Combinatorics

Release 10.6

The Sage Development Team

Apr 01, 2025

CONTENTS

CHAPTER

ONE

INTRODUCTORY MATERIAL

- *Combinatorics quickref*
- *Introduction to combinatorics in Sage*

CHAPTER

TWO

THEMATIC INDEXES

- *Algebraic combinatorics*
  - *Combinatorial Hopf algebras*
  - *Cluster algebras and quivers*
  - *Crystals*
  - *Root Systems*
  - *Symmetric Functions*
  - `FullyCommutativeElements`
- *Counting*
- *Enumerated sets and combinatorial objects*
- *Enumerated sets of partitions, tableaux, …*
- *Finite state machines, automata, transducers*
- *Combinatorial species*
- *Combinatorial designs and incidence structures*
- *Posets*
- *Combinatorics on words*
- *A bijectionist's toolkit*

CHAPTER

THREE

UTILITIES

CHAPTER

FOUR

RELATED TOPICS

- Coding Theory
- Discrete dynamics
- Graph Theory

CHAPTER

FIVE

COMPREHENSIVE MODULE LIST

5.1 Comprehensive Module List

Note

This list is currently sorted in alphabetical order w.r.t. the module names. It can be updated semi-automatically by running in `src/sage/combinat`:

```
find -name "*.py*" | sed 's|\.pyx\?$||; s|\./|    sage/combinat/|' | LANG=en_US.UTF-
→8 LC_COLLATE=C sort > /tmp/module_list.rst
```

and copy pasting the result back there.

Todo

See Issue #17421 for desirable improvements.

5.1.1 Abstract Recursive Trees

The purpose of this class is to help implement trees with a specific structure on the children of each node. For instance, one could want to define a tree in which each node sees its children as linearly (see the *Ordered Trees* module) or cyclically ordered.

Tree structures

Conceptually, one can define a tree structure from any object that can contain others. Indeed, a list can contain lists which contain lists which contain lists, and thus define a tree … The same can be done with sets, or any kind of iterable objects.

While any iterable is sufficient to encode trees, it can prove useful to have other methods available like isomorphism tests (see next section), conversions to DiGraphs objects (see *as_digraph()*) or computation of the number of automorphisms constrained by the structure on children. Providing such methods is the whole purpose of the *AbstractTree* class.

As a result, the *AbstractTree* class is not meant to be instantiated, but extended. It is expected that classes extending this one may also inherit from classes representing iterables, for instance `ClonableArray` or `ClonableList`

Constrained Trees

The tree built from a specific container will reflect the properties of the container. Indeed, if `A` is an iterable class whose elements are linearly ordered, a class `B` extending both of *AbstractTree* and `A` will be such that the children of a node will be linearly ordered. If `A` behaves like a set (i.e. if there is no order on the elements it contains), then two trees will be considered as equal if one can be obtained from the other through permutations between the children of a same node (see next section).

Paths and ID

It is expected that each element of a set of children should be identified by its index in the container. This way, any node of the tree can be identified by a word describing a path from the root node.

Canonical labellings

Equality between instances of classes extending both `AbstractTree` and `A` is entirely defined by the equality defined on the elements of `A`. A canonical labelling of such a tree, however, should be such that two trees `a` and `b` satisfying `a == b` have the same canonical labellings. On the other hand, the canonical labellings of trees `a` and `b` satisfying `a != b` are expected to be different.

For this reason, the values returned by the `canonical_labelling` method heavily depend on the data structure used for a node's children and **should be overridden** by most of the classes extending `AbstractTree` if it is incoherent with the data structure.

Authors

- Florent Hivert (2010-2011): initial revision
- Frédéric Chapoton (2011): contributed some methods

class `sage.combinat.abstract_tree.`**`AbstractClonableTree`**

Bases: `AbstractTree`

Abstract Clonable Tree.

An abstract class for trees with clone protocol (see `list_clone`). It is expected that classes extending this one may also inherit from classes like `ClonableArray` or `ClonableList` depending whether one wants to build trees where adding a child is allowed.

> **Note**
>
> Due to the limitation of Cython inheritance, one cannot inherit here from `ClonableElement`, because it would prevent us from later inheriting from `ClonableArray` or `ClonableList`.

How should this class be extended ?

A class extending `AbstractClonableTree` should satisfy the following assumptions:

- An instantiable class extending `AbstractClonableTree` should also extend the `ClonableElement` class or one of its subclasses generally, at least `ClonableArray`.
- To respect the Clone protocol, the `AbstractClonableTree.check()` method should be overridden by the new class.

See also the assumptions in `AbstractTree`.

`check()`

Check that `self` is a correct tree.

This method does nothing. It is implemented here because many extensions of `AbstractClonableTree` also extend `sage.structure.list_clone.ClonableElement`, which requires it.

It should be overridden in subclasses in order to check that the characterizing property of the respective kind of tree holds (eg: two children for binary trees).

EXAMPLES:

```
sage: OrderedTree([[],[[]]]).check()
sage: BinaryTree([[],[[],[]]]).check()
```

```
>>> from sage.all import *
>>> OrderedTree([[],[[]]]).check()
>>> BinaryTree([[],[[],[]]]).check()
```

class sage.combinat.abstract_tree.**AbstractLabelledClonableTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: *AbstractLabelledTree*, *AbstractClonableTree*

Abstract Labelled Clonable Tree.

This class takes care of modification for the label by the clone protocol.

Note

Due to the limitation of Cython inheritance, one cannot inherit here from `ClonableArray`, because it would prevent us to inherit later from `ClonableList`.

map_labels(*f*)

Apply the function f to the labels of `self`.

This method returns a copy of `self` on which the function f has been applied on all labels (a label x is replaced by $f(x)$).

EXAMPLES:

```
sage: LT = LabelledOrderedTree
sage: t = LT([LT([],label=1),LT([],label=7)],label=3); t
3[1[], 7[]]
sage: t.map_labels(lambda z:z+1)
4[2[], 8[]]

sage: LBT = LabelledBinaryTree
sage: bt = LBT([LBT([],label=1),LBT([],label=4)],label=2); bt
2[1[., .], 4[., .]]
sage: bt.map_labels(lambda z:z+1)
3[2[., .], 5[., .]]
```

```
>>> from sage.all import *
>>> LT = LabelledOrderedTree
>>> t = LT([LT([],label=Integer(1)),LT([],label=Integer(7))],
↪label=Integer(3)); t
3[1[], 7[]]
>>> t.map_labels(lambda z:z+Integer(1))
4[2[], 8[]]

>>> LBT = LabelledBinaryTree
>>> bt = LBT([LBT([],label=Integer(1)),LBT([],label=Integer(4))],
↪label=Integer(2)); bt
2[1[., .], 4[., .]]
```

(continues on next page)

(continued from previous page)

```
>>> bt.map_labels(lambda z:z+Integer(1))
3[2[., .], 5[., .]]
```

set_label(*path*, *label*)

Change the label of subtree indexed by `path` to `label`.

INPUT:

- `path` – `None` (default) or a path (list or tuple of children index in the tree)
- `label` – any sage object

OUTPUT: nothing, `self` is modified in place

Note

`self` must be in a mutable state. See `sage.structure.list_clone` for more details about mutability.

EXAMPLES:

```
sage: t = LabelledOrderedTree([[],[[],[]]])
sage: t.set_label((0,), 4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with t.clone() as t:
....:     t.set_label((0,), 4)
sage: t
None[4[], None[None[], None[]]]
sage: with t.clone() as t:
....:     t.set_label((1,0), label = 42)
sage: t
None[4[], None[42[], None[]]]
```

```
>>> from sage.all import *
>>> t = LabelledOrderedTree([[],[[],[]]])
>>> t.set_label((Integer(0),), Integer(4))
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
>>> with t.clone() as t:
...     t.set_label((Integer(0),), Integer(4))
>>> t
None[4[], None[None[], None[]]]
>>> with t.clone() as t:
...     t.set_label((Integer(1),Integer(0)), label = Integer(42))
>>> t
None[4[], None[42[], None[]]]
```

Todo

Do we want to implement the following syntactic sugar:

```
with t.clone() as tt:
    tt.labels[1,2] = 3 ?
```

set_root_label(*label*)

Set the label of the root of `self`.

INPUT:

- `label` – any Sage object

OUTPUT: none, `self` is modified in place

Note

`self` must be in a mutable state. See `sage.structure.list_clone` for more details about mutability.

EXAMPLES:

```
sage: t = LabelledOrderedTree([[],[[],[]]])
sage: t.set_root_label(3)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with t.clone() as t:
....:     t.set_root_label(3)
sage: t.label()
3
sage: t
3[None[], None[None[], None[]]]
```

```
>>> from sage.all import *
>>> t = LabelledOrderedTree([[],[[],[]]])
>>> t.set_root_label(Integer(3))
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
>>> with t.clone() as t:
...     t.set_root_label(Integer(3))
>>> t.label()
3
>>> t
3[None[], None[None[], None[]]]
```

This also works for binary trees:

```
sage: bt = LabelledBinaryTree([[],[]])
sage: bt.set_root_label(3)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with bt.clone() as bt:
....:     bt.set_root_label(3)
```

(continues on next page)

(continued from previous page)

```
sage: bt.label()
3
sage: bt
3[None[., .], None[., .]]
```

```
>>> from sage.all import *
>>> bt = LabelledBinaryTree([[],[]])
>>> bt.set_root_label(Integer(3))
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
>>> with bt.clone() as bt:
...     bt.set_root_label(Integer(3))
>>> bt.label()
3
>>> bt
3[None[., .], None[., .]]
```

class sage.combinat.abstract_tree.**AbstractLabelledTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: *AbstractTree*

Abstract Labelled Tree.

Typically a class for labelled trees is constructed by inheriting from a class for unlabelled trees and *AbstractLabelledTree*.

How should this class be extended ?

A class extending *AbstractLabelledTree* should respect the following assumptions:

- For a labelled tree `T` the call `T.parent().unlabelled_trees()` should return a parent for unlabelled trees of the same kind: for example,
  - if `T` is a binary labelled tree, `T.parent()` is `LabelledBinaryTrees()` and `T.parent().unlabelled_trees()` is `BinaryTrees()`
  - if `T` is an ordered labelled tree, `T.parent()` is `LabelledOrderedTrees()` and `T.parent().unlabelled_trees()` is `OrderedTrees()`
- In the same vein, the class of `T` should contain an attribute `_UnLabelled` which should be the class for the corresponding unlabelled trees.

See also the assumptions in *AbstractTree*.

See also

AbstractTree

as_digraph()

Return a directed graph version of `self`.

Warning

At this time, the output makes sense only if `self` is a labelled binary tree with no repeated labels and no `None` labels.

EXAMPLES:

```
sage: LT = LabelledOrderedTrees()
sage: t1 = LT([LT([],label=6),LT([],label=1)],label=9)
sage: t1.as_digraph()
Digraph on 3 vertices

sage: t = BinaryTree([[None, None],[[],None]])
sage: lt = t.canonical_labelling()
sage: lt.as_digraph()
Digraph on 4 vertices
```

```
>>> from sage.all import *
>>> LT = LabelledOrderedTrees()
>>> t1 = LT([LT([],label=Integer(6)),LT([],label=Integer(1))],
↪label=Integer(9))
>>> t1.as_digraph()
Digraph on 3 vertices

>>> t = BinaryTree([[None, None],[[],None]])
>>> lt = t.canonical_labelling()
>>> lt.as_digraph()
Digraph on 4 vertices
```

label(*path=None*)

Return the label of `self`.

INPUT:

- `path` – `None` (default) or a path (list or tuple of children index in the tree)

OUTPUT: the label of the subtree indexed by `path`

EXAMPLES:

```
sage: t = LabelledOrderedTree([[],[]], label = 3)
sage: t.label()
3
sage: t[0].label()
sage: t = LabelledOrderedTree([LabelledOrderedTree([], 5),[]], label = 3)
sage: t.label()
3
sage: t[0].label()
5
sage: t[1].label()
sage: t.label([0])
5
```

```
>>> from sage.all import *
>>> t = LabelledOrderedTree([[],[]], label = Integer(3))
>>> t.label()
```

(continues on next page)

(continued from previous page)

```
3
>>> t[Integer(0)].label()
>>> t = LabelledOrderedTree([LabelledOrderedTree([], Integer(5)),[]], label =␣
↪Integer(3))
>>> t.label()
3
>>> t[Integer(0)].label()
5
>>> t[Integer(1)].label()
>>> t.label([Integer(0)])
5
```

labels()

Return the list of labels of `self`.

EXAMPLES:

```
sage: LT = LabelledOrderedTree
sage: t = LT([LT([],label='b'),LT([],label='c')],label='a')
sage: t.labels()
['a', 'b', 'c']

sage: LBT = LabelledBinaryTree
sage: LBT([LBT([],label=1),LBT([],label=4)],label=2).labels()
[2, 1, 4]
```

```
>>> from sage.all import *
>>> LT = LabelledOrderedTree
>>> t = LT([LT([],label='b'),LT([],label='c')],label='a')
>>> t.labels()
['a', 'b', 'c']

>>> LBT = LabelledBinaryTree
>>> LBT([LBT([],label=Integer(1)),LBT([],label=Integer(4))],label=Integer(2)).
↪labels()
[2, 1, 4]
```

leaf_labels()

Return the list of labels of the leaves of `self`.

In case of a labelled binary tree, these "leaves" are not actually the leaves of the binary trees, but the nodes whose both children are leaves!

EXAMPLES:

```
sage: LT = LabelledOrderedTree
sage: t = LT([LT([],label='b'),LT([],label='c')],label='a')
sage: t.leaf_labels()
['b', 'c']

sage: LBT = LabelledBinaryTree
sage: bt = LBT([LBT([],label='b'),LBT([],label='c')],label='a')
sage: bt.leaf_labels()
```

(continues on next page)

(continued from previous page)

```
['b', 'c']
sage: LBT([], label='1').leaf_labels()
['1']
sage: LBT(None).leaf_labels()
[]
```

```
>>> from sage.all import *
>>> LT = LabelledOrderedTree
>>> t = LT([LT([],label='b'),LT([],label='c')],label='a')
>>> t.leaf_labels()
['b', 'c']

>>> LBT = LabelledBinaryTree
>>> bt = LBT([LBT([],label='b'),LBT([],label='c')],label='a')
>>> bt.leaf_labels()
['b', 'c']
>>> LBT([], label='1').leaf_labels()
['1']
>>> LBT(None).leaf_labels()
[]
```

shape()

Return the unlabelled tree associated to `self`.

EXAMPLES:

```
sage: t = LabelledOrderedTree([[],[[]]], label = 25).shape(); t
[[], [[]]]

sage: LabelledBinaryTree([[],[[],[]]], label = 25).shape()
[[., .], [[., .], [., .]]]

sage: LRT = LabelledRootedTree
sage: tb = LRT([],label='b')
sage: LRT([tb, tb], label='a').shape()
[[], []]
```

```
>>> from sage.all import *
>>> t = LabelledOrderedTree([[],[[]]], label = Integer(25)).shape(); t
[[], [[]]]

>>> LabelledBinaryTree([[],[[],[]]], label = Integer(25)).shape()
[[., .], [[., .], [., .]]]

>>> LRT = LabelledRootedTree
>>> tb = LRT([],label='b')
>>> LRT([tb, tb], label='a').shape()
[[], []]
```

class sage.combinat.abstract_tree.**AbstractTree**

Bases: `object`

Abstract Tree.

There is no data structure defined here, as this class is meant to be extended, not instantiated.

How should this class be extended?

A class extending *AbstractTree* should respect several assumptions:

- For a tree `T`, the call `iter(T)` should return an iterator on the children of the root `T`.
- The *canonical_labelling* method should return the same value for trees that are considered equal (see the "canonical labellings" section in the documentation of the *AbstractTree* class).
- For a tree `T` the call `T.parent().labelled_trees()` should return a parent for labelled trees of the same kind: for example,
  - if `T` is a binary tree, `T.parent()` is `BinaryTrees()` and `T.parent().labelled_trees()` is `LabelledBinaryTrees()`
  - if `T` is an ordered tree, `T.parent()` is `OrderedTrees()` and `T.parent().labelled_trees()` is `LabelledOrderedTrees()`

breadth_first_order_traversal(*action=None*)

Run the breadth-first post-order traversal algorithm and subject every node encountered to some procedure `action`. The algorithm is:

```
queue <- [ root ];
while the queue is not empty:
    node <- pop( queue );
    manipulate the node;
    prepend to the queue the list of all subtrees of
        the node (from the rightmost to the leftmost).
```

INPUT:

- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

EXAMPLES:

For example, on the following binary tree b:

```
|   ___3____      |
|  /        \     |
| 1         _7_   |
|  \       /   \  |
|   2     5     8 |
|        / \      |
|       4   6     |
```

the breadth-first order traversal algorithm explores b in the following order of nodes: $3, 1, 7, 2, 5, 8, 4, 6$.

canonical_labelling(*shift=1*)

Return a labelled version of `self`.

The actual canonical labelling is currently unspecified. However, it is guaranteed to have labels in $1...n$ where n is the number of nodes of the tree. Moreover, two (unlabelled) trees compare as equal if and only if their canonical labelled trees compare as equal.

EXAMPLES:

```
sage: t = OrderedTree([[], [[], [[], []], [[], []]], [[], []]])
sage: t.canonical_labelling()
1[2[], 3[4[], 5[6[], 7[]], 8[9[], 10[]]], 11[12[], 13[]]]

sage: BinaryTree([]).canonical_labelling()
1[., .]
sage: BinaryTree([[],[[],[]]]).canonical_labelling()
2[1[., .], 4[3[., .], 5[., .]]]
```

```
>>> from sage.all import *
>>> t = OrderedTree([[], [[], [[], []], [[], []]], [[], []]])
>>> t.canonical_labelling()
1[2[], 3[4[], 5[6[], 7[]], 8[9[], 10[]]], 11[12[], 13[]]]

>>> BinaryTree([]).canonical_labelling()
1[., .]
>>> BinaryTree([[],[[],[]]]).canonical_labelling()
2[1[., .], 4[3[., .], 5[., .]]]
```

contour_traversal(*first_action=None*, *middle_action=None*, *final_action=None*, *leaf_action=None*)

Run the counterclockwise countour traversal algorithm (iterative implementation) and subject every node encountered to some procedure `first_action`, `middle_action` or `final_action` each time it reaches it.

ALGORITHM:

- if the root is a leaf, apply $leaf_action$
- **else**
  - apply $first_action$ to the root
  - iteratively apply $middle_action$ to the root and traverse each subtree from the leftmost one to the rightmost one
  - apply $final_action$ to the root

INPUT:

- `first_action` – (optional) a function which takes a node as input, and does something the first time it is reached during exploration
- `middle_action` – (optional) a function which takes a node as input, and does something each time it explore one of its children
- `final_action` – (optional) a function which takes a node as input, and does something the last time it is reached during exploration
- `leaf_action` – (optional) a function which takes a leaf as input, and does something when it is reached during exploration.

OUTPUT:

`None`. (This is *not* an iterator.)

depth()

Return the depth of `self`.

EXAMPLES:

```
sage: OrderedTree().depth()
1
sage: OrderedTree([]).depth()
1
sage: OrderedTree([[],[]]).depth()
2
sage: OrderedTree([[],[[]]]).depth()
3
sage: OrderedTree([[], [[], [[], []], [[], []]], [[], []]]).depth()
4

sage: BinaryTree().depth()
0
sage: BinaryTree([[],[[],[]]]).depth()
3
```

```
>>> from sage.all import *
>>> OrderedTree().depth()
1
>>> OrderedTree([]).depth()
1
>>> OrderedTree([[],[]]).depth()
2
>>> OrderedTree([[],[[]]]).depth()
3
>>> OrderedTree([[], [[], [[], []], [[], []]], [[], []]]).depth()
4

>>> BinaryTree().depth()
0
>>> BinaryTree([[],[[],[]]]).depth()
3
```

iterative_post_order_traversal(*action=None*)

Run the depth-first post-order traversal algorithm (iterative implementation) and subject every node encountered to some procedure `action`. The algorithm is:

```
explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one;
then manipulate the root with function `action` (in the
    case of a binary tree, only if the root is not a leaf).
```

INPUT:

- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

See also

- *post_order_traversal_iter()*

iterative_pre_order_traversal(*action=None*)

Run the depth-first pre-order traversal algorithm (iterative implementation) and subject every node encountered to some procedure `action`. The algorithm is:

```
manipulate the root with function `action` (in the case
    of a binary tree, only if the root is not a leaf);
then explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one.
```

INPUT:

- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

See also

- *pre_order_traversal_iter()*
- *pre_order_traversal()*

node_number()

Return the number of nodes of `self`.

See also

node_number_at_depth(), *node_number_to_the_right()*

EXAMPLES:

```
sage: OrderedTree().node_number()
1
sage: OrderedTree([]).node_number()
1
sage: OrderedTree([[],[]]).node_number()
3
sage: OrderedTree([[],[[]]]).node_number()
4
sage: OrderedTree([[], [[], [[], []], [[], []]], [[], []]]).node_number()
13
```

```
>>> from sage.all import *
>>> OrderedTree().node_number()
1
>>> OrderedTree([]).node_number()
1
>>> OrderedTree([[],[]]).node_number()
3
>>> OrderedTree([[],[[]]]).node_number()
4
```

(continues on next page)

(continued from previous page)

```
>>> OrderedTree([[], [[], [[], []], [[], []]], [[], []]]).node_number()
13
```

EXAMPLES:

```
sage: BinaryTree(None).node_number()
0
sage: BinaryTree([]).node_number()
1
sage: BinaryTree([[], None]).node_number()
2
sage: BinaryTree([[None, [[], []]], None]).node_number()
5
```

```
>>> from sage.all import *
>>> BinaryTree(None).node_number()
0
>>> BinaryTree([]).node_number()
1
>>> BinaryTree([[], None]).node_number()
2
>>> BinaryTree([[None, [[], []]], None]).node_number()
5
```

node_number_at_depth(*depth*)

Return the number of nodes at a given depth.

This counts all nodes that are at the given depth.

Here the root is considered to have depth 0.

INPUT:

- `depth` – integer

See also

node_number(), *node_number_to_the_right()*, *paths_at_depth()*

EXAMPLES:

```
sage: T = OrderedTree([[[], [[]]], [[], [[[]]]], []])
sage: ascii_art(T)
    ___o____
   /    /  /
  o_   o_ o
 / /  / /
o o  o o
  |    |
  o    o
       |
       o
```

(continues on next page)

(continued from previous page)

```
sage: [T.node_number_at_depth(i) for i in range(6)]
[1, 3, 4, 2, 1, 0]
```

```
>>> from sage.all import *
>>> T = OrderedTree([[[], [[]]], [[], [[[]]]], []])
>>> ascii_art(T)
    ___o____
   /    /  /
  o_   o_ o
 / /  / /
o o  o o
  |    |
  o    o
       |
       o
>>> [T.node_number_at_depth(i) for i in range(Integer(6))]
[1, 3, 4, 2, 1, 0]
```

node_number_to_the_right(*path*)

Return the number of nodes at the same depth and to the right of the node identified by `path`.

This counts the nodes that are at the same depth as the given one, and strictly to its right.

See also

node_number(), *node_number_at_depth()*, *paths_to_the_right()*

EXAMPLES:

```
sage: T = OrderedTree([[[], [[]]], [[], [[[]]]], []])
sage: ascii_art(T)
    ___o____
   /    /  /
  o_   o_ o
 / /  / /
o o  o o
  |    |
  o    o
       |
       o
sage: T.node_number_to_the_right(())
0
sage: T.node_number_to_the_right((0,))
2
sage: T.node_number_to_the_right((0,1))
2
sage: T.node_number_to_the_right((0,1,0))
1

sage: T = OrderedTree([])
sage: T.node_number_to_the_right(())
0
```

```
>>> from sage.all import *
>>> T = OrderedTree([[[], [[]]], [[], [[[]]]], []])
>>> ascii_art(T)
    ___o____
   /     /  /
  o_    o_ o
 / /   / /
o o   o o
  |     |
  o     o
        |
        o
>>> T.node_number_to_the_right(())
0
>>> T.node_number_to_the_right((Integer(0),))
2
>>> T.node_number_to_the_right((Integer(0),Integer(1)))
2
>>> T.node_number_to_the_right((Integer(0),Integer(1),Integer(0)))
1

>>> T = OrderedTree([])
>>> T.node_number_to_the_right(())
0
```

paths()

Return a generator for all paths to nodes of `self`.

OUTPUT:

This method returns a list of sequences of integers. Each of these sequences represents a path from the root node to some node. For instance, $(1, 3, 2, 5, 0, 3)$ represents the node obtained by choosing the 1st child of the root node (in the ordering returned by `iter`), then the 3rd child of its child, then the 2nd child of the latter, etc. (where the labelling of the children is zero-based).

The root element is represented by the empty tuple `()`.

See also

paths_at_depth(), *paths_to_the_right()*

EXAMPLES:

```
sage: list(OrderedTree([]).paths())
[()]
sage: list(OrderedTree([[],[[]]]).paths())
[(), (0,), (1,), (1, 0)]

sage: list(BinaryTree([[],[[],[]]]).paths())
[(), (0,), (1,), (1, 0), (1, 1)]
```

```
>>> from sage.all import *
>>> list(OrderedTree([]).paths())
```

(continues on next page)

(continued from previous page)

```
[()]
>>> list(OrderedTree([[],[[]]]).paths())
[(), (0,), (1,), (1, 0)]

>>> list(BinaryTree([[],[[],[]]]).paths())
[(), (0,), (1,), (1, 0), (1, 1)]
```

paths_at_depth(*depth*, *path=[]*)

Return a generator for all paths at a fixed depth.

This iterates over all paths for nodes that are at the given depth.

Here the root is considered to have depth 0.

INPUT:

- `depth` – integer
- `path` – (optional) list; given path used in the recursion

Warning

The `path` option should not be used directly.

See also

paths(), *paths_to_the_right()*, *node_number_at_depth()*

EXAMPLES:

```
sage: T = OrderedTree([[[], [[], [[]]]], [], [[[],[]]], [], []])
sage: ascii_art(T)
     ______o_______
    /    /   /  / /
  _o__  o   o  o o
 /   /      |
o   o_      o_
   / /     / /
  o o     o o
    |
    o
sage: list(T.paths_at_depth(0))
[()]
sage: list(T.paths_at_depth(2))
[(0, 0), (0, 1), (2, 0)]
sage: list(T.paths_at_depth(4))
[(0, 1, 1, 0)]
sage: list(T.paths_at_depth(5))
[]

sage: T2 = OrderedTree([])
```

(continues on next page)

(continued from previous page)

```
sage: list(T2.paths_at_depth(0))
[()]
```

```
>>> from sage.all import *
>>> T = OrderedTree([[[], [[], [[]]]], [], [[[],[]]], [], []])
>>> ascii_art(T)
     ______o_______
    /     /   /  / /
  _o__   o   o  o o
 /   /       |
o   o_       o_
   / /      / /
  o o      o o
    |
    o
>>> list(T.paths_at_depth(Integer(0)))
[()]
>>> list(T.paths_at_depth(Integer(2)))
[(0, 0), (0, 1), (2, 0)]
>>> list(T.paths_at_depth(Integer(4)))
[(0, 1, 1, 0)]
>>> list(T.paths_at_depth(Integer(5)))
[]

>>> T2 = OrderedTree([])
>>> list(T2.paths_at_depth(Integer(0)))
[()]
```

paths_to_the_right(*path*)

Return a generator of paths for all nodes at the same depth and to the right of the node identified by `path`.

This iterates over the paths for nodes that are at the same depth as the given one, and strictly to its right.

INPUT:

- `path` – any path in the tree

See also

paths(), *paths_at_depth()*, *node_number_to_the_right()*

EXAMPLES:

```
sage: T = OrderedTree([[[], [[]]], [[], [[[]]]], []])
sage: ascii_art(T)
    ___o____
   /     /  /
  o_    o_ o
 / /   / /
o o   o o
  |     |
  o     o
```

(continues on next page)

(continued from previous page)

```
       |
       o
sage: g = T.paths_to_the_right(())
sage: list(g)
[]

sage: g = T.paths_to_the_right((0,))
sage: list(g)
[(1,), (2,)]

sage: g = T.paths_to_the_right((0,1))
sage: list(g)
[(1, 0), (1, 1)]

sage: g = T.paths_to_the_right((0,1,0))
sage: list(g)
[(1, 1, 0)]

sage: g = T.paths_to_the_right((1,2))
sage: list(g)
[]
```

```
>>> from sage.all import *
>>> T = OrderedTree([[[], [[]]], [[], [[[]]]], []])
>>> ascii_art(T)
   ___o____
  /    /  /
 o_   o_ o
/ /  / /
o o  o o
 |    |
 o    o
      |
      o
>>> g = T.paths_to_the_right(())
>>> list(g)
[]

>>> g = T.paths_to_the_right((Integer(0),))
>>> list(g)
[(1,), (2,)]

>>> g = T.paths_to_the_right((Integer(0),Integer(1)))
>>> list(g)
[(1, 0), (1, 1)]

>>> g = T.paths_to_the_right((Integer(0),Integer(1),Integer(0)))
>>> list(g)
[(1, 1, 0)]

>>> g = T.paths_to_the_right((Integer(1),Integer(2)))
>>> list(g)
```

(continues on next page)

(continued from previous page)

```
[]
```

post_order_traversal(*action=None*)

Run the depth-first post-order traversal algorithm (recursive implementation) and subject every node encountered to some procedure `action`. The algorithm is:

```
explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one;
then manipulate the root with function `action` (in the
    case of a binary tree, only if the root is not a leaf).
```

INPUT:

- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

See also:

- *post_order_traversal_iter()*
- *iterative_post_order_traversal()*

post_order_traversal_iter()

The depth-first post-order traversal iterator.

This method iters each node following the depth-first post-order traversal algorithm (recursive implementation). The algorithm is:

```
explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one;
then yield the root (in the case of binary trees, only if
    it is not a leaf).
```

EXAMPLES:

For example on the following binary tree b:

```
|   ___3____       |
|  /        \      |
| 1         _7_    |
|  \       /   \   |
|   2     5     8  |
|        / \       |
|       4   6      |
```

(only the nodes are shown), the depth-first post-order traversal algorithm explores b in the following order of nodes: $2, 1, 4, 6, 5, 8, 7, 3$.

For another example, consider the labelled tree:

```
|      __1_____ |
|     /  /    / |
|    2   6    8_ |
|    |   |   / / |
|    3_  7  9 10 |
|   / /          |
| 4  5           |
```

The algorithm explores this tree in the following order: $4, 5, 3, 2, 7, 6, 9, 10, 8, 1$.

pre_order_traversal(*action=None*)

Run the depth-first pre-order traversal algorithm (recursive implementation) and subject every node encountered to some procedure `action`. The algorithm is:

```
manipulate the root with function `action` (in the case
    of a binary tree, only if the root is not a leaf);
then explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one.
```

INPUT:

- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

EXAMPLES:

For example, on the following binary tree b:

```
|    ____3_____        |
|   /          \       |
| 1           _7_      |
|   \        /   \     |
|    2      5      8   |
|          / \         |
|         4    6       |
```

the depth-first pre-order traversal algorithm explores b in the following order of nodes: $3, 1, 2, 7, 5, 4, 6, 8$.

Another example:

```
|      __1_____ |
|     /  /    / |
|    2   6    8_ |
|    |   |   / / |
|    3_  7  9 10 |
|   / /          |
| 4  5           |
```

The algorithm explores this tree in the following order: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

See also

- *pre_order_traversal_iter()*

- *iterative_pre_order_traversal()*

pre_order_traversal_iter()

The depth-first pre-order traversal iterator.

This method iters each node following the depth-first pre-order traversal algorithm (recursive implementation). The algorithm is:

```
yield the root (in the case of binary trees, if it is not
    a leaf);
then explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one.
```

EXAMPLES:

For example, on the following binary tree b:

```
|   ___3____      |
|  /        \     |
| 1         _7_   |
|  \       /   \  |
|   2     5     8 |
|        / \      |
|       4   6     |
```

(only the nodes shown), the depth-first pre-order traversal algorithm explores b in the following order of nodes: $3, 1, 2, 7, 5, 4, 6, 8$.

Another example:

```
|     __1____ |
|    /  /   / |
|   2   6   8_ |
|   |   |  / / |
|   3_  7  9 10 |
|  / /         |
| 4 5          |
```

The algorithm explores this labelled tree in the following order: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

subtrees()

Return a generator for all nonempty subtrees of `self`.

The number of nonempty subtrees of a tree is its number of nodes. (The word "nonempty" makes a difference only in the case of binary trees. For ordered trees, for example, all trees are nonempty.)

EXAMPLES:

```
sage: list(OrderedTree([]).subtrees())
[[]]
sage: list(OrderedTree([[],[[]]]).subtrees())
[[[], [[]]], [], [[]], []]

sage: list(OrderedTree([[],[[]]]).canonical_labelling().subtrees())
[1[2[], 3[4[]]], 2[], 3[4[]], 4[]]
```

(continues on next page)

(continued from previous page)

```
sage: list(BinaryTree([[],[[],[]]]).subtrees())
[[[., .], [[., .], [., .]]], [., .], [[., .], [., .]], [., .], [., .]]

sage: v = BinaryTree([[],[]])
sage: list(v.canonical_labelling().subtrees())
[2[1[., .], 3[., .]], 1[., .], 3[., .]]
```

```
>>> from sage.all import *
>>> list(OrderedTree([]).subtrees())
[[]]
>>> list(OrderedTree([[],[[]]]).subtrees())
[[[], [[]]], [], [[]], []]

>>> list(OrderedTree([[],[[]]]).canonical_labelling().subtrees())
[1[2[], 3[4[]]], 2[], 3[4[]], 4[]]

>>> list(BinaryTree([[],[[],[]]]).subtrees())
[[[., .], [[., .], [., .]]], [., .], [[., .], [., .]], [., .], [., .]]

>>> v = BinaryTree([[],[]])
>>> list(v.canonical_labelling().subtrees())
[2[1[., .], 3[., .]], 1[., .], 3[., .]]
```

to_hexacode()

Transform a tree into a hexadecimal string.

The definition of the hexacode is recursive. The first letter is the valence of the root as a hexadecimal (up to 15), followed by the concatenation of the hexacodes of the subtrees.

This method only works for trees where every vertex has valency at most 15.

See *from_hexacode()* for the reverse transformation.

EXAMPLES:

```
sage: from sage.combinat.abstract_tree import from_hexacode
sage: LT = LabelledOrderedTrees()
sage: from_hexacode('2010', LT).to_hexacode()
'2010'
sage: LT.an_element().to_hexacode()
'3020010'
sage: t = from_hexacode('a0000000000000000', LT)
sage: t.to_hexacode()
'a0000000000'

sage: OrderedTrees(6).an_element().to_hexacode()
'500000'
```

```
>>> from sage.all import *
>>> from sage.combinat.abstract_tree import from_hexacode
>>> LT = LabelledOrderedTrees()
>>> from_hexacode('2010', LT).to_hexacode()
```

(continues on next page)

(continued from previous page)

```
'2010'
>>> LT.an_element().to_hexacode()
'3020010'
>>> t = from_hexacode('a0000000000000000', LT)
>>> t.to_hexacode()
'a0000000000'

>>> OrderedTrees(Integer(6)).an_element().to_hexacode()
'500000'
```

tree_factorial()

Return the tree-factorial of `self`.

Definition:

The tree-factorial $T!$ of a tree T is the product $\prod_{v \in T} \#\text{children}(v)$.

EXAMPLES:

```
sage: LT = LabelledOrderedTrees()
sage: t = LT([LT([],label=6),LT([],label=1)],label=9)
sage: t.tree_factorial()
3

sage: BinaryTree([[],[[],[]]]).tree_factorial()
15
```

```
>>> from sage.all import *
>>> LT = LabelledOrderedTrees()
>>> t = LT([LT([],label=Integer(6)),LT([],label=Integer(1))],label=Integer(9))
>>> t.tree_factorial()
3

>>> BinaryTree([[],[[],[]]]).tree_factorial()
15
```

sage.combinat.abstract_tree.**from_hexacode**(*ch*, *parent=None*, *label='@'*)

Transform a hexadecimal string into a tree.

INPUT:

- `ch` – a hexadecimal string
- `parent` – kind of trees to be produced. If `None`, this will be `LabelledOrderedTrees`
- `label` – a label (default: `'@'`) to be used for every vertex of the tree

See *AbstractTree.to_hexacode()* for the description of the encoding

See `_from_hexacode_aux()` for the actual code

EXAMPLES:

```
sage: from sage.combinat.abstract_tree import from_hexacode
sage: from_hexacode('12000', LabelledOrderedTrees())
@[@[@[], @[]]]
sage: from_hexacode('12000')
```

(continues on next page)

(continued from previous page)

```
@[@[@[], @[]]]

sage: from_hexacode('1200', LabelledOrderedTrees())
@[@[@[], @[]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.abstract_tree import from_hexacode
>>> from_hexacode('12000', LabelledOrderedTrees())
@[@[@[], @[]]]
>>> from_hexacode('12000')
@[@[@[], @[]]]

>>> from_hexacode('1200', LabelledOrderedTrees())
@[@[@[], @[]]]
```

It can happen that only a prefix of the word is used:

```
sage: from_hexacode('a'+14*'0', LabelledOrderedTrees())
@[@[], @[], @[], @[], @[], @[], @[], @[], @[], @[]]
```

```
>>> from sage.all import *
>>> from_hexacode('a'+Integer(14)*'0', LabelledOrderedTrees())
@[@[], @[], @[], @[], @[], @[], @[], @[], @[], @[]]
```

One can choose the label:

```
sage: from_hexacode('1200', LabelledOrderedTrees(), label='o')
o[o[o[], o[]]]
```

```
>>> from sage.all import *
>>> from_hexacode('1200', LabelledOrderedTrees(), label='o')
o[o[o[], o[]]]
```

One can also create other kinds of trees:

```
sage: from_hexacode('1200', OrderedTrees())
[[[], []]]
```

```
>>> from sage.all import *
>>> from_hexacode('1200', OrderedTrees())
[[[], []]]
```

5.1.2 Affine Permutations

class sage.combinat.affine_permutation.**AffinePermutation**(*parent*, *lst*, *check=True*)

Bases: ClonableArray

An affine permutation, represented in the window notation, and considered as a bijection from $\mathbf{Z}$ to $\mathbf{Z}$.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p = A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p = A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

apply_simple_reflection(*i*, *side='right'*)

Apply a simple reflection.

INPUT:

- `i` – integer
- `side` – (default: `'right'`) determines whether to apply the reflection on the `'right'` or `'left'`

EXAMPLES:

```
sage: p = AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.apply_simple_reflection(3)
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
sage: p.apply_simple_reflection(11)
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
sage: p.apply_simple_reflection(3, 'left')
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
sage: p.apply_simple_reflection(11, 'left')
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
```

```
>>> from sage.all import *
>>> p = AffinePermutationGroup(['A',Integer(7),Integer(1)])([Integer(3), -
↪Integer(1), Integer(0), Integer(6), Integer(5), Integer(4), Integer(10),␣
↪Integer(9)])
>>> p.apply_simple_reflection(Integer(3))
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
>>> p.apply_simple_reflection(Integer(11))
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
>>> p.apply_simple_reflection(Integer(3), 'left')
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
>>> p.apply_simple_reflection(Integer(11), 'left')
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
```

grassmannian_quotient(*i=0*, *side='right'*)

Return the Grassmannian quotient.

Factors `self` into a unique product of a Grassmannian and a finite-type element. Returns a tuple containing the Grassmannian and finite elements, in order according to `side`.

INPUT:

- `i` – (default: 0) an element of the index set; the descent checked for

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: gq=p.grassmannian_quotient()
sage: gq
(Type A affine permutation with window [-1, 0, 3, 4, 5, 6, 9, 10],
 Type A affine permutation with window [3, 1, 2, 6, 5, 4, 8, 7])
sage: gq[0].is_i_grassmannian()
True
sage: 0 not in gq[1].reduced_word()
True
sage: prod(gq)==p
True

sage: gqLeft=p.grassmannian_quotient(side='left')
sage: 0 not in gqLeft[0].reduced_word()
True
sage: gqLeft[1].is_i_grassmannian(side='left')
True
sage: prod(gqLeft)==p
True
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
→Integer(4), Integer(10), Integer(9)])
>>> gq=p.grassmannian_quotient()
>>> gq
(Type A affine permutation with window [-1, 0, 3, 4, 5, 6, 9, 10],
 Type A affine permutation with window [3, 1, 2, 6, 5, 4, 8, 7])
>>> gq[Integer(0)].is_i_grassmannian()
True
>>> Integer(0) not in gq[Integer(1)].reduced_word()
True
>>> prod(gq)==p
True

>>> gqLeft=p.grassmannian_quotient(side='left')
>>> Integer(0) not in gqLeft[Integer(0)].reduced_word()
True
>>> gqLeft[Integer(1)].is_i_grassmannian(side='left')
True
>>> prod(gqLeft)==p
True
```

index_set()

Index set of the affine permutation group.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: A.index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> A.index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

is_i_grassmannian(*i=0*, *side='right'*)

Test whether `self` is i-grassmannian, i.e., either is the identity or has `i` as the sole descent.

INPUT:

- `i` – an element of the index set
- `side` – determines the side on which to check the descents

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.is_i_grassmannian()
False
sage: q=A.from_word([3,2,1,0])
sage: q.is_i_grassmannian()
True
sage: q=A.from_word([2,3,4,5])
sage: q.is_i_grassmannian(5)
True
sage: q.is_i_grassmannian(2, side='left')
True
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.is_i_grassmannian()
False
>>> q=A.from_word([Integer(3),Integer(2),Integer(1),Integer(0)])
>>> q.is_i_grassmannian()
True
>>> q=A.from_word([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> q.is_i_grassmannian(Integer(5))
True
>>> q.is_i_grassmannian(Integer(2), side='left')
True
```

is_one()

Test whether the affine permutation is the identity.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.is_one()
False
sage: q=A.one()
```

(continues on next page)

(continued from previous page)

```
sage: q.is_one()
True
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.is_one()
False
>>> q=A.one()
>>> q.is_one()
True
```

lower_covers(*side='right'*)

Return lower covers of `self`.

The set of affine permutations of one less length related by multiplication by a simple transposition on the indicated side. These are the elements that `self` covers in weak order.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.lower_covers()
[Type A affine permutation with window [-1, 3, 0, 6, 5, 4, 10, 9],
 Type A affine permutation with window [3, -1, 0, 5, 6, 4, 10, 9],
 Type A affine permutation with window [3, -1, 0, 6, 4, 5, 10, 9],
 Type A affine permutation with window [3, -1, 0, 6, 5, 4, 9, 10]]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.lower_covers()
[Type A affine permutation with window [-1, 3, 0, 6, 5, 4, 10, 9],
 Type A affine permutation with window [3, -1, 0, 5, 6, 4, 10, 9],
 Type A affine permutation with window [3, -1, 0, 6, 4, 5, 10, 9],
 Type A affine permutation with window [3, -1, 0, 6, 5, 4, 9, 10]]
```

reduced_word()

Return a reduced word for the affine permutation.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.reduced_word()
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
```

(continues on next page)

(continued from previous page)

```
>>> p.reduced_word()
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
```

signature()

Signature of the affine permutation, $(-1)^l$, where l is the length of the permutation.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.signature()
1
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.signature()
1
```

to_weyl_group_element()

The affine Weyl group element corresponding to the affine permutation.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.to_weyl_group_element()
[ 0 -1  0  1  0  0  1  0]
[ 1 -1  0  1  0  0  1 -1]
[ 1 -1  0  1  0  0  0  0]
[ 0  0  0  1  0  0  0  0]
[ 0  0  0  1  0 -1  1  0]
[ 0  0  0  1 -1  0  1  0]
[ 0  0  0  0  0  0  1  0]
[ 0 -1  1  0  0  0  1  0]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.to_weyl_group_element()
[ 0 -1  0  1  0  0  1  0]
[ 1 -1  0  1  0  0  1 -1]
[ 1 -1  0  1  0  0  0  0]
[ 0  0  0  1  0  0  0  0]
[ 0  0  0  1  0 -1  1  0]
[ 0  0  0  1 -1  0  1  0]
[ 0  0  0  0  0  0  1  0]
[ 0 -1  1  0  0  0  1  0]
```

sage.combinat.affine_permutation.**AffinePermutationGroup**(*cartan_type*)

Wrapper function for specific affine permutation groups.

These are combinatorial implementations of the affine Weyl groups of types A, B, C, D, and G as permutations of the set of all integers. the basic algorithms are derived from [BB2005] and [Eri1995].

EXAMPLES:

```
sage: ct = CartanType(['A',7,1])
sage: A = AffinePermutationGroup(ct)
sage: A
The group of affine permutations of type ['A', 7, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(7),Integer(1)])
>>> A = AffinePermutationGroup(ct)
>>> A
The group of affine permutations of type ['A', 7, 1]
```

We define an element of `A`:

```
sage: p = A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

```
>>> from sage.all import *
>>> p = A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

We find the value $p(1)$, considering p as a bijection on the integers. This is the same as calling the `value()` method:

```
sage: p.value(1)
3
sage: p(1) == p.value(1)
True
```

```
>>> from sage.all import *
>>> p.value(Integer(1))
3
>>> p(Integer(1)) == p.value(Integer(1))
True
```

We can also find the position of the integer 3 in p considered as a sequence, equivalent to finding $p^{-1}(3)$:

```
sage: p.position(3)
1
sage: (p^-1)(3)
1
```

```
>>> from sage.all import *
>>> p.position(Integer(3))
1
>>> (p**-Integer(1))(Integer(3))
1
```

Since the affine permutation group is a group, we demonstrate its group properties:

```
sage: A.one()
Type A affine permutation with window [1, 2, 3, 4, 5, 6, 7, 8]

sage: q = A([0, 2, 3, 4, 5, 6, 7, 9])
sage: p * q
Type A affine permutation with window [1, -1, 0, 6, 5, 4, 10, 11]
sage: q * p
Type A affine permutation with window [3, -1, 1, 6, 5, 4, 10, 8]

sage: p^-1
Type A affine permutation with window [0, -1, 1, 6, 5, 4, 10, 11]
sage: p^-1 * p == A.one()
True
sage: p * p^-1 == A.one()
True
```

```
>>> from sage.all import *
>>> A.one()
Type A affine permutation with window [1, 2, 3, 4, 5, 6, 7, 8]

>>> q = A([Integer(0), Integer(2), Integer(3), Integer(4), Integer(5), Integer(6),
↪ Integer(7), Integer(9)])
>>> p * q
Type A affine permutation with window [1, -1, 0, 6, 5, 4, 10, 11]
>>> q * p
Type A affine permutation with window [3, -1, 1, 6, 5, 4, 10, 8]

>>> p**-Integer(1)
Type A affine permutation with window [0, -1, 1, 6, 5, 4, 10, 11]
>>> p**-Integer(1) * p == A.one()
True
>>> p * p**-Integer(1) == A.one()
True
```

If we decide we prefer the Weyl Group implementation of the affine Weyl group, we can easily get it:

```
sage: p.to_weyl_group_element()
[ 0 -1  0  1  0  0  1  0]
[ 1 -1  0  1  0  0  1 -1]
[ 1 -1  0  1  0  0  0  0]
[ 0  0  0  1  0  0  0  0]
[ 0  0  0  1  0 -1  1  0]
[ 0  0  0  1 -1  0  1  0]
[ 0  0  0  0  0  0  1  0]
[ 0 -1  1  0  0  0  1  0]
```

```
>>> from sage.all import *
>>> p.to_weyl_group_element()
[ 0 -1  0  1  0  0  1  0]
[ 1 -1  0  1  0  0  1 -1]
[ 1 -1  0  1  0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  1  0  0  0  0]
[ 0  0  0  1  0 -1  1  0]
[ 0  0  0  1 -1  0  1  0]
[ 0  0  0  0  0  0  1  0]
[ 0 -1  1  0  0  0  1  0]
```

We can find a reduced word and do all of the other things one expects in a Coxeter group:

```
sage: p.has_right_descent(1)
True
sage: p.apply_simple_reflection(1)
Type A affine permutation with window [-1, 3, 0, 6, 5, 4, 10, 9]
sage: p.apply_simple_reflection(0)
Type A affine permutation with window [1, -1, 0, 6, 5, 4, 10, 11]
sage: p.reduced_word()
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
sage: p.length()
10
```

```
>>> from sage.all import *
>>> p.has_right_descent(Integer(1))
True
>>> p.apply_simple_reflection(Integer(1))
Type A affine permutation with window [-1, 3, 0, 6, 5, 4, 10, 9]
>>> p.apply_simple_reflection(Integer(0))
Type A affine permutation with window [1, -1, 0, 6, 5, 4, 10, 11]
>>> p.reduced_word()
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
>>> p.length()
10
```

The following methods are particular to type A. We can check if the element is fully commutative:

```
sage: p.is_fully_commutative()
False
sage: q.is_fully_commutative()
True
```

```
>>> from sage.all import *
>>> p.is_fully_commutative()
False
>>> q.is_fully_commutative()
True
```

We can also compute the affine Lehmer code of the permutation, a weak composition with $k + 1$ entries:

```
sage: p.to_lehmer_code()
[0, 3, 3, 0, 1, 2, 0, 1]
```

```
>>> from sage.all import *
>>> p.to_lehmer_code()
[0, 3, 3, 0, 1, 2, 0, 1]
```

Once we have the Lehmer code, we can obtain a k-bounded partition by sorting the Lehmer code, and then reading the row lengths. There is a unique 0-Grassmanian (dominant) affine permutation associated to this k-bounded partition, and a k-core as well.

```
sage: p.to_bounded_partition()
[5, 3, 2]
sage: p.to_dominant()
Type A affine permutation with window [-2, -1, 1, 3, 4, 8, 10, 13]
sage: p.to_core()
[5, 3, 2]
```

```
>>> from sage.all import *
>>> p.to_bounded_partition()
[5, 3, 2]
>>> p.to_dominant()
Type A affine permutation with window [-2, -1, 1, 3, 4, 8, 10, 13]
>>> p.to_core()
[5, 3, 2]
```

Finally, we can take a reduced word for p and insert it to find a standard composition tableau associated uniquely to that word:

```
sage: p.tableau_of_word(p.reduced_word())
[[], [1, 6, 9], [2, 7, 10], [], [3], [4, 8], [], [5]]
```

```
>>> from sage.all import *
>>> p.tableau_of_word(p.reduced_word())
[[], [1, 6, 9], [2, 7, 10], [], [3], [4, 8], [], [5]]
```

We can also form affine permutation groups in types B, C, D, and G:

```
sage: B = AffinePermutationGroup(['B',4,1])
sage: B.an_element()
Type B affine permutation with window [-1, 3, 4, 11]

sage: C = AffinePermutationGroup(['C',4,1])
sage: C.an_element()
Type C affine permutation with window [2, 3, 4, 10]

sage: D = AffinePermutationGroup(['D',4,1])
sage: D.an_element()
Type D affine permutation with window [-1, 3, 11, 5]

sage: G = AffinePermutationGroup(['G',2,1])
sage: G.an_element()
Type G affine permutation with window [0, 4, -1, 8, 3, 7]
```

```
>>> from sage.all import *
>>> B = AffinePermutationGroup(['B',Integer(4),Integer(1)])
>>> B.an_element()
Type B affine permutation with window [-1, 3, 4, 11]

>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> C.an_element()
Type C affine permutation with window [2, 3, 4, 10]

>>> D = AffinePermutationGroup(['D',Integer(4),Integer(1)])
>>> D.an_element()
Type D affine permutation with window [-1, 3, 11, 5]

>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> G.an_element()
Type G affine permutation with window [0, 4, -1, 8, 3, 7]
```

class sage.combinat.affine_permutation.**AffinePermutationGroupGeneric**(*cartan_type*)

Bases: UniqueRepresentation, Parent

The generic affine permutation group class, in which we define all type-free methods for the specific affine permutation groups.

cartan_matrix()

Return the Cartan matrix of self.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).cartan_matrix()
[ 2 -1  0  0  0  0  0 -1]
[-1  2 -1  0  0  0  0  0]
[ 0 -1  2 -1  0  0  0  0]
[ 0  0 -1  2 -1  0  0  0]
[ 0  0  0 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[-1  0  0  0  0  0 -1  2]
```

```
>>> from sage.all import *
>>> AffinePermutationGroup(['A',Integer(7),Integer(1)]).cartan_matrix()
[ 2 -1  0  0  0  0  0 -1]
[-1  2 -1  0  0  0  0  0]
[ 0 -1  2 -1  0  0  0  0]
[ 0  0 -1  2 -1  0  0  0]
[ 0  0  0 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[-1  0  0  0  0  0 -1  2]
```

cartan_type()

Return the Cartan type of self.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).cartan_type()
['A', 7, 1]
```

```
>>> from sage.all import *
>>> AffinePermutationGroup(['A',Integer(7),Integer(1)]).cartan_type()
['A', 7, 1]
```

classical()

Return the finite permutation group.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: A.classical()
Symmetric group of order 8! as a permutation group
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> A.classical()
Symmetric group of order 8! as a permutation group
```

from_word(*w*)

Build an affine permutation from a given word. Note: Already in category as `from_reduced_word`, but this is less typing!

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: A.from_word([0, 7, 4, 1, 0, 7, 5, 4, 2, 1])
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),
↪Integer(4), Integer(10), Integer(9)])
>>> A.from_word([Integer(0), Integer(7), Integer(4), Integer(1), Integer(0),
↪Integer(7), Integer(5), Integer(4), Integer(2), Integer(1)])
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

index_set()

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

```
>>> from sage.all import *
>>> AffinePermutationGroup(['A',Integer(7),Integer(1)]).index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

is_crystallographic()

Tells whether the affine permutation group is crystallographic.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).is_crystallographic()
True
```

```
>>> from sage.all import *
>>> AffinePermutationGroup(['A',Integer(7),Integer(1)]).is_crystallographic()
True
```

random_element(*n=None*)

Return a random affine permutation of length `n`.

If `n` is not specified, then `n` is chosen as a random nonnegative integer in $[0, 1000]$.

Starts at the identity, then chooses an upper cover at random. Not very uniform: actually constructs a uniformly random reduced word of length n. Thus we most likely get elements with lots of reduced words!

For the actual code, see `sage.categories.coxeter_group.random_element_of_length()`.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: A.random_element() # random
Type A affine permutation with window [-12, 16, 19, -1, -2, 10, -3, 9]
sage: p = A.random_element(10)
sage: p.length() == 10
True
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> A.random_element() # random
Type A affine permutation with window [-12, 16, 19, -1, -2, 10, -3, 9]
>>> p = A.random_element(Integer(10))
>>> p.length() == Integer(10)
True
```

rank()

Rank of the affine permutation group, equal to $k + 1$.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).rank()
8
```

```
>>> from sage.all import *
>>> AffinePermutationGroup(['A',Integer(7),Integer(1)]).rank()
8
```

reflection_index_set()

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).reflection_index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

```
>>> from sage.all import *
>>> AffinePermutationGroup(['A',Integer(7),Integer(1)]).reflection_index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

weyl_group()

Return the Weyl Group of the same type as `self`.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: A.weyl_group()
Weyl Group of type ['A', 7, 1] (as a matrix group acting on the root space)
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> A.weyl_group()
Weyl Group of type ['A', 7, 1] (as a matrix group acting on the root space)
```

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeA**(*cartan_type*)

Bases: *AffinePermutationGroupGeneric*

Element

alias of *AffinePermutationTypeA*

from_lehmer_code(*C*, *typ='decreasing'*, *side='right'*)

Return the affine permutation with the supplied Lehmer code (a weak composition with $k+1$ parts, at least one of which is 0).

INPUT:

- `typ` – `'increasing'` or `'decreasing'` (default: `'decreasing'`); type of product
- `side` – `'right'` or `'left'` (default: `'right'`); whether the decomposition is from the right or left

EXAMPLES:

```
sage: import itertools
sage: A = AffinePermutationGroup(['A',7,1])
sage: p = A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.to_lehmer_code()
[0, 3, 3, 0, 1, 2, 0, 1]
sage: A.from_lehmer_code(p.to_lehmer_code()) == p
True
sage: orders = ('increasing','decreasing')
sage: sides = ('left','right')
sage: all(A.from_lehmer_code(p.to_lehmer_code(o,s),o,s) == p
....:     for o,s in itertools.product(orders,sides))
True
```

```
>>> from sage.all import *
>>> import itertools
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p = A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),
→Integer(4), Integer(10), Integer(9)])
>>> p.to_lehmer_code()
[0, 3, 3, 0, 1, 2, 0, 1]
>>> A.from_lehmer_code(p.to_lehmer_code()) == p
True
>>> orders = ('increasing','decreasing')
>>> sides = ('left','right')
>>> all(A.from_lehmer_code(p.to_lehmer_code(o,s),o,s) == p
...     for o,s in itertools.product(orders,sides))
True
```

one()

Return the identity element.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).one()
Type A affine permutation with window [1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> from sage.all import *
>>> AffinePermutationGroup(['A',Integer(7),Integer(1)]).one()
Type A affine permutation with window [1, 2, 3, 4, 5, 6, 7, 8]
```

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeB**(*cartan_type*)

Bases: *AffinePermutationGroupTypeC*

Element

alias of *AffinePermutationTypeB*

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeC**(*cartan_type*)

Bases: *AffinePermutationGroupGeneric*

Element

alias of *AffinePermutationTypeC*

one()

Return the identity element.

EXAMPLES:

```
sage: ct=CartanType(['C',4,1])
sage: C = AffinePermutationGroup(ct)
sage: C.one()
Type C affine permutation with window [1, 2, 3, 4]
sage: C.one()*C.one()==C.one()
True
```

```
>>> from sage.all import *
>>> ct=CartanType(['C',Integer(4),Integer(1)])
>>> C = AffinePermutationGroup(ct)
>>> C.one()
Type C affine permutation with window [1, 2, 3, 4]
>>> C.one()*C.one()==C.one()
True
```

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeD**(*cartan_type*)

Bases: *AffinePermutationGroupTypeC*

Element

alias of *AffinePermutationTypeD*

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeG**(*cartan_type*)

Bases: *AffinePermutationGroupGeneric*

Element

alias of *AffinePermutationTypeG*

one()

Return the identity element.

EXAMPLES:

```
sage: AffinePermutationGroup(['G',2,1]).one()
Type G affine permutation with window [1, 2, 3, 4, 5, 6]
```

```
>>> from sage.all import *
>>> AffinePermutationGroup(['G',Integer(2),Integer(1)]).one()
Type G affine permutation with window [1, 2, 3, 4, 5, 6]
```

class sage.combinat.affine_permutation.**AffinePermutationTypeA**(*parent*, *lst*, *check=True*)

Bases: `AffinePermutation`

apply_simple_reflection_left(*i*)

Apply the simple reflection to the values i, $i + 1$.

EXAMPLES:

```
sage: p = AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.apply_simple_reflection_left(3)
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
sage: p.apply_simple_reflection_left(11)
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
```

```
>>> from sage.all import *
>>> p = AffinePermutationGroup(['A',Integer(7),Integer(1)])([Integer(3), -
↪Integer(1), Integer(0), Integer(6), Integer(5), Integer(4), Integer(10),␣
↪Integer(9)])
>>> p.apply_simple_reflection_left(Integer(3))
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
>>> p.apply_simple_reflection_left(Integer(11))
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
```

apply_simple_reflection_right(*i*)

Apply the simple reflection to positions i, $i + 1$.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: p = AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.apply_simple_reflection_right(3)
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
sage: p.apply_simple_reflection_right(11)
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
```

```
>>> from sage.all import *
>>> p = AffinePermutationGroup(['A',Integer(7),Integer(1)])([Integer(3), -
↪Integer(1), Integer(0), Integer(6), Integer(5), Integer(4), Integer(10),␣
↪Integer(9)])
>>> p.apply_simple_reflection_right(Integer(3))
```

(continues on next page)

(continued from previous page)

```
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
>>> p.apply_simple_reflection_right(Integer(11))
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p = A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
sage: q = A([1,2,3])  # indirect doctest
Traceback (most recent call last):
...
ValueError: length of list must be k+1=8
sage: q = A([1,2,3,4,5,6,7,0])  # indirect doctest
Traceback (most recent call last):
...
ValueError: window does not sum to 36
sage: q = A([1,1,3,4,5,6,7,9])  # indirect doctest
Traceback (most recent call last):
...
ValueError: entries must have distinct residues
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p = A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
>>> q = A([Integer(1),Integer(2),Integer(3)])  # indirect doctest
Traceback (most recent call last):
...
ValueError: length of list must be k+1=8
>>> q = A([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(0)])  # indirect doctest
Traceback (most recent call last):
...
ValueError: window does not sum to 36
>>> q = A([Integer(1),Integer(1),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(9)])  # indirect doctest
Traceback (most recent call last):
...
ValueError: entries must have distinct residues
```

flip_automorphism()

The Dynkin diagram automorphism which fixes s_0 and reverses all other indices.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.flip_automorphism()
Type A affine permutation with window [0, -1, 5, 4, 3, 9, 10, 6]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.flip_automorphism()
Type A affine permutation with window [0, -1, 5, 4, 3, 9, 10, 6]
```

has_left_descent (*i*)

Determine whether there is a descent at `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: p = AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.has_left_descent(1)
True
sage: p.has_left_descent(9)
True
sage: p.has_left_descent(0)
True
```

```
>>> from sage.all import *
>>> p = AffinePermutationGroup(['A',Integer(7),Integer(1)])([Integer(3), -
↪Integer(1), Integer(0), Integer(6), Integer(5), Integer(4), Integer(10),␣
↪Integer(9)])
>>> p.has_left_descent(Integer(1))
True
>>> p.has_left_descent(Integer(9))
True
>>> p.has_left_descent(Integer(0))
True
```

has_right_descent (*i*)

Determine whether there is a descent at `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: p = AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.has_right_descent(1)
True
sage: p.has_right_descent(9)
True
```

(continues on next page)

(continued from previous page)

```
sage: p.has_right_descent(0)
False
```

```
>>> from sage.all import *
>>> p = AffinePermutationGroup(['A',Integer(7),Integer(1)])([Integer(3), -
↪Integer(1), Integer(0), Integer(6), Integer(5), Integer(4), Integer(10),␣
↪Integer(9)])
>>> p.has_right_descent(Integer(1))
True
>>> p.has_right_descent(Integer(9))
True
>>> p.has_right_descent(Integer(0))
False
```

is_fully_commutative()

Determine whether `self` is fully commutative.

This means that it has no reduced word with a braid.

This uses a specific algorithm.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p = A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.is_fully_commutative()
False
sage: q = A([-3, -2, 0, 7, 9, 2, 11, 12])
sage: q.is_fully_commutative()
True
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p = A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.is_fully_commutative()
False
>>> q = A([-Integer(3), -Integer(2), Integer(0), Integer(7), Integer(9),␣
↪Integer(2), Integer(11), Integer(12)])
>>> q.is_fully_commutative()
True
```

maximal_cyclic_decomposition(*typ='decreasing'*, *side='right'*, *verbose=False*)

Find the unique maximal decomposition of `self` into cyclically decreasing/increasing elements.

INPUT:

- `typ` – `'increasing'` or `'decreasing'` (default: `'decreasing'`); chooses whether to find increasing or decreasing sets
- `side` – `'right'` or `'left'` (default: `'right'`) chooses whether to find maximal sets starting from the left or the right
- `verbose` – boolean (default: `False`); print extra information while finding the decomposition

EXAMPLES:

```
sage: p = AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.maximal_cyclic_decomposition()
[[0, 7], [4, 1, 0], [7, 5, 4, 2, 1]]
sage: p.maximal_cyclic_decomposition(side='left')
[[1, 0, 7, 5, 4], [1, 0, 5], [2, 1]]
sage: p.maximal_cyclic_decomposition(typ='increasing', side='right')
[[1], [5, 0, 1, 2], [4, 5, 7, 0, 1]]
sage: p.maximal_cyclic_decomposition(typ='increasing', side='left')
[[0, 1, 2, 4, 5], [4, 7, 0, 1], [7]]
```

```
>>> from sage.all import *
>>> p = AffinePermutationGroup(['A',Integer(7),Integer(1)])([Integer(3), -
↪Integer(1), Integer(0), Integer(6), Integer(5), Integer(4), Integer(10),␣
↪Integer(9)])
>>> p.maximal_cyclic_decomposition()
[[0, 7], [4, 1, 0], [7, 5, 4, 2, 1]]
>>> p.maximal_cyclic_decomposition(side='left')
[[1, 0, 7, 5, 4], [1, 0, 5], [2, 1]]
>>> p.maximal_cyclic_decomposition(typ='increasing', side='right')
[[1], [5, 0, 1, 2], [4, 5, 7, 0, 1]]
>>> p.maximal_cyclic_decomposition(typ='increasing', side='left')
[[0, 1, 2, 4, 5], [4, 7, 0, 1], [7]]
```

maximal_cyclic_factor(*typ='decreasing'*, *side='right'*, *verbose=False*)

For an affine permutation x, find the unique maximal subset A of the index set such that $x = yd_A$ is a reduced product.

INPUT:

- `typ` – `'increasing'` or `'decreasing'` (default: `'decreasing'`); chooses whether to find increasing or decreasing sets
- `side` – `'right'` or `'left'` (default: `'right'`); chooses whether to find maximal sets starting from the left or the right
- `verbose` – boolean; if `True`, outputs information about how the cyclically increasing element was found

EXAMPLES:

```
sage: p = AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.maximal_cyclic_factor()
[7, 5, 4, 2, 1]
sage: p.maximal_cyclic_factor(side='left')
[1, 0, 7, 5, 4]
sage: p.maximal_cyclic_factor('increasing','right')
[4, 5, 7, 0, 1]
sage: p.maximal_cyclic_factor('increasing','left')
[0, 1, 2, 4, 5]
```

```
>>> from sage.all import *
>>> p = AffinePermutationGroup(['A',Integer(7),Integer(1)])([Integer(3), -
↪Integer(1), Integer(0), Integer(6), Integer(5), Integer(4), Integer(10),␣
↪Integer(9)])
>>> p.maximal_cyclic_factor()
[7, 5, 4, 2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> p.maximal_cyclic_factor(side='left')
[1, 0, 7, 5, 4]
>>> p.maximal_cyclic_factor('increasing','right')
[4, 5, 7, 0, 1]
>>> p.maximal_cyclic_factor('increasing','left')
[0, 1, 2, 4, 5]
```

position(*i*)

Find the position `j` such the `self.value(j) == i`.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p = A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.position(3)
1
sage: p.position(11)
9
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p = A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.position(Integer(3))
1
>>> p.position(Integer(11))
9
```

promotion()

The Dynkin diagram automorphism which sends s_i to s_{i+1}.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p = A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.promotion()
Type A affine permutation with window [2, 4, 0, 1, 7, 6, 5, 11]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p = A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.promotion()
Type A affine permutation with window [2, 4, 0, 1, 7, 6, 5, 11]
```

tableau_of_word(*w*, *typ='decreasing'*, *side='right'*, *alpha=None*)

Finds a tableau on the Lehmer code of `self` corresponding to the given reduced word.

For a full description of this algorithm, see [Den2012].

INPUT:

- `w` – a reduced word for `self`
- `typ` – `'increasing'` or `'decreasing'`; the type of Lehmer code used

- side – 'right' or 'left'
- alpha – a content vector; w should be of type alpha; specifying alpha produces semistandard tableaux

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.tableau_of_word(p.reduced_word())
[[], [1, 6, 9], [2, 7, 10], [], [3], [4, 8], [], [5]]
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: w=p.reduced_word()
sage: w
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
sage: alpha=[5,3,2]
sage: p.tableau_of_word(p.reduced_word(), alpha=alpha)
[[], [1, 2, 3], [1, 2, 3], [], [1], [1, 2], [], [1]]
sage: p.tableau_of_word(p.reduced_word(), side='left')
[[1, 4, 9], [6], [], [], [3, 7], [8], [], [2, 5, 10]]
sage: p.tableau_of_word(p.reduced_word(), typ='increasing', side='right')
[[9, 10], [1, 2], [], [], [3, 4], [8], [], [5, 6, 7]]
sage: p.tableau_of_word(p.reduced_word(), typ='increasing', side='left')
[[1, 2], [4, 5, 6], [9, 10], [], [3], [7, 8], [], []]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.tableau_of_word(p.reduced_word())
[[], [1, 6, 9], [2, 7, 10], [], [3], [4, 8], [], [5]]
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> w=p.reduced_word()
>>> w
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
>>> alpha=[Integer(5),Integer(3),Integer(2)]
>>> p.tableau_of_word(p.reduced_word(), alpha=alpha)
[[], [1, 2, 3], [1, 2, 3], [], [1], [1, 2], [], [1]]
>>> p.tableau_of_word(p.reduced_word(), side='left')
[[1, 4, 9], [6], [], [], [3, 7], [8], [], [2, 5, 10]]
>>> p.tableau_of_word(p.reduced_word(), typ='increasing', side='right')
[[9, 10], [1, 2], [], [], [3, 4], [8], [], [5, 6, 7]]
>>> p.tableau_of_word(p.reduced_word(), typ='increasing', side='left')
[[1, 2], [4, 5, 6], [9, 10], [], [3], [7, 8], [], []]
```

to_bounded_partition(*typ='decreasing'*, *side='right'*)

Return the k-bounded partition associated to the dominant element obtained by sorting the Lehmer code.

INPUT:

- typ – 'increasing' or 'decreasing' (default: 'decreasing'); chooses whether to find increasing or decreasing sets
- side – 'right' or 'left' (default: 'right'); chooses whether to find maximal sets starting from

the left or the right

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',2,1])
sage: p=A.from_lehmer_code([4,1,0])
sage: p.to_bounded_partition()
[2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(2),Integer(1)])
>>> p=A.from_lehmer_code([Integer(4),Integer(1),Integer(0)])
>>> p.to_bounded_partition()
[2, 1, 1, 1]
```

to_core (*typ='decreasing'*, *side='right'*)

Return the core associated to the dominant element obtained by sorting the Lehmer code.

INPUT:

- `typ` – `'increasing'` or `'decreasing'` (default: `'decreasing'`.)
- `side` – `'right'` or `'left'` (default: `'right'`.) Chooses whether to find maximal sets starting from the left or the right

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',2,1])
sage: p=A.from_lehmer_code([4,1,0])
sage: p.to_bounded_partition()
[2, 1, 1, 1]
sage: p.to_core()
[4, 2, 1, 1]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(2),Integer(1)])
>>> p=A.from_lehmer_code([Integer(4),Integer(1),Integer(0)])
>>> p.to_bounded_partition()
[2, 1, 1, 1]
>>> p.to_core()
[4, 2, 1, 1]
```

to_dominant (*typ='decreasing'*, *side='right'*)

Finds the Lehmer code and then sorts it. Returns the affine permutation with the given sorted Lehmer code; this element is 0-dominant.

INPUT:

- `typ` – `'increasing'` or `'decreasing'` (default: `'decreasing'`) chooses whether to find increasing or decreasing sets
- `side` – `'right'` or `'left'` (default: `'right'`) chooses whether to find maximal sets starting from the left or the right

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.to_dominant()
Type A affine permutation with window [-2, -1, 1, 3, 4, 8, 10, 13]
sage: p.to_dominant(typ='increasing', side='left')
Type A affine permutation with window [3, 4, -1, 5, 0, 9, 6, 10]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
>>> p.to_dominant()
Type A affine permutation with window [-2, -1, 1, 3, 4, 8, 10, 13]
>>> p.to_dominant(typ='increasing', side='left')
Type A affine permutation with window [3, 4, -1, 5, 0, 9, 6, 10]
```

to_lehmer_code(*typ='decreasing'*, *side='right'*)

Return the affine Lehmer code.

There are four such codes; the options `typ` and `side` determine which code is generated. The codes generated are the shape of the maximal cyclic decompositions of `self` according to the given `typ` and `side` options.

INPUT:

- `typ` – `'increasing'` or `'decreasing'` (default: `'decreasing'`); chooses whether to find increasing or decreasing sets
- `side` – `'right'` or `'left'` (default: `'right'`) chooses whether to find maximal sets starting from the left or the right

EXAMPLES:

```
sage: import itertools
sage: A = AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: orders = ('increasing','decreasing')
sage: sides = ('left','right')
sage: for o,s in itertools.product(orders, sides):
....:     p.to_lehmer_code(o,s)
[2, 3, 2, 0, 1, 2, 0, 0]
[2, 2, 0, 0, 2, 1, 0, 3]
[3, 1, 0, 0, 2, 1, 0, 3]
[0, 3, 3, 0, 1, 2, 0, 1]
sage: for a in itertools.product(orders, sides):
....:     A.from_lehmer_code(p.to_lehmer_code(a[0],a[1]), a[0],a[1])==p
True
True
True
True
```

```
>>> from sage.all import *
>>> import itertools
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p=A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),␣
↪Integer(4), Integer(10), Integer(9)])
```

(continues on next page)

(continued from previous page)

```
>>> orders = ('increasing','decreasing')
>>> sides = ('left','right')
>>> for o,s in itertools.product(orders, sides):
...   p.to_lehmer_code(o,s)
[2, 3, 2, 0, 1, 2, 0, 0]
[2, 2, 0, 0, 2, 1, 0, 3]
[3, 1, 0, 0, 2, 1, 0, 3]
[0, 3, 3, 0, 1, 2, 0, 1]
>>> for a in itertools.product(orders, sides):
...   A.from_lehmer_code(p.to_lehmer_code(a[Integer(0)],a[Integer(1)]),
↪a[Integer(0)],a[Integer(1)])==p
True
True
True
True
```

to_type_a()

Return an embedding of `self` into the affine permutation group of type A. (For type A, just returns `self`.)

EXAMPLES:

```
sage: p = AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.to_type_a() is p
True
```

```
>>> from sage.all import *
>>> p = AffinePermutationGroup(['A',Integer(7),Integer(1)])([Integer(3), -
↪Integer(1), Integer(0), Integer(6), Integer(5), Integer(4), Integer(10),
↪Integer(9)])
>>> p.to_type_a() is p
True
```

value(*i*, *base_window=False*)

Return the image of the integer `i` under this permutation.

INPUT:

- `base_window` – boolean; indicating whether `i` is in the base window; if `True`, will run a bit faster, but the method will screw up if `i` is not actually in the index set

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: p = A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.value(1)
3
sage: p.value(9)
11
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A',Integer(7),Integer(1)])
>>> p = A([Integer(3), -Integer(1), Integer(0), Integer(6), Integer(5),
↪Integer(4), Integer(10), Integer(9)])
```

(continues on next page)

(continued from previous page)

```
>>> p.value(Integer(1))
3
>>> p.value(Integer(9))
11
```

class sage.combinat.affine_permutation.**AffinePermutationTypeB**(*parent*, *lst*, *check=True*)

Bases: *AffinePermutationTypeC*

apply_simple_reflection_left(*i*)

Apply the simple reflection indexed by i on values.

EXAMPLES:

```
sage: B = AffinePermutationGroup(['B',4,1])
sage: p=B([-5,1,6,-2])
sage: p.apply_simple_reflection_left(0)
Type B affine permutation with window [-5, -2, 6, 1]
sage: p.apply_simple_reflection_left(2)
Type B affine permutation with window [-5, 1, 7, -3]
sage: p.apply_simple_reflection_left(4)
Type B affine permutation with window [-4, 1, 6, -2]
```

```
>>> from sage.all import *
>>> B = AffinePermutationGroup(['B',Integer(4),Integer(1)])
>>> p=B([-Integer(5),Integer(1),Integer(6),-Integer(2)])
>>> p.apply_simple_reflection_left(Integer(0))
Type B affine permutation with window [-5, -2, 6, 1]
>>> p.apply_simple_reflection_left(Integer(2))
Type B affine permutation with window [-5, 1, 7, -3]
>>> p.apply_simple_reflection_left(Integer(4))
Type B affine permutation with window [-4, 1, 6, -2]
```

apply_simple_reflection_right(*i*)

Apply the simple reflection indexed by i on positions.

EXAMPLES:

```
sage: B = AffinePermutationGroup(['B',4,1])
sage: p=B([-5,1,6,-2])
sage: p.apply_simple_reflection_right(1)
Type B affine permutation with window [1, -5, 6, -2]
sage: p.apply_simple_reflection_right(0)
Type B affine permutation with window [-1, 5, 6, -2]
sage: p.apply_simple_reflection_right(4)
Type B affine permutation with window [-5, 1, 6, 11]
```

```
>>> from sage.all import *
>>> B = AffinePermutationGroup(['B',Integer(4),Integer(1)])
>>> p=B([-Integer(5),Integer(1),Integer(6),-Integer(2)])
>>> p.apply_simple_reflection_right(Integer(1))
Type B affine permutation with window [1, -5, 6, -2]
>>> p.apply_simple_reflection_right(Integer(0))
Type B affine permutation with window [-1, 5, 6, -2]
```

(continues on next page)

(continued from previous page)

```
>>> p.apply_simple_reflection_right(Integer(4))
Type B affine permutation with window [-5, 1, 6, 11]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: B = AffinePermutationGroup(['B',4,1])
sage: x = B([-5,1,6,-2])
sage: x
Type B affine permutation with window [-5, 1, 6, -2]
```

```
>>> from sage.all import *
>>> B = AffinePermutationGroup(['B',Integer(4),Integer(1)])
>>> x = B([-Integer(5),Integer(1),Integer(6),-Integer(2)])
>>> x
Type B affine permutation with window [-5, 1, 6, -2]
```

has_left_descent(*i*)

Determines whether there is a descent at `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: B = AffinePermutationGroup(['B',4,1])
sage: p=B([-5,1,6,-2])
sage: [p.has_left_descent(i) for i in B.index_set()]
[True, True, False, False, True]
```

```
>>> from sage.all import *
>>> B = AffinePermutationGroup(['B',Integer(4),Integer(1)])
>>> p=B([-Integer(5),Integer(1),Integer(6),-Integer(2)])
>>> [p.has_left_descent(i) for i in B.index_set()]
[True, True, False, False, True]
```

has_right_descent(*i*)

Determines whether there is a descent at index `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: B = AffinePermutationGroup(['B',4,1])
sage: p = B([-5,1,6,-2])
sage: [p.has_right_descent(i) for i in B.index_set()]
[True, False, False, True, False]
```

```
>>> from sage.all import *
>>> B = AffinePermutationGroup(['B',Integer(4),Integer(1)])
>>> p = B([-Integer(5),Integer(1),Integer(6),-Integer(2)])
>>> [p.has_right_descent(i) for i in B.index_set()]
[True, False, False, True, False]
```

class sage.combinat.affine_permutation.**AffinePermutationTypeC**(*parent*, *lst*, *check=True*)

Bases: `AffinePermutation`

apply_simple_reflection_left(*i*)

Apply the simple reflection indexed by `i` on values.

EXAMPLES:

```
sage: C = AffinePermutationGroup(['C',4,1])
sage: x = C([-1,5,3,7])
sage: for i in C.index_set(): x.apply_simple_reflection_left(i)
Type C affine permutation with window [1, 5, 3, 7]
Type C affine permutation with window [-2, 5, 3, 8]
Type C affine permutation with window [-1, 5, 2, 6]
Type C affine permutation with window [-1, 6, 4, 7]
Type C affine permutation with window [-1, 4, 3, 7]
```

```
>>> from sage.all import *
>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
>>> x = C([-Integer(1),Integer(5),Integer(3),Integer(7)])
>>> for i in C.index_set(): x.apply_simple_reflection_left(i)
Type C affine permutation with window [1, 5, 3, 7]
Type C affine permutation with window [-2, 5, 3, 8]
Type C affine permutation with window [-1, 5, 2, 6]
Type C affine permutation with window [-1, 6, 4, 7]
Type C affine permutation with window [-1, 4, 3, 7]
```

apply_simple_reflection_right(*i*)

Apply the simple reflection indexed by `i` on positions.

EXAMPLES:

```
sage: C = AffinePermutationGroup(['C',4,1])
sage: x=C([-1,5,3,7])
sage: for i in C.index_set(): x.apply_simple_reflection_right(i)
Type C affine permutation with window [1, 5, 3, 7]
Type C affine permutation with window [5, -1, 3, 7]
Type C affine permutation with window [-1, 3, 5, 7]
Type C affine permutation with window [-1, 5, 7, 3]
Type C affine permutation with window [-1, 5, 3, 2]
```

```
>>> from sage.all import *
>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
>>> x=C([-Integer(1),Integer(5),Integer(3),Integer(7)])
>>> for i in C.index_set(): x.apply_simple_reflection_right(i)
Type C affine permutation with window [1, 5, 3, 7]
Type C affine permutation with window [5, -1, 3, 7]
```

(continues on next page)

(continued from previous page)

```
Type C affine permutation with window [-1, 3, 5, 7]
Type C affine permutation with window [-1, 5, 7, 3]
Type C affine permutation with window [-1, 5, 3, 2]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: C = AffinePermutationGroup(['C',4,1])
sage: x = C([-1,5,3,7])
sage: x
Type C affine permutation with window [-1, 5, 3, 7]
```

```
>>> from sage.all import *
>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
>>> x = C([-Integer(1),Integer(5),Integer(3),Integer(7)])
>>> x
Type C affine permutation with window [-1, 5, 3, 7]
```

has_left_descent(*i*)

Determine whether there is a descent at `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: C = AffinePermutationGroup(['C',4,1])
sage: x = C([-1,5,3,7])
sage: for i in C.index_set(): x.has_left_descent(i)
True
False
True
False
True
```

```
>>> from sage.all import *
>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
>>> x = C([-Integer(1),Integer(5),Integer(3),Integer(7)])
>>> for i in C.index_set(): x.has_left_descent(i)
True
False
True
False
True
```

has_right_descent(*i*)

Determine whether there is a descent at index `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: C = AffinePermutationGroup(['C',4,1])
sage: x = C([-1,5,3,7])
sage: for i in C.index_set(): x.has_right_descent(i)
True
False
True
False
True
```

```
>>> from sage.all import *
>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
>>> x = C([-Integer(1),Integer(5),Integer(3),Integer(7)])
>>> for i in C.index_set(): x.has_right_descent(i)
True
False
True
False
True
```

position(i)

Find the position j such the `self.value(j)=i`.

EXAMPLES:

```
sage: C = AffinePermutationGroup(['C',4,1])
sage: x = C.one()
sage: all(x.position(i) == i for i in range(-10,10))
True
```

```
>>> from sage.all import *
>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
>>> x = C.one()
>>> all(x.position(i) == i for i in range(-Integer(10),Integer(10)))
True
```

to_type_a()

Return an embedding of `self` into the affine permutation group of type A.

EXAMPLES:

```
sage: C = AffinePermutationGroup(['C',4,1])
sage: x = C([-1,5,3,7])
sage: x.to_type_a()
Type A affine permutation with window [-1, 5, 3, 7, 2, 6, 4, 10, 9]
```

```
>>> from sage.all import *
>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
>>> x = C([-Integer(1),Integer(5),Integer(3),Integer(7)])
>>> x.to_type_a()
Type A affine permutation with window [-1, 5, 3, 7, 2, 6, 4, 10, 9]
```

value(*i*)

Return the image of the integer `i` under this permutation.

EXAMPLES:

```
sage: C = AffinePermutationGroup(['C',4,1])
sage: x = C.one()
sage: all(x.value(i) == i for i in range(-10,10))
True
```

```
>>> from sage.all import *
>>> C = AffinePermutationGroup(['C',Integer(4),Integer(1)])
>>> x = C.one()
>>> all(x.value(i) == i for i in range(-Integer(10),Integer(10)))
True
```

class sage.combinat.affine_permutation.**AffinePermutationTypeD**(*parent*, *lst*, *check=True*)

Bases: *AffinePermutationTypeC*

apply_simple_reflection_left(*i*)

Apply simple reflection indexed by `i` on values.

EXAMPLES:

```
sage: D = AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: p.apply_simple_reflection_left(0)
Type D affine permutation with window [-2, -6, 5, 1]
sage: p.apply_simple_reflection_left(1)
Type D affine permutation with window [2, -6, 5, -1]
sage: p.apply_simple_reflection_left(4)
Type D affine permutation with window [1, -4, 3, -2]
```

```
>>> from sage.all import *
>>> D = AffinePermutationGroup(['D',Integer(4),Integer(1)])
>>> p=D([Integer(1),-Integer(6),Integer(5),-Integer(2)])
>>> p.apply_simple_reflection_left(Integer(0))
Type D affine permutation with window [-2, -6, 5, 1]
>>> p.apply_simple_reflection_left(Integer(1))
Type D affine permutation with window [2, -6, 5, -1]
>>> p.apply_simple_reflection_left(Integer(4))
Type D affine permutation with window [1, -4, 3, -2]
```

apply_simple_reflection_right(*i*)

Apply the simple reflection indexed by `i` on positions.

EXAMPLES:

```
sage: D = AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: p.apply_simple_reflection_right(0)
Type D affine permutation with window [6, -1, 5, -2]
sage: p.apply_simple_reflection_right(1)
Type D affine permutation with window [-6, 1, 5, -2]
```

(continues on next page)

(continued from previous page)

```
sage: p.apply_simple_reflection_right(4)
Type D affine permutation with window [1, -6, 11, 4]
```

```
>>> from sage.all import *
>>> D = AffinePermutationGroup(['D',Integer(4),Integer(1)])
>>> p=D([Integer(1),-Integer(6),Integer(5),-Integer(2)])
>>> p.apply_simple_reflection_right(Integer(0))
Type D affine permutation with window [6, -1, 5, -2]
>>> p.apply_simple_reflection_right(Integer(1))
Type D affine permutation with window [-6, 1, 5, -2]
>>> p.apply_simple_reflection_right(Integer(4))
Type D affine permutation with window [1, -6, 11, 4]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: D = AffinePermutationGroup(['D',4,1])
sage: p = D([1,-6,5,-2])
sage: p
Type D affine permutation with window [1, -6, 5, -2]
```

```
>>> from sage.all import *
>>> D = AffinePermutationGroup(['D',Integer(4),Integer(1)])
>>> p = D([Integer(1),-Integer(6),Integer(5),-Integer(2)])
>>> p
Type D affine permutation with window [1, -6, 5, -2]
```

has_left_descent(*i*)

Determine whether there is a descent at `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: D = AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: [p.has_left_descent(i) for i in D.index_set()]
[True, True, False, True, True]
```

```
>>> from sage.all import *
>>> D = AffinePermutationGroup(['D',Integer(4),Integer(1)])
>>> p=D([Integer(1),-Integer(6),Integer(5),-Integer(2)])
>>> [p.has_left_descent(i) for i in D.index_set()]
[True, True, False, True, True]
```

has_right_descent(*i*)

Determine whether there is a descent at index `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: D = AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: [p.has_right_descent(i) for i in D.index_set()]
[True, True, False, True, False]
```

```
>>> from sage.all import *
>>> D = AffinePermutationGroup(['D',Integer(4),Integer(1)])
>>> p=D([Integer(1),-Integer(6),Integer(5),-Integer(2)])
>>> [p.has_right_descent(i) for i in D.index_set()]
[True, True, False, True, False]
```

class sage.combinat.affine_permutation.**AffinePermutationTypeG**(*parent*, *lst*, *check=True*)

Bases: *AffinePermutation*

apply_simple_reflection_left(*i*)

Apply simple reflection indexed by i on values.

EXAMPLES:

```
sage: G = AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: p.apply_simple_reflection_left(0)
Type G affine permutation with window [0, 10, -7, 14, -3, 7]
sage: p.apply_simple_reflection_left(1)
Type G affine permutation with window [1, 9, -4, 11, -2, 6]
sage: p.apply_simple_reflection_left(2)
Type G affine permutation with window [3, 11, -5, 12, -4, 4]
```

```
>>> from sage.all import *
>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> p=G([Integer(2), Integer(10), -Integer(5), Integer(12), -Integer(3),␣
↪Integer(5)])
>>> p.apply_simple_reflection_left(Integer(0))
Type G affine permutation with window [0, 10, -7, 14, -3, 7]
>>> p.apply_simple_reflection_left(Integer(1))
Type G affine permutation with window [1, 9, -4, 11, -2, 6]
>>> p.apply_simple_reflection_left(Integer(2))
Type G affine permutation with window [3, 11, -5, 12, -4, 4]
```

apply_simple_reflection_right(*i*)

Apply the simple reflection indexed by `i` on positions.

EXAMPLES:

```
sage: G = AffinePermutationGroup(['G',2,1])
sage: p = G([2, 10, -5, 12, -3, 5])
sage: p.apply_simple_reflection_right(0)
Type G affine permutation with window [-9, -1, -5, 12, 8, 16]
sage: p.apply_simple_reflection_right(1)
Type G affine permutation with window [10, 2, 12, -5, 5, -3]
sage: p.apply_simple_reflection_right(2)
Type G affine permutation with window [2, -5, 10, -3, 12, 5]
```

```
>>> from sage.all import *
>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> p = G([Integer(2), Integer(10), -Integer(5), Integer(12), -Integer(3),␣
↪Integer(5)])
>>> p.apply_simple_reflection_right(Integer(0))
Type G affine permutation with window [-9, -1, -5, 12, 8, 16]
>>> p.apply_simple_reflection_right(Integer(1))
Type G affine permutation with window [10, 2, 12, -5, 5, -3]
>>> p.apply_simple_reflection_right(Integer(2))
Type G affine permutation with window [2, -5, 10, -3, 12, 5]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: G = AffinePermutationGroup(['G',2,1])
sage: p = G([2, 10, -5, 12, -3, 5])
sage: p
Type G affine permutation with window [2, 10, -5, 12, -3, 5]
```

```
>>> from sage.all import *
>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> p = G([Integer(2), Integer(10), -Integer(5), Integer(12), -Integer(3),␣
↪Integer(5)])
>>> p
Type G affine permutation with window [2, 10, -5, 12, -3, 5]
```

has_left_descent(*i*)

Determines whether there is a descent at `i`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: G = AffinePermutationGroup(['G',2,1])
sage: p = G([2, 10, -5, 12, -3, 5])
sage: [p.has_left_descent(i) for i in G.index_set()]
[False, True, False]
```

```
>>> from sage.all import *
>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> p = G([Integer(2), Integer(10), -Integer(5), Integer(12), -Integer(3),␣
↪Integer(5)])
>>> [p.has_left_descent(i) for i in G.index_set()]
[False, True, False]
```

has_right_descent(*i*)

Determines whether there is a descent at index i.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: G = AffinePermutationGroup(['G',2,1])
sage: p = G([2, 10, -5, 12, -3, 5])
sage: [p.has_right_descent(i) for i in G.index_set()]
[False, False, True]
```

```
>>> from sage.all import *
>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> p = G([Integer(2), Integer(10), -Integer(5), Integer(12), -Integer(3),␣
↪Integer(5)])
>>> [p.has_right_descent(i) for i in G.index_set()]
[False, False, True]
```

position(*i*)

Find the position `j` such the `self.value(j) == i`.

EXAMPLES:

```
sage: G = AffinePermutationGroup(['G',2,1])
sage: p = G([2, 10, -5, 12, -3, 5])
sage: [p.position(i) for i in p]
[1, 2, 3, 4, 5, 6]
```

```
>>> from sage.all import *
>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> p = G([Integer(2), Integer(10), -Integer(5), Integer(12), -Integer(3),␣
↪Integer(5)])
>>> [p.position(i) for i in p]
[1, 2, 3, 4, 5, 6]
```

to_type_a()

Return an embedding of `self` into the affine permutation group of type A.

EXAMPLES:

```
sage: G = AffinePermutationGroup(['G',2,1])
sage: p = G([2, 10, -5, 12, -3, 5])
sage: p.to_type_a()
Type A affine permutation with window [2, 10, -5, 12, -3, 5]
```

```
>>> from sage.all import *
>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> p = G([Integer(2), Integer(10), -Integer(5), Integer(12), -Integer(3),␣
↪Integer(5)])
>>> p.to_type_a()
Type A affine permutation with window [2, 10, -5, 12, -3, 5]
```

value(*i*, *base_window=False*)

Return the image of the integer `i` under this permutation.

INPUT:

- `base_window` – boolean indicating whether `i` is between 1 and $k+1$; if `True`, will run a bit faster, but the method will screw up if `i` is not actually in the index set

EXAMPLES:

```
sage: G = AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: [p.value(i) for i in [1..12]]
[2, 10, -5, 12, -3, 5, 8, 16, 1, 18, 3, 11]
```

```
>>> from sage.all import *
>>> G = AffinePermutationGroup(['G',Integer(2),Integer(1)])
>>> p=G([Integer(2), Integer(10), -Integer(5), Integer(12), -Integer(3),␣
↪Integer(5)])
>>> [p.value(i) for i in (ellipsis_range(Integer(1),Ellipsis,Integer(12)))]
[2, 10, -5, 12, -3, 5, 8, 16, 1, 18, 3, 11]
```

5.1.3 Algebraic combinatorics

Thematic tutorials

- Algebraic Combinatorics in Sage
- Lie Methods and Related Combinatorics in Sage
- Linear Programming (Mixed Integer)

Enumerated sets of combinatorial objects

- *Enumerated sets of partitions, tableaux, …*
- `GelfandTsetlinPattern`, `GelfandTsetlinPatterns`
- `KnutsonTaoPuzzleSolver`

Groups and Algebras

- Catalog of algebras
- Groups
- `SymmetricGroup`, `CoxeterGroup`, `WeylGroup`
- `PartitionAlgebra`
- `IwahoriHeckeAlgebra`
- `SymmetricGroupAlgebra`
- `NilCoxeterAlgebra`
- `AffineNilTemperleyLiebTypeA`
- *Descent Algebras*
- *Diagram and Partition Algebras*
- *Blob Algebras*

Combinatorial Representation Theory

- *Root Systems*
- *Crystals*

- *Rigged configurations*
- *Cluster algebras and quivers*
- `KazhdanLusztigPolynomial`
- `SymmetricGroupRepresentation`
- *Specht Modules*
- *Yang-Baxter Graphs*
- *Hall Polynomials*
- *Key polynomials*

Operads and their algebras

- *Free Dendriform Algebras*
- *Free Pre-Lie Algebras*
- Free Zinbiel Algebras

5.1.4 Combinatorics

Introductory material

- *Combinatorics quickref*
- *Introduction to combinatorics in Sage*

Thematic indexes

- *Algebraic combinatorics*
  - *Combinatorial Hopf algebras*
  - *Cluster algebras and quivers*
  - *Crystals*
  - *Root Systems*
  - *Symmetric Functions*
  - `FullyCommutativeElements`
- *Counting*
- *Enumerated sets and combinatorial objects*
- *Enumerated sets of partitions, tableaux, …*
- *Finite state machines, automata, transducers*
- *Combinatorial species*
- *Combinatorial designs and incidence structures*
- *Posets*
- *Combinatorics on words*
- *A bijectionist's toolkit*

Utilities

- *Output functions*
- *Rankers*
- `Combinatorial maps`
- *Miscellaneous*

Related topics

- Coding Theory
- Discrete dynamics
- Graph Theory

5.1.5 Alternating Sign Matrices

AUTHORS:

- Mike Hansen (2007): Initial version
- Pierre Cange, Luis Serrano (2012): Added monotone triangles
- Travis Scrimshaw (2013-28-03): Added element class for ASM's and made `MonotoneTriangles` inherit from `GelfandTsetlinPatterns`
- Jessica Striker (2013): Added additional methods
- Vincent Delecroix (2017): cleaning

class `sage.combinat.alternating_sign_matrix.`**`AlternatingSignMatrices`**`(n)`

Bases: `UniqueRepresentation`, `Parent`

Class of all $n \times n$ alternating sign matrices.

An alternating sign matrix of size n is an $n \times n$ matrix of 0's, 1's and -1's such that the sum of each row and column is 1 and the nonzero entries in each row and column alternate in sign.

Alternating sign matrices of size n are in bijection with `monotone triangles` with n rows.

INPUT:

- `n` – integer; the size of the matrices

EXAMPLES:

This will create an instance to manipulate the alternating sign matrices of size 3:

```
sage: A = AlternatingSignMatrices(3)
sage: A
Alternating sign matrices of size 3
sage: A.cardinality()
7
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A
Alternating sign matrices of size 3
>>> A.cardinality()
7
```

Notably, this implementation allows to make a lattice of it:

```
sage: L = A.lattice()
sage: L
Finite lattice containing 7 elements
sage: L.category()
Category of facade finite enumerated lattice posets
```

```
>>> from sage.all import *
>>> L = A.lattice()
>>> L
Finite lattice containing 7 elements
>>> L.category()
Category of facade finite enumerated lattice posets
```

Element

alias of *AlternatingSignMatrix*

cardinality()

Return the cardinality of `self`.

The number of $n \times n$ alternating sign matrices is equal to

$$\prod_{k=0}^{n-1} \frac{(3k+1)!}{(n+k)!} = \frac{1!4!7!10!\cdots(3n-2)!}{n!(n+1)!(n+2)!(n+3)!\cdots(2n-1)!}$$

EXAMPLES:

```
sage: [AlternatingSignMatrices(n).cardinality() for n in range(11)]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

```
>>> from sage.all import *
>>> [AlternatingSignMatrices(n).cardinality() for n in range(Integer(11))]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

cover_relations()

Iterate on the cover relations between the alternating sign matrices.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: for (a,b) in A.cover_relations():
....:     eval('a, b')
(
[1 0 0]  [0 1 0]
[0 1 0]  [1 0 0]
[0 0 1], [0 0 1]
)
(
[1 0 0]  [1 0 0]
[0 1 0]  [0 0 1]
[0 0 1], [0 1 0]
)
(
```

(continues on next page)

(continued from previous page)

```
[0 1 0]   [ 0  1  0]
[1 0 0]   [ 1 -1  1]
[0 0 1], [ 0  1  0]
)
(
[1 0 0]   [ 0  1  0]
[0 0 1]   [ 1 -1  1]
[0 1 0], [ 0  1  0]
)
(
[ 0  1  0]   [0 0 1]
[ 1 -1  1]   [1 0 0]
[ 0  1  0], [0 1 0]
)
(
[ 0  1  0]   [0 1 0]
[ 1 -1  1]   [0 0 1]
[ 0  1  0], [1 0 0]
)
(
[0 0 1]   [0 0 1]
[1 0 0]   [0 1 0]
[0 1 0], [1 0 0]
)
(
[0 1 0]   [0 0 1]
[0 0 1]   [0 1 0]
[1 0 0], [1 0 0]
)
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> for (a,b) in A.cover_relations():
...     eval('a, b')
(
[1 0 0]   [0 1 0]
[0 1 0]   [1 0 0]
[0 0 1], [0 0 1]
)
(
[1 0 0]   [1 0 0]
[0 1 0]   [0 0 1]
[0 0 1], [0 1 0]
)
(
[0 1 0]   [ 0  1  0]
[1 0 0]   [ 1 -1  1]
[0 0 1], [ 0  1  0]
)
(
[1 0 0]   [ 0  1  0]
[0 0 1]   [ 1 -1  1]
```

(continues on next page)

(continued from previous page)

```
[0 1 0], [ 0  1  0]
)
(
[ 0  1  0]  [0 0 1]
[ 1 -1  1]  [1 0 0]
[ 0  1  0], [0 1 0]
)
(
[ 0  1  0]  [0 1 0]
[ 1 -1  1]  [0 0 1]
[ 0  1  0], [1 0 0]
)
(
[0 0 1]  [0 0 1]
[1 0 0]  [0 1 0]
[0 1 0], [1 0 0]
)
(
[0 1 0]  [0 0 1]
[0 0 1]  [0 1 0]
[1 0 0], [1 0 0]
)
```

first()

Return the first alternating sign matrix.

EXAMPLES:

```
sage: AlternatingSignMatrices(5).first()
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

```
>>> from sage.all import *
>>> AlternatingSignMatrices(Integer(5)).first()
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

from_contre_tableau(*comps*)

Return an alternating sign matrix from a contre-tableau.

EXAMPLES:

```
sage: ASM = AlternatingSignMatrices(3)
sage: ASM.from_contre_tableau([[1, 2, 3], [1, 2], [1]])
[0 0 1]
[0 1 0]
[1 0 0]
```

(continues on next page)

(continued from previous page)

```
sage: ASM.from_contre_tableau([[1, 2, 3], [2, 3], [3]])
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> ASM = AlternatingSignMatrices(Integer(3))
>>> ASM.from_contre_tableau([[Integer(1), Integer(2), Integer(3)],␣
↪[Integer(1), Integer(2)], [Integer(1)]])
[0 0 1]
[0 1 0]
[1 0 0]
>>> ASM.from_contre_tableau([[Integer(1), Integer(2), Integer(3)],␣
↪[Integer(2), Integer(3)], [Integer(3)]])
[1 0 0]
[0 1 0]
[0 0 1]
```

from_corner_sum(*corner*)

Return an alternating sign matrix from a corner sum matrix.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A.from_corner_sum(matrix([[0,0,0,0],[0,1,1,1],[0,1,2,2],[0,1,2,3]]))
[1 0 0]
[0 1 0]
[0 0 1]
sage: A.from_corner_sum(matrix([[0,0,0,0],[0,0,1,1],[0,1,1,2],[0,1,2,3]]))
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A.from_corner_sum(matrix([[Integer(0),Integer(0),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(1),Integer(1)],[Integer(0),Integer(1),
↪Integer(2),Integer(2)],[Integer(0),Integer(1),Integer(2),Integer(3)]]))
[1 0 0]
[0 1 0]
[0 0 1]
>>> A.from_corner_sum(matrix([[Integer(0),Integer(0),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1),Integer(1)],[Integer(0),Integer(1),
↪Integer(1),Integer(2)],[Integer(0),Integer(1),Integer(2),Integer(3)]]))
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
```

from_height_function(*height*)

Return an alternating sign matrix from a height function.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A.from_height_function(matrix([[0,1,2,3],[1,2,1,2],[2,3,2,1],[3,2,1,
↪0]]))
[0 0 1]
[1 0 0]
[0 1 0]
sage: A.from_height_function(matrix([[0,1,2,3],[1,2,1,2],[2,1,2,1],[3,2,1,
↪0]]))
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A.from_height_function(matrix([[Integer(0),Integer(1),Integer(2),
↪Integer(3)],[Integer(1),Integer(2),Integer(1),Integer(2)],[Integer(2),
↪Integer(3),Integer(2),Integer(1)],[Integer(3),Integer(2),Integer(1),
↪Integer(0)]]))
[0 0 1]
[1 0 0]
[0 1 0]
>>> A.from_height_function(matrix([[Integer(0),Integer(1),Integer(2),
↪Integer(3)],[Integer(1),Integer(2),Integer(1),Integer(2)],[Integer(2),
↪Integer(1),Integer(2),Integer(1)],[Integer(3),Integer(2),Integer(1),
↪Integer(0)]]))
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
```

from_monotone_triangle(*triangle*, *check=True*)

Return an alternating sign matrix from a monotone triangle.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A.from_monotone_triangle([[3, 2, 1], [2, 1], [1]])
[1 0 0]
[0 1 0]
[0 0 1]
sage: A.from_monotone_triangle([[3, 2, 1], [3, 2], [3]])
[0 0 1]
[0 1 0]
[1 0 0]


sage: A.from_monotone_triangle([[3, 2, 1], [2, 2], [1]])
Traceback (most recent call last):
...
ValueError: not a valid triangle
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A.from_monotone_triangle([[Integer(3), Integer(2), Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(2), Integer(1)], [Integer(1)]])
[1 0 0]
[0 1 0]
[0 0 1]
>>> A.from_monotone_triangle([[Integer(3), Integer(2), Integer(1)],␣
↪[Integer(3), Integer(2)], [Integer(3)]])
[0 0 1]
[0 1 0]
[1 0 0]


>>> A.from_monotone_triangle([[Integer(3), Integer(2), Integer(1)],␣
↪[Integer(2), Integer(2)], [Integer(1)]])
Traceback (most recent call last):
...
ValueError: not a valid triangle
```

gyration_orbit_sizes()

Return the sizes of gyration orbits of `self`.

EXAMPLES:

```
sage: AlternatingSignMatrices(3).gyration_orbit_sizes()
[3, 2, 2]
sage: AlternatingSignMatrices(4).gyration_orbit_sizes()
[4, 8, 2, 8, 8, 8, 2, 2]

sage: A = AlternatingSignMatrices(5)
sage: li = [5,10,10,10,10,10,2,5,10,10,10,10,10,10,10,10,10,10,10,10,
....: 4,10,10,10,10,10,10,4,5,10,10,10,10,10,10,10,2,4,5,10,10,10,10,10,10,
....: 4,5,10,10,2,2]
sage: A.gyration_orbit_sizes() == li
True
```

```
>>> from sage.all import *
>>> AlternatingSignMatrices(Integer(3)).gyration_orbit_sizes()
[3, 2, 2]
>>> AlternatingSignMatrices(Integer(4)).gyration_orbit_sizes()
[4, 8, 2, 8, 8, 8, 2, 2]

>>> A = AlternatingSignMatrices(Integer(5))
>>> li = [Integer(5),Integer(10),Integer(10),Integer(10),Integer(10),
↪Integer(10),Integer(2),Integer(5),Integer(10),Integer(10),Integer(10),
↪Integer(10),Integer(10),Integer(10),Integer(10),Integer(10),Integer(10),
↪Integer(10),Integer(10),Integer(10),
... Integer(4),Integer(10),Integer(10),Integer(10),Integer(10),Integer(10),
↪Integer(10),Integer(4),Integer(5),Integer(10),Integer(10),Integer(10),
↪Integer(10),Integer(10),Integer(10),Integer(10),Integer(2),Integer(4),
↪Integer(5),Integer(10),Integer(10),Integer(10),Integer(10),Integer(10),
↪Integer(10),
... Integer(4),Integer(5),Integer(10),Integer(10),Integer(2),Integer(2)]
>>> A.gyration_orbit_sizes() == li
```

(continues on next page)

(continued from previous page)

```
True
```

gyration_orbits()

Return the list of gyration orbits of `self`.

EXAMPLES:

```
sage: AlternatingSignMatrices(3).gyration_orbits()
((
  [1 0 0]  [0 0 1]  [ 0  1  0]
  [0 1 0]  [0 1 0]  [ 1 -1  1]
  [0 0 1], [1 0 0], [ 0  1  0]
 ),
 (
  [0 1 0]  [1 0 0]
  [1 0 0]  [0 0 1]
  [0 0 1], [0 1 0]
 ),
 (
  [0 0 1]  [0 1 0]
  [1 0 0]  [0 0 1]
  [0 1 0], [1 0 0]
 ))
```

```
>>> from sage.all import *
>>> AlternatingSignMatrices(Integer(3)).gyration_orbits()
((
  [1 0 0]  [0 0 1]  [ 0  1  0]
  [0 1 0]  [0 1 0]  [ 1 -1  1]
  [0 0 1], [1 0 0], [ 0  1  0]
 ),
 (
  [0 1 0]  [1 0 0]
  [1 0 0]  [0 0 1]
  [0 0 1], [0 1 0]
 ),
 (
  [0 0 1]  [0 1 0]
  [1 0 0]  [0 0 1]
  [0 1 0], [1 0 0]
 ))
```

last()

Return the last alternating sign matrix.

EXAMPLES:

```
sage: AlternatingSignMatrices(5).last()
[0 0 0 0 1]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]
[1 0 0 0 0]
```

```
>>> from sage.all import *
>>> AlternatingSignMatrices(Integer(5)).last()
[0 0 0 0 1]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]
[1 0 0 0 0]
```

lattice()

Return the lattice of the alternating sign matrices of size n, created by `LatticePoset`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: L = A.lattice()
sage: L
Finite lattice containing 7 elements
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> L = A.lattice()
>>> L
Finite lattice containing 7 elements
```

matrix_space()

Return the underlying matrix space.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A.matrix_space()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A.matrix_space()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

random_element()

Return a uniformly random alternating sign matrix.

EXAMPLES:

```
sage: AlternatingSignMatrices(7).random_element()   # random
[ 0  0  0  0  1  0  0]
[ 0  0  1  0 -1  0  1]
[ 0  0  0  0  1  0  0]
[ 0  1 -1  0  0  1  0]
[ 1 -1  1  0  0  0  0]
[ 0  0  0  1  0  0  0]
[ 0  1  0  0  0  0  0]
sage: a = AlternatingSignMatrices(5).random_element()
sage: bool(a.number_negative_ones()) or a.is_permutation()
True
```

```
>>> from sage.all import *
>>> AlternatingSignMatrices(Integer(7)).random_element()  # random
[ 0  0  0  0  1  0  0]
[ 0  0  1  0 -1  0  1]
[ 0  0  0  0  1  0  0]
[ 0  1 -1  0  0  1  0]
[ 1 -1  1  0  0  0  0]
[ 0  0  0  1  0  0  0]
[ 0  1  0  0  0  0  0]
>>> a = AlternatingSignMatrices(Integer(5)).random_element()
>>> bool(a.number_negative_ones()) or a.is_permutation()
True
```

This is done using a modified version of Propp and Wilson's "coupling from the past" algorithm. It creates a uniformly random Gelfand-Tsetlin triangle with top row $[n, n-1, \ldots 2, 1]$, and then converts it to an alternating sign matrix.

size()

Return the size of the matrices in `self`.

class sage.combinat.alternating_sign_matrix.**AlternatingSignMatrix**(*parent*, *asm*)

Bases: `Element`

An alternating sign matrix.

An alternating sign matrix is a square matrix of 0's, 1's and -1's such that the sum of each row and column is 1 and the nonzero entries in each row and column alternate in sign.

These were introduced in [MRR1983].

ASM_compatible(B)

Return `True` if `self` and `B` are compatible alternating sign matrices in the sense of [EKLP1992]. (If `self` is of size n, `B` must be of size $n+1$.)

In [EKLP1992], there is a notion of a pair of ASM's with sizes differing by 1 being compatible, in the sense that they can be combined to encode a tiling of the Aztec Diamond.

EXAMPLES:

```
sage: A = AlternatingSignMatrix(matrix([[0,0,1,0],[0,1,-1,1],[1,0,0,0],[0,0,1,
↪0]]))
sage: B = AlternatingSignMatrix(matrix([[0,0,1,0,0],[0,0,0,1,0],[1,0,0,-1,1],
↪[0,1,0,0,0],[0,0,0,1,0]]))
sage: A.ASM_compatible(B)
True
sage: A = AlternatingSignMatrix(matrix([[0,1,0],[1,-1,1],[0,1,0]]))
sage: B = AlternatingSignMatrix(matrix([[0,0,1,0],[0,0,0,1],[1,0,0,0],[0,1,0,
↪0]]))
sage: A.ASM_compatible(B)
False
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix(matrix([[Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(0),Integer(1),-Integer(1),Integer(1)],[Integer(1),
↪Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(1),
↪Integer(0)]]))
```

(continues on next page)

(continued from previous page)

```
>>> B = AlternatingSignMatrix(matrix([[Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(1),Integer(0),Integer(0),-Integer(1),Integer(1)],
↪[Integer(0),Integer(1),Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(0),Integer(1),Integer(0)]]))
>>> A.ASM_compatible(B)
True
>>> A = AlternatingSignMatrix(matrix([[Integer(0),Integer(1),Integer(0)],
↪[Integer(1),-Integer(1),Integer(1)],[Integer(0),Integer(1),Integer(0)]]))
>>> B = AlternatingSignMatrix(matrix([[Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(1)],[Integer(1),
↪Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(1),Integer(0),
↪Integer(0)]]))
>>> A.ASM_compatible(B)
False
```

ASM_compatible_bigger()

Return all ASM's compatible with `self` that are of size one greater than `self`.

Given an $n \times n$ alternating sign matrix A, there are as many ASM's of size $n + 1$ compatible with A as 2 raised to the power of the number of 1s in A [EKLP1992].

EXAMPLES:

```
sage: A = AlternatingSignMatrix([[1,0],[0,1]])
sage: A.ASM_compatible_bigger()
[
[ 0  1  0]  [1 0 0]  [0 1 0]  [1 0 0]
[ 1 -1  1]  [0 0 1]  [1 0 0]  [0 1 0]
[ 0  1  0], [0 1 0], [0 0 1], [0 0 1]
]
sage: B = AlternatingSignMatrix([[0,1],[1,0]])
sage: B.ASM_compatible_bigger()
[
[0 0 1]  [0 0 1]  [0 1 0]  [ 0  1  0]
[0 1 0]  [1 0 0]  [0 0 1]  [ 1 -1  1]
[1 0 0], [0 1 0], [1 0 0], [ 0  1  0]
]

sage: B = AlternatingSignMatrix([[0,1,0],[1,-1,1],[0,1,0]])
sage: len(B.ASM_compatible_bigger()) == 2**4
True
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)]])
>>> A.ASM_compatible_bigger()
[
[ 0  1  0]  [1 0 0]  [0 1 0]  [1 0 0]
[ 1 -1  1]  [0 0 1]  [1 0 0]  [0 1 0]
[ 0  1  0], [0 1 0], [0 0 1], [0 0 1]
]
>>> B = AlternatingSignMatrix([[Integer(0),Integer(1)],[Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)]])
>>> B.ASM_compatible_bigger()
[
[0 0 1]  [0 0 1]  [0 1 0]  [ 0  1  0]
[0 1 0]  [1 0 0]  [0 0 1]  [ 1 -1  1]
[1 0 0], [0 1 0], [1 0 0], [ 0  1  0]
]

>>> B = AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0)],[Integer(1),
↪-Integer(1),Integer(1)],[Integer(0),Integer(1),Integer(0)]])
>>> len(B.ASM_compatible_bigger()) == Integer(2)**Integer(4)
True
```

ASM_compatible_smaller()

Return the list of all ASMs compatible with `self` that are of size one smaller than `self`.

Given an alternating sign matrix A of size n, there are as many ASM's of size $n-1$ compatible with it as 2 raised to the power of the number of -1's in A [EKLP1992].

EXAMPLES:

```
sage: A = AlternatingSignMatrix(matrix([[0,0,1,0],[0,1,-1,1],[1,0,0,0],[0,0,1,
↪0]]))
sage: A.ASM_compatible_smaller()
[
[0 0 1]  [ 0  1  0]
[1 0 0]  [ 1 -1  1]
[0 1 0], [ 0  1  0]
]
sage: B = AlternatingSignMatrix(matrix([[1,0,0],[0,0,1],[0,1,0]]))
sage: B.ASM_compatible_smaller()
[
[1 0]
[0 1]
]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix(matrix([[Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(0),Integer(1),-Integer(1),Integer(1)],[Integer(1),
↪Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(1),
↪Integer(0)]]))
>>> A.ASM_compatible_smaller()
[
[0 0 1]  [ 0  1  0]
[1 0 0]  [ 1 -1  1]
[0 1 0], [ 0  1  0]
]
>>> B = AlternatingSignMatrix(matrix([[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(1),Integer(0)]]))
>>> B.ASM_compatible_smaller()
[
[1 0]
[0 1]
```

(continues on next page)

(continued from previous page)

```
]
```

corner_sum_matrix()

Return the corner sum matrix of `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).corner_sum_matrix()
[0 0 0 0]
[0 1 1 1]
[0 1 2 2]
[0 1 2 3]
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.corner_sum_matrix()
[0 0 0 0]
[0 0 1 1]
[0 1 1 2]
[0 1 2 3]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.corner_sum_matrix()
[0 0 0 0]
[0 0 0 1]
[0 1 1 2]
[0 1 2 3]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
↪Integer(0)],[Integer(0), Integer(0), Integer(1)]]).corner_sum_matrix()
[0 0 0 0]
[0 1 1 1]
[0 1 2 2]
[0 1 2 3]
>>> asm = A([[Integer(0), Integer(1), Integer(0)],[Integer(1), -Integer(1),␣
↪Integer(1)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.corner_sum_matrix()
[0 0 0 0]
[0 0 1 1]
[0 1 1 2]
[0 1 2 3]
>>> asm = A([[Integer(0), Integer(0), Integer(1)],[Integer(1), Integer(0),␣
↪Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.corner_sum_matrix()
[0 0 0 0]
[0 0 0 1]
[0 1 1 2]
[0 1 2 3]
```

gyration()

Return the alternating sign matrix obtained by applying gyration to the height function in bijection with `self`.

Gyration acts on height functions as follows. Go through the entries of the matrix, first those for which the sum of the row and column indices is even, then for those for which it is odd, and increment or decrement the squares by 2 wherever possible such that the resulting matrix is still a height function. Gyration was first defined in [Wie2000] as an action on fully-packed loops.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).gyration()
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.gyration()
[1 0 0]
[0 1 0]
[0 0 1]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.gyration()
[0 1 0]
[0 0 1]
[1 0 0]
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).gyration().gyration()
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).gyration().gyration().gyration()
[1 0 0]
[0 1 0]
[0 0 1]

sage: A = AlternatingSignMatrices(4)
sage: M = A([[0,0,1,0],[1,0,0,0],[0,1,-1,1],[0,0,1,0]])
sage: for i in range(5):
....:     M = M.gyration()
sage: M
[1 0 0 0]
[0 0 0 1]
[0 1 0 0]
[0 0 1 0]

sage: a0 = a = AlternatingSignMatrices(5).random_element()
sage: for i in range(20):
....:     a = a.gyration()
sage: a == a0
True
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
→Integer(0)],[Integer(0), Integer(0), Integer(1)]]).gyration()
[0 0 1]
[0 1 0]
[1 0 0]
```

(continues on next page)

(continued from previous page)

```
>>> asm = A([[Integer(0), Integer(1), Integer(0)],[Integer(1), -Integer(1),␣
↪Integer(1)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.gyration()
[1 0 0]
[0 1 0]
[0 0 1]
>>> asm = A([[Integer(0), Integer(0), Integer(1)],[Integer(1), Integer(0),␣
↪Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.gyration()
[0 1 0]
[0 0 1]
[1 0 0]
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
↪Integer(0)],[Integer(0), Integer(0), Integer(1)]]).gyration().gyration()
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
↪Integer(0)],[Integer(0), Integer(0), Integer(1)]]).gyration().gyration().
↪gyration()
[1 0 0]
[0 1 0]
[0 0 1]

>>> A = AlternatingSignMatrices(Integer(4))
>>> M = A([[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(1),
↪Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(0),Integer(1),Integer(0)]])
>>> for i in range(Integer(5)):
...     M = M.gyration()
>>> M
[1 0 0 0]
[0 0 0 1]
[0 1 0 0]
[0 0 1 0]

>>> a0 = a = AlternatingSignMatrices(Integer(5)).random_element()
>>> for i in range(Integer(20)):
...     a = a.gyration()
>>> a == a0
True
```

gyration_orbit()

Return the gyration orbit of `self` (including `self`).

EXAMPLES:

```
sage: AlternatingSignMatrix([[0,1,0],[1,-1,1],[0,1,0]]).gyration_orbit()
[
[ 0  1  0]  [1 0 0]  [0 0 1]
[ 1 -1  1]  [0 1 0]  [0 1 0]
```

(continues on next page)

(continued from previous page)

```
[ 0  1  0], [0 0 1], [1 0 0]
]

sage: AlternatingSignMatrix([[0,1,0,0],[1,-1,1,0],[0,1,-1,1],[0,0,1,0]]).
↪gyration_orbit()
[
[ 0  1  0  0]  [1 0 0 0]  [ 0  0  1  0]  [0 0 0 1]
[ 1 -1  1  0]  [0 1 0 0]  [ 0  1 -1  1]  [0 0 1 0]
[ 0  1 -1  1]  [0 0 1 0]  [ 1 -1  1  0]  [0 1 0 0]
[ 0  0  1  0], [0 0 0 1], [ 0  1  0  0], [1 0 0 0]
]

sage: len(AlternatingSignMatrix([[0,1,0,0,0,0],[0,0,1,0,0,0],[1,-1,0,0,0,1],
....: [0,1,0,0,0,0],[0,0,0,1,0,0],[0,0,0,0,1,0]]).gyration_orbit())
12
```

```
>>> from sage.all import *
>>> AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0)],[Integer(1),-
↪Integer(1),Integer(1)],[Integer(0),Integer(1),Integer(0)]]).gyration_orbit()
[
[ 0  1  0]  [1 0 0]  [0 0 1]
[ 1 -1  1]  [0 1 0]  [0 1 0]
[ 0  1  0], [0 0 1], [1 0 0]
]

>>> AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0),Integer(0)],
↪[Integer(1),-Integer(1),Integer(1),Integer(0)],[Integer(0),Integer(1),-
↪Integer(1),Integer(1)],[Integer(0),Integer(0),Integer(1),Integer(0)]]).
↪gyration_orbit()
[
[ 0  1  0  0]  [1 0 0 0]  [ 0  0  1  0]  [0 0 0 1]
[ 1 -1  1  0]  [0 1 0 0]  [ 0  1 -1  1]  [0 0 1 0]
[ 0  1 -1  1]  [0 0 1 0]  [ 1 -1  1  0]  [0 1 0 0]
[ 0  0  1  0], [0 0 0 1], [ 0  1  0  0], [1 0 0 0]
]

>>> len(AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0)],[Integer(1),-Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(1)],
... [Integer(0),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(0),Integer(0),Integer(1),Integer(0)]]).
↪gyration_orbit())
12
```

height_function()

Return the height function from `self`.

A height function corresponding to an $n \times n$ ASM is an $(n+1) \times (n+1)$ matrix such that the first row is $0, 1, \ldots, n$, the last row is $n, n-1, \ldots, 1, 0$, and the difference between adjacent entries is 1.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).height_function()
[0 1 2 3]
[1 0 1 2]
[2 1 0 1]
[3 2 1 0]
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.height_function()
[0 1 2 3]
[1 2 1 2]
[2 1 2 1]
[3 2 1 0]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.height_function()
[0 1 2 3]
[1 2 1 2]
[2 3 2 1]
[3 2 1 0]

sage: A = AlternatingSignMatrices(4)
sage: all(A.from_height_function(a.height_function()) == a for a in A)
True
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
→Integer(0)],[Integer(0), Integer(0), Integer(1)]]).height_function()
[0 1 2 3]
[1 0 1 2]
[2 1 0 1]
[3 2 1 0]
>>> asm = A([[Integer(0), Integer(1), Integer(0)],[Integer(1), -Integer(1),␣
→Integer(1)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.height_function()
[0 1 2 3]
[1 2 1 2]
[2 1 2 1]
[3 2 1 0]
>>> asm = A([[Integer(0), Integer(0), Integer(1)],[Integer(1), Integer(0),␣
→Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.height_function()
[0 1 2 3]
[1 2 1 2]
[2 3 2 1]
[3 2 1 0]

>>> A = AlternatingSignMatrices(Integer(4))
>>> all(A.from_height_function(a.height_function()) == a for a in A)
True
```

inversion_number()

Return the inversion number of `self`.

If we denote the entries of the alternating sign matrix as $a_{i,j}$, the inversion number is defined as $\sum_{i>k}\sum_{j<l} a_{i,j}a_{k,l}$. When restricted to permutation matrices, this gives the usual inversion number of the permutation.

This definition is equivalent to the one given in [MRR1983].

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).inversion_number()
0
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.inversion_number()
2
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.inversion_number()
2
sage: P = Permutations(5)
sage: all(p.number_of_inversions()==AlternatingSignMatrix(p.to_matrix()).
↪inversion_number() for p in P)
True
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
↪Integer(0)],[Integer(0), Integer(0), Integer(1)]]).inversion_number()
0
>>> asm = A([[Integer(0), Integer(0), Integer(1)],[Integer(1), Integer(0),␣
↪Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.inversion_number()
2
>>> asm = A([[Integer(0), Integer(1), Integer(0)],[Integer(1), -Integer(1),␣
↪Integer(1)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.inversion_number()
2
>>> P = Permutations(Integer(5))
>>> all(p.number_of_inversions()==AlternatingSignMatrix(p.to_matrix()).
↪inversion_number() for p in P)
True
```

is_permutation()

Return `True` if `self` is a permutation matrix and `False` otherwise.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0,1,0],[1,0,0],[0,0,1]])
sage: asm.is_permutation()
True
sage: asm = A([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.is_permutation()
False
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> asm = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> asm.is_permutation()
True
>>> asm = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(1),Integer(0)]])
>>> asm.is_permutation()
False
```

left_key()

Return the left key of the alternating sign matrix `self`.

The left key of an alternating sign matrix was defined by Lascoux in [Lasc] and is obtained by successively removing all the -1's until what remains is a permutation matrix. This notion corresponds to the notion of left key for semistandard tableaux. So our algorithm proceeds as follows: we map `self` to its corresponding monotone triangle, view that monotone triangle as a semistandard tableau, take its left key, and then map back through monotone triangles to the permutation matrix which is the left key.

See also [Ava2007].

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[0,0,1],[1,0,0],[0,1,0]]).left_key()
[0 0 1]
[1 0 0]
[0 1 0]
sage: t = A([[0,1,0],[1,-1,1],[0,1,0]]).left_key(); t
[1 0 0]
[0 0 1]
[0 1 0]
sage: parent(t)
Alternating sign matrices of size 3
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(0),Integer(0),Integer(1)],[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(0)]]).left_key()
[0 0 1]
[1 0 0]
[0 1 0]
>>> t = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(1),Integer(0)]]).left_key(); t
[1 0 0]
[0 0 1]
[0 1 0]
>>> parent(t)
Alternating sign matrices of size 3
```

left_key_as_permutation()

Return the permutation of the left key of `self`.

See also

- *left_key()*

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[0,0,1],[1,0,0],[0,1,0]]).left_key_as_permutation()
[3, 1, 2]
sage: t = A([[0,1,0],[1,-1,1],[0,1,0]]).left_key_as_permutation(); t
[1, 3, 2]
sage: parent(t)
Standard permutations
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(0),Integer(0),Integer(1)],[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(0)]]).left_key_as_permutation()
[3, 1, 2]
>>> t = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(1),Integer(0)]]).left_key_as_permutation();
↪t
[1, 3, 2]
>>> parent(t)
Standard permutations
```

link_pattern()

Return the link pattern corresponding to the fully packed loop corresponding to `self`.

EXAMPLES:

We can extract the underlying link pattern (a non-crossing partition) from a fully packed loop:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: A.link_pattern()
[(1, 2), (3, 6), (4, 5)]

sage: B = AlternatingSignMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: B.link_pattern()
[(1, 6), (2, 5), (3, 4)]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(0), Integer(1), Integer(0)],
↪[Integer(1), -Integer(1), Integer(1)], [Integer(0), Integer(1),
↪Integer(0)]])
>>> A.link_pattern()
[(1, 2), (3, 6), (4, 5)]

>>> B = AlternatingSignMatrix([[Integer(1), Integer(0), Integer(0)],
↪[Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0), Integer(1)]])
>>> B.link_pattern()
[(1, 6), (2, 5), (3, 4)]
```

number_negative_ones()

Return the number of entries in `self` equal to -1.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0,1,0],[1,0,0],[0,0,1]])
sage: asm.number_negative_ones()
0
sage: asm = A([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.number_negative_ones()
1
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> asm = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> asm.number_negative_ones()
0
>>> asm = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(1),Integer(0)]])
>>> asm.number_negative_ones()
1
```

rotate_ccw()

Return the counterclockwise quarter turn rotation of `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).rotate_ccw()
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.rotate_ccw()
[1 0 0]
[0 0 1]
[0 1 0]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
↪Integer(0)],[Integer(0), Integer(0), Integer(1)]]).rotate_ccw()
[0 0 1]
[0 1 0]
[1 0 0]
>>> asm = A([[Integer(0), Integer(0), Integer(1)],[Integer(1), Integer(0),␣
↪Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.rotate_ccw()
[1 0 0]
[0 0 1]
[0 1 0]
```

rotate_cw()

Return the clockwise quarter turn rotation of self.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).rotate_cw()
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.rotate_cw()
[0 1 0]
[1 0 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
↪Integer(0)],[Integer(0), Integer(0), Integer(1)]]).rotate_cw()
[0 0 1]
[0 1 0]
[1 0 0]
>>> asm = A([[Integer(0), Integer(0), Integer(1)],[Integer(1), Integer(0),␣
↪Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.rotate_cw()
[0 1 0]
[1 0 0]
[0 0 1]
```

to_dyck_word(*algorithm*)

Return a Dyck word determined by the specified algorithm.

The algorithm 'last_diagonal' uses the last diagonal of the monotone triangle corresponding to self. The algorithm 'link_pattern' returns the Dyck word in bijection with the link pattern of the fully packed loop.

Note that these two algorithms in general yield different Dyck words for a given alternating sign matrix.

INPUT:

- algorithm – either 'last_diagonal' or 'link_pattern'

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[0,1,0],[1,0,0],[0,0,1]]).to_dyck_word(algorithm = 'last_diagonal')
[1, 1, 0, 0, 1, 0]
sage: d = A([[0,1,0],[1,-1,1],[0,1,0]]).to_dyck_word(algorithm = 'last_
↪diagonal'); d
[1, 1, 0, 1, 0, 0]
sage: parent(d)
Complete Dyck words
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0,1,0],[1,0,0],[0,0,1]])
sage: asm.to_dyck_word(algorithm = 'link_pattern')
[1, 0, 1, 0, 1, 0]
```

(continues on next page)

(continued from previous page)

```
sage: asm = A([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.to_dyck_word(algorithm = 'link_pattern')
[1, 0, 1, 1, 0, 0]
sage: A = AlternatingSignMatrices(4)
sage: asm = A([[0,0,1,0],[1,0,0,0],[0,1,-1,1],[0,0,1,0]])
sage: asm.to_dyck_word(algorithm = 'link_pattern')
[1, 1, 1, 0, 1, 0, 0, 0]
sage: asm.to_dyck_word()
Traceback (most recent call last):
...
TypeError: ...to_dyck_word() ...argument...
sage: asm.to_dyck_word(algorithm = 'notamethod')
Traceback (most recent call last):
...
ValueError: unknown algorithm 'notamethod'
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1)]]).to_dyck_word(algorithm = 'last_diagonal
↪')
[1, 1, 0, 0, 1, 0]
>>> d = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(1),Integer(0)]]).to_dyck_word(algorithm =
↪'last_diagonal'); d
[1, 1, 0, 1, 0, 0]
>>> parent(d)
Complete Dyck words
>>> A = AlternatingSignMatrices(Integer(3))
>>> asm = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> asm.to_dyck_word(algorithm = 'link_pattern')
[1, 0, 1, 0, 1, 0]
>>> asm = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(1),Integer(0)]])
>>> asm.to_dyck_word(algorithm = 'link_pattern')
[1, 0, 1, 1, 0, 0]
>>> A = AlternatingSignMatrices(Integer(4))
>>> asm = A([[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(1),
↪Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(0),Integer(1),Integer(0)]])
>>> asm.to_dyck_word(algorithm = 'link_pattern')
[1, 1, 1, 0, 1, 0, 0, 0]
>>> asm.to_dyck_word()
Traceback (most recent call last):
...
TypeError: ...to_dyck_word() ...argument...
>>> asm.to_dyck_word(algorithm = 'notamethod')
Traceback (most recent call last):
...
ValueError: unknown algorithm 'notamethod'
```

to_fully_packed_loop()

Return the fully packed loop configuration from `self`.

See also

FullyPackedLoop

EXAMPLES:

```
sage: asm = AlternatingSignMatrix([[1,0,0],[0,1,0],[0,0,1]])
sage: fpl = asm.to_fully_packed_loop()
sage: fpl
    |         |
    |         |
    +    + -- +
    |    |
    |    |
 -- +    +    + --
         |    |
         |    |
    + -- +    +
    |         |
    |         |
```

```
>>> from sage.all import *
>>> asm = AlternatingSignMatrix([[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> fpl = asm.to_fully_packed_loop()
>>> fpl
    |         |
    |         |
    +    + -- +
    |    |
    |    |
 -- +    +    + --
         |    |
         |    |
    + -- +    +
    |         |
    |         |
```

to_matrix()

Return `self` as a regular matrix.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[1, 0, 0],[0, 1, 0],[0, 0, 1]])
sage: m = asm.to_matrix(); m
[1 0 0]
[0 1 0]
[0 0 1]
```

(continues on next page)

(continued from previous page)

```
sage: m.parent()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> asm = A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
→Integer(0)],[Integer(0), Integer(0), Integer(1)]])
>>> m = asm.to_matrix(); m
[1 0 0]
[0 1 0]
[0 0 1]
>>> m.parent()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

to_monotone_triangle()

Return a monotone triangle from `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).to_monotone_triangle()
[[3, 2, 1], [2, 1], [1]]
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.to_monotone_triangle()
[[3, 2, 1], [3, 1], [2]]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.to_monotone_triangle()
[[3, 2, 1], [3, 1], [3]]
sage: A.from_monotone_triangle(asm.to_monotone_triangle()) == asm
True
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
→Integer(0)],[Integer(0), Integer(0), Integer(1)]]).to_monotone_triangle()
[[3, 2, 1], [2, 1], [1]]
>>> asm = A([[Integer(0), Integer(1), Integer(0)],[Integer(1), -Integer(1),␣
→Integer(1)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.to_monotone_triangle()
[[3, 2, 1], [3, 1], [2]]
>>> asm = A([[Integer(0), Integer(0), Integer(1)],[Integer(1), Integer(0),␣
→Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.to_monotone_triangle()
[[3, 2, 1], [3, 1], [3]]
>>> A.from_monotone_triangle(asm.to_monotone_triangle()) == asm
True
```

to_permutation()

Return the corresponding permutation if `self` is a permutation matrix.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0,1,0],[1,0,0],[0,0,1]])
sage: p = asm.to_permutation(); p
[2, 1, 3]
sage: parent(p)
Standard permutations
sage: asm = A([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.to_permutation()
Traceback (most recent call last):
...
ValueError: not a permutation matrix
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> asm = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> p = asm.to_permutation(); p
[2, 1, 3]
>>> parent(p)
Standard permutations
>>> asm = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(1),Integer(0)]])
>>> asm.to_permutation()
Traceback (most recent call last):
...
ValueError: not a permutation matrix
```

to_semistandard_tableau()

Return the semistandard tableau corresponding the monotone triangle corresponding to `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[0,0,1],[1,0,0],[0,1,0]]).to_semistandard_tableau()
[[1, 1, 3], [2, 3], [3]]
sage: t = A([[0,1,0],[1,-1,1],[0,1,0]]).to_semistandard_tableau(); t
[[1, 1, 2], [2, 3], [3]]
sage: parent(t)
Semistandard tableaux
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
>>> A([[Integer(0),Integer(0),Integer(1)],[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(0)]]).to_semistandard_tableau()
[[1, 1, 3], [2, 3], [3]]
>>> t = A([[Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(1)],[Integer(0),Integer(1),Integer(0)]]).to_semistandard_tableau();␣
↪t
[[1, 1, 2], [2, 3], [3]]
>>> parent(t)
Semistandard tableaux
```

to_six_vertex_model()

Return the six vertex model configuration from `self`.

This method calls `sage.combinat.six_vertex_model.from_alternating_sign_matrix()`.

EXAMPLES:

```
sage: asm = AlternatingSignMatrix([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.to_six_vertex_model()
    ^    ^    ^
    |    |    |
--> # -> # <- # <--
    ^    |    ^
    |    V    |
--> # <- # -> # <--
    |    ^    |
    V    |    V
--> # -> # <- # <--
    |    |    |
    V    V    V
```

```
>>> from sage.all import *
>>> asm = AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0)],
↪[Integer(1),-Integer(1),Integer(1)],[Integer(0),Integer(1),Integer(0)]])
>>> asm.to_six_vertex_model()
    ^    ^    ^
    |    |    |
--> # -> # <- # <--
    ^    |    ^
    |    V    |
--> # <- # -> # <--
    |    ^    |
    V    |    V
--> # -> # <- # <--
    |    |    |
    V    V    V
```

transpose()

Return `self` transposed.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).transpose()
[1 0 0]
[0 1 0]
[0 0 1]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.transpose()
[0 1 0]
[0 0 1]
[1 0 0]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrices(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> A([[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1),␣
↪Integer(0)],[Integer(0), Integer(0), Integer(1)]]).transpose()
[1 0 0]
[0 1 0]
[0 0 1]
>>> asm = A([[Integer(0), Integer(0), Integer(1)],[Integer(1), Integer(0),␣
↪Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> asm.transpose()
[0 1 0]
[0 0 1]
[1 0 0]
```

class sage.combinat.alternating_sign_matrix.**ContreTableaux**

Bases: Parent

Factory class for the combinatorial class of contre tableaux of size n.

EXAMPLES:

```
sage: ct4 = ContreTableaux(4); ct4
Contre tableaux of size 4
sage: ct4.cardinality()
42
```

```
>>> from sage.all import *
>>> ct4 = ContreTableaux(Integer(4)); ct4
Contre tableaux of size 4
>>> ct4.cardinality()
42
```

class sage.combinat.alternating_sign_matrix.**ContreTableaux_n**(*n*)

Bases: *ContreTableaux*

cardinality()

EXAMPLES:

```
sage: [ContreTableaux(n).cardinality() for n in range(11)]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

```
>>> from sage.all import *
>>> [ContreTableaux(n).cardinality() for n in range(Integer(11))]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

class sage.combinat.alternating_sign_matrix.**MonotoneTriangles**(*n*)

Bases: *GelfandTsetlinPatternsTopRow*

Monotone triangles with n rows.

A monotone triangle is a number triangle $(a_{i,j})_{1 \leq i \leq n, 1 \leq j \leq i}$ on $\{1, \ldots, n\}$ such that:

- $a_{i,j} < a_{i,j+1}$
- $a_{i+1,j} < a_{i,j} \leq a_{i+1,j+1}$

This notably requires that the bottom column is `[1,...,n]`.

Alternatively a monotone triangle is a strict Gelfand-Tsetlin pattern with top row $(n, \ldots, 2, 1)$.

INPUT:

- `n` – the number of rows in the monotone triangles

EXAMPLES:

This represents the monotone triangles with base `[3,2,1]`:

```
sage: M = MonotoneTriangles(3)
sage: M
Monotone triangles with 3 rows
sage: M.cardinality()
7
```

```
>>> from sage.all import *
>>> M = MonotoneTriangles(Integer(3))
>>> M
Monotone triangles with 3 rows
>>> M.cardinality()
7
```

The monotone triangles are a lattice:

```
sage: M.lattice()
Finite lattice containing 7 elements
```

```
>>> from sage.all import *
>>> M.lattice()
Finite lattice containing 7 elements
```

Monotone triangles can be converted to alternating sign matrices and back:

```
sage: M = MonotoneTriangles(5)
sage: A = AlternatingSignMatrices(5)
sage: all(A.from_monotone_triangle(m).to_monotone_triangle() == m for m in M)
True
```

```
>>> from sage.all import *
>>> M = MonotoneTriangles(Integer(5))
>>> A = AlternatingSignMatrices(Integer(5))
>>> all(A.from_monotone_triangle(m).to_monotone_triangle() == m for m in M)
True
```

cardinality()

Cardinality of `self`.

The number of monotone triangles with n rows is equal to

$$\prod_{k=0}^{n-1} \frac{(3k+1)!}{(n+k)!} = \frac{1!4!7!10!\cdots(3n-2)!}{n!(n+1)!(n+2)!(n+3)!\cdots(2n-1)!}$$

EXAMPLES:

```
sage: M = MonotoneTriangles(4)
sage: M.cardinality()
42
```

```
>>> from sage.all import *
>>> M = MonotoneTriangles(Integer(4))
>>> M.cardinality()
42
```

cover_relations()

Iterate on the cover relations in the set of monotone triangles with n rows.

EXAMPLES:

```
sage: M = MonotoneTriangles(3)
sage: for (a,b) in M.cover_relations():
....:     eval('a, b')
([[3, 2, 1], [2, 1], [1]], [[3, 2, 1], [2, 1], [2]])
([[3, 2, 1], [2, 1], [1]], [[3, 2, 1], [3, 1], [1]])
([[3, 2, 1], [2, 1], [2]], [[3, 2, 1], [3, 1], [2]])
([[3, 2, 1], [3, 1], [1]], [[3, 2, 1], [3, 1], [2]])
([[3, 2, 1], [3, 1], [2]], [[3, 2, 1], [3, 1], [3]])
([[3, 2, 1], [3, 1], [2]], [[3, 2, 1], [3, 2], [2]])
([[3, 2, 1], [3, 1], [3]], [[3, 2, 1], [3, 2], [3]])
([[3, 2, 1], [3, 2], [2]], [[3, 2, 1], [3, 2], [3]])
```

```
>>> from sage.all import *
>>> M = MonotoneTriangles(Integer(3))
>>> for (a,b) in M.cover_relations():
...     eval('a, b')
([[3, 2, 1], [2, 1], [1]], [[3, 2, 1], [2, 1], [2]])
([[3, 2, 1], [2, 1], [1]], [[3, 2, 1], [3, 1], [1]])
([[3, 2, 1], [2, 1], [2]], [[3, 2, 1], [3, 1], [2]])
([[3, 2, 1], [3, 1], [1]], [[3, 2, 1], [3, 1], [2]])
([[3, 2, 1], [3, 1], [2]], [[3, 2, 1], [3, 1], [3]])
([[3, 2, 1], [3, 1], [2]], [[3, 2, 1], [3, 2], [2]])
([[3, 2, 1], [3, 1], [3]], [[3, 2, 1], [3, 2], [3]])
([[3, 2, 1], [3, 2], [2]], [[3, 2, 1], [3, 2], [3]])
```

lattice()

Return the lattice of the monotone triangles with n rows.

EXAMPLES:

```
sage: M = MonotoneTriangles(3)
sage: P = M.lattice()
sage: P
Finite lattice containing 7 elements
```

```
>>> from sage.all import *
>>> M = MonotoneTriangles(Integer(3))
>>> P = M.lattice()
```

(continues on next page)

(continued from previous page)

```
>>> P
Finite lattice containing 7 elements
```

class sage.combinat.alternating_sign_matrix.**TruncatedStaircases**

Bases: Parent

Factory class for the combinatorial class of truncated staircases of size n with last column last_column.

EXAMPLES:

```
sage: t4 = TruncatedStaircases(4, [2,3]); t4
Truncated staircases of size 4 with last column [2, 3]
sage: t4.cardinality()
4
```

```
>>> from sage.all import *
>>> t4 = TruncatedStaircases(Integer(4), [Integer(2),Integer(3)]); t4
Truncated staircases of size 4 with last column [2, 3]
>>> t4.cardinality()
4
```

class sage.combinat.alternating_sign_matrix.**TruncatedStaircases_nlastcolumn**(*n*, *last_column*)

Bases: *TruncatedStaircases*

cardinality()

EXAMPLES:

```
sage: T = TruncatedStaircases(4, [2,3])
sage: T.cardinality()
4
```

```
>>> from sage.all import *
>>> T = TruncatedStaircases(Integer(4), [Integer(2),Integer(3)])
>>> T.cardinality()
4
```

5.1.6 Backtracking

This library contains a generic tool for constructing large sets whose elements can be enumerated by exploring a search space with a (lazy) tree or graph structure.

- *GenericBacktracker*: Depth first search through a tree described by a children function, with branch pruning, etc.

This module has mostly been superseded by RecursivelyEnumeratedSet.

class sage.combinat.backtrack.**GenericBacktracker**(*initial_data*, *initial_state*)

Bases: object

A generic backtrack tool for exploring a search space organized as a tree, with branch pruning, etc.

See also RecursivelyEnumeratedSet_forest for handling simple special cases.

class sage.combinat.backtrack.**PositiveIntegerSemigroup**

Bases: UniqueRepresentation, RecursivelyEnumeratedSet_forest

The commutative additive semigroup of positive integers.

This class provides an example of algebraic structure which inherits from RecursivelyEnumeratedSet_forest. It builds the positive integers a la Peano, and endows it with its natural commutative additive semigroup structure.

EXAMPLES:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
sage: PP.category()
Join of Category of monoids and Category of commutative additive semigroups and␣
↪Category of infinite enumerated sets and Category of facade sets
sage: PP.cardinality()
+Infinity
sage: PP.one()
1
sage: PP.an_element()
1
sage: some_elements = list(PP.some_elements()); some_elements
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,␣
↪23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,␣
↪43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,␣
↪63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,␣
↪83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

```
>>> from sage.all import *
>>> from sage.combinat.backtrack import PositiveIntegerSemigroup
>>> PP = PositiveIntegerSemigroup()
>>> PP.category()
Join of Category of monoids and Category of commutative additive semigroups and␣
↪Category of infinite enumerated sets and Category of facade sets
>>> PP.cardinality()
+Infinity
>>> PP.one()
1
>>> PP.an_element()
1
>>> some_elements = list(PP.some_elements()); some_elements
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,␣
↪23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,␣
↪43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,␣
↪63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,␣
↪83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

children(*x*)

Return the single child x+1 of the integer x.

EXAMPLES:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
```

(continues on next page)

(continued from previous page)

```
sage: list(PP.children(1))
[2]
sage: list(PP.children(42))
[43]
```

```
>>> from sage.all import *
>>> from sage.combinat.backtrack import PositiveIntegerSemigroup
>>> PP = PositiveIntegerSemigroup()
>>> list(PP.children(Integer(1)))
[2]
>>> list(PP.children(Integer(42)))
[43]
```

one()

Return the unit of `self`.

EXAMPLES:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
sage: PP.one()
1
```

```
>>> from sage.all import *
>>> from sage.combinat.backtrack import PositiveIntegerSemigroup
>>> PP = PositiveIntegerSemigroup()
>>> PP.one()
1
```

roots()

Return the single root of `self`.

EXAMPLES:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
sage: list(PP.roots())
[1]
```

```
>>> from sage.all import *
>>> from sage.combinat.backtrack import PositiveIntegerSemigroup
>>> PP = PositiveIntegerSemigroup()
>>> list(PP.roots())
[1]
```

5.1.7 Baxter permutations

class sage.combinat.baxter_permutations.**BaxterPermutations**

Bases: `UniqueRepresentation`, `Parent`

The combinatorial class of Baxter permutations.

A Baxter permutation is a permutation avoiding the generalized permutation patterns $2-41-3$ and $3-14-2$. In other words, a permutation σ is a Baxter permutation if for any subword $u := u_1 u_2 u_3 u_4$ of σ such that the letters u_2 and u_3 are adjacent in σ, the standardized version of u is neither 2413 nor 3142.

See [Gir2012] for a study of Baxter permutations.

INPUT:

- `n` – nonnegative integer (default: `None`); the size of the permutations

OUTPUT:

Return the combinatorial class of the Baxter permutations of size `n` if `n` is not `None`. Otherwise, return the combinatorial class of all Baxter permutations.

EXAMPLES:

```
sage: BaxterPermutations(5)
Baxter permutations of size 5
sage: BaxterPermutations()
Baxter permutations
```

```
>>> from sage.all import *
>>> BaxterPermutations(Integer(5))
Baxter permutations of size 5
>>> BaxterPermutations()
Baxter permutations
```

class sage.combinat.baxter_permutations.**BaxterPermutations_all**(*n=None*)

Bases: `DisjointUnionEnumeratedSets`, *`BaxterPermutations`*

The enumerated set of all Baxter permutations.

See *`BaxterPermutations`* for the definition of Baxter permutations.

EXAMPLES:

```
sage: from sage.combinat.baxter_permutations import BaxterPermutations_all
sage: BaxterPermutations_all()
Baxter permutations
```

```
>>> from sage.all import *
>>> from sage.combinat.baxter_permutations import BaxterPermutations_all
>>> BaxterPermutations_all()
Baxter permutations
```

to_pair_of_twin_binary_trees(*p*)

Apply a bijection between Baxter permutations of size `self._n` and the set of pairs of twin binary trees with `self._n` nodes.

INPUT:

- `p` – a Baxter permutation

OUTPUT:

The pair of twin binary trees (T_L, T_R) where T_L (resp. T_R) is obtained by inserting the letters of `p` from left to right (resp. right to left) following the binary search tree insertion algorithm. This is called the *Baxter P-symbol* in [Gir2012] Definition 4.1.

Note

This method only works when `p` is a permutation. For words with repeated letters, it would return two "right binary search trees" (in the terminology of [Gir2012]), which conflicts with the definition in [Gir2012].

EXAMPLES:

```
sage: BP = BaxterPermutations()
sage: BP.to_pair_of_twin_binary_trees(Permutation([]))                          #␣
↪needs sage.graphs
(., .)
sage: BP.to_pair_of_twin_binary_trees(Permutation([1, 2, 3]))                   #␣
↪needs sage.graphs
(1[., 2[., 3[., .]]], 3[2[1[., .], .], .])
sage: BP.to_pair_of_twin_binary_trees(Permutation([3, 4, 1, 2]))                #␣
↪needs sage.graphs
(3[1[., 2[., .]], 4[., .]], 2[1[., .], 4[3[., .], .]])
```

```
>>> from sage.all import *
>>> BP = BaxterPermutations()
>>> BP.to_pair_of_twin_binary_trees(Permutation([]))                          #␣
↪needs sage.graphs
(., .)
>>> BP.to_pair_of_twin_binary_trees(Permutation([Integer(1), Integer(2),␣
↪Integer(3)]))                 # needs sage.graphs
(1[., 2[., 3[., .]]], 3[2[1[., .], .], .])
>>> BP.to_pair_of_twin_binary_trees(Permutation([Integer(3), Integer(4),␣
↪Integer(1), Integer(2)]))              # needs sage.graphs
(3[1[., 2[., .]], 4[., .]], 2[1[., .], 4[3[., .], .]])
```

class sage.combinat.baxter_permutations.**BaxterPermutations_size**(*n*)

Bases: *BaxterPermutations*

The enumerated set of Baxter permutations of a given size.

See *BaxterPermutations* for the definition of Baxter permutations.

EXAMPLES:

```
sage: from sage.combinat.baxter_permutations import BaxterPermutations_size
sage: BaxterPermutations_size(5)
Baxter permutations of size 5
```

```
>>> from sage.all import *
>>> from sage.combinat.baxter_permutations import BaxterPermutations_size
>>> BaxterPermutations_size(Integer(5))
Baxter permutations of size 5
```

cardinality()

Return the number of Baxter permutations of size `self._n`.

For any positive integer n, the number of Baxter permutations of size n equals

$$\sum_{k=1}^{n} \frac{\binom{n+1}{k-1}\binom{n+1}{k}\binom{n+1}{k+1}}{\binom{n+1}{1}\binom{n+1}{2}}.$$

This is OEIS sequence A001181.

EXAMPLES:

```
sage: [BaxterPermutations(n).cardinality() for n in range(13)]
[1, 1, 2, 6, 22, 92, 422, 2074, 10754, 58202, 326240, 1882960, 11140560]

sage: BaxterPermutations(3r).cardinality()
6
sage: parent(_)
Integer Ring
```

```
>>> from sage.all import *
>>> [BaxterPermutations(n).cardinality() for n in range(Integer(13))]
[1, 1, 2, 6, 22, 92, 422, 2074, 10754, 58202, 326240, 1882960, 11140560]

>>> BaxterPermutations(3).cardinality()
6
>>> parent(_)
Integer Ring
```

5.1.8 A bijectionist's toolkit

AUTHORS:

- Alexander Grosz, Tobias Kietreiber, Stephan Pfannerer and Martin Rubey (2020-2022): Initial version

Quick reference

set_statistics()	Declare statistics that are preserved by the bijection.
set_value_restrictions()	Restrict the values of the statistic on an element.
set_constant_blocks()	Declare that the statistic is constant on some sets.
set_distributions()	Restrict the distribution of values of the statistic on some elements.
set_intertwining_relations()	Declare that the statistic intertwines with other maps.
set_quadratic_relation()	Declare that the statistic satisfies a certain relation.
set_homomesic()	Declare that the statistic is homomesic with respect to a given set partition.
statistics_table()	Print a table collecting information on the given statistics.
statistics_fibers()	Collect elements with the same statistics.
constant_blocks()	Return the blocks on which the statistic is constant.
solutions_iterator()	Iterate over all possible solutions.
possible_values()	Return all possible values for a given element.
minimal_subdistributions_iterator()	Iterate over the minimal subdistributions.
minimal_subdistributions_blocks_iterator()	Iterate over the minimal subdistributions.

A guided tour

Consider the following combinatorial statistics on a permutation:

- wex, the number of weak excedences,
- fix, the number of fixed points,
- des, the number of descents (after appending 0),
- adj, the number of adjacencies (after appending 0), and
- $llis$, the length of a longest increasing subsequence.

Moreover, let rot be action of rotation on a permutation, i.e., the conjugation with the long cycle.

It is known that there must exist a statistic s on permutations, which is equidistributed with $llis$ but additionally invariant under rot. However, at least very small cases do not contradict the possibility that one can even find a statistic s, invariant under rot and such that $(s, wex, fix) \sim (llis, des, adj)$. Let us check this for permutations of size at most 3:

```
sage: N = 3
sage: A = B = [pi for n in range(N+1) for pi in Permutations(n)]
sage: def alpha1(p): return len(p.weak_excedences())
sage: def alpha2(p): return len(p.fixed_points())
sage: def beta1(p): return len(p.descents(final_descent=True)) if p else 0
sage: def beta2(p): return len([e for (e, f) in zip(p, p[1:]+[0]) if e == f+1])
sage: tau = Permutation.longest_increasing_subsequence_length
sage: def rotate_permutation(p):
....:     cycle = Permutation(tuple(range(1, len(p)+1)))
....:     return Permutation([cycle.inverse()(p(cycle(i))) for i in range(1,
↪len(p)+1)])
```

(continues on next page)

(continued from previous page)

```
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_statistics((len, len), (alpha1, beta1), (alpha2, beta2))
sage: a, b = bij.statistics_table()
sage: table(a, header_row=True, frame=True)
┌───────────┬────────┬────────┬────────┐
│ a         │ α_1(a) │ α_2(a) │ α_3(a) │
╞═══════════╪════════╪════════╪════════╡
│ []        │ 0      │ 0      │ 0      │
├───────────┼────────┼────────┼────────┤
│ [1]       │ 1      │ 1      │ 1      │
├───────────┼────────┼────────┼────────┤
│ [1, 2]    │ 2      │ 2      │ 2      │
├───────────┼────────┼────────┼────────┤
│ [2, 1]    │ 2      │ 1      │ 0      │
├───────────┼────────┼────────┼────────┤
│ [1, 2, 3] │ 3      │ 3      │ 3      │
├───────────┼────────┼────────┼────────┤
│ [1, 3, 2] │ 3      │ 2      │ 1      │
├───────────┼────────┼────────┼────────┤
│ [2, 1, 3] │ 3      │ 2      │ 1      │
├───────────┼────────┼────────┼────────┤
│ [2, 3, 1] │ 3      │ 2      │ 0      │
├───────────┼────────┼────────┼────────┤
│ [3, 1, 2] │ 3      │ 1      │ 0      │
├───────────┼────────┼────────┼────────┤
│ [3, 2, 1] │ 3      │ 2      │ 1      │
└───────────┴────────┴────────┴────────┘

sage: table(b, header_row=True, frame=True)
┌───────────┬───┬────────┬────────┬────────┐
│ b         │ τ │ β_1(b) │ β_2(b) │ β_3(b) │
╞═══════════╪═══╪════════╪════════╪════════╡
│ []        │ 0 │ 0      │ 0      │ 0      │
├───────────┼───┼────────┼────────┼────────┤
│ [1]       │ 1 │ 1      │ 1      │ 1      │
├───────────┼───┼────────┼────────┼────────┤
│ [1, 2]    │ 2 │ 2      │ 1      │ 0      │
├───────────┼───┼────────┼────────┼────────┤
│ [2, 1]    │ 1 │ 2      │ 2      │ 2      │
├───────────┼───┼────────┼────────┼────────┤
│ [1, 2, 3] │ 3 │ 3      │ 1      │ 0      │
├───────────┼───┼────────┼────────┼────────┤
│ [1, 3, 2] │ 2 │ 3      │ 2      │ 1      │
├───────────┼───┼────────┼────────┼────────┤
│ [2, 1, 3] │ 2 │ 3      │ 2      │ 1      │
├───────────┼───┼────────┼────────┼────────┤
│ [2, 3, 1] │ 2 │ 3      │ 2      │ 1      │
├───────────┼───┼────────┼────────┼────────┤
│ [3, 1, 2] │ 2 │ 3      │ 2      │ 0      │
├───────────┼───┼────────┼────────┼────────┤
│ [3, 2, 1] │ 1 │ 3      │ 3      │ 3      │
└───────────┴───┴────────┴────────┴────────┘
```

(continues on next page)

(continued from previous page)

```
sage: from sage.combinat.cyclic_sieving_phenomenon import orbit_decomposition
sage: bij.set_constant_blocks(orbit_decomposition(A, rotate_permutation))
sage: bij.constant_blocks()
{{[1, 3, 2], [2, 1, 3], [3, 2, 1]}}
sage: next(bij.solutions_iterator())
{[]: 0,
 [1]: 1,
 [1, 2]: 1,
 [1, 2, 3]: 1,
 [1, 3, 2]: 2,
 [2, 1]: 2,
 [2, 1, 3]: 2,
 [2, 3, 1]: 2,
 [3, 1, 2]: 3,
 [3, 2, 1]: 2}
```

```
>>> from sage.all import *
>>> N = Integer(3)
>>> A = B = [pi for n in range(N+Integer(1)) for pi in Permutations(n)]
>>> def alpha1(p): return len(p.weak_excedences())
>>> def alpha2(p): return len(p.fixed_points())
>>> def beta1(p): return len(p.descents(final_descent=True)) if p else Integer(0)
>>> def beta2(p): return len([e for (e, f) in zip(p, p[Integer(1):]+[Integer(0)]) if␣
↪e == f+Integer(1)])
>>> tau = Permutation.longest_increasing_subsequence_length
>>> def rotate_permutation(p):
...     cycle = Permutation(tuple(range(Integer(1), len(p)+Integer(1))))
...     return Permutation([cycle.inverse()(p(cycle(i))) for i in range(Integer(1),␣
↪len(p)+Integer(1))])
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_statistics((len, len), (alpha1, beta1), (alpha2, beta2))
>>> a, b = bij.statistics_table()
>>> table(a, header_row=True, frame=True)
┌───────────┬────────┬────────┬────────┐
│ a         │ α_1(a) │ α_2(a) │ α_3(a) │
╞═══════════╪════════╪════════╪════════╡
│ []        │ 0      │ 0      │ 0      │
├───────────┼────────┼────────┼────────┤
│ [1]       │ 1      │ 1      │ 1      │
├───────────┼────────┼────────┼────────┤
│ [1, 2]    │ 2      │ 2      │ 2      │
├───────────┼────────┼────────┼────────┤
│ [2, 1]    │ 2      │ 1      │ 0      │
├───────────┼────────┼────────┼────────┤
│ [1, 2, 3] │ 3      │ 3      │ 3      │
├───────────┼────────┼────────┼────────┤
│ [1, 3, 2] │ 3      │ 2      │ 1      │
├───────────┼────────┼────────┼────────┤
│ [2, 1, 3] │ 3      │ 2      │ 1      │
├───────────┼────────┼────────┼────────┤
│ [2, 3, 1] │ 3      │ 2      │ 0      │
```

(continues on next page)

(continued from previous page)

```
├───────────┼───────┼───────┼───────┤
│ [3, 1, 2] │ 3     │ 1     │ 0     │
├───────────┼───────┼───────┼───────┤
│ [3, 2, 1] │ 3     │ 2     │ 1     │
└───────────┴───────┴───────┴───────┘

>>> table(b, header_row=True, frame=True)
┌───────────┬───┬────────┬────────┬────────┐
│ b         │ τ │ β_1(b) │ β_2(b) │ β_3(b) │
╞═══════════╪═══╪════════╪════════╪════════╡
│ []        │ 0 │ 0      │ 0      │ 0      │
├───────────┼───┼────────┼────────┼────────┤
│ [1]       │ 1 │ 1      │ 1      │ 1      │
├───────────┼───┼────────┼────────┼────────┤
│ [1, 2]    │ 2 │ 2      │ 1      │ 0      │
├───────────┼───┼────────┼────────┼────────┤
│ [2, 1]    │ 1 │ 2      │ 2      │ 2      │
├───────────┼───┼────────┼────────┼────────┤
│ [1, 2, 3] │ 3 │ 3      │ 1      │ 0      │
├───────────┼───┼────────┼────────┼────────┤
│ [1, 3, 2] │ 2 │ 3      │ 2      │ 1      │
├───────────┼───┼────────┼────────┼────────┤
│ [2, 1, 3] │ 2 │ 3      │ 2      │ 1      │
├───────────┼───┼────────┼────────┼────────┤
│ [2, 3, 1] │ 2 │ 3      │ 2      │ 1      │
├───────────┼───┼────────┼────────┼────────┤
│ [3, 1, 2] │ 2 │ 3      │ 2      │ 0      │
├───────────┼───┼────────┼────────┼────────┤
│ [3, 2, 1] │ 1 │ 3      │ 3      │ 3      │
└───────────┴───┴────────┴────────┴────────┘

>>> from sage.combinat.cyclic_sieving_phenomenon import orbit_decomposition
>>> bij.set_constant_blocks(orbit_decomposition(A, rotate_permutation))
>>> bij.constant_blocks()
{{[1, 3, 2], [2, 1, 3], [3, 2, 1]}}
>>> next(bij.solutions_iterator())
{[]: 0,
 [1]: 1,
 [1, 2]: 1,
 [1, 2, 3]: 1,
 [1, 3, 2]: 2,
 [2, 1]: 2,
 [2, 1, 3]: 2,
 [2, 3, 1]: 2,
 [3, 1, 2]: 3,
 [3, 2, 1]: 2}
```

On the other hand, we can check that there is no rotation invariant statistic on non-crossing set partitions which is equidistributed with the Strahler number on ordered trees:

```
sage: N = 8
sage: A = [SetPartition(d.to_noncrossing_partition()) for n in range(N) for d in␣
```

(continues on next page)

(continued from previous page)

```
↪DyckWords(n)]
sage: B = [t for n in range(1, N+1) for t in OrderedTrees(n)]
sage: def theta(m): return SetPartition([[i % m.size() + 1 for i in b] for b in m])
```

```
>>> from sage.all import *
>>> N = Integer(8)
>>> A = [SetPartition(d.to_noncrossing_partition()) for n in range(N) for d in␣
↪DyckWords(n)]
>>> B = [t for n in range(Integer(1), N+Integer(1)) for t in OrderedTrees(n)]
>>> def theta(m): return SetPartition([[i % m.size() + Integer(1) for i in b] for b␣
↪in m])
```

Code for the Strahler number can be obtained from FindStat. The following code is equivalent to `tau = findstat(397)`:

```
sage: def tau(T):
....:     if len(T) == 0:
....:         return 1
....:     else:
....:         l = [tau(S) for S in T]
....:         m = max(l)
....:         if l.count(m) == 1:
....:             return m
....:         else:
....:             return m+1
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_statistics((lambda a: a.size(), lambda b: b.node_number()-1))
sage: from sage.combinat.cyclic_sieving_phenomenon import orbit_decomposition
sage: bij.set_constant_blocks(orbit_decomposition(A, theta))
sage: list(bij.solutions_iterator())
[]
```

```
>>> from sage.all import *
>>> def tau(T):
...     if len(T) == Integer(0):
...         return Integer(1)
...     else:
...         l = [tau(S) for S in T]
...         m = max(l)
...         if l.count(m) == Integer(1):
...             return m
...         else:
...             return m+Integer(1)
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_statistics((lambda a: a.size(), lambda b: b.node_number()-Integer(1)))
>>> from sage.combinat.cyclic_sieving_phenomenon import orbit_decomposition
>>> bij.set_constant_blocks(orbit_decomposition(A, theta))
>>> list(bij.solutions_iterator())
[]
```

Next we demonstrate how to search for a bijection, instead An example identifying s and S:

```
sage: N = 4
sage: A = [dyck_word for n in range(1, N) for dyck_word in DyckWords(n)]
sage: B = [binary_tree for n in range(1, N) for binary_tree in BinaryTrees(n)]
sage: concat_path = lambda D1, D2: DyckWord(list(D1) + list(D2))
sage: concat_tree = lambda B1, B2: concat_path(B1.to_dyck_word(),
....:                                          B2.to_dyck_word()).to_binary_tree()
sage: bij = Bijectionist(A, B)
sage: bij.set_intertwining_relations((2, concat_path, concat_tree))
sage: bij.set_statistics((lambda d: d.semilength(), lambda t: t.node_number()))
sage: for D in sorted(bij.minimal_subdistributions_iterator(), key=lambda x:␣
↪(len(x[0][0]), x)):
....:     ascii_art(D)
( [ /\ ], [ o ] )
(           [ o   ] )
(           [  \  ] )
( [ /\/\ ], [   o ] )
(           [   o ] )
( [  /\  ]  [  /  ] )
( [ /  \ ], [ o   ] )
(             [ o     ] )
(             [  \    ] )
(             [   o   ] )
(             [    \  ] )
( [ /\/\/\ ], [     o ] )
(             [ o   ] )
(             [  \  ] )
(             [   o ] )
( [    /\  ]  [  /  ] )
( [ /\/  \ ], [ o   ] )
(             [   o   ] )
( [  /\    ]  [  / \  ] )
( [ /  \/\ ], [ o   o ] )
(                     [   o,      o ] )
(                     [  /      /  ] )
( [           /\   ]  [ o      o   ] )
( [  /\/\    /  \  ]  [  \    /    ] )
( [ /    \, /    \ ], [   o  o     ] )
```

```
>>> from sage.all import *
>>> N = Integer(4)
>>> A = [dyck_word for n in range(Integer(1), N) for dyck_word in DyckWords(n)]
>>> B = [binary_tree for n in range(Integer(1), N) for binary_tree in BinaryTrees(n)]
>>> concat_path = lambda D1, D2: DyckWord(list(D1) + list(D2))
>>> concat_tree = lambda B1, B2: concat_path(B1.to_dyck_word(),
...                                          B2.to_dyck_word()).to_binary_tree()
>>> bij = Bijectionist(A, B)
>>> bij.set_intertwining_relations((Integer(2), concat_path, concat_tree))
>>> bij.set_statistics((lambda d: d.semilength(), lambda t: t.node_number()))
>>> for D in sorted(bij.minimal_subdistributions_iterator(), key=lambda x:␣
↪(len(x[Integer(0)][Integer(0)]), x)):
...     ascii_art(D)
( [ /\ ], [ o ] )
```

(continues on next page)

(continued from previous page)

```
(           [ o   ] )
(           [  \  ] )
( [ /\/\ ], [   o ] )
(           [   o ] )
( [  /\  ]  [  /  ] )
( [ /  \ ], [ o   ] )
(             [ o     ] )
(             [  \    ] )
(             [   o   ] )
(             [    \  ] )
( [ /\/\/\ ], [     o ] )
(             [ o   ] )
(             [  \  ] )
(             [   o ] )
( [   /\  ]  [  /  ] )
( [ /\/  \ ], [ o   ] )
(             [   o   ] )
( [  /\    ]  [  / \  ] )
( [ /  \/\ ], [ o   o ] )
(                     [   o,     o ] )
(                     [  /      /  ] )
( [           /\   ]  [ o      o   ] )
( [  /\/\    /  \  ]  [  \    /    ] )
( [ /    \, /    \ ], [   o  o     ] )
```

The output is in a form suitable for FindStat:

```
sage: findmap(list(bij.minimal_subdistributions_iterator()))          # optional --
↪internet
0: Mp00034 (quality [100])
1: Mp00061oMp00023 (quality [100])
2: Mp00018oMp00140 (quality [100])
```

```
>>> from sage.all import *
>>> findmap(list(bij.minimal_subdistributions_iterator()))          # optional --
↪internet
0: Mp00034 (quality [100])
1: Mp00061oMp00023 (quality [100])
2: Mp00018oMp00140 (quality [100])
```

class sage.combinat.bijectionist.**Bijectionist**(*A*, *B*, *tau=None*, *alpha_beta=()*, *P=None*, *pi_rho=()*, *phi_psi=()*, *Q=None*, *elements_distributions=()*, *value_restrictions=()*, *solver=None*, *key=None*)

Bases: `SageObject`

A toolbox to list all possible bijections between two finite sets under various constraints.

INPUT:

- `A`, `B` – sets of equal size, given as a list
- `tau` – (optional) a function from `B` to `Z`, in case of `None`, the identity map `lambda x: x` is used
- `alpha_beta` – (optional) a list of pairs of statistics `alpha` from `A` to `W` and `beta` from `B` to `W`

- `P` – (optional) a partition of `A`
- `pi_rho` – (optional) a list of triples `(k, pi, rho)`, where
  - `pi` – a `k`-ary operation composing objects in `A` and
  - `rho` – a `k`-ary function composing statistic values in `Z`
- `elements_distributions` – (optional) a list of pairs `(tA, tZ)`, specifying the distributions of `tA`
- `value_restrictions` – (optional) a list of pairs `(a, tZ)`, restricting the possible values of `a`
- `solver` – (optional) the backend used to solve the mixed integer linear programs

`W` and `Z` can be arbitrary sets. As a natural example we may think of the natural numbers or tuples of integers.

We are looking for a statistic $s : A \to Z$ and a bijection $S : A \to B$ such that

- $s = \tau \circ S$: the statistics s and τ are equidistributed and S is an intertwining bijection.
- $\alpha = \beta \circ S$: the statistics α and β are equidistributed and S is an intertwining bijection.
- s is constant on the blocks of P.
- $s(\pi(a_1, \ldots, a_k)) = \rho(s(a_1), \ldots, s(a_k))$.

Additionally, we may require that

- $s(a) \in Z_a$ for specified sets $Z_a \subseteq Z$, and
- $s|_{\tilde A}$ has a specified distribution for specified sets $\tilde A \subset A$.

If τ is the identity, the two unknown functions s and S coincide. Although we do not exclude other bijective choices for τ, they probably do not make sense.

If we want that S is graded, i.e. if elements of A and B have a notion of size and S should preserve this size, we can add grading statistics as α and β. Since α and β will be equidistributed with S as an intertwining bijection, S will then also be graded.

In summary, we have the following two commutative diagrams, where s and S are unknown functions.

$$\begin{tikzcd} & A \arrow[dl, "\alpha"'] \arrow[d, "S"'] \arrow[dr, "s"] & \\ W & B \arrow[l, "\beta"'] \arrow[r, "\tau"] & Z \end{tikzcd} \qquad \begin{tikzcd} A^k \arrow[r, "\pi"] \arrow[d, "s^k"] & A \arrow[d, "s"] \\ Z^k \arrow[r, "\rho"] & Z \end{tikzcd}$$

> **Note**
>
> If τ is the identity map, the partition P of A necessarily consists only of singletons.

> **Note**
>
> The order of invocation of the methods with prefix `set`, i.e., *`set_statistics()`*, *`set_intertwining_relations()`*, *`set_constant_blocks()`*, etc., is irrelevant. Calling any of these methods a second time overrides the previous specification.

`constant_blocks`(*singletons=False*, *optimal=False*)

Return the set partition P of A such that $s : A \to Z$ is known to be constant on the blocks of P.

INPUT:

- `singletons` – boolean (default: `False`); whether or not to include singleton blocks in the output

- `optimal` – boolean (default: `False`); whether or not to compute the coarsest possible partition

Note: computing the coarsest possible partition may be computationally expensive, but may speed up generating solutions.

EXAMPLES:

```
sage: A = B = ["a", "b", "c"]
sage: bij = Bijectionist(A, B, lambda x: 0)
sage: bij.set_constant_blocks([["a", "b"]])
sage: bij.constant_blocks()
{{'a', 'b'}}

sage: bij.constant_blocks(singletons=True)
{{'a', 'b'}, {'c'}}
```

```
>>> from sage.all import *
>>> A = B = ["a", "b", "c"]
>>> bij = Bijectionist(A, B, lambda x: Integer(0))
>>> bij.set_constant_blocks([["a", "b"]])
>>> bij.constant_blocks()
{{'a', 'b'}}

>>> bij.constant_blocks(singletons=True)
{{'a', 'b'}, {'c'}}
```

minimal_subdistributions_blocks_iterator()

Return all representatives of minimal subsets $\tilde{P}$ of P together with submultisets $\tilde{Z}$ with $s(\tilde{P}) = \tilde{Z}$ as multisets.

Warning: If there are several solutions with the same support (i.e., the sets of block representatives are the same), only one of these will be found, even if the distributions are different, see the doctest below. To find all solutions, use *minimal_subdistributions_iterator()*, which is, however, computationally more expensive.

EXAMPLES:

```
sage: A = B = [permutation for n in range(3) for permutation in␣
↪Permutations(n)]
sage: bij = Bijectionist(A, B, len)
sage: bij.set_statistics((len, len))
sage: for sol in bij.solutions_iterator():
....:     print(sol)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 2}
sage: sorted(bij.minimal_subdistributions_blocks_iterator())
[([[]], [0]), ([[1]], [1]), ([[1, 2]], [2]), ([[2, 1]], [2])]
```

```
>>> from sage.all import *
>>> A = B = [permutation for n in range(Integer(3)) for permutation in␣
↪Permutations(n)]
>>> bij = Bijectionist(A, B, len)
>>> bij.set_statistics((len, len))
>>> for sol in bij.solutions_iterator():
...     print(sol)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 2}
>>> sorted(bij.minimal_subdistributions_blocks_iterator())
[([[]], [0]), ([[1]], [1]), ([[1, 2]], [2]), ([[2, 1]], [2])]
```

Another example:

```
sage: N = 2; A = B = [dyck_word for n in range(N+1) for dyck_word in␣
↪DyckWords(n)]
sage: def tau(D): return D.number_of_touch_points()
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_statistics((lambda d: d.semilength(), lambda d: d.semilength()))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1, 0]: 1, [1, 0, 1, 0]: 1, [1, 1, 0, 0]: 2}
{[]: 0, [1, 0]: 1, [1, 0, 1, 0]: 2, [1, 1, 0, 0]: 1}
sage: for subdistribution in bij.minimal_subdistributions_blocks_iterator():
....:     print(subdistribution)
([[]], [0])
([[1, 0]], [1])
([[1, 0, 1, 0], [1, 1, 0, 0]], [1, 2])
```

```
>>> from sage.all import *
>>> N = Integer(2); A = B = [dyck_word for n in range(N+Integer(1)) for dyck_
↪word in DyckWords(n)]
>>> def tau(D): return D.number_of_touch_points()
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_statistics((lambda d: d.semilength(), lambda d: d.semilength()))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1, 0]: 1, [1, 0, 1, 0]: 1, [1, 1, 0, 0]: 2}
{[]: 0, [1, 0]: 1, [1, 0, 1, 0]: 2, [1, 1, 0, 0]: 1}
>>> for subdistribution in bij.minimal_subdistributions_blocks_iterator():
...     print(subdistribution)
([[]], [0])
([[1, 0]], [1])
([[1, 0, 1, 0], [1, 1, 0, 0]], [1, 2])
```

An example with two elements of the same block in a subdistribution:

```
sage: A = B = ["a", "b", "c", "d", "e"]
sage: tau = {"a": 1, "b": 1, "c": 2, "d": 2, "e": 3}.get
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_constant_blocks([["a", "b"]])
sage: bij.set_value_restrictions(("a", [1, 2]))
```

(continues on next page)

(continued from previous page)

```
sage: bij.constant_blocks(optimal=True)
{{'a', 'b'}}
sage: list(bij.minimal_subdistributions_blocks_iterator())
[(['b', 'b', 'c', 'd', 'e'], [1, 1, 2, 2, 3])]
```

```
>>> from sage.all import *
>>> A = B = ["a", "b", "c", "d", "e"]
>>> tau = {"a": Integer(1), "b": Integer(1), "c": Integer(2), "d": Integer(2),
↪ "e": Integer(3)}.get
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_constant_blocks([["a", "b"]])
>>> bij.set_value_restrictions(("a", [Integer(1), Integer(2)]))
>>> bij.constant_blocks(optimal=True)
{{'a', 'b'}}
>>> list(bij.minimal_subdistributions_blocks_iterator())
[(['b', 'b', 'c', 'd', 'e'], [1, 1, 2, 2, 3])]
```

An example with overlapping minimal subdistributions:

```
sage: A = B = ["a", "b", "c", "d", "e"]
sage: tau = {"a": 1, "b": 1, "c": 2, "d": 2, "e": 3}.get
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_distributions((["a", "b"], [1, 2]), (["a", "c", "d"], [1, 2,␣
↪3]))
sage: sorted(bij.solutions_iterator(), key=lambda d: tuple(sorted(d.items())))
[{'a': 1, 'b': 2, 'c': 2, 'd': 3, 'e': 1},
 {'a': 1, 'b': 2, 'c': 3, 'd': 2, 'e': 1},
 {'a': 2, 'b': 1, 'c': 1, 'd': 3, 'e': 2},
 {'a': 2, 'b': 1, 'c': 3, 'd': 1, 'e': 2}]
sage: bij.constant_blocks(optimal=True)
{{'a', 'e'}}
sage: list(bij.minimal_subdistributions_blocks_iterator())
[(['a', 'b'], [1, 2]), (['a', 'c', 'd'], [1, 2, 3])]
```

```
>>> from sage.all import *
>>> A = B = ["a", "b", "c", "d", "e"]
>>> tau = {"a": Integer(1), "b": Integer(1), "c": Integer(2), "d": Integer(2),
↪ "e": Integer(3)}.get
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_distributions((["a", "b"], [Integer(1), Integer(2)]), (["a", "c",
↪"d"], [Integer(1), Integer(2), Integer(3)]))
>>> sorted(bij.solutions_iterator(), key=lambda d: tuple(sorted(d.items())))
[{'a': 1, 'b': 2, 'c': 2, 'd': 3, 'e': 1},
 {'a': 1, 'b': 2, 'c': 3, 'd': 2, 'e': 1},
 {'a': 2, 'b': 1, 'c': 1, 'd': 3, 'e': 2},
 {'a': 2, 'b': 1, 'c': 3, 'd': 1, 'e': 2}]
>>> bij.constant_blocks(optimal=True)
{{'a', 'e'}}
>>> list(bij.minimal_subdistributions_blocks_iterator())
[(['a', 'b'], [1, 2]), (['a', 'c', 'd'], [1, 2, 3])]
```

Fedor Petrov's example from https://mathoverflow.net/q/424187:

```
sage: A = B = ["a"+str(i) for i in range(1, 9)] + ["b"+str(i) for i in␣
↪range(3, 9)] + ["d"]
sage: tau = {b: 0 if i < 10 else 1 for i, b in enumerate(B)}.get
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_constant_blocks([["a"+str(i), "b"+str(i)] for i in range(1, 9)␣
↪if "b"+str(i) in A])
sage: d = [0]*8+[1]*4
sage: bij.set_distributions((A[:8] + A[8+2:-1], d), (A[:8] + A[8:-3], d))
sage: sorted([s[a] for a in A] for s in bij.solutions_iterator())
[[0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1],
 [0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0],
 [0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
 [0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0],
 [0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0],
 [1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0],
 [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
 [1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0],
 [1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0],
 [1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
 [1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1]]

sage: sorted(bij.minimal_subdistributions_blocks_iterator())        # random
[(['a1', 'a2', 'a3', 'a4', 'a5', 'a5', 'a6', 'a6', 'a7', 'a7', 'a8', 'a8'],
  [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]),
 (['a3', 'a4', 'd'], [0, 0, 1]),
 (['a7', 'a8', 'd'], [0, 0, 1])]
```

```
>>> from sage.all import *
>>> A = B = ["a"+str(i) for i in range(Integer(1), Integer(9))] + ["b"+str(i)␣
↪for i in range(Integer(3), Integer(9))] + ["d"]
>>> tau = {b: Integer(0) if i < Integer(10) else Integer(1) for i, b in␣
↪enumerate(B)}.get
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_constant_blocks([["a"+str(i), "b"+str(i)] for i in␣
↪range(Integer(1), Integer(9)) if "b"+str(i) in A])
>>> d = [Integer(0)]*Integer(8)+[Integer(1)]*Integer(4)
>>> bij.set_distributions((A[:Integer(8)] + A[Integer(8)+Integer(2):-
↪Integer(1)], d), (A[:Integer(8)] + A[Integer(8):-Integer(3)], d))
>>> sorted([s[a] for a in A] for s in bij.solutions_iterator())
[[0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1],
 [0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0],
 [0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
 [0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0],
 [0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0],
 [1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0],
 [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
 [1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0],
 [1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0],
 [1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
 [1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1]]

>>> sorted(bij.minimal_subdistributions_blocks_iterator())        # random
```

(continues on next page)

(continued from previous page)

```
[(['a1', 'a2', 'a3', 'a4', 'a5', 'a5', 'a6', 'a6', 'a7', 'a7', 'a8', 'a8'],
  [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]),
 (['a3', 'a4', 'd'], [0, 0, 1]),
 (['a7', 'a8', 'd'], [0, 0, 1])]
```

The following solution is not found, because it happens to have the same support as the other:

```
sage: D = set(A).difference(['b7', 'b8', 'd'])
sage: sorted(a.replace("b", "a") for a in D)
['a1', 'a2', 'a3', 'a3', 'a4', 'a4', 'a5', 'a5', 'a6', 'a6', 'a7', 'a8']
sage: set(tuple(sorted(s[a] for a in D)) for s in bij.solutions_iterator())
{(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1)}
```

```
>>> from sage.all import *
>>> D = set(A).difference(['b7', 'b8', 'd'])
>>> sorted(a.replace("b", "a") for a in D)
['a1', 'a2', 'a3', 'a3', 'a4', 'a4', 'a5', 'a5', 'a6', 'a6', 'a7', 'a8']
>>> set(tuple(sorted(s[a] for a in D)) for s in bij.solutions_iterator())
{(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1)}
```

But it is, by design, included here:

```
sage: sorted(D) in [d for d, _ in bij.minimal_subdistributions_iterator()]
True
```

```
>>> from sage.all import *
>>> sorted(D) in [d for d, _ in bij.minimal_subdistributions_iterator()]
True
```

minimal_subdistributions_iterator()

Return all minimal subsets $\tilde{A}$ of A together with submultisets $\tilde{Z}$ with $s(\tilde{A}) = \tilde{Z}$ as multisets.

EXAMPLES:

```
sage: A = B = [permutation for n in range(3) for permutation in␣
↪Permutations(n)]
sage: bij = Bijectionist(A, B, len)
sage: bij.set_statistics((len, len))
sage: for sol in bij.solutions_iterator():
....:     print(sol)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 2}
sage: sorted(bij.minimal_subdistributions_iterator())
[([[]], [0]), ([[1]], [1]), ([[1, 2]], [2]), ([[2, 1]], [2])]
```

```
>>> from sage.all import *
>>> A = B = [permutation for n in range(Integer(3)) for permutation in␣
↪Permutations(n)]
>>> bij = Bijectionist(A, B, len)
>>> bij.set_statistics((len, len))
>>> for sol in bij.solutions_iterator():
...     print(sol)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 2}
```

(continues on next page)

(continued from previous page)

```
>>> sorted(bij.minimal_subdistributions_iterator())
[(([]], [0]), ([[1]], [1]), ([[1, 2]], [2]), ([[2, 1]], [2])]
```

Another example:

```
sage: N = 2; A = B = [dyck_word for n in range(N+1) for dyck_word in␣
↪DyckWords(n)]
sage: def tau(D): return D.number_of_touch_points()
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_statistics((lambda d: d.semilength(), lambda d: d.semilength()))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1, 0]: 1, [1, 0, 1, 0]: 1, [1, 1, 0, 0]: 2}
{[]: 0, [1, 0]: 1, [1, 0, 1, 0]: 2, [1, 1, 0, 0]: 1}
sage: for subdistribution in bij.minimal_subdistributions_iterator():
....:     print(subdistribution)
([[]], [0])
([[1, 0]], [1])
([[1, 0, 1, 0], [1, 1, 0, 0]], [1, 2])
```

```
>>> from sage.all import *
>>> N = Integer(2); A = B = [dyck_word for n in range(N+Integer(1)) for dyck_
↪word in DyckWords(n)]
>>> def tau(D): return D.number_of_touch_points()
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_statistics((lambda d: d.semilength(), lambda d: d.semilength()))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1, 0]: 1, [1, 0, 1, 0]: 1, [1, 1, 0, 0]: 2}
{[]: 0, [1, 0]: 1, [1, 0, 1, 0]: 2, [1, 1, 0, 0]: 1}
>>> for subdistribution in bij.minimal_subdistributions_iterator():
...     print(subdistribution)
([[]], [0])
([[1, 0]], [1])
([[1, 0, 1, 0], [1, 1, 0, 0]], [1, 2])
```

An example with two elements of the same block in a subdistribution:

```
sage: A = B = ["a", "b", "c", "d", "e"]
sage: tau = {"a": 1, "b": 1, "c": 2, "d": 2, "e": 3}.get
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_constant_blocks([["a", "b"]])
sage: bij.set_value_restrictions(("a", [1, 2]))
sage: bij.constant_blocks(optimal=True)
{{'a', 'b'}}
sage: list(bij.minimal_subdistributions_iterator())
[(['a', 'b', 'c', 'd', 'e'], [1, 1, 2, 2, 3])]
```

```
>>> from sage.all import *
>>> A = B = ["a", "b", "c", "d", "e"]
```

(continues on next page)

(continued from previous page)

```
>>> tau = {"a": Integer(1), "b": Integer(1), "c": Integer(2), "d": Integer(2),
↪ "e": Integer(3)}.get
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_constant_blocks([["a", "b"]])
>>> bij.set_value_restrictions(("a", [Integer(1), Integer(2)]))
>>> bij.constant_blocks(optimal=True)
{{'a', 'b'}}
>>> list(bij.minimal_subdistributions_iterator())
[(['a', 'b', 'c', 'd', 'e'], [1, 1, 2, 2, 3])]
```

possible_values(*p=None*, *optimal=False*)

Return for each block the values of s compatible with the imposed restrictions.

INPUT:

- `p` – (optional) a block of P, or an element of a block of P, or a list of these
- `optimal` – boolean (default: `False`); whether or not to compute the minimal possible set of statistic values

Note

Computing the minimal possible set of statistic values may be computationally expensive.

Todo

currently, calling this method with `optimal=True` does not update the internal dictionary, because this would interfere with the variables of the MILP.

EXAMPLES:

```
sage: A = B = ["a", "b", "c", "d"]
sage: tau = {"a": 1, "b": 1, "c": 1, "d": 2}.get
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_constant_blocks([["a", "b"]])
sage: bij.possible_values(A)
{'a': {1, 2}, 'b': {1, 2}, 'c': {1, 2}, 'd': {1, 2}}
sage: bij.possible_values(A, optimal=True)
{'a': {1}, 'b': {1}, 'c': {1, 2}, 'd': {1, 2}}
```

```
>>> from sage.all import *
>>> A = B = ["a", "b", "c", "d"]
>>> tau = {"a": Integer(1), "b": Integer(1), "c": Integer(1), "d": Integer(2)}
↪.get
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_constant_blocks([["a", "b"]])
>>> bij.possible_values(A)
{'a': {1, 2}, 'b': {1, 2}, 'c': {1, 2}, 'd': {1, 2}}
>>> bij.possible_values(A, optimal=True)
{'a': {1}, 'b': {1}, 'c': {1, 2}, 'd': {1, 2}}
```

The internal dictionary is not updated:

```
sage: bij.possible_values(A)
{'a': {1, 2}, 'b': {1, 2}, 'c': {1, 2}, 'd': {1, 2}}
```

```
>>> from sage.all import *
>>> bij.possible_values(A)
{'a': {1, 2}, 'b': {1, 2}, 'c': {1, 2}, 'd': {1, 2}}
```

set_constant_blocks(P)

Declare that $s : A \to Z$ is constant on each block of P.

> **Warning**
>
> Any restriction imposed by a previous invocation of *set_constant_blocks()* will be overwritten, including restrictions discovered by *set_intertwining_relations()* and *solutions_iterator()*!

A common example is to use the orbits of a bijection acting on A. This can be achieved using the function *orbit_decomposition()*.

INPUT:

- `P` – set partition of A, singletons may be omitted

EXAMPLES:

Initially the partitions are set to singleton blocks. The current partition can be reviewed using *constant_blocks()*:

```
sage: A = B = 'abcd'
sage: bij = Bijectionist(A, B, lambda x: B.index(x) % 2)
sage: bij.constant_blocks()
{}

sage: bij.set_constant_blocks([['a', 'c']])
sage: bij.constant_blocks()
{{'a', 'c'}}
```

```
>>> from sage.all import *
>>> A = B = 'abcd'
>>> bij = Bijectionist(A, B, lambda x: B.index(x) % Integer(2))
>>> bij.constant_blocks()
{}

>>> bij.set_constant_blocks([['a', 'c']])
>>> bij.constant_blocks()
{{'a', 'c'}}
```

We now add a map that combines some blocks:

```
sage: def pi(p1, p2): return 'abcdefgh'[A.index(p1) + A.index(p2)]
sage: def rho(s1, s2): return (s1 + s2) % 2
sage: bij.set_intertwining_relations((2, pi, rho))
sage: list(bij.solutions_iterator())
```

(continues on next page)

(continued from previous page)

```
[{'a': 0, 'b': 1, 'c': 0, 'd': 1}]
sage: bij.constant_blocks()
{{'a', 'c'}, {'b', 'd'}}
```

```
>>> from sage.all import *
>>> def pi(p1, p2): return 'abcdefgh'[A.index(p1) + A.index(p2)]
>>> def rho(s1, s2): return (s1 + s2) % Integer(2)
>>> bij.set_intertwining_relations((Integer(2), pi, rho))
>>> list(bij.solutions_iterator())
[{'a': 0, 'b': 1, 'c': 0, 'd': 1}]
>>> bij.constant_blocks()
{{'a', 'c'}, {'b', 'd'}}
```

Setting constant blocks overrides any previous assignment:

```
sage: bij.set_constant_blocks([['a', 'b']])
sage: bij.constant_blocks()
{{'a', 'b'}}
```

```
>>> from sage.all import *
>>> bij.set_constant_blocks([['a', 'b']])
>>> bij.constant_blocks()
{{'a', 'b'}}
```

If there is no solution, and the coarsest partition is requested, an error is raised:

```
sage: bij.constant_blocks(optimal=True)
Traceback (most recent call last):
...
StopIteration
```

```
>>> from sage.all import *
>>> bij.constant_blocks(optimal=True)
Traceback (most recent call last):
...
StopIteration
```

set_distributions(**elements_distributions*)

Specify the distribution of s for a subset of elements.

> **Warning**
>
> Any restriction imposed by a previous invocation of `set_distributions()` will be overwritten!

INPUT:

- one or more pairs of $(\tilde{A}, \tilde{Z})$, where $\tilde{A} \subseteq A$ and $\tilde{Z}$ is a list of values in Z of the same size as $\tilde{A}$

This method specifies that $\{s(a) | a \in \tilde{A}\}$ equals $\tilde{Z}$ as a multiset for each of the pairs.

When specifying several distributions, the subsets of A do not have to be disjoint.

ALGORITHM:

We add

$$\sum_{a \in \tilde{A}} x_{p(a),z} t^z = \sum_{z \in \tilde{Z}} t^z,$$

where $p(a)$ is the block containing a, for each given distribution as a vector equation to the MILP.

EXAMPLES:

```
sage: A = B = [permutation for n in range(4) for permutation in␣
↪Permutations(n)]
sage: tau = Permutation.longest_increasing_subsequence_length
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_statistics((len, len))
sage: bij.set_distributions(([Permutation([1, 2, 3]), Permutation([1, 3, 2])],
↪ [1, 3]))
sage: for sol in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(sol)
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 3, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 1, [1, 2, 3]: 1, [1, 3, 2]: 3, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 1, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}

sage: bij.constant_blocks(optimal=True)
{{[2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]}}
sage: sorted(bij.minimal_subdistributions_blocks_iterator(), key=lambda d:␣
↪(len(d[0]), d[0]))
[([[]], [0]),
 ([[1]], [1]),
 ([[2, 1, 3]], [2]),
 ([[1, 2], [2, 1]], [1, 2]),
 ([[1, 2, 3], [1, 3, 2]], [1, 3])]
```

```
>>> from sage.all import *
>>> A = B = [permutation for n in range(Integer(4)) for permutation in␣
↪Permutations(n)]
>>> tau = Permutation.longest_increasing_subsequence_length
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_statistics((len, len))
>>> bij.set_distributions(([Permutation([Integer(1), Integer(2), Integer(3)]),
↪ Permutation([Integer(1), Integer(3), Integer(2)])], [Integer(1),␣
↪Integer(3)]))
>>> for sol in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(sol)
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 3, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
```

(continues on next page)

(continued from previous page)

```
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 1, [1, 2, 3]: 1, [1, 3, 2]: 3, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 1, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}

>>> bij.constant_blocks(optimal=True)
{{[2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]}}
>>> sorted(bij.minimal_subdistributions_blocks_iterator(), key=lambda d:␣
↪(len(d[Integer(0)]), d[Integer(0)]))
[([[]], [0]),
 ([[1]], [1]),
 ([[2, 1, 3]], [2]),
 ([[1, 2], [2, 1]], [1, 2]),
 ([[1, 2, 3], [1, 3, 2]], [1, 3])]
```

We may also specify multiple, possibly overlapping distributions:

```
sage: bij.set_distributions(([Permutation([1, 2, 3]), Permutation([1, 3, 2])],
↪ [1, 3]),
....:                          ([Permutation([1, 3, 2]), Permutation([3, 2, 1]),
....:                           Permutation([2, 1, 3])], [1, 2, 2]))
sage: for sol in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(sol)
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 1, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}

sage: bij.constant_blocks(optimal=True)
{{[1], [1, 3, 2]}, {[2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]}}
sage: sorted(bij.minimal_subdistributions_blocks_iterator(), key=lambda d:␣
↪(len(d[0]), d[0]))
[([[]], [0]),
 ([[1]], [1]),
 ([[1, 2, 3]], [3]),
 ([[2, 3, 1]], [2]),
 ([[1, 2], [2, 1]], [1, 2])]
```

```
>>> from sage.all import *
>>> bij.set_distributions(([Permutation([Integer(1), Integer(2), Integer(3)]),
↪ Permutation([Integer(1), Integer(3), Integer(2)])], [Integer(1),␣
↪Integer(3)]),
...                        ([Permutation([Integer(1), Integer(3), Integer(2)]),
↪ Permutation([Integer(3), Integer(2), Integer(1)]),
...                         Permutation([Integer(2), Integer(1), Integer(3)])],
↪ [Integer(1), Integer(2), Integer(2)]))
>>> for sol in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(sol)
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
```

(continues on next page)

(continued from previous page)

```
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 1, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}

>>> bij.constant_blocks(optimal=True)
{{[1], [1, 3, 2]}, {[2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]}}
>>> sorted(bij.minimal_subdistributions_blocks_iterator(), key=lambda d:␣
↪(len(d[Integer(0)]), d[Integer(0)]))
[([[]], [0]),
 ([[1]], [1]),
 ([[1, 2, 3]], [3]),
 ([[2, 3, 1]], [2]),
 ([[1, 2], [2, 1]], [1, 2])]
```

set_homomesic(*Q*)

Assert that the average of s on each block of Q is constant.

INPUT:

- `Q` – set partition of `A`

EXAMPLES:

```
sage: A = B = [1,2,3]
sage: bij = Bijectionist(A, B, lambda b: b % 3)
sage: bij.set_homomesic([[1,2], [3]])
sage: list(bij.solutions_iterator())
[{1: 2, 2: 0, 3: 1}, {1: 0, 2: 2, 3: 1}]
```

```
>>> from sage.all import *
>>> A = B = [Integer(1),Integer(2),Integer(3)]
>>> bij = Bijectionist(A, B, lambda b: b % Integer(3))
>>> bij.set_homomesic([[Integer(1),Integer(2)], [Integer(3)]])
>>> list(bij.solutions_iterator())
[{1: 2, 2: 0, 3: 1}, {1: 0, 2: 2, 3: 1}]
```

set_intertwining_relations(**pi_rho*)

Add restrictions of the form $s(\pi(a_1, \dots, a_k)) = \rho(s(a_1), \dots, s(a_k))$.

> **Warning**
>
> Any restriction imposed by a previous invocation of `set_intertwining_relations()` will be overwritten!

INPUT:

- `pi_rho` – one or more tuples $(k, \pi : A^k \to A, \rho : Z^k \to Z, \tilde{A})$ where $\tilde{A}$ (optional) is a k-ary function that returns true if and only if a k-tuple of objects in A is in the domain of π

ALGORITHM:

The relation

$$s(\pi(a_1, \dots, a_k)) = \rho(s(a_1), \dots, s(a_k))$$

for each pair (π, ρ) implies immediately that $s(\pi(a_1, \dots, a_k))$ only depends on the blocks of $a_1, \dots, a_k$.

The MILP formulation is as follows. Let $a_1, \dots, a_k \in A$ and let $a = \pi(a_1, \dots, a_k)$. Let $z_1, \dots, z_k \in Z$ and let $z = \rho(z_1, \dots, z_k)$. Suppose that $a_i \in p_i$ for all i and that $a \in p$.

We then want to model the implication

$$x_{p_1,z_1} = 1, \dots, x_{p_k,z_k} = 1 \Rightarrow x_{p,z} = 1.$$

We achieve this by requiring

$$x_{p,z} \geq 1 - k + \sum_{i=1}^{k} x_{p_i,z_i}.$$

Note that z must be a possible value of p and each z_i must be a possible value of p_i.

EXAMPLES:

We can concatenate two permutations by increasing the values of the second permutation by the length of the first permutation:

```
sage: def concat(p1, p2): return Permutation(p1 + [i + len(p1) for i in p2])
```

```
>>> from sage.all import *
>>> def concat(p1, p2): return Permutation(p1 + [i + len(p1) for i in p2])
```

We may be interested in statistics on permutations which are equidistributed with the number of fixed points, such that concatenating permutations corresponds to adding statistic values:

```
sage: A = B = [permutation for n in range(4) for permutation in␣
↪Permutations(n)]
sage: bij = Bijectionist(A, B, Permutation.number_of_fixed_points)
sage: bij.set_statistics((len, len))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
...
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
...
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪3, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
...

sage: bij.set_intertwining_relations((2, concat, lambda x, y: x + y))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 0}
```

```
>>> from sage.all import *
>>> A = B = [permutation for n in range(Integer(4)) for permutation in␣
```

(continues on next page)

(continued from previous page)

```
↪Permutations(n)]
>>> bij = Bijectionist(A, B, Permutation.number_of_fixed_points)
>>> bij.set_statistics((len, len))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
...
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
...
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪3, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
...

>>> bij.set_intertwining_relations((Integer(2), concat, lambda x, y: x + y))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 0}
```

The domain of the composition may be restricted. E.g., if we concatenate only permutations starting with a 1, we obtain fewer forced elements:

```
sage: in_domain = lambda p1, p2: (not p1 or p1(1) == 1) and (not p2 or p2(1)␣
↪== 1)
sage: bij.set_intertwining_relations((2, concat, lambda x, y: x + y, in_
↪domain))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
```

(continues on next page)

(continued from previous page)

```
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 0}
```

```
>>> from sage.all import *
>>> in_domain = lambda p1, p2: (not p1 or p1(Integer(1)) == Integer(1)) and
↪(not p2 or p2(Integer(1)) == Integer(1))
>>> bij.set_intertwining_relations((Integer(2), concat, lambda x, y: x + y,
↪in_domain))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:
↪1, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
↪0, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
↪0, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
↪1, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 0}
```

We can also restrict according to several composition functions. For example, we may additionally concatenate permutations by incrementing the elements of the first:

```
sage: skew_concat = lambda p1, p2: Permutation([i + len(p2) for i in p1] +
↪list(p2))
sage: bij.set_intertwining_relations((2, skew_concat, lambda x, y: x + y))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 0, [1, 3, 2]: 0, [2, 1, 3]:
↪1, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 0, [1, 3, 2]: 1, [2, 1, 3]:
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
```

```
>>> from sage.all import *
>>> skew_concat = lambda p1, p2: Permutation([i + len(p2) for i in p1] +
↪list(p2))
```

(continues on next page)

(continued from previous page)

```
>>> bij.set_intertwining_relations((Integer(2), skew_concat, lambda x, y: x +␣
↪y))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 0, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 0, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
```

However, this yields no solution:

```
sage: bij.set_intertwining_relations((2, concat, lambda x, y: x + y), (2,␣
↪skew_concat, lambda x, y: x + y))
sage: list(bij.solutions_iterator())
[]
```

```
>>> from sage.all import *
>>> bij.set_intertwining_relations((Integer(2), concat, lambda x, y: x + y),␣
↪(Integer(2), skew_concat, lambda x, y: x + y))
>>> list(bij.solutions_iterator())
[]
```

set_quadratic_relation(**phi_psi*)

Add restrictions of the form $s \circ \psi \circ s = \phi$.

INPUT:

- phi_psi – (optional) a list of pairs (ϕ, ρ) where $\phi : A \to Z$ and $\psi : Z \to A$

ALGORITHM:

We add

$$x_{p(a),z} = x_{p(\psi(z)),\phi(a)}$$

for $a \in A$ and $z \in Z$ to the MILP, where $\phi : A \to Z$ and $\psi : Z \to A$. Note that, in particular, ϕ must be constant on blocks.

EXAMPLES:

```
sage: A = B = DyckWords(3)
sage: bij = Bijectionist(A, B)
sage: bij.set_statistics((lambda D: D.number_of_touch_points(), lambda D: D.
↪number_of_initial_rises()))
sage: ascii_art(sorted(bij.minimal_subdistributions_iterator()))
[ (             [   /\   ] )
[ (             [  /  \  ] )  ( [    /\    /\    ]  [  /\      /\/\  ] )
[ ( [ /\/\/\ ], [ /    \ ] ), ( [ /\/  \, /  \/\ ], [ /  \/\, /    \ ] ),

 ( [           /\   ]                     ) ]
 ( [  /\/\    /  \   ]  [            /\  ] ) ]
 ( [ /    \, /    \ ], [ /\/\/\, /\/  \ ] ) ]
```

(continues on next page)

(continued from previous page)

```
sage: bij.set_quadratic_relation((lambda D: D, lambda D: D))
sage: ascii_art(sorted(bij.minimal_subdistributions_iterator()))
[ (             [   /\   ] )
[ (             [  /  \  ] )  ( [    /\  ]  [  /\/\  ] )
[ ( [ /\/\/\ ], [ /    \ ] ), ( [ /\/  \ ], [ /    \ ] ),


 ( [  /\    ]  [  /\    ] )  ( [  /\/\  ]  [    /\  ] )
 ( [ /  \/\ ], [ /  \/\ ] ), ( [ /    \ ], [ /\/  \ ] ),

 ( [   /\   ]              ) ]
 ( [  /  \  ]              ) ]
 ( [ /    \ ], [ /\/\/\ ] ) ]
```

```
>>> from sage.all import *
>>> A = B = DyckWords(Integer(3))
>>> bij = Bijectionist(A, B)
>>> bij.set_statistics((lambda D: D.number_of_touch_points(), lambda D: D.
→number_of_initial_rises()))
>>> ascii_art(sorted(bij.minimal_subdistributions_iterator()))
[ (             [   /\   ] )
[ (             [  /  \  ] )  ( [    /\    /\    ]  [  /\      /\/\  ] )
[ ( [ /\/\/\ ], [ /    \ ] ), ( [ /\/  \, /  \/\ ], [ /  \/\, /    \ ] ),
<BLANKLINE>
 ( [           /\   ]                     ) ]
 ( [  /\/\    /  \  ]  [            /\  ] ) ]
 ( [ /    \, /    \ ], [ /\/\/\, /\/  \ ] ) ]
>>> bij.set_quadratic_relation((lambda D: D, lambda D: D))
>>> ascii_art(sorted(bij.minimal_subdistributions_iterator()))
[ (             [   /\   ] )
[ (             [  /  \  ] )  ( [    /\  ]  [  /\/\  ] )
[ ( [ /\/\/\ ], [ /    \ ] ), ( [ /\/  \ ], [ /    \ ] ),
<BLANKLINE>
<BLANKLINE>
 ( [  /\    ]  [  /\    ] )  ( [  /\/\  ]  [    /\  ] )
 ( [ /  \/\ ], [ /  \/\ ] ), ( [ /    \ ], [ /\/  \ ] ),
<BLANKLINE>
 ( [   /\   ]              ) ]
 ( [  /  \  ]              ) ]
 ( [ /    \ ], [ /\/\/\ ] ) ]
```

set_semi_conjugacy(**pi_rho*)

Add restrictions of the form $s(\pi(a_1, \ldots, a_k)) = \rho(s(a_1), \ldots, s(a_k))$.

> **Warning**
>
> Any restriction imposed by a previous invocation of *set_intertwining_relations()* will be overwritten!

INPUT:

- `pi_rho` – one or more tuples $(k, \pi : A^k \to A, \rho : Z^k \to Z, \tilde{A})$ where $\tilde{A}$ (optional) is a k-ary function that returns true if and only if a k-tuple of objects in A is in the domain of π

ALGORITHM:

The relation

$$s(\pi(a_1, \dots, a_k)) = \rho(s(a_1), \dots, s(a_k))$$

for each pair (π, ρ) implies immediately that $s(\pi(a_1, \dots, a_k))$ only depends on the blocks of $a_1, \dots, a_k$.

The MILP formulation is as follows. Let $a_1, \dots, a_k \in A$ and let $a = \pi(a_1, \dots, a_k)$. Let $z_1, \dots, z_k \in Z$ and let $z = \rho(z_1, \dots, z_k)$. Suppose that $a_i \in p_i$ for all i and that $a \in p$.

We then want to model the implication

$$x_{p_1, z_1} = 1, \dots, x_{p_k, z_k} = 1 \Rightarrow x_{p, z} = 1.$$

We achieve this by requiring

$$x_{p,z} \geq 1 - k + \sum_{i=1}^{k} x_{p_i, z_i}.$$

Note that z must be a possible value of p and each z_i must be a possible value of p_i.

EXAMPLES:

We can concatenate two permutations by increasing the values of the second permutation by the length of the first permutation:

```
sage: def concat(p1, p2): return Permutation(p1 + [i + len(p1) for i in p2])
```

```
>>> from sage.all import *
>>> def concat(p1, p2): return Permutation(p1 + [i + len(p1) for i in p2])
```

We may be interested in statistics on permutations which are equidistributed with the number of fixed points, such that concatenating permutations corresponds to adding statistic values:

```
sage: A = B = [permutation for n in range(4) for permutation in␣
↪Permutations(n)]
sage: bij = Bijectionist(A, B, Permutation.number_of_fixed_points)
sage: bij.set_statistics((len, len))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
...
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
...
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪3, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
...

sage: bij.set_intertwining_relations((2, concat, lambda x, y: x + y))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
```

(continues on next page)

(continued from previous page)

```
↪1, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 0}
```

```
>>> from sage.all import *
>>> A = B = [permutation for n in range(Integer(4)) for permutation in␣
↪Permutations(n)]
>>> bij = Bijectionist(A, B, Permutation.number_of_fixed_points)
>>> bij.set_statistics((len, len))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
...
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
...
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪3, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
...

>>> bij.set_intertwining_relations((Integer(2), concat, lambda x, y: x + y))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 0}
```

The domain of the composition may be restricted. E.g., if we concatenate only permutations starting with a 1, we obtain fewer forced elements:

```
sage: in_domain = lambda p1, p2: (not p1 or p1(1) == 1) and (not p2 or p2(1)␣
↪== 1)
sage: bij.set_intertwining_relations((2, concat, lambda x, y: x + y, in_
↪domain))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
```

(continues on next page)

(continued from previous page)

```
↪0, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 0}
```

```
>>> from sage.all import *
>>> in_domain = lambda p1, p2: (not p1 or p1(Integer(1)) == Integer(1)) and␣
↪(not p2 or p2(Integer(1)) == Integer(1))
>>> bij.set_intertwining_relations((Integer(2), concat, lambda x, y: x + y,␣
↪in_domain))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 0, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 1, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 2, [2, 1]: 0, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 0, [3, 2, 1]: 0}
```

We can also restrict according to several composition functions. For example, we may additionally concatenate permutations by incrementing the elements of the first:

```
sage: skew_concat = lambda p1, p2: Permutation([i + len(p2) for i in p1] +␣
↪list(p2))
sage: bij.set_intertwining_relations((2, skew_concat, lambda x, y: x + y))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 0, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
```

(continues on next page)

(continued from previous page)

```
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 0, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
```

```
>>> from sage.all import *
>>> skew_concat = lambda p1, p2: Permutation([i + len(p2) for i in p1] +␣
↪list(p2))
>>> bij.set_intertwining_relations((Integer(2), skew_concat, lambda x, y: x +␣
↪y))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 0, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 0, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 1, [3, 2, 1]: 3}
```

However, this yields no solution:

```
sage: bij.set_intertwining_relations((2, concat, lambda x, y: x + y), (2,␣
↪skew_concat, lambda x, y: x + y))
sage: list(bij.solutions_iterator())
[]
```

```
>>> from sage.all import *
>>> bij.set_intertwining_relations((Integer(2), concat, lambda x, y: x + y),␣
↪(Integer(2), skew_concat, lambda x, y: x + y))
>>> list(bij.solutions_iterator())
[]
```

set_statistics(**alpha_beta*)

Set constraints of the form $\alpha = \beta \circ S$.

> **Warning**
>
> Any restriction imposed by a previous invocation of *set_statistics()* will be overwritten!

INPUT:

- alpha_beta – one or more pairs $(\alpha : A \to W, \beta : B \to W)$

If the statistics α and β are not equidistributed, an error is raised.

ALGORITHM:

We add

$$\sum_{a\in A, z\in Z} x_{p(a),z} s^z t^{\alpha(a)} = \sum_{b\in B} s^{\tau(b)} t(\beta(b))$$

as a matrix equation to the MILP.

EXAMPLES:

We look for bijections S on permutations such that the number of weak exceedences of $S(\pi)$ equals the number of descents of π, and statistics s, such that the number of fixed points of $S(\pi)$ equals $s(\pi)$:

```
sage: N = 4; A = B = [permutation for n in range(N) for permutation in␣
↪Permutations(n)]
sage: def wex(p): return len(p.weak_excedences())
sage: def fix(p): return len(p.fixed_points())
sage: def des(p): return len(p.descents(final_descent=True)) if p else 0
sage: def adj(p): return len([e for (e, f) in zip(p, p[1:]+[0]) if e == f+1])
sage: bij = Bijectionist(A, B, fix)
sage: bij.set_statistics((wex, des), (len, len))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 3, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 3, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 0}

sage: bij = Bijectionist(A, B, fix)
sage: bij.set_statistics((wex, des), (fix, adj), (len, len))
sage: for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
....:     print(solution)
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 0}
```

```
>>> from sage.all import *
>>> N = Integer(4); A = B = [permutation for n in range(N) for permutation in␣
↪Permutations(n)]
>>> def wex(p): return len(p.weak_excedences())
>>> def fix(p): return len(p.fixed_points())
>>> def des(p): return len(p.descents(final_descent=True)) if p else␣
↪Integer(0)
>>> def adj(p): return len([e for (e, f) in zip(p,␣
↪p[Integer(1):]+[Integer(0)]) if e == f+Integer(1)])
>>> bij = Bijectionist(A, B, fix)
>>> bij.set_statistics((wex, des), (len, len))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
```

(continues on next page)

(continued from previous page)

```
...     print(solution)
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 1, [3, 1, 2]: 3, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 1, [3, 1, 2]: 3, [3, 2, 1]: 0}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 0}

>>> bij = Bijectionist(A, B, fix)
>>> bij.set_statistics((wex, des), (fix, adj), (len, len))
>>> for solution in sorted(list(bij.solutions_iterator()), key=lambda d:␣
↪tuple(sorted(d.items()))):
...     print(solution)
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 0, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪0, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 1}
{[]: 0, [1]: 1, [1, 2]: 0, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [2, 3, 1]: 0, [3, 1, 2]: 3, [3, 2, 1]: 0}
```

Calling this with non-equidistributed statistics yields an error:

```
sage: bij = Bijectionist(A, B, fix)
sage: bij.set_statistics((wex, fix))
Traceback (most recent call last):
...
ValueError: statistics alpha and beta are not equidistributed
```

```
>>> from sage.all import *
>>> bij = Bijectionist(A, B, fix)
>>> bij.set_statistics((wex, fix))
Traceback (most recent call last):
...
ValueError: statistics alpha and beta are not equidistributed
```

set_value_restrictions(**value_restrictions*)

Restrict the set of possible values $s(a)$ for a given element a.

> **Warning**
>
> Any restriction imposed by a previous invocation of `set_value_restrictions()` will be overwritten!

INPUT:

- `value_restrictions` – one or more pairs $(a \in A, \tilde{Z} \subseteq Z)$

EXAMPLES:

We may want to restrict the value of a given element to a single or multiple values. We do not require that the specified values are in the image of τ. In some cases, the restriction may not be able to provide a better solution, as for size 3 in the following example.

```
sage: A = B = [permutation for n in range(4) for permutation in␣
↪Permutations(n)]
sage: tau = Permutation.longest_increasing_subsequence_length
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_statistics((len, len))
sage: bij.set_value_restrictions((Permutation([1, 2]), [1]),
....:                           (Permutation([3, 2, 1]), [2, 3, 4]))
sage: for sol in sorted(bij.solutions_iterator(), key=lambda d: sorted(d.
↪items())):
....:     print(sol)
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 2, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 2, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 3, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 2, [2, 1, 3]:␣
↪2, [2, 3, 1]: 3, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 2, [2, 1, 3]:␣
↪3, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 3, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 3, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 1, [2, 1, 3]:␣
↪2, [2, 3, 1]: 3, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 1, [2, 1, 3]:␣
↪3, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪1, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪1, [2, 3, 1]: 2, [3, 1, 2]: 3, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪1, [2, 3, 1]: 3, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪2, [2, 3, 1]: 1, [3, 1, 2]: 2, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪2, [2, 3, 1]: 1, [3, 1, 2]: 3, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪2, [2, 3, 1]: 2, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪2, [2, 3, 1]: 3, [3, 1, 2]: 1, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪3, [2, 3, 1]: 1, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:␣
↪3, [2, 3, 1]: 2, [3, 1, 2]: 1, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 3, [2, 1, 3]:␣
↪1, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
```

(continues on next page)

(continued from previous page)

```
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 3, [2, 1, 3]:
→2, [2, 3, 1]: 1, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 3, [2, 1, 3]:
→2, [2, 3, 1]: 2, [3, 1, 2]: 1, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
→2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 2, [2, 1, 3]:
→1, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 2, [2, 1, 3]:
→2, [2, 3, 1]: 1, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 2, [2, 1, 3]:
→2, [2, 3, 1]: 2, [3, 1, 2]: 1, [3, 2, 1]: 2}
```

```
>>> from sage.all import *
>>> A = B = [permutation for n in range(Integer(4)) for permutation in
→Permutations(n)]
>>> tau = Permutation.longest_increasing_subsequence_length
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_statistics((len, len))
>>> bij.set_value_restrictions((Permutation([Integer(1), Integer(2)]),
→[Integer(1)]),
...                            (Permutation([Integer(3), Integer(2),
→Integer(1)]), [Integer(2), Integer(3), Integer(4)]))
>>> for sol in sorted(bij.solutions_iterator(), key=lambda d: sorted(d.
→items())):
...     print(sol)
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 2, [2, 1, 3]:
→2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 2, [2, 1, 3]:
→2, [2, 3, 1]: 2, [3, 1, 2]: 3, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 2, [2, 1, 3]:
→2, [2, 3, 1]: 3, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 2, [2, 1, 3]:
→3, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 1, [1, 3, 2]: 3, [2, 1, 3]:
→2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 1, [2, 1, 3]:
→2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 1, [2, 1, 3]:
→2, [2, 3, 1]: 2, [3, 1, 2]: 3, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 1, [2, 1, 3]:
→2, [2, 3, 1]: 3, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 1, [2, 1, 3]:
→3, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
→1, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
→1, [2, 3, 1]: 2, [3, 1, 2]: 3, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
→1, [2, 3, 1]: 3, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
→2, [2, 3, 1]: 1, [3, 1, 2]: 2, [3, 2, 1]: 3}
```

(continues on next page)

(continued from previous page)

```
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
↪2, [2, 3, 1]: 1, [3, 1, 2]: 3, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
↪2, [2, 3, 1]: 2, [3, 1, 2]: 1, [3, 2, 1]: 3}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
↪2, [2, 3, 1]: 3, [3, 1, 2]: 1, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
↪3, [2, 3, 1]: 1, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 2, [2, 1, 3]:
↪3, [2, 3, 1]: 2, [3, 1, 2]: 1, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 3, [2, 1, 3]:
↪1, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 3, [2, 1, 3]:
↪2, [2, 3, 1]: 1, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 2, [1, 3, 2]: 3, [2, 1, 3]:
↪2, [2, 3, 1]: 2, [3, 1, 2]: 1, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 1, [2, 1, 3]:
↪2, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 2, [2, 1, 3]:
↪1, [2, 3, 1]: 2, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 2, [2, 1, 3]:
↪2, [2, 3, 1]: 1, [3, 1, 2]: 2, [3, 2, 1]: 2}
{[]: 0, [1]: 1, [1, 2]: 1, [2, 1]: 2, [1, 2, 3]: 3, [1, 3, 2]: 2, [2, 1, 3]:
↪2, [2, 3, 1]: 2, [3, 1, 2]: 1, [3, 2, 1]: 2}
```

However, an error occurs if the set of possible values is empty. In this example, the image of τ under any legal bijection is disjoint to the specified values.

solutions_iterator()

An iterator over all solutions of the problem.

OUTPUT: an iterator over all possible mappings $s : A \to Z$

ALGORITHM:

We solve an integer linear program with a binary variable $x_{p,z}$ for each partition block $p \in P$ and each statistic value $z \in Z$:

- $x_{p,z} = 1$ if and only if $s(a) = z$ for all $a \in p$.

Then we add the constraint $\sum_{x\in V} x < |V|$, where V is the set containing all x with $x = 1$, that is, those indicator variables representing the current solution. Therefore, a solution of this new program must be different from all those previously obtained.

INTEGER LINEAR PROGRAM:

- Let $m_w(p)$, for a block p of P, be the multiplicity of the value w in W under α, that is, the number of elements $a \in p$ with $\alpha(a) = w$.
- Let $n_w(z)$ be the number of elements $b \in B$ with $\beta(b) = w$ and $\tau(b) = z$ for $w \in W$, $z \in Z$.
- Let k be the arity of a pair (π, ρ) in an intertwining relation.

and the following constraints:

- because every block is assigned precisely one value, for all $p \in P$,

$$\sum_z x_{p,z} = 1.$$

- because the statistics s and τ and also α and β are equidistributed, for all $w \in W$ and $z \in Z$,

$$\sum_p m_w(p) x_{p,z} = n_w(z).$$

- for each intertwining relation $s(\pi(a_1, \dots, a_k)) = \rho(s(a_1), \dots, s(a_r))$, and for all k-combinations of blocks $p_i \in P$ such that there exist $(a_1, \dots, a_k) \in p_1 \times \dots \times p_k$ with $\pi(a_1, \dots, a_k) \in W$ and $z = \rho(z_1, \dots, z_k)$,

$$x_{p,z} \geq 1 - k + \sum_{i=1}^{k} x_{p_i, z_i}.$$

- for each distribution restriction, i.e. a set of elements $\tilde{A}$ and a distribution of values given by integers d_z representing the multiplicity of each $z \in Z$, and $r_p = |p \cap \tilde{A}|$ indicating the relative size of block p in the set of elements of the distribution,

$$\sum_p r_p x_{p,z} = d_z.$$

EXAMPLES:

```
sage: A = B = 'abc'
sage: bij = Bijectionist(A, B, lambda x: B.index(x) % 2, solver='GLPK')
sage: next(bij.solutions_iterator())
{'a': 0, 'b': 1, 'c': 0}

sage: list(bij.solutions_iterator())
[{'a': 0, 'b': 1, 'c': 0},
 {'a': 1, 'b': 0, 'c': 0},
 {'a': 0, 'b': 0, 'c': 1}]

sage: N = 4
sage: A = B = [permutation for n in range(N) for permutation in␣
↪Permutations(n)]
```

```
>>> from sage.all import *
>>> A = B = 'abc'
>>> bij = Bijectionist(A, B, lambda x: B.index(x) % Integer(2), solver='GLPK')
>>> next(bij.solutions_iterator())
{'a': 0, 'b': 1, 'c': 0}

>>> list(bij.solutions_iterator())
[{'a': 0, 'b': 1, 'c': 0},
 {'a': 1, 'b': 0, 'c': 0},
 {'a': 0, 'b': 0, 'c': 1}]

>>> N = Integer(4)
>>> A = B = [permutation for n in range(N) for permutation in Permutations(n)]
```

Let τ be the number of non-left-to-right-maxima of a permutation:

```
sage: def tau(pi):
....:     pi = list(pi)
....:     i = count = 0
....:     for j in range(len(pi)):
....:         if pi[j] > i:
```

(continues on next page)

(continued from previous page)

```
....:             i = pi[j]
....:         else:
....:             count += 1
....:     return count
```

```
>>> from sage.all import *
>>> def tau(pi):
...     pi = list(pi)
...     i = count = Integer(0)
...     for j in range(len(pi)):
...         if pi[j] > i:
...             i = pi[j]
...         else:
...             count += Integer(1)
...     return count
```

We look for a statistic which is constant on conjugacy classes:

```
sage: P = [list(a) for n in range(N) for a in Permutations(n).conjugacy_
↪classes()]

sage: bij = Bijectionist(A, B, tau, solver='GLPK')
sage: bij.set_statistics((len, len))
sage: bij.set_constant_blocks(P)
sage: for solution in bij.solutions_iterator():
....:     print(solution)
{[]: 0, [1]: 0, [1, 2]: 1, [2, 1]: 0, [1, 2, 3]: 0, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [3, 2, 1]: 1, [2, 3, 1]: 2, [3, 1, 2]: 2}
{[]: 0, [1]: 0, [1, 2]: 0, [2, 1]: 1, [1, 2, 3]: 0, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [3, 2, 1]: 1, [2, 3, 1]: 2, [3, 1, 2]: 2}
```

```
>>> from sage.all import *
>>> P = [list(a) for n in range(N) for a in Permutations(n).conjugacy_
↪classes()]

>>> bij = Bijectionist(A, B, tau, solver='GLPK')
>>> bij.set_statistics((len, len))
>>> bij.set_constant_blocks(P)
>>> for solution in bij.solutions_iterator():
...     print(solution)
{[]: 0, [1]: 0, [1, 2]: 1, [2, 1]: 0, [1, 2, 3]: 0, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [3, 2, 1]: 1, [2, 3, 1]: 2, [3, 1, 2]: 2}
{[]: 0, [1]: 0, [1, 2]: 0, [2, 1]: 1, [1, 2, 3]: 0, [1, 3, 2]: 1, [2, 1, 3]:␣
↪1, [3, 2, 1]: 1, [2, 3, 1]: 2, [3, 1, 2]: 2}
```

Changing or re-setting problem parameters clears the internal cache. Setting the verbosity prints the MILP which is solved.:

```
sage: set_verbose(2)
sage: bij.set_constant_blocks(P)
sage: _ = list(bij.solutions_iterator())
Constraints are:
```

(continues on next page)

(continued from previous page)

```
    block []: 1 <= x_0 <= 1
    block [1]: 1 <= x_1 <= 1
    block [1, 2]: 1 <= x_2 + x_3 <= 1
    block [2, 1]: 1 <= x_4 + x_5 <= 1
    block [1, 2, 3]: 1 <= x_6 + x_7 + x_8 <= 1
    block [1, 3, 2]: 1 <= x_9 + x_10 + x_11 <= 1
    block [2, 3, 1]: 1 <= x_12 + x_13 + x_14 <= 1
    statistics: 1 <= x_0 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_1 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_2 + x_4 <= 1
    statistics: 1 <= x_3 + x_5 <= 1
    statistics: 0 <= <= 0
    statistics: 1 <= x_6 + 3 x_9 + 2 x_12 <= 1
    statistics: 3 <= x_7 + 3 x_10 + 2 x_13 <= 3
    statistics: 2 <= x_8 + 3 x_11 + 2 x_14 <= 2
Variables are:
    x_0: s([]) = 0
    x_1: s([1]) = 0
    x_2: s([1, 2]) = 0
    x_3: s([1, 2]) = 1
    x_4: s([2, 1]) = 0
    x_5: s([2, 1]) = 1
    x_6: s([1, 2, 3]) = 0
    x_7: s([1, 2, 3]) = 1
    x_8: s([1, 2, 3]) = 2
    x_9: s([1, 3, 2]) = s([2, 1, 3]) = s([3, 2, 1]) = 0
    x_10: s([1, 3, 2]) = s([2, 1, 3]) = s([3, 2, 1]) = 1
    x_11: s([1, 3, 2]) = s([2, 1, 3]) = s([3, 2, 1]) = 2
    x_12: s([2, 3, 1]) = s([3, 1, 2]) = 0
    x_13: s([2, 3, 1]) = s([3, 1, 2]) = 1
    x_14: s([2, 3, 1]) = s([3, 1, 2]) = 2
after vetoing
Constraints are:
    block []: 1 <= x_0 <= 1
    block [1]: 1 <= x_1 <= 1
    block [1, 2]: 1 <= x_2 + x_3 <= 1
    block [2, 1]: 1 <= x_4 + x_5 <= 1
    block [1, 2, 3]: 1 <= x_6 + x_7 + x_8 <= 1
    block [1, 3, 2]: 1 <= x_9 + x_10 + x_11 <= 1
    block [2, 3, 1]: 1 <= x_12 + x_13 + x_14 <= 1
    statistics: 1 <= x_0 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_1 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_2 + x_4 <= 1
    statistics: 1 <= x_3 + x_5 <= 1
```

(continues on next page)

(continued from previous page)

```
    statistics: 0 <= <= 0
    statistics: 1 <= x_6 + 3 x_9 + 2 x_12 <= 1
    statistics: 3 <= x_7 + 3 x_10 + 2 x_13 <= 3
    statistics: 2 <= x_8 + 3 x_11 + 2 x_14 <= 2
    veto: x_0 + x_1 + x_3 + x_4 + x_6 + x_10 + x_14 <= 6
after vetoing
Constraints are:
    block []: 1 <= x_0 <= 1
    block [1]: 1 <= x_1 <= 1
    block [1, 2]: 1 <= x_2 + x_3 <= 1
    block [2, 1]: 1 <= x_4 + x_5 <= 1
    block [1, 2, 3]: 1 <= x_6 + x_7 + x_8 <= 1
    block [1, 3, 2]: 1 <= x_9 + x_10 + x_11 <= 1
    block [2, 3, 1]: 1 <= x_12 + x_13 + x_14 <= 1
    statistics: 1 <= x_0 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_1 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_2 + x_4 <= 1
    statistics: 1 <= x_3 + x_5 <= 1
    statistics: 0 <= <= 0
    statistics: 1 <= x_6 + 3 x_9 + 2 x_12 <= 1
    statistics: 3 <= x_7 + 3 x_10 + 2 x_13 <= 3
    statistics: 2 <= x_8 + 3 x_11 + 2 x_14 <= 2
    veto: x_0 + x_1 + x_3 + x_4 + x_6 + x_10 + x_14 <= 6
    veto: x_0 + x_1 + x_2 + x_5 + x_6 + x_10 + x_14 <= 6

sage: set_verbose(0)
```

```
>>> from sage.all import *
>>> set_verbose(Integer(2))
>>> bij.set_constant_blocks(P)
>>> _ = list(bij.solutions_iterator())
Constraints are:
    block []: 1 <= x_0 <= 1
    block [1]: 1 <= x_1 <= 1
    block [1, 2]: 1 <= x_2 + x_3 <= 1
    block [2, 1]: 1 <= x_4 + x_5 <= 1
    block [1, 2, 3]: 1 <= x_6 + x_7 + x_8 <= 1
    block [1, 3, 2]: 1 <= x_9 + x_10 + x_11 <= 1
    block [2, 3, 1]: 1 <= x_12 + x_13 + x_14 <= 1
    statistics: 1 <= x_0 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_1 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_2 + x_4 <= 1
    statistics: 1 <= x_3 + x_5 <= 1
    statistics: 0 <= <= 0
```

(continues on next page)

(continued from previous page)

```
    statistics: 1 <= x_6 + 3 x_9 + 2 x_12 <= 1
    statistics: 3 <= x_7 + 3 x_10 + 2 x_13 <= 3
    statistics: 2 <= x_8 + 3 x_11 + 2 x_14 <= 2
Variables are:
    x_0: s([]) = 0
    x_1: s([1]) = 0
    x_2: s([1, 2]) = 0
    x_3: s([1, 2]) = 1
    x_4: s([2, 1]) = 0
    x_5: s([2, 1]) = 1
    x_6: s([1, 2, 3]) = 0
    x_7: s([1, 2, 3]) = 1
    x_8: s([1, 2, 3]) = 2
    x_9: s([1, 3, 2]) = s([2, 1, 3]) = s([3, 2, 1]) = 0
    x_10: s([1, 3, 2]) = s([2, 1, 3]) = s([3, 2, 1]) = 1
    x_11: s([1, 3, 2]) = s([2, 1, 3]) = s([3, 2, 1]) = 2
    x_12: s([2, 3, 1]) = s([3, 1, 2]) = 0
    x_13: s([2, 3, 1]) = s([3, 1, 2]) = 1
    x_14: s([2, 3, 1]) = s([3, 1, 2]) = 2
after vetoing
Constraints are:
    block []: 1 <= x_0 <= 1
    block [1]: 1 <= x_1 <= 1
    block [1, 2]: 1 <= x_2 + x_3 <= 1
    block [2, 1]: 1 <= x_4 + x_5 <= 1
    block [1, 2, 3]: 1 <= x_6 + x_7 + x_8 <= 1
    block [1, 3, 2]: 1 <= x_9 + x_10 + x_11 <= 1
    block [2, 3, 1]: 1 <= x_12 + x_13 + x_14 <= 1
    statistics: 1 <= x_0 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_1 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_2 + x_4 <= 1
    statistics: 1 <= x_3 + x_5 <= 1
    statistics: 0 <= <= 0
    statistics: 1 <= x_6 + 3 x_9 + 2 x_12 <= 1
    statistics: 3 <= x_7 + 3 x_10 + 2 x_13 <= 3
    statistics: 2 <= x_8 + 3 x_11 + 2 x_14 <= 2
    veto: x_0 + x_1 + x_3 + x_4 + x_6 + x_10 + x_14 <= 6
after vetoing
Constraints are:
    block []: 1 <= x_0 <= 1
    block [1]: 1 <= x_1 <= 1
    block [1, 2]: 1 <= x_2 + x_3 <= 1
    block [2, 1]: 1 <= x_4 + x_5 <= 1
    block [1, 2, 3]: 1 <= x_6 + x_7 + x_8 <= 1
    block [1, 3, 2]: 1 <= x_9 + x_10 + x_11 <= 1
    block [2, 3, 1]: 1 <= x_12 + x_13 + x_14 <= 1
    statistics: 1 <= x_0 <= 1
    statistics: 0 <= <= 0
```

(continues on next page)

(continued from previous page)

```
    statistics: 0 <= <= 0
    statistics: 1 <= x_1 <= 1
    statistics: 0 <= <= 0
    statistics: 0 <= <= 0
    statistics: 1 <= x_2 + x_4 <= 1
    statistics: 1 <= x_3 + x_5 <= 1
    statistics: 0 <= <= 0
    statistics: 1 <= x_6 + 3 x_9 + 2 x_12 <= 1
    statistics: 3 <= x_7 + 3 x_10 + 2 x_13 <= 3
    statistics: 2 <= x_8 + 3 x_11 + 2 x_14 <= 2
    veto: x_0 + x_1 + x_3 + x_4 + x_6 + x_10 + x_14 <= 6
    veto: x_0 + x_1 + x_2 + x_5 + x_6 + x_10 + x_14 <= 6

>>> set_verbose(Integer(0))
```

statistics_fibers()

Return a dictionary mapping statistic values in W to their preimages in A and B.

This is a (computationally) fast way to obtain a first impression which objects in A should be mapped to which objects in B.

EXAMPLES:

```
sage: A = B = [permutation for n in range(4) for permutation in␣
↪Permutations(n)]
sage: tau = Permutation.longest_increasing_subsequence_length
sage: def wex(p): return len(p.weak_excedences())
sage: def fix(p): return len(p.fixed_points())
sage: def des(p): return len(p.descents(final_descent=True)) if p else 0
sage: def adj(p): return len([e for (e, f) in zip(p, p[1:]+[0]) if e == f+1])
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_statistics((len, len), (wex, des), (fix, adj))
sage: table([[key, AB[0], AB[1]] for key, AB in bij.statistics_fibers().
↪items()])
  (0, 0, 0)   [[]]                                   [[]]
  (1, 1, 1)   [[1]]                                  [[1]]
  (2, 2, 2)   [[1, 2]]                               [[2, 1]]
  (2, 1, 0)   [[2, 1]]                               [[1, 2]]
  (3, 3, 3)   [[1, 2, 3]]                            [[3, 2, 1]]
  (3, 2, 1)   [[1, 3, 2], [2, 1, 3], [3, 2, 1]]    [[1, 3, 2], [2, 1, 3], [2,␣
↪3, 1]]
  (3, 2, 0)   [[2, 3, 1]]                            [[3, 1, 2]]
  (3, 1, 0)   [[3, 1, 2]]                            [[1, 2, 3]]
```

```
>>> from sage.all import *
>>> A = B = [permutation for n in range(Integer(4)) for permutation in␣
↪Permutations(n)]
>>> tau = Permutation.longest_increasing_subsequence_length
>>> def wex(p): return len(p.weak_excedences())
>>> def fix(p): return len(p.fixed_points())
>>> def des(p): return len(p.descents(final_descent=True)) if p else␣
↪Integer(0)
>>> def adj(p): return len([e for (e, f) in zip(p,␣
```

(continues on next page)

(continued from previous page)

```
↪p[Integer(1):]+[Integer(0)]) if e == f+Integer(1)])
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_statistics((len, len), (wex, des), (fix, adj))
>>> table([[key, AB[Integer(0)], AB[Integer(1)]] for key, AB in bij.
↪statistics_fibers().items()])
  (0, 0, 0)   [[]]                               [[]]
  (1, 1, 1)   [[1]]                              [[1]]
  (2, 2, 2)   [[1, 2]]                           [[2, 1]]
  (2, 1, 0)   [[2, 1]]                           [[1, 2]]
  (3, 3, 3)   [[1, 2, 3]]                        [[3, 2, 1]]
  (3, 2, 1)   [[1, 3, 2], [2, 1, 3], [3, 2, 1]]  [[1, 3, 2], [2, 1, 3], [2,␣
↪3, 1]]
  (3, 2, 0)   [[2, 3, 1]]                        [[3, 1, 2]]
  (3, 1, 0)   [[3, 1, 2]]                        [[1, 2, 3]]
```

statistics_table(*header=True*)

Provide information about all elements of A with corresponding α values and all elements of B with corresponding β and τ values.

INPUT:

- `header` – boolean (default: `True`); whether to include a header with the standard Greek letters

OUTPUT:

A pair of lists suitable for `table`, where

- the first contains the elements of A together with the values of α
- the second contains the elements of B together with the values of τ and β

EXAMPLES:

```
sage: A = B = [permutation for n in range(4) for permutation in␣
↪Permutations(n)]
sage: tau = Permutation.longest_increasing_subsequence_length
sage: def wex(p): return len(p.weak_excedences())
sage: def fix(p): return len(p.fixed_points())
sage: def des(p): return len(p.descents(final_descent=True)) if p else 0
sage: def adj(p): return len([e for (e, f) in zip(p, p[1:]+[0]) if e == f+1])
sage: bij = Bijectionist(A, B, tau)
sage: bij.set_statistics((wex, des), (fix, adj))
sage: a, b = bij.statistics_table()
sage: table(a, header_row=True, frame=True)
┌───────────┬────────┬────────┐
│ a         │ α_1(a) │ α_2(a) │
╞═══════════╪════════╪════════╡
│ []        │ 0      │ 0      │
├───────────┼────────┼────────┤
│ [1]       │ 1      │ 1      │
├───────────┼────────┼────────┤
│ [1, 2]    │ 2      │ 2      │
├───────────┼────────┼────────┤
│ [2, 1]    │ 1      │ 0      │
├───────────┼────────┼────────┤
```

(continues on next page)

(continued from previous page)

```
│ [1, 2, 3] │ 3         │ 3         │
├───────────┼───────────┼───────────┤
│ [1, 3, 2] │ 2         │ 1         │
├───────────┼───────────┼───────────┤
│ [2, 1, 3] │ 2         │ 1         │
├───────────┼───────────┼───────────┤
│ [2, 3, 1] │ 2         │ 0         │
├───────────┼───────────┼───────────┤
│ [3, 1, 2] │ 1         │ 0         │
├───────────┼───────────┼───────────┤
│ [3, 2, 1] │ 2         │ 1         │
└───────────┴───────────┴───────────┘
sage: table(b, header_row=True, frame=True)
┌───────────┬───┬───────────┬───────────┐
│ b         │ τ │ β_1(b)    │ β_2(b)    │
╞═══════════╪═══╪═══════════╪═══════════╡
│ []        │ 0 │ 0         │ 0         │
├───────────┼───┼───────────┼───────────┤
│ [1]       │ 1 │ 1         │ 1         │
├───────────┼───┼───────────┼───────────┤
│ [1, 2]    │ 2 │ 1         │ 0         │
├───────────┼───┼───────────┼───────────┤
│ [2, 1]    │ 1 │ 2         │ 2         │
├───────────┼───┼───────────┼───────────┤
│ [1, 2, 3] │ 3 │ 1         │ 0         │
├───────────┼───┼───────────┼───────────┤
│ [1, 3, 2] │ 2 │ 2         │ 1         │
├───────────┼───┼───────────┼───────────┤
│ [2, 1, 3] │ 2 │ 2         │ 1         │
├───────────┼───┼───────────┼───────────┤
│ [2, 3, 1] │ 2 │ 2         │ 1         │
├───────────┼───┼───────────┼───────────┤
│ [3, 1, 2] │ 2 │ 2         │ 0         │
├───────────┼───┼───────────┼───────────┤
│ [3, 2, 1] │ 1 │ 3         │ 3         │
└───────────┴───┴───────────┴───────────┘
```

```
>>> from sage.all import *
>>> A = B = [permutation for n in range(Integer(4)) for permutation in␣
↪Permutations(n)]
>>> tau = Permutation.longest_increasing_subsequence_length
>>> def wex(p): return len(p.weak_excedences())
>>> def fix(p): return len(p.fixed_points())
>>> def des(p): return len(p.descents(final_descent=True)) if p else␣
↪Integer(0)
>>> def adj(p): return len([e for (e, f) in zip(p,␣
↪p[Integer(1):]+[Integer(0)]) if e == f+Integer(1)])
>>> bij = Bijectionist(A, B, tau)
>>> bij.set_statistics((wex, des), (fix, adj))
>>> a, b = bij.statistics_table()
>>> table(a, header_row=True, frame=True)
┌───────────┬───────────┬───────────┐
```

(continues on next page)

(continued from previous page)

```
┌─────────────┬─────────┬─────────┐
│ a           │ α_1(a)  │ α_2(a)  │
╞═════════════╪═════════╪═════════╡
│ []          │ 0       │ 0       │
├─────────────┼─────────┼─────────┤
│ [1]         │ 1       │ 1       │
├─────────────┼─────────┼─────────┤
│ [1, 2]      │ 2       │ 2       │
├─────────────┼─────────┼─────────┤
│ [2, 1]      │ 1       │ 0       │
├─────────────┼─────────┼─────────┤
│ [1, 2, 3]   │ 3       │ 3       │
├─────────────┼─────────┼─────────┤
│ [1, 3, 2]   │ 2       │ 1       │
├─────────────┼─────────┼─────────┤
│ [2, 1, 3]   │ 2       │ 1       │
├─────────────┼─────────┼─────────┤
│ [2, 3, 1]   │ 2       │ 0       │
├─────────────┼─────────┼─────────┤
│ [3, 1, 2]   │ 1       │ 0       │
├─────────────┼─────────┼─────────┤
│ [3, 2, 1]   │ 2       │ 1       │
└─────────────┴─────────┴─────────┘
>>> table(b, header_row=True, frame=True)
┌─────────────┬───┬─────────┬─────────┐
│ b           │ τ │ β_1(b)  │ β_2(b)  │
╞═════════════╪═══╪═════════╪═════════╡
│ []          │ 0 │ 0       │ 0       │
├─────────────┼───┼─────────┼─────────┤
│ [1]         │ 1 │ 1       │ 1       │
├─────────────┼───┼─────────┼─────────┤
│ [1, 2]      │ 2 │ 1       │ 0       │
├─────────────┼───┼─────────┼─────────┤
│ [2, 1]      │ 1 │ 2       │ 2       │
├─────────────┼───┼─────────┼─────────┤
│ [1, 2, 3]   │ 3 │ 1       │ 0       │
├─────────────┼───┼─────────┼─────────┤
│ [1, 3, 2]   │ 2 │ 2       │ 1       │
├─────────────┼───┼─────────┼─────────┤
│ [2, 1, 3]   │ 2 │ 2       │ 1       │
├─────────────┼───┼─────────┼─────────┤
│ [2, 3, 1]   │ 2 │ 2       │ 1       │
├─────────────┼───┼─────────┼─────────┤
│ [3, 1, 2]   │ 2 │ 2       │ 0       │
├─────────────┼───┼─────────┼─────────┤
│ [3, 2, 1]   │ 1 │ 3       │ 3       │
└─────────────┴───┴─────────┴─────────┘
```

5.1.9 Binary Recurrence Sequences

This class implements several methods relating to general linear binary recurrence sequences, including a sieve to find perfect powers in integral linear binary recurrence sequences.

EXAMPLES:

```
sage: R = BinaryRecurrenceSequence(1,1)        #the Fibonacci sequence
sage: R(137)        #the 137th term of the Fibonacci sequence
19134702400093278081449423917
sage: R(137) == fibonacci(137)
True
sage: [R(i) % 4 for i in range(12)]
[0, 1, 1, 2, 3, 1, 0, 1, 1, 2, 3, 1]
sage: R.period(4)        #the period of the fibonacci sequence modulo 4
6
sage: R.pthpowers(2, 10**10)        # long time (7 seconds) -- in fact these are all␣
↪squares, c.f. [BMS06]
[0, 1, 2, 12]

sage: S = BinaryRecurrenceSequence(8,1)  #a Lucas sequence
sage: S.period(73)
148
sage: S(5) % 73 == S(5 +148) %73
True
sage: S.pthpowers(3, 10**10)    # long time (3 seconds) -- provably finds the indices␣
↪of all 3rd powers less than 10^10
[0, 1, 2]

sage: T = BinaryRecurrenceSequence(2,0,1,2)
sage: [T(i) for i in range(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
sage: T.is_degenerate()
True
sage: T.is_geometric()
True
sage: T.pthpowers(7, 10**30)                                                     #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: the degenerate binary recurrence sequence is geometric or quasigeometric
and has many pth powers
```

```
>>> from sage.all import *
>>> R = BinaryRecurrenceSequence(Integer(1),Integer(1))        #the Fibonacci sequence
>>> R(Integer(137))        #the 137th term of the Fibonacci sequence
19134702400093278081449423917
>>> R(Integer(137)) == fibonacci(Integer(137))
True
>>> [R(i) % Integer(4) for i in range(Integer(12))]
[0, 1, 1, 2, 3, 1, 0, 1, 1, 2, 3, 1]
>>> R.period(Integer(4))        #the period of the fibonacci sequence modulo 4
6
>>> R.pthpowers(Integer(2), Integer(10)**Integer(10))        # long time (7 seconds) -
↪- in fact these are all squares, c.f. [BMS06]
[0, 1, 2, 12]

>>> S = BinaryRecurrenceSequence(Integer(8),Integer(1))  #a Lucas sequence
>>> S.period(Integer(73))
```

(continues on next page)

(continued from previous page)

```
148
>>> S(Integer(5)) % Integer(73) == S(Integer(5) +Integer(148)) %Integer(73)
True
>>> S.pthpowers(Integer(3), Integer(10)**Integer(10))     # long time (3 seconds) --␣
↪provably finds the indices of all 3rd powers less than 10^10
[0, 1, 2]

>>> T = BinaryRecurrenceSequence(Integer(2),Integer(0),Integer(1),Integer(2))
>>> [T(i) for i in range(Integer(10))]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>> T.is_degenerate()
True
>>> T.is_geometric()
True
>>> T.pthpowers(Integer(7), Integer(10)**Integer(30))                                ␣
↪                         # needs sage.symbolic
Traceback (most recent call last):
...
ValueError: the degenerate binary recurrence sequence is geometric or quasigeometric
and has many pth powers
```

AUTHORS:

- Isabel Vogt (2013): initial version

See [SV2013], [BMS2006], and [SS1983].

class sage.combinat.binary_recurrence_sequences.**BinaryRecurrenceSequence**(*b*, *c*, *u0=0*, *u1=1*)

Bases: SageObject

Create a linear binary recurrence sequence defined by initial conditions u_0 and u_1 and recurrence relation $u_{n+2} = b * u_{n+1} + c * u_n$.

INPUT:

- b – integer; (partially determining the recurrence relation)
- c – integer; (partially determining the recurrence relation)
- u0 – integer; (the 0-th term of the binary recurrence sequence)
- u1 – integer; (the 1-st term of the binary recurrence sequence)

OUTPUT: an integral linear binary recurrence sequence defined by u_0, u_1, and $u_{n+2} = bu_{n+1} + cu_n$

See also

fibonacci(), *lucas_number1()*, *lucas_number2()*

EXAMPLES:

```
sage: R = BinaryRecurrenceSequence(3,3,2,1)
sage: R
Binary recurrence sequence defined by: u_n = 3 * u_{n-1} + 3 * u_{n-2};
With initial conditions: u_0 = 2, and u_1 = 1
```

```
>>> from sage.all import *
>>> R = BinaryRecurrenceSequence(Integer(3),Integer(3),Integer(2),Integer(1))
>>> R
Binary recurrence sequence defined by: u_n = 3 * u_{n-1} + 3 * u_{n-2};
With initial conditions: u_0 = 2, and u_1 = 1
```

is_arithmetic()

Decide whether the sequence is degenerate and an arithmetic sequence.

The sequence is arithmetic if and only if $u_1 - u_0 = u_2 - u_1 = u_3 - u_2$.

This corresponds to the matrix $F = [[0, 1], [c, b]]$ being nondiagonalizable and $\alpha/\beta = 1$.

EXAMPLES:

```
sage: S = BinaryRecurrenceSequence(2,-1)
sage: [S(i) for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: S.is_arithmetic()
True
```

```
>>> from sage.all import *
>>> S = BinaryRecurrenceSequence(Integer(2),-Integer(1))
>>> [S(i) for i in range(Integer(10))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> S.is_arithmetic()
True
```

is_degenerate()

Decide whether the binary recurrence sequence is degenerate.

Let α and β denote the roots of the characteristic polynomial $p(x) = x^2 - bx - c$. Let $a = u_1 - u_0\beta/(\beta - \alpha)$ and $b = u_1 - u_0\alpha/(\beta - \alpha)$. The sequence is, thus, given by $u_n = a\alpha^n - b\beta^n$. Then we say that the sequence is nondegenerate if and only if $a * b * \alpha * \beta \neq 0$ and α/β is not a root of unity.

More concretely, there are 4 classes of degeneracy, that can all be formulated in terms of the matrix $F = [[0, 1], [c, b]]$.

- F is singular – this corresponds to `c` = 0, and thus $\alpha * \beta = 0$. This sequence is geometric after term `u0` and so we call it `quasigeometric`
- $v = [[u_0], [u_1]]$ is an eigenvector of F – this corresponds to a `geometric` sequence with $a * b = 0$
- F is nondiagonalizable – this corresponds to $\alpha = \beta$. This sequence will be the point-wise product of an arithmetic and geometric sequence.
- F^k is scalar, for some $k > 1$ – this corresponds to α/β a k th root of unity. This sequence is a union of several geometric sequences, and so we again call it `quasigeometric`.

EXAMPLES:

```
sage: S = BinaryRecurrenceSequence(0,1)
sage: S.is_degenerate()
True
sage: S.is_geometric()
False
sage: S.is_quasigeometric()
```

(continues on next page)

(continued from previous page)

```
True

sage: R = BinaryRecurrenceSequence(3,-2)
sage: R.is_degenerate()
False

sage: T = BinaryRecurrenceSequence(2,-1)
sage: T.is_degenerate()
True
sage: T.is_arithmetic()
True
```

```
>>> from sage.all import *
>>> S = BinaryRecurrenceSequence(Integer(0),Integer(1))
>>> S.is_degenerate()
True
>>> S.is_geometric()
False
>>> S.is_quasigeometric()
True

>>> R = BinaryRecurrenceSequence(Integer(3),-Integer(2))
>>> R.is_degenerate()
False

>>> T = BinaryRecurrenceSequence(Integer(2),-Integer(1))
>>> T.is_degenerate()
True
>>> T.is_arithmetic()
True
```

is_geometric()

Decide whether the binary recurrence sequence is geometric - ie a geometric sequence.

This is a subcase of a degenerate binary recurrence sequence, for which $ab = 0$, i.e. $u_n/u_{n-1} = r$ for some value of r.

See *is_degenerate()* for a description of degeneracy and definitions of a and b.

EXAMPLES:

```
sage: S = BinaryRecurrenceSequence(2,0,1,2)
sage: [S(i) for i in range(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
sage: S.is_geometric()
True
```

```
>>> from sage.all import *
>>> S = BinaryRecurrenceSequence(Integer(2),Integer(0),Integer(1),Integer(2))
>>> [S(i) for i in range(Integer(10))]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>> S.is_geometric()
True
```

is_quasigeometric()

Decide whether the binary recurrence sequence is degenerate and similar to a geometric sequence, i.e. the union of multiple geometric sequences, or geometric after term `u0`.

If α/β is a k th root of unity, where $k > 1$, then necessarily $k = 2, 3, 4, 6$. Then $F = [[0, 1], [c, b]$ is diagonalizable, and $F^k = [[\alpha^k, 0], [0, \beta^k]]$ is a diagonal matrix. Thus for all values of j mod k, the j mod k terms of u_n form a geometric series.

If α or β is zero, this implies that $c = 0$. This is the case when F is singular. In this case, $u_1, u_2, u_3, ...$ is geometric.

EXAMPLES:

```
sage: S = BinaryRecurrenceSequence(0,1)
sage: [S(i) for i in range(10)]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
sage: S.is_quasigeometric()
True

sage: R = BinaryRecurrenceSequence(3,0)
sage: [R(i) for i in range(10)]
[0, 1, 3, 9, 27, 81, 243, 729, 2187, 6561]
sage: R.is_quasigeometric()
True
```

```
>>> from sage.all import *
>>> S = BinaryRecurrenceSequence(Integer(0),Integer(1))
>>> [S(i) for i in range(Integer(10))]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
>>> S.is_quasigeometric()
True

>>> R = BinaryRecurrenceSequence(Integer(3),Integer(0))
>>> [R(i) for i in range(Integer(10))]
[0, 1, 3, 9, 27, 81, 243, 729, 2187, 6561]
>>> R.is_quasigeometric()
True
```

period(*m*, *eventual*)

Return the period of the binary recurrence sequence modulo an integer `m`.

If n_1 is congruent to n_2 modulo `period(m)`, then u_{n_1} is is congruent to u_{n_2} modulo `m`.

INPUT:

- `m` – integer; modulo which the period of the recurrence relation is calculated
- `eventual` – boolean (default: `False`); if `True`, allow the sequence to be eventually periodic, rather than requiring it to be purely periodic. So n_1 might not be congruent to n_2 modulo `period(m)` unless n_1 and n_2 are large.

OUTPUT: integer (the period of the sequence modulo m)

EXAMPLES:

If $p = \pm 1 \mod 5$, then the period of the Fibonacci sequence mod p is $p-1$ (c.f. Lemma 3.3 of [BMS2006]).

```
sage: R = BinaryRecurrenceSequence(1,1)
sage: R.period(31)
30

sage: [R(i) % 4 for i in range(12)]
[0, 1, 1, 2, 3, 1, 0, 1, 1, 2, 3, 1]
sage: R.period(4)
6
```

```
>>> from sage.all import *
>>> R = BinaryRecurrenceSequence(Integer(1),Integer(1))
>>> R.period(Integer(31))
30

>>> [R(i) % Integer(4) for i in range(Integer(12))]
[0, 1, 1, 2, 3, 1, 0, 1, 1, 2, 3, 1]
>>> R.period(Integer(4))
6
```

This function works for degenerate sequences as well.

```
sage: S = BinaryRecurrenceSequence(2,0,1,2)
sage: S.is_degenerate()
True
sage: S.is_geometric()
True
sage: [S(i) % 17 for i in range(16)]
[1, 2, 4, 8, 16, 15, 13, 9, 1, 2, 4, 8, 16, 15, 13, 9]
sage: S.period(17)
8
```

```
>>> from sage.all import *
>>> S = BinaryRecurrenceSequence(Integer(2),Integer(0),Integer(1),Integer(2))
>>> S.is_degenerate()
True
>>> S.is_geometric()
True
>>> [S(i) % Integer(17) for i in range(Integer(16))]
[1, 2, 4, 8, 16, 15, 13, 9, 1, 2, 4, 8, 16, 15, 13, 9]
>>> S.period(Integer(17))
8
```

Letting `eventual` be `True` allows us to find the period of a sequence that is not purely periodic.

```
sage: T = BinaryRecurrenceSequence(5,12,u0=0,u1=1)
sage: [T(n) % 10 for n in range(20)]
[0, 1, 5, 7, 5, 9, 5, 3, 5, 1, 5, 7, 5, 9, 5, 3, 5, 1, 5, 7]
sage: T.period(10)
Traceback (most recent call last):
...
ValueError: Binary recurrence sequence modulo 10 is not a purely
periodic sequence.
```

(continues on next page)

(continued from previous page)

```
sage: T.period(10,eventual=True)
8
```

```
>>> from sage.all import *
>>> T = BinaryRecurrenceSequence(Integer(5),Integer(12),u0=Integer(0),
↪u1=Integer(1))
>>> [T(n) % Integer(10) for n in range(Integer(20))]
[0, 1, 5, 7, 5, 9, 5, 3, 5, 1, 5, 7, 5, 9, 5, 3, 5, 1, 5, 7]
>>> T.period(Integer(10))
Traceback (most recent call last):
...
ValueError: Binary recurrence sequence modulo 10 is not a purely
periodic sequence.
>>> T.period(Integer(10),eventual=True)
8
```

Note

The answer is cached.

pthpowers (*p*, *Bound*)

Find the indices of proveably all p-th powers in the recurrence sequence bounded by Bound.

Let u_n be a binary recurrence sequence. A `p` th power in u_n is a solution to $u_n = y^p$ for some integer y. There are only finitely many `p` th powers in any recurrence sequence [SS1983].

INPUT:

- `p` – a rational prime integer (the fixed p in $u_n = y^p$)
- `Bound` – a natural number (the maximum index n in $u_n = y^p$ that is checked)

OUTPUT:

A list of the indices of all `p` th powers less bounded by `Bound`. If the sequence is degenerate and there are many `p` th powers, raises `ValueError`.

EXAMPLES:

```
sage: R = BinaryRecurrenceSequence(1,1)        #the Fibonacci sequence
sage: R.pthpowers(2, 10**10)        # long time (7 seconds) -- in fact these␣
↪are all squares, c.f. [BMS2006]_
[0, 1, 2, 12]

sage: S = BinaryRecurrenceSequence(8,1) #a Lucas sequence
sage: S.pthpowers(3,10**10)    # long time (3 seconds) -- provably finds the␣
↪indices of all 3rd powers less than 10^10
[0, 1, 2]

sage: Q = BinaryRecurrenceSequence(3,3,2,1)
sage: Q.pthpowers(11,10**10)          # long time (7.5 seconds)
[1]
```

```
>>> from sage.all import *
>>> R = BinaryRecurrenceSequence(Integer(1),Integer(1))        #the Fibonacci␣
↪sequence
>>> R.pthpowers(Integer(2), Integer(10)**Integer(10))        # long time (7␣
↪seconds) -- in fact these are all squares, c.f. [BMS2006]_
[0, 1, 2, 12]

>>> S = BinaryRecurrenceSequence(Integer(8),Integer(1)) #a Lucas sequence
>>> S.pthpowers(Integer(3),Integer(10)**Integer(10))     # long time (3␣
↪seconds) -- provably finds the indices of all 3rd powers less than 10^10
[0, 1, 2]

>>> Q = BinaryRecurrenceSequence(Integer(3),Integer(3),Integer(2),Integer(1))
>>> Q.pthpowers(Integer(11),Integer(10)**Integer(10))          # long time (7.
↪5 seconds)
[1]
```

If the sequence is degenerate, and there are no `p` th powers, returns []. Otherwise, if there are many `p` th powers, raises `ValueError`.

```
sage: T = BinaryRecurrenceSequence(2,0,1,2)
sage: T.is_degenerate()
True
sage: T.is_geometric()
True
sage: T.pthpowers(7, 10**30)                                                  #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: the degenerate binary recurrence sequence is geometric or
quasigeometric and has many pth powers

sage: L = BinaryRecurrenceSequence(4,0,2,2)
sage: [L(i).factor() for i in range(10)]
[2, 2, 2^3, 2^5, 2^7, 2^9, 2^11, 2^13, 2^15, 2^17]
sage: L.is_quasigeometric()
True
sage: L.pthpowers(2, 10**30)                                                  #␣
↪needs sage.symbolic
[]
```

```
>>> from sage.all import *
>>> T = BinaryRecurrenceSequence(Integer(2),Integer(0),Integer(1),Integer(2))
>>> T.is_degenerate()
True
>>> T.is_geometric()
True
>>> T.pthpowers(Integer(7), Integer(10)**Integer(30))                          ␣
↪                       # needs sage.symbolic
Traceback (most recent call last):
...
ValueError: the degenerate binary recurrence sequence is geometric or
quasigeometric and has many pth powers
```

(continues on next page)

(continued from previous page)

```
>>> L = BinaryRecurrenceSequence(Integer(4),Integer(0),Integer(2),Integer(2))
>>> [L(i).factor() for i in range(Integer(10))]
[2, 2, 2^3, 2^5, 2^7, 2^9, 2^11, 2^13, 2^15, 2^17]
>>> L.is_quasigeometric()
True
>>> L.pthpowers(Integer(2), Integer(10)**Integer(30))                    ␣
↪                         # needs sage.symbolic
[]
```

Note

This function is primarily optimized in the range where `Bound` is much larger than `p`.

5.1.10 Binary Trees

This module deals with binary trees as mathematical (in particular immutable) objects.

Note

If you need the data-structure for example to represent sets or hash tables with AVL trees, you should have a look at `sage.misc.sagex_ds`.

AUTHORS:

- Florent Hivert (2010-2011): initial implementation.
- Adrien Boussicault (2015): Hook statistics.

class `sage.combinat.binary_tree.`**`BinaryTree`**(*parent*, *children=None*, *check=True*)

Bases: *`AbstractClonableTree`*, `ClonableArray`

Binary trees.

Binary trees here mean ordered (a.k.a. plane) finite binary trees, where "ordered" means that the children of each node are ordered.

Binary trees contain nodes and leaves, where each node has two children while each leaf has no children. The number of leaves of a binary tree always equals the number of nodes plus 1.

INPUT:

- `children` – `None` (default) or a list, tuple or iterable of length 2 of binary trees or convertible objects. This corresponds to the standard recursive definition of a binary tree as either a leaf or a pair of binary trees. Syntactic sugar allows leaving out all but the outermost calls of the `BinaryTree()` constructor, so that, e. g., `BinaryTree([BinaryTree(None),BinaryTree(None)])` can be shortened to `BinaryTree([None, None])`. It is also allowed to abbreviate `[None, None]` by `[]`.
- `check` – boolean (default: `True`); whether check for binary should be performed or not

EXAMPLES:

```
sage: BinaryTree()
.
```

(continues on next page)

(continued from previous page)

```
sage: BinaryTree(None)
.
sage: BinaryTree([])
[., .]
sage: BinaryTree([None, None])
[., .]
sage: BinaryTree([None, []])
[., [., .]]
sage: BinaryTree([[], None])
[[., .], .]
sage: BinaryTree("[[], .]")
[[., .], .]
sage: BinaryTree([None, BinaryTree([None, None])])
[., [., .]]

sage: BinaryTree([[], None, []])
Traceback (most recent call last):
...
ValueError: this is not a binary tree
```

```
>>> from sage.all import *
>>> BinaryTree()
.
>>> BinaryTree(None)
.
>>> BinaryTree([])
[., .]
>>> BinaryTree([None, None])
[., .]
>>> BinaryTree([None, []])
[., [., .]]
>>> BinaryTree([[], None])
[[., .], .]
>>> BinaryTree("[[], .]")
[[., .], .]
>>> BinaryTree([None, BinaryTree([None, None])])
[., [., .]]

>>> BinaryTree([[], None, []])
Traceback (most recent call last):
...
ValueError: this is not a binary tree
```

as_ordered_tree(*with_leaves=True*)

Return the same tree seen as an ordered tree. By default, leaves are transformed into actual nodes, but this can be avoided by setting the optional variable `with_leaves` to `False`.

EXAMPLES:

```
sage: bt = BinaryTree([]); bt
[., .]
sage: bt.as_ordered_tree()
```

(continues on next page)

(continued from previous page)

```
[[], []]
sage: bt.as_ordered_tree(with_leaves = False)
[]
sage: bt = bt.canonical_labelling(); bt
1[., .]
sage: bt.as_ordered_tree()
1[None[], None[]]
sage: bt.as_ordered_tree(with_leaves=False)
1[]
```

```
>>> from sage.all import *
>>> bt = BinaryTree([]); bt
[., .]
>>> bt.as_ordered_tree()
[[], []]
>>> bt.as_ordered_tree(with_leaves = False)
[]
>>> bt = bt.canonical_labelling(); bt
1[., .]
>>> bt.as_ordered_tree()
1[None[], None[]]
>>> bt.as_ordered_tree(with_leaves=False)
1[]
```

canonical_labelling(*shift=1*)

Return a labelled version of `self`.

The canonical labelling of a binary tree is a certain labelling of the nodes (not the leaves) of the tree. The actual canonical labelling is currently unspecified. However, it is guaranteed to have labels in $1...n$ where n is the number of nodes of the tree. Moreover, two (unlabelled) trees compare as equal if and only if their canonical labelled trees compare as equal.

EXAMPLES:

```
sage: BinaryTree().canonical_labelling()
.
sage: BinaryTree([]).canonical_labelling()
1[., .]
sage: BinaryTree([[[], [[], None]], [[], []]]).canonical_labelling()
5[2[1[., .], 4[3[., .], .]], 7[6[., .], 8[., .]]]
```

```
>>> from sage.all import *
>>> BinaryTree().canonical_labelling()
.
>>> BinaryTree([]).canonical_labelling()
1[., .]
>>> BinaryTree([[[], [[], None]], [[], []]]).canonical_labelling()
5[2[1[., .], 4[3[., .], .]], 7[6[., .], 8[., .]]]
```

canopee()

Return the canopee of `self`.

The *canopee* of a non-empty binary tree T with n internal nodes is the list l of 0 and 1 of length $n-1$ obtained by going along the leaves of T from left to right except the two extremal ones, writing 0 if the leaf is a right leaf and 1 if the leaf is a left leaf.

EXAMPLES:

```
sage: BinaryTree([]).canopee()
[]
sage: BinaryTree([None, []]).canopee()
[1]
sage: BinaryTree([[], None]).canopee()
[0]
sage: BinaryTree([[], []]).canopee()
[0, 1]
sage: BinaryTree([[[], [[], None]], [[], []]]).canopee()
[0, 1, 0, 0, 1, 0, 1]
```

```
>>> from sage.all import *
>>> BinaryTree([]).canopee()
[]
>>> BinaryTree([None, []]).canopee()
[1]
>>> BinaryTree([[], None]).canopee()
[0]
>>> BinaryTree([[], []]).canopee()
[0, 1]
>>> BinaryTree([[[], [[], None]], [[], []]]).canopee()
[0, 1, 0, 0, 1, 0, 1]
```

The number of pairs (t_1, t_2) of binary trees of size n such that the canopee of t_1 is the complementary of the canopee of t_2 is also the number of Baxter permutations (see [DG1994], see also OEIS sequence A001181). We check this in small cases:

```
sage: [len([(u,v) for u in BinaryTrees(n) for v in BinaryTrees(n)
....:       if [1 - x for x in u.canopee()] == v.canopee()])
....:    for n in range(1, 5)]
[1, 2, 6, 22]
```

```
>>> from sage.all import *
>>> [len([(u,v) for u in BinaryTrees(n) for v in BinaryTrees(n)
...       if [Integer(1) - x for x in u.canopee()] == v.canopee()])
...    for n in range(Integer(1), Integer(5))]
[1, 2, 6, 22]
```

Here is a less trivial implementation of this:

```
sage: from sage.sets.finite_set_map_cy import fibers
sage: def baxter(n):
....:     f = fibers(lambda t: tuple(t.canopee()),
....:                   BinaryTrees(n))
....:     return sum(len(f[i])*len(f[tuple(1-x for x in i)])
....:                for i in f)
sage: [baxter(n) for n in range(1, 7)]
[1, 2, 6, 22, 92, 422]
```

```
>>> from sage.all import *
>>> from sage.sets.finite_set_map_cy import fibers
>>> def baxter(n):
...     f = fibers(lambda t: tuple(t.canopee()),
...                   BinaryTrees(n))
...     return sum(len(f[i])*len(f[tuple(Integer(1)-x for x in i)])
...                for i in f)
>>> [baxter(n) for n in range(Integer(1), Integer(7))]
[1, 2, 6, 22, 92, 422]
```

check()

Check that `self` is a binary tree.

EXAMPLES:

```
sage: BinaryTree([[], []])     # indirect doctest
[[., .], [., .]]
sage: BinaryTree([[], [], []]) # indirect doctest
Traceback (most recent call last):
...
ValueError: this is not a binary tree
sage: BinaryTree([[]])         # indirect doctest
Traceback (most recent call last):
...
ValueError: this is not a binary tree
```

```
>>> from sage.all import *
>>> BinaryTree([[], []])     # indirect doctest
[[., .], [., .]]
>>> BinaryTree([[], [], []]) # indirect doctest
Traceback (most recent call last):
...
ValueError: this is not a binary tree
>>> BinaryTree([[]])         # indirect doctest
Traceback (most recent call last):
...
ValueError: this is not a binary tree
```

comb(*side='left'*)

Return the comb of a tree.

There are two combs in a binary tree: a left comb and a right comb.

Consider all the vertices of the leftmost (resp. rightmost) branch of the root. The left (resp. right) comb is the list of right (resp. left) subtrees of each of these vertices.

INPUT:

- `side` – (default: `'left'`) set to `'left'` to obtain a left comb, and to `'right'` to obtain a right comb

OUTPUT: list of binary trees

See also

over_decomposition(), *under_decomposition()*

EXAMPLES:

```
sage: BT = BinaryTree( '.' )
sage: [BT.comb('left'), BT.comb('right')]
[[], []]
sage: BT = BinaryTree( '[.,.]' )
sage: [BT.comb('left'), BT.comb('right')]
[[], []]
sage: BT = BinaryTree( '[[[.,.], .], [.,.]]' )
sage: BT.comb('left')
[., .]
sage: BT.comb('right')
[.]
sage: BT = BinaryTree( '[[[[., [., .]], .], [[., .], [[[., .], [., .]], [., .
↪]]]], [., [[[., .], [[[., .], [., .]], .]], .]]]' )
sage: ascii_art(BT)
        ________o________
       /                 \
    __o__                 o
   /     \                 \
  o     __o___              o
 /     /      \            /
o     o       _o_       __o__
 \           /   \     /     \
  o         o     o   o       o
           / \               /
          o   o             o
                           / \
                          o   o
sage: BT.comb('left')
[[[., .], [[[., .], [., .]], [., .]]], ., [., .]]
sage: ascii_art(BT.comb('left'))
[   __o___     , , o ]
[  /      \          ]
[ o       _o_        ]
[        /   \       ]
[       o     o      ]
[      / \           ]
[     o   o          ]
sage: BT.comb('right')
[., [[., .], [[[., .], [., .]], .]]]
sage: ascii_art(BT.comb('right'))
[ ,   __o__   ]
[    /     \  ]
[   o       o ]
[          /  ]
[         o   ]
[        / \  ]
[       o   o ]
```

```
>>> from sage.all import *
>>> BT = BinaryTree( '.' )
>>> [BT.comb('left'), BT.comb('right')]
```

(continues on next page)

(continued from previous page)

```
[[], []]
>>> BT = BinaryTree( '[.,.]' )
>>> [BT.comb('left'), BT.comb('right')]
[[], []]
>>> BT = BinaryTree( '[[[.,.], .], [.,.]]' )
>>> BT.comb('left')
[., .]
>>> BT.comb('right')
[.]
>>> BT = BinaryTree( '[[[[., [., .]], .], [[., .], [[[., .], [., .]], [., .]]]], [., [[[., .], [[[., .], [., .]], .]], .]]]' )
>>> ascii_art(BT)
        ________o________
       /                 \
    __o__                 o
   /     \                 \
  o     __o___              o
 /     /      \            /
o     o       _o_       __o__
 \           /   \     /     \
  o         o     o   o       o
           / \               /
          o   o             o
                           / \
                          o   o
>>> BT.comb('left')
[[[., .], [[[., .], [., .]], [., .]]], ., [., .]]
>>> ascii_art(BT.comb('left'))
[   __o___     , , o ]
[  /      \          ]
[ o       _o_        ]
[        /   \       ]
[       o     o      ]
[      / \           ]
[     o   o          ]
>>> BT.comb('right')
[., [[., .], [[[., .], [., .]], .]]]
>>> ascii_art(BT.comb('right'))
[ ,   __o__   ]
[    /     \  ]
[   o       o ]
[          /  ]
[         o   ]
[        / \  ]
[       o   o ]
```

dendriform_shuffle(*other*)

Return the list of terms in the dendriform product.

This is the list of all binary trees that can be obtained by identifying the rightmost path in `self` and the leftmost path in `other`. Every term corresponds to a shuffle of the vertices on the rightmost path in `self` and the vertices on the leftmost path in `other`.

EXAMPLES:

```
sage: u = BinaryTree()
sage: g = BinaryTree([])
sage: l = BinaryTree([g, u])
sage: r = BinaryTree([u, g])

sage: list(g.dendriform_shuffle(g))                                    #␣
↪needs sage.combinat
[[[., .], .], [., [., .]]]

sage: list(l.dendriform_shuffle(l))                                    #␣
↪needs sage.combinat
[[[[[., .], .], .], .], [[[., .], [., .]], .],
[[., .], [[., .], .]]]

sage: list(l.dendriform_shuffle(r))                                    #␣
↪needs sage.combinat
[[[[., .], .], [., .]], [[., .], [., [., .]]]]
```

```
>>> from sage.all import *
>>> u = BinaryTree()
>>> g = BinaryTree([])
>>> l = BinaryTree([g, u])
>>> r = BinaryTree([u, g])

>>> list(g.dendriform_shuffle(g))                                    #␣
↪needs sage.combinat
[[[., .], .], [., [., .]]]

>>> list(l.dendriform_shuffle(l))                                    #␣
↪needs sage.combinat
[[[[[., .], .], .], .], [[[., .], [., .]], .],
[[., .], [[., .], .]]]

>>> list(l.dendriform_shuffle(r))                                    #␣
↪needs sage.combinat
[[[[., .], .], [., .]], [[., .], [., [., .]]]]
```

graph(*with_leaves=True*)

Convert `self` to a digraph.

By default, this graph contains both nodes and leaves, hence is never empty. To obtain a graph which contains only the nodes, the `with_leaves` optional keyword variable has to be set to `False`.

The resulting digraph is endowed with a combinatorial embedding, in order to be displayed correctly.

INPUT:

- `with_leaves` – boolean (default: `True`); whether the resulting graph will be formed from the leaves and the nodes of `self` (if `True`), or only from the nodes of `self` (if `False`)

EXAMPLES:

```
sage: t1 = BinaryTree([[], None])
sage: t1.graph()
```

(continues on next page)

(continued from previous page)

```
Digraph on 5 vertices
sage: t1.graph(with_leaves=False)
Digraph on 2 vertices

sage: t1 = BinaryTree([[], [[], None]])
sage: t1.graph()
Digraph on 9 vertices
sage: t1.graph().edges(sort=True)
[(0, 1, None), (0, 4, None), (1, 2, None), (1, 3, None), (4, 5, None), (4, 8,␣
↪None), (5, 6, None), (5, 7, None)]
sage: t1.graph(with_leaves=False)
Digraph on 4 vertices
sage: t1.graph(with_leaves=False).edges(sort=True)
[(0, 1, None), (0, 2, None), (2, 3, None)]

sage: t1 = BinaryTree()
sage: t1.graph()
Digraph on 1 vertex
sage: t1.graph(with_leaves=False)
Digraph on 0 vertices

sage: BinaryTree([]).graph()
Digraph on 3 vertices
sage: BinaryTree([]).graph(with_leaves=False)
Digraph on 1 vertex

sage: t1 = BinaryTree([[], [[], []]])
sage: t1.graph(with_leaves=False)
Digraph on 5 vertices
sage: t1.graph(with_leaves=False).edges(sort=True)
[(0, 1, None), (0, 2, None), (2, 3, None), (2, 4, None)]
```

```
>>> from sage.all import *
>>> t1 = BinaryTree([[], None])
>>> t1.graph()
Digraph on 5 vertices
>>> t1.graph(with_leaves=False)
Digraph on 2 vertices

>>> t1 = BinaryTree([[], [[], None]])
>>> t1.graph()
Digraph on 9 vertices
>>> t1.graph().edges(sort=True)
[(0, 1, None), (0, 4, None), (1, 2, None), (1, 3, None), (4, 5, None), (4, 8,␣
↪None), (5, 6, None), (5, 7, None)]
>>> t1.graph(with_leaves=False)
Digraph on 4 vertices
>>> t1.graph(with_leaves=False).edges(sort=True)
[(0, 1, None), (0, 2, None), (2, 3, None)]

>>> t1 = BinaryTree()
>>> t1.graph()
```

(continues on next page)

(continued from previous page)

```
Digraph on 1 vertex
>>> t1.graph(with_leaves=False)
Digraph on 0 vertices

>>> BinaryTree([]).graph()
Digraph on 3 vertices
>>> BinaryTree([]).graph(with_leaves=False)
Digraph on 1 vertex

>>> t1 = BinaryTree([[], [[], []]])
>>> t1.graph(with_leaves=False)
Digraph on 5 vertices
>>> t1.graph(with_leaves=False).edges(sort=True)
[(0, 1, None), (0, 2, None), (2, 3, None), (2, 4, None)]
```

hook_number()

Return the number of hooks.

Recalling that a branch is a path from a vertex of the tree to a leaf, the leftmost (resp. rightmost) branch of a vertex v is the branch from v made only of left (resp. right) edges.

The hook of a vertex v is a set of vertices formed by the union of v, and the vertices of its leftmost and rightmost branches.

There is a unique way to partition the set of vertices in hooks. The number of hooks in such a partition is the hook number of the tree.

We can obtain this partition recursively by extracting the root's hook and iterating the processus on each tree of the remaining forest.

EXAMPLES:

```
sage: BT = BinaryTree( '.' )
sage: BT.hook_number()
0
sage: BT = BinaryTree( '[.,.]' )
sage: BT.hook_number()
1
sage: BT = BinaryTree( '[[[.,.], .], [.,.]]' ); ascii_art(BT)
    o
   / \
  o   o
 /
o
sage: BT.hook_number()
1
sage: BT = BinaryTree( '[[[[., [., .]], .], [[., .], [[[., .], [., .]], [., .
↪]]]], [., [[[., .], [[[., .], [., .]], .]], .]]]' )
sage: ascii_art(BT)
        ________o________
       /                  \
    __o__                  o
   /     \                  \
  o     __o___               o
```

(continues on next page)

(continued from previous page)

```
 /     /       \            /
o     o       _o_       __o__
 \           /   \     /     \
  o         o     o   o       o
           / \               /
          o   o             o
                           / \
                          o   o
sage: BT.hook_number()
6
```

```
>>> from sage.all import *
>>> BT = BinaryTree( '.' )
>>> BT.hook_number()
0
>>> BT = BinaryTree( '[.,.]' )
>>> BT.hook_number()
1
>>> BT = BinaryTree( '[[[.,.], .], [.,.]]' ); ascii_art(BT)
    o
   / \
  o   o
 /
o
>>> BT.hook_number()
1
>>> BT = BinaryTree( '[[[[., [., .]], .], [[., .], [[[., .], [., .]], [., .
↪]]]], [., [[[., .], [[[., .], [., .]], .]], .]]]' )
>>> ascii_art(BT)
        ________o________
       /                 \
    __o__                 o
   /     \                 \
  o     __o___              o
 /     /      \            /
o     o       _o_       __o__
 \           /   \     /     \
  o         o     o   o       o
           / \               /
          o   o             o
                           / \
                          o   o
>>> BT.hook_number()
6
```

in_order_traversal(*node_action=None*, *leaf_action=None*)

Explore the binary tree `self` using the depth-first infix-order traversal algorithm, executing the `node_action` function whenever traversing a node and executing the `leaf_action` function whenever traversing a leaf.

In more detail, what this method does to a tree T is the following:

```
if the root of `T` is a node:
    apply in_order_traversal to the left subtree of `T`
        (with the same node_action and leaf_action);
    apply node_action to the root of `T`;
    apply in_order_traversal to the right subtree of `T`
        (with the same node_action and leaf_action);
else:
    apply leaf_action to the root of `T`.
```

For example on the following binary tree T, where we denote leaves by $a, b, c, \ldots$ and nodes by $1, 2, 3, \ldots$:

```
|     ____3____          |
|    /         \         |
|   1          __7__     |
|  / \        /     \    |
| a   2      _5_     8   |
|    / \    /   \   / \  |
|   b   c  4     6 h   i |
|         / \   / \      |
|        d   e f   g     |
```

this method first applies `leaf_action` to a, then applies `node_action` to 1, then `leaf_action` to b, then `node_action` to 2, etc., with the vertices being traversed in the order $a, 1, b, 2, c, 3, d, 4, e, 5, f, 6, g, 7, h, 8, i$.

See *in_order_traversal_iter()* for a version of this algorithm which only iterates through the vertices rather than applying any function to them.

INPUT:

- `node_action` – (optional) a function which takes a node in input and does something during the exploration
- `leaf_action` – (optional) a function which takes a leaf in input and does something during the exploration

in_order_traversal_iter()

The depth-first infix-order traversal iterator for the binary tree `self`.

This method iters each vertex (node and leaf alike) of the given binary tree following the depth-first infix order traversal algorithm.

The *depth-first infix order traversal algorithm* iterates through a binary tree as follows:

```
iterate through the left subtree (by the depth-first infix
    order traversal algorithm);
yield the root;
iterate through the right subtree (by the depth-first infix
    order traversal algorithm).
```

For example on the following binary tree T, where we denote leaves by $a, b, c, \ldots$ and nodes by $1, 2, 3, \ldots$:

```
|     ____3____          |
|    /         \         |
|   1          __7__     |
|  / \        /     \    |
```

(continues on next page)

(continued from previous page)

```
| a   2      _5_     8   |
|    / \    /   \   / \  |
|   b   c  4     6 h   i |
|         / \   / \      |
|        d   e f   g     |
```

the depth-first infix-order traversal algorithm iterates through the vertices of T in the following order: $a, 1, b, 2, c, 3, d, 4, e, 5, f, 6, g, 7, h, 8, i$.

See *in_order_traversal()* for a version of this algorithm which not only iterates through, but actually does something at the vertices of tree.

is_complete()

Return `True` if `self` is complete, else return `False`.

In a nutshell, a complete binary tree is a perfect binary tree except possibly in the last level, with all nodes in the last level "flush to the left".

In more detail: A complete binary tree (also called binary heap) is a binary tree in which every level, except possibly the last one (the deepest), is completely filled. At depth n, all nodes must be as far left as possible.

For example:

```
|         ___o___   |
|        /       \  |
|     __o__       o |
|    /     \        |
|   o       o       |
|  / \     / \      |
| o   o   o   o     |
```

is not complete but the following ones are:

```
|     __o__          _o_            ___o___     |
|    /     \        /   \          /       \    |
|   o       o      o     o      __o__       o   |
|  / \     / \    / \          /     \     / \  |
| o   o   o   o, o   o    ,   o       o   o   o |
|                            / \     /          |
|                           o   o   o           |
```

EXAMPLES:

```
sage: def lst(i):
....:     return [bt for bt in BinaryTrees(i) if bt.is_complete()]
sage: for i in range(8): ascii_art(lst(i)) # long time
[   ]
[ o ]
[   o ]
[  /  ]
[ o   ]
[   o   ]
[  / \  ]
[ o   o ]
[     o   ]
```

(continues on next page)

(continued from previous page)

```
[    / \  ]
[   o   o ]
[  /      ]
[ o       ]
[     _o_   ]
[    /   \  ]
[   o     o ]
[  / \      ]
[ o   o     ]
[     __o__   ]
[    /     \  ]
[   o       o ]
[  / \     /  ]
[ o   o   o   ]
[     __o__     ]
[    /     \    ]
[   o       o   ]
[  / \     / \  ]
[ o   o   o   o ]
```

```
>>> from sage.all import *
>>> def lst(i):
...     return [bt for bt in BinaryTrees(i) if bt.is_complete()]
>>> for i in range(Integer(8)): ascii_art(lst(i)) # long time
[  ]
[ o ]
[   o ]
[  /  ]
[ o   ]
[   o   ]
[  / \  ]
[ o   o ]
[     o   ]
[    / \  ]
[   o   o ]
[  /      ]
[ o       ]
[     _o_   ]
[    /   \  ]
[   o     o ]
[  / \      ]
[ o   o     ]
[     __o__   ]
[    /     \  ]
[   o       o ]
[  / \     /  ]
[ o   o   o   ]
[     __o__     ]
[    /     \    ]
[   o       o   ]
[  / \     / \  ]
[ o   o   o   o ]
```

is_empty()

Return whether `self` is empty.

The notion of emptiness employed here is the one which defines a binary tree to be empty if its root is a leaf. There is precisely one empty binary tree.

EXAMPLES:

```
sage: BinaryTree().is_empty()
True
sage: BinaryTree([]).is_empty()
False
sage: BinaryTree([[], None]).is_empty()
False
```

```
>>> from sage.all import *
>>> BinaryTree().is_empty()
True
>>> BinaryTree([]).is_empty()
False
>>> BinaryTree([[], None]).is_empty()
False
```

is_full()

Return `True` if `self` is full, else return `False`.

A full binary tree is a tree in which every node either has two child nodes or has two child leaves.

This is also known as *proper binary tree* or *2-tree* or *strictly binary tree*.

For example:

```
|       __o__   |
|      /     \  |
|     o       o |
|    / \        |
|   o   o       |
|  /     \      |
| o       o     |
```

is not full but the next one is:

```
|         ___o___   |
|        /       \  |
|     __o__       o |
|    /     \        |
|   o       o       |
|  / \     / \      |
| o   o   o   o     |
```

EXAMPLES:

```
sage: BinaryTree([[[[],None],[None,[]]], []]).is_full()
False
sage: BinaryTree([[[[],[]],[[],[]]], []]).is_full()
True
```

(continues on next page)

(continued from previous page)

```
sage: ascii_art([bt for bt in BinaryTrees(5) if bt.is_full()])
[   _o_     ,      _o_   ]
[  /   \          /   \  ]
[ o     o       o     o ]
[      / \     / \       ]
[     o   o   o   o      ]
```

```
>>> from sage.all import *
>>> BinaryTree([[[[],None],[None,[]]], []]).is_full()
False
>>> BinaryTree([[[[],[]],[[],[]]], []]).is_full()
True
>>> ascii_art([bt for bt in BinaryTrees(Integer(5)) if bt.is_full()])
[   _o_     ,      _o_   ]
[  /   \          /   \  ]
[ o     o       o     o ]
[      / \     / \       ]
[     o   o   o   o      ]
```

is_perfect()

Return `True` if `self` is perfect, else return `False`.

A perfect binary tree is a full tree in which all leaves are at the same depth.

For example:

```
|          ___o___    |
|         /       \   |
|      __o__       o  |
|     /     \         |
|    o       o        |
|   / \     / \       |
|  o   o   o   o      |
```

is not perfect but the next one is:

```
|      __o__      |
|     /     \     |
|    o       o    |
|   / \     / \   |
|  o   o   o   o  |
```

EXAMPLES:

```
sage: def lst(i):
....:     return [bt for bt in BinaryTrees(i) if bt.is_perfect()]
sage: for i in range(8): ascii_art(lst(i)) # long time
[   ]
[ o ]
[   ]
[   o   ]
[  / \  ]
[ o   o ]
```

(continues on next page)

(continued from previous page)

```
[   ]
[   ]
[   ]
[      __o__      ]
[     /     \     ]
[    o       o    ]
[   / \     / \   ]
[ o   o   o   o ]
```

```
>>> from sage.all import *
>>> def lst(i):
...     return [bt for bt in BinaryTrees(i) if bt.is_perfect()]
>>> for i in range(Integer(8)): ascii_art(lst(i)) # long time
[   ]
[ o ]
[   ]
[    o    ]
[   / \   ]
[ o   o ]
[   ]
[   ]
[   ]
[      __o__      ]
[     /     \     ]
[    o       o    ]
[   / \     / \   ]
[ o   o   o   o ]
```

left_border_symmetry()

Return the tree where a symmetry has been applied recursively on all left borders. If a tree is made of three trees $[T_1, T_2, T_3]$ on its left border, it becomes $[T_3', T_2', T_1']$ where same symmetry has been applied to T_1, T_2, T_3.

EXAMPLES:

```
sage: BinaryTree().left_border_symmetry()
.
sage: BinaryTree([]).left_border_symmetry()
[., .]
sage: BinaryTree([[None,[]],None]).left_border_symmetry()
[[., .], [., .]]
sage: BinaryTree([[None,[None,[]]],None]).left_border_symmetry()
[[., .], [., [., .]]]
sage: bt = BinaryTree([[None,[None,[]]],None]).canonical_labelling()
sage: bt
4[1[., 2[., 3[., .]]], .]
sage: bt.left_border_symmetry()
1[4[., .], 2[., 3[., .]]]
```

```
>>> from sage.all import *
>>> BinaryTree().left_border_symmetry()
.
```

(continues on next page)

(continued from previous page)

```
>>> BinaryTree([]).left_border_symmetry()
[., .]
>>> BinaryTree([[None,[]],None]).left_border_symmetry()
[[., .], [., .]]
>>> BinaryTree([[None,[None,[]]],None]).left_border_symmetry()
[[., .], [., [., .]]]
>>> bt = BinaryTree([[None,[None,[]]],None]).canonical_labelling()
>>> bt
4[1[., 2[., 3[., .]]], .]
>>> bt.left_border_symmetry()
1[4[., .], 2[., 3[., .]]]
```

left_children_node_number(*direction='left'*)

Return the number of nodes which are left children in `self`.

Every node (except the root) is either the left child or the right child of its parent node. The total number of nodes is 1 plus the number of left-children nodes plus the number of right-children nodes.

INPUT:

- `direction` – either `'left'` (default) or `'right'` ; if set to `'right'`, instead count nodes that are right children

EXAMPLES:

```
sage: bt = BinaryTree([[None,[[],[]]],[None,[[],None]]])
sage: ascii_art(bt)
  __o__
 /     \
o       o
 \       \
  o       o
 / \     /
o   o   o
sage: bt.left_children_node_number('left')
3
sage: bt.left_children_node_number('right')
4

sage: all(5 == 1 + bt.left_children_node_number()
....:             + bt.left_children_node_number('right')
....:     for bt in BinaryTrees(5))
True
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[None,[[],[]]],[None,[[],None]]])
>>> ascii_art(bt)
  __o__
 /     \
o       o
 \       \
  o       o
 / \     /
o   o   o
```

(continues on next page)

(continued from previous page)

```
>>> bt.left_children_node_number('left')
3
>>> bt.left_children_node_number('right')
4

>>> all(Integer(5) == Integer(1) + bt.left_children_node_number()
...                 + bt.left_children_node_number('right')
...         for bt in BinaryTrees(Integer(5)))
True
```

left_right_symmetry()

Return the left-right symmetrized tree of `self`.

EXAMPLES:

```
sage: BinaryTree().left_right_symmetry()
.
sage: BinaryTree([]).left_right_symmetry()
[., .]
sage: BinaryTree([[],None]).left_right_symmetry()
[., [., .]]
sage: BinaryTree([[None, []],None]).left_right_symmetry()
[., [[., .], .]]
```

```
>>> from sage.all import *
>>> BinaryTree().left_right_symmetry()
.
>>> BinaryTree([]).left_right_symmetry()
[., .]
>>> BinaryTree([[],None]).left_right_symmetry()
[., [., .]]
>>> BinaryTree([[None, []],None]).left_right_symmetry()
[., [[., .], .]]
```

left_rotate()

Return the result of left rotation applied to the binary tree `self`.

Left rotation on binary trees is defined as follows: Let T be a binary tree such that the right child of the root of T is a node. Let A be the left child of the root of T, and let B and C be the left and right children of the right child of the root of T. (Keep in mind that nodes of trees are identified with the subtrees consisting of their descendants.) Then, the left rotation of T is the binary tree in which the right child of the root is C, whereas the left child of the root is a node whose left and right children are A and B. In pictures:

```
|    *                          *    |
|   / \                        / \   |
| A    *   -left-rotate->     *    C |
|     / \                    / \     |
|    B    C                 A    B   |
```

where asterisks signify a single node each (but A, B and C might be empty).

For example,

```
|   _o_                       o |
|  /   \                     /  |
| o     o  -left-rotate->   o   |
|      /                   / \  |
|     o                   o   o |
<BLANKLINE>
|       __o__                           o |
|      /     \                         /  |
|     o       o  -left-rotate->       o   |
|    / \                             /    |
|   o   o                           o     |
|  /     \                         / \    |
| o       o                       o   o   |
|                                /     \  |
|                               o       o |
```

Left rotation is the inverse operation to right rotation (*right_rotate()*).

See also:

right_rotate()

EXAMPLES:

```
sage: b = BinaryTree([[],[[],None]]); ascii_art([b])
[   _o_   ]
[  /   \  ]
[ o     o ]
[      /  ]
[     o   ]
sage: ascii_art([b.left_rotate()])
[     o ]
[    /  ]
[   o   ]
[  / \  ]
[ o   o ]
sage: b.left_rotate().right_rotate() == b
True
```

```
>>> from sage.all import *
>>> b = BinaryTree([[],[[],None]]); ascii_art([b])
[   _o_   ]
[  /   \  ]
[ o     o ]
[      /  ]
[     o   ]
>>> ascii_art([b.left_rotate()])
[     o ]
[    /  ]
[   o   ]
[  / \  ]
[ o   o ]
```

(continues on next page)

(continued from previous page)

```
>>> b.left_rotate().right_rotate() == b
True
```

make_leaf()

Modify `self` so that it becomes a leaf (i. e., an empty tree).

> **Note**
>
> `self` must be in a mutable state.

> **See also**
>
> *make_node*

EXAMPLES:

```
sage: t = BinaryTree([None, None])
sage: t.make_leaf()
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with t.clone() as t1:
....:     t1.make_leaf()
sage: t, t1
([., .], .)
```

```
>>> from sage.all import *
>>> t = BinaryTree([None, None])
>>> t.make_leaf()
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
>>> with t.clone() as t1:
...     t1.make_leaf()
>>> t, t1
([., .], .)
```

make_node(*child_list=[None, None]*)

Modify `self` so that it becomes a node with children `child_list`.

INPUT:

- `child_list` – a pair of binary trees (or objects convertible to)

> **Note**
>
> `self` must be in a mutable state.

See also

make_leaf

EXAMPLES:

```
sage: t = BinaryTree()
sage: t.make_node([None, None])
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with t.clone() as t1:
....:     t1.make_node([None, None])
sage: t, t1
(., [., .])
sage: with t.clone() as t:
....:     t.make_node([BinaryTree(), BinaryTree(), BinaryTree([])])
Traceback (most recent call last):
...
ValueError: the list must have length 2
sage: with t1.clone() as t2:
....:     t2.make_node([t1, t1])
sage: with t2.clone() as t3:
....:     t3.make_node([t1, t2])
sage: t1, t2, t3
([., .], [[., .], [., .]], [[., .], [[., .], [., .]]])
```

```
>>> from sage.all import *
>>> t = BinaryTree()
>>> t.make_node([None, None])
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
>>> with t.clone() as t1:
...     t1.make_node([None, None])
>>> t, t1
(., [., .])
>>> with t.clone() as t:
...     t.make_node([BinaryTree(), BinaryTree(), BinaryTree([])])
Traceback (most recent call last):
...
ValueError: the list must have length 2
>>> with t1.clone() as t2:
...     t2.make_node([t1, t1])
>>> with t2.clone() as t3:
...     t3.make_node([t1, t2])
>>> t1, t2, t3
([., .], [[., .], [., .]], [[., .], [[., .], [., .]]])
```

over(*bt*)

Return `self` over `bt`, where "over" is the `over` (/) operation.

If T and T' are two binary trees, then T over T' (written T/T') is defined as the tree obtained by grafting

T' on the rightmost leaf of T. More precisely, T/T' is defined by identifying the root of the T' with the rightmost leaf of T. See section 4.5 of [HNT2005].

If T is empty, then $T/T' = T'$.

The definition of this "over" operation goes back to Loday-Ronco [LR0102066] (Definition 2.2), but it is denoted by $\backslash$ and called the "under" operation there. In fact, trees in sage have their root at the top, contrary to the trees in [LR0102066] which are growing upwards. For this reason, the names of the over and under operations are swapped, in order to keep a graphical meaning. (Our notation follows that of section 4.5 of [HNT2005].)

See also

under()

EXAMPLES:

Showing only the nodes of a binary tree, here is an example for the over operation:

```
|   o       __o__       _o_         |
|  / \  /  /     \  =  /   \        |
| o   o   o       o   o     o       |
|          \     /           \      |
|           o   o           __o__   |
|                          /     \  |
|                         o       o |
|                          \     /  |
|                           o   o   |
```

A Sage example:

```
sage: b1 = BinaryTree([[],[[],[]]])
sage: b2 = BinaryTree([[None, []],[]])
sage: ascii_art((b1, b2, b1/b2))
(   _o_     ,   _o_  ,   _o_           )
(  /   \       /   \    /   \          )
( o     o     o     o  o     o_        )
(      / \     \            /  \       )
(     o   o     o          o    o      )
(                                \     )
(                               _o_    )
(                              /   \   )
(                             o     o  )
(                              \       )
(                               o      )
```

```
>>> from sage.all import *
>>> b1 = BinaryTree([[],[[],[]]])
>>> b2 = BinaryTree([[None, []],[]])
>>> ascii_art((b1, b2, b1/b2))
(   _o_     ,   _o_  ,   _o_           )
(  /   \       /   \    /   \          )
( o     o     o     o  o     o_        )
(      / \     \            /  \       )
```

(continues on next page)

(continued from previous page)

```
(      o   o    o          o    o       )
(                                \      )
(                               _o_     )
(                              /   \    )
(                             o      o  )
(                              \        )
(                               o       )
```

over_decomposition()

Return the unique maximal decomposition as an over product.

This means that the tree is cut along all edges of its rightmost path.

Beware that the factors are ordered starting from the root.

See also

comb(), *under_decomposition()*

EXAMPLES:

```
sage: g = BinaryTree([])
sage: r = g.over(g); r
[., [., .]]
sage: l = g.under(g); l
[[., .], .]
sage: r.over_decomposition()
[[., .], [., .]]
sage: l.over_decomposition() == [l]
True

sage: x = g.over(l).over(l).over(g).over(g)
sage: ascii_art(x)
 o
               _o_
 /              o     o
     /                o   o
                              o
sage: x.over_decomposition() == [g,l,l,g,g]
True
```

```
>>> from sage.all import *
>>> g = BinaryTree([])
>>> r = g.over(g); r
[., [., .]]
>>> l = g.under(g); l
[[., .], .]
>>> r.over_decomposition()
[[., .], [., .]]
>>> l.over_decomposition() == [l]
True
```

(continues on next page)

(continued from previous page)

```
>>> x = g.over(l).over(l).over(g).over(g)
>>> ascii_art(x)
 o
                _o_
 /               o     o
     /                 o   o
                               o
>>> x.over_decomposition() == [g,l,l,g,g]
True
```

prune()

Return the binary tree obtained by deleting each leaf of `self`.

The operation of pruning is the left inverse of attaching as many leaves as possible to each node of a binary tree. That is to say, for all binary trees `bt`, we have:

```
bt == bt.to_full().prune()
```

However, it is only a right inverse if and only if `bt` is a full binary tree:

```
bt == bt.prune().to_full()
```

OUTPUT: a binary tree

See also

to_full()

EXAMPLES:

```
sage: bt = BinaryTree([[[None, []], [[], []]], None])
sage: ascii_art(bt)
      o
     /
  __o__
 /     \
o       o
 \     / \
  o   o   o
sage: ascii_art(bt.prune())
    o
   /
  o
 / \
o   o
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[[None, []], [[], []]], None])
>>> ascii_art(bt)
      o
     /
  __o__
```

(continues on next page)

(continued from previous page)

```
 /      \
o        o
 \      / \
  o   o   o
>>> ascii_art(bt.prune())
    o
   /
  o
 / \
o   o
```

We check the relationship with `to_full()`:

```
sage: bt = BinaryTree([[[], [[None, []], []]], [[],[]]])
sage: bt == bt.to_full().prune()
True
sage: bt == bt.prune().to_full()
False

sage: bt = BinaryTree([[[], []], [[], [[[], []], []]]])
sage: bt.is_full()
True
sage: bt == bt.prune().to_full()
True
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[[], [[None, []], []]], [[],[]]])
>>> bt == bt.to_full().prune()
True
>>> bt == bt.prune().to_full()
False

>>> bt = BinaryTree([[[], []], [[], [[[], []], []]]])
>>> bt.is_full()
True
>>> bt == bt.prune().to_full()
True
```

Pruning the empty tree is again the empty tree:

```
sage: bt = BinaryTree(None)
sage: bt.prune()
.
```

```
>>> from sage.all import *
>>> bt = BinaryTree(None)
>>> bt.prune()
.
```

q_hook_length_fraction(*q=None*, *q_factor=False*)

Compute the `q`-hook length fraction of the binary tree `self`, with an additional "q-factor" if desired.

If T is a (plane) binary tree and q is a polynomial indeterminate over some ring, then the q-hook length fraction $h_q(T)$ of T is defined by

$$h_q(T) = \frac{[|T|]_q!}{\prod_{t \in T} [|T_t|]_q},$$

where the product ranges over all nodes t of T, where T_t denotes the subtree of T consisting of t and its all descendants, and where for every tree S, we denote by $|S|$ the number of nodes of S. While this definition only shows that $h_q(T)$ is a rational function in T, it is in fact easy to show that $h_q(T)$ is actually a polynomial in T, and thus makes sense when any element of a commutative ring is substituted for q. This can also be explicitly seen from the following recursive formula for $h_q(T)$:

$$h_q(T) = \binom{|T| - 1}{|T_1|}_q h_q(T_1) h_q(T_2),$$

where T is any nonempty binary tree, and T_1 and T_2 are the two child trees of the root of T, and where $\binom{a}{b}_q$ denotes a q-binomial coefficient.

A variation of the q-hook length fraction is the following "q-hook length fraction with q-factor":

$$f_q(T) = h_q(T) \cdot \prod_{t \in T} q^{|T_{\mathrm{right}(t)}|},$$

where for every node t, we denote by $\mathrm{right}(t)$ the right child of t. This $f_q(T)$ differs from $h_q(T)$ only in a multiplicative factor, which is a power of q.

When $q = 1$, both $f_q(T)$ and $h_q(T)$ equal the number of permutations whose binary search tree (see [HNT2005] for the definition) is T (after dropping the labels). For example, there are 20 permutations which give a binary tree of the following shape:

```
|     __o__   |
|    /     \  |
|   o       o |
|  / \     /  |
| o   o   o   |
```

by the binary search insertion algorithm, in accordance with the fact that this tree satisfies $f_1(T) = 20$.

When q is considered as a polynomial indeterminate, $f_q(T)$ is the generating function for all permutations whose binary search tree is T (after dropping the labels) with respect to the number of inversions (i. e., the Coxeter length) of the permutations.

Objects similar to $h_q(T)$ also make sense for general ordered forests (rather than just binary trees), see e. g. [BW1988], Theorem 9.1.

INPUT:

- `q` – a ring element which is to be substituted as q into the q-hook length fraction (by default, this is set to be the indeterminate q in the polynomial ring $\mathbf{Z}[q]$)
- `q_factor` – a Boolean (default: `False`); which determines whether to compute $h_q(T)$ or to compute $f_q(T)$ (namely, $h_q(T)$ is obtained when `q_factor == False`, and $f_q(T)$ is obtained when `q_factor == True`)

EXAMPLES:

Let us start with a simple example. Actually, let us start with the easiest possible example – the binary tree with only one vertex (which is a leaf):

```
sage: b = BinaryTree()
sage: b.q_hook_length_fraction()                                           #␣
↪needs sage.combinat
1
sage: b.q_hook_length_fraction(q_factor=True)                              #␣
↪needs sage.combinat
1
```

```
>>> from sage.all import *
>>> b = BinaryTree()
>>> b.q_hook_length_fraction()                                           #␣
↪needs sage.combinat
1
>>> b.q_hook_length_fraction(q_factor=True)                              #␣
↪needs sage.combinat
1
```

Nothing different for a tree with one node and two leaves:

```
sage: b = BinaryTree([]); b
[., .]
sage: b.q_hook_length_fraction()                                           #␣
↪needs sage.combinat
1
sage: b.q_hook_length_fraction(q_factor=True)                              #␣
↪needs sage.combinat
1
```

```
>>> from sage.all import *
>>> b = BinaryTree([]); b
[., .]
>>> b.q_hook_length_fraction()                                           #␣
↪needs sage.combinat
1
>>> b.q_hook_length_fraction(q_factor=True)                              #␣
↪needs sage.combinat
1
```

Let us get to a more interesting tree:

```
sage: # needs sage.combinat
sage: b = BinaryTree([[[],[]],[[],None]]); b
[[[., .], [., .]], [[., .], .]]
sage: b.q_hook_length_fraction()(q=1)
20
sage: b.q_hook_length_fraction()
q^7 + 2*q^6 + 3*q^5 + 4*q^4 + 4*q^3 + 3*q^2 + 2*q + 1
sage: b.q_hook_length_fraction(q_factor=True)
q^10 + 2*q^9 + 3*q^8 + 4*q^7 + 4*q^6 + 3*q^5 + 2*q^4 + q^3
sage: b.q_hook_length_fraction(q=2)
465
sage: b.q_hook_length_fraction(q=2, q_factor=True)
3720
```

(continues on next page)

(continued from previous page)

```
sage: q = PolynomialRing(ZZ, 'q').gen()
sage: b.q_hook_length_fraction(q=q**2)
q^14 + 2*q^12 + 3*q^10 + 4*q^8 + 4*q^6 + 3*q^4 + 2*q^2 + 1
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> b = BinaryTree([[[],[]],[[],None]]); b
[[[., .], [., .]], [[., .], .]]
>>> b.q_hook_length_fraction()(q=Integer(1))
20
>>> b.q_hook_length_fraction()
q^7 + 2*q^6 + 3*q^5 + 4*q^4 + 4*q^3 + 3*q^2 + 2*q + 1
>>> b.q_hook_length_fraction(q_factor=True)
q^10 + 2*q^9 + 3*q^8 + 4*q^7 + 4*q^6 + 3*q^5 + 2*q^4 + q^3
>>> b.q_hook_length_fraction(q=Integer(2))
465
>>> b.q_hook_length_fraction(q=Integer(2), q_factor=True)
3720
>>> q = PolynomialRing(ZZ, 'q').gen()
>>> b.q_hook_length_fraction(q=q**Integer(2))
q^14 + 2*q^12 + 3*q^10 + 4*q^8 + 4*q^6 + 3*q^4 + 2*q^2 + 1
```

Let us check the fact that $f_q(T)$ is the generating function for all permutations whose binary search tree is T (after dropping the labels) with respect to the number of inversions of the permutations:

```
sage: def q_hook_length_fraction_2(T):
....:     P = PolynomialRing(ZZ, 'q')
....:     q = P.gen()
....:     res = P.zero()
....:     for w in T.sylvester_class():
....:         res += q ** Permutation(w).length()
....:     return res
sage: def test_genfun(i):
....:     return all( q_hook_length_fraction_2(T)
....:                 == T.q_hook_length_fraction(q_factor=True)
....:                 for T in BinaryTrees(i) )
sage: test_genfun(4)                                                      #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> def q_hook_length_fraction_2(T):
...     P = PolynomialRing(ZZ, 'q')
...     q = P.gen()
...     res = P.zero()
...     for w in T.sylvester_class():
...         res += q ** Permutation(w).length()
...     return res
>>> def test_genfun(i):
...     return all( q_hook_length_fraction_2(T)
...                 == T.q_hook_length_fraction(q_factor=True)
...                 for T in BinaryTrees(i) )
```

(continues on next page)

(continued from previous page)

```
>>> test_genfun(Integer(4))                                    ␣
↪     # needs sage.combinat
True
```

right_rotate()

Return the result of right rotation applied to the binary tree `self`.

Right rotation on binary trees is defined as follows: Let T be a binary tree such that the left child of the root of T is a node. Let C be the right child of the root of T, and let A and B be the left and right children of the left child of the root of T. (Keep in mind that nodes of trees are identified with the subtrees consisting of their descendants.) Then, the right rotation of T is the binary tree in which the left child of the root is A, whereas the right child of the root is a node whose left and right children are B and C. In pictures:

```
|     *                      *     |
|    / \                    / \    |
|   *   C -right-rotate->  A   *   |
|  / \                        / \  |
| A   B                      B   C |
```

where asterisks signify a single node each (but A, B and C might be empty).

For example,

```
|     o                     _o_   |
|    /                     /   \  |
|   o    -right-rotate->  o     o |
|  / \                         /  |
| o   o                       o   |
<BLANKLINE>
|       __o__                         _o__      |
|      /     \                       /    \     |
|     o       o  -right-rotate->    o     _o_   |
|    / \                           /     /   \  |
|   o   o                         o     o     o |
|  /     \                               \      |
| o       o                               o     |
```

Right rotation is the inverse operation to left rotation (*left_rotate()*).

The right rotation operation introduced here is the one defined in Definition 2.1 of [CP2012].

See also

left_rotate()

EXAMPLES:

```
sage: b = BinaryTree([[[],[]], None]); ascii_art([b])
[     o ]
[    /  ]
[   o   ]
[  / \  ]
[ o   o ]
```

(continues on next page)

(continued from previous page)

```
sage: ascii_art([b.right_rotate()])
[   _o_   ]
[  /   \  ]
[ o     o ]
[      /  ]
[     o   ]
sage: b = BinaryTree([[[[],None],[None,[]]], []]); ascii_art([b])
[       __o__   ]
[      /     \  ]
[     o       o ]
[    / \        ]
[   o   o       ]
[  /     \      ]
[ o       o     ]
sage: ascii_art([b.right_rotate()])
[     _o__      ]
[    /    \     ]
[   o     _o_   ]
[  /     /   \  ]
[ o     o     o ]
[        \      ]
[         o     ]
```

```
>>> from sage.all import *
>>> b = BinaryTree([[[],[]], None]); ascii_art([b])
[     o ]
[    /  ]
[   o   ]
[  / \  ]
[ o   o ]
>>> ascii_art([b.right_rotate()])
[   _o_   ]
[  /   \  ]
[ o     o ]
[      /  ]
[     o   ]
>>> b = BinaryTree([[[[],None],[None,[]]], []]); ascii_art([b])
[       __o__   ]
[      /     \  ]
[     o       o ]
[    / \        ]
[   o   o       ]
[  /     \      ]
[ o       o     ]
>>> ascii_art([b.right_rotate()])
[     _o__      ]
[    /    \     ]
[   o     _o_   ]
[  /     /   \  ]
[ o     o     o ]
[        \      ]
[         o     ]
```

show(*with_leaves=False*)

Show the binary tree `show`, with or without leaves depending on the Boolean keyword variable `with_leaves`.

> **Warning**
>
> For a labelled binary tree, the labels shown in the picture are not (in general) the ones given by the labelling!
>
> Use `_latex_()`, `view`, `_ascii_art_()` or `pretty_print` for more faithful representations of the data of the tree.

single_edge_cut_shapes()

Return the list of possible single-edge cut shapes for the binary tree.

This is used in `sage.combinat.interval_posets.TamariIntervalPoset.is_new()`.

OUTPUT:

a list of triples (m, i, n) of integers

This is a list running over all inner edges (i.e., edges joining two non-leaf vertices) of the binary tree. The removal of each inner edge defines two binary trees (connected components), the root-tree and the sub-tree. Thus, to every inner edge, we can assign three positive integers: m is the node number of the root-tree R, and n is the node number of the sub-tree S. The integer i is the index of the leaf of R on which S is grafted to obtain the original tree. The leaves of R are numbered starting from 1 (from left to right), hence $1 \leq i \leq m + 1$.

In fact, each of m and n determines the other, as the total node number of R and S is the node number of `self`.

EXAMPLES:

```
sage: BT = BinaryTrees(3)
sage: [t.single_edge_cut_shapes() for t in BT]
[[(2, 3, 1), (1, 2, 2)],
[(2, 2, 1), (1, 2, 2)],
[(2, 1, 1), (2, 3, 1)],
[(2, 2, 1), (1, 1, 2)],
[(2, 1, 1), (1, 1, 2)]]

sage: BT = BinaryTrees(2)
sage: [t.single_edge_cut_shapes() for t in BT]
[[(1, 2, 1)], [(1, 1, 1)]]

sage: BT = BinaryTrees(1)
sage: [t.single_edge_cut_shapes() for t in BT]
[[]]
```

```
>>> from sage.all import *
>>> BT = BinaryTrees(Integer(3))
>>> [t.single_edge_cut_shapes() for t in BT]
[[(2, 3, 1), (1, 2, 2)],
[(2, 2, 1), (1, 2, 2)],
[(2, 1, 1), (2, 3, 1)],
[(2, 2, 1), (1, 1, 2)],
[(2, 1, 1), (1, 1, 2)]]
```

(continues on next page)

(continued from previous page)

```
>>> BT = BinaryTrees(Integer(2))
>>> [t.single_edge_cut_shapes() for t in BT]
[[(1, 2, 1)], [(1, 1, 1)]]

>>> BT = BinaryTrees(Integer(1))
>>> [t.single_edge_cut_shapes() for t in BT]
[[]]
```

sylvester_class(*left_to_right=False*)

Iterate over the sylvester class corresponding to the binary tree `self`.

The sylvester class of a tree T is the set of permutations σ whose right-to-left binary search tree (a notion defined in [HNT2005], Definition 7) is T after forgetting the labels. This is an equivalence class of the sylvester congruence (the congruence on words which holds two words $uacvbw$ and $ucavbw$ congruent whenever a, b, c are letters satisfying $a \leq b < c$, and extends by transitivity) on the symmetric group.

For example the following tree's sylvester class consists of the permutations $(1, 3, 2)$ and $(3, 1, 2)$:

```
[    o    ]
[   / \   ]
[  o   o  ]
```

(only the nodes are drawn here).

The right-to-left binary search tree of a word is constructed by an RSK-like insertion algorithm which proceeds as follows: Start with an empty labelled binary tree, and read the word from right to left. Each time a letter is read from the word, insert this letter in the existing tree using binary search tree insertion (*binary_search_insert()*). This is what the *binary_search_tree()* method computes if it is given the keyword `left_to_right=False`.

Here are two more descriptions of the sylvester class of a binary search tree:

- The sylvester class of a binary search tree T is the set of all linear extensions of the poset corresponding to T (that is, of the poset whose Hasse diagram is T, with the root on top), provided that the nodes of T are labelled with $1, 2, \ldots, n$ in a binary-search-tree way (i.e., every left descendant of a node has a label smaller than that of the node, and every right descendant of a node has a label higher than that of the node).
- The sylvester class of a binary search tree T (with vertex labels $1, 2, \ldots, n$) is the interval $[u, v]$ in the right permutohedron order (*permutohedron_lequal()*), where u is the 312-avoiding permutation corresponding to T (*to_312_avoiding_permutation()*), and where v is the 132-avoiding permutation corresponding to T (*to_132_avoiding_permutation()*).

If the optional keyword variable `left_to_right` is set to `True`, then the *left* sylvester class of `self` is returned instead. This is the set of permutations σ whose left-to-right binary search tree (that is, the result of the *binary_search_tree()* with `left_to_right` set to `True`) is `self`. It is an equivalence class of the left sylvester congruence.

> **Warning**
>
> This method yields the elements of the sylvester class as raw lists, not as permutations!

EXAMPLES:

Verifying the claim that the right-to-left binary search trees of the permutations in the sylvester class of a tree t all equal t:

```
sage: def test_bst_of_sc(n, left_to_right):
....:     for t in BinaryTrees(n):
....:         for p in t.sylvester_class(left_to_right=left_to_right):
....:             p_per = Permutation(p)
....:             tree = p_per.binary_search_tree(left_to_right=left_to_right)
....:             if not BinaryTree(tree) == t:
....:                 return False
....:     return True
sage: test_bst_of_sc(4, False)                                                  #␣
↪needs sage.combinat
True
sage: test_bst_of_sc(5, False)      # long time                                 #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> def test_bst_of_sc(n, left_to_right):
...     for t in BinaryTrees(n):
...         for p in t.sylvester_class(left_to_right=left_to_right):
...             p_per = Permutation(p)
...             tree = p_per.binary_search_tree(left_to_right=left_to_right)
...             if not BinaryTree(tree) == t:
...                 return False
...     return True
>>> test_bst_of_sc(Integer(4), False)                                           ␣
↪      # needs sage.combinat
True
>>> test_bst_of_sc(Integer(5), False)      # long time                          ␣
↪      # needs sage.combinat
True
```

The same with the left-to-right version of binary search:

```
sage: test_bst_of_sc(4, True)                                                   #␣
↪needs sage.combinat
True
sage: test_bst_of_sc(5, True)       # long time                                 #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> test_bst_of_sc(Integer(4), True)                                            ␣
↪      # needs sage.combinat
True
>>> test_bst_of_sc(Integer(5), True)       # long time                          ␣
↪      # needs sage.combinat
True
```

Checking that the sylvester class is the set of linear extensions of the poset of the tree:

```
sage: all(sorted(t.canonical_labelling().sylvester_class())                    #␣
↪needs sage.combinat sage.modules
....:         == sorted(list(v)
....:                   for v in t.canonical_labelling().to_poset().linear_
↪extensions())
....:     for t in BinaryTrees(4))
True
```

```
>>> from sage.all import *
>>> all(sorted(t.canonical_labelling().sylvester_class())                    #␣
↪needs sage.combinat sage.modules
...         == sorted(list(v)
...                   for v in t.canonical_labelling().to_poset().linear_
↪extensions())
...     for t in BinaryTrees(Integer(4)))
True
```

tamari_greater()

The list of all trees greater or equal to `self` in the Tamari order.

This is the order filter of the Tamari order generated by `self`.

See *tamari_lequal()* for the definition of the Tamari poset.

See also

tamari_smaller()

EXAMPLES:

For example, the tree:

```
|     __o__   |
|    /     \  |
|   o       o |
|  / \     /  |
| o   o   o   |
```

has these trees greater or equal to it:

```
|o          , o         , o         , o         ,  o        ,   o       ,|
| \            \           \           \            \            \       |
|  o            o           o           o           _o_        __o__     |
|   \            \           \           \         /   \      /     \    |
|    o            o           o         _o_       o     o    o       o   |
|     \            \         / \       /   \       \     \    \     /    |
|      o            o       o   o     o     o       o     o    o   o     |
|       \            \           \         /                             |
|        o            o           o       o                              |
|         \          /                                                   |
|          o        o                                                    |
<BLANKLINE>
|   o         ,   o       ,   _o_       ,   _o__      ,   __o__     ,   ___o___   ,|
```

(continues on next page)

(continued from previous page)

```
|  / \          / \        /   \        /    \       /     \      /       \  |
| o   o        o   o      o     o      o     _o_    o       o    o         o |
|      \            \          / \          /   \    \       \    \       /  |
|       o            o        o   o        o     o    o       o    o     o   |
|        \            \            \            /      \            \        |
|         o            o            o          o        o            o       |
|          \          /                                                      |
|           o        o                                                       |
<BLANKLINE>
|     _o_    ,     __o__  |
|    /   \        /     \ |
|   o     o      o       o|
|  / \     \    / \     / |
| o   o     o  o   o   o  |
```

tamari_interval(*other*)

Return the Tamari interval between `self` and `other` as a *TamariIntervalPoset*.

A "Tamari interval" is an interval in the Tamari poset. See *tamari_lequal()* for the definition of the Tamari poset.

INPUT:

- `other` – a binary tree greater or equal to `self` in the Tamari order

EXAMPLES:

```
sage: bt = BinaryTree([[None, [[], None]], None])
sage: ip = bt.tamari_interval(BinaryTree([None, [[None, []], None]])); ip
The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (3, 1),␣
↪(2, 1)]
sage: ip.lower_binary_tree()
[[., [[., .], .]], .]
sage: ip.upper_binary_tree()
[., [[., [., .]], .]]
sage: ip.interval_cardinality()
4
sage: ip.number_of_tamari_inversions()
2
sage: list(ip.binary_trees())
[[., [[., [., .]], .]],
 [[., [., [., .]]], .],
 [., [[[., .], .], .]],
 [[., [[., .], .]], .]]
sage: bt.tamari_interval(BinaryTree([[None,[]],[]]))
Traceback (most recent call last):
...
ValueError: the two binary trees are not comparable on the Tamari lattice
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[None, [[], None]], None])
>>> ip = bt.tamari_interval(BinaryTree([None, [[None, []], None]])); ip
The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (3, 1),␣
↪(2, 1)]
```

(continues on next page)

(continued from previous page)

```
>>> ip.lower_binary_tree()
[[., [[., .], .]], .]
>>> ip.upper_binary_tree()
[., [[., [., .]], .]]
>>> ip.interval_cardinality()
4
>>> ip.number_of_tamari_inversions()
2
>>> list(ip.binary_trees())
[[., [[., [., .]], .]],
 [[., [., [., .]]], .],
 [., [[[., .], .], .]],
 [[., [[., .], .]], .]]
>>> bt.tamari_interval(BinaryTree([[None,[]],[]]))
Traceback (most recent call last):
...
ValueError: the two binary trees are not comparable on the Tamari lattice
```

tamari_join(*other*)

Return the join of the binary trees `self` and `other` (of equal size) in the n-th Tamari poset (where n is the size of these trees).

The n-th Tamari poset (defined in *tamari_lequal()*) is known to be a lattice, and the map from the n-th symmetric group S_n to the n-th Tamari poset defined by sending every permutation $p \in S_n$ to the binary search tree of p (more precisely, to `p.binary_search_tree_shape()`) is a lattice homomorphism. (See Theorem 6.2 in [Rea2004].)

See also:

tamari_lequal(), *tamari_meet()*.

AUTHORS:

Viviane Pons and Darij Grinberg, 18 June 2014; Frédéric Chapoton, 9 January 2018.

EXAMPLES:

```
sage: a = BinaryTree([None, [None, []]])
sage: b = BinaryTree([None, [[], None]])
sage: c = BinaryTree([[None, []], None])
sage: d = BinaryTree([[[], None], None])
sage: e = BinaryTree([[], []])
sage: a.tamari_join(c) == a
True
sage: b.tamari_join(c) == b
True
sage: c.tamari_join(e) == a
True
sage: d.tamari_join(e) == e
True
sage: e.tamari_join(b) == a
True
```

(continues on next page)

(continued from previous page)

```
sage: e.tamari_join(a) == a
True
```

```
>>> from sage.all import *
>>> a = BinaryTree([None, [None, []]])
>>> b = BinaryTree([None, [[], None]])
>>> c = BinaryTree([[None, []], None])
>>> d = BinaryTree([[[], None], None])
>>> e = BinaryTree([[], []])
>>> a.tamari_join(c) == a
True
>>> b.tamari_join(c) == b
True
>>> c.tamari_join(e) == a
True
>>> d.tamari_join(e) == e
True
>>> e.tamari_join(b) == a
True
>>> e.tamari_join(a) == a
True
```

```
sage: b1 = BinaryTree([None, [[[], None], None]])
sage: b2 = BinaryTree([[[], None], []])
sage: b1.tamari_join(b2)
[., [[., .], [., .]]]
sage: b3 = BinaryTree([[], [[], None]])
sage: b1.tamari_join(b3)
[., [., [[., .], .]]]
sage: b2.tamari_join(b3)
[[., .], [., [., .]]]
```

```
>>> from sage.all import *
>>> b1 = BinaryTree([None, [[[], None], None]])
>>> b2 = BinaryTree([[[], None], []])
>>> b1.tamari_join(b2)
[., [[., .], [., .]]]
>>> b3 = BinaryTree([[], [[], None]])
>>> b1.tamari_join(b3)
[., [., [[., .], .]]]
>>> b2.tamari_join(b3)
[[., .], [., [., .]]]
```

The universal property of the meet operation is satisfied:

```
sage: def test_uni_join(p, q):
....:     j = p.tamari_join(q)
....:     if not p.tamari_lequal(j):
....:         return False
....:     if not q.tamari_lequal(j):
....:         return False
```

(continues on next page)

(continued from previous page)

```
....:     for r in p.tamari_greater():
....:         if q.tamari_lequal(r) and not j.tamari_lequal(r):
....:             return False
....:     return True
sage: all( test_uni_join(p, q) for p in BinaryTrees(3) for q in␣
↪BinaryTrees(3) )
True
sage: p = BinaryTrees(6).random_element()                              #␣
↪needs sage.combinat
sage: q = BinaryTrees(6).random_element()                              #␣
↪needs sage.combinat
sage: test_uni_join(p, q)                                              #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> def test_uni_join(p, q):
...     j = p.tamari_join(q)
...     if not p.tamari_lequal(j):
...         return False
...     if not q.tamari_lequal(j):
...         return False
...     for r in p.tamari_greater():
...         if q.tamari_lequal(r) and not j.tamari_lequal(r):
...             return False
...     return True
>>> all( test_uni_join(p, q) for p in BinaryTrees(Integer(3)) for q in␣
↪BinaryTrees(Integer(3)) )
True
>>> p = BinaryTrees(Integer(6)).random_element()                       ␣
↪     # needs sage.combinat
>>> q = BinaryTrees(Integer(6)).random_element()                       ␣
↪     # needs sage.combinat
>>> test_uni_join(p, q)                                              #␣
↪needs sage.combinat
True
```

Border cases:

```
sage: b = BinaryTree(None)
sage: b.tamari_join(b)
.
sage: b = BinaryTree([])
sage: b.tamari_join(b)
[., .]
```

```
>>> from sage.all import *
>>> b = BinaryTree(None)
>>> b.tamari_join(b)
.
>>> b = BinaryTree([])
>>> b.tamari_join(b)
```

(continues on next page)

(continued from previous page)

```
[., .]
```

tamari_lequal(*t2*)

Return `True` if `self` is less or equal to another binary tree `t2` (of the same size as `self`) in the Tamari order.

The Tamari order on binary trees of size n is the partial order on the set of all binary trees of size n generated by the following requirement: If a binary tree T' is obtained by right rotation (see `right_rotate()`) from a binary tree T, then $T < T'$. This not only is a well-defined partial order, but actually is a lattice structure on the set of binary trees of size n, and is a quotient of the weak order on the n-th symmetric group (also known as the right permutohedron order, see `permutohedron_lequal()`). See [CP2012]. The set of binary trees of size n equipped with the Tamari order is called the n-th Tamari poset.

The Tamari order can equivalently be defined as follows:

If T and S are two binary trees of size n, then the following four statements are equivalent:

- We have $T \leq S$ in the Tamari order.
- There exist elements t and s of the Sylvester classes (`sylvester_class()`) of T and S, respectively, such that $t \leq s$ in the weak order on the symmetric group.
- The 132-avoiding permutation corresponding to T (see `to_132_avoiding_permutation()`) is $\leq$ to the 132-avoiding permutation corresponding to S in the weak order on the symmetric group.
- The 312-avoiding permutation corresponding to T (see `to_312_avoiding_permutation()`) is $\leq$ to the 312-avoiding permutation corresponding to S in the weak order on the symmetric group.

See also

`tamari_smaller()`, `tamari_greater()`, `tamari_pred()`, `tamari_succ()`, `tamari_interval()`

EXAMPLES:

This tree:

```
|       o      |
|      / \     |
|     o   o    |
|    /         |
|   o          |
|  / \         |
| o   o        |
```

is Tamari-$\leq$ to the following tree:

```
|     _o_      |
|    /   \     |
|   o     o    |
|  / \     \   |
| o   o     o  |
```

Checking this:

```
sage: b = BinaryTree([[[[], []], None], []])
sage: c = BinaryTree([[[],[]],[None,[]]])
sage: b.tamari_lequal(c)
True
```

```
>>> from sage.all import *
>>> b = BinaryTree([[[[], []], None], []])
>>> c = BinaryTree([[[],[]],[None,[]]])
>>> b.tamari_lequal(c)
True
```

tamari_meet (*other*, *side='right'*)

Return the meet of the binary trees `self` and `other` (of equal size) in the n-th Tamari poset (where n is the size of these trees).

The n-th Tamari poset (defined in *tamari_lequal()*) is known to be a lattice, and the map from the n-th symmetric group S_n to the n-th Tamari poset defined by sending every permutation $p \in S_n$ to the binary search tree of p (more precisely, to `p.binary_search_tree_shape()`) is a lattice homomorphism. (See Theorem 6.2 in [Rea2004].)

See also

tamari_lequal(), *tamari_join()*.

AUTHORS:

Viviane Pons and Darij Grinberg, 18 June 2014.

EXAMPLES:

```
sage: a = BinaryTree([None, [None, []]])
sage: b = BinaryTree([None, [[], None]])
sage: c = BinaryTree([[None, []], None])
sage: d = BinaryTree([[[], None], None])
sage: e = BinaryTree([[], []])
sage: a.tamari_meet(c) == c
True
sage: b.tamari_meet(c) == c
True
sage: c.tamari_meet(e) == d
True
sage: d.tamari_meet(e) == d
True
sage: e.tamari_meet(b) == d
True
sage: e.tamari_meet(a) == e
True
```

```
>>> from sage.all import *
>>> a = BinaryTree([None, [None, []]])
>>> b = BinaryTree([None, [[], None]])
>>> c = BinaryTree([[None, []], None])
>>> d = BinaryTree([[[], None], None])
```

(continues on next page)

(continued from previous page)

```
>>> e = BinaryTree([[], []])
>>> a.tamari_meet(c) == c
True
>>> b.tamari_meet(c) == c
True
>>> c.tamari_meet(e) == d
True
>>> d.tamari_meet(e) == d
True
>>> e.tamari_meet(b) == d
True
>>> e.tamari_meet(a) == e
True
```

```
sage: b1 = BinaryTree([None, [[[], None], None]])
sage: b2 = BinaryTree([[[], None], []])
sage: b1.tamari_meet(b2)
[[[[., .], .], .], .]
sage: b3 = BinaryTree([[], [[], None]])
sage: b1.tamari_meet(b3)
[[[[., .], .], .], .]
sage: b2.tamari_meet(b3)
[[[[., .], .], .], .]
```

```
>>> from sage.all import *
>>> b1 = BinaryTree([None, [[[], None], None]])
>>> b2 = BinaryTree([[[], None], []])
>>> b1.tamari_meet(b2)
[[[[., .], .], .], .]
>>> b3 = BinaryTree([[], [[], None]])
>>> b1.tamari_meet(b3)
[[[[., .], .], .], .]
>>> b2.tamari_meet(b3)
[[[[., .], .], .], .]
```

The universal property of the meet operation is satisfied:

```
sage: def test_uni_meet(p, q):
....:     m = p.tamari_meet(q)
....:     if not m.tamari_lequal(p):
....:         return False
....:     if not m.tamari_lequal(q):
....:         return False
....:     for r in p.tamari_smaller():
....:         if r.tamari_lequal(q) and not r.tamari_lequal(m):
....:             return False
....:     return True
sage: all( test_uni_meet(p, q) for p in BinaryTrees(3) for q in␣
↪BinaryTrees(3) )
True
sage: p = BinaryTrees(6).random_element()                                    #␣
↪needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
sage: q = BinaryTrees(6).random_element()                                      #␣
↪needs sage.combinat
sage: test_uni_meet(p, q)                                                      #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> def test_uni_meet(p, q):
...     m = p.tamari_meet(q)
...     if not m.tamari_lequal(p):
...         return False
...     if not m.tamari_lequal(q):
...         return False
...     for r in p.tamari_smaller():
...         if r.tamari_lequal(q) and not r.tamari_lequal(m):
...             return False
...     return True
>>> all( test_uni_meet(p, q) for p in BinaryTrees(Integer(3)) for q in␣
↪BinaryTrees(Integer(3)) )
True
>>> p = BinaryTrees(Integer(6)).random_element()                               ␣
↪     # needs sage.combinat
>>> q = BinaryTrees(Integer(6)).random_element()                               ␣
↪     # needs sage.combinat
>>> test_uni_meet(p, q)                                                      #␣
↪needs sage.combinat
True
```

Border cases:

```
sage: b = BinaryTree(None)
sage: b.tamari_meet(b)
.
sage: b = BinaryTree([])
sage: b.tamari_meet(b)
[., .]
```

```
>>> from sage.all import *
>>> b = BinaryTree(None)
>>> b.tamari_meet(b)
.
>>> b = BinaryTree([])
>>> b.tamari_meet(b)
[., .]
```

tamari_pred()

Compute the list of predecessors of `self` in the Tamari poset.

This list is computed by performing all left rotates possible on its nodes.

See *tamari_lequal()* for the definition of the Tamari poset.

EXAMPLES:

For this tree:

```
|     __o__   |
|    /     \  |
|   o       o |
|  / \     /  |
| o   o   o   |
```

the list is:

```
|        o ,       _o_   |
|       /         /   \  |
|     _o_        o     o |
|    /   \      /     /  |
|   o     o    o     o   |
|  / \        /          |
| o   o      o           |
```

tamari_smaller()

The list of all trees smaller or equal to `self` in the Tamari order.

This is the order ideal of the Tamari order generated by `self`.

See *tamari_lequal()* for the definition of the Tamari poset.

> **See also**
>
> *tamari_greater()*

EXAMPLES:

The tree:

```
|     __o__   |
|    /     \  |
|   o       o |
|  / \     /  |
| o   o   o   |
```

has these trees smaller or equal to it:

```
|    __o__  ,       _o_  ,        o ,         o,         o,           o |
|   /     \        /   \         /           /          /            /  |
|  o       o      o     o      _o_          o          o            o   |
| / \     /      /     /      /   \        / \        /            /    |
|o   o   o      o     o      o     o      o   o      o            o     |
|              /            / \          /          /            /      |
|             o            o   o        o          o            o       |
|                                      /          / \          /        |
|                                     o          o   o        o         |
|                                                            /          |
|                                                           o           |
```

tamari_sorting_tuple(*reverse=False*)

Return the Tamari sorting tuple of `self` and the size of `self`.

This is a pair (w, n), where n is the number of nodes of `self`, and w is an n-tuple whose i-th entry is the number of all nodes among the descendants of the right child of the i-th node of `self`. Here, the nodes of `self` are numbered from left to right.

INPUT:

- `reverse` – boolean (default: `False`); if `True`, return instead the result for the left-right symmetric of the binary tree

OUTPUT:

a pair (w, n), where w is a tuple of integers, and n the size

Two binary trees of the same size are comparable in the Tamari order if and only if the associated tuples w are componentwise comparable. (This is essentially the Theorem in [HT1972].) This is used in *tamari_lequal()*.

EXAMPLES:

```
sage: [t.tamari_sorting_tuple() for t in BinaryTrees(3)]
[((2, 1, 0), 3),
 ((2, 0, 0), 3),
 ((0, 1, 0), 3),
 ((1, 0, 0), 3),
 ((0, 0, 0), 3)]

sage: t = BinaryTrees(10).random_element()                                    #␣
↪needs sage.combinat
sage: u = t.left_right_symmetry()                                              #␣
↪needs sage.combinat
sage: t.tamari_sorting_tuple(True) == u.tamari_sorting_tuple()                #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> [t.tamari_sorting_tuple() for t in BinaryTrees(Integer(3))]
[((2, 1, 0), 3),
 ((2, 0, 0), 3),
 ((0, 1, 0), 3),
 ((1, 0, 0), 3),
 ((0, 0, 0), 3)]

>>> t = BinaryTrees(Integer(10)).random_element()                             ␣
↪     # needs sage.combinat
>>> u = t.left_right_symmetry()                                              #␣
↪needs sage.combinat
>>> t.tamari_sorting_tuple(True) == u.tamari_sorting_tuple()                #␣
↪needs sage.combinat
True
```

REFERENCES:

- [HT1972]

tamari_succ()

Compute the list of successors of `self` in the Tamari poset.

This is the list of all trees obtained by a right rotate of one of its nodes.

See `tamari_lequal()` for the definition of the Tamari poset.

EXAMPLES:

The list of successors of:

```
|     __o__    |
|    /     \   |
|   o       o  |
|  / \     /   |
| o   o   o    |
```

is:

```
|   _o__     ,   ___o___  ,     _o_     |
|  /    \       /       \      /   \    |
| o     _o_    o         o    o     o   |
|      /   \    \       /    / \     \  |
|     o     o    o     o    o   o     o |
|          /      \                     |
|         o        o                    |
```

to_132_avoiding_permutation()

Return a 132-avoiding permutation corresponding to the binary tree.

The linear extensions of a binary tree form an interval of the weak order called the sylvester class of the tree. This permutation is the maximal element of this sylvester class.

EXAMPLES:

```
sage: bt = BinaryTree([[],[]])
sage: bt.to_132_avoiding_permutation()
[3, 1, 2]
sage: bt = BinaryTree([[[], [[], None]], [[], []]])
sage: bt.to_132_avoiding_permutation()
[8, 6, 7, 3, 4, 1, 2, 5]
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[],[]])
>>> bt.to_132_avoiding_permutation()
[3, 1, 2]
>>> bt = BinaryTree([[[], [[], None]], [[], []]])
>>> bt.to_132_avoiding_permutation()
[8, 6, 7, 3, 4, 1, 2, 5]
```

to_312_avoiding_permutation()

Return a 312-avoiding permutation corresponding to the binary tree.

The linear extensions of a binary tree form an interval of the weak order called the sylvester class of the tree. This permutation is the minimal element of this sylvester class.

EXAMPLES:

```
sage: bt = BinaryTree([[],[]])
sage: bt.to_312_avoiding_permutation()
[1, 3, 2]
```

(continues on next page)

(continued from previous page)

```
sage: bt = BinaryTree([[[], [[], None]], [[], []]])
sage: bt.to_312_avoiding_permutation()
[1, 3, 4, 2, 6, 8, 7, 5]
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[],[]])
>>> bt.to_312_avoiding_permutation()
[1, 3, 2]
>>> bt = BinaryTree([[[], [[], None]], [[], []]])
>>> bt.to_312_avoiding_permutation()
[1, 3, 4, 2, 6, 8, 7, 5]
```

to_dyck_word(*usemap='1L0R'*)

Return the Dyck word associated with `self` using the given map.

INPUT:

- `usemap` – string; either `'1L0R'`, `'1R0L'`, `'L1R0'`, `'R1L0'`

The bijection is defined recursively as follows:

- a leaf is associated to the empty Dyck Word
- a tree with children l, r is associated with the Dyck word described by `usemap` where L and R are respectively the Dyck words associated with the trees l and r.

EXAMPLES:

```
sage: # needs sage.combinat
sage: BinaryTree().to_dyck_word()
[]
sage: BinaryTree([]).to_dyck_word()
[1, 0]
sage: BinaryTree([[[], [[], None]], [[], []]]).to_dyck_word()
[1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word()
[1, 1, 0, 1, 0, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word("1R0L")
[1, 0, 1, 1, 0, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word("L1R0")
[1, 1, 0, 0, 1, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word("R1L0")
[1, 1, 0, 1, 0, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word("R10L")
Traceback (most recent call last):
...
ValueError: R10L is not a correct map
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> BinaryTree().to_dyck_word()
[]
>>> BinaryTree([]).to_dyck_word()
[1, 0]
```

(continues on next page)

(continued from previous page)

```
>>> BinaryTree([[[], [[], None]], [[], []]]).to_dyck_word()
[1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0]
>>> BinaryTree([[None,[]],None]).to_dyck_word()
[1, 1, 0, 1, 0, 0]
>>> BinaryTree([[None,[]],None]).to_dyck_word("1R0L")
[1, 0, 1, 1, 0, 0]
>>> BinaryTree([[None,[]],None]).to_dyck_word("L1R0")
[1, 1, 0, 0, 1, 0]
>>> BinaryTree([[None,[]],None]).to_dyck_word("R1L0")
[1, 1, 0, 1, 0, 0]
>>> BinaryTree([[None,[]],None]).to_dyck_word("R10L")
Traceback (most recent call last):
...
ValueError: R10L is not a correct map
```

to_dyck_word_tamari()

Return the Dyck word associated with `self` in consistency with the Tamari order on Dyck words and binary trees.

The bijection is defined recursively as follows:

- a leaf is associated with an empty Dyck word;
- a tree with children l, r is associated with the Dyck word $T(l)1T(r)0$.

EXAMPLES:

```
sage: # needs sage.combinat
sage: BinaryTree().to_dyck_word_tamari()
[]
sage: BinaryTree([]).to_dyck_word_tamari()
[1, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word_tamari()
[1, 1, 0, 0, 1, 0]
sage: BinaryTree([[[], [[], None]], [[], []]]).to_dyck_word_tamari()
[1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> BinaryTree().to_dyck_word_tamari()
[]
>>> BinaryTree([]).to_dyck_word_tamari()
[1, 0]
>>> BinaryTree([[None,[]],None]).to_dyck_word_tamari()
[1, 1, 0, 0, 1, 0]
>>> BinaryTree([[[], [[], None]], [[], []]]).to_dyck_word_tamari()
[1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
```

to_full()

Return the full binary tree constructed from `self`.

Let T be a binary tree with n nodes. We construct a full binary tree F from T by attaching a leaf to each node of T which does not have 2 children. The resulting tree will have $2n + 1$ nodes.

OUTPUT: a full binary tree. See *is_full()* for the definition of full

See also

`prune()`

EXAMPLES:

```
sage: bt = BinaryTree([[None,[]],None])
sage: bt.to_full().is_full()
True
sage: ascii_art(bt)
  o
 /
o
 \
  o
sage: ascii_art(bt.to_full())
     __o___
    /      \
  _o_       o
 /   \
o     o
     / \
    o   o

sage: bt = BinaryTree([[],[]])
sage: ascii_art(bt)
  o
 / \
o   o
sage: ascii_art(bt.to_full())
    __o__
   /     \
  o       o
 / \     / \
o   o   o   o

sage: BinaryTree(None).to_full()
[., .]
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[None,[]],None])
>>> bt.to_full().is_full()
True
>>> ascii_art(bt)
  o
 /
o
 \
  o
>>> ascii_art(bt.to_full())
     __o___
    /      \
```

(continues on next page)

(continued from previous page)

```
  _o_       o
 /   \
o     o
     / \
    o   o

>>> bt = BinaryTree([[],[]])
>>> ascii_art(bt)
  o
 / \
o   o
>>> ascii_art(bt.to_full())
    __o__
   /     \
  o       o
 / \     / \
o   o   o   o

>>> BinaryTree(None).to_full()
[., .]
```

to_ordered_tree_left_branch()

Return an ordered tree of size $n+1$ by the following recursive rule:

- if x is the left child of y, x becomes the left brother of y
- if x is the right child of y, x becomes the last child of y

EXAMPLES:

```
sage: bt = BinaryTree([[],[]])
sage: bt.to_ordered_tree_left_branch()
[[], [[]]]
sage: bt = BinaryTree([[[], [], None]], [[], []]])
sage: bt.to_ordered_tree_left_branch()
[[], [[], []], [[], [[]]]]
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[],[]])
>>> bt.to_ordered_tree_left_branch()
[[], [[]]]
>>> bt = BinaryTree([[[], [], None]], [[], []]])
>>> bt.to_ordered_tree_left_branch()
[[], [[], []], [[], [[]]]]
```

to_ordered_tree_right_branch()

Return an ordered tree of size $n+1$ by the following recursive rule:

- if x is the right child of y, x becomes the right brother of y
- if x is the left child of y, x becomes the first child of y

EXAMPLES:

```
sage: bt = BinaryTree([[],[]])
sage: bt.to_ordered_tree_right_branch()
[[[]], []]
sage: bt = BinaryTree([[[], [[], None]], [[], []]])
sage: bt.to_ordered_tree_right_branch()
[[[[]], [[]]], [[]], []]
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[],[]])
>>> bt.to_ordered_tree_right_branch()
[[[]], []]
>>> bt = BinaryTree([[[], [[], None]], [[], []]])
>>> bt.to_ordered_tree_right_branch()
[[[[]], [[]]], [[]], []]
```

to_poset(*with_leaves=False*, *root_to_leaf=False*)

Return the poset obtained by interpreting the tree as a Hasse diagram.

The default orientation is from leaves to root but you can pass `root_to_leaf=True` to obtain the inverse orientation.

Leaves are ignored by default, but one can set `with_leaves` to `True` to obtain the poset of the complete tree.

INPUT:

- `with_leaves` – boolean (default: `False`); whether the resulting poset will be formed from the leaves and the nodes of `self` (if `True`), or only from the nodes of `self` (if `False`)
- `root_to_leaf` – boolean (default: `False`); whether the poset orientation should be from root to leaves (if `True`) or from leaves to root (if `False`)

EXAMPLES:

```
sage: bt = BinaryTree([])
sage: bt.to_poset()
Finite poset containing 1 elements
sage: bt.to_poset(with_leaves=True)
Finite poset containing 3 elements
sage: P1 = bt.to_poset(with_leaves=True)
sage: len(P1.maximal_elements())
1
sage: len(P1.minimal_elements())
2
sage: bt = BinaryTree([])
sage: P2 = bt.to_poset(with_leaves=True,root_to_leaf=True)
sage: len(P2.maximal_elements())
2
sage: len(P2.minimal_elements())
1
```

```
>>> from sage.all import *
>>> bt = BinaryTree([])
>>> bt.to_poset()
Finite poset containing 1 elements
```

(continues on next page)

(continued from previous page)

```
>>> bt.to_poset(with_leaves=True)
Finite poset containing 3 elements
>>> P1 = bt.to_poset(with_leaves=True)
>>> len(P1.maximal_elements())
1
>>> len(P1.minimal_elements())
2
>>> bt = BinaryTree([])
>>> P2 = bt.to_poset(with_leaves=True,root_to_leaf=True)
>>> len(P2.maximal_elements())
2
>>> len(P2.minimal_elements())
1
```

If the tree is labelled, we use its labelling to label the poset. Otherwise, we use the poset canonical labelling:

```
sage: bt = BinaryTree([[],[None,[]]]).canonical_labelling()
sage: bt
2[1[., .], 3[., 4[., .]]]
sage: bt.to_poset().cover_relations()
[[4, 3], [3, 2], [1, 2]]
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[],[None,[]]]).canonical_labelling()
>>> bt
2[1[., .], 3[., 4[., .]]]
>>> bt.to_poset().cover_relations()
[[4, 3], [3, 2], [1, 2]]
```

Let us check that the empty binary tree is correctly handled:

```
sage: bt = BinaryTree()
sage: bt.to_poset()
Finite poset containing 0 elements
sage: bt.to_poset(with_leaves=True)
Finite poset containing 1 elements
```

```
>>> from sage.all import *
>>> bt = BinaryTree()
>>> bt.to_poset()
Finite poset containing 0 elements
>>> bt.to_poset(with_leaves=True)
Finite poset containing 1 elements
```

to_tilting()

Transform a binary tree into a tilting object.

Let t be a binary tree with n nodes. There exists a unique depiction of t (above the diagonal) such that all leaves are regularly distributed on the diagonal line from $(0, 0)$ to (n, n) and all edges are either horizontal or vertical. This method provides the coordinates of this depiction, with the root as the top-left vertex.

OUTPUT: list of pairs of integers

Every vertex of the binary tree is mapped to a pair of integers. The conventions are the following. The root has coordinates $(0, n)$ where n is the node number. If a vertex is the left (right) son of another vertex, they share the first (second) coordinate.

EXAMPLES:

```
sage: t = BinaryTrees(1)[0]
sage: t.to_tilting()
[(0, 1)]

sage: for t in BinaryTrees(2):
....:     print(t.to_tilting())
[(1, 2), (0, 2)]
[(0, 1), (0, 2)]

sage: from sage.combinat.abstract_tree import from_hexacode
sage: t = from_hexacode('2020222002000', BinaryTrees())
sage: print(t.to_tilting())
[(0, 1), (2, 3), (4, 5), (6, 7), (4, 7), (8, 9), (10, 11),
 (8, 11), (4, 11), (12, 13), (4, 13), (2, 13), (0, 13)]

sage: w = DyckWord([1,1,1,1,0,1,1,0,0,0,1,1,0,1,0,1,1,0,1,1,0,0,0,0,0,0])    #␣
↪needs sage.combinat
sage: t2 = w.to_binary_tree()                                                #␣
↪needs sage.combinat
sage: len(t2.to_tilting()) == t2.node_number()                               #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> t = BinaryTrees(Integer(1))[Integer(0)]
>>> t.to_tilting()
[(0, 1)]

>>> for t in BinaryTrees(Integer(2)):
...     print(t.to_tilting())
[(1, 2), (0, 2)]
[(0, 1), (0, 2)]

>>> from sage.combinat.abstract_tree import from_hexacode
>>> t = from_hexacode('2020222002000', BinaryTrees())
>>> print(t.to_tilting())
[(0, 1), (2, 3), (4, 5), (6, 7), (4, 7), (8, 9), (10, 11),
 (8, 11), (4, 11), (12, 13), (4, 13), (2, 13), (0, 13)]

>>> w = DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0)])    # needs sage.combinat
>>> t2 = w.to_binary_tree()                                                #␣
↪needs sage.combinat
>>> len(t2.to_tilting()) == t2.node_number()                               #␣
↪needs sage.combinat
True
```

to_undirected_graph(*with_leaves=False*)

Return the undirected graph obtained from the tree nodes and edges.

Leaves are ignored by default, but one can set `with_leaves` to `True` to obtain the graph of the complete tree.

INPUT:

- `with_leaves` – boolean (default: `False`); whether the resulting graph will be formed from the leaves and the nodes of `self` (if `True`), or only from the nodes of `self` (if `False`)

EXAMPLES:

```
sage: bt = BinaryTree([])
sage: bt.to_undirected_graph()
Graph on 1 vertex
sage: bt.to_undirected_graph(with_leaves=True)
Graph on 3 vertices

sage: bt = BinaryTree()
sage: bt.to_undirected_graph()
Graph on 0 vertices
sage: bt.to_undirected_graph(with_leaves=True)
Graph on 1 vertex
```

```
>>> from sage.all import *
>>> bt = BinaryTree([])
>>> bt.to_undirected_graph()
Graph on 1 vertex
>>> bt.to_undirected_graph(with_leaves=True)
Graph on 3 vertices

>>> bt = BinaryTree()
>>> bt.to_undirected_graph()
Graph on 0 vertices
>>> bt.to_undirected_graph(with_leaves=True)
Graph on 1 vertex
```

If the tree is labelled, we use its labelling to label the graph. Otherwise, we use the graph canonical labelling which means that two different trees can have the same graph.

EXAMPLES:

```
sage: bt = BinaryTree([[],[None,[]]])
sage: bt.canonical_labelling().to_undirected_graph() == bt.to_undirected_
↪graph()
False
sage: BinaryTree([[],[]]).to_undirected_graph() == BinaryTree([[[],None],
↪None]).to_undirected_graph()
True
```

```
>>> from sage.all import *
>>> bt = BinaryTree([[],[None,[]]])
>>> bt.canonical_labelling().to_undirected_graph() == bt.to_undirected_graph()
False
```

(continues on next page)

(continued from previous page)

```
>>> BinaryTree([[],[]]).to_undirected_graph() == BinaryTree([[[],None],None]).
↪to_undirected_graph()
True
```

twisting_number()

Return a pair (number of maximal left branches, number of maximal right branches).

Recalling that a branch of a vertex v is a path from a vertex of the tree to a leaf, a left (resp. right) branch is a branch made only of left (resp. right) edges. The length of a branch is the number of edges composing it. A left (resp. right) branch is maximal if it is not included in a strictly longer left (resp. right) branch.

OUTPUT: list of two integers

EXAMPLES:

```
sage: BT = BinaryTree( '.' )
sage: BT.twisting_number()
[0, 0]
sage: BT = BinaryTree( '[.,.]' )
sage: BT.twisting_number()
[0, 0]
sage: BT = BinaryTree( '[[[.,.], .], [.,.]]' ); ascii_art(BT)
    o
   / \
  o   o
 /
o
sage: BT.twisting_number()
[1, 1]
sage: BT = BinaryTree( '[[[[., [., .]], .], [[., .], [[[., .], [., .]], [., .
↪]]]], [., [[[., .], [[[., .], [., .]], .]], .]]]' )
sage: ascii_art(BT)
        ________o________
       /                 \
    __o__                 o
   /     \                 \
  o     __o___              o
 /     /      \            /
o     o       _o_       __o__
 \           /   \     /     \
  o         o     o   o       o
           / \               /
          o   o             o
                           / \
                          o   o
sage: BT.twisting_number()
[5, 6]
sage: BT = BinaryTree( '[.,[[[.,.],.],.]]' ); ascii_art(BT)
  o
   \
    o
   /
  o
```

(continues on next page)

(continued from previous page)

```
 /
o
sage: BT.twisting_number()
[1, 1]
```

```
>>> from sage.all import *
>>> BT = BinaryTree( '.' )
>>> BT.twisting_number()
[0, 0]
>>> BT = BinaryTree( '[.,.]' )
>>> BT.twisting_number()
[0, 0]
>>> BT = BinaryTree( '[[[.,.], .], [.,.]]' ); ascii_art(BT)
    o
   / \
  o   o
 /
o
>>> BT.twisting_number()
[1, 1]
>>> BT = BinaryTree( '[[[[[., [., .]], .], [[., .], [[[., .], [., .]], [., .]]]], [., [[[., .], [[[., .], [., .]], .]], .]]]' )
>>> ascii_art(BT)
        ________o________
       /                 \
    __o__                 o
   /     \                 \
  o     __o___              o
 /     /      \            /
o     o       _o_       __o__
 \           /   \     /     \
  o         o     o   o       o
           / \               /
          o   o             o
                           / \
                          o   o
>>> BT.twisting_number()
[5, 6]
>>> BT = BinaryTree( '[.,[[[.,.],.],.]]' ); ascii_art(BT)
  o
   \
    o
   /
  o
 /
o
>>> BT.twisting_number()
[1, 1]
```

under (*bt*)

Return `self` under `bt`, where "under" is the `under` ($\backslash$) operation.

If T and T' are two binary trees, then T under T' (written $T\backslash T'$) is defined as the tree obtained by grafting

T on the leftmost leaf of T'. More precisely, $T \backslash T'$ is defined by identifying the root of T with the leftmost leaf of T'.

If T' is empty, then $T \backslash T' = T$.

The definition of this "under" operation goes back to Loday-Ronco [LR0102066] (Definition 2.2), but it is denoted by $/$ and called the "over" operation there. In fact, trees in sage have their root at the top, contrary to the trees in [LR0102066] which are growing upwards. For this reason, the names of the over and under operations are swapped, in order to keep a graphical meaning. (Our notation follows that of section 4.5 of [HNT2005].)

See also:

over()

EXAMPLES:

Showing only the nodes of a binary tree, here is an example for the under operation:

```
sage: b1 = BinaryTree([[],[]])
sage: b2 = BinaryTree([None,[]])
sage: ascii_art((b1, b2, b1.under(b2)))
(   o  , o  ,     _o_   )
(  / \    \      /   \  )
( o   o    o    o     o )
(              / \      )
(             o   o     )
```

```
>>> from sage.all import *
>>> b1 = BinaryTree([[],[]])
>>> b2 = BinaryTree([None,[]])
>>> ascii_art((b1, b2, b1.under(b2)))
(   o  , o  ,     _o_   )
(  / \    \      /   \  )
( o   o    o    o     o )
(              / \      )
(             o   o     )
```

under_decomposition()

Return the unique maximal decomposition as an under product.

This means that the tree is cut along all edges of its leftmost path.

Beware that the factors are ordered starting from the root.

See also:

comb(), *over_decomposition()*

EXAMPLES:

```
sage: g = BinaryTree([])
sage: r = g.over(g); r
[., [., .]]
```

(continues on next page)

(continued from previous page)

```
sage: l = g.under(g); l
[[., .], .]
sage: l.under_decomposition()
[[., .], [., .]]
sage: r.under_decomposition() == [r]
True

sage: x = r.under(g).under(r).under(g)
sage: ascii_art(x)
      o
     /
    o
   / \
  o   o
 /
o
 \
  o
sage: x.under_decomposition() == [g,r,g,r]
True
```

```
>>> from sage.all import *
>>> g = BinaryTree([])
>>> r = g.over(g); r
[., [., .]]
>>> l = g.under(g); l
[[., .], .]
>>> l.under_decomposition()
[[., .], [., .]]
>>> r.under_decomposition() == [r]
True

>>> x = r.under(g).under(r).under(g)
>>> ascii_art(x)
      o
     /
    o
   / \
  o   o
 /
o
 \
  o
>>> x.under_decomposition() == [g,r,g,r]
True
```

class sage.combinat.binary_tree.**BinaryTrees**

Bases: UniqueRepresentation, Parent

Factory for binary trees.

A binary tree is a tree with at most 2 children. The binary trees considered here are also ordered (a.k.a. planar), that is to say, their children are ordered.

A full binary tree is a binary tree with no nodes with 1 child.

INPUT:

- `size` – integer (optional)
- `full` – boolean (optional)

OUTPUT:

The set of all (full if `full=True`) binary trees (of the given `size` if specified).

See also

BinaryTree, *BinaryTree.is_full()*

EXAMPLES:

```
sage: BinaryTrees()
Binary trees

sage: BinaryTrees(2)
Binary trees of size 2

sage: BinaryTrees(full=True)
Full binary trees

sage: BinaryTrees(3, full=True)
Full binary trees of size 3

sage: BinaryTrees(4, full=True)
Traceback (most recent call last):
...
ValueError: n must be 0 or odd
```

```
>>> from sage.all import *
>>> BinaryTrees()
Binary trees

>>> BinaryTrees(Integer(2))
Binary trees of size 2

>>> BinaryTrees(full=True)
Full binary trees

>>> BinaryTrees(Integer(3), full=True)
Full binary trees of size 3

>>> BinaryTrees(Integer(4), full=True)
Traceback (most recent call last):
...
ValueError: n must be 0 or odd
```

Note

This is a factory class whose constructor returns instances of subclasses.

Note

The fact that BinaryTrees is a class instead of a simple callable is an implementation detail. It could be changed in the future and one should not rely on it.

leaf()

Return a leaf tree with `self` as parent.

EXAMPLES:

```
sage: BinaryTrees().leaf()
.
```

```
>>> from sage.all import *
>>> BinaryTrees().leaf()
.
```

class sage.combinat.binary_tree.**BinaryTrees_all**

Bases: `DisjointUnionEnumeratedSets`, *BinaryTrees*

Element

alias of *BinaryTree*

labelled_trees()

Return the set of labelled trees associated to `self`.

EXAMPLES:

```
sage: BinaryTrees().labelled_trees()
Labelled binary trees
```

```
>>> from sage.all import *
>>> BinaryTrees().labelled_trees()
Labelled binary trees
```

unlabelled_trees()

Return the set of unlabelled trees associated to `self`.

EXAMPLES:

```
sage: BinaryTrees().unlabelled_trees()
Binary trees
```

```
>>> from sage.all import *
>>> BinaryTrees().unlabelled_trees()
Binary trees
```

class sage.combinat.binary_tree.**BinaryTrees_size**(*size*)

Bases: *BinaryTrees*

The enumerated sets of binary trees of given size.

cardinality()

The cardinality of `self`.

This is a Catalan number.

random_element()

Return a random `BinaryTree` with uniform probability.

This method generates a random `DyckWord` and then uses a bijection between Dyck words and binary trees.

EXAMPLES:

```
sage: BinaryTrees(5).random_element()  # random                              #␣
↪needs sage.combinat
[., [., [., [., [., .]]]]]
sage: BinaryTrees(0).random_element()                                        #␣
↪needs sage.combinat
.
sage: BinaryTrees(1).random_element()                                        #␣
↪needs sage.combinat
[., .]
```

```
>>> from sage.all import *
>>> BinaryTrees(Integer(5)).random_element()  # random                      ␣
↪    # needs sage.combinat
[., [., [., [., [., .]]]]]
>>> BinaryTrees(Integer(0)).random_element()                                ␣
↪    # needs sage.combinat
.
>>> BinaryTrees(Integer(1)).random_element()                                ␣
↪    # needs sage.combinat
[., .]
```

class sage.combinat.binary_tree.**FullBinaryTrees_all**

Bases: `DisjointUnionEnumeratedSets`, *BinaryTrees*

All full binary trees.

class sage.combinat.binary_tree.**FullBinaryTrees_size**(*size*)

Bases: *BinaryTrees*

Full binary trees of a fixed size (number of nodes).

cardinality()

The cardinality of `self`.

This is a Catalan number.

random_element()

Return a random `FullBinaryTree` with uniform probability.

This method generates a random `DyckWord` of size $(s-1)/2$, where s is the size of `self`, which uses a bijection between Dyck words and binary trees to get a binary tree, and convert it to a full binary tree.

EXAMPLES:

```
sage: BinaryTrees(5, full=True).random_element()  # random                    #␣
↪needs sage.combinat
[[], [[], []]]
sage: BinaryTrees(0, full=True).random_element()                              #␣
↪needs sage.combinat
.
sage: BinaryTrees(1, full=True).random_element()                              #␣
↪needs sage.combinat
[., .]
```

```
>>> from sage.all import *
>>> BinaryTrees(Integer(5), full=True).random_element()  # random            ␣
↪     # needs sage.combinat
[[], [[], []]]
>>> BinaryTrees(Integer(0), full=True).random_element()                      ␣
↪     # needs sage.combinat
.
>>> BinaryTrees(Integer(1), full=True).random_element()                      ␣
↪     # needs sage.combinat
[., .]
```

class sage.combinat.binary_tree.**LabelledBinaryTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: *AbstractLabelledClonableTree*, *BinaryTree*

Labelled binary trees.

A labelled binary tree is a binary tree (see *BinaryTree* for the meaning of this) with a label assigned to each node. The labels need not be integers, nor are they required to be distinct. `None` can be used as a label.

Warning

While it is possible to assign values to leaves (not just nodes) using this class, these labels are disregarded by various methods such as *labels()*, `map_labels()`, and (ironically) *leaf_labels()*.

INPUT:

- `children` – `None` (default) or a list, tuple or iterable of length 2 of labelled binary trees or convertible objects. This corresponds to the standard recursive definition of a labelled binary tree as being either a leaf, or a pair of:
  - a pair of labelled binary trees,
  - and a label.

  (The label is specified in the keyword variable `label`; see below.)

  Syntactic sugar allows leaving out all but the outermost calls of the `LabelledBinaryTree()` constructor, so that, e. g., `LabelledBinaryTree([LabelledBinaryTree(None), LabelledBinaryTree(None)])` can be shortened to `LabelledBinaryTree([None,None])`. However, using this shorthand, it is impossible to label any vertex of the tree other than the root (because there is no way to pass a `label` variable without calling `LabelledBinaryTree` explicitly).

  It is also allowed to abbreviate `[None, None]` by `[]` if one does not want to label the leaves (which one should not do anyway!).

- `label` – (default: `None`) the label to be put on the root of this tree

- `check` – boolean (default: `True`); whether checks should be performed or not

Todo

It is currently not possible to use `LabelledBinaryTree()` as a shorthand for `LabelledBinaryTree(None)` (in analogy to similar syntax in the `BinaryTree` class).

EXAMPLES:

```
sage: LabelledBinaryTree(None)
.
sage: LabelledBinaryTree(None, label='ae')    # not well supported
'ae'
sage: LabelledBinaryTree([])
None[., .]
sage: LabelledBinaryTree([], label=3)    # not well supported
3[., .]
sage: LabelledBinaryTree([None, None])
None[., .]
sage: LabelledBinaryTree([None, None], label=5)
5[., .]
sage: LabelledBinaryTree([None, []])
None[., None[., .]]
sage: LabelledBinaryTree([None, []], label=4)
4[., None[., .]]
sage: LabelledBinaryTree([[], None])
None[None[., .], .]
sage: LabelledBinaryTree("[[], .]", label=False)
False[None[., .], .]
sage: LabelledBinaryTree([None, LabelledBinaryTree([None, None], label=4)],
↪label=3)
3[., 4[., .]]
sage: LabelledBinaryTree([None, BinaryTree([None, None])], label=3)
3[., None[., .]]

sage: LabelledBinaryTree([[], None, []])
Traceback (most recent call last):
...
ValueError: this is not a binary tree

sage: LBT = LabelledBinaryTree
sage: t1 = LBT([[LBT([], label=2), None], None], label=4); t1
4[None[2[., .], .], .]
```

```
>>> from sage.all import *
>>> LabelledBinaryTree(None)
.
>>> LabelledBinaryTree(None, label='ae')    # not well supported
'ae'
>>> LabelledBinaryTree([])
None[., .]
>>> LabelledBinaryTree([], label=Integer(3))    # not well supported
```

(continues on next page)

(continued from previous page)

```
3[., .]
>>> LabelledBinaryTree([None, None])
None[., .]
>>> LabelledBinaryTree([None, None], label=Integer(5))
5[., .]
>>> LabelledBinaryTree([None, []])
None[., None[., .]]
>>> LabelledBinaryTree([None, []], label=Integer(4))
4[., None[., .]]
>>> LabelledBinaryTree([[], None])
None[None[., .], .]
>>> LabelledBinaryTree("[[], .]", label=False)
False[None[., .], .]
>>> LabelledBinaryTree([None, LabelledBinaryTree([None, None], label=Integer(4))],
↪ label=Integer(3))
3[., 4[., .]]
>>> LabelledBinaryTree([None, BinaryTree([None, None])], label=Integer(3))
3[., None[., .]]

>>> LabelledBinaryTree([[], None, []])
Traceback (most recent call last):
...
ValueError: this is not a binary tree

>>> LBT = LabelledBinaryTree
>>> t1 = LBT([[LBT([], label=Integer(2)), None], None], label=Integer(4)); t1
4[None[2[., .], .], .]
```

binary_search_insert(*letter*)

Return the result of inserting a letter `letter` into the right strict binary search tree `self`.

INPUT:

- `letter` – any object comparable with the labels of `self`

OUTPUT:

The right strict binary search tree `self` with `letter` inserted into it according to the binary search insertion algorithm.

Note

`self` is supposed to be a binary search tree. This is not being checked!

A right strict binary search tree is defined to be a labelled binary tree such that for each node n with label x, every descendant of the left child of n has a label $\leq x$, and every descendant of the right child of n has a label $> x$. (Here, only nodes count as descendants, and every node counts as its own descendant too.) Leaves are assumed to have no labels.

Given a right strict binary search tree t and a letter i, the result of inserting i into t (denoted $Ins(i,t)$ in the following) is defined recursively as follows:

- If t is empty, then $Ins(i,t)$ is the tree with one node only, and this node is labelled with i.

- Otherwise, let j be the label of the root of t. If $i > j$, then $Ins(i, t)$ is obtained by replacing the right child of t by $Ins(i, r)$ in t, where r denotes the right child of t. If $i \leq j$, then $Ins(i, t)$ is obtained by replacing the left child of t by $Ins(i, l)$ in t, where l denotes the left child of t.

See, for example, [HNT2005] for properties of this algorithm.

Warning

If t is nonempty, then inserting i into t does not change the root label of t. Hence, as opposed to algorithms like Robinson-Schensted-Knuth, binary search tree insertion involves no bumping.

EXAMPLES:

The example from Fig. 2 of [HNT2005]:

```
sage: LBT = LabelledBinaryTree
sage: x = LBT(None)
sage: x
.
sage: x = x.binary_search_insert("b"); x
b[., .]
sage: x = x.binary_search_insert("d"); x
b[., d[., .]]
sage: x = x.binary_search_insert("e"); x
b[., d[., e[., .]]]
sage: x = x.binary_search_insert("a"); x
b[a[., .], d[., e[., .]]]
sage: x = x.binary_search_insert("b"); x
b[a[., b[., .]], d[., e[., .]]]
sage: x = x.binary_search_insert("d"); x
b[a[., b[., .]], d[d[., .], e[., .]]]
sage: x = x.binary_search_insert("a"); x
b[a[a[., .], b[., .]], d[d[., .], e[., .]]]
sage: x = x.binary_search_insert("c"); x
b[a[a[., .], b[., .]], d[d[c[., .], .], e[., .]]]
```

```
>>> from sage.all import *
>>> LBT = LabelledBinaryTree
>>> x = LBT(None)
>>> x
.
>>> x = x.binary_search_insert("b"); x
b[., .]
>>> x = x.binary_search_insert("d"); x
b[., d[., .]]
>>> x = x.binary_search_insert("e"); x
b[., d[., e[., .]]]
>>> x = x.binary_search_insert("a"); x
b[a[., .], d[., e[., .]]]
>>> x = x.binary_search_insert("b"); x
b[a[., b[., .]], d[., e[., .]]]
>>> x = x.binary_search_insert("d"); x
b[a[., b[., .]], d[d[., .], e[., .]]]
```

(continues on next page)

(continued from previous page)

```
>>> x = x.binary_search_insert("a"); x
b[a[a[., .], b[., .]], d[d[., .], e[., .]]]
>>> x = x.binary_search_insert("c"); x
b[a[a[., .], b[., .]], d[d[c[., .], .], e[., .]]]
```

Other examples:

```
sage: LBT = LabelledBinaryTree
sage: LBT(None).binary_search_insert(3)
3[., .]
sage: LBT([], label = 1).binary_search_insert(3)
1[., 3[., .]]
sage: LBT([], label = 3).binary_search_insert(1)
3[1[., .], .]
sage: res = LBT(None)
sage: for i in [3,1,5,2,4,6]:
....:     res = res.binary_search_insert(i)
sage: res
3[1[., 2[., .]], 5[4[., .], 6[., .]]]
```

```
>>> from sage.all import *
>>> LBT = LabelledBinaryTree
>>> LBT(None).binary_search_insert(Integer(3))
3[., .]
>>> LBT([], label = Integer(1)).binary_search_insert(Integer(3))
1[., 3[., .]]
>>> LBT([], label = Integer(3)).binary_search_insert(Integer(1))
3[1[., .], .]
>>> res = LBT(None)
>>> for i in [Integer(3),Integer(1),Integer(5),Integer(2),Integer(4),
↪Integer(6)]:
...     res = res.binary_search_insert(i)
>>> res
3[1[., 2[., .]], 5[4[., .], 6[., .]]]
```

heap_insert(*l*)

Return the result of inserting a letter `l` into the binary heap (tree) `self`.

A binary heap is a labelled complete binary tree such that for each node, the label at the node is greater or equal to the label of each of its child nodes. (More precisely, this is called a max-heap.)

For example:

```
|      _7_    |
|     /   \   |
|    5      6 |
|   / \       |
| 3    4      |
```

is a binary heap.

See Wikipedia article Binary_heap#Insert for a description of how to insert a letter into a binary heap. The result is another binary heap.

INPUT:

- `letter` – any object comparable with the labels of `self`

Note: `self` is assumed to be a binary heap (tree). No check is performed.

left_rotate()

Return the result of left rotation applied to the labelled binary tree `self`.

Left rotation on labelled binary trees is defined as follows: Let T be a labelled binary tree such that the right child of the root of T is a node. Let A be the left child of the root of T, and let B and C be the left and right children of the right child of the root of T. (Keep in mind that nodes of trees are identified with the subtrees consisting of their descendants.) Furthermore, let x be the label at the root of T, and y be the label at the right child of the root of T. Then, the left rotation of T is the labelled binary tree in which the root is labelled y, the right child of the root is C, whereas the left child of the root is a node labelled x whose left and right children are A and B. In pictures:

```
|     y                    x     |
|    / \                  / \    |
|   x   C <-left-rotate- A   y   |
|  / \                      / \  |
| A   B                    B   C |
```

Left rotation is the inverse operation to right rotation (*right_rotate()*).

right_rotate()

Return the result of right rotation applied to the labelled binary tree `self`.

Right rotation on labelled binary trees is defined as follows: Let T be a labelled binary tree such that the left child of the root of T is a node. Let C be the right child of the root of T, and let A and B be the left and right children of the left child of the root of T. (Keep in mind that nodes of trees are identified with the subtrees consisting of their descendants.) Furthermore, let y be the label at the root of T, and x be the label at the left child of the root of T. Then, the right rotation of T is the labelled binary tree in which the root is labelled x, the left child of the root is A, whereas the right child of the root is a node labelled y whose left and right children are B and C. In pictures:

```
|     y                      x     |
|    / \                    / \    |
|   x   C -right-rotate->  A   y   |
|  / \                        / \  |
| A   B                      B   C |
```

Right rotation is the inverse operation to left rotation (*left_rotate()*).

semistandard_insert(*letter*)

Return the result of inserting a letter `letter` into the semistandard tree `self` using the bumping algorithm.

INPUT:

- `letter` – any object comparable with the labels of `self`

OUTPUT:

The semistandard tree `self` with `letter` inserted into it according to the bumping algorithm.

Note: `self` is supposed to be a semistandard tree. This is not being checked!

A semistandard tree is defined to be a labelled binary tree such that for each node n with label x, every descendant of the left child of n has a label $> x$, and every descendant of the right child of n has a label $\geq x$. (Here, only nodes count as descendants, and every node counts as its own descendant too.) Leaves are assumed to have no labels.

Given a semistandard tree t and a letter i, the result of inserting i into t (denoted $Ins(i, t)$ in the following) is defined recursively as follows:

- If t is empty, then $Ins(i, t)$ is the tree with one node only, and this node is labelled with i.
- Otherwise, let j be the label of the root of t. If $i \geq j$, then $Ins(i, t)$ is obtained by replacing the right child of t by $Ins(i, r)$ in t, where r denotes the right child of t. If $i < j$, then $Ins(i, t)$ is obtained by replacing the label at the root of t by i, and replacing the left child of t by $Ins(j, l)$ in t, where l denotes the left child of t.

This algorithm is similar to the Robinson-Schensted-Knuth insertion algorithm for semistandard Young tableaux.

AUTHORS:

- Darij Grinberg (10 Nov 2013).

EXAMPLES:

```
sage: LBT = LabelledBinaryTree
sage: x = LBT(None)
sage: x
.
sage: x = x.semistandard_insert("b"); x
b[., .]
sage: x = x.semistandard_insert("d"); x
b[., d[., .]]
sage: x = x.semistandard_insert("e"); x
b[., d[., e[., .]]]
sage: x = x.semistandard_insert("a"); x
a[b[., .], d[., e[., .]]]
sage: x = x.semistandard_insert("b"); x
a[b[., .], b[d[., .], e[., .]]]
sage: x = x.semistandard_insert("d"); x
a[b[., .], b[d[., .], d[e[., .], .]]]
sage: x = x.semistandard_insert("a"); x
a[b[., .], a[b[d[., .], .], d[e[., .], .]]]
sage: x = x.semistandard_insert("c"); x
a[b[., .], a[b[d[., .], .], c[d[e[., .], .], .]]]
```

```
>>> from sage.all import *
>>> LBT = LabelledBinaryTree
>>> x = LBT(None)
>>> x
.
>>> x = x.semistandard_insert("b"); x
b[., .]
```

(continues on next page)

(continued from previous page)

```
>>> x = x.semistandard_insert("d"); x
b[., d[., .]]
>>> x = x.semistandard_insert("e"); x
b[., d[., e[., .]]]
>>> x = x.semistandard_insert("a"); x
a[b[., .], d[., e[., .]]]
>>> x = x.semistandard_insert("b"); x
a[b[., .], b[d[., .], e[., .]]]
>>> x = x.semistandard_insert("d"); x
a[b[., .], b[d[., .], d[e[., .], .]]]
>>> x = x.semistandard_insert("a"); x
a[b[., .], a[b[d[., .], .], d[e[., .], .]]]
>>> x = x.semistandard_insert("c"); x
a[b[., .], a[b[d[., .], .], c[d[e[., .], .], .]]]
```

Other examples:

```
sage: LBT = LabelledBinaryTree
sage: LBT(None).semistandard_insert(3)
3[., .]
sage: LBT([], label = 1).semistandard_insert(3)
1[., 3[., .]]
sage: LBT([], label = 3).semistandard_insert(1)
1[3[., .], .]
sage: res = LBT(None)
sage: for i in [3,1,5,2,4,6]:
....:     res = res.semistandard_insert(i)
sage: res
1[3[., .], 2[5[., .], 4[., 6[., .]]]]
```

```
>>> from sage.all import *
>>> LBT = LabelledBinaryTree
>>> LBT(None).semistandard_insert(Integer(3))
3[., .]
>>> LBT([], label = Integer(1)).semistandard_insert(Integer(3))
1[., 3[., .]]
>>> LBT([], label = Integer(3)).semistandard_insert(Integer(1))
1[3[., .], .]
>>> res = LBT(None)
>>> for i in [Integer(3),Integer(1),Integer(5),Integer(2),Integer(4),
↪Integer(6)]:
...     res = res.semistandard_insert(i)
>>> res
1[3[., .], 2[5[., .], 4[., 6[., .]]]]
```

class sage.combinat.binary_tree.**LabelledBinaryTrees**(*category=None*)

Bases: *LabelledOrderedTrees*

This is a parent stub to serve as a factory class for trees with various labels constraints.

Element

alias of *LabelledBinaryTree*

labelled_trees()

Return the set of labelled trees associated to self.

EXAMPLES:

```
sage: LabelledBinaryTrees().labelled_trees()
Labelled binary trees
```

```
>>> from sage.all import *
>>> LabelledBinaryTrees().labelled_trees()
Labelled binary trees
```

unlabelled_trees()

Return the set of unlabelled trees associated to self.

EXAMPLES:

```
sage: LabelledBinaryTrees().unlabelled_trees()
Binary trees
```

```
>>> from sage.all import *
>>> LabelledBinaryTrees().unlabelled_trees()
Binary trees
```

This is used to compute the shape:

```
sage: t = LabelledBinaryTrees().an_element().shape(); t
[[[., .], [., .]], [[., .], [., .]]]
sage: t.parent()
Binary trees
```

```
>>> from sage.all import *
>>> t = LabelledBinaryTrees().an_element().shape(); t
[[[., .], [., .]], [[., .], [., .]]]
>>> t.parent()
Binary trees
```

sage.combinat.binary_tree.**binary_search_tree_shape**(*w*, *left_to_right=True*)

Direct computation of the binary search tree shape of a list of integers.

INPUT:

- w – list of integers
- left_to_right – boolean (default: True)

OUTPUT: a non labelled binary tree

This is used under the same name as a method for permutations.

EXAMPLES:

```
sage: from sage.combinat.binary_tree import binary_search_tree_shape
sage: binary_search_tree_shape([1,4,3,2])
[., [[[., .], .], .]]
sage: binary_search_tree_shape([5,1,3,2])
[[., [[., .], .]], .]
```

```
>>> from sage.all import *
>>> from sage.combinat.binary_tree import binary_search_tree_shape
>>> binary_search_tree_shape([Integer(1),Integer(4),Integer(3),Integer(2)])
[., [[[., .], .], .]]
>>> binary_search_tree_shape([Integer(5),Integer(1),Integer(3),Integer(2)])
[[., [[., .], .]], .]
```

By passing the option `left_to_right=False` one can have the insertion going from right to left:

```
sage: binary_search_tree_shape([1,6,4,2], False)
[[., .], [., [., .]]]
```

```
>>> from sage.all import *
>>> binary_search_tree_shape([Integer(1),Integer(6),Integer(4),Integer(2)], False)
[[., .], [., [., .]]]
```

sage.combinat.binary_tree.**from_tamari_sorting_tuple**(*key*)

Return a binary tree from its Tamari-sorting tuple.

See *tamari_sorting_tuple()*

INPUT:

- `key` – tuple of integers

EXAMPLES:

```
sage: from sage.combinat.binary_tree import from_tamari_sorting_tuple
sage: t = BinaryTrees(60).random_element()                                  #␣
↪needs sage.combinat
sage: from_tamari_sorting_tuple(t.tamari_sorting_tuple()[0]) == t           #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> from sage.combinat.binary_tree import from_tamari_sorting_tuple
>>> t = BinaryTrees(Integer(60)).random_element()                              ␣
↪    # needs sage.combinat
>>> from_tamari_sorting_tuple(t.tamari_sorting_tuple()[Integer(0)]) == t       ␣
↪    # needs sage.combinat
True
```

5.1.11 Blob Algebras

AUTHORS:

- Travis Scrimshaw (2020-05-16): Initial version

class sage.combinat.blob_algebra.**BlobAlgebra**(*k*, *q1*, *q2*, *q3*, *base_ring*, *prefix*)

Bases: *CombinatorialFreeModule*

The blob algebra.

The *blob algebra* (also known as the Temperley-Lieb algebra of type B in [ILZ2018], but is a quotient of the Temperley-Lieb algebra of type B defined in [Graham1985]) is a diagram-type algebra introduced in [MS1994] whose basis consists of `Temperley-Lieb diagrams`, noncrossing perfect matchings, that may contain blobs on strands that can be deformed so that the blob touches the left side (which we can think of as a frozen pole).

The form we give here has 3 parameters, the natural one from the `Temperley-Lieb algebra`, one for the idempotent relation, and one for a loop with a blob.

INPUT:

- `k` – the order
- `q1` – the loop parameter
- `q2` – the idempotent parameter
- `q3` – the blob loop parameter

EXAMPLES:

```
sage: R.<q,r,s> = ZZ[]
sage: B4 = algebras.Blob(4, q, r, s)
sage: B = sorted(B4.basis())
sage: B[14]
B({{-4, -3}}, {{-2, -1}, {1, 2}, {3, 4}})
sage: B[40]
B({{3, 4}}, {{-4, -3}, {-2, -1}, {1, 2}})
sage: B[14] * B[40]
q*r*s*B({}, {{-4, -3}, {-2, -1}, {1, 2}, {3, 4}})
```

```
>>> from sage.all import *
>>> R = ZZ['q, r, s']; (q, r, s,) = R._first_ngens(3)
>>> B4 = algebras.Blob(Integer(4), q, r, s)
>>> B = sorted(B4.basis())
>>> B[Integer(14)]
B({{-4, -3}}, {{-2, -1}, {1, 2}, {3, 4}})
>>> B[Integer(40)]
B({{3, 4}}, {{-4, -3}, {-2, -1}, {1, 2}})
>>> B[Integer(14)] * B[Integer(40)]
q*r*s*B({}, {{-4, -3}, {-2, -1}, {1, 2}, {3, 4}})
```

REFERENCES:

- [MS1994]
- [ILZ2018]

one_basis()

Return the index of the basis element 1.

EXAMPLES:

```
sage: R.<q,r,s> = ZZ[]
sage: B4 = algebras.Blob(4, q, r, s)
sage: B4.one_basis()
({}, {{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}})
```

```
>>> from sage.all import *
>>> R = ZZ['q, r, s']; (q, r, s,) = R._first_ngens(3)
>>> B4 = algebras.Blob(Integer(4), q, r, s)
```

(continues on next page)

(continued from previous page)

```
>>> B4.one_basis()
({}, {{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}})
```

order()

Return the order of `self`.

The order of a partition algebra is defined as half of the number of nodes in the diagrams.

EXAMPLES:

```
sage: R.<q,r,s> = ZZ[]
sage: B4 = algebras.Blob(4, q, r, s)
sage: B4.order()
4
```

```
>>> from sage.all import *
>>> R = ZZ['q, r, s']; (q, r, s,) = R._first_ngens(3)
>>> B4 = algebras.Blob(Integer(4), q, r, s)
>>> B4.order()
4
```

product_on_basis(*top*, *bot*)

Return the product of the basis elements indexed by `top` and `bot`.

EXAMPLES:

```
sage: R.<q,r,s> = ZZ[]
sage: B4 = algebras.Blob(4, q, r, s)
sage: B = B4.basis()
sage: BD = sorted(B.keys())
sage: BD[14]
({{-4, -3}}, {{-2, -1}, {1, 2}, {3, 4}})
sage: BD[40]
({{3, 4}}, {{-4, -3}, {-2, -1}, {1, 2}})
sage: B4.product_on_basis(BD[14], BD[40])
q*r*s*B({}, {{-4, -3}, {-2, -1}, {1, 2}, {3, 4}})
sage: all(len((x*y).support()) == 1 for x in B for y in B)
True
```

```
>>> from sage.all import *
>>> R = ZZ['q, r, s']; (q, r, s,) = R._first_ngens(3)
>>> B4 = algebras.Blob(Integer(4), q, r, s)
>>> B = B4.basis()
>>> BD = sorted(B.keys())
>>> BD[Integer(14)]
({{-4, -3}}, {{-2, -1}, {1, 2}, {3, 4}})
>>> BD[Integer(40)]
({{3, 4}}, {{-4, -3}, {-2, -1}, {1, 2}})
>>> B4.product_on_basis(BD[Integer(14)], BD[Integer(40)])
q*r*s*B({}, {{-4, -3}, {-2, -1}, {1, 2}, {3, 4}})
>>> all(len((x*y).support()) == Integer(1) for x in B for y in B)
True
```

class sage.combinat.blob_algebra.**BlobDiagram**(*parent*, *marked*, *unmarked*)

Bases: Element

A blob diagram.

A blob diagram consists of a perfect matching of the set $\{1, \ldots, n\} \sqcup \{-1, \ldots, -n\}$ such that the result is a noncrossing matching (a *Temperley-Lieb diagram*), divided into two sets of pairs: one for the pairs with blobs and one for those without. The blobed pairs must either be either the leftmost propagating strand or to the left of it and not nested.

temperley_lieb_diagram()

Return the Temperley-Lieb diagram corresponding to self.

EXAMPLES:

```
sage: from sage.combinat.blob_algebra import BlobDiagrams
sage: BD4 = BlobDiagrams(4)
sage: B = BD4([[1,-3]], [[2,-4], [3,4], [-1,-2]])
sage: B.temperley_lieb_diagram()
{{-4, 2}, {-3, 1}, {-2, -1}, {3, 4}}
```

```
>>> from sage.all import *
>>> from sage.combinat.blob_algebra import BlobDiagrams
>>> BD4 = BlobDiagrams(Integer(4))
>>> B = BD4([[Integer(1),-Integer(3)]], [[Integer(2),-Integer(4)],
↪[Integer(3),Integer(4)], [-Integer(1),-Integer(2)]])
>>> B.temperley_lieb_diagram()
{{-4, 2}, {-3, 1}, {-2, -1}, {3, 4}}
```

class sage.combinat.blob_algebra.**BlobDiagrams**(*n*)

Bases: Parent, UniqueRepresentation

The set of all blob diagrams.

Element

alias of *BlobDiagram*

base_set()

Return the base set of self.

EXAMPLES:

```
sage: from sage.combinat.blob_algebra import BlobDiagrams
sage: BD4 = BlobDiagrams(4)
sage: sorted(BD4.base_set())
[-4, -3, -2, -1, 1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> from sage.combinat.blob_algebra import BlobDiagrams
>>> BD4 = BlobDiagrams(Integer(4))
>>> sorted(BD4.base_set())
[-4, -3, -2, -1, 1, 2, 3, 4]
```

cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: from sage.combinat.blob_algebra import BlobDiagrams
sage: BD4 = BlobDiagrams(4)
sage: BD4.cardinality()
70
```

```
>>> from sage.all import *
>>> from sage.combinat.blob_algebra import BlobDiagrams
>>> BD4 = BlobDiagrams(Integer(4))
>>> BD4.cardinality()
70
```

order()

Return the order of `self`.

EXAMPLES:

```
sage: from sage.combinat.blob_algebra import BlobDiagrams
sage: BD4 = BlobDiagrams(4)
sage: BD4.order()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.blob_algebra import BlobDiagrams
>>> BD4 = BlobDiagrams(Integer(4))
>>> BD4.order()
4
```

5.1.12 Cartesian Products

class sage.combinat.cartesian_product.**CartesianProduct_iters**(**iters*)

Bases: `EnumeratedSetFromIterator`

Cartesian product of finite sets.

This class will soon be deprecated (see Issue #18411 and Issue #19195). One should instead use the functorial construction `cartesian_product`. The main differences in behavior are:

- construction: `CartesianProduct` takes as many argument as there are factors whereas `cartesian_product` takes a single list (or iterable) of factors;
- representation of elements: elements are represented by plain Python list for `CartesianProduct` versus a custom element class for `cartesian_product`;
- membership testing: because of the above, plain Python lists are not considered as elements of a `cartesian_product`.

All of these is illustrated in the examples below.

EXAMPLES:

```
sage: F1 = ['a', 'b']
sage: F2 = [1, 2, 3, 4]
sage: F3 = Permutations(3)
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: C = CartesianProduct_iters(F1, F2, F3)
```

(continues on next page)

(continued from previous page)

```
sage: c = cartesian_product([F1, F2, F3])

sage: type(C.an_element())
<class 'list'>
sage: type(c.an_element())
<class 'sage.sets.cartesian_product.CartesianProduct_with_category.element_class'>

sage: l = ['a', 1, Permutation([3,2,1])]
sage: l in C
True
sage: l in c
False
sage: elt = c(l)
sage: elt
('a', 1, [3, 2, 1])
sage: elt in c
True
sage: elt.parent() is c
True
```

```
>>> from sage.all import *
>>> F1 = ['a', 'b']
>>> F2 = [Integer(1), Integer(2), Integer(3), Integer(4)]
>>> F3 = Permutations(Integer(3))
>>> from sage.combinat.cartesian_product import CartesianProduct_iters
>>> C = CartesianProduct_iters(F1, F2, F3)
>>> c = cartesian_product([F1, F2, F3])

>>> type(C.an_element())
<class 'list'>
>>> type(c.an_element())
<class 'sage.sets.cartesian_product.CartesianProduct_with_category.element_class'>

>>> l = ['a', Integer(1), Permutation([Integer(3),Integer(2),Integer(1)])]
>>> l in C
True
>>> l in c
False
>>> elt = c(l)
>>> elt
('a', 1, [3, 2, 1])
>>> elt in c
True
>>> elt.parent() is c
True
```

cardinality()

Return the number of elements in the Cartesian product of everything in *iters.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
```

(continues on next page)

(continued from previous page)

```
sage: CartesianProduct_iters(range(2), range(3)).cardinality()
6
sage: CartesianProduct_iters(range(2), range(3)).cardinality()
6
sage: CartesianProduct_iters(range(2), range(3), range(4)).cardinality()
24
```

```
>>> from sage.all import *
>>> from sage.combinat.cartesian_product import CartesianProduct_iters
>>> CartesianProduct_iters(range(Integer(2)), range(Integer(3))).cardinality()
6
>>> CartesianProduct_iters(range(Integer(2)), range(Integer(3))).cardinality()
6
>>> CartesianProduct_iters(range(Integer(2)), range(Integer(3)),
→range(Integer(4))).cardinality()
24
```

This works correctly for infinite objects:

```
sage: CartesianProduct_iters(ZZ, QQ).cardinality()
+Infinity
sage: CartesianProduct_iters(ZZ, []).cardinality()
0
```

```
>>> from sage.all import *
>>> CartesianProduct_iters(ZZ, QQ).cardinality()
+Infinity
>>> CartesianProduct_iters(ZZ, []).cardinality()
0
```

is_finite()

The Cartesian product is finite if all of its inputs are finite, or if any input is empty.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: CartesianProduct_iters(ZZ, []).is_finite()
True
sage: CartesianProduct_iters(4,4).is_finite()
Traceback (most recent call last):
...
ValueError: unable to determine whether this product is finite
```

```
>>> from sage.all import *
>>> from sage.combinat.cartesian_product import CartesianProduct_iters
>>> CartesianProduct_iters(ZZ, []).is_finite()
True
>>> CartesianProduct_iters(Integer(4),Integer(4)).is_finite()
Traceback (most recent call last):
...
ValueError: unable to determine whether this product is finite
```

list()

Return.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: CartesianProduct_iters(range(3), range(3)).list()
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
sage: CartesianProduct_iters('dog', 'cat').list()
[['d', 'c'],
 ['d', 'a'],
 ['d', 't'],
 ['o', 'c'],
 ['o', 'a'],
 ['o', 't'],
 ['g', 'c'],
 ['g', 'a'],
 ['g', 't']]
```

```
>>> from sage.all import *
>>> from sage.combinat.cartesian_product import CartesianProduct_iters
>>> CartesianProduct_iters(range(Integer(3)), range(Integer(3))).list()
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
>>> CartesianProduct_iters('dog', 'cat').list()
[['d', 'c'],
 ['d', 'a'],
 ['d', 't'],
 ['o', 'c'],
 ['o', 'a'],
 ['o', 't'],
 ['g', 'c'],
 ['g', 'a'],
 ['g', 't']]
```

random_element()

Return a random element from the Cartesian product of *iters.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: c = CartesianProduct_iters('dog', 'cat').random_element()
sage: c in CartesianProduct_iters('dog', 'cat')
True
```

```
>>> from sage.all import *
>>> from sage.combinat.cartesian_product import CartesianProduct_iters
>>> c = CartesianProduct_iters('dog', 'cat').random_element()
>>> c in CartesianProduct_iters('dog', 'cat')
True
```

unrank(x)

For finite Cartesian products, we can reduce unrank to the constituent iterators.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: C = CartesianProduct_iters(range(1000), range(1000), range(1000))
sage: C[238792368]
[238, 792, 368]
```

```
>>> from sage.all import *
>>> from sage.combinat.cartesian_product import CartesianProduct_iters
>>> C = CartesianProduct_iters(range(Integer(1000)), range(Integer(1000)),␣
→range(Integer(1000)))
>>> C[Integer(238792368)]
[238, 792, 368]
```

Check for Issue #15919:

```
sage: FF = IntegerModRing(29)
sage: C = CartesianProduct_iters(FF, FF, FF)
sage: C.unrank(0)
[0, 0, 0]
```

```
>>> from sage.all import *
>>> FF = IntegerModRing(Integer(29))
>>> C = CartesianProduct_iters(FF, FF, FF)
>>> C.unrank(Integer(0))
[0, 0, 0]
```

5.1.13 Enumerated sets of partitions, tableaux, ...

Partitions

- *Integer partitions*
- *Skew Partitions*
- *Partition tuples*
- *Super Partitions*
- *Tableaux*
- *TableauTuples*
- *Skew Tableaux*
- *Ribbons*
- *Ribbon Tableaux*
- *Strong and weak tableaux*
- *Shifted primed tableaux*
- *Residue sequences of tableaux*

RSK

- *Robinson-Schensted-Knuth correspondence*
- *Growth diagrams and dual graded graphs*

5.1.14 Combinatorial Hopf algebras

- *Symmetric Functions*
- *Non-commutative symmetric functions and quasi-symmetric functions*
- *Symmetric functions in non-commuting variables*
- *Schubert Polynomials*
- *Poirier-Reutenauer Hopf algebra of standard tableaux*
- *Free Quasi-symmetric functions*
- *Grossman-Larson Hopf Algebras*
- *Word Quasi-symmetric functions*

5.1.15 Poirier-Reutenauer Hopf algebra of standard tableaux

AUTHORS:

- Franco Saliola (2012): initial implementation
- Travis Scrimshaw (2018-04-11): added missing doctests and reorganization

class sage.combinat.chas.fsym.**FSymBases**(*parent_with_realization*)

Bases: Category_realization_of_parent

The category of graded bases of $FSym$ and $FSym^*$ indexed by standard tableaux.

class ElementMethods

Bases: object

duality_pairing(*other*)

Compute the pairing between self and an element other of the dual.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: G = FSym.G()
sage: F = G.dual_basis()
sage: elt = G[[1,3],[2]] - 3*G[[1,2],[3]]
sage: elt.duality_pairing(F[[1,3],[2]])
1
sage: elt.duality_pairing(F[[1,2],[3]])
-3
sage: elt.duality_pairing(F[[1,2]])
0
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> G = FSym.G()
>>> F = G.dual_basis()
>>> elt = G[[Integer(1),Integer(3)],[Integer(2)]] -␣
↪Integer(3)*G[[Integer(1),Integer(2)],[Integer(3)]]
>>> elt.duality_pairing(F[[Integer(1),Integer(3)],[Integer(2)]])
1
>>> elt.duality_pairing(F[[Integer(1),Integer(2)],[Integer(3)]])
-3
```

(continues on next page)

(continued from previous page)

```
>>> elt.duality_pairing(F[[Integer(1),Integer(2)]])
0
```

class ParentMethods

Bases: `object`

basis(*degree=None*)

The basis elements (optionally: of the specified degree).

OUTPUT: family

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: TG = FSym.G()
sage: TG.basis()
Lazy family (Term map from Standard tableaux to Hopf algebra of standard␣
↪tableaux
 over the Rational Field in the Fundamental basis(i))_{i in Standard␣
↪tableaux}
sage: TG.basis().keys()
Standard tableaux
sage: TG.basis(degree=3).keys()
Standard tableaux of size 3
sage: TG.basis(degree=3).list()
[G[123], G[13|2], G[12|3], G[1|2|3]]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> TG = FSym.G()
>>> TG.basis()
Lazy family (Term map from Standard tableaux to Hopf algebra of standard␣
↪tableaux
 over the Rational Field in the Fundamental basis(i))_{i in Standard␣
↪tableaux}
>>> TG.basis().keys()
Standard tableaux
>>> TG.basis(degree=Integer(3)).keys()
Standard tableaux of size 3
>>> TG.basis(degree=Integer(3)).list()
[G[123], G[13|2], G[12|3], G[1|2|3]]
```

degree_on_basis(*t*)

Return the degree of a standard tableau in the algebra of free symmetric functions.

This is the size of the tableau `t`.

EXAMPLES:

```
sage: G = algebras.FSym(QQ).G()
sage: t = StandardTableau([[1,3],[2]])
sage: G.degree_on_basis(t)
3
sage: u = StandardTableau([[1,3,4,5],[2]])
```

(continues on next page)

(continued from previous page)

```
sage: G.degree_on_basis(u)
5
```

```
>>> from sage.all import *
>>> G = algebras.FSym(QQ).G()
>>> t = StandardTableau([[Integer(1),Integer(3)],[Integer(2)]])
>>> G.degree_on_basis(t)
3
>>> u = StandardTableau([[Integer(1),Integer(3),Integer(4),Integer(5)],
↪[Integer(2)]])
>>> G.degree_on_basis(u)
5
```

duality_pairing(*x*, *y*)

The canonical pairing between $FSym$ and $FSym^*$.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: G = FSym.G()
sage: F = G.dual_basis()
sage: t1 = StandardTableau([[1,3,5],[2,4]])
sage: t2 = StandardTableau([[1,3],[2,5],[4]])
sage: G.duality_pairing(G[t1], F[t2])
0
sage: G.duality_pairing(G[t1], F[t1])
1
sage: G.duality_pairing(G[t2], F[t2])
1
sage: F.duality_pairing(F[t2], G[t2])
1

sage: z = G[[1,3,5],[2,4]]
sage: all(F.duality_pairing(F[p1] * F[p2], z) == c
....:     for ((p1, p2), c) in z.coproduct())
True
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> G = FSym.G()
>>> F = G.dual_basis()
>>> t1 = StandardTableau([[Integer(1),Integer(3),Integer(5)],[Integer(2),
↪Integer(4)]])
>>> t2 = StandardTableau([[Integer(1),Integer(3)],[Integer(2),Integer(5)],
↪[Integer(4)]])
>>> G.duality_pairing(G[t1], F[t2])
0
>>> G.duality_pairing(G[t1], F[t1])
1
>>> G.duality_pairing(G[t2], F[t2])
1
>>> F.duality_pairing(F[t2], G[t2])
```

(continues on next page)

(continued from previous page)

```
1

>>> z = G[[Integer(1),Integer(3),Integer(5)],[Integer(2),Integer(4)]]
>>> all(F.duality_pairing(F[p1] * F[p2], z) == c
...     for ((p1, p2), c) in z.coproduct())
True
```

duality_pairing_matrix(*basis*, *degree*)

The matrix of scalar products between elements of $FSym$ and elements of $FSym^*$.

INPUT:

- basis – a basis of the dual Hopf algebra
- degree – nonnegative integer

OUTPUT:

- the matrix of scalar products between the basis self and the basis basis in the dual Hopf algebra of degree degree

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: G = FSym.G()
sage: G.duality_pairing_matrix(G.dual_basis(), 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> G = FSym.G()
>>> G.duality_pairing_matrix(G.dual_basis(), Integer(3))
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

one_basis()

Return the basis index corresponding to 1.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: TG = FSym.G()
sage: TG.one_basis()
[]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> TG = FSym.G()
>>> TG.one_basis()
[]
```

super_categories()

The super categories of self.

EXAMPLES:

```
sage: from sage.combinat.chas.fsym import FSymBases
sage: FSym = algebras.FSym(ZZ)
sage: bases = FSymBases(FSym)
sage: bases.super_categories()
[Category of realizations of Hopf algebra of standard tableaux over the
↪Integer Ring,
 Join of Category of realizations of Hopf algebras over Integer Ring
     and Category of graded algebras over Integer Ring
     and Category of graded coalgebras over Integer Ring,
 Category of graded connected Hopf algebras with basis over Integer Ring]
```

```
>>> from sage.all import *
>>> from sage.combinat.chas.fsym import FSymBases
>>> FSym = algebras.FSym(ZZ)
>>> bases = FSymBases(FSym)
>>> bases.super_categories()
[Category of realizations of Hopf algebra of standard tableaux over the
↪Integer Ring,
 Join of Category of realizations of Hopf algebras over Integer Ring
     and Category of graded algebras over Integer Ring
     and Category of graded coalgebras over Integer Ring,
 Category of graded connected Hopf algebras with basis over Integer Ring]
```

class sage.combinat.chas.fsym.**FSymBasis_abstract**(*alg*, *graded=True*)

Bases: *CombinatorialFreeModule*, BindableClass

Abstract base class for graded bases of $FSym$ and of $FSym^*$ indexed by standard tableaux.

This must define the following attributes:

- `_prefix` – the basis prefix

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: G = algebras.FSym(QQ).G()
sage: G.some_elements()
[G[], G[1], G[12], G[1] + G[1|2], G[] + 1/2*G[1]]
```

```
>>> from sage.all import *
>>> G = algebras.FSym(QQ).G()
>>> G.some_elements()
[G[], G[1], G[12], G[1] + G[1|2], G[] + 1/2*G[1]]
```

class sage.combinat.chas.fsym.**FreeSymmetricFunctions**(*base_ring*)

Bases: UniqueRepresentation, Parent

The free symmetric functions.

The *free symmetric functions* is a combinatorial Hopf algebra defined using tableaux and denoted $FSym$.

Consider the Hopf algebra $FQSym$ (*FreeQuasisymmetricFunctions*) over a commutative ring R, and its bases (F_w) and (G_w) (where w, in both cases, ranges over all permutations in all symmetric groups $S_0, S_1, S_2, \ldots$).

For each word w, let $P(w)$ be the P-tableau of w (that is, the first of the two tableaux obtained by applying the RSK algorithm to w; see `RSK()`). If t is a standard tableau of size n, then we define $\mathcal{G}_t \in FQSym$ to be the sum of the F_w with w ranging over all permutations of $\{1, 2, \ldots, n\}$ satisfying $P(w) = t$. Equivalently, $\mathcal{G}_t$ is the sum of the G_w with w ranging over all permutations of $\{1, 2, \ldots, n\}$ satisfying $Q(w) = t$ (where $Q(w)$ denotes the Q-tableau of w).

The R-linear span of the $\mathcal{G}_t$ (for t ranging over all standard tableaux) is a Hopf subalgebra of $FQSym$, denoted by $FSym$ and known as the *free symmetric functions* or the *Poirier-Reutenauer Hopf algebra of tableaux*. It has been introduced in [PoiReu95], where it was denoted by $(\mathbf{Z}T, *, \delta)$. (What we call $\mathcal{G}_t$ has just been called t in [PoiReu95].) The family $(\mathcal{G}_t)$ (with t ranging over all standard tableaux) is a basis of $FSym$, called the *Fundamental basis*.

EXAMPLES:

As explained above, $FSym$ is constructed as a Hopf subalgebra of $FQSym$:

```
sage: G = algebras.FSym(QQ).G()
sage: F = algebras.FQSym(QQ).F()
sage: G[[1,3],[2]]
G[13|2]
sage: G[[1,3],[2]].to_fqsym()
G[2, 1, 3] + G[3, 1, 2]
sage: F(G[[1,3],[2]])
F[2, 1, 3] + F[2, 3, 1]
```

```
>>> from sage.all import *
>>> G = algebras.FSym(QQ).G()
>>> F = algebras.FQSym(QQ).F()
>>> G[[Integer(1),Integer(3)],[Integer(2)]]
G[13|2]
>>> G[[Integer(1),Integer(3)],[Integer(2)]].to_fqsym()
G[2, 1, 3] + G[3, 1, 2]
>>> F(G[[Integer(1),Integer(3)],[Integer(2)]])
F[2, 1, 3] + F[2, 3, 1]
```

This embedding is a Hopf algebra morphism:

```
sage: all(F(G[t1] * G[t2]) == F(G[t1]) * F(G[t2])
....:     for t1 in StandardTableaux(2)
....:     for t2 in StandardTableaux(3))
True

sage: FF = F.tensor_square()
sage: all(FF(G[t].coproduct()) == F(G[t]).coproduct()
....:     for t in StandardTableaux(4))
True
```

```
>>> from sage.all import *
>>> all(F(G[t1] * G[t2]) == F(G[t1]) * F(G[t2])
...     for t1 in StandardTableaux(Integer(2))
...     for t2 in StandardTableaux(Integer(3)))
True

>>> FF = F.tensor_square()
>>> all(FF(G[t].coproduct()) == F(G[t]).coproduct()
```

(continues on next page)

(continued from previous page)

```
...     for t in StandardTableaux(Integer(4)))
True
```

There is a Hopf algebra map from $FSym$ onto the Hopf algebra of symmetric functions, which maps a tableau t to the Schur function indexed by the shape of t:

```
sage: TG = algebras.FSym(QQ).G()
sage: t = StandardTableau([[1,3],[2,4],[5]])
sage: TG[t]
G[13|24|5]
sage: TG[t].to_symmetric_function()
s[2, 2, 1]
```

```
>>> from sage.all import *
>>> TG = algebras.FSym(QQ).G()
>>> t = StandardTableau([[Integer(1),Integer(3)],[Integer(2),Integer(4)],
↪[Integer(5)]])
>>> TG[t]
G[13|24|5]
>>> TG[t].to_symmetric_function()
s[2, 2, 1]
```

class Fundamental(*alg*, *graded=True*)

Bases: *FSymBasis_abstract*

The Hopf algebra of tableaux on the Fundamental basis.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: TG = FSym.G()
sage: TG
Hopf algebra of standard tableaux over the Rational Field
 in the Fundamental basis
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> TG = FSym.G()
>>> TG
Hopf algebra of standard tableaux over the Rational Field
 in the Fundamental basis
```

Elements of the algebra look like:

```
sage: TG.an_element()
2*G[] + 2*G[1] + 3*G[12]
```

```
>>> from sage.all import *
>>> TG.an_element()
2*G[] + 2*G[1] + 3*G[12]
```

class Element

Bases: IndexedFreeModuleElement

to_fqsym()

Return the image of `self` under the natural inclusion map to $FQSym$.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: G = FSym.G()
sage: t = StandardTableau([[1,3],[2,4],[5]])
sage: G[t].to_fqsym()
G[2, 1, 5, 4, 3] + G[3, 1, 5, 4, 2] + G[3, 2, 5, 4, 1]
 + G[4, 1, 5, 3, 2] + G[4, 2, 5, 3, 1]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> G = FSym.G()
>>> t = StandardTableau([[Integer(1),Integer(3)],[Integer(2),
↪Integer(4)],[Integer(5)]])
>>> G[t].to_fqsym()
G[2, 1, 5, 4, 3] + G[3, 1, 5, 4, 2] + G[3, 2, 5, 4, 1]
 + G[4, 1, 5, 3, 2] + G[4, 2, 5, 3, 1]
```

to_symmetric_function()

Return the image of `self` under the natural projection map to Sym.

The natural projection map $FSym \to Sym$ sends each standard tableau t to the Schur function s_λ, where λ is the shape of t. This map is a surjective Hopf algebra homomorphism.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: G = FSym.G()
sage: t = StandardTableau([[1,3],[2,4],[5]])
sage: G[t].to_symmetric_function()
s[2, 2, 1]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> G = FSym.G()
>>> t = StandardTableau([[Integer(1),Integer(3)],[Integer(2),
↪Integer(4)],[Integer(5)]])
>>> G[t].to_symmetric_function()
s[2, 2, 1]
```

coproduct_on_basis(*t*)

Return the coproduct of the basis element indexed by `t`.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: G = FSym.G()
sage: t = StandardTableau([[1,2,5], [3,4]])
sage: G.coproduct_on_basis(t)
G[] # G[125|34] + G[1] # G[12|34] + G[1] # G[124|3]
 + G[1|2] # G[13|2] + G[12] # G[12|3] + G[12] # G[123]
```

(continues on next page)

(continued from previous page)

```
 + G[12|34] # G[1] + G[123] # G[12] + G[125|34] # G[]
 + G[13|2] # G[1|2] + G[13|2] # G[12] + G[134|2] # G[1]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> G = FSym.G()
>>> t = StandardTableau([[Integer(1),Integer(2),Integer(5)], [Integer(3),
↪Integer(4)]])
>>> G.coproduct_on_basis(t)
G[] # G[125|34] + G[1] # G[12|34] + G[1] # G[124|3]
 + G[1|2] # G[13|2] + G[12] # G[12|3] + G[12] # G[123]
 + G[12|34] # G[1] + G[123] # G[12] + G[125|34] # G[]
 + G[13|2] # G[1|2] + G[13|2] # G[12] + G[134|2] # G[1]
```

dual_basis()

Return the dual basis to `self`.

EXAMPLES:

```
sage: G = algebras.FSym(QQ).G()
sage: G.dual_basis()
Dual Hopf algebra of standard tableaux over the Rational Field
 in the FundamentalDual basis
```

```
>>> from sage.all import *
>>> G = algebras.FSym(QQ).G()
>>> G.dual_basis()
Dual Hopf algebra of standard tableaux over the Rational Field
 in the FundamentalDual basis
```

product_on_basis(*t1*, *t2*)

Return the product of basis elements indexed by `t1` and `t2`.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: G = FSym.G()
sage: t1 = StandardTableau([[1,2], [3]])
sage: t2 = StandardTableau([[1,2,3]])
sage: G.product_on_basis(t1, t2)
G[12456|3] + G[1256|3|4] + G[1256|34] + G[126|35|4]

sage: t1 = StandardTableau([[1],[2]])
sage: t2 = StandardTableau([[1,2]])
sage: G.product_on_basis(t1, t2)
G[134|2] + G[14|2|3]

sage: t1 = StandardTableau([[1,2],[3]])
sage: t2 = StandardTableau([[1],[2]])
sage: G.product_on_basis(t1, t2)
G[12|3|4|5] + G[12|34|5] + G[124|3|5] + G[124|35]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> G = FSym.G()
>>> t1 = StandardTableau([[Integer(1),Integer(2)], [Integer(3)]])
>>> t2 = StandardTableau([[Integer(1),Integer(2),Integer(3)]])
>>> G.product_on_basis(t1, t2)
G[12456|3] + G[1256|3|4] + G[1256|34] + G[126|35|4]

>>> t1 = StandardTableau([[Integer(1)],[Integer(2)]])
>>> t2 = StandardTableau([[Integer(1),Integer(2)]])
>>> G.product_on_basis(t1, t2)
G[134|2] + G[14|2|3]

>>> t1 = StandardTableau([[Integer(1),Integer(2)],[Integer(3)]])
>>> t2 = StandardTableau([[Integer(1)],[Integer(2)]])
>>> G.product_on_basis(t1, t2)
G[12|3|4|5] + G[12|34|5] + G[124|3|5] + G[124|35]
```

G

alias of *Fundamental*

a_realization()

Return a particular realization of `self` (the Fundamental basis).

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: FSym.a_realization()
Hopf algebra of standard tableaux over the Rational Field
 in the Fundamental basis
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> FSym.a_realization()
Hopf algebra of standard tableaux over the Rational Field
 in the Fundamental basis
```

dual()

Return the dual Hopf algebra of $FSym$.

EXAMPLES:

```
sage: algebras.FSym(QQ).dual()
Dual Hopf algebra of standard tableaux over the Rational Field
```

```
>>> from sage.all import *
>>> algebras.FSym(QQ).dual()
Dual Hopf algebra of standard tableaux over the Rational Field
```

class sage.combinat.chas.fsym.**FreeSymmetricFunctions_Dual**(*base_ring*)

Bases: `UniqueRepresentation`, `Parent`

The Hopf dual $FSym^*$ of the free symmetric functions $FSym$.

See *FreeSymmetricFunctions* for the definition of the latter.

Recall that the fundamental basis of $FSym$ consists of the elements $\mathcal{G}_t$ for t ranging over all standard tableaux. The dual basis of this is called the *dual fundamental basis* of $FSym^*$, and is denoted by $(\mathcal{G}_t^*)$. The Hopf dual $FSym^*$ is isomorphic to the Hopf algebra $(\mathbf{Z}T, *', \delta')$ from [PoiReu95]; the isomorphism sends a basis element $\mathcal{G}_t^*$ to t.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: TF = FSym.dual().F()
sage: TF[1,2] * TF[[1],[2]]
F[12|3|4] + F[123|4] + F[124|3] + F[13|2|4] + F[134|2] + F[14|2|3]
sage: TF[[1,2],[3]].coproduct()
F[] # F[12|3] + F[1] # F[1|2] + F[12] # F[1] + F[12|3] # F[]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> TF = FSym.dual().F()
>>> TF[Integer(1),Integer(2)] * TF[[Integer(1)],[Integer(2)]]
F[12|3|4] + F[123|4] + F[124|3] + F[13|2|4] + F[134|2] + F[14|2|3]
>>> TF[[Integer(1),Integer(2)],[Integer(3)]].coproduct()
F[] # F[12|3] + F[1] # F[1|2] + F[12] # F[1] + F[12|3] # F[]
```

The Hopf algebra $FSym^*$ is a Hopf quotient of $FQSym$; the canonical projection sends F_w (for a permutation w) to $\mathcal{G}_{Q(w)}^*$, where $Q(w)$ is the Q-tableau of w. This projection is implemented as a coercion:

```
sage: FQSym = algebras.FQSym(QQ)
sage: F = FQSym.F()
sage: TF(F[[1, 3, 2]])
F[12|3]
sage: TF(F[[5, 1, 4, 2, 3]])
F[135|2|4]
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(QQ)
>>> F = FQSym.F()
>>> TF(F[[Integer(1), Integer(3), Integer(2)]])
F[12|3]
>>> TF(F[[Integer(5), Integer(1), Integer(4), Integer(2), Integer(3)]])
F[135|2|4]
```

F

alias of *FundamentalDual*

class FundamentalDual(*alg*, *graded=True*)

Bases: *FSymBasis_abstract*

The dual to the Hopf algebra of tableaux, on the fundamental dual basis.

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: TF = FSym.dual().F()
sage: TF
Dual Hopf algebra of standard tableaux over the Rational Field
 in the FundamentalDual basis
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> TF = FSym.dual().F()
>>> TF
Dual Hopf algebra of standard tableaux over the Rational Field
 in the FundamentalDual basis
```

Elements of the algebra look like:

```
sage: TF.an_element()
2*F[] + 2*F[1] + 3*F[12]
```

```
>>> from sage.all import *
>>> TF.an_element()
2*F[] + 2*F[1] + 3*F[12]
```

class Element

Bases: `IndexedFreeModuleElement`

to_quasisymmetric_function()

Return the image of `self` under the canonical projection $FSym^* \to QSym$ to the ring of quasi-symmetric functions.

This projection is the adjoint of the canonical injection $NSym \to FSym$ (see `to_fsym()`). It sends each tableau t to the fundamental quasi-symmetric function F_α, where α is the descent composition of t.

EXAMPLES:

```
sage: F = algebras.FSym(QQ).dual().F()
sage: F[[1,3,5],[2,4]].to_quasisymmetric_function()
F[1, 2, 2]
```

```
>>> from sage.all import *
>>> F = algebras.FSym(QQ).dual().F()
>>> F[[Integer(1),Integer(3),Integer(5)],[Integer(2),Integer(4)]].to_
↪quasisymmetric_function()
F[1, 2, 2]
```

coproduct_on_basis(*t*)

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: TF = FSym.dual().F()
sage: t = StandardTableau([[1,2,5], [3,4]])
sage: TF.coproduct_on_basis(t)
F[] # F[125|34] + F[1] # F[134|2] + F[12] # F[123]
 + F[12|3] # F[12] + F[12|34] # F[1] + F[125|34] # F[]
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> TF = FSym.dual().F()
>>> t = StandardTableau([[Integer(1),Integer(2),Integer(5)], [Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(4)]])
>>> TF.coproduct_on_basis(t)
F[] # F[125|34] + F[1] # F[134|2] + F[12] # F[123]
 + F[12|3] # F[12] + F[12|34] # F[1] + F[125|34] # F[]
```

dual_basis()

Return the dual basis to `self`.

EXAMPLES:

```
sage: F = algebras.FSym(QQ).dual().F()
sage: F.dual_basis()
Hopf algebra of standard tableaux over the Rational Field
 in the Fundamental basis
```

```
>>> from sage.all import *
>>> F = algebras.FSym(QQ).dual().F()
>>> F.dual_basis()
Hopf algebra of standard tableaux over the Rational Field
 in the Fundamental basis
```

product_on_basis(*t1*, *t2*)

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ)
sage: TF = FSym.dual().F()
sage: t1 = StandardTableau([[1,2]])
sage: TF.product_on_basis(t1, t1)
F[12|34] + F[123|4] + F[1234] + F[124|3] + F[13|24] + F[134|2]
sage: t0 = StandardTableau([])
sage: TF.product_on_basis(t1, t0) == TF[t1] == TF.product_on_basis(t0, t1)
True
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ)
>>> TF = FSym.dual().F()
>>> t1 = StandardTableau([[Integer(1),Integer(2)]])
>>> TF.product_on_basis(t1, t1)
F[12|34] + F[123|4] + F[1234] + F[124|3] + F[13|24] + F[134|2]
>>> t0 = StandardTableau([])
>>> TF.product_on_basis(t1, t0) == TF[t1] == TF.product_on_basis(t0, t1)
True
```

a_realization()

Return a particular realization of `self` (the Fundamental dual basis).

EXAMPLES:

```
sage: FSym = algebras.FSym(QQ).dual()
sage: FSym.a_realization()
Dual Hopf algebra of standard tableaux over the Rational Field
 in the FundamentalDual basis
```

```
>>> from sage.all import *
>>> FSym = algebras.FSym(QQ).dual()
>>> FSym.a_realization()
Dual Hopf algebra of standard tableaux over the Rational Field
 in the FundamentalDual basis
```

dual()

Return the dual Hopf algebra of `self`, which is $FSym$.

EXAMPLES:

```
sage: D = algebras.FSym(QQ).dual()
sage: D.dual()
Hopf algebra of standard tableaux over the Rational Field
```

```
>>> from sage.all import *
>>> D = algebras.FSym(QQ).dual()
>>> D.dual()
Hopf algebra of standard tableaux over the Rational Field
```

sage.combinat.chas.fsym.**ascent_set**(*t*)

Return the ascent set of a standard tableau `t` (encoded as a sorted list).

The *ascent set* of a standard tableau t is defined as the set of all entries i of t such that the number $i + 1$ either appears to the right of i or appears in a row above i or does not appear in t at all.

EXAMPLES:

```
sage: from sage.combinat.chas.fsym import ascent_set
sage: t = StandardTableau([[1,3,4,7],[2,5,6],[8]])
sage: ascent_set(t)
[2, 3, 5, 6, 8]
sage: ascent_set(StandardTableau([]))
[]
sage: ascent_set(StandardTableau([[1, 2, 3]]))
[1, 2, 3]
sage: ascent_set(StandardTableau([[1, 2, 4], [3]]))
[1, 3, 4]
sage: ascent_set([[1, 3, 5], [2, 4]])
[2, 4, 5]
```

```
>>> from sage.all import *
>>> from sage.combinat.chas.fsym import ascent_set
>>> t = StandardTableau([[Integer(1),Integer(3),Integer(4),Integer(7)],
↪[Integer(2),Integer(5),Integer(6)],[Integer(8)]])
>>> ascent_set(t)
[2, 3, 5, 6, 8]
>>> ascent_set(StandardTableau([]))
[]
>>> ascent_set(StandardTableau([[Integer(1), Integer(2), Integer(3)]]))
[1, 2, 3]
>>> ascent_set(StandardTableau([[Integer(1), Integer(2), Integer(4)],␣
↪[Integer(3)]]))
[1, 3, 4]
```

(continues on next page)

(continued from previous page)

```
>>> ascent_set([[Integer(1), Integer(3), Integer(5)], [Integer(2), Integer(4)]])
[2, 4, 5]
```

sage.combinat.chas.fsym.**descent_composition**(*t*)

Return the descent composition of a standard tableau t.

This is the composition of the size of t whose partial sums are the elements of the descent set of t (see *descent_set()*).

EXAMPLES:

```
sage: from sage.combinat.chas.fsym import descent_composition
sage: t = StandardTableau([[1,3,4,7],[2,5,6],[8]])
sage: descent_composition(t)
[1, 3, 3, 1]
sage: descent_composition([[1, 3, 5], [2, 4]])
[1, 2, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.chas.fsym import descent_composition
>>> t = StandardTableau([[Integer(1),Integer(3),Integer(4),Integer(7)],
↪[Integer(2),Integer(5),Integer(6)],[Integer(8)]])
>>> descent_composition(t)
[1, 3, 3, 1]
>>> descent_composition([[Integer(1), Integer(3), Integer(5)], [Integer(2),␣
↪Integer(4)]])
[1, 2, 2]
```

sage.combinat.chas.fsym.**descent_set**(*t*)

Return the descent set of a standard tableau t (encoded as a sorted list).

The *descent set* of a standard tableau t is defined as the set of all entries i of t such that the number $i + 1$ appears in a row below i in t.

EXAMPLES:

```
sage: from sage.combinat.chas.fsym import descent_set
sage: t = StandardTableau([[1,3,4,7],[2,5,6],[8]])
sage: descent_set(t)
[1, 4, 7]
sage: descent_set(StandardTableau([]))
[]
sage: descent_set(StandardTableau([[1, 2, 3]]))
[]
sage: descent_set(StandardTableau([[1, 2, 4], [3]]))
[2]
sage: descent_set([[1, 3, 5], [2, 4]])
[1, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.chas.fsym import descent_set
>>> t = StandardTableau([[Integer(1),Integer(3),Integer(4),Integer(7)],
↪[Integer(2),Integer(5),Integer(6)],[Integer(8)]])
```

(continues on next page)

(continued from previous page)

```
>>> descent_set(t)
[1, 4, 7]
>>> descent_set(StandardTableau([]))
[]
>>> descent_set(StandardTableau([[Integer(1), Integer(2), Integer(3)]]))
[]
>>> descent_set(StandardTableau([[Integer(1), Integer(2), Integer(4)],␣
↪[Integer(3)]]))
[2]
>>> descent_set([[Integer(1), Integer(3), Integer(5)], [Integer(2), Integer(4)]])
[1, 3]
```

sage.combinat.chas.fsym.**standardize**(*t*)

Return the standard tableau corresponding to a given semistandard tableau `t` with no repeated entries.

> **Note**
>
> This is an optimized version of *Tableau.standardization()* for computations in $FSym$ by using the assumption of no repeated entries in `t`.

EXAMPLES:

```
sage: from sage.combinat.chas.fsym import standardize
sage: t = Tableau([[1,3,5,7],[2,4,8],[9]])
sage: standardize(t)
[[1, 3, 5, 6], [2, 4, 7], [8]]
sage: t = Tableau([[3,8,9,15],[5,10,12],[133]])
sage: standardize(t)
[[1, 3, 4, 7], [2, 5, 6], [8]]
```

```
>>> from sage.all import *
>>> from sage.combinat.chas.fsym import standardize
>>> t = Tableau([[Integer(1),Integer(3),Integer(5),Integer(7)],[Integer(2),
↪Integer(4),Integer(8)],[Integer(9)]])
>>> standardize(t)
[[1, 3, 5, 6], [2, 4, 7], [8]]
>>> t = Tableau([[Integer(3),Integer(8),Integer(9),Integer(15)],[Integer(5),
↪Integer(10),Integer(12)],[Integer(133)]])
>>> standardize(t)
[[1, 3, 4, 7], [2, 5, 6], [8]]
```

5.1.16 Word Quasi-symmetric functions

AUTHORS:

- Travis Scrimshaw (2018-04-09): initial implementation
- Darij Grinberg and Amy Pang (2018-04-12): further bases and methods

class sage.combinat.chas.wqsym.**WQSymBases**(*base*, *graded*)

Bases: `Category_realization_of_parent`

The category of bases of $WQSym$.

class ElementMethods

Bases: `object`

algebraic_complement()

Return the image of the element `self` of $WQSym$ under the algebraic complement involution.

If $u = (u_1, u_2, \ldots, u_n)$ is a packed word that contains the letters $1, 2, \ldots, k$ and no others, then the *complement* of u is defined to be the packed word $\overline{u} := (k + 1 - u_1, k + 1 - u_2, \ldots, k + 1 - u_n)$.

The algebraic complement involution is defined as the linear map $WQSym \to WQSym$ that sends each basis element $\mathbf{M}_u$ of the monomial basis of $WQSym$ to the basis element $\mathbf{M}_{\overline{u}}$. This is a graded algebra automorphism and a coalgebra anti-automorphism of $WQSym$. Denoting by $\overline{f}$ the image of an element $f \in WQSym$ under the algebraic complement involution, it can be shown that every packed word u satisfies

$$\overline{\mathbf{M}_u} = \mathbf{M}_{\overline{u}}, \qquad \overline{X_u} = X_{\overline{u}},$$

where standard notations for classical bases of $WQSym$ are being used (that is, $\mathbf{M}$ for the monomial basis, and X for the characteristic basis).

This can be restated in terms of ordered set partitions: For any ordered set partition $R = (R_1, R_2, \ldots, R_k)$, let R^r denote the ordered set partition $(R_k, R_{k-1}, \ldots, R_1)$; this is known as the *reversal* of R. Then,

$$\overline{\mathbf{M}_A} = \mathbf{M}_{A^r}, \qquad \overline{X_A} = X_{A^r}$$

for any ordered set partition A.

The formula describing algebraic complements on the Q basis (`WordQuasiSymmetricFunctions.StronglyCoarser`) is more complicated, and requires some definitions. We define a partial order $\leq$ on the set of all ordered set partitions as follows: $A \leq B$ if and only if A is strongly finer than B (see `is_strongly_finer()` for a definition of this). The *length* $\ell(R)$ of an ordered set partition R shall be defined as the number of parts of R. Use the notation Q for the Q basis. For any ordered set partition A of $[n]$, we have

$$\overline{Q_A} = \sum_P c_{A,P} Q_P,$$

where the sum is over all ordered set partitions P of $[n]$, and where the coefficient $c_{A,P}$ is defined as follows:

- If there exists an ordered set partition R satisfying $R \leq P$ and $A \leq R^r$, then this R is unique, and $c_{A,P} = (-1)^{\ell(R)-\ell(P)}$.
- If there exists no such R, then $c_{A,P} = 0$.

The formula describing algebraic complements on the Φ basis (`WordQuasiSymmetricFunctions.StronglyFiner`) is identical to the above formula for the Q basis, except that the $\leq$ sign has to be replaced by $\geq$ in the definition of the coefficients $c_{A,P}$. In fact, both formulas are particular cases of a general formula for involutions: Assume that V is an (additive) abelian group, and that I is a poset. For each $i \in I$, let M_i be an element of V. Also, let ω be an involution of the set I (not necessarily order-preserving or order-reversing), and let ω' be an involutive group endomorphism of V such that each $i \in I$ satisfies $\omega'(M_i) = M_{\omega(i)}$. For each $i \in I$, let $F_i = \sum_{j \geq i} M_j$, where we assume that the sum is finite. Then, each $i \in I$ satisfies

$$\omega'(F_i) = \sum_j \sum_{\substack{k \leq j; \\ \omega(k) \geq i}} \mu(k, j) F_j,$$

where μ denotes the Möbius function. This formula becomes particularly useful when the k satisfying $k \leq j$ and $\omega(k) \geq i$ is unique (if it exists). In our situation, V is $WQSym$, and I is the set of ordered set partitions equipped either with the $\leq$ partial order defined above or with its opposite order. The M_i is the $\mathbf{M}_A$, whereas the F_i is either Q_i or Φ_i.

If we denote the star involution (`star_involution()`) of the quasisymmetric functions by $f \mapsto f^*$, and if we let π be the canonical projection $WQSym \to QSym$, then each $f \in WQSym$ satisfies $\pi(\overline{f}) = (\pi(f))^*$.

See also

`coalgebraic_complement()`, `star_involution()`

EXAMPLES:

Recall that the index set for the bases of $WQSym$ is given by ordered set partitions, not packed words. Translated into the language of ordered set partitions, the algebraic complement involution acts on the Monomial basis by reversing the ordered set partition. In other words, we have

$$\overline{\mathbf{M}_{(P_1,P_2,\ldots,P_k)}} = \mathbf{M}_{(P_k,P_{k-1},\ldots,P_1)}$$

for any standard ordered set partition $(P_1, P_2, \ldots, P_k)$. Let us check this in practice:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: M = WQSym.M()
sage: M[[1,3],[2]].algebraic_complement()
M[{2}, {1, 3}]
sage: M[[1,4],[2,5],[3,6]].algebraic_complement()
M[{3, 6}, {2, 5}, {1, 4}]
sage: (3*M[[1]] - 4*M[[]] + 5*M[[1],[2]]).algebraic_complement()
-4*M[] + 3*M[{1}] + 5*M[{2}, {1}]
sage: X = WQSym.X()
sage: X[[1,3],[2]].algebraic_complement()
X[{2}, {1, 3}]
sage: C = WQSym.C()
sage: C[[1,3],[2]].algebraic_complement()
-C[{1, 2, 3}] - C[{1, 3}, {2}] + C[{2}, {1, 3}]
sage: Q = WQSym.Q()
sage: Q[[1,2],[5,6],[3,4]].algebraic_complement()
Q[{3, 4}, {1, 2, 5, 6}] + Q[{3, 4}, {5, 6}, {1, 2}] - Q[{3, 4, 5, 6}, {1,
→2}]
sage: Phi = WQSym.Phi()
sage: Phi[[2], [1,3]].algebraic_complement()
-Phi[{1}, {3}, {2}] + Phi[{1, 3}, {2}] + Phi[{3}, {1}, {2}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> M = WQSym.M()
>>> M[[Integer(1),Integer(3)],[Integer(2)]].algebraic_complement()
M[{2}, {1, 3}]
>>> M[[Integer(1),Integer(4)],[Integer(2),Integer(5)],[Integer(3),
→Integer(6)]].algebraic_complement()
M[{3, 6}, {2, 5}, {1, 4}]
>>> (Integer(3)*M[[Integer(1)]] - Integer(4)*M[[]] +
→Integer(5)*M[[Integer(1)],[Integer(2)]]).algebraic_complement()
-4*M[] + 3*M[{1}] + 5*M[{2}, {1}]
```

(continues on next page)

(continued from previous page)

```
>>> X = WQSym.X()
>>> X[[Integer(1),Integer(3)],[Integer(2)]].algebraic_complement()
X[{2}, {1, 3}]
>>> C = WQSym.C()
>>> C[[Integer(1),Integer(3)],[Integer(2)]].algebraic_complement()
-C[{1, 2, 3}] - C[{1, 3}, {2}] + C[{2}, {1, 3}]
>>> Q = WQSym.Q()
>>> Q[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].algebraic_complement()
Q[{3, 4}, {1, 2, 5, 6}] + Q[{3, 4}, {5, 6}, {1, 2}] - Q[{3, 4, 5, 6}, {1,␣
↪2}]
>>> Phi = WQSym.Phi()
>>> Phi[[Integer(2)], [Integer(1),Integer(3)]].algebraic_complement()
-Phi[{1}, {3}, {2}] + Phi[{1, 3}, {2}] + Phi[{3}, {1}, {2}]
```

The algebraic complement involution intertwines the antipode and the inverse of the antipode:

```
sage: all( M(I).antipode().algebraic_complement().antipode()  # long time
....:      == M(I).algebraic_complement()
....:      for I in OrderedSetPartitions(4) )
True
```

```
>>> from sage.all import *
>>> all( M(I).antipode().algebraic_complement().antipode()  # long time
...      == M(I).algebraic_complement()
...      for I in OrderedSetPartitions(Integer(4)) )
True
```

Testing the $\pi(\overline{f}) = (\pi(f))^*$ relation:

```
sage: all( M[I].algebraic_complement().to_quasisymmetric_function()
....:      == M[I].to_quasisymmetric_function().star_involution()
....:      for I in OrderedSetPartitions(4) )
True
```

```
>>> from sage.all import *
>>> all( M[I].algebraic_complement().to_quasisymmetric_function()
...      == M[I].to_quasisymmetric_function().star_involution()
...      for I in OrderedSetPartitions(Integer(4)) )
True
```

Todo

Check further commutative squares.

coalgebraic_complement()

Return the image of the element `self` of $WQSym$ under the coalgebraic complement involution.

If $u = (u_1, u_2, \ldots, u_n)$ is a packed word, then the *reversal* of u is defined to be the packed word $(u_n, u_{n-1}, \ldots, u_1)$. This reversal is denoted by u^r.

The coalgebraic complement involution is defined as the linear map $WQSym \to WQSym$ that sends each basis element $\mathbf{M}_u$ of the monomial basis of $WQSym$ to the basis element $\mathbf{M}_{u^r}$. This is a graded coalgebra automorphism and an algebra anti-automorphism of $WQSym$. Denoting by f^r the image of an element $f \in WQSym$ under the coalgebraic complement involution, it can be shown that every packed word u satisfies

$$(\mathbf{M}_u)^r = \mathbf{M}_{u^r}, \qquad (X_u)^r = X_{u^r},$$

where standard notations for classical bases of $WQSym$ are being used (that is, $\mathbf{M}$ for the monomial basis, and X for the characteristic basis).

This can be restated in terms of ordered set partitions: For any ordered set partition R of $[n]$, let $\overline{R}$ denote the complement of R (defined in `complement()`). Then,

$$(\mathbf{M}_A)^r = \mathbf{M}_{\overline{A}}, \qquad (X_A)^r = X_{\overline{A}}$$

for any ordered set partition A.

Recall that $WQSym$ is a subring of the ring of all bounded-degree noncommutative power series in countably many indeterminates. The latter ring has an obvious continuous algebra anti-endomorphism which sends each letter x_i to x_i (and thus sends each monomial $x_{i_1} x_{i_2} \cdots x_{i_n}$ to $x_{i_n} x_{i_{n-1}} \cdots x_{i_1}$). This anti-endomorphism is actually an involution. The coalgebraic complement involution is simply the restriction of this involution to the subring $WQSym$.

The formula describing coalgebraic complements on the Q basis (`WordQuasiSymmetricFunctions.StronglyCoarser`) is more complicated, and requires some definitions. We define a partial order $\leq$ on the set of all ordered set partitions as follows: $A \leq B$ if and only if A is strongly finer than B (see `is_strongly_finer()` for a definition of this). The *length* $\ell(R)$ of an ordered set partition R shall be defined as the number of parts of R. Use the notation Q for the Q basis. For any ordered set partition A of $[n]$, we have

$$(Q_A)^r = \sum_P c_{A,P} Q_P,$$

where the sum is over all ordered set partitions P of $[n]$, and where the coefficient $c_{A,P}$ is defined as follows:

- If there exists an ordered set partition R satisfying $R \leq P$ and $A \leq \overline{R}$, then this R is unique, and $c_{A,P} = (-1)^{\ell(R) - \ell(P)}$.
- If there exists no such R, then $c_{A,P} = 0$.

The formula describing coalgebraic complements on the Φ basis (`WordQuasiSymmetricFunctions.StronglyFiner`) is identical to the above formula for the Q basis, except that the $\leq$ sign has to be replaced by $\geq$ in the definition of the coefficients $c_{A,P}$. In fact, both formulas are particular cases of the general formula for involutions described in the documentation of `algebraic_complement()`.

If we let π be the canonical projection $WQSym \to QSym$, then each $f \in WQSym$ satisfies $\pi(f^r) = \pi(f)$.

See also:

`algebraic_complement()`, `star_involution()`

EXAMPLES:

Recall that the index set for the bases of $WQSym$ is given by ordered set partitions, not packed words. Translated into the language of ordered set partitions, the coalgebraic complement involution acts on the Monomial basis by complementing the ordered set partition. In other words, we have

$$(\mathbf{M}_A)^r = \mathbf{M}_{\overline{A}}$$

for any standard ordered set partition P. Let us check this in practice:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: M = WQSym.M()
sage: M[[1,3],[2]].coalgebraic_complement()
M[{1, 3}, {2}]
sage: M[[1,2],[3]].coalgebraic_complement()
M[{2, 3}, {1}]
sage: M[[1], [4], [2,3]].coalgebraic_complement()
M[{4}, {1}, {2, 3}]
sage: M[[1,4],[2,5],[3,6]].coalgebraic_complement()
M[{3, 6}, {2, 5}, {1, 4}]
sage: (3*M[[1]] - 4*M[[]] + 5*M[[1],[2]]).coalgebraic_complement()
-4*M[] + 3*M[{1}] + 5*M[{2}, {1}]
sage: X = WQSym.X()
sage: X[[1,3],[2]].coalgebraic_complement()
X[{1, 3}, {2}]
sage: C = WQSym.C()
sage: C[[1,3],[2]].coalgebraic_complement()
C[{1, 3}, {2}]
sage: Q = WQSym.Q()
sage: Q[[1,2],[5,6],[3,4]].coalgebraic_complement()
Q[{1, 2, 5, 6}, {3, 4}] + Q[{5, 6}, {1, 2}, {3, 4}] - Q[{5, 6}, {1, 2, 3,␣
↪4}]
sage: Phi = WQSym.Phi()
sage: Phi[[2], [1,3]].coalgebraic_complement()
-Phi[{2}, {1}, {3}] + Phi[{2}, {1, 3}] + Phi[{2}, {3}, {1}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> M = WQSym.M()
>>> M[[Integer(1),Integer(3)],[Integer(2)]].coalgebraic_complement()
M[{1, 3}, {2}]
>>> M[[Integer(1),Integer(2)],[Integer(3)]].coalgebraic_complement()
M[{2, 3}, {1}]
>>> M[[Integer(1)], [Integer(4)], [Integer(2),Integer(3)]].coalgebraic_
↪complement()
M[{4}, {1}, {2, 3}]
>>> M[[Integer(1),Integer(4)],[Integer(2),Integer(5)],[Integer(3),
↪Integer(6)]].coalgebraic_complement()
M[{3, 6}, {2, 5}, {1, 4}]
>>> (Integer(3)*M[[Integer(1)]] - Integer(4)*M[[]] +␣
↪Integer(5)*M[[Integer(1)],[Integer(2)]]).coalgebraic_complement()
-4*M[] + 3*M[{1}] + 5*M[{2}, {1}]
>>> X = WQSym.X()
>>> X[[Integer(1),Integer(3)],[Integer(2)]].coalgebraic_complement()
X[{1, 3}, {2}]
>>> C = WQSym.C()
>>> C[[Integer(1),Integer(3)],[Integer(2)]].coalgebraic_complement()
C[{1, 3}, {2}]
>>> Q = WQSym.Q()
>>> Q[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].coalgebraic_complement()
Q[{1, 2, 5, 6}, {3, 4}] + Q[{5, 6}, {1, 2}, {3, 4}] - Q[{5, 6}, {1, 2, 3,␣
```

(continues on next page)

(continued from previous page)

```
↪4}]
>>> Phi = WQSym.Phi()
>>> Phi[[Integer(2)], [Integer(1),Integer(3)]].coalgebraic_complement()
-Phi[{2}, {1}, {3}] + Phi[{2}, {1, 3}] + Phi[{2}, {3}, {1}]
```

The coalgebraic complement involution intertwines the antipode and the inverse of the antipode:

```
sage: all( M(I).antipode().coalgebraic_complement().antipode()  # long␣
↪time
....:      == M(I).coalgebraic_complement()
....:      for I in OrderedSetPartitions(4) )
True
```

```
>>> from sage.all import *
>>> all( M(I).antipode().coalgebraic_complement().antipode()  # long time
...      == M(I).coalgebraic_complement()
...      for I in OrderedSetPartitions(Integer(4)) )
True
```

Testing the $\pi(f^r) = \pi(f)$ relation above:

```
sage: all( M[I].coalgebraic_complement().to_quasisymmetric_function()
....:      == M[I].to_quasisymmetric_function()
....:      for I in OrderedSetPartitions(4) )
True
```

```
>>> from sage.all import *
>>> all( M[I].coalgebraic_complement().to_quasisymmetric_function()
...      == M[I].to_quasisymmetric_function()
...      for I in OrderedSetPartitions(Integer(4)) )
True
```

Todo

Check further commutative squares.

star_involution()

Return the image of the element `self` of $WQSym$ under the star involution.

The star involution is the composition of the algebraic complement involution (*algebraic_complement()*) with the coalgebraic complement involution (*coalgebraic_complement()*). The composition can be performed in either order, as the involutions commute.

The star involution is a graded Hopf algebra anti-automorphism of $WQSym$. Let f^* denote the image of an element $f \in WQSym$ under the star involution. Let $\mathbf{M}$, X, Q and Φ stand for the monomial, characteristic, Q and Phi bases of $WQSym$. For any ordered set partition A of $[n]$, we let A^* denote the complement (*complement()*) of the reversal (*reversed()*) of A. Then, for any ordered set partition A of $[n]$, we have

$$(\mathbf{M}_A)^* = \mathbf{M}_{A^*}, \qquad (X_A)^* = X_{A^*}, \qquad (Q_A)^* = Q_{A^*}, \qquad (\Phi_A)^* = \Phi_{A^*}.$$

The star involution (*star_involution()*) on the ring of noncommutative symmetric functions is a restriction of the star involution on $WQSym$.

If we denote the star involution (*star_involution()*) of the quasisymmetric functions by $f \mapsto f^*$, and if we let π be the canonical projection $WQSym \to QSym$, then each $f \in WQSym$ satisfies $\pi(f^*) = (\pi(f))^*$.

Todo

More commutative diagrams? FQSym and FSym need their own star_involution methods defined first.

See also

algebraic_complement(), *coalgebraic_complement()*

EXAMPLES:

Keep in mind that the default input method for basis keys of $WQSym$ is by entering an ordered set partition, not a packed word. Let us check the basis formulas for the star involution:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: M = WQSym.M()
sage: M[[1,3], [2,4,5]].star_involution()
M[{1, 2, 4}, {3, 5}]
sage: M[[1,3],[2]].star_involution()
M[{2}, {1, 3}]
sage: M[[1,4],[2,5],[3,6]].star_involution()
M[{1, 4}, {2, 5}, {3, 6}]
sage: (3*M[[1]] - 4*M[[]] + 5*M[[1],[2]]).star_involution()
-4*M[] + 3*M[{1}] + 5*M[{1}, {2}]
sage: X = WQSym.X()
sage: X[[1,3],[2]].star_involution()
X[{2}, {1, 3}]
sage: C = WQSym.C()
sage: C[[1,3],[2]].star_involution()
-C[{1, 2, 3}] - C[{1, 3}, {2}] + C[{2}, {1, 3}]
sage: Q = WQSym.Q()
sage: Q[[1,3], [2,4,5]].star_involution()
Q[{1, 2, 4}, {3, 5}]
sage: Phi = WQSym.Phi()
sage: Phi[[1,3], [2,4,5]].star_involution()
Phi[{1, 2, 4}, {3, 5}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> M = WQSym.M()
>>> M[[Integer(1),Integer(3)], [Integer(2),Integer(4),Integer(5)]].star_
↪involution()
M[{1, 2, 4}, {3, 5}]
>>> M[[Integer(1),Integer(3)],[Integer(2)]].star_involution()
M[{2}, {1, 3}]
>>> M[[Integer(1),Integer(4)],[Integer(2),Integer(5)],[Integer(3),
↪Integer(6)]].star_involution()
```

(continues on next page)

(continued from previous page)

```
M[{1, 4}, {2, 5}, {3, 6}]
>>> (Integer(3)*M[[Integer(1)]] - Integer(4)*M[[]] +␣
↪Integer(5)*M[[Integer(1)],[Integer(2)]]).star_involution()
-4*M[] + 3*M[{1}] + 5*M[{1}, {2}]
>>> X = WQSym.X()
>>> X[[Integer(1),Integer(3)],[Integer(2)]].star_involution()
X[{2}, {1, 3}]
>>> C = WQSym.C()
>>> C[[Integer(1),Integer(3)],[Integer(2)]].star_involution()
-C[{1, 2, 3}] - C[{1, 3}, {2}] + C[{2}, {1, 3}]
>>> Q = WQSym.Q()
>>> Q[[Integer(1),Integer(3)], [Integer(2),Integer(4),Integer(5)]].star_
↪involution()
Q[{1, 2, 4}, {3, 5}]
>>> Phi = WQSym.Phi()
>>> Phi[[Integer(1),Integer(3)], [Integer(2),Integer(4),Integer(5)]].star_
↪involution()
Phi[{1, 2, 4}, {3, 5}]
```

Testing the formulas for $(Q_A)^*$ and $(\Phi_A)^*$:

```
sage: all(Q[A].star_involution() == Q[A.complement().reversed()] for A in␣
↪OrderedSetPartitions(4))
True
sage: all(Phi[A].star_involution() == Phi[A.complement().reversed()] for␣
↪A in OrderedSetPartitions(4))
True
```

```
>>> from sage.all import *
>>> all(Q[A].star_involution() == Q[A.complement().reversed()] for A in␣
↪OrderedSetPartitions(Integer(4)))
True
>>> all(Phi[A].star_involution() == Phi[A.complement().reversed()] for A␣
↪in OrderedSetPartitions(Integer(4)))
True
```

The star involution commutes with the antipode:

```
sage: all( M(I).antipode().star_involution()   # long time
....:      == M(I).star_involution().antipode()
....:      for I in OrderedSetPartitions(4) )
True
```

```
>>> from sage.all import *
>>> all( M(I).antipode().star_involution()   # long time
...      == M(I).star_involution().antipode()
...      for I in OrderedSetPartitions(Integer(4)) )
True
```

Testing the $\pi(f^*) = (\pi(f))^*$ relation:

```
sage: all( M[I].star_involution().to_quasisymmetric_function()
....:      == M[I].to_quasisymmetric_function().star_involution()
....:      for I in OrderedSetPartitions(4) )
True
```

```
>>> from sage.all import *
>>> all( M[I].star_involution().to_quasisymmetric_function()
...      == M[I].to_quasisymmetric_function().star_involution()
...      for I in OrderedSetPartitions(Integer(4)) )
True
```

Testing the fact that the star involution on the noncommutative symmetric functions is a restriction of the star involution on $WQSym$:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: R = NCSF.R()
sage: all(R[I].star_involution().to_fqsym().to_wqsym()
....:     == R[I].to_fqsym().to_wqsym().star_involution()
....:     for I in Compositions(4))
True
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> R = NCSF.R()
>>> all(R[I].star_involution().to_fqsym().to_wqsym()
...     == R[I].to_fqsym().to_wqsym().star_involution()
...     for I in Compositions(Integer(4)))
True
```

Todo

Check further commutative squares.

to_quasisymmetric_function()

The projection of `self` to the ring $QSym$ of quasisymmetric functions.

There is a canonical projection $\pi : WQSym \to QSym$ that sends every element $\mathbf{M}_P$ of the monomial basis of $WQSym$ to the monomial quasisymmetric function M_c, where c is the composition whose parts are the sizes of the blocks of P. This π is a ring homomorphism.

OUTPUT: an element of the quasisymmetric functions in the monomial basis

EXAMPLES:

```
sage: M = algebras.WQSym(QQ).M()
sage: M[[1,3],[2]].to_quasisymmetric_function()
M[2, 1]
sage: (M[[1,3],[2]] + 3*M[[2,3],[1]] - M[[1,2,3],]).to_quasisymmetric_
↪function()
4*M[2, 1] - M[3]
sage: X, Y = M[[1,3],[2]], M[[1,2,3],]
sage: X.to_quasisymmetric_function() * Y.to_quasisymmetric_function() ==␣
```

(continues on next page)

(continued from previous page)

```
↪(X*Y).to_quasisymmetric_function()
True

sage: C = algebras.WQSym(QQ).C()
sage: C[[2,3],[1,4]].to_quasisymmetric_function() == M(C[[2,3],[1,4]]).to_
↪quasisymmetric_function()
True

sage: C2 = algebras.WQSym(GF(2)).C()
sage: C2[[1,2],[3,4]].to_quasisymmetric_function()
M[2, 2]
sage: C2[[2,3],[1,4]].to_quasisymmetric_function()
M[4]
```

```
>>> from sage.all import *
>>> M = algebras.WQSym(QQ).M()
>>> M[[Integer(1),Integer(3)],[Integer(2)]].to_quasisymmetric_function()
M[2, 1]
>>> (M[[Integer(1),Integer(3)],[Integer(2)]] + Integer(3)*M[[Integer(2),
↪Integer(3)],[Integer(1)]] - M[[Integer(1),Integer(2),Integer(3)],]).to_
↪quasisymmetric_function()
4*M[2, 1] - M[3]
>>> X, Y = M[[Integer(1),Integer(3)],[Integer(2)]], M[[Integer(1),
↪Integer(2),Integer(3)],]
>>> X.to_quasisymmetric_function() * Y.to_quasisymmetric_function() ==␣
↪(X*Y).to_quasisymmetric_function()
True

>>> C = algebras.WQSym(QQ).C()
>>> C[[Integer(2),Integer(3)],[Integer(1),Integer(4)]].to_quasisymmetric_
↪function() == M(C[[Integer(2),Integer(3)],[Integer(1),Integer(4)]]).to_
↪quasisymmetric_function()
True

>>> C2 = algebras.WQSym(GF(Integer(2))).C()
>>> C2[[Integer(1),Integer(2)],[Integer(3),Integer(4)]].to_quasisymmetric_
↪function()
M[2, 2]
>>> C2[[Integer(2),Integer(3)],[Integer(1),Integer(4)]].to_quasisymmetric_
↪function()
M[4]
```

class ParentMethods

Bases: `object`

degree_on_basis(*t*)

Return the degree of an ordered set partition in the algebra of word quasi-symmetric functions.

This is the sum of the sizes of the blocks of the ordered set partition.

EXAMPLES:

```
sage: A = algebras.WQSym(QQ).M()
sage: u = OrderedSetPartition([[2,1],])
sage: A.degree_on_basis(u)
2
sage: u = OrderedSetPartition([[2], [1]])
sage: A.degree_on_basis(u)
2
```

```
>>> from sage.all import *
>>> A = algebras.WQSym(QQ).M()
>>> u = OrderedSetPartition([[Integer(2),Integer(1)],])
>>> A.degree_on_basis(u)
2
>>> u = OrderedSetPartition([[Integer(2)], [Integer(1)]])
>>> A.degree_on_basis(u)
2
```

is_commutative()

Return whether `self` is commutative.

EXAMPLES:

```
sage: M = algebras.WQSym(ZZ).M()
sage: M.is_commutative()
False
```

```
>>> from sage.all import *
>>> M = algebras.WQSym(ZZ).M()
>>> M.is_commutative()
False
```

is_field(*proof=True*)

Return whether `self` is a field.

EXAMPLES:

```
sage: M = algebras.WQSym(QQ).M()
sage: M.is_field()
False
```

```
>>> from sage.all import *
>>> M = algebras.WQSym(QQ).M()
>>> M.is_field()
False
```

one_basis()

Return the index of the unit.

EXAMPLES:

```
sage: A = algebras.WQSym(QQ).M()
sage: A.one_basis()
[]
```

```
>>> from sage.all import *
>>> A = algebras.WQSym(QQ).M()
>>> A.one_basis()
[]
```

super_categories()

The super categories of `self`.

EXAMPLES:

```
sage: from sage.combinat.chas.wqsym import WQSymBases
sage: WQSym = algebras.WQSym(ZZ)
sage: bases = WQSymBases(WQSym, True)
sage: bases.super_categories()
[Category of realizations of Word Quasi-symmetric functions over Integer Ring,
 Join of Category of realizations of Hopf algebras over Integer Ring
     and Category of graded algebras over Integer Ring
     and Category of graded coalgebras over Integer Ring,
 Category of graded connected Hopf algebras with basis over Integer Ring]

sage: bases = WQSymBases(WQSym, False)
sage: bases.super_categories()
[Category of realizations of Word Quasi-symmetric functions over Integer Ring,
 Join of Category of realizations of Hopf algebras over Integer Ring
     and Category of graded algebras over Integer Ring
     and Category of graded coalgebras over Integer Ring,
 Join of Category of filtered connected Hopf algebras with basis over Integer␣
↪Ring
     and Category of graded algebras over Integer Ring
     and Category of graded coalgebras over Integer Ring]
```

```
>>> from sage.all import *
>>> from sage.combinat.chas.wqsym import WQSymBases
>>> WQSym = algebras.WQSym(ZZ)
>>> bases = WQSymBases(WQSym, True)
>>> bases.super_categories()
[Category of realizations of Word Quasi-symmetric functions over Integer Ring,
 Join of Category of realizations of Hopf algebras over Integer Ring
     and Category of graded algebras over Integer Ring
     and Category of graded coalgebras over Integer Ring,
 Category of graded connected Hopf algebras with basis over Integer Ring]

>>> bases = WQSymBases(WQSym, False)
>>> bases.super_categories()
[Category of realizations of Word Quasi-symmetric functions over Integer Ring,
 Join of Category of realizations of Hopf algebras over Integer Ring
     and Category of graded algebras over Integer Ring
     and Category of graded coalgebras over Integer Ring,
 Join of Category of filtered connected Hopf algebras with basis over Integer␣
↪Ring
     and Category of graded algebras over Integer Ring
     and Category of graded coalgebras over Integer Ring]
```

class sage.combinat.chas.wqsym.**WQSymBasis_abstract**(*alg*, *graded=True*)

Bases: *CombinatorialFreeModule*, BindableClass

Abstract base class for bases of $WQSym$.

This must define two attributes:

- `_prefix` – the basis prefix
- `_basis_name` – the name of the basis (must match one of the names that the basis can be constructed from $WQSym$)

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: M = algebras.WQSym(QQ).M()
sage: M.an_element()
M[{1}] + 2*M[{1}, {2}]
```

```
>>> from sage.all import *
>>> M = algebras.WQSym(QQ).M()
>>> M.an_element()
M[{1}] + 2*M[{1}, {2}]
```

options = Current options for WordQuasiSymmetricFunctions element - display: normal - objects: compositions

some_elements()

Return some elements of the word quasi-symmetric functions.

EXAMPLES:

```
sage: M = algebras.WQSym(QQ).M()
sage: M.some_elements()
[M[], M[{1}], M[{1, 2}],
 M[{1}] + M[{1}, {2}],
 M[] + 1/2*M[{1}]]
```

```
>>> from sage.all import *
>>> M = algebras.WQSym(QQ).M()
>>> M.some_elements()
[M[], M[{1}], M[{1, 2}],
 M[{1}] + M[{1}, {2}],
 M[] + 1/2*M[{1}]]
```

class sage.combinat.chas.wqsym.**WordQuasiSymmetricFunctions**(R)

Bases: UniqueRepresentation, Parent

The word quasi-symmetric functions.

The ring of word quasi-symmetric functions can be defined as a subring of the ring of all bounded-degree noncommutative power series in countably many indeterminates (i.e., elements in $R\langle\langle x_1, x_2, x_3, \ldots \rangle\rangle$ of bounded degree). Namely, consider words over the alphabet $\{1, 2, 3, \ldots\}$; every noncommutative power series is an infinite R-linear combination of these words. For each such word w, we define the *packing* of w to be the word $\mathrm{pack}(w)$ that is obtained from w by replacing the smallest letter that appears in w by 1, the second-smallest letter that appears in w by 2, etc. (for example, $\mathrm{pack}(4112774) = 3112443$). A word w is said to be *packed* if $\mathrm{pack}(w) = w$. For each packed word u, we define the noncommutative power series $\mathbf{M}_u = \sum w$, where the sum ranges over all words w satisfying $\mathrm{pack}(w) = u$. The span of these power series $\mathbf{M}_u$ is a subring of the ring of all noncommutative power series; it is called the ring of word quasi-symmetric functions, and is denoted by $WQSym$.

For each nonnegative integer n, there is a bijection between packed words of length n and ordered set partitions of $\{1, 2, \ldots, n\}$. Under this bijection, a packed word $u = (u_1, u_2, \ldots, u_n)$ of length n corresponds to the ordered set partition $P = (P_1, P_2, \ldots, P_k)$ of $\{1, 2, \ldots, n\}$ whose i-th part P_i (for each i) is the set of all $j \in \{1, 2, \ldots, n\}$ such that $u_j = i$.

The basis element $\mathbf{M}_u$ is also denoted as $\mathbf{M}_P$ in this situation. The basis $(\mathbf{M}_P)_P$ is called the *Monomial basis* and is implemented as `Monomial`.

Other bases are the cone basis (aka C basis), the characteristic basis (aka X basis), the Q basis and the Phi basis.

Bases of $WQSym$ are implemented (internally) using ordered set partitions. However, the user may access specific basis vectors using either packed words or ordered set partitions. See the examples below, noting especially the section on ambiguities.

$WQSym$ is endowed with a connected graded Hopf algebra structure (see Section 2.2 of [NoThWi08], Section 1.1 of [FoiMal14] and Section 4.3.2 of [MeNoTh11]) given by

$$\Delta(\mathbf{M}_{(P_1,\ldots,P_\ell)}) = \sum_{i=0}^{\ell} \mathbf{M}_{\mathrm{st}(P_1,\ldots,P_i)} \otimes \mathbf{M}_{\mathrm{st}(P_{i+1},\ldots,P_\ell)}.$$

Here, for any ordered set partition $(Q_1, \ldots, Q_k)$ of a finite set Z of integers, we let $\mathrm{st}(Q_1, \ldots, Q_k)$ denote the set partition obtained from Z by replacing the smallest element appearing in it by 1, the second-smallest element by 2, and so on.

A rule for multiplying elements of the monomial basis relies on the *quasi-shuffle product* of two ordered set partitions. The quasi-shuffle product $\Box$ is given by `ShuffleProduct_overlapping` with the + operation in the overlapping of the shuffles being the union of the sets. The product $\mathbf{M}_P \mathbf{M}_Q$ for two ordered set partitions P and Q of $[n]$ and $[m]$ is then given by

$$\mathbf{M}_P \mathbf{M}_Q = \sum_{R \in P \Box Q^+} \mathbf{M}_R,$$

where Q^+ means Q with all numbers shifted upwards by n.

Sometimes, $WQSym$ is also denoted as $NCQSym$.

REFERENCES:

- [FoiMal14]
- [MeNoTh11]
- [NoThWi08]
- [BerZab05]

EXAMPLES:

Constructing the algebra and its Monomial basis:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: WQSym
Word Quasi-symmetric functions over Integer Ring
sage: M = WQSym.M()
sage: M
Word Quasi-symmetric functions over Integer Ring in the Monomial basis
sage: M[[]]
M[]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> WQSym
Word Quasi-symmetric functions over Integer Ring
>>> M = WQSym.M()
>>> M
Word Quasi-symmetric functions over Integer Ring in the Monomial basis
>>> M[[]]
M[]
```

Calling basis elements using packed words:

```
sage: x = M[1,2,1]; x
M[{1, 3}, {2}]
sage: x == M[[1,2,1]] == M[Word([1,2,1])]
True
sage: y = M[1,1,2] - M[1,2,2]; y
-M[{1}, {2, 3}] + M[{1, 2}, {3}]
```

```
>>> from sage.all import *
>>> x = M[Integer(1),Integer(2),Integer(1)]; x
M[{1, 3}, {2}]
>>> x == M[[Integer(1),Integer(2),Integer(1)]] == M[Word([Integer(1),Integer(2),
↪Integer(1)])]
True
>>> y = M[Integer(1),Integer(1),Integer(2)] - M[Integer(1),Integer(2),Integer(2)];
↪ y
-M[{1}, {2, 3}] + M[{1, 2}, {3}]
```

Calling basis elements using ordered set partitions:

```
sage: z = M[[1,2,3],]; z
M[{1, 2, 3}]
sage: z == M[[[1,2,3]]] == M[OrderedSetPartition([[1,2,3]])]
True
sage: M[[1,2],[3]]
M[{1, 2}, {3}]
```

```
>>> from sage.all import *
>>> z = M[[Integer(1),Integer(2),Integer(3)],]; z
M[{1, 2, 3}]
>>> z == M[[[Integer(1),Integer(2),Integer(3)]]] ==␣
↪M[OrderedSetPartition([[Integer(1),Integer(2),Integer(3)]])]
True
>>> M[[Integer(1),Integer(2)],[Integer(3)]]
M[{1, 2}, {3}]
```

Note that expressions above are output in terms of ordered set partitions, even when input as packed words. Output as packed words can be achieved by modifying the global options. (See `OrderedSetPartitions.options()` for further details.):

```
sage: M.options.objects = "words"
sage: y
```

(continues on next page)

(continued from previous page)

```
-M[1, 2, 2] + M[1, 1, 2]
sage: M.options.display = "compact"
sage: y
-M[122] + M[112]
sage: z
M[111]
```

```
>>> from sage.all import *
>>> M.options.objects = "words"
>>> y
-M[1, 2, 2] + M[1, 1, 2]
>>> M.options.display = "compact"
>>> y
-M[122] + M[112]
>>> z
M[111]
```

The options should be reset to display as ordered set partitions:

```
sage: M.options._reset()
sage: z
M[{1, 2, 3}]
```

```
>>> from sage.all import *
>>> M.options._reset()
>>> z
M[{1, 2, 3}]
```

Illustration of the Hopf algebra structure:

```
sage: M[[2, 3], [5], [6], [4], [1]].coproduct()
M[] # M[{2, 3}, {5}, {6}, {4}, {1}] + M[{1, 2}] # M[{3}, {4}, {2}, {1}]
 + M[{1, 2}, {3}] # M[{3}, {2}, {1}] + M[{1, 2}, {3}, {4}] # M[{2}, {1}]
 + M[{1, 2}, {4}, {5}, {3}] # M[{1}] + M[{2, 3}, {5}, {6}, {4}, {1}] # M[]
sage: _ == M[5,1,1,4,2,3].coproduct()
True
sage: M[[1,1,1]] * M[[1,1,2]]   # packed words
M[{1, 2, 3}, {4, 5}, {6}] + M[{1, 2, 3, 4, 5}, {6}]
 + M[{4, 5}, {1, 2, 3}, {6}] + M[{4, 5}, {1, 2, 3, 6}]
 + M[{4, 5}, {6}, {1, 2, 3}]
sage: M[[1,2,3],].antipode()  # ordered set partition
-M[{1, 2, 3}]
sage: M[[1], [2], [3]].antipode()
-M[{1, 2, 3}] - M[{2, 3}, {1}] - M[{3}, {1, 2}] - M[{3}, {2}, {1}]
sage: x = M[[1],[2],[3]] + 3*M[[2],[1]]
sage: x.counit()
0
sage: x.antipode()
3*M[{1}, {2}] + 3*M[{1, 2}] - M[{1, 2, 3}] - M[{2, 3}, {1}]
 - M[{3}, {1, 2}] - M[{3}, {2}, {1}]
```

```
>>> from sage.all import *
>>> M[[Integer(2), Integer(3)], [Integer(5)], [Integer(6)], [Integer(4)],␣
↪[Integer(1)]].coproduct()
M[] # M[{2, 3}, {5}, {6}, {4}, {1}] + M[{1, 2}] # M[{3}, {4}, {2}, {1}]
 + M[{1, 2}, {3}] # M[{3}, {2}, {1}] + M[{1, 2}, {3}, {4}] # M[{2}, {1}]
 + M[{1, 2}, {4}, {5}, {3}] # M[{1}] + M[{2, 3}, {5}, {6}, {4}, {1}] # M[]
>>> _ == M[Integer(5),Integer(1),Integer(1),Integer(4),Integer(2),Integer(3)].
↪coproduct()
True
>>> M[[Integer(1),Integer(1),Integer(1)]] * M[[Integer(1),Integer(1),Integer(2)]]␣
↪  # packed words
M[{1, 2, 3}, {4, 5}, {6}] + M[{1, 2, 3, 4, 5}, {6}]
 + M[{4, 5}, {1, 2, 3}, {6}] + M[{4, 5}, {1, 2, 3, 6}]
 + M[{4, 5}, {6}, {1, 2, 3}]
>>> M[[Integer(1),Integer(2),Integer(3)],].antipode()  # ordered set partition
-M[{1, 2, 3}]
>>> M[[Integer(1)], [Integer(2)], [Integer(3)]].antipode()
-M[{1, 2, 3}] - M[{2, 3}, {1}] - M[{3}, {1, 2}] - M[{3}, {2}, {1}]
>>> x = M[[Integer(1)],[Integer(2)],[Integer(3)]] + Integer(3)*M[[Integer(2)],
↪[Integer(1)]]
>>> x.counit()
0
>>> x.antipode()
3*M[{1}, {2}] + 3*M[{1, 2}] - M[{1, 2, 3}] - M[{2, 3}, {1}]
 - M[{3}, {1, 2}] - M[{3}, {2}, {1}]
```

Ambiguities

Some ambiguity arises when accessing basis vectors with the dictionary syntax, i.e., `M[...]`. A common example is when referencing an ordered set partition with one part. For example, in the expression `M[[1,2]]`, does `[[1, 2]]` refer to an ordered set partition or does `[1,2]` refer to a packed word? We choose the latter: if the received arguments do not behave like a tuple of iterables, then view them as describing a packed word. (In the running example, one argument is received, which behaves as a tuple of integers.) Here are a variety of ways to get the same basis vector:

```
sage: x = M[1,1]; x
M[{1, 2}]
sage: x == M[[1,1]]  # treated as word
True
sage: x == M[[1,2],] == M[[[1,2]]]  # treated as ordered set partitions
True

sage: M[[1,3],[2]]  # treat as ordered set partition
M[{1, 3}, {2}]
sage: M[[1,3],[2]] == M[1,2,1]  # treat as word
True
```

```
>>> from sage.all import *
>>> x = M[Integer(1),Integer(1)]; x
M[{1, 2}]
>>> x == M[[Integer(1),Integer(1)]]  # treated as word
True
```

(continues on next page)

(continued from previous page)

```
>>> x == M[[Integer(1),Integer(2)],] == M[[[Integer(1),Integer(2)]]]  # treated␣
↪as ordered set partitions
True

>>> M[[Integer(1),Integer(3)],[Integer(2)]]  # treat as ordered set partition
M[{1, 3}, {2}]
>>> M[[Integer(1),Integer(3)],[Integer(2)]] == M[Integer(1),Integer(2),
↪Integer(1)]  # treat as word
True
```

Todo

- Dendriform structure.

C

alias of `Cone`

class Characteristic(*alg*)

Bases: `WQSymBasis_abstract`

The Characteristic basis of $WQSym$.

The *Characteristic basis* is a graded basis (X_P) of $WQSym$, indexed by ordered set partitions P. It is defined by

$$X_P = (-1)^{\ell(P)} \mathbf{M}_P,$$

where $(\mathbf{M}_P)_P$ denotes the Monomial basis, and where $\ell(P)$ denotes the number of blocks in an ordered set partition P.

EXAMPLES:

```
sage: WQSym = algebras.WQSym(QQ)
sage: X = WQSym.X(); X
Word Quasi-symmetric functions over Rational Field in the Characteristic basis

sage: X[[[1,2,3]]] * X[[1,2],[3]]
X[{1, 2, 3}, {4, 5}, {6}] - X[{1, 2, 3, 4, 5}, {6}]
 + X[{4, 5}, {1, 2, 3}, {6}] - X[{4, 5}, {1, 2, 3, 6}]
 + X[{4, 5}, {6}, {1, 2, 3}]

sage: X[[1, 4], [3], [2]].coproduct()
X[] # X[{1, 4}, {3}, {2}] + X[{1, 2}] # X[{2}, {1}]
 + X[{1, 3}, {2}] # X[{1}] + X[{1, 4}, {3}, {2}] # X[]

sage: M = WQSym.M()
sage: M(X[[1, 2, 3],])
-M[{1, 2, 3}]
sage: M(X[[1, 3], [2]])
M[{1, 3}, {2}]
sage: X(M[[1, 2, 3],])
-X[{1, 2, 3}]
```

(continues on next page)

(continued from previous page)

```
sage: X(M[[1, 3], [2]])
X[{1, 3}, {2}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(QQ)
>>> X = WQSym.X(); X
Word Quasi-symmetric functions over Rational Field in the Characteristic basis

>>> X[[[Integer(1),Integer(2),Integer(3)]]] * X[[Integer(1),Integer(2)],
↪[Integer(3)]]
X[{1, 2, 3}, {4, 5}, {6}] - X[{1, 2, 3, 4, 5}, {6}]
 + X[{4, 5}, {1, 2, 3}, {6}] - X[{4, 5}, {1, 2, 3, 6}]
 + X[{4, 5}, {6}, {1, 2, 3}]

>>> X[[Integer(1), Integer(4)], [Integer(3)], [Integer(2)]].coproduct()
X[] # X[{1, 4}, {3}, {2}] + X[{1, 2}] # X[{2}, {1}]
 + X[{1, 3}, {2}] # X[{1}] + X[{1, 4}, {3}, {2}] # X[]

>>> M = WQSym.M()
>>> M(X[[Integer(1), Integer(2), Integer(3)],])
-M[{1, 2, 3}]
>>> M(X[[Integer(1), Integer(3)], [Integer(2)]])
M[{1, 3}, {2}]
>>> X(M[[Integer(1), Integer(2), Integer(3)],])
-X[{1, 2, 3}]
>>> X(M[[Integer(1), Integer(3)], [Integer(2)]])
X[{1, 3}, {2}]
```

class Element

Bases: `IndexedFreeModuleElement`

algebraic_complement()

Return the image of the element `self` of $WQSym$ under the algebraic complement involution.

See *WQSymBases.ElementMethods.algebraic_complement()* for a definition of the involution and for examples.

See also

coalgebraic_complement(), *star_involution()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: X = WQSym.X()
sage: X[[1,2],[5,6],[3,4]].algebraic_complement()
X[{3, 4}, {5, 6}, {1, 2}]
sage: X[[3], [1, 2], [4]].algebraic_complement()
X[{4}, {1, 2}, {3}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> X = WQSym.X()
>>> X[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].algebraic_complement()
X[{3, 4}, {5, 6}, {1, 2}]
>>> X[[Integer(3)], [Integer(1), Integer(2)], [Integer(4)]].algebraic_
↪complement()
X[{4}, {1, 2}, {3}]
```

coalgebraic_complement()

Return the image of the element `self` of $WQSym$ under the coalgebraic complement involution.

See *WQSymBases.ElementMethods.coalgebraic_complement()* for a definition of the involution and for examples.

See also:

algebraic_complement(), *star_involution()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: X = WQSym.X()
sage: X[[1,2],[5,6],[3,4]].coalgebraic_complement()
X[{5, 6}, {1, 2}, {3, 4}]
sage: X[[3], [1, 2], [4]].coalgebraic_complement()
X[{2}, {3, 4}, {1}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> X = WQSym.X()
>>> X[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].coalgebraic_complement()
X[{5, 6}, {1, 2}, {3, 4}]
>>> X[[Integer(3)], [Integer(1), Integer(2)], [Integer(4)]].
↪coalgebraic_complement()
X[{2}, {3, 4}, {1}]
```

star_involution()

Return the image of the element `self` of $WQSym$ under the star involution.

See *WQSymBases.ElementMethods.star_involution()* for a definition of the involution and for examples.

See also:

algebraic_complement(), *coalgebraic_complement()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: X = WQSym.X()
sage: X[[1,2],[5,6],[3,4]].star_involution()
X[{3, 4}, {1, 2}, {5, 6}]
sage: X[[3], [1, 2], [4]].star_involution()
X[{1}, {3, 4}, {2}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> X = WQSym.X()
>>> X[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].star_involution()
X[{3, 4}, {1, 2}, {5, 6}]
>>> X[[Integer(3)], [Integer(1), Integer(2)], [Integer(4)]].star_
↪involution()
X[{1}, {3, 4}, {2}]
```

class Cone(*alg*)

Bases: `WQSymBasis_abstract`

The Cone basis of $WQSym$.

Let $(X_P)_P$ denote the Characteristic basis of $WQSym$. Denote the quasi-shuffle of two ordered set partitions A and B by $A \square B$. For an ordered set partition $P = (P_1, \ldots, P_\ell)$, we form a list of ordered set partitions $[P] := (P'_1, \ldots, P'_k)$ as follows. Define a strictly decreasing sequence of integers $\ell + 1 = i_0 > i_1 > \cdots > i_k = 1$ recursively by requiring that $\min P_{i_j} \leq \min P_a$ for all $a < i_{j-1}$. Set $P'_j = (P_{i_j}, \ldots, P_{i_{j-1}-1})$.

The *Cone basis* $(C_P)_P$ is defined by

$$C_P = \sum_Q X_Q,$$

where the sum is over all elements Q of the quasi-shuffle product $P'_1 \square P'_2 \square \cdots \square P'_k$ with $[P] = (P'_1, \ldots, P'_k)$.

EXAMPLES:

```
sage: WQSym = algebras.WQSym(QQ)
sage: C = WQSym.C()
sage: C
Word Quasi-symmetric functions over Rational Field in the Cone basis

sage: X = WQSym.X()
sage: X(C[[2,3],[1,4]])
X[{1, 2, 3, 4}] + X[{1, 4}, {2, 3}] + X[{2, 3}, {1, 4}]
sage: X(C[[1,4],[2,3]])
X[{1, 4}, {2, 3}]
sage: X(C[[2,3],[1],[4]])
X[{1}, {2, 3}, {4}] + X[{1}, {2, 3, 4}] + X[{1}, {4}, {2, 3}]
 + X[{1, 2, 3}, {4}] + X[{2, 3}, {1}, {4}]
sage: X(C[[3], [2, 5], [1, 4]])
X[{1, 2, 3, 4, 5}] + X[{1, 2, 4, 5}, {3}] + X[{1, 3, 4}, {2, 5}]
 + X[{1, 4}, {2, 3, 5}] + X[{1, 4}, {2, 5}, {3}]
 + X[{1, 4}, {3}, {2, 5}] + X[{2, 3, 5}, {1, 4}]
 + X[{2, 5}, {1, 3, 4}] + X[{2, 5}, {1, 4}, {3}]
 + X[{2, 5}, {3}, {1, 4}] + X[{3}, {1, 2, 4, 5}]
```

(continues on next page)

(continued from previous page)

```
 + X[{3}, {1, 4}, {2, 5}] + X[{3}, {2, 5}, {1, 4}]
sage: C(X[[2,3],[1,4]])
-C[{1, 2, 3, 4}] - C[{1, 4}, {2, 3}] + C[{2, 3}, {1, 4}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(QQ)
>>> C = WQSym.C()
>>> C
Word Quasi-symmetric functions over Rational Field in the Cone basis

>>> X = WQSym.X()
>>> X(C[[Integer(2),Integer(3)],[Integer(1),Integer(4)]])
X[{1, 2, 3, 4}] + X[{1, 4}, {2, 3}] + X[{2, 3}, {1, 4}]
>>> X(C[[Integer(1),Integer(4)],[Integer(2),Integer(3)]])
X[{1, 4}, {2, 3}]
>>> X(C[[Integer(2),Integer(3)],[Integer(1)],[Integer(4)]])
X[{1}, {2, 3}, {4}] + X[{1}, {2, 3, 4}] + X[{1}, {4}, {2, 3}]
 + X[{1, 2, 3}, {4}] + X[{2, 3}, {1}, {4}]
>>> X(C[[Integer(3)], [Integer(2), Integer(5)], [Integer(1), Integer(4)]])
X[{1, 2, 3, 4, 5}] + X[{1, 2, 4, 5}, {3}] + X[{1, 3, 4}, {2, 5}]
 + X[{1, 4}, {2, 3, 5}] + X[{1, 4}, {2, 5}, {3}]
 + X[{1, 4}, {3}, {2, 5}] + X[{2, 3, 5}, {1, 4}]
 + X[{2, 5}, {1, 3, 4}] + X[{2, 5}, {1, 4}, {3}]
 + X[{2, 5}, {3}, {1, 4}] + X[{3}, {1, 2, 4, 5}]
 + X[{3}, {1, 4}, {2, 5}] + X[{3}, {2, 5}, {1, 4}]
>>> C(X[[Integer(2),Integer(3)],[Integer(1),Integer(4)]])
-C[{1, 2, 3, 4}] - C[{1, 4}, {2, 3}] + C[{2, 3}, {1, 4}]
```

REFERENCES:

- Section 4 of [Early2017]

Todo

Experiments suggest that `algebraic_complement()`, `coalgebraic_complement()`, and `star_involution()` should have reasonable formulas on the C basis; at least the coefficients of the outputs on any element of the C basis seem to be always $0, 1, -1$. Is this true? What is the formula?

some_elements()

Return some elements of the word quasi-symmetric functions in the Cone basis.

EXAMPLES:

```
sage: C = algebras.WQSym(QQ).C()
sage: C.some_elements()
[C[], C[{1}], C[{1, 2}], C[] + 1/2*C[{1}]]
```

```
>>> from sage.all import *
>>> C = algebras.WQSym(QQ).C()
>>> C.some_elements()
[C[], C[{1}], C[{1, 2}], C[] + 1/2*C[{1}]]
```

M

alias of *Monomial*

class Monomial(*alg*, *graded=True*)

Bases: *WQSymBasis_abstract*

The Monomial basis of $WQSym$.

The family $(\mathbf{M}_u)$, as defined in *WordQuasiSymmetricFunctions* with u ranging over all packed words, is a basis for the free R-module $WQSym$ and called the *Monomial basis*. Here it is labelled using ordered set partitions.

EXAMPLES:

```
sage: WQSym = algebras.WQSym(QQ)
sage: M = WQSym.M(); M
Word Quasi-symmetric functions over Rational Field in the Monomial basis
sage: sorted(M.basis(2))
[M[{1}, {2}], M[{2}, {1}], M[{1, 2}]]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(QQ)
>>> M = WQSym.M(); M
Word Quasi-symmetric functions over Rational Field in the Monomial basis
>>> sorted(M.basis(Integer(2)))
[M[{1}, {2}], M[{2}, {1}], M[{1, 2}]]
```

coproduct_on_basis(*x*)

Return the coproduct of `self` on the basis element indexed by the ordered set partition `x`.

EXAMPLES:

```
sage: M = algebras.WQSym(QQ).M()

sage: M.coproduct(M.one())  # indirect doctest
M[] # M[]
sage: M.coproduct( M([[1]]) )  # indirect doctest
M[] # M[{1}] + M[{1}] # M[]
sage: M.coproduct( M([[1,2]]) )
M[] # M[{1, 2}] + M[{1, 2}] # M[]
sage: M.coproduct( M([[1], [2]]) )
M[] # M[{1}, {2}] + M[{1}] # M[{1}] + M[{1}, {2}] # M[]
```

```
>>> from sage.all import *
>>> M = algebras.WQSym(QQ).M()

>>> M.coproduct(M.one())  # indirect doctest
M[] # M[]
>>> M.coproduct( M([[Integer(1)]]) )  # indirect doctest
M[] # M[{1}] + M[{1}] # M[]
>>> M.coproduct( M([[Integer(1),Integer(2)]]) )
M[] # M[{1, 2}] + M[{1, 2}] # M[]
>>> M.coproduct( M([[Integer(1)], [Integer(2)]]) )
M[] # M[{1}, {2}] + M[{1}] # M[{1}] + M[{1}, {2}] # M[]
```

product_on_basis (*x*, *y*)

Return the (associative) $*$ product of the basis elements of `self` indexed by the ordered set partitions x and y.

This is the shifted quasi-shuffle product of x and y.

EXAMPLES:

```
sage: A = algebras.WQSym(QQ).M()
sage: x = OrderedSetPartition([[1],[2,3]])
sage: y = OrderedSetPartition([[1,2]])
sage: z = OrderedSetPartition([[1,2],[3]])
sage: A.product_on_basis(x, y)
M[{1}, {2, 3}, {4, 5}] + M[{1}, {2, 3, 4, 5}]
 + M[{1}, {4, 5}, {2, 3}] + M[{1, 4, 5}, {2, 3}]
 + M[{4, 5}, {1}, {2, 3}]
sage: A.product_on_basis(x, z)
M[{1}, {2, 3}, {4, 5}, {6}] + M[{1}, {2, 3, 4, 5}, {6}]
 + M[{1}, {4, 5}, {2, 3}, {6}] + M[{1}, {4, 5}, {2, 3, 6}]
 + M[{1}, {4, 5}, {6}, {2, 3}] + M[{1, 4, 5}, {2, 3}, {6}]
 + M[{1, 4, 5}, {2, 3, 6}] + M[{1, 4, 5}, {6}, {2, 3}]
 + M[{4, 5}, {1}, {2, 3}, {6}] + M[{4, 5}, {1}, {2, 3, 6}]
 + M[{4, 5}, {1}, {6}, {2, 3}] + M[{4, 5}, {1, 6}, {2, 3}]
 + M[{4, 5}, {6}, {1}, {2, 3}]
sage: A.product_on_basis(y, y)
M[{1, 2}, {3, 4}] + M[{1, 2, 3, 4}] + M[{3, 4}, {1, 2}]
```

```
>>> from sage.all import *
>>> A = algebras.WQSym(QQ).M()
>>> x = OrderedSetPartition([[Integer(1)],[Integer(2),Integer(3)]])
>>> y = OrderedSetPartition([[Integer(1),Integer(2)]])
>>> z = OrderedSetPartition([[Integer(1),Integer(2)],[Integer(3)]])
>>> A.product_on_basis(x, y)
M[{1}, {2, 3}, {4, 5}] + M[{1}, {2, 3, 4, 5}]
 + M[{1}, {4, 5}, {2, 3}] + M[{1, 4, 5}, {2, 3}]
 + M[{4, 5}, {1}, {2, 3}]
>>> A.product_on_basis(x, z)
M[{1}, {2, 3}, {4, 5}, {6}] + M[{1}, {2, 3, 4, 5}, {6}]
 + M[{1}, {4, 5}, {2, 3}, {6}] + M[{1}, {4, 5}, {2, 3, 6}]
 + M[{1}, {4, 5}, {6}, {2, 3}] + M[{1, 4, 5}, {2, 3}, {6}]
 + M[{1, 4, 5}, {2, 3, 6}] + M[{1, 4, 5}, {6}, {2, 3}]
 + M[{4, 5}, {1}, {2, 3}, {6}] + M[{4, 5}, {1}, {2, 3, 6}]
 + M[{4, 5}, {1}, {6}, {2, 3}] + M[{4, 5}, {1, 6}, {2, 3}]
 + M[{4, 5}, {6}, {1}, {2, 3}]
>>> A.product_on_basis(y, y)
M[{1, 2}, {3, 4}] + M[{1, 2, 3, 4}] + M[{3, 4}, {1, 2}]
```

Phi

alias of *StronglyFiner*

Q

alias of *StronglyCoarser*

class StronglyCoarser (*alg*)

Bases: *WQSymBasis_abstract*

The Q basis of $WQSym$.

We define a partial order $\leq$ on the set of all ordered set partitions as follows: $A \leq B$ if and only if A is strongly finer than B (see `is_strongly_finer()` for a definition of this).

The *Q basis* $(Q_P)_P$ is a basis of $WQSym$ indexed by ordered set partitions, and is defined by

$$Q_P = \sum \mathbf{M}_W,$$

where the sum is over ordered set partitions W satisfying $P \leq W$.

EXAMPLES:

```
sage: WQSym = algebras.WQSym(QQ)
sage: M = WQSym.M(); Q = WQSym.Q()
sage: Q
Word Quasi-symmetric functions over Rational Field in the Q basis

sage: Q(M[[2,3],[1,4]])
Q[{2, 3}, {1, 4}]
sage: Q(M[[1,2],[3,4]])
Q[{1, 2}, {3, 4}] - Q[{1, 2, 3, 4}]
sage: M(Q[[1,2],[3,4]])
M[{1, 2}, {3, 4}] + M[{1, 2, 3, 4}]
sage: M(Q[[2,3],[1],[4]])
M[{2, 3}, {1}, {4}] + M[{2, 3}, {1, 4}]
sage: M(Q[[3], [2, 5], [1, 4]])
M[{3}, {2, 5}, {1, 4}]
sage: M(Q[[1, 4], [2, 3], [5], [6]])
M[{1, 4}, {2, 3}, {5}, {6}] + M[{1, 4}, {2, 3}, {5, 6}]
 + M[{1, 4}, {2, 3, 5}, {6}] + M[{1, 4}, {2, 3, 5, 6}]

sage: Q[[1, 3], [2]] * Q[[1], [2]]
Q[{1, 3}, {2}, {4}, {5}] + Q[{1, 3}, {4}, {2}, {5}]
 + Q[{1, 3}, {4}, {5}, {2}] + Q[{4}, {1, 3}, {2}, {5}]
 + Q[{4}, {1, 3}, {5}, {2}] + Q[{4}, {5}, {1, 3}, {2}]

sage: Q[[1, 3], [2]].coproduct()
Q[] # Q[{1, 3}, {2}] + Q[{1, 2}] # Q[{1}] + Q[{1, 3}, {2}] # Q[]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(QQ)
>>> M = WQSym.M(); Q = WQSym.Q()
>>> Q
Word Quasi-symmetric functions over Rational Field in the Q basis

>>> Q(M[[Integer(2),Integer(3)],[Integer(1),Integer(4)]])
Q[{2, 3}, {1, 4}]
>>> Q(M[[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
Q[{1, 2}, {3, 4}] - Q[{1, 2, 3, 4}]
>>> M(Q[[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
M[{1, 2}, {3, 4}] + M[{1, 2, 3, 4}]
>>> M(Q[[Integer(2),Integer(3)],[Integer(1)],[Integer(4)]])
M[{2, 3}, {1}, {4}] + M[{2, 3}, {1, 4}]
>>> M(Q[[Integer(3)], [Integer(2), Integer(5)], [Integer(1), Integer(4)]])
```

(continues on next page)

(continued from previous page)

```
M[{3}, {2, 5}, {1, 4}]
>>> M(Q[[Integer(1), Integer(4)], [Integer(2), Integer(3)], [Integer(5)],␣
↪[Integer(6)]]])
M[{1, 4}, {2, 3}, {5}, {6}] + M[{1, 4}, {2, 3}, {5, 6}]
 + M[{1, 4}, {2, 3, 5}, {6}] + M[{1, 4}, {2, 3, 5, 6}]

>>> Q[[Integer(1), Integer(3)], [Integer(2)]] * Q[[Integer(1)], [Integer(2)]]
Q[{1, 3}, {2}, {4}, {5}] + Q[{1, 3}, {4}, {2}, {5}]
 + Q[{1, 3}, {4}, {5}, {2}] + Q[{4}, {1, 3}, {2}, {5}]
 + Q[{4}, {1, 3}, {5}, {2}] + Q[{4}, {5}, {1, 3}, {2}]

>>> Q[[Integer(1), Integer(3)], [Integer(2)]].coproduct()
Q[] # Q[{1, 3}, {2}] + Q[{1, 2}] # Q[{1}] + Q[{1, 3}, {2}] # Q[]
```

REFERENCES:

- Section 6 of [BerZab05]

class Element

Bases: IndexedFreeModuleElement

algebraic_complement()

Return the image of the element `self` of $WQSym$ under the algebraic complement involution.

See *WQSymBases.ElementMethods.algebraic_complement()* for a definition of the involution and for examples.

See also

coalgebraic_complement(), *star_involution()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: Q = WQSym.Q()
sage: Q[[1,2],[5,6],[3,4]].algebraic_complement()
Q[{3, 4}, {1, 2, 5, 6}] + Q[{3, 4}, {5, 6}, {1, 2}]
 - Q[{3, 4, 5, 6}, {1, 2}]
sage: Q[[3], [1, 2], [4]].algebraic_complement()
Q[{1, 2, 4}, {3}] + Q[{4}, {1, 2}, {3}] - Q[{4}, {1, 2, 3}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> Q = WQSym.Q()
>>> Q[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].algebraic_complement()
Q[{3, 4}, {1, 2, 5, 6}] + Q[{3, 4}, {5, 6}, {1, 2}]
 - Q[{3, 4, 5, 6}, {1, 2}]
>>> Q[[Integer(3)], [Integer(1), Integer(2)], [Integer(4)]].algebraic_
↪complement()
Q[{1, 2, 4}, {3}] + Q[{4}, {1, 2}, {3}] - Q[{4}, {1, 2, 3}]
```

coalgebraic_complement()

Return the image of the element `self` of $WQSym$ under the coalgebraic complement involution.

See *WQSymBases.ElementMethods.coalgebraic_complement()* for a definition of the involution and for examples.

See also

algebraic_complement(), *star_involution()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: Q = WQSym.Q()
sage: Q[[1,2],[5,6],[3,4]].coalgebraic_complement()
Q[{1, 2, 5, 6}, {3, 4}] + Q[{5, 6}, {1, 2}, {3, 4}] - Q[{5, 6}, {1, 2,␣
↪3, 4}]
sage: Q[[3], [1, 2], [4]].coalgebraic_complement()
Q[{2}, {1, 3, 4}] + Q[{2}, {3, 4}, {1}] - Q[{2, 3, 4}, {1}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> Q = WQSym.Q()
>>> Q[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].coalgebraic_complement()
Q[{1, 2, 5, 6}, {3, 4}] + Q[{5, 6}, {1, 2}, {3, 4}] - Q[{5, 6}, {1, 2,␣
↪3, 4}]
>>> Q[[Integer(3)], [Integer(1), Integer(2)], [Integer(4)]].
↪coalgebraic_complement()
Q[{2}, {1, 3, 4}] + Q[{2}, {3, 4}, {1}] - Q[{2, 3, 4}, {1}]
```

star_involution()

Return the image of the element `self` of $WQSym$ under the star involution.

See *WQSymBases.ElementMethods.star_involution()* for a definition of the involution and for examples.

See also

algebraic_complement(), *coalgebraic_complement()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: Q = WQSym.Q()
sage: Q[[1,2],[5,6],[3,4]].star_involution()
Q[{3, 4}, {1, 2}, {5, 6}]
sage: Q[[3], [1, 2], [4]].star_involution()
Q[{1}, {3, 4}, {2}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
```

(continues on next page)

(continued from previous page)

```
>>> Q = WQSym.Q()
>>> Q[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].star_involution()
Q[{3, 4}, {1, 2}, {5, 6}]
>>> Q[[Integer(3)], [Integer(1), Integer(2)], [Integer(4)]].star_
↪involution()
Q[{1}, {3, 4}, {2}]
```

coproduct_on_basis(x)

Return the coproduct of `self` on the basis element indexed by the ordered set partition `x`.

EXAMPLES:

```
sage: Q = algebras.WQSym(QQ).Q()

sage: Q.coproduct(Q.one())  # indirect doctest
Q[] # Q[]
sage: Q.coproduct( Q([[1]]) )  # indirect doctest
Q[] # Q[{1}] + Q[{1}] # Q[]
sage: Q.coproduct( Q([[1,2]]) )
Q[] # Q[{1, 2}] + Q[{1, 2}] # Q[]
sage: Q.coproduct( Q([[1], [2]]) )
Q[] # Q[{1}, {2}] + Q[{1}] # Q[{1}] + Q[{1}, {2}] # Q[]
sage: Q[[1,2],[3],[4]].coproduct()
Q[] # Q[{1, 2}, {3}, {4}] + Q[{1, 2}] # Q[{1}, {2}]
 + Q[{1, 2}, {3}] # Q[{1}] + Q[{1, 2}, {3}, {4}] # Q[]
```

```
>>> from sage.all import *
>>> Q = algebras.WQSym(QQ).Q()

>>> Q.coproduct(Q.one())  # indirect doctest
Q[] # Q[]
>>> Q.coproduct( Q([[Integer(1)]]) )  # indirect doctest
Q[] # Q[{1}] + Q[{1}] # Q[]
>>> Q.coproduct( Q([[Integer(1),Integer(2)]]) )
Q[] # Q[{1, 2}] + Q[{1, 2}] # Q[]
>>> Q.coproduct( Q([[Integer(1)], [Integer(2)]]) )
Q[] # Q[{1}, {2}] + Q[{1}] # Q[{1}] + Q[{1}, {2}] # Q[]
>>> Q[[Integer(1),Integer(2)],[Integer(3)],[Integer(4)]].coproduct()
Q[] # Q[{1, 2}, {3}, {4}] + Q[{1, 2}] # Q[{1}, {2}]
 + Q[{1, 2}, {3}] # Q[{1}] + Q[{1, 2}, {3}, {4}] # Q[]
```

product_on_basis(x, y)

Return the (associative) $*$ product of the basis elements of the Q basis `self` indexed by the ordered set partitions x and y.

This is the shifted shuffle product of x and y.

EXAMPLES:

```
sage: A = algebras.WQSym(QQ).Q()
sage: x = OrderedSetPartition([[1],[2,3]])
sage: y = OrderedSetPartition([[1,2]])
```

(continues on next page)

(continued from previous page)

```
sage: z = OrderedSetPartition([[1,2],[3]])
sage: A.product_on_basis(x, y)
Q[{1}, {2, 3}, {4, 5}] + Q[{1}, {4, 5}, {2, 3}]
 + Q[{4, 5}, {1}, {2, 3}]
sage: A.product_on_basis(x, z)
Q[{1}, {2, 3}, {4, 5}, {6}] + Q[{1}, {4, 5}, {2, 3}, {6}]
 + Q[{1}, {4, 5}, {6}, {2, 3}] + Q[{4, 5}, {1}, {2, 3}, {6}]
 + Q[{4, 5}, {1}, {6}, {2, 3}] + Q[{4, 5}, {6}, {1}, {2, 3}]
sage: A.product_on_basis(y, y)
Q[{1, 2}, {3, 4}] + Q[{3, 4}, {1, 2}]
```

```
>>> from sage.all import *
>>> A = algebras.WQSym(QQ).Q()
>>> x = OrderedSetPartition([[Integer(1)],[Integer(2),Integer(3)]])
>>> y = OrderedSetPartition([[Integer(1),Integer(2)]])
>>> z = OrderedSetPartition([[Integer(1),Integer(2)],[Integer(3)]])
>>> A.product_on_basis(x, y)
Q[{1}, {2, 3}, {4, 5}] + Q[{1}, {4, 5}, {2, 3}]
 + Q[{4, 5}, {1}, {2, 3}]
>>> A.product_on_basis(x, z)
Q[{1}, {2, 3}, {4, 5}, {6}] + Q[{1}, {4, 5}, {2, 3}, {6}]
 + Q[{1}, {4, 5}, {6}, {2, 3}] + Q[{4, 5}, {1}, {2, 3}, {6}]
 + Q[{4, 5}, {1}, {6}, {2, 3}] + Q[{4, 5}, {6}, {1}, {2, 3}]
>>> A.product_on_basis(y, y)
Q[{1, 2}, {3, 4}] + Q[{3, 4}, {1, 2}]
```

some_elements()

Return some elements of the word quasi-symmetric functions in the Q basis.

EXAMPLES:

```
sage: Q = algebras.WQSym(QQ).Q()
sage: Q.some_elements()
[Q[], Q[{1}], Q[{1, 2}], Q[] + 1/2*Q[{1}]]
```

```
>>> from sage.all import *
>>> Q = algebras.WQSym(QQ).Q()
>>> Q.some_elements()
[Q[], Q[{1}], Q[{1, 2}], Q[] + 1/2*Q[{1}]]
```

class StronglyFiner(*alg*)

Bases: `WQSymBasis_abstract`

The Phi basis of $WQSym$.

We define a partial order $\leq$ on the set of all ordered set partitions as follows: $A \leq B$ if and only if A is strongly finer than B (see `is_strongly_finer()` for a definition of this).

The *Phi basis* $(\Phi_P)_P$ is a basis of $WQSym$ indexed by ordered set partitions, and is defined by

$$\Phi_P = \sum \mathbf{M}_W,$$

where the sum is over ordered set partitions W satisfying $W \leq P$.

Novelli and Thibon introduced this basis in [NovThi06] Section 2.7.2, and called it the quasi-ribbon basis. It later reappeared in [MeNoTh11] Section 4.3.2.

EXAMPLES:

```
sage: WQSym = algebras.WQSym(QQ)
sage: M = WQSym.M(); Phi = WQSym.Phi()
sage: Phi
Word Quasi-symmetric functions over Rational Field in the Phi basis

sage: Phi(M[[2,3],[1,4]])
Phi[{2}, {3}, {1}, {4}] - Phi[{2}, {3}, {1, 4}]
 - Phi[{2, 3}, {1}, {4}] + Phi[{2, 3}, {1, 4}]
sage: Phi(M[[1,2],[3,4]])
Phi[{1}, {2}, {3}, {4}] - Phi[{1}, {2}, {3, 4}]
 - Phi[{1, 2}, {3}, {4}] + Phi[{1, 2}, {3, 4}]
sage: M(Phi[[1,2],[3,4]])
M[{1}, {2}, {3}, {4}] + M[{1}, {2}, {3, 4}]
 + M[{1, 2}, {3}, {4}] + M[{1, 2}, {3, 4}]
sage: M(Phi[[2,3],[1],[4]])
M[{2}, {3}, {1}, {4}] + M[{2, 3}, {1}, {4}]
sage: M(Phi[[3], [2, 5], [1, 4]])
M[{3}, {2}, {5}, {1}, {4}] + M[{3}, {2}, {5}, {1, 4}]
 + M[{3}, {2, 5}, {1}, {4}] + M[{3}, {2, 5}, {1, 4}]
sage: M(Phi[[1, 4], [2, 3], [5], [6]])
M[{1}, {4}, {2}, {3}, {5}, {6}] + M[{1}, {4}, {2, 3}, {5}, {6}]
 + M[{1, 4}, {2}, {3}, {5}, {6}] + M[{1, 4}, {2, 3}, {5}, {6}]

sage: Phi[[1],] * Phi[[1, 3], [2]]
Phi[{1, 2, 4}, {3}] + Phi[{2}, {1, 4}, {3}]
 + Phi[{2, 4}, {1, 3}] + Phi[{2, 4}, {3}, {1}]
sage: Phi[[3, 5], [1, 4], [2]].coproduct()
Phi[] # Phi[{3, 5}, {1, 4}, {2}]
 + Phi[{1}] # Phi[{4}, {1, 3}, {2}]
 + Phi[{1, 2}] # Phi[{1, 3}, {2}]
 + Phi[{2, 3}, {1}] # Phi[{2}, {1}]
 + Phi[{2, 4}, {1, 3}] # Phi[{1}]
 + Phi[{3, 5}, {1, 4}, {2}] # Phi[]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(QQ)
>>> M = WQSym.M(); Phi = WQSym.Phi()
>>> Phi
Word Quasi-symmetric functions over Rational Field in the Phi basis

>>> Phi(M[[Integer(2),Integer(3)],[Integer(1),Integer(4)]])
Phi[{2}, {3}, {1}, {4}] - Phi[{2}, {3}, {1, 4}]
 - Phi[{2, 3}, {1}, {4}] + Phi[{2, 3}, {1, 4}]
>>> Phi(M[[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
Phi[{1}, {2}, {3}, {4}] - Phi[{1}, {2}, {3, 4}]
 - Phi[{1, 2}, {3}, {4}] + Phi[{1, 2}, {3, 4}]
>>> M(Phi[[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
M[{1}, {2}, {3}, {4}] + M[{1}, {2}, {3, 4}]
 + M[{1, 2}, {3}, {4}] + M[{1, 2}, {3, 4}]
```

(continues on next page)

(continued from previous page)

```
>>> M(Phi[[Integer(2),Integer(3)],[Integer(1)],[Integer(4)]])
M[{2}, {3}, {1}, {4}] + M[{2, 3}, {1}, {4}]
>>> M(Phi[[Integer(3)], [Integer(2), Integer(5)], [Integer(1), Integer(4)]])
M[{3}, {2}, {5}, {1}, {4}] + M[{3}, {2}, {5}, {1, 4}]
 + M[{3}, {2, 5}, {1}, {4}] + M[{3}, {2, 5}, {1, 4}]
>>> M(Phi[[Integer(1), Integer(4)], [Integer(2), Integer(3)], [Integer(5)],
↪[Integer(6)]])
M[{1}, {4}, {2}, {3}, {5}, {6}] + M[{1}, {4}, {2, 3}, {5}, {6}]
 + M[{1, 4}, {2}, {3}, {5}, {6}] + M[{1, 4}, {2, 3}, {5}, {6}]

>>> Phi[[Integer(1)],] * Phi[[Integer(1), Integer(3)], [Integer(2)]]
Phi[{1, 2, 4}, {3}] + Phi[{2}, {1, 4}, {3}]
 + Phi[{2, 4}, {1, 3}] + Phi[{2, 4}, {3}, {1}]
>>> Phi[[Integer(3), Integer(5)], [Integer(1), Integer(4)], [Integer(2)]].
↪coproduct()
Phi[] # Phi[{3, 5}, {1, 4}, {2}]
 + Phi[{1}] # Phi[{4}, {1, 3}, {2}]
 + Phi[{1, 2}] # Phi[{1, 3}, {2}]
 + Phi[{2, 3}, {1}] # Phi[{2}, {1}]
 + Phi[{2, 4}, {1, 3}] # Phi[{1}]
 + Phi[{3, 5}, {1, 4}, {2}] # Phi[]
```

REFERENCES:

- Section 2.7.2 of [NovThi06]

class Element

Bases: `IndexedFreeModuleElement`

algebraic_complement()

Return the image of the element `self` of $WQSym$ under the algebraic complement involution.

See *WQSymBases.ElementMethods.algebraic_complement()* for a definition of the involution and for examples.

See also

coalgebraic_complement(), *star_involution()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: Phi = WQSym.Phi()
sage: Phi[[1],[2,4],[3]].algebraic_complement()
-Phi[{3}, {2}, {4}, {1}] + Phi[{3}, {2, 4}, {1}] + Phi[{3}, {4}, {2},
↪{1}]
sage: Phi[[1],[2,3],[4]].algebraic_complement()
-Phi[{4}, {2}, {3}, {1}] + Phi[{4}, {2, 3}, {1}] + Phi[{4}, {3}, {2},
↪{1}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> Phi = WQSym.Phi()
```

(continues on next page)

(continued from previous page)

```
>>> Phi[[Integer(1)],[Integer(2),Integer(4)],[Integer(3)]].algebraic_
↪complement()
-Phi[{3}, {2}, {4}, {1}] + Phi[{3}, {2, 4}, {1}] + Phi[{3}, {4}, {2},
↪{1}]
>>> Phi[[Integer(1)],[Integer(2),Integer(3)],[Integer(4)]].algebraic_
↪complement()
-Phi[{4}, {2}, {3}, {1}] + Phi[{4}, {2, 3}, {1}] + Phi[{4}, {3}, {2},
↪{1}]
```

coalgebraic_complement()

Return the image of the element `self` of $WQSym$ under the coalgebraic complement involution.

See *WQSymBases.ElementMethods.coalgebraic_complement()* for a definition of the involution and for examples.

See also

algebraic_complement(), *star_involution()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: Phi = WQSym.Phi()
sage: Phi[[1],[2],[3,4]].coalgebraic_complement()
-Phi[{4}, {3}, {1}, {2}] + Phi[{4}, {3}, {1, 2}] + Phi[{4}, {3}, {2},
↪{1}]
sage: Phi[[2],[1,4],[3]].coalgebraic_complement()
-Phi[{3}, {1}, {4}, {2}] + Phi[{3}, {1, 4}, {2}] + Phi[{3}, {4}, {1},
↪{2}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> Phi = WQSym.Phi()
>>> Phi[[Integer(1)],[Integer(2)],[Integer(3),Integer(4)]].coalgebraic_
↪complement()
-Phi[{4}, {3}, {1}, {2}] + Phi[{4}, {3}, {1, 2}] + Phi[{4}, {3}, {2},
↪{1}]
>>> Phi[[Integer(2)],[Integer(1),Integer(4)],[Integer(3)]].coalgebraic_
↪complement()
-Phi[{3}, {1}, {4}, {2}] + Phi[{3}, {1, 4}, {2}] + Phi[{3}, {4}, {1},
↪{2}]
```

star_involution()

Return the image of the element `self` of $WQSym$ under the star involution.

See *WQSymBases.ElementMethods.star_involution()* for a definition of the involution and for examples.

See also

algebraic_complement(), *coalgebraic_complement()*

EXAMPLES:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: Phi = WQSym.Phi()
sage: Phi[[1,2],[5,6],[3,4]].star_involution()
Phi[{3, 4}, {1, 2}, {5, 6}]
sage: Phi[[3], [1, 2], [4]].star_involution()
Phi[{1}, {3, 4}, {2}]
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> Phi = WQSym.Phi()
>>> Phi[[Integer(1),Integer(2)],[Integer(5),Integer(6)],[Integer(3),
↪Integer(4)]].star_involution()
Phi[{3, 4}, {1, 2}, {5, 6}]
>>> Phi[[Integer(3)], [Integer(1), Integer(2)], [Integer(4)]].star_
↪involution()
Phi[{1}, {3, 4}, {2}]
```

coproduct_on_basis(x)

Return the coproduct of `self` on the basis element indexed by the ordered set partition `x`.

The coproduct of the basis element Φ_x indexed by an ordered set partition x of $[n]$ can be computed by the following formula ([NovThi06]):

$$\Delta \Phi_x = \sum \Phi_y \otimes \Phi_z,$$

where the sum ranges over all pairs (y, z) of ordered set partitions y and z such that:

- y and z are ordered set partitions of two complementary subsets of $[n]$;
- x is obtained either by concatenating y and z, or by first concatenating y and z and then merging the two "middle blocks" (i.e., the last block of y and the first block of z); in the latter case, the maximum of the last block of y has to be smaller than the minimum of the first block of z (so that when merging these blocks, their entries don't need to be sorted).

EXAMPLES:

```
sage: Phi = algebras.WQSym(QQ).Phi()

sage: Phi.coproduct(Phi.one())  # indirect doctest
Phi[] # Phi[]
sage: Phi.coproduct( Phi([[1]]) )  # indirect doctest
Phi[] # Phi[{1}] + Phi[{1}] # Phi[]
sage: Phi.coproduct( Phi([[1,2]]) )
Phi[] # Phi[{1, 2}] + Phi[{1}] # Phi[{1}] + Phi[{1, 2}] # Phi[]
sage: Phi.coproduct( Phi([[1], [2]]) )
Phi[] # Phi[{1}, {2}] + Phi[{1}] # Phi[{1}] + Phi[{1}, {2}] # Phi[]
sage: Phi[[1,2],[3],[4]].coproduct()
Phi[] # Phi[{1, 2}, {3}, {4}] + Phi[{1}] # Phi[{1}, {2}, {3}]
 + Phi[{1, 2}] # Phi[{1}, {2}] + Phi[{1, 2}, {3}] # Phi[{1}]
 + Phi[{1, 2}, {3}, {4}] # Phi[]
```

```
>>> from sage.all import *
>>> Phi = algebras.WQSym(QQ).Phi()

>>> Phi.coproduct(Phi.one())  # indirect doctest
```

(continues on next page)

(continued from previous page)

```
Phi[] # Phi[]
>>> Phi.coproduct( Phi([[Integer(1)]]) )  # indirect doctest
Phi[] # Phi[{1}] + Phi[{1}] # Phi[]
>>> Phi.coproduct( Phi([[Integer(1),Integer(2)]]) )
Phi[] # Phi[{1, 2}] + Phi[{1}] # Phi[{1}] + Phi[{1, 2}] # Phi[]
>>> Phi.coproduct( Phi([[Integer(1)], [Integer(2)]]) )
Phi[] # Phi[{1}, {2}] + Phi[{1}] # Phi[{1}] + Phi[{1}, {2}] # Phi[]
>>> Phi[[Integer(1),Integer(2)],[Integer(3)],[Integer(4)]].coproduct()
Phi[] # Phi[{1, 2}, {3}, {4}] + Phi[{1}] # Phi[{1}, {2}, {3}]
 + Phi[{1, 2}] # Phi[{1}, {2}] + Phi[{1, 2}, {3}] # Phi[{1}]
 + Phi[{1, 2}, {3}, {4}] # Phi[]
```

product_on_basis(x, y)

Return the (associative) $*$ product of the basis elements of the Phi basis `self` indexed by the ordered set partitions x and y.

This is obtained by the following algorithm (going back to [NovThi06]):

Let x be an ordered set partition of $[m]$, and y an ordered set partition of $[n]$. Transform x into a list u of all the m elements of $[m]$ by writing out each block of x (in increasing order) and putting bars between each two consecutive blocks; this is called a barred permutation. Do the same for y, but also shift each entry of the resulting barred permutation by m. Let v be the barred permutation of $[m + n] \setminus [m]$ thus obtained. Now, shuffle the two barred permutations u and v (ignoring the bars) in all the $\binom{n+m}{n}$ possible ways. For each shuffle obtained, place bars between some entries of the shuffle, according to the following rule:

- If two consecutive entries of the shuffle both come from u, then place a bar between them if the corresponding entries of u had a bar between them.
- If the first of two consecutive entries of the shuffle comes from v and the second from u, then place a bar between them.

This results in a barred permutation of $[m + n]$. Transform it into an ordered set partition of $[m + n]$, by treating the bars as dividers separating consecutive blocks.

The product $\Phi_x \Phi_y$ is the sum of Φ_p with p ranging over all ordered set partitions obtained this way.

EXAMPLES:

```
sage: A = algebras.WQSym(QQ).Phi()
sage: x = OrderedSetPartition([[1],[2,3]])
sage: y = OrderedSetPartition([[1,2]])
sage: z = OrderedSetPartition([[1,2],[3]])
sage: A.product_on_basis(x, y)
Phi[{1}, {2, 3, 4, 5}] + Phi[{1}, {2, 4}, {3, 5}]
 + Phi[{1}, {2, 4, 5}, {3}] + Phi[{1, 4}, {2, 3, 5}]
 + Phi[{1, 4}, {2, 5}, {3}] + Phi[{1, 4, 5}, {2, 3}]
 + Phi[{4}, {1}, {2, 3, 5}] + Phi[{4}, {1}, {2, 5}, {3}]
 + Phi[{4}, {1, 5}, {2, 3}] + Phi[{4, 5}, {1}, {2, 3}]
sage: A.product_on_basis(x, z)
Phi[{1}, {2, 3, 4, 5}, {6}] + Phi[{1}, {2, 4}, {3, 5}, {6}]
 + Phi[{1}, {2, 4, 5}, {3, 6}] + Phi[{1}, {2, 4, 5}, {6}, {3}]
 + Phi[{1, 4}, {2, 3, 5}, {6}] + Phi[{1, 4}, {2, 5}, {3, 6}]
 + Phi[{1, 4}, {2, 5}, {6}, {3}] + Phi[{1, 4, 5}, {2, 3, 6}]
 + Phi[{1, 4, 5}, {2, 6}, {3}] + Phi[{1, 4, 5}, {6}, {2, 3}]
 + Phi[{4}, {1}, {2, 3, 5}, {6}]
 + Phi[{4}, {1}, {2, 5}, {3, 6}]
```

(continues on next page)

(continued from previous page)

```
 + Phi[{4}, {1}, {2, 5}, {6}, {3}]
 + Phi[{4}, {1, 5}, {2, 3, 6}] + Phi[{4}, {1, 5}, {2, 6}, {3}]
 + Phi[{4}, {1, 5}, {6}, {2, 3}] + Phi[{4, 5}, {1}, {2, 3, 6}]
 + Phi[{4, 5}, {1}, {2, 6}, {3}] + Phi[{4, 5}, {1, 6}, {2, 3}]
 + Phi[{4, 5}, {6}, {1}, {2, 3}]
sage: A.product_on_basis(y, y)
Phi[{1, 2, 3, 4}] + Phi[{1, 3}, {2, 4}] + Phi[{1, 3, 4}, {2}]
 + Phi[{3}, {1, 2, 4}] + Phi[{3}, {1, 4}, {2}]
 + Phi[{3, 4}, {1, 2}]
```

```
>>> from sage.all import *
>>> A = algebras.WQSym(QQ).Phi()
>>> x = OrderedSetPartition([[Integer(1)],[Integer(2),Integer(3)]])
>>> y = OrderedSetPartition([[Integer(1),Integer(2)]])
>>> z = OrderedSetPartition([[Integer(1),Integer(2)],[Integer(3)]])
>>> A.product_on_basis(x, y)
Phi[{1}, {2, 3, 4, 5}] + Phi[{1}, {2, 4}, {3, 5}]
 + Phi[{1}, {2, 4, 5}, {3}] + Phi[{1, 4}, {2, 3, 5}]
 + Phi[{1, 4}, {2, 5}, {3}] + Phi[{1, 4, 5}, {2, 3}]
 + Phi[{4}, {1}, {2, 3, 5}] + Phi[{4}, {1}, {2, 5}, {3}]
 + Phi[{4}, {1, 5}, {2, 3}] + Phi[{4, 5}, {1}, {2, 3}]
>>> A.product_on_basis(x, z)
Phi[{1}, {2, 3, 4, 5}, {6}] + Phi[{1}, {2, 4}, {3, 5}, {6}]
 + Phi[{1}, {2, 4, 5}, {3, 6}] + Phi[{1}, {2, 4, 5}, {6}, {3}]
 + Phi[{1, 4}, {2, 3, 5}, {6}] + Phi[{1, 4}, {2, 5}, {3, 6}]
 + Phi[{1, 4}, {2, 5}, {6}, {3}] + Phi[{1, 4, 5}, {2, 3, 6}]
 + Phi[{1, 4, 5}, {2, 6}, {3}] + Phi[{1, 4, 5}, {6}, {2, 3}]
 + Phi[{4}, {1}, {2, 3, 5}, {6}]
 + Phi[{4}, {1}, {2, 5}, {3, 6}]
 + Phi[{4}, {1}, {2, 5}, {6}, {3}]
 + Phi[{4}, {1, 5}, {2, 3, 6}] + Phi[{4}, {1, 5}, {2, 6}, {3}]
 + Phi[{4}, {1, 5}, {6}, {2, 3}] + Phi[{4, 5}, {1}, {2, 3, 6}]
 + Phi[{4, 5}, {1}, {2, 6}, {3}] + Phi[{4, 5}, {1, 6}, {2, 3}]
 + Phi[{4, 5}, {6}, {1}, {2, 3}]
>>> A.product_on_basis(y, y)
Phi[{1, 2, 3, 4}] + Phi[{1, 3}, {2, 4}] + Phi[{1, 3, 4}, {2}]
 + Phi[{3}, {1, 2, 4}] + Phi[{3}, {1, 4}, {2}]
 + Phi[{3, 4}, {1, 2}]
```

some_elements()

Return some elements of the word quasi-symmetric functions in the Phi basis.

EXAMPLES:

```
sage: Phi = algebras.WQSym(QQ).Phi()
sage: Phi.some_elements()
[Phi[], Phi[{1}], Phi[{1, 2}], Phi[] + 1/2*Phi[{1}]]
```

```
>>> from sage.all import *
>>> Phi = algebras.WQSym(QQ).Phi()
>>> Phi.some_elements()
[Phi[], Phi[{1}], Phi[{1, 2}], Phi[] + 1/2*Phi[{1}]]
```

X

alias of *Characteristic*

a_realization()

Return a particular realization of `self` (the M-basis).

EXAMPLES:

```
sage: WQSym = algebras.WQSym(QQ)
sage: WQSym.a_realization()
Word Quasi-symmetric functions over Rational Field in the Monomial basis
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(QQ)
>>> WQSym.a_realization()
Word Quasi-symmetric functions over Rational Field in the Monomial basis
```

options = Current options for WordQuasiSymmetricFunctions element - display: normal - objects: compositions

5.1.17 Cluster algebras and quivers

- A compendium on the cluster algebra and quiver package in Sage [MS2011]
- *Quiver mutation types*
- *Quiver*
- *ClusterSeed*

5.1.18 ClusterSeed

A *cluster seed* is a pair $(B, \mathbf{x})$ with B being a *skew-symmetrizable* $(n + m) \times n$ *-matrix* and with $\mathbf{x}$ being an n-tuple of *independent elements* in the field of rational functions in n variables.

For the compendium on the cluster algebra and quiver package see [MS2011].

AUTHORS:

- Gregg Musiker: Initial Version
- Christian Stump: Initial Version
- Aram Dermenjian (2015-07-01): Updating ability to not rely solely on clusters
- Jesse Levitt (2015-07-01): Updating ability to not rely solely on clusters

REFERENCES:

- [FZ2007]
- [BDP2013]

See also

For mutation types of cluster seeds, see `sage.combinat.cluster_algebra_quiver.quiver_mutation_type.QuiverMutationType()`. Cluster seeds are closely related to `sage.combinat.cluster_algebra_quiver.quiver.ClusterQuiver()`.

class sage.combinat.cluster_algebra_quiver.cluster_seed.**ClusterSeed**(*data*, *frozen=None*, *is_principal=False*, *user_labels=None*, *user_labels_prefix='x'*)

Bases: SageObject

The *cluster seed* associated to an *exchange matrix*.

INPUT:

- data – can be any of the following:

```
* :class:`QuiverMutationType`
```

  - str – string representing a *QuiverMutationType* or a common quiver type (see Examples)
  - *ClusterQuiver*
  - Matrix – a skew-symmetrizable matrix
  - DiGraph – must be the input data for a quiver
  - List of edges – must be the edge list of a digraph for a quiver

EXAMPLES:

```
sage: S = ClusterSeed(['A',5]); S
A seed for a cluster algebra of rank 5 of type ['A', 5]

sage: S = ClusterSeed(['A',[2,5],1]); S
A seed for a cluster algebra of rank 7 of type ['A', [2, 5], 1]

sage: T = ClusterSeed(S); T
A seed for a cluster algebra of rank 7 of type ['A', [2, 5], 1]

sage: T = ClusterSeed(S._M); T
A seed for a cluster algebra of rank 7

sage: T = ClusterSeed(S.quiver()._digraph); T
A seed for a cluster algebra of rank 7

sage: T = ClusterSeed(S.quiver()._digraph.edges(sort=True)); T
A seed for a cluster algebra of rank 7

sage: S = ClusterSeed(['B',2]); S
A seed for a cluster algebra of rank 2 of type ['B', 2]

sage: S = ClusterSeed(['C',2]); S
A seed for a cluster algebra of rank 2 of type ['B', 2]

sage: S = ClusterSeed(['A', [5,0],1]); S
A seed for a cluster algebra of rank 5 of type ['D', 5]

sage: S = ClusterSeed(['GR',[3,7]]); S
A seed for a cluster algebra of rank 6 of type ['E', 6]
```

(continues on next page)

(continued from previous page)

```
sage: S = ClusterSeed(['F', 4, [2,1]]); S
A seed for a cluster algebra of rank 6 of type ['F', 4, [1, 2]]

sage: S = ClusterSeed(['A',4]); S._use_fpolys
True
sage: S._use_d_vec
True
sage: S._use_g_vec
True
sage: S._use_c_vec
True

sage: S = ClusterSeed(['A', 4]); S.use_fpolys(False); S._use_fpolys
False

sage: S = ClusterSeed(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd']]),
....:                 frozen=['c']); S
A seed for a cluster algebra of rank 4 with 1 frozen variable

sage: S = ClusterSeed(['D', 4], user_labels=[-1, 0, 1, 2]); S
A seed for a cluster algebra of rank 4 of type ['D', 4]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(5)]); S
A seed for a cluster algebra of rank 5 of type ['A', 5]

>>> S = ClusterSeed(['A',[Integer(2),Integer(5)],Integer(1)]); S
A seed for a cluster algebra of rank 7 of type ['A', [2, 5], 1]

>>> T = ClusterSeed(S); T
A seed for a cluster algebra of rank 7 of type ['A', [2, 5], 1]

>>> T = ClusterSeed(S._M); T
A seed for a cluster algebra of rank 7

>>> T = ClusterSeed(S.quiver()._digraph); T
A seed for a cluster algebra of rank 7

>>> T = ClusterSeed(S.quiver()._digraph.edges(sort=True)); T
A seed for a cluster algebra of rank 7

>>> S = ClusterSeed(['B',Integer(2)]); S
A seed for a cluster algebra of rank 2 of type ['B', 2]

>>> S = ClusterSeed(['C',Integer(2)]); S
A seed for a cluster algebra of rank 2 of type ['B', 2]

>>> S = ClusterSeed(['A', [Integer(5),Integer(0)],Integer(1)]); S
A seed for a cluster algebra of rank 5 of type ['D', 5]

>>> S = ClusterSeed(['GR',[Integer(3),Integer(7)]]); S
A seed for a cluster algebra of rank 6 of type ['E', 6]
```

(continues on next page)

(continued from previous page)

```
>>> S = ClusterSeed(['F', Integer(4), [Integer(2),Integer(1)]]); S
A seed for a cluster algebra of rank 6 of type ['F', 4, [1, 2]]

>>> S = ClusterSeed(['A',Integer(4)]); S._use_fpolys
True
>>> S._use_d_vec
True
>>> S._use_g_vec
True
>>> S._use_c_vec
True

>>> S = ClusterSeed(['A', Integer(4)]); S.use_fpolys(False); S._use_fpolys
False

>>> S = ClusterSeed(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd']]),
...                 frozen=['c']); S
A seed for a cluster algebra of rank 4 with 1 frozen variable

>>> S = ClusterSeed(['D', Integer(4)], user_labels=[-Integer(1), Integer(0),
↪Integer(1), Integer(2)]); S
A seed for a cluster algebra of rank 4 of type ['D', 4]
```

LLM_gen_set(*size_limit=-1*)

Produce a list of upper cluster algebra elements corresponding to all vectors in $\{0, 1\}^n$.

INPUT:

- `B` – a skew-symmetric matrix
- `size_limit` – a limit on how many vectors you want the function to return

OUTPUT: an array of elements in the upper cluster algebra

EXAMPLES:

```
sage: B = matrix([[0,1,0],[-1,0,1],[0,-1,0],[1,0,0],[0,1,0],[0,0,1]])
sage: C = ClusterSeed(B)
sage: C.LLM_gen_set()
[1,
 (x1 + x3)/x0,
 (x0*x4 + x2)/x1,
 (x0*x3*x4 + x1*x2 + x2*x3)/(x0*x1),
 (x1*x5 + 1)/x2,
 (x1^2*x5 + x1*x3*x5 + x1 + x3)/(x0*x2),
 (x0*x1*x4*x5 + x0*x4 + x2)/(x1*x2),
 (x0*x1*x3*x4*x5 + x0*x3*x4 + x1*x2 + x2*x3)/(x0*x1*x2)]
```

```
>>> from sage.all import *
>>> B = matrix([[Integer(0),Integer(1),Integer(0)],[-Integer(1),Integer(0),
↪Integer(1)],[Integer(0),-Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> C = ClusterSeed(B)
>>> C.LLM_gen_set()
[1,
 (x1 + x3)/x0,
 (x0*x4 + x2)/x1,
 (x0*x3*x4 + x1*x2 + x2*x3)/(x0*x1),
 (x1*x5 + 1)/x2,
 (x1^2*x5 + x1*x3*x5 + x1 + x3)/(x0*x2),
 (x0*x1*x4*x5 + x0*x4 + x2)/(x1*x2),
 (x0*x1*x3*x4*x5 + x0*x3*x4 + x1*x2 + x2*x3)/(x0*x1*x2)]
```

b_matrix()

Return the B *-matrix* of `self`.

EXAMPLES:

```
sage: ClusterSeed(['A',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: ClusterSeed(['B',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -2  0]

sage: ClusterSeed(['D',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1 -1]
[ 0  1  0  0]
[ 0  1  0  0]

sage: ClusterSeed(QuiverMutationType([['A',2],['B',2]])).b_matrix()
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -2  0]
```

```
>>> from sage.all import *
>>> ClusterSeed(['A',Integer(4)]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

>>> ClusterSeed(['B',Integer(4)]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -2  0]
```

(continues on next page)

(continued from previous page)

```
>>> ClusterSeed(['D',Integer(4)]).b_matrix()
[ 0  1  0  0]
[-1  0 -1 -1]
[ 0  1  0  0]
[ 0  1  0  0]

>>> ClusterSeed(QuiverMutationType([['A',Integer(2)],['B',Integer(2)]])).b_
↪matrix()
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -2  0]
```

b_matrix_class(*depth=+Infinity*, *up_to_equivalence=True*)

Return all B-matrices in the mutation class of `self`.

INPUT:

- `depth` – (default: infinity) integer or infinity, only seeds with distance at most `depth` from `self` are returned
- `up_to_equivalence` – boolean (default: `True`); if `True`, only B-matrices up to equivalence are considered

EXAMPLES:

- for examples see *b_matrix_class_iter()*

b_matrix_class_iter(*depth=+Infinity*, *up_to_equivalence=True*)

Return an iterator through all B-matrices in the mutation class of `self`.

INPUT:

- `depth` – (default: infinity) integer or infinity, only seeds with distance at most `depth` from `self` are returned
- `up_to_equivalence` – boolean (default: `True`); if `True`, only B-matrices up to equivalence are considered

EXAMPLES:

A standard finite type example:

```
sage: S = ClusterSeed(['A',4])
sage: it = S.b_matrix_class_iter()
sage: for T in it: print(T)
[ 0  0  0  1]
[ 0  0  1  1]
[ 0 -1  0  0]
[-1 -1  0  0]
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  1]
[-1  0 -1  0]
[ 0  0  1  1]
[ 0  0  0 -1]
```

(continues on next page)

(continued from previous page)

```
[-1  0  0  0]
[-1  1  0  0]
[ 0  0  0  1]
[ 0  0 -1  1]
[ 0  1  0 -1]
[-1 -1  1  0]
[ 0  0  0  1]
[ 0  0 -1  0]
[ 0  1  0 -1]
[-1  0  1  0]
[ 0  0  0 -1]
[ 0  0 -1  1]
[ 0  1  0 -1]
[ 1 -1  1  0]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)])
>>> it = S.b_matrix_class_iter()
>>> for T in it: print(T)
[ 0  0  0  1]
[ 0  0  1  1]
[ 0 -1  0  0]
[-1 -1  0  0]
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  1]
[-1  0 -1  0]
[ 0  0  1  1]
[ 0  0  0 -1]
[-1  0  0  0]
[-1  1  0  0]
[ 0  0  0  1]
[ 0  0 -1  1]
[ 0  1  0 -1]
[-1 -1  1  0]
[ 0  0  0  1]
[ 0  0 -1  0]
[ 0  1  0 -1]
[-1  0  1  0]
[ 0  0  0 -1]
[ 0  0 -1  1]
[ 0  1  0 -1]
[ 1 -1  1  0]
```

A finite type example with given depth:

```
sage: it = S.b_matrix_class_iter(depth=1)
sage: for T in it: print(T)
[ 0  0  0  1]
[ 0  0  1  1]
[ 0 -1  0  0]
[-1 -1  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  1]
[-1  0 -1  0]
[ 0  0  1  1]
[ 0  0  0 -1]
[-1  0  0  0]
[-1  1  0  0]
```

```
>>> from sage.all import *
>>> it = S.b_matrix_class_iter(depth=Integer(1))
>>> for T in it: print(T)
[ 0  0  0  1]
[ 0  0  1  1]
[ 0 -1  0  0]
[-1 -1  0  0]
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  1]
[-1  0 -1  0]
[ 0  0  1  1]
[ 0  0  0 -1]
[-1  0  0  0]
[-1  1  0  0]
```

Finite type example not considered up to equivalence:

```
sage: S = ClusterSeed(['A',3])
sage: it = S.b_matrix_class_iter(up_to_equivalence=False)
sage: b_matrix_class = list(it)
sage: len(b_matrix_class)
14
sage: b_matrix_class[0]
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> it = S.b_matrix_class_iter(up_to_equivalence=False)
>>> b_matrix_class = list(it)
>>> len(b_matrix_class)
14
>>> b_matrix_class[Integer(0)]
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
```

Infinite (but finite mutation) type example:

```
sage: S = ClusterSeed(['A',[1,2],1])
sage: it = S.b_matrix_class_iter()
```

(continues on next page)

(continued from previous page)

```
sage: for T in it: print(T)
[ 0  1  1]
[-1  0  1]
[-1 -1  0]
[ 0 -2  1]
[ 2  0 -1]
[-1  1  0]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',[Integer(1),Integer(2)],Integer(1)])
>>> it = S.b_matrix_class_iter()
>>> for T in it: print(T)
[ 0  1  1]
[-1  0  1]
[-1 -1  0]
[ 0 -2  1]
[ 2  0 -1]
[-1  1  0]
```

Infinite mutation type example:

```
sage: S = ClusterSeed(['E',10])
sage: it = S.b_matrix_class_iter(depth=3)
sage: len ([T for T in it])
266
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['E',Integer(10)])
>>> it = S.b_matrix_class_iter(depth=Integer(3))
>>> len ([T for T in it])
266
```

For a cluster seed from an arbitrarily labelled digraph:

```
sage: dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
sage: S = ClusterSeed(dg, frozen=['b'])
sage: S.b_matrix_class()
[
[ 0  0]  [ 0  0]  [0 0]
[ 0  0]  [ 0  0]  [0 0]
[-1  1], [-1 -1], [1 1]
]
```

```
>>> from sage.all import *
>>> dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
>>> S = ClusterSeed(dg, frozen=['b'])
>>> S.b_matrix_class()
[
[ 0  0]  [ 0  0]  [0 0]
[ 0  0]  [ 0  0]  [0 0]
[-1  1], [-1 -1], [1 1]
]
```

c_matrix(*show_warnings=True*)

Return all *c-vectors* of `self`.

> **Warning**
>
> This method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: S.c_matrix()
[ 1  0  0]
[ 0  0 -1]
[ 0 -1  0]

sage: S = ClusterSeed(['A',4])
sage: S.use_g_vectors(False); S.use_fpolys(False)
sage: S.use_c_vectors(False); S.use_d_vectors(False); S.track_mutations(False)
sage: S.c_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate c-vectors. Need to use c vectors.
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> S.c_matrix()
[ 1  0  0]
[ 0  0 -1]
[ 0 -1  0]

>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_g_vectors(False); S.use_fpolys(False)
>>> S.use_c_vectors(False); S.use_d_vectors(False); S.track_mutations(False)
>>> S.c_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate c-vectors. Need to use c vectors.
```

c_vector(*k*)

Return the `k`-th *c-vector* of `self`. It is obtained as the `k`-th column vector of the bottom part of the `B`-matrix of `self`.

> **Warning**
>
> This method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: [S.c_vector(k) for k in range(3)]
[(1, 0, 0), (0, 0, -1), (0, -1, 0)]

sage: S = ClusterSeed(Matrix([[0,1],[-1,0],[1,0],[-1,1]])); S
A seed for a cluster algebra of rank 2 with 2 frozen variables
sage: S.c_vector(0)
(1, 0)

sage: S = ClusterSeed(Matrix([[0,1],[-1,0],[1,0],[-1,1]]))
sage: S.use_c_vectors(bot_is_c=True); S
A seed for a cluster algebra of rank 2 with 2 frozen variables
sage: S.c_vector(0)
(1, -1)
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> [S.c_vector(k) for k in range(Integer(3))]
[(1, 0, 0), (0, 0, -1), (0, -1, 0)]

>>> S = ClusterSeed(Matrix([[Integer(0),Integer(1)],[-Integer(1),Integer(0)],
↪[Integer(1),Integer(0)],[-Integer(1),Integer(1)]])); S
A seed for a cluster algebra of rank 2 with 2 frozen variables
>>> S.c_vector(Integer(0))
(1, 0)

>>> S = ClusterSeed(Matrix([[Integer(0),Integer(1)],[-Integer(1),Integer(0)],
↪[Integer(1),Integer(0)],[-Integer(1),Integer(1)]]))
>>> S.use_c_vectors(bot_is_c=True); S
A seed for a cluster algebra of rank 2 with 2 frozen variables
>>> S.c_vector(Integer(0))
(1, -1)
```

cluster()

Return a copy of the *cluster* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.cluster()
[x0, x1, x2]

sage: S.mutate(1)
sage: S.cluster()
[x0, (x0*x2 + 1)/x1, x2]

sage: S.mutate(2)
sage: S.cluster()
[x0, (x0*x2 + 1)/x1, (x0*x2 + x1 + 1)/(x1*x2)]

sage: S.mutate([2,1])
```

(continues on next page)

(continued from previous page)

```
sage: S.cluster()
[x0, x1, x2]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.cluster()
[x0, x1, x2]

>>> S.mutate(Integer(1))
>>> S.cluster()
[x0, (x0*x2 + 1)/x1, x2]

>>> S.mutate(Integer(2))
>>> S.cluster()
[x0, (x0*x2 + 1)/x1, (x0*x2 + x1 + 1)/(x1*x2)]

>>> S.mutate([Integer(2),Integer(1)])
>>> S.cluster()
[x0, x1, x2]
```

cluster_class(*depth=+Infinity*, *show_depth=False*, *up_to_equivalence=True*)

Return the cluster class of `self` with respect to certain constraints.

INPUT:

- `depth` – (default: infinity) integer, only seeds with distance at most `depth` from `self` are returned
- `return_depth` – (default: `False`) if `True`, ignored if `depth` is set; returns the depth of the mutation class, i.e., the maximal distance from `self` of an element in the mutation class
- `up_to_equivalence` – (default: `True`) if `True`, only clusters up to equivalence are considered

EXAMPLES:

- for examples see *cluster_class_iter()*

cluster_class_iter(*depth=+Infinity*, *show_depth=False*, *up_to_equivalence=True*)

Return an iterator through all clusters in the mutation class of `self`.

INPUT:

- `depth` – (default: infinity) integer or infinity, only seeds with distance at most `depth` from `self` are returned
- `show_depth` – boolean (default: `False`); if `True`, ignored if `depth` is set; returns the depth of the mutation class, i.e., the maximal distance from `self` of an element in the mutation class
- `up_to_equivalence` – boolean (default: `True`); if `True`, only clusters up to equivalence are considered

EXAMPLES:

A standard finite type example:

```
sage: S = ClusterSeed(['A',3])
sage: it = S.cluster_class_iter()
sage: cluster_class = list(it)
sage: len(cluster_class)
```

(continues on next page)

(continued from previous page)

```
14
sage: cluster_class[0]
[x0, x1, x2]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> it = S.cluster_class_iter()
>>> cluster_class = list(it)
>>> len(cluster_class)
14
>>> cluster_class[Integer(0)]
[x0, x1, x2]
```

A finite type example with given depth:

```
sage: it = S.cluster_class_iter(depth=1)
sage: for T in it: print(T)
[x0, x1, x2]
[x0, x1, (x1 + 1)/x2]
[x0, (x0*x2 + 1)/x1, x2]
[(x1 + 1)/x0, x1, x2]
```

```
>>> from sage.all import *
>>> it = S.cluster_class_iter(depth=Integer(1))
>>> for T in it: print(T)
[x0, x1, x2]
[x0, x1, (x1 + 1)/x2]
[x0, (x0*x2 + 1)/x1, x2]
[(x1 + 1)/x0, x1, x2]
```

A finite type example where the depth is returned while computing:

```
sage: it = S.cluster_class_iter(show_depth=True)
sage: _ = list(it)
Depth: 0      found: 1          Time: ... s
Depth: 1      found: 4          Time: ... s
Depth: 2      found: 9          Time: ... s
Depth: 3      found: 13         Time: ... s
Depth: 4      found: 14         Time: ... s
```

```
>>> from sage.all import *
>>> it = S.cluster_class_iter(show_depth=True)
>>> _ = list(it)
Depth: 0      found: 1          Time: ... s
Depth: 1      found: 4          Time: ... s
Depth: 2      found: 9          Time: ... s
Depth: 3      found: 13         Time: ... s
Depth: 4      found: 14         Time: ... s
```

Finite type examples not considered up to equivalence:

```
sage: it = S.cluster_class_iter(up_to_equivalence=False)
sage: len([T for T in it])
84

sage: it = ClusterSeed(['A',2]).cluster_class_iter(up_to_equivalence=False)
sage: cluster_class = list(it)
sage: len(cluster_class)
10
sage: cluster_class[0]
[x0, x1]
sage: cluster_class[-1]
[x1, x0]
```

```
>>> from sage.all import *
>>> it = S.cluster_class_iter(up_to_equivalence=False)
>>> len([T for T in it])
84

>>> it = ClusterSeed(['A',Integer(2)]).cluster_class_iter(up_to_
↪equivalence=False)
>>> cluster_class = list(it)
>>> len(cluster_class)
10
>>> cluster_class[Integer(0)]
[x0, x1]
>>> cluster_class[-Integer(1)]
[x1, x0]
```

Infinite type examples:

```
sage: S = ClusterSeed(['A',[1,1],1])
sage: it = S.cluster_class_iter()
sage: next(it)
[x0, x1]
sage: next(it)
[x0, (x0^2 + 1)/x1]
sage: next(it)
[(x1^2 + 1)/x0, x1]
sage: next(it)
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^2 + 1)/x1]
sage: next(it)
[(x1^2 + 1)/x0, (x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)]

sage: it = S.cluster_class_iter(depth=3)
sage: for T in it: print(T)
[x0, x1]
[x0, (x0^2 + 1)/x1]
[(x1^2 + 1)/x0, x1]
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^2 + 1)/x1]
[(x1^2 + 1)/x0, (x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)]
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^6 + 3*x0^4 + 2*x0^2*x1^2 + x1^4 +␣
↪3*x0^2 + 2*x1^2 + 1)/(x0^2*x1^3)]
[(x1^6 + x0^4 + 2*x0^2*x1^2 + 3*x1^4 + 2*x0^2 + 3*x1^2 + 1)/(x0^3*x1^2), (x1^
```

(continues on next page)

(continued from previous page)

```
↪4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',[Integer(1),Integer(1)],Integer(1)])
>>> it = S.cluster_class_iter()
>>> next(it)
[x0, x1]
>>> next(it)
[x0, (x0^2 + 1)/x1]
>>> next(it)
[(x1^2 + 1)/x0, x1]
>>> next(it)
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^2 + 1)/x1]
>>> next(it)
[(x1^2 + 1)/x0, (x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)]

>>> it = S.cluster_class_iter(depth=Integer(3))
>>> for T in it: print(T)
[x0, x1]
[x0, (x0^2 + 1)/x1]
[(x1^2 + 1)/x0, x1]
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^2 + 1)/x1]
[(x1^2 + 1)/x0, (x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)]
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^6 + 3*x0^4 + 2*x0^2*x1^2 + x1^4 +␣
↪3*x0^2 + 2*x1^2 + 1)/(x0^2*x1^3)]
[(x1^6 + x0^4 + 2*x0^2*x1^2 + 3*x1^4 + 2*x0^2 + 3*x1^2 + 1)/(x0^3*x1^2), (x1^
↪4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)]
```

For a cluster seed from an arbitrarily labelled digraph:

```
sage: dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
sage: S = ClusterSeed(dg, frozen=['b'])
sage: S.cluster_class()
[[a, c], [a, (b + 1)/c], [(b + 1)/a, c], [(b + 1)/a, (b + 1)/c]]

sage: S2 = ClusterSeed(dg, frozen=[])
sage: S2.cluster_class()[0]
[a, b, c]
```

```
>>> from sage.all import *
>>> dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
>>> S = ClusterSeed(dg, frozen=['b'])
>>> S.cluster_class()
[[a, c], [a, (b + 1)/c], [(b + 1)/a, c], [(b + 1)/a, (b + 1)/c]]

>>> S2 = ClusterSeed(dg, frozen=[])
>>> S2.cluster_class()[Integer(0)]
[a, b, c]
```

cluster_index(*cluster_str*)

Return the index of a cluster if `use_fpolys` is on.

INPUT:

- `cluster_str` – the string to look for in the cluster

OUTPUT: integer or `None` if the string is not a cluster variable

EXAMPLES:

```
sage: S = ClusterSeed(['A', 4], user_labels=['x', 'y', 'z', 'w']); S.mutate('x
↪')
sage: S.cluster_index('x')
sage: S.cluster_index('(y+1)/x')
0
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A', Integer(4)], user_labels=['x', 'y', 'z', 'w']); S.
↪mutate('x')
>>> S.cluster_index('x')
>>> S.cluster_index('(y+1)/x')
0
```

cluster_variable(*k*)

Generates a cluster variable using F-polynomials.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutate([0,1])
sage: S.cluster_variable(0)
(x1 + 1)/x0
sage: S.cluster_variable(1)
(x0*x2 + x1 + 1)/(x0*x1)
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.mutate([Integer(0),Integer(1)])
>>> S.cluster_variable(Integer(0))
(x1 + 1)/x0
>>> S.cluster_variable(Integer(1))
(x0*x2 + x1 + 1)/(x0*x1)
```

coefficient(*k*)

Return the *coefficient* of `self` at index `k`, or vertex `k` if `k` is not an index.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: [S.coefficient(k) for k in range(3)]
[y0, 1/y2, 1/y1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> [S.coefficient(k) for k in range(Integer(3))]
[y0, 1/y2, 1/y1]
```

coefficients()

Return all *coefficients* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: S.coefficients()
[y0, 1/y2, 1/y1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> S.coefficients()
[y0, 1/y2, 1/y1]
```

d_matrix(*show_warnings=True*)

Return the matrix of *d-vectors* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]); S.d_matrix()
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
sage: S.mutate([1,2,1,0,1,3]); S.d_matrix()
[1 1 0 1]
[1 1 1 1]
[1 0 1 1]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)]); S.d_matrix()
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
>>> S.mutate([Integer(1),Integer(2),Integer(1),Integer(0),Integer(1),
↪Integer(3)]); S.d_matrix()
[1 1 0 1]
[1 1 1 1]
[1 0 1 1]
[0 0 0 1]
```

d_vector(*k*)

Return the `k`-th *d-vector* of `self`. This is the exponent vector of the denominator of the `k`-th cluster variable.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutate([2,1,2])
sage: [S.d_vector(k) for k in range(3)]
[(-1, 0, 0), (0, 1, 1), (0, 1, 0)]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> [S.d_vector(k) for k in range(Integer(3))]
[(-1, 0, 0), (0, 1, 1), (0, 1, 0)]
```

exchangeable_part()

Return the restriction to the principal part (i.e. the exchangeable variables) of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',4])
sage: T = ClusterSeed(S.quiver().digraph().edges(sort=True), frozen=[3])
sage: T.quiver().digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: T.exchangeable_part().quiver().digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1))]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)])
>>> T = ClusterSeed(S.quiver().digraph().edges(sort=True),␣
↪frozen=[Integer(3)])
>>> T.quiver().digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

>>> T.exchangeable_part().quiver().digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1))]
```

f_polynomial(k)

Return the `k`-th *F-polynomial* of `self`. It is obtained from the `k`-th cluster variable by setting all x_i to 1.

Warning

This method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: [S.f_polynomial(k) for k in range(3)]
[1, y1*y2 + y2 + 1, y1 + 1]

sage: S = ClusterSeed(Matrix([[0,1],[-1,0],[1,0],[-1,1]]))
sage: S.use_c_vectors(bot_is_c=True); S
A seed for a cluster algebra of rank 2 with 2 frozen variables
sage: T = ClusterSeed(Matrix([[0,1],[-1,0]])).principal_extension(); T
A seed for a cluster algebra of rank 2 with principal coefficients
sage: S.mutate(0)
sage: T.mutate(0)
sage: S.f_polynomials()
```

(continues on next page)

(continued from previous page)

```
[y0 + y1, 1]
sage: T.f_polynomials()
[y0 + 1, 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> [S.f_polynomial(k) for k in range(Integer(3))]
[1, y1*y2 + y2 + 1, y1 + 1]

>>> S = ClusterSeed(Matrix([[Integer(0),Integer(1)],[-Integer(1),Integer(0)],
↪[Integer(1),Integer(0)],[-Integer(1),Integer(1)]]))
>>> S.use_c_vectors(bot_is_c=True); S
A seed for a cluster algebra of rank 2 with 2 frozen variables
>>> T = ClusterSeed(Matrix([[Integer(0),Integer(1)],[-Integer(1),
↪Integer(0)]])).principal_extension(); T
A seed for a cluster algebra of rank 2 with principal coefficients
>>> S.mutate(Integer(0))
>>> T.mutate(Integer(0))
>>> S.f_polynomials()
[y0 + y1, 1]
>>> T.f_polynomials()
[y0 + 1, 1]
```

f_polynomials()

Return all *F-polynomials* of `self`. These are obtained from the cluster variables by setting all x_i's to 1.

> **Warning**
>
> This method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: S.f_polynomials()
[1, y1*y2 + y2 + 1, y1 + 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> S.f_polynomials()
[1, y1*y2 + y2 + 1, y1 + 1]
```

find_upper_bound(*verbose=False*)

Return the upper bound of the given cluster algebra as a quotient ring.

The upper bound is the intersection of the Laurent polynomial rings of the initial cluster and its neighboring clusters. As such, it always contains both the cluster algebra and the upper cluster algebra. This function uses the algorithm from [MM2015].

When the initial seed is totally coprime (for example, when the unfrozen part of the exchange matrix has full rank), the upper bound is equal to the upper cluster algebra by [BFZ2005].

Warning: The computation time grows rapidly with the size of the seed and the number of steps. For most seeds larger than four vertices, the algorithm may take an infeasible amount of time. Additionally, it will run forever without terminating whenever the upper bound is infinitely-generated (such as the example in [Spe2013]).

INPUT:

- `verbose` – boolean (default: `False`); if `True`, prints output during the computation

EXAMPLES:

- finite type:

```
sage: S = ClusterSeed(['A',3])
sage: S.find_upper_bound()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x0p, x1p, x2p, z0
 over Rational Field
 by the ideal (x0*x0p - x1 - 1, x1*x1p - x0*x2 - 1, x2*x2p - x1 - 1,
              x0*z0 - x2p, x1*z0 + z0 - x0p*x2p, x2*z0 - x0p,
              x1p*z0 + z0 - x0p*x1p*x2p + x1 + 1)
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.find_upper_bound()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x0p, x1p, x2p, z0
 over Rational Field
 by the ideal (x0*x0p - x1 - 1, x1*x1p - x0*x2 - 1, x2*x2p - x1 - 1,
              x0*z0 - x2p, x1*z0 + z0 - x0p*x2p, x2*z0 - x0p,
              x1p*z0 + z0 - x0p*x1p*x2p + x1 + 1)
```

- Markov:

```
sage: B = matrix([[0,2,-2],[-2,0,2],[2,-2,0]])
sage: S = ClusterSeed(B)
sage: S.find_upper_bound()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x0p, x1p, x2p, z0
 over Rational Field
 by the ideal (x0*x0p - x2^2 - x1^2, x1*x1p - x2^2 - x0^2, x2*x2p - x1^2 -
↪ x0^2,
              x0p*x1p*x2p - x0*x1*x2p - x0*x2*x1p - x1*x2*x0p -␣
↪2*x0*x1*x2,
              x0^3*z0 - x1p*x2p + x1*x2, x0*x1*z0 - x2p - x2,
              x1^3*z0 - x0p*x2p + x0*x2, x0*x2*z0 - x1p - x1,
              x1*x2*z0 - x0p - x0, x2^3*z0 - x0p*x1p + x0*x1)
```

```
>>> from sage.all import *
>>> B = matrix([[Integer(0),Integer(2),-Integer(2)],[-Integer(2),
↪Integer(0),Integer(2)],[Integer(2),-Integer(2),Integer(0)]])
```

(continues on next page)

(continued from previous page)

```
>>> S = ClusterSeed(B)
>>> S.find_upper_bound()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x0p, x1p, x2p, z0
 over Rational Field
 by the ideal (x0*x0p - x2^2 - x1^2, x1*x1p - x2^2 - x0^2, x2*x2p - x1^2 -
↪ x0^2,
               x0p*x1p*x2p - x0*x1*x2p - x0*x2*x1p - x1*x2*x0p -␣
↪2*x0*x1*x2,
               x0^3*z0 - x1p*x2p + x1*x2, x0*x1*z0 - x2p - x2,
               x1^3*z0 - x0p*x2p + x0*x2, x0*x2*z0 - x1p - x1,
               x1*x2*z0 - x0p - x0, x2^3*z0 - x0p*x1p + x0*x1)
```

first_green_vertex()

Return the first green vertex of `self`.

A vertex is defined to be green if its c-vector has all nonpositive entries. More information on green vertices can be found at [BDP2013]

EXAMPLES:

```
sage: ClusterSeed(['A',3]).principal_extension().first_green_vertex()
0

sage: ClusterSeed(['A',[3,3],1]).principal_extension().first_green_vertex()
0
```

```
>>> from sage.all import *
>>> ClusterSeed(['A',Integer(3)]).principal_extension().first_green_vertex()
0

>>> ClusterSeed(['A',[Integer(3),Integer(3)],Integer(1)]).principal_
↪extension().first_green_vertex()
0
```

first_red_vertex()

Return the first red vertex of `self`.

A vertex is defined to be red if its c-vector has all nonnegative entries. More information on red vertices can be found at [BDP2013].

EXAMPLES:

```
sage: ClusterSeed(['A',3]).principal_extension().first_red_vertex()

sage: ClusterSeed(['A',[3,3],1]).principal_extension().first_red_vertex()

sage: Q = ClusterSeed(['A',[3,3],1]).principal_extension()
sage: Q.mutate(1)
sage: Q.first_red_vertex()
1
```

```
>>> from sage.all import *
>>> ClusterSeed(['A',Integer(3)]).principal_extension().first_red_vertex()
```

(continues on next page)

(continued from previous page)

```
>>> ClusterSeed(['A',[Integer(3),Integer(3)],Integer(1)]).principal_
↪extension().first_red_vertex()

>>> Q = ClusterSeed(['A',[Integer(3),Integer(3)],Integer(1)]).principal_
↪extension()
>>> Q.mutate(Integer(1))
>>> Q.first_red_vertex()
1
```

first_urban_renewal()

Return the first urban renewal vertex.

An urban renewal vertex is one in which there are two arrows pointing toward the vertex and two arrows pointing away.

EXAMPLES:

```
sage: G = ClusterSeed(['GR',[4,9]]); G.first_urban_renewal()
5
```

```
>>> from sage.all import *
>>> G = ClusterSeed(['GR',[Integer(4),Integer(9)]]); G.first_urban_renewal()
5
```

free_vertices()

Return the list of *exchangeable vertices* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd
↪']]),
....:                         frozen=['b', 'd'])
sage: S.free_vertices()
['a', 'c', 'e']

sage: S = ClusterSeed(DiGraph([[5, 'b']]))
sage: S.free_vertices()
[5, 'b']
```

```
>>> from sage.all import *
>>> S = ClusterSeed(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd']]),
...                         frozen=['b', 'd'])
>>> S.free_vertices()
['a', 'c', 'e']

>>> S = ClusterSeed(DiGraph([[Integer(5), 'b']]))
>>> S.free_vertices()
[5, 'b']
```

frozen_vertices()

Return the list of *frozen vertices* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd
↪']]),
....:                         frozen=['b', 'd'])
sage: sorted(S.frozen_vertices())
['b', 'd']
```

```
>>> from sage.all import *
>>> S = ClusterSeed(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd']]),
...                         frozen=['b', 'd'])
>>> sorted(S.frozen_vertices())
['b', 'd']
```

g_matrix(*show_warnings=True*)

Return the matrix of all *g-vectors* of `self`. These are the degree vectors of the cluster variables after setting all y_i's to 0.

> **Warning**
>
> This method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: S.g_matrix()
[ 1  0  0]
[ 0  0 -1]
[ 0 -1  0]

sage: S = ClusterSeed(['A',3])
sage: S.mutate([0,1])
sage: S.g_matrix()
[-1 -1  0]
[ 1  0  0]
[ 0  0  1]

sage: S = ClusterSeed(['A',4])
sage: S.use_g_vectors(False); S.use_fpolys(False); S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: S = ClusterSeed(['A',4])
sage: S.use_g_vectors(False); S.use_c_vectors(False); S.use_fpolys(False)
sage: S.track_mutations(False); S.g_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate g-vectors. Need to use g vectors.
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> S.g_matrix()
[ 1  0  0]
[ 0  0 -1]
[ 0 -1  0]

>>> S = ClusterSeed(['A',Integer(3)])
>>> S.mutate([Integer(0),Integer(1)])
>>> S.g_matrix()
[-1 -1  0]
[ 1  0  0]
[ 0  0  1]

>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_g_vectors(False); S.use_fpolys(False); S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_g_vectors(False); S.use_c_vectors(False); S.use_fpolys(False)
>>> S.track_mutations(False); S.g_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate g-vectors. Need to use g vectors.
```

g_vector(*k*)

Return the k-th *g-vector* of self. This is the degree vector of the k-th cluster variable after setting all y_i's to 0.

> **Warning**
>
> This method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: [S.g_vector(k) for k in range(3)]
[(1, 0, 0), (0, 0, -1), (0, -1, 0)]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutate([Integer(2),Integer(1),Integer(2)])
>>> [S.g_vector(k) for k in range(Integer(3))]
[(1, 0, 0), (0, 0, -1), (0, -1, 0)]
```

get_upper_cluster_algebra_element(*a*)

Compute an element in the upper cluster algebra of B corresponding to the vector $a \in \mathbf{Z}^n$.

See [LLM2014] for more details.

INPUT:

- B – a skew-symmetric matrix. Must have the same number of columns as the length of the vectors in vd
- a – a vector in $\mathbf{Z}^n$ where n is the number of columns in B

OUTPUT: an element in the upper cluster algebra. Depending on the input it may or may not be irreducible

EXAMPLES:

```
sage: B = matrix([[0,3,-3],[-3,0,3],[3,-3,0],[1,0,0],[0,1,0],[0,0,1]])
sage: C = ClusterSeed(B)
sage: C.get_upper_cluster_algebra_element([1,1,0])
(x0^3*x2^3*x3*x4 + x2^6*x3 + x1^3*x2^3)/(x0*x1)
sage: C.get_upper_cluster_algebra_element([1,1,1])
x0^2*x1^2*x2^2*x3*x4*x5 + x0^2*x1^2*x2^2

sage: B = matrix([[0,3,0],[-3,0,3],[0,-3,0]])
sage: C = ClusterSeed(B)
sage: C.get_upper_cluster_algebra_element([1,1,0])
(x1^3*x2^3 + x0^3 + x2^3)/(x0*x1)
sage: C.get_upper_cluster_algebra_element([1,1,1])
(x0^3*x1^3 + x1^3*x2^3 + x0^3 + x2^3)/(x0*x1*x2)

sage: B = matrix([[0,2],[-3,0],[4,-5]])
sage: C = ClusterSeed(B)
sage: C.get_upper_cluster_algebra_element([1,1])
(x2^9 + x1^3*x2^5 + x0^2*x2^4)/(x0*x1)

sage: B = matrix([[0,3,-5],[-3,0,4],[5,-4,0]])
sage: C = ClusterSeed(B)
sage: C.get_upper_cluster_algebra_element([1,1,1])
x0^4*x1^2*x2^3 + x0^2*x1^3*x2^4
```

```
>>> from sage.all import *
>>> B = matrix([[Integer(0),Integer(3),-Integer(3)],[-Integer(3),Integer(0),
↪Integer(3)],[Integer(3),-Integer(3),Integer(0)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(1)]])
>>> C = ClusterSeed(B)
>>> C.get_upper_cluster_algebra_element([Integer(1),Integer(1),Integer(0)])
(x0^3*x2^3*x3*x4 + x2^6*x3 + x1^3*x2^3)/(x0*x1)
>>> C.get_upper_cluster_algebra_element([Integer(1),Integer(1),Integer(1)])
x0^2*x1^2*x2^2*x3*x4*x5 + x0^2*x1^2*x2^2

>>> B = matrix([[Integer(0),Integer(3),Integer(0)],[-Integer(3),Integer(0),
↪Integer(3)],[Integer(0),-Integer(3),Integer(0)]])
>>> C = ClusterSeed(B)
>>> C.get_upper_cluster_algebra_element([Integer(1),Integer(1),Integer(0)])
(x1^3*x2^3 + x0^3 + x2^3)/(x0*x1)
>>> C.get_upper_cluster_algebra_element([Integer(1),Integer(1),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
(x0^3*x1^3 + x1^3*x2^3 + x0^3 + x2^3)/(x0*x1*x2)

>>> B = matrix([[Integer(0),Integer(2)],[-Integer(3),Integer(0)],[Integer(4),-
↪Integer(5)]])
>>> C = ClusterSeed(B)
>>> C.get_upper_cluster_algebra_element([Integer(1),Integer(1)])
(x2^9 + x1^3*x2^5 + x0^2*x2^4)/(x0*x1)

>>> B = matrix([[Integer(0),Integer(3),-Integer(5)],[-Integer(3),Integer(0),
↪Integer(4)],[Integer(5),-Integer(4),Integer(0)]])
>>> C = ClusterSeed(B)
>>> C.get_upper_cluster_algebra_element([Integer(1),Integer(1),Integer(1)])
x0^4*x1^2*x2^3 + x0^2*x1^3*x2^4
```

greedy (*a1*, *a2*, *algorithm='by_recursion'*)

Return the greedy element $x[a_1, a_2]$ assuming that `self` is rank two.

The third input can be `'by_recursion'`, `'by_combinatorics'`, or `'just_numbers'` to specify if the user wants the element computed by the recurrence, combinatorial formula, or wants to set x_1 and x_2 to be one.

See [LLZ2014] for more details.

EXAMPLES:

```
sage: S = ClusterSeed(['R2', [3, 3]])
sage: S.greedy(4, 4)
(x0^12 + x1^12 + 4*x0^9 + 4*x1^9 + 6*x0^6
 + 4*x0^3*x1^3 + 6*x1^6 + 4*x0^3 + 4*x1^3 + 1)/(x0^4*x1^4)
sage: S.greedy(4, 4, 'by_combinatorics')
(x0^12 + x1^12 + 4*x0^9 + 4*x1^9 + 6*x0^6
 + 4*x0^3*x1^3 + 6*x1^6 + 4*x0^3 + 4*x1^3 + 1)/(x0^4*x1^4)
sage: S.greedy(4, 4, 'just_numbers')
35
sage: S = ClusterSeed(['R2', [2, 2]])
sage: S.greedy(1, 2)
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
sage: S.greedy(1, 2, 'by_combinatorics')
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['R2', [Integer(3), Integer(3)]])
>>> S.greedy(Integer(4), Integer(4))
(x0^12 + x1^12 + 4*x0^9 + 4*x1^9 + 6*x0^6
 + 4*x0^3*x1^3 + 6*x1^6 + 4*x0^3 + 4*x1^3 + 1)/(x0^4*x1^4)
>>> S.greedy(Integer(4), Integer(4), 'by_combinatorics')
(x0^12 + x1^12 + 4*x0^9 + 4*x1^9 + 6*x0^6
 + 4*x0^3*x1^3 + 6*x1^6 + 4*x0^3 + 4*x1^3 + 1)/(x0^4*x1^4)
>>> S.greedy(Integer(4), Integer(4), 'just_numbers')
35
>>> S = ClusterSeed(['R2', [Integer(2), Integer(2)]])
>>> S.greedy(Integer(1), Integer(2))
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
```

(continues on next page)

(continued from previous page)

```
>>> S.greedy(Integer(1), Integer(2), 'by_combinatorics')
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
```

green_vertices()

Return the list of green vertices of `self`.

A vertex is defined to be green if its c-vector has all nonpositive entries. More information on green vertices can be found at [BDP2013]

OUTPUT: the green vertices as a list of integers

EXAMPLES:

```
sage: ClusterSeed(['A',3]).principal_extension().green_vertices()
[0, 1, 2]

sage: ClusterSeed(['A',[3,3],1]).principal_extension().green_vertices()
[0, 1, 2, 3, 4, 5]
```

```
>>> from sage.all import *
>>> ClusterSeed(['A',Integer(3)]).principal_extension().green_vertices()
[0, 1, 2]

>>> ClusterSeed(['A',[Integer(3),Integer(3)],Integer(1)]).principal_
↪extension().green_vertices()
[0, 1, 2, 3, 4, 5]
```

ground_field()

Return the *ground field* of the cluster of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.ground_field()
Multivariate Polynomial Ring in x0, x1, x2, y0, y1, y2 over Rational Field
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.ground_field()
Multivariate Polynomial Ring in x0, x1, x2, y0, y1, y2 over Rational Field
```

highest_degree_denominator(*filter=None*)

Return the vertex of the cluster polynomial with highest degree in the denominator.

INPUT:

- `filter` – list or iterable

OUTPUT: integer

EXAMPLES:

```
sage: B = matrix([[0,-1,0,-1,1,1], [1,0,1,0,-1,-1], [0,-1,0,-1,1,1],
....:             [1,0,1,0,-1,-1], [-1,1,-1,1,0,0], [-1,1,-1,1,0,0]])
sage: C = ClusterSeed(B).principal_extension(); C.mutate([0,1,2,4,3,2,5,4,3])
```

(continues on next page)

(continued from previous page)

```
sage: C.highest_degree_denominator()
5
```

```
>>> from sage.all import *
>>> B = matrix([[Integer(0),-Integer(1),Integer(0),-Integer(1),Integer(1),
↪Integer(1)], [Integer(1),Integer(0),Integer(1),Integer(0),-Integer(1),-
↪Integer(1)], [Integer(0),-Integer(1),Integer(0),-Integer(1),Integer(1),
↪Integer(1)],
...             [Integer(1),Integer(0),Integer(1),Integer(0),-Integer(1),-
↪Integer(1)], [-Integer(1),Integer(1),-Integer(1),Integer(1),Integer(0),
↪Integer(0)], [-Integer(1),Integer(1),-Integer(1),Integer(1),Integer(0),
↪Integer(0)]])
>>> C = ClusterSeed(B).principal_extension(); C.mutate([Integer(0),Integer(1),
↪Integer(2),Integer(4),Integer(3),Integer(2),Integer(5),Integer(4),
↪Integer(3)])
>>> C.highest_degree_denominator()
5
```

interact (*fig_size=1*, *circular=True*)

Start an interactive window for cluster seed mutations.

Only in *Jupyter notebook mode*.

INPUT:

- `fig_size` – (default: 1) factor by which the size of the plot is multiplied
- `circular` – boolean (default: `True`); if `True`, the circular plot is chosen, otherwise >>spring<< is used

is_acyclic ()

Return `True` iff `self` is acyclic (i.e., if the underlying quiver is acyclic).

EXAMPLES:

```
sage: ClusterSeed(['A',4]).is_acyclic()
True

sage: ClusterSeed(['A',[2,1],1]).is_acyclic()
True

sage: ClusterSeed([[0,1],[1,2],[2,0]]).is_acyclic()
False
```

```
>>> from sage.all import *
>>> ClusterSeed(['A',Integer(4)]).is_acyclic()
True

>>> ClusterSeed(['A',[Integer(2),Integer(1)],Integer(1)]).is_acyclic()
True

>>> ClusterSeed([[Integer(0),Integer(1)],[Integer(1),Integer(2)],[Integer(2),
↪Integer(0)]]).is_acyclic()
False
```

is_bipartite(*return_bipartition=False*)

Return `True` iff `self` is bipartite (i.e., if the underlying quiver is bipartite).

INPUT:

- `return_bipartition` – boolean (default: `False`); if `True`, the bipartition is returned in the case of `self` being bipartite

EXAMPLES:

```
sage: ClusterSeed(['A',[3,3],1]).is_bipartite()
True

sage: ClusterSeed(['A',[4,3],1]).is_bipartite()
False
```

```
>>> from sage.all import *
>>> ClusterSeed(['A',[Integer(3),Integer(3)],Integer(1)]).is_bipartite()
True

>>> ClusterSeed(['A',[Integer(4),Integer(3)],Integer(1)]).is_bipartite()
False
```

is_finite()

Return `True` if `self` is of finite type.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.is_finite()
True

sage: S = ClusterSeed(['A',[2,2],1])
sage: S.is_finite()
False
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.is_finite()
True

>>> S = ClusterSeed(['A',[Integer(2),Integer(2)],Integer(1)])
>>> S.is_finite()
False
```

is_mutation_finite(*nr_of_checks=None*, *return_path=False*)

Return `True` if `self` is of finite mutation type.

INPUT:

- `nr_of_checks` – (default: `None`) number of mutations applied; standard is 500 times the number of vertices of `self`
- `return_path` – boolean (default: `False`); if `True`, in case of `self` not being mutation finite, a path from `self` to a quiver with an edge label $(a, -b)$ and $a * b > 4$ is returned

ALGORITHM:

- A cluster seed is mutation infinite if and only if every $b_{ij} * b_{ji} > -4$. Thus, we apply random mutations in random directions

Warning

- Uses a non-deterministic method by random mutations in various directions.
- In theory, it can return a wrong `True`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',10])
sage: S._mutation_type = None
sage: S.is_mutation_finite()
True

sage: S = ClusterSeed([(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(2,9)])
sage: S.is_mutation_finite()
False
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(10)])
>>> S._mutation_type = None
>>> S.is_mutation_finite()
True

>>> S = ClusterSeed([(Integer(0),Integer(1)),(Integer(1),Integer(2)),
↪(Integer(2),Integer(3)),(Integer(3),Integer(4)),(Integer(4),Integer(5)),
↪(Integer(5),Integer(6)),(Integer(6),Integer(7)),(Integer(7),Integer(8)),
↪(Integer(2),Integer(9))])
>>> S.is_mutation_finite()
False
```

m()

Return the number of *frozen variables* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.n()
3

sage: S.m()
0

sage: S = S.principal_extension()
sage: S.m()
3
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.n()
```

(continues on next page)

(continued from previous page)

```
3

>>> S.m()
0

>>> S = S.principal_extension()
>>> S.m()
3
```

most_decreased_denominator_after_mutation()

Return the vertex that will produce the most decrease in denominator degrees after mutation.

EXAMPLES:

```
sage: S = ClusterSeed(['A',5])
sage: S.mutate([0,2,3,1,2,3,1,2,0,2,3])
sage: S.most_decreased_denominator_after_mutation()
2
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(5)])
>>> S.mutate([Integer(0),Integer(2),Integer(3),Integer(1),Integer(2),
↪Integer(3),Integer(1),Integer(2),Integer(0),Integer(2),Integer(3)])
>>> S.most_decreased_denominator_after_mutation()
2
```

most_decreased_edge_after_mutation()

Return the vertex that will produce the least degrees after mutation.

EXAMPLES:

```
sage: S = ClusterSeed(['A',5])
sage: S.mutate([0,2,3,1,2,3,1,2,0,2,3])
sage: S.most_decreased_edge_after_mutation()
2
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(5)])
>>> S.mutate([Integer(0),Integer(2),Integer(3),Integer(1),Integer(2),
↪Integer(3),Integer(1),Integer(2),Integer(0),Integer(2),Integer(3)])
>>> S.most_decreased_edge_after_mutation()
2
```

mutate(*sequence*, *inplace=True*, *input_type=None*)

Mutate `self` at a vertex or a sequence of vertices.

INPUT:

- `sequence` – a vertex of `self`, an iterator of vertices of `self`, a function which takes in the *ClusterSeed* and returns a vertex or an iterator of vertices, or a string representing a type of vertices to mutate
- `inplace` – boolean (default: `True`); if `False`, the result is returned, otherwise `self` is modified
- `input_type` – (default: `None`) indicates the type of data contained in the sequence

Possible values for vertex types in `sequence` are:

- `'first_source'` – mutates at first found source vertex
- `'sources'` – mutates at all sources
- `'first_sink'` – mutates at first sink
- `'sinks'` – mutates at all sink vertices
- `'green'` – mutates at the first green vertex
- `'red'` – mutates at the first red vertex
- `'urban_renewal'` or `'urban'` – mutates at first urban renewal vertex
- `'all_urban_renewals'` or `'all_urban'` – mutates at all urban renewal vertices

For `input_type`, if no value is given, preference will be given to vertex names, then indices, then cluster variables. If all input is not of the same type, an error is given. Possible values for `input_type` are:

- `'vertices'` – interprets the input sequence as vertices
- `'indices'` – interprets the input sequence as indices
- `'cluster_vars'` – interprets the input sequence as cluster variables. This must be selected if inputting a sequence of cluster variables.

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]); S.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: S.mutate(0); S.b_matrix()
[ 0 -1  0  0]
[ 1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: T = S.mutate(0, inplace=False); T
A seed for a cluster algebra of rank 4 of type ['A', 4]

sage: S.mutate(0)
sage: S == T
True

sage: S.mutate([0,1,0])
sage: S.b_matrix()
[ 0 -1  1  0]
[ 1  0  0  0]
[-1  0  0  1]
[ 0  0 -1  0]

sage: S = ClusterSeed(QuiverMutationType([['A',1],['A',3]]))
sage: S.b_matrix()
[ 0  0  0  0]
[ 0  0  1  0]
```

(continues on next page)

(continued from previous page)

```
[ 0 -1  0 -1]
[ 0  0  1  0]

sage: T = S.mutate(0,inplace=False)
sage: S == T
False

sage: Q = ClusterSeed(['A',3]);Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]

sage: Q.mutate('first_sink');Q.b_matrix()
[ 0 -1  0]
[ 1  0  1]
[ 0 -1  0]

sage: def last_vertex(self): return self._n - 1
sage: Q.mutate(last_vertex); Q.b_matrix()
[ 0 -1  0]
[ 1  0 -1]
[ 0  1  0]

sage: S = ClusterSeed(['A', 4], user_labels=['a', 'b', 'c', 'd'])
sage: S.mutate('a'); S.mutate('(b+1)/a')
sage: S.cluster()
[a, b, c, d]

sage: S = ClusterSeed(['A', 4], user_labels=['a', 'b', 'c'])
Traceback (most recent call last):
...
ValueError: the number of user-defined labels is not
 the number of exchangeable and frozen variables

sage: S = ClusterSeed(['A', 4], user_labels=['x', 'y', 'w', 'z'])
sage: S.mutate('x')
sage: S.cluster()
[(y + 1)/x, y, w, z]
sage: S.mutate('(y+1)/x')
sage: S.cluster()
[x, y, w, z]
sage: S.mutate('y')
sage: S.cluster()
[x, (x*w + 1)/y, w, z]
sage: S.mutate('(x*w+1)/y')
sage: S.cluster()
[x, y, w, z]

sage: S = ClusterSeed(['A', 4], user_labels=[[1, 2], [2, 3], [4, 5], [5, 6]])
sage: S.cluster()
[x_1_2, x_2_3, x_4_5, x_5_6]
sage: S.mutate('[1,2]')
```

(continues on next page)

(continued from previous page)

```
sage: S.cluster()
[(x_2_3 + 1)/x_1_2, x_2_3, x_4_5, x_5_6]

sage: S = ClusterSeed(['A', 4], user_labels=[[1, 2], [2, 3], [4, 5], [5, 6]],
....:                  user_labels_prefix='P');
sage: S.cluster()
[P_1_2, P_2_3, P_4_5, P_5_6]
sage: S.mutate('[1,2]')
sage: S.cluster()
[(P_2_3 + 1)/P_1_2, P_2_3, P_4_5, P_5_6]
sage: S.mutate('P_4_5')
sage: S.cluster()
[(P_2_3 + 1)/P_1_2, P_2_3, (P_2_3*P_5_6 + 1)/P_4_5, P_5_6]

sage: S = ClusterSeed(['A', 4])
sage: S.mutate([0, 1, 0, 1, 0, 2, 1])
sage: T = ClusterSeed(S)
sage: S.use_fpolys(False)
sage: S.use_g_vectors(False)
sage: S.use_c_vectors(False)
sage: S._C
sage: S._G
sage: S._F
sage: S.g_matrix()
[ 0 -1  0  0]
[ 1  1  1  0]
[ 0  0 -1  0]
[ 0  0  1  1]
sage: S.c_matrix()
[ 1 -1  0  0]
[ 1  0  0  0]
[ 1  0 -1  1]
[ 0  0  0  1]
sage: S.f_polynomials() == T.f_polynomials()
True

sage: S.cluster() == T.cluster()
True
sage: S._mut_path
[0, 1, 0, 1, 0, 2, 1]

sage: S = ClusterSeed(DiGraph([[1, 2], [2, 'c']]))
sage: S.mutate(1)
Input can be ambiguously interpreted as both vertices and indices.
 Mutating at vertices by default.
sage: S.cluster()
[(x2 + 1)/x1, x2, c]
sage: S.mutate(1, input_type='indices')
sage: S.cluster()
[(x2 + 1)/x1, (x2*c + x1 + c)/(x1*x2), c]

sage: S = ClusterSeed(DiGraph([['a', 'b'], ['c', 'b'], ['d', 'b']]))
```

(continues on next page)

(continued from previous page)

```
sage: S.mutate(['a', 'b', 'a', 'b', 'a'])
sage: S.cluster()
[b, a, c, d]
sage: S.mutate('a')
Input can be ambiguously interpreted as both vertices and cluster variables.
 Mutating at vertices by default.
sage: S.cluster()
[(a*c*d + 1)/b, a, c, d]
sage: S.mutate('a', input_type='cluster_vars')
sage: S.cluster()
[(a*c*d + 1)/b, (a*c*d + b + 1)/(a*b), c, d]
sage: S.mutate(['(a*c*d + 1)/b', 'd'])
sage: S.cluster()
[(b + 1)/a, (a*c*d + b + 1)/(a*b), c, (a*c*d + b^2 + 2*b + 1)/(a*b*d)]

sage: S = ClusterSeed(DiGraph([[5, 'b']]))
sage: S.mutate(5)
sage: S.cluster()
[(b + 1)/x5, b]
sage: S.mutate([5])
sage: S.cluster()
[x5, b]
sage: S.mutate(0)
sage: S.cluster()
[(b + 1)/x5, b]

sage: S = ClusterSeed(DiGraph([[1, 2]]))
sage: S.cluster()
[x1, x2]
sage: S.mutate(1)
Input can be ambiguously interpreted as both vertices and indices.
 Mutating at vertices by default.
sage: S.cluster()
[(x2 + 1)/x1, x2]

sage: S = ClusterSeed(DiGraph([[-1, 0], [0, 1]]))
sage: S.cluster()
[xneg1, x0, x1]
sage: S.mutate(-1);S.cluster()
[(x0 + 1)/xneg1, x0, x1]
sage: S.mutate(0, input_type='vertices');S.cluster()
[(x0 + 1)/xneg1, (x0*x1 + xneg1 + x1)/(xneg1*x0), x1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)]); S.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

>>> S.mutate(Integer(0)); S.b_matrix()
[ 0 -1  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

>>> T = S.mutate(Integer(0), inplace=False); T
A seed for a cluster algebra of rank 4 of type ['A', 4]

>>> S.mutate(Integer(0))
>>> S == T
True

>>> S.mutate([Integer(0),Integer(1),Integer(0)])
>>> S.b_matrix()
[ 0 -1  1  0]
[ 1  0  0  0]
[-1  0  0  1]
[ 0  0 -1  0]

>>> S = ClusterSeed(QuiverMutationType([['A',Integer(1)],['A',Integer(3)]]))
>>> S.b_matrix()
[ 0  0  0  0]
[ 0  0  1  0]
[ 0 -1  0 -1]
[ 0  0  1  0]

>>> T = S.mutate(Integer(0),inplace=False)
>>> S == T
False

>>> Q = ClusterSeed(['A',Integer(3)]);Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]

>>> Q.mutate('first_sink');Q.b_matrix()
[ 0 -1  0]
[ 1  0  1]
[ 0 -1  0]

>>> def last_vertex(self): return self._n - Integer(1)
>>> Q.mutate(last_vertex); Q.b_matrix()
[ 0 -1  0]
[ 1  0 -1]
[ 0  1  0]

>>> S = ClusterSeed(['A', Integer(4)], user_labels=['a', 'b', 'c', 'd'])
>>> S.mutate('a'); S.mutate('(b+1)/a')
>>> S.cluster()
[a, b, c, d]

>>> S = ClusterSeed(['A', Integer(4)], user_labels=['a', 'b', 'c'])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: the number of user-defined labels is not
 the number of exchangeable and frozen variables

>>> S = ClusterSeed(['A', Integer(4)], user_labels=['x', 'y', 'w', 'z'])
>>> S.mutate('x')
>>> S.cluster()
[(y + 1)/x, y, w, z]
>>> S.mutate('(y+1)/x')
>>> S.cluster()
[x, y, w, z]
>>> S.mutate('y')
>>> S.cluster()
[x, (x*w + 1)/y, w, z]
>>> S.mutate('(x*w+1)/y')
>>> S.cluster()
[x, y, w, z]

>>> S = ClusterSeed(['A', Integer(4)], user_labels=[[Integer(1), Integer(2)],
↪[Integer(2), Integer(3)], [Integer(4), Integer(5)], [Integer(5),
↪Integer(6)]])
>>> S.cluster()
[x_1_2, x_2_3, x_4_5, x_5_6]
>>> S.mutate('[1,2]')
>>> S.cluster()
[(x_2_3 + 1)/x_1_2, x_2_3, x_4_5, x_5_6]

>>> S = ClusterSeed(['A', Integer(4)], user_labels=[[Integer(1), Integer(2)],
↪[Integer(2), Integer(3)], [Integer(4), Integer(5)], [Integer(5),
↪Integer(6)]],
...                 user_labels_prefix='P');
>>> S.cluster()
[P_1_2, P_2_3, P_4_5, P_5_6]
>>> S.mutate('[1,2]')
>>> S.cluster()
[(P_2_3 + 1)/P_1_2, P_2_3, P_4_5, P_5_6]
>>> S.mutate('P_4_5')
>>> S.cluster()
[(P_2_3 + 1)/P_1_2, P_2_3, (P_2_3*P_5_6 + 1)/P_4_5, P_5_6]

>>> S = ClusterSeed(['A', Integer(4)])
>>> S.mutate([Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),
↪Integer(2), Integer(1)])
>>> T = ClusterSeed(S)
>>> S.use_fpolys(False)
>>> S.use_g_vectors(False)
>>> S.use_c_vectors(False)
>>> S._C
>>> S._G
>>> S._F
>>> S.g_matrix()
[ 0 -1  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 1  1  1  0]
[ 0  0 -1  0]
[ 0  0  1  1]
>>> S.c_matrix()
[ 1 -1  0  0]
[ 1  0  0  0]
[ 1  0 -1  1]
[ 0  0  0  1]
>>> S.f_polynomials() == T.f_polynomials()
True

>>> S.cluster() == T.cluster()
True
>>> S._mut_path
[0, 1, 0, 1, 0, 2, 1]

>>> S = ClusterSeed(DiGraph([[Integer(1), Integer(2)], [Integer(2), 'c']]))
>>> S.mutate(Integer(1))
Input can be ambiguously interpreted as both vertices and indices.
 Mutating at vertices by default.
>>> S.cluster()
[(x2 + 1)/x1, x2, c]
>>> S.mutate(Integer(1), input_type='indices')
>>> S.cluster()
[(x2 + 1)/x1, (x2*c + x1 + c)/(x1*x2), c]

>>> S = ClusterSeed(DiGraph([['a', 'b'], ['c', 'b'], ['d', 'b']]))
>>> S.mutate(['a', 'b', 'a', 'b', 'a'])
>>> S.cluster()
[b, a, c, d]
>>> S.mutate('a')
Input can be ambiguously interpreted as both vertices and cluster variables.
 Mutating at vertices by default.
>>> S.cluster()
[(a*c*d + 1)/b, a, c, d]
>>> S.mutate('a', input_type='cluster_vars')
>>> S.cluster()
[(a*c*d + 1)/b, (a*c*d + b + 1)/(a*b), c, d]
>>> S.mutate(['(a*c*d + 1)/b', 'd'])
>>> S.cluster()
[(b + 1)/a, (a*c*d + b + 1)/(a*b), c, (a*c*d + b^2 + 2*b + 1)/(a*b*d)]

>>> S = ClusterSeed(DiGraph([[Integer(5), 'b']]))
>>> S.mutate(Integer(5))
>>> S.cluster()
[(b + 1)/x5, b]
>>> S.mutate([Integer(5)])
>>> S.cluster()
[x5, b]
>>> S.mutate(Integer(0))
>>> S.cluster()
[(b + 1)/x5, b]
```

(continues on next page)

(continued from previous page)

```
>>> S = ClusterSeed(DiGraph([[Integer(1), Integer(2)]]))
>>> S.cluster()
[x1, x2]
>>> S.mutate(Integer(1))
Input can be ambiguously interpreted as both vertices and indices.
 Mutating at vertices by default.
>>> S.cluster()
[(x2 + 1)/x1, x2]

>>> S = ClusterSeed(DiGraph([[-Integer(1), Integer(0)], [Integer(0),␣
↪Integer(1)]]))
>>> S.cluster()
[xneg1, x0, x1]
>>> S.mutate(-Integer(1));S.cluster()
[(x0 + 1)/xneg1, x0, x1]
>>> S.mutate(Integer(0), input_type='vertices');S.cluster()
[(x0 + 1)/xneg1, (x0*x1 + xneg1 + x1)/(xneg1*x0), x1]
```

mutation_analysis(*options=['all']*, *filter=None*)

Run an analysis of all potential mutation options. Note that this might take a long time on large seeds.

> **Note**
>
> Edges are only returned if we have a non-valued quiver. Green and red vertices are only returned if the cluster is principal.

INPUT:

- `options` – (default: `['all']`) a list of mutation options
- `filter` – (default: `None`) a vertex or interval of vertices to limit our search to

Possible options are:

- `'all'` – all options below
- `'edges'` – number of edges (works with skew-symmetric quivers)
- `'edge_diff'` – edges added/deleted (works with skew-symmetric quivers)
- `'green_vertices'` – list of green vertices (works with principals)
- `'green_vertices_diff'` – green vertices added/removed (works with principals)
- `'red_vertices'` – list of red vertices (works with principals)
- `'red_vertices_diff'` – red vertices added/removed (works with principals)
- `'urban_renewals'` – list of urban renewal vertices
- `'urban_renewals_diff'` – urban renewal vertices added/removed
- `'sources'` – list of source vertices
- `'sources_diff'` – source vertices added/removed
- `'sinks'` – list of sink vertices

- `'sinks_diff'` – sink vertices added/removed
- `'denominators'` – list of all denominators of the cluster variables

OUTPUT:

Outputs a dictionary indexed by the vertex numbers. Each vertex will itself also be a dictionary with each desired option included as a key in the dictionary. As an example you would get something similar to: `{0: {'edges': 1}, 1: {'edges': 2}}`. This represents that if you were to do a mutation at the current seed then mutating at vertex 0 would result in a quiver with 1 edge and mutating at vertex 0 would result in a quiver with 2 edges.

EXAMPLES:

```
sage: B = [[0, 4, 0, -1],[-4,0, 3, 0],[0, -3, 0, 1],[1, 0, -1, 0]]
sage: S = ClusterSeed(matrix(B)); S.mutate([2,3,1,2,1,3,0,2])
sage: S.mutation_analysis()
{0: {'d_matrix': [ 0  0  1  0]
                 [ 0 -1  0  0]
                 [ 0  0  0 -1]
                 [-1  0  0  0],
     'denominators': [1, 1, x0, 1],
     'edge_diff': 6,
     'edges': 13,
     'green_vertices': [0, 1, 3],
     'green_vertices_diff': {'added': [0], 'removed': []},
     'red_vertices': [2],
     'red_vertices_diff': {'added': [], 'removed': [0]},
     'sinks': [],
     'sinks_diff': {'added': [], 'removed': [2]},
     'sources': [],
     'sources_diff': {'added': [], 'removed': []},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}},
 1: {'d_matrix': [ 1  4  1  0]
                 [ 0  1  0  0]
                 [ 0  0  0 -1]
                 [ 1  4  0  0],
     'denominators': [x0*x3, x0^4*x1*x3^4, x0, 1],
     'edge_diff': 2,
     'edges': 9,
     'green_vertices': [0, 3],
     'green_vertices_diff': {'added': [0], 'removed': [1]},
     'red_vertices': [1, 2],
     'red_vertices_diff': {'added': [1], 'removed': [0]},
     'sinks': [2],
     'sinks_diff': {'added': [], 'removed': []},
     'sources': [],
     'sources_diff': {'added': [], 'removed': []},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}},
 2: {'d_matrix': [ 1  0  0  0]
                 [ 0 -1  0  0]
                 [ 0  0  0 -1]
                 [ 1  0  1  0],
```

(continues on next page)

(continued from previous page)

```
     'denominators': [x0*x3, 1, x3, 1],
     'edge_diff': 0,
     'edges': 7,
     'green_vertices': [1, 2, 3],
     'green_vertices_diff': {'added': [2], 'removed': []},
     'red_vertices': [0],
     'red_vertices_diff': {'added': [], 'removed': [2]},
     'sinks': [],
     'sinks_diff': {'added': [], 'removed': [2]},
     'sources': [2],
     'sources_diff': {'added': [2], 'removed': []},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}},
 3: {'d_matrix': [ 1  0  1  1]
                 [ 0 -1  0  0]
                 [ 0  0  0  1]
                 [ 1  0  0  1],
     'denominators': [x0*x3, 1, x0, x0*x2*x3],
     'edge_diff': -1,
     'edges': 6,
     'green_vertices': [1],
     'green_vertices_diff': {'added': [], 'removed': [3]},
     'red_vertices': [0, 2, 3],
     'red_vertices_diff': {'added': [3], 'removed': []},
     'sinks': [2],
     'sinks_diff': {'added': [], 'removed': []},
     'sources': [1],
     'sources_diff': {'added': [1], 'removed': []},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}}}

sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutation_analysis()
{0: {'d_matrix': [ 1  0  0]
                 [ 0 -1  0]
                 [ 0  0 -1],
     'denominators': [x0, 1, 1],
     'green_vertices': [1, 2],
     'green_vertices_diff': {'added': [], 'removed': [0]},
     'red_vertices': [0],
     'red_vertices_diff': {'added': [0], 'removed': []},
     'sinks': [],
     'sinks_diff': {'added': [], 'removed': [1]},
     'sources': [4, 5],
     'sources_diff': {'added': [], 'removed': [3]},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}},
 1: {'d_matrix': [-1  0  0]
                 [ 0  1  0]
                 [ 0  0 -1],
     'denominators': [1, x1, 1],
     'green_vertices': [0, 2],
```

(continues on next page)

(continued from previous page)

```
     'green_vertices_diff': {'added': [], 'removed': [1]},
     'red_vertices': [1],
     'red_vertices_diff': {'added': [1], 'removed': []},
     'sinks': [0, 2, 4],
     'sinks_diff': {'added': [0, 2, 4], 'removed': [1]},
     'sources': [1, 3, 5],
     'sources_diff': {'added': [1], 'removed': [4]},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}},
 2: {'d_matrix': [-1  0  0]
                 [ 0 -1  0]
                 [ 0  0  1],
     'denominators': [1, 1, x2],
     'green_vertices': [0, 1],
     'green_vertices_diff': {'added': [], 'removed': [2]},
     'red_vertices': [2],
     'red_vertices_diff': {'added': [2], 'removed': []},
     'sinks': [],
     'sinks_diff': {'added': [], 'removed': [1]},
     'sources': [3, 4],
     'sources_diff': {'added': [], 'removed': [5]},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}}}
```

```
>>> from sage.all import *
>>> B = [[Integer(0), Integer(4), Integer(0), -Integer(1)],[-Integer(4),
↪Integer(0), Integer(3), Integer(0)],[Integer(0), -Integer(3), Integer(0),␣
↪Integer(1)],[Integer(1), Integer(0), -Integer(1), Integer(0)]]
>>> S = ClusterSeed(matrix(B)); S.mutate([Integer(2),Integer(3),Integer(1),
↪Integer(2),Integer(1),Integer(3),Integer(0),Integer(2)])
>>> S.mutation_analysis()
{0: {'d_matrix': [ 0  0  1  0]
                 [ 0 -1  0  0]
                 [ 0  0  0 -1]
                 [-1  0  0  0],
     'denominators': [1, 1, x0, 1],
     'edge_diff': 6,
     'edges': 13,
     'green_vertices': [0, 1, 3],
     'green_vertices_diff': {'added': [0], 'removed': []},
     'red_vertices': [2],
     'red_vertices_diff': {'added': [], 'removed': [0]},
     'sinks': [],
     'sinks_diff': {'added': [], 'removed': [2]},
     'sources': [],
     'sources_diff': {'added': [], 'removed': []},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}},
 1: {'d_matrix': [ 1  4  1  0]
                 [ 0  1  0  0]
                 [ 0  0  0 -1]
                 [ 1  4  0  0],
```

(continues on next page)

(continued from previous page)

```
     'denominators': [x0*x3, x0^4*x1*x3^4, x0, 1],
     'edge_diff': 2,
     'edges': 9,
     'green_vertices': [0, 3],
     'green_vertices_diff': {'added': [0], 'removed': [1]},
     'red_vertices': [1, 2],
     'red_vertices_diff': {'added': [1], 'removed': [0]},
     'sinks': [2],
     'sinks_diff': {'added': [], 'removed': []},
     'sources': [],
     'sources_diff': {'added': [], 'removed': []},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}},
 2: {'d_matrix': [ 1  0  0  0]
                 [ 0 -1  0  0]
                 [ 0  0  0 -1]
                 [ 1  0  1  0],
     'denominators': [x0*x3, 1, x3, 1],
     'edge_diff': 0,
     'edges': 7,
     'green_vertices': [1, 2, 3],
     'green_vertices_diff': {'added': [2], 'removed': []},
     'red_vertices': [0],
     'red_vertices_diff': {'added': [], 'removed': [2]},
     'sinks': [],
     'sinks_diff': {'added': [], 'removed': [2]},
     'sources': [2],
     'sources_diff': {'added': [2], 'removed': []},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}},
 3: {'d_matrix': [ 1  0  1  1]
                 [ 0 -1  0  0]
                 [ 0  0  0  1]
                 [ 1  0  0  1],
     'denominators': [x0*x3, 1, x0, x0*x2*x3],
     'edge_diff': -1,
     'edges': 6,
     'green_vertices': [1],
     'green_vertices_diff': {'added': [], 'removed': [3]},
     'red_vertices': [0, 2, 3],
     'red_vertices_diff': {'added': [3], 'removed': []},
     'sinks': [2],
     'sinks_diff': {'added': [], 'removed': []},
     'sources': [1],
     'sources_diff': {'added': [1], 'removed': []},
     'urban_renewals': [],
     'urban_renewals_diff': {'added': [], 'removed': []}}}

>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.mutation_analysis()
{0: {'d_matrix': [ 1  0  0]
                 [ 0 -1  0]
```

(continues on next page)

(continued from previous page)

```
                  [ 0  0 -1],
    'denominators': [x0, 1, 1],
    'green_vertices': [1, 2],
    'green_vertices_diff': {'added': [], 'removed': [0]},
    'red_vertices': [0],
    'red_vertices_diff': {'added': [0], 'removed': []},
    'sinks': [],
    'sinks_diff': {'added': [], 'removed': [1]},
    'sources': [4, 5],
    'sources_diff': {'added': [], 'removed': [3]},
    'urban_renewals': [],
    'urban_renewals_diff': {'added': [], 'removed': []}},
1: {'d_matrix': [-1  0  0]
                [ 0  1  0]
                [ 0  0 -1],
    'denominators': [1, x1, 1],
    'green_vertices': [0, 2],
    'green_vertices_diff': {'added': [], 'removed': [1]},
    'red_vertices': [1],
    'red_vertices_diff': {'added': [1], 'removed': []},
    'sinks': [0, 2, 4],
    'sinks_diff': {'added': [0, 2, 4], 'removed': [1]},
    'sources': [1, 3, 5],
    'sources_diff': {'added': [1], 'removed': [4]},
    'urban_renewals': [],
    'urban_renewals_diff': {'added': [], 'removed': []}},
2: {'d_matrix': [-1  0  0]
                [ 0 -1  0]
                [ 0  0  1],
    'denominators': [1, 1, x2],
    'green_vertices': [0, 1],
    'green_vertices_diff': {'added': [], 'removed': [2]},
    'red_vertices': [2],
    'red_vertices_diff': {'added': [2], 'removed': []},
    'sinks': [],
    'sinks_diff': {'added': [], 'removed': [1]},
    'sources': [3, 4],
    'sources_diff': {'added': [], 'removed': [5]},
    'urban_renewals': [],
    'urban_renewals_diff': {'added': [], 'removed': []}}}
```

mutation_class(*depth=+Infinity*, *show_depth=False*, *return_paths=False*, *up_to_equivalence=True*, *only_sink_source=False*)

Return the mutation class of `self` with respect to certain constraints.

Note

Vertex labels are not tracked in this method.

See also

mutation_class_iter()

INPUT:

- `depth` – (default: `infinity`) integer, only seeds with distance at most depth from `self` are returned
- `show_depth` – boolean (default: `False`); if `True`, the actual depth of the mutation is shown
- `return_paths` – boolean (default: `False`); if `True`, a shortest path of mutation sequences from self to the given quiver is returned as well
- `up_to_equivalence` – boolean (default: `True`); if `True`, only seeds up to equivalence are considered
- `sink_source` – boolean (default: `False`); if `True`, only mutations at sinks and sources are applied

EXAMPLES:

- for examples see *mutation_class_iter()*

mutation_class_iter(*depth=+Infinity*, *show_depth=False*, *return_paths=False*, *up_to_equivalence=True*, *only_sink_source=False*)

Return an iterator for the mutation class of `self` with respect to certain constraints.

INPUT:

- `depth` – (default: infinity) integer or infinity, only seeds with distance at most `depth` from `self` are returned
- `show_depth` – boolean (default: `False`); if `True`, the current depth of the mutation is shown while computing
- `return_paths` – boolean (default: `False`); if `True`, a shortest path of mutations from `self` to the given quiver is returned as well
- `up_to_equivalence` – boolean (default: `True`); if `True`, only one seed up to simultaneous permutation of rows and columns of the exchange matrix is recorded
- `sink_source` – boolean (default: `False`); if `True`, only mutations at sinks and sources are applied

EXAMPLES:

A standard finite type example:

```
sage: S = ClusterSeed(['A',3])
sage: it = S.mutation_class_iter()
sage: for T in it: print(T)
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
```

(continues on next page)

(continued from previous page)

```
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> it = S.mutation_class_iter()
>>> for T in it: print(T)
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
```

A finite type example with given depth:

```
sage: it = S.mutation_class_iter(depth=1)
sage: for T in it: print(T)
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
```

```
>>> from sage.all import *
>>> it = S.mutation_class_iter(depth=Integer(1))
>>> for T in it: print(T)
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
```

A finite type example where the depth is shown while computing:

```
sage: it = S.mutation_class_iter(show_depth=True)
sage: for T in it: pass
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 4          Time: ... s
Depth: 2     found: 9          Time: ... s
Depth: 3     found: 13         Time: ... s
Depth: 4     found: 14         Time: ... s
```

```
>>> from sage.all import *
>>> it = S.mutation_class_iter(show_depth=True)
>>> for T in it: pass
```

(continues on next page)

(continued from previous page)

```
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 4          Time: ... s
Depth: 2     found: 9          Time: ... s
Depth: 3     found: 13         Time: ... s
Depth: 4     found: 14         Time: ... s
```

A finite type example with shortest paths returned:

```
sage: it = S.mutation_class_iter(return_paths=True)
sage: mutation_class = list(it)
sage: len(mutation_class)
14
sage: mutation_class[0]
(A seed for a cluster algebra of rank 3 of type ['A', 3], [])
```

```
>>> from sage.all import *
>>> it = S.mutation_class_iter(return_paths=True)
>>> mutation_class = list(it)
>>> len(mutation_class)
14
>>> mutation_class[Integer(0)]
(A seed for a cluster algebra of rank 3 of type ['A', 3], [])
```

Finite type examples not considered up to equivalence:

```
sage: it = S.mutation_class_iter(up_to_equivalence=False)
sage: len([T for T in it])
84

sage: it = ClusterSeed(['A',2]).mutation_class_iter(return_paths=True,
....:                                                 up_to_equivalence=False)
sage: mutation_class = list(it)
sage: len(mutation_class)
10
sage: mutation_class[0]
(A seed for a cluster algebra of rank 2 of type ['A', 2], [])
```

```
>>> from sage.all import *
>>> it = S.mutation_class_iter(up_to_equivalence=False)
>>> len([T for T in it])
84

>>> it = ClusterSeed(['A',Integer(2)]).mutation_class_iter(return_paths=True,
...                                                 up_to_equivalence=False)
>>> mutation_class = list(it)
>>> len(mutation_class)
10
>>> mutation_class[Integer(0)]
(A seed for a cluster algebra of rank 2 of type ['A', 2], [])
```

Check that Issue #14638 is fixed:

```
sage: S = ClusterSeed(['E',6])
sage: MC = S.mutation_class(depth=7); len(MC)   # long time
534
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['E',Integer(6)])
>>> MC = S.mutation_class(depth=Integer(7)); len(MC)   # long time
534
```

Infinite type examples:

```
sage: S = ClusterSeed(['A',[1,1],1])
sage: it = S.mutation_class_iter()
sage: next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
sage: next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
sage: next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
sage: next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]

sage: it = S.mutation_class_iter(depth=3, return_paths=True)
sage: for T in it: print(T)
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1, 0])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1, 0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0, 1, 0])
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',[Integer(1),Integer(1)],Integer(1)])
>>> it = S.mutation_class_iter()
>>> next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
>>> next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
>>> next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
>>> next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]

>>> it = S.mutation_class_iter(depth=Integer(3), return_paths=True)
>>> for T in it: print(T)
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1, 0])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1, 0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0, 1, 0])
```

mutation_sequence(*sequence*, *show_sequence=False*, *fig_size=1.2*, *return_output='seed'*)

Return the seeds obtained by mutating `self` at all vertices in `sequence`.

INPUT:

- `sequence` – an iterable of vertices of self
- `show_sequence` – boolean (default: `False`); if `True`, a png containing the associated quivers is shown
- `fig_size` – (default: 1.2) factor by which the size of the plot is multiplied
- `return_output` – (default: `'seed'`) determines what output is to be returned:
  - if `'seed'`, outputs all the cluster seeds obtained by the `sequence` of mutations
  - if `'matrix'`, outputs a list of exchange matrices
  - if `'var'`, outputs a list of new cluster variables obtained at each step

EXAMPLES:

```
sage: S = ClusterSeed(['A',2])
sage: for T in S.mutation_sequence([0,1,0]):
....:     print(T.b_matrix())
[ 0 -1]
[ 1  0]
[ 0  1]
[-1  0]
[ 0 -1]
[ 1  0]

sage: S = ClusterSeed(['A',2])
sage: S.mutation_sequence([0,1,0,1], return_output='var')
[(x1 + 1)/x0, (x0 + x1 + 1)/(x0*x1), (x0 + 1)/x1, x0]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(2)])
>>> for T in S.mutation_sequence([Integer(0),Integer(1),Integer(0)]):
...     print(T.b_matrix())
[ 0 -1]
[ 1  0]
[ 0  1]
[-1  0]
[ 0 -1]
[ 1  0]

>>> S = ClusterSeed(['A',Integer(2)])
>>> S.mutation_sequence([Integer(0),Integer(1),Integer(0),Integer(1)], return_
↪output='var')
[(x1 + 1)/x0, (x0 + x1 + 1)/(x0*x1), (x0 + 1)/x1, x0]
```

mutation_type()

Return the mutation type of each connected component of `self`, if it can be determined.

Otherwise, the mutation type of this component is set to be unknown.

The mutation types of the components are ordered by vertex labels.

Warning

- All finite types can be detected,
- All affine types can be detected, **except** affine type D (the algorithm is not yet implemented)
- All exceptional types can be detected.
- Might fail to work if it is used within different Sage processes simultaneously (that happened in the doctesting).

EXAMPLES:

- finite types:

```
sage: S = ClusterSeed(['A',5])
sage: S._mutation_type = S._quiver._mutation_type = None
sage: S.mutation_type()
['A', 5]

sage: S = ClusterSeed([(0,1),(1,2),(2,3),(3,4)])
sage: S.mutation_type()
['A', 5]

sage: S = ClusterSeed(DiGraph([['a','b'],['c','b'],['c','d'],['e','d']]),
....:                 frozen=['c'])
sage: S.mutation_type()
[ ['A', 2], ['A', 2] ]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(5)])
>>> S._mutation_type = S._quiver._mutation_type = None
>>> S.mutation_type()
['A', 5]

>>> S = ClusterSeed([(Integer(0),Integer(1)),(Integer(1),Integer(2)),
↪(Integer(2),Integer(3)),(Integer(3),Integer(4))])
>>> S.mutation_type()
['A', 5]

>>> S = ClusterSeed(DiGraph([['a','b'],['c','b'],['c','d'],['e','d']]),
...                 frozen=['c'])
>>> S.mutation_type()
[ ['A', 2], ['A', 2] ]
```

- affine types:

```
sage: S = ClusterSeed(['E',8,[1,1]]); S
A seed for a cluster algebra of rank 10 of type ['E', 8, [1, 1]]
sage: S._mutation_type = S._quiver._mutation_type = None; S
A seed for a cluster algebra of rank 10
sage: S.mutation_type() # long time
['E', 8, [1, 1]]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['E',Integer(8),[Integer(1),Integer(1)]]); S
A seed for a cluster algebra of rank 10 of type ['E', 8, [1, 1]]
>>> S._mutation_type = S._quiver._mutation_type = None; S
A seed for a cluster algebra of rank 10
>>> S.mutation_type() # long time
['E', 8, [1, 1]]
```

- the not yet working affine type D:

```
sage: S = ClusterSeed(['D',4,1])
sage: S._mutation_type = S._quiver._mutation_type = None
sage: S.mutation_type() # todo: not implemented
['D', 4, 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['D',Integer(4),Integer(1)])
>>> S._mutation_type = S._quiver._mutation_type = None
>>> S.mutation_type() # todo: not implemented
['D', 4, 1]
```

- the exceptional types:

```
sage: S = ClusterSeed(['X',6])
sage: S._mutation_type = S._quiver._mutation_type = None
sage: S.mutation_type() # long time
['X', 6]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['X',Integer(6)])
>>> S._mutation_type = S._quiver._mutation_type = None
>>> S.mutation_type() # long time
['X', 6]
```

- infinite types:

```
sage: S = ClusterSeed(['GR',[4,9]])
sage: S._mutation_type = S._quiver._mutation_type = None
sage: S.mutation_type()
'undetermined infinite mutation type'
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['GR',[Integer(4),Integer(9)]])
>>> S._mutation_type = S._quiver._mutation_type = None
>>> S.mutation_type()
'undetermined infinite mutation type'
```

mutations()

Return the list of mutations `self` has undergone if they are being tracked.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutations()
[]

sage: S.mutate([0,1,0,2])
sage: S.mutations()
[0, 1, 0, 2]

sage: S.track_mutations(False)
sage: S.mutations()
Traceback (most recent call last):
...
ValueError: Not recording mutation sequence.  Need to track mutations.
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.mutations()
[]

>>> S.mutate([Integer(0),Integer(1),Integer(0),Integer(2)])
>>> S.mutations()
[0, 1, 0, 2]

>>> S.track_mutations(False)
>>> S.mutations()
Traceback (most recent call last):
...
ValueError: Not recording mutation sequence.  Need to track mutations.
```

n()

Return the number of *exchangeable variables* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A', 3])
sage: S.n()
3
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A', Integer(3)])
>>> S.n()
3
```

oriented_exchange_graph()

Return the oriented exchange graph of `self` as a directed graph.

The seed must be a cluster seed for a cluster algebra of finite type with principal coefficients (the corresponding quiver must have mutable vertices $0, 1, ..., n-1$).

EXAMPLES:

```
sage: S = ClusterSeed(['A', 2]).principal_extension()
sage: G = S.oriented_exchange_graph(); G
Digraph on 5 vertices
```

(continues on next page)

(continued from previous page)

```
sage: G.out_degree_sequence()
[2, 1, 1, 1, 0]

sage: S = ClusterSeed(['B', 2]).principal_extension()
sage: G = S.oriented_exchange_graph(); G
Digraph on 6 vertices
sage: G.out_degree_sequence()
[2, 1, 1, 1, 1, 0]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A', Integer(2)]).principal_extension()
>>> G = S.oriented_exchange_graph(); G
Digraph on 5 vertices
>>> G.out_degree_sequence()
[2, 1, 1, 1, 0]

>>> S = ClusterSeed(['B', Integer(2)]).principal_extension()
>>> G = S.oriented_exchange_graph(); G
Digraph on 6 vertices
>>> G.out_degree_sequence()
[2, 1, 1, 1, 1, 0]
```

plot (*circular=False*, *mark=None*, *save_pos=False*, *force_c=False*, *with_greens=False*, *add_labels=False*)

Return the plot of the quiver of `self`.

INPUT:

- `circular` – boolean (default: `False`); if `True`, the circular plot is chosen, otherwise >>spring<< is used
- `mark` – (default: `None`) if set to i, the vertex i is highlighted
- `save_pos` – boolean (default: `False`); if `True`, the positions of the vertices are saved
- `force_c` – boolean (default: `False`); if `True`, will show the frozen vertices even if they were never initialized
- `with_greens` – boolean (default: `False`); if `True`, will display the green vertices in green
- `add_labels` – boolean (default: `False`); if `True`, will use the initial variables as labels

EXAMPLES:

```
sage: S = ClusterSeed(['A',5])
sage: S.plot()                                                          #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
sage: S.plot(circular=True)                                             #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
sage: S.plot(circular=True, mark=1)                                     #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(5)])
>>> S.plot()                                                            #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
>>> S.plot(circular=True)                                               #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
>>> S.plot(circular=True, mark=Integer(1))                                ␣
↪     # needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
```

principal_extension()

Return the principal extension of `self`, yielding a $2n \times n$ matrix.

Raises an error if the input seed has a non-square exchange matrix. In this case, the method instead adds n frozen variables to any previously frozen variables. I.e., the seed obtained by adding a frozen variable to every exchangeable variable of `self`.

EXAMPLES:

```
sage: S = ClusterSeed([[0,1],[1,2],[2,3],[2,4]]); S
A seed for a cluster algebra of rank 5

sage: T = S.principal_extension(); T
A seed for a cluster algebra of rank 5 with principal coefficients

sage: T.b_matrix()
[ 0  1  0  0  0]
[-1  0  1  0  0]
[ 0 -1  0  1  1]
[ 0  0 -1  0  0]
[ 0  0 -1  0  0]
[ 1  0  0  0  0]
[ 0  1  0  0  0]
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[ 0  0  0  0  1]

sage: S = ClusterSeed(['A', 4], user_labels=['a', 'b', 'c', 'd'])
sage: T = S.principal_extension()
sage: T.cluster()
[a, b, c, d]
sage: T.coefficients()
[y0, y1, y2, y3]
sage: S2 = ClusterSeed(['A', 4], user_labels={0:'a', 1:'b', 2:'c', 3:'d'})
sage: S2 == S
True
sage: T2 = S2.principal_extension()
sage: T2 == T
True
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> S = ClusterSeed([[Integer(0),Integer(1)],[Integer(1),Integer(2)],
↪[Integer(2),Integer(3)],[Integer(2),Integer(4)]]); S
A seed for a cluster algebra of rank 5

>>> T = S.principal_extension(); T
A seed for a cluster algebra of rank 5 with principal coefficients

>>> T.b_matrix()
[ 0  1  0  0  0]
[-1  0  1  0  0]
[ 0 -1  0  1  1]
[ 0  0 -1  0  0]
[ 0  0 -1  0  0]
[ 1  0  0  0  0]
[ 0  1  0  0  0]
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[ 0  0  0  0  1]

>>> S = ClusterSeed(['A', Integer(4)], user_labels=['a', 'b', 'c', 'd'])
>>> T = S.principal_extension()
>>> T.cluster()
[a, b, c, d]
>>> T.coefficients()
[y0, y1, y2, y3]
>>> S2 = ClusterSeed(['A', Integer(4)], user_labels={Integer(0):'a',␣
↪Integer(1):'b', Integer(2):'c', Integer(3):'d'})
>>> S2 == S
True
>>> T2 = S2.principal_extension()
>>> T2 == T
True
```

quiver()

Return the *quiver* associated to `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.quiver()
Quiver on 3 vertices of type ['A', 3]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.quiver()
Quiver on 3 vertices of type ['A', 3]
```

red_vertices()

Return the list of red vertices of `self`.

A vertex is defined to be red if its c-vector has all nonnegative entries. More information on red vertices can be found at [BDP2013].

OUTPUT: the red vertices as a list of integers

EXAMPLES:

```
sage: ClusterSeed(['A',3]).principal_extension().red_vertices()
[]

sage: ClusterSeed(['A',[3,3],1]).principal_extension().red_vertices()
[]

sage: Q = ClusterSeed(['A',[3,3],1]).principal_extension()
sage: Q.mutate(1)
sage: Q.red_vertices()
[1]
```

```
>>> from sage.all import *
>>> ClusterSeed(['A',Integer(3)]).principal_extension().red_vertices()
[]

>>> ClusterSeed(['A',[Integer(3),Integer(3)],Integer(1)]).principal_
↪extension().red_vertices()
[]

>>> Q = ClusterSeed(['A',[Integer(3),Integer(3)],Integer(1)]).principal_
↪extension()
>>> Q.mutate(Integer(1))
>>> Q.red_vertices()
[1]
```

reorient(*data*)

Reorients `self` with respect to the given total order, or with respect to an iterator of ordered pairs.

> **Warning**
>
> - This operation might change the mutation type of `self`.
> - Ignores ordered pairs (i, j) for which neither (i, j) nor (j, i) is an edge of `self`.

INPUT:

- `data` – an iterator defining a total order on `self.vertices()`, or an iterator of ordered pairs in `self` defining the new orientation of these edges.

EXAMPLES:

```
sage: S = ClusterSeed(['A',[2,3],1])
sage: S.mutation_type()
['A', [2, 3], 1]

sage: S.reorient([(0,1),(2,3)])
sage: S.mutation_type()
['D', 5]

sage: S.reorient([(1,0),(2,3)])
sage: S.mutation_type()
```

(continues on next page)

(continued from previous page)

```
['A', [1, 4], 1]

sage: S.reorient([0,1,2,3,4])
sage: S.mutation_type()
['A', [1, 4], 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',[Integer(2),Integer(3)],Integer(1)])
>>> S.mutation_type()
['A', [2, 3], 1]

>>> S.reorient([(Integer(0),Integer(1)),(Integer(2),Integer(3))])
>>> S.mutation_type()
['D', 5]

>>> S.reorient([(Integer(1),Integer(0)),(Integer(2),Integer(3))])
>>> S.mutation_type()
['A', [1, 4], 1]

>>> S.reorient([Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)])
>>> S.mutation_type()
['A', [1, 4], 1]
```

reset_cluster()

Reset the cluster of `self` to the initial cluster.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutate([1,2,1])
sage: S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]

sage: S.reset_cluster()
sage: S.cluster()
[x0, x1, x2]

sage: T = S.principal_extension()
sage: T.cluster()
[x0, x1, x2]
sage: T.mutate([1,2,1])
sage: T.cluster()
[x0, (x1*y2 + x0)/x2, (x1*y1*y2 + x0*y1 + x2)/(x1*x2)]

sage: T.reset_cluster()
sage: T.cluster()
[x0, x1, x2]

sage: S = ClusterSeed(['B',3], user_labels=[[1,2],[2,3],[3,4]],
....:                   user_labels_prefix='p')
sage: S.mutate([0,1])
sage: S.cluster()
```

(continues on next page)

(continued from previous page)

```
[(p_2_3 + 1)/p_1_2, (p_1_2*p_3_4^2 + p_2_3 + 1)/(p_1_2*p_2_3), p_3_4]

sage: S.reset_cluster()
sage: S.cluster()
[p_1_2, p_2_3, p_3_4]
sage: S.g_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
sage: S.f_polynomials()
[1, 1, 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> S.mutate([Integer(1),Integer(2),Integer(1)])
>>> S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]

>>> S.reset_cluster()
>>> S.cluster()
[x0, x1, x2]

>>> T = S.principal_extension()
>>> T.cluster()
[x0, x1, x2]
>>> T.mutate([Integer(1),Integer(2),Integer(1)])
>>> T.cluster()
[x0, (x1*y2 + x0)/x2, (x1*y1*y2 + x0*y1 + x2)/(x1*x2)]

>>> T.reset_cluster()
>>> T.cluster()
[x0, x1, x2]

>>> S = ClusterSeed(['B',Integer(3)], user_labels=[[Integer(1),Integer(2)],
↪[Integer(2),Integer(3)],[Integer(3),Integer(4)]],
...                     user_labels_prefix='p')
>>> S.mutate([Integer(0),Integer(1)])
>>> S.cluster()
[(p_2_3 + 1)/p_1_2, (p_1_2*p_3_4^2 + p_2_3 + 1)/(p_1_2*p_2_3), p_3_4]

>>> S.reset_cluster()
>>> S.cluster()
[p_1_2, p_2_3, p_3_4]
>>> S.g_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
>>> S.f_polynomials()
[1, 1, 1]
```

reset_coefficients()

Reset the coefficients of `self` to the frozen variables but keep the current cluster.

This raises an error if the number of frozen variables is different from the number of exchangeable variables.

Warning

This command to be phased out since *use_c_vectors()* does this more effectively.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
sage: S.mutate([1,2,1])
sage: S.b_matrix()
[ 0  1 -1]
[-1  0  1]
[ 1 -1  0]
[ 1  0  0]
[ 0  1 -1]
[ 0  0 -1]
sage: S.reset_coefficients()
sage: S.b_matrix()
[ 0  1 -1]
[-1  0  1]
[ 1 -1  0]
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)]).principal_extension()
>>> S.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
>>> S.mutate([Integer(1),Integer(2),Integer(1)])
>>> S.b_matrix()
[ 0  1 -1]
[-1  0  1]
[ 1 -1  0]
[ 1  0  0]
[ 0  1 -1]
[ 0  0 -1]
>>> S.reset_coefficients()
>>> S.b_matrix()
```

(continues on next page)

(continued from previous page)

```
[ 0  1 -1]
[-1  0  1]
[ 1 -1  0]
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
```

save_image(*filename*, *circular=False*, *mark=None*, *save_pos=False*)

Save the plot of the underlying digraph of the quiver of `self`.

INPUT:

- `filename` – the filename the image is saved to
- `circular` – boolean (default: `False`); if `True`, the circular plot is chosen, otherwise >>spring<< is used
- `mark` – (default: `None`) if set to i, the vertex i is highlighted
- `save_pos` – boolean (default: `False`); if `True`, the positions of the vertices are saved

EXAMPLES:

```
sage: S = ClusterSeed(['F',4,[1,2]])
sage: import tempfile
sage: with tempfile.NamedTemporaryFile(suffix='.png') as f:                    #␣
↪needs sage.plot sage.symbolic
....:     S.save_image(f.name)
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['F',Integer(4),[Integer(1),Integer(2)]])
>>> import tempfile
>>> with tempfile.NamedTemporaryFile(suffix='.png') as f:                    #␣
↪needs sage.plot sage.symbolic
...     S.save_image(f.name)
```

set_c_matrix(*data*)

Will force set the c-matrix according to a matrix, a quiver, or a seed.

INPUT:

- `data` – the matrix to set the c-matrix to; also allowed to be a quiver or cluster seed, in which case the b-matrix is used

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: X = matrix([[0,0,1],[0,1,0],[1,0,0]])
sage: S.set_c_matrix(X)
sage: S.c_matrix()
[0 0 1]
[0 1 0]
[1 0 0]

sage: Y = matrix([[-1,0,1],[0,1,0],[1,0,0]])
sage: S.set_c_matrix(Y)
```

(continues on next page)

(continued from previous page)

```
C matrix does not look to be valid - there exists a column
containing positive and negative entries.
Continuing...

sage: Z = matrix([[1,0,1],[0,1,0],[2,0,2]])
sage: S.set_c_matrix(Z)
C matrix does not look to be valid - not a linearly independent set.
Continuing...
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> X = matrix([[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(1),
↪Integer(0)],[Integer(1),Integer(0),Integer(0)]])
>>> S.set_c_matrix(X)
>>> S.c_matrix()
[0 0 1]
[0 1 0]
[1 0 0]

>>> Y = matrix([[-Integer(1),Integer(0),Integer(1)],[Integer(0),Integer(1),
↪Integer(0)],[Integer(1),Integer(0),Integer(0)]])
>>> S.set_c_matrix(Y)
C matrix does not look to be valid - there exists a column
containing positive and negative entries.
Continuing...

>>> Z = matrix([[Integer(1),Integer(0),Integer(1)],[Integer(0),Integer(1),
↪Integer(0)],[Integer(2),Integer(0),Integer(2)]])
>>> S.set_c_matrix(Z)
C matrix does not look to be valid - not a linearly independent set.
Continuing...
```

set_cluster(*cluster*, *force=False*)

Set the cluster for `self` to `cluster`.

> **Warning**
>
> Initialization may lead to inconsistent data.

INPUT:

- `cluster` – an iterable defining a cluster for `self`

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: cluster = S.cluster()
sage: S.mutate([1,2,1])
sage: S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]
sage: cluster2 = S.cluster()
```

(continues on next page)

(continued from previous page)

```
sage: S.set_cluster(cluster)
Warning: using set_cluster at this point could lead to inconsistent seed data.

sage: S.set_cluster(cluster, force=True)
sage: S.cluster()
[x0, x1, x2]
sage: S.set_cluster(cluster2, force=True)
sage: S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]

sage: S = ClusterSeed(['A',3]); S.use_fpolys(False)
sage: S.set_cluster([1,1,1])
Warning: clusters not being tracked so this command is ignored.
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> cluster = S.cluster()
>>> S.mutate([Integer(1),Integer(2),Integer(1)])
>>> S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]
>>> cluster2 = S.cluster()

>>> S.set_cluster(cluster)
Warning: using set_cluster at this point could lead to inconsistent seed data.

>>> S.set_cluster(cluster, force=True)
>>> S.cluster()
[x0, x1, x2]
>>> S.set_cluster(cluster2, force=True)
>>> S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]

>>> S = ClusterSeed(['A',Integer(3)]); S.use_fpolys(False)
>>> S.set_cluster([Integer(1),Integer(1),Integer(1)])
Warning: clusters not being tracked so this command is ignored.
```

show(*fig_size=1*, *circular=False*, *mark=None*, *save_pos=False*, *force_c=False*, *with_greens=False*, *add_labels=False*)

Shows the plot of the quiver of `self`.

INPUT:

- `fig_size` – (default: 1) factor by which the size of the plot is multiplied
- `circular` – boolean (default: `False`); if `True`, the circular plot is chosen, otherwise >>spring<< is used
- `mark` – (default: `None`) if set to i, the vertex i is highlighted
- `save_pos` – boolean (default: `False`); if `True`, the positions of the vertices are saved
- `force_c` – boolean (default: `False`); if `True`, will show the frozen vertices even if they were never initialized
- `with_greens` – boolean (default: `False`); if `True`, will display the green vertices in green

- `add_labels` – boolean (default: `False`); if `True`, will use the initial variables as labels

smallest_c_vector()

Return the vertex with the smallest c-vector.

OUTPUT: integer

EXAMPLES:

```
sage: B = matrix([[0,2], [-2,0]])
sage: C = ClusterSeed(B).principal_extension()
sage: C.mutate(0)
sage: C.smallest_c_vector()
0
```

```
>>> from sage.all import *
>>> B = matrix([[Integer(0),Integer(2)], [-Integer(2),Integer(0)]])
>>> C = ClusterSeed(B).principal_extension()
>>> C.mutate(Integer(0))
>>> C.smallest_c_vector()
0
```

track_mutations(*use=True*)

Begin tracking the mutation path.

Warning

May initialize all other data to ensure that all c-, d-, and g-vectors agree on the start of mutations.

INPUT:

- `use` – boolean (default: `True`); if `True`, will begin filling the mutation path

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]); S.track_mutations(False)
sage: S.mutate(0)
sage: S.mutations()
Traceback (most recent call last):
...
ValueError: Not recording mutation sequence.  Need to track mutations.
sage: S.track_mutations(True)
sage: S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: S.mutate([0,1])
sage: S.mutations()
[0, 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)]); S.track_mutations(False)
```

(continues on next page)

(continued from previous page)

```
>>> S.mutate(Integer(0))
>>> S.mutations()
Traceback (most recent call last):
...
ValueError: Not recording mutation sequence.  Need to track mutations.
>>> S.track_mutations(True)
>>> S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

>>> S.mutate([Integer(0),Integer(1)])
>>> S.mutations()
[0, 1]
```

universal_extension()

Return the universal extension of `self`.

This is the initial seed of the associated cluster algebra with universal coefficients, as defined in section 12 of [FZ2007].

This method works only if `self` is a bipartite, finite-type seed.

Due to some limitations in the current implementation of `CartanType`, we need to construct the set of almost positive coroots by hand. As a consequence their ordering is not the standard one (the rows of the bottom part of the exchange matrix might be a shuffling of those you would expect).

EXAMPLES:

```
sage: S = ClusterSeed(['A',2])
sage: T = S.universal_extension()
sage: T.b_matrix()
[ 0  1]
[-1  0]
[-1  0]
[ 1  0]
[ 1 -1]
[ 0  1]
[ 0 -1]

sage: S = ClusterSeed(['A',3])
sage: T = S.universal_extension()
sage: T.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
[-1  0  0]
[ 1  0  0]
[ 1 -1  0]
[ 1 -1  1]
[ 0  1  0]
[ 0 -1  0]
[ 0 -1  1]
```

(continues on next page)

(continued from previous page)

```
[ 0  0 -1]
[ 0  0  1]

sage: S = ClusterSeed(['B',2])
sage: T = S.universal_extension()
sage: T.b_matrix()
[ 0  1]
[-2  0]
[-1  0]
[ 1  0]
[ 1 -1]
[ 2 -1]
[ 0  1]
[ 0 -1]

sage: S = ClusterSeed(['A', 5], user_labels=[-2, -1, 0, 1 ,2])
sage: U = S.universal_extension()
sage: U.b_matrix() == ClusterSeed(['A', 5]).universal_extension().b_matrix()
True
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(2)])
>>> T = S.universal_extension()
>>> T.b_matrix()
[ 0  1]
[-1  0]
[-1  0]
[ 1  0]
[ 1 -1]
[ 0  1]
[ 0 -1]

>>> S = ClusterSeed(['A',Integer(3)])
>>> T = S.universal_extension()
>>> T.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
[-1  0  0]
[ 1  0  0]
[ 1 -1  0]
[ 1 -1  1]
[ 0  1  0]
[ 0 -1  0]
[ 0 -1  1]
[ 0  0 -1]
[ 0  0  1]

>>> S = ClusterSeed(['B',Integer(2)])
>>> T = S.universal_extension()
>>> T.b_matrix()
[ 0  1]
```

(continues on next page)

(continued from previous page)

```
[-2  0]
[-1  0]
[ 1  0]
[ 1 -1]
[ 2 -1]
[ 0  1]
[ 0 -1]

>>> S = ClusterSeed(['A', Integer(5)], user_labels=[-Integer(2), -Integer(1),␣
↪Integer(0), Integer(1) ,Integer(2)])
>>> U = S.universal_extension()
>>> U.b_matrix() == ClusterSeed(['A', Integer(5)]).universal_extension().b_
↪matrix()
True
```

urban_renewals (*return_first=False*)

Return the list of the urban renewal vertices of `self`.

An urban renewal vertex is one in which there are two arrows pointing toward the vertex and two arrows pointing away.

INPUT:

- `return_first` – boolean (default: `False`); if `True`, will return the first urban renewal

OUTPUT:

A list of vertices (as integers)

EXAMPLES:

```
sage: G = ClusterSeed(['GR',[4,9]]); G.urban_renewals()
[5, 6]
```

```
>>> from sage.all import *
>>> G = ClusterSeed(['GR',[Integer(4),Integer(9)]]); G.urban_renewals()
[5, 6]
```

use_c_vectors (*use=True*, *bot_is_c=False*, *force=False*)

Reconstruct c-vectors from other data or initialize if no usable data exists.

Warning: Initialization may lead to inconsistent data.

INPUT:

- `use` – boolean (default: `True`); if `True`, will use c-vectors
- `bot_is_c` – boolean (default: `False`); if `True` and `ClusterSeed` `self` has `self._m == self._n`, then will assume bottom half of the extended exchange matrix is the c-matrix. If `True`, lets the `ClusterSeed` know c-vectors can be calculated.

EXAMPLES:

```
sage: S = ClusterSeed(['A',4])
sage: S.use_c_vectors(False); S.use_g_vectors(False)
sage: S.use_fpolys(False); S.track_mutations(False)
sage: S.use_c_vectors(True)
```

(continues on next page)

(continued from previous page)

```
Warning: Initializing c-vectors at this point
could lead to inconsistent seed data.

sage: S.use_c_vectors(True, force=True)
sage: S.c_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: S = ClusterSeed(['A',4])
sage: S.use_c_vectors(False); S.use_g_vectors(False)
sage: S.use_fpolys(False); S.track_mutations(False)
sage: S.mutate(1)
sage: S.use_c_vectors(True, force=True)
sage: S.c_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_c_vectors(False); S.use_g_vectors(False)
>>> S.use_fpolys(False); S.track_mutations(False)
>>> S.use_c_vectors(True)
Warning: Initializing c-vectors at this point
could lead to inconsistent seed data.

>>> S.use_c_vectors(True, force=True)
>>> S.c_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_c_vectors(False); S.use_g_vectors(False)
>>> S.use_fpolys(False); S.track_mutations(False)
>>> S.mutate(Integer(1))
>>> S.use_c_vectors(True, force=True)
>>> S.c_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

use_d_vectors(*use=True*, *force=False*)

Reconstruct d-vectors from other data or initialize if no usable data exists.

> **Warning**
>
> Initialization may lead to inconsistent data.

INPUT:

- `use` – boolean (default: `True`); if `True`, will use d-vectors

EXAMPLES:

```
sage: S = ClusterSeed(['A',4])
sage: S.use_d_vectors(True)
sage: S.d_matrix()
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]

sage: S = ClusterSeed(['A',4]); S.use_d_vectors(False)
sage: S.track_mutations(False); S.mutate(1); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
sage: S.use_fpolys(False)
sage: S.d_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate d-vectors. Need to use d vectors.

sage: S = ClusterSeed(['A',4]); S.use_d_vectors(False)
sage: S.track_mutations(False); S.mutate(1); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
sage: S.use_fpolys(False)
sage: S.use_d_vectors(True)
Warning: Initializing d-vectors at this point
could lead to inconsistent seed data.

sage: S.use_d_vectors(True, force=True)
sage: S.d_matrix()
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]

sage: S = ClusterSeed(['A',4]); S.mutate(1); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
```

(continues on next page)

(continued from previous page)

```
sage: S = ClusterSeed(['A',4])
sage: S.use_d_vectors(True); S.mutate(1); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_d_vectors(True)
>>> S.d_matrix()
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]

>>> S = ClusterSeed(['A',Integer(4)]); S.use_d_vectors(False)
>>> S.track_mutations(False); S.mutate(Integer(1)); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
>>> S.use_fpolys(False)
>>> S.d_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate d-vectors. Need to use d vectors.

>>> S = ClusterSeed(['A',Integer(4)]); S.use_d_vectors(False)
>>> S.track_mutations(False); S.mutate(Integer(1)); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
>>> S.use_fpolys(False)
>>> S.use_d_vectors(True)
Warning: Initializing d-vectors at this point
could lead to inconsistent seed data.

>>> S.use_d_vectors(True, force=True)
>>> S.d_matrix()
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]

>>> S = ClusterSeed(['A',Integer(4)]); S.mutate(Integer(1)); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
>>> S = ClusterSeed(['A',Integer(4)])
```

(continues on next page)

(continued from previous page)

```
>>> S.use_d_vectors(True); S.mutate(Integer(1)); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
```

use_fpolys(*use=True*, *user_labels=None*, *user_labels_prefix=None*)

Use F-polynomials in our Cluster Seed.

Note: This will automatically try to recompute the cluster variables if possible

INPUT:

- `use` – boolean (default: `True`); if `True`, will use F-polynomials
- `user_labels` – (default: `None`) if set, will overwrite the default cluster variable `labels`
- `user_labels_prefix` – (default: `None`) if set, will overwrite the default

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]); S.use_fpolys(False); S._cluster
sage: S.use_fpolys(True)
sage: S.cluster()
[x0, x1, x2, x3]

sage: S = ClusterSeed(['A',4]); S.use_fpolys(False); S.track_mutations(False)
sage: S.mutate(1)
sage: S.use_fpolys(True)
Traceback (most recent call last):
...
ValueError: F-polynomials and Cluster Variables cannot be reconstructed
from given data.
sage: S.cluster()
Traceback (most recent call last):
...
ValueError: Clusters not being tracked
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)]); S.use_fpolys(False); S._cluster
>>> S.use_fpolys(True)
>>> S.cluster()
[x0, x1, x2, x3]

>>> S = ClusterSeed(['A',Integer(4)]); S.use_fpolys(False); S.track_
↪mutations(False)
>>> S.mutate(Integer(1))
>>> S.use_fpolys(True)
Traceback (most recent call last):
...
ValueError: F-polynomials and Cluster Variables cannot be reconstructed
from given data.
>>> S.cluster()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: Clusters not being tracked
```

use_g_vectors(*use=True*, *force=False*)

Reconstruct g-vectors from other data or initialize if no usable data exists.

> **Warning**
>
> Initialization may lead to inconsistent data.

INPUT:

- `use` – boolean (default: `True`); if `True`, will use g-vectors

EXAMPLES:

```
sage: S = ClusterSeed(['A',4])
sage: S.use_g_vectors(False); S.use_fpolys(False)
sage: S.use_g_vectors(True)
sage: S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: S = ClusterSeed(['A',4])
sage: S.use_g_vectors(False); S.use_fpolys(False)
sage: S.mutate(1)
sage: S.use_g_vectors(True)
sage: S.g_matrix()
[ 1  0  0  0]
[ 0 -1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]

sage: S = ClusterSeed(['A',4])
sage: S.use_g_vectors(False); S.use_fpolys(False); S.track_mutations(False)
sage: S.mutate(1)
sage: S.use_c_vectors(False)
sage: S.g_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate g-vectors. Need to use g vectors.

sage: S = ClusterSeed(['A',4])
sage: S.use_g_vectors(False); S.use_fpolys(False); S.track_mutations(False)
sage: S.mutate(1)
sage: S.use_c_vectors(False)
sage: S.use_g_vectors(True)
Warning: Initializing g-vectors at this point
could lead to inconsistent seed data.
```

(continues on next page)

(continued from previous page)

```
sage: S.use_g_vectors(True, force=True)
sage: S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_g_vectors(False); S.use_fpolys(False)
>>> S.use_g_vectors(True)
>>> S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_g_vectors(False); S.use_fpolys(False)
>>> S.mutate(Integer(1))
>>> S.use_g_vectors(True)
>>> S.g_matrix()
[ 1  0  0  0]
[ 0 -1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]

>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_g_vectors(False); S.use_fpolys(False); S.track_mutations(False)
>>> S.mutate(Integer(1))
>>> S.use_c_vectors(False)
>>> S.g_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate g-vectors. Need to use g vectors.

>>> S = ClusterSeed(['A',Integer(4)])
>>> S.use_g_vectors(False); S.use_fpolys(False); S.track_mutations(False)
>>> S.mutate(Integer(1))
>>> S.use_c_vectors(False)
>>> S.use_g_vectors(True)
Warning: Initializing g-vectors at this point
could lead to inconsistent seed data.

>>> S.use_g_vectors(True, force=True)
>>> S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

variable_class(*depth=+Infinity*, *ignore_bipartite_belt=False*)

Return all cluster variables in the mutation class of `self`.

INPUT:

- `depth` – (default: infinity) integer, only seeds with distance at most `depth` from `self` are returned
- `ignore_bipartite_belt` – boolean (default: `False`); if `True`, the algorithm does not use the bipartite belt

EXAMPLES:

- for examples see `variable_class_iter()`

variable_class_iter(*depth=+Infinity*, *ignore_bipartite_belt=False*)

Return an iterator for all cluster variables in the mutation class of `self`.

INPUT:

- `depth` – (default: infinity) integer, only seeds with distance at most `depth` from `self` are returned
- `ignore_bipartite_belt` – boolean (default: `False`); if `True`, the algorithm does not use the bipartite belt

EXAMPLES:

A standard finite type example:

```
sage: S = ClusterSeed(['A',3])
sage: it = S.variable_class_iter()
sage: for T in it: print(T)
x0
x1
x2
(x1 + 1)/x0
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
(x1 + 1)/x2
(x0*x2 + x1 + 1)/(x0*x1)
(x0*x2 + 1)/x1
(x0*x2 + x1 + 1)/(x1*x2)
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> it = S.variable_class_iter()
>>> for T in it: print(T)
x0
x1
x2
(x1 + 1)/x0
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
(x1 + 1)/x2
(x0*x2 + x1 + 1)/(x0*x1)
(x0*x2 + 1)/x1
(x0*x2 + x1 + 1)/(x1*x2)
```

Finite type examples with given depth:

```
sage: it = S.variable_class_iter(depth=1)
sage: for T in it: print(T)
Found a bipartite seed - restarting the depth counter at zero
```

(continues on next page)

(continued from previous page)

```
and constructing the variable class using its bipartite belt.
x0
x1
x2
(x1 + 1)/x0
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
(x1 + 1)/x2
(x0*x2 + x1 + 1)/(x0*x1)
(x0*x2 + 1)/x1
(x0*x2 + x1 + 1)/(x1*x2)
```

```
>>> from sage.all import *
>>> it = S.variable_class_iter(depth=Integer(1))
>>> for T in it: print(T)
Found a bipartite seed - restarting the depth counter at zero
and constructing the variable class using its bipartite belt.
x0
x1
x2
(x1 + 1)/x0
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
(x1 + 1)/x2
(x0*x2 + x1 + 1)/(x0*x1)
(x0*x2 + 1)/x1
(x0*x2 + x1 + 1)/(x1*x2)
```

Note that the notion of *depth* depends on whether a bipartite seed is found or not, or if it is manually ignored:

```
sage: it = S.variable_class_iter(depth=1, ignore_bipartite_belt=True)
sage: for T in it: print(T)
x0
x1
x2
(x1 + 1)/x2
(x0*x2 + 1)/x1
(x1 + 1)/x0

sage: S.mutate([0,1])
sage: it2 = S.variable_class_iter(depth=1)
sage: for T in it2: print(T)
(x1 + 1)/x0
(x0*x2 + x1 + 1)/(x0*x1)
x2
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
x1
(x0*x2 + 1)/x1
```

```
>>> from sage.all import *
>>> it = S.variable_class_iter(depth=Integer(1), ignore_bipartite_belt=True)
>>> for T in it: print(T)
x0
x1
```

(continues on next page)

(continued from previous page)

```
x2
(x1 + 1)/x2
(x0*x2 + 1)/x1
(x1 + 1)/x0

>>> S.mutate([Integer(0),Integer(1)])
>>> it2 = S.variable_class_iter(depth=Integer(1))
>>> for T in it2: print(T)
(x1 + 1)/x0
(x0*x2 + x1 + 1)/(x0*x1)
x2
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
x1
(x0*x2 + 1)/x1
```

Infinite type examples:

```
sage: S = ClusterSeed(['A',[1,1],1])
sage: it = S.variable_class_iter(depth=2)
sage: for T in it: print(T)
Found a bipartite seed - restarting the depth counter at zero
and constructing the variable class using its bipartite belt.
x0
x1
(x1^2 + 1)/x0
(x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
(x0^2 + 1)/x1
(x1^6 + x0^4 + 2*x0^2*x1^2 + 3*x1^4 + 2*x0^2 + 3*x1^2 + 1)/(x0^3*x1^2)
(x1^8 + x0^6 + 2*x0^4*x1^2 + 3*x0^2*x1^4 + 4*x1^6 + 3*x0^4 + 6*x0^2*x1^2 +
↪6*x1^4 + 3*x0^2 + 4*x1^2 + 1)/(x0^4*x1^3)
(x0^8 + 4*x0^6 + 3*x0^4*x1^2 + 2*x0^2*x1^4 + x1^6 + 6*x0^4 + 6*x0^2*x1^2 +
↪3*x1^4 + 4*x0^2 + 3*x1^2 + 1)/(x0^3*x1^4)
(x0^6 + 3*x0^4 + 2*x0^2*x1^2 + x1^4 + 3*x0^2 + 2*x1^2 + 1)/(x0^2*x1^3)
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',[Integer(1),Integer(1)],Integer(1)])
>>> it = S.variable_class_iter(depth=Integer(2))
>>> for T in it: print(T)
Found a bipartite seed - restarting the depth counter at zero
and constructing the variable class using its bipartite belt.
x0
x1
(x1^2 + 1)/x0
(x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
(x0^2 + 1)/x1
(x1^6 + x0^4 + 2*x0^2*x1^2 + 3*x1^4 + 2*x0^2 + 3*x1^2 + 1)/(x0^3*x1^2)
(x1^8 + x0^6 + 2*x0^4*x1^2 + 3*x0^2*x1^4 + 4*x1^6 + 3*x0^4 + 6*x0^2*x1^2 +
↪6*x1^4 + 3*x0^2 + 4*x1^2 + 1)/(x0^4*x1^3)
(x0^8 + 4*x0^6 + 3*x0^4*x1^2 + 2*x0^2*x1^4 + x1^6 + 6*x0^4 + 6*x0^2*x1^2 +
↪3*x1^4 + 4*x0^2 + 3*x1^2 + 1)/(x0^3*x1^4)
```

(continues on next page)

(continued from previous page)

```
(x0^6 + 3*x0^4 + 2*x0^2*x1^2 + x1^4 + 3*x0^2 + 2*x1^2 + 1)/(x0^2*x1^3)
```

x(*k*)

Return the k *-th initial cluster variable* for the associated cluster seed, or the cluster variable of the corresponding vertex in `self.quiver`.

EXAMPLES:

```
sage: S = ClusterSeed(['A', 3])
sage: S.mutate([2, 1])
sage: S.x(0)
x0

sage: S.x(1)
x1

sage: S.x(2)
x2

sage: dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
sage: S = ClusterSeed(dg, frozen=['c'])
sage: S.x(0)
a
sage: S.x('a')
a
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A', Integer(3)])
>>> S.mutate([Integer(2), Integer(1)])
>>> S.x(Integer(0))
x0

>>> S.x(Integer(1))
x1

>>> S.x(Integer(2))
x2

>>> dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
>>> S = ClusterSeed(dg, frozen=['c'])
>>> S.x(Integer(0))
a
>>> S.x('a')
a
```

y(*k*)

Return the k *-th initial coefficient (frozen variable)* for the associated cluster seed, or the cluster variable of the corresponding vertex in `self.quiver`.

EXAMPLES:

```
sage: S = ClusterSeed(['A', 3]).principal_extension()
sage: S.mutate([2, 1])
```

(continues on next page)

(continued from previous page)

```
sage: S.y(0)
y0

sage: S.y(1)
y1

sage: S.y(2)
y2

sage: dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
sage: S = ClusterSeed(dg, frozen=['c'])
sage: S.y(0)
c
sage: S.y('c')
c
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A', Integer(3)]).principal_extension()
>>> S.mutate([Integer(2), Integer(1)])
>>> S.y(Integer(0))
y0

>>> S.y(Integer(1))
y1

>>> S.y(Integer(2))
y2

>>> dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
>>> S = ClusterSeed(dg, frozen=['c'])
>>> S.y(Integer(0))
c
>>> S.y('c')
c
```

class sage.combinat.cluster_algebra_quiver.cluster_seed.**ClusterVariable**(*parent*, *numerator*, *denominator*, *coerce=True*, *reduce=True*, *mutation_type=None*, *variable_type=None*, *xdim=0*)

Bases: `FractionFieldElement`

This class is a thin wrapper for cluster variables in cluster seeds.

It provides the extra feature to store if a variable is frozen or not.

- the associated positive root:

```
sage: S = ClusterSeed(['A',3])
sage: for T in S.variable_class_iter():
```

(continues on next page)

(continued from previous page)

```
....:     print("{} {}".format(T, T.almost_positive_root()))
x0 -alpha[1]
x1 -alpha[2]
x2 -alpha[3]
(x1 + 1)/x0 alpha[1]
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2) alpha[1] + alpha[2] + alpha[3]
(x1 + 1)/x2 alpha[3]
(x0*x2 + x1 + 1)/(x0*x1) alpha[1] + alpha[2]
(x0*x2 + 1)/x1 alpha[2]
(x0*x2 + x1 + 1)/(x1*x2) alpha[2] + alpha[3]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> for T in S.variable_class_iter():
...     print("{} {}".format(T, T.almost_positive_root()))
x0 -alpha[1]
x1 -alpha[2]
x2 -alpha[3]
(x1 + 1)/x0 alpha[1]
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2) alpha[1] + alpha[2] + alpha[3]
(x1 + 1)/x2 alpha[3]
(x0*x2 + x1 + 1)/(x0*x1) alpha[1] + alpha[2]
(x0*x2 + 1)/x1 alpha[2]
(x0*x2 + x1 + 1)/(x1*x2) alpha[2] + alpha[3]
```

almost_positive_root()

Return the *almost positive root* associated to `self` if `self` is of finite type.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: for T in S.variable_class_iter():
....:     print("{} {}".format(T, T.almost_positive_root()))
x0 -alpha[1]
x1 -alpha[2]
x2 -alpha[3]
(x1 + 1)/x0 alpha[1]
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2) alpha[1] + alpha[2] + alpha[3]
(x1 + 1)/x2 alpha[3]
(x0*x2 + x1 + 1)/(x0*x1) alpha[1] + alpha[2]
(x0*x2 + 1)/x1 alpha[2]
(x0*x2 + x1 + 1)/(x1*x2) alpha[2] + alpha[3]
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> for T in S.variable_class_iter():
...     print("{} {}".format(T, T.almost_positive_root()))
x0 -alpha[1]
x1 -alpha[2]
x2 -alpha[3]
(x1 + 1)/x0 alpha[1]
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2) alpha[1] + alpha[2] + alpha[3]
```

(continues on next page)

(continued from previous page)

```
(x1 + 1)/x2 alpha[3]
(x0*x2 + x1 + 1)/(x0*x1) alpha[1] + alpha[2]
(x0*x2 + 1)/x1 alpha[2]
(x0*x2 + x1 + 1)/(x1*x2) alpha[2] + alpha[3]
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**PathSubset**(*n*, *m*)

Encode a *maximal* Dyck path from $(0,0)$ to (n,m) (for $n \geq m \geq 0$) as a subset of $\{0,1,2,...,2n-1\}$.

The encoding is given by indexing horizontal edges by odd numbers and vertical edges by evens.

The horizontal between (i,j) and $(i+1,j)$ is indexed by the odd number $2*i+1$. The vertical between (i,j) and $(i,j+1)$ is indexed by the even number $2*j$.

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import PathSubset
sage: PathSubset(4,0)
{1, 3, 5, 7}
sage: PathSubset(4,1)
{1, 3, 5, 6, 7}
sage: PathSubset(4,2)
{1, 2, 3, 5, 6, 7}
sage: PathSubset(4,3)
{1, 2, 3, 4, 5, 6, 7}
sage: PathSubset(4,4)
{0, 1, 2, 3, 4, 5, 6, 7}
```

```
>>> from sage.all import *
>>> from sage.combinat.cluster_algebra_quiver.cluster_seed import PathSubset
>>> PathSubset(Integer(4),Integer(0))
{1, 3, 5, 7}
>>> PathSubset(Integer(4),Integer(1))
{1, 3, 5, 6, 7}
>>> PathSubset(Integer(4),Integer(2))
{1, 2, 3, 5, 6, 7}
>>> PathSubset(Integer(4),Integer(3))
{1, 2, 3, 4, 5, 6, 7}
>>> PathSubset(Integer(4),Integer(4))
{0, 1, 2, 3, 4, 5, 6, 7}
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**SetToPath**(*T*)

Rearrange the encoding for a *maximal* Dyck path (as a set) so that it is a list in the proper order of the edges.

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import PathSubset
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import SetToPath
sage: SetToPath(PathSubset(4,0))
[1, 3, 5, 7]
sage: SetToPath(PathSubset(4,1))
[1, 3, 5, 7, 6]
sage: SetToPath(PathSubset(4,2))
[1, 3, 2, 5, 7, 6]
sage: SetToPath(PathSubset(4,3))
```

(continues on next page)

(continued from previous page)

```
[1, 3, 2, 5, 4, 7, 6]
sage: SetToPath(PathSubset(4,4))
[1, 0, 3, 2, 5, 4, 7, 6]
```

```
>>> from sage.all import *
>>> from sage.combinat.cluster_algebra_quiver.cluster_seed import PathSubset
>>> from sage.combinat.cluster_algebra_quiver.cluster_seed import SetToPath
>>> SetToPath(PathSubset(Integer(4),Integer(0)))
[1, 3, 5, 7]
>>> SetToPath(PathSubset(Integer(4),Integer(1)))
[1, 3, 5, 7, 6]
>>> SetToPath(PathSubset(Integer(4),Integer(2)))
[1, 3, 2, 5, 7, 6]
>>> SetToPath(PathSubset(Integer(4),Integer(3)))
[1, 3, 2, 5, 4, 7, 6]
>>> SetToPath(PathSubset(Integer(4),Integer(4)))
[1, 0, 3, 2, 5, 4, 7, 6]
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**coeff_recurs**(p, q, $a1$, $a2$, b, c)

Coefficients in Laurent expansion of greedy element, as defined by recursion.

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import coeff_recurs
sage: coeff_recurs(1, 1, 5, 5, 3, 3)
10
```

```
>>> from sage.all import *
>>> from sage.combinat.cluster_algebra_quiver.cluster_seed import coeff_recurs
>>> coeff_recurs(Integer(1), Integer(1), Integer(5), Integer(5), Integer(3),␣
→Integer(3))
10
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**get_green_vertices**(C)

Get the green vertices from a matrix.

Will go through each column and return the ones where no entry is greater than 0.

INPUT:

- `C` – the C-matrix to check

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import get_green_
→vertices
sage: S = ClusterSeed(['A',4]); S.mutate([1,2,3,2,0,1,2,0,3])
sage: get_green_vertices(S.c_matrix())
[0, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.cluster_algebra_quiver.cluster_seed import get_green_
→vertices
>>> S = ClusterSeed(['A',Integer(4)]); S.mutate([Integer(1),Integer(2),Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),Integer(3)])
>>> get_green_vertices(S.c_matrix())
[0, 3]
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**get_red_vertices**(C)

Get the red vertices from a matrix.

Will go through each column and return the ones where no entry is less than 0.

INPUT:

- `C` – the C-matrix to check

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import get_red_
↪vertices
sage: S = ClusterSeed(['A',4]); S.mutate([1,2,3,2,0,1,2,0,3])
sage: get_red_vertices(S.c_matrix())
[1, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.cluster_algebra_quiver.cluster_seed import get_red_vertices
>>> S = ClusterSeed(['A',Integer(4)]); S.mutate([Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),Integer(3)])
>>> get_red_vertices(S.c_matrix())
[1, 2]
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**is_LeeLiZel_allowable**(T, n, m, b, c)

Check if the subset T contributes to the computation of the greedy element $x[m, n]$ in the rank two (b, c)-cluster algebra.

This uses the conditions of Lee-Li-Zelevinsky's paper [LLZ2014].

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import is_LeeLiZel_
↪allowable
sage: is_LeeLiZel_allowable({1,3,2,5,7,6},4,2,6,6)
False
sage: is_LeeLiZel_allowable({1,2,5},3,3,1,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.cluster_algebra_quiver.cluster_seed import is_LeeLiZel_
↪allowable
>>> is_LeeLiZel_allowable({Integer(1),Integer(3),Integer(2),Integer(5),Integer(7),
↪Integer(6)},Integer(4),Integer(2),Integer(6),Integer(6))
False
>>> is_LeeLiZel_allowable({Integer(1),Integer(2),Integer(5)},Integer(3),
↪Integer(3),Integer(1),Integer(1))
True
```

5.1.19 mutation_class

This file contains helper functions for compute the mutation class of a cluster algebra or quiver.

For the compendium on the cluster algebra and quiver package see [MS2011]

AUTHORS:

- Gregg Musiker
- Christian Stump

5.1.20 Helper functions for mutation types of quivers

This file contains helper functions for detecting the mutation type of a cluster algebra or quiver.

For the compendium on the cluster algebra and quiver package see [MS2011]

AUTHORS:

- Gregg Musiker
- Christian Stump

sage.combinat.cluster_algebra_quiver.mutation_type.**is_mutation_finite**(*M*, *nr_of_checks=None*)

Use a non-deterministic method by random mutations in various directions. Can result in a wrong answer.

Warning

This method modifies the input matrix `M`!

INPUT:

- `nr_of_checks` – number of mutations applied (default: `None`); standard is 500*(number of vertices of self)

ALGORITHM:

A quiver is mutation infinite if and only if every edge label (a,-b) satisfy a*b > 4. Thus, we apply random mutations in random directions

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.mutation_type import is_mutation_
↪finite

sage: Q = ClusterQuiver(['A',10])                                          #␣
↪needs sage.modules
sage: M = Q.b_matrix()                                                     #␣
↪needs sage.modules
sage: is_mutation_finite(M)                                                #␣
↪needs sage.modules
(True, None)

sage: # needs sage.modules
sage: Q = ClusterQuiver([(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(2,9)])
sage: M = Q.b_matrix()
sage: is_mutation_finite(M)   # random
(False, [9, 6, 9, 8, 9, 4, 0, 4, 5, 2, 1, 0, 1, 0, 7, 1, 9, 2, 5, 7, 8, 6, 3, 0,␣
```

(continues on next page)

(continued from previous page)

```
↪2, 5, 4, 2, 6, 9, 2, 7, 3, 5, 3, 7, 9, 5, 9, 0, 2, 7, 9, 2, 4, 2, 1, 6, 9, 4, 3,
↪ 5, 0, 8, 2, 9, 5, 3, 7, 0, 1, 8, 3, 7, 2, 7, 3, 4, 8, 0, 4, 9, 5, 2, 8, 4, 8,␣
↪1, 7, 8, 9, 1, 5, 0, 8, 7, 4, 8, 9, 8, 0, 7, 4, 7, 1, 2, 8, 6, 1, 3, 9, 3, 9, 1,
↪ 3, 2, 4, 9, 5, 1, 2, 9, 4, 8, 5, 3, 4, 6, 8, 9, 2, 5, 9, 4, 6, 2, 1, 4, 9, 6,␣
↪0, 9, 8, 0, 4, 7, 9, 2, 1, 6])
```

```
>>> from sage.all import *
>>> from sage.combinat.cluster_algebra_quiver.mutation_type import is_mutation_
↪finite

>>> Q = ClusterQuiver(['A',Integer(10)])                                        ␣
↪     # needs sage.modules
>>> M = Q.b_matrix()                                                           #␣
↪needs sage.modules
>>> is_mutation_finite(M)                                                      #␣
↪needs sage.modules
(True, None)

>>> # needs sage.modules
>>> Q = ClusterQuiver([(Integer(0),Integer(1)),(Integer(1),Integer(2)),
↪(Integer(2),Integer(3)),(Integer(3),Integer(4)),(Integer(4),Integer(5)),
↪(Integer(5),Integer(6)),(Integer(6),Integer(7)),(Integer(7),Integer(8)),
↪(Integer(2),Integer(9))])
>>> M = Q.b_matrix()
>>> is_mutation_finite(M)  # random
(False, [9, 6, 9, 8, 9, 4, 0, 4, 5, 2, 1, 0, 1, 0, 7, 1, 9, 2, 5, 7, 8, 6, 3, 0,␣
↪2, 5, 4, 2, 6, 9, 2, 7, 3, 5, 3, 7, 9, 5, 9, 0, 2, 7, 9, 2, 4, 2, 1, 6, 9, 4, 3,
↪ 5, 0, 8, 2, 9, 5, 3, 7, 0, 1, 8, 3, 7, 2, 7, 3, 4, 8, 0, 4, 9, 5, 2, 8, 4, 8,␣
↪1, 7, 8, 9, 1, 5, 0, 8, 7, 4, 8, 9, 8, 0, 7, 4, 7, 1, 2, 8, 6, 1, 3, 9, 3, 9, 1,
↪ 3, 2, 4, 9, 5, 1, 2, 9, 4, 8, 5, 3, 4, 6, 8, 9, 2, 5, 9, 4, 6, 2, 1, 4, 9, 6,␣
↪0, 9, 8, 0, 4, 7, 9, 2, 1, 6])
```

Check that Issue #19495 is fixed:

```
sage: dg = DiGraph(); dg.add_vertex(0); S = ClusterSeed(dg); S                  #␣
↪needs sage.modules
A seed for a cluster algebra of rank 1
sage: S.is_mutation_finite()                                                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> dg = DiGraph(); dg.add_vertex(Integer(0)); S = ClusterSeed(dg); S           ␣
↪     # needs sage.modules
A seed for a cluster algebra of rank 1
>>> S.is_mutation_finite()                                                     #␣
↪needs sage.modules
True
```

sage.combinat.cluster_algebra_quiver.mutation_type.**load_data**(*user=True*)

Load a dict with keys being tuples representing exceptional QuiverMutationTypes, and with values being lists or sets containing all mutation equivalent quivers as dig6 data.

We check

- the data stored by the user (unless `user=False` was given)
- and the data installed by the optional package `database_mutation_class`.

INPUT:

- `user` – boolean (default: `True`); whether to look at user data. If not, only consider the optional package.

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.mutation_type import load_data
sage: load_data(2) # random - depends on the data the user has stored
{('G', 2): [('AO', (((0, 1), (1, -3)),)), ('AO', (((0, 1), (3, -1)),))]}
```

```
>>> from sage.all import *
>>> from sage.combinat.cluster_algebra_quiver.mutation_type import load_data
>>> load_data(Integer(2)) # random - depends on the data the user has stored
{('G', 2): [('AO', (((0, 1), (1, -3)),)), ('AO', (((0, 1), (3, -1)),))]}
```

5.1.21 Quiver

A *quiver* is an oriented graph without loops, two-cycles, or multiple edges. The edges are labelled by pairs $(i, -j)$ (with i and j being positive integers) such that the matrix $M = (m_{ab})$ with $m_{ab} = i, m_{ba} = -j$ for an edge $(i, -j)$ between vertices a and b is skew-symmetrizable.

Warning

This is not the standard definition of a quiver. Normally, in cluster algebra theory, a quiver is defined as an oriented graph without loops and two-cycles but with multiple edges allowed; the edges are unlabelled. This notion of quivers, however, can be seen as a particular case of our notion of quivers. Namely, if we have a quiver (in the regular sense of this word) with (precisely) i edges from a to b, then we represent it by a quiver (in our sense of this word) with an edge from a to b labelled by the pair $(i, -i)$.

For the compendium on the cluster algebra and quiver package see [MS2011]

AUTHORS:

- Gregg Musiker
- Christian Stump

See also

For mutation types of combinatorial quivers, see *`QuiverMutationType()`*. Cluster seeds are closely related to *`ClusterSeed()`*.

class `sage.combinat.cluster_algebra_quiver.quiver.`**`ClusterQuiver`**(*data*, *frozen=None*, *user_labels=None*)

Bases: `SageObject`

The *quiver* associated to an *exchange matrix*.

INPUT:

- `data` – can be any of the following:

```
- :class:`QuiverMutationType`
```

  - `str` – string representing a *QuiverMutationType* or a common quiver type (see Examples)
  - *ClusterQuiver*
  - `Matrix` – a skew-symmetrizable matrix
  - `DiGraph` – must be the input data for a quiver
  - List of edges – must be the edge list of a digraph for a quiver

- `frozen` – (default: `None`) sets the list of frozen variables if the input type is a `DiGraph`, it is ignored otherwise
- `user_labels` – (default: `None`) sets the names of the labels for the vertices of the quiver if the input type is not a `DiGraph`; otherwise it is ignored

EXAMPLES:

From a *QuiverMutationType*:

```
sage: Q = ClusterQuiver(['A',5]); Q
Quiver on 5 vertices of type ['A', 5]

sage: Q = ClusterQuiver(['B',2]); Q
Quiver on 2 vertices of type ['B', 2]
sage: Q2 = ClusterQuiver(['C',2]); Q2
Quiver on 2 vertices of type ['B', 2]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
True
sage: MT = Q2.mutation_type(); MT.standard_quiver() == Q2
False

sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]

sage: Q = ClusterQuiver(['A', [5,0],1]); Q
Quiver on 5 vertices of type ['D', 5]
sage: Q.is_finite()
True
sage: Q.is_acyclic()
False

sage: Q = ClusterQuiver(['F', 4, [2,1]]); Q
Quiver on 6 vertices of type ['F', 4, [1, 2]]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
False
sage: dg = Q.digraph(); Q.mutate([2,1,4,0,5,3])
sage: dg2 = Q.digraph(); dg2.is_isomorphic(dg,edge_labels=True)
False
sage: dg2.is_isomorphic(MT.standard_quiver().digraph(),edge_labels=True)
True

sage: Q = ClusterQuiver(['G',2, (3,1)]); Q
Quiver on 4 vertices of type ['G', 2, [1, 3]]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
```

(continues on next page)

(continued from previous page)

```
False

sage: Q = ClusterQuiver(['GR',[3,6]]); Q
Quiver on 4 vertices of type ['D', 4]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
False

sage: Q = ClusterQuiver(['GR',[3,7]]); Q
Quiver on 6 vertices of type ['E', 6]

sage: Q = ClusterQuiver(['TR',2]); Q
Quiver on 3 vertices of type ['A', 3]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
False
sage: Q.mutate([1,0]); MT.standard_quiver() == Q
True

sage: Q = ClusterQuiver(['TR',3]); Q
Quiver on 6 vertices of type ['D', 6]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
False
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(5)]); Q
Quiver on 5 vertices of type ['A', 5]

>>> Q = ClusterQuiver(['B',Integer(2)]); Q
Quiver on 2 vertices of type ['B', 2]
>>> Q2 = ClusterQuiver(['C',Integer(2)]); Q2
Quiver on 2 vertices of type ['B', 2]
>>> MT = Q.mutation_type(); MT.standard_quiver() == Q
True
>>> MT = Q2.mutation_type(); MT.standard_quiver() == Q2
False

>>> Q = ClusterQuiver(['A',[Integer(2),Integer(5)],Integer(1)]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]

>>> Q = ClusterQuiver(['A', [Integer(5),Integer(0)],Integer(1)]); Q
Quiver on 5 vertices of type ['D', 5]
>>> Q.is_finite()
True
>>> Q.is_acyclic()
False

>>> Q = ClusterQuiver(['F', Integer(4), [Integer(2),Integer(1)]]); Q
Quiver on 6 vertices of type ['F', 4, [1, 2]]
>>> MT = Q.mutation_type(); MT.standard_quiver() == Q
False
>>> dg = Q.digraph(); Q.mutate([Integer(2),Integer(1),Integer(4),Integer(0),
↪Integer(5),Integer(3)])
>>> dg2 = Q.digraph(); dg2.is_isomorphic(dg,edge_labels=True)
```

(continues on next page)

(continued from previous page)

```
False
>>> dg2.is_isomorphic(MT.standard_quiver().digraph(),edge_labels=True)
True

>>> Q = ClusterQuiver(['G',Integer(2), (Integer(3),Integer(1))]); Q
Quiver on 4 vertices of type ['G', 2, [1, 3]]
>>> MT = Q.mutation_type(); MT.standard_quiver() == Q
False

>>> Q = ClusterQuiver(['GR',[Integer(3),Integer(6)]]); Q
Quiver on 4 vertices of type ['D', 4]
>>> MT = Q.mutation_type(); MT.standard_quiver() == Q
False

>>> Q = ClusterQuiver(['GR',[Integer(3),Integer(7)]]); Q
Quiver on 6 vertices of type ['E', 6]

>>> Q = ClusterQuiver(['TR',Integer(2)]); Q
Quiver on 3 vertices of type ['A', 3]
>>> MT = Q.mutation_type(); MT.standard_quiver() == Q
False
>>> Q.mutate([Integer(1),Integer(0)]); MT.standard_quiver() == Q
True

>>> Q = ClusterQuiver(['TR',Integer(3)]); Q
Quiver on 6 vertices of type ['D', 6]
>>> MT = Q.mutation_type(); MT.standard_quiver() == Q
False
```

From a `ClusterQuiver`:

```
sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
sage: T = ClusterQuiver(Q); T
Quiver on 7 vertices of type ['A', [2, 5], 1]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',[Integer(2),Integer(5)],Integer(1)]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
>>> T = ClusterQuiver(Q); T
Quiver on 7 vertices of type ['A', [2, 5], 1]
```

From a Matrix:

```
sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
sage: T = ClusterQuiver(Q._M); T
Quiver on 7 vertices

sage: Q = ClusterQuiver(matrix([[0,1,-1],[-1,0,1],[1,-1,0],[1,2,3]])); Q
Quiver on 4 vertices with 1 frozen vertex
```

(continues on next page)

(continued from previous page)

```
sage: Q = ClusterQuiver(matrix([])); Q
Quiver without vertices
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',[Integer(2),Integer(5)],Integer(1)]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
>>> T = ClusterQuiver(Q._M); T
Quiver on 7 vertices

>>> Q = ClusterQuiver(matrix([[Integer(0),Integer(1),-Integer(1)],[-Integer(1),
↪Integer(0),Integer(1)],[Integer(1),-Integer(1),Integer(0)],[Integer(1),
↪Integer(2),Integer(3)]])); Q
Quiver on 4 vertices with 1 frozen vertex

>>> Q = ClusterQuiver(matrix([])); Q
Quiver without vertices
```

From a DiGraph:

```
sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
sage: T = ClusterQuiver(Q._digraph); T
Quiver on 7 vertices

sage: Q = ClusterQuiver(DiGraph([[1,2],[2,3],[3,4],[4,1]])); Q
Quiver on 4 vertices

sage: Q = ClusterQuiver(DiGraph([['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'e']]),
....:           frozen=['c']); Q
Quiver on 5 vertices with 1 frozen vertex
sage: Q.mutation_type()
[ ['A', 2], ['A', 2] ]
sage: Q
Quiver on 5 vertices of type [ ['A', 2], ['A', 2] ]
with 1 frozen vertex
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',[Integer(2),Integer(5)],Integer(1)]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
>>> T = ClusterQuiver(Q._digraph); T
Quiver on 7 vertices

>>> Q = ClusterQuiver(DiGraph([[Integer(1),Integer(2)],[Integer(2),Integer(3)],
↪[Integer(3),Integer(4)],[Integer(4),Integer(1)]])); Q
Quiver on 4 vertices

>>> Q = ClusterQuiver(DiGraph([['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'e']]),
...           frozen=['c']); Q
Quiver on 5 vertices with 1 frozen vertex
>>> Q.mutation_type()
[ ['A', 2], ['A', 2] ]
>>> Q
```

(continues on next page)

(continued from previous page)

```
Quiver on 5 vertices of type [ ['A', 2], ['A', 2] ]
with 1 frozen vertex
```

From a List of edges:

```
sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
sage: T = ClusterQuiver(Q._digraph.edges(sort=True)); T
Quiver on 7 vertices

sage: Q = ClusterQuiver([[1, 2], [2, 3], [3, 4], [4, 1]]); Q
Quiver on 4 vertices
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',[Integer(2),Integer(5)],Integer(1)]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
>>> T = ClusterQuiver(Q._digraph.edges(sort=True)); T
Quiver on 7 vertices

>>> Q = ClusterQuiver([[Integer(1), Integer(2)], [Integer(2), Integer(3)],
↪[Integer(3), Integer(4)], [Integer(4), Integer(1)]]); Q
Quiver on 4 vertices
```

b_matrix()

Return the b-matrix of `self`.

EXAMPLES:

```
sage: ClusterQuiver(['A',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: ClusterQuiver(['B',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -2  0]

sage: ClusterQuiver(['D',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1 -1]
[ 0  1  0  0]
[ 0  1  0  0]

sage: ClusterQuiver(QuiverMutationType([['A',2],['B',2]])).b_matrix()
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -2  0]
```

```
>>> from sage.all import *
>>> ClusterQuiver(['A',Integer(4)]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

>>> ClusterQuiver(['B',Integer(4)]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -2  0]

>>> ClusterQuiver(['D',Integer(4)]).b_matrix()
[ 0  1  0  0]
[-1  0 -1 -1]
[ 0  1  0  0]
[ 0  1  0  0]

>>> ClusterQuiver(QuiverMutationType([['A',Integer(2)],['B',Integer(2)]])).b_
→matrix()
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -2  0]
```

canonical_label(*certificate=False*)

Return the canonical labelling of `self`.

See `sage.graphs.generic_graph.GenericGraph.canonical_label()`.

INPUT:

- `certificate` – boolean (default: `False`); if `True`, the dictionary from `self.vertices()` to the vertices of the returned quiver is returned as well

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4]); Q.digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: T = Q.canonical_label(); T.digraph().edges(sort=True)
[(0, 3, (1, -1)), (1, 2, (1, -1)), (1, 3, (1, -1))]

sage: T, iso = Q.canonical_label(certificate=True)
sage: T.digraph().edges(sort=True); iso
[(0, 3, (1, -1)), (1, 2, (1, -1)), (1, 3, (1, -1))]
{0: 0, 1: 3, 2: 1, 3: 2}

sage: Q = ClusterQuiver(QuiverMutationType([['B',2],['A',1]])); Q
Quiver on 3 vertices of type [ ['B', 2], ['A', 1] ]

sage: Q.canonical_label()
Quiver on 3 vertices of type [ ['A', 1], ['B', 2] ]
```

(continues on next page)

(continued from previous page)

```
sage: Q.canonical_label(certificate=True)
(Quiver on 3 vertices of type [ ['A', 1], ['B', 2] ], {0: 1, 1: 2, 2: 0})
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(4)]); Q.digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

>>> T = Q.canonical_label(); T.digraph().edges(sort=True)
[(0, 3, (1, -1)), (1, 2, (1, -1)), (1, 3, (1, -1))]

>>> T, iso = Q.canonical_label(certificate=True)
>>> T.digraph().edges(sort=True); iso
[(0, 3, (1, -1)), (1, 2, (1, -1)), (1, 3, (1, -1))]
{0: 0, 1: 3, 2: 1, 3: 2}

>>> Q = ClusterQuiver(QuiverMutationType([['B',Integer(2)],['A',
→Integer(1)]])); Q
Quiver on 3 vertices of type [ ['B', 2], ['A', 1] ]

>>> Q.canonical_label()
Quiver on 3 vertices of type [ ['A', 1], ['B', 2] ]

>>> Q.canonical_label(certificate=True)
(Quiver on 3 vertices of type [ ['A', 1], ['B', 2] ], {0: 1, 1: 2, 2: 0})
```

d_vector_fan()

Return the d-vector fan associated with the quiver.

It is the fan whose maximal cones are generated by the d-matrices of the clusters.

This is a complete simplicial fan (and even smooth when the initial quiver is acyclic). It only makes sense for quivers of finite type.

EXAMPLES:

```
sage: # needs sage.geometry.polyhedron sage.libs.singular
sage: Fd = ClusterQuiver([[1,2]]).d_vector_fan(); Fd
Rational polyhedral fan in 2-d lattice N
sage: Fd.ngenerating_cones()
5
sage: Fd = ClusterQuiver([[1,2],[2,3]]).d_vector_fan(); Fd
Rational polyhedral fan in 3-d lattice N
sage: Fd.ngenerating_cones()
14
sage: Fd.is_smooth()
True
sage: Fd = ClusterQuiver([[1,2],[2,3],[3,1]]).d_vector_fan(); Fd
Rational polyhedral fan in 3-d lattice N
sage: Fd.ngenerating_cones()
14
sage: Fd.is_smooth()
False
```

```
>>> from sage.all import *
>>> # needs sage.geometry.polyhedron sage.libs.singular
>>> Fd = ClusterQuiver([[Integer(1),Integer(2)]]).d_vector_fan(); Fd
Rational polyhedral fan in 2-d lattice N
>>> Fd.ngenerating_cones()
5
>>> Fd = ClusterQuiver([[Integer(1),Integer(2)],[Integer(2),Integer(3)]]).d_
↪vector_fan(); Fd
Rational polyhedral fan in 3-d lattice N
>>> Fd.ngenerating_cones()
14
>>> Fd.is_smooth()
True
>>> Fd = ClusterQuiver([[Integer(1),Integer(2)],[Integer(2),Integer(3)],
↪[Integer(3),Integer(1)]]).d_vector_fan(); Fd
Rational polyhedral fan in 3-d lattice N
>>> Fd.ngenerating_cones()
14
>>> Fd.is_smooth()
False
```

digraph()

Return the underlying digraph of `self`.

EXAMPLES:

```
sage: ClusterQuiver(['A',1]).digraph()
Digraph on 1 vertex
sage: list(ClusterQuiver(['A',1]).digraph())
[0]
sage: ClusterQuiver(['A',1]).digraph().edges(sort=True)
[]

sage: ClusterQuiver(['A',4]).digraph()
Digraph on 4 vertices
sage: ClusterQuiver(['A',4]).digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: ClusterQuiver(['B',4]).digraph()
Digraph on 4 vertices
sage: ClusterQuiver(['A',4]).digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: ClusterQuiver(QuiverMutationType([['A',2],['B',2]])).digraph()
Digraph on 4 vertices

sage: ClusterQuiver(QuiverMutationType([['A',2],['B',2]])).digraph().
↪edges(sort=True)
[(0, 1, (1, -1)), (2, 3, (1, -2))]

sage: ClusterQuiver(['C', 4], user_labels = ['x', 'y', 'z', 'w']).digraph().
↪edges(sort=True)
[('x', 'y', (1, -1)), ('z', 'w', (2, -1)), ('z', 'y', (1, -1))]
```

```
>>> from sage.all import *
>>> ClusterQuiver(['A',Integer(1)]).digraph()
Digraph on 1 vertex
>>> list(ClusterQuiver(['A',Integer(1)]).digraph())
[0]
>>> ClusterQuiver(['A',Integer(1)]).digraph().edges(sort=True)
[]

>>> ClusterQuiver(['A',Integer(4)]).digraph()
Digraph on 4 vertices
>>> ClusterQuiver(['A',Integer(4)]).digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

>>> ClusterQuiver(['B',Integer(4)]).digraph()
Digraph on 4 vertices
>>> ClusterQuiver(['A',Integer(4)]).digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

>>> ClusterQuiver(QuiverMutationType([['A',Integer(2)],['B',Integer(2)]])).
↪digraph()
Digraph on 4 vertices

>>> ClusterQuiver(QuiverMutationType([['A',Integer(2)],['B',Integer(2)]])).
↪digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 3, (1, -2))]

>>> ClusterQuiver(['C', Integer(4)], user_labels = ['x', 'y', 'z', 'w']).
↪digraph().edges(sort=True)
[('x', 'y', (1, -1)), ('z', 'w', (2, -1)), ('z', 'y', (1, -1))]
```

exchangeable_part()

Return the restriction to the principal part (i.e. exchangeable part) of `self`, the subquiver obtained by deleting the frozen vertices of `self`.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4])
sage: T = ClusterQuiver(Q.digraph().edges(sort=True), frozen=[3])
sage: T.digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: T.exchangeable_part().digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1))]

sage: Q2 = Q.principal_extension()
sage: Q3 = Q2.principal_extension()
sage: Q2.exchangeable_part() == Q3.exchangeable_part()
True
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(4)])
>>> T = ClusterQuiver(Q.digraph().edges(sort=True), frozen=[Integer(3)])
>>> T.digraph().edges(sort=True)
```

(continues on next page)

(continued from previous page)

```
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

>>> T.exchangeable_part().digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 1, (1, -1))]

>>> Q2 = Q.principal_extension()
>>> Q3 = Q2.principal_extension()
>>> Q2.exchangeable_part() == Q3.exchangeable_part()
True
```

first_sink()

Return the first vertex of `self` that is a sink.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([1,2,4,3,2])
sage: Q.first_sink()
0
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(5)])
>>> Q.mutate([Integer(1),Integer(2),Integer(4),Integer(3),Integer(2)])
>>> Q.first_sink()
0
```

first_source()

Return the first vertex of `self` that is a source.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([2,1,3,4,2])
sage: Q.first_source()
1
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(5)])
>>> Q.mutate([Integer(2),Integer(1),Integer(3),Integer(4),Integer(2)])
>>> Q.first_source()
1
```

free_vertices()

Return the list of free vertices of `self`.

EXAMPLES:

```
sage: Q = ClusterQuiver(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd
↪']]),
....:                       frozen=['b', 'd'])
sage: Q.free_vertices()
['a', 'c', 'e']
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd
↪']]),
...                         frozen=['b', 'd'])
>>> Q.free_vertices()
['a', 'c', 'e']
```

frozen_vertices()

Return the list of frozen vertices of `self`.

EXAMPLES:

```
sage: Q = ClusterQuiver(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd
↪']]),
....:                         frozen=['b', 'd'])
sage: sorted(Q.frozen_vertices())
['b', 'd']
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd
↪']]),
...                         frozen=['b', 'd'])
>>> sorted(Q.frozen_vertices())
['b', 'd']
```

g_vector_fan()

Return the g-vector fan associated with the quiver.

It is the fan whose maximal cones are generated by the g-matrices of the clusters.

This is a complete simplicial fan. It is only supported for quivers of finite type.

EXAMPLES:

```
sage: Fg = ClusterQuiver([[1,2]]).g_vector_fan(); Fg
Rational polyhedral fan in 2-d lattice N
sage: Fg.ngenerating_cones()
5

sage: Fg = ClusterQuiver([[1,2],[2,3]]).g_vector_fan(); Fg
Rational polyhedral fan in 3-d lattice N
sage: Fg.ngenerating_cones()
14
sage: Fg.is_smooth()
True

sage: Fg = ClusterQuiver([[1,2],[2,3],[3,1]]).g_vector_fan(); Fg
Rational polyhedral fan in 3-d lattice N
sage: Fg.ngenerating_cones()
14
sage: Fg.is_smooth()
True
```

```
>>> from sage.all import *
>>> Fg = ClusterQuiver([[Integer(1),Integer(2)]]).g_vector_fan(); Fg
Rational polyhedral fan in 2-d lattice N
>>> Fg.ngenerating_cones()
5

>>> Fg = ClusterQuiver([[Integer(1),Integer(2)],[Integer(2),Integer(3)]]).g_
↪vector_fan(); Fg
Rational polyhedral fan in 3-d lattice N
>>> Fg.ngenerating_cones()
14
>>> Fg.is_smooth()
True

>>> Fg = ClusterQuiver([[Integer(1),Integer(2)],[Integer(2),Integer(3)],
↪[Integer(3),Integer(1)]]).g_vector_fan(); Fg
Rational polyhedral fan in 3-d lattice N
>>> Fg.ngenerating_cones()
14
>>> Fg.is_smooth()
True
```

interact(*fig_size=1*, *circular=True*)

Start an interactive window for cluster quiver mutations.

Only in *Jupyter notebook mode*.

INPUT:

- `fig_size` – (default: 1) factor by which the size of the plot is multiplied
- `circular` – boolean (default: `True`); if `True`, the circular plot is chosen, otherwise >>spring<< is used

is_acyclic()

Return true if `self` is acyclic.

EXAMPLES:

```
sage: ClusterQuiver(['A',4]).is_acyclic()
True

sage: ClusterQuiver(['A',[2,1],1]).is_acyclic()
True

sage: ClusterQuiver([[0,1],[1,2],[2,0]]).is_acyclic()
False
```

```
>>> from sage.all import *
>>> ClusterQuiver(['A',Integer(4)]).is_acyclic()
True

>>> ClusterQuiver(['A',[Integer(2),Integer(1)],Integer(1)]).is_acyclic()
True

>>> ClusterQuiver([[Integer(0),Integer(1)],[Integer(1),Integer(2)],
```

(continues on next page)

(continued from previous page)

```
↪[Integer(2),Integer(0)]]).is_acyclic()
False
```

is_bipartite(*return_bipartition=False*)

Return `True` if `self` is bipartite.

EXAMPLES:

```
sage: ClusterQuiver(['A',[3,3],1]).is_bipartite()
True

sage: ClusterQuiver(['A',[4,3],1]).is_bipartite()
False
```

```
>>> from sage.all import *
>>> ClusterQuiver(['A',[Integer(3),Integer(3)],Integer(1)]).is_bipartite()
True

>>> ClusterQuiver(['A',[Integer(4),Integer(3)],Integer(1)]).is_bipartite()
False
```

is_finite()

Return `True` if `self` is of finite type.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',3])
sage: Q.is_finite()
True
sage: Q = ClusterQuiver(['A',[2,2],1])
sage: Q.is_finite()
False
sage: Q = ClusterQuiver([['A',3],['B',3]])
sage: Q.is_finite()
True
sage: Q = ClusterQuiver(['T',[4,4,4]])
sage: Q.is_finite()
False
sage: Q = ClusterQuiver([['A',3],['T',[4,4,4]]])
sage: Q.is_finite()
False
sage: Q = ClusterQuiver([['A',3],['T',[2,2,3]]])
sage: Q.is_finite()
True
sage: Q = ClusterQuiver([['A',3],['D',5]])
sage: Q.is_finite()
True
sage: Q = ClusterQuiver([['A',3],['D',5,1]])
sage: Q.is_finite()
False

sage: Q = ClusterQuiver([[0,1,2],[1,2,2],[2,0,2]])
sage: Q.is_finite()
```

(continues on next page)

(continued from previous page)

```
False

sage: Q = ClusterQuiver([[0,1,2],[1,2,2],[2,0,2],[3,4,1],[4,5,1]])
sage: Q.is_finite()
False
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(3)])
>>> Q.is_finite()
True
>>> Q = ClusterQuiver(['A',[Integer(2),Integer(2)],Integer(1)])
>>> Q.is_finite()
False
>>> Q = ClusterQuiver([['A',Integer(3)],['B',Integer(3)]])
>>> Q.is_finite()
True
>>> Q = ClusterQuiver(['T',[Integer(4),Integer(4),Integer(4)]])
>>> Q.is_finite()
False
>>> Q = ClusterQuiver([['A',Integer(3)],['T',[Integer(4),Integer(4),
↪Integer(4)]]])
>>> Q.is_finite()
False
>>> Q = ClusterQuiver([['A',Integer(3)],['T',[Integer(2),Integer(2),
↪Integer(3)]]])
>>> Q.is_finite()
True
>>> Q = ClusterQuiver([['A',Integer(3)],['D',Integer(5)]])
>>> Q.is_finite()
True
>>> Q = ClusterQuiver([['A',Integer(3)],['D',Integer(5),Integer(1)]])
>>> Q.is_finite()
False

>>> Q = ClusterQuiver([[Integer(0),Integer(1),Integer(2)],[Integer(1),
↪Integer(2),Integer(2)],[Integer(2),Integer(0),Integer(2)]])
>>> Q.is_finite()
False

>>> Q = ClusterQuiver([[Integer(0),Integer(1),Integer(2)],[Integer(1),
↪Integer(2),Integer(2)],[Integer(2),Integer(0),Integer(2)],[Integer(3),
↪Integer(4),Integer(1)],[Integer(4),Integer(5),Integer(1)]])
>>> Q.is_finite()
False
```

is_mutation_finite(*nr_of_checks=None*, *return_path=False*)

Return whether `self` is mutation-finite.

This uses a non-deterministic method by random mutations in various directions. This can result in a wrong answer.

INPUT:

- `nr_of_checks` – (default: `None`) number of mutations applied; Standard is 500*(number of vertices of `self`)

- `return_path` – (default: `False`) if `True`, in case of `self` not being mutation finite, a path from `self` to a quiver with an edge label $(a, -b)$ and $a * b > 4$ is returned

ALGORITHM:

A quiver is mutation infinite if and only if every edge label (a,-b) satisfy a*b > 4. Thus, we apply random mutations in random directions

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',10])
sage: Q._mutation_type = None
sage: Q.is_mutation_finite()
True

sage: Q = ClusterQuiver([(0,1),(1,2),(2,3),(3,4),
....:                    (4,5),(5,6),(6,7),(7,8),(2,9)])
sage: Q.is_mutation_finite()
False
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(10)])
>>> Q._mutation_type = None
>>> Q.is_mutation_finite()
True

>>> Q = ClusterQuiver([(Integer(0),Integer(1)),(Integer(1),Integer(2)),
↪(Integer(2),Integer(3)),(Integer(3),Integer(4)),
...                    (Integer(4),Integer(5)),(Integer(5),Integer(6)),
↪(Integer(6),Integer(7)),(Integer(7),Integer(8)),(Integer(2),Integer(9))])
>>> Q.is_mutation_finite()
False
```

m()

Return the number of frozen vertices of `self`.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4])
sage: Q.m()
0

sage: T = ClusterQuiver(Q.digraph().edges(sort=True), frozen=[3])
sage: T.n()
3
sage: T.m()
1
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(4)])
>>> Q.m()
0

```

(continues on next page)

(continued from previous page)

```
>>> T = ClusterQuiver(Q.digraph().edges(sort=True), frozen=[Integer(3)])
>>> T.n()
3
>>> T.m()
1
```

mutate(*data*, *inplace=True*)

Mutate `self` at a sequence of vertices.

INPUT:

- `sequence` – a vertex of `self`, an iterator of vertices of `self`, a function which takes in the ClusterQuiver and returns a vertex or an iterator of vertices, or a string of the parameter wanting to be called on `ClusterQuiver` that will return a vertex or an iterator of vertices
- `inplace` – boolean (default: `True`); if `False`, the result is returned, otherwise `self` is modified

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4]); Q.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: Q.mutate(0); Q.b_matrix()
[ 0 -1  0  0]
[ 1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: T = Q.mutate(0, inplace=False); T
Quiver on 4 vertices of type ['A', 4]

sage: Q.mutate(0)
sage: Q == T
True

sage: Q.mutate([0,1,0])
sage: Q.b_matrix()
[ 0 -1  1  0]
[ 1  0  0  0]
[-1  0  0  1]
[ 0  0 -1  0]

sage: Q = ClusterQuiver(QuiverMutationType([['A',1],['A',3]]))
sage: Q.b_matrix()
[ 0  0  0  0]
[ 0  0  1  0]
[ 0 -1  0 -1]
[ 0  0  1  0]

sage: T = Q.mutate(0,inplace=False)
sage: Q == T
```

(continues on next page)

(continued from previous page)

```
True

sage: Q = ClusterQuiver(['A',3]); Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
sage: Q.mutate('first_sink'); Q.b_matrix()
[ 0 -1  0]
[ 1  0  1]
[ 0 -1  0]
sage: Q.mutate('first_source'); Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]

sage: dg = DiGraph()
sage: dg.add_vertices(['a','b','c','d','e'])
sage: dg.add_edges([['a','b'], ['b','c'], ['c','d'], ['d','e']])
sage: Q2 = ClusterQuiver(dg, frozen=['c']); Q2.b_matrix()
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -1  0]
[ 0 -1  1  0]
sage: Q2.mutate('a'); Q2.b_matrix()
[ 0 -1  0  0]
[ 1  0  0  0]
[ 0  0  0  1]
[ 0  0 -1  0]
[ 0 -1  1  0]

sage: dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
sage: Q = ClusterQuiver(dg);Q
Quiver on 3 vertices
sage: Q.mutate(['a','b'],inplace=False).digraph().edges(sort=True)
[('a', 'b', (1, -1)), ('c', 'b', (1, -1))]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(4)]); Q.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

>>> Q.mutate(Integer(0)); Q.b_matrix()
[ 0 -1  0  0]
[ 1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

>>> T = Q.mutate(Integer(0), inplace=False); T
Quiver on 4 vertices of type ['A', 4]
```

(continues on next page)

(continued from previous page)

```
>>> Q.mutate(Integer(0))
>>> Q == T
True

>>> Q.mutate([Integer(0),Integer(1),Integer(0)])
>>> Q.b_matrix()
[ 0 -1  1  0]
[ 1  0  0  0]
[-1  0  0  1]
[ 0  0 -1  0]

>>> Q = ClusterQuiver(QuiverMutationType([['A',Integer(1)],['A',Integer(3)]]))
>>> Q.b_matrix()
[ 0  0  0  0]
[ 0  0  1  0]
[ 0 -1  0 -1]
[ 0  0  1  0]

>>> T = Q.mutate(Integer(0),inplace=False)
>>> Q == T
True

>>> Q = ClusterQuiver(['A',Integer(3)]); Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
>>> Q.mutate('first_sink'); Q.b_matrix()
[ 0 -1  0]
[ 1  0  1]
[ 0 -1  0]
>>> Q.mutate('first_source'); Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]

>>> dg = DiGraph()
>>> dg.add_vertices(['a','b','c','d','e'])
>>> dg.add_edges([['a','b'], ['b','c'], ['c','d'], ['d','e']])
>>> Q2 = ClusterQuiver(dg, frozen=['c']); Q2.b_matrix()
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -1  0]
[ 0 -1  1  0]
>>> Q2.mutate('a'); Q2.b_matrix()
[ 0 -1  0  0]
[ 1  0  0  0]
[ 0  0  0  1]
[ 0  0 -1  0]
[ 0 -1  1  0]
```

(continues on next page)

(continued from previous page)

```
>>> dg = DiGraph([['a', 'b'], ['b', 'c']], format='list_of_edges')
>>> Q = ClusterQuiver(dg);Q
Quiver on 3 vertices
>>> Q.mutate(['a','b'],inplace=False).digraph().edges(sort=True)
[('a', 'b', (1, -1)), ('c', 'b', (1, -1))]
```

mutation_class(*depth=+Infinity*, *show_depth=False*, *return_paths=False*, *data_type='quiver'*, *up_to_equivalence=True*, *sink_source=False*)

Return the mutation class of `self` together with certain constraints.

INPUT:

- `depth` – (default: `infinity`) integer, only seeds with distance at most depth from `self` are returned
- `show_depth` – boolean (default: `False`); if `True`, the actual depth of the mutation is shown
- `return_paths` – boolean (default: `False`); if `True`, a shortest path of mutation sequences from `self` to the given quiver is returned as well
- `data_type` – (default: `'quiver'`) can be one of the following:
  - `'quiver'` – the quiver is returned
  - `'dig6'` – the dig6-data is returned
  - `'path'` – shortest paths of mutation sequences from `self` are returned
- `sink_source` – boolean (default: `False`); if `True`, only mutations at sinks and sources are applied

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',3])
sage: Ts = Q.mutation_class()
sage: for T in Ts: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: Ts = Q.mutation_class(depth=1)
sage: for T in Ts: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: Ts = Q.mutation_class(show_depth=True)
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

sage: Ts = Q.mutation_class(return_paths=True)
sage: for T in Ts: print(T)
(Quiver on 3 vertices of type ['A', 3], [])
(Quiver on 3 vertices of type ['A', 3], [1])
(Quiver on 3 vertices of type ['A', 3], [0])
(Quiver on 3 vertices of type ['A', 3], [0, 1])
```

(continues on next page)

(continued from previous page)

```
sage: Ts = Q.mutation_class(up_to_equivalence=False)
sage: for T in Ts: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: Ts = Q.mutation_class(return_paths=True,up_to_equivalence=False)
sage: len(Ts)
14
sage: Ts[0]
(Quiver on 3 vertices of type ['A', 3], [])

sage: Ts = Q.mutation_class(show_depth=True)
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

sage: Ts = Q.mutation_class(show_depth=True, up_to_equivalence=False)
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 4          Time: ... s
Depth: 2     found: 6          Time: ... s
Depth: 3     found: 10         Time: ... s
Depth: 4     found: 14         Time: ... s
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(3)])
>>> Ts = Q.mutation_class()
>>> for T in Ts: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

>>> Ts = Q.mutation_class(depth=Integer(1))
>>> for T in Ts: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

>>> Ts = Q.mutation_class(show_depth=True)
Depth: 0     found: 1          Time: ... s
```

(continues on next page)

(continued from previous page)

```
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

>>> Ts = Q.mutation_class(return_paths=True)
>>> for T in Ts: print(T)
(Quiver on 3 vertices of type ['A', 3], [])
(Quiver on 3 vertices of type ['A', 3], [1])
(Quiver on 3 vertices of type ['A', 3], [0])
(Quiver on 3 vertices of type ['A', 3], [0, 1])

>>> Ts = Q.mutation_class(up_to_equivalence=False)
>>> for T in Ts: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

>>> Ts = Q.mutation_class(return_paths=True,up_to_equivalence=False)
>>> len(Ts)
14
>>> Ts[Integer(0)]
(Quiver on 3 vertices of type ['A', 3], [])

>>> Ts = Q.mutation_class(show_depth=True)
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

>>> Ts = Q.mutation_class(show_depth=True, up_to_equivalence=False)
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 4          Time: ... s
Depth: 2     found: 6          Time: ... s
Depth: 3     found: 10        Time: ... s
Depth: 4     found: 14        Time: ... s
```

mutation_class_iter(*depth=+Infinity*, *show_depth=False*, *return_paths=False*, *data_type='quiver'*, *up_to_equivalence=True*, *sink_source=False*)

Return an iterator for the mutation class of `self` together with certain constraints.

INPUT:

- `depth` – integer (default: infinity); only quivers with distance at most depth from `self` are returned.
- `show_depth` – boolean (default: `False`); if `True`, the actual depth of the mutation is shown

- `return_paths` – boolean (default: `False`); if `True`, a shortest path of mutation sequences from `self` to the given quiver is returned as well
- `data_type` – (default: `'quiver'`) can be one of the following:

```
* "quiver"
* "matrix"
* "digraph"
* "dig6"
* "path"
```

- `up_to_equivalence` – boolean (default: `True`); if `True`, only one quiver for each graph-isomorphism class is recorded
- `sink_source` – boolean (default: `False`); if `True`, only mutations at sinks and sources are applied

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',3])
sage: it = Q.mutation_class_iter()
sage: for T in it: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: it = Q.mutation_class_iter(depth=1)
sage: for T in it: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: it = Q.mutation_class_iter(show_depth=True)
sage: for T in it: pass
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

sage: it = Q.mutation_class_iter(return_paths=True)
sage: for T in it: print(T)
(Quiver on 3 vertices of type ['A', 3], [])
(Quiver on 3 vertices of type ['A', 3], [1])
(Quiver on 3 vertices of type ['A', 3], [0])
(Quiver on 3 vertices of type ['A', 3], [0, 1])

sage: it = Q.mutation_class_iter(up_to_equivalence=False)
sage: for T in it: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
```

(continues on next page)

(continued from previous page)

```
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: it = Q.mutation_class_iter(return_paths=True,
....:         up_to_equivalence=False)
sage: mutation_class = list(it)
sage: len(mutation_class)
14
sage: mutation_class[0]
(Quiver on 3 vertices of type ['A', 3], [])

sage: Q = ClusterQuiver(['A',3])
sage: it = Q.mutation_class_iter(data_type='path')
sage: for T in it: print(T)
[]
[1]
[0]
[0, 1]

sage: Q = ClusterQuiver(['A',3])
sage: it = Q.mutation_class_iter(return_paths=True,
....:         data_type='matrix')
sage: next(it)
(
[ 0  0  1]
[ 0  0  1]
[-1 -1  0], []
)

sage: dg = DiGraph([['a', 'b'], ['b', 'c']],
....:         format='list_of_edges')
sage: S = ClusterQuiver(dg, frozen=['b'])
sage: S.mutation_class()
[Quiver on 3 vertices with 1 frozen vertex,
 Quiver on 3 vertices with 1 frozen vertex,
 Quiver on 3 vertices with 1 frozen vertex]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(3)])
>>> it = Q.mutation_class_iter()
>>> for T in it: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

>>> it = Q.mutation_class_iter(depth=Integer(1))
>>> for T in it: print(T)
```

(continues on next page)

(continued from previous page)

```
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

>>> it = Q.mutation_class_iter(show_depth=True)
>>> for T in it: pass
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

>>> it = Q.mutation_class_iter(return_paths=True)
>>> for T in it: print(T)
(Quiver on 3 vertices of type ['A', 3], [])
(Quiver on 3 vertices of type ['A', 3], [1])
(Quiver on 3 vertices of type ['A', 3], [0])
(Quiver on 3 vertices of type ['A', 3], [0, 1])

>>> it = Q.mutation_class_iter(up_to_equivalence=False)
>>> for T in it: print(T)
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

>>> it = Q.mutation_class_iter(return_paths=True,
...          up_to_equivalence=False)
>>> mutation_class = list(it)
>>> len(mutation_class)
14
>>> mutation_class[Integer(0)]
(Quiver on 3 vertices of type ['A', 3], [])

>>> Q = ClusterQuiver(['A',Integer(3)])
>>> it = Q.mutation_class_iter(data_type='path')
>>> for T in it: print(T)
[]
[1]
[0]
[0, 1]

>>> Q = ClusterQuiver(['A',Integer(3)])
>>> it = Q.mutation_class_iter(return_paths=True,
```

(continues on next page)

(continued from previous page)

```
...         data_type='matrix')
>>> next(it)
(
[ 0  0  1]
[ 0  0  1]
[-1 -1  0], []
)

>>> dg = DiGraph([['a', 'b'], ['b', 'c']],
...          format='list_of_edges')
>>> S = ClusterQuiver(dg, frozen=['b'])
>>> S.mutation_class()
[Quiver on 3 vertices with 1 frozen vertex,
 Quiver on 3 vertices with 1 frozen vertex,
 Quiver on 3 vertices with 1 frozen vertex]
```

mutation_sequence(*sequence*, *show_sequence=False*, *fig_size=1.2*)

Return a list containing the sequence of quivers obtained from `self` by a sequence of mutations on vertices.

INPUT:

- `sequence` – list or tuple of vertices of `self`
- `show_sequence` – boolean (default: `False`); if `True`, a png containing the mutation sequence is shown
- `fig_size` – (default: 1.2) factor by which the size of the sequence is expanded

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4])
sage: seq = Q.mutation_sequence([0,1]); seq
[Quiver on 4 vertices of type ['A', 4],
 Quiver on 4 vertices of type ['A', 4],
 Quiver on 4 vertices of type ['A', 4]]
sage: [T.b_matrix() for T in seq]
[
[ 0  1  0  0]  [ 0 -1  0  0]  [ 0  1 -1  0]
[-1  0 -1  0]  [ 1  0 -1  0]  [-1  0  1  0]
[ 0  1  0  1]  [ 0  1  0  1]  [ 1 -1  0  1]
[ 0  0 -1  0], [ 0  0 -1  0], [ 0  0 -1  0]
]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(4)])
>>> seq = Q.mutation_sequence([Integer(0),Integer(1)]); seq
[Quiver on 4 vertices of type ['A', 4],
 Quiver on 4 vertices of type ['A', 4],
 Quiver on 4 vertices of type ['A', 4]]
>>> [T.b_matrix() for T in seq]
[
[ 0  1  0  0]  [ 0 -1  0  0]  [ 0  1 -1  0]
[-1  0 -1  0]  [ 1  0 -1  0]  [-1  0  1  0]
[ 0  1  0  1]  [ 0  1  0  1]  [ 1 -1  0  1]
[ 0  0 -1  0], [ 0  0 -1  0], [ 0  0 -1  0]
```

(continues on next page)

(continued from previous page)

```
]
```

mutation_type()

Return the mutation type of `self`.

Return the mutation_type of each connected component of `self` if it can be determined, otherwise, the mutation type of this component is set to be unknown.

The mutation types of the components are ordered by vertex labels.

If you do many type recognitions, you should consider to save exceptional mutation types using `save_quiver_data()`

WARNING:

- All finite types can be detected,
- All affine types can be detected, EXCEPT affine type D (the algorithm is not yet implemented)
- All exceptional types can be detected.

EXAMPLES:

```
sage: ClusterQuiver(['A',4]).mutation_type()
['A', 4]
sage: ClusterQuiver(['A',(3,1),1]).mutation_type()
['A', [1, 3], 1]
sage: ClusterQuiver(['C',2]).mutation_type()
['B', 2]
sage: ClusterQuiver(['B',4,1]).mutation_type()
['BD', 4, 1]
```

```
>>> from sage.all import *
>>> ClusterQuiver(['A',Integer(4)]).mutation_type()
['A', 4]
>>> ClusterQuiver(['A',(Integer(3),Integer(1)),Integer(1)]).mutation_type()
['A', [1, 3], 1]
>>> ClusterQuiver(['C',Integer(2)]).mutation_type()
['B', 2]
>>> ClusterQuiver(['B',Integer(4),Integer(1)]).mutation_type()
['BD', 4, 1]
```

finite types:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q._mutation_type = None
sage: Q.mutation_type()
['A', 5]

sage: Q = ClusterQuiver([(0,1),(1,2),(2,3),(3,4)])
sage: Q.mutation_type()
['A', 5]

sage: Q = ClusterQuiver(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd
↪']]),
....:                     frozen=['c'])
```

(continues on next page)

(continued from previous page)

```
sage: Q.mutation_type()
[ ['A', 2], ['A', 2] ]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(5)])
>>> Q._mutation_type = None
>>> Q.mutation_type()
['A', 5]

>>> Q = ClusterQuiver([(Integer(0),Integer(1)),(Integer(1),Integer(2)),
↪(Integer(2),Integer(3)),(Integer(3),Integer(4))])
>>> Q.mutation_type()
['A', 5]

>>> Q = ClusterQuiver(DiGraph([['a', 'b'], ['c', 'b'], ['c', 'd'], ['e', 'd
↪']]),
...                   frozen=['c'])
>>> Q.mutation_type()
[ ['A', 2], ['A', 2] ]
```

affine types:

```
sage: Q = ClusterQuiver(['E',8,[1,1]]); Q
Quiver on 10 vertices of type ['E', 8, [1, 1]]
sage: Q._mutation_type = None; Q
Quiver on 10 vertices
sage: Q.mutation_type() # long time
['E', 8, [1, 1]]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['E',Integer(8),[Integer(1),Integer(1)]]); Q
Quiver on 10 vertices of type ['E', 8, [1, 1]]
>>> Q._mutation_type = None; Q
Quiver on 10 vertices
>>> Q.mutation_type() # long time
['E', 8, [1, 1]]
```

the not yet working affine type D (unless user has saved small classical quiver data):

```
sage: Q = ClusterQuiver(['D',4,1])
sage: Q._mutation_type = None
sage: Q.mutation_type() # todo: not implemented
['D', 4, 1]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['D',Integer(4),Integer(1)])
>>> Q._mutation_type = None
>>> Q.mutation_type() # todo: not implemented
['D', 4, 1]
```

the exceptional types:

```
sage: Q = ClusterQuiver(['X',6])
sage: Q._mutation_type = None
sage: Q.mutation_type() # long time
['X', 6]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['X',Integer(6)])
>>> Q._mutation_type = None
>>> Q.mutation_type() # long time
['X', 6]
```

examples from page 8 of [Ke2008]:

```
sage: dg = DiGraph(); dg.add_edges([(9,0),(9,4),(4,6),(6,7),(7,8),(8,3),(3,5),
↪(5,6),(8,1),(2,3)])
sage: ClusterQuiver(dg).mutation_type() # long time
['E', 8, [1, 1]]

sage: dg = DiGraph({ 0:[3], 1:[0,4], 2:[0,6], 3:[1,2,7], 4:[3,8], 5:[2], 6:[3,
↪5], 7:[4,6], 8:[7] })
sage: ClusterQuiver(dg).mutation_type() # long time
['E', 8, 1]

sage: dg = DiGraph({ 0:[3,9], 1:[0,4], 2:[0,6], 3:[1,2,7], 4:[3,8], 5:[2],␣
↪6:[3,5], 7:[4,6], 8:[7], 9:[1] })
sage: ClusterQuiver(dg).mutation_type() # long time
['E', 8, [1, 1]]
```

```
>>> from sage.all import *
>>> dg = DiGraph(); dg.add_edges([(Integer(9),Integer(0)),(Integer(9),
↪Integer(4)),(Integer(4),Integer(6)),(Integer(6),Integer(7)),(Integer(7),
↪Integer(8)),(Integer(8),Integer(3)),(Integer(3),Integer(5)),(Integer(5),
↪Integer(6)),(Integer(8),Integer(1)),(Integer(2),Integer(3))])
>>> ClusterQuiver(dg).mutation_type() # long time
['E', 8, [1, 1]]

>>> dg = DiGraph({ Integer(0):[Integer(3)], Integer(1):[Integer(0),
↪Integer(4)], Integer(2):[Integer(0),Integer(6)], Integer(3):[Integer(1),
↪Integer(2),Integer(7)], Integer(4):[Integer(3),Integer(8)],␣
↪Integer(5):[Integer(2)], Integer(6):[Integer(3),Integer(5)],␣
↪Integer(7):[Integer(4),Integer(6)], Integer(8):[Integer(7)] })
>>> ClusterQuiver(dg).mutation_type() # long time
['E', 8, 1]

>>> dg = DiGraph({ Integer(0):[Integer(3),Integer(9)], Integer(1):[Integer(0),
↪Integer(4)], Integer(2):[Integer(0),Integer(6)], Integer(3):[Integer(1),
↪Integer(2),Integer(7)], Integer(4):[Integer(3),Integer(8)],␣
↪Integer(5):[Integer(2)], Integer(6):[Integer(3),Integer(5)],␣
↪Integer(7):[Integer(4),Integer(6)], Integer(8):[Integer(7)],␣
↪Integer(9):[Integer(1)] })
>>> ClusterQuiver(dg).mutation_type() # long time
['E', 8, [1, 1]]
```

infinite types:

```
sage: Q = ClusterQuiver(['GR',[4,9]])
sage: Q._mutation_type = None
sage: Q.mutation_type()
'undetermined infinite mutation type'
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['GR',[Integer(4),Integer(9)]])
>>> Q._mutation_type = None
>>> Q.mutation_type()
'undetermined infinite mutation type'
```

reducible types:

```
sage: Q = ClusterQuiver([['A', 3], ['B', 3]])
sage: Q._mutation_type = None
sage: Q.mutation_type()
[ ['A', 3], ['B', 3] ]

sage: Q = ClusterQuiver([['A', 3], ['T', [4,4,4]]])
sage: Q._mutation_type = None
sage: Q.mutation_type()
[['A', 3], 'undetermined infinite mutation type']

sage: Q = ClusterQuiver([['A', 3], ['B', 3], ['T', [4,4,4]]])
sage: Q._mutation_type = None
sage: Q.mutation_type()
[['A', 3], ['B', 3], 'undetermined infinite mutation type']

sage: Q = ClusterQuiver([[0,1,2],[1,2,2],[2,0,2],[3,4,1],[4,5,1]])
sage: Q.mutation_type()
['undetermined finite mutation type', ['A', 3]]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver([['A', Integer(3)], ['B', Integer(3)]])
>>> Q._mutation_type = None
>>> Q.mutation_type()
[ ['A', 3], ['B', 3] ]

>>> Q = ClusterQuiver([['A', Integer(3)], ['T', [Integer(4),Integer(4),
↪Integer(4)]]])
>>> Q._mutation_type = None
>>> Q.mutation_type()
[['A', 3], 'undetermined infinite mutation type']

>>> Q = ClusterQuiver([['A', Integer(3)], ['B', Integer(3)], ['T',␣
↪[Integer(4),Integer(4),Integer(4)]]])
>>> Q._mutation_type = None
>>> Q.mutation_type()
[['A', 3], ['B', 3], 'undetermined infinite mutation type']

>>> Q = ClusterQuiver([[Integer(0),Integer(1),Integer(2)],[Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(2)],[Integer(2),Integer(0),Integer(2)],[Integer(3),
↪Integer(4),Integer(1)],[Integer(4),Integer(5),Integer(1)]])
>>> Q.mutation_type()
['undetermined finite mutation type', ['A', 3]]
```

n()

Return the number of free vertices of `self`.

EXAMPLES:

```
sage: ClusterQuiver(['A',4]).n()
4
sage: ClusterQuiver(['A',(3,1),1]).n()
4
sage: ClusterQuiver(['B',4]).n()
4
sage: ClusterQuiver(['B',4,1]).n()
5
```

```
>>> from sage.all import *
>>> ClusterQuiver(['A',Integer(4)]).n()
4
>>> ClusterQuiver(['A',(Integer(3),Integer(1)),Integer(1)]).n()
4
>>> ClusterQuiver(['B',Integer(4)]).n()
4
>>> ClusterQuiver(['B',Integer(4),Integer(1)]).n()
5
```

number_of_edges()

Return the total number of edges on the quiver.

> **Note**
>
> This only works with non-valued quivers. If used on a non-valued quiver then the positive value is taken to be the number of edges added.

OUTPUT: integer of the number of edges

EXAMPLES:

```
sage: S = ClusterQuiver(['A',4]); S.number_of_edges()
3

sage: S = ClusterQuiver(['B',4]); S.number_of_edges()
3
```

```
>>> from sage.all import *
>>> S = ClusterQuiver(['A',Integer(4)]); S.number_of_edges()
3
```

(continues on next page)

(continued from previous page)

```
>>> S = ClusterQuiver(['B',Integer(4)]); S.number_of_edges()
3
```

plot(*circular=True*, *center=(0, 0)*, *directed=True*, *mark=None*, *save_pos=False*, *greens=[]*)

Return the plot of the underlying digraph of `self`.

INPUT:

- `circular` – boolean (default: `True`); if `True`, the circular plot is chosen, otherwise >>spring<< is used
- `center` – (default: (0,0)) sets the center of the circular plot, otherwise it is ignored
- `directed` – boolean (default: `True`); if `True`, the directed version is shown, otherwise the undirected
- `mark` – (default: `None`) if set to i, the vertex i is highlighted
- `save_pos` – boolean (default: `False`); if `True`, the positions of the vertices are saved
- `greens` – (default: `[]`) if set to a list, will display the green vertices as green

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q.plot()                                                          #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
sage: Q.plot(circular=True)                                             #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
sage: Q.plot(circular=True, mark=1)                                     #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(5)])
>>> Q.plot()                                                          #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
>>> Q.plot(circular=True)                                             #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
>>> Q.plot(circular=True, mark=Integer(1))                              ␣
↪     # needs sage.plot sage.symbolic
Graphics object consisting of 15 graphics primitives
```

poincare_semistable(*theta*, *d*)

Return the Poincaré polynomial of the moduli space of semi-stable representations of dimension vector d.

INPUT:

- `theta` – stability weight, as list or vector of rationals
- `d` – dimension vector, as list or vector of coprime integers

The semi-stability is taken with respect to the slope function

$$\mu(d) = \theta(d)/\dim(d)$$

where d is a dimension vector.

This uses the matrix-inversion algorithm from [Rei2002].

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',2])
sage: Q.poincare_semistable([1,0],[1,0])
1
sage: Q.poincare_semistable([1,0],[1,1])
1

sage: K2 = ClusterQuiver(matrix([[0,2],[-2,0]]))
sage: theta = (1, 0)
sage: K2.poincare_semistable(theta, [1,0])
1
sage: K2.poincare_semistable(theta, [1,1])
v^2 + 1
sage: K2.poincare_semistable(theta, [1,2])
1
sage: K2.poincare_semistable(theta, [1,3])
0

sage: K3 = ClusterQuiver(matrix([[0,3],[-3,0]]))
sage: theta = (1, 0)
sage: K3.poincare_semistable(theta, (2,3))
v^12 + v^10 + 3*v^8 + 3*v^6 + 3*v^4 + v^2 + 1
sage: K3.poincare_semistable(theta, (3,4))(1)
68
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(2)])
>>> Q.poincare_semistable([Integer(1),Integer(0)],[Integer(1),Integer(0)])
1
>>> Q.poincare_semistable([Integer(1),Integer(0)],[Integer(1),Integer(1)])
1

>>> K2 = ClusterQuiver(matrix([[Integer(0),Integer(2)],[-Integer(2),
↪Integer(0)]]))
>>> theta = (Integer(1), Integer(0))
>>> K2.poincare_semistable(theta, [Integer(1),Integer(0)])
1
>>> K2.poincare_semistable(theta, [Integer(1),Integer(1)])
v^2 + 1
>>> K2.poincare_semistable(theta, [Integer(1),Integer(2)])
1
>>> K2.poincare_semistable(theta, [Integer(1),Integer(3)])
0

>>> K3 = ClusterQuiver(matrix([[Integer(0),Integer(3)],[-Integer(3),
↪Integer(0)]]))
>>> theta = (Integer(1), Integer(0))
>>> K3.poincare_semistable(theta, (Integer(2),Integer(3)))
v^12 + v^10 + 3*v^8 + 3*v^6 + 3*v^4 + v^2 + 1
>>> K3.poincare_semistable(theta, (Integer(3),Integer(4)))(Integer(1))
68
```

REFERENCES:

principal_extension(*inplace=False*)

Return the principal extension of `self`, adding n frozen vertices to any previously frozen vertices.

This is the quiver obtained by adding an outgoing edge to every mutable vertex of `self`.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',2]); Q
Quiver on 2 vertices of type ['A', 2]
sage: T = Q.principal_extension(); T
Quiver on 4 vertices of type ['A', 2] with 2 frozen vertices
sage: T2 = T.principal_extension(); T2
Quiver on 6 vertices of type ['A', 2] with 4 frozen vertices
sage: Q.digraph().edges(sort=True)
[(0, 1, (1, -1))]
sage: T.digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 0, (1, -1)), (3, 1, (1, -1))]
sage: T2.digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 0, (1, -1)), (3, 1, (1, -1)),
 (4, 0, (1, -1)), (5, 1, (1, -1))]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(2)]); Q
Quiver on 2 vertices of type ['A', 2]
>>> T = Q.principal_extension(); T
Quiver on 4 vertices of type ['A', 2] with 2 frozen vertices
>>> T2 = T.principal_extension(); T2
Quiver on 6 vertices of type ['A', 2] with 4 frozen vertices
>>> Q.digraph().edges(sort=True)
[(0, 1, (1, -1))]
>>> T.digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 0, (1, -1)), (3, 1, (1, -1))]
>>> T2.digraph().edges(sort=True)
[(0, 1, (1, -1)), (2, 0, (1, -1)), (3, 1, (1, -1)),
 (4, 0, (1, -1)), (5, 1, (1, -1))]
```

qmu_save(*filename=None*)

Save `self` in a `.qmu` file.

This file can then be opened in Bernhard Keller's Quiver Applet.

See https://webusers.imj-prg.fr/~bernhard.keller/quivermutation/

INPUT:

- `filename` – the filename the image is saved to

If a filename is not specified, the default name is `from_sage.qmu` in the current sage directory.

EXAMPLES:

```
sage: Q = ClusterQuiver(['F',4,[1,2]])
sage: import tempfile
sage: with tempfile.NamedTemporaryFile(suffix='.qmu') as f:                   #␣
↪needs sage.plot sage.symbolic
....:     Q.qmu_save(f.name)
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['F',Integer(4),[Integer(1),Integer(2)]])
>>> import tempfile
>>> with tempfile.NamedTemporaryFile(suffix='.qmu') as f:                    #␣
↪needs sage.plot sage.symbolic
...     Q.qmu_save(f.name)
```

Make sure we can save quivers with $m! = n$ frozen variables, see Issue #14851:

```
sage: S = ClusterSeed(['A',3])
sage: T1 = S.principal_extension()
sage: Q = T1.quiver()
sage: import tempfile
sage: with tempfile.NamedTemporaryFile(suffix='.qmu') as f:                  #␣
↪needs sage.plot sage.symbolic
....:     Q.qmu_save(f.name)
```

```
>>> from sage.all import *
>>> S = ClusterSeed(['A',Integer(3)])
>>> T1 = S.principal_extension()
>>> Q = T1.quiver()
>>> import tempfile
>>> with tempfile.NamedTemporaryFile(suffix='.qmu') as f:                    #␣
↪needs sage.plot sage.symbolic
...     Q.qmu_save(f.name)
```

relabel (*relabelling*, *inplace=True*)

Return the quiver after doing a relabelling.

Will relabel the vertices of the quiver.

INPUT:

- `relabelling` – dictionary of labels to move around
- `inplace` – boolean (default: `True`); if `True`, will return a duplicate of the quiver

EXAMPLES:

```
sage: S = ClusterQuiver(['A',4]).relabel({1:'5',2:'go'})
```

```
>>> from sage.all import *
>>> S = ClusterQuiver(['A',Integer(4)]).relabel({Integer(1):'5',Integer(2):'go
↪'})
```

reorient (*data*)

Reorient `self` with respect to the given total order, or with respect to an iterator of edges in `self` to be reverted.

Warning

This operation might change the mutation type of `self`.

INPUT:

- `data` – an iterator defining a total order on `self.vertices()`, or an iterator of edges in `self` to be reoriented.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',(2,3),1])
sage: Q.mutation_type()
['A', [2, 3], 1]

sage: Q.reorient([(0,1),(1,2),(2,3),(3,4)])
sage: Q.mutation_type()
['D', 5]

sage: Q.reorient([0,1,2,3,4])
sage: Q.mutation_type()
['A', [1, 4], 1]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',(Integer(2),Integer(3)),Integer(1)])
>>> Q.mutation_type()
['A', [2, 3], 1]

>>> Q.reorient([(Integer(0),Integer(1)),(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(3),Integer(4))])
>>> Q.mutation_type()
['D', 5]

>>> Q.reorient([Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)])
>>> Q.mutation_type()
['A', [1, 4], 1]
```

save_image(*filename*, *circular=False*)

Save the plot of the underlying digraph of `self`.

INPUT:

- `filename` – the filename the image is saved to
- `circular` – boolean (default: `False`); if `True`, the circular plot is chosen, otherwise >>spring<< is used

EXAMPLES:

```
sage: Q = ClusterQuiver(['F',4,[1,2]])
sage: import tempfile
sage: with tempfile.NamedTemporaryFile(suffix='.png') as f:                #␣
↪needs sage.plot sage.symbolic
....:     Q.save_image(f.name)
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['F',Integer(4),[Integer(1),Integer(2)]])
>>> import tempfile
>>> with tempfile.NamedTemporaryFile(suffix='.png') as f:                #␣
↪needs sage.plot sage.symbolic
...     Q.save_image(f.name)
```

show(*fig_size=1*, *circular=False*, *directed=True*, *mark=None*, *save_pos=False*, *greens=[]*)

Show the plot of the underlying digraph of `self`.

INPUT:

- `fig_size` – (default: 1) factor by which the size of the plot is multiplied
- `circular` – boolean (default: `False`); if `True`, the circular plot is chosen, otherwise >>spring<< is used
- `directed` – boolean (default: `True`); if `True`, the directed version is shown, otherwise the undirected
- `mark` – boolean (default: `None`); if set to i, the vertex i is highlighted
- `save_pos` – boolean (default: `False`); if `True`, the positions of the vertices are saved
- `greens` – (default: `[]`) if set to a list, will display the green vertices as green

sinks()

Return all vertices of `self` that are sinks.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([1,2,4,3,2])
sage: Q.sinks()
[0, 2]

sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([2,1,3,4,2])
sage: Q.sinks()
[3]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(5)])
>>> Q.mutate([Integer(1),Integer(2),Integer(4),Integer(3),Integer(2)])
>>> Q.sinks()
[0, 2]

>>> Q = ClusterQuiver(['A',Integer(5)])
>>> Q.mutate([Integer(2),Integer(1),Integer(3),Integer(4),Integer(2)])
>>> Q.sinks()
[3]
```

sources()

Return all vertices of `self` that are sources.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([1,2,4,3,2])
sage: Q.sources()
[]

sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([2,1,3,4,2])
sage: Q.sources()
[1]
```

```
>>> from sage.all import *
>>> Q = ClusterQuiver(['A',Integer(5)])
>>> Q.mutate([Integer(1),Integer(2),Integer(4),Integer(3),Integer(2)])
>>> Q.sources()
[]

>>> Q = ClusterQuiver(['A',Integer(5)])
>>> Q.mutate([Integer(2),Integer(1),Integer(3),Integer(4),Integer(2)])
>>> Q.sources()
[1]
```

5.1.22 Quiver mutation types

AUTHORS:

- Gregg Musiker (2012, initial version)
- Christian Stump (2012, initial version)
- Hugh Thomas (2012, initial version)

sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationType**(**args*)

Quiver mutation types **can be seen as a slight generalization of**

generalized Cartan types.

Background on generalized Cartan types can be found at

Wikipedia article Generalized_Cartan_matrix

For the compendium on the cluster algebra and quiver package in Sage see [MS2011]

A B-matrix is a skew-symmetrizable $(n \times n)$-matrix M. I.e., there exists an invertible diagonal matrix D such that DM is skew-symmetric. M can be encoded as a *quiver* by having a directed edge from vertex i to vertex j with label (a, b) if $a = M_{i,j} > 0$ and $b = M_{j,i} < 0$. We consider quivers up to *mutation equivalence*.

To a quiver mutation type we can associate a *generalized Cartan type* by sending M to the generalized Cartan matrix $C(M)$ obtained by replacing all positive entries by their negatives and adding 2's on the main diagonal.

`QuiverMutationType` constructs a quiver mutation type object. For more detail on the possible different types, please see the compendium.

INPUT:

The input consists either of a quiver mutation type, or of a `letter` (a string), a `rank` (one integer or a list/tuple of integers), and an optional `twist` (an integer or a list of integers). There are several different naming conventions for quiver mutation types.

- Finite type – `letter` is a Dynkin type (A-G), and `rank` is the rank.
- Affine type – there is more than one convention for naming affine types.
  - Kac's notation: `letter` is a Dynkin type, `rank` is the rank of the associated finite Dynkin diagram, and `twist` is the twist, which could be 1, 2, or 3. In the special case of affine type A, there is more than one quiver mutation type associated to the Cartan type. In this case only, `rank` is a pair of integers (i,j), giving the number of edges pointing clockwise and the number of edges pointing counter-clockwise. The total number of vertices is given by i+j in this case.
  - Naive notation: `letter` is one of 'BB', 'BC', 'BD', 'CC', 'CD'. The name specifies the two ends of the diagram, which are joined by a path. The total number of vertices is given by `rank +1` (to match the

indexing people expect because these are affine types). In general, `rank` must be large enough for the picture to make sense, but we accept `letter` is `BC` and `rank=1`.

  – Macdonald notation: for the dual of an untwisted affine type (such as ['C', 6, 1]), we accept a twist of -1 (i.e., ['C',6,-1]).

- Elliptic type – `letter` is a Dynkin type, `rank` is the rank of the finite Dynkin diagram, and `twist` is a tuple of two integers. We follow Saito's notation.
- Other shapes:
  – Rank 2: `letter` is 'R2', and `rank` is a pair of integers specifying the label on the unique edge.
  – Triangle: `letter` is `TR`, and `rank` is the number of vertices along a side.
  – T: This defines a quiver shaped like a T. `letter` is 'T', and the `rank` is a triple, whose entries specify the number of vertices along each path from the branch point (counting the branch point).
  – Grassmannian: This defines the cluster algebra (without coefficients) corresponding to the cluster algebra with coefficients which is the coordinate ring of a Grassmannian. `letter` is 'GR'. `rank` is a pair of integers (k, n) with 'k' < 'n' specifying the Grassmannian of k-planes in n-space. This defines a quiver given by a (k-1) x (n-k-1) grid where each square is cyclically oriented.
  – Exceptional mutation finite quivers: The two exceptional mutation finite quivers, found by Derksen-Owen, have `letter` as 'X' and `rank` 6 or 7, equal to the number of vertices.
  – AE, BE, CE, DE: Quivers are built of one end which looks like type (affine A), B, C, or D, and the other end which looks like type E (i.e., it consists of two antennae, one of length one, and one of length two). `letter` is 'AE', 'BE', 'CE', or 'DE', and `rank` is the total number of vertices. Note that 'AE' is of a slightly different form and requires `rank` to be a pair of integers (i,j) just as in the case of affine type A. See Exercise 4.3 in Kac's book Infinite Dimensional Lie Algebras for more details.
  – Infinite type E: It is also possible to obtain infinite-type E quivers by specifying `letter` as 'E' and `rank` as the number of vertices.

REFERENCES:

- A good reference for finite and affine Dynkin diagrams, including Kac's notation, is the Wikipedia article Dynkin_diagram.
- A good reference for the skew-symmetrizable elliptic diagrams is "Cluster algebras of finite mutation type via unfolding" by A. Felikson, M. Shapiro, and P. Tumarkin, [FST2012].

EXAMPLES:

Finite types:

```
sage: QuiverMutationType('A', 1)
['A', 1]
sage: QuiverMutationType('A',5)
['A', 5]

sage: QuiverMutationType('B', 2)
['B', 2]
sage: QuiverMutationType('B',5)
['B', 5]

sage: QuiverMutationType('C', 2)
['B', 2]
sage: QuiverMutationType('C',5)
['C', 5]
```

(continues on next page)

(continued from previous page)

```
sage: QuiverMutationType('D', 2)
[ ['A', 1], ['A', 1] ]
sage: QuiverMutationType('D',3)
['A', 3]
sage: QuiverMutationType('D',4)
['D', 4]

sage: QuiverMutationType('E',6)
['E', 6]

sage: QuiverMutationType('G', 2)
['G', 2]

sage: QuiverMutationType('A',(1,0), 1)
['A', 1]

sage: QuiverMutationType('A',(2,0), 1)
[ ['A', 1], ['A', 1] ]

sage: QuiverMutationType('A',(7,0), 1)
['D', 7]
```

```
>>> from sage.all import *
>>> QuiverMutationType('A', Integer(1))
['A', 1]
>>> QuiverMutationType('A',Integer(5))
['A', 5]

>>> QuiverMutationType('B', Integer(2))
['B', 2]
>>> QuiverMutationType('B',Integer(5))
['B', 5]

>>> QuiverMutationType('C', Integer(2))
['B', 2]
>>> QuiverMutationType('C',Integer(5))
['C', 5]

>>> QuiverMutationType('D', Integer(2))
[ ['A', 1], ['A', 1] ]
>>> QuiverMutationType('D',Integer(3))
['A', 3]
>>> QuiverMutationType('D',Integer(4))
['D', 4]

>>> QuiverMutationType('E',Integer(6))
['E', 6]

>>> QuiverMutationType('G', Integer(2))
['G', 2]
```

(continues on next page)

(continued from previous page)

```
>>> QuiverMutationType('A',(Integer(1),Integer(0)), Integer(1))
['A', 1]

>>> QuiverMutationType('A',(Integer(2),Integer(0)), Integer(1))
[ ['A', 1], ['A', 1] ]

>>> QuiverMutationType('A',(Integer(7),Integer(0)), Integer(1))
['D', 7]
```

Affine types:

```
sage: QuiverMutationType('A',(1, 1), 1)
['A', [1, 1], 1]
sage: QuiverMutationType('A',(2,4), 1)
['A', [2, 4], 1]

sage: QuiverMutationType('BB', 2, 1)
['BB', 2, 1]
sage: QuiverMutationType('BB',4, 1)
['BB', 4, 1]

sage: QuiverMutationType('CC', 2, 1)
['CC', 2, 1]
sage: QuiverMutationType('CC',4, 1)
['CC', 4, 1]

sage: QuiverMutationType('BC', 1, 1)
['BC', 1, 1]
sage: QuiverMutationType('BC',5, 1)
['BC', 5, 1]

sage: QuiverMutationType('BD',3, 1)
['BD', 3, 1]
sage: QuiverMutationType('BD',5, 1)
['BD', 5, 1]

sage: QuiverMutationType('CD',3, 1)
['CD', 3, 1]
sage: QuiverMutationType('CD',5, 1)
['CD', 5, 1]

sage: QuiverMutationType('D',4, 1)
['D', 4, 1]
sage: QuiverMutationType('D',6, 1)
['D', 6, 1]

sage: QuiverMutationType('E',6, 1)
['E', 6, 1]
sage: QuiverMutationType('E',7, 1)
['E', 7, 1]
sage: QuiverMutationType('E',8, 1)
['E', 8, 1]
```

(continues on next page)

(continued from previous page)

```
sage: QuiverMutationType('F',4, 1)
['F', 4, 1]
sage: QuiverMutationType('F',4,-1)
['F', 4, -1]

sage: QuiverMutationType('G', 2, 1)
['G', 2, 1]
sage: QuiverMutationType('G', 2,-1)
['G', 2, -1]
sage: QuiverMutationType('A',3, 2) == QuiverMutationType('D',3, 2)
True
```

```
>>> from sage.all import *
>>> QuiverMutationType('A',(Integer(1), Integer(1)), Integer(1))
['A', [1, 1], 1]
>>> QuiverMutationType('A',(Integer(2),Integer(4)), Integer(1))
['A', [2, 4], 1]

>>> QuiverMutationType('BB', Integer(2), Integer(1))
['BB', 2, 1]
>>> QuiverMutationType('BB',Integer(4), Integer(1))
['BB', 4, 1]

>>> QuiverMutationType('CC', Integer(2), Integer(1))
['CC', 2, 1]
>>> QuiverMutationType('CC',Integer(4), Integer(1))
['CC', 4, 1]

>>> QuiverMutationType('BC', Integer(1), Integer(1))
['BC', 1, 1]
>>> QuiverMutationType('BC',Integer(5), Integer(1))
['BC', 5, 1]

>>> QuiverMutationType('BD',Integer(3), Integer(1))
['BD', 3, 1]
>>> QuiverMutationType('BD',Integer(5), Integer(1))
['BD', 5, 1]

>>> QuiverMutationType('CD',Integer(3), Integer(1))
['CD', 3, 1]
>>> QuiverMutationType('CD',Integer(5), Integer(1))
['CD', 5, 1]

>>> QuiverMutationType('D',Integer(4), Integer(1))
['D', 4, 1]
>>> QuiverMutationType('D',Integer(6), Integer(1))
['D', 6, 1]

>>> QuiverMutationType('E',Integer(6), Integer(1))
['E', 6, 1]
>>> QuiverMutationType('E',Integer(7), Integer(1))
```

(continues on next page)

(continued from previous page)

```
['E', 7, 1]
>>> QuiverMutationType('E',Integer(8), Integer(1))
['E', 8, 1]

>>> QuiverMutationType('F',Integer(4), Integer(1))
['F', 4, 1]
>>> QuiverMutationType('F',Integer(4),-Integer(1))
['F', 4, -1]

>>> QuiverMutationType('G', Integer(2), Integer(1))
['G', 2, 1]
>>> QuiverMutationType('G', Integer(2),-Integer(1))
['G', 2, -1]
>>> QuiverMutationType('A',Integer(3), Integer(2)) == QuiverMutationType('D',
↪Integer(3), Integer(2))
True
```

Affine types using Kac's Notation:

```
sage: QuiverMutationType('A', 1, 1)
['A', [1, 1], 1]
sage: QuiverMutationType('B',5, 1)
['BD', 5, 1]
sage: QuiverMutationType('C',5, 1)
['CC', 5, 1]
sage: QuiverMutationType('A', 2, 2)
['BC', 1, 1]
sage: QuiverMutationType('A',7, 2)
['CD', 4, 1]
sage: QuiverMutationType('A',8, 2)
['BC', 4, 1]
sage: QuiverMutationType('D',6, 2)
['BB', 5, 1]
sage: QuiverMutationType('E',6, 2)
['F', 4, -1]
sage: QuiverMutationType('D',4,3)
['G', 2, -1]
```

```
>>> from sage.all import *
>>> QuiverMutationType('A', Integer(1), Integer(1))
['A', [1, 1], 1]
>>> QuiverMutationType('B',Integer(5), Integer(1))
['BD', 5, 1]
>>> QuiverMutationType('C',Integer(5), Integer(1))
['CC', 5, 1]
>>> QuiverMutationType('A', Integer(2), Integer(2))
['BC', 1, 1]
>>> QuiverMutationType('A',Integer(7), Integer(2))
['CD', 4, 1]
>>> QuiverMutationType('A',Integer(8), Integer(2))
['BC', 4, 1]
>>> QuiverMutationType('D',Integer(6), Integer(2))
```

(continues on next page)

(continued from previous page)

```
['BB', 5, 1]
>>> QuiverMutationType('E',Integer(6), Integer(2))
['F', 4, -1]
>>> QuiverMutationType('D',Integer(4),Integer(3))
['G', 2, -1]
```

Elliptic types:

```
sage: QuiverMutationType('E',6,[1, 1])
['E', 6, [1, 1]]
sage: QuiverMutationType('F',4,[2, 1])
['F', 4, [1, 2]]
sage: QuiverMutationType('G', 2,[3,3])
['G', 2, [3, 3]]
```

```
>>> from sage.all import *
>>> QuiverMutationType('E',Integer(6),[Integer(1), Integer(1)])
['E', 6, [1, 1]]
>>> QuiverMutationType('F',Integer(4),[Integer(2), Integer(1)])
['F', 4, [1, 2]]
>>> QuiverMutationType('G', Integer(2),[Integer(3),Integer(3)])
['G', 2, [3, 3]]
```

Mutation finite types:

Rank 2 cases:

```
sage: QuiverMutationType('R2',(1, 1))
['A', 2]
sage: QuiverMutationType('R2',(1, 2))
['B', 2]
sage: QuiverMutationType('R2',(1,3))
['G', 2]
sage: QuiverMutationType('R2',(1,4))
['BC', 1, 1]
sage: QuiverMutationType('R2',(1,5))
['R2', [1, 5]]
sage: QuiverMutationType('R2',(2, 2))
['A', [1, 1], 1]
sage: QuiverMutationType('R2',(3,5))
['R2', [3, 5]]
```

```
>>> from sage.all import *
>>> QuiverMutationType('R2',(Integer(1), Integer(1)))
['A', 2]
>>> QuiverMutationType('R2',(Integer(1), Integer(2)))
['B', 2]
>>> QuiverMutationType('R2',(Integer(1),Integer(3)))
['G', 2]
>>> QuiverMutationType('R2',(Integer(1),Integer(4)))
['BC', 1, 1]
>>> QuiverMutationType('R2',(Integer(1),Integer(5)))
```

(continues on next page)

(continued from previous page)

```
['R2', [1, 5]]
>>> QuiverMutationType('R2',(Integer(2), Integer(2)))
['A', [1, 1], 1]
>>> QuiverMutationType('R2',(Integer(3),Integer(5)))
['R2', [3, 5]]
```

Exceptional Derksen-Owen quivers:

```
sage: QuiverMutationType('X',6)
['X', 6]
```

```
>>> from sage.all import *
>>> QuiverMutationType('X',Integer(6))
['X', 6]
```

(Mainly) mutation infinite types:

Infinite type E:

```
sage: QuiverMutationType('E',9)
['E', 8, 1]
sage: QuiverMutationType('E', 10)
['E', 10]
sage: QuiverMutationType('E', 12)
['E', 12]

sage: QuiverMutationType('AE',(2,3))
['AE', [2, 3]]
sage: QuiverMutationType('BE',5)
['BE', 5]
sage: QuiverMutationType('CE',5)
['CE', 5]
sage: QuiverMutationType('DE',6)
['DE', 6]
```

```
>>> from sage.all import *
>>> QuiverMutationType('E',Integer(9))
['E', 8, 1]
>>> QuiverMutationType('E', Integer(10))
['E', 10]
>>> QuiverMutationType('E', Integer(12))
['E', 12]

>>> QuiverMutationType('AE',(Integer(2),Integer(3)))
['AE', [2, 3]]
>>> QuiverMutationType('BE',Integer(5))
['BE', 5]
>>> QuiverMutationType('CE',Integer(5))
['CE', 5]
>>> QuiverMutationType('DE',Integer(6))
['DE', 6]
```

Grassmannian types:

```
sage: QuiverMutationType('GR',(2,4))
['A', 1]
sage: QuiverMutationType('GR',(2,6))
['A', 3]
sage: QuiverMutationType('GR',(3,6))
['D', 4]
sage: QuiverMutationType('GR',(3,7))
['E', 6]
sage: QuiverMutationType('GR',(3,8))
['E', 8]
sage: QuiverMutationType('GR',(3, 10))
['GR', [3, 10]]
```

```
>>> from sage.all import *
>>> QuiverMutationType('GR',(Integer(2),Integer(4)))
['A', 1]
>>> QuiverMutationType('GR',(Integer(2),Integer(6)))
['A', 3]
>>> QuiverMutationType('GR',(Integer(3),Integer(6)))
['D', 4]
>>> QuiverMutationType('GR',(Integer(3),Integer(7)))
['E', 6]
>>> QuiverMutationType('GR',(Integer(3),Integer(8)))
['E', 8]
>>> QuiverMutationType('GR',(Integer(3), Integer(10)))
['GR', [3, 10]]
```

Triangular types:

```
sage: QuiverMutationType('TR', 2)
['A', 3]
sage: QuiverMutationType('TR',3)
['D', 6]
sage: QuiverMutationType('TR',4)
['E', 8, [1, 1]]
sage: QuiverMutationType('TR',5)
['TR', 5]
```

```
>>> from sage.all import *
>>> QuiverMutationType('TR', Integer(2))
['A', 3]
>>> QuiverMutationType('TR',Integer(3))
['D', 6]
>>> QuiverMutationType('TR',Integer(4))
['E', 8, [1, 1]]
>>> QuiverMutationType('TR',Integer(5))
['TR', 5]
```

T types:

```
sage: QuiverMutationType('T',(1, 1, 1))
['A', 1]
sage: QuiverMutationType('T',(1, 1,4))
```

(continues on next page)

(continued from previous page)

```
['A', 4]
sage: QuiverMutationType('T',(1,4,4))
['A', 7]
sage: QuiverMutationType('T',(2, 2, 2))
['D', 4]
sage: QuiverMutationType('T',(2, 2,4))
['D', 6]
sage: QuiverMutationType('T',(2,3,3))
['E', 6]
sage: QuiverMutationType('T',(2,3,4))
['E', 7]
sage: QuiverMutationType('T',(2,3,5))
['E', 8]
sage: QuiverMutationType('T',(2,3,6))
['E', 8, 1]
sage: QuiverMutationType('T',(2,3,7))
['E', 10]
sage: QuiverMutationType('T',(3,3,3))
['E', 6, 1]
sage: QuiverMutationType('T',(3,3,4))
['T', [3, 3, 4]]
```

```
>>> from sage.all import *
>>> QuiverMutationType('T',(Integer(1), Integer(1), Integer(1)))
['A', 1]
>>> QuiverMutationType('T',(Integer(1), Integer(1),Integer(4)))
['A', 4]
>>> QuiverMutationType('T',(Integer(1),Integer(4),Integer(4)))
['A', 7]
>>> QuiverMutationType('T',(Integer(2), Integer(2), Integer(2)))
['D', 4]
>>> QuiverMutationType('T',(Integer(2), Integer(2),Integer(4)))
['D', 6]
>>> QuiverMutationType('T',(Integer(2),Integer(3),Integer(3)))
['E', 6]
>>> QuiverMutationType('T',(Integer(2),Integer(3),Integer(4)))
['E', 7]
>>> QuiverMutationType('T',(Integer(2),Integer(3),Integer(5)))
['E', 8]
>>> QuiverMutationType('T',(Integer(2),Integer(3),Integer(6)))
['E', 8, 1]
>>> QuiverMutationType('T',(Integer(2),Integer(3),Integer(7)))
['E', 10]
>>> QuiverMutationType('T',(Integer(3),Integer(3),Integer(3)))
['E', 6, 1]
>>> QuiverMutationType('T',(Integer(3),Integer(3),Integer(4)))
['T', [3, 3, 4]]
```

Reducible types:

```
sage: QuiverMutationType(['A',3],['B',4])
[ ['A', 3], ['B', 4] ]
```

```
>>> from sage.all import *
>>> QuiverMutationType(['A',Integer(3)],['B',Integer(4)])
[ ['A', 3], ['B', 4] ]
```

class sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationTypeFactory**

Bases: SageObject

samples (*finite=None*, *affine=None*, *elliptic=None*, *mutation_finite=None*)

Return a sample of the available quiver mutations types.

INPUT:

- finite
- affine
- elliptic
- mutation_finite

All four input keywords default values are None. If set to True or False, only these samples are returned.

EXAMPLES:

```
sage: QuiverMutationType.samples()
[['A', 1], ['A', 5], ['B', 2], ['B', 5], ['C', 3],
 ['C', 5], [ ['A', 1], ['A', 1] ], ['D', 5], ['E', 6],
 ['E', 7], ['E', 8], ['F', 4], ['G', 2],
 ['A', [1, 1], 1], ['A', [4, 5], 1], ['D', 4, 1],
 ['BB', 5, 1], ['E', 6, [1, 1]], ['E', 7, [1, 1]],
 ['R2', [1, 5]], ['R2', [3, 5]], ['E', 10], ['BE', 5],
 ['GR', [3, 10]], ['T', [3, 3, 4]]]

sage: QuiverMutationType.samples(finite=True)
[['A', 1], ['A', 5], ['B', 2], ['B', 5], ['C', 3],
 ['C', 5], [ ['A', 1], ['A', 1] ], ['D', 5], ['E', 6],
 ['E', 7], ['E', 8], ['F', 4], ['G', 2]]

sage: QuiverMutationType.samples(affine=True)
[['A', [1, 1], 1], ['A', [4, 5], 1], ['D', 4, 1], ['BB', 5, 1]]

sage: QuiverMutationType.samples(elliptic=True)
[['E', 6, [1, 1]], ['E', 7, [1, 1]]]

sage: QuiverMutationType.samples(mutation_finite=False)
[['R2', [1, 5]], ['R2', [3, 5]], ['E', 10], ['BE', 5],
 ['GR', [3, 10]], ['T', [3, 3, 4]]]
```

```
>>> from sage.all import *
>>> QuiverMutationType.samples()
[['A', 1], ['A', 5], ['B', 2], ['B', 5], ['C', 3],
 ['C', 5], [ ['A', 1], ['A', 1] ], ['D', 5], ['E', 6],
 ['E', 7], ['E', 8], ['F', 4], ['G', 2],
 ['A', [1, 1], 1], ['A', [4, 5], 1], ['D', 4, 1],
 ['BB', 5, 1], ['E', 6, [1, 1]], ['E', 7, [1, 1]],
```

(continues on next page)

(continued from previous page)

```
 ['R2', [1, 5]], ['R2', [3, 5]], ['E', 10], ['BE', 5],
 ['GR', [3, 10]], ['T', [3, 3, 4]]]

>>> QuiverMutationType.samples(finite=True)
[['A', 1], ['A', 5], ['B', 2], ['B', 5], ['C', 3],
 ['C', 5], [ ['A', 1], ['A', 1] ], ['D', 5], ['E', 6],
 ['E', 7], ['E', 8], ['F', 4], ['G', 2]]

>>> QuiverMutationType.samples(affine=True)
[['A', [1, 1], 1], ['A', [4, 5], 1], ['D', 4, 1], ['BB', 5, 1]]

>>> QuiverMutationType.samples(elliptic=True)
[['E', 6, [1, 1]], ['E', 7, [1, 1]]]

>>> QuiverMutationType.samples(mutation_finite=False)
[['R2', [1, 5]], ['R2', [3, 5]], ['E', 10], ['BE', 5],
 ['GR', [3, 10]], ['T', [3, 3, 4]]]
```

class sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationType_Irreducible**(*letter*, *rank*, *twist=N*

Bases: *QuiverMutationType_abstract*

The mutation type for a cluster algebra or a quiver. Should not be called directly, but through *QuiverMutationType*.

class_size()

If it is known, the size of the mutation class of all quivers which are mutation equivalent to the standard quiver of `self` (up to isomorphism) is returned.

Otherwise, `NotImplemented` is returned.

Formula for finite type A is taken from Torkildsen - Counting cluster-tilted algebras of type A_n. Formulas for affine type A and finite type D are taken from Bastian, Prellberg, Rubey, Stump - Counting the number of elements in the mutation classes of $\widetilde{A}_n$ quivers. Formulas for finite and affine types B and C are proven but not yet published. Conjectural formulas for several other non-simply-laced affine types are implemented. Exceptional Types (finite, affine, and elliptic) E, F, G, and X are hardcoded.

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.class_size()
19

sage: mut_type = QuiverMutationType( ['A',[10,3], 1] ); mut_type
['A', [3, 10], 1]
sage: mut_type.class_size()
142120

sage: mut_type = QuiverMutationType( ['B',6] ); mut_type
['B', 6]
sage: mut_type.class_size()
```

(continues on next page)

(continued from previous page)

```
132

sage: mut_type = QuiverMutationType( ['BD',6, 1] ); mut_type
['BD', 6, 1]
sage: mut_type.class_size()
Warning: This method uses a formula which has not been proved correct.
504
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType( ['A',Integer(5)] ); mut_type
['A', 5]
>>> mut_type.class_size()
19

>>> mut_type = QuiverMutationType( ['A',[Integer(10),Integer(3)], Integer(1)]␣
↪); mut_type
['A', [3, 10], 1]
>>> mut_type.class_size()
142120

>>> mut_type = QuiverMutationType( ['B',Integer(6)] ); mut_type
['B', 6]
>>> mut_type.class_size()
132

>>> mut_type = QuiverMutationType( ['BD',Integer(6), Integer(1)] ); mut_type
['BD', 6, 1]
>>> mut_type.class_size()
Warning: This method uses a formula which has not been proved correct.
504
```

Check that Issue #14048 is fixed:

```
sage: mut_type = QuiverMutationType( ['F',4,(2, 1)] )
sage: mut_type.class_size()
90
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType( ['F',Integer(4),(Integer(2), Integer(1))] )
>>> mut_type.class_size()
90
```

dual()

Return the *QuiverMutationType* which is dual to `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType('A',5); mut_type
['A', 5]
sage: mut_type.dual()
['A', 5]
```

(continues on next page)

(continued from previous page)

```
sage: mut_type = QuiverMutationType('B',5); mut_type
['B', 5]
sage: mut_type.dual()
['C', 5]
sage: mut_type.dual().dual()
['B', 5]
sage: mut_type.dual().dual() == mut_type
True
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType('A',Integer(5)); mut_type
['A', 5]
>>> mut_type.dual()
['A', 5]

>>> mut_type = QuiverMutationType('B',Integer(5)); mut_type
['B', 5]
>>> mut_type.dual()
['C', 5]
>>> mut_type.dual().dual()
['B', 5]
>>> mut_type.dual().dual() == mut_type
True
```

irreducible_components()

Return a list of all irreducible components of `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType('A',3); mut_type
['A', 3]
sage: mut_type.irreducible_components()
(['A', 3],)
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType('A',Integer(3)); mut_type
['A', 3]
>>> mut_type.irreducible_components()
(['A', 3],)
```

class sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationType_Reducible**(**args*)

Bases: `QuiverMutationType_abstract`

The mutation type for a cluster algebra or a quiver. Should not be called directly, but through `QuiverMutationType`. Inherits from `QuiverMutationType_abstract`.

class_size()

If it is known, the size of the mutation class of all quivers which are mutation equivalent to the standard quiver of `self` (up to isomorphism) is returned.

Otherwise, `NotImplemented` is returned.

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.class_size()
20

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6])
sage: mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.class_size()
100
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)]); mut_type
[ ['A', 3], ['B', 3] ]
>>> mut_type.class_size()
20

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)],['X',
↪Integer(6)])
>>> mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
>>> mut_type.class_size()
100
```

dual()

Return the *QuiverMutationType* which is dual to `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',5],['B',6],['C',5],['D',4]); mut_type
[ ['A', 5], ['B', 6], ['C', 5], ['D', 4] ]
sage: mut_type.dual()
[ ['A', 5], ['C', 6], ['B', 5], ['D', 4] ]
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType(['A',Integer(5)],['B',Integer(6)],['C',
↪Integer(5)],['D',Integer(4)]); mut_type
[ ['A', 5], ['B', 6], ['C', 5], ['D', 4] ]
>>> mut_type.dual()
[ ['A', 5], ['C', 6], ['B', 5], ['D', 4] ]
```

irreducible_components()

Return a list of all irreducible components of `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType('A',3); mut_type
['A', 3]
sage: mut_type.irreducible_components()
(['A', 3],)

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
```

(continues on next page)

(continued from previous page)

```
sage: mut_type.irreducible_components()
(['A', 3], ['B', 3])

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6])
sage: mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.irreducible_components()
(['A', 3], ['B', 3], ['X', 6])
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType('A',Integer(3)); mut_type
['A', 3]
>>> mut_type.irreducible_components()
(['A', 3],)

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)]); mut_type
[ ['A', 3], ['B', 3] ]
>>> mut_type.irreducible_components()
(['A', 3], ['B', 3])

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)],['X',
↪Integer(6)])
>>> mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
>>> mut_type.irreducible_components()
(['A', 3], ['B', 3], ['X', 6])
```

class sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationType_abstract**

Bases: UniqueRepresentation, SageObject

EXAMPLES:

```
sage: mut_type1 = QuiverMutationType('A',5)
sage: mut_type2 = QuiverMutationType('A',5)
sage: mut_type3 = QuiverMutationType('A',6)
sage: mut_type1 == mut_type2
True
sage: mut_type1 == mut_type3
False
```

```
>>> from sage.all import *
>>> mut_type1 = QuiverMutationType('A',Integer(5))
>>> mut_type2 = QuiverMutationType('A',Integer(5))
>>> mut_type3 = QuiverMutationType('A',Integer(6))
>>> mut_type1 == mut_type2
True
>>> mut_type1 == mut_type3
False
```

b_matrix()

Return the B-matrix of the standard quiver of `self`.

The conventions for B-matrices agree with Fomin-Zelevinsky (up to a reordering of the simple roots).

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',5]); mut_type
['A', 5]
sage: mut_type.b_matrix()                                                    #␣
↪needs sage.modules
[ 0  1  0  0  0]
[-1  0 -1  0  0]
[ 0  1  0  1  0]
[ 0  0 -1  0 -1]
[ 0  0  0  1  0]

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.b_matrix()                                                    #␣
↪needs sage.modules
[ 0  1  0  0  0  0]
[-1  0 -1  0  0  0]
[ 0  1  0  0  0  0]
[ 0  0  0  0  1  0]
[ 0  0  0 -1  0 -1]
[ 0  0  0  0  2  0]
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType(['A',Integer(5)]); mut_type
['A', 5]
>>> mut_type.b_matrix()                                                    #␣
↪needs sage.modules
[ 0  1  0  0  0]
[-1  0 -1  0  0]
[ 0  1  0  1  0]
[ 0  0 -1  0 -1]
[ 0  0  0  1  0]

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)]); mut_type
[ ['A', 3], ['B', 3] ]
>>> mut_type.b_matrix()                                                    #␣
↪needs sage.modules
[ 0  1  0  0  0  0]
[-1  0 -1  0  0  0]
[ 0  1  0  0  0  0]
[ 0  0  0  0  1  0]
[ 0  0  0 -1  0 -1]
[ 0  0  0  0  2  0]
```

cartan_matrix()

Return the Cartan matrix of `self`.

Note that (up to a reordering of the simple roots) the convention for the definition of Cartan matrix, used here and elsewhere in Sage, agrees with the conventions of Kac, Fulton-Harris, and Fomin-Zelevinsky, but disagrees with the convention of Bourbaki. The (i, j) entry is $2(\alpha_i, \alpha_j)/(\alpha_i, \alpha_i)$.

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',5]); mut_type
['A', 5]
sage: mut_type.cartan_matrix()                                          #␣
↪needs sage.modules
[ 2 -1  0  0  0]
[-1  2 -1  0  0]
[ 0 -1  2 -1  0]
[ 0  0 -1  2 -1]
[ 0  0  0 -1  2]

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.cartan_matrix()                                          #␣
↪needs sage.modules
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2  0  0  0]
[ 0  0  0  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -2  2]
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType(['A',Integer(5)]); mut_type
['A', 5]
>>> mut_type.cartan_matrix()                                          #␣
↪needs sage.modules
[ 2 -1  0  0  0]
[-1  2 -1  0  0]
[ 0 -1  2 -1  0]
[ 0  0 -1  2 -1]
[ 0  0  0 -1  2]

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)]); mut_type
[ ['A', 3], ['B', 3] ]
>>> mut_type.cartan_matrix()                                          #␣
↪needs sage.modules
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2  0  0  0]
[ 0  0  0  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -2  2]
```

is_affine()

Return `True` if `self` is of affine type.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A', 2])
sage: mt.is_affine()
False

sage: mt = QuiverMutationType(['A',[4, 2], 1])
```

(continues on next page)

(continued from previous page)

```
sage: mt.is_affine()
True
```

```
>>> from sage.all import *
>>> mt = QuiverMutationType(['A', Integer(2)])
>>> mt.is_affine()
False

>>> mt = QuiverMutationType(['A',[Integer(4), Integer(2)], Integer(1)])
>>> mt.is_affine()
True
```

is_elliptic()

Return True if self is of elliptic type.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A', 2])
sage: mt.is_elliptic()
False

sage: mt = QuiverMutationType(['E',6,[1, 1]])
sage: mt.is_elliptic()
True
```

```
>>> from sage.all import *
>>> mt = QuiverMutationType(['A', Integer(2)])
>>> mt.is_elliptic()
False

>>> mt = QuiverMutationType(['E',Integer(6),[Integer(1), Integer(1)]])
>>> mt.is_elliptic()
True
```

is_finite()

Return True if self is of finite type.

This means that the cluster algebra associated to self has only a finite number of cluster variables.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A', 2])
sage: mt.is_finite()
True

sage: mt = QuiverMutationType(['A',[4, 2], 1])
sage: mt.is_finite()
False
```

```
>>> from sage.all import *
>>> mt = QuiverMutationType(['A', Integer(2)])
>>> mt.is_finite()
True
```

(continues on next page)

(continued from previous page)

```
>>> mt = QuiverMutationType(['A',[Integer(4), Integer(2)], Integer(1)])
>>> mt.is_finite()
False
```

is_irreducible()

Return `True` if `self` is irreducible.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A', 2])
sage: mt.is_irreducible()
True
```

```
>>> from sage.all import *
>>> mt = QuiverMutationType(['A', Integer(2)])
>>> mt.is_irreducible()
True
```

is_mutation_finite()

Return `True` if `self` is of finite mutation type.

This means that its mutation class has only finitely many different B-matrices.

EXAMPLES:

```
sage: mt = QuiverMutationType(['D',5, 1])
sage: mt.is_mutation_finite()
True
```

```
>>> from sage.all import *
>>> mt = QuiverMutationType(['D',Integer(5), Integer(1)])
>>> mt.is_mutation_finite()
True
```

is_simply_laced()

Return `True` if `self` is simply laced.

This means that the only arrows that appear in the quiver of `self` are single unlabelled arrows.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A', 2])
sage: mt.is_simply_laced()
True

sage: mt = QuiverMutationType(['B', 2])
sage: mt.is_simply_laced()
False

sage: mt = QuiverMutationType(['A',(1, 1), 1])
sage: mt.is_simply_laced()
False
```

```
>>> from sage.all import *
>>> mt = QuiverMutationType(['A', Integer(2)])
>>> mt.is_simply_laced()
True

>>> mt = QuiverMutationType(['B', Integer(2)])
>>> mt.is_simply_laced()
False

>>> mt = QuiverMutationType(['A',(Integer(1), Integer(1)), Integer(1)])
>>> mt.is_simply_laced()
False
```

is_skew_symmetric()

Return `True` if the B-matrix of `self` is skew-symmetric.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A', 2])
sage: mt.is_skew_symmetric()
True

sage: mt = QuiverMutationType(['B', 2])
sage: mt.is_skew_symmetric()
False

sage: mt = QuiverMutationType(['A',(1, 1), 1])
sage: mt.is_skew_symmetric()
True
```

```
>>> from sage.all import *
>>> mt = QuiverMutationType(['A', Integer(2)])
>>> mt.is_skew_symmetric()
True

>>> mt = QuiverMutationType(['B', Integer(2)])
>>> mt.is_skew_symmetric()
False

>>> mt = QuiverMutationType(['A',(Integer(1), Integer(1)), Integer(1)])
>>> mt.is_skew_symmetric()
True
```

letter()

Return the classification letter of `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.letter()
'A'
```

(continues on next page)

(continued from previous page)

```
sage: mut_type = QuiverMutationType( ['BC',5, 1] ); mut_type
['BC', 5, 1]
sage: mut_type.letter()
'BC'

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.letter()
'A x B'

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.letter()
'A x B x X'
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType( ['A',Integer(5)] ); mut_type
['A', 5]
>>> mut_type.letter()
'A'

>>> mut_type = QuiverMutationType( ['BC',Integer(5), Integer(1)] ); mut_type
['BC', 5, 1]
>>> mut_type.letter()
'BC'

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)]); mut_type
[ ['A', 3], ['B', 3] ]
>>> mut_type.letter()
'A x B'

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)],['X',
↪Integer(6)]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
>>> mut_type.letter()
'A x B x X'
```

plot(*circular=False*, *directed=True*)

Return the plot of the underlying graph or digraph of `self`.

INPUT:

- `circular` – boolean (default: `False`); if `True`, the circular plot is chosen, otherwise >>spring<< is used
- `directed` – boolean (default: `True`); if `True`, the directed version is shown, otherwise the undirected

EXAMPLES:

```
sage: QMT = QuiverMutationType(['A',5])
sage: pl = QMT.plot()                                                  #␣
↪needs sage.plot sage.symbolic
sage: pl = QMT.plot(circular=True)                                     #␣
↪needs sage.plot sage.symbolic
```

```
>>> from sage.all import *
>>> QMT = QuiverMutationType(['A',Integer(5)])
>>> pl = QMT.plot()                                                          #␣
↪needs sage.plot sage.symbolic
>>> pl = QMT.plot(circular=True)                                             #␣
↪needs sage.plot sage.symbolic
```

properties()

Print a scheme of all properties of `self`.

Most properties have natural definitions for either irreducible or reducible types. `affine` and `elliptic` are only defined for irreducible types.

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',3]); mut_type
['A', 3]
sage: mut_type.properties()
['A', 3] has rank 3 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      True
    - skew-symmetric:    True
    - finite:            True
    - affine:            False
    - elliptic:          False

sage: mut_type = QuiverMutationType(['B',3]); mut_type
['B', 3]
sage: mut_type.properties()
['B', 3] has rank 3 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      False
    - skew-symmetric:    False
    - finite:            True
    - affine:            False
    - elliptic:          False

sage: mut_type = QuiverMutationType(['B',3, 1]); mut_type
['BD', 3, 1]
sage: mut_type.properties()
['BD', 3, 1] has rank 4 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      False
    - skew-symmetric:    False
    - finite:            False
    - affine:            True
    - elliptic:          False

sage: mut_type = QuiverMutationType(['E',6,[1, 1]]); mut_type
['E', 6, [1, 1]]
sage: mut_type.properties()
```

(continues on next page)

(continued from previous page)

```
['E', 6, [1, 1]] has rank 8 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      False
    - skew-symmetric:    True
    - finite:            False
    - affine:            False
    - elliptic:          True

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.properties()
[ ['A', 3], ['B', 3] ] has rank 6 and the following properties:
    - irreducible:       False
    - mutation finite:   True
    - simply-laced:      False
    - skew-symmetric:    False
    - finite:            True

sage: mut_type = QuiverMutationType('GR',[4,9]); mut_type
['GR', [4, 9]]
sage: mut_type.properties()
['GR', [4, 9]] has rank 12 and the following properties:
    - irreducible:       True
    - mutation finite:   False
    - simply-laced:      True
    - skew-symmetric:    True
    - finite:            False
    - affine:            False
    - elliptic:          False
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType(['A',Integer(3)]); mut_type
['A', 3]
>>> mut_type.properties()
['A', 3] has rank 3 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      True
    - skew-symmetric:    True
    - finite:            True
    - affine:            False
    - elliptic:          False

>>> mut_type = QuiverMutationType(['B',Integer(3)]); mut_type
['B', 3]
>>> mut_type.properties()
['B', 3] has rank 3 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      False
    - skew-symmetric:    False
```

(continues on next page)

(continued from previous page)

```
    - finite:             True
    - affine:             False
    - elliptic:           False

>>> mut_type = QuiverMutationType(['B',Integer(3), Integer(1)]); mut_type
['BD', 3, 1]
>>> mut_type.properties()
['BD', 3, 1] has rank 4 and the following properties:
    - irreducible:        True
    - mutation finite:    True
    - simply-laced:       False
    - skew-symmetric:     False
    - finite:             False
    - affine:             True
    - elliptic:           False

>>> mut_type = QuiverMutationType(['E',Integer(6),[Integer(1), Integer(1)]]);␣
↪mut_type
['E', 6, [1, 1]]
>>> mut_type.properties()
['E', 6, [1, 1]] has rank 8 and the following properties:
    - irreducible:        True
    - mutation finite:    True
    - simply-laced:       False
    - skew-symmetric:     True
    - finite:             False
    - affine:             False
    - elliptic:           True

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)]); mut_type
[ ['A', 3], ['B', 3] ]
>>> mut_type.properties()
[ ['A', 3], ['B', 3] ] has rank 6 and the following properties:
    - irreducible:        False
    - mutation finite:    True
    - simply-laced:       False
    - skew-symmetric:     False
    - finite:             True

>>> mut_type = QuiverMutationType('GR',[Integer(4),Integer(9)]); mut_type
['GR', [4, 9]]
>>> mut_type.properties()
['GR', [4, 9]] has rank 12 and the following properties:
    - irreducible:        True
    - mutation finite:    False
    - simply-laced:       True
    - skew-symmetric:     True
    - finite:             False
    - affine:             False
    - elliptic:           False
```

rank()

Return the rank in the standard quiver of `self`.

The rank is the number of vertices.

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.rank()
5

sage: mut_type = QuiverMutationType( ['A',[4,5], 1] ); mut_type
['A', [4, 5], 1]
sage: mut_type.rank()
9

sage: mut_type = QuiverMutationType( ['BC',5, 1] ); mut_type
['BC', 5, 1]
sage: mut_type.rank()
6

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.rank()
6

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.rank()
12
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType( ['A',Integer(5)] ); mut_type
['A', 5]
>>> mut_type.rank()
5

>>> mut_type = QuiverMutationType( ['A',[Integer(4),Integer(5)], Integer(1)]␣
→); mut_type
['A', [4, 5], 1]
>>> mut_type.rank()
9

>>> mut_type = QuiverMutationType( ['BC',Integer(5), Integer(1)] ); mut_type
['BC', 5, 1]
>>> mut_type.rank()
6

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)]); mut_type
[ ['A', 3], ['B', 3] ]
>>> mut_type.rank()
6

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)],['X',
```

(continues on next page)

(continued from previous page)

```
↪Integer(6)]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
>>> mut_type.rank()
12
```

show(*circular=False*, *directed=True*)

Show the plot of the underlying digraph of `self`.

INPUT:

- `circular` – boolean (default: `False`); if `True`, the circular plot is chosen, otherwise >>spring<< is used
- `directed` – boolean (default: `True`); if `True`, the directed version is shown, otherwise the undirected

standard_quiver()

Return the standard quiver of `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.standard_quiver()                                          #␣
↪needs sage.modules
Quiver on 5 vertices of type ['A', 5]

sage: mut_type = QuiverMutationType( ['A',[5,3], 1] ); mut_type
['A', [3, 5], 1]
sage: mut_type.standard_quiver()                                          #␣
↪needs sage.modules
Quiver on 8 vertices of type ['A', [3, 5], 1]

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.standard_quiver()                                          #␣
↪needs sage.modules
Quiver on 6 vertices of type [ ['A', 3], ['B', 3] ]

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.standard_quiver()                                          #␣
↪needs sage.modules
Quiver on 12 vertices of type [ ['A', 3], ['B', 3], ['X', 6] ]
```

```
>>> from sage.all import *
>>> mut_type = QuiverMutationType( ['A',Integer(5)] ); mut_type
['A', 5]
>>> mut_type.standard_quiver()                                          #␣
↪needs sage.modules
Quiver on 5 vertices of type ['A', 5]

>>> mut_type = QuiverMutationType( ['A',[Integer(5),Integer(3)], Integer(1)]␣
↪); mut_type
['A', [3, 5], 1]
```

(continues on next page)

(continued from previous page)

```
>>> mut_type.standard_quiver()                                              #␣
↪needs sage.modules
Quiver on 8 vertices of type ['A', [3, 5], 1]

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)]); mut_type
[ ['A', 3], ['B', 3] ]
>>> mut_type.standard_quiver()                                              #␣
↪needs sage.modules
Quiver on 6 vertices of type [ ['A', 3], ['B', 3] ]

>>> mut_type = QuiverMutationType(['A',Integer(3)],['B',Integer(3)],['X',
↪Integer(6)]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
>>> mut_type.standard_quiver()                                              #␣
↪needs sage.modules
Quiver on 12 vertices of type [ ['A', 3], ['B', 3], ['X', 6] ]
```

`sage.combinat.cluster_algebra_quiver.quiver_mutation_type.`**`save_quiver_data`**`(`*n*, *up_to=True*, *types='ClassicalExceptional'*, *verbose=True*`)`

Save mutation classes of certain quivers of ranks up to and equal to n or equal to n to `DOT_SAGE/cluster_algebra_quiver/mutation_classes_n.dig6`.

This data will then be used to determine quiver mutation types.

INPUT:

- `n` – the rank (or the upper limit on the rank) of the mutation classes that are being saved
- `up_to` – (default: `True`) if `True`, saves data for ranks smaller than or equal to n; if `False`, saves data for rank exactly n
- `types` – (default: `'ClassicalExceptional'`) if all, saves data for both exceptional mutation-finite quivers and for classical quiver; the input 'Exceptional' or 'Classical' is also allowed to save only part of this data

5.1.23 Cluster complex (or generalized dual associahedron)

EXAMPLES:

A first example of a cluster complex:

```
sage: C = ClusterComplex(['A', 2]); C
Cluster complex of type ['A', 2] with 5 vertices and 5 facets
```

```
>>> from sage.all import *
>>> C = ClusterComplex(['A', Integer(2)]); C
Cluster complex of type ['A', 2] with 5 vertices and 5 facets
```

Its vertices, facets, and minimal non-faces:

```
sage: C.vertices()
(0, 1, 2, 3, 4)

sage: C.facets()
```

(continues on next page)

(continued from previous page)

```
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

sage: for F in C.facets(): F.cluster()
[(-1, 0), (0, -1)]
[(-1, 0), (0, 1)]
[(0, -1), (1, 0)]
[(1, 0), (1, 1)]
[(1, 1), (0, 1)]

sage: C.minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

```
>>> from sage.all import *
>>> C.vertices()
(0, 1, 2, 3, 4)

>>> C.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

>>> for F in C.facets(): F.cluster()
[(-1, 0), (0, -1)]
[(-1, 0), (0, 1)]
[(0, -1), (1, 0)]
[(1, 0), (1, 1)]
[(1, 1), (0, 1)]

>>> C.minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

We can do everything we can do on simplicial complexes, e.g. computing its homology:

```
sage: C.homology()
{0: 0, 1: Z}
```

```
>>> from sage.all import *
>>> C.homology()
{0: 0, 1: Z}
```

AUTHORS:

- Christian Stump (2011) Initial version

class sage.combinat.cluster_complex.**ClusterComplex**(*W*, *k*, *coxeter_element*, *algorithm*)

Bases: *SubwordComplex*

A cluster complex (or generalized dual associahedron).

The cluster complex (or generalized dual associahedron) is a simplicial complex constructed from a cluster algebra. Its vertices are the cluster variables and its facets are the clusters, i.e., maximal subsets of compatible cluster variables.

The cluster complex of type A_n is the simplicial complex with vertices being (proper) diagonals in a convex $(n+3)$-gon and with facets being triangulations.

The implementation of the cluster complex depends on its connection to subword complexes, see [CLS2014]. Let

c be a Coxeter element with reduced word $\mathbf{c}$ in a finite Coxeter group W, and let $\mathbf{w}_\circ$ be the c-sorting word for the longest element $w_\circ \in W$.

The `multi-cluster complex` $\Delta(W, k)$ has vertices in one-to-one correspondence with letters in the word $Q = \mathbf{c^k w_\circ}$ and with facets being complements in Q of reduced expressions for $w_\circ$.

For $k = 1$, the multi-cluster complex is isomorphic to the cluster complex as defined above.

EXAMPLES:

A first example of a cluster complex:

```
sage: C = ClusterComplex(['A', 2]); C
Cluster complex of type ['A', 2] with 5 vertices and 5 facets
```

```
>>> from sage.all import *
>>> C = ClusterComplex(['A', Integer(2)]); C
Cluster complex of type ['A', 2] with 5 vertices and 5 facets
```

Its vertices, facets, and minimal non-faces:

```
sage: C.vertices()
(0, 1, 2, 3, 4)

sage: C.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

sage: C.minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

```
>>> from sage.all import *
>>> C.vertices()
(0, 1, 2, 3, 4)

>>> C.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

>>> C.minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

We can do everything we can do on simplicial complexes, e.g. computing its homology:

```
sage: C.homology()
{0: 0, 1: Z}
```

```
>>> from sage.all import *
>>> C.homology()
{0: 0, 1: Z}
```

We can also create a multi-cluster complex:

```
sage: ClusterComplex(['A', 2], k=2)
Multi-cluster complex of type ['A', 2] with 7 vertices and 14 facets
```

```
>>> from sage.all import *
>>> ClusterComplex(['A', Integer(2)], k=Integer(2))
Multi-cluster complex of type ['A', 2] with 7 vertices and 14 facets
```

REFERENCES:

- [CLS2014]

Element

alias of *ClusterComplexFacet*

cyclic_rotation()

Return the operation on the facets of `self` obtained by the cyclic rotation as defined in [CLS2014].

EXAMPLES:

```
sage: ClusterComplex(['A', 2]).cyclic_rotation()
<function ...act at ...>
```

```
>>> from sage.all import *
>>> ClusterComplex(['A', Integer(2)]).cyclic_rotation()
<function ...act at ...>
```

k()

Return the index k of `self`.

EXAMPLES:

```
sage: ClusterComplex(['A', 2]).k()
1
```

```
>>> from sage.all import *
>>> ClusterComplex(['A', Integer(2)]).k()
1
```

minimal_nonfaces()

Return the minimal non-faces of `self`.

EXAMPLES:

```
sage: ClusterComplex(['A', 2]).minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

```
>>> from sage.all import *
>>> ClusterComplex(['A', Integer(2)]).minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

class sage.combinat.cluster_complex.**ClusterComplexFacet**(*parent*, *positions*, *facet_test=True*)

Bases: *SubwordComplexFacet*

A cluster (i.e., a facet) of a cluster complex.

cluster()

Return this cluster as a set of almost positive roots.

EXAMPLES:

```
sage: C = ClusterComplex(['A', 2])
sage: F = C((0, 1))
sage: F.cluster()
[(-1, 0), (0, -1)]
```

```
>>> from sage.all import *
>>> C = ClusterComplex(['A', Integer(2)])
>>> F = C((Integer(0), Integer(1)))
>>> F.cluster()
[(-1, 0), (0, -1)]
```

product_of_upper_cluster()

Return the product of the upper cluster in reversed order.

EXAMPLES:

```
sage: C = ClusterComplex(['A', 2])
sage: for F in C: F.product_of_upper_cluster().reduced_word()
[]
[2]
[1]
[1, 2]
[1, 2]
```

```
>>> from sage.all import *
>>> C = ClusterComplex(['A', Integer(2)])
>>> for F in C: F.product_of_upper_cluster().reduced_word()
[]
[2]
[1]
[1, 2]
[1, 2]
```

upper_cluster()

Return the part of the cluster that contains positive roots.

EXAMPLES:

```
sage: C = ClusterComplex(['A', 2])
sage: F = C((0, 1))
sage: F.upper_cluster()
[]
```

```
>>> from sage.all import *
>>> C = ClusterComplex(['A', Integer(2)])
>>> F = C((Integer(0), Integer(1)))
>>> F.upper_cluster()
[]
```

5.1.24 Colored Permutations

Todo

Much of the colored permutations (and element) class can be generalized to $G \wr S_n$

class sage.combinat.colored_permutations.**ColoredPermutation**(*parent*, *colors*, *perm*)

Bases: MultiplicativeGroupElement

A colored permutation.

colors()

Return the colors of self.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: x = s1*s2*t
sage: x.colors()
[1, 0, 0]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> s1,s2,t = C.gens()
>>> x = s1*s2*t
>>> x.colors()
[1, 0, 0]
```

has_left_descent(*i*)

Return True if i is a left descent of self.

Let $p = ((s_1, \ldots s_n), \sigma)$ be a colored permutation. We say p has a left n-descent if $s_n > 0$. If $i < n$, then we say p has a left i-descent if either

- $s_i \neq 0, s_{i+1} = 0$ and $\sigma_i < \sigma_{i+1}$ or
- $s_i = s_{i+1}$ and $\sigma_i > \sigma_{i+1}$.

This notion of a left i-descent is done in order to recursively construct $w(p) = \sigma_i w(\sigma_i^{-1} p)$, where $w(p)$ denotes a reduced word of p.

EXAMPLES:

```
sage: C = ColoredPermutations(2, 4)
sage: s1,s2,s3,s4 = C.gens()
sage: x = s4*s1*s2*s3*s4
sage: [x.has_left_descent(i) for i in C.index_set()]
[True, False, False, True]

sage: C = ColoredPermutations(1, 5)
sage: s1,s2,s3,s4 = C.gens()
sage: x = s4*s1*s2*s3*s4
sage: [x.has_left_descent(i) for i in C.index_set()]
[True, False, False, True]
```

(continues on next page)

(continued from previous page)

```
sage: C = ColoredPermutations(3, 3)
sage: x = C([[2,1,0],[3,1,2]])
sage: [x.has_left_descent(i) for i in C.index_set()]
[False, True, False]

sage: C = ColoredPermutations(4, 4)
sage: x = C([[2,1,0,1],[3,2,4,1]])
sage: [x.has_left_descent(i) for i in C.index_set()]
[False, True, False, True]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(2), Integer(4))
>>> s1,s2,s3,s4 = C.gens()
>>> x = s4*s1*s2*s3*s4
>>> [x.has_left_descent(i) for i in C.index_set()]
[True, False, False, True]

>>> C = ColoredPermutations(Integer(1), Integer(5))
>>> s1,s2,s3,s4 = C.gens()
>>> x = s4*s1*s2*s3*s4
>>> [x.has_left_descent(i) for i in C.index_set()]
[True, False, False, True]

>>> C = ColoredPermutations(Integer(3), Integer(3))
>>> x = C([[Integer(2),Integer(1),Integer(0)],[Integer(3),Integer(1),
↪Integer(2)]])
>>> [x.has_left_descent(i) for i in C.index_set()]
[False, True, False]

>>> C = ColoredPermutations(Integer(4), Integer(4))
>>> x = C([[Integer(2),Integer(1),Integer(0),Integer(1)],[Integer(3),
↪Integer(2),Integer(4),Integer(1)]])
>>> [x.has_left_descent(i) for i in C.index_set()]
[False, True, False, True]
```

length()

Return the length of `self` in generating reflections.

This is the minimal numbers of generating reflections needed to obtain `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(3, 3)
sage: x = C([[2,1,0],[3,1,2]])
sage: x.length()
7

sage: C = ColoredPermutations(4, 4)
sage: x = C([[2,1,0,1],[3,2,4,1]])
sage: x.length()
12
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(3), Integer(3))
>>> x = C([[Integer(2),Integer(1),Integer(0)],[Integer(3),Integer(1),
↪Integer(2)]])
>>> x.length()
7

>>> C = ColoredPermutations(Integer(4), Integer(4))
>>> x = C([[Integer(2),Integer(1),Integer(0),Integer(1)],[Integer(3),
↪Integer(2),Integer(4),Integer(1)]])
>>> x.length()
12
```

one_line_form()

Return the one line form of `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: x = s1*s2*t
sage: x
[[1, 0, 0], [3, 1, 2]]
sage: x.one_line_form()
[(1, 3), (0, 1), (0, 2)]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> s1,s2,t = C.gens()
>>> x = s1*s2*t
>>> x
[[1, 0, 0], [3, 1, 2]]
>>> x.one_line_form()
[(1, 3), (0, 1), (0, 2)]
```

permutation()

Return the permutation of `self`.

This is obtained by forgetting the colors.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: x = s1*s2*t
sage: x.permutation()
[3, 1, 2]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> s1,s2,t = C.gens()
>>> x = s1*s2*t
>>> x.permutation()
[3, 1, 2]
```

reduced_word()

Return a word in the simple reflections to obtain `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(3, 3)
sage: x = C([[2,1,0],[3,1,2]])
sage: x.reduced_word()
[2, 1, 3, 2, 1, 3, 3]

sage: C = ColoredPermutations(4, 4)
sage: x = C([[2,1,0,1],[3,2,4,1]])
sage: x.reduced_word()
[2, 1, 4, 3, 2, 1, 4, 3, 2, 4, 4, 3]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(3), Integer(3))
>>> x = C([[Integer(2),Integer(1),Integer(0)],[Integer(3),Integer(1),
↪Integer(2)]])
>>> x.reduced_word()
[2, 1, 3, 2, 1, 3, 3]

>>> C = ColoredPermutations(Integer(4), Integer(4))
>>> x = C([[Integer(2),Integer(1),Integer(0),Integer(1)],[Integer(3),
↪Integer(2),Integer(4),Integer(1)]])
>>> x.reduced_word()
[2, 1, 4, 3, 2, 1, 4, 3, 2, 4, 4, 3]
```

to_matrix()

Return a matrix of `self`.

The colors are mapped to roots of unity.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: x = s1*s2*t*s2; x.one_line_form()
[(1, 2), (0, 1), (0, 3)]
sage: M = x.to_matrix(); M                                              #␣
↪needs sage.rings.number_field
[    0     1     0]
[zeta4     0     0]
[    0     0     1]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> s1,s2,t = C.gens()
>>> x = s1*s2*t*s2; x.one_line_form()
[(1, 2), (0, 1), (0, 3)]
>>> M = x.to_matrix(); M                                                #␣
↪needs sage.rings.number_field
[    0     1     0]
[zeta4     0     0]
[    0     0     1]
```

The matrix multiplication is in the *opposite* order:

```
sage: M == s2.to_matrix()*t.to_matrix()*s2.to_matrix()*s1.to_matrix()        #␣
↪needs sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> M == s2.to_matrix()*t.to_matrix()*s2.to_matrix()*s1.to_matrix()        #␣
↪needs sage.rings.number_field
True
```

class sage.combinat.colored_permutations.**ColoredPermutations**(*m*, *n*)

Bases: *ShephardToddFamilyGroup*

The group of m-colored permutations on $\{1, 2, \ldots, n\}$.

Let S_n be the symmetric group on n letters and C_m be the cyclic group of order m. The m-colored permutation group on n letters is given by $P_n^m = C_m \wr S_n$. This is also the complex reflection group $G(m, 1, n)$.

We define our multiplication by

$$((s_1, \ldots s_n), \sigma) \cdot ((t_1, \ldots, t_n), \tau) = ((s_1 t_{\sigma(1)}, \ldots, s_n t_{\sigma(n)}), \tau\sigma).$$

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3); C
4-colored permutations of size 3
sage: s1,s2,t = C.gens()
sage: (s1, s2, t)
([[0, 0, 0], [2, 1, 3]], [[0, 0, 0], [1, 3, 2]], [[0, 0, 1], [1, 2, 3]])
sage: s1*s2
[[0, 0, 0], [3, 1, 2]]
sage: s1*s2*s1 == s2*s1*s2
True
sage: t^4 == C.one()
True
sage: s2*t*s2
[[0, 1, 0], [1, 2, 3]]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3)); C
4-colored permutations of size 3
>>> s1,s2,t = C.gens()
>>> (s1, s2, t)
([[0, 0, 0], [2, 1, 3]], [[0, 0, 0], [1, 3, 2]], [[0, 0, 1], [1, 2, 3]])
>>> s1*s2
[[0, 0, 0], [3, 1, 2]]
>>> s1*s2*s1 == s2*s1*s2
True
>>> t**Integer(4) == C.one()
True
>>> s2*t*s2
[[0, 1, 0], [1, 2, 3]]
```

We can also create a colored permutation by passing an iterable consisting of tuples consisting of `(color, element)`:

```
sage: x = C([(2,1), (3,3), (3,2)]); x
[[2, 3, 3], [1, 3, 2]]
```

```
>>> from sage.all import *
>>> x = C([(Integer(2),Integer(1)), (Integer(3),Integer(3)), (Integer(3),
↪Integer(2))]); x
[[2, 3, 3], [1, 3, 2]]
```

or a list of colors and a permutation:

```
sage: C([[3,3,1], [1,3,2]])
[[3, 3, 1], [1, 3, 2]]
sage: C(([3,3,1], [1,3,2]))
[[3, 3, 1], [1, 3, 2]]
```

```
>>> from sage.all import *
>>> C([[Integer(3),Integer(3),Integer(1)], [Integer(1),Integer(3),Integer(2)]])
[[3, 3, 1], [1, 3, 2]]
>>> C(([Integer(3),Integer(3),Integer(1)], [Integer(1),Integer(3),Integer(2)]))
[[3, 3, 1], [1, 3, 2]]
```

There is also the natural lift from permutations:

```
sage: P = Permutations(3)
sage: C(P.an_element())
[[0, 0, 0], [3, 1, 2]]
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(3))
>>> C(P.an_element())
[[0, 0, 0], [3, 1, 2]]
```

A colored permutation:

```
sage: C(C.an_element()) == C.an_element()
True
```

```
>>> from sage.all import *
>>> C(C.an_element()) == C.an_element()
True
```

REFERENCES:

- Wikipedia article Generalized_symmetric_group
- Wikipedia article Complex_reflection_group

class sage.combinat.colored_permutations.**ShephardToddFamilyGroup**(*m*, *p*, *n*)

Bases: `UniqueRepresentation`, `Parent`

The Shephard-Todd family complex reflection group $G(m, p, n)$ realized as a subgroup of *colored permutations*.

A general complex reflection group is a subgroup of $GL(V)$, where V is a **C** vector space, that is generated by *reflections*, diagonalizable matrices with at most one eigenvalue not equal to 1. The group of colored permutations

$G(m, 1, n)$ are the generalized permutation matrices whose entries are m-th roots of unity. For $p|m$, the group $G(m, p, n)$ is the index p subgroup such that the product of the entries is a m/p-th root of unity.

By the (Chevalley-)Shephard-Todd classification of irreducible finite complex reflection groups, the groups $G(m, p, n)$ (with $G(2, 2, 2)$ being exceptionally reducible since it is the Klein four group) form the only infinite family with an additional 34 exceptional groups G_k, where $4 \leq k \leq 37$. To avoid ambiguities, we refer to $G(m, p, n)$ as the *Shephard-Todd family complex reflection group*.

INPUT:

- m – positive integer
- p – positive integer dividing m
- n – positive integer

REFERENCES:

- Wikipedia article Complex_reflection_group

EXAMPLES:

```
sage: groups.misc.ShephardToddFamily(6, 1, 4)
6-colored permutations of size 4
sage: groups.misc.ShephardToddFamily(6, 2, 4)
Complex reflection group G(6, 2, 4)
sage: groups.misc.ShephardToddFamily(6, 3, 4)
Complex reflection group G(6, 3, 4)
sage: groups.misc.ShephardToddFamily(6, 6, 4)
Complex reflection group G(6, 6, 4)
```

```
>>> from sage.all import *
>>> groups.misc.ShephardToddFamily(Integer(6), Integer(1), Integer(4))
6-colored permutations of size 4
>>> groups.misc.ShephardToddFamily(Integer(6), Integer(2), Integer(4))
Complex reflection group G(6, 2, 4)
>>> groups.misc.ShephardToddFamily(Integer(6), Integer(3), Integer(4))
Complex reflection group G(6, 3, 4)
>>> groups.misc.ShephardToddFamily(Integer(6), Integer(6), Integer(4))
Complex reflection group G(6, 6, 4)
```

Element

alias of *ColoredPermutation*

as_permutation_group()

Return the permutation group corresponding to self.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.as_permutation_group()                                          #␣
↪needs sage.groups
Complex reflection group G(4, 1, 3) as a permutation group
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.as_permutation_group()                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
Complex reflection group G(4, 1, 3) as a permutation group
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.cardinality()
384
sage: C.cardinality() == 4**3 * factorial(3)
True
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.cardinality()
384
>>> C.cardinality() == Integer(4)**Integer(3) * factorial(Integer(3))
True
```

codegrees()

Return the codegrees of `self`.

Let G be a complex reflection group. The codegrees $d_1^* \leq d_2^* \leq \cdots \leq d_\ell^*$ of G can be defined by:

$$\prod_{i=1}^{\ell}(q - d_i^* - 1) = \sum_{g \in G} \det(g) q^{\dim(V^g)},$$

where V is the natural complex vector space that G acts on and ℓ is the `rank()`.

If $m = 1$, then we are in the special case of the symmetric group and the codegrees are $(n-2, n-3, \ldots 1, 0)$. Otherwise the degrees are $((n-1)m, (n-2)m, \ldots, m, 0)$.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.codegrees()
(8, 4, 0)
sage: S = ColoredPermutations(1, 3)
sage: S.codegrees()
(1, 0)
sage: G = groups.misc.ShephardToddFamily(6, 2, 3)
sage: G.codegrees()
(12, 6, 0)
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.codegrees()
(8, 4, 0)
>>> S = ColoredPermutations(Integer(1), Integer(3))
>>> S.codegrees()
(1, 0)
```

(continues on next page)

(continued from previous page)

```
>>> G = groups.misc.ShephardToddFamily(Integer(6), Integer(2), Integer(3))
>>> G.codegrees()
(12, 6, 0)
```

coxeter_matrix()

Return the Coxeter matrix of `self`.

When the group is imprimitive and not a Coxeter group, this returns `None`.

EXAMPLES:

```
sage: C = ColoredPermutations(3, 4)
sage: C.coxeter_matrix()                                              #␣
↪needs sage.modules
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]

sage: C = ColoredPermutations(1, 4)
sage: C.coxeter_matrix()                                              #␣
↪needs sage.modules
[1 3 2]
[3 1 3]
[2 3 1]

sage: G = groups.misc.ShephardToddFamily(2, 2, 3)
sage: G.coxeter_matrix()
[1 3 3]
[3 1 2]
[3 2 1]

sage: G = groups.misc.ShephardToddFamily(2, 2, 2)
sage: G.coxeter_matrix()
[1 2]
[2 1]

sage: G = groups.misc.ShephardToddFamily(2, 2, 1)
sage: G.coxeter_matrix()
[1]

sage: G = groups.misc.ShephardToddFamily(5, 5, 1)
sage: G.coxeter_matrix()
[]

sage: G = groups.misc.ShephardToddFamily(4, 4, 2)
sage: G.coxeter_matrix()
[1 4]
[4 1]

sage: G = groups.misc.ShephardToddFamily(7, 7, 2)
sage: G.coxeter_matrix()
[1 7]
```

(continues on next page)

(continued from previous page)

```
[7 1]

sage: G = groups.misc.ShephardToddFamily(6, 3, 1)
sage: G.coxeter_matrix() is None
True
sage: G = groups.misc.ShephardToddFamily(6, 3, 4)
sage: G.coxeter_matrix() is None
True
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(3), Integer(4))
>>> C.coxeter_matrix()                                                    #␣
↪needs sage.modules
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]

>>> C = ColoredPermutations(Integer(1), Integer(4))
>>> C.coxeter_matrix()                                                    #␣
↪needs sage.modules
[1 3 2]
[3 1 3]
[2 3 1]

>>> G = groups.misc.ShephardToddFamily(Integer(2), Integer(2), Integer(3))
>>> G.coxeter_matrix()
[1 3 3]
[3 1 2]
[3 2 1]

>>> G = groups.misc.ShephardToddFamily(Integer(2), Integer(2), Integer(2))
>>> G.coxeter_matrix()
[1 2]
[2 1]

>>> G = groups.misc.ShephardToddFamily(Integer(2), Integer(2), Integer(1))
>>> G.coxeter_matrix()
[1]

>>> G = groups.misc.ShephardToddFamily(Integer(5), Integer(5), Integer(1))
>>> G.coxeter_matrix()
[]

>>> G = groups.misc.ShephardToddFamily(Integer(4), Integer(4), Integer(2))
>>> G.coxeter_matrix()
[1 4]
[4 1]

>>> G = groups.misc.ShephardToddFamily(Integer(7), Integer(7), Integer(2))
>>> G.coxeter_matrix()
[1 7]
```

(continues on next page)

(continued from previous page)

```
[7 1]

>>> G = groups.misc.ShephardToddFamily(Integer(6), Integer(3), Integer(1))
>>> G.coxeter_matrix() is None
True
>>> G = groups.misc.ShephardToddFamily(Integer(6), Integer(3), Integer(4))
>>> G.coxeter_matrix() is None
True
```

degrees()

Return the degrees of `self`.

The degrees of a complex reflection group are the degrees of the fundamental invariants of the ring of polynomial invariants.

If $m = 1$, then we are in the special case of the symmetric group and the degrees are $(2, 3, \ldots, n, n + 1)$. Otherwise the degrees are $(m, 2m, \ldots, nm)$.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.degrees()
(4, 8, 12)
sage: S = ColoredPermutations(1, 3)
sage: S.degrees()
(2, 3)
sage: G = groups.misc.ShephardToddFamily(6, 2, 3)
sage: G.degrees()
(6, 9, 12)
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.degrees()
(4, 8, 12)
>>> S = ColoredPermutations(Integer(1), Integer(3))
>>> S.degrees()
(2, 3)
>>> G = groups.misc.ShephardToddFamily(Integer(6), Integer(2), Integer(3))
>>> G.degrees()
(6, 9, 12)
```

We now check that the product of the degrees is equal to the cardinality of `self`:

```
sage: prod(C.degrees()) == C.cardinality()
True
sage: prod(S.degrees()) == S.cardinality()
True
sage: prod(G.degrees()) == G.cardinality()
True
```

```
>>> from sage.all import *
>>> prod(C.degrees()) == C.cardinality()
True
```

(continues on next page)

(continued from previous page)

```
>>> prod(S.degrees()) == S.cardinality()
True
>>> prod(G.degrees()) == G.cardinality()
True
```

fixed_point_polynomial(*q=None*)

The fixed point polynomial of `self`.

The fixed point polynomial f_G of a complex reflection group G is counting the dimensions of fixed points subspaces:

$$f_G(q) = \sum_{w \in W} q^{\dim V^w}.$$

Furthermore, let $d_1, d_2, \ldots, d_\ell$ be the degrees of G, where ℓ is the `rank()`. Then the fixed point polynomial is given by

$$f_G(q) = \prod_{i=1}^{\ell} (q + d_i - 1).$$

INPUT:

- `q` – (default: the generator of `ZZ['q']`) the parameter q

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.fixed_point_polynomial()
q^3 + 21*q^2 + 131*q + 231

sage: S = ColoredPermutations(1, 3)
sage: S.fixed_point_polynomial()
q^2 + 3*q + 2
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.fixed_point_polynomial()
q^3 + 21*q^2 + 131*q + 231

>>> S = ColoredPermutations(Integer(1), Integer(3))
>>> S.fixed_point_polynomial()
q^2 + 3*q + 2
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.gens()
([[0, 0, 0], [2, 1, 3]],
 [[0, 0, 0], [1, 3, 2]],
 [[0, 0, 1], [1, 2, 3]])
```

(continues on next page)

(continued from previous page)

```
sage: S = SignedPermutations(4)
sage: S.gens()
([2, 1, 3, 4], [1, 3, 2, 4], [1, 2, 4, 3], [1, 2, 3, -4])
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.gens()
([[0, 0, 0], [2, 1, 3]],
 [[0, 0, 0], [1, 3, 2]],
 [[0, 0, 1], [1, 2, 3]])

>>> S = SignedPermutations(Integer(4))
>>> S.gens()
([2, 1, 3, 4], [1, 3, 2, 4], [1, 2, 4, 3], [1, 2, 3, -4])
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(3, 4)
sage: C.index_set()
(1, 2, 3, 4)

sage: C = ColoredPermutations(1, 4)
sage: C.index_set()
(1, 2, 3)

sage: G = groups.misc.ShephardToddFamily(6, 6, 4)
sage: G.index_set()
(1, 2, 3, 4)

sage: G = groups.misc.ShephardToddFamily(6, 2, 4)
sage: G.index_set()
(1, 2, 3, 4, 5)

sage: G = groups.misc.ShephardToddFamily(6, 6, 1)
sage: G.index_set()
()

sage: G = groups.misc.ShephardToddFamily(6, 2, 1)
sage: G.index_set()
(1,)
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(3), Integer(4))
>>> C.index_set()
(1, 2, 3, 4)

>>> C = ColoredPermutations(Integer(1), Integer(4))
>>> C.index_set()
(1, 2, 3)
```

(continues on next page)

(continued from previous page)

```
>>> G = groups.misc.ShephardToddFamily(Integer(6), Integer(6), Integer(4))
>>> G.index_set()
(1, 2, 3, 4)

>>> G = groups.misc.ShephardToddFamily(Integer(6), Integer(2), Integer(4))
>>> G.index_set()
(1, 2, 3, 4, 5)

>>> G = groups.misc.ShephardToddFamily(Integer(6), Integer(6), Integer(1))
>>> G.index_set()
()

>>> G = groups.misc.ShephardToddFamily(Integer(6), Integer(2), Integer(1))
>>> G.index_set()
(1,)
```

is_well_generated()

Return if `self` is a well-generated complex reflection group.

A complex reflection group G is well-generated if it is generated by ℓ reflections. Equivalently, G is well-generated if $d_i + d_i^* = d_\ell$ for all $1 \leq i \leq \ell$.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.is_well_generated()
True
sage: C = ColoredPermutations(2, 8)
sage: C.is_well_generated()
True
sage: C = ColoredPermutations(1, 4)
sage: C.is_well_generated()
True
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.is_well_generated()
True
>>> C = ColoredPermutations(Integer(2), Integer(8))
>>> C.is_well_generated()
True
>>> C = ColoredPermutations(Integer(1), Integer(4))
>>> C.is_well_generated()
True
```

matrix_group()

Return the matrix group corresponding to `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.matrix_group()                                                         #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
Matrix group over Cyclotomic Field of order 4 and degree 2 with 3 generators (
[0 1 0]  [1 0 0]  [    1     0     0]
[1 0 0]  [0 0 1]  [    0     1     0]
[0 0 1], [0 1 0], [    0     0 zeta4]
)
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.matrix_group()                                                           #␣
↪needs sage.modules
Matrix group over Cyclotomic Field of order 4 and degree 2 with 3 generators (
[0 1 0]  [1 0 0]  [    1     0     0]
[1 0 0]  [0 0 1]  [    0     1     0]
[0 0 1], [0 1 0], [    0     0 zeta4]
)
```

number_of_reflection_hyperplanes()

Return the number of reflection hyperplanes of `self`.

The number of reflection hyperplanes of a complex reflection group is equal to the sum of the codegrees plus the rank.

EXAMPLES:

```
sage: C = ColoredPermutations(1, 2)
sage: C.number_of_reflection_hyperplanes()
1
sage: C = ColoredPermutations(1, 3)
sage: C.number_of_reflection_hyperplanes()
3
sage: C = ColoredPermutations(4, 12)
sage: C.number_of_reflection_hyperplanes()
276
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(1), Integer(2))
>>> C.number_of_reflection_hyperplanes()
1
>>> C = ColoredPermutations(Integer(1), Integer(3))
>>> C.number_of_reflection_hyperplanes()
3
>>> C = ColoredPermutations(Integer(4), Integer(12))
>>> C.number_of_reflection_hyperplanes()
276
```

one()

Return the identity element of `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.one()
```

(continues on next page)

(continued from previous page)

```
[[0, 0, 0], [1, 2, 3]]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.one()
[[0, 0, 0], [1, 2, 3]]
```

order()

Return the cardinality of `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.cardinality()
384
sage: C.cardinality() == 4**3 * factorial(3)
True
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.cardinality()
384
>>> C.cardinality() == Integer(4)**Integer(3) * factorial(Integer(3))
True
```

random_element()

Return an element of `self` at random.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s = C.random_element(); s  # random
[[0, 2, 1], [2, 1, 3]]
sage: s in C
True
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> s = C.random_element(); s  # random
[[0, 2, 1], [2, 1, 3]]
>>> s in C
True
```

rank()

Return the rank of `self`.

The rank of a complex reflection group is equal to the dimension of the complex vector space the group acts on.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 12)
sage: C.rank()
```

(continues on next page)

(continued from previous page)

```
12
sage: C = ColoredPermutations(7, 4)
sage: C.rank()
4
sage: C = ColoredPermutations(1, 4)
sage: C.rank()
3
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(12))
>>> C.rank()
12
>>> C = ColoredPermutations(Integer(7), Integer(4))
>>> C.rank()
4
>>> C = ColoredPermutations(Integer(1), Integer(4))
>>> C.rank()
3
```

simple_reflection(*i*)

Return the i-th simple reflection of self.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.gens()
([[0, 0, 0], [2, 1, 3]], [[0, 0, 0], [1, 3, 2]], [[0, 0, 1], [1, 2, 3]])
sage: C.simple_reflection(2)
[[0, 0, 0], [1, 3, 2]]
sage: C.simple_reflection(3)
[[0, 0, 1], [1, 2, 3]]

sage: S = SignedPermutations(4)
sage: S.simple_reflection(1)
[2, 1, 3, 4]
sage: S.simple_reflection(4)
[1, 2, 3, -4]

sage: G = groups.misc.ShephardToddFamily(4, 2, 3)
sage: list(G.simple_reflections())
[[[0, 0, 0], [2, 1, 3]],
 [[0, 0, 0], [1, 3, 2]],
 [[0, 1, 3], [1, 3, 2]],
 [[0, 0, 2], [1, 2, 3]]]

sage: G = groups.misc.ShephardToddFamily(8, 4, 1)
sage: G.simple_reflections()
Finite family {1: [[4], [1]]}

sage: G = groups.misc.ShephardToddFamily(8, 8, 1)
sage: G.simple_reflections()
Finite family {}
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(4), Integer(3))
>>> C.gens()
([[0, 0, 0], [2, 1, 3]], [[0, 0, 0], [1, 3, 2]], [[0, 0, 1], [1, 2, 3]])
>>> C.simple_reflection(Integer(2))
[[0, 0, 0], [1, 3, 2]]
>>> C.simple_reflection(Integer(3))
[[0, 0, 1], [1, 2, 3]]

>>> S = SignedPermutations(Integer(4))
>>> S.simple_reflection(Integer(1))
[2, 1, 3, 4]
>>> S.simple_reflection(Integer(4))
[1, 2, 3, -4]

>>> G = groups.misc.ShephardToddFamily(Integer(4), Integer(2), Integer(3))
>>> list(G.simple_reflections())
[[[0, 0, 0], [2, 1, 3]],
 [[0, 0, 0], [1, 3, 2]],
 [[0, 1, 3], [1, 3, 2]],
 [[0, 0, 2], [1, 2, 3]]]

>>> G = groups.misc.ShephardToddFamily(Integer(8), Integer(4), Integer(1))
>>> G.simple_reflections()
Finite family {1: [[4], [1]]}

>>> G = groups.misc.ShephardToddFamily(Integer(8), Integer(8), Integer(1))
>>> G.simple_reflections()
Finite family {}
```

class sage.combinat.colored_permutations.**SignedPermutation**(*parent*, *colors*, *perm*)

Bases: *ColoredPermutation*

A signed permutation.

cycle_type()

Return a pair of partitions of `len(self)` corresponding to the signed cycle type of `self`.

A *cycle* is a tuple $C = (c_0, \ldots, c_{k-1})$ with $\pi(c_i) = c_{i+1}$ for $0 \leq i < k$ and $\pi(c_{k-1}) = c_0$. If C is a cycle, $\overline{C} = (-c_0, \ldots, -c_{k-1})$ is also a cycle. A cycle is *negative*, if $C = \overline{C}$ up to cyclic reordering. In this case, k is necessarily even and the length of C is $k/2$. A *positive cycle* is a pair $C\overline{C}$, its length is k.

Let α be the partition whose parts are the lengths of the positive cycles and let β be the partition whose parts are the lengths of the negative cycles. Then (α, β) is the cycle type of π.

EXAMPLES:

```
sage: G = SignedPermutations(7)
sage: pi = G([2, -1, 4, -6, -5, -3, 7])
sage: pi.cycle_type()
([3, 1], [2, 1])

sage: G = SignedPermutations(5)
sage: all(pi.cycle_type().size() == 5 for pi in G)
True
```

(continues on next page)

(continued from previous page)

```
sage: set(pi.cycle_type() for pi in G) == set(PartitionTuples(2, 5))
True
```

```
>>> from sage.all import *
>>> G = SignedPermutations(Integer(7))
>>> pi = G([Integer(2), -Integer(1), Integer(4), -Integer(6), -Integer(5), -
↪Integer(3), Integer(7)])
>>> pi.cycle_type()
([3, 1], [2, 1])

>>> G = SignedPermutations(Integer(5))
>>> all(pi.cycle_type().size() == Integer(5) for pi in G)
True
>>> set(pi.cycle_type() for pi in G) == set(PartitionTuples(Integer(2),
↪Integer(5)))
True
```

has_left_descent (*i*)

Return `True` if `i` is a left descent of `self`.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: s1,s2,s3,s4 = S.gens()
sage: x = s4*s1*s2*s3*s4
sage: [x.has_left_descent(i) for i in S.index_set()]
[True, False, False, True]
```

```
>>> from sage.all import *
>>> S = SignedPermutations(Integer(4))
>>> s1,s2,s3,s4 = S.gens()
>>> x = s4*s1*s2*s3*s4
>>> [x.has_left_descent(i) for i in S.index_set()]
[True, False, False, True]
```

order ()

Return the multiplicative order of the signed permutation.

EXAMPLES:

```
sage: pi = SignedPermutations(7)([2,-1,4,-6,-5,-3,7])
sage: pi.to_cycles(singletons=False)
[(1, 2, -1, -2), (3, 4, -6), (-3, -4, 6), (5, -5)]
sage: pi.order()
12
```

```
>>> from sage.all import *
>>> pi = SignedPermutations(Integer(7))([Integer(2),-Integer(1),Integer(4),-
↪Integer(6),-Integer(5),-Integer(3),Integer(7)])
>>> pi.to_cycles(singletons=False)
[(1, 2, -1, -2), (3, 4, -6), (-3, -4, 6), (5, -5)]
>>> pi.order()
12
```

to_cycles(*singletons=True*, *use_min=True*, *negative_cycles=True*)

Return the signed permutation `self` as a list of disjoint cycles.

The cycles are returned in the order of increasing smallest elements, and each cycle is returned as a tuple which starts with its smallest positive element.

INPUT:

- `singletons` – boolean (default: `True`); whether to include singleton cycles or not
- `use_min` – boolean (default: `True`); if `False`, the cycles are returned in the order of increasing *largest* (not smallest) elements, and each cycle starts with its largest element
- `negative_cycles` – boolean (default: `True`); if `False`, for any two cycles $C^{\pm} = \{\pm c_1, \ldots, \pm c_k\}$ such that $C^+ \neq C^-$, this does not include the cycle C^-

Warning

The arugment `negative_cycles` does not refer to the usual definition of a negative cycle; see `cycle_type()`.

EXAMPLES:

```
sage: pi = SignedPermutations(7)([2,-1,4,-6,-5,-3,7])
sage: pi.to_cycles()
[(1, 2, -1, -2), (3, 4, -6), (-3, -4, 6), (5, -5), (7,), (-7,)]
sage: pi.to_cycles(singletons=False)
[(1, 2, -1, -2), (3, 4, -6), (-3, -4, 6), (5, -5)]
sage: pi.to_cycles(negative_cycles=False)
[(1, 2, -1, -2), (3, 4, -6), (5, -5), (7,)]
sage: pi.to_cycles(singletons=False, negative_cycles=False)
[(1, 2, -1, -2), (3, 4, -6), (5, -5)]
sage: pi.to_cycles(use_min=False)
[(7,), (-7,), (6, -3, -4), (-6, 3, 4), (5, -5), (2, -1, -2, 1)]
sage: pi.to_cycles(singletons=False, use_min=False)
[(6, -3, -4), (-6, 3, 4), (5, -5), (2, -1, -2, 1)]
```

```
>>> from sage.all import *
>>> pi = SignedPermutations(Integer(7))([Integer(2),-Integer(1),Integer(4),-
↪Integer(6),-Integer(5),-Integer(3),Integer(7)])
>>> pi.to_cycles()
[(1, 2, -1, -2), (3, 4, -6), (-3, -4, 6), (5, -5), (7,), (-7,)]
>>> pi.to_cycles(singletons=False)
[(1, 2, -1, -2), (3, 4, -6), (-3, -4, 6), (5, -5)]
>>> pi.to_cycles(negative_cycles=False)
[(1, 2, -1, -2), (3, 4, -6), (5, -5), (7,)]
>>> pi.to_cycles(singletons=False, negative_cycles=False)
[(1, 2, -1, -2), (3, 4, -6), (5, -5)]
>>> pi.to_cycles(use_min=False)
[(7,), (-7,), (6, -3, -4), (-6, 3, 4), (5, -5), (2, -1, -2, 1)]
>>> pi.to_cycles(singletons=False, use_min=False)
[(6, -3, -4), (-6, 3, 4), (5, -5), (2, -1, -2, 1)]
```

to_matrix()

Return a matrix of `self`.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: s1,s2,s3,s4 = S.gens()
sage: x = s4*s1*s2*s3*s4
sage: M = x.to_matrix(); M                                              #␣
↪needs sage.modules
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0 -1]
[-1  0  0  0]
```

```
>>> from sage.all import *
>>> S = SignedPermutations(Integer(4))
>>> s1,s2,s3,s4 = S.gens()
>>> x = s4*s1*s2*s3*s4
>>> M = x.to_matrix(); M                                              #␣
↪needs sage.modules
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0 -1]
[-1  0  0  0]
```

The matrix multiplication is in the *opposite* order:

```
sage: m1,m2,m3,m4 = [g.to_matrix() for g in S.gens()]                   #␣
↪needs sage.modules
sage: M == m4 * m3 * m2 * m1 * m4                                        #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> m1,m2,m3,m4 = [g.to_matrix() for g in S.gens()]                   #␣
↪needs sage.modules
>>> M == m4 * m3 * m2 * m1 * m4                                        #␣
↪needs sage.modules
True
```

class sage.combinat.colored_permutations.**SignedPermutations**(*n*)

Bases: *ColoredPermutations*

Group of signed permutations.

The group of signed permutations is also known as the hyperoctahedral group, the Coxeter group of type B_n, and the 2-colored permutation group. Thus it can be constructed as the wreath product $S_2 \wr S_n$.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: s1,s2,s3,s4 = S.group_generators()
sage: x = s4*s1*s2*s3*s4; x
[-4, 1, 2, -3]
sage: x^4 == S.one()
True
```

```
>>> from sage.all import *
>>> S = SignedPermutations(Integer(4))
>>> s1,s2,s3,s4 = S.group_generators()
>>> x = s4*s1*s2*s3*s4; x
[-4, 1, 2, -3]
>>> x**Integer(4) == S.one()
True
```

This is a finite Coxeter group of type B_n:

```
sage: S.canonical_representation()                                          #␣
↪needs sage.modules
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 2
 with a = 1.414213562373095? with Coxeter matrix:
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
sage: S.long_element()
[-1, -2, -3, -4]
sage: S.long_element().reduced_word()
[1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 4, 3, 2, 4, 3, 4]
```

```
>>> from sage.all import *
>>> S.canonical_representation()                                        #␣
↪needs sage.modules
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 2
 with a = 1.414213562373095? with Coxeter matrix:
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
>>> S.long_element()
[-1, -2, -3, -4]
>>> S.long_element().reduced_word()
[1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 4, 3, 2, 4, 3, 4]
```

We can also go between the 2-colored permutation group:

```
sage: C = ColoredPermutations(2, 3)
sage: S = SignedPermutations(3)
sage: S.an_element()
[-3, 1, 2]
sage: C(S.an_element())
[[1, 0, 0], [3, 1, 2]]
sage: S(C(S.an_element())) == S.an_element()
True
sage: S(C.an_element())
[-3, 1, 2]
```

```
>>> from sage.all import *
>>> C = ColoredPermutations(Integer(2), Integer(3))
>>> S = SignedPermutations(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> S.an_element()
[-3, 1, 2]
>>> C(S.an_element())
[[1, 0, 0], [3, 1, 2]]
>>> S(C(S.an_element())) == S.an_element()
True
>>> S(C.an_element())
[-3, 1, 2]
```

There is also the natural lift from permutations:

```
sage: P = Permutations(3)
sage: x = S(P.an_element()); x
[3, 1, 2]
sage: x.parent()
Signed permutations of 3
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(3))
>>> x = S(P.an_element()); x
[3, 1, 2]
>>> x.parent()
Signed permutations of 3
```

REFERENCES:

- Wikipedia article Hyperoctahedral_group

Element

alias of *SignedPermutation*

conjugacy_class_representative(*nu*)

Return a permutation with (signed) cycle type `nu`.

EXAMPLES:

```
sage: G = SignedPermutations(4)
sage: for nu in PartitionTuples(2, 4):
....:     print(nu, G.conjugacy_class_representative(nu))
....:     assert nu == G.conjugacy_class_representative(nu).cycle_type(), nu
([4], []) [2, 3, 4, 1]
([3, 1], []) [2, 3, 1, 4]
([2, 2], []) [2, 1, 4, 3]
([2, 1, 1], []) [2, 1, 3, 4]
([1, 1, 1, 1], []) [1, 2, 3, 4]
([3], [1]) [2, 3, 1, -4]
([2, 1], [1]) [2, 1, 3, -4]
([1, 1, 1], [1]) [1, 2, 3, -4]
([2], [2]) [2, 1, 4, -3]
([2], [1, 1]) [2, 1, -3, -4]
([1, 1], [2]) [1, 2, 4, -3]
([1, 1], [1, 1]) [1, 2, -3, -4]
([1], [3]) [1, 3, 4, -2]
([1], [2, 1]) [1, 3, -2, -4]
```

(continues on next page)

(continued from previous page)

```
([1], [1, 1, 1]) [1, -2, -3, -4]
([], [4]) [2, 3, 4, -1]
([], [3, 1]) [2, 3, -1, -4]
([], [2, 2]) [2, -1, 4, -3]
([], [2, 1, 1]) [2, -1, -3, -4]
([], [1, 1, 1, 1]) [-1, -2, -3, -4]
```

```
>>> from sage.all import *
>>> G = SignedPermutations(Integer(4))
>>> for nu in PartitionTuples(Integer(2), Integer(4)):
...     print(nu, G.conjugacy_class_representative(nu))
...     assert nu == G.conjugacy_class_representative(nu).cycle_type(), nu
([4], []) [2, 3, 4, 1]
([3, 1], []) [2, 3, 1, 4]
([2, 2], []) [2, 1, 4, 3]
([2, 1, 1], []) [2, 1, 3, 4]
([1, 1, 1, 1], []) [1, 2, 3, 4]
([3], [1]) [2, 3, 1, -4]
([2, 1], [1]) [2, 1, 3, -4]
([1, 1, 1], [1]) [1, 2, 3, -4]
([2], [2]) [2, 1, 4, -3]
([2], [1, 1]) [2, 1, -3, -4]
([1, 1], [2]) [1, 2, 4, -3]
([1, 1], [1, 1]) [1, 2, -3, -4]
([1], [3]) [1, 3, 4, -2]
([1], [2, 1]) [1, 3, -2, -4]
([1], [1, 1, 1]) [1, -2, -3, -4]
([], [4]) [2, 3, 4, -1]
([], [3, 1]) [2, 3, -1, -4]
([], [2, 2]) [2, -1, 4, -3]
([], [2, 1, 1]) [2, -1, -3, -4]
([], [1, 1, 1, 1]) [-1, -2, -3, -4]
```

long_element(*index_set=None*)

Return the longest element of `self`, or of the parabolic subgroup corresponding to the given `index_set`.

INPUT:

- `index_set` – (optional) a subset (as a list or iterable) of the nodes of the indexing set

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.long_element()
[-1, -2, -3, -4]
```

```
>>> from sage.all import *
>>> S = SignedPermutations(Integer(4))
>>> S.long_element()
[-1, -2, -3, -4]
```

one()

Return the identity element of `self`.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.one()
[1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> S = SignedPermutations(Integer(4))
>>> S.one()
[1, 2, 3, 4]
```

random_element()

Return an element drawn uniformly at random.

EXAMPLES:

```
sage: C = SignedPermutations(7)
sage: s = C.random_element(); s # random
[7, 6, -4, -5, 2, 3, -1]
sage: s in C
True
```

```
>>> from sage.all import *
>>> C = SignedPermutations(Integer(7))
>>> s = C.random_element(); s # random
[7, 6, -4, -5, 2, 3, -1]
>>> s in C
True
```

simple_reflection(*i*)

Return the `i`-th simple reflection of `self`.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.simple_reflection(1)
[2, 1, 3, 4]
sage: S.simple_reflection(4)
[1, 2, 3, -4]
```

```
>>> from sage.all import *
>>> S = SignedPermutations(Integer(4))
>>> S.simple_reflection(Integer(1))
[2, 1, 3, 4]
>>> S.simple_reflection(Integer(4))
[1, 2, 3, -4]
```

5.1.25 Combinatorial Functions

This module implements some combinatorial functions, as listed below. For a more detailed description, see the relevant docstrings.

Sequences:

- Bell numbers, *bell_number()*

- Catalan numbers, `catalan_number()` (not to be confused with the Catalan constant)
- Narayana numbers, `narayana_number()`
- Euler numbers, `euler_number()` (Maxima)
- Eulerian numbers, `eulerian_number()`
- Eulerian polynomial, `eulerian_polynomial()`
- Fibonacci numbers, `fibonacci()` (PARI) and `fibonacci_number()` (GAP) The PARI version is better.
- Lucas numbers, `lucas_number1()`, `lucas_number2()`.
- Stirling numbers, `stirling_number1()`, `stirling_number2()`.
- Polygonal numbers, `polygonal_number()`

Set-theoretic constructions:

- Derangements of a multiset, `derangements()` and `number_of_derangements()`.
- Tuples of a multiset, `tuples()` and `number_of_tuples()`. An ordered tuple of length k of set S is a ordered selection with repetitions of S and is represented by a sorted list of length k containing elements from S.
- Unordered tuples of a set, `unordered_tuples()` and `number_of_unordered_tuples()`. An unordered tuple of length k of set S is an unordered selection with repetitions of S and is represented by a sorted list of length k containing elements from S.

Warning

The following function is deprecated and will soon be removed.

- Permutations of a multiset, `permutations()`, `permutations_iterator()`, `number_of_permutations()`. A permutation is a list that contains exactly the same elements but possibly in different order.

Related functions:

- Bernoulli polynomials, `bernoulli_polynomial()`

Implemented in other modules (listed for completeness):

The package `sage.arith` contains the following combinatorial functions:

- `binomial()` the binomial coefficient (wrapped from PARI)
- `factorial()` (wrapped from PARI)
- `falling_factorial()` Definition: for integer $a \geq 0$ we have $x(x-1)\cdots(x-a+1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+1)}{\Gamma(x-a+1)}$.
- `rising_factorial()` Definition: for integer $a \geq 0$ we have $x(x+1)\cdots(x+a-1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+a)}{\Gamma(x)}$.

From other modules:

- `number_of_partitions()` (wrapped from PARI) the *number* of partitions:
- `sage.combinat.q_analogues.gaussian_binomial()` the Gaussian binomial

$$\binom{n}{k}_q = \frac{(1-q^m)(1-q^{m-1})\cdots(1-q^{m-r+1})}{(1-q)(1-q^2)\cdots(1-q^r)}.$$

The `sage.groups.perm_gps.permgroup_elements` contains the following combinatorial functions:

- matrix method of PermutationGroupElement yielding the permutation matrix of the group element.

Todo

GUAVA commands:

- VandermondeMat
- GrayMat returns a list of all different vectors of length n over the field F, using Gray ordering.

Not in GAP:

- Rencontres numbers (Wikipedia article Rencontres_number)

REFERENCES:

- Wikipedia article Twelvefold_way (general reference)

AUTHORS:

- David Joyner (2006-07): initial implementation.
- William Stein (2006-07): editing of docs and code; many optimizations, refinements, and bug fixes in corner cases
- David Joyner (2006-09): bug fix for combinations, added permutations_iterator, combinations_iterator from Python Cookbook, edited docs.
- David Joyner (2007-11): changed permutations, added hadamard_matrix
- Florent Hivert (2009-02): combinatorial class cleanup
- Fredrik Johansson (2010-07): fast implementation of `stirling_number2`
- Punarbasu Purkayastha (2012-12): deprecate arrangements, combinations, combinations_iterator, and clean up very old deprecated methods.

Functions and classes

class `sage.combinat.combinat.`**`CombinatorialElement`**(*parent*, **args*, ***kwds*)

Bases: *`CombinatorialObject`*, `Element`

`CombinatorialElement` is both a *`CombinatorialObject`* and an `Element`. So it represents a list which is an element of some parent.

A `CombinatorialElement` subclass also automatically supports the `__classcall__` mechanism.

Warning

This class is slowly being deprecated. Use `ClonableList` instead.

INPUT:

- `parent` – the `Parent` class for this element
- `lst` – list or any object that can be converted to a list by calling `list()`

EXAMPLES:

```
sage: # needs sage.combinat
sage: from sage.combinat.combinat import CombinatorialElement
sage: e = CombinatorialElement(Partitions(6), [3,2,1])
sage: e == loads(dumps(e))
True
sage: parent(e)
Partitions of the integer 6
sage: list(e)
[3, 2, 1]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> from sage.combinat.combinat import CombinatorialElement
>>> e = CombinatorialElement(Partitions(Integer(6)), [Integer(3),Integer(2),
↪Integer(1)])
>>> e == loads(dumps(e))
True
>>> parent(e)
Partitions of the integer 6
>>> list(e)
[3, 2, 1]
```

Check classcalls:

```
sage: class Foo(CombinatorialElement):                                  #␣
↪needs sage.combinat
....:     @staticmethod
....:     def __classcall__(cls, x):
....:         return x
sage: Foo(17)                                                            #␣
↪needs sage.combinat
17
```

```
>>> from sage.all import *
>>> class Foo(CombinatorialElement):                                   #␣
↪needs sage.combinat
...     @staticmethod
...     def __classcall__(cls, x):
...         return x
>>> Foo(Integer(17))                                                     ␣
↪     # needs sage.combinat
17
```

class sage.combinat.combinat.**CombinatorialObject**(*l*, *copy=True*)

Bases: SageObject

CombinatorialObject provides a thin wrapper around a list. The main differences are that __setitem__ is disabled so that CombinatorialObjects are shallowly immutable, and the intention is that they are semantically immutable.

Because of this, CombinatorialObjects provide a __hash__ function which computes the hash of the string representation of a list and the hash of its parent's class. Thus, each CombinatorialObject should have a unique string representation.

See also

`CombinatorialElement` if you want a combinatorial object which is an element of a parent.

Warning

This class is slowly being deprecated. Use `ClonableList` instead.

INPUT:

- `l` – list or any object that can be converted to a list by calling `list()`
- `copy` – boolean (default: `True`); if `False`, then `l` must be a `list`, which is assigned to `self._list` without copying

EXAMPLES:

```
sage: c = CombinatorialObject([1,2,3])
sage: c == loads(dumps(c))
True
sage: c._list
[1, 2, 3]
sage: c._hash is None
True
```

```
>>> from sage.all import *
>>> c = CombinatorialObject([Integer(1),Integer(2),Integer(3)])
>>> c == loads(dumps(c))
True
>>> c._list
[1, 2, 3]
>>> c._hash is None
True
```

For efficiency, you can specify `copy=False` if you know what you are doing:

```
sage: from sage.combinat.combinat import CombinatorialObject
sage: x = [3, 2, 1]
sage: C = CombinatorialObject(x, copy=False)
sage: C
[3, 2, 1]
sage: x[0] = 5
sage: C
[5, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat import CombinatorialObject
>>> x = [Integer(3), Integer(2), Integer(1)]
>>> C = CombinatorialObject(x, copy=False)
>>> C
[3, 2, 1]
>>> x[Integer(0)] = Integer(5)
```

(continues on next page)

(continued from previous page)

```
>>> C
[5, 2, 1]
```

index (*key*)

EXAMPLES:

```
sage: c = CombinatorialObject([1,2,3])
sage: c.index(1)
0
sage: c.index(3)
2
```

```
>>> from sage.all import *
>>> c = CombinatorialObject([Integer(1),Integer(2),Integer(3)])
>>> c.index(Integer(1))
0
>>> c.index(Integer(3))
2
```

sage.combinat.combinat.**bell_number** (*n*, *algorithm='flint'*, ***options*)

Return the n-th Bell number.

This is the number of ways to partition a set of n elements into pairwise disjoint nonempty subsets.

INPUT:

- `n` – positive integer
- `algorithm` – (default: `'flint'`) any one of the following:
  - `'dobinski'` – use Dobinski's formula implemented in Sage
  - `'flint'` – wrap FLINT's `arith_bell_number`
  - `'gap'` – wrap GAP's `Bell`
  - `'mpmath'` – wrap mpmath's `bell`

Warning

When using the mpmath algorithm to compute Bell numbers and you specify `prec`, it can return incorrect results due to low precision. See the examples section.

Let B_n denote the n-th Bell number. Dobinski's formula is:

$$B_n = e^{-1} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

To show our implementation of Dobinski's method works, suppose that $n \geq 5$ and let k_0 be the smallest positive integer such that $\frac{k_0^n}{k_0!} < 1$. Note that $k_0 > n$ and $k_0 \leq 2n$ because we can prove that $\frac{(2n)^n}{(2n)!} < 1$ by Stirling.

If $k > k_0$, then we have $\frac{k^n}{k!} < \frac{1}{2^{k-k_0}}$. We show this by induction: let $c_k = \frac{k^n}{k!}$, if $k > n$ then

$$\frac{c_{k+1}}{c_k} = \frac{(1+k^{-1})^n}{k+1} < \frac{(1+n^{-1})^n}{n} < \frac{1}{2}.$$

The last inequality can easily be checked numerically for $n \geq 5$.

Using this, we can see that $\frac{c_k}{c_{k_0}} < \frac{1}{2^{k-k_0}}$ for $k > k_0 > n$. So summing this it gives that $\sum_{k=k_0+1}^{\infty} \frac{k^n}{k!} < 1$, and hence

$$B_n = e^{-1}\left(\sum_{k=0}^{k_0} \frac{k^n}{k!} + E_1\right) = e^{-1}\sum_{k=0}^{k_0} \frac{k^n}{k!} + E_2,$$

where $0 < E_1 < 1$ and $0 < E_2 < e^{-1}$. Next we have for any $q > 0$

$$\sum_{k=0}^{k_0} \frac{k^n}{k!} = \frac{1}{q}\sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor + \frac{E_3}{q}$$

where $0 \leq E_3 \leq k_0 + 1 \leq 2n + 1$. Let $E_4 = \frac{E_3}{q}$ and let $q = 2n + 1$. We find $0 \leq E_4 \leq 1$. These two bounds give:

$$\begin{aligned} B_n &= \frac{e^{-1}}{q}\sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor + e^{-1}E_4 + E_2 \\ &= \frac{e^{-1}}{q}\sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor + E_5 \end{aligned}$$

where

$$0 < E_5 = e^{-1}E_4 + E_2 \leq e^{-1} + e^{-1} < \frac{3}{4}.$$

It follows that

$$B_n = \left\lceil \frac{e^{-1}}{q}\sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor \right\rceil.$$

Now define

$$b = \sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor.$$

This b can be computed exactly using integer arithmetic. To avoid the costly integer division by $k!$, we collect more terms and do only one division, for example with 3 terms:

$$\frac{k^n}{k!} + \frac{(k+1)^n}{(k+1)!} + \frac{(k+2)^n}{(k+2)!} = \frac{k^n(k+1)(k+2) + (k+1)^n(k+2) + (k+2)^n}{(k+2)!}$$

In the implementation, we collect $\sqrt{n}/2$ terms.

To actually compute B_n from b, we let $p = \lfloor \log_2(b) \rfloor + 1$ such that $b < 2^p$ and we compute with p bits of precision. This implies that b (and $q < b$) can be represented exactly.

We compute $\frac{e^{-1}}{q}b$, rounding down, and we must have an absolute error of at most $1/4$ (given that $E_5 < 3/4$). This means that we need a relative error of at most

$$\frac{eq}{4b} > \frac{(eq)/4}{2^p} > \frac{7}{2^p}$$

(assuming $n \geq 5$). With a precision of p bits and rounding down, every rounding has a relative error of at most $2^{1-p} = 2/2^p$. Since we do 3 roundings (b and q do not require rounding), we get a relative error of at most $6/2^p$. All this implies that the precision of p bits is sufficient.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: bell_number(10)
115975
sage: bell_number(2)
2
sage: bell_number(-10)
Traceback (most recent call last):
...
ArithmeticError: Bell numbers not defined for negative indices
sage: bell_number(1)
1
sage: bell_number(1/3)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

```
>>> from sage.all import *
>>> # needs sage.libs.flint
>>> bell_number(Integer(10))
115975
>>> bell_number(Integer(2))
2
>>> bell_number(-Integer(10))
Traceback (most recent call last):
...
ArithmeticError: Bell numbers not defined for negative indices
>>> bell_number(Integer(1))
1
>>> bell_number(Integer(1)/Integer(3))
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

When using the mpmath algorithm, we are required have mpmath's precision set to at least $\log_2(B_n)$ bits. If upon computing the Bell number the first time, we deem the precision too low, we use our guess to (temporarily) raise mpmath's precision and the Bell number is recomputed.

```
sage: k = bell_number(30, 'mpmath'); k                                      #␣
↪needs mpmath
846749014511809332450147
sage: k == bell_number(30)                                                   #␣
↪needs mpmath sage.libs.flint
True
```

```
>>> from sage.all import *
>>> k = bell_number(Integer(30), 'mpmath'); k                               ␣
↪     # needs mpmath
846749014511809332450147
>>> k == bell_number(Integer(30))                                            ␣
↪     # needs mpmath sage.libs.flint
True
```

If you knows what precision is necessary before computing the Bell number, you can use the `prec` option:

```
sage: k2 = bell_number(30, 'mpmath', prec=30); k2                              #␣
↪needs mpmath
846749014511809332450147
sage: k == k2                                                                   #␣
↪needs mpmath
True
```

```
>>> from sage.all import *
>>> k2 = bell_number(Integer(30), 'mpmath', prec=Integer(30)); k2             ␣
↪                 # needs mpmath
846749014511809332450147
>>> k == k2                                                                    #␣
↪needs mpmath
True
```

Warning

Running mpmath with the precision set too low can result in incorrect results:

```
sage: k = bell_number(30, 'mpmath', prec=15); k                                #␣
↪needs mpmath
846749014511809388871680
sage: k == bell_number(30)                                                     #␣
↪needs mpmath sage.libs.flint
False
```

```
>>> from sage.all import *
>>> k = bell_number(Integer(30), 'mpmath', prec=Integer(15)); k               ␣
↪          # needs mpmath
846749014511809388871680
>>> k == bell_number(Integer(30))
↪# needs mpmath sage.libs.flint
False
```

AUTHORS:

- Robert Gerbicz
- Jeroen Demeyer: improved implementation of Dobinski formula with more accurate error estimates (Issue #17157)

REFERENCES:

- Wikipedia article Bell_number
- http://fredrik-j.blogspot.com/2009/03/computing-generalized-bell-numbers.html
- http://mathworld.wolfram.com/DobinskisFormula.html

sage.combinat.combinat.**bell_polynomial**(*n*, *k=None*, *ordinary=False*)

Return the (partial) (exponential/ordinary) bell Polynomial.

The partial (exponential) *Bell polynomial* is defined by the formula

$$B_{n,k}(x_0, x_1, \ldots, x_{n-k}) = \sum_{\substack{j_0+\ldots+j_{n-k}=k \\ 1j_0+\ldots+(n-k+1)j_{n-k}=n}} \frac{n!}{j_0!j_1!\cdots j_{n-k}!} \left(\frac{x_0}{(0+1)!}\right)^{j_0} \left(\frac{x_1}{(1+1)!}\right)^{j_1} \cdots \left(\frac{x_{n-k}}{(n-k+1)!}\right)^{j_{n-k}}.$$

The complete (exponential) Bell Polynomial is defined as

$$B_n(x_0, x_1, \ldots, x_{n-k}) = \sum_{k=0}^{n} B_{n,k}(x_0, x_1, \ldots, x_{n-k}).$$

The ordinary variant of the partial Bell polynomial is defined by

$$\hat{B}_{n,k}(x_0, x_1, \ldots, x_{n-k}) = \sum_{\substack{j_0+\ldots+j_{n-k}=k \\ 1j_0+\ldots+(n-k+1)j_{n-k}=n}} \binom{k}{j_0, j_1, \ldots, j_{n-k}} x_0^{j_0} x_1^{j_1} \cdots x_{n-k}^{j_{n-k}},$$

where we have used the multinomial coefficient. The complete version has the same definition as its exponential counterpart.

If we define $f(z) = \sum_{n=1}^{\infty} x_{n-1} z^n/n!$ then these are alternative definitions for exponential Bell polynomials

$$\exp(f(z)) = \sum_{n=0}^{\infty} B_n(x_0, \ldots, x_{n-1}) \frac{z^n}{n!},$$
$$\frac{f(z)^k}{k!} = \sum_{n=k}^{\infty} B_{n,k}(x_0, \ldots, x_{n-k}) \frac{z^n}{n!}.$$

Defining $g(z) = \sum_{n=1}^{\infty} x_{n-1} z^n$, we have the analoguous alternative definitions

$$\frac{1}{1-f(z)} = \sum_{n=0}^{\infty} \hat{B}_n(x_0, \ldots, x_{n-1}) z^n,$$
$$f(z)^k = \sum_{n=k}^{\infty} \hat{B}_{n,k}(x_0, \ldots, x_{n-k}) z^n,$$

(see reference).

INPUT:

- `k` – (optional) if specified, returns the partial Bell polynomial, otherwise returns the complete Bell polynomial
- `ordinary` – boolean (default: `False`); if `True`, returns the (partial) ordinary Bell polynomial, otherwise returns the (partial) exponential Bell polynomial

EXAMPLES:

The complete and partial Bell polynomials:

```
sage: # needs sage.combinat
sage: bell_polynomial(3)
x0^3 + 3*x0*x1 + x2
sage: bell_polynomial(4)
x0^4 + 6*x0^2*x1 + 3*x1^2 + 4*x0*x2 + x3
sage: bell_polynomial(6, 3)
15*x1^3 + 60*x0*x1*x2 + 15*x0^2*x3
sage: bell_polynomial(6, 6)
x0^6
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> bell_polynomial(Integer(3))
x0^3 + 3*x0*x1 + x2
>>> bell_polynomial(Integer(4))
x0^4 + 6*x0^2*x1 + 3*x1^2 + 4*x0*x2 + x3
>>> bell_polynomial(Integer(6), Integer(3))
15*x1^3 + 60*x0*x1*x2 + 15*x0^2*x3
>>> bell_polynomial(Integer(6), Integer(6))
x0^6
```

The ordinary variants are:

```
sage: # needs sage.combinat sage.arith
sage: bell_polynomial(3, ordinary=True)
x0^3 + 2*x0*x1 + x2
sage: bell_polynomial(4, ordinary=True)
x0^4 + 3*x0^2*x1 + x1^2 + 2*x0*x2 + x3
sage: bell_polynomial(6, 3, True)
x1^3 + 6*x0*x1*x2 + 3*x0^2*x3
sage: bell_polynomial(6, 6, True)
x0^6
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.arith
>>> bell_polynomial(Integer(3), ordinary=True)
x0^3 + 2*x0*x1 + x2
>>> bell_polynomial(Integer(4), ordinary=True)
x0^4 + 3*x0^2*x1 + x1^2 + 2*x0*x2 + x3
>>> bell_polynomial(Integer(6), Integer(3), True)
x1^3 + 6*x0*x1*x2 + 3*x0^2*x3
>>> bell_polynomial(Integer(6), Integer(6), True)
x0^6
```

We verify the alternative definition of the different Bell polynomials using the functions f and g given above:

```
sage: # needs sage.combinat sage.arith
sage: n = 6 # positive integer
sage: k = 4 # positive integer
sage: R.<x> = InfinitePolynomialRing(QQ)
sage: PR = PolynomialRing(QQ, 'x', n)
sage: d = {x[i]: PR.gen(i) for i in range(n)} #substitution dictionnary
sage: L.<z> = LazyPowerSeriesRing(R)
sage: f = L(lambda i: x[i-1]/factorial(i), valuation=1)
sage: all(exp(f)[i].subs(d) * factorial(i) == bell_polynomial(i) for i in␣
↪range(n+1))
True
sage: all((f^k/factorial(k))[i].subs(d) * factorial(i) == bell_polynomial(i, k)␣
↪for i in range(k, n+k))
True
sage: g = L(lambda i: x[i-1], valuation=1)
sage: all((1/(1-g))[i].subs(d) == bell_polynomial(i, ordinary=True) for i in␣
↪range(n+1))
```

(continues on next page)

(continued from previous page)

```
True
sage: all((g^k)[i].subs(d) == bell_polynomial(i, k, True) for i in range(k, n+k))
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.arith
>>> n = Integer(6) # positive integer
>>> k = Integer(4) # positive integer
>>> R = InfinitePolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> PR = PolynomialRing(QQ, 'x', n)
>>> d = {x[i]: PR.gen(i) for i in range(n)} #substitution dictionnary
>>> L = LazyPowerSeriesRing(R, names=('z',)); (z,) = L._first_ngens(1)
>>> f = L(lambda i: x[i-Integer(1)]/factorial(i), valuation=Integer(1))
>>> all(exp(f)[i].subs(d) * factorial(i) == bell_polynomial(i) for i in␣
↪range(n+Integer(1)))
True
>>> all((f**k/factorial(k))[i].subs(d) * factorial(i) == bell_polynomial(i, k)␣
↪for i in range(k, n+k))
True
>>> g = L(lambda i: x[i-Integer(1)], valuation=Integer(1))
>>> all((Integer(1)/(Integer(1)-g))[i].subs(d) == bell_polynomial(i,␣
↪ordinary=True) for i in range(n+Integer(1)))
True
>>> all((g**k)[i].subs(d) == bell_polynomial(i, k, True) for i in range(k, n+k))
True
```

REFERENCES:

- [Bel1927]
- [Com1974]

AUTHORS:

- Blair Sutton (2009-01-26)
- Thierry Monteil (2015-09-29): the result must always be a polynomial.
- Kei Beauduin (2024-04-06): when univariate, the polynomial is in variable `x0`. extended to complete exponential, partial ordinary and complete ordinary Bell polynomials.

`sage.combinat.combinat.`**`bernoulli_polynomial`**`(x, n)`

Return the `n`-th Bernoulli polynomial evaluated at `x`.

The generating function for the Bernoulli polynomials is

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \frac{t^n}{n!},$$

and they are given directly by

$$B_n(x) = \sum_{i=0}^{n} \binom{n}{i} B_{n-i} x^i.$$

One has $B_n(x) = -n\zeta(1-n, x)$, where $\zeta(s, x)$ is the Hurwitz zeta function. Thus, in a certain sense, the Hurwitz zeta function generalizes the Bernoulli polynomials to non-integer values of n.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: y = QQ['y'].0
sage: bernoulli_polynomial(y, 5)
y^5 - 5/2*y^4 + 5/3*y^3 - 1/6*y
sage: bernoulli_polynomial(y, 5)(12)
199870
sage: bernoulli_polynomial(12, 5)
199870
sage: bernoulli_polynomial(y^2 + 1, 5)
y^10 + 5/2*y^8 + 5/3*y^6 - 1/6*y^2
sage: P.<t> = ZZ[]
sage: p = bernoulli_polynomial(t, 6)
sage: p.parent()
Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> # needs sage.libs.flint
>>> y = QQ['y'].gen(0)
>>> bernoulli_polynomial(y, Integer(5))
y^5 - 5/2*y^4 + 5/3*y^3 - 1/6*y
>>> bernoulli_polynomial(y, Integer(5))(Integer(12))
199870
>>> bernoulli_polynomial(Integer(12), Integer(5))
199870
>>> bernoulli_polynomial(y**Integer(2) + Integer(1), Integer(5))
y^10 + 5/2*y^8 + 5/3*y^6 - 1/6*y^2
>>> P = ZZ['t']; (t,) = P._first_ngens(1)
>>> p = bernoulli_polynomial(t, Integer(6))
>>> p.parent()
Univariate Polynomial Ring in t over Rational Field
```

We verify an instance of the formula which is the origin of the Bernoulli polynomials (and numbers):

```
sage: power_sum = sum(k^4 for k in range(10))
sage: 5*power_sum == bernoulli_polynomial(10, 5) - bernoulli(5)                 #␣
↪needs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> power_sum = sum(k**Integer(4) for k in range(Integer(10)))
>>> Integer(5)*power_sum == bernoulli_polynomial(Integer(10), Integer(5)) -␣
↪bernoulli(Integer(5))                          # needs sage.libs.flint
True
```

REFERENCES:

- Wikipedia article Bernoulli_polynomials

`sage.combinat.combinat.`**`catalan_number`**`(n)`

Return the n-th Catalan number.

The n-th Catalan number is given directly in terms of binomial coefficients by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} \qquad \text{for} \quad n \geq 0.$$

Consider the set $S = \{1, ..., n\}$. A noncrossing partition of S is a partition in which no two blocks "cross" each other, i.e., if a and b belong to one block and x and y to another, they are not arranged in the order $axby$. C_n is the number of noncrossing partitions of the set S. There are many other interpretations (see REFERENCES).

When $n = -1$, this function returns the limit value $-1/2$. For other $n < 0$ it returns 0.

INPUT:

- n – integer

OUTPUT: integer

EXAMPLES:

```
sage: [catalan_number(i) for i in range(7)]
[1, 1, 2, 5, 14, 42, 132]
sage: x = (QQ[['x']].0).O(8)
sage: (-1/2)*sqrt(1 - 4*x)
-1/2 + x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + 42*x^6 + 132*x^7 + O(x^8)
sage: [catalan_number(i) for i in range(-7,7)]
[0, 0, 0, 0, 0, 0, -1/2, 1, 1, 2, 5, 14, 42, 132]
sage: [catalan_number(n).mod(2) for n in range(16)]
[1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
```

```
>>> from sage.all import *
>>> [catalan_number(i) for i in range(Integer(7))]
[1, 1, 2, 5, 14, 42, 132]
>>> x = (QQ[['x']].gen(0)).O(Integer(8))
>>> (-Integer(1)/Integer(2))*sqrt(Integer(1) - Integer(4)*x)
-1/2 + x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + 42*x^6 + 132*x^7 + O(x^8)
>>> [catalan_number(i) for i in range(-Integer(7),Integer(7))]
[0, 0, 0, 0, 0, 0, -1/2, 1, 1, 2, 5, 14, 42, 132]
>>> [catalan_number(n).mod(Integer(2)) for n in range(Integer(16))]
[1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
```

REFERENCES:

- Wikipedia article Catalan_number
- http://www-history.mcs.st-andrews.ac.uk/~history/Miscellaneous/CatalanNumbers/catalan.html

sage.combinat.combinat.**euler_number**(*n*, *algorithm='flint'*)

Return the n-th Euler number.

INPUT:

- n – positive integer
- algorithm – (default: 'flint') any one of the following:
  - 'maxima' – wraps Maxima's euler
  - 'flint' – wrap FLINT's arith_euler_number

EXAMPLES:

```
sage: [euler_number(i) for i in range(10)]                              #␣
↪needs sage.libs.flint
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
sage: x = PowerSeriesRing(QQ, 'x').gen().O(10)
```

(continues on next page)

(continued from previous page)

```
sage: 2/(exp(x)+exp(-x))
1 - 1/2*x^2 + 5/24*x^4 - 61/720*x^6 + 277/8064*x^8 + O(x^10)
sage: [euler_number(i)/factorial(i) for i in range(11)]                    #␣
↪needs sage.libs.flint
[1, 0, -1/2, 0, 5/24, 0, -61/720, 0, 277/8064, 0, -50521/3628800]
sage: euler_number(-1)
Traceback (most recent call last):
...
ValueError: n (=-1) must be a nonnegative integer
```

```
>>> from sage.all import *
>>> [euler_number(i) for i in range(Integer(10))]                          ␣
↪     # needs sage.libs.flint
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
>>> x = PowerSeriesRing(QQ, 'x').gen().O(Integer(10))
>>> Integer(2)/(exp(x)+exp(-x))
1 - 1/2*x^2 + 5/24*x^4 - 61/720*x^6 + 277/8064*x^8 + O(x^10)
>>> [euler_number(i)/factorial(i) for i in range(Integer(11))]             ␣
↪     # needs sage.libs.flint
[1, 0, -1/2, 0, 5/24, 0, -61/720, 0, 277/8064, 0, -50521/3628800]
>>> euler_number(-Integer(1))
Traceback (most recent call last):
...
ValueError: n (=-1) must be a nonnegative integer
```

REFERENCES:

- Wikipedia article Euler_number

`sage.combinat.combinat.`**`eulerian_number`**`(`*k*, *algorithm='recursive'*`)`

Return the Eulerian number of index `(n, k)`.

This is the coefficient of t^k in the Eulerian polynomial $A_n(t)$.

INPUT:

- `n` – integer
- `k` – integer between `0` and `n - 1`
- `algorithm` – `'recursive'` (default) or `'formula'`

OUTPUT: integer

See also

`eulerian_polynomial()`

EXAMPLES:

```
sage: from sage.combinat.combinat import eulerian_number
sage: [eulerian_number(5,i) for i in range(5)]
[1, 26, 66, 26, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat import eulerian_number
>>> [eulerian_number(Integer(5),i) for i in range(Integer(5))]
[1, 26, 66, 26, 1]
```

sage.combinat.combinat.**eulerian_polynomial**(*algorithm='derivative'*)

Return the Eulerian polynomial of index `n`.

This is the generating polynomial counting permutations in the symmetric group S_n according to their number of descents.

INPUT:

- `n` – integer
- `algorithm` – `'derivative'` (default) or `'coeffs'`

OUTPUT: polynomial in one variable `t`

See also

eulerian_number()

EXAMPLES:

```
sage: from sage.combinat.combinat import eulerian_polynomial
sage: eulerian_polynomial(5)
t^4 + 26*t^3 + 66*t^2 + 26*t + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat import eulerian_polynomial
>>> eulerian_polynomial(Integer(5))
t^4 + 26*t^3 + 66*t^2 + 26*t + 1
```

REFERENCES:

- Wikipedia article Eulerian_number

sage.combinat.combinat.**fibonacci**(*n*, *algorithm='pari'*)

Return the n-th Fibonacci number.

The Fibonacci sequence F_n is defined by the initial conditions $F_1 = F_2 = 1$ and the recurrence relation $F_{n+2} = F_{n+1} + F_n$. For negative n we define $F_n = (-1)^{n+1} F_{-n}$, which is consistent with the recurrence relation.

INPUT:

- `algorithm` – string; one of
  - `'pari'` – (default) use the PARI C library's pari:fibo function
  - `'gap'` – use GAP's Fibonacci function

Note

PARI is tens to hundreds of times faster than GAP here. Moreover, PARI works for every large input whereas GAP does not.

EXAMPLES:

```
sage: fibonacci(10)                                                          #␣
↪needs sage.libs.pari
55
sage: fibonacci(10, algorithm='gap')                                         #␣
↪needs sage.libs.gap
55
```

```
>>> from sage.all import *
>>> fibonacci(Integer(10))                                                   ␣
↪      # needs sage.libs.pari
55
>>> fibonacci(Integer(10), algorithm='gap')                                  ␣
↪      # needs sage.libs.gap
55
```

```
sage: fibonacci(-100)                                                        #␣
↪needs sage.libs.pari
-354224848179261915075
sage: fibonacci(100)                                                         #␣
↪needs sage.libs.pari
354224848179261915075
```

```
>>> from sage.all import *
>>> fibonacci(-Integer(100))                                                 ␣
↪      # needs sage.libs.pari
-354224848179261915075
>>> fibonacci(Integer(100))                                                  ␣
↪      # needs sage.libs.pari
354224848179261915075
```

```
sage: fibonacci(0)                                                           #␣
↪needs sage.libs.pari
0
sage: fibonacci(1/2)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

```
>>> from sage.all import *
>>> fibonacci(Integer(0))                                                    ␣
↪      # needs sage.libs.pari
0
>>> fibonacci(Integer(1)/Integer(2))
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

`sage.combinat.combinat.`**`fibonacci_sequence`**(*start*, *stop=None*, *algorithm=None*)

Return an iterator over the Fibonacci sequence, for all fibonacci numbers f_n from `n = start` up to (but not including) `n = stop`

INPUT:

- `start` – starting value
- `stop` – stopping value
- `algorithm` – (default: `None`) passed on to fibonacci function (or not passed on if `None`, i.e., use the default)

EXAMPLES:

```
sage: fibs = [i for i in fibonacci_sequence(10, 20)]; fibs                    #␣
↪needs sage.libs.pari
[55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

```
>>> from sage.all import *
>>> fibs = [i for i in fibonacci_sequence(Integer(10), Integer(20))]; fibs    ␣
↪                # needs sage.libs.pari
[55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

```
sage: sum([i for i in fibonacci_sequence(100, 110)])                          #␣
↪needs sage.libs.pari
69919376923075308730013
```

```
>>> from sage.all import *
>>> sum([i for i in fibonacci_sequence(Integer(100), Integer(110))])         ␣
↪                # needs sage.libs.pari
69919376923075308730013
```

See also

fibonacci_xrange()

AUTHORS:

- Bobby Moretti

`sage.combinat.combinat.`**`fibonacci_xrange`**(*start*, *stop=None*, *algorithm='pari'*)

Return an iterator over all of the Fibonacci numbers in the given range, including `f_n = start` up to, but not including, `f_n = stop`.

EXAMPLES:

```
sage: fibs_in_some_range = [i for i in fibonacci_xrange(10^7, 10^8)]          #␣
↪needs sage.libs.pari
sage: len(fibs_in_some_range)                                                 #␣
↪needs sage.libs.pari
4
sage: fibs_in_some_range                                                      #␣
↪needs sage.libs.pari
[14930352, 24157817, 39088169, 63245986]
```

```
>>> from sage.all import *
>>> fibs_in_some_range = [i for i in fibonacci_xrange(Integer(10)**Integer(7),␣
↪Integer(10)**Integer(8))]              # needs sage.libs.pari
```

(continues on next page)

(continued from previous page)

```
>>> len(fibs_in_some_range)                                                  #␣
↪needs sage.libs.pari
4
>>> fibs_in_some_range                                                       #␣
↪needs sage.libs.pari
[14930352, 24157817, 39088169, 63245986]
```

```
sage: fibs = [i for i in fibonacci_xrange(10, 100)]; fibs                    #␣
↪needs sage.libs.pari
[13, 21, 34, 55, 89]
```

```
>>> from sage.all import *
>>> fibs = [i for i in fibonacci_xrange(Integer(10), Integer(100))]; fibs    ␣
↪                # needs sage.libs.pari
[13, 21, 34, 55, 89]
```

```
sage: list(fibonacci_xrange(13, 34))                                         #␣
↪needs sage.libs.pari
[13, 21]
```

```
>>> from sage.all import *
>>> list(fibonacci_xrange(Integer(13), Integer(34)))                         ␣
↪                # needs sage.libs.pari
[13, 21]
```

A solution to the second Project Euler problem:

```
sage: sum([i for i in fibonacci_xrange(10^6) if is_even(i)])                 #␣
↪needs sage.libs.pari
1089154
```

```
>>> from sage.all import *
>>> sum([i for i in fibonacci_xrange(Integer(10)**Integer(6)) if is_even(i)])    ␣
↪                 # needs sage.libs.pari
1089154
```

See also

fibonacci_sequence()

AUTHORS:

- Bobby Moretti

sage.combinat.combinat.**lucas_number1**(n, P, Q)

Return the n-th Lucas number "of the first kind" (this is not standard terminology). The Lucas sequence $L^{(1)}_n$ is defined by the initial conditions $L^{(1)}_1 = 0$, $L^{(1)}_2 = 1$ and the recurrence relation $L^{(1)}_{n+2} = P \cdot L^{(1)}_{n+1} - Q \cdot L^{(1)}_n$.

Wraps GAP's `Lucas(...)[1]`.

$P = 1$, $Q = -1$ gives the Fibonacci sequence.

INPUT:

- `n` – integer
- `P`, `Q` – integer or rational numbers

OUTPUT: integer or rational number

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: lucas_number1(5,1,-1)
5
sage: lucas_number1(6,1,-1)
8
sage: lucas_number1(7,1,-1)
13
sage: lucas_number1(7,1,-2)
43
sage: lucas_number1(5,2,3/5)
229/25
sage: lucas_number1(5,2,1.5)
1/4
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> lucas_number1(Integer(5),Integer(1),-Integer(1))
5
>>> lucas_number1(Integer(6),Integer(1),-Integer(1))
8
>>> lucas_number1(Integer(7),Integer(1),-Integer(1))
13
>>> lucas_number1(Integer(7),Integer(1),-Integer(2))
43
>>> lucas_number1(Integer(5),Integer(2),Integer(3)/Integer(5))
229/25
>>> lucas_number1(Integer(5),Integer(2),RealNumber('1.5'))
1/4
```

There was a conjecture that the sequence L_n defined by $L_{n+2} = L_{n+1} + L_n$, $L_1 = 1$, $L_2 = 3$, has the property that n prime implies that L_n is prime.

```
sage: def lucas(n):
....:     return Integer((5/2)*lucas_number1(n,1,-1) + (1/2)*lucas_number2(n,1,-
↪1))
sage: [[lucas(n), is_prime(lucas(n)), n+1, is_prime(n+1)] for n in range(15)]    #␣
↪needs sage.libs.gap
[[1, False, 1, False],
 [3, True, 2, True],
 [4, False, 3, True],
 [7, True, 4, False],
 [11, True, 5, True],
 [18, False, 6, False],
 [29, True, 7, True],
 [47, True, 8, False],
```

(continues on next page)

(continued from previous page)

```
 [76, False, 9, False],
 [123, False, 10, False],
 [199, True, 11, True],
 [322, False, 12, False],
 [521, True, 13, True],
 [843, False, 14, False],
 [1364, False, 15, False]]
```

```
>>> from sage.all import *
>>> def lucas(n):
...     return Integer((Integer(5)/Integer(2))*lucas_number1(n,Integer(1),-
↪Integer(1)) + (Integer(1)/Integer(2))*lucas_number2(n,Integer(1),-Integer(1)))
>>> [[lucas(n), is_prime(lucas(n)), n+Integer(1), is_prime(n+Integer(1))] for n␣
↪in range(Integer(15))]    # needs sage.libs.gap
[[1, False, 1, False],
 [3, True, 2, True],
 [4, False, 3, True],
 [7, True, 4, False],
 [11, True, 5, True],
 [18, False, 6, False],
 [29, True, 7, True],
 [47, True, 8, False],
 [76, False, 9, False],
 [123, False, 10, False],
 [199, True, 11, True],
 [322, False, 12, False],
 [521, True, 13, True],
 [843, False, 14, False],
 [1364, False, 15, False]]
```

Can you use Sage to find a counterexample to the conjecture?

sage.combinat.combinat.**lucas_number2**(n, P, Q)

Return the n-th Lucas number "of the second kind" (this is not standard terminology). The Lucas sequence $L^{(2)}_n$ is defined by the initial conditions $L^{(2)}_1 = 2$, $L^{(2)}_2 = P$ and the recurrence relation $L^{(2)}_{n+2} = P \cdot L^{(2)}_{n+1} - Q \cdot L^{(2)}_n$.

Wraps GAP's Lucas(…)[2].

INPUT:

- n – integer
- P, Q – integer or rational numbers

OUTPUT: integer or rational number

EXAMPLES:

```
sage: [lucas_number2(i,1,-1) for i in range(10)]                              #␣
↪needs sage.libs.gap
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
sage: [fibonacci(i-1)+fibonacci(i+1) for i in range(10)]                       #␣
↪needs sage.libs.pari
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

```
>>> from sage.all import *
>>> [lucas_number2(i,Integer(1),-Integer(1)) for i in range(Integer(10))]                      ␣
↪                           # needs sage.libs.gap
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
>>> [fibonacci(i-Integer(1))+fibonacci(i+Integer(1)) for i in range(Integer(10))]␣
↪                           # needs sage.libs.pari
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

```
sage: # needs sage.libs.gap
sage: n = lucas_number2(5,2,3); n
2
sage: type(n)
<class 'sage.rings.integer.Integer'>
sage: n = lucas_number2(5,2,-3/9); n
418/9
sage: type(n)
<class 'sage.rings.rational.Rational'>
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> n = lucas_number2(Integer(5),Integer(2),Integer(3)); n
2
>>> type(n)
<class 'sage.rings.integer.Integer'>
>>> n = lucas_number2(Integer(5),Integer(2),-Integer(3)/Integer(9)); n
418/9
>>> type(n)
<class 'sage.rings.rational.Rational'>
```

The case $P = 1$, $Q = -1$ is the Lucas sequence in Brualdi's Introductory Combinatorics, 4th ed., Prentice-Hall, 2004:

```
sage: [lucas_number2(n,1,-1) for n in range(10)]                                     #␣
↪needs sage.libs.gap
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

```
>>> from sage.all import *
>>> [lucas_number2(n,Integer(1),-Integer(1)) for n in range(Integer(10))]                      ␣
↪                           # needs sage.libs.gap
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

sage.combinat.combinat.**narayana_number**(n, k)

Return the Narayana number of index `(n, k)`.

For every integer $n \geq 1$, the sum of Narayana numbers $\sum_k N_{n,k}$ is the Catalan number C_n.

INPUT:

- `n` – integer
- `k` – integer between `0` and `n - 1`

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.combinat import narayana_number
sage: [narayana_number(3, i) for i in range(3)]
[1, 3, 1]
sage: sum(narayana_number(7,i) for i in range(7)) == catalan_number(7)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat import narayana_number
>>> [narayana_number(Integer(3), i) for i in range(Integer(3))]
[1, 3, 1]
>>> sum(narayana_number(Integer(7),i) for i in range(Integer(7))) == catalan_
↪number(Integer(7))
True
```

REFERENCES:

- Wikipedia article Narayana_number

sage.combinat.combinat.**number_of_tuples**(*S*, *k*, *algorithm='naive'*)

Return the size of `tuples(S, k)` when S is a set. More generally, return the size of `tuples(set(S), k)`. (So, unlike `tuples()`, this method removes redundant entries from S.)

INPUT:

- `S` – the base set
- `k` – the length of the tuples
- `algorithm` – can be one of the following:
  - `'naive'` – (default) use the naive counting $|S|^k$
  - `'gap'` – wraps GAP's `NrTuples`

Warning

When using `algorithm='gap'`, `S` must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If `S` consists of at all complicated Sage objects, this function might *not* do what you expect.

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: number_of_tuples(S,2)
25
sage: number_of_tuples(S,2, algorithm='gap')                                #␣
↪needs sage.libs.gap
25
sage: S = [1,1,2,3,4,5]
sage: number_of_tuples(S,2)
25
sage: number_of_tuples(S,2, algorithm='gap')                                #␣
↪needs sage.libs.gap
25
sage: number_of_tuples(S,0)
```

(continues on next page)

(continued from previous page)

```
1
sage: number_of_tuples(S,0, algorithm='gap')                                   #␣
↪needs sage.libs.gap
1
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]
>>> number_of_tuples(S,Integer(2))
25
>>> number_of_tuples(S,Integer(2), algorithm='gap')                            ␣
↪     # needs sage.libs.gap
25
>>> S = [Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]
>>> number_of_tuples(S,Integer(2))
25
>>> number_of_tuples(S,Integer(2), algorithm='gap')                            ␣
↪     # needs sage.libs.gap
25
>>> number_of_tuples(S,Integer(0))
1
>>> number_of_tuples(S,Integer(0), algorithm='gap')                            ␣
↪     # needs sage.libs.gap
1
```

sage.combinat.combinat.**number_of_unordered_tuples**(*S*, *k*, *algorithm='naive'*)

Return the size of `unordered_tuples(S, k)` when S is a set.

INPUT:

- `S` – the base set
- `k` – the length of the tuples
- `algorithm` – can be one of the following:
  - `'naive'` – (default) use the naive counting $\binom{|S|+k-1}{k}$
  - `'gap'` – wraps GAP's `NrUnorderedTuples`

Warning

When using `algorithm='gap'`, `S` must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If `S` consists of at all complicated Sage objects, this function might *not* do what you expect.

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: number_of_unordered_tuples(S,2)
15
sage: number_of_unordered_tuples(S,2, algorithm='gap')                         #␣
↪needs sage.libs.gap
15
```

(continues on next page)

(continued from previous page)

```
sage: S = [1,1,2,3,4,5]
sage: number_of_unordered_tuples(S,2)
15
sage: number_of_unordered_tuples(S,2, algorithm='gap')                        #␣
↪needs sage.libs.gap
15
sage: number_of_unordered_tuples(S,0)
1
sage: number_of_unordered_tuples(S,0, algorithm='gap')                        #␣
↪needs sage.libs.gap
1
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]
>>> number_of_unordered_tuples(S,Integer(2))
15
>>> number_of_unordered_tuples(S,Integer(2), algorithm='gap')                 ␣
↪     # needs sage.libs.gap
15
>>> S = [Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]
>>> number_of_unordered_tuples(S,Integer(2))
15
>>> number_of_unordered_tuples(S,Integer(2), algorithm='gap')                 ␣
↪     # needs sage.libs.gap
15
>>> number_of_unordered_tuples(S,Integer(0))
1
>>> number_of_unordered_tuples(S,Integer(0), algorithm='gap')                 ␣
↪     # needs sage.libs.gap
1
```

sage.combinat.combinat.**polygonal_number**(*s*, *n*)

Return the n-th s-gonal number.

Polygonal sequences are represented by dots forming a regular polygon. Two famous sequences are the triangular numbers (3rd column of Pascal's Triangle) and the square numbers. The n-th term in a polygonal sequence is defined by

$$P(s,n) = \frac{n^2(s-2) - n(s-4)}{2},$$

where s is the number of sides of the polygon.

INPUT:

- `s` – integer greater than 1; the number of sides of the polygon
- `n` – integer; the index of the returned s-gonal number

OUTPUT: integer

EXAMPLES:

The triangular numbers:

```
sage: [polygonal_number(3, n) for n in range(10)]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]

sage: [polygonal_number(3, n) for n in range(-10, 0)]
[45, 36, 28, 21, 15, 10, 6, 3, 1, 0]
```

```
>>> from sage.all import *
>>> [polygonal_number(Integer(3), n) for n in range(Integer(10))]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]

>>> [polygonal_number(Integer(3), n) for n in range(-Integer(10), Integer(0))]
[45, 36, 28, 21, 15, 10, 6, 3, 1, 0]
```

The square numbers:

```
sage: [polygonal_number(4, n) for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> from sage.all import *
>>> [polygonal_number(Integer(4), n) for n in range(Integer(10))]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The pentagonal numbers:

```
sage: [polygonal_number(5, n) for n in range(10)]
[0, 1, 5, 12, 22, 35, 51, 70, 92, 117]
```

```
>>> from sage.all import *
>>> [polygonal_number(Integer(5), n) for n in range(Integer(10))]
[0, 1, 5, 12, 22, 35, 51, 70, 92, 117]
```

The hexagonal numbers:

```
sage: [polygonal_number(6, n) for n in range(10)]
[0, 1, 6, 15, 28, 45, 66, 91, 120, 153]
```

```
>>> from sage.all import *
>>> [polygonal_number(Integer(6), n) for n in range(Integer(10))]
[0, 1, 6, 15, 28, 45, 66, 91, 120, 153]
```

The input is converted into an integer:

```
sage: polygonal_number(3.0, 2.0)
3
```

```
>>> from sage.all import *
>>> polygonal_number(RealNumber('3.0'), RealNumber('2.0'))
3
```

A non-integer input returns an error:

```
sage: polygonal_number(3.5, 1)                                                  #␣
↪needs sage.rings.real_mpfr
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

```
>>> from sage.all import *
>>> polygonal_number(RealNumber('3.5'), Integer(1))                              ␣
↪                     # needs sage.rings.real_mpfr
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

s must be greater than 1:

```
sage: polygonal_number(1, 4)
Traceback (most recent call last):
...
ValueError: s (=1) must be greater than 1
```

```
>>> from sage.all import *
>>> polygonal_number(Integer(1), Integer(4))
Traceback (most recent call last):
...
ValueError: s (=1) must be greater than 1
```

REFERENCES:

- Wikipedia article Polygonal_number

sage.combinat.combinat.**stirling_number1**(*n*, *k*, *algorithm='gap'*)

Return the n-th Stirling number $S_1(n, k)$ of the first kind.

This is the number of permutations of n points with k cycles.

See Wikipedia article Stirling_numbers_of_the_first_kind.

INPUT:

- `n` – nonnegative machine-size integer
- `k` – nonnegative machine-size integer
- `algorithm`:
  - `'gap'` – default; use GAP's `Stirling1` function
  - `'flint'` – use flint's `arith_stirling_number_1u` function

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: stirling_number1(3,2)
3
sage: stirling_number1(5,2)
50
sage: 9*stirling_number1(9,5) + stirling_number1(9,4)
269325
```

(continues on next page)

(continued from previous page)

```
sage: stirling_number1(10,5)
269325
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> stirling_number1(Integer(3),Integer(2))
3
>>> stirling_number1(Integer(5),Integer(2))
50
>>> Integer(9)*stirling_number1(Integer(9),Integer(5)) + stirling_
↪number1(Integer(9),Integer(4))
269325
>>> stirling_number1(Integer(10),Integer(5))
269325
```

Indeed, $S_1(n,k) = S_1(n-1,k-1) + (n-1)S_1(n-1,k)$.

`sage.combinat.combinat.`**`stirling_number2`**`(`*n*, *k*, *algorithm=None*`)`

Return the n-th Stirling number $S_2(n,k)$ of the second kind.

This is the number of ways to partition a set of n elements into k pairwise disjoint nonempty subsets. The n-th Bell number is the sum of the $S_2(n,k)$'s, $k = 0,...,n$.

See Wikipedia article Stirling_numbers_of_the_second_kind.

INPUT:

- `n` – nonnegative machine-size integer
- `k` – nonnegative machine-size integer
- `algorithm`:
  - `None` – default; use native implementation
  - `'flint'` – use flint's `arith_stirling_number_2` function
  - `'gap'` – use GAP's `Stirling2` function
  - `'maxima'` – use Maxima's `stirling2` function

EXAMPLES:

Print a table of the first several Stirling numbers of the second kind:

```
sage: for n in range(10):
....:     for k in range(10):
....:         print(str(stirling_number2(n,k)).rjust(k and 6))
1      0      0      0      0      0      0      0      0      0
0      1      0      0      0      0      0      0      0      0
0      1      1      0      0      0      0      0      0      0
0      1      3      1      0      0      0      0      0      0
0      1      7      6      1      0      0      0      0      0
0      1     15     25     10      1      0      0      0      0
0      1     31     90     65     15      1      0      0      0
0      1     63    301    350    140     21      1      0      0
0      1    127    966   1701   1050    266     28      1      0
0      1    255   3025   7770   6951   2646    462     36      1
```

```
>>> from sage.all import *
>>> for n in range(Integer(10)):
...     for k in range(Integer(10)):
...         print(str(stirling_number2(n,k)).rjust(k and Integer(6)))
1      0      0      0      0      0      0      0      0      0
0      1      0      0      0      0      0      0      0      0
0      1      1      0      0      0      0      0      0      0
0      1      3      1      0      0      0      0      0      0
0      1      7      6      1      0      0      0      0      0
0      1     15     25     10      1      0      0      0      0
0      1     31     90     65     15      1      0      0      0
0      1     63    301    350    140     21      1      0      0
0      1    127    966   1701   1050    266     28      1      0
0      1    255   3025   7770   6951   2646    462     36      1
```

Stirling numbers satisfy $S_2(n,k) = S_2(n-1,k-1) + kS_2(n-1,k)$:

```
sage: 5*stirling_number2(9,5) + stirling_number2(9,4)
42525
sage: stirling_number2(10,5)
42525
```

```
>>> from sage.all import *
>>> Integer(5)*stirling_number2(Integer(9),Integer(5)) + stirling_
→number2(Integer(9),Integer(4))
42525
>>> stirling_number2(Integer(10),Integer(5))
42525
```

sage.combinat.combinat.**tuples**(*S*, *k*, *algorithm='itertools'*)

Return a list of all k-tuples of elements of a given set `S`.

This function accepts the set `S` in the form of any iterable (list, tuple or iterator), and returns a list of k-tuples. If `S` contains duplicate entries, then you should expect the method to return tuples multiple times!

Recall that k-tuples are ordered (in the sense that two k-tuples differing in the order of their entries count as different) and can have repeated entries (even if `S` is a list with no repetition).

INPUT:

- `S` – the base set
- `k` – the length of the tuples
- `algorithm` – can be one of the following:
  - `'itertools'` – (default) use python's itertools
  - `'native'` – use a native Sage implementation

Note

The ordering of the list of tuples depends on the algorithm.

EXAMPLES:

```
sage: S = [1,2]
sage: tuples(S,3)
[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2),
 (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
sage: mset = ["s","t","e","i","n"]
sage: tuples(mset, 2)
[('s', 's'), ('s', 't'), ('s', 'e'), ('s', 'i'), ('s', 'n'),
 ('t', 's'), ('t', 't'), ('t', 'e'), ('t', 'i'), ('t', 'n'),
 ('e', 's'), ('e', 't'), ('e', 'e'), ('e', 'i'), ('e', 'n'),
 ('i', 's'), ('i', 't'), ('i', 'e'), ('i', 'i'), ('i', 'n'),
 ('n', 's'), ('n', 't'), ('n', 'e'), ('n', 'i'), ('n', 'n')]
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2)]
>>> tuples(S,Integer(3))
[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2),
 (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
>>> mset = ["s","t","e","i","n"]
>>> tuples(mset, Integer(2))
[('s', 's'), ('s', 't'), ('s', 'e'), ('s', 'i'), ('s', 'n'),
 ('t', 's'), ('t', 't'), ('t', 'e'), ('t', 'i'), ('t', 'n'),
 ('e', 's'), ('e', 't'), ('e', 'e'), ('e', 'i'), ('e', 'n'),
 ('i', 's'), ('i', 't'), ('i', 'e'), ('i', 'i'), ('i', 'n'),
 ('n', 's'), ('n', 't'), ('n', 'e'), ('n', 'i'), ('n', 'n')]
```

```
sage: K.<a> = GF(4, 'a')                                                    #␣
↪needs sage.rings.finite_rings
sage: mset = [x for x in K if x != 0]                                       #␣
↪needs sage.rings.finite_rings
sage: tuples(mset, 2)                                                       #␣
↪needs sage.rings.finite_rings
[(a, a), (a, a + 1), (a, 1), (a + 1, a), (a + 1, a + 1),
 (a + 1, 1), (1, a), (1, a + 1), (1, 1)]
```

```
>>> from sage.all import *
>>> K = GF(Integer(4), 'a', names=('a',)); (a,) = K._first_ngens(1)# needs sage.
↪rings.finite_rings
>>> mset = [x for x in K if x != Integer(0)]                                 ␣
↪     # needs sage.rings.finite_rings
>>> tuples(mset, Integer(2))                                                 ␣
↪     # needs sage.rings.finite_rings
[(a, a), (a, a + 1), (a, 1), (a + 1, a), (a + 1, a + 1),
 (a + 1, 1), (1, a), (1, a + 1), (1, 1)]
```

We check that the implementations agree (up to ordering):

```
sage: tuples(S, 3, 'native')
[(1, 1, 1), (2, 1, 1), (1, 2, 1), (2, 2, 1),
 (1, 1, 2), (2, 1, 2), (1, 2, 2), (2, 2, 2)]
```

```
>>> from sage.all import *
>>> tuples(S, Integer(3), 'native')
```

(continues on next page)

(continued from previous page)

```
[(1, 1, 1), (2, 1, 1), (1, 2, 1), (2, 2, 1),
 (1, 1, 2), (2, 1, 2), (1, 2, 2), (2, 2, 2)]
```

Lastly we check on a multiset:

```
sage: S = [1,1,2]
sage: sorted(tuples(S, 3)) == sorted(tuples(S, 3, 'native'))
True
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(1),Integer(2)]
>>> sorted(tuples(S, Integer(3))) == sorted(tuples(S, Integer(3), 'native'))
True
```

AUTHORS:

- Jon Hanke (2006-08)

`sage.combinat.combinat.`**`unordered_tuples`**`(`*S*, *k*, *algorithm='itertools'*`)`

Return a list of all unordered tuples of length `k` of the set `S`.

An unordered tuple of length k of set S is a unordered selection with repetitions of S and is represented by a sorted list of length k containing elements from S.

Unlike *`tuples()`*, the result of this method does not depend on how often an element appears in S; only the *set* S is being used. For example, `unordered_tuples([1, 1, 1], 2)` will return `[(1, 1)]`. If you want it to return `[(1, 1), (1, 1), (1, 1)]`, use Python's `itertools.combinations_with_replacement` instead.

INPUT:

- `S` – the base set
- `k` – the length of the tuples
- `algorithm` – can be one of the following:
  - `'itertools'` – (default) use python's itertools
  - `'gap'` – wraps GAP's `UnorderedTuples`

> **Warning**
>
> When using `algorithm='gap'`, `S` must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If `S` consists of at all complicated Sage objects, this function might *not* do what you expect.

EXAMPLES:

```
sage: S = [1,2]
sage: unordered_tuples(S, 3)
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2)]
```

(continues on next page)

(continued from previous page)

```
>>> unordered_tuples(S, Integer(3))
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

We check that this agrees with GAP:

```
sage: unordered_tuples(S, 3, algorithm='gap')                              #␣
↪needs sage.libs.gap
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

```
>>> from sage.all import *
>>> unordered_tuples(S, Integer(3), algorithm='gap')                        ␣
↪     # needs sage.libs.gap
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

We check the result on strings:

```
sage: S = ["a","b","c"]
sage: unordered_tuples(S, 2)
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
sage: unordered_tuples(S, 2, algorithm='gap')                              #␣
↪needs sage.libs.gap
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
```

```
>>> from sage.all import *
>>> S = ["a","b","c"]
>>> unordered_tuples(S, Integer(2))
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
>>> unordered_tuples(S, Integer(2), algorithm='gap')                        ␣
↪     # needs sage.libs.gap
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
```

Lastly we check on a multiset:

```
sage: S = [1,1,2]
sage: unordered_tuples(S, 3) == unordered_tuples(S, 3, 'gap')              #␣
↪needs sage.libs.gap
True
sage: unordered_tuples(S, 3)
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(1),Integer(2)]
>>> unordered_tuples(S, Integer(3)) == unordered_tuples(S, Integer(3), 'gap')    ␣
↪             # needs sage.libs.gap
True
>>> unordered_tuples(S, Integer(3))
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

sage.combinat.combinat.**unshuffle_iterator**(*a*, *one=1*)

Iterate over the unshuffles of a list (or tuple) a, also yielding the signs of the respective permutations.

If n and k are integers satisfying $0 \leq k \leq n$, then a $(k, n-k)$-*unshuffle* means a permutation $\pi \in S_n$ such that $\pi(1) < \pi(2) < \cdots < \pi(k)$ and $\pi(k+1) < \pi(k+2) < \cdots < \pi(n)$. This method provides, for a list

$a = (a_1, a_2, \ldots, a_n)$ of length n, an iterator yielding all pairs:

$$\Big(\big((a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(k)}), (a_{\pi(k+1)}, a_{\pi(k+2)}, \ldots, a_{\pi(n)}) \big), (-1)^{\pi} \Big)$$

for all $k \in \{0, 1, \ldots, n\}$ and all $(k, n-k)$-unshuffles π. The optional variable `one` can be set to a different value which results in the $(-1)^{\pi}$ component being multiplied by said value.

The iterator does not yield these in order of increasing k.

EXAMPLES:

```
sage: from sage.combinat.combinat import unshuffle_iterator
sage: list(unshuffle_iterator([1, 3, 4]))
[(((), (1, 3, 4)), 1), (((1,), (3, 4)), 1), (((3,), (1, 4)), -1),
 (((1, 3), (4,)), 1), (((4,), (1, 3)), 1), (((1, 4), (3,)), -1),
 (((3, 4), (1,)), 1), (((1, 3, 4), ()), 1)]
sage: list(unshuffle_iterator([3, 1]))
[(((), (3, 1)), 1), (((3,), (1,)), 1), (((1,), (3,)), -1),
 (((3, 1), ()), 1)]
sage: list(unshuffle_iterator([8]))
[(((), (8,)), 1), (((8,), ()), 1)]
sage: list(unshuffle_iterator([]))
[(((), ()), 1)]
sage: list(unshuffle_iterator([3, 1], 3/2))
[(((), (3, 1)), 3/2), (((3,), (1,)), 3/2), (((1,), (3,)), -3/2),
 (((3, 1), ()), 3/2)]
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat import unshuffle_iterator
>>> list(unshuffle_iterator([Integer(1), Integer(3), Integer(4)]))
[(((), (1, 3, 4)), 1), (((1,), (3, 4)), 1), (((3,), (1, 4)), -1),
 (((1, 3), (4,)), 1), (((4,), (1, 3)), 1), (((1, 4), (3,)), -1),
 (((3, 4), (1,)), 1), (((1, 3, 4), ()), 1)]
>>> list(unshuffle_iterator([Integer(3), Integer(1)]))
[(((), (3, 1)), 1), (((3,), (1,)), 1), (((1,), (3,)), -1),
 (((3, 1), ()), 1)]
>>> list(unshuffle_iterator([Integer(8)]))
[(((), (8,)), 1), (((8,), ()), 1)]
>>> list(unshuffle_iterator([]))
[(((), ()), 1)]
>>> list(unshuffle_iterator([Integer(3), Integer(1)], Integer(3)/Integer(2)))
[(((), (3, 1)), 3/2), (((3,), (1,)), 3/2), (((1,), (3,)), -3/2),
 (((3, 1), ()), 3/2)]
```

5.1.26 Fast computation of combinatorial functions (Cython + mpz)

Currently implemented:

- Stirling numbers of the second kind
- iterators for set partitions
- iterator for Lyndon words
- iterator for perfect matchings
- conjugate of partitions

AUTHORS:

- Fredrik Johansson (2010-10): Stirling numbers of second kind
- Martin Rubey and Travis Scrimshaw (2018): iterators for set partitions, Lyndon words, and perfect matchings

`sage.combinat.combinat_cython.`**`conjugate`**`(p)`

Return the conjugate partition associated to the partition `p` as a list.

EXAMPLES:

```
sage: from sage.combinat.combinat_cython import conjugate
sage: conjugate([2,2])
[2, 2]
sage: conjugate([6,3,1])
[3, 2, 2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat_cython import conjugate
>>> conjugate([Integer(2),Integer(2)])
[2, 2]
>>> conjugate([Integer(6),Integer(3),Integer(1)])
[3, 2, 2, 1, 1, 1]
```

`sage.combinat.combinat_cython.`**`lyndon_word_iterator`**`(n, k)`

Generate the Lyndon words of fixed length `k` with `n` letters.

The resulting Lyndon words will be words represented as lists whose alphabet is `range(n)` $(= \{0, 1, \ldots, n-1\})$.

ALGORITHM:

The iterative FKM Algorithm 7.2 from [Rus2003].

EXAMPLES:

```
sage: from sage.combinat.combinat_cython import lyndon_word_iterator
sage: list(lyndon_word_iterator(4, 2))
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
sage: list(lyndon_word_iterator(2, 4))
[[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat_cython import lyndon_word_iterator
>>> list(lyndon_word_iterator(Integer(4), Integer(2)))
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
>>> list(lyndon_word_iterator(Integer(2), Integer(4)))
[[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1]]
```

`sage.combinat.combinat_cython.`**`perfect_matchings_iterator`**`(n)`

Iterate over all perfect matchings with `n` parts.

This iterates over all perfect matchings of $\{0, 1, \ldots, 2n-1\}$ using a Gray code for fixed-point-free involutions due to Walsh [Wal2001].

EXAMPLES:

```
sage: from sage.combinat.combinat_cython import perfect_matchings_iterator
sage: list(perfect_matchings_iterator(1))
[[(0, 1)]]
sage: list(perfect_matchings_iterator(2))
[[(0, 1), (2, 3)], [(0, 2), (1, 3)], [(0, 3), (1, 2)]]

sage: list(perfect_matchings_iterator(0))
[[]]
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat_cython import perfect_matchings_iterator
>>> list(perfect_matchings_iterator(Integer(1)))
[[(0, 1)]]
>>> list(perfect_matchings_iterator(Integer(2)))
[[(0, 1), (2, 3)], [(0, 2), (1, 3)], [(0, 3), (1, 2)]]

>>> list(perfect_matchings_iterator(Integer(0)))
[[]]
```

REFERENCES:

- [Wal2001]

sage.combinat.combinat_cython.**set_partition_composition**(*sp1*, *sp2*)

Return a tuple consisting of the composition of the set partitions `sp1` and `sp2` and the number of components removed from the middle rows of the graph.

EXAMPLES:

```
sage: from sage.combinat.combinat_cython import set_partition_composition
sage: sp1 = ((1,-2),(2,-1))
sage: sp2 = ((1,-2),(2,-1))
sage: p, c = set_partition_composition(sp1, sp2)
sage: (SetPartition(p), c) == (SetPartition([[1,-1],[2,-2]]), 0)           #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> from sage.combinat.combinat_cython import set_partition_composition
>>> sp1 = ((Integer(1),-Integer(2)),(Integer(2),-Integer(1)))
>>> sp2 = ((Integer(1),-Integer(2)),(Integer(2),-Integer(1)))
>>> p, c = set_partition_composition(sp1, sp2)
>>> (SetPartition(p), c) == (SetPartition([[Integer(1),-Integer(1)],[Integer(2),-
↪Integer(2)]]), Integer(0))               # needs sage.combinat
True
```

5.1.27 Combinations

AUTHORS:

- Mike Hansen (2007): initial implementation
- Vincent Delecroix (2011): cleaning, bug corrections, doctests
- Antoine Genitrini (2020) : new implementation of the lexicographic unranking of combinations

class sage.combinat.combination.**ChooseNK**(*mset*, *k*)

Bases: *Combinations_setk*

sage.combinat.combination.**Combinations**(*mset*, *k=None*)

Return the combinatorial class of combinations of the multiset mset. If k is specified, then it returns the combinatorial class of combinations of mset of size k.

A *combination* of a multiset M is an unordered selection of k objects of M, where every object can appear at most as many times as it appears in M.

The combinatorial classes correctly handle the cases where mset has duplicate elements.

EXAMPLES:

```
sage: C = Combinations(range(4)); C
Combinations of [0, 1, 2, 3]
sage: C.list()
[[],
 [0],
 [1],
 [2],
 [3],
 [0, 1],
 [0, 2],
 [0, 3],
 [1, 2],
 [1, 3],
 [2, 3],
 [0, 1, 2],
 [0, 1, 3],
 [0, 2, 3],
 [1, 2, 3],
 [0, 1, 2, 3]]
 sage: C.cardinality()
 16
```

```
>>> from sage.all import *
>>> C = Combinations(range(Integer(4))); C
Combinations of [0, 1, 2, 3]
>>> C.list()
[[],
 [0],
 [1],
 [2],
 [3],
 [0, 1],
 [0, 2],
 [0, 3],
 [1, 2],
 [1, 3],
 [2, 3],
 [0, 1, 2],
 [0, 1, 3],
 [0, 2, 3],
 [1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
 [0, 1, 2, 3]]
>>> C.cardinality()
16
```

```
sage: C2 = Combinations(range(4),2); C2
Combinations of [0, 1, 2, 3] of length 2
sage: C2.list()
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
sage: C2.cardinality()
6
```

```
>>> from sage.all import *
>>> C2 = Combinations(range(Integer(4)),Integer(2)); C2
Combinations of [0, 1, 2, 3] of length 2
>>> C2.list()
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
>>> C2.cardinality()
6
```

```
sage: Combinations([1,2,2,3]).list()
[[],
 [1],
 [2],
 [3],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
 [1, 2, 2],
 [1, 2, 3],
 [2, 2, 3],
 [1, 2, 2, 3]]
```

```
>>> from sage.all import *
>>> Combinations([Integer(1),Integer(2),Integer(2),Integer(3)]).list()
[[],
 [1],
 [2],
 [3],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
 [1, 2, 2],
 [1, 2, 3],
 [2, 2, 3],
 [1, 2, 2, 3]]
```

```
sage: Combinations([1,2,3], 2).list()
[[1, 2], [1, 3], [2, 3]]
```

```
>>> from sage.all import *
>>> Combinations([Integer(1),Integer(2),Integer(3)], Integer(2)).list()
[[1, 2], [1, 3], [2, 3]]
```

```
sage: mset = [1,1,2,3,4,4,5]
sage: Combinations(mset,2).list()
[[1, 1],
 [1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [2, 3],
 [2, 4],
 [2, 5],
 [3, 4],
 [3, 5],
 [4, 4],
 [4, 5]]
```

```
>>> from sage.all import *
>>> mset = [Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(4),
↪Integer(5)]
>>> Combinations(mset,Integer(2)).list()
[[1, 1],
 [1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [2, 3],
 [2, 4],
 [2, 5],
 [3, 4],
 [3, 5],
 [4, 4],
 [4, 5]]
```

```
sage: mset = ["d","a","v","i","d"]
sage: Combinations(mset,3).list()
[['d', 'd', 'a'],
 ['d', 'd', 'v'],
 ['d', 'd', 'i'],
 ['d', 'a', 'v'],
 ['d', 'a', 'i'],
 ['d', 'v', 'i'],
 ['a', 'v', 'i']]
```

```
>>> from sage.all import *
>>> mset = ["d","a","v","i","d"]
>>> Combinations(mset,Integer(3)).list()
[['d', 'd', 'a'],
 ['d', 'd', 'v'],
 ['d', 'd', 'i'],
```

(continues on next page)

(continued from previous page)

```
 ['d', 'a', 'v'],
 ['d', 'a', 'i'],
 ['d', 'v', 'i'],
 ['a', 'v', 'i']]
```

```
sage: X = Combinations([1,2,3,4,5],3)
sage: [x for x in X]
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 4],
 [1, 3, 5],
 [1, 4, 5],
 [2, 3, 4],
 [2, 3, 5],
 [2, 4, 5],
 [3, 4, 5]]
```

```
>>> from sage.all import *
>>> X = Combinations([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],
↪Integer(3))
>>> [x for x in X]
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 4],
 [1, 3, 5],
 [1, 4, 5],
 [2, 3, 4],
 [2, 3, 5],
 [2, 4, 5],
 [3, 4, 5]]
```

It is possible to take combinations of Sage objects:

```
sage: Combinations([vector([1,1]), vector([2,2]), vector([3,3])], 2).list()      #␣
↪needs sage.modules
[[(1, 1), (2, 2)], [(1, 1), (3, 3)], [(2, 2), (3, 3)]]
```

```
>>> from sage.all import *
>>> Combinations([vector([Integer(1),Integer(1)]), vector([Integer(2),
↪Integer(2)]), vector([Integer(3),Integer(3)])], Integer(2)).list()      # needs␣
↪sage.modules
[[(1, 1), (2, 2)], [(1, 1), (3, 3)], [(2, 2), (3, 3)]]
```

class sage.combinat.combination.**Combinations_mset**(*mset*)

Bases: Parent

cardinality()

class sage.combinat.combination.**Combinations_msetk**(*mset*, *k*)

Bases: Parent

cardinality()

Return the size of combinations(mset, k).

IMPLEMENTATION: Wraps GAP's NrCombinations.

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: Combinations(mset,2).cardinality()                                    #␣
↪needs sage.libs.gap
12
```

```
>>> from sage.all import *
>>> mset = [Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(4),
↪Integer(5)]
>>> Combinations(mset,Integer(2)).cardinality()                              ␣
↪     # needs sage.libs.gap
12
```

class sage.combinat.combination.**Combinations_set**(*mset*)

Bases: *Combinations_mset*

cardinality()

Return the size of Combinations(set).

EXAMPLES:

```
sage: Combinations(range(16000)).cardinality() == 2^16000
True
```

```
>>> from sage.all import *
>>> Combinations(range(Integer(16000))).cardinality() ==␣
↪Integer(2)**Integer(16000)
True
```

rank(*x*)

EXAMPLES:

```
sage: c = Combinations([1,2,3])
sage: list(range(c.cardinality())) == list(map(c.rank, c))
True
```

```
>>> from sage.all import *
>>> c = Combinations([Integer(1),Integer(2),Integer(3)])
>>> list(range(c.cardinality())) == list(map(c.rank, c))
True
```

unrank(*r*)

EXAMPLES:

```
sage: c = Combinations([1,2,3])
sage: c.list() == list(map(c.unrank, range(c.cardinality())))
True
```

```
>>> from sage.all import *
>>> c = Combinations([Integer(1),Integer(2),Integer(3)])
>>> c.list() == list(map(c.unrank, range(c.cardinality())))
True
```

class sage.combinat.combination.**Combinations_setk**(*mset*, *k*)

Bases: *Combinations_msetk*

cardinality()

Return the size of combinations(set, k).

EXAMPLES:

```
sage: Combinations(range(16000), 5).cardinality()
8732673194560003200
```

```
>>> from sage.all import *
>>> Combinations(range(Integer(16000)), Integer(5)).cardinality()
8732673194560003200
```

list()

EXAMPLES:

```
sage: Combinations([1,2,3,4,5],3).list()
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 4],
 [1, 3, 5],
 [1, 4, 5],
 [2, 3, 4],
 [2, 3, 5],
 [2, 4, 5],
 [3, 4, 5]]
```

```
>>> from sage.all import *
>>> Combinations([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],
↪Integer(3)).list()
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 4],
 [1, 3, 5],
 [1, 4, 5],
 [2, 3, 4],
 [2, 3, 5],
 [2, 4, 5],
 [3, 4, 5]]
```

rank(*x*)

EXAMPLES:

```
sage: c = Combinations([1,2,3], 2)
sage: list(range(c.cardinality())) == list(map(c.rank, c.list()))
True
```

```
>>> from sage.all import *
>>> c = Combinations([Integer(1),Integer(2),Integer(3)], Integer(2))
>>> list(range(c.cardinality())) == list(map(c.rank, c.list()))
True
```

unrank(*r*)

EXAMPLES:

```
sage: c = Combinations([1,2,3], 2)
sage: c.list() == list(map(c.unrank, range(c.cardinality())))
True
```

```
>>> from sage.all import *
>>> c = Combinations([Integer(1),Integer(2),Integer(3)], Integer(2))
>>> c.list() == list(map(c.unrank, range(c.cardinality())))
True
```

sage.combinat.combination.**from_rank**(*r*, *n*, *k*)

Return the combination of rank `r` in the subsets of `range(n)` of size `k` when listed in lexicographic order.

The algorithm used is based on factoradics and presented in [DGH2020]. It is there compared to the other from the literature.

EXAMPLES:

```
sage: import sage.combinat.combination as combination
sage: combination.from_rank(0,3,0)
()
sage: combination.from_rank(0,3,1)
(0,)
sage: combination.from_rank(1,3,1)
(1,)
sage: combination.from_rank(2,3,1)
(2,)
sage: combination.from_rank(0,3,2)
(0, 1)
sage: combination.from_rank(1,3,2)
(0, 2)
sage: combination.from_rank(2,3,2)
(1, 2)
sage: combination.from_rank(0,3,3)
(0, 1, 2)
```

```
>>> from sage.all import *
>>> import sage.combinat.combination as combination
>>> combination.from_rank(Integer(0),Integer(3),Integer(0))
()
>>> combination.from_rank(Integer(0),Integer(3),Integer(1))
(0,)
```

(continues on next page)

(continued from previous page)

```
>>> combination.from_rank(Integer(1),Integer(3),Integer(1))
(1,)
>>> combination.from_rank(Integer(2),Integer(3),Integer(1))
(2,)
>>> combination.from_rank(Integer(0),Integer(3),Integer(2))
(0, 1)
>>> combination.from_rank(Integer(1),Integer(3),Integer(2))
(0, 2)
>>> combination.from_rank(Integer(2),Integer(3),Integer(2))
(1, 2)
>>> combination.from_rank(Integer(0),Integer(3),Integer(3))
(0, 1, 2)
```

sage.combinat.combination.**rank**(*comb*, *n*, *check=True*)

Return the rank of `comb` in the subsets of `range(n)` of size `k` where `k` is the length of `comb`.

The algorithm used is based on combinadics and James McCaffrey's MSDN article. See: Wikipedia article Combinadic.

EXAMPLES:

```
sage: import sage.combinat.combination as combination
sage: combination.rank((), 3)
0
sage: combination.rank((0,), 3)
0
sage: combination.rank((1,), 3)
1
sage: combination.rank((2,), 3)
2
sage: combination.rank((0,1), 3)
0
sage: combination.rank((0,2), 3)
1
sage: combination.rank((1,2), 3)
2
sage: combination.rank((0,1,2), 3)
0

sage: combination.rank((0,1,2,3), 3)
Traceback (most recent call last):
...
ValueError: len(comb) must be <= n
sage: combination.rank((0,0), 2)
Traceback (most recent call last):
...
ValueError: comb must be a subword of (0,1,...,n)

sage: combination.rank([1,2], 3)
2
sage: combination.rank([0,1,2], 3)
0
```

```
>>> from sage.all import *
>>> import sage.combinat.combination as combination
>>> combination.rank((), Integer(3))
0
>>> combination.rank((Integer(0),), Integer(3))
0
>>> combination.rank((Integer(1),), Integer(3))
1
>>> combination.rank((Integer(2),), Integer(3))
2
>>> combination.rank((Integer(0),Integer(1)), Integer(3))
0
>>> combination.rank((Integer(0),Integer(2)), Integer(3))
1
>>> combination.rank((Integer(1),Integer(2)), Integer(3))
2
>>> combination.rank((Integer(0),Integer(1),Integer(2)), Integer(3))
0

>>> combination.rank((Integer(0),Integer(1),Integer(2),Integer(3)), Integer(3))
Traceback (most recent call last):
...
ValueError: len(comb) must be <= n
>>> combination.rank((Integer(0),Integer(0)), Integer(2))
Traceback (most recent call last):
...
ValueError: comb must be a subword of (0,1,...,n)

>>> combination.rank([Integer(1),Integer(2)], Integer(3))
2
>>> combination.rank([Integer(0),Integer(1),Integer(2)], Integer(3))
0
```

5.1.28 Combinatorial maps

This module provides a decorator that can be used to add semantic to a Python method by marking it as implementing a *combinatorial map*, that is a map between two enumerated sets:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map
sage: class MyPermutation():
....:     @combinatorial_map()
....:     def reverse(self):
....:         '''
....:         Reverse the permutation
....:         '''
....:         # ... code ...
```

```
>>> from sage.all import *
>>> from sage.combinat.combinatorial_map import combinatorial_map
>>> class MyPermutation():
...     @combinatorial_map()
...     def reverse(self):
```

(continues on next page)

(continued from previous page)

```
...         '''
...         Reverse the permutation
...         '''
...         # ... code ...
```

By default, this decorator is a no-op: it returns the decorated method as is:

```
sage: MyPermutation.reverse
<function MyPermutation.reverse at ...>
```

```
>>> from sage.all import *
>>> MyPermutation.reverse
<function MyPermutation.reverse at ...>
```

See `combinatorial_map_wrapper()` for the various options this decorator can take.

Projects built on top of Sage are welcome to customize locally this hook to instrument the Sage code and exploit this semantic information. Typically, the decorator could be used to populate a database of maps. For a real-life application, see the project $FindStat < http : //findstat.org/ >$. As a basic example, a variant of the decorator is provided as `combinatorial_map_wrapper()`; it wraps the decorated method, so that one can later use `combinatorial_maps_in_class()` to query an object, or class thereof, for all the combinatorial maps that apply to it.

Note

Since decorators are evaluated upon loading Python modules, customizing `combinatorial map` needs to be done before the modules using it are loaded. In the examples below, where we illustrate the customized `combinatorial_map` decorator on the `sage.combinat.permutation` module, we resort to force a reload of this module after dynamically changing `sage.combinat.combinatorial_map.combinatorial_map`. This is good enough for those doctests, but remains fragile.

For real use cases, it is probably best to just edit this source file statically (see below).

class `sage.combinat.combinatorial_map.`**`CombinatorialMap`**(*f*, *order=None*, *name=None*)

Bases: `object`

This is a wrapper class for methods that are *combinatorial maps*.

For further details and doctests, see *Combinatorial maps* and `combinatorial_map_wrapper()`.

name()

Return the name of a combinatorial map. This is used for the string representation of `self`.

EXAMPLES:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map
sage: class CombinatorialClass:
....:     @combinatorial_map(name='map1')
....:     def to_self_1(): pass
....:     @combinatorial_map()
....:     def to_self_2(): pass
sage: CombinatorialClass.to_self_1.name()
'map1'
sage: CombinatorialClass.to_self_2.name()
'to_self_2'
```

```
>>> from sage.all import *
>>> from sage.combinat.combinatorial_map import combinatorial_map
>>> class CombinatorialClass:
...     @combinatorial_map(name='map1')
...     def to_self_1(): pass
...     @combinatorial_map()
...     def to_self_2(): pass
>>> CombinatorialClass.to_self_1.name()
'map1'
>>> CombinatorialClass.to_self_2.name()
'to_self_2'
```

order()

Return the order of `self`, or `None` if the order is not known.

EXAMPLES:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map
sage: class CombinatorialClass:
....:     @combinatorial_map(order=2)
....:     def to_self_1(): pass
....:     @combinatorial_map()
....:     def to_self_2(): pass
sage: CombinatorialClass.to_self_1.order()
2
sage: CombinatorialClass.to_self_2.order() is None
True
```

```
>>> from sage.all import *
>>> from sage.combinat.combinatorial_map import combinatorial_map
>>> class CombinatorialClass:
...     @combinatorial_map(order=Integer(2))
...     def to_self_1(): pass
...     @combinatorial_map()
...     def to_self_2(): pass
>>> CombinatorialClass.to_self_1.order()
2
>>> CombinatorialClass.to_self_2.order() is None
True
```

unbounded_map()

Return the unbounded version of `self`.

You can use this method to return a function which takes as input an element in the domain of the combinatorial map. See the example below.

EXAMPLES:

```
sage: sage.combinat.combinatorial_map.combinatorial_map = sage.combinat.
↪combinatorial_map.combinatorial_map_wrapper
sage: from importlib import reload
sage: _ = reload(sage.combinat.permutation)
sage: from sage.combinat.permutation import Permutation
sage: pi = Permutation([1,3,2])
```

(continues on next page)

(continued from previous page)

```
sage: f = pi.reverse
sage: F = f.unbounded_map()
sage: F(pi)
[2, 3, 1]
```

```
>>> from sage.all import *
>>> sage.combinat.combinatorial_map.combinatorial_map = sage.combinat.
↪combinatorial_map.combinatorial_map_wrapper
>>> from importlib import reload
>>> _ = reload(sage.combinat.permutation)
>>> from sage.combinat.permutation import Permutation
>>> pi = Permutation([Integer(1),Integer(3),Integer(2)])
>>> f = pi.reverse
>>> F = f.unbounded_map()
>>> F(pi)
[2, 3, 1]
```

sage.combinat.combinatorial_map.**combinatorial_map**(*f=None*, *order=None*, *name=None*)

Combinatorial map decorator.

See *Combinatorial maps* for a description of this decorator and its purpose. This default implementation does nothing.

INPUT:

- f – (default: None, if combinatorial_map is used as a decorator) a function
- name – (default: None) the name for nicer outputs on combinatorial maps
- order – (default: None) the order of the combinatorial map, if it is known. Is not used, but might be helpful later

OUTPUT: f unchanged

EXAMPLES:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map_trivial as␣
↪combinatorial_map
sage: class MyPermutation():
....:     @combinatorial_map
....:     def reverse(self):
....:         '''
....:         Reverse the permutation
....:         '''
....:         # ... code ...
....:     @combinatorial_map(name='descent set of permutation')
....:     def descent_set(self):
....:         '''
....:         The descent set of the permutation
....:         '''
....:         # ... code ...

sage: MyPermutation.reverse
<function MyPermutation.reverse at ...>
```

(continues on next page)

(continued from previous page)

```
sage: MyPermutation.descent_set
<function MyPermutation.descent_set at ...>
```

```
>>> from sage.all import *
>>> from sage.combinat.combinatorial_map import combinatorial_map_trivial as␣
↪combinatorial_map
>>> class MyPermutation():
...     @combinatorial_map
...     def reverse(self):
...         '''
...         Reverse the permutation
...         '''
...         # ... code ...
...     @combinatorial_map(name='descent set of permutation')
...     def descent_set(self):
...         '''
...         The descent set of the permutation
...         '''
...         # ... code ...

>>> MyPermutation.reverse
<function MyPermutation.reverse at ...>

>>> MyPermutation.descent_set
<function MyPermutation.descent_set at ...>
```

sage.combinat.combinatorial_map.**combinatorial_map_trivial**(*f=None*, *order=None*, *name=None*)

Combinatorial map decorator.

See *Combinatorial maps* for a description of this decorator and its purpose. This default implementation does nothing.

INPUT:

- f – (default: None, if combinatorial_map is used as a decorator) a function
- name – (default: None) the name for nicer outputs on combinatorial maps
- order – (default: None) the order of the combinatorial map, if it is known. Is not used, but might be helpful later

OUTPUT: f unchanged

EXAMPLES:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map_trivial as␣
↪combinatorial_map
sage: class MyPermutation():
....:     @combinatorial_map
....:     def reverse(self):
....:         '''
....:         Reverse the permutation
....:         '''
....:         # ... code ...
....:     @combinatorial_map(name='descent set of permutation')
```

(continues on next page)

(continued from previous page)

```
....:     def descent_set(self):
....:         '''
....:         The descent set of the permutation
....:         '''
....:         # ... code ...

sage: MyPermutation.reverse
<function MyPermutation.reverse at ...>

sage: MyPermutation.descent_set
<function MyPermutation.descent_set at ...>
```

```
>>> from sage.all import *
>>> from sage.combinat.combinatorial_map import combinatorial_map_trivial as␣
↪combinatorial_map
>>> class MyPermutation():
...     @combinatorial_map
...     def reverse(self):
...         '''
...         Reverse the permutation
...         '''
...         # ... code ...
...     @combinatorial_map(name='descent set of permutation')
...     def descent_set(self):
...         '''
...         The descent set of the permutation
...         '''
...         # ... code ...

>>> MyPermutation.reverse
<function MyPermutation.reverse at ...>

>>> MyPermutation.descent_set
<function MyPermutation.descent_set at ...>
```

sage.combinat.combinatorial_map.**combinatorial_map_wrapper**(*f=None*, *order=None*, *name=None*)

Combinatorial map decorator (basic example).

See *Combinatorial maps* for a description of the `combinatorial_map` decorator and its purpose. This implementation, together with `combinatorial_maps_in_class()` illustrates how to use this decorator as a hook to instrument the Sage code.

INPUT:

- `f` – (default: `None`, if combinatorial_map is used as a decorator) a function
- `name` – (default: `None`) the name for nicer outputs on combinatorial maps
- `order` – (default: `None`) the order of the combinatorial map, if it is known. Is not used, but might be helpful later

OUTPUT: a combinatorial map; this is an instance of the `CombinatorialMap`

EXAMPLES:

We define a class illustrating the use of this implementation of the `combinatorial_map` decorator with its various arguments:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map_wrapper as␣
↪combinatorial_map
sage: class MyPermutation():
....:     @combinatorial_map()
....:     def reverse(self):
....:         '''
....:         Reverse the permutation
....:         '''
....:         pass
....:     @combinatorial_map(order=2)
....:     def inverse(self):
....:         '''
....:         The inverse of the permutation
....:         '''
....:         pass
....:     @combinatorial_map(name='descent set of permutation')
....:     def descent_set(self):
....:         '''
....:         The descent set of the permutation
....:         '''
....:         pass
....:     def major_index(self):
....:         '''
....:         The major index of the permutation
....:         '''
....:         pass
sage: MyPermutation.reverse
Combinatorial map: reverse
sage: MyPermutation.descent_set
Combinatorial map: descent set of permutation
sage: MyPermutation.inverse
Combinatorial map: inverse
```

```
>>> from sage.all import *
>>> from sage.combinat.combinatorial_map import combinatorial_map_wrapper as␣
↪combinatorial_map
>>> class MyPermutation():
...     @combinatorial_map()
...     def reverse(self):
...         '''
...         Reverse the permutation
...         '''
...         pass
...     @combinatorial_map(order=Integer(2))
...     def inverse(self):
...         '''
...         The inverse of the permutation
...         '''
...         pass
...     @combinatorial_map(name='descent set of permutation')
```

(continues on next page)

(continued from previous page)

```
...     def descent_set(self):
...         '''
...         The descent set of the permutation
...         '''
...         pass
...     def major_index(self):
...         '''
...         The major index of the permutation
...         '''
...         pass
>>> MyPermutation.reverse
Combinatorial map: reverse
>>> MyPermutation.descent_set
Combinatorial map: descent set of permutation
>>> MyPermutation.inverse
Combinatorial map: inverse
```

One can now determine all the combinatorial maps associated with a given object as follows:

```
sage: from sage.combinat.combinatorial_map import combinatorial_maps_in_class
sage: X = combinatorial_maps_in_class(MyPermutation); X # random
[Combinatorial map: reverse,
 Combinatorial map: descent set of permutation,
 Combinatorial map: inverse]
```

```
>>> from sage.all import *
>>> from sage.combinat.combinatorial_map import combinatorial_maps_in_class
>>> X = combinatorial_maps_in_class(MyPermutation); X # random
[Combinatorial map: reverse,
 Combinatorial map: descent set of permutation,
 Combinatorial map: inverse]
```

The method `major_index` defined about is not a combinatorial map:

```
sage: MyPermutation.major_index
<function MyPermutation.major_index at ...>
```

```
>>> from sage.all import *
>>> MyPermutation.major_index
<function MyPermutation.major_index at ...>
```

But one can define a function that turns `major_index` into a combinatorial map:

```
sage: def major_index(p):
....:     return p.major_index()
sage: major_index
<function major_index at ...>
sage: combinatorial_map(major_index)
Combinatorial map: major_index
```

```
>>> from sage.all import *
>>> def major_index(p):
```

(continues on next page)

(continued from previous page)

```
...     return p.major_index()
>>> major_index
<function major_index at ...>
>>> combinatorial_map(major_index)
Combinatorial map: major_index
```

sage.combinat.combinatorial_map.**combinatorial_maps_in_class**(*cls*)

Return the combinatorial maps of the class as a list of combinatorial maps.

For further details and doctests, see *Combinatorial maps* and `combinatorial_map_wrapper()`.

EXAMPLES:

```
sage: sage.combinat.combinatorial_map.combinatorial_map = sage.combinat.
↪combinatorial_map.combinatorial_map_wrapper
sage: from importlib import reload
sage: _ = reload(sage.combinat.permutation)
sage: from sage.combinat.combinatorial_map import combinatorial_maps_in_class
sage: p = Permutation([1,3,2,4])
sage: cmaps = combinatorial_maps_in_class(p)
sage: cmaps # random
[Combinatorial map: Robinson-Schensted insertion tableau,
 Combinatorial map: Robinson-Schensted recording tableau,
 Combinatorial map: Robinson-Schensted tableau shape,
 Combinatorial map: complement,
 Combinatorial map: descent composition,
 Combinatorial map: inverse, ...]
sage: p.left_tableau in cmaps
True
sage: p.right_tableau in cmaps
True
sage: p.complement in cmaps
True
```

```
>>> from sage.all import *
>>> sage.combinat.combinatorial_map.combinatorial_map = sage.combinat.
↪combinatorial_map.combinatorial_map_wrapper
>>> from importlib import reload
>>> _ = reload(sage.combinat.permutation)
>>> from sage.combinat.combinatorial_map import combinatorial_maps_in_class
>>> p = Permutation([Integer(1),Integer(3),Integer(2),Integer(4)])
>>> cmaps = combinatorial_maps_in_class(p)
>>> cmaps # random
[Combinatorial map: Robinson-Schensted insertion tableau,
 Combinatorial map: Robinson-Schensted recording tableau,
 Combinatorial map: Robinson-Schensted tableau shape,
 Combinatorial map: complement,
 Combinatorial map: descent composition,
 Combinatorial map: inverse, ...]
>>> p.left_tableau in cmaps
True
>>> p.right_tableau in cmaps
True
```

(continues on next page)

(continued from previous page)

```
>>> p.complement in cmaps
True
```

5.1.29 Integer compositions

A composition c of a nonnegative integer n is a list of positive integers (the *parts* of the composition) with total sum n.

This module provides tools for manipulating compositions and enumerated sets of compositions.

EXAMPLES:

```
sage: Composition([5, 3, 1, 3])
[5, 3, 1, 3]
sage: list(Compositions(4))
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
```

```
>>> from sage.all import *
>>> Composition([Integer(5), Integer(3), Integer(1), Integer(3)])
[5, 3, 1, 3]
>>> list(Compositions(Integer(4)))
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
```

AUTHORS:

- Mike Hansen, Nicolas M. Thiéry
- MuPAD-Combinat developers (algorithms and design inspiration)
- Travis Scrimshaw (2013-02-03): Removed `CombinatorialClass`

class sage.combinat.composition.**Composition**(*parent*, *lst*)

Bases: `CombinatorialElement`

Integer compositions.

A composition of a nonnegative integer n is a list $(i_1, \ldots, i_k)$ of positive integers with total sum n.

EXAMPLES:

The simplest way to create a composition is by specifying its entries as a list, tuple (or other iterable):

```
sage: Composition([3,1,2])
[3, 1, 2]
sage: Composition((3,1,2))
[3, 1, 2]
sage: Composition(i for i in range(2,5))
[2, 3, 4]
```

```
>>> from sage.all import *
>>> Composition([Integer(3),Integer(1),Integer(2)])
[3, 1, 2]
>>> Composition((Integer(3),Integer(1),Integer(2)))
[3, 1, 2]
>>> Composition(i for i in range(Integer(2),Integer(5)))
[2, 3, 4]
```

You can also create a composition from its code. The *code* of a composition $(i_1, i_2, \ldots, i_k)$ of n is a list of length n that consists of a 1 followed by $i_1 - 1$ zeros, then a 1 followed by $i_2 - 1$ zeros, and so on.

```
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Composition(code=_)
[4, 1, 2, 3, 5]
sage: Composition([3,1,2,3,5]).to_code()
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Composition(code=_)
[3, 1, 2, 3, 5]
```

```
>>> from sage.all import *
>>> Composition([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).to_
↪code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
>>> Composition(code=_)
[4, 1, 2, 3, 5]
>>> Composition([Integer(3),Integer(1),Integer(2),Integer(3),Integer(5)]).to_
↪code()
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
>>> Composition(code=_)
[3, 1, 2, 3, 5]
```

You can also create the composition of n corresponding to a subset of $\{1, 2, \ldots, n-1\}$ under the bijection that maps the composition $(i_1, i_2, \ldots, i_k)$ of n to the subset $\{i_1, i_1 + i_2, i_1 + i_2 + i_3, \ldots, i_1 + \cdots + i_{k-1}\}$ (see `to_subset()`):

```
sage: Composition(from_subset=({1, 2, 4}, 5))
[1, 1, 2, 1]
sage: Composition([1, 1, 2, 1]).to_subset()
{1, 2, 4}
```

```
>>> from sage.all import *
>>> Composition(from_subset=({Integer(1), Integer(2), Integer(4)}, Integer(5)))
[1, 1, 2, 1]
>>> Composition([Integer(1), Integer(1), Integer(2), Integer(1)]).to_subset()
{1, 2, 4}
```

The following notation equivalently specifies the composition from the set $\{i_1 - 1, i_1 + i_2 - 1, i_1 + i_2 + i_3 - 1, \ldots, i_1 + \cdots + i_{k-1} - 1, n - 1\}$ or $\{i_1 - 1, i_1 + i_2 - 1, i_1 + i_2 + i_3 - 1, \ldots, i_1 + \cdots + i_{k-1} - 1\}$ and n. This provides compatibility with Python's 0-indexing.

```
sage: Composition(descents=[1,0,4,8,11])
[1, 1, 3, 4, 3]
sage: Composition(descents=[0,1,3,4])
[1, 1, 2, 1]
sage: Composition(descents=([0,1,3],5))
[1, 1, 2, 1]
sage: Composition(descents=({0,1,3},5))
[1, 1, 2, 1]
```

```
>>> from sage.all import *
>>> Composition(descents=[Integer(1),Integer(0),Integer(4),Integer(8),
↪Integer(11)])
[1, 1, 3, 4, 3]
>>> Composition(descents=[Integer(0),Integer(1),Integer(3),Integer(4)])
[1, 1, 2, 1]
>>> Composition(descents=([Integer(0),Integer(1),Integer(3)],Integer(5)))
[1, 1, 2, 1]
>>> Composition(descents=({Integer(0),Integer(1),Integer(3)},Integer(5)))
[1, 1, 2, 1]
```

An integer composition may be regarded as a sequence. Thus it is an instance of the Python abstract base class `Sequence` allows us to check if objects behave "like" sequences based on implemented methods. Note that `collections.abc.Sequence` is not the same as `sage.structure.sequence.Sequence`:

```
sage: import collections.abc
sage: C = Composition([3,2,3])
sage: isinstance(C, collections.abc.Sequence)
True
sage: issubclass(C.__class__, collections.abc.Sequence)
True
```

```
>>> from sage.all import *
>>> import collections.abc
>>> C = Composition([Integer(3),Integer(2),Integer(3)])
>>> isinstance(C, collections.abc.Sequence)
True
>>> issubclass(C.__class__, collections.abc.Sequence)
True
```

Typically, instances of `collections.abc.Sequence` have a `.count` method. `Composition.count` counts the number of parts of a specified size:

```
sage: C.count(3)
2
```

```
>>> from sage.all import *
>>> C.count(Integer(3))
2
```

EXAMPLES:

```
sage: C = Composition([3,1,2])
sage: TestSuite(C).run()
```

```
>>> from sage.all import *
>>> C = Composition([Integer(3),Integer(1),Integer(2)])
>>> TestSuite(C).run()
```

complement()

Return the complement of the composition `self`.

The complement of a composition I is defined as follows:

If I is the empty composition, then the complement is the empty composition as well. Otherwise, let S be the descent set of I (that is, the subset $\{i_1, i_1 + i_2, \ldots, i_1 + i_2 + \cdots + i_{k-1}\}$ of $\{1, 2, \ldots, |I| - 1\}$, where I is written as $(i_1, i_2, \ldots, i_k)$). Then, the complement of I is defined as the composition of size $|I|$ whose descent set is $\{1, 2, \ldots, |I| - 1\} \setminus S$.

The complement of a composition I also is the reverse composition (`reversed()`) of the conjugate (`conjugate()`) of I.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).conjugate()
[1, 1, 3, 3, 1, 3]
sage: Composition([1, 1, 3, 1, 2, 1, 3]).complement()
[3, 1, 3, 3, 1, 1]
```

```
>>> from sage.all import *
>>> Composition([Integer(1), Integer(1), Integer(3), Integer(1), Integer(2),␣
→Integer(1), Integer(3)]).conjugate()
[1, 1, 3, 3, 1, 3]
>>> Composition([Integer(1), Integer(1), Integer(3), Integer(1), Integer(2),␣
→Integer(1), Integer(3)]).complement()
[3, 1, 3, 3, 1, 1]
```

conjugate()

Return the conjugate of the composition `self`.

The conjugate of a composition I is defined as the complement (see `complement()`) of the reverse composition (see `reversed()`) of I.

An equivalent definition of the conjugate goes by saying that the ribbon shape of the conjugate of a composition I is the conjugate of the ribbon shape of I. (The ribbon shape of a composition is returned by `to_skew_partition()`.)

This implementation uses the algorithm from mupad-combinat.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).conjugate()
[1, 1, 3, 3, 1, 3]
```

```
>>> from sage.all import *
>>> Composition([Integer(1), Integer(1), Integer(3), Integer(1), Integer(2),␣
→Integer(1), Integer(3)]).conjugate()
[1, 1, 3, 3, 1, 3]
```

The ribbon shape of the conjugate of I is the conjugate of the ribbon shape of I:

```
sage: all( I.conjugate().to_skew_partition()                              #␣
→needs sage.combinat
....:      == I.to_skew_partition().conjugate()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( I.conjugate().to_skew_partition()                                #␣
→needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
...       == I.to_skew_partition().conjugate()
...       for I in Compositions(Integer(4)) )
True
```

count (*n*)

Return the number of parts of size `n`.

EXAMPLES:

```
sage: C = Composition([3,2,3])
sage: C.count(3)
2
sage: C.count(2)
1
sage: C.count(1)
0
```

```
>>> from sage.all import *
>>> C = Composition([Integer(3),Integer(2),Integer(3)])
>>> C.count(Integer(3))
2
>>> C.count(Integer(2))
1
>>> C.count(Integer(1))
0
```

descents (*final_descent=False*)

This gives one fewer than the partial sums of the composition.

This is here to maintain some sort of backward compatibility, even through the original implementation was broken (it gave the wrong answer). The same information can be found in *partial_sums()*.

See also

partial_sums()

INPUT:

- `final_descent` – boolean (default: `False`)

OUTPUT:

- the list of partial sums of `self` with each part decremented by 1. This includes the sum of all entries when `final_descent` is `True`.

EXAMPLES:

```
sage: c = Composition([2,1,3,2])
sage: c.descents()
[1, 2, 5]
sage: c.descents(final_descent=True)
[1, 2, 5, 7]
```

```
>>> from sage.all import *
>>> c = Composition([Integer(2),Integer(1),Integer(3),Integer(2)])
>>> c.descents()
[1, 2, 5]
>>> c.descents(final_descent=True)
[1, 2, 5, 7]
```

fatten(*grouping*)

Return the composition fatter than `self`, obtained by grouping together consecutive parts according to `grouping`.

INPUT:

- `grouping` – a composition whose sum is the length of `self`

EXAMPLES:

Let us start with the composition:

```
sage: c = Composition([4,5,2,7,1])
```

```
>>> from sage.all import *
>>> c = Composition([Integer(4),Integer(5),Integer(2),Integer(7),Integer(1)])
```

With `grouping` equal to $(1, \dots, 1)$, c is left unchanged:

```
sage: c.fatten(Composition([1,1,1,1,1]))
[4, 5, 2, 7, 1]
```

```
>>> from sage.all import *
>>> c.fatten(Composition([Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1)]))
[4, 5, 2, 7, 1]
```

With `grouping` equal to (ℓ) where ℓ is the length of c, this yields the coarsest composition above c:

```
sage: c.fatten(Composition([5]))
[19]
```

```
>>> from sage.all import *
>>> c.fatten(Composition([Integer(5)]))
[19]
```

Other values for `grouping` yield (all the) other compositions coarser than c:

```
sage: c.fatten(Composition([2,1,2]))
[9, 2, 8]
sage: c.fatten(Composition([3,1,1]))
[11, 7, 1]
```

```
>>> from sage.all import *
>>> c.fatten(Composition([Integer(2),Integer(1),Integer(2)]))
[9, 2, 8]
>>> c.fatten(Composition([Integer(3),Integer(1),Integer(1)]))
[11, 7, 1]
```

fatter()

Return the set of compositions which are fatter than `self`.

Complexity for generation: $O(|c|)$ memory, $O(|r|)$ time where $|c|$ is the size of `self` and r is the result.

EXAMPLES:

```
sage: C = Composition([4,5,2]).fatter()
sage: C.cardinality()
4
sage: list(C)
[[4, 5, 2], [4, 7], [9, 2], [11]]
```

```
>>> from sage.all import *
>>> C = Composition([Integer(4),Integer(5),Integer(2)]).fatter()
>>> C.cardinality()
4
>>> list(C)
[[4, 5, 2], [4, 7], [9, 2], [11]]
```

Some extreme cases:

```
sage: list(Composition([5]).fatter())
[[5]]
sage: list(Composition([]).fatter())
[[]]
sage: list(Composition([1,1,1,1]).fatter()) == list(Compositions(4))
True
```

```
>>> from sage.all import *
>>> list(Composition([Integer(5)]).fatter())
[[5]]
>>> list(Composition([]).fatter())
[[]]
>>> list(Composition([Integer(1),Integer(1),Integer(1),Integer(1)]).fatter())␣
↪== list(Compositions(Integer(4)))
True
```

finer()

Return the set of compositions which are finer than `self`.

EXAMPLES:

```
sage: C = Composition([3,2]).finer()
sage: C.cardinality()
8
sage: C.list()
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 2, 1, 1], [1, 2, 2], [2, 1, 1, 1], [2, 1,␣
↪2], [3, 1, 1], [3, 2]]

sage: Composition([]).finer()
{[]}
```

```
>>> from sage.all import *
>>> C = Composition([Integer(3),Integer(2)]).finer()
>>> C.cardinality()
8
>>> C.list()
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 2, 1, 1], [1, 2, 2], [2, 1, 1, 1], [2, 1,␣
↪2], [3, 1, 1], [3, 2]]

>>> Composition([]).finer()
{[]}
```

inf(*other*, *check=True*)

Return the meet of `self` with a composition `other` of the same size.

The meet of two compositions I and J of size n is the finest composition of n which is coarser than each of I and J. It can be described as the composition whose descent set is the intersection of the descent sets of I and J.

INPUT:

- `other` – composition of same size as `self`
- `check` – boolean (default: `True`); whether to check the input compositions for having the same size

OUTPUT: the meet of the compositions `self` and `other`

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).meet([4, 3, 2])
[4, 5]
sage: Composition([9, 6]).meet([1, 3, 6, 3, 2])
[15]
sage: Composition([9, 6]).meet([1, 3, 5, 1, 3, 2])
[9, 6]
sage: Composition([1, 1, 1, 1, 1]).meet([3, 2])
[3, 2]
sage: Composition([4, 2]).meet([3, 3])
[6]
sage: Composition([]).meet([])
[]
sage: Composition([1]).meet([1])
[1]
```

```
>>> from sage.all import *
>>> Composition([Integer(3), Integer(1), Integer(1), Integer(3), Integer(1)]).
↪meet([Integer(4), Integer(3), Integer(2)])
[4, 5]
>>> Composition([Integer(9), Integer(6)]).meet([Integer(1), Integer(3),␣
↪Integer(6), Integer(3), Integer(2)])
[15]
>>> Composition([Integer(9), Integer(6)]).meet([Integer(1), Integer(3),␣
↪Integer(5), Integer(1), Integer(3), Integer(2)])
[9, 6]
>>> Composition([Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)]).
↪meet([Integer(3), Integer(2)])
```

(continues on next page)

(continued from previous page)

```
[3, 2]
>>> Composition([Integer(4), Integer(2)]).meet([Integer(3), Integer(3)])
[6]
>>> Composition([]).meet([])
[]
>>> Composition([Integer(1)]).meet([Integer(1)])
[1]
```

Let us verify on small examples that the meet of I and J is coarser than both of I and J:

```
sage: all( all( I.is_finer(I.meet(J)) and
....:           J.is_finer(I.meet(J))
....:           for J in Compositions(4) )
....:       for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( I.is_finer(I.meet(J)) and
...           J.is_finer(I.meet(J))
...           for J in Compositions(Integer(4)) )
...       for I in Compositions(Integer(4)) )
True
```

and is the finest composition to do so:

```
sage: all( all( all( I.meet(J).is_finer(K)
....:                for K in I.fatter()
....:                if J.is_finer(K) )
....:           for J in Compositions(3) )
....:       for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> all( all( all( I.meet(J).is_finer(K)
...                for K in I.fatter()
...                if J.is_finer(K) )
...           for J in Compositions(Integer(3)) )
...       for I in Compositions(Integer(3)) )
True
```

The descent set of the meet of I and J is the intersection of the descent sets of I and J:

```
sage: def test_meet(n):
....:     return all( all( I.to_subset().intersection(J.to_subset())
....:                      == I.meet(J).to_subset()
....:                      for J in Compositions(n) )
....:                 for I in Compositions(n) )
sage: all( test_meet(n) for n in range(1, 5) )
True
```

```
>>> from sage.all import *
>>> def test_meet(n):
```

(continues on next page)

(continued from previous page)

```
...     return all( all( I.to_subset().intersection(J.to_subset())
...                      == I.meet(J).to_subset()
...                      for J in Compositions(n) )
...                 for I in Compositions(n) )
>>> all( test_meet(n) for n in range(Integer(1), Integer(5)) )
True
```

See also

join()

AUTHORS:

- Darij Grinberg (2013-09-05)

is_finer(*co2*)

Return `True` if the composition `self` is finer than the composition `co2`; otherwise, return `False`.

EXAMPLES:

```
sage: Composition([4,1,2]).is_finer([3,1,3])
False
sage: Composition([3,1,3]).is_finer([4,1,2])
False
sage: Composition([1,2,2,1,1,2]).is_finer([5,1,3])
True
sage: Composition([2,2,2]).is_finer([4,2])
True
```

```
>>> from sage.all import *
>>> Composition([Integer(4),Integer(1),Integer(2)]).is_finer([Integer(3),
↪Integer(1),Integer(3)])
False
>>> Composition([Integer(3),Integer(1),Integer(3)]).is_finer([Integer(4),
↪Integer(1),Integer(2)])
False
>>> Composition([Integer(1),Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(2)]).is_finer([Integer(5),Integer(1),Integer(3)])
True
>>> Composition([Integer(2),Integer(2),Integer(2)]).is_finer([Integer(4),
↪Integer(2)])
True
```

join(*other*, *check=True*)

Return the join of `self` with a composition `other` of the same size.

The join of two compositions I and J of size n is the coarsest composition of n which refines each of I and J. It can be described as the composition whose descent set is the union of the descent sets of I and J. It is also the concatenation of $I_1, I_2, \cdots, I_m$, where $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is the ribbon decomposition of I with respect to J (see *ribbon_decomposition()*).

INPUT:

- `other` – composition of same size as `self`

- `check` – boolean (default: `True`); whether to check the input compositions for having the same size

OUTPUT: the join of the compositions `self` and `other`

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).join([4, 3, 2])
[3, 1, 1, 2, 1, 1]
sage: Composition([9, 6]).join([1, 3, 6, 3, 2])
[1, 3, 5, 1, 3, 2]
sage: Composition([9, 6]).join([1, 3, 5, 1, 3, 2])
[1, 3, 5, 1, 3, 2]
sage: Composition([1, 1, 1, 1, 1]).join([3, 2])
[1, 1, 1, 1, 1]
sage: Composition([4, 2]).join([3, 3])
[3, 1, 2]
sage: Composition([]).join([])
[]
```

```
>>> from sage.all import *
>>> Composition([Integer(3), Integer(1), Integer(1), Integer(3), Integer(1)]).
↪join([Integer(4), Integer(3), Integer(2)])
[3, 1, 1, 2, 1, 1]
>>> Composition([Integer(9), Integer(6)]).join([Integer(1), Integer(3),␣
↪Integer(6), Integer(3), Integer(2)])
[1, 3, 5, 1, 3, 2]
>>> Composition([Integer(9), Integer(6)]).join([Integer(1), Integer(3),␣
↪Integer(5), Integer(1), Integer(3), Integer(2)])
[1, 3, 5, 1, 3, 2]
>>> Composition([Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)]).
↪join([Integer(3), Integer(2)])
[1, 1, 1, 1, 1]
>>> Composition([Integer(4), Integer(2)]).join([Integer(3), Integer(3)])
[3, 1, 2]
>>> Composition([]).join([])
[]
```

Let us verify on small examples that the join of I and J refines both of I and J:

```
sage: all( all( I.join(J).is_finer(I) and
....:           I.join(J).is_finer(J)
....:           for J in Compositions(4) )
....:       for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( I.join(J).is_finer(I) and
...           I.join(J).is_finer(J)
...           for J in Compositions(Integer(4)) )
...       for I in Compositions(Integer(4)) )
True
```

and is the coarsest composition to do so:

```
sage: all( all( all( K.is_finer(I.join(J))
....:                for K in I.finer()
....:                if K.is_finer(J) )
....:           for J in Compositions(3) )
....:      for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> all( all( all( K.is_finer(I.join(J))
...                for K in I.finer()
...                if K.is_finer(J) )
...           for J in Compositions(Integer(3)) )
...      for I in Compositions(Integer(3)) )
True
```

Let us check that the join of I and J is indeed the concatenation of $I_1, I_2, \cdots, I_m$, where $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is the ribbon decomposition of I with respect to J:

```
sage: all( all( Composition.sum(I.ribbon_decomposition(J)[0])
....:           == I.join(J) for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( Composition.sum(I.ribbon_decomposition(J)[Integer(0)])
...           == I.join(J) for J in Compositions(Integer(4)) )
...      for I in Compositions(Integer(4)) )
True
```

Also, the descent set of the join of I and J is the union of the descent sets of I and J:

```
sage: all( all( I.to_subset().union(J.to_subset())
....:           == I.join(J).to_subset()
....:           for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( I.to_subset().union(J.to_subset())
...           == I.join(J).to_subset()
...           for J in Compositions(Integer(4)) )
...      for I in Compositions(Integer(4)) )
True
```

See also

meet(), *ribbon_decomposition()*

AUTHORS:

- Darij Grinberg (2013-09-05)

major_index()

Return the major index of `self`. The major index is defined as the sum of the descents.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).major_index()
31
```

```
>>> from sage.all import *
>>> Composition([Integer(1), Integer(1), Integer(3), Integer(1), Integer(2),␣
↪Integer(1), Integer(3)]).major_index()
31
```

meet(*other*, *check=True*)

Return the meet of `self` with a composition `other` of the same size.

The meet of two compositions I and J of size n is the finest composition of n which is coarser than each of I and J. It can be described as the composition whose descent set is the intersection of the descent sets of I and J.

INPUT:

- `other` – composition of same size as `self`
- `check` – boolean (default: `True`); whether to check the input compositions for having the same size

OUTPUT: the meet of the compositions `self` and `other`

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).meet([4, 3, 2])
[4, 5]
sage: Composition([9, 6]).meet([1, 3, 6, 3, 2])
[15]
sage: Composition([9, 6]).meet([1, 3, 5, 1, 3, 2])
[9, 6]
sage: Composition([1, 1, 1, 1, 1]).meet([3, 2])
[3, 2]
sage: Composition([4, 2]).meet([3, 3])
[6]
sage: Composition([]).meet([])
[]
sage: Composition([1]).meet([1])
[1]
```

```
>>> from sage.all import *
>>> Composition([Integer(3), Integer(1), Integer(1), Integer(3), Integer(1)]).
↪meet([Integer(4), Integer(3), Integer(2)])
[4, 5]
>>> Composition([Integer(9), Integer(6)]).meet([Integer(1), Integer(3),␣
↪Integer(6), Integer(3), Integer(2)])
[15]
>>> Composition([Integer(9), Integer(6)]).meet([Integer(1), Integer(3),␣
↪Integer(5), Integer(1), Integer(3), Integer(2)])
[9, 6]
>>> Composition([Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)]).
```

(continues on next page)

(continued from previous page)

```
↪meet([Integer(3), Integer(2)])
[3, 2]
>>> Composition([Integer(4), Integer(2)]).meet([Integer(3), Integer(3)])
[6]
>>> Composition([]).meet([])
[]
>>> Composition([Integer(1)]).meet([Integer(1)])
[1]
```

Let us verify on small examples that the meet of I and J is coarser than both of I and J:

```
sage: all( all( I.is_finer(I.meet(J)) and
....:           J.is_finer(I.meet(J))
....:           for J in Compositions(4) )
....:       for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( I.is_finer(I.meet(J)) and
...           J.is_finer(I.meet(J))
...           for J in Compositions(Integer(4)) )
...       for I in Compositions(Integer(4)) )
True
```

and is the finest composition to do so:

```
sage: all( all( all( I.meet(J).is_finer(K)
....:                for K in I.fatter()
....:                if J.is_finer(K) )
....:           for J in Compositions(3) )
....:       for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> all( all( all( I.meet(J).is_finer(K)
...                for K in I.fatter()
...                if J.is_finer(K) )
...           for J in Compositions(Integer(3)) )
...       for I in Compositions(Integer(3)) )
True
```

The descent set of the meet of I and J is the intersection of the descent sets of I and J:

```
sage: def test_meet(n):
....:     return all( all( I.to_subset().intersection(J.to_subset())
....:                      == I.meet(J).to_subset()
....:                      for J in Compositions(n) )
....:                  for I in Compositions(n) )
sage: all( test_meet(n) for n in range(1, 5) )
True
```

```
>>> from sage.all import *
>>> def test_meet(n):
...     return all( all( I.to_subset().intersection(J.to_subset())
...                      == I.meet(J).to_subset()
...                      for J in Compositions(n) )
...                 for I in Compositions(n) )
>>> all( test_meet(n) for n in range(Integer(1), Integer(5)) )
True
```

See also:

join()

AUTHORS:

- Darij Grinberg (2013-09-05)

near_concatenation(*other*)

Return the near-concatenation of two nonempty compositions `self` and `other`.

The near-concatenation $I \odot J$ of two nonempty compositions I and J is defined as the composition $(i_1, i_2, \ldots, i_{n-1}, i_n + j_1, j_2, j_3, \ldots, j_m)$, where $(i_1, i_2, \ldots, i_n) = I$ and $(j_1, j_2, \ldots, j_m) = J$.

This method returns `None` if one of the two input compositions is empty.

EXAMPLES:

```
sage: Composition([1, 1, 3]).near_concatenation(Composition([4, 1, 2]))
[1, 1, 7, 1, 2]
sage: Composition([6]).near_concatenation(Composition([1, 5]))
[7, 5]
sage: Composition([1, 5]).near_concatenation(Composition([6]))
[1, 11]
```

```
>>> from sage.all import *
>>> Composition([Integer(1), Integer(1), Integer(3)]).near_
→concatenation(Composition([Integer(4), Integer(1), Integer(2)]))
[1, 1, 7, 1, 2]
>>> Composition([Integer(6)]).near_concatenation(Composition([Integer(1),␣
→Integer(5)]))
[7, 5]
>>> Composition([Integer(1), Integer(5)]).near_
→concatenation(Composition([Integer(6)]))
[1, 11]
```

partial_sums(*final=True*)

The partial sums of the sequence defined by the entries of the composition.

If $I = (i_1, \ldots, i_m)$ is a composition, then the partial sums of the entries of the composition are $[i_1, i_1 + i_2, \ldots, i_1 + i_2 + \cdots + i_m]$.

INPUT:

- `final` – boolean (default: `True`); whether or not to include the final partial sum, which is always the size of the composition

See also

to_subset()

EXAMPLES:

```
sage: Composition([1,1,3,1,2,1,3]).partial_sums()
[1, 2, 5, 6, 8, 9, 12]
```

```
>>> from sage.all import *
>>> Composition([Integer(1),Integer(1),Integer(3),Integer(1),Integer(2),
↪Integer(1),Integer(3)]).partial_sums()
[1, 2, 5, 6, 8, 9, 12]
```

With `final = False`, the last partial sum is not included:

```
sage: Composition([1,1,3,1,2,1,3]).partial_sums(final=False)
[1, 2, 5, 6, 8, 9]
```

```
>>> from sage.all import *
>>> Composition([Integer(1),Integer(1),Integer(3),Integer(1),Integer(2),
↪Integer(1),Integer(3)]).partial_sums(final=False)
[1, 2, 5, 6, 8, 9]
```

peaks()

Return a list of the peaks of the composition `self`.

The peaks of a composition are the descents which do not immediately follow another descent.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).peaks()
[4, 7]
```

```
>>> from sage.all import *
>>> Composition([Integer(1), Integer(1), Integer(3), Integer(1), Integer(2),␣
↪Integer(1), Integer(3)]).peaks()
[4, 7]
```

refinement_splitting(*J*)

Return the refinement splitting of `self` according to `J`.

INPUT:

- `J` – a composition such that `self` is finer than `J`

OUTPUT:

- the unique list of compositions $(I^{(p)})_{p=1,\ldots,m}$, obtained by splitting I, such that $|I^{(p)}| = J_p$ for all $p = 1, \ldots, m$.

See also

refinement_splitting_lengths()

EXAMPLES:

```
sage: Composition([1,2,2,1,1,2]).refinement_splitting([5,1,3])
[[1, 2, 2], [1], [1, 2]]
sage: Composition([]).refinement_splitting([])
[]
sage: Composition([3]).refinement_splitting([2])
Traceback (most recent call last):
...
ValueError: compositions self (= [3]) and J (= [2]) must be of the same size
sage: Composition([2,1]).refinement_splitting([1,2])
Traceback (most recent call last):
...
ValueError: composition J (= [2, 1]) does not refine self (= [1, 2])
```

```
>>> from sage.all import *
>>> Composition([Integer(1),Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(2)]).refinement_splitting([Integer(5),Integer(1),Integer(3)])
[[1, 2, 2], [1], [1, 2]]
>>> Composition([]).refinement_splitting([])
[]
>>> Composition([Integer(3)]).refinement_splitting([Integer(2)])
Traceback (most recent call last):
...
ValueError: compositions self (= [3]) and J (= [2]) must be of the same size
>>> Composition([Integer(2),Integer(1)]).refinement_splitting([Integer(1),
↪Integer(2)])
Traceback (most recent call last):
...
ValueError: composition J (= [2, 1]) does not refine self (= [1, 2])
```

refinement_splitting_lengths(*J*)

Return the lengths of the compositions in the refinement splitting of `self` according to `J`.

See also

refinement_splitting() for the definition of refinement splitting

EXAMPLES:

```
sage: Composition([1,2,2,1,1,2]).refinement_splitting_lengths([5,1,3])
[3, 1, 2]
sage: Composition([]).refinement_splitting_lengths([])
[]
sage: Composition([3]).refinement_splitting_lengths([2])
Traceback (most recent call last):
...
ValueError: compositions self (= [3]) and J (= [2]) must be of the same size
sage: Composition([2,1]).refinement_splitting_lengths([1,2])
Traceback (most recent call last):
...
ValueError: composition J (= [2, 1]) does not refine self (= [1, 2])
```

```
>>> from sage.all import *
>>> Composition([Integer(1),Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(2)]).refinement_splitting_lengths([Integer(5),Integer(1),
↪Integer(3)])
[3, 1, 2]
>>> Composition([]).refinement_splitting_lengths([])
[]
>>> Composition([Integer(3)]).refinement_splitting_lengths([Integer(2)])
Traceback (most recent call last):
...
ValueError: compositions self (= [3]) and J (= [2]) must be of the same size
>>> Composition([Integer(2),Integer(1)]).refinement_splitting_
↪lengths([Integer(1),Integer(2)])
Traceback (most recent call last):
...
ValueError: composition J (= [2, 1]) does not refine self (= [1, 2])
```

reversed()

Return the reverse composition of `self`.

The reverse composition of a composition $(i_1, i_2, \ldots, i_k)$ is defined as the composition $(i_k, i_{k-1}, \ldots, i_1)$.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).reversed()
[3, 1, 2, 1, 3, 1, 1]
```

```
>>> from sage.all import *
>>> Composition([Integer(1), Integer(1), Integer(3), Integer(1), Integer(2),␣
↪Integer(1), Integer(3)]).reversed()
[3, 1, 2, 1, 3, 1, 1]
```

ribbon_decomposition(*other*, *check=True*)

Return a pair describing the ribbon decomposition of a composition `self` with respect to a composition `other` of the same size.

If I and J are two compositions of the same nonzero size, then the ribbon decomposition of I with respect to J is defined as follows: Write I and J as $I = (i_1, i_2, \ldots, i_n)$ and $J = (j_1, j_2, \ldots, j_m)$. Then, the equality $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ holds for a unique m-tuple $(I_1, I_2, \ldots, I_m)$ of compositions such that each I_k has size j_k and for a unique choice of $m-1$ signs $\bullet$ each of which is either the concatenation sign $\cdot$ or the near-concatenation sign $\odot$ (see `__add__()` and `near_concatenation()` for the definitions of these two signs). This m-tuple and this choice of signs together are said to form the ribbon decomposition of I with respect to J. If I and J are empty, then the same definition applies, except that there are 0 rather than $m-1$ signs.

See Section 4.8 of [NCSF1].

INPUT:

- `other` – composition of same size as `self`
- `check` – boolean (default: `True`); whether to check the input compositions for having the same size

OUTPUT:

- a pair `(u, v)`, where `u` is a tuple of compositions (corresponding to the m-tuple $(I_1, I_2, \ldots, I_m)$ in the above definition), and `v` is a tuple of $0`sand`1`s(encodingthechoiceofsigns`\bullet$ in the above definition, with a 0 standing for $\cdot$ and a 1 standing for $\odot$).

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).ribbon_decomposition([4, 3, 2])
(([3, 1], [1, 2], [1, 1]), (0, 1))
sage: Composition([9, 6]).ribbon_decomposition([1, 3, 6, 3, 2])
(([1], [3], [5, 1], [3], [2]), (1, 1, 1, 1))
sage: Composition([9, 6]).ribbon_decomposition([1, 3, 5, 1, 3, 2])
(([1], [3], [5], [1], [3], [2]), (1, 1, 0, 1, 1))
sage: Composition([1, 1, 1, 1, 1]).ribbon_decomposition([3, 2])
(([1, 1, 1], [1, 1]), (0,))
sage: Composition([4, 2]).ribbon_decomposition([6])
(([4, 2],), ())
sage: Composition([]).ribbon_decomposition([])
((), ())
```

```
>>> from sage.all import *
>>> Composition([Integer(3), Integer(1), Integer(1), Integer(3), Integer(1)]).
↪ribbon_decomposition([Integer(4), Integer(3), Integer(2)])
(([3, 1], [1, 2], [1, 1]), (0, 1))
>>> Composition([Integer(9), Integer(6)]).ribbon_decomposition([Integer(1),␣
↪Integer(3), Integer(6), Integer(3), Integer(2)])
(([1], [3], [5, 1], [3], [2]), (1, 1, 1, 1))
>>> Composition([Integer(9), Integer(6)]).ribbon_decomposition([Integer(1),␣
↪Integer(3), Integer(5), Integer(1), Integer(3), Integer(2)])
(([1], [3], [5], [1], [3], [2]), (1, 1, 0, 1, 1))
>>> Composition([Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)]).
↪ribbon_decomposition([Integer(3), Integer(2)])
(([1, 1, 1], [1, 1]), (0,))
>>> Composition([Integer(4), Integer(2)]).ribbon_decomposition([Integer(6)])
(([4, 2],), ())
>>> Composition([]).ribbon_decomposition([])
((), ())
```

Let us check that the defining property $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is satisfied:

```
sage: def compose_back(u, v):
....:     comp = u[0]
....:     r = len(v)
....:     if len(u) != r + 1:
....:         raise ValueError("something is wrong")
....:     for i in range(r):
....:         if v[i] == 0:
....:             comp += u[i + 1]
....:         else:
....:             comp = comp.near_concatenation(u[i + 1])
....:     return comp
sage: all( all( all( compose_back(*(I.ribbon_decomposition(J))) == I
....:                 for J in Compositions(n) )
....:             for I in Compositions(n) )
....:       for n in range(1, 5) )
True
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> def compose_back(u, v):
...     comp = u[Integer(0)]
...     r = len(v)
...     if len(u) != r + Integer(1):
...         raise ValueError("something is wrong")
...     for i in range(r):
...         if v[i] == Integer(0):
...             comp += u[i + Integer(1)]
...         else:
...             comp = comp.near_concatenation(u[i + Integer(1)])
...     return comp
>>> all( all( all( compose_back(*(I.ribbon_decomposition(J))) == I
...                for J in Compositions(n) )
...           for I in Compositions(n) )
...      for n in range(Integer(1), Integer(5)) )
True
```

AUTHORS:

- Darij Grinberg (2013-08-29)

shuffle_product(*other*, *overlap=False*)

The (overlapping) shuffles of `self` and `other`.

Suppose $I = (i_1, \ldots, i_k)$ and $J = (j_1, \ldots, j_l)$ are two compositions. A *shuffle* of I and J is a composition of length $k + l$ that contains both I and J as subsequences.

More generally, an *overlapping shuffle* of I and J is obtained by distributing the elements of I and J (preserving the relative ordering of these elements) among the positions of an empty list; an element of I and an element of J are permitted to share the same position, in which case they are replaced by their sum. In particular, a shuffle of I and J is an overlapping shuffle of I and J.

INPUT:

- `other` – composition
- `overlap` – boolean (default: `False`); if `True`, the overlapping shuffle product is returned

OUTPUT:

An enumerated set (allowing for multiplicities)

EXAMPLES:

The shuffle product of $[2, 2]$ and $[1, 1, 3]$:

```
sage: alph = Composition([2,2])
sage: beta = Composition([1,1,3])
sage: S = alph.shuffle_product(beta); S                                 #␣
↪needs sage.combinat
Shuffle product of [2, 2] and [1, 1, 3]
sage: S.list()                                                          #␣
↪needs sage.combinat
[[2, 2, 1, 1, 3], [2, 1, 2, 1, 3], [2, 1, 1, 2, 3], [2, 1, 1, 3, 2],
 [1, 2, 2, 1, 3], [1, 2, 1, 2, 3], [1, 2, 1, 3, 2], [1, 1, 2, 2, 3],
 [1, 1, 2, 3, 2], [1, 1, 3, 2, 2]]
```

```
>>> from sage.all import *
>>> alph = Composition([Integer(2),Integer(2)])
>>> beta = Composition([Integer(1),Integer(1),Integer(3)])
>>> S = alph.shuffle_product(beta); S                                    #␣
↪needs sage.combinat
Shuffle product of [2, 2] and [1, 1, 3]
>>> S.list()                                                             #␣
↪needs sage.combinat
[[2, 2, 1, 1, 3], [2, 1, 2, 1, 3], [2, 1, 1, 2, 3], [2, 1, 1, 3, 2],
 [1, 2, 2, 1, 3], [1, 2, 1, 2, 3], [1, 2, 1, 3, 2], [1, 1, 2, 2, 3],
 [1, 1, 2, 3, 2], [1, 1, 3, 2, 2]]
```

The *overlapping* shuffle product of $[2, 2]$ and $[1, 1, 3]$:

```
sage: alph = Composition([2,2])
sage: beta = Composition([1,1,3])
sage: O = alph.shuffle_product(beta, overlap=True); O                     #␣
↪needs sage.combinat
Overlapping shuffle product of [2, 2] and [1, 1, 3]
sage: O.list()                                                            #␣
↪needs sage.combinat
[[2, 2, 1, 1, 3], [2, 1, 2, 1, 3], [2, 1, 1, 2, 3], [2, 1, 1, 3, 2],
 [1, 2, 2, 1, 3], [1, 2, 1, 2, 3], [1, 2, 1, 3, 2], [1, 1, 2, 2, 3],
 [1, 1, 2, 3, 2], [1, 1, 3, 2, 2],
 [3, 2, 1, 3], [2, 3, 1, 3], [3, 1, 2, 3], [2, 1, 3, 3], [3, 1, 3, 2],
 [2, 1, 1, 5], [1, 3, 2, 3], [1, 2, 3, 3], [1, 3, 3, 2], [1, 2, 1, 5],
 [1, 1, 5, 2], [1, 1, 2, 5],
 [3, 3, 3], [3, 1, 5], [1, 3, 5]]
```

```
>>> from sage.all import *
>>> alph = Composition([Integer(2),Integer(2)])
>>> beta = Composition([Integer(1),Integer(1),Integer(3)])
>>> O = alph.shuffle_product(beta, overlap=True); O                      #␣
↪needs sage.combinat
Overlapping shuffle product of [2, 2] and [1, 1, 3]
>>> O.list()                                                             #␣
↪needs sage.combinat
[[2, 2, 1, 1, 3], [2, 1, 2, 1, 3], [2, 1, 1, 2, 3], [2, 1, 1, 3, 2],
 [1, 2, 2, 1, 3], [1, 2, 1, 2, 3], [1, 2, 1, 3, 2], [1, 1, 2, 2, 3],
 [1, 1, 2, 3, 2], [1, 1, 3, 2, 2],
 [3, 2, 1, 3], [2, 3, 1, 3], [3, 1, 2, 3], [2, 1, 3, 3], [3, 1, 3, 2],
 [2, 1, 1, 5], [1, 3, 2, 3], [1, 2, 3, 3], [1, 3, 3, 2], [1, 2, 1, 5],
 [1, 1, 5, 2], [1, 1, 2, 5],
 [3, 3, 3], [3, 1, 5], [1, 3, 5]]
```

Note that the shuffle product of two compositions can include the same composition more than once since a composition can be a shuffle of two compositions in several ways. For example:

```
sage: # needs sage.combinat
sage: w1 = Composition([1])
sage: S = w1.shuffle_product(w1); S
Shuffle product of [1] and [1]
sage: S.list()
```

(continues on next page)

(continued from previous page)

```
[[1, 1], [1, 1]]
sage: O = w1.shuffle_product(w1, overlap=True); O
Overlapping shuffle product of [1] and [1]
sage: O.list()
[[1, 1], [1, 1], [2]]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> w1 = Composition([Integer(1)])
>>> S = w1.shuffle_product(w1); S
Shuffle product of [1] and [1]
>>> S.list()
[[1, 1], [1, 1]]
>>> O = w1.shuffle_product(w1, overlap=True); O
Overlapping shuffle product of [1] and [1]
>>> O.list()
[[1, 1], [1, 1], [2]]
```

size()

Return the size of `self`, that is the sum of its parts.

EXAMPLES:

```
sage: Composition([7,1,3]).size()
11
```

```
>>> from sage.all import *
>>> Composition([Integer(7),Integer(1),Integer(3)]).size()
11
```

specht_module(*base_ring=None*)

Return the Specht module corresponding to `self`.

EXAMPLES:

```
sage: SM = Composition([1,2,2]).specht_module(QQ); SM                      #␣
↪needs sage.combinat sage.modules
Specht module of [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)] over Rational Field
sage: s = SymmetricFunctions(QQ).s()                                      #␣
↪needs sage.combinat sage.modules
sage: s(SM.frobenius_image())                                             #␣
↪needs sage.combinat sage.modules
s[2, 2, 1]
```

```
>>> from sage.all import *
>>> SM = Composition([Integer(1),Integer(2),Integer(2)]).specht_module(QQ);␣
↪SM                         # needs sage.combinat sage.modules
Specht module of [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)] over Rational Field
>>> s = SymmetricFunctions(QQ).s()                                        #␣
↪needs sage.combinat sage.modules
>>> s(SM.frobenius_image())                                               #␣
↪needs sage.combinat sage.modules
s[2, 2, 1]
```

specht_module_dimension(*base_ring=None*)

Return the dimension of the Specht module corresponding to `self`.

INPUT:

- `base_ring` – (default: **Q**) the base ring

EXAMPLES:

```
sage: Composition([1,2,2]).specht_module_dimension()                          #␣
↪needs sage.combinat sage.modules
5
sage: Composition([1,2,2]).specht_module_dimension(GF(2))                     #␣
↪needs sage.combinat sage.modules sage.rings.finite_rings
5
```

```
>>> from sage.all import *
>>> Composition([Integer(1),Integer(2),Integer(2)]).specht_module_dimension()␣
↪                         # needs sage.combinat sage.modules
5
>>> Composition([Integer(1),Integer(2),Integer(2)]).specht_module_
↪dimension(GF(Integer(2)))                        # needs sage.combinat sage.
↪modules sage.rings.finite_rings
5
```

static sum(*compositions*)

Return the concatenation of the given compositions.

INPUT:

- `compositions` – list (or iterable) of compositions

EXAMPLES:

```
sage: Composition.sum([Composition([1, 1, 3]), Composition([4, 1, 2]),␣
↪Composition([3,1])])
[1, 1, 3, 4, 1, 2, 3, 1]
```

```
>>> from sage.all import *
>>> Composition.sum([Composition([Integer(1), Integer(1), Integer(3)]),␣
↪Composition([Integer(4), Integer(1), Integer(2)]), Composition([Integer(3),
↪Integer(1)])])
[1, 1, 3, 4, 1, 2, 3, 1]
```

Any iterable can be provided as input:

```
sage: Composition.sum([Composition([i,i]) for i in [4,1,3]])
[4, 4, 1, 1, 3, 3]
```

```
>>> from sage.all import *
>>> Composition.sum([Composition([i,i]) for i in [Integer(4),Integer(1),
↪Integer(3)]])
[4, 4, 1, 1, 3, 3]
```

Empty inputs are handled gracefully:

```
sage: Composition.sum([]) == Composition([])
True
```

```
>>> from sage.all import *
>>> Composition.sum([]) == Composition([])
True
```

sup(*other*, *check=True*)

Return the join of `self` with a composition `other` of the same size.

The join of two compositions I and J of size n is the coarsest composition of n which refines each of I and J. It can be described as the composition whose descent set is the union of the descent sets of I and J. It is also the concatenation of $I_1, I_2, \cdots, I_m$, where $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is the ribbon decomposition of I with respect to J (see `ribbon_decomposition()`).

INPUT:

- `other` – composition of same size as `self`
- `check` – boolean (default: `True`); whether to check the input compositions for having the same size

OUTPUT: the join of the compositions `self` and `other`

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).join([4, 3, 2])
[3, 1, 1, 2, 1, 1]
sage: Composition([9, 6]).join([1, 3, 6, 3, 2])
[1, 3, 5, 1, 3, 2]
sage: Composition([9, 6]).join([1, 3, 5, 1, 3, 2])
[1, 3, 5, 1, 3, 2]
sage: Composition([1, 1, 1, 1, 1]).join([3, 2])
[1, 1, 1, 1, 1]
sage: Composition([4, 2]).join([3, 3])
[3, 1, 2]
sage: Composition([]).join([])
[]
```

```
>>> from sage.all import *
>>> Composition([Integer(3), Integer(1), Integer(1), Integer(3), Integer(1)]).
↪join([Integer(4), Integer(3), Integer(2)])
[3, 1, 1, 2, 1, 1]
>>> Composition([Integer(9), Integer(6)]).join([Integer(1), Integer(3),␣
↪Integer(6), Integer(3), Integer(2)])
[1, 3, 5, 1, 3, 2]
>>> Composition([Integer(9), Integer(6)]).join([Integer(1), Integer(3),␣
↪Integer(5), Integer(1), Integer(3), Integer(2)])
[1, 3, 5, 1, 3, 2]
>>> Composition([Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)]).
↪join([Integer(3), Integer(2)])
[1, 1, 1, 1, 1]
>>> Composition([Integer(4), Integer(2)]).join([Integer(3), Integer(3)])
[3, 1, 2]
>>> Composition([]).join([])
[]
```

Let us verify on small examples that the join of I and J refines both of I and J:

```
sage: all( all( I.join(J).is_finer(I) and
....:           I.join(J).is_finer(J)
....:           for J in Compositions(4) )
....:       for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( I.join(J).is_finer(I) and
...           I.join(J).is_finer(J)
...           for J in Compositions(Integer(4)) )
...       for I in Compositions(Integer(4)) )
True
```

and is the coarsest composition to do so:

```
sage: all( all( all( K.is_finer(I.join(J))
....:                for K in I.finer()
....:                if K.is_finer(J) )
....:           for J in Compositions(3) )
....:       for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> all( all( all( K.is_finer(I.join(J))
...                for K in I.finer()
...                if K.is_finer(J) )
...           for J in Compositions(Integer(3)) )
...       for I in Compositions(Integer(3)) )
True
```

Let us check that the join of I and J is indeed the concatenation of $I_1, I_2, \cdots, I_m$, where $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is the ribbon decomposition of I with respect to J:

```
sage: all( all( Composition.sum(I.ribbon_decomposition(J)[0])
....:           == I.join(J) for J in Compositions(4) )
....:       for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( Composition.sum(I.ribbon_decomposition(J)[Integer(0)])
...           == I.join(J) for J in Compositions(Integer(4)) )
...       for I in Compositions(Integer(4)) )
True
```

Also, the descent set of the join of I and J is the union of the descent sets of I and J:

```
sage: all( all( I.to_subset().union(J.to_subset())
....:           == I.join(J).to_subset()
....:           for J in Compositions(4) )
....:       for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( I.to_subset().union(J.to_subset())
...           == I.join(J).to_subset()
...           for J in Compositions(Integer(4)) )
...       for I in Compositions(Integer(4)) )
True
```

See also

meet(), *ribbon_decomposition()*

AUTHORS:

- Darij Grinberg (2013-09-05)

to_code()

Return the code of the composition `self`.

The code of a composition I is a list of length size(I) of 1s and 0s such that there is a 1 wherever a new part starts. (Exceptional case: When the composition is empty, the code is `[0]`.)

EXAMPLES:

```
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> Composition([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).to_
↪code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
```

to_partition()

Return the partition obtained by sorting `self` into decreasing order.

EXAMPLES:

```
sage: Composition([2,1,3]).to_partition()                                    #␣
↪needs sage.combinat
[3, 2, 1]
sage: Composition([4,2,2]).to_partition()                                    #␣
↪needs sage.combinat
[4, 2, 2]
sage: Composition([]).to_partition()                                         #␣
↪needs sage.combinat
[]
```

```
>>> from sage.all import *
>>> Composition([Integer(2),Integer(1),Integer(3)]).to_partition()          ␣
↪                   # needs sage.combinat
[3, 2, 1]
>>> Composition([Integer(4),Integer(2),Integer(2)]).to_partition()          ␣
↪                   # needs sage.combinat
[4, 2, 2]
```

(continues on next page)

(continued from previous page)

```
>>> Composition([]).to_partition()                                          #␣
↪needs sage.combinat
[]
```

to_skew_partition(*overlap=1*)

Return the skew partition obtained from `self`.

This is a skew partition whose rows have the entries of `self` as their length, taken in reverse order (so the first entry of `self` is the length of the lowermost row, etc.). The parameter `overlap` indicates the number of cells on each row that are directly below cells of the previous row. When it is set to 1 (its default value), the result is the ribbon shape of `self`.

EXAMPLES:

```
sage: # needs sage.combinat
sage: Composition([3,4,1]).to_skew_partition()
[6, 6, 3] / [5, 2]
sage: Composition([3,4,1]).to_skew_partition(overlap=0)
[8, 7, 3] / [7, 3]
sage: Composition([]).to_skew_partition()
[] / []
sage: Composition([1,2]).to_skew_partition()
[2, 1] / []
sage: Composition([2,1]).to_skew_partition()
[2, 2] / [1]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> Composition([Integer(3),Integer(4),Integer(1)]).to_skew_partition()
[6, 6, 3] / [5, 2]
>>> Composition([Integer(3),Integer(4),Integer(1)]).to_skew_
↪partition(overlap=Integer(0))
[8, 7, 3] / [7, 3]
>>> Composition([]).to_skew_partition()
[] / []
>>> Composition([Integer(1),Integer(2)]).to_skew_partition()
[2, 1] / []
>>> Composition([Integer(2),Integer(1)]).to_skew_partition()
[2, 2] / [1]
```

to_subset(*final=False*)

The subset corresponding to `self` under the bijection (see below) between compositions of n and subsets of $\{1, 2, \ldots, n-1\}$.

The bijection maps a composition $(i_1, \ldots, i_k)$ of n to $\{i_1, i_1 + i_2, i_1 + i_2 + i_3, \ldots, i_1 + \cdots + i_{k-1}\}$.

INPUT:

- `final` – boolean (default: `False`); whether or not to include the final partial sum, which is always the size of the composition

See also

partial_sums()

EXAMPLES:

```
sage: Composition([1,1,3,1,2,1,3]).to_subset()
{1, 2, 5, 6, 8, 9}
sage: for I in Compositions(3): print(I.to_subset())
{1, 2}
{1}
{2}
{}
```

```
>>> from sage.all import *
>>> Composition([Integer(1),Integer(1),Integer(3),Integer(1),Integer(2),
↪Integer(1),Integer(3)]).to_subset()
{1, 2, 5, 6, 8, 9}
>>> for I in Compositions(Integer(3)): print(I.to_subset())
{1, 2}
{1}
{2}
{}
```

With `final=True`, the sum of all the elements of the composition is included in the subset:

```
sage: Composition([1,1,3,1,2,1,3]).to_subset(final=True)
{1, 2, 5, 6, 8, 9, 12}
```

```
>>> from sage.all import *
>>> Composition([Integer(1),Integer(1),Integer(3),Integer(1),Integer(2),
↪Integer(1),Integer(3)]).to_subset(final=True)
{1, 2, 5, 6, 8, 9, 12}
```

wll_gt(*co2*)

Return `True` if the composition `self` is greater than the composition `co2` with respect to the wll-ordering; otherwise, return `False`.

The wll-ordering is a total order on the set of all compositions defined as follows: A composition I is greater than a composition J if and only if one of the following conditions holds:

- The size of I is greater than the size of J.
- The size of I equals the size of J, but the length of I is greater than the length of J.
- The size of I equals the size of J, and the length of I equals the length of J, but I is lexicographically greater than J.

("wll-ordering" is short for "weight, length, lexicographic ordering".)

EXAMPLES:

```
sage: Composition([4,1,2]).wll_gt([3,1,3])
True
sage: Composition([7]).wll_gt([4,1,2])
False
sage: Composition([8]).wll_gt([4,1,2])
True
sage: Composition([3,2,2,2]).wll_gt([5,2])
True
```

(continues on next page)

(continued from previous page)

```
sage: Composition([]).wll_gt([3])
False
sage: Composition([2,1]).wll_gt([2,1])
False
sage: Composition([2,2,2]).wll_gt([4,2])
True
sage: Composition([4,2]).wll_gt([2,2,2])
False
sage: Composition([1,1,2]).wll_gt([2,2])
True
sage: Composition([2,2]).wll_gt([1,3])
True
sage: Composition([2,1,2]).wll_gt([])
True
```

```
>>> from sage.all import *
>>> Composition([Integer(4),Integer(1),Integer(2)]).wll_gt([Integer(3),
↪Integer(1),Integer(3)])
True
>>> Composition([Integer(7)]).wll_gt([Integer(4),Integer(1),Integer(2)])
False
>>> Composition([Integer(8)]).wll_gt([Integer(4),Integer(1),Integer(2)])
True
>>> Composition([Integer(3),Integer(2),Integer(2),Integer(2)]).wll_
↪gt([Integer(5),Integer(2)])
True
>>> Composition([]).wll_gt([Integer(3)])
False
>>> Composition([Integer(2),Integer(1)]).wll_gt([Integer(2),Integer(1)])
False
>>> Composition([Integer(2),Integer(2),Integer(2)]).wll_gt([Integer(4),
↪Integer(2)])
True
>>> Composition([Integer(4),Integer(2)]).wll_gt([Integer(2),Integer(2),
↪Integer(2)])
False
>>> Composition([Integer(1),Integer(1),Integer(2)]).wll_gt([Integer(2),
↪Integer(2)])
True
>>> Composition([Integer(2),Integer(2)]).wll_gt([Integer(1),Integer(3)])
True
>>> Composition([Integer(2),Integer(1),Integer(2)]).wll_gt([])
True
```

class sage.combinat.composition.**Compositions**(*is_infinite=False*, *category=None*)

Bases: UniqueRepresentation, Parent

Set of integer compositions.

A composition c of a nonnegative integer n is a list of positive integers with total sum n.

See also

- *Composition*
- *Partitions*
- *IntegerVectors*

EXAMPLES:

There are 8 compositions of 4:

```
sage: Compositions(4).cardinality()
8
```

```
>>> from sage.all import *
>>> Compositions(Integer(4)).cardinality()
8
```

Here is the list of them:

```
sage: Compositions(4).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4)).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
```

You can use the `.first()` method to get the 'first' composition of a number:

```
sage: Compositions(4).first()
[1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4)).first()
[1, 1, 1, 1]
```

You can also calculate the 'next' composition given the current one:

```
sage: Compositions(4).next([1,1,2])
[1, 2, 1]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4)).next([Integer(1),Integer(1),Integer(2)])
[1, 2, 1]
```

If n is not specified, this returns the combinatorial class of all (nonnegative) integer compositions:

```
sage: Compositions()
Compositions of nonnegative integers
sage: [] in Compositions()
True
sage: [2,3,1] in Compositions()
```

(continues on next page)

(continued from previous page)

```
True
sage: [-2,3,1] in Compositions()
False
```

```
>>> from sage.all import *
>>> Compositions()
Compositions of nonnegative integers
>>> [] in Compositions()
True
>>> [Integer(2),Integer(3),Integer(1)] in Compositions()
True
>>> [-Integer(2),Integer(3),Integer(1)] in Compositions()
False
```

If n is specified, it returns the class of compositions of n:

```
sage: Compositions(3)
Compositions of 3
sage: list(Compositions(3))
[[1, 1, 1], [1, 2], [2, 1], [3]]
sage: Compositions(3).cardinality()
4
```

```
>>> from sage.all import *
>>> Compositions(Integer(3))
Compositions of 3
>>> list(Compositions(Integer(3)))
[[1, 1, 1], [1, 2], [2, 1], [3]]
>>> Compositions(Integer(3)).cardinality()
4
```

The following examples show how to test whether or not an object is a composition:

```
sage: [3,4] in Compositions()
True
sage: [3,4] in Compositions(7)
True
sage: [3,4] in Compositions(5)
False
```

```
>>> from sage.all import *
>>> [Integer(3),Integer(4)] in Compositions()
True
>>> [Integer(3),Integer(4)] in Compositions(Integer(7))
True
>>> [Integer(3),Integer(4)] in Compositions(Integer(5))
False
```

Similarly, one can check whether or not an object is a composition which satisfies further constraints:

```
sage: [4,2] in Compositions(6, inner=[2,2])
True
```

(continues on next page)

(continued from previous page)

```
sage: [4,2] in Compositions(6, inner=[2,3])
False
sage: [4,1] in Compositions(5, inner=[2,1], max_slope = 0)
True
```

```
>>> from sage.all import *
>>> [Integer(4),Integer(2)] in Compositions(Integer(6), inner=[Integer(2),
↪Integer(2)])
True
>>> [Integer(4),Integer(2)] in Compositions(Integer(6), inner=[Integer(2),
↪Integer(3)])
False
>>> [Integer(4),Integer(1)] in Compositions(Integer(5), inner=[Integer(2),
↪Integer(1)], max_slope = Integer(0))
True
```

An example with incompatible constraints:

```
sage: [4,2] in Compositions(6, inner=[2,2], min_part=3)
False
```

```
>>> from sage.all import *
>>> [Integer(4),Integer(2)] in Compositions(Integer(6), inner=[Integer(2),
↪Integer(2)], min_part=Integer(3))
False
```

The options `length`, `min_length`, and `max_length` can be used to set length constraints on the compositions. For example, the compositions of 4 of length equal to, at least, and at most 2 are given by:

```
sage: Compositions(4, length=2).list()
[[3, 1], [2, 2], [1, 3]]
sage: Compositions(4, min_length=2).list()
[[3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
sage: Compositions(4, max_length=2).list()
[[4], [3, 1], [2, 2], [1, 3]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4), length=Integer(2)).list()
[[3, 1], [2, 2], [1, 3]]
>>> Compositions(Integer(4), min_length=Integer(2)).list()
[[3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
>>> Compositions(Integer(4), max_length=Integer(2)).list()
[[4], [3, 1], [2, 2], [1, 3]]
```

Setting both `min_length` and `max_length` to the same value is equivalent to setting `length` to this value:

```
sage: Compositions(4, min_length=2, max_length=2).list()
[[3, 1], [2, 2], [1, 3]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4), min_length=Integer(2), max_length=Integer(2)).list()
[[3, 1], [2, 2], [1, 3]]
```

The options `inner` and `outer` can be used to set part-by-part containment constraints. The list of compositions of 4 bounded above by `[3,1,2]` is given by:

```
sage: list(Compositions(4, outer=[3,1,2]))
[[3, 1], [2, 1, 1], [1, 1, 2]]
```

```
>>> from sage.all import *
>>> list(Compositions(Integer(4), outer=[Integer(3),Integer(1),Integer(2)]))
[[3, 1], [2, 1, 1], [1, 1, 2]]
```

`outer` sets `max_length` to the length of its argument. Moreover, the parts of `outer` may be infinite to clear the constraint on specific parts. This is the list of compositions of 4 of length at most 3 such that the first and third parts are at most 1:

```
sage: Compositions(4, outer=[1,oo,1]).list()
[[1, 3], [1, 2, 1]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4), outer=[Integer(1),oo,Integer(1)]).list()
[[1, 3], [1, 2, 1]]
```

This is the list of compositions of 4 bounded below by `[1,1,1]`:

```
sage: Compositions(4, inner=[1,1,1]).list()
[[2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4), inner=[Integer(1),Integer(1),Integer(1)]).list()
[[2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

The options `min_slope` and `max_slope` can be used to set constraints on the slope, that is the difference `p[i+1]-p[i]` of two consecutive parts. The following is the list of weakly increasing compositions of 4:

```
sage: Compositions(4, min_slope=0).list()
[[4], [2, 2], [1, 3], [1, 1, 2], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4), min_slope=Integer(0)).list()
[[4], [2, 2], [1, 3], [1, 1, 2], [1, 1, 1, 1]]
```

Here are the weakly decreasing ones:

```
sage: Compositions(4, max_slope=0).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4), max_slope=Integer(0)).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

The following is the list of compositions of 4 such that two consecutive parts differ by at most one:

```
sage: Compositions(4, min_slope=-1, max_slope=1).list()
[[4], [2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(4), min_slope=-Integer(1), max_slope=Integer(1)).list()
[[4], [2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

The constraints can be combined together in all reasonable ways. This is the list of compositions of 5 of length between 2 and 4 such that the difference between consecutive parts is between -2 and 1:

```
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4).
↪list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [2, 1, 1, 1], [1, 2, 2], [1, 2,␣
↪1, 1], [1, 1, 2, 1], [1, 1, 1, 2]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(5), max_slope=Integer(1), min_slope=-Integer(2), min_
↪length=Integer(2), max_length=Integer(4)).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [2, 1, 1, 1], [1, 2, 2], [1, 2,␣
↪1, 1], [1, 1, 2, 1], [1, 1, 1, 2]]
```

We can do the same thing with an outer constraint:

```
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4,␣
↪outer=[2,5,2]).list()
[[2, 3], [2, 2, 1], [2, 1, 2], [1, 2, 2]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(5), max_slope=Integer(1), min_slope=-Integer(2), min_
↪length=Integer(2), max_length=Integer(4), outer=[Integer(2),Integer(5),
↪Integer(2)]).list()
[[2, 3], [2, 2, 1], [2, 1, 2], [1, 2, 2]]
```

However, providing incoherent constraints may yield strange results. It is up to the user to ensure that the inner and outer compositions themselves satisfy the parts and slope constraints.

Note that setting `min_part=0` is not allowed:

```
sage: Compositions(2, length=3, min_part=0)
Traceback (most recent call last):
...
ValueError: setting min_part=0 is not allowed for Compositions
```

```
>>> from sage.all import *
>>> Compositions(Integer(2), length=Integer(3), min_part=Integer(0))
Traceback (most recent call last):
...
ValueError: setting min_part=0 is not allowed for Compositions
```

Instead you must use `IntegerVectors`:

```
sage: list(IntegerVectors(2, 3))
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

```
>>> from sage.all import *
>>> list(IntegerVectors(Integer(2), Integer(3)))
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

The generation algorithm is constant amortized time, and handled by the generic tool `IntegerListsLex`.

Element

alias of `Composition`

from_code(*code*)

Return the composition from its code. The code of a composition I is a list of length $\mathrm{size}(I)$ consisting of 1s and 0s such that there is a 1 wherever a new part starts. (Exceptional case: When the composition is empty, the code is `[0]`.)

EXAMPLES:

```
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Compositions().from_code(_)
[4, 1, 2, 3, 5]
sage: Composition([3,1,2,3,5]).to_code()
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Compositions().from_code(_)
[3, 1, 2, 3, 5]
```

```
>>> from sage.all import *
>>> Composition([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).to_
↪code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
>>> Compositions().from_code(_)
[4, 1, 2, 3, 5]
>>> Composition([Integer(3),Integer(1),Integer(2),Integer(3),Integer(5)]).to_
↪code()
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
>>> Compositions().from_code(_)
[3, 1, 2, 3, 5]
```

from_descents(*descents*, *nps=None*)

Return a composition from the list of descents.

INPUT:

- `descents` – an iterable
- `nps` – integer or `None` (default: `None`)

OUTPUT:

- The composition of `nps` whose descents are listed in `descents`, assuming that `nps` is not `None` (otherwise, the last element of `descents` is removed from `descents`, and `nps` is set to be this last element plus 1).

EXAMPLES:

```
sage: [x-1 for x in Composition([1, 1, 3, 4, 3]).to_subset()]
[0, 1, 4, 8]
```

(continues on next page)

(continued from previous page)

```
sage: Compositions().from_descents([1,0,4,8],12)
[1, 1, 3, 4, 3]
sage: Compositions().from_descents([1,0,4,8,11])
[1, 1, 3, 4, 3]
```

```
>>> from sage.all import *
>>> [x-Integer(1) for x in Composition([Integer(1), Integer(1), Integer(3),␣
↪Integer(4), Integer(3)]).to_subset()]
[0, 1, 4, 8]
>>> Compositions().from_descents([Integer(1),Integer(0),Integer(4),
↪Integer(8)],Integer(12))
[1, 1, 3, 4, 3]
>>> Compositions().from_descents([Integer(1),Integer(0),Integer(4),Integer(8),
↪Integer(11)])
[1, 1, 3, 4, 3]
```

from_subset(S, n)

The composition of n corresponding to the subset `S` of $\{1, 2, \ldots, n-1\}$ under the bijection that maps the composition $(i_1, i_2, \ldots, i_k)$ of n to the subset $\{i_1, i_1 + i_2, i_1 + i_2 + i_3, \ldots, i_1 + \cdots + i_{k-1}\}$ (see *Composition.to_subset()*).

INPUT:

- `S` – an iterable, a subset of $\{1, 2, \ldots, n-1\}$
- `n` – integer

EXAMPLES:

```
sage: Compositions().from_subset([2,1,5,9], 12)
[1, 1, 3, 4, 3]
sage: Compositions().from_subset({2,1,5,9}, 12)
[1, 1, 3, 4, 3]

sage: Compositions().from_subset([], 12)
[12]
sage: Compositions().from_subset([], 0)
[]
```

```
>>> from sage.all import *
>>> Compositions().from_subset([Integer(2),Integer(1),Integer(5),Integer(9)],␣
↪Integer(12))
[1, 1, 3, 4, 3]
>>> Compositions().from_subset({Integer(2),Integer(1),Integer(5),Integer(9)},␣
↪Integer(12))
[1, 1, 3, 4, 3]

>>> Compositions().from_subset([], Integer(12))
[12]
>>> Compositions().from_subset([], Integer(0))
[]
```

class sage.combinat.composition.**Compositions_all**

Bases: *Compositions*

Class of all compositions.

subset(*size=None*)

Return the set of compositions of the given size.

EXAMPLES:

```
sage: C = Compositions()
sage: C.subset(4)
Compositions of 4
sage: C.subset(size=3)
Compositions of 3
```

```
>>> from sage.all import *
>>> C = Compositions()
>>> C.subset(Integer(4))
Compositions of 4
>>> C.subset(size=Integer(3))
Compositions of 3
```

zero()

Return the zero of the additive monoid.

This is the empty composition.

EXAMPLES:

```
sage: C = Compositions()
sage: C.zero()
[]
```

```
>>> from sage.all import *
>>> C = Compositions()
>>> C.zero()
[]
```

class sage.combinat.composition.**Compositions_constraints**(**args*, ***kwds*)

Bases: `IntegerListsLex`

class sage.combinat.composition.**Compositions_n**(*n*)

Bases: `Compositions`

Class of compositions of a fixed n.

cardinality()

Return the number of compositions of n.

random_element()

Return a random `Composition` with uniform probability.

This method generates a random binary word starting with a 1 and then uses the bijection between compositions and their code.

EXAMPLES:

```
sage: Compositions(5).random_element() # random
[2, 1, 1, 1]
sage: Compositions(0).random_element()
[]
sage: Compositions(1).random_element()
[1]
```

```
>>> from sage.all import *
>>> Compositions(Integer(5)).random_element() # random
[2, 1, 1, 1]
>>> Compositions(Integer(0)).random_element()
[]
>>> Compositions(Integer(1)).random_element()
[1]
```

sage.combinat.composition.**composition_iterator_fast**(n)

Iterator over compositions of n yielded as lists.

5.1.30 Signed Compositions

class sage.combinat.composition_signed.**SignedCompositions**(n)

Bases: *Compositions_n*

The class of signed compositions of n.

EXAMPLES:

```
sage: SC3 = SignedCompositions(3); SC3
Signed compositions of 3
sage: SC3.cardinality()
18
sage: len(SC3.list())
18
sage: SC3.first()
[1, 1, 1]
sage: SC3.last()
[-3]
sage: SC3.random_element() # random
[1, -1, 1]
sage: SC3.list()
[[1, 1, 1],
 [1, 1, -1],
 [1, -1, 1],
 [1, -1, -1],
 [-1, 1, 1],
 [-1, 1, -1],
 [-1, -1, 1],
 [-1, -1, -1],
 [1, 2],
 [1, -2],
 [-1, 2],
 [-1, -2],
 [2, 1],
```

(continues on next page)

(continued from previous page)

```
 [2, -1],
 [-2, 1],
 [-2, -1],
 [3],
 [-3]]
```

```
>>> from sage.all import *
>>> SC3 = SignedCompositions(Integer(3)); SC3
Signed compositions of 3
>>> SC3.cardinality()
18
>>> len(SC3.list())
18
>>> SC3.first()
[1, 1, 1]
>>> SC3.last()
[-3]
>>> SC3.random_element() # random
[1, -1, 1]
>>> SC3.list()
[[1, 1, 1],
 [1, 1, -1],
 [1, -1, 1],
 [1, -1, -1],
 [-1, 1, 1],
 [-1, 1, -1],
 [-1, -1, 1],
 [-1, -1, -1],
 [1, 2],
 [1, -2],
 [-1, 2],
 [-1, -2],
 [2, 1],
 [2, -1],
 [-2, 1],
 [-2, -1],
 [3],
 [-3]]
```

cardinality()

Return the number of elements in `self`.

The number of signed compositions of n is equal to

$$\sum_{i=1}^{n+1} \binom{n-1}{i-1} 2^i$$

EXAMPLES:

```
sage: SC4 = SignedCompositions(4)
sage: SC4.cardinality() == len(SC4.list())
True
```

(continues on next page)

(continued from previous page)

```
sage: SignedCompositions(3).cardinality()
18
```

```
>>> from sage.all import *
>>> SC4 = SignedCompositions(Integer(4))
>>> SC4.cardinality() == len(SC4.list())
True
>>> SignedCompositions(Integer(3)).cardinality()
18
```

5.1.31 Composition Tableaux

AUTHORS:

- Chris Berg, Jeff Ferreira (2012-9): Initial version

class sage.combinat.composition_tableau.**CompositionTableau**(*parent*, *t*)

Bases: *CombinatorialElement*

A composition tableau.

A *composition tableau* t of shape $I = (I_1, \ldots, I_\ell)$ is an array of boxes in rows, I_i boxes in row i, filled with positive integers such that:

1) the entries in the rows of t weakly decrease from left to right,
2) the left-most column of t strictly increase from top to bottom.
3) Add zero entries to the rows of t until the resulting array is rectangular of shape $\ell \times m$. For $1 \leq i < j \leq \ell, 2 \leq k \leq m$ and $(t(j,k) \neq 0$, and also if $t(j,k) \geq t(i,k))$ implies $t(j,k) > t(i,k-1)$.

INPUT:

- t – list of lists

EXAMPLES:

```
sage: CompositionTableau([[1],[2,2]])
[[1], [2, 2]]
sage: CompositionTableau([[1],[3,2],[4,4]])
[[1], [3, 2], [4, 4]]
sage: CompositionTableau([])
[]
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1)],[Integer(2),Integer(2)]])
[[1], [2, 2]]
>>> CompositionTableau([[Integer(1)],[Integer(3),Integer(2)],[Integer(4),
↪Integer(4)]])
[[1], [3, 2], [4, 4]]
>>> CompositionTableau([])
[]
```

descent_composition()

Return the composition corresponding to the set of all i that do not have $i+1$ appearing strictly to the left of i in self.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).descent_composition()
[1, 2, 2]
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1)],[Integer(3),Integer(2)],[Integer(4),
↪Integer(4)]]).descent_composition()
[1, 2, 2]
```

descent_set()

Return the set of all i that do *not* have $i+1$ appearing strictly to the left of i in `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).descent_set()
[1, 3]
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1)],[Integer(3),Integer(2)],[Integer(4),
↪Integer(4)]]).descent_set()
[1, 3]
```

is_standard()

Return `True` if `self` is a standard composition tableau and `False` otherwise.

EXAMPLES:

```
sage: CompositionTableau([[1,1],[3,2],[4,4,3]]).is_standard()
False
sage: CompositionTableau([[2,1],[3],[4]]).is_standard()
True
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1),Integer(1)],[Integer(3),Integer(2)],
↪[Integer(4),Integer(4),Integer(3)]]).is_standard()
False
>>> CompositionTableau([[Integer(2),Integer(1)],[Integer(3)],[Integer(4)]]).
↪is_standard()
True
```

pp()

Return a pretty print string of `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).pp()
1
3  2
4  4
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1)],[Integer(3),Integer(2)],[Integer(4),
↪Integer(4)]]).pp()
```

(continues on next page)

(continued from previous page)

```
1
3  2
4  4
```

shape_composition()

Return a Composition object which is the shape of `self`.

EXAMPLES:

```
sage: CompositionTableau([[1,1],[3,2],[4,4,3]]).shape_composition()
[2, 2, 3]
sage: CompositionTableau([[2,1],[3],[4]]).shape_composition()
[2, 1, 1]
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1),Integer(1)],[Integer(3),Integer(2)],
→[Integer(4),Integer(4),Integer(3)]]).shape_composition()
[2, 2, 3]
>>> CompositionTableau([[Integer(2),Integer(1)],[Integer(3)],[Integer(4)]]).
→shape_composition()
[2, 1, 1]
```

shape_partition()

Return a partition which is the shape of `self` sorted into weakly decreasing order.

EXAMPLES:

```
sage: CompositionTableau([[1,1],[3,2],[4,4,3]]).shape_partition()
[3, 2, 2]
sage: CompositionTableau([[2,1],[3],[4]]).shape_partition()
[2, 1, 1]
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1),Integer(1)],[Integer(3),Integer(2)],
→[Integer(4),Integer(4),Integer(3)]]).shape_partition()
[3, 2, 2]
>>> CompositionTableau([[Integer(2),Integer(1)],[Integer(3)],[Integer(4)]]).
→shape_partition()
[2, 1, 1]
```

size()

Return the number of boxes in `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).size()
5
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1)],[Integer(3),Integer(2)],[Integer(4),
→Integer(4)]]).size()
5
```

weight()

Return a composition where entry i is the number of times that i appears in `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).weight()
[1, 1, 1, 2, 0]
```

```
>>> from sage.all import *
>>> CompositionTableau([[Integer(1)],[Integer(3),Integer(2)],[Integer(4),
↪Integer(4)]]).weight()
[1, 1, 1, 2, 0]
```

class sage.combinat.composition_tableau.**CompositionTableaux**(****kwds*)

Bases: `UniqueRepresentation`, `Parent`

Composition tableaux.

INPUT:

- `size` – the size of the composition tableaux
- `shape` – the shape of the composition tableaux
- `max_entry` – the maximum entry for the composition tableaux

The first argument is interpreted as `size` or `shape` depending on whether it is an integer or a composition.

EXAMPLES:

```
sage: CT = CompositionTableaux(3); CT
Composition Tableaux of size 3 and maximum entry 3
sage: list(CT)
[[[1], [2], [3]],
 [[1], [2, 2]],
 [[1], [3, 2]],
 [[1], [3, 3]],
 [[2], [3, 3]],
 [[1, 1], [2]],
 [[1, 1], [3]],
 [[2, 1], [3]],
 [[2, 2], [3]],
 [[1, 1, 1]],
 [[2, 1, 1]],
 [[2, 2, 1]],
 [[2, 2, 2]],
 [[3, 1, 1]],
 [[3, 2, 1]],
 [[3, 2, 2]],
 [[3, 3, 1]],
 [[3, 3, 2]],
 [[3, 3, 3]]]

sage: CT = CompositionTableaux([1,2,1]); CT
Composition tableaux of shape [1, 2, 1] and maximum entry 4
sage: list(CT)
[[[1], [2, 2], [3]],
```

(continues on next page)

(continued from previous page)

```
 [[1], [2, 2], [4]],
 [[1], [3, 2], [4]],
 [[1], [3, 3], [4]],
 [[2], [3, 3], [4]]]

sage: CT = CompositionTableaux(shape=[1,2,1],max_entry=3); CT
Composition tableaux of shape [1, 2, 1] and maximum entry 3
sage: list(CT)
[[[1], [2, 2], [3]]]

sage: CT = CompositionTableaux(2,max_entry=3); CT
Composition Tableaux of size 2 and maximum entry 3
sage: list(CT)
[[[1], [2]],
 [[1], [3]],
 [[2], [3]],
 [[1, 1]],
 [[2, 1]],
 [[2, 2]],
 [[3, 1]],
 [[3, 2]],
 [[3, 3]]]

sage: CT = CompositionTableaux(0); CT
Composition Tableaux of size 0 and maximum entry 0
sage: list(CT)
[[]]
```

```
>>> from sage.all import *
>>> CT = CompositionTableaux(Integer(3)); CT
Composition Tableaux of size 3 and maximum entry 3
>>> list(CT)
[[[1], [2], [3]],
 [[1], [2, 2]],
 [[1], [3, 2]],
 [[1], [3, 3]],
 [[2], [3, 3]],
 [[1, 1], [2]],
 [[1, 1], [3]],
 [[2, 1], [3]],
 [[2, 2], [3]],
 [[1, 1, 1]],
 [[2, 1, 1]],
 [[2, 2, 1]],
 [[2, 2, 2]],
 [[3, 1, 1]],
 [[3, 2, 1]],
 [[3, 2, 2]],
 [[3, 3, 1]],
 [[3, 3, 2]],
 [[3, 3, 3]]]
```

(continues on next page)

(continued from previous page)

```
>>> CT = CompositionTableaux([Integer(1),Integer(2),Integer(1)]); CT
Composition tableaux of shape [1, 2, 1] and maximum entry 4
>>> list(CT)
[[[1], [2, 2], [3]],
 [[1], [2, 2], [4]],
 [[1], [3, 2], [4]],
 [[1], [3, 3], [4]],
 [[2], [3, 3], [4]]]

>>> CT = CompositionTableaux(shape=[Integer(1),Integer(2),Integer(1)],max_
↪entry=Integer(3)); CT
Composition tableaux of shape [1, 2, 1] and maximum entry 3
>>> list(CT)
[[[1], [2, 2], [3]]]

>>> CT = CompositionTableaux(Integer(2),max_entry=Integer(3)); CT
Composition Tableaux of size 2 and maximum entry 3
>>> list(CT)
[[[1], [2]],
 [[1], [3]],
 [[2], [3]],
 [[1, 1]],
 [[2, 1]],
 [[2, 2]],
 [[3, 1]],
 [[3, 2]],
 [[3, 3]]]

>>> CT = CompositionTableaux(Integer(0)); CT
Composition Tableaux of size 0 and maximum entry 0
>>> list(CT)
[[]]
```

Element

alias of *CompositionTableau*

class sage.combinat.composition_tableau.**CompositionTableauxBacktracker**(*shape*, *max_entry=None*)

Bases: *GenericBacktracker*

A backtracker class for generating sets of composition tableaux.

get_next_pos(*ii*, *jj*)

EXAMPLES:

```
sage: from sage.combinat.composition_tableau import␣
↪CompositionTableauxBacktracker
sage: T = CompositionTableau([[2,1],[5,4,3,2],[6],[7,7,6]])
sage: n = CompositionTableauxBacktracker(T.shape_composition())
sage: n.get_next_pos(1,1)
(1, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.composition_tableau import␣
↪CompositionTableauxBacktracker
>>> T = CompositionTableau([[Integer(2),Integer(1)],[Integer(5),Integer(4),
↪Integer(3),Integer(2)],[Integer(6)],[Integer(7),Integer(7),Integer(6)]])
>>> n = CompositionTableauxBacktracker(T.shape_composition())
>>> n.get_next_pos(Integer(1),Integer(1))
(1, 2)
```

class sage.combinat.composition_tableau.**CompositionTableaux_all**(*max_entry=None*)

Bases: *CompositionTableaux*, DisjointUnionEnumeratedSets

All composition tableaux.

an_element()

Return a particular element of self.

EXAMPLES:

```
sage: CT = CompositionTableaux()
sage: CT.an_element()
[[1, 1], [2]]
```

```
>>> from sage.all import *
>>> CT = CompositionTableaux()
>>> CT.an_element()
[[1, 1], [2]]
```

class sage.combinat.composition_tableau.**CompositionTableaux_shape**(*comp*, *max_entry=None*)

Bases: *CompositionTableaux*

Composition tableaux of a fixed shape comp with a given max entry.

INPUT:

- comp – a composition
- max_entry – nonnegative integer (default: size of comp)

an_element()

Return a particular element of *CompositionTableaux_shape*.

EXAMPLES:

```
sage: CT = CompositionTableaux([1,2,1])
sage: CT.an_element()
[[1], [2, 2], [3]]
```

```
>>> from sage.all import *
>>> CT = CompositionTableaux([Integer(1),Integer(2),Integer(1)])
>>> CT.an_element()
[[1], [2, 2], [3]]
```

class sage.combinat.composition_tableau.**CompositionTableaux_size**(*n*, *max_entry=None*)

Bases: *CompositionTableaux*

Composition tableaux of a fixed size n.

INPUT:

- n – nonnegative integer
- max_entry – nonnegative integer (default: n)

OUTPUT: the class of composition tableaux of size n

5.1.32 Constellations

A constellation is a tuple $(g_0, g_1, \dots, g_k)$ of permutations such that the product $g_0 g_1 \dots g_k$ is the identity. One often assumes that the group generated by $g_0, g_1, \dots, g_k$ acts transitively ([LZ2004] definition 1). Geometrically, it corresponds to a covering of the 2-sphere ramified over k points (the transitivity condition corresponds to the connectivity of the covering).

EXAMPLES:

```
sage: c = Constellation(['(1,2)', '(1,3)', None])
sage: c
Constellation of length 3 and degree 3
g0 (1,2)(3)
g1 (1,3)(2)
g2 (1,3,2)
sage: C = Constellations(3,4); C
Connected constellations of length 3 and degree 4 on {1, 2, 3, 4}
sage: C.cardinality()  # long time
426

sage: C = Constellations(3, 4, domain=('a', 'b', 'c', 'd'))
sage: C
Connected constellations of length 3 and degree 4 on {'a', 'b', 'c', 'd'}
sage: c = C(('a','c'),(('b','c'),('a','d')), None)
sage: c
Constellation of length 3 and degree 4
g0 ('a','c')('b')('d')
g1 ('a','d')('b','c')
g2 ('a','d','c','b')
sage: c.is_connected()
True
sage: c.euler_characteristic()
2
sage: TestSuite(C).run()
```

```
>>> from sage.all import *
>>> c = Constellation(['(1,2)', '(1,3)', None])
>>> c
Constellation of length 3 and degree 3
g0 (1,2)(3)
g1 (1,3)(2)
g2 (1,3,2)
>>> C = Constellations(Integer(3),Integer(4)); C
Connected constellations of length 3 and degree 4 on {1, 2, 3, 4}
>>> C.cardinality()  # long time
426

```

(continues on next page)

(continued from previous page)

```
>>> C = Constellations(Integer(3), Integer(4), domain=('a', 'b', 'c', 'd'))
>>> C
Connected constellations of length 3 and degree 4 on {'a', 'b', 'c', 'd'}
>>> c = C(('a','c'),(('b','c'),('a','d')), None)
>>> c
Constellation of length 3 and degree 4
g0 ('a','c')('b')('d')
g1 ('a','d')('b','c')
g2 ('a','d','c','b')
>>> c.is_connected()
True
>>> c.euler_characteristic()
2
>>> TestSuite(C).run()
```

sage.combinat.constellation.**Constellation**(*g=None*, *mutable=False*, *connected=True*, *check=True*)

Constellation.

INPUT:

- `g` – list of permutations
- `mutable` – boolean (default: `False`); whether the result is mutable or not
- `connected` – boolean (default: `True`); whether the result should be connected
- `check` – boolean (default: `True`); whether or not to check. If it is `True`, then the list `g` must contain no `None`.

EXAMPLES:

Simple initialization:

```
sage: Constellation(['(0,1)','(0,3)(1,2)','(0,3,1,2)'])
Constellation of length 3 and degree 4
g0 (0,1)(2)(3)
g1 (0,3)(1,2)
g2 (0,3,1,2)
```

```
>>> from sage.all import *
>>> Constellation(['(0,1)','(0,3)(1,2)','(0,3,1,2)'])
Constellation of length 3 and degree 4
g0 (0,1)(2)(3)
g1 (0,3)(1,2)
g2 (0,3,1,2)
```

One of the permutation can be omitted:

```
sage: Constellation(['(0,1)', None, '(0,4)(1,2,3)'])
Constellation of length 3 and degree 5
g0 (0,1)(2)(3)(4)
g1 (0,3,2,1,4)
g2 (0,4)(1,2,3)
```

```
>>> from sage.all import *
>>> Constellation(['(0,1)', None, '(0,4)(1,2,3)'])
Constellation of length 3 and degree 5
g0 (0,1)(2)(3)(4)
g1 (0,3,2,1,4)
g2 (0,4)(1,2,3)
```

One can define mutable constellations:

```
sage: Constellation(([0,2,1], [2,1,0], [1,2,0]), mutable=True)
Constellation of length 3 and degree 3
g0 (0)(1,2)
g1 (0,2)(1)
g2 (0,1,2)
```

```
>>> from sage.all import *
>>> Constellation(([Integer(0),Integer(2),Integer(1)], [Integer(2),Integer(1),
↪Integer(0)], [Integer(1),Integer(2),Integer(0)]), mutable=True)
Constellation of length 3 and degree 3
g0 (0)(1,2)
g1 (0,2)(1)
g2 (0,1,2)
```

class sage.combinat.constellation.**Constellation_class**(*parent*, *g*, *connected*, *mutable*, *check*)

Bases: Element

Constellation.

A constellation or a tuple of permutations $(g_0, g_1, ..., g_k)$ such that the product $g_0 g_1 ... g_k$ is the identity.

braid_group_action(*i*)

Act on self as the braid group generator that exchanges position i and $i + 1$.

INPUT:

- i – integer in $[0, n - 1]$ where n is the length of self

EXAMPLES:

```
sage: sigma = lambda c, i: c.braid_group_action(i)

sage: c = Constellation(['(0,1)(2,3,4)','(1,4)',None]); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2)(3)
g2 (0,1,3,2,4)
sage: sigma(c, 1)
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0,1,3,2,4)
g2 (0,3)(1)(2)(4)
```

```
>>> from sage.all import *
>>> sigma = lambda c, i: c.braid_group_action(i)
```

(continues on next page)

(continued from previous page)

```
>>> c = Constellation(['(0,1)(2,3,4)','(1,4)',None]); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2)(3)
g2 (0,1,3,2,4)
>>> sigma(c, Integer(1))
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0,1,3,2,4)
g2 (0,3)(1)(2)(4)
```

Check the commutation relation:

```
sage: c = Constellation(['(0,1)(2,3,4)','(1,4)','(2,5)(0,4)',None])
sage: d = Constellation(['(0,1,3,5)','(2,3,4)','(0,3,5)',None])
sage: c13 = sigma(sigma(c, 0), 2)
sage: c31 = sigma(sigma(c, 2), 0)
sage: c13 == c31
True
sage: d13 = sigma(sigma(d, 0), 2)
sage: d31 = sigma(sigma(d, 2), 0)
sage: d13 == d31
True
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)(2,3,4)','(1,4)','(2,5)(0,4)',None])
>>> d = Constellation(['(0,1,3,5)','(2,3,4)','(0,3,5)',None])
>>> c13 = sigma(sigma(c, Integer(0)), Integer(2))
>>> c31 = sigma(sigma(c, Integer(2)), Integer(0))
>>> c13 == c31
True
>>> d13 = sigma(sigma(d, Integer(0)), Integer(2))
>>> d31 = sigma(sigma(d, Integer(2)), Integer(0))
>>> d13 == d31
True
```

Check the braid relation:

```
sage: c121 = sigma(sigma(sigma(c, 1), 2), 1)
sage: c212 = sigma(sigma(sigma(c, 2), 1), 2)
sage: c121 == c212
True
sage: d121 = sigma(sigma(sigma(d, 1), 2), 1)
sage: d212 = sigma(sigma(sigma(d, 2), 1), 2)
sage: d121 == d212
True
```

```
>>> from sage.all import *
>>> c121 = sigma(sigma(sigma(c, Integer(1)), Integer(2)), Integer(1))
>>> c212 = sigma(sigma(sigma(c, Integer(2)), Integer(1)), Integer(2))
>>> c121 == c212
True
```

(continues on next page)

(continued from previous page)

```
>>> d121 = sigma(sigma(sigma(d, Integer(1)), Integer(2)), Integer(1))
>>> d212 = sigma(sigma(sigma(d, Integer(2)), Integer(1)), Integer(2))
>>> d121 == d212
True
```

braid_group_orbit()

Return the graph of the action of the braid group.

The action is considered up to isomorphism of constellation.

EXAMPLES:

```
sage: c = Constellation(['(0,1)(2,3,4)','(1,4)',None]); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2)(3)
g2 (0,1,3,2,4)
sage: G = c.braid_group_orbit()
sage: G.num_verts()
4
sage: G.num_edges()
12
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)(2,3,4)','(1,4)',None]); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2)(3)
g2 (0,1,3,2,4)
>>> G = c.braid_group_orbit()
>>> G.num_verts()
4
>>> G.num_edges()
12
```

connected_components()

Return the connected components.

OUTPUT: list of connected constellations

EXAMPLES:

```
sage: c = Constellation(['(0,1)(2)', None, '(0,1)(2)'], connected=False)
sage: cc = c.connected_components(); cc
[Constellation of length 3 and degree 2
g0 (0,1)
g1 (0)(1)
g2 (0,1),
Constellation of length 3 and degree 1
g0 (0)
g1 (0)
g2 (0)]
sage: all(c2.is_connected() for c2 in cc)
True
```

(continues on next page)

(continued from previous page)

```
sage: c = Constellation(['(0,1,2)', None], connected=False)
sage: c.connected_components()
[Constellation of length 2 and degree 3
g0 (0,1,2)
g1 (0,2,1)]
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)(2)', None, '(0,1)(2)'], connected=False)
>>> cc = c.connected_components(); cc
[Constellation of length 3 and degree 2
g0 (0,1)
g1 (0)(1)
g2 (0,1),
Constellation of length 3 and degree 1
g0 (0)
g1 (0)
g2 (0)]
>>> all(c2.is_connected() for c2 in cc)
True

>>> c = Constellation(['(0,1,2)', None], connected=False)
>>> c.connected_components()
[Constellation of length 2 and degree 3
g0 (0,1,2)
g1 (0,2,1)]
```

copy()

Return a copy of `self`.

degree()

Return the degree of the constellation.

The degree of a constellation is the number n that corresponds to the symmetric group $S(n)$ in which the permutations of the constellation are defined.

EXAMPLES:

```
sage: c = Constellation([])
sage: c.degree()
0
sage: c = Constellation(['(0,1)',None])
sage: c.degree()
2
sage: c = Constellation(['(0,1)','(0,3,2)(1,5)',None,'(4,3,2,1)'])
sage: c.degree()
6
```

```
>>> from sage.all import *
>>> c = Constellation([])
>>> c.degree()
0
>>> c = Constellation(['(0,1)',None])
```

(continues on next page)

(continued from previous page)

```
>>> c.degree()
2
>>> c = Constellation(['(0,1)','(0,3,2)(1,5)',None,'(4,3,2,1)'])
>>> c.degree()
6
```

euler_characteristic()

Return the Euler characteristic of the surface.

ALGORITHM:

Hurwitz formula

EXAMPLES:

```
sage: c = Constellation(['(0,1)', '(0,2)', None])
sage: c.euler_characteristic()
2

sage: c = Constellation(['(0,1,2,3)','(1,3,0,2)', '(0,3,1,2)', None])
sage: c.euler_characteristic()
-4
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)', '(0,2)', None])
>>> c.euler_characteristic()
2

>>> c = Constellation(['(0,1,2,3)','(1,3,0,2)', '(0,3,1,2)', None])
>>> c.euler_characteristic()
-4
```

g(*i=None*)

Return the permutation g_i of the constellation.

INPUT:

- `i` – integer or `None` (default)

If `None` , return instead the list of all g_i.

EXAMPLES:

```
sage: c = Constellation(['(0,1,2)(3,4)','(0,3)',None])
sage: c.g(0)
(0,1,2)(3,4)
sage: c.g(1)
(0,3)
sage: c.g(2)
(0,4,3,2,1)
sage: c.g()
[(0,1,2)(3,4), (0,3), (0,4,3,2,1)]
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1,2)(3,4)','(0,3)',None])
```

(continues on next page)

(continued from previous page)

```
>>> c.g(Integer(0))
(0,1,2)(3,4)
>>> c.g(Integer(1))
(0,3)
>>> c.g(Integer(2))
(0,4,3,2,1)
>>> c.g()
[(0,1,2)(3,4), (0,3), (0,4,3,2,1)]
```

genus()

Return the genus of the surface.

EXAMPLES:

```
sage: c = Constellation(['(0,1)', '(0,2)', None])
sage: c.genus()
0

sage: c = Constellation(['(0,1)(2,3,4)','(1,3,4)(2,0)', None])
sage: c.genus()
1
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)', '(0,2)', None])
>>> c.genus()
0

>>> c = Constellation(['(0,1)(2,3,4)','(1,3,4)(2,0)', None])
>>> c.genus()
1
```

is_connected()

Test of connectedness.

EXAMPLES:

```
sage: c = Constellation(['(0,1)(2)', None, '(0,1)(2)'], connected=False)
sage: c.is_connected()
False
sage: c = Constellation(['(0,1,2)', None], connected=False)
sage: c.is_connected()
True
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)(2)', None, '(0,1)(2)'], connected=False)
>>> c.is_connected()
False
>>> c = Constellation(['(0,1,2)', None], connected=False)
>>> c.is_connected()
True
```

is_isomorphic(*other*, *return_map=False*)

Test of isomorphism.

Return `True` if the constellations are isomorphic (*i.e.* related by a common conjugacy) and return the permutation that conjugate the two permutations if `return_map` is `True` in such a way that `self.relabel(m) == other`.

ALGORITHM:

uses canonical labels obtained from the method `relabel()`.

EXAMPLES:

```
sage: c = Constellation([[1,0,2],[2,1,0],[0,2,1],None])
sage: d = Constellation([[2,1,0],[0,2,1],[1,0,2],None])
sage: answer, mapping = c.is_isomorphic(d,return_map=True)
sage: answer
True
sage: c.relabel(mapping) == d
True
```

```
>>> from sage.all import *
>>> c = Constellation([[Integer(1),Integer(0),Integer(2)],[Integer(2),
↪Integer(1),Integer(0)],[Integer(0),Integer(2),Integer(1)],None])
>>> d = Constellation([[Integer(2),Integer(1),Integer(0)],[Integer(0),
↪Integer(2),Integer(1)],[Integer(1),Integer(0),Integer(2)],None])
>>> answer, mapping = c.is_isomorphic(d,return_map=True)
>>> answer
True
>>> c.relabel(mapping) == d
True
```

is_mutable()

Return `False` as `self` is immutable.

EXAMPLES:

```
sage: c = Constellation(([0,2,1],[2,1,0],[1,2,0]), mutable=False)
sage: c.is_mutable()
False
```

```
>>> from sage.all import *
>>> c = Constellation(([Integer(0),Integer(2),Integer(1)],[Integer(2),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]), mutable=False)
>>> c.is_mutable()
False
```

length()

Return the number of permutations.

EXAMPLES:

```
sage: c = Constellation(['(0,1)','(0,2)','(0,3)',None])
sage: c.length()
4
sage: c = Constellation(['(0,1,3)',None,'(1,2)'])
sage: c.length()
3
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)','(0,2)','(0,3)',None])
>>> c.length()
4
>>> c = Constellation(['(0,1,3)',None,'(1,2)'])
>>> c.length()
3
```

mutable_copy()

Return a mutable copy of `self`.

EXAMPLES:

```
sage: c = Constellation(([0,2,1],[2,1,0],[1,2,0]), mutable=False)
sage: d = c.mutable_copy()
sage: d.is_mutable()
True
```

```
>>> from sage.all import *
>>> c = Constellation(([Integer(0),Integer(2),Integer(1)],[Integer(2),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]), mutable=False)
>>> d = c.mutable_copy()
>>> d.is_mutable()
True
```

passport(*i=None*)

Return the profile of `self`.

The profile of a constellation is the tuple of partitions associated to the conjugacy classes of the permutations of the constellation.

This is also called the passport.

EXAMPLES:

```
sage: c = Constellation(['(0,1,2)(3,4)','(0,3)',None])
sage: c.profile()
([3, 2], [2, 1, 1, 1], [5])
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1,2)(3,4)','(0,3)',None])
>>> c.profile()
([3, 2], [2, 1, 1, 1], [5])
```

profile(*i=None*)

Return the profile of `self`.

The profile of a constellation is the tuple of partitions associated to the conjugacy classes of the permutations of the constellation.

This is also called the passport.

EXAMPLES:

```
sage: c = Constellation(['(0,1,2)(3,4)','(0,3)',None])
sage: c.profile()
([3, 2], [2, 1, 1, 1], [5])
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1,2)(3,4)','(0,3)',None])
>>> c.profile()
([3, 2], [2, 1, 1, 1], [5])
```

relabel(*perm=None*, *return_map=False*)

Relabel `self`.

If `perm` is provided then relabel with respect to `perm`. Otherwise use canonical labels. In that case, if `return_map` is provided, the return also the map used for canonical labels.

Algorithm:

the cycle for g(0) are adjacent and the cycle are joined with respect to the other permutations. The minimum is taken for all possible renumerotations.

EXAMPLES:

```
sage: c = Constellation(['(0,1)(2,3,4)','(1,4)',None]); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2)(3)
g2 (0,1,3,2,4)
sage: c2 = c.relabel(); c2
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,2)(3)(4)
g2 (0,1,4,3,2)
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)(2,3,4)','(1,4)',None]); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2)(3)
g2 (0,1,3,2,4)
>>> c2 = c.relabel(); c2
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,2)(3)(4)
g2 (0,1,4,3,2)
```

The map returned when the option `return_map` is set to `True` can be used to set the relabelling:

```
sage: c3, perm = c.relabel(return_map=True)
sage: c3 == c2 and c3 == c.relabel(perm=perm)
True

sage: S5 = SymmetricGroup(range(5))
sage: d = c.relabel(S5([4,3,1,0,2])); d
Constellation of length 3 and degree 5
```

(continues on next page)

(continued from previous page)

```
g0 (0,2,1)(3,4)
g1 (0)(1)(2,3)(4)
g2 (0,1,2,4,3)
sage: d.is_isomorphic(c)
True
```

```
>>> from sage.all import *
>>> c3, perm = c.relabel(return_map=True)
>>> c3 == c2 and c3 == c.relabel(perm=perm)
True

>>> S5 = SymmetricGroup(range(Integer(5)))
>>> d = c.relabel(S5([Integer(4),Integer(3),Integer(1),Integer(0),
↪Integer(2)])); d
Constellation of length 3 and degree 5
g0 (0,2,1)(3,4)
g1 (0)(1)(2,3)(4)
g2 (0,1,2,4,3)
>>> d.is_isomorphic(c)
True
```

We check that after a random relabelling the new constellation is isomorphic to the initial one:

```
sage: c = Constellation(['(0,1)(2,3,4)','(1,4)',None])
sage: p = S5.random_element()
sage: cc = c.relabel(perm=p)
sage: cc.is_isomorphic(c)
True
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)(2,3,4)','(1,4)',None])
>>> p = S5.random_element()
>>> cc = c.relabel(perm=p)
>>> cc.is_isomorphic(c)
True
```

Check that it works for “non standard” labels:

```
sage: c = Constellation([(('a','b'),('c','d','e')),('b','d'), None])
sage: c.relabel()
Constellation of length 3 and degree 5
g0 ('a','b')('c','d','e')
g1 ('a')('b','c')('d')('e')
g2 ('a','b','e','d','c')
```

```
>>> from sage.all import *
>>> c = Constellation([(('a','b'),('c','d','e')),('b','d'), None])
>>> c.relabel()
Constellation of length 3 and degree 5
g0 ('a','b')('c','d','e')
g1 ('a')('b','c')('d')('e')
g2 ('a','b','e','d','c')
```

set_immutable()

Do nothing, as `self` is already immutable.

EXAMPLES:

```
sage: c = Constellation(([0,2,1],[2,1,0],[1,2,0]), mutable=False)
sage: c.set_immutable()
sage: c.is_mutable()
False
```

```
>>> from sage.all import *
>>> c = Constellation(([Integer(0),Integer(2),Integer(1)],[Integer(2),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]), mutable=False)
>>> c.set_immutable()
>>> c.is_mutable()
False
```

switch(*i*, *j0*, *j1*)

Perform the multiplication by the transposition $(j0, j1)$ between the permutations g_i and g_{i+1}.

The modification is local in the sense that it modifies g_i and g_{i+1} but does not modify the product $g_i g_{i+1}$. The new constellation is

$$(g_0, \ldots, g_{i-1}, g_i(j0j1), (j0j1)g_{i+1}, g_{i+2}, \ldots, g_k)$$

EXAMPLES:

```
sage: c = Constellation(['(0,1)(2,3,4)','(1,4)',None], mutable=True); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2)(3)
g2 (0,1,3,2,4)
sage: c.is_mutable()
True
sage: c.switch(1,2,3); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2,3)
g2 (0,1,3,4)(2)
sage: c._check()
sage: c.switch(2,1,3); c
Constellation of length 3 and degree 5
g0 (0,1,4,2,3)
g1 (0)(1,4)(2,3)
g2 (0,3,4)(1)(2)
sage: c._check()
sage: c.switch(0,0,1); c
Constellation of length 3 and degree 5
g0 (0)(1,4,2,3)
g1 (0,4,1)(2,3)
g2 (0,3,4)(1)(2)
sage: c._check()
```

```
>>> from sage.all import *
>>> c = Constellation(['(0,1)(2,3,4)','(1,4)',None], mutable=True); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2)(3)
g2 (0,1,3,2,4)
>>> c.is_mutable()
True
>>> c.switch(Integer(1),Integer(2),Integer(3)); c
Constellation of length 3 and degree 5
g0 (0,1)(2,3,4)
g1 (0)(1,4)(2,3)
g2 (0,1,3,4)(2)
>>> c._check()
>>> c.switch(Integer(2),Integer(1),Integer(3)); c
Constellation of length 3 and degree 5
g0 (0,1,4,2,3)
g1 (0)(1,4)(2,3)
g2 (0,3,4)(1)(2)
>>> c._check()
>>> c.switch(Integer(0),Integer(0),Integer(1)); c
Constellation of length 3 and degree 5
g0 (0)(1,4,2,3)
g1 (0,4,1)(2,3)
g2 (0,3,4)(1)(2)
>>> c._check()
```

sage.combinat.constellation.**Constellations**(**data*, ***options*)

Build a set of constellations.

INPUT:

- `profile` – an optional profile
- `length` – an optional length
- `degree` – an optional degree
- `connected` – an optional boolean

EXAMPLES:

```
sage: Constellations(4,2)
Connected constellations of length 4 and degree 2 on {1, 2}

sage: Constellations([[3,2,1],[3,3],[3,3]])
Connected constellations with profile ([3, 2, 1], [3, 3], [3, 3]) on {1, 2, 3, 4,
↪5, 6}
```

```
>>> from sage.all import *
>>> Constellations(Integer(4),Integer(2))
Connected constellations of length 4 and degree 2 on {1, 2}

>>> Constellations([[Integer(3),Integer(2),Integer(1)],[Integer(3),Integer(3)],
↪[Integer(3),Integer(3)]])
```

(continues on next page)

(continued from previous page)

```
Connected constellations with profile ([3, 2, 1], [3, 3], [3, 3]) on {1, 2, 3, 4,␣
↪5, 6}
```

class sage.combinat.constellation.**Constellations_ld**(*length*, *degree*, *sym=None*, *connected=True*)

Bases: UniqueRepresentation, Parent

Constellations of given length and degree.

EXAMPLES:

```
sage: C = Constellations(2,3); C
Connected constellations of length 2 and degree 3 on {1, 2, 3}
sage: C([[2,3,1],[3,1,2]])
Constellation of length 2 and degree 3
g0 (1,2,3)
g1 (1,3,2)
sage: C.cardinality()
2
sage: Constellations(2,3,connected=False).cardinality()
6
```

```
>>> from sage.all import *
>>> C = Constellations(Integer(2),Integer(3)); C
Connected constellations of length 2 and degree 3 on {1, 2, 3}
>>> C([[Integer(2),Integer(3),Integer(1)],[Integer(3),Integer(1),Integer(2)]])
Constellation of length 2 and degree 3
g0 (1,2,3)
g1 (1,3,2)
>>> C.cardinality()
2
>>> Constellations(Integer(2),Integer(3),connected=False).cardinality()
6
```

Element

alias of *Constellation_class*

braid_group_action()

Return a list of graphs that corresponds to the braid group action on `self` up to isomorphism.

OUTPUT: list of graphs

EXAMPLES:

```
sage: C = Constellations(3,3)
sage: C.braid_group_action()
[Looped multi-digraph on 3 vertices,
 Looped multi-digraph on 1 vertex,
 Looped multi-digraph on 3 vertices]
```

```
>>> from sage.all import *
>>> C = Constellations(Integer(3),Integer(3))
>>> C.braid_group_action()
[Looped multi-digraph on 3 vertices,
```

(continues on next page)

(continued from previous page)

```
Looped multi-digraph on 1 vertex,
Looped multi-digraph on 3 vertices]
```

braid_group_orbits()

Return the orbits under the action of braid group.

EXAMPLES:

```
sage: C = Constellations(3,3)
sage: O = C.braid_group_orbits()
sage: len(O)
3
sage: [x.profile() for x in O[0]]
[([1, 1, 1], [3], [3]), ([3], [1, 1, 1], [3]), ([3], [3], [1, 1, 1])]
sage: [x.profile() for x in O[1]]
[([3], [3], [3])]
sage: [x.profile() for x in O[2]]
[([2, 1], [2, 1], [3]), ([2, 1], [3], [2, 1]), ([3], [2, 1], [2, 1])]
```

```
>>> from sage.all import *
>>> C = Constellations(Integer(3),Integer(3))
>>> O = C.braid_group_orbits()
>>> len(O)
3
>>> [x.profile() for x in O[Integer(0)]]
[([1, 1, 1], [3], [3]), ([3], [1, 1, 1], [3]), ([3], [3], [1, 1, 1])]
>>> [x.profile() for x in O[Integer(1)]]
[([3], [3], [3])]
>>> [x.profile() for x in O[Integer(2)]]
[([2, 1], [2, 1], [3]), ([2, 1], [3], [2, 1]), ([3], [2, 1], [2, 1])]
```

is_empty()

Return whether this set of constellations is empty.

EXAMPLES:

```
sage: Constellations(2, 3).is_empty()
False
sage: Constellations(1, 2).is_empty()
True
sage: Constellations(1, 2, connected=False).is_empty()
False
```

```
>>> from sage.all import *
>>> Constellations(Integer(2), Integer(3)).is_empty()
False
>>> Constellations(Integer(1), Integer(2)).is_empty()
True
>>> Constellations(Integer(1), Integer(2), connected=False).is_empty()
False
```

random_element(*mutable=False*)

Return a random element.

This is found by trial and rejection, starting from a random list of permutations.

EXAMPLES:

```
sage: const = Constellations(3,3)
sage: const.random_element()
Constellation of length 3 and degree 3
...
...
...
sage: c = const.random_element()
sage: c.degree() == 3 and c.length() == 3
True
```

```
>>> from sage.all import *
>>> const = Constellations(Integer(3),Integer(3))
>>> const.random_element()
Constellation of length 3 and degree 3
...
...
...
>>> c = const.random_element()
>>> c.degree() == Integer(3) and c.length() == Integer(3)
True
```

class sage.combinat.constellation.**Constellations_p**(*profile*, *domain=None*, *connected=True*)

Bases: UniqueRepresentation, Parent

Constellations with fixed profile.

EXAMPLES:

```
sage: C = Constellations([[3,1],[3,1],[2,2]]); C
Connected constellations with profile ([3, 1], [3, 1], [2, 2]) on {1, 2, 3, 4}
sage: C.cardinality()
24
sage: C.first()
Constellation of length 3 and degree 4
g0 (1)(2,3,4)
g1 (1,2,3)(4)
g2 (1,2)(3,4)
sage: C.last()
Constellation of length 3 and degree 4
g0 (1,4,3)(2)
g1 (1,4,2)(3)
g2 (1,2)(3,4)
```

```
>>> from sage.all import *
>>> C = Constellations([[Integer(3),Integer(1)],[Integer(3),Integer(1)],
↪[Integer(2),Integer(2)]]); C
Connected constellations with profile ([3, 1], [3, 1], [2, 2]) on {1, 2, 3, 4}
>>> C.cardinality()
24
>>> C.first()
```

(continues on next page)

(continued from previous page)

```
Constellation of length 3 and degree 4
g0 (1)(2,3,4)
g1 (1,2,3)(4)
g2 (1,2)(3,4)
>>> C.last()
Constellation of length 3 and degree 4
g0 (1,4,3)(2)
g1 (1,4,2)(3)
g2 (1,2)(3,4)
```

Note that the cardinality can also be computed using characters of the symmetric group (Frobenius formula):

```
sage: P = Partitions(4)
sage: p1 = Partition([3,1])
sage: p2 = Partition([3,1])
sage: p3 = Partition([2,2])
sage: i1 = P.cardinality() - P.rank(p1) - 1
sage: i2 = P.cardinality() - P.rank(p2) - 1
sage: i3 = P.cardinality() - P.rank(p3) - 1
sage: s = 0
sage: for c in SymmetricGroup(4).irreducible_characters():
....:     v = c.values()
....:     s += v[i1] * v[i2] * v[i3] / v[0]
sage: c1 = p1.conjugacy_class_size()
sage: c2 = p2.conjugacy_class_size()
sage: c3 = p3.conjugacy_class_size()
sage: c1 * c2 * c3 / factorial(4)**2 * s
1
```

```
>>> from sage.all import *
>>> P = Partitions(Integer(4))
>>> p1 = Partition([Integer(3),Integer(1)])
>>> p2 = Partition([Integer(3),Integer(1)])
>>> p3 = Partition([Integer(2),Integer(2)])
>>> i1 = P.cardinality() - P.rank(p1) - Integer(1)
>>> i2 = P.cardinality() - P.rank(p2) - Integer(1)
>>> i3 = P.cardinality() - P.rank(p3) - Integer(1)
>>> s = Integer(0)
>>> for c in SymmetricGroup(Integer(4)).irreducible_characters():
...     v = c.values()
...     s += v[i1] * v[i2] * v[i3] / v[Integer(0)]
>>> c1 = p1.conjugacy_class_size()
>>> c2 = p2.conjugacy_class_size()
>>> c3 = p3.conjugacy_class_size()
>>> c1 * c2 * c3 / factorial(Integer(4))**Integer(2) * s
1
```

The number obtained above is up to isomorphism. And we can check:

```
sage: len(C.isomorphism_representatives())
1
```

```
>>> from sage.all import *
>>> len(C.isomorphism_representatives())
1
```

isomorphism_representatives()

Return a set of isomorphism representative of `self`.

EXAMPLES:

```
sage: C = Constellations([[5], [4,1], [3,2]])
sage: C.cardinality()
240
sage: ir = sorted(C.isomorphism_representatives())
sage: len(ir)
2
sage: ir[0]
Constellation of length 3 and degree 5
g0 (1,2,3,4,5)
g1 (1)(2,3,4,5)
g2 (1,5,3)(2,4)
sage: ir[1]
Constellation of length 3 and degree 5
g0 (1,2,3,4,5)
g1 (1)(2,5,3,4)
g2 (1,5)(2,3,4)
```

```
>>> from sage.all import *
>>> C = Constellations([[Integer(5)], [Integer(4),Integer(1)], [Integer(3),
↪Integer(2)]])
>>> C.cardinality()
240
>>> ir = sorted(C.isomorphism_representatives())
>>> len(ir)
2
>>> ir[Integer(0)]
Constellation of length 3 and degree 5
g0 (1,2,3,4,5)
g1 (1)(2,3,4,5)
g2 (1,5,3)(2,4)
>>> ir[Integer(1)]
Constellation of length 3 and degree 5
g0 (1,2,3,4,5)
g1 (1)(2,5,3,4)
g2 (1,5)(2,3,4)
```

sage.combinat.constellation.**perm_conjugate**(p, s)

Return the conjugate of the permutation p by the permutation s.

INPUT:

- `p, s` – two permutations of {0,..,n-1} given by lists of values

OUTPUT: a permutation of {0,..,n-1} given by a list of values

EXAMPLES:

```
sage: from sage.combinat.constellation import perm_conjugate
sage: perm_conjugate([3,1,2,0], [3,2,0,1])
[0, 3, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.constellation import perm_conjugate
>>> perm_conjugate([Integer(3),Integer(1),Integer(2),Integer(0)], [Integer(3),
↪Integer(2),Integer(0),Integer(1)])
[0, 3, 2, 1]
```

sage.combinat.constellation.**perm_invert**(*p*)

Return the inverse of the permutation p.

INPUT:

- p – a permutation of {0,..,n-1} given by a list of values

OUTPUT: a permutation of {0,..,n-1} given by a list of values

EXAMPLES:

```
sage: from sage.combinat.constellation import perm_invert
sage: perm_invert([3,2,0,1])
[2, 3, 1, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.constellation import perm_invert
>>> perm_invert([Integer(3),Integer(2),Integer(0),Integer(1)])
[2, 3, 1, 0]
```

sage.combinat.constellation.**perm_sym_domain**(*g*)

Return the domain of a single permutation (before initialization).

EXAMPLES:

```
sage: from sage.combinat.constellation import perm_sym_domain
sage: perm_sym_domain([1,2,3,4])
{1, 2, 3, 4}
sage: perm_sym_domain(((1,2),(0,4)))
{0, 1, 2, 4}
sage: sorted(perm_sym_domain('(1,2,0,5)'))
[0, 1, 2, 5]
```

```
>>> from sage.all import *
>>> from sage.combinat.constellation import perm_sym_domain
>>> perm_sym_domain([Integer(1),Integer(2),Integer(3),Integer(4)])
{1, 2, 3, 4}
>>> perm_sym_domain(((Integer(1),Integer(2)),(Integer(0),Integer(4))))
{0, 1, 2, 4}
>>> sorted(perm_sym_domain('(1,2,0,5)'))
[0, 1, 2, 5]
```

sage.combinat.constellation.**perms_are_connected**(*g*, *n*)

Check that the action of the generated group is transitive.

INPUT:

- g – list of permutations of $[0, n-1]$ (in a SymmetricGroup)
- n – integer

EXAMPLES:

```
sage: from sage.combinat.constellation import perms_are_connected
sage: S = SymmetricGroup(range(3))
sage: perms_are_connected([S([0,1,2]),S([0,2,1])],3)
False
sage: perms_are_connected([S([0,1,2]),S([1,2,0])],3)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.constellation import perms_are_connected
>>> S = SymmetricGroup(range(Integer(3)))
>>> perms_are_connected([S([Integer(0),Integer(1),Integer(2)]),S([Integer(0),
↪Integer(2),Integer(1)])],Integer(3))
False
>>> perms_are_connected([S([Integer(0),Integer(1),Integer(2)]),S([Integer(1),
↪Integer(2),Integer(0)])],Integer(3))
True
```

sage.combinat.constellation.**perms_canonical_labels**(*p*, *e=None*)

Relabel a list with a common conjugation such that two conjugated lists are relabeled the same way.

INPUT:

- p – list of at least 2 permutations
- e – None or a list of integer in the domain of the permutations. If provided, then the renumbering algorithm is only performed from the elements of e.

OUTPUT: a pair made of a list of permutations (as a list of lists) and a list that corresponds to the conjugacy used.

EXAMPLES:

```
sage: from sage.combinat.constellation import perms_canonical_labels
sage: l0 = [[2,0,3,1], [3,1,2,0], [0,2,1,3]]
sage: l, m = perms_canonical_labels(l0); l
[[1, 2, 3, 0], [0, 3, 2, 1], [2, 1, 0, 3]]

sage: S = SymmetricGroup(range(4))
sage: [~S(m) * S(u) * S(m) for u in l0] == list(map(S, l))
True

sage: perms_canonical_labels([])
Traceback (most recent call last):
...
ValueError: input must have length >= 2
```

```
>>> from sage.all import *
>>> from sage.combinat.constellation import perms_canonical_labels
>>> l0 = [[Integer(2),Integer(0),Integer(3),Integer(1)], [Integer(3),Integer(1),
↪Integer(2),Integer(0)], [Integer(0),Integer(2),Integer(1),Integer(3)]]
>>> l, m = perms_canonical_labels(l0); l
```

(continues on next page)

(continued from previous page)

```
[[1, 2, 3, 0], [0, 3, 2, 1], [2, 1, 0, 3]]

>>> S = SymmetricGroup(range(Integer(4)))
>>> [~S(m) * S(u) * S(m) for u in l0] == list(map(S, l))
True

>>> perms_canonical_labels([])
Traceback (most recent call last):
...
ValueError: input must have length >= 2
```

sage.combinat.constellation.**perms_canonical_labels_from**(*x*, *y*, *j0*, *verbose=False*)

Return canonical labels for `x`, `y` that starts at `j0`.

> **Warning**
>
> The group generated by `x` and the elements of `y` should be transitive.

INPUT:

- `x` – list; a permutation of $[0, \ldots, n]$ as a list
- `y` – list of permutations of $[0, \ldots, n]$ as a list of lists
- `j0` – an index in [0, …, n]

OUTPUT: mapping: a permutation that specify the new labels

EXAMPLES:

```
sage: from sage.combinat.constellation import perms_canonical_labels_from
sage: perms_canonical_labels_from([0,1,2],[[1,2,0]], 0)
[0, 1, 2]
sage: perms_canonical_labels_from([1,0,2], [[2,0,1]], 0)
[0, 1, 2]
sage: perms_canonical_labels_from([1,0,2], [[2,0,1]], 1)
[1, 0, 2]
sage: perms_canonical_labels_from([1,0,2], [[2,0,1]], 2)
[2, 1, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.constellation import perms_canonical_labels_from
>>> perms_canonical_labels_from([Integer(0),Integer(1),Integer(2)],[[Integer(1),
→Integer(2),Integer(0)]], Integer(0))
[0, 1, 2]
>>> perms_canonical_labels_from([Integer(1),Integer(0),Integer(2)], [[Integer(2),
→Integer(0),Integer(1)]], Integer(0))
[0, 1, 2]
>>> perms_canonical_labels_from([Integer(1),Integer(0),Integer(2)], [[Integer(2),
→Integer(0),Integer(1)]], Integer(1))
[1, 0, 2]
>>> perms_canonical_labels_from([Integer(1),Integer(0),Integer(2)], [[Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0),Integer(1)]], Integer(2))
[2, 1, 0]
```

sage.combinat.constellation.**perms_sym_init**(*g*, *sym=None*)

Initialize a list of permutations (in the same symmetric group).

OUTPUT:

- `sym` – a symmetric group
- `gg` – list of permutations

EXAMPLES:

```
sage: from sage.combinat.constellation import perms_sym_init
sage: S, g = perms_sym_init([[0,2,1,3], [1,3,2,0]])
sage: S.domain()
{0, 1, 2, 3}
sage: g
[(1,2), (0,1,3)]

sage: S, g = perms_sym_init(['(2,1)', '(0,3)'])
sage: S.domain()
{0, 1, 2, 3}
sage: g
[(1,2), (0,3)]

sage: S, g = perms_sym_init([(1,0), (2,1)])
sage: S.domain()
{0, 1, 2}
sage: g
[(0,1), (1,2)]

sage: S, g = perms_sym_init([((1,0),(2,3)), '(0,1,4)'])
sage: S.domain()
{0, 1, 2, 3, 4}
sage: g
[(0,1)(2,3), (0,1,4)]
```

```
>>> from sage.all import *
>>> from sage.combinat.constellation import perms_sym_init
>>> S, g = perms_sym_init([[Integer(0),Integer(2),Integer(1),Integer(3)],␣
↪[Integer(1),Integer(3),Integer(2),Integer(0)]])
>>> S.domain()
{0, 1, 2, 3}
>>> g
[(1,2), (0,1,3)]

>>> S, g = perms_sym_init(['(2,1)', '(0,3)'])
>>> S.domain()
{0, 1, 2, 3}
>>> g
[(1,2), (0,3)]
```

(continues on next page)

(continued from previous page)

```
>>> S, g = perms_sym_init([(Integer(1),Integer(0)), (Integer(2),Integer(1))])
>>> S.domain()
{0, 1, 2}
>>> g
[(0,1), (1,2)]

>>> S, g = perms_sym_init([((Integer(1),Integer(0)),(Integer(2),Integer(3))), '(0,
↪1,4)'])
>>> S.domain()
{0, 1, 2, 3, 4}
>>> g
[(0,1)(2,3), (0,1,4)]
```

5.1.33 Cores

A k-core is a partition from which no rim hook of size k can be removed. Alternatively, a k-core is an integer partition such that the Ferrers diagram for the partition contains no cells with a hook of size (a multiple of) k.

Authors:

- Anne Schilling and Mike Zabrocki (2011): initial version
- Travis Scrimshaw (2012): Added latex output for Core class

class sage.combinat.core.**Core**(*parent*, *core*)

Bases: *CombinatorialElement*

A k-core is an integer partition from which no rim hook of size k can be removed.

EXAMPLES:

```
sage: c = Core([2,1],4); c
[2, 1]
sage: c = Core([3,1],4); c
Traceback (most recent call last):
...
ValueError: [3, 1] is not a 4-core
```

```
>>> from sage.all import *
>>> c = Core([Integer(2),Integer(1)],Integer(4)); c
[2, 1]
>>> c = Core([Integer(3),Integer(1)],Integer(4)); c
Traceback (most recent call last):
...
ValueError: [3, 1] is not a 4-core
```

affine_symmetric_group_action(*w*, *transposition=False*)

Return the (left) action of the affine symmetric group on self.

INPUT:

- w – tuple of integers $[w_1, \ldots, w_m]$ with $0 \leq w_j < k$. If transposition is set to be True, then $w = [w_0, w_1]$ is interpreted as a transposition t_{w_0,w_1} (see _transposition_to_reduced_word()).
- transposition – boolean (default: False)

The output is the (left) action of the product of the corresponding simple transpositions on `self`, that is $s_{w_1} \cdots s_{w_m}(self)$. See `affine_symmetric_group_simple_action()`.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: c.affine_symmetric_group_action([0,1,0,2,1])
[8, 6, 4, 2]
sage: c.affine_symmetric_group_action([0,2], transposition=True)
[4, 2, 1, 1]

sage: c = Core([11,8,5,5,3,3,1,1,1],4)
sage: c.affine_symmetric_group_action([2,5],transposition=True)
[11, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> c = Core([Integer(4),Integer(2)],Integer(3))
>>> c.affine_symmetric_group_action([Integer(0),Integer(1),Integer(0),
↪Integer(2),Integer(1)])
[8, 6, 4, 2]
>>> c.affine_symmetric_group_action([Integer(0),Integer(2)],␣
↪transposition=True)
[4, 2, 1, 1]

>>> c = Core([Integer(11),Integer(8),Integer(5),Integer(5),Integer(3),
↪Integer(3),Integer(1),Integer(1),Integer(1)],Integer(4))
>>> c.affine_symmetric_group_action([Integer(2),Integer(5)],
↪transposition=True)
[11, 8, 7, 6, 5, 4, 3, 2, 1]
```

affine_symmetric_group_simple_action(*i*)

Return the action of the simple transposition s_i of the affine symmetric group on `self`.

This gives the action of the affine symmetric group of type $A_k^{(1)}$ on the k-core `self`. If `self` has outside (resp. inside) corners of content i modulo k, then these corners are added (resp. removed). Otherwise the action is trivial.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: c.affine_symmetric_group_simple_action(0)                              #␣
↪needs sage.modules
[3, 1]
sage: c.affine_symmetric_group_simple_action(1)                              #␣
↪needs sage.modules
[5, 3, 1]
sage: c.affine_symmetric_group_simple_action(2)                              #␣
↪needs sage.modules
[4, 2]
```

```
>>> from sage.all import *
>>> c = Core([Integer(4),Integer(2)],Integer(3))
>>> c.affine_symmetric_group_simple_action(Integer(0))                       ␣
↪     # needs sage.modules
```

(continues on next page)

(continued from previous page)

```
[3, 1]
>>> c.affine_symmetric_group_simple_action(Integer(1))                      ␣
↪     # needs sage.modules
[5, 3, 1]
>>> c.affine_symmetric_group_simple_action(Integer(2))                      ␣
↪     # needs sage.modules
[4, 2]
```

This action corresponds to the left action by the i-th simple reflection in the affine symmetric group:

```
sage: c = Core([4,2],3)
sage: W = c.to_grassmannian().parent()                                     #␣
↪needs sage.modules
sage: i = 0
sage: (c.affine_symmetric_group_simple_action(i).to_grassmannian()         #␣
↪needs sage.modules
....:      == W.simple_reflection(i)*c.to_grassmannian())
True
sage: i = 1
sage: (c.affine_symmetric_group_simple_action(i).to_grassmannian()         #␣
↪needs sage.modules
....:      == W.simple_reflection(i)*c.to_grassmannian())
True
```

```
>>> from sage.all import *
>>> c = Core([Integer(4),Integer(2)],Integer(3))
>>> W = c.to_grassmannian().parent()                                     #␣
↪needs sage.modules
>>> i = Integer(0)
>>> (c.affine_symmetric_group_simple_action(i).to_grassmannian()         #␣
↪needs sage.modules
...      == W.simple_reflection(i)*c.to_grassmannian())
True
>>> i = Integer(1)
>>> (c.affine_symmetric_group_simple_action(i).to_grassmannian()         #␣
↪needs sage.modules
...      == W.simple_reflection(i)*c.to_grassmannian())
True
```

contains(*other*)

Check whether `self` contains `other`.

INPUT:

- `other` – another k-core or a list

OUTPUT: boolean

This returns `True` if the Ferrers diagram of `self` contains the Ferrers diagram of `other`.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: x = Core([4,2,2,1,1],3)
```

(continues on next page)

(continued from previous page)

```
sage: x.contains(c)
True
sage: c.contains(x)
False
```

```
>>> from sage.all import *
>>> c = Core([Integer(4),Integer(2)],Integer(3))
>>> x = Core([Integer(4),Integer(2),Integer(2),Integer(1),Integer(1)],
↪Integer(3))
>>> x.contains(c)
True
>>> c.contains(x)
False
```

k()

Return k of the k-core `self`.

EXAMPLES:

```
sage: c = Core([2,1],4)
sage: c.k()
4
```

```
>>> from sage.all import *
>>> c = Core([Integer(2),Integer(1)],Integer(4))
>>> c.k()
4
```

length()

Return the length of `self`.

The length of a k-core is the size of the corresponding $(k-1)$-bounded partition which agrees with the length of the corresponding Grassmannian element, see `to_grassmannian()`.

EXAMPLES:

```
sage: c = Core([4,2],3); c.length()
4
sage: c.to_grassmannian().length()                                     #␣
↪needs sage.modules
4

sage: Core([9,5,3,2,1,1], 5).length()
13
```

```
>>> from sage.all import *
>>> c = Core([Integer(4),Integer(2)],Integer(3)); c.length()
4
>>> c.to_grassmannian().length()                                     #␣
↪needs sage.modules
4

>>> Core([Integer(9),Integer(5),Integer(3),Integer(2),Integer(1),Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(5)).length()
13
```

size()

Return the size of self as a partition.

EXAMPLES:

```
sage: Core([2,1],4).size()
3
sage: Core([4,2],3).size()
6
```

```
>>> from sage.all import *
>>> Core([Integer(2),Integer(1)],Integer(4)).size()
3
>>> Core([Integer(4),Integer(2)],Integer(3)).size()
6
```

strong_covers()

Return a list of all elements that cover self in strong order.

EXAMPLES:

```
sage: c = Core([1],3)
sage: c.strong_covers()
[[2], [1, 1]]
sage: c = Core([4,2],3)
sage: c.strong_covers()
[[5, 3, 1], [4, 2, 1, 1]]
```

```
>>> from sage.all import *
>>> c = Core([Integer(1)],Integer(3))
>>> c.strong_covers()
[[2], [1, 1]]
>>> c = Core([Integer(4),Integer(2)],Integer(3))
>>> c.strong_covers()
[[5, 3, 1], [4, 2, 1, 1]]
```

strong_down_list()

Return a list of all elements that are covered by self in strong order.

EXAMPLES:

```
sage: c = Core([1],3)
sage: c.strong_down_list()
[[]]
sage: c = Core([5,3,1],3)
sage: c.strong_down_list()
[[4, 2], [3, 1, 1]]
```

```
>>> from sage.all import *
>>> c = Core([Integer(1)],Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> c.strong_down_list()
[[]]
>>> c = Core([Integer(5),Integer(3),Integer(1)],Integer(3))
>>> c.strong_down_list()
[[4, 2], [3, 1, 1]]
```

strong_le(*other*)

Strong order (Bruhat) comparison on cores.

INPUT:

- `other` – another k-core

OUTPUT: boolean

This returns whether `self` <= `other` in Bruhat (or strong) order.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: x = Core([4,2,2,1,1],3)
sage: c.strong_le(x)
True
sage: c.strong_le([4,2,2,1,1])
True

sage: x = Core([4,1],4)
sage: c.strong_le(x)
Traceback (most recent call last):
...
ValueError: the two cores do not have the same k
```

```
>>> from sage.all import *
>>> c = Core([Integer(4),Integer(2)],Integer(3))
>>> x = Core([Integer(4),Integer(2),Integer(2),Integer(1),Integer(1)],
↪Integer(3))
>>> c.strong_le(x)
True
>>> c.strong_le([Integer(4),Integer(2),Integer(2),Integer(1),Integer(1)])
True

>>> x = Core([Integer(4),Integer(1)],Integer(4))
>>> c.strong_le(x)
Traceback (most recent call last):
...
ValueError: the two cores do not have the same k
```

to_bounded_partition()

Bijection between k-cores and $(k-1)$-bounded partitions.

This maps the k-core `self` to the corresponding $(k-1)$-bounded partition. This bijection is achieved by deleting all cells in `self` of hook length greater than k.

EXAMPLES:

```
sage: gamma = Core([9,5,3,2,1,1], 5)
sage: gamma.to_bounded_partition()
[4, 3, 2, 2, 1, 1]
```

```
>>> from sage.all import *
>>> gamma = Core([Integer(9),Integer(5),Integer(3),Integer(2),Integer(1),
↪Integer(1)], Integer(5))
>>> gamma.to_bounded_partition()
[4, 3, 2, 2, 1, 1]
```

to_grassmannian()

Bijection between k-cores and Grassmannian elements in the affine Weyl group of type $A_{k-1}^{(1)}$.

For further details, see the documentation of the method *from_kbounded_to_reduced_word()* and *from_kbounded_to_grassmannian()*.

EXAMPLES:

```
sage: c = Core([3,1,1],3)
sage: w = c.to_grassmannian(); w                                        #␣
↪needs sage.modules
[-1  1  1]
[-2  2  1]
[-2  1  2]
sage: c.parent()
3-Cores of length 4
sage: w.parent()                                                        #␣
↪needs sage.modules
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root space)

sage: c = Core([],3)
sage: c.to_grassmannian()                                               #␣
↪needs sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> c = Core([Integer(3),Integer(1),Integer(1)],Integer(3))
>>> w = c.to_grassmannian(); w                                        #␣
↪needs sage.modules
[-1  1  1]
[-2  2  1]
[-2  1  2]
>>> c.parent()
3-Cores of length 4
>>> w.parent()                                                        #␣
↪needs sage.modules
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root space)

>>> c = Core([],Integer(3))
>>> c.to_grassmannian()                                               #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
[1 0 0]
[0 1 0]
[0 0 1]
```

to_partition()

Turn the core `self` into the partition identical to `self`.

EXAMPLES:

```
sage: mu = Core([2,1,1],3)
sage: mu.to_partition()
[2, 1, 1]
```

```
>>> from sage.all import *
>>> mu = Core([Integer(2),Integer(1),Integer(1)],Integer(3))
>>> mu.to_partition()
[2, 1, 1]
```

weak_covers()

Return a list of all elements that cover `self` in weak order.

EXAMPLES:

```
sage: c = Core([1],3)
sage: c.weak_covers()                                                    #␣
↪needs sage.modules
[[1, 1], [2]]

sage: c = Core([4,2],3)
sage: c.weak_covers()                                                    #␣
↪needs sage.modules
[[5, 3, 1]]
```

```
>>> from sage.all import *
>>> c = Core([Integer(1)],Integer(3))
>>> c.weak_covers()                                                    #␣
↪needs sage.modules
[[1, 1], [2]]

>>> c = Core([Integer(4),Integer(2)],Integer(3))
>>> c.weak_covers()                                                    #␣
↪needs sage.modules
[[5, 3, 1]]
```

weak_le(*other*)

Weak order comparison on cores.

INPUT:

- `other` – another k-core

OUTPUT: boolean

This returns whether `self` <= `other` in weak order.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: x = Core([5,3,1],3)
sage: c.weak_le(x)                                                          #␣
↪needs sage.modules
True
sage: c.weak_le([5,3,1])                                                    #␣
↪needs sage.modules
True

sage: x = Core([4,2,2,1,1],3)
sage: c.weak_le(x)                                                          #␣
↪needs sage.modules
False

sage: x = Core([5,3,1],6)
sage: c.weak_le(x)
Traceback (most recent call last):
...
ValueError: the two cores do not have the same k
```

```
>>> from sage.all import *
>>> c = Core([Integer(4),Integer(2)],Integer(3))
>>> x = Core([Integer(5),Integer(3),Integer(1)],Integer(3))
>>> c.weak_le(x)                                                          #␣
↪needs sage.modules
True
>>> c.weak_le([Integer(5),Integer(3),Integer(1)])                         ␣
↪                          # needs sage.modules
True

>>> x = Core([Integer(4),Integer(2),Integer(2),Integer(1),Integer(1)],
↪Integer(3))
>>> c.weak_le(x)                                                          #␣
↪needs sage.modules
False

>>> x = Core([Integer(5),Integer(3),Integer(1)],Integer(6))
>>> c.weak_le(x)
Traceback (most recent call last):
...
ValueError: the two cores do not have the same k
```

sage.combinat.core.**Cores**(*k*, *length=None*, ***kwargs*)

A k-core is a partition from which no rim hook of size k can be removed. Alternatively, a k-core is an integer partition such that the Ferrers diagram for the partition contains no cells with a hook of size (a multiple of) k.

The k-cores generally have two notions of size which are useful for different applications. One is the number of cells in the Ferrers diagram with hook less than k, the other is the total number of cells of the Ferrers diagram. In the implementation in Sage, the first of notion is referred to as the `length` of the k-core and the second is the `size` of the k-core. The class of Cores requires that either the size or the length of the elements in the class is specified.

EXAMPLES:

We create the set of the 4-cores of length 6. Here the length of a k-core is the size of the corresponding $(k-1)$-bounded partition, see also `length()`:

```
sage: C = Cores(4, 6); C
4-Cores of length 6
sage: C.list()
[[6, 3], [5, 2, 1], [4, 1, 1, 1], [4, 2, 2], [3, 3, 1, 1], [3, 2, 1, 1, 1], [2, 2,
↪ 2, 1, 1, 1]]
sage: C.cardinality()
7
sage: C.an_element()
[6, 3]
```

```
>>> from sage.all import *
>>> C = Cores(Integer(4), Integer(6)); C
4-Cores of length 6
>>> C.list()
[[6, 3], [5, 2, 1], [4, 1, 1, 1], [4, 2, 2], [3, 3, 1, 1], [3, 2, 1, 1, 1], [2, 2,
↪ 2, 1, 1, 1]]
>>> C.cardinality()
7
>>> C.an_element()
[6, 3]
```

We may also list the set of 4-cores of size 6, where the size is the number of boxes in the core, see also `size()`:

```
sage: C = Cores(4, size=6); C
4-Cores of size 6
sage: C.list()
[[4, 1, 1], [3, 2, 1], [3, 1, 1, 1]]
sage: C.cardinality()
3
sage: C.an_element()
[4, 1, 1]
```

```
>>> from sage.all import *
>>> C = Cores(Integer(4), size=Integer(6)); C
4-Cores of size 6
>>> C.list()
[[4, 1, 1], [3, 2, 1], [3, 1, 1, 1]]
>>> C.cardinality()
3
>>> C.an_element()
[4, 1, 1]
```

class `sage.combinat.core.`**`Cores_length`**`(k, n)`

Bases: `UniqueRepresentation`, `Parent`

The class of k-cores of length n.

`Element`

alias of `Core`

`from_partition``(part)`

Convert the partition `part` into a core (as the identity map).

This is the inverse method to *to_partition()*.

EXAMPLES:

```
sage: C = Cores(3,4)
sage: c = C.from_partition([4,2]); c
[4, 2]

sage: mu = Partition([2,1,1])
sage: C = Cores(3,3)
sage: C.from_partition(mu).to_partition() == mu
True

sage: mu = Partition([])
sage: C = Cores(3,0)
sage: C.from_partition(mu).to_partition() == mu
True
```

```
>>> from sage.all import *
>>> C = Cores(Integer(3),Integer(4))
>>> c = C.from_partition([Integer(4),Integer(2)]); c
[4, 2]

>>> mu = Partition([Integer(2),Integer(1),Integer(1)])
>>> C = Cores(Integer(3),Integer(3))
>>> C.from_partition(mu).to_partition() == mu
True

>>> mu = Partition([])
>>> C = Cores(Integer(3),Integer(0))
>>> C.from_partition(mu).to_partition() == mu
True
```

list()

Return the list of all k-cores of length n.

EXAMPLES:

```
sage: C = Cores(3,4)
sage: C.list()
[[4, 2], [3, 1, 1], [2, 2, 1, 1]]
```

```
>>> from sage.all import *
>>> C = Cores(Integer(3),Integer(4))
>>> C.list()
[[4, 2], [3, 1, 1], [2, 2, 1, 1]]
```

class sage.combinat.core.**Cores_size**(k, n)

Bases: UniqueRepresentation, Parent

The class of k-cores of size n.

Element

alias of *Core*

from_partition(*part*)

Convert the partition `part` into a core (as the identity map).

This is the inverse method to `to_partition()`.

EXAMPLES:

```
sage: C = Cores(3,size=4)
sage: c = C.from_partition([2,1,1]); c
[2, 1, 1]

sage: mu = Partition([2,1,1])
sage: C = Cores(3,size=4)
sage: C.from_partition(mu).to_partition() == mu
True

sage: mu = Partition([])
sage: C = Cores(3,size=0)
sage: C.from_partition(mu).to_partition() == mu
True
```

```
>>> from sage.all import *
>>> C = Cores(Integer(3),size=Integer(4))
>>> c = C.from_partition([Integer(2),Integer(1),Integer(1)]); c
[2, 1, 1]

>>> mu = Partition([Integer(2),Integer(1),Integer(1)])
>>> C = Cores(Integer(3),size=Integer(4))
>>> C.from_partition(mu).to_partition() == mu
True

>>> mu = Partition([])
>>> C = Cores(Integer(3),size=Integer(0))
>>> C.from_partition(mu).to_partition() == mu
True
```

list()

Return the list of all k-cores of size n.

EXAMPLES:

```
sage: C = Cores(3, size = 4)
sage: C.list()
[[3, 1], [2, 1, 1]]
```

```
>>> from sage.all import *
>>> C = Cores(Integer(3), size = Integer(4))
>>> C.list()
[[3, 1], [2, 1, 1]]
```

5.1.34 Counting

- The On-Line Encyclopedia of Integer Sequences (OEIS)
- *Functions that compute some of the sequences in Sloane's tables*
- *Compute Bell and Uppuluri-Carpenter numbers*
- *q-Analogues*, *q-Bernoulli Numbers and Polynomials*
- *Binary Recurrence Sequences*
- *C-Finite Sequences*
- *Combinatorial Functions*

Todo

Mention sage/combinat/degree_sequences?

5.1.35 Affine Crystals

class sage.combinat.crystals.affine.**AffineCrystalFromClassical**(*cartan_type*, *classical_crystal*, *category=None*)

Bases: `UniqueRepresentation`, `Parent`

This abstract class can be used for affine crystals that are constructed from a classical crystal. The zero arrows can be implemented using different methods (for example using a Dynkin diagram automorphisms or virtual crystals).

This is a helper class, mostly used to implement Kirillov-Reshetikhin crystals (see: *KirillovReshetikhin-Crystal()*).

For general information about crystals see `sage.combinat.crystals`.

INPUT:

- `cartan_type` – the Cartan type of the resulting affine crystal
- `classical_crystal` – instance of a classical crystal

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: A.list()
[[[1]], [[2]], [[3]]]
sage: A.cartan_type()
['A', 2, 1]
sage: A.index_set()
(0, 1, 2)
sage: b = A(rows=[[1]])
sage: b.weight()
-Lambda[0] + Lambda[1]
sage: b.classical_weight()
(1, 0, 0)
sage: [x.s(0) for x in A.list()]
```

(continues on next page)

(continued from previous page)

```
[[[3]], [[2]], [[1]]]
sage: [x.s(1) for x in A.list()]
[[[2]], [[1]], [[3]]]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> A.list()
[[[1]], [[2]], [[3]]]
>>> A.cartan_type()
['A', 2, 1]
>>> A.index_set()
(0, 1, 2)
>>> b = A(rows=[[Integer(1)]])
>>> b.weight()
-Lambda[0] + Lambda[1]
>>> b.classical_weight()
(1, 0, 0)
>>> [x.s(Integer(0)) for x in A.list()]
[[[3]], [[2]], [[1]]]
>>> [x.s(Integer(1)) for x in A.list()]
[[[2]], [[1]], [[3]]]
```

Element

alias of *AffineCrystalFromClassicalElement*

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: C = crystals.Tableaux(['A',3],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',3,1],C,pr,pr_inverse,
↪1)
sage: A.cardinality() == C.cardinality()
True
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',Integer(3)],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',Integer(3),Integer(1)],
↪C,pr,pr_inverse,Integer(1))
>>> A.cardinality() == C.cardinality()
True
```

lift(*affine_elt*)

Lift an affine crystal element to the corresponding classical crystal element.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A.list()[0]
sage: A.lift(b)
[[1]]
sage: A.lift(b).parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A.list()[Integer(0)]
>>> A.lift(b)
[[1]]
>>> A.lift(b).parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
```

retract(*classical_elt*)

Transform a classical crystal element to the corresponding affine crystal element.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: t = C(rows=[[1]])
sage: t.parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
sage: A.retract(t)
[[1]]
sage: A.retract(t).parent() is A
True
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
```

(continues on next page)

(continued from previous page)

```
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> t = C(rows=[[Integer(1)]])
>>> t.parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
>>> A.retract(t)
[[1]]
>>> A.retract(t).parent() is A
True
```

class sage.combinat.crystals.affine.**AffineCrystalFromClassicalAndPromotion**(*cartan_type*, *classical_crystal*, *p_automorphism*, *p_inverse_automorphism*, *dynkin_node*, *category=None*)

Bases: *AffineCrystalFromClassical*

Crystals that are constructed from a classical crystal and a Dynkin diagram automorphism σ. In type A_n, the Dynkin diagram automorphism is $i \to i+1 \pmod{n}+1$ and the corresponding map on the crystal is the promotion operation pr on tableaux. The affine crystal operators are given by $f_0 = \mathrm{pr}^{-1} f_{\sigma(0)} \mathrm{pr}$.

INPUT:

- `cartan_type` – the Cartan type of the resulting affine crystal
- `classical_crystal` – instance of a classical crystal
- `automorphism, inverse_automorphism` – a function on the elements of the `classical_crystal`
- `dynkin_node` – integer specifying the classical node in the image of the zero node under the automorphism sigma

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: A.list()
[[[1]], [[2]], [[3]]]
sage: A.cartan_type()
['A', 2, 1]
sage: A.index_set()
(0, 1, 2)
sage: b = A(rows=[[1]])
sage: b.weight()
-Lambda[0] + Lambda[1]
sage: b.classical_weight()
(1, 0, 0)
```

(continues on next page)

(continued from previous page)

```
sage: [x.s(0) for x in A.list()]
[[[3]], [[2]], [[1]]]
sage: [x.s(1) for x in A.list()]
[[[2]], [[1]], [[3]]]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> A.list()
[[[1]], [[2]], [[3]]]
>>> A.cartan_type()
['A', 2, 1]
>>> A.index_set()
(0, 1, 2)
>>> b = A(rows=[[Integer(1)]])
>>> b.weight()
-Lambda[0] + Lambda[1]
>>> b.classical_weight()
(1, 0, 0)
>>> [x.s(Integer(0)) for x in A.list()]
[[[3]], [[2]], [[1]]]
>>> [x.s(Integer(1)) for x in A.list()]
[[[2]], [[1]], [[3]]]
```

Element

alias of *AffineCrystalFromClassicalAndPromotionElement*

automorphism(x)

Give the analogue of the affine Dynkin diagram automorphism on the level of crystals.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A.list()[0]
sage: A.automorphism(b)
[[2]]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
```

(continues on next page)

(continued from previous page)

```
↪inverse,Integer(1))
>>> b = A.list()[Integer(0)]
>>> A.automorphism(b)
[[2]]
```

inverse_automorphism(*x*)

Give the analogue of the inverse of the affine Dynkin diagram automorphism on the level of crystals.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A.list()[0]
sage: A.inverse_automorphism(b)
[[3]]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A.list()[Integer(0)]
>>> A.inverse_automorphism(b)
[[3]]
```

class sage.combinat.crystals.affine.**AffineCrystalFromClassicalAndPromotionElement**

Bases: *AffineCrystalFromClassicalElement*

Elements of crystals that are constructed from a classical crystal and a Dynkin diagram automorphism. In type A, the automorphism is the promotion operation on tableaux.

This class is not instantiated directly but rather `__call__`-ed from *AffineCrystalFromClassicalAndPromotion*. The syntax of this is governed by the (classical) crystal.

Since this class inherits from *AffineCrystalFromClassicalElement*, the methods that need to be implemented are *e0()*, *f0()* and possibly *epsilon0()* and *phi0()* if more efficient algorithms exist.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b = A(rows=[[1]])
sage: b._repr_()
'[[1]]'
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A(rows=[[Integer(1)]])
>>> b._repr_()
'[[1]]'
```

e0()

Implement e_0 using the automorphism as $e_0 = \mathrm{pr}^{-1} e_{dynkin_node} \mathrm{pr}$

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A(rows=[[1]])
sage: b.e0()
[[3]]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A(rows=[[Integer(1)]])
>>> b.e0()
[[3]]
```

epsilon0()

Implement $epsilon_0$ using the automorphism.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: [x.epsilon0() for x in A.list()]
[1, 0, 0]
```

```
>>> from sage.all import *
>>> n = Integer(2)
```

(continues on next page)

(continued from previous page)

```
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> [x.epsilon0() for x in A.list()]
[1, 0, 0]
```

f0()

Implement f_0 using the automorphism as $f_0 = \mathrm{pr}^{-1} f_{dynkin_node} \mathrm{pr}$

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A(rows=[[3]])
sage: b.f0()
[[1]]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A(rows=[[Integer(3)]])
>>> b.f0()
[[1]]
```

phi0()

Implement phi_0 using the automorphism.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: [x.phi0() for x in A.list()]
[0, 0, 1]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
```

(continues on next page)

(continued from previous page)

```
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> [x.phi0() for x in A.list()]
[0, 0, 1]
```

class sage.combinat.crystals.affine.**AffineCrystalFromClassicalElement**

Bases: ElementWrapper

Elements of crystals that are constructed from a classical crystal.

The elements inherit many of their methods from the classical crystal using lift and retract.

This class is not instantiated directly but rather `__call__`-ed from *AffineCrystalFromClassical*. The syntax of this is governed by the (classical) crystal.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b = A(rows=[[1]])
sage: b._repr_()
'[[1]]'
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A(rows=[[Integer(1)]])
>>> b._repr_()
'[[1]]'
```

classical_weight()

Return the classical weight corresponding to `self`.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A(rows=[[1]])
sage: b.classical_weight()
(1, 0, 0)
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A(rows=[[Integer(1)]])
>>> b.classical_weight()
(1, 0, 0)
```

e(i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A(rows=[[1]])
sage: b.e(0)
[[3]]
sage: b.e(1)
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A(rows=[[Integer(1)]])
>>> b.e(Integer(0))
[[3]]
>>> b.e(Integer(1))
```

e0()

Assumes that e_0 is implemented separately.

epsilon(i)

Return the maximal time the crystal operator e_i can be applied to `self`.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: [x.epsilon(0) for x in A.list()]
```

(continues on next page)

(continued from previous page)

```
[1, 0, 0]
sage: [x.epsilon(1) for x in A.list()]
[0, 1, 0]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> [x.epsilon(Integer(0)) for x in A.list()]
[1, 0, 0]
>>> [x.epsilon(Integer(1)) for x in A.list()]
[0, 1, 0]
```

epsilon0()

Uses ε_0 from the super class, but should be implemented if a faster implementation exists.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: [x.epsilon0() for x in A.list()]
[1, 0, 0]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> [x.epsilon0() for x in A.list()]
[1, 0, 0]
```

f(i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A(rows=[[3]])
```

(continues on next page)

(continued from previous page)

```
sage: b.f(0)
[[1]]
sage: b.f(2)
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A(rows=[[Integer(3)]])
>>> b.f(Integer(0))
[[1]]
>>> b.f(Integer(2))
```

f0()

Assumes that f_0 is implemented separately.

lift()

Lift an affine crystal element to the corresponding classical crystal element.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: b = A.list()[0]
sage: b.lift()
[[1]]
sage: b.lift().parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> b = A.list()[Integer(0)]
>>> b.lift()
[[1]]
>>> b.lift().parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
```

phi(i)

Return the maximal time the crystal operator f_i can be applied to `self`.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: [x.phi(0) for x in A.list()]
[0, 0, 1]
sage: [x.phi(1) for x in A.list()]
[1, 0, 0]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> [x.phi(Integer(0)) for x in A.list()]
[0, 0, 1]
>>> [x.phi(Integer(1)) for x in A.list()]
[1, 0, 0]
```

phi0()

Uses φ_0 from the super class, but should be implemented if a faster implementation exists.

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,
↪1)
sage: [x.phi0() for x in A.list()]
[0, 0, 1]
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> C = crystals.Tableaux(['A',n],shape=[Integer(1)])
>>> pr = attrcall("promotion")
>>> pr_inverse = attrcall("promotion_inverse")
>>> A = crystals.AffineFromClassicalAndPromotion(['A',n,Integer(1)],C,pr,pr_
↪inverse,Integer(1))
>>> [x.phi0() for x in A.list()]
[0, 0, 1]
```

pp()

Method for pretty printing.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',3,2],1,1)
sage: t=K(rows=[[1]])
sage: t.pp()
1
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(3),Integer(2)],Integer(1),
↪Integer(1))
>>> t=K(rows=[[Integer(1)]])
>>> t.pp()
1
```

5.1.36 Affine factorization crystal of type A

class sage.combinat.crystals.affine_factorization.**AffineFactorizationCrystal**(*w*, *n*, *x=None*)

Bases: UniqueRepresentation, Parent

The crystal on affine factorizations with a cut-point, as introduced by [MS2015].

INPUT:

- w – an element in an (affine) Weyl group or a skew shape of k-bounded partitions (if k was specified)
- n – the number of factors in the factorization
- x – (default: None) the cut point; if not specified it is determined as the minimal missing residue in w
- k – (default: None) positive integer, specifies that w is k-bounded or a $k + 1$-core when specified

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,3,2,1])
sage: B = crystals.AffineFactorization(w,3); B
Crystal on affine factorizations of type A2 associated to s2*s3*s2*s1
sage: B.list()
[(1, s2, s3*s2*s1),
 (1, s3*s2, s3*s1),
 (1, s3*s2*s1, s3),
 (s3, s2, s3*s1),
 (s3, s2*s1, s3),
 (s3*s2, s1, s3),
 (s3*s2*s1, 1, s3),
 (s3*s2*s1, s3, 1),
 (s3*s2, 1, s3*s1),
 (s3*s2, s3, s1),
 (s3*s2, s3*s1, 1),
 (s2, 1, s3*s2*s1),
 (s2, s3, s2*s1),
 (s2, s3*s2, s1),
 (s2, s3*s2*s1, 1)]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3),Integer(1)], prefix='s')
```

(continues on next page)

(continued from previous page)

```
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(2),Integer(1)])
>>> B = crystals.AffineFactorization(w,Integer(3)); B
Crystal on affine factorizations of type A2 associated to s2*s3*s2*s1
>>> B.list()
[(1, s2, s3*s2*s1),
 (1, s3*s2, s3*s1),
 (1, s3*s2*s1, s3),
 (s3, s2, s3*s1),
 (s3, s2*s1, s3),
 (s3*s2, s1, s3),
 (s3*s2*s1, 1, s3),
 (s3*s2*s1, s3, 1),
 (s3*s2, 1, s3*s1),
 (s3*s2, s3, s1),
 (s3*s2, s3*s1, 1),
 (s2, 1, s3*s2*s1),
 (s2, s3, s2*s1),
 (s2, s3*s2, s1),
 (s2, s3*s2*s1, 1)]
```

We can also access the crystal by specifying a skew shape in terms of k-bounded partitions:

```
sage: crystals.AffineFactorization([[3,1,1],[1]], 3, k=3)
Crystal on affine factorizations of type A2 associated to s2*s3*s2*s1
```

```
>>> from sage.all import *
>>> crystals.AffineFactorization([[Integer(3),Integer(1),Integer(1)],
↪[Integer(1)]], Integer(3), k=Integer(3))
Crystal on affine factorizations of type A2 associated to s2*s3*s2*s1
```

We can compute the highest weight elements:

```
sage: hw = [w for w in B if w.is_highest_weight()]
sage: hw
[(1, s2, s3*s2*s1)]
sage: hw[0].weight()
(3, 1, 0)
```

```
>>> from sage.all import *
>>> hw = [w for w in B if w.is_highest_weight()]
>>> hw
[(1, s2, s3*s2*s1)]
>>> hw[Integer(0)].weight()
(3, 1, 0)
```

And show that this crystal is isomorphic to the tableau model of the same weight:

```
sage: C = crystals.Tableaux(['A',2],shape=[3,1])
sage: GC = C.digraph()
sage: GB = B.digraph()
sage: GC.is_isomorphic(GB, edge_labels=True)
True
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',Integer(2)],shape=[Integer(3),Integer(1)])
>>> GC = C.digraph()
>>> GB = B.digraph()
>>> GC.is_isomorphic(GB, edge_labels=True)
True
```

The crystal operators themselves move elements between adjacent factors:

```
sage: b = hw[0];b
(1, s2, s3*s2*s1)
sage: b.f(1)
(1, s3*s2, s3*s1)
```

```
>>> from sage.all import *
>>> b = hw[Integer(0)];b
(1, s2, s3*s2*s1)
>>> b.f(Integer(1))
(1, s3*s2, s3*s1)
```

The cut point x is not supposed to occur in the reduced words for w:

```
sage: B = crystals.AffineFactorization([[3,2],[2]],4,x=0,k=3)
Traceback (most recent call last):
...
ValueError: x cannot be in reduced word of s0*s3*s2
```

```
>>> from sage.all import *
>>> B = crystals.AffineFactorization([[Integer(3),Integer(2)],[Integer(2)]],
↪Integer(4),x=Integer(0),k=Integer(3))
Traceback (most recent call last):
...
ValueError: x cannot be in reduced word of s0*s3*s2
```

class Element

Bases: `ElementWrapper`

bracketing(*i*)

Removes all bracketed letters between i-th and $i + 1$-th entry.

EXAMPLES:

```
sage: B = crystals.AffineFactorization([[3,1],[1]], 3, k=3, x=4)
sage: W = B.w.parent()
sage: t = B((W.one(),W.from_reduced_word([3]),W.from_reduced_word([2,
↪1]))); t
(1, s3, s2*s1)
sage: t.bracketing(1)
[[3], [2, 1]]
```

```
>>> from sage.all import *
>>> B = crystals.AffineFactorization([[Integer(3),Integer(1)],
↪[Integer(1)]], Integer(3), k=Integer(3), x=Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> W = B.w.parent()
>>> t = B((W.one(),W.from_reduced_word([Integer(3)]),W.from_reduced_
↪word([Integer(2),Integer(1)]))); t
(1, s3, s2*s1)
>>> t.bracketing(Integer(1))
[[3], [2, 1]]
```

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: B = crystals.AffineFactorization([[3,1],[1]], 4, k=3)
sage: W = B.w.parent()
sage: t = B((W.one(),W.one(),W.from_reduced_word([3]),W.from_reduced_
↪word([2,1]))); t
(1, 1, s3, s2*s1)
sage: t.e(1)
(1, 1, 1, s3*s2*s1)
```

```
>>> from sage.all import *
>>> B = crystals.AffineFactorization([[Integer(3),Integer(1)],
↪[Integer(1)]], Integer(4), k=Integer(3))
>>> W = B.w.parent()
>>> t = B((W.one(),W.one(),W.from_reduced_word([Integer(3)]),W.from_
↪reduced_word([Integer(2),Integer(1)]))); t
(1, 1, s3, s2*s1)
>>> t.e(Integer(1))
(1, 1, 1, s3*s2*s1)
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: B = crystals.AffineFactorization([[3,1],[1]], 4, k=3)
sage: W = B.w.parent()
sage: t = B((W.one(),W.one(),W.from_reduced_word([3]),W.from_reduced_
↪word([2,1]))); t
(1, 1, s3, s2*s1)
sage: t.f(2)
(1, s3, 1, s2*s1)
sage: t.f(1)
(1, 1, s3*s2, s1)
```

```
>>> from sage.all import *
>>> B = crystals.AffineFactorization([[Integer(3),Integer(1)],
↪[Integer(1)]], Integer(4), k=Integer(3))
>>> W = B.w.parent()
>>> t = B((W.one(),W.one(),W.from_reduced_word([Integer(3)]),W.from_
↪reduced_word([Integer(2),Integer(1)]))); t
(1, 1, s3, s2*s1)
```

(continues on next page)

(continued from previous page)

```
>>> t.f(Integer(2))
(1, s3, 1, s2*s1)
>>> t.f(Integer(1))
(1, 1, s3*s2, s1)
```

to_tableau()

Return the tableau representation of `self`.

Uses the recording tableau of a minor variation of Edelman-Greene insertion. See Theorem 4.11 in [MS2015].

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2])
sage: B = crystals.AffineFactorization(w,3)
sage: for x in B:
....:     x
....:     x.to_tableau().pp()
(1, s2*s1, s3*s2)
  1  1
  2  2
(s2, s1, s3*s2)
  1  1
  2  3
(s2, s3*s1, s2)
  1  2
  2  3
(s2*s1, 1, s3*s2)
  1  1
  3  3
(s2*s1, s3, s2)
  1  2
  3  3
(s2*s1, s3*s2, 1)
  2  2
  3  3
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3),Integer(1)], prefix='s')
>>> w = W.from_reduced_word([Integer(2),Integer(1),Integer(3),Integer(2)])
>>> B = crystals.AffineFactorization(w,Integer(3))
>>> for x in B:
...     x
...     x.to_tableau().pp()
(1, s2*s1, s3*s2)
  1  1
  2  2
(s2, s1, s3*s2)
  1  1
  2  3
(s2, s3*s1, s2)
  1  2
```

(continues on next page)

(continued from previous page)

```
  2  3
(s2*s1, 1, s3*s2)
  1  1
  3  3
(s2*s1, s3, s2)
  1  2
  3  3
(s2*s1, s3*s2, 1)
  2  2
  3  3
```

class sage.combinat.crystals.affine_factorization.**FactorizationToTableaux**(*parent*, *cartan_type=None*, *virtualization=None*, *scaling_factors=None*)

Bases: CrystalMorphism

is_embedding()

Return True as this is an isomorphism.

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2])
sage: B = crystals.AffineFactorization(w,3)
sage: phi = B._tableaux_isomorphism
sage: phi.is_isomorphism()
True
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3),Integer(1)], prefix='s')
>>> w = W.from_reduced_word([Integer(2),Integer(1),Integer(3),Integer(2)])
>>> B = crystals.AffineFactorization(w,Integer(3))
>>> phi = B._tableaux_isomorphism
>>> phi.is_isomorphism()
True
```

is_isomorphism()

Return True as this is an isomorphism.

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2])
sage: B = crystals.AffineFactorization(w,3)
sage: phi = B._tableaux_isomorphism
sage: phi.is_isomorphism()
True
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3),Integer(1)], prefix='s')
```

(continues on next page)

(continued from previous page)

```
>>> w = W.from_reduced_word([Integer(2),Integer(1),Integer(3),Integer(2)])
>>> B = crystals.AffineFactorization(w,Integer(3))
>>> phi = B._tableaux_isomorphism
>>> phi.is_isomorphism()
True
```

is_surjective()

Return `True` as this is an isomorphism.

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2])
sage: B = crystals.AffineFactorization(w,3)
sage: phi = B._tableaux_isomorphism
sage: phi.is_isomorphism()
True
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3),Integer(1)], prefix='s')
>>> w = W.from_reduced_word([Integer(2),Integer(1),Integer(3),Integer(2)])
>>> B = crystals.AffineFactorization(w,Integer(3))
>>> phi = B._tableaux_isomorphism
>>> phi.is_isomorphism()
True
```

sage.combinat.crystals.affine_factorization.**affine_factorizations**(*w*, *l*, *weight=None*)

Return all factorizations of w into l factors or of weight `weight`.

INPUT:

- `w` – an (affine) permutation or element of the (affine) Weyl group
- `l` – nonnegative integer
- `weight` – (default: `None`) tuple of nonnegative integers specifying the length of the factors

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([3,2,3,1,0,1])
sage: from sage.combinat.crystals.affine_factorization import affine_
↪factorizations
sage: affine_factorizations(w,4)
[[s2, s3, s0, s2*s1*s0],
[s2, s3, s2*s0, s1*s0],
[s2, s3, s2*s1*s0, s1],
[s2, s3*s2, s0, s1*s0],
[s2, s3*s2, s1*s0, s1],
[s2, s3*s2*s1, s0, s1],
[s3*s2, s3, s0, s1*s0],
[s3*s2, s3, s1*s0, s1],
[s3*s2, s3*s1, s0, s1],
[s3*s2*s1, s3, s0, s1]]
```

(continues on next page)

(continued from previous page)

```
sage: W = WeylGroup(['A',2], prefix='s')
sage: w0 = W.long_element()
sage: affine_factorizations(w0,3)
[[1, s1, s2*s1],
[1, s2*s1, s2],
[s1, 1, s2*s1],
[s1, s2, s1],
[s1, s2*s1, 1],
[s2, s1, s2],
[s2*s1, 1, s2],
[s2*s1, s2, 1]]
sage: affine_factorizations(w0,3,(0,1,2))
[[1, s1, s2*s1]]
sage: affine_factorizations(w0,3,(1,1,1))
[[s1, s2, s1], [s2, s1, s2]]
sage: W = WeylGroup(['A',3], prefix='s')
sage: w0 = W.long_element()
sage: affine_factorizations(w0,6,(1,1,1,1,1,1))   # long time
[[s1, s2, s1, s3, s2, s1],
[s1, s2, s3, s1, s2, s1],
[s1, s2, s3, s2, s1, s2],
[s1, s3, s2, s1, s3, s2],
[s1, s3, s2, s3, s1, s2],
[s2, s1, s2, s3, s2, s1],
[s2, s1, s3, s2, s1, s3],
[s2, s1, s3, s2, s3, s1],
[s2, s3, s1, s2, s1, s3],
[s2, s3, s1, s2, s3, s1],
[s2, s3, s2, s1, s2, s3],
[s3, s1, s2, s1, s3, s2],
[s3, s1, s2, s3, s1, s2],
[s3, s2, s1, s2, s3, s2],
[s3, s2, s1, s3, s2, s3],
[s3, s2, s3, s1, s2, s3]]
sage: affine_factorizations(w0,6,(0,0,0,1,2,3))
[[1, 1, 1, s1, s2*s1, s3*s2*s1]]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3),Integer(1)], prefix='s')
>>> w = W.from_reduced_word([Integer(3),Integer(2),Integer(3),Integer(1),
↪Integer(0),Integer(1)])
>>> from sage.combinat.crystals.affine_factorization import affine_factorizations
>>> affine_factorizations(w,Integer(4))
[[s2, s3, s0, s2*s1*s0],
[s2, s3, s2*s0, s1*s0],
[s2, s3, s2*s1*s0, s1],
[s2, s3*s2, s0, s1*s0],
[s2, s3*s2, s1*s0, s1],
[s2, s3*s2*s1, s0, s1],
[s3*s2, s3, s0, s1*s0],
[s3*s2, s3, s1*s0, s1],
[s3*s2, s3*s1, s0, s1],
```

(continues on next page)

(continued from previous page)

```
[s3*s2*s1, s3, s0, s1]]

>>> W = WeylGroup(['A',Integer(2)], prefix='s')
>>> w0 = W.long_element()
>>> affine_factorizations(w0,Integer(3))
[[1, s1, s2*s1],
[1, s2*s1, s2],
[s1, 1, s2*s1],
[s1, s2, s1],
[s1, s2*s1, 1],
[s2, s1, s2],
[s2*s1, 1, s2],
[s2*s1, s2, 1]]
>>> affine_factorizations(w0,Integer(3),(Integer(0),Integer(1),Integer(2)))
[[1, s1, s2*s1]]
>>> affine_factorizations(w0,Integer(3),(Integer(1),Integer(1),Integer(1)))
[[s1, s2, s1], [s2, s1, s2]]
>>> W = WeylGroup(['A',Integer(3)], prefix='s')
>>> w0 = W.long_element()
>>> affine_factorizations(w0,Integer(6),(Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(1),Integer(1)))   # long time
[[s1, s2, s1, s3, s2, s1],
[s1, s2, s3, s1, s2, s1],
[s1, s2, s3, s2, s1, s2],
[s1, s3, s2, s1, s3, s2],
[s1, s3, s2, s3, s1, s2],
[s2, s1, s2, s3, s2, s1],
[s2, s1, s3, s2, s1, s3],
[s2, s1, s3, s2, s3, s1],
[s2, s3, s1, s2, s1, s3],
[s2, s3, s1, s2, s3, s1],
[s2, s3, s2, s1, s2, s3],
[s3, s1, s2, s1, s3, s2],
[s3, s1, s2, s3, s1, s2],
[s3, s2, s1, s2, s3, s2],
[s3, s2, s1, s3, s2, s3],
[s3, s2, s3, s1, s2, s3]]
>>> affine_factorizations(w0,Integer(6),(Integer(0),Integer(0),Integer(0),
↪Integer(1),Integer(2),Integer(3)))
[[1, 1, 1, s1, s2*s1, s3*s2*s1]]
```

5.1.37 Affinization Crystals

class sage.combinat.crystals.affinization.**AffinizationOfCrystal**(B)

Bases: `UniqueRepresentation`, `Parent`

An affinization of a crystal.

Let $\mathfrak{g}$ be a Kac-Moody algebra of affine type. The affinization of a finite $U_q'(\mathfrak{g})$-crystal B is the (infinite) $U_q(\mathfrak{g})$-crystal with underlying set:

$$B^{\mathrm{aff}} = \{b(m) \mid b \in B, m \in \mathbf{Z}\}$$

and crystal structure determined by:

$$e_i(b(m)) = \begin{cases} (e_0 b)(m+1) & i = 0, \\ (e_i b)(m) & i \neq 0, \end{cases}$$
$$f_i(b(m)) = \begin{cases} (f_0 b)(m-1) & i = 0, \\ (f_i b)(m) & i \neq 0, \end{cases}$$
$$\mathrm{wt}(b(m)) = \mathrm{wt}(b) + m\delta.$$

EXAMPLES:

We first construct a Kirillov-Reshetikhin crystal and then take it's corresponding affinization:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 2, 2)
sage: A = K.affinization()
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)], Integer(2),␣
→Integer(2))
>>> A = K.affinization()
```

Next we construct an affinization crystal from a tensor product of KR crystals:

```
sage: KT = crystals.TensorProductOfKirillovReshetikhinTableaux(['C',2,1], [[1,2],
→[2,1]])
sage: A = crystals.AffinizationOf(KT)
```

```
>>> from sage.all import *
>>> KT = crystals.TensorProductOfKirillovReshetikhinTableaux(['C',Integer(2),
→Integer(1)], [[Integer(1),Integer(2)],[Integer(2),Integer(1)]])
>>> A = crystals.AffinizationOf(KT)
```

REFERENCES:

- [HK2002] Chapter 10

class Element(*parent*, *b*, *m*)

Bases: `Element`

An element in an affinization crystal.

e(*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.e(0)
[[1, 2], [2, 3]](1)
sage: mg.e(1)
sage: mg.e(0).e(1)
[[1, 1], [2, 3]](1)
```

```
>>> from sage.all import *
>>> A = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(2),Integer(2)).affinization()
>>> mg = A.module_generators[Integer(0)]
>>> mg.e(Integer(0))
[[1, 2], [2, 3]](1)
>>> mg.e(Integer(1))
>>> mg.e(Integer(0)).e(Integer(1))
[[1, 1], [2, 3]](1)
```

epsilon(*i*)

Return ε_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.epsilon(0)
2
sage: mg.epsilon(1)
0
```

```
>>> from sage.all import *
>>> A = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(2),Integer(2)).affinization()
>>> mg = A.module_generators[Integer(0)]
>>> mg.epsilon(Integer(0))
2
>>> mg.epsilon(Integer(1))
0
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.f(2)
[[1, 1], [2, 3]](0)
sage: mg.f(2).f(2).f(0)
sage: mg.f_string([2,1,1])
sage: mg.f_string([2,1])
[[1, 2], [2, 3]](0)
sage: mg.f_string([2,1,0])
[[1, 1], [2, 2]](-1)
```

```
>>> from sage.all import *
>>> A = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(2)).affinization()
>>> mg = A.module_generators[Integer(0)]
>>> mg.f(Integer(2))
[[1, 1], [2, 3]](0)
>>> mg.f(Integer(2)).f(Integer(2)).f(Integer(0))
>>> mg.f_string([Integer(2),Integer(1),Integer(1)])
>>> mg.f_string([Integer(2),Integer(1)])
[[1, 2], [2, 3]](0)
>>> mg.f_string([Integer(2),Integer(1),Integer(0)])
[[1, 1], [2, 2]](-1)
```

phi (i)

Return φ_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.phi(0)
0
sage: mg.phi(2)
2
```

```
>>> from sage.all import *
>>> A = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(2),Integer(2)).affinization()
>>> mg = A.module_generators[Integer(0)]
>>> mg.phi(Integer(0))
0
>>> mg.phi(Integer(2))
2
```

weight ()

Return the weight of `self`.

The weight wt of an element is:

$$\mathrm{wt}\big(b(m)\big) = \mathrm{wt}(b) + m\delta,$$

where δ is the null root.

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.weight()
-2*Lambda[0] + 2*Lambda[2]
sage: mg.e(0).weight()
-Lambda[1] + Lambda[2] + delta
sage: mg.e(0).e(0).weight()
2*Lambda[0] - 2*Lambda[1] + 2*delta
```

```
>>> from sage.all import *
>>> A = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],
↪Integer(2),Integer(2)).affinization()
>>> mg = A.module_generators[Integer(0)]
>>> mg.weight()
-2*Lambda[0] + 2*Lambda[2]
>>> mg.e(Integer(0)).weight()
-Lambda[1] + Lambda[2] + delta
>>> mg.e(Integer(0)).e(Integer(0)).weight()
2*Lambda[0] - 2*Lambda[1] + 2*delta
```

5.1.38 Alcove paths

AUTHORS:

- Brant Jones (2008): initial version
- Arthur Lubovsky (2013-03-07): rewritten to implement affine type
- Travis Scrimshaw (2016-06-23): implemented $\mathcal{B}(\infty)$

Special thanks to: Nicolas Borie, Anne Schilling, Travis Scrimshaw, and Nicolas Thiéry.

class sage.combinat.crystals.alcove_path.**CrystalOfAlcovePaths**(*starting_weight*, *highest_weight_crystal*)

Bases: `UniqueRepresentation`, `Parent`

Crystal of alcove paths generated from a "straight-line" path to the negative of a given dominant weight.

INPUT:

- `cartan_type` – Cartan type of a finite or affine untwisted root system
- `weight` – dominant weight as a list of (integral) coefficients of the fundamental weights
- `highest_weight_crystal` – (default: `True`) if `True` returns the highest weight crystal. If `False` returns an object which is close to being isomorphic to the tensor product of Kirillov-Reshetikhin crystals of column shape in the following sense: We get all the vertices, but only some of the edges. We'll call the included edges pseudo-Demazure. They are all nonzero edges and the 0-edges not at the end of a 0-string of edges, i.e. not those with $f_0(b) = b'$ with $\varphi_0(b) = 1$. (Whereas Demazure 0-edges are those that are not at the beginning of a zero string.) In this case the weight $[c_1, c_2, \ldots, c_k]$ represents $\sum_{i=1}^k c_i \omega_i$.

Note

If `highest_weight_crystal` = `False`, since we do not get the full crystal, `TestSuite` will fail on the Stembridge axioms.

See also

- `Crystals`

EXAMPLES:

The following example appears in Figure 2 of [LP2008]:

```
sage: C = crystals.AlcovePaths(['G',2],[0,1])
sage: G = C.digraph()
sage: GG = DiGraph({
....:     ()        : {(0)           : 2 },
....:     (0)       : {(0,8)         : 1 },
....:     (0,1)     : {(0,1,7)       : 2 },
....:     (0,1,2)   : {(0,1,2,9)     : 1 },
....:     (0,1,2,3) : {(0,1,2,3,4)   : 2 },
....:     (0,1,2,6) : {(0,1,2,3)     : 1 },
....:     (0,1,2,9) : {(0,1,2,6)     : 1 },
....:     (0,1,7)   : {(0,1,2)       : 2 },
....:     (0,1,7,9) : {(0,1,2,9)     : 2 },
....:     (0,5)     : {(0,1)         : 1, (0,5,7) : 2 },
....:     (0,5,7)   : {(0,5,7,9)     : 1 },
....:     (0,5,7,9) : {(0,1,7,9)     : 1 },
....:     (0,8)     : {(0,5)         : 1 },
....:     })
sage: G.is_isomorphic(GG)
True
sage: for (u,v,i) in G.edges(sort=True):
....:     print((u.integer_sequence() , v.integer_sequence(), i))
([], [0], 2)
([0], [0, 8], 1)
([0, 1], [0, 1, 7], 2)
([0, 1, 2], [0, 1, 2, 9], 1)
([0, 1, 2, 3], [0, 1, 2, 3, 4], 2)
([0, 1, 2, 6], [0, 1, 2, 3], 1)
([0, 1, 2, 9], [0, 1, 2, 6], 1)
([0, 1, 7], [0, 1, 2], 2)
([0, 1, 7, 9], [0, 1, 2, 9], 2)
([0, 5], [0, 1], 1)
([0, 5], [0, 5, 7], 2)
([0, 5, 7], [0, 5, 7, 9], 1)
([0, 5, 7, 9], [0, 1, 7, 9], 1)
([0, 8], [0, 5], 1)
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['G',Integer(2)],[Integer(0),Integer(1)])
>>> G = C.digraph()
>>> GG = DiGraph({
...     ()        : {(Integer(0))          : Integer(2) },
...     (Integer(0))        : {(Integer(0),Integer(8))        : Integer(1) },
...     (Integer(0),Integer(1))      : {(Integer(0),Integer(1),Integer(7))      :␣
↪Integer(2) },
...     (Integer(0),Integer(1),Integer(2))    : {(Integer(0),Integer(1),Integer(2),
↪Integer(9))    : Integer(1) },
...     (Integer(0),Integer(1),Integer(2),Integer(3)) : {(Integer(0),Integer(1),
↪Integer(2),Integer(3),Integer(4)) : Integer(2) },
...     (Integer(0),Integer(1),Integer(2),Integer(6)) : {(Integer(0),Integer(1),
↪Integer(2),Integer(3))    : Integer(1) },
...     (Integer(0),Integer(1),Integer(2),Integer(9)) : {(Integer(0),Integer(1),
↪Integer(2),Integer(6))    : Integer(1) },
```

(continues on next page)

(continued from previous page)

```
...     (Integer(0),Integer(1),Integer(7))    : {(Integer(0),Integer(1),
↪Integer(2))     : Integer(2) },
...     (Integer(0),Integer(1),Integer(7),Integer(9)) : {(Integer(0),Integer(1),
↪Integer(2),Integer(9))    : Integer(2) },
...     (Integer(0),Integer(5))     : {(Integer(0),Integer(1))        : Integer(1),
↪ (Integer(0),Integer(5),Integer(7)) : Integer(2) },
...     (Integer(0),Integer(5),Integer(7))    : {(Integer(0),Integer(5),Integer(7),
↪Integer(9))    : Integer(1) },
...     (Integer(0),Integer(5),Integer(7),Integer(9)) : {(Integer(0),Integer(1),
↪Integer(7),Integer(9))    : Integer(1) },
...     (Integer(0),Integer(8))     : {(Integer(0),Integer(5))        : Integer(1)␣
↪},
...     })
>>> G.is_isomorphic(GG)
True
>>> for (u,v,i) in G.edges(sort=True):
...     print((u.integer_sequence() , v.integer_sequence(), i))
([], [0], 2)
([0], [0, 8], 1)
([0, 1], [0, 1, 7], 2)
([0, 1, 2], [0, 1, 2, 9], 1)
([0, 1, 2, 3], [0, 1, 2, 3, 4], 2)
([0, 1, 2, 6], [0, 1, 2, 3], 1)
([0, 1, 2, 9], [0, 1, 2, 6], 1)
([0, 1, 7], [0, 1, 2], 2)
([0, 1, 7, 9], [0, 1, 2, 9], 2)
([0, 5], [0, 1], 1)
([0, 5], [0, 5, 7], 2)
([0, 5, 7], [0, 5, 7, 9], 1)
([0, 5, 7, 9], [0, 1, 7, 9], 1)
([0, 8], [0, 5], 1)
```

Alcove path crystals are a discrete version of Littelmann paths. We verify that the alcove path crystal is isomorphic to the LS path crystal:

```
sage: C1 = crystals.AlcovePaths(['C',3],[2,1,0])
sage: g1 = C1.digraph() #long time
sage: C2 = crystals.LSPaths(['C',3],[2,1,0])
sage: g2 = C2.digraph() #long time
sage: g1.is_isomorphic(g2, edge_labels=True) #long time
True
```

```
>>> from sage.all import *
>>> C1 = crystals.AlcovePaths(['C',Integer(3)],[Integer(2),Integer(1),Integer(0)])
>>> g1 = C1.digraph() #long time
>>> C2 = crystals.LSPaths(['C',Integer(3)],[Integer(2),Integer(1),Integer(0)])
>>> g2 = C2.digraph() #long time
>>> g1.is_isomorphic(g2, edge_labels=True) #long time
True
```

The preferred initialization method is via explicit weights rather than a Cartan type and the coefficients of the fundamental weights:

```
sage: R = RootSystem(['C',3])
sage: P = R.weight_lattice()
sage: La = P.fundamental_weights()
sage: C = crystals.AlcovePaths(2*La[1]+La[2]); C
Highest weight crystal of alcove paths of type ['C', 3] and weight 2*Lambda[1] +␣
↪Lambda[2]
sage: C1==C
True
```

```
>>> from sage.all import *
>>> R = RootSystem(['C',Integer(3)])
>>> P = R.weight_lattice()
>>> La = P.fundamental_weights()
>>> C = crystals.AlcovePaths(Integer(2)*La[Integer(1)]+La[Integer(2)]); C
Highest weight crystal of alcove paths of type ['C', 3] and weight 2*Lambda[1] +␣
↪Lambda[2]
>>> C1==C
True
```

We now explain the data structure:

```
sage: C = crystals.AlcovePaths(['A',2],[2,0]) ; C
Highest weight crystal of alcove paths of type ['A', 2] and weight 2*Lambda[1]
sage: C._R.lambda_chain()
[(alpha[1], 0), (alpha[1] + alpha[2], 0), (alpha[1], 1), (alpha[1] + alpha[2], 1)]
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(2),Integer(0)]) ; C
Highest weight crystal of alcove paths of type ['A', 2] and weight 2*Lambda[1]
>>> C._R.lambda_chain()
[(alpha[1], 0), (alpha[1] + alpha[2], 0), (alpha[1], 1), (alpha[1] + alpha[2], 1)]
```

The previous list gives the initial "straight line" path from the fundamental alcove A_o to its translation $A_o - \lambda$ where $\lambda = 2\omega_1$ in this example. The initial path for weight λ is called the λ-chain. This path is constructed from the ordered pairs (β, k), by crossing the hyperplane orthogonal to β at height $-k$. We can view a plot of this path as follows:

```
sage: x=C( () )
sage: x.plot() # not tested - outputs a pdf
```

```
>>> from sage.all import *
>>> x=C( () )
>>> x.plot() # not tested - outputs a pdf
```

An element of the crystal is given by a subset of the λ-chain. This subset indicates the hyperplanes where the initial path should be folded. The highest weight element is given by the empty subset.

```
sage: x
()
sage: x.f(1).f(2)
((alpha[1], 1), (alpha[1] + alpha[2], 1))
sage: x.f(1).f(2).integer_sequence()
```

(continues on next page)

(continued from previous page)

```
[2, 3]
sage: C([2,3])
((alpha[1], 1), (alpha[1] + alpha[2], 1))
sage: C([2,3]).is_admissible() #check if a valid vertex
True
sage: C([1,3]).is_admissible() #check if a valid vertex
False
```

```
>>> from sage.all import *
>>> x
()
>>> x.f(Integer(1)).f(Integer(2))
((alpha[1], 1), (alpha[1] + alpha[2], 1))
>>> x.f(Integer(1)).f(Integer(2)).integer_sequence()
[2, 3]
>>> C([Integer(2),Integer(3)])
((alpha[1], 1), (alpha[1] + alpha[2], 1))
>>> C([Integer(2),Integer(3)]).is_admissible() #check if a valid vertex
True
>>> C([Integer(1),Integer(3)]).is_admissible() #check if a valid vertex
False
```

Alcove path crystals now works in affine type (Issue #14143):

```
sage: C = crystals.AlcovePaths(['A',2,1],[1,0,0]) ; C
Highest weight crystal of alcove paths of type ['A', 2, 1] and weight Lambda[0]
sage: x=C(  () )
sage: x.f(0)
((alpha[0], 0),)
sage: C.R
Root system of type ['A', 2, 1]
sage: C.weight
Lambda[0]
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2),Integer(1)],[Integer(1),Integer(0),
↪Integer(0)]) ; C
Highest weight crystal of alcove paths of type ['A', 2, 1] and weight Lambda[0]
>>> x=C(  () )
>>> x.f(Integer(0))
((alpha[0], 0),)
>>> C.R
Root system of type ['A', 2, 1]
>>> C.weight
Lambda[0]
```

Test that the tensor products of Kirillov-Reshetikhin crystals minus non-pseudo-Demazure arrows is in bijection with alcove path construction:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],2,1)
sage: T = crystals.TensorProduct(K,K)
sage: g = T.digraph() #long time
```

(continues on next page)

(continued from previous page)

```
sage: for e in g.edges(sort=False):  #long time
....:     if e[0].phi(0) == 1 and e[2] == 0:
....:         g.delete_edge(e)

sage: C = crystals.AlcovePaths(['B',3,1],[0,2,0], highest_weight_crystal=False)
sage: g2 = C.digraph() #long time
sage: g.is_isomorphic(g2, edge_labels = True) #long time
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(2),
↪Integer(1))
>>> T = crystals.TensorProduct(K,K)
>>> g = T.digraph() #long time
>>> for e in g.edges(sort=False):  #long time
...     if e[Integer(0)].phi(Integer(0)) == Integer(1) and e[Integer(2)] ==␣
↪Integer(0):
...         g.delete_edge(e)

>>> C = crystals.AlcovePaths(['B',Integer(3),Integer(1)],[Integer(0),Integer(2),
↪Integer(0)], highest_weight_crystal=False)
>>> g2 = C.digraph() #long time
>>> g.is_isomorphic(g2, edge_labels = True) #long time
True
```

Note

In type $C_n^{(1)}$, the Kirillov-Reshetikhin crystal is not connected when restricted to pseudo-Demazure arrows, hence the previous example will fail for type $C_n^{(1)}$ crystals.

```
sage: R = RootSystem(['B',3])
sage: P = R.weight_lattice()
sage: La = P.fundamental_weights()
sage: D = crystals.AlcovePaths(2*La[2], highest_weight_crystal=False)
sage: C == D
True
```

```
>>> from sage.all import *
>>> R = RootSystem(['B',Integer(3)])
>>> P = R.weight_lattice()
>>> La = P.fundamental_weights()
>>> D = crystals.AlcovePaths(Integer(2)*La[Integer(2)], highest_weight_
↪crystal=False)
>>> C == D
True
```

Warning

Weights from finite root systems index non-highest weight crystals.

Element

alias of `CrystalOfAlcovePathsElement`

vertices()

Return a list of all the vertices of the crystal.

The vertices are represented as lists of integers recording the folding positions.

One can compute all vertices of the crystal by finding all the admissible subsets of the λ-chain (see method is_admissible, for definition). We use the breadth first search algorithm.

Warning

This method is (currently) only useful for the case when `highest_weight_crystal = False`, where you cannot always reach all vertices of the crystal using crystal operators, starting from the highest weight vertex. This method is typically slower than generating the crystal graph using crystal operators.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['C', 2], [1, 0])
sage: C.vertices()
[[], [0], [0, 1], [0, 1, 2]]
sage: C = crystals.AlcovePaths(['C', 2, 1], [2, 1], False)
sage: len(C.vertices())
80
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['C', Integer(2)], [Integer(1), Integer(0)])
>>> C.vertices()
[[], [0], [0, 1], [0, 1, 2]]
>>> C = crystals.AlcovePaths(['C', Integer(2), Integer(1)], [Integer(2),
→Integer(1)], False)
>>> len(C.vertices())
80
```

The number of elements reachable using the crystal operators from the module generator:

```
sage: len(list(C))
55
```

```
>>> from sage.all import *
>>> len(list(C))
55
```

class sage.combinat.crystals.alcove_path.**CrystalOfAlcovePathsElement**

Bases: `ElementWrapper`

Crystal of alcove paths element.

INPUT:

- `data` – list of folding positions in the lambda chain (indexing starts at 0) or a tuple of *RootsWithHeight* giving folding positions in the lambda chain.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[3,2])
sage: x = C ( () )
sage: x.f(1).f(2)
((alpha[1], 2), (alpha[1] + alpha[2], 4))
sage: x.f(1).f(2).integer_sequence()
[8, 9]
sage: C([8,9])
((alpha[1], 2), (alpha[1] + alpha[2], 4))
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(3),Integer(2)])
>>> x = C ( () )
>>> x.f(Integer(1)).f(Integer(2))
((alpha[1], 2), (alpha[1] + alpha[2], 4))
>>> x.f(Integer(1)).f(Integer(2)).integer_sequence()
[8, 9]
>>> C([Integer(8),Integer(9)])
((alpha[1], 2), (alpha[1] + alpha[2], 4))
```

e(i)

Return the i-th crystal raising operator on `self`.

INPUT:

- `i` – element of the index set of the underlying root system

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[2,0]); C
Highest weight crystal of alcove paths of type ['A', 2] and weight 2*Lambda[1]
sage: x = C( () )
sage: x.e(1)
sage: x.f(1) == x.f(1).f(2).e(2)
True
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(2),Integer(0)]); C
Highest weight crystal of alcove paths of type ['A', 2] and weight 2*Lambda[1]
>>> x = C( () )
>>> x.e(Integer(1))
>>> x.f(Integer(1)) == x.f(Integer(1)).f(Integer(2)).e(Integer(2))
True
```

epsilon(i)

Return the distance to the start of the i-string.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[1,1])
sage: [c.epsilon(1) for c in C]
```

(continues on next page)

(continued from previous page)

```
[0, 1, 0, 0, 1, 0, 1, 2]
sage: [c.epsilon(2) for c in C]
[0, 0, 1, 2, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> [c.epsilon(Integer(1)) for c in C]
[0, 1, 0, 0, 1, 0, 1, 2]
>>> [c.epsilon(Integer(2)) for c in C]
[0, 0, 1, 2, 1, 1, 0, 0]
```

f(*i*)

Return the i-th crystal lowering operator on `self`.

INPUT:

- `i` – element of the index_set of the underlying root_system

EXAMPLES:

```
sage: C=crystals.AlcovePaths(['B',2],[1,1])
sage: x=C(  () )
sage: x.f(1)
((alpha[1], 0),)
sage: x.f(1).f(2)
((alpha[1], 0), (alpha[1] + alpha[2], 2))
```

```
>>> from sage.all import *
>>> C=crystals.AlcovePaths(['B',Integer(2)],[Integer(1),Integer(1)])
>>> x=C(  () )
>>> x.f(Integer(1))
((alpha[1], 0),)
>>> x.f(Integer(1)).f(Integer(2))
((alpha[1], 0), (alpha[1] + alpha[2], 2))
```

integer_sequence()

Return a list of integers corresponding to positions in the λ-chain where it is folded.

Todo

Incorporate this method into the `_repr_` for finite Cartan type.

Note

Only works for finite Cartan types and indexing starts at 0.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[3,2])
sage: x = C( () )
```

(continues on next page)

(continued from previous page)

```
sage: x.f(1).f(2).integer_sequence()
[8, 9]
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(3),Integer(2)])
>>> x = C( () )
>>> x.f(Integer(1)).f(Integer(2)).integer_sequence()
[8, 9]
```

is_admissible()

Diagnostic test to check if `self` is a valid element of the crystal.

If `self.value` is given by

$$(\beta_1, i_1), (\beta_2, i_2), \ldots, (\beta_k, i_k),$$

for highest weight crystals this checks if the sequence

$$1 \rightarrow s_{\beta_1} \rightarrow s_{\beta_1} s_{\beta_2} \rightarrow \cdots \rightarrow s_{\beta_1} s_{\beta_2} \ldots s_{\beta_k}$$

is a path in the Bruhat graph. If `highest_weight_crystal=False`, then the method checks if the above sequence is a path in the quantum Bruhat graph.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[1,1]); C
Highest weight crystal of alcove paths of type ['A', 2] and weight Lambda[1]␣
↪+ Lambda[2]
sage: roots = sorted(C._R._root_lattice.positive_roots()); roots
[alpha[1], alpha[1] + alpha[2], alpha[2]]
sage: r1 = C._R(roots[0],0); r1
(alpha[1], 0)
sage: r2 = C._R(roots[2],0); r2
(alpha[2], 0)
sage: r3 = C._R(roots[1],1); r3
(alpha[1] + alpha[2], 1)
sage: x = C( ( r1,r2) )
sage: x.is_admissible()
True
sage: x = C( (r3,) ); x
((alpha[1] + alpha[2], 1),)
sage: x.is_admissible()
False
sage: C = crystals.AlcovePaths(['C',2,1],[2,1],False)
sage: C([7,8]).is_admissible()
True
sage: C = crystals.AlcovePaths(['A',2],[3,2])
sage: C([2,3]).is_admissible()
True
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(1),Integer(1)]); C
Highest weight crystal of alcove paths of type ['A', 2] and weight Lambda[1]␣
```

(continues on next page)

(continued from previous page)

```
↪+ Lambda[2]
>>> roots = sorted(C._R._root_lattice.positive_roots()); roots
[alpha[1], alpha[1] + alpha[2], alpha[2]]
>>> r1 = C._R(roots[Integer(0)],Integer(0)); r1
(alpha[1], 0)
>>> r2 = C._R(roots[Integer(2)],Integer(0)); r2
(alpha[2], 0)
>>> r3 = C._R(roots[Integer(1)],Integer(1)); r3
(alpha[1] + alpha[2], 1)
>>> x = C( ( r1,r2) )
>>> x.is_admissible()
True
>>> x = C( (r3,) ); x
((alpha[1] + alpha[2], 1),)
>>> x.is_admissible()
False
>>> C = crystals.AlcovePaths(['C',Integer(2),Integer(1)],[Integer(2),
↪Integer(1)],False)
>>> C([Integer(7),Integer(8)]).is_admissible()
True
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(3),Integer(2)])
>>> C([Integer(2),Integer(3)]).is_admissible()
True
```

Todo

Better doctest

path()

Return the path in the (quantum) Bruhat graph corresponding to `self`.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['B', 3], [3,1,2])
sage: b = C.highest_weight_vector().f_string([1,3,2,1,3,1])
sage: b.path()
[1, s1, s3*s1, s2*s3*s1, s3*s2*s3*s1]
sage: b = C.highest_weight_vector().f_string([2,3,3,2])
sage: b.path()
[1, s2, s3*s2, s2*s3*s2]
sage: b = C.highest_weight_vector().f_string([2,3,3,2,1])
sage: b.path()
[1, s2, s3*s2, s2*s3*s2, s1*s2*s3*s2]
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['B', Integer(3)], [Integer(3),Integer(1),
↪Integer(2)])
>>> b = C.highest_weight_vector().f_string([Integer(1),Integer(3),Integer(2),
↪Integer(1),Integer(3),Integer(1)])
>>> b.path()
[1, s1, s3*s1, s2*s3*s1, s3*s2*s3*s1]
```

(continues on next page)

(continued from previous page)

```
>>> b = C.highest_weight_vector().f_string([Integer(2),Integer(3),Integer(3),
↪Integer(2)])
>>> b.path()
[1, s2, s3*s2, s2*s3*s2]
>>> b = C.highest_weight_vector().f_string([Integer(2),Integer(3),Integer(3),
↪Integer(2),Integer(1)])
>>> b.path()
[1, s2, s3*s2, s2*s3*s2, s1*s2*s3*s2]
```

phi (i)

Return the distance to the end of the i-string.

This method overrides the generic implementation in the category of crystals since this computation is more efficient.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[1,1])
sage: [c.phi(1) for c in C]
[1, 0, 0, 1, 0, 2, 1, 0]
sage: [c.phi(2) for c in C]
[1, 2, 1, 0, 0, 0, 0, 1]
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> [c.phi(Integer(1)) for c in C]
[1, 0, 0, 1, 0, 2, 1, 0]
>>> [c.phi(Integer(2)) for c in C]
[1, 2, 1, 0, 0, 0, 0, 1]
```

plot ()

Return a plot self.

Note

Currently only implemented for types A_2, B_2, and C_2.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[2,0])
sage: x = C( () ).f(1).f(2)
sage: x.plot() # Not tested - creates a pdf
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(2),Integer(0)])
>>> x = C( () ).f(Integer(1)).f(Integer(2))
>>> x.plot() # Not tested - creates a pdf
```

weight ()

Return the weight of self.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[2,0])
sage: for i in C: i.weight()
(2, 0, 0)
(1, 1, 0)
(0, 2, 0)
(0, -1, 0)
(-1, 0, 0)
(-2, -2, 0)
sage: B = crystals.AlcovePaths(['A',2,1],[1,0,0])
sage: p = B.module_generators[0].f_string([0,1,2])
sage: p.weight()
Lambda[0] - delta
```

```
>>> from sage.all import *
>>> C = crystals.AlcovePaths(['A',Integer(2)],[Integer(2),Integer(0)])
>>> for i in C: i.weight()
(2, 0, 0)
(1, 1, 0)
(0, 2, 0)
(0, -1, 0)
(-1, 0, 0)
(-2, -2, 0)
>>> B = crystals.AlcovePaths(['A',Integer(2),Integer(1)],[Integer(1),
↪Integer(0),Integer(0)])
>>> p = B.module_generators[Integer(0)].f_string([Integer(0),Integer(1),
↪Integer(2)])
>>> p.weight()
Lambda[0] - delta
```

class sage.combinat.crystals.alcove_path.**InfinityCrystalOfAlcovePaths**(*cartan_type*)

Bases: UniqueRepresentation, Parent

$\mathcal{B}(\infty)$ crystal of alcove paths.

class Element(*parent*, *elt*, *shift*)

Bases: ElementWrapper

Initialize self.

EXAMPLES:

```
sage: A = crystals.infinity.AlcovePaths(['F',4])
sage: mg = A.highest_weight_vector()
sage: x = mg.f_string([2,3,1,4,4,2,3,1])
sage: TestSuite(x).run()
```

```
>>> from sage.all import *
>>> A = crystals.infinity.AlcovePaths(['F',Integer(4)])
>>> mg = A.highest_weight_vector()
>>> x = mg.f_string([Integer(2),Integer(3),Integer(1),Integer(4),Integer(4),
↪Integer(2),Integer(3),Integer(1)])
>>> TestSuite(x).run()
```

e(*i*)

Return the action of e_i on self.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: A = crystals.infinity.AlcovePaths(['D',5,1])
sage: mg = A.highest_weight_vector()
sage: x = mg.f_string([1,3,4,2,5,4,5,5])
sage: x.f(4).e(5) == x.e(5).f(4)
True
```

```
>>> from sage.all import *
>>> A = crystals.infinity.AlcovePaths(['D',Integer(5),Integer(1)])
>>> mg = A.highest_weight_vector()
>>> x = mg.f_string([Integer(1),Integer(3),Integer(4),Integer(2),
↪Integer(5),Integer(4),Integer(5),Integer(5)])
>>> x.f(Integer(4)).e(Integer(5)) == x.e(Integer(5)).f(Integer(4))
True
```

epsilon(*i*)

Return ε_i of self.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: A = crystals.infinity.AlcovePaths(['A',7,2])
sage: mg = A.highest_weight_vector()
sage: x = mg.f_string([1,0,2,3,4,4,4,2,3,3,3])
sage: [x.epsilon(i) for i in A.index_set()]
[0, 0, 0, 3, 0]
sage: x = mg.f_string([2,2,1,1,0,1,0,2,3,3,3,4])
sage: [x.epsilon(i) for i in A.index_set()]
[1, 2, 0, 1, 1]
```

```
>>> from sage.all import *
>>> A = crystals.infinity.AlcovePaths(['A',Integer(7),Integer(2)])
>>> mg = A.highest_weight_vector()
>>> x = mg.f_string([Integer(1),Integer(0),Integer(2),Integer(3),
↪Integer(4),Integer(4),Integer(4),Integer(2),Integer(3),Integer(3),
↪Integer(3)])
>>> [x.epsilon(i) for i in A.index_set()]
[0, 0, 0, 3, 0]
>>> x = mg.f_string([Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(0),Integer(1),Integer(0),Integer(2),Integer(3),Integer(3),
↪Integer(3),Integer(4)])
>>> [x.epsilon(i) for i in A.index_set()]
[1, 2, 0, 1, 1]
```

f(*i*)

Return the action of f_i on self.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: A = crystals.infinity.AlcovePaths(['E',7,1])
sage: mg = A.highest_weight_vector()
sage: mg.f_string([1,3,5,6,4,2,0,2,1,0,2,4,7,4,2])
((alpha[2], -3), (alpha[5], -1), (alpha[1], -1),
 (alpha[0] + alpha[1], -2),
 (alpha[2] + alpha[4] + alpha[5], -2),
 (alpha[5] + alpha[6], -1), (alpha[1] + alpha[3], -1),
 (alpha[5] + alpha[6] + alpha[7], -1),
 (alpha[0] + alpha[1] + alpha[3], -1),
 (alpha[1] + alpha[3] + alpha[4] + alpha[5], -1))
```

```
>>> from sage.all import *
>>> A = crystals.infinity.AlcovePaths(['E',Integer(7),Integer(1)])
>>> mg = A.highest_weight_vector()
>>> mg.f_string([Integer(1),Integer(3),Integer(5),Integer(6),Integer(4),
↪Integer(2),Integer(0),Integer(2),Integer(1),Integer(0),Integer(2),
↪Integer(4),Integer(7),Integer(4),Integer(2)])
((alpha[2], -3), (alpha[5], -1), (alpha[1], -1),
 (alpha[0] + alpha[1], -2),
 (alpha[2] + alpha[4] + alpha[5], -2),
 (alpha[5] + alpha[6], -1), (alpha[1] + alpha[3], -1),
 (alpha[5] + alpha[6] + alpha[7], -1),
 (alpha[0] + alpha[1] + alpha[3], -1),
 (alpha[1] + alpha[3] + alpha[4] + alpha[5], -1))
```

phi(*i*)

Return φ_i of `self`.

Let $A \in \mathcal{B}(\infty)$ Define $\varphi_i(A) := \varepsilon_i(A) + \langle h_i, \mathrm{wt}(A) \rangle$, where h_i is the i-th simple coroot and $\mathrm{wt}(A)$ is the *weight()* of A.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.infinity.AlcovePaths(['A',8,2])
sage: mg = A.highest_weight_vector()
sage: x = mg.f_string([1,0,2,3,4,4,4,2,3,3,3])
sage: [x.phi(i) for i in A.index_set()]
[1, 1, 1, 3, -2]
sage: x = mg.f_string([2,2,1,1,0,1,0,2,3,3,3,4])
sage: [x.phi(i) for i in A.index_set()]
[4, -1, 0, 0, 2]
```

```
>>> from sage.all import *
>>> A = crystals.infinity.AlcovePaths(['A',Integer(8),Integer(2)])
>>> mg = A.highest_weight_vector()
>>> x = mg.f_string([Integer(1),Integer(0),Integer(2),Integer(3),
↪Integer(4),Integer(4),Integer(4),Integer(2),Integer(3),Integer(3),
↪Integer(3)])
>>> [x.phi(i) for i in A.index_set()]
[1, 1, 1, 3, -2]
>>> x = mg.f_string([Integer(2),Integer(2),Integer(1),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0),Integer(1),Integer(0),Integer(2),Integer(3),Integer(3),
↪Integer(3),Integer(4)])
>>> [x.phi(i) for i in A.index_set()]
[4, -1, 0, 0, 2]
```

projection(*k=None*)

Return the projection `self` onto $B(k\rho)$.

INPUT:

- `k` – (optional) if not given, defaults to the smallest value such that `self` is not `None` under the projection

EXAMPLES:

```
sage: A = crystals.infinity.AlcovePaths(['G',2])
sage: mg = A.highest_weight_vector()
sage: x = mg.f_string([2,1,1,2,2,2,1,1]); x
((alpha[2], -3), (alpha[1] + alpha[2], -3),
 (3*alpha[1] + 2*alpha[2], -1), (2*alpha[1] + alpha[2], -1))
sage: x.projection()
((alpha[2], 0), (alpha[1] + alpha[2], 9),
 (3*alpha[1] + 2*alpha[2], 8), (2*alpha[1] + alpha[2], 14))
sage: x.projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight 3*Lambda[1] + 3*Lambda[2]

sage: mg.projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight 0
sage: mg.f(1).projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight Lambda[1] + Lambda[2]
sage: mg.f(1).f(2).projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight Lambda[1] + Lambda[2]
sage: b = mg.f_string([1,2,2,1,2])
sage: b.projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight 2*Lambda[1] + 2*Lambda[2]
sage: b.projection(3).parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight 3*Lambda[1] + 3*Lambda[2]
sage: b.projection(1)
```

```
>>> from sage.all import *
>>> A = crystals.infinity.AlcovePaths(['G',Integer(2)])
>>> mg = A.highest_weight_vector()
>>> x = mg.f_string([Integer(2),Integer(1),Integer(1),Integer(2),
↪Integer(2),Integer(2),Integer(1),Integer(1)]); x
((alpha[2], -3), (alpha[1] + alpha[2], -3),
 (3*alpha[1] + 2*alpha[2], -1), (2*alpha[1] + alpha[2], -1))
>>> x.projection()
((alpha[2], 0), (alpha[1] + alpha[2], 9),
```

(continues on next page)

(continued from previous page)

```
 (3*alpha[1] + 2*alpha[2], 8), (2*alpha[1] + alpha[2], 14))
>>> x.projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight 3*Lambda[1] + 3*Lambda[2]

>>> mg.projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight 0
>>> mg.f(Integer(1)).projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight Lambda[1] + Lambda[2]
>>> mg.f(Integer(1)).f(Integer(2)).projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight Lambda[1] + Lambda[2]
>>> b = mg.f_string([Integer(1),Integer(2),Integer(2),Integer(1),
↪Integer(2)])
>>> b.projection().parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight 2*Lambda[1] + 2*Lambda[2]
>>> b.projection(Integer(3)).parent()
Highest weight crystal of alcove paths of type ['G', 2]
 and weight 3*Lambda[1] + 3*Lambda[2]
>>> b.projection(Integer(1))
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: A = crystals.infinity.AlcovePaths(['E',6])
sage: mg = A.highest_weight_vector()
sage: fstr = [1,3,4,2,1,2,3,6,5,3,2,6,2]
sage: x = mg.f_string(fstr)
sage: al = A.weight_lattice_realization().simple_roots()
sage: x.weight() == -sum(al[i]*fstr.count(i) for i in A.index_set())
True
```

```
>>> from sage.all import *
>>> A = crystals.infinity.AlcovePaths(['E',Integer(6)])
>>> mg = A.highest_weight_vector()
>>> fstr = [Integer(1),Integer(3),Integer(4),Integer(2),Integer(1),
↪Integer(2),Integer(3),Integer(6),Integer(5),Integer(3),Integer(2),
↪Integer(6),Integer(2)]
>>> x = mg.f_string(fstr)
>>> al = A.weight_lattice_realization().simple_roots()
>>> x.weight() == -sum(al[i]*fstr.count(i) for i in A.index_set())
True
```

class sage.combinat.crystals.alcove_path.**RootsWithHeight**(*weight*)

Bases: `UniqueRepresentation`, `Parent`

Data structure of the ordered pairs (β, k), where β is a positive root and k is a nonnegative integer. A total order is implemented on this set, and depends on the weight.

INPUT:

- `cartan_type` – Cartan type of a finite or affine untwisted root system
- `weight` – dominant weight as a list of (integral) coefficients of the fundamental weights

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import RootsWithHeight
sage: R = RootsWithHeight(['A',2],[1,1]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] +
→Lambda[2]

sage: r1 = R._root_lattice.from_vector(vector([1,0])); r1
alpha[1]
sage: r2 = R._root_lattice.from_vector(vector([1,1])); r2
alpha[1] + alpha[2]

sage: x = R(r1,0); x
(alpha[1], 0)
sage: y = R(r2,1); y
(alpha[1] + alpha[2], 1)
sage: x < y
True
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.alcove_path import RootsWithHeight
>>> R = RootsWithHeight(['A',Integer(2)],[Integer(1),Integer(1)]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] +
→Lambda[2]

>>> r1 = R._root_lattice.from_vector(vector([Integer(1),Integer(0)])); r1
alpha[1]
>>> r2 = R._root_lattice.from_vector(vector([Integer(1),Integer(1)])); r2
alpha[1] + alpha[2]

>>> x = R(r1,Integer(0)); x
(alpha[1], 0)
>>> y = R(r2,Integer(1)); y
(alpha[1] + alpha[2], 1)
>>> x < y
True
```

Element

alias of *RootsWithHeightElement*

lambda_chain()

Return the unfolded λ-chain.

Note

Only works in root systems of finite type.

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import RootsWithHeight
sage: R = RootsWithHeight(['A',2],[1,1]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] +
→Lambda[2]
sage: R.lambda_chain()
[(alpha[2], 0), (alpha[1] + alpha[2], 0), (alpha[1], 0), (alpha[1] + alpha[2],
→ 1)]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.alcove_path import RootsWithHeight
>>> R = RootsWithHeight(['A',Integer(2)],[Integer(1),Integer(1)]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] +
→Lambda[2]
>>> R.lambda_chain()
[(alpha[2], 0), (alpha[1] + alpha[2], 0), (alpha[1], 0), (alpha[1] + alpha[2],
→ 1)]
```

word()

Give the initial alcove path (λ-chain) in terms of simple roots. Used for plotting the path.

Note

Currently only implemented for finite Cartan types.

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import RootsWithHeight
sage: R = RootsWithHeight(['A',2],[3,2])
sage: R.word()
[2, 1, 2, 0, 1, 2, 1, 0, 1, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.alcove_path import RootsWithHeight
>>> R = RootsWithHeight(['A',Integer(2)],[Integer(3),Integer(2)])
>>> R.word()
[2, 1, 2, 0, 1, 2, 1, 0, 1, 2]
```

class sage.combinat.crystals.alcove_path.**RootsWithHeightElement**(*parent*, *root*, *height*)

Bases: `Element`

Element of *RootsWithHeight*.

INPUT:

- `root` – a positive root β in our root system
- `height` – integer such that $0 \leq l \leq \langle \lambda, \beta^\vee \rangle$

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import RootsWithHeight
sage: rl = RootSystem(['A',2]).root_lattice()
sage: x = rl.from_vector(vector([1,1])); x
```

(continues on next page)

(continued from previous page)

```
alpha[1] + alpha[2]
sage: R = RootsWithHeight(['A',2],[1,1]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] +␣
↪Lambda[2]
sage: y = R(x, 1); y
(alpha[1] + alpha[2], 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.alcove_path import RootsWithHeight
>>> rl = RootSystem(['A',Integer(2)]).root_lattice()
>>> x = rl.from_vector(vector([Integer(1),Integer(1)])); x
alpha[1] + alpha[2]
>>> R = RootsWithHeight(['A',Integer(2)],[Integer(1),Integer(1)]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] +␣
↪Lambda[2]
>>> y = R(x, Integer(1)); y
(alpha[1] + alpha[2], 1)
```

sage.combinat.crystals.alcove_path.**compare_graphs**(*g1*, *g2*, *node1*, *node2*)

Compare two edge-labeled `graphs` obtained from `Crystal.digraph()`, starting from the root nodes of each graph.

- `g1` – `graphs`, first digraph
- `g2` – `graphs`, second digraph
- `node1` – element of `g1`
- `node2` – element of `g2`

Traverse `g1` starting at `node1` and compare this graph with the one obtained by traversing `g2` starting with `node2`. If the graphs match (including labels) then return `True`. Return `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import compare_graphs
sage: G1 = crystals.Tableaux(['A',3], shape=[1,1]).digraph()
sage: C = crystals.AlcovePaths(['A',3],[0,1,0])
sage: G2 = C.digraph()
sage: compare_graphs(G1, G2, C( () ), G2.vertices(sort=True)[0])
True
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.alcove_path import compare_graphs
>>> G1 = crystals.Tableaux(['A',Integer(3)], shape=[Integer(1),Integer(1)]).
↪digraph()
>>> C = crystals.AlcovePaths(['A',Integer(3)],[Integer(0),Integer(1),Integer(0)])
>>> G2 = C.digraph()
>>> compare_graphs(G1, G2, C( () ), G2.vertices(sort=True)[Integer(0)])
True
```

5.1.39 Crystals

Introductory material

- *An introduction to crystals*
- The Lie Methods and Related Combinatorics thematic tutorial

Catalogs of crystals

- *Catalog Of Crystals*

See also

- The categories for crystals: `Crystals`, `HighestWeightCrystals`, `FiniteCrystals`, `ClassicalCrystals`, `RegularCrystals`, `RegularSuperCrystals` – The categories for crystals
- *Root Systems*

5.1.40 Benkart-Kang-Kashiwara crystals for the general-linear Lie superalgebra

class sage.combinat.crystals.bkk_crystals.**CrystalOfBKKTableaux**(*ct*, *shape*)

Bases: *CrystalOfWords*

Crystal of tableaux for type $A(m|n)$.

This is an implementation of the tableaux model of the Benkart-Kang-Kashiwara crystal [BKK2000] for the Lie superalgebra $\mathfrak{gl}(m+1, n+1)$.

INPUT:

- `ct` – a super Lie Cartan type of type $A(m|n)$
- `shape` – shape specifying the highest weight; this should be a partition contained in a hook of height $n+1$ and width $m+1$

EXAMPLES:

```
sage: T = crystals.Tableaux(['A', [1,1]], shape = [2,1])
sage: T.cardinality()
20
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A', [Integer(1),Integer(1)]], shape = [Integer(2),
→Integer(1)])
>>> T.cardinality()
20
```

class Element

Bases: *CrystalOfBKKTableauxElement*

genuine_highest_weight_vectors(*index_set=None*)

Return a tuple of genuine highest weight elements.

A *fake highest weight vector* is one which is annihilated by e_i for all i in the index set, but whose weight is not bigger in dominance order than all other elements in the crystal. A *genuine highest weight vector* is a highest weight element that is not fake.

EXAMPLES:

```
sage: B = crystals.Tableaux(['A', [1,1]], shape=[3,2,1])
sage: B.genuine_highest_weight_vectors()
([[-2, -2, -2], [-1, -1], [1]],)
sage: B.highest_weight_vectors()
([[-2, -2, -2], [-1, -1], [1]],
 [[-2, -2, -2], [-1, 2], [1]],
 [[-2, -2, 2], [-1, -1], [1]])
```

```
>>> from sage.all import *
>>> B = crystals.Tableaux(['A', [Integer(1),Integer(1)]], shape=[Integer(3),
↪Integer(2),Integer(1)])
>>> B.genuine_highest_weight_vectors()
([[-2, -2, -2], [-1, -1], [1]],)
>>> B.highest_weight_vectors()
([[-2, -2, -2], [-1, -1], [1]],
 [[-2, -2, -2], [-1, 2], [1]],
 [[-2, -2, 2], [-1, -1], [1]])
```

shape()

Return the shape of `self`.

EXAMPLES:

```
sage: T = crystals.Tableaux(['A', [1, 2]], shape=[2,1])
sage: T.shape()
[2, 1]
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A', [Integer(1), Integer(2)]], shape=[Integer(2),
↪Integer(1)])
>>> T.shape()
[2, 1]
```

5.1.41 Catalog Of Crystals

Let I be an index set and let $(A, \Pi, \Pi^\vee, P, P^\vee)$ be a Cartan datum associated with generalized Cartan matrix $A = (a_{ij})_{i,j\in I}$. An *abstract crystal* associated to this Cartan datum is a set B together with maps

$$e_i, f_i \colon B \to B \cup \{0\}, \qquad \varepsilon_i, \varphi_i \colon B \to \mathbf{Z} \cup \{-\infty\}, \qquad \mathrm{wt}\colon B \to P,$$

subject to the following conditions:

1. $\varphi_i(b) = \varepsilon_i(b) + \langle h_i, \mathrm{wt}(b)\rangle$ for all $b \in B$ and $i \in I$;
2. $\mathrm{wt}(e_i b) = \mathrm{wt}(b) + \alpha_i$ if $e_i b \in B$;
3. $\mathrm{wt}(f_i b) = \mathrm{wt}(b) - \alpha_i$ if $f_i b \in B$;
4. $\varepsilon_i(e_i b) = \varepsilon_i(b) - 1$, $\varphi_i(e_i b) = \varphi_i(b) + 1$ if $e_i b \in B$;
5. $\varepsilon_i(f_i b) = \varepsilon_i(b) + 1$, $\varphi_i(f_i b) = \varphi_i(b) - 1$ if $f_i b \in B$;
6. $f_i b = b'$ if and only if $b = e_i b'$ for $b, b' \in B$ and $i \in I$;
7. if $\varphi_i(b) = -\infty$ for $b \in B$, then $e_i b = f_i b = 0$.

See also

- `sage.categories.crystals`
- `sage.combinat.crystals.crystals`

Catalog

This is a catalog of crystals that are currently implemented in Sage:

- `AffineCrystalFromClassical`
- `AffineCrystalFromClassicalAndPromotion`
- `AffineFactorization`
- `AffinizationOf`
- `AlcovePaths`
- `FastRankTwo`
- `FullyCommutativeStableGrothendieck`
- `GeneralizedYoungWalls`
- `HighestWeight`
- `Induced`
- `KacModule`
- `KirillovReshetikhin`
- `KleshchevPartitions`
- `KyotoPathModel`
- `Letters`
- `LSPaths`
- `Minimaj`
- `NakajimaMonomials`
- `OddNegativeRoots`
- `ProjectedLevelZeroLSPaths`
- `RiggedConfigurations`
- `ShiftedPrimedTableaux`
- `Spins`
- `SpinsPlus`
- `SpinsMinus`
- `Tableaux`

Subcatalogs:

- *Catalog Of Crystal Models For B(\infty)*
- *Catalog Of Elementary Crystals*
- *Catalog Of Crystal Models For Kirillov-Reshetikhin Crystals*

Functorial constructions:

- *DirectSum*
- *TensorProduct*

5.1.42 Catalog Of Elementary Crystals

See *elementary_crystals*.

- *Component*
- *Elementary* or *B*
- *R*
- *T*

5.1.43 Catalog Of Crystal Models For $B(\infty)$

We currently have the following models:

- *AlcovePaths*
- *GeneralizedYoungWalls*
- *LSPaths*
- *Multisegments*
- *MVPolytopes*
- *NakajimaMonomials*
- *PBW*
- *PolyhedralRealization*
- *RiggedConfigurations*
- *Star*
- *Tableaux*

5.1.44 Catalog Of Crystal Models For Kirillov-Reshetikhin Crystals

We currently have the following models:

- *KashiwaraNakashimaTableaux*
- *KirillovReshetikhinTableaux*
- *LSPaths*
- *RiggedConfigurations*

5.1.45 An introduction to crystals

Informally, a crystal $\mathcal{B}$ is an oriented graph with edges colored in some set I such that, for each $i \in I$, each node x has:

- at most one i-successor, denoted $f_i x$;
- at most one i-predecessor, denoted $e_i x$.

By convention, one writes $f_i x = \emptyset$ and $e_i x = \emptyset$ when x has no successor resp. predecessor.

One may think of $\mathcal{B}$ as essentially a deterministic automaton whose dual is also deterministic; in this context, the f_i's and e_i's are respectively the transition functions of the automaton and of its dual, and $\emptyset$ is the sink.

A crystal comes further endowed with a weight function $\mathrm{wt} : \mathcal{B} \to L$ which satisfies appropriate conditions.

In combinatorial representation theory, crystals are used as combinatorial data to model representations of Lie algebra.

Axiomatic definition

Let C be a Cartan type (`CartanType`) with index set I, and L be a realization of the weight lattice of the type C. Let α_i and $\alpha_i^\vee$ denote the simple roots and coroots respectively.

A type C crystal is a non-empty set $\mathcal{B}$ endowed with maps $\mathrm{wt} : \mathcal{B} \to L$, $e_i, f_i : \mathcal{B} \to \mathcal{B} \cup \{\emptyset\}$, and $\varepsilon_i, \varphi_i : \mathcal{B} \to \mathbf{Z} \cup \{-\infty\}$ for $i \in I$ satisfying the following properties for all $i \in I$:

- for $b, b' \in \mathcal{B}$, we have $f_i b' = b$ if and only if $e_i b = b'$;
- if $e_i b \in \mathcal{B}$, then:
  - $\mathrm{wt}(e_i b) = \mathrm{wt}(b) + \alpha_i$,
  - $\varepsilon_i(e_i b) = \varepsilon_i(b) - 1$,
  - $\varphi_i(e_i b) = \varphi_i(b) + 1$;
- if $f_i b \in \mathcal{B}$, then:
  - $\mathrm{wt}(f_i b) = \mathrm{wt}(b) - \alpha_i$,
  - $\varepsilon_i(f_i b) = \varepsilon_i(b) + 1$,
  - $\varphi_i(f_i b) = \varphi_i(b) - 1$;
- $\varphi_i(b) = \varepsilon_i(b) + \langle \alpha_i^\vee, \mathrm{wt}(b) \rangle$,
- if $\varphi_i(b) = -\infty$ for $b \in \mathcal{B}$, then $e_i b = f_i b = \emptyset$.

Some further conditions are required to guarantee that this data indeed models a representation of a Lie algebra. For finite simply laced types a complete characterization is given by Stembridge's local axioms [Ste2003].

EXAMPLES:

We construct the type A_5 crystal on letters (or in representation theoretic terms, the highest weight crystal of type A_5 corresponding to the highest weight Λ_1):

```
sage: C = crystals.Letters(['A',5]); C
The crystal of letters for type ['A', 5]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)]); C
The crystal of letters for type ['A', 5]
```

It has a single highest weight element:

```
sage: C.highest_weight_vectors()
(1,)
```

```
>>> from sage.all import *
>>> C.highest_weight_vectors()
(1,)
```

A crystal is an enumerated set (see `EnumeratedSets`); and we can count and list its elements in the usual way:

```
sage: C.cardinality()
6
sage: C.list()
[1, 2, 3, 4, 5, 6]
```

```
>>> from sage.all import *
>>> C.cardinality()
6
>>> C.list()
[1, 2, 3, 4, 5, 6]
```

as well as use it in for loops:

```
sage: [x for x in C]
[1, 2, 3, 4, 5, 6]
```

```
>>> from sage.all import *
>>> [x for x in C]
[1, 2, 3, 4, 5, 6]
```

Here are some more elaborate crystals (see their respective documentations):

```
sage: Tens = crystals.TensorProduct(C, C)
sage: Spin = crystals.Spins(['B', 3])
sage: Tab  = crystals.Tableaux(['A', 3], shape = [2,1,1])
sage: Fast = crystals.FastRankTwo(['B', 2], shape = [3/2, 1/2])
sage: KR = crystals.KirillovReshetikhin(['A',2,1],1,1)
```

```
>>> from sage.all import *
>>> Tens = crystals.TensorProduct(C, C)
>>> Spin = crystals.Spins(['B', Integer(3)])
>>> Tab  = crystals.Tableaux(['A', Integer(3)], shape = [Integer(2),Integer(1),
↪Integer(1)])
>>> Fast = crystals.FastRankTwo(['B', Integer(2)], shape = [Integer(3)/Integer(2),␣
↪Integer(1)/Integer(2)])
>>> KR = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],Integer(1),
↪Integer(1))
```

One can get (currently) crude plotting via:

```
sage: Tab.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 52 graphics primitives
```

```
>>> from sage.all import *
>>> Tab.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 52 graphics primitives
```

If dot2tex is installed, one can obtain nice latex pictures via:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 1,1)
sage: view(K, pdflatex=True) # optional - dot2tex graphviz, not tested (opens␣
↪external window)
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(1),
↪Integer(1))
>>> view(K, pdflatex=True) # optional - dot2tex graphviz, not tested (opens external␣
↪window)
```

or with colored edges:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 1,1)
sage: G = K.digraph()
sage: G.set_latex_options(color_by_label={0:"black", 1:"red", 2:"blue", 3:"green"})
sage: view(G, pdflatex=True) # optional - dot2tex graphviz, not tested (opens␣
↪external window)
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(1),
↪Integer(1))
>>> G = K.digraph()
>>> G.set_latex_options(color_by_label={Integer(0):"black", Integer(1):"red",␣
↪Integer(2):"blue", Integer(3):"green"})
>>> view(G, pdflatex=True) # optional - dot2tex graphviz, not tested (opens external␣
↪window)
```

For rank two crystals, there is an alternative method of getting metapost pictures. For more information see `C.metapost?`
.

See also

The overview of crystal features in Sage

Todo

- Vocabulary and conventions:
  - For a classical crystal: connected / highest weight / irreducible
  - …
- Layout instructions for plot() for rank 2 types
- RestrictionOfCrystal

The crystals library in Sage grew up from an initial implementation in MuPAD-Combinat (see <MuPAD-Combinat>/lib/COMBINAT/crystals.mu).

class `sage.combinat.crystals.crystals.`**`CrystalBacktracker`**(*crystal*, *index_set=None*)

Bases: `GenericBacktracker`

Time complexity: $O(nF)$ amortized for each produced element, where n is the size of the index set, and F is the cost of computing e and f operators.

Memory complexity: $O(D)$ where D is the depth of the crystal.

Principle of the algorithm:

Let C be a classical crystal. It's an acyclic graph where each connected component has a unique element without predecessors (the highest weight element for this component). Let's assume for simplicity that C is irreducible (i.e. connected) with highest weight element u.

One can define a natural spanning tree of C by taking u as the root of the tree, and for any other element y taking as ancestor the element x such that there is an i-arrow from x to y with i minimal. Then, a path from u to y describes the lexicographically smallest sequence $i_1, \dots, i_k$ such that $(f_{i_k} \circ f_{i_1})(u) = y$.

Morally, the iterator implemented below just does a depth first search walk through this spanning tree. In practice, this can be achieved recursively as follows: take an element x, and consider in turn each successor $y = f_i(x)$, ignoring those such that $y = f_j(x')$ for some x' and $j < i$ (this can be tested by computing $e_j(y)$ for $j < i$).

EXAMPLES:

```
sage: from sage.combinat.crystals.crystals import CrystalBacktracker
sage: C = crystals.Tableaux(['B',3],shape=[3,2,1])
sage: CB = CrystalBacktracker(C)
sage: len(list(CB))
1617
sage: CB = CrystalBacktracker(C, [1,2])
sage: len(list(CB))
8
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.crystals import CrystalBacktracker
>>> C = crystals.Tableaux(['B',Integer(3)],shape=[Integer(3),Integer(2),
↪Integer(1)])
>>> CB = CrystalBacktracker(C)
>>> len(list(CB))
1617
>>> CB = CrystalBacktracker(C, [Integer(1),Integer(2)])
>>> len(list(CB))
8
```

5.1.46 Direct Sum of Crystals

class sage.combinat.crystals.direct_sum.**DirectSumOfCrystals**(*crystals*, *facade*, *keepkey*, *category*, ***options*)

Bases: DisjointUnionEnumeratedSets

Direct sum of crystals.

Given a list of crystals $B_0, \dots, B_k$ of the same Cartan type, one can form the direct sum $B_0 \oplus \cdots \oplus B_k$.

INPUT:

- `crystals` – list of crystals of the same Cartan type
- `keepkey` – boolean

The option `keepkey` is by default set to `False`, assuming that the crystals are all distinct. In this case the elements of the direct sum are just represented by the elements in the crystals B_i. If the crystals are not all distinct, one should set the `keepkey` option to `True`. In this case, the elements of the direct sum are represented as tuples (i, b) where $b \in B_i$.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: C1 = crystals.Tableaux(['A',2],shape=[1,1])
sage: B = crystals.DirectSum([C,C1])
sage: B.list()
[1, 2, 3, [[1], [2]], [[1], [3]], [[2], [3]]]
sage: [b.f(1) for b in B]
[2, None, None, None, [[2], [3]], None]
sage: B.module_generators
(1, [[1], [2]])
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> C1 = crystals.Tableaux(['A',Integer(2)],shape=[Integer(1),Integer(1)])
>>> B = crystals.DirectSum([C,C1])
>>> B.list()
[1, 2, 3, [[1], [2]], [[1], [3]], [[2], [3]]]
>>> [b.f(Integer(1)) for b in B]
[2, None, None, None, [[2], [3]], None]
>>> B.module_generators
(1, [[1], [2]])
```

```
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: B.list()
[(0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3)]
sage: B.module_generators
((0, 1), (1, 1))
sage: b = B( tuple([0,C(1)]) )
sage: b
(0, 1)
sage: b.weight()
(1, 0, 0)
```

```
>>> from sage.all import *
>>> B = crystals.DirectSum([C,C], keepkey=True)
>>> B.list()
[(0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3)]
>>> B.module_generators
((0, 1), (1, 1))
>>> b = B( tuple([Integer(0),C(Integer(1))]) )
>>> b
(0, 1)
>>> b.weight()
(1, 0, 0)
```

The following is required, because *`DirectSumOfCrystals`* takes the same arguments as `DisjointUnionEnumeratedSets` (which see for details).

class Element

Bases: `ElementWrapper`

A class for elements of direct sums of crystals.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: [[b, b.e(2)] for b in B]
[[(0, 1), None], [(0, 2), None], [(0, 3), (0, 2)], [(1, 1), None], [(1,
→2), None], [(1, 3), (1, 2)]]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> B = crystals.DirectSum([C,C], keepkey=True)
>>> [[b, b.e(Integer(2))] for b in B]
[[(0, 1), None], [(0, 2), None], [(0, 3), (0, 2)], [(1, 1), None], [(1,
→2), None], [(1, 3), (1, 2)]]
```

epsilon(*i*)

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: b = B( tuple([0,C(2)]) )
sage: b.epsilon(2)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> B = crystals.DirectSum([C,C], keepkey=True)
>>> b = B( tuple([Integer(0),C(Integer(2))]) )
>>> b.epsilon(Integer(2))
0
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: [[b,b.f(1)] for b in B]
[[(0, 1), (0, 2)], [(0, 2), None], [(0, 3), None], [(1, 1), (1, 2)], [(1,
→2), None], [(1, 3), None]]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> B = crystals.DirectSum([C,C], keepkey=True)
>>> [[b,b.f(Integer(1))] for b in B]
[[(0, 1), (0, 2)], [(0, 2), None], [(0, 3), None], [(1, 1), (1, 2)], [(1,
→2), None], [(1, 3), None]]
```

phi(*i*)

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: b = B( tuple([0,C(2)]) )
sage: b.phi(2)
1
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> B = crystals.DirectSum([C,C], keepkey=True)
>>> b = B( tuple([Integer(0),C(Integer(2))]) )
>>> b.phi(Integer(2))
1
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: b = B( tuple([0,C(2)]) )
sage: b
(0, 2)
sage: b.weight()
(0, 1, 0)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> B = crystals.DirectSum([C,C], keepkey=True)
>>> b = B( tuple([Integer(0),C(Integer(2))]) )
>>> b
(0, 2)
>>> b.weight()
(0, 1, 0)
```

weight_lattice_realization()

Return the weight lattice realization used to express weights.

The weight lattice realization is the common parent which all weight lattice realizations of the crystals of `self` coerce into.

EXAMPLES:

```
sage: LaZ = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: LaQ = RootSystem(['A',2,1]).weight_space(extended=True).fundamental_
↪weights()
sage: B = crystals.LSPaths(LaQ[1])
sage: B.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['A',
↪ 2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: C = crystals.AlcovePaths(LaZ[1])
sage: C.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
sage: D = crystals.DirectSum([B,C])
sage: D.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['A',
↪ 2, 1]
```

```
>>> from sage.all import *
>>> LaZ = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> LaQ = RootSystem(['A',Integer(2),Integer(1)]).weight_space(extended=True).
↪fundamental_weights()
>>> B = crystals.LSPaths(LaQ[Integer(1)])
>>> B.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['A',
↪ 2, 1]
>>> C = crystals.AlcovePaths(LaZ[Integer(1)])
>>> C.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
>>> D = crystals.DirectSum([B,C])
>>> D.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['A',
↪ 2, 1]
```

5.1.47 Elementary Crystals

Let λ be a weight. The crystals T_λ, R_λ, B_i, and C are important objects in the tensor category of crystals. For example, the crystal T_0 is the neutral object in this category; i.e., $T_0 \otimes B \cong B \otimes T_0 \cong B$ for any crystal B. We list some other properties of these crystals:

- The crystal $T_\lambda \otimes B(\infty)$ is the crystal of the Verma module with highest weight λ, where λ is a dominant integral weight.
- Let u_∞ be the highest weight vector of $B(\infty)$ and λ be a dominant integral weight. There is an embedding of crystals $B(\lambda) \longrightarrow T_\lambda \otimes B(\infty)$ sending $u_\lambda \mapsto t_\lambda \otimes u_\infty$ which is not strict, but the embedding $B(\lambda) \longrightarrow C \otimes T_\lambda \otimes B(\infty)$ by $u_\lambda \mapsto c \otimes t_\lambda \otimes u_\infty$ is a strict embedding.
- For any dominant integral weight λ, there is a surjective crystal morphism $\Psi_\lambda \colon R_\lambda \otimes B(\infty) \longrightarrow B(\lambda)$. More precisely, if $B = \{r_\lambda \otimes b \in R_\lambda \otimes B(\infty) : \Psi_\lambda(r_\lambda \otimes b) \neq 0\}$, then $B \cong B(\lambda)$ as crystals.
- For all Cartan types and all weights λ, we have $R_\lambda \cong C \otimes T_\lambda$ as crystals.
- For each i, there is a strict crystal morphism $\Psi_i \colon B(\infty) \longrightarrow B_i \otimes B(\infty)$ defined by $u_\infty \mapsto b_i(0) \otimes u_\infty$, where u_∞ is the highest weight vector of $B(\infty)$.

For more information on $B(\infty)$, see `InfinityCrystalOfTableaux`.

Note

As with `TensorProductOfCrystals`, we are using the opposite of Kashiwara's convention.

AUTHORS:

- Ben Salisbury: Initial version

REFERENCES:

- [Ka1993]
- [NZ1997]

class sage.combinat.crystals.elementary_crystals.**AbstractSingleCrystalElement**

Bases: Element

Abstract base class for elements in crystals with a single element.

e(i)

Return e_i of self, which is None for all i.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: ct = CartanType(['A',2])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[1])
sage: t = T.highest_weight_vector()
sage: t.e(1)
sage: t.e(2)
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(2)])
>>> la = RootSystem(ct).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(ct,la[Integer(1)])
>>> t = T.highest_weight_vector()
>>> t.e(Integer(1))
>>> t.e(Integer(2))
```

f(i)

Return f_i of self, which is None for all i.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: ct = CartanType(['A',2])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[1])
sage: t = T.highest_weight_vector()
sage: t.f(1)
sage: t.f(2)
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(2)])
>>> la = RootSystem(ct).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(ct,la[Integer(1)])
>>> t = T.highest_weight_vector()
>>> t.f(Integer(1))
>>> t.f(Integer(2))
```

class sage.combinat.crystals.elementary_crystals.**ComponentCrystal**(*cartan_type*, *P*)

Bases: UniqueRepresentation, Parent

The component crystal.

Defined in [Ka1993], the component crystal $C = \{c\}$ is the single element crystal whose crystal structure is defined by

$$\mathrm{wt}(c) = 0, \quad e_i c = f_i c = 0, \quad \varepsilon_i(c) = \varphi_i(c) = 0.$$

Note $C \cong B(0)$, where $B(0)$ is the highest weight crystal of highest weight 0.

INPUT:

- cartan_type – a Cartan type

class Element

Bases: *AbstractSingleCrystalElement*

Element of a component crystal.

epsilon(*i*)

Return ε_i of self, which is 0 for all i.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: C = crystals.elementary.Component("C5")
sage: c = C.highest_weight_vector()
sage: [c.epsilon(i) for i in C.index_set()]
[0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> C = crystals.elementary.Component("C5")
>>> c = C.highest_weight_vector()
>>> [c.epsilon(i) for i in C.index_set()]
[0, 0, 0, 0, 0]
```

phi(*i*)

Return φ_i of self, which is 0 for all i.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: C = crystals.elementary.Component("C5")
sage: c = C.highest_weight_vector()
sage: [c.phi(i) for i in C.index_set()]
[0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> C = crystals.elementary.Component("C5")
>>> c = C.highest_weight_vector()
>>> [c.phi(i) for i in C.index_set()]
[0, 0, 0, 0, 0]
```

weight()

Return the weight of `self`, which is always 0.

EXAMPLES:

```
sage: C = crystals.elementary.Component("F4")
sage: c = C.highest_weight_vector()
sage: c.weight()
(0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> C = crystals.elementary.Component("F4")
>>> c = C.highest_weight_vector()
>>> c.weight()
(0, 0, 0, 0)
```

cardinality()

Return the cardinality of `self`, which is always 1.

EXAMPLES:

```
sage: C = crystals.elementary.Component("E6")
sage: c = C.highest_weight_vector()
sage: C.cardinality()
1
```

```
>>> from sage.all import *
>>> C = crystals.elementary.Component("E6")
>>> c = C.highest_weight_vector()
>>> C.cardinality()
1
```

weight_lattice_realization()

Return the weight lattice realization of `self`.

EXAMPLES:

```
sage: C = crystals.elementary.Component("A2")
sage: C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 2]

sage: P = RootSystem(['A',2]).weight_lattice()
sage: C = crystals.elementary.Component(P)
sage: C.weight_lattice_realization() is P
True
```

```
>>> from sage.all import *
>>> C = crystals.elementary.Component("A2")
>>> C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 2]

>>> P = RootSystem(['A',Integer(2)]).weight_lattice()
>>> C = crystals.elementary.Component(P)
```

(continues on next page)

(continued from previous page)

```
>>> C.weight_lattice_realization() is P
True
```

class sage.combinat.crystals.elementary_crystals.**ElementaryCrystal**(*cartan_type*, *i*)

Bases: UniqueRepresentation, Parent

The elementary crystal B_i.

For i an element of the index set of type X, the crystal B_i of type X is the set

$$B_i = \{b_i(m) : m \in \mathbf{Z}\},$$

where the crystal structure is given by

$$\begin{aligned} \mathrm{wt}\big(b_i(m)\big) &= m\alpha_i \\ \varphi_j\big(b_i(m)\big) &= \begin{cases} m & \text{if } j = i, \\ -\infty & \text{if } j \neq i, \end{cases} \\ \varepsilon_j\big(b_i(m)\big) &= \begin{cases} -m & \text{if } j = i, \\ -\infty & \text{if } j \neq i, \end{cases} \\ e_j b_i(m) &= \begin{cases} b_i(m+1) & \text{if } j = i, \\ 0 & \text{if } j \neq i, \end{cases} \\ f_j b_i(m) &= \begin{cases} b_i(m-1) & \text{if } j = i, \\ 0 & \text{if } j \neq i. \end{cases} \end{aligned}$$

The *Kashiwara embedding theorem* asserts there is a unique strict crystal embedding of crystals

$$B(\infty) \hookrightarrow B_i \otimes B(\infty),$$

satisfying certain properties (see [Ka1993]). The above embedding may be iterated to obtain a new embedding

$$B(\infty) \hookrightarrow B_{i_N} \otimes B_{i_{N-1}} \otimes \cdots \otimes B_{i_2} \otimes B_{i_1} \otimes B(\infty),$$

which is a foundational object in the study of *polyhedral realizations of crystals* (see, for example, [NZ1997]).

class Element(*parent*, *m*)

Bases: Element

Element of a B_i crystal.

e(*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.elementary.Elementary(['E',7],1)
sage: B(3).e(1)
4
sage: B(172).e_string([1]*171)
343
sage: B(0).e(2)
```

```
>>> from sage.all import *
>>> B = crystals.elementary.Elementary(['E',Integer(7)],Integer(1))
>>> B(Integer(3)).e(Integer(1))
4
>>> B(Integer(172)).e_string([Integer(1)]*Integer(171))
343
>>> B(Integer(0)).e(Integer(2))
```

epsilon(*i*)

Return ε_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.elementary.Elementary(['F',4],3)
sage: [[B(j).epsilon(i) for i in B.index_set()] for j in range(5)]
[[-inf, -inf, 0, -inf],
 [-inf, -inf, -1, -inf],
 [-inf, -inf, -2, -inf],
 [-inf, -inf, -3, -inf],
 [-inf, -inf, -4, -inf]]
```

```
>>> from sage.all import *
>>> B = crystals.elementary.Elementary(['F',Integer(4)],Integer(3))
>>> [[B(j).epsilon(i) for i in B.index_set()] for j in range(Integer(5))]
[[-inf, -inf, 0, -inf],
 [-inf, -inf, -1, -inf],
 [-inf, -inf, -2, -inf],
 [-inf, -inf, -3, -inf],
 [-inf, -inf, -4, -inf]]
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.elementary.Elementary(['E',7],1)
sage: B(3).f(1)
2
sage: B(172).f_string([1]*171)
1
sage: B(0).e(2)
```

```
>>> from sage.all import *
>>> B = crystals.elementary.Elementary(['E',Integer(7)],Integer(1))
>>> B(Integer(3)).f(Integer(1))
2
>>> B(Integer(172)).f_string([Integer(1)]*Integer(171))
1
>>> B(Integer(0)).e(Integer(2))
```

phi (i)

Return φ_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.elementary.Elementary(['E',8,1],4)
sage: [[B(m).phi(j) for j in B.index_set()] for m in range(44,49)]
[[-inf, -inf, -inf, -inf, 44, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 45, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 46, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 47, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 48, -inf, -inf, -inf, -inf]]
```

```
>>> from sage.all import *
>>> B = crystals.elementary.Elementary(['E',Integer(8),Integer(1)],
↪Integer(4))
>>> [[B(m).phi(j) for j in B.index_set()] for m in range(Integer(44),
↪Integer(49))]
[[-inf, -inf, -inf, -inf, 44, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 45, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 46, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 47, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 48, -inf, -inf, -inf, -inf]]
```

weight ()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.elementary.Elementary(['C',14],12)
sage: B(-385).weight()
-385*alpha[12]
```

```
>>> from sage.all import *
>>> B = crystals.elementary.Elementary(['C',Integer(14)],Integer(12))
>>> B(-Integer(385)).weight()
-385*alpha[12]
```

weight_lattice_realization ()

Return a realization of the lattice containing the weights of `self`.

EXAMPLES:

```
sage: B = crystals.elementary.Elementary(['A',4, 1], 2)
sage: B.weight_lattice_realization()
Root lattice of the Root system of type ['A', 4, 1]
```

```
>>> from sage.all import *
>>> B = crystals.elementary.Elementary(['A',Integer(4), Integer(1)],␣
↪Integer(2))
>>> B.weight_lattice_realization()
Root lattice of the Root system of type ['A', 4, 1]
```

class sage.combinat.crystals.elementary_crystals.**RCrystal**(*cartan_type*, *weight*, *dual*)

Bases: UniqueRepresentation, Parent

The crystal R_λ.

For a fixed weight λ, the crystal $R_\lambda = \{r_\lambda\}$ is a single element crystal with the crystal structure defined by

$$\mathrm{wt}(r_\lambda) = \lambda, \quad e_i r_\lambda = f_i r_\lambda = 0, \quad \varepsilon_i(r_\lambda) = -\langle h_i, \lambda \rangle, \quad \varphi_i(r_\lambda) = 0,$$

where $\{h_i\}$ are the simple coroots.

Tensoring R_λ with a crystal B results in shifting the weights of the vertices in B by λ and may also cut a subset out of the original graph of B. That is, $\mathrm{wt}(r_\lambda \otimes b) = \mathrm{wt}(b) + \lambda$, where $b \in B$, provided $r_\lambda \otimes b \neq 0$. For example, the crystal graph of $B(\lambda)$ is the same as the crystal graph of $R_\lambda \otimes B(\infty)$ generated from the component $r_\lambda \otimes u_\infty$.

There is also a dual version of this crystal given by $R^\vee_\lambda = \{r^\vee_\lambda\}$ with the crystal structure defined by

$$\mathrm{wt}(r^\vee_\lambda) = \lambda, \quad e_i r^\vee_\lambda = f_i r^\vee_\lambda = 0, \quad \varepsilon_i(r^\vee_\lambda) = 0, \quad \varphi_i(r^\vee_\lambda) = \langle h_i, \lambda \rangle.$$

INPUT:

- `cartan_type` – a Cartan type
- `weight` – an element of the weight lattice of type `cartan_type`
- `dual` – boolean (default: `False`)

EXAMPLES:

We check by tensoring R_λ with $B(\infty)$ results in a component of $B(\lambda)$:

```
sage: B = crystals.infinity.Tableaux("A2")
sage: R = crystals.elementary.R("A2", B.Lambda()[1]+B.Lambda()[2])
sage: T = crystals.TensorProduct(R, B)
sage: mg = T(R.highest_weight_vector(), B.highest_weight_vector())
sage: S = T.subcrystal(generators=[mg])
sage: sorted([x.weight() for x in S], key=str)
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 1, 1),
 (1, 1, 1), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: sorted([x.weight() for x in C], key=str)
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 1, 1),
 (1, 1, 1), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
sage: GT = T.digraph(subset=S)
sage: GC = C.digraph()
sage: GT.is_isomorphic(GC, edge_labels=True)
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("A2")
>>> R = crystals.elementary.R("A2", B.Lambda()[Integer(1)]+B.Lambda()[Integer(2)])
>>> T = crystals.TensorProduct(R, B)
>>> mg = T(R.highest_weight_vector(), B.highest_weight_vector())
>>> S = T.subcrystal(generators=[mg])
>>> sorted([x.weight() for x in S], key=str)
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 1, 1),
 (1, 1, 1), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
>>> C = crystals.Tableaux("A2", shape=[Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> sorted([x.weight() for x in C], key=str)
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 1, 1),
 (1, 1, 1), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
>>> GT = T.digraph(subset=S)
>>> GC = C.digraph()
>>> GT.is_isomorphic(GC, edge_labels=True)
True
```

class Element

Bases: *AbstractSingleCrystalElement*

Element of a R_λ crystal.

epsilon(*i*)

Return ε_i of `self`.

We have $\varepsilon_i(r_\lambda) = -\langle h_i, \lambda \rangle$ for all i, where h_i is a simple coroot.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: la = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R("A2", la[1])
sage: r = R.highest_weight_vector()
sage: [r.epsilon(i) for i in R.index_set()]
[-1, 0]

sage: R = crystals.elementary.R("A2", la[1], dual=True)
sage: r = R.highest_weight_vector()
sage: [r.epsilon(i) for i in R.index_set()]
[0, 0]
```

```
>>> from sage.all import *
>>> la = RootSystem(['A',Integer(2)]).weight_lattice().fundamental_
↪weights()
>>> R = crystals.elementary.R("A2", la[Integer(1)])
>>> r = R.highest_weight_vector()
>>> [r.epsilon(i) for i in R.index_set()]
[-1, 0]

>>> R = crystals.elementary.R("A2", la[Integer(1)], dual=True)
>>> r = R.highest_weight_vector()
>>> [r.epsilon(i) for i in R.index_set()]
[0, 0]
```

phi(*i*)

Return φ_i of `self`, which is 0 for all i.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: la = RootSystem("C5").weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R("C5", la[4]+la[5])
sage: r = R.highest_weight_vector()
sage: [r.phi(i) for i in R.index_set()]
[0, 0, 0, 0, 0]

sage: R = crystals.elementary.R("C5", la[4]+la[5], dual=True)
sage: r = R.highest_weight_vector()
sage: [r.phi(i) for i in R.index_set()]
[0, 0, 0, 1, 1]
```

```
>>> from sage.all import *
>>> la = RootSystem("C5").weight_lattice().fundamental_weights()
>>> R = crystals.elementary.R("C5", la[Integer(4)]+la[Integer(5)])
>>> r = R.highest_weight_vector()
>>> [r.phi(i) for i in R.index_set()]
[0, 0, 0, 0, 0]

>>> R = crystals.elementary.R("C5", la[Integer(4)]+la[Integer(5)],␣
↪dual=True)
>>> r = R.highest_weight_vector()
>>> [r.phi(i) for i in R.index_set()]
[0, 0, 0, 1, 1]
```

weight()

Return the weight of `self`, which is always λ.

EXAMPLES:

```
sage: ct = CartanType(['C',5])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[4]+la[5]-la[1]-la[2])
sage: t = T.highest_weight_vector()
sage: t.weight()
-Lambda[1] - Lambda[2] + Lambda[4] + Lambda[5]
```

```
>>> from sage.all import *
>>> ct = CartanType(['C',Integer(5)])
>>> la = RootSystem(ct).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(ct,la[Integer(4)]+la[Integer(5)]-
↪la[Integer(1)]-la[Integer(2)])
>>> t = T.highest_weight_vector()
>>> t.weight()
-Lambda[1] - Lambda[2] + Lambda[4] + Lambda[5]
```

cardinality()

Return the cardinality of `self`, which is always 1.

EXAMPLES:

```
sage: La = RootSystem(['C',12]).weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R(['C',12],La[9])
```

(continues on next page)

(continued from previous page)

```
sage: R.cardinality()
1
```

```
>>> from sage.all import *
>>> La = RootSystem(['C',Integer(12)]).weight_lattice().fundamental_weights()
>>> R = crystals.elementary.R(['C',Integer(12)],La[Integer(9)])
>>> R.cardinality()
1
```

weight_lattice_realization()

Return a realization of the lattice containing the weights of `self`.

EXAMPLES:

```
sage: La = RootSystem(['C',12]).weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R(['C',12], La[9])
sage: R.weight_lattice_realization()
Weight lattice of the Root system of type ['C', 12]

sage: ct = CartanMatrix([[2, -4], [-5, 2]])
sage: La = RootSystem(ct).weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R(ct, La[1])
sage: R.weight_lattice_realization()
Weight lattice of the Root system of type
[ 2 -4]
[-5  2]
```

```
>>> from sage.all import *
>>> La = RootSystem(['C',Integer(12)]).weight_lattice().fundamental_weights()
>>> R = crystals.elementary.R(['C',Integer(12)], La[Integer(9)])
>>> R.weight_lattice_realization()
Weight lattice of the Root system of type ['C', 12]

>>> ct = CartanMatrix([[Integer(2), -Integer(4)], [-Integer(5), Integer(2)]])
>>> La = RootSystem(ct).weight_lattice().fundamental_weights()
>>> R = crystals.elementary.R(ct, La[Integer(1)])
>>> R.weight_lattice_realization()
Weight lattice of the Root system of type
[ 2 -4]
[-5  2]
```

class sage.combinat.crystals.elementary_crystals.**TCrystal**(*cartan_type*, *weight*)

Bases: `UniqueRepresentation`, `Parent`

The crystal T_λ.

Let λ be a weight. As defined in [Ka1993] the crystal $T_\lambda = \{t_\lambda\}$ is a single element crystal with the crystal structure defined by

$$\mathrm{wt}(t_\lambda) = \lambda, \quad e_i t_\lambda = f_i t_\lambda = 0, \quad \varepsilon_i(t_\lambda) = \varphi_i(t_\lambda) = -\infty.$$

The crystal T_λ shifts the weights of the vertices in a crystal B by λ when tensored with B, but leaves the graph structure of B unchanged. That is to say, for all $b \in B$, we have $\mathrm{wt}(b \otimes t_\lambda) = \mathrm{wt}(b) + \lambda$.

INPUT:

- `cartan_type` – a Cartan type
- `weight` – an element of the weight lattice of type `cartan_type`

EXAMPLES:

```
sage: ct = CartanType(['A',2])
sage: C = crystals.Tableaux(ct, shape=[1])
sage: for x in C: x.weight()
(1, 0, 0)
(0, 1, 0)
(0, 0, 1)
sage: La = RootSystem(ct).ambient_space().fundamental_weights()
sage: TLa = crystals.elementary.T(ct, 3*(La[1] + La[2]))
sage: TP = crystals.TensorProduct(TLa, C)
sage: for x in TP: x.weight()
(7, 3, 0)
(6, 4, 0)
(6, 3, 1)
sage: G = C.digraph()
sage: H = TP.digraph()
sage: G.is_isomorphic(H,edge_labels=True)
True
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(2)])
>>> C = crystals.Tableaux(ct, shape=[Integer(1)])
>>> for x in C: x.weight()
(1, 0, 0)
(0, 1, 0)
(0, 0, 1)
>>> La = RootSystem(ct).ambient_space().fundamental_weights()
>>> TLa = crystals.elementary.T(ct, Integer(3)*(La[Integer(1)] + La[Integer(2)]))
>>> TP = crystals.TensorProduct(TLa, C)
>>> for x in TP: x.weight()
(7, 3, 0)
(6, 4, 0)
(6, 3, 1)
>>> G = C.digraph()
>>> H = TP.digraph()
>>> G.is_isomorphic(H,edge_labels=True)
True
```

class Element

Bases: *AbstractSingleCrystalElement*

Element of a T_λ crystal.

epsilon(i)

Return ε_i of `self`, which is $-\infty$ for all i.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: ct = CartanType(['C',5])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[4]+la[5]-la[1]-la[2])
sage: t = T.highest_weight_vector()
sage: [t.epsilon(i) for i in T.index_set()]
[-inf, -inf, -inf, -inf, -inf]
```

```
>>> from sage.all import *
>>> ct = CartanType(['C',Integer(5)])
>>> la = RootSystem(ct).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(ct,la[Integer(4)]+la[Integer(5)]-
↪la[Integer(1)]-la[Integer(2)])
>>> t = T.highest_weight_vector()
>>> [t.epsilon(i) for i in T.index_set()]
[-inf, -inf, -inf, -inf, -inf]
```

phi (i)

Return φ_i of `self`, which is $-\infty$ for all i.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: ct = CartanType(['C',5])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[4]+la[5]-la[1]-la[2])
sage: t = T.highest_weight_vector()
sage: [t.phi(i) for i in T.index_set()]
[-inf, -inf, -inf, -inf, -inf]
```

```
>>> from sage.all import *
>>> ct = CartanType(['C',Integer(5)])
>>> la = RootSystem(ct).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(ct,la[Integer(4)]+la[Integer(5)]-
↪la[Integer(1)]-la[Integer(2)])
>>> t = T.highest_weight_vector()
>>> [t.phi(i) for i in T.index_set()]
[-inf, -inf, -inf, -inf, -inf]
```

weight ()

Return the weight of `self`, which is always λ.

EXAMPLES:

```
sage: ct = CartanType(['C',5])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[4]+la[5]-la[1]-la[2])
sage: t = T.highest_weight_vector()
sage: t.weight()
-Lambda[1] - Lambda[2] + Lambda[4] + Lambda[5]
```

```
>>> from sage.all import *
>>> ct = CartanType(['C',Integer(5)])
```

(continues on next page)

(continued from previous page)

```
>>> la = RootSystem(ct).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(ct,la[Integer(4)]+la[Integer(5)]-
↪la[Integer(1)]-la[Integer(2)])
>>> t = T.highest_weight_vector()
>>> t.weight()
-Lambda[1] - Lambda[2] + Lambda[4] + Lambda[5]
```

cardinality()

Return the cardinality of `self`, which is always 1.

EXAMPLES:

```
sage: La = RootSystem(['C',12]).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(['C',12], La[9])
sage: T.cardinality()
1
```

```
>>> from sage.all import *
>>> La = RootSystem(['C',Integer(12)]).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(['C',Integer(12)], La[Integer(9)])
>>> T.cardinality()
1
```

weight_lattice_realization()

Return a realization of the lattice containing the weights of `self`.

EXAMPLES:

```
sage: La = RootSystem(['C',12]).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(['C',12], La[9])
sage: T.weight_lattice_realization()
Weight lattice of the Root system of type ['C', 12]

sage: ct = CartanMatrix([[2, -4], [-5, 2]])
sage: La = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct, La[1])
sage: T.weight_lattice_realization()
Weight lattice of the Root system of type
[ 2 -4]
[-5  2]
```

```
>>> from sage.all import *
>>> La = RootSystem(['C',Integer(12)]).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(['C',Integer(12)], La[Integer(9)])
>>> T.weight_lattice_realization()
Weight lattice of the Root system of type ['C', 12]

>>> ct = CartanMatrix([[Integer(2), -Integer(4)], [-Integer(5), Integer(2)]])
>>> La = RootSystem(ct).weight_lattice().fundamental_weights()
>>> T = crystals.elementary.T(ct, La[Integer(1)])
>>> T.weight_lattice_realization()
Weight lattice of the Root system of type
```

(continues on next page)

(continued from previous page)

```
[ 2 -4]
[-5  2]
```

5.1.48 Fast Rank Two Crystals

class sage.combinat.crystals.fast_crystals.**FastCrystal**(*ct*, *shape*, *format*)

Bases: UniqueRepresentation, Parent

An alternative implementation of rank 2 crystals. The root operators are implemented in memory by table lookup. This means that in comparison with the CrystalsOfTableaux, these crystals are slow to instantiate but faster for computation. Implemented for types A_2, B_2, and C_2.

INPUT:

- cartan_type – the Cartan type and must be either type A_2, B_2, or C_2
- shape – a shape is of the form [l1,l2] where l1 and l2 are either integers or (in type B_2) half integers such that l1 - l2 is integral. It is assumed that l1 >= l2 >= 0. If l1 and l2 are integers, this will produce a crystal isomorphic to the one obtained by crystals.Tableaux(type, shape=[l1,l2]). Furthermore crystals.FastRankTwo(['B', 2], l1+1/2, l2+1/2) produces a crystal isomorphic to the following crystal T:

```
sage: C = crystals.Tableaux(['B',2], shape=[l1,l2])                    # not tested
sage: D = crystals.Spins(['B',2])                                      # not tested
sage: T = crystals.TensorProduct(C, D, C.list()[0], D.list()[0])   # not tested
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['B',Integer(2)], shape=[l1,l2])                  #␣
↪not tested
>>> D = crystals.Spins(['B',Integer(2)])                                    #␣
↪not tested
>>> T = crystals.TensorProduct(C, D, C.list()[Integer(0)], D.
↪list()[Integer(0)])  # not tested
```

- format – (default: 'string') the default representation of elements is in term of theBerenstein-Zelevinsky-Littelmann (BZL) strings [a1, a2, ...] described under metapost in crystals. Alternative representations may be obtained by the options 'dual_string' or 'simple'. In the 'simple' format, the element is represented by and integer, and in the 'dual_string' format, it is represented by the BZL string, but the underlying decomposition of the long Weyl group element into simple reflections is changed.

class Element(*parent*, *value*, *format*)

Bases: Element

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: c = C(0); c
[0, 0, 0]
sage: C[0].parent()
The fast crystal for A2 with shape [2,1]
sage: TestSuite(c).run()
```

```
>>> from sage.all import *
>>> C = crystals.FastRankTwo(['A',Integer(2)],shape=[Integer(2),Integer(1)])
>>> c = C(Integer(0)); c
[0, 0, 0]
>>> C[Integer(0)].parent()
The fast crystal for A2 with shape [2,1]
>>> TestSuite(c).run()
```

e(i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: C(1).e(1)
[0, 0, 0]
sage: C(0).e(1) is None
True
```

```
>>> from sage.all import *
>>> C = crystals.FastRankTwo(['A',Integer(2)],shape=[Integer(2),
↪Integer(1)])
>>> C(Integer(1)).e(Integer(1))
[0, 0, 0]
>>> C(Integer(0)).e(Integer(1)) is None
True
```

f(i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: C(6).f(1)
[1, 2, 1]
sage: C(7).f(1) is None
True
```

```
>>> from sage.all import *
>>> C = crystals.FastRankTwo(['A',Integer(2)],shape=[Integer(2),
↪Integer(1)])
>>> C(Integer(6)).f(Integer(1))
[1, 2, 1]
>>> C(Integer(7)).f(Integer(1)) is None
True
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.FastRankTwo(['A',2], shape=[2,1])]
[(2, 1, 0), (1, 2, 0), (1, 1, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (1, 1,␣
```

(continues on next page)

(continued from previous page)

```
↪1), (0, 2, 1)]
sage: [v.weight() for v in crystals.FastRankTwo(['B',2], shape=[1,0])]
[(1, 0), (0, 1), (0, 0), (0, -1), (-1, 0)]
sage: [v.weight() for v in crystals.FastRankTwo(['B',2], shape=[1/2,1/2])]
[(1/2, 1/2), (1/2, -1/2), (-1/2, 1/2), (-1/2, -1/2)]
sage: [v.weight() for v in crystals.FastRankTwo(['C',2], shape=[1,0])]
[(1, 0), (0, 1), (0, -1), (-1, 0)]
sage: [v.weight() for v in crystals.FastRankTwo(['C',2], shape=[1,1])]
[(1, 1), (1, -1), (0, 0), (-1, 1), (-1, -1)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.FastRankTwo(['A',Integer(2)],␣
↪shape=[Integer(2),Integer(1)])]
[(2, 1, 0), (1, 2, 0), (1, 1, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (1, 1,␣
↪1), (0, 2, 1)]
>>> [v.weight() for v in crystals.FastRankTwo(['B',Integer(2)],␣
↪shape=[Integer(1),Integer(0)])]
[(1, 0), (0, 1), (0, 0), (0, -1), (-1, 0)]
>>> [v.weight() for v in crystals.FastRankTwo(['B',Integer(2)],␣
↪shape=[Integer(1)/Integer(2),Integer(1)/Integer(2)])]
[(1/2, 1/2), (1/2, -1/2), (-1/2, 1/2), (-1/2, -1/2)]
>>> [v.weight() for v in crystals.FastRankTwo(['C',Integer(2)],␣
↪shape=[Integer(1),Integer(0)])]
[(1, 0), (0, 1), (0, -1), (-1, 0)]
>>> [v.weight() for v in crystals.FastRankTwo(['C',Integer(2)],␣
↪shape=[Integer(1),Integer(1)])]
[(1, 1), (1, -1), (0, 0), (-1, 1), (-1, -1)]
```

cmp_elements(x, y)

Return `True` if and only if there is a path from x to y in the crystal graph.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: x = C(0)
sage: y = C(1)
sage: C.cmp_elements(x,y)
-1
sage: C.cmp_elements(y,x)
1
sage: C.cmp_elements(x,x)
0
```

```
>>> from sage.all import *
>>> C = crystals.FastRankTwo(['A',Integer(2)],shape=[Integer(2),Integer(1)])
>>> x = C(Integer(0))
>>> y = C(Integer(1))
>>> C.cmp_elements(x,y)
-1
```

(continues on next page)

(continued from previous page)

```
>>> C.cmp_elements(y,x)
1
>>> C.cmp_elements(x,x)
0
```

digraph()

Return the digraph associated to `self`.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: C.digraph()
Digraph on 8 vertices
```

```
>>> from sage.all import *
>>> C = crystals.FastRankTwo(['A',Integer(2)],shape=[Integer(2),Integer(1)])
>>> C.digraph()
Digraph on 8 vertices
```

5.1.49 Fully commutative stable Grothendieck crystal

AUTHORS:

- Jianping Pan (2020-08-31): initial version
- Wencin Poh (2020-08-31): initial version
- Anne Schilling (2020-08-31): initial version

class sage.combinat.crystals.fully_commutative_stable_grothendieck.**DecreasingHeckeFactorization**(*parent*, *t*)

Bases: `Element`

Class of decreasing factorizations in the 0-Hecke monoid.

INPUT:

- `t` – decreasing factorization inputted as list of lists
- `max_value` – maximal value of entries

EXAMPLES:

```
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
sage: t = [[3, 2], [], [2, 1]]
sage: h = DecreasingHeckeFactorization(t, 3); h
(3, 2)()(2, 1)
sage: h.excess
1
sage: h.factors
3
sage: h.max_value
3
sage: h.value
```

(continues on next page)

(continued from previous page)

```
((3, 2), (), (2, 1))

sage: u = [[3, 2, 1], [3], [2, 1]]
sage: h = DecreasingHeckeFactorization(u); h
(3, 2, 1)(3)(2, 1)
sage: h.weight()
(2, 1, 3)
sage: h.parent()
Decreasing Hecke factorizations with 3 factors associated to [2, 1, 3, 2, 1] with␣
↪excess 1
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
>>> t = [[Integer(3), Integer(2)], [], [Integer(2), Integer(1)]]
>>> h = DecreasingHeckeFactorization(t, Integer(3)); h
(3, 2)()(2, 1)
>>> h.excess
1
>>> h.factors
3
>>> h.max_value
3
>>> h.value
((3, 2), (), (2, 1))

>>> u = [[Integer(3), Integer(2), Integer(1)], [Integer(3)], [Integer(2),␣
↪Integer(1)]]
>>> h = DecreasingHeckeFactorization(u); h
(3, 2, 1)(3)(2, 1)
>>> h.weight()
(2, 1, 3)
>>> h.parent()
Decreasing Hecke factorizations with 3 factors associated to [2, 1, 3, 2, 1] with␣
↪excess 1
```

to_increasing_hecke_biword()

Return the associated increasing Hecke biword of `self`.

EXAMPLES:

```
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck␣
↪import DecreasingHeckeFactorization
sage: t = [[2], [], [2, 1],[4, 3, 1]]
sage: h = DecreasingHeckeFactorization(t, 4)
sage: h.to_increasing_hecke_biword()
[[1, 1, 1, 2, 2, 4], [1, 3, 4, 1, 2, 2]]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
>>> t = [[Integer(2)], [], [Integer(2), Integer(1)],[Integer(4), Integer(3),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)]]
>>> h = DecreasingHeckeFactorization(t, Integer(4))
>>> h.to_increasing_hecke_biword()
[[1, 1, 1, 2, 2, 4], [1, 3, 4, 1, 2, 2]]
```

to_word()

Return the word associated to `self` in the 0-Hecke monoid.

EXAMPLES:

```
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck␣
↪import DecreasingHeckeFactorization
sage: t = [[2], [], [2, 1], [4, 3, 1]]
sage: h = DecreasingHeckeFactorization(t)
sage: h.to_word()
[2, 2, 1, 4, 3, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
>>> t = [[Integer(2)], [], [Integer(2), Integer(1)], [Integer(4), Integer(3),␣
↪Integer(1)]]
>>> h = DecreasingHeckeFactorization(t)
>>> h.to_word()
[2, 2, 1, 4, 3, 1]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck␣
↪import DecreasingHeckeFactorization
sage: t = [[2], [2, 1], [], [4, 3, 1]]
sage: h = DecreasingHeckeFactorization(t, 6)
sage: h.weight()
(3, 0, 2, 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
>>> t = [[Integer(2)], [Integer(2), Integer(1)], [], [Integer(4), Integer(3),␣
↪Integer(1)]]
>>> h = DecreasingHeckeFactorization(t, Integer(6))
>>> h.weight()
(3, 0, 2, 1)
```

class `sage.combinat.crystals.fully_commutative_stable_grothendieck.`**DecreasingHeckeFactorizations**(*w*, *factors*, *excess*)

Bases: `UniqueRepresentation`, `Parent`

Set of decreasing factorizations in the 0-Hecke monoid.

INPUT:

- `w` – an element in the symmetric group
- `factors` – the number of factors in the factorization
- `excess` – the total number of letters in the factorization minus the length of a reduced word for `w`

EXAMPLES:

```
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorizations
sage: S = SymmetricGroup(3+1)
sage: w = S.from_reduced_word([1, 3, 2, 1])
sage: F = DecreasingHeckeFactorizations(w, 3, 3); F
Decreasing Hecke factorizations with 3 factors associated to [1, 3, 2, 1] with␣
↪excess 3
sage: F.list()
[(3, 1)(3, 1)(3, 2, 1), (3, 1)(3, 2, 1)(2, 1), (3, 2, 1)(2, 1)(2, 1)]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorizations
>>> S = SymmetricGroup(Integer(3)+Integer(1))
>>> w = S.from_reduced_word([Integer(1), Integer(3), Integer(2), Integer(1)])
>>> F = DecreasingHeckeFactorizations(w, Integer(3), Integer(3)); F
Decreasing Hecke factorizations with 3 factors associated to [1, 3, 2, 1] with␣
↪excess 3
>>> F.list()
[(3, 1)(3, 1)(3, 2, 1), (3, 1)(3, 2, 1)(2, 1), (3, 2, 1)(2, 1)(2, 1)]
```

Element

alias of *DecreasingHeckeFactorization*

list()

Return list of all elements of `self`.

EXAMPLES:

```
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck␣
↪import DecreasingHeckeFactorizations
sage: S = SymmetricGroup(3+1)
sage: w = S.from_reduced_word([1, 3, 2, 1])
sage: F = DecreasingHeckeFactorizations(w, 3, 3)
sage: F.list()
[(3, 1)(3, 1)(3, 2, 1), (3, 1)(3, 2, 1)(2, 1), (3, 2, 1)(2, 1)(2, 1)]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorizations
>>> S = SymmetricGroup(Integer(3)+Integer(1))
>>> w = S.from_reduced_word([Integer(1), Integer(3), Integer(2), Integer(1)])
>>> F = DecreasingHeckeFactorizations(w, Integer(3), Integer(3))
>>> F.list()
[(3, 1)(3, 1)(3, 2, 1), (3, 1)(3, 2, 1)(2, 1), (3, 2, 1)(2, 1)(2, 1)]
```

class sage.combinat.crystals.fully_commutative_stable_grothendieck.**FullyCommutativeStableGrothendiec**

Bases: UniqueRepresentation, Parent

The crystal on fully commutative decreasing factorizations in the 0-Hecke monoid, as introduced by [MPPS2020].

INPUT:

- w – an element in the symmetric group or a (skew) shape
- factors – the number of factors in the factorization
- excess – the total number of letters in the factorization minus the length of a reduced word for w
- shape – boolean (default: False); indicator for input w, True if w is entered as a (skew) shape and False otherwise

EXAMPLES:

```
sage: S = SymmetricGroup(3+1)
sage: w = S.from_reduced_word([1, 3, 2])
sage: B = crystals.FullyCommutativeStableGrothendieck(w, 3, 2); B
Fully commutative stable Grothendieck crystal of type A_2 associated to [1, 3, 2]
↪with excess 2
sage: B.list()
[(1)(3, 1)(3, 2),
 (3, 1)(1)(3, 2),
 (3, 1)(3, 1)(2),
 (3)(3, 1)(3, 2),
 (3, 1)(3)(3, 2),
 (3, 1)(3, 2)(2)]
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(3)+Integer(1))
>>> w = S.from_reduced_word([Integer(1), Integer(3), Integer(2)])
>>> B = crystals.FullyCommutativeStableGrothendieck(w, Integer(3), Integer(2)); B
Fully commutative stable Grothendieck crystal of type A_2 associated to [1, 3, 2]
↪with excess 2
>>> B.list()
[(1)(3, 1)(3, 2),
 (3, 1)(1)(3, 2),
 (3, 1)(3, 1)(2),
 (3)(3, 1)(3, 2),
 (3, 1)(3)(3, 2),
 (3, 1)(3, 2)(2)]
```

We can also access the crystal by specifying a skew shape:

```
sage: crystals.FullyCommutativeStableGrothendieck([[2, 2], [1]], 4, 1, shape=True)
Fully commutative stable Grothendieck crystal of type A_3 associated to [2, 1, 3]
↪with excess 1
```

```
>>> from sage.all import *
>>> crystals.FullyCommutativeStableGrothendieck([[Integer(2), Integer(2)],
→[Integer(1)]], Integer(4), Integer(1), shape=True)
Fully commutative stable Grothendieck crystal of type A_3 associated to [2, 1, 3]
→with excess 1
```

We can compute the highest weight elements:

```
sage: hw = [w for w in B if w.is_highest_weight()]
sage: hw
[(1)(3, 1)(3, 2), (3)(3, 1)(3, 2)]
sage: hw[0].weight()
(2, 2, 1)
```

```
>>> from sage.all import *
>>> hw = [w for w in B if w.is_highest_weight()]
>>> hw
[(1)(3, 1)(3, 2), (3)(3, 1)(3, 2)]
>>> hw[Integer(0)].weight()
(2, 2, 1)
```

The crystal operators themselves move elements between adjacent factors:

```
sage: b = hw[0]; b
(1)(3, 1)(3, 2)
sage: b.f(2)
(3, 1)(1)(3, 2)
```

```
>>> from sage.all import *
>>> b = hw[Integer(0)]; b
(1)(3, 1)(3, 2)
>>> b.f(Integer(2))
(3, 1)(1)(3, 2)
```

class Element(*parent*, *t*)

Bases: *DecreasingHeckeFactorization*

Create an instance self of element t.

This method takes into account the constraints on the word, the number of factors, and excess statistic associated to the parent class.

EXAMPLES:

```
sage: S = SymmetricGroup(3+1)
sage: w = S.from_reduced_word([1, 3, 2])
sage: B = crystals.FullyCommutativeStableGrothendieck(w, 3, 2)
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck
→import DecreasingHeckeFactorization
sage: h = DecreasingHeckeFactorization([[3, 1], [3], [3, 2]], 4)
sage: u = B(h); u.value
((3, 1), (3,), (3, 2))
sage: v = B([[3, 1], [3], [3, 2]]); v.value
((3, 1), (3,), (3, 2))
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(3)+Integer(1))
>>> w = S.from_reduced_word([Integer(1), Integer(3), Integer(2)])
>>> B = crystals.FullyCommutativeStableGrothendieck(w, Integer(3), Integer(2))
>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
>>> h = DecreasingHeckeFactorization([[Integer(3), Integer(1)], [Integer(3)],␣
↪[Integer(3), Integer(2)]], Integer(4))
>>> u = B(h); u.value
((3, 1), (3,), (3, 2))
>>> v = B([[Integer(3), Integer(1)], [Integer(3)], [Integer(3), Integer(2)]]);
↪ v.value
((3, 1), (3,), (3, 2))
```

bracketing(*i*)

Remove all bracketed letters between i-th and $(i+1)$-th entry.

EXAMPLES:

```
sage: S = SymmetricGroup(4+1)
sage: w = S.from_reduced_word([3, 2, 1, 4, 3])
sage: B = crystals.FullyCommutativeStableGrothendieck(w, 3, 2)
sage: h = B([[3], [4, 2, 1], [4, 3, 1]])
sage: h.bracketing(1)
[[], []]
sage: h.bracketing(2)
[[], [2, 1]]
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(4)+Integer(1))
>>> w = S.from_reduced_word([Integer(3), Integer(2), Integer(1),␣
↪Integer(4), Integer(3)])
>>> B = crystals.FullyCommutativeStableGrothendieck(w, Integer(3),␣
↪Integer(2))
>>> h = B([[Integer(3)], [Integer(4), Integer(2), Integer(1)],␣
↪[Integer(4), Integer(3), Integer(1)]])
>>> h.bracketing(Integer(1))
[[], []]
>>> h.bracketing(Integer(2))
[[], [2, 1]]
```

e(*i*)

Return the action of e_i on `self` using the rules described in [MPPS2020].

EXAMPLES:

```
sage: S = SymmetricGroup(4+1)
sage: w = S.from_reduced_word([2, 1, 4, 3, 2])
sage: B = crystals.FullyCommutativeStableGrothendieck(w, 4, 3)
sage: h = B([[4, 2], [4, 2, 1], [3, 2], [2]]); h
(4, 2)(4, 2, 1)(3, 2)(2)
sage: h.e(1)
(4, 2)(4, 2, 1)(3)(3, 2)
```

(continues on next page)

(continued from previous page)

```
sage: h.e(2)
(4, 2)(2, 1)(4, 3, 2)(2)
sage: h.e(3)
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(4)+Integer(1))
>>> w = S.from_reduced_word([Integer(2), Integer(1), Integer(4),␣
↪Integer(3), Integer(2)])
>>> B = crystals.FullyCommutativeStableGrothendieck(w, Integer(4),␣
↪Integer(3))
>>> h = B([[Integer(4), Integer(2)], [Integer(4), Integer(2), Integer(1)],
↪ [Integer(3), Integer(2)], [Integer(2)]]); h
(4, 2)(4, 2, 1)(3, 2)(2)
>>> h.e(Integer(1))
(4, 2)(4, 2, 1)(3)(3, 2)
>>> h.e(Integer(2))
(4, 2)(2, 1)(4, 3, 2)(2)
>>> h.e(Integer(3))
```

f(*i*)

Return the action of f_i on `self` using the rules described in [MPPS2020].

EXAMPLES:

```
sage: S = SymmetricGroup(4+1)
sage: w = S.from_reduced_word([3, 2, 1, 4, 3])
sage: B = crystals.FullyCommutativeStableGrothendieck(w, 4, 3)
sage: h = B([[3, 2], [2, 1], [4, 3], [3, 1]]); h
(3, 2)(2, 1)(4, 3)(3, 1)
sage: h.f(1)
(3, 2)(2, 1)(4, 3, 1)(3)
sage: h.f(2)
sage: h.f(3)
(3, 2, 1)(1)(4, 3)(3, 1)
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(4)+Integer(1))
>>> w = S.from_reduced_word([Integer(3), Integer(2), Integer(1),␣
↪Integer(4), Integer(3)])
>>> B = crystals.FullyCommutativeStableGrothendieck(w, Integer(4),␣
↪Integer(3))
>>> h = B([[Integer(3), Integer(2)], [Integer(2), Integer(1)],␣
↪[Integer(4), Integer(3)], [Integer(3), Integer(1)]]); h
(3, 2)(2, 1)(4, 3)(3, 1)
>>> h.f(Integer(1))
(3, 2)(2, 1)(4, 3, 1)(3)
>>> h.f(Integer(2))
>>> h.f(Integer(3))
(3, 2, 1)(1)(4, 3)(3, 1)
```

module_generators()

Return generators for `self` as a crystal.

EXAMPLES:

```
sage: S = SymmetricGroup(3+1)
sage: w = S.from_reduced_word([1, 3, 2])
sage: B = crystals.FullyCommutativeStableGrothendieck(w, 3, 2)
sage: B.module_generators
((1)(3, 1)(3, 2), (3)(3, 1)(3, 2))
sage: C = crystals.FullyCommutativeStableGrothendieck(w, 4, 2)
sage: C.module_generators
(()(1)(3, 1)(3, 2),
 ()(3)(3, 1)(3, 2),
 (1)(1)(1)(3, 2),
 (1)(1)(3)(3, 2),
 (1)(3)(3)(3, 2))
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(3)+Integer(1))
>>> w = S.from_reduced_word([Integer(1), Integer(3), Integer(2)])
>>> B = crystals.FullyCommutativeStableGrothendieck(w, Integer(3), Integer(2))
>>> B.module_generators
((1)(3, 1)(3, 2), (3)(3, 1)(3, 2))
>>> C = crystals.FullyCommutativeStableGrothendieck(w, Integer(4), Integer(2))
>>> C.module_generators
(()(1)(3, 1)(3, 2),
 ()(3)(3, 1)(3, 2),
 (1)(1)(1)(3, 2),
 (1)(1)(3)(3, 2),
 (1)(3)(3)(3, 2))
```

5.1.50 Crystals of Generalized Young Walls

AUTHORS:

- Lucas David-Roesler: Initial version
- Ben Salisbury: Initial version
- Travis Scrimshaw: Initial version

Generalized Young walls are certain generalizations of Young tableaux introduced in [KS2010] and designed to be a realization of the crystals $\mathcal{B}(\infty)$ and $\mathcal{B}(\lambda)$ in type $A_n^{(1)}$.

REFERENCES:

- [KLRS2016]
- [KS2010]

class sage.combinat.crystals.generalized_young_walls.**CrystalOfGeneralizedYoungWalls**(*n*, *La*)

Bases: *InfinityCrystalOfGeneralizedYoungWalls*

The crystal $\mathcal{Y}(\lambda)$ of generalized Young walls of the given type with highest weight λ.

These were characterized in Theorem 4.1 of [KS2010]. See *GeneralizedYoungWall.in_highest_weight_crystal()*.

INPUT:

- n – type $A_n^{(1)}$
- weight – dominant integral weight

EXAMPLES:

```
sage: La = RootSystem(['A',3,1]).weight_lattice(extended=True).fundamental_
↪weights()[1]
sage: YLa = crystals.GeneralizedYoungWalls(3,La)
sage: y = YLa([[0],[1,0,3,2,1],[2,1,0],[3]])
sage: y.pp()
        3|
    0|1|2|
1|2|3|0|1|
        0|
sage: y.weight()
-Lambda[0] + Lambda[2] + Lambda[3] - 3*delta
sage: y.in_highest_weight_crystal(La)
True
sage: y.f(1)
[[0], [1, 0, 3, 2, 1], [2, 1, 0], [3], [], [1]]
sage: y.f(1).f(1)
sage: yy = crystals.infinity.GeneralizedYoungWalls(3)([[0], [1, 0, 3, 2, 1], [2,␣
↪1, 0], [3], [], [1]])
sage: yy.f(1)
[[0], [1, 0, 3, 2, 1], [2, 1, 0], [3], [], [1], [], [], [], [1]]
sage: yyy = yy.f(1)
sage: yyy.in_highest_weight_crystal(La)
False

sage: LS = crystals.LSPaths(['A',3,1],[1,0,0,0])
sage: C = LS.subcrystal(max_depth=4)
sage: G = LS.digraph(subset=C)
sage: P = RootSystem(['A',3,1]).weight_lattice(extended=True)
sage: La = P.fundamental_weights()
sage: YW = crystals.GeneralizedYoungWalls(3,La[0])
sage: CW = YW.subcrystal(max_depth=4)
sage: GW = YW.digraph(subset=CW)
sage: GW.is_isomorphic(G,edge_labels=True)
True
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(3),Integer(1)]).weight_lattice(extended=True).
↪fundamental_weights()[Integer(1)]
>>> YLa = crystals.GeneralizedYoungWalls(Integer(3),La)
>>> y = YLa([[Integer(0)],[Integer(1),Integer(0),Integer(3),Integer(2),
↪Integer(1)],[Integer(2),Integer(1),Integer(0)],[Integer(3)]])
>>> y.pp()
        3|
    0|1|2|
1|2|3|0|1|
        0|
>>> y.weight()
-Lambda[0] + Lambda[2] + Lambda[3] - 3*delta
```

(continues on next page)

(continued from previous page)

```
>>> y.in_highest_weight_crystal(La)
True
>>> y.f(Integer(1))
[[0], [1, 0, 3, 2, 1], [2, 1, 0], [3], [], [1]]
>>> y.f(Integer(1)).f(Integer(1))
>>> yy = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[Integer(0)],␣
↪[Integer(1), Integer(0), Integer(3), Integer(2), Integer(1)], [Integer(2),␣
↪Integer(1), Integer(0)], [Integer(3)], [], [Integer(1)]])
>>> yy.f(Integer(1))
[[0], [1, 0, 3, 2, 1], [2, 1, 0], [3], [], [1], [], [], [], [1]]
>>> yyy = yy.f(Integer(1))
>>> yyy.in_highest_weight_crystal(La)
False

>>> LS = crystals.LSPaths(['A',Integer(3),Integer(1)],[Integer(1),Integer(0),
↪Integer(0),Integer(0)])
>>> C = LS.subcrystal(max_depth=Integer(4))
>>> G = LS.digraph(subset=C)
>>> P = RootSystem(['A',Integer(3),Integer(1)]).weight_lattice(extended=True)
>>> La = P.fundamental_weights()
>>> YW = crystals.GeneralizedYoungWalls(Integer(3),La[Integer(0)])
>>> CW = YW.subcrystal(max_depth=Integer(4))
>>> GW = YW.digraph(subset=CW)
>>> GW.is_isomorphic(G,edge_labels=True)
True
```

To display the crystal down to a specified depth:

```
sage: S = YLa.subcrystal(max_depth=4)
sage: G = YLa.digraph(subset=S)
sage: view(G) # not tested
```

```
>>> from sage.all import *
>>> S = YLa.subcrystal(max_depth=Integer(4))
>>> G = YLa.digraph(subset=S)
>>> view(G) # not tested
```

Element

alias of *CrystalOfGeneralizedYoungWallsElement*

class sage.combinat.crystals.generalized_young_walls.**CrystalOfGeneralizedYoungWallsElement**(*parent*, *data*)

Bases: *GeneralizedYoungWall*

Element of the highest weight crystal of generalized Young walls.

e(*i*)

Compute the action of e_i restricted to the highest weight crystal.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()[1]
sage: hwy = crystals.GeneralizedYoungWalls(2,La)([[],[1,0],[2,1]])
sage: hwy.e(1)
[[], [1, 0], [2]]
sage: hwy.e(2)
sage: hwy.e(3)
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()[Integer(1)]
>>> hwy = crystals.GeneralizedYoungWalls(Integer(2),La)([[],[Integer(1),
↪Integer(0)],[Integer(2),Integer(1)]])
>>> hwy.e(Integer(1))
[[], [1, 0], [2]]
>>> hwy.e(Integer(2))
>>> hwy.e(Integer(3))
```

f(*i*)

Compute the action of f_i restricted to the highest weight crystal.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()[1]
sage: GYW = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = GYW([[],[1,0],[2,1]])
sage: y.f(1)
[[], [1, 0], [2, 1], [], [1]]
sage: hwy = crystals.GeneralizedYoungWalls(2,La)([[],[1,0],[2,1]])
sage: hwy.f(1)
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()[Integer(1)]
>>> GYW = crystals.infinity.GeneralizedYoungWalls(Integer(2))
>>> y = GYW([[],[Integer(1),Integer(0)],[Integer(2),Integer(1)]])
>>> y.f(Integer(1))
[[], [1, 0], [2, 1], [], [1]]
>>> hwy = crystals.GeneralizedYoungWalls(Integer(2),La)([[],[Integer(1),
↪Integer(0)],[Integer(2),Integer(1)]])
>>> hwy.f(Integer(1))
```

phi(*i*)

Return the value $\varepsilon_i(Y) + \langle h_i, \mathrm{wt}(Y)\rangle$, where h_i is the i-th simple coroot and Y is `self`.

EXAMPLES:

```
sage: La = RootSystem(['A',3,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: y = crystals.GeneralizedYoungWalls(3,La[0])([])
sage: y.phi(1)
0
```

(continues on next page)

(continued from previous page)

```
sage: y.phi(2)
0
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(3),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> y = crystals.GeneralizedYoungWalls(Integer(3),La[Integer(0)])([])
>>> y.phi(Integer(1))
0
>>> y.phi(Integer(2))
0
```

weight()

Return the weight of `self` in the highest weight crystal as an element of the weight lattice $\bigoplus_{i=0}^{n} \mathbf{Z}\Lambda_i$.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()[1]
sage: hwy = crystals.GeneralizedYoungWalls(2,La)([[],[1,0],[2,1]])
sage: hwy.weight()
Lambda[0] - Lambda[1] + Lambda[2] - delta
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()[Integer(1)]
>>> hwy = crystals.GeneralizedYoungWalls(Integer(2),La)([[],[Integer(1),
↪Integer(0)],[Integer(2),Integer(1)]])
>>> hwy.weight()
Lambda[0] - Lambda[1] + Lambda[2] - delta
```

class sage.combinat.crystals.generalized_young_walls.**GeneralizedYoungWall**(*parent*, *data*)

Bases: *CombinatorialElement*

A generalized Young wall.

For more information, see *InfinityCrystalOfGeneralizedYoungWalls*.

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(4)
sage: mg = Y.module_generators[0]; mg.pp()
0
sage: mg.f_string([1,2,0,1]).pp()
1|2|
0|1|
   |
```

```
>>> from sage.all import *
>>> Y = crystals.infinity.GeneralizedYoungWalls(Integer(4))
>>> mg = Y.module_generators[Integer(0)]; mg.pp()
0
>>> mg.f_string([Integer(1),Integer(2),Integer(0),Integer(1)]).pp()
```

(continues on next page)

(continued from previous page)

```
1|2|
0|1|
   |
```

Epsilon()

Return $\sum_{i=0}^{n} \varepsilon_i(Y)\Lambda_i$ where Y is `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,
↪2,1,0,3,2],[0],[],[2]])
sage: y.Epsilon()
Lambda[0] + 3*Lambda[2]
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[Integer(0)],
↪[Integer(1),Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],
↪[Integer(3),Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],
↪[Integer(0)],[],[Integer(2)]])
>>> y.Epsilon()
Lambda[0] + 3*Lambda[2]
```

Phi()

Return $\sum_{i=0}^{n} \varphi_i(Y)\Lambda_i$ where Y is `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,
↪2,1,0,3,2],[0],[],[2]])
sage: y.Phi()
-Lambda[0] + 3*Lambda[1] - Lambda[2] + 3*Lambda[3]

sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,
↪1,0,3,2],[],[],[2]])
sage: x.Phi()
2*Lambda[0] + Lambda[1] - Lambda[2] + Lambda[3]
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[Integer(0)],
↪[Integer(1),Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],
↪[Integer(3),Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],
↪[Integer(0)],[],[Integer(2)]])
>>> y.Phi()
-Lambda[0] + 3*Lambda[1] - Lambda[2] + 3*Lambda[3]

>>> x = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[],[Integer(1),
↪Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],[],[],[Integer(2)]])
>>> x.Phi()
2*Lambda[0] + Lambda[1] - Lambda[2] + Lambda[3]
```

a(i, k)

Return the number $a_i(k)$ of i-colored boxes in the `k`-th column of `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,
→2,1,0,3,2],[0],[],[2]])
sage: y.a(1,2)
1
sage: y.a(0,2)
1
sage: y.a(3,2)
0
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[Integer(0)],
→[Integer(1),Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],
→[Integer(3),Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],
→[Integer(0)],[],[Integer(2)]])
>>> y.a(Integer(1),Integer(2))
1
>>> y.a(Integer(0),Integer(2))
1
>>> y.a(Integer(3),Integer(2))
0
```

column(*k*)

Return the list of boxes from the `k`-th column of `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,
→2,1,0,3,2],[0],[],[2]])
sage: y.column(2)
[None, 0, 1, 2, None, None, None]

sage: hw = crystals.infinity.GeneralizedYoungWalls(5)([])
sage: hw.column(1)
[]
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[Integer(0)],
→[Integer(1),Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],
→[Integer(3),Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],
→[Integer(0)],[],[Integer(2)]])
>>> y.column(Integer(2))
[None, 0, 1, 2, None, None, None]

>>> hw = crystals.infinity.GeneralizedYoungWalls(Integer(5))([])
>>> hw.column(Integer(1))
[]
```

content()

Return total number of blocks in `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(2)([[0],[1,0],[2,1,0,2],[],
↪[1]])
sage: y.content()
8

sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,
↪1,0,3,2],[],[],[2]])
sage: x.content()
13
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(2))([[Integer(0)],
↪[Integer(1),Integer(0)],[Integer(2),Integer(1),Integer(0),Integer(2)],[],
↪[Integer(1)]])
>>> y.content()
8

>>> x = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[],[Integer(1),
↪Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],[],[],[Integer(2)]])
>>> x.content()
13
```

e(*i*)

Return the application of the Kashiwara raising operator e_i on `self`.

This will remove the i-colored box corresponding to the rightmost $+$ in `self.signature(i)`.

EXAMPLES:

```
sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,
↪1,0,3,2],[],[],[2]])
sage: x.e(2)
[[], [1, 0, 3, 2], [2, 1], [3, 2, 1, 0, 3, 2]]
sage: _.e(2)
[[], [1, 0, 3], [2, 1], [3, 2, 1, 0, 3, 2]]
sage: _.e(2)
[[], [1, 0, 3], [2, 1], [3, 2, 1, 0, 3]]
sage: _.e(2)
```

```
>>> from sage.all import *
>>> x = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[],[Integer(1),
↪Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],[],[],[Integer(2)]])
>>> x.e(Integer(2))
[[], [1, 0, 3, 2], [2, 1], [3, 2, 1, 0, 3, 2]]
>>> _.e(Integer(2))
[[], [1, 0, 3], [2, 1], [3, 2, 1, 0, 3, 2]]
>>> _.e(Integer(2))
[[], [1, 0, 3], [2, 1], [3, 2, 1, 0, 3]]
>>> _.e(Integer(2))
```

epsilon(*i*)

Return the number of i-colored arrows in the i-string above `self` in the crystal graph.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,
↪1,0,3,2],[],[],[2]])
sage: y.epsilon(1)
0
sage: y.epsilon(2)
3
sage: y.epsilon(0)
0
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[],[Integer(1),
↪Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],[],[],[Integer(2)]])
>>> y.epsilon(Integer(1))
0
>>> y.epsilon(Integer(2))
3
>>> y.epsilon(Integer(0))
0
```

f(i)

Return the application of the Kashiwara lowering operator f_i on `self`.

This will add an i-colored colored box to the site corresponding to the leftmost plus in `self.signature(i)`.

EXAMPLES:

```
sage: hw = crystals.infinity.GeneralizedYoungWalls(2)([])
sage: hw.f(1)
[[], [1]]
sage: _.f(2)
[[], [1], [2]]
sage: _.f(0)
[[], [1, 0], [2]]
sage: _.f(0)
[[0], [1, 0], [2]]
```

```
>>> from sage.all import *
>>> hw = crystals.infinity.GeneralizedYoungWalls(Integer(2))([])
>>> hw.f(Integer(1))
[[], [1]]
>>> _.f(Integer(2))
[[], [1], [2]]
>>> _.f(Integer(0))
[[], [1, 0], [2]]
>>> _.f(Integer(0))
[[0], [1, 0], [2]]
```

generate_signature(i)

The i-signature of `self` (with whitespace where cancellation occurs) together with the unreduced sequence from $\{+, -\}$. The result also records to the row and column position of the sign.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(2)([[0],[1,0],[2,1,0,2],[],
↪[1]])
sage: y.generate_signature(1)
([['+', 2, 5], ['-', 4, 1]], '  ')
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(2))([[Integer(0)],
↪[Integer(1),Integer(0)],[Integer(2),Integer(1),Integer(0),Integer(2)],[],
↪[Integer(1)]])
>>> y.generate_signature(Integer(1))
([['+', 2, 5], ['-', 4, 1]], '  ')
```

in_highest_weight_crystal(*La*)

Return a boolean indicating if the generalized Young wall element is in the highest weight crystal cut out by the given highest weight La.

By Theorem 4.1 of [KS2010], a generalized Young wall Y represents a vertex in the highest weight crystal $Y(\lambda)$, with $\lambda = \Lambda_{i_1} + \Lambda_{i_2} + \cdots + \Lambda_{i_\ell}$ a dominant integral weight of level $\ell > 0$, if it satisfies the following condition. For each positive integer k, if there exists $j \in I$ such that $a_j(k) - a_{j-1}(k) > 0$, then for some $p = 1, \ldots, \ell$,

$$j + k \equiv i_p + 1 \bmod n + 1 \text{ and } a_j(k) - a_{j-1}(k) \le \lambda(h_{i_p}),$$

where $\{h_0, h_1, \ldots, h_n\}$ is the set of simple coroots attached to $A_n^{(1)}$.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()[1]
sage: GYW = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = GYW([[],[1,0],[2,1]])
sage: y.in_highest_weight_crystal(La)
True
sage: x = GYW([[],[1],[2],[],[],[2],[],[],[2]])
sage: x.in_highest_weight_crystal(La)
False
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()[Integer(1)]
>>> GYW = crystals.infinity.GeneralizedYoungWalls(Integer(2))
>>> y = GYW([[],[Integer(1),Integer(0)],[Integer(2),Integer(1)]])
>>> y.in_highest_weight_crystal(La)
True
>>> x = GYW([[],[Integer(1)],[Integer(2)],[],[],[Integer(2)],[],[],
↪[Integer(2)]])
>>> x.in_highest_weight_crystal(La)
False
```

latex_large()

Generate LaTeX code for self but the output is larger.

This requires TikZ.

EXAMPLES:

```
sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,
↪1,0,3,2],[],[],[2]])
sage: x.latex_large()
'\\begin{tikzpicture}[baseline=5,scale=.45] \n \\foreach \\x [count=\\s from␣
↪0] in \n{{},{1,0,3,2},{2,1},{3,2,1,0,3,2},{},{},{2}} \n{\\foreach \\y␣
↪[count=\\t from 0] in \\x {  \\node[font=\\scriptsize] at (-\\t,\\s) {$\\y$}
↪; \n \\draw (-\\t+.5,\\s+.5) to (-\\t-.5,\\s+.5); \n \\draw (-\\t+.5,\\s-.
↪5) to (-\\t-.5,\\s-.5); \n \\draw (-\\t-.5,\\s-.5) to (-\\t-.5,\\s+.5);  } \
↪n \\draw[-,thick] (.5,\\s+1) to (.5,-.5) to (-\\s-1,-.5); } \n \\end
↪{tikzpicture} \n'
```

```
>>> from sage.all import *
>>> x = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[],[Integer(1),
↪Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],[],[],[Integer(2)]])
>>> x.latex_large()
'\\begin{tikzpicture}[baseline=5,scale=.45] \n \\foreach \\x [count=\\s from␣
↪0] in \n{{},{1,0,3,2},{2,1},{3,2,1,0,3,2},{},{},{2}} \n{\\foreach \\y␣
↪[count=\\t from 0] in \\x {  \\node[font=\\scriptsize] at (-\\t,\\s) {$\\y$}
↪; \n \\draw (-\\t+.5,\\s+.5) to (-\\t-.5,\\s+.5); \n \\draw (-\\t+.5,\\s-.
↪5) to (-\\t-.5,\\s-.5); \n \\draw (-\\t-.5,\\s-.5) to (-\\t-.5,\\s+.5);  } \
↪n \\draw[-,thick] (.5,\\s+1) to (.5,-.5) to (-\\s-1,-.5); } \n \\end
↪{tikzpicture} \n'
```

number_of_parts()

Return the value of $\mathscr{N}$ on `self`.

In [KLRS2016], the statistic $\mathscr{N}$ was defined on elements in $\mathcal{Y}(\infty)$ which counts how many parts are in the corresponding Kostant partition. Specifically, the computation of $\mathscr{N}(Y)$ is done using the following algorithm:

- If Y has no rows whose right-most box is colored n and such that the length of this row is a multiple of $n+1$, then $\mathscr{N}(Y)$ is the total number of distinct rows in Y, not counting multiplicity.
- Otherwise, search Y for the longest row such that the right-most box is colored n and such that the total number of boxes in the row is $k(n+1)$ for some $k \geq 1$. Replace this row by $n+1$ distinct rows of length k, reordering all rows, if necessary, so that the result is a proper wall. (Note that the resulting wall may no longer be reduced.) Repeat the search and replace process for all other rows of the above form for each $k' < k$. Then $\mathscr{N}(Y)$ is the number of distinct rows, not counting multiplicity, in the wall resulting from this process.

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(3)
sage: y = Y([[0],[],[],[],[0],[],[],[],[0]])
sage: y.number_of_parts()
1

sage: Y = crystals.infinity.GeneralizedYoungWalls(3)
sage: y = Y([[0,3,2],[1,0],[],[],[0,3],[1,0],[],[],[0]])
sage: y.number_of_parts()
4

sage: Y = crystals.infinity.GeneralizedYoungWalls(2)
```

(continues on next page)

(continued from previous page)

```
sage: y = Y([[0,2,1],[1,0],[2,1,0,2,1,0,2,1,0],[],[2,1,0,2,1,0]])
sage: y.number_of_parts()
8
```

```
>>> from sage.all import *
>>> Y = crystals.infinity.GeneralizedYoungWalls(Integer(3))
>>> y = Y([[Integer(0)],[],[],[],[Integer(0)],[],[],[],[Integer(0)]])
>>> y.number_of_parts()
1

>>> Y = crystals.infinity.GeneralizedYoungWalls(Integer(3))
>>> y = Y([[Integer(0),Integer(3),Integer(2)],[Integer(1),Integer(0)],[],[],
↪[Integer(0),Integer(3)],[Integer(1),Integer(0)],[],[],[Integer(0)]])
>>> y.number_of_parts()
4

>>> Y = crystals.infinity.GeneralizedYoungWalls(Integer(2))
>>> y = Y([[Integer(0),Integer(2),Integer(1)],[Integer(1),Integer(0)],
↪[Integer(2),Integer(1),Integer(0),Integer(2),Integer(1),Integer(0),
↪Integer(2),Integer(1),Integer(0)],[],[Integer(2),Integer(1),Integer(0),
↪Integer(2),Integer(1),Integer(0)]])
>>> y.number_of_parts()
8
```

phi(*i*)

Return the value $\varepsilon_i(Y) + \langle h_i, \mathrm{wt}(Y)\rangle$, where h_i is the i-th simple coroot and Y is `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,
↪2,1,0,3,2],[0],[],[2]])
sage: y.phi(1)
3
sage: y.phi(2)
-1
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[Integer(0)],
↪[Integer(1),Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],
↪[Integer(3),Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],
↪[Integer(0)],[],[Integer(2)]])
>>> y.phi(Integer(1))
3
>>> y.phi(Integer(2))
-1
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(2)([[0,2,1],[1,0,2,1,0],[],
↪[0],[1,0,2],[],[],[1]])
```

(continues on next page)

(continued from previous page)

```
sage: y.pp()
        1|
         |
         |
    2|0|1|
        0|
         |
0|1|2|0|1|
    1|2|0|
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(2))([[Integer(0),
↪Integer(2),Integer(1)],[Integer(1),Integer(0),Integer(2),Integer(1),
↪Integer(0)],[],[Integer(0)],[Integer(1),Integer(0),Integer(2)],[],[],
↪[Integer(1)]])
>>> y.pp()
        1|
         |
         |
    2|0|1|
        0|
         |
0|1|2|0|1|
    1|2|0|
```

raw_signature(*i*)

Return the sequence from $\{+, -\}$ obtained from all i-admissible slots and removable i-boxes without canceling any $(+, -)$-pairs. The result also notes the row and column of the sign.

EXAMPLES:

```
sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,
↪1,0,3,2],[],[],[2]])
sage: x.raw_signature(2)
[['-', 3, 6], ['-', 1, 4], ['-', 6, 1]]
```

```
>>> from sage.all import *
>>> x = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[],[Integer(1),
↪Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],[],[],[Integer(2)]])
>>> x.raw_signature(Integer(2))
[['-', 3, 6], ['-', 1, 4], ['-', 6, 1]]
```

signature(*i*)

Return the i-signature of `self`.

The signature is obtained by reading `self` in columns bottom to top starting from the left. Then add a $-$ at every i-box which may be removed from `self` and still obtain a legal generalized Young wall, and add a $+$ at each site for which an i-box may be added and still obtain a valid generalized Young wall. Then successively cancel any $(+, -)$-pair to obtain a sequence of the form $- \cdots - + \cdots +$. This resulting sequence is the output.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(2)([[0],[1,0],[2,1,0,2],[],
↪[1]])
sage: y.signature(1)
''

sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,
↪1,0,3,2],[],[],[2]])
sage: x.signature(2)
'---'
```

```
>>> from sage.all import *
>>> y = crystals.infinity.GeneralizedYoungWalls(Integer(2))([[Integer(0)],
↪[Integer(1),Integer(0)],[Integer(2),Integer(1),Integer(0),Integer(2)],[],
↪[Integer(1)]])
>>> y.signature(Integer(1))
''

>>> x = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[],[Integer(1),
↪Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],[],[],[Integer(2)]])
>>> x.signature(Integer(2))
'---'
```

sum_of_weighted_row_lengths()

Return the value of $\mathscr{M}$ on `self`.

Let $\mathcal{Y}_0 \subset \mathcal{Y}(\infty)$ be the set of generalized Young walls which have no rows whose right-most box is colored n. For $Y \in \mathcal{Y}_0$,

$$\mathscr{M}(Y) = \sum_{i=1}^{n} (i+1) M_i(Y),$$

where $M_i(Y)$ is the number of nonempty rows in Y whose right-most box is colored $i-1$.

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = Y([[0,2,1,0,2],[1,0,2],[],[0,2],[1,0],[],[0],[1,0]])
sage: y.sum_of_weighted_row_lengths()
15
```

```
>>> from sage.all import *
>>> Y = crystals.infinity.GeneralizedYoungWalls(Integer(2))
>>> y = Y([[Integer(0),Integer(2),Integer(1),Integer(0),Integer(2)],
↪[Integer(1),Integer(0),Integer(2)],[],[Integer(0),Integer(2)],[Integer(1),
↪Integer(0)],[],[Integer(0)],[Integer(1),Integer(0)]])
>>> y.sum_of_weighted_row_lengths()
15
```

weight(*root_lattice=False*)

Return the weight of `self`.

INPUT:

- `root_lattice` – boolean determining whether weight should appear in root lattice or not in extended affine weight lattice

EXAMPLES:

```
sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,
↪1,0,3,2],[],[],[2]])
sage: x.weight()
2*Lambda[0] + Lambda[1] - 4*Lambda[2] + Lambda[3] - 2*delta
sage: x.weight(root_lattice=True)
-2*alpha[0] - 3*alpha[1] - 5*alpha[2] - 3*alpha[3]
```

```
>>> from sage.all import *
>>> x = crystals.infinity.GeneralizedYoungWalls(Integer(3))([[],[Integer(1),
↪Integer(0),Integer(3),Integer(2)],[Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(1),Integer(0),Integer(3),Integer(2)],[],[],[Integer(2)]])
>>> x.weight()
2*Lambda[0] + Lambda[1] - 4*Lambda[2] + Lambda[3] - 2*delta
>>> x.weight(root_lattice=True)
-2*alpha[0] - 3*alpha[1] - 5*alpha[2] - 3*alpha[3]
```

class `sage.combinat.crystals.generalized_young_walls.`**`InfinityCrystalOfGeneralizedYoungWalls`**`(`*n*, *category*`)`

Bases: `UniqueRepresentation`, `Parent`

The crystal $\mathcal{Y}(\infty)$ of generalized Young walls of type $A_n^{(1)}$ as defined in [KS2010].

A generalized Young wall is a collection of boxes stacked on a fixed board, such that color of the box at the site located in the j-th row from the bottom and the i-th column from the right is $j-1 \bmod n+1$. There are several growth conditions on elements in $Y \in \mathcal{Y}(\infty)$:

- Walls grow in rows from right to left. That is, for every box $y \in Y$ that is not in the rightmost column, there must be a box immediately to the right of y.
- For all $p > q$ such that $p - q \equiv 0 \bmod n+1$, the p-th row has most as many boxes as the q-th row.
- There does not exist a column in the wall such that if one i-colored box, for every $i = 0, 1, \ldots, n$, is removed from that column, then the result satisfies the above conditions.

There is a crystal structure on $\mathcal{Y}(\infty)$ defined as follows. Define maps

$$e_i,\ f_i \colon \mathcal{Y}(\infty) \longrightarrow \mathcal{Y}(\infty) \sqcup \{0\}, \qquad \varepsilon_i,\ \varphi_i \colon \mathcal{Y}(\infty) \longrightarrow \mathbf{Z}, \qquad \mathrm{wt}\colon \mathcal{Y}(\infty) \longrightarrow \bigoplus_{i=0}^{n} \mathbf{Z}\Lambda_i \oplus \mathbf{Z}\delta,$$

by

$$\mathrm{wt}(Y) = -\sum_{i=0}^{n} m_i(Y)\alpha_i,$$

where $m_i(Y)$ is the number of i-boxes in Y, $\varepsilon_i(Y)$ is the number of $-$ in the i-signature of Y, and

$$\varphi_i(Y) = \varepsilon_i(Y) + \langle h_i, \mathrm{wt}(Y)\rangle.$$

See *`GeneralizedYoungWall.e()`*, *`GeneralizedYoungWall.f()`*, and *`GeneralizedYoungWall.signature()`* for more about e_i, f_i, and i-signatures.

INPUT:

- n – type $A_n^{(1)}$

EXAMPLES:

```
sage: Yinf = crystals.infinity.GeneralizedYoungWalls(3)
sage: y = Yinf([[0],[1,0,3,2],[],[3,2,1],[0],[1,0]])
sage: y.pp()
    0|1|
      0|
  1|2|3|
        |
2|3|0|1|
      0|
sage: y.weight(root_lattice=True)
-4*alpha[0] - 3*alpha[1] - 2*alpha[2] - 2*alpha[3]
sage: y.f(0)
[[0], [1, 0, 3, 2], [], [3, 2, 1], [0], [1, 0], [], [], [0]]
sage: y.e(0).pp()
    0|1|
        |
  1|2|3|
        |
2|3|0|1|
      0|
```

```
>>> from sage.all import *
>>> Yinf = crystals.infinity.GeneralizedYoungWalls(Integer(3))
>>> y = Yinf([[Integer(0)],[Integer(1),Integer(0),Integer(3),Integer(2)],[],
↪[Integer(3),Integer(2),Integer(1)],[Integer(0)],[Integer(1),Integer(0)]])
>>> y.pp()
    0|1|
      0|
  1|2|3|
        |
2|3|0|1|
      0|
>>> y.weight(root_lattice=True)
-4*alpha[0] - 3*alpha[1] - 2*alpha[2] - 2*alpha[3]
>>> y.f(Integer(0))
[[0], [1, 0, 3, 2], [], [3, 2, 1], [0], [1, 0], [], [], [0]]
>>> y.e(Integer(0)).pp()
    0|1|
        |
  1|2|3|
        |
2|3|0|1|
      0|
```

To display the crystal down to depth 3:

```
sage: S = Yinf.subcrystal(max_depth=3)
sage: G = Yinf.digraph(subset=S) # long time
sage: view(G) # not tested
```

```
>>> from sage.all import *
>>> S = Yinf.subcrystal(max_depth=Integer(3))
>>> G = Yinf.digraph(subset=S) # long time
>>> view(G) # not tested
```

Element

alias of *GeneralizedYoungWall*

5.1.51 Highest weight crystals

class sage.combinat.crystals.highest_weight_crystals.**FiniteDimensionalHighestWeightCrystal_TypeE**(*dominant_weight*)

Bases: *TensorProductOfCrystals*

Commonalities for all finite dimensional type E highest weight crystals.

Subclasses should setup an attribute column_crystal in their `__init__` method before calling the `__init__` method of this class.

Element

alias of *TensorProductOfRegularCrystalsElement*

module_generator()

Yield the module generator (or highest weight element) of the classical crystal of given dominant weight in `self`.

EXAMPLES:

```
sage: C=CartanType(['E',6])
sage: La=C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[2])
sage: T.module_generator()
[[(2, -1), (1,)]]
sage: T = crystals.HighestWeight(0*La[2])
sage: T.module_generator()
[]

sage: C=CartanType(['E',7])
sage: La=C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: T.module_generator()
[[(-7, 1), (7,)]]
```

```
>>> from sage.all import *
>>> C=CartanType(['E',Integer(6)])
>>> La=C.root_system().weight_lattice().fundamental_weights()
>>> T = crystals.HighestWeight(La[Integer(2)])
>>> T.module_generator()
[[(2, -1), (1,)]]
>>> T = crystals.HighestWeight(Integer(0)*La[Integer(2)])
>>> T.module_generator()
[]
```

(continues on next page)

(continued from previous page)

```
>>> C=CartanType(['E',Integer(7)])
>>> La=C.root_system().weight_lattice().fundamental_weights()
>>> T = crystals.HighestWeight(La[Integer(1)])
>>> T.module_generator()
[[(-7, 1), (7,)]]
```

class sage.combinat.crystals.highest_weight_crystals.**FiniteDimensionalHighestWeightCrystal_TypeE6**(*dom i- nar*

Bases: *FiniteDimensionalHighestWeightCrystal_TypeE*

Class of finite dimensional highest weight crystals of type E_6.

EXAMPLES:

```
sage: C=CartanType(['E',6])
sage: La=C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[2]); T
Finite dimensional highest weight crystal of type ['E', 6] and highest weight
↪Lambda[2]
sage: B1 = T.column_crystal[1]; B1
The crystal of letters for type ['E', 6]
sage: B6 = T.column_crystal[6]; B6
The crystal of letters for type ['E', 6] (dual)
sage: t = T(B6([-1]),B1([-1,3])); t
[(-1,), (-1, 3)]
sage: [t.epsilon(i) for i in T.index_set()]
[2, 0, 0, 0, 0, 0]
sage: [t.phi(i) for i in T.index_set()]
[0, 0, 1, 0, 0, 0]
sage: TestSuite(t).run()
```

```
>>> from sage.all import *
>>> C=CartanType(['E',Integer(6)])
>>> La=C.root_system().weight_lattice().fundamental_weights()
>>> T = crystals.HighestWeight(La[Integer(2)]); T
Finite dimensional highest weight crystal of type ['E', 6] and highest weight
↪Lambda[2]
>>> B1 = T.column_crystal[Integer(1)]; B1
The crystal of letters for type ['E', 6]
>>> B6 = T.column_crystal[Integer(6)]; B6
The crystal of letters for type ['E', 6] (dual)
>>> t = T(B6([-Integer(1)]),B1([-Integer(1),Integer(3)])); t
[(-1,), (-1, 3)]
>>> [t.epsilon(i) for i in T.index_set()]
[2, 0, 0, 0, 0, 0]
>>> [t.phi(i) for i in T.index_set()]
[0, 0, 1, 0, 0, 0]
>>> TestSuite(t).run()
```

class sage.combinat.crystals.highest_weight_crystals.**FiniteDimensionalHighestWeightCrystal_TypeE7**(*dom i- nar*

Bases: *FiniteDimensionalHighestWeightCrystal_TypeE*

Class of finite dimensional highest weight crystals of type E_7.

EXAMPLES:

```
sage: C=CartanType(['E',7])
sage: La=C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: T.cardinality()
133
sage: B7 = T.column_crystal[7]; B7
The crystal of letters for type ['E', 7]
sage: t = T(B7([-5, 6]), B7([-2, 3])); t
[(-5, 6), (-2, 3)]
sage: [t.epsilon(i) for i in T.index_set()]
[0, 1, 0, 0, 1, 0, 0]
sage: [t.phi(i) for i in T.index_set()]
[0, 0, 1, 0, 0, 1, 0]
sage: TestSuite(t).run()
```

```
>>> from sage.all import *
>>> C=CartanType(['E',Integer(7)])
>>> La=C.root_system().weight_lattice().fundamental_weights()
>>> T = crystals.HighestWeight(La[Integer(1)])
>>> T.cardinality()
133
>>> B7 = T.column_crystal[Integer(7)]; B7
The crystal of letters for type ['E', 7]
>>> t = T(B7([-Integer(5), Integer(6)]), B7([-Integer(2), Integer(3)])); t
[(-5, 6), (-2, 3)]
>>> [t.epsilon(i) for i in T.index_set()]
[0, 1, 0, 0, 1, 0, 0]
>>> [t.phi(i) for i in T.index_set()]
[0, 0, 1, 0, 0, 1, 0]
>>> TestSuite(t).run()
```

`sage.combinat.crystals.highest_weight_crystals.`**`HighestWeightCrystal`**(*dominant_weight*, *model=None*)

Return the highest weight crystal of highest weight `dominant_weight` of the given `model`.

INPUT:

- `dominant_weight` – a dominant weight
- `model` – (optional) if not specified, then we have the following default models:
  - types A_n, B_n, C_n, D_n, G_2 - *tableaux*
  - types $E_{6,7}$ - *type E finite dimensional crystal*
  - all other types - *LS paths*

  otherwise can be one of the following:

  - `'Tableaux'` - *KN tableaux*
  - `'TypeE'` - *type E finite dimensional crystal*

– 'NakajimaMonomials' - *Nakajima monomials*
– 'LSPaths' - *LS paths*
– 'AlcovePaths' - *alcove paths*
– 'GeneralizedYoungWalls' - *generalized Young walls*
– 'RiggedConfigurations' - *rigged configurations*

EXAMPLES:

```
sage: La = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: wt = La[1] + La[2]
sage: crystals.HighestWeight(wt)
The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]]

sage: La = RootSystem(['C',2]).weight_lattice().fundamental_weights()
sage: wt = 5*La[1] + La[2]
sage: crystals.HighestWeight(wt)
The crystal of tableaux of type ['C', 2] and shape(s) [[6, 1]]

sage: La = RootSystem(['B',2]).weight_lattice().fundamental_weights()
sage: wt = La[1] + La[2]
sage: crystals.HighestWeight(wt)
The crystal of tableaux of type ['B', 2] and shape(s) [[3/2, 1/2]]
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2)]).weight_lattice().fundamental_weights()
>>> wt = La[Integer(1)] + La[Integer(2)]
>>> crystals.HighestWeight(wt)
The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]]

>>> La = RootSystem(['C',Integer(2)]).weight_lattice().fundamental_weights()
>>> wt = Integer(5)*La[Integer(1)] + La[Integer(2)]
>>> crystals.HighestWeight(wt)
The crystal of tableaux of type ['C', 2] and shape(s) [[6, 1]]

>>> La = RootSystem(['B',Integer(2)]).weight_lattice().fundamental_weights()
>>> wt = La[Integer(1)] + La[Integer(2)]
>>> crystals.HighestWeight(wt)
The crystal of tableaux of type ['B', 2] and shape(s) [[3/2, 1/2]]
```

Some type E examples:

```
sage: C = CartanType(['E',6])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: T.cardinality()
27
sage: T = crystals.HighestWeight(La[6])
sage: T.cardinality()
27
sage: T = crystals.HighestWeight(La[2])
sage: T.cardinality()
78
```

(continues on next page)

(continued from previous page)

```
sage: T = crystals.HighestWeight(La[4])
sage: T.cardinality()
2925
sage: T = crystals.HighestWeight(La[3])
sage: T.cardinality()
351
sage: T = crystals.HighestWeight(La[5])
sage: T.cardinality()
351

sage: C = CartanType(['E',7])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: T.cardinality()
133
sage: T = crystals.HighestWeight(La[2])
sage: T.cardinality()
912
sage: T = crystals.HighestWeight(La[3])
sage: T.cardinality()
8645
sage: T = crystals.HighestWeight(La[4])
sage: T.cardinality()
365750
sage: T = crystals.HighestWeight(La[5])
sage: T.cardinality()
27664
sage: T = crystals.HighestWeight(La[6])
sage: T.cardinality()
1539
sage: T = crystals.HighestWeight(La[7])
sage: T.cardinality()
56
```

```
>>> from sage.all import *
>>> C = CartanType(['E',Integer(6)])
>>> La = C.root_system().weight_lattice().fundamental_weights()
>>> T = crystals.HighestWeight(La[Integer(1)])
>>> T.cardinality()
27
>>> T = crystals.HighestWeight(La[Integer(6)])
>>> T.cardinality()
27
>>> T = crystals.HighestWeight(La[Integer(2)])
>>> T.cardinality()
78
>>> T = crystals.HighestWeight(La[Integer(4)])
>>> T.cardinality()
2925
>>> T = crystals.HighestWeight(La[Integer(3)])
>>> T.cardinality()
351
```

(continues on next page)

(continued from previous page)

```
>>> T = crystals.HighestWeight(La[Integer(5)])
>>> T.cardinality()
351

>>> C = CartanType(['E',Integer(7)])
>>> La = C.root_system().weight_lattice().fundamental_weights()
>>> T = crystals.HighestWeight(La[Integer(1)])
>>> T.cardinality()
133
>>> T = crystals.HighestWeight(La[Integer(2)])
>>> T.cardinality()
912
>>> T = crystals.HighestWeight(La[Integer(3)])
>>> T.cardinality()
8645
>>> T = crystals.HighestWeight(La[Integer(4)])
>>> T.cardinality()
365750
>>> T = crystals.HighestWeight(La[Integer(5)])
>>> T.cardinality()
27664
>>> T = crystals.HighestWeight(La[Integer(6)])
>>> T.cardinality()
1539
>>> T = crystals.HighestWeight(La[Integer(7)])
>>> T.cardinality()
56
```

An example with an affine type:

```
sage: C = CartanType(['C',2,1])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: sorted(T.subcrystal(max_depth=3), key=str)
[(-Lambda[0] + 3*Lambda[1] - Lambda[2] - delta,),
 (-Lambda[0] + Lambda[1] + Lambda[2] - delta,),
 (-Lambda[1] + 2*Lambda[2] - delta,),
 (2*Lambda[0] - Lambda[1],),
 (Lambda[0] + Lambda[1] - Lambda[2],),
 (Lambda[0] - Lambda[1] + Lambda[2],),
 (Lambda[1],)]
```

```
>>> from sage.all import *
>>> C = CartanType(['C',Integer(2),Integer(1)])
>>> La = C.root_system().weight_lattice().fundamental_weights()
>>> T = crystals.HighestWeight(La[Integer(1)])
>>> sorted(T.subcrystal(max_depth=Integer(3)), key=str)
[(-Lambda[0] + 3*Lambda[1] - Lambda[2] - delta,),
 (-Lambda[0] + Lambda[1] + Lambda[2] - delta,),
 (-Lambda[1] + 2*Lambda[2] - delta,),
 (2*Lambda[0] - Lambda[1],),
 (Lambda[0] + Lambda[1] - Lambda[2],),
```

(continues on next page)

(continued from previous page)

```
(Lambda[0] - Lambda[1] + Lambda[2],),
(Lambda[1],)]
```

Using the various models:

```
sage: La = RootSystem(['F',4]).weight_lattice().fundamental_weights()
sage: wt = La[1] + La[4]
sage: crystals.HighestWeight(wt)
The crystal of LS paths of type ['F', 4] and weight Lambda[1] + Lambda[4]
sage: crystals.HighestWeight(wt, model='NakajimaMonomials')
Highest weight crystal of modified Nakajima monomials of
 Cartan type ['F', 4] and highest weight Lambda[1] + Lambda[4]
sage: crystals.HighestWeight(wt, model='AlcovePaths')
Highest weight crystal of alcove paths of type ['F', 4] and weight Lambda[1] +␣
↪Lambda[4]
sage: crystals.HighestWeight(wt, model='RiggedConfigurations')
Crystal of rigged configurations of type ['F', 4] and weight Lambda[1] + Lambda[4]
sage: La = RootSystem(['A',3,1]).weight_lattice().fundamental_weights()
sage: wt = La[0] + La[2]
sage: crystals.HighestWeight(wt, model='GeneralizedYoungWalls')
Highest weight crystal of generalized Young walls of
 Cartan type ['A', 3, 1] and highest weight Lambda[0] + Lambda[2]
```

```
>>> from sage.all import *
>>> La = RootSystem(['F',Integer(4)]).weight_lattice().fundamental_weights()
>>> wt = La[Integer(1)] + La[Integer(4)]
>>> crystals.HighestWeight(wt)
The crystal of LS paths of type ['F', 4] and weight Lambda[1] + Lambda[4]
>>> crystals.HighestWeight(wt, model='NakajimaMonomials')
Highest weight crystal of modified Nakajima monomials of
 Cartan type ['F', 4] and highest weight Lambda[1] + Lambda[4]
>>> crystals.HighestWeight(wt, model='AlcovePaths')
Highest weight crystal of alcove paths of type ['F', 4] and weight Lambda[1] +␣
↪Lambda[4]
>>> crystals.HighestWeight(wt, model='RiggedConfigurations')
Crystal of rigged configurations of type ['F', 4] and weight Lambda[1] + Lambda[4]
>>> La = RootSystem(['A',Integer(3),Integer(1)]).weight_lattice().fundamental_
↪weights()
>>> wt = La[Integer(0)] + La[Integer(2)]
>>> crystals.HighestWeight(wt, model='GeneralizedYoungWalls')
Highest weight crystal of generalized Young walls of
 Cartan type ['A', 3, 1] and highest weight Lambda[0] + Lambda[2]
```

5.1.52 Induced Crystals

We construct a crystal structure on a set induced by a bijection Φ.

AUTHORS:

- Travis Scrimshaw (2014-05-15): Initial implementation

class sage.combinat.crystals.induced_structure.**InducedCrystal**(*X*, *phi*, *inverse*)

Bases: UniqueRepresentation, Parent

A crystal induced from an injection.

Let X be a set and let C be crystal and consider any injection $\Phi : X \to C$. We induce a crystal structure on X by considering Φ to be a crystal morphism.

Alternatively we can induce a crystal structure on some (sub)set of X by considering an injection $\Phi : C \to X$ considered as a crystal morphism. This form is also useful when the set X is not explicitly known.

INPUT:

- `X` – the base set
- `phi` – the map Φ
- `inverse` – (optional) the inverse map Φ^{-1}
- `from_crystal` – boolean (default: `False`); if the induced structure is of the second type $\Phi : C \to X$

EXAMPLES:

We construct a crystal structure of Gelfand-Tsetlin patterns by going through their bijection with semistandard tableaux:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: I.digraph().is_isomorphic(D.digraph(), edge_labels=True)
True
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)], shapes=PartitionsInBox(Integer(4),
↪Integer(3)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(3))
>>> phi = lambda x: D(x.to_tableau())
>>> phi_inv = lambda x: G(x.to_tableau())
>>> I = crystals.Induced(G, phi, phi_inv)
>>> I.digraph().is_isomorphic(D.digraph(), edge_labels=True)
True
```

Now we construct the above example but inducing the structure going the other way (from tableaux to Gelfand-Tsetlin patterns). This can also give us more information coming from the crystal.

```
sage: D2 = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G2 = GelfandTsetlinPatterns(4, 1)
sage: phi2 = lambda x: D2(x.to_tableau())
sage: phi2_inv = lambda x: G2(x.to_tableau())
sage: I2 = crystals.Induced(D2, phi2_inv, phi2, from_crystal=True)
sage: I2.module_generators
([[0, 0, 0, 0], [0, 0, 0], [0, 0], [0]],
 [[1, 0, 0, 0], [1, 0, 0], [1, 0], [1]],
 [[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]],
 [[1, 1, 1, 0], [1, 1, 1], [1, 1], [1]],
 [[1, 1, 1, 1], [1, 1, 1], [1, 1], [1]])
```

```
>>> from sage.all import *
>>> D2 = crystals.Tableaux(['A',Integer(3)], shapes=PartitionsInBox(Integer(4),
↪Integer(1)))
>>> G2 = GelfandTsetlinPatterns(Integer(4), Integer(1))
>>> phi2 = lambda x: D2(x.to_tableau())
>>> phi2_inv = lambda x: G2(x.to_tableau())
>>> I2 = crystals.Induced(D2, phi2_inv, phi2, from_crystal=True)
>>> I2.module_generators
([[0, 0, 0, 0], [0, 0, 0], [0, 0], [0]],
 [[1, 0, 0, 0], [1, 0, 0], [1, 0], [1]],
 [[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]],
 [[1, 1, 1, 0], [1, 1, 1], [1, 1], [1]],
 [[1, 1, 1, 1], [1, 1, 1], [1, 1], [1]])
```

We check an example when the codomain is larger than the domain (although here the crystal structure is trivial):

```
sage: P = Permutations(4)
sage: D = crystals.Tableaux(['A',3], shapes=Partitions(4))
sage: T = crystals.TensorProduct(D, D)
sage: phi = lambda p: T(D(RSK(p)[0]), D(RSK(p)[1]))
sage: phi_inv = lambda d: RSK_inverse(d[0].to_tableau(), d[1].to_tableau(),␣
↪output='permutation')
sage: all(phi_inv(phi(p)) == p for p in P) # Check it really is the inverse
True
sage: I = crystals.Induced(P, phi, phi_inv)
sage: I.digraph()
Digraph on 24 vertices
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
>>> D = crystals.Tableaux(['A',Integer(3)], shapes=Partitions(Integer(4)))
>>> T = crystals.TensorProduct(D, D)
>>> phi = lambda p: T(D(RSK(p)[Integer(0)]), D(RSK(p)[Integer(1)]))
>>> phi_inv = lambda d: RSK_inverse(d[Integer(0)].to_tableau(), d[Integer(1)].to_
↪tableau(), output='permutation')
>>> all(phi_inv(phi(p)) == p for p in P) # Check it really is the inverse
True
>>> I = crystals.Induced(P, phi, phi_inv)
>>> I.digraph()
Digraph on 24 vertices
```

We construct an example without a specified inverse map:

```
sage: X = Words(2,4)
sage: L = crystals.Letters(['A',1])
sage: T = crystals.TensorProduct(*[L]*4)
sage: Phi = lambda x : T(*[L(i) for i in x])
sage: I = crystals.Induced(X, Phi)
sage: I.digraph()
Digraph on 16 vertices
```

```
>>> from sage.all import *
>>> X = Words(Integer(2),Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> L = crystals.Letters(['A',Integer(1)])
>>> T = crystals.TensorProduct(*[L]*Integer(4))
>>> Phi = lambda x : T(*[L(i) for i in x])
>>> I = crystals.Induced(X, Phi)
>>> I.digraph()
Digraph on 16 vertices
```

class Element

Bases: `ElementWrapper`

An element of an induced crystal.

e(*i*)

Return e_i of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.e(1)
sage: elt.e(2)
[[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]]
sage: elt.e(3)
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],␣
↪shapes=PartitionsInBox(Integer(4),Integer(3)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(3))
>>> phi = lambda x: D(x.to_tableau())
>>> phi_inv = lambda x: G(x.to_tableau())
>>> I = crystals.Induced(G, phi, phi_inv)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],␣
↪[Integer(1)]])
>>> elt.e(Integer(1))
>>> elt.e(Integer(2))
[[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]]
>>> elt.e(Integer(3))
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
```

(continues on next page)

(continued from previous page)

```
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],
↪shapes=PartitionsInBox(Integer(4),Integer(3)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(3))
>>> phi = lambda x: D(x.to_tableau())
>>> phi_inv = lambda x: G(x.to_tableau())
>>> I = crystals.Induced(G, phi, phi_inv)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],
↪[Integer(1)]])
>>> [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

f (i)

Return f_i of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.f(1)
[[1, 1, 0, 0], [1, 1, 0], [1, 0], [0]]
sage: elt.f(2)
sage: elt.f(3)
[[1, 1, 0, 0], [1, 0, 0], [1, 0], [1]]
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],
↪shapes=PartitionsInBox(Integer(4),Integer(3)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(3))
>>> phi = lambda x: D(x.to_tableau())
>>> phi_inv = lambda x: G(x.to_tableau())
>>> I = crystals.Induced(G, phi, phi_inv)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],
↪[Integer(1)]])
>>> elt.f(Integer(1))
[[1, 1, 0, 0], [1, 1, 0], [1, 0], [0]]
>>> elt.f(Integer(2))
>>> elt.f(Integer(3))
[[1, 1, 0, 0], [1, 0, 0], [1, 0], [1]]
```

phi (i)

Return φ_i of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: [elt.phi(i) for i in I.index_set()]
[1, 0, 1]
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],␣
↪shapes=PartitionsInBox(Integer(4),Integer(3)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(3))
>>> phi = lambda x: D(x.to_tableau())
>>> phi_inv = lambda x: G(x.to_tableau())
>>> I = crystals.Induced(G, phi, phi_inv)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],␣
↪[Integer(1)]])
>>> [elt.phi(i) for i in I.index_set()]
[1, 0, 1]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.weight()
(1, 0, 1, 0)
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],␣
↪shapes=PartitionsInBox(Integer(4),Integer(3)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(3))
>>> phi = lambda x: D(x.to_tableau())
>>> phi_inv = lambda x: G(x.to_tableau())
>>> I = crystals.Induced(G, phi, phi_inv)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],␣
↪[Integer(1)]])
>>> elt.weight()
(1, 0, 1, 0)
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Permutations(4)
sage: D = crystals.Tableaux(['A',3], shapes=Partitions(4))
sage: T = crystals.TensorProduct(D, D)
sage: phi = lambda p: T(D(RSK(p)[0]), D(RSK(p)[1]))
sage: phi_inv = lambda d: RSK_inverse(d[0].to_tableau(), d[1].to_tableau(),␣
↪output='permutation')
sage: I = crystals.Induced(P, phi, phi_inv)
sage: I.cardinality() == factorial(4)
True
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
>>> D = crystals.Tableaux(['A',Integer(3)], shapes=Partitions(Integer(4)))
>>> T = crystals.TensorProduct(D, D)
>>> phi = lambda p: T(D(RSK(p)[Integer(0)]), D(RSK(p)[Integer(1)]))
>>> phi_inv = lambda d: RSK_inverse(d[Integer(0)].to_tableau(), d[Integer(1)].
↪to_tableau(), output='permutation')
>>> I = crystals.Induced(P, phi, phi_inv)
>>> I.cardinality() == factorial(Integer(4))
True
```

class sage.combinat.crystals.induced_structure.**InducedFromCrystal**(*X*, *phi*, *inverse*)

Bases: UniqueRepresentation, Parent

A crystal induced from an injection.

Alternatively we can induce a crystal structure on some (sub)set of X by considering an injection $\Phi : C \to X$ considered as a crystal morphism.

See also

InducedCrystal

INPUT:

- X – the base set
- phi – the map Φ
- inverse – (optional) the inverse map Φ^{-1}

EXAMPLES:

We construct a crystal structure on generalized permutations with a fixed first row by using RSK:

```
sage: C = crystals.Tableaux(['A',3], shape=[2,1])
sage: def psi(x):
....:     ret = RSK_inverse(x.to_tableau(), Tableau([[1,1],[2]]))
....:     return (tuple(ret[0]), tuple(ret[1]))
sage: psi_inv = lambda x: C(RSK(*x)[0])
sage: I = crystals.Induced(C, psi, psi_inv, from_crystal=True)
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',Integer(3)], shape=[Integer(2),Integer(1)])
>>> def psi(x):
```

(continues on next page)

(continued from previous page)

```
...     ret = RSK_inverse(x.to_tableau(), Tableau([[Integer(1),Integer(1)],
↪[Integer(2)]]))
...     return (tuple(ret[Integer(0)]), tuple(ret[Integer(1)]))
>>> psi_inv = lambda x: C(RSK(*x)[Integer(0)])
>>> I = crystals.Induced(C, psi, psi_inv, from_crystal=True)
```

class Element

Bases: `ElementWrapper`

An element of an induced crystal.

e(*i*)

Return e_i of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.e(1)
sage: elt.e(2)
[[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]]
sage: elt.e(3)
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],␣
↪shapes=PartitionsInBox(Integer(4),Integer(1)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(1))
>>> def phi(x): return G(x.to_tableau())
>>> def phi_inv(x): return D(G(x).to_tableau())
>>> I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],␣
↪[Integer(1)]])
>>> elt.e(Integer(1))
>>> elt.e(Integer(2))
[[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]]
>>> elt.e(Integer(3))
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
```

(continues on next page)

(continued from previous page)

```
sage: [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],␣
↪shapes=PartitionsInBox(Integer(4),Integer(1)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(1))
>>> def phi(x): return G(x.to_tableau())
>>> def phi_inv(x): return D(G(x).to_tableau())
>>> I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],␣
↪[Integer(1)]])
>>> [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

f (*i*)

Return f_i of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.f(1)
[[1, 1, 0, 0], [1, 1, 0], [1, 0], [0]]
sage: elt.f(2)
sage: elt.f(3)
[[1, 1, 0, 0], [1, 0, 0], [1, 0], [1]]
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],␣
↪shapes=PartitionsInBox(Integer(4),Integer(1)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(1))
>>> def phi(x): return G(x.to_tableau())
>>> def phi_inv(x): return D(G(x).to_tableau())
>>> I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],␣
↪[Integer(1)]])
>>> elt.f(Integer(1))
[[1, 1, 0, 0], [1, 1, 0], [1, 0], [0]]
>>> elt.f(Integer(2))
>>> elt.f(Integer(3))
[[1, 1, 0, 0], [1, 0, 0], [1, 0], [1]]
```

phi (*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],␣
↪shapes=PartitionsInBox(Integer(4),Integer(1)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(1))
>>> def phi(x): return G(x.to_tableau())
>>> def phi_inv(x): return D(G(x).to_tableau())
>>> I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],␣
↪[Integer(1)]])
>>> [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.weight()
(1, 0, 1, 0)
```

```
>>> from sage.all import *
>>> D = crystals.Tableaux(['A',Integer(3)],␣
↪shapes=PartitionsInBox(Integer(4),Integer(1)))
>>> G = GelfandTsetlinPatterns(Integer(4), Integer(1))
>>> def phi(x): return G(x.to_tableau())
>>> def phi_inv(x): return D(G(x).to_tableau())
>>> I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
>>> elt = I([[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0)],␣
↪[Integer(1)]])
>>> elt.weight()
(1, 0, 1, 0)
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: C = crystals.Tableaux(['A',3], shape=[2,1])
sage: def psi(x):
....:     ret = RSK_inverse(x.to_tableau(), Tableau([[1,1],[2]]))
....:     return (tuple(ret[0]), tuple(ret[1]))
sage: psi_inv = lambda x: C(RSK(*x)[0])
sage: I = crystals.Induced(C, psi, psi_inv, from_crystal=True)
sage: I.cardinality() == C.cardinality()
True
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',Integer(3)], shape=[Integer(2),Integer(1)])
>>> def psi(x):
...     ret = RSK_inverse(x.to_tableau(), Tableau([[Integer(1),Integer(1)],
↪[Integer(2)]]))
...     return (tuple(ret[Integer(0)]), tuple(ret[Integer(1)]))
>>> psi_inv = lambda x: C(RSK(*x)[Integer(0)])
>>> I = crystals.Induced(C, psi, psi_inv, from_crystal=True)
>>> I.cardinality() == C.cardinality()
True
```

5.1.53 $\mathcal{B}(\infty)$ Crystals of Tableaux in Nonexceptional Types and G_2

A tableau model for $\mathcal{B}(\infty)$. For more information, see *InfinityCrystalOfTableaux*.

AUTHORS:

- Ben Salisbury: Initial version
- Travis Scrimshaw: Initial version

class sage.combinat.crystals.infinity_crystals.**DualInfinityQueerCrystalOfTableaux**(*cartan_type*)

Bases: *CrystalOfWords*

Initialize self.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',2])
sage: TestSuite(B).run() # long time
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['A',Integer(2)])
>>> TestSuite(B).run() # long time
```

class Element

Bases: *InfinityQueerCrystalOfTableauxElement*

index_set()

Return the index set of self.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(["Q",3])
sage: B.index_set()
(1, 2, -1)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(["Q",Integer(3)])
>>> B.index_set()
(1, 2, -1)
```

module_generator()

Return the module generator (or highest weight element) of `self`.

The module generator is the unique semistandard hook tableau of shape $(n, n-1, \ldots, 2, 1)$ with weight 0.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(["Q",5])
sage: B.module_generator()
[[5, 5, 5, 5, 5], [4, 4, 4, 4], [3, 3, 3], [2, 2], [1]]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(["Q",Integer(5)])
>>> B.module_generator()
[[5, 5, 5, 5, 5], [4, 4, 4, 4], [3, 3, 3], [2, 2], [1]]
```

class sage.combinat.crystals.infinity_crystals.**InfinityCrystalOfTableaux**(*cartan_type*)

Bases: *CrystalOfWords*

$\mathcal{B}(\infty)$ crystal of tableaux.

A tableaux model $\mathcal{T}(\infty)$ for the crystal $\mathcal{B}(\infty)$ is introduced by Hong and Lee in [HL2008]. This model is currently valid for types A_n, B_n, C_n, D_n, and G_2, and builds on the tableaux model given by Kashiwara and Nakashima [KN1994] in types A_n, B_n, C_n, and D_n, and by Kang and Misra [KM1994] in type G_2.

Note

We are using the English convention for our tableaux.

We say a tableau T is *marginally large* if:

- for each $1 \leq i \leq n$, the leftmost box in the i-th row from the top in T is an i-box,
- for each $1 \leq i \leq n$, the number of i-boxes in the i-th row from the top in T is greater than the total number of boxes in the $(i+1)$-th row by exactly one.

We now will describe this tableaux model type-by-type.

Type A_n

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly n rows over the alphabet $\{1 \prec 2 \prec \cdots \prec n+1\}$.

Type B_n

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly n rows over the alphabet $\{1 \prec \cdots \prec n \prec 0 \prec \overline{n} \prec \cdots \prec \overline{1}\}$ and subject to the following constraints:

- for each $1 \leq i \leq n$, the contents of the boxes in the i-th row are $\preceq \overline{i}$,
- the entry 0 can appear at most once in a single row.

Type C_n

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly n rows over the alphabet $\{1 \prec \cdots \prec n \prec \overline{n} \prec \cdots \prec \overline{1}\}$ and for each $1 \leq i \leq n$, the contents of the boxes in the i-th row are $\preceq \overline{i}$.

Type D_n

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly $n-1$ rows over the alphabet $\{1 \prec \cdots \prec n, \overline{n} \prec \cdots \prec \overline{1}\}$ and subject to the following constraints:

- for each $1 \leq i \leq n$, the contents of the boxes in the i-th row are $\preceq \overline{i}$,
- the entries n and $\overline{n}$ may not appear simultaneously in a single row.

Type G_2

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly 2 rows over the ordered alphabet $\{1 \prec 2 \prec 3 \prec 0 \prec \overline{3} \prec \overline{2} \prec \overline{1}\}$ and subject to the following constraints:

- the contents of the boxes in the first row are $\preceq \overline{i}$,
- the contents of the boxes in the second row are $\preceq 3$,
- the entry 0 can appear at most once in the first row and not at all in the second row.

In particular, the shape of the tableaux is not fixed in any instance of $\mathcal{T}(\infty)$; the row lengths of a tableau can be arbitrarily long.

INPUT:

- `cartan_type` – one of `['A',n]`, `['B',n]`, `['C',n]`, `['D',n]`, or `['G',2]`, where `n` is a positive integer

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',2])
sage: b = B.highest_weight_vector(); b.pp()
1  1
2
sage: b.f_string([2,1,1,2,2,2]).pp()
1  1  1  1  1  2  3
2  3  3  3

sage: B = crystals.infinity.Tableaux(['G',2])
sage: b = B(rows=[[1,1,1,1,1,2,3,3,0,-3,-1,-1,-1],[2,3,3,3]])
sage: b.e_string([2,1,1,1,1,1,1]).pp()
1  1  1  1  2  3  3  3  3 -2 -2 -2
2  3  3
sage: b.e_string([2,1,1,1,1,1,1,1])
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['A',Integer(2)])
>>> b = B.highest_weight_vector(); b.pp()
1  1
2
>>> b.f_string([Integer(2),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(2)]).pp()
1  1  1  1  1  2  3
2  3  3  3
```

(continues on next page)

(continued from previous page)

```
>>> B = crystals.infinity.Tableaux(['G',Integer(2)])
>>> b = B(rows=[[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(2),Integer(3),Integer(3),Integer(0),-Integer(3),-Integer(1),-Integer(1),
↪-Integer(1)],[Integer(2),Integer(3),Integer(3),Integer(3)]])
>>> b.e_string([Integer(2),Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1)]).pp()
1  1  1  1  2  3  3  3  3 -2 -2 -2
2  3  3
>>> b.e_string([Integer(2),Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(1)])
```

We check that a few classical crystals embed into $\mathcal{T}(\infty)$:

```
sage: def crystal_test(B, C):
....:     T = crystals.elementary.T(C.cartan_type(), C.module_generators[0].
↪weight())
....:     TP = crystals.TensorProduct(T, B)
....:     mg = TP(T[0], B.module_generators[0])
....:     g = {C.module_generators[0]: mg}
....:     f = C.crystal_morphism(g, category=HighestWeightCrystals())
....:     G = B.digraph(subset=[f(x) for x in C])
....:     return G.is_isomorphic(C.digraph(), edge_labels=True)
sage: B = crystals.infinity.Tableaux(['A',2])
sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: crystal_test(B, C)
True
sage: C = crystals.Tableaux(['A',2], shape=[6,2])
sage: crystal_test(B, C)
True
sage: B = crystals.infinity.Tableaux(['B',2])
sage: C = crystals.Tableaux(['B',2], shape=[3])
sage: crystal_test(B, C)
True
sage: C = crystals.Tableaux(['B',2], shape=[2,1])
sage: crystal_test(B, C)
True
sage: B = crystals.infinity.Tableaux(['C',3])
sage: C = crystals.Tableaux(['C',3], shape=[2,1])
sage: crystal_test(B, C)
True
sage: B = crystals.infinity.Tableaux(['D',4])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: crystal_test(B, C)
True
sage: C = crystals.Tableaux(['D',4], shape=[1,1,1,1])
sage: crystal_test(B, C)
True
sage: B = crystals.infinity.Tableaux(['G',2])
sage: C = crystals.Tableaux(['G',2], shape=[3])
sage: crystal_test(B, C)
True
```

```
>>> from sage.all import *
>>> def crystal_test(B, C):
...     T = crystals.elementary.T(C.cartan_type(), C.module_
↪generators[Integer(0)].weight())
...     TP = crystals.TensorProduct(T, B)
...     mg = TP(T[Integer(0)], B.module_generators[Integer(0)])
...     g = {C.module_generators[Integer(0)]: mg}
...     f = C.crystal_morphism(g, category=HighestWeightCrystals())
...     G = B.digraph(subset=[f(x) for x in C])
...     return G.is_isomorphic(C.digraph(), edge_labels=True)
>>> B = crystals.infinity.Tableaux(['A',Integer(2)])
>>> C = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2),Integer(1)])
>>> crystal_test(B, C)
True
>>> C = crystals.Tableaux(['A',Integer(2)], shape=[Integer(6),Integer(2)])
>>> crystal_test(B, C)
True
>>> B = crystals.infinity.Tableaux(['B',Integer(2)])
>>> C = crystals.Tableaux(['B',Integer(2)], shape=[Integer(3)])
>>> crystal_test(B, C)
True
>>> C = crystals.Tableaux(['B',Integer(2)], shape=[Integer(2),Integer(1)])
>>> crystal_test(B, C)
True
>>> B = crystals.infinity.Tableaux(['C',Integer(3)])
>>> C = crystals.Tableaux(['C',Integer(3)], shape=[Integer(2),Integer(1)])
>>> crystal_test(B, C)
True
>>> B = crystals.infinity.Tableaux(['D',Integer(4)])
>>> C = crystals.Tableaux(['D',Integer(4)], shape=[Integer(2)])
>>> crystal_test(B, C)
True
>>> C = crystals.Tableaux(['D',Integer(4)], shape=[Integer(1),Integer(1),
↪Integer(1),Integer(1)])
>>> crystal_test(B, C)
True
>>> B = crystals.infinity.Tableaux(['G',Integer(2)])
>>> C = crystals.Tableaux(['G',Integer(2)], shape=[Integer(3)])
>>> crystal_test(B, C)
True
```

class Element

Bases: *InfinityCrystalOfTableauxElement*

Elements in $\mathcal{B}(\infty)$ crystal of tableaux.

content()

Return the content of `self`.

The content $|T|$ of $T \in \mathcal{B}(\infty)$ is the number of blocks added to the highest weight to obtain T with any $\bar{\imath}$-boxes in the i-th row counted with multiplicity 2 provided the underlying Cartan type is of type B, D, or G.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("D5")
sage: b = B.highest_weight_vector().f_string([5,4,3,1,1,3,4,5,3,4,5,1,4,5,
↪2,3,5,3,2,4])
sage: b.content()
13

sage: B = crystals.infinity.Tableaux("B2")
sage: b = B(rows=[[1,1,1,1,1,1,2,2,2,-2,-2],[2,0,-2,-2,-2]])
sage: b.content()
12

sage: B = crystals.infinity.Tableaux("C2")
sage: b = B(rows=[[1,1,1,1,1,1,2,2,2,-2,-2],[2,-2,-2,-2]])
sage: b.content()
8
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("D5")
>>> b = B.highest_weight_vector().f_string([Integer(5),Integer(4),
↪Integer(3),Integer(1),Integer(1),Integer(3),Integer(4),Integer(5),
↪Integer(3),Integer(4),Integer(5),Integer(1),Integer(4),Integer(5),
↪Integer(2),Integer(3),Integer(5),Integer(3),Integer(2),Integer(4)])
>>> b.content()
13

>>> B = crystals.infinity.Tableaux("B2")
>>> b = B(rows=[[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(2),Integer(2),Integer(2),-Integer(2),-Integer(2)],
↪[Integer(2),Integer(0),-Integer(2),-Integer(2),-Integer(2)]])
>>> b.content()
12

>>> B = crystals.infinity.Tableaux("C2")
>>> b = B(rows=[[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(2),Integer(2),Integer(2),-Integer(2),-Integer(2)],
↪[Integer(2),-Integer(2),-Integer(2),-Integer(2)]])
>>> b.content()
8
```

phi(*i*)

Return φ_i of `self`.

Let $T \in \mathcal{B}(\infty)$ Define $\varphi_i(T) := \varepsilon_i(T) + \langle h_i, \mathrm{wt}(T) \rangle$, where h_i is the i-th simple coroot and $\mathrm{wt}(T)$ is the `weight()` of T.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("A3")
sage: [B.highest_weight_vector().f_string([1,3,2,3,1,3,2,1]).phi(i) for i␣
↪in B.index_set()]
[-3, 4, -3]
```

(continues on next page)

(continued from previous page)

```
sage: B = crystals.infinity.Tableaux("G2")
sage: [B.highest_weight_vector().f_string([2,2,1,2,1,1,1,2]).phi(i) for i␣
↪in B.index_set()]
[5, -3]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("A3")
>>> [B.highest_weight_vector().f_string([Integer(1),Integer(3),Integer(2),
↪Integer(3),Integer(1),Integer(3),Integer(2),Integer(1)]).phi(i) for i␣
↪in B.index_set()]
[-3, 4, -3]

>>> B = crystals.infinity.Tableaux("G2")
>>> [B.highest_weight_vector().f_string([Integer(2),Integer(2),Integer(1),
↪Integer(2),Integer(1),Integer(1),Integer(1),Integer(2)]).phi(i) for i␣
↪in B.index_set()]
[5, -3]
```

reduced_form()

Return the reduced form of `self`.

The reduced form of a tableaux $T \in \mathcal{T}(\infty)$ is the (not necessarily semistandard) tableaux obtained from T by removing all i-boxes in the i-th row, subject to the condition that if the row is empty, a $*$ is put as a placeholder. This is described in [BN2010] and [LS2012].

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',3])
sage: b = B.highest_weight_vector().f_string([2,2,2,3,3,3,3,3])
sage: b.pp()
1  1  1  1  1  1  1  1
2  2  2  2  4  4  4
3  4  4
sage: b.reduced_form()
[['*'], [4, 4, 4], [4, 4]]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['A',Integer(3)])
>>> b = B.highest_weight_vector().f_string([Integer(2),Integer(2),
↪Integer(2),Integer(3),Integer(3),Integer(3),Integer(3),Integer(3)])
>>> b.pp()
1  1  1  1  1  1  1  1
2  2  2  2  4  4  4
3  4  4
>>> b.reduced_form()
[['*'], [4, 4, 4], [4, 4]]
```

seg()

Return the statistic seg of `self`.

More precisely, following [LS2012], define a k-segment of a tableau T in $\mathcal{B}(\infty)$ to be a maximal string of k-boxes in a single row of T. Set $\mathrm{seg}'(T)$ to be the number of k-segments in T, as k varies over all possible values. Then $\mathrm{seg}(T)$ is determined type-by-type.

- In types A_n and C_n, define $\mathrm{seg}(T) := \mathrm{seg}'(T)$.
- In types B_n and G_2, set $e(T)$ to be the number of rows in T which contain both a 0-box and an $\bar{\imath}$-box. Define $\mathrm{seg}(T) := \mathrm{seg}'(T) - e(T)$.
- In type D_n, set $d(T)$ to be the number of rows in T which contain an $\bar{\imath}$-box, but no n-box nor $\overline{n}$-box. Define $\mathrm{seg}(T) := \mathrm{seg}'(T) + d(T)$.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',3])
sage: b = B.highest_weight_vector().f_string([1,3,2,2,3,1,1,3])
sage: b.pp()
1  1  1  1  1  1  2  2  4
2  2  2  2  3
3  4  4
sage: b.seg()
4

sage: B = crystals.infinity.Tableaux(['D',4])
sage: b = B(rows=[[1,1,1,1,1,1,3,-2,-1],[2,2,2,4,-2],[3,3],[4]])
sage: b.pp()
1  1  1  1  1  1  3 -2 -1
2  2  2  4 -2
3  3
4
sage: b.seg()
6

sage: B = crystals.infinity.Tableaux(['G',2])
sage: b = B.highest_weight_vector().f_string([2,1,1,1,2,1,2,2,1,2,2,2,1,2,
↪2,1])
sage: b.pp()
1  1  1  1  1  1  1  1  2  3  0 -3
2  3  3  3  3  3  3
sage: b.seg()
5
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['A',Integer(3)])
>>> b = B.highest_weight_vector().f_string([Integer(1),Integer(3),
↪Integer(2),Integer(2),Integer(3),Integer(1),Integer(1),Integer(3)])
>>> b.pp()
1  1  1  1  1  1  2  2  4
2  2  2  2  3
3  4  4
>>> b.seg()
4

>>> B = crystals.infinity.Tableaux(['D',Integer(4)])
>>> b = B(rows=[[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(3),-Integer(2),-Integer(1)],[Integer(2),Integer(2),
↪Integer(2),Integer(4),-Integer(2)],[Integer(3),Integer(3)],
↪[Integer(4)]])
>>> b.pp()
1  1  1  1  1  1  3 -2 -1
```

(continues on next page)

(continued from previous page)

```
2  2  2  4 -2
3  3
4
>>> b.seg()
6

>>> B = crystals.infinity.Tableaux(['G',Integer(2)])
>>> b = B.highest_weight_vector().f_string([Integer(2),Integer(1),
↪Integer(1),Integer(1),Integer(2),Integer(1),Integer(2),Integer(2),
↪Integer(1),Integer(2),Integer(2),Integer(2),Integer(1),Integer(2),
↪Integer(2),Integer(1)])
>>> b.pp()
1  1  1  1  1  1  1  1  2  3  0 -3
2  3  3  3  3  3  3
>>> b.seg()
5
```

weight()

Return the weight of `self`.

From the definition of a crystal and that the highest weight element b_∞ of $\mathcal{B}(\infty)$ is 0, the weight of $T \in \mathcal{B}(\infty)$ can be defined as $\mathrm{wt}(T) := -\sum_j \alpha_{i_j}$ where $\widetilde{e}_{i_1} \cdots \widetilde{e}_{i_\ell} T = b_\infty$ and $\{\alpha_i\}$ is the set of simple roots. (Note that the weight is independent of the path chosen to get to the highest weight.)

However we can also take advantage of the fact that $\rho \colon R_\lambda \otimes \mathcal{B}(\infty) \longrightarrow B(\lambda)$, where λ is the shape of T, preserves the tableau representation of T. Therefore

$$\mathrm{wt}(T) = \mathrm{wt}\bigl(\rho(T)\bigr) - \lambda$$

where $\mathrm{wt}\bigl(\rho(T)\bigr)$ is just the usual weight of the tableau T.

Let Λ_i be the i-th fundamental weight. In type D, the height $n-1$ columns corresponds to $\Lambda_{n-1} + \Lambda_n$ and the in type B, the height n columns corresponds to $2\Lambda_n$.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("C7")
sage: b = B.highest_weight_vector().f_string([1,6,4,7,4,2,4,6,2,4,6,7,1,2,
↪4,7])
sage: b.weight()
(-2, -1, 3, -5, 5, -3, -3)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("C7")
>>> b = B.highest_weight_vector().f_string([Integer(1),Integer(6),
↪Integer(4),Integer(7),Integer(4),Integer(2),Integer(4),Integer(6),
↪Integer(2),Integer(4),Integer(6),Integer(7),Integer(1),Integer(2),
↪Integer(4),Integer(7)])
>>> b.weight()
(-2, -1, 3, -5, 5, -3, -3)
```

Check that the definitions agree:

```
sage: P = B.weight_lattice_realization()
sage: alpha = P.simple_roots()
sage: b.weight() == -2*alpha[1] - 3*alpha[2] - 5*alpha[4] - 3*alpha[6] -␣
↪3*alpha[7]
True
```

```
>>> from sage.all import *
>>> P = B.weight_lattice_realization()
>>> alpha = P.simple_roots()
>>> b.weight() == -Integer(2)*alpha[Integer(1)] -␣
↪Integer(3)*alpha[Integer(2)] - Integer(5)*alpha[Integer(4)] -␣
↪Integer(3)*alpha[Integer(6)] - Integer(3)*alpha[Integer(7)]
True
```

Check that it works for type B:

```
sage: B = crystals.infinity.Tableaux("B2")
sage: B.highest_weight_vector().weight()
(0, 0)
sage: b = B.highest_weight_vector().f_string([1,2,2,2,1,2])
sage: P = B.weight_lattice_realization()
sage: alpha = P.simple_roots()
sage: b.weight() == -2*alpha[1] - 4*alpha[2]
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("B2")
>>> B.highest_weight_vector().weight()
(0, 0)
>>> b = B.highest_weight_vector().f_string([Integer(1),Integer(2),
↪Integer(2),Integer(2),Integer(1),Integer(2)])
>>> P = B.weight_lattice_realization()
>>> alpha = P.simple_roots()
>>> b.weight() == -Integer(2)*alpha[Integer(1)] -␣
↪Integer(4)*alpha[Integer(2)]
True
```

Check that it works for type D:

```
sage: B = crystals.infinity.Tableaux("D4")
sage: B.highest_weight_vector().weight()
(0, 0, 0, 0)
sage: b = B.highest_weight_vector().f_string([1,4,4,2,4,3,2,4,1,3,2,4])
sage: P = B.weight_lattice_realization()
sage: alpha = P.simple_roots()
sage: b.weight() == -2*alpha[1] - 3*alpha[2] - 2*alpha[3] - 5*alpha[4]
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("D4")
>>> B.highest_weight_vector().weight()
(0, 0, 0, 0)
```

(continues on next page)

(continued from previous page)

```
>>> b = B.highest_weight_vector().f_string([Integer(1),Integer(4),
↪Integer(4),Integer(2),Integer(4),Integer(3),Integer(2),Integer(4),
↪Integer(1),Integer(3),Integer(2),Integer(4)])
>>> P = B.weight_lattice_realization()
>>> alpha = P.simple_roots()
>>> b.weight() == -Integer(2)*alpha[Integer(1)] -␣
↪Integer(3)*alpha[Integer(2)] - Integer(2)*alpha[Integer(3)] -␣
↪Integer(5)*alpha[Integer(4)]
True
```

module_generator()

Return the module generator (or highest weight element) of `self`.

The module generator is the unique tableau of shape $(n, n-1, \ldots, 2, 1)$ with weight 0.

EXAMPLES:

```
sage: T = crystals.infinity.Tableaux(['A',3])
sage: T.module_generator()
[[1, 1, 1], [2, 2], [3]]
```

```
>>> from sage.all import *
>>> T = crystals.infinity.Tableaux(['A',Integer(3)])
>>> T.module_generator()
[[1, 1, 1], [2, 2], [3]]
```

class sage.combinat.crystals.infinity_crystals.**InfinityCrystalOfTableauxTypeD**(*cartan_type*)

Bases: *InfinityCrystalOfTableaux*

$\mathcal{B}(\infty)$ crystal of tableaux for type D_n.

This is the set $\mathcal{T}(\infty)$ of marginally large semistandard tableaux with exactly $n-1$ rows over the alphabet $\{1 \prec \cdots \prec n, \overline{n} \prec \cdots \prec \overline{1}\}$ and subject to the following constraints:

- for each $1 \leq i \leq n$, the contents of the boxes in the i-th row are $\preceq \overline{i}$,
- the entries n and $\overline{n}$ may not appear simultaneously in a single row.

For more information, see *InfinityCrystalOfTableaux*.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("D4")
sage: b = B.highest_weight_vector().f_string([4,3,2,1,4])
sage: b.pp()
1  1  1  1  1  1  2
2  2  2  2  3
3 -4 -3
sage: b.weight()
(-1, 0, -2, -1)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("D4")
>>> b = B.highest_weight_vector().f_string([Integer(4),Integer(3),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(4)])
>>> b.pp()
1  1  1  1  1  1  2
2  2  2  2  3
3 -4 -3
>>> b.weight()
(-1, 0, -2, -1)
```

class Element

Bases: *InfinityCrystalOfTableauxElementTypeD*, *Element*

Elements in $\mathcal{B}(\infty)$ crystal of tableaux for type D_n.

module_generator()

Return the module generator (or highest weight element) of `self`.

The module generator is the unique tableau of shape $(n-1,\ldots,2,1)$ with weight 0.

EXAMPLES:

```
sage: T = crystals.infinity.Tableaux(['D',4])
sage: T.module_generator()
[[1, 1, 1], [2, 2], [3]]
```

```
>>> from sage.all import *
>>> T = crystals.infinity.Tableaux(['D',Integer(4)])
>>> T.module_generator()
[[1, 1, 1], [2, 2], [3]]
```

5.1.54 Crystals of Kac modules of the general-linear Lie superalgebra

class sage.combinat.crystals.kac_modules.**CrystalOfKacModule**(*cartan_type*, *la*, *mu*)

Bases: `UniqueRepresentation`, `Parent`

Crystal of a Kac module.

Let $\mathfrak{g}$ be the general linear Lie superalgebra $\mathfrak{gl}(m|n)$. Let λ and μ be dominant weights for $\mathfrak{gl}_m$ and $\mathfrak{gl}_n$, respectively. Let K be the module $K = \langle f_\alpha \rangle$, where α ranges over all odd positive roots. A *Kac module* is the $U_q(\mathfrak{g})$-module constructed from the highest weight $U_q(\mathfrak{gl}_m \oplus \mathfrak{gl}_n)$-module $V(\lambda,\mu)$ (induced to a $U_q(\mathfrak{g})$-module in the natural way) by

$$K(\lambda,\mu) := K \otimes_L V(\lambda,\mu),$$

where L is the subalgebra generated by e_0 and $U_q(\mathfrak{gl}_m \oplus \mathfrak{gl}_n)$.

The Kac module admits a $U_q(\mathfrak{g})$-crystal structure by taking the crystal structure of K as given by *CrystalOfOddNegativeRoots* and the crystal $B(\lambda,\mu)$ (the natural crystal structure of $V(\lambda,\mu)$).

Note

Our notation differs slightly from [Kwon2012] in that our last tableau is transposed.

EXAMPLES:

```
sage: K = crystals.KacModule(['A', [1,2]], [2], [1,1])
sage: K.cardinality()
576
sage: K.cardinality().factor()
2^6 * 3^2
sage: len(K.cartan_type().root_system().ambient_space().positive_odd_roots())
6
sage: mg = K.module_generator()
sage: mg
({}, [[-2, -2]], [[1], [2]])
sage: mg.weight()
(2, 0, 1, 1, 0)
sage: mg.f(-1)
({}, [[-2, -1]], [[1], [2]])
sage: mg.f(0)
({-e[-1]+e[1]}, [[-2, -2]], [[1], [2]])
sage: mg.f(1)
sage: mg.f(2)
({}, [[-2, -2]], [[1], [3]])

sage: sorted(K.highest_weight_vectors(), key=str)
[({-e[-1]+e[3]}, [[-2, -1]], [[1], [2]]),
 ({-e[-1]+e[3]}, [[-2, -2]], [[1], [2]]),
 ({}, [[-2, -2]], [[1], [2]])]
```

```
>>> from sage.all import *
>>> K = crystals.KacModule(['A', [Integer(1),Integer(2)]], [Integer(2)],
↪[Integer(1),Integer(1)])
>>> K.cardinality()
576
>>> K.cardinality().factor()
2^6 * 3^2
>>> len(K.cartan_type().root_system().ambient_space().positive_odd_roots())
6
>>> mg = K.module_generator()
>>> mg
({}, [[-2, -2]], [[1], [2]])
>>> mg.weight()
(2, 0, 1, 1, 0)
>>> mg.f(-Integer(1))
({}, [[-2, -1]], [[1], [2]])
>>> mg.f(Integer(0))
({-e[-1]+e[1]}, [[-2, -2]], [[1], [2]])
>>> mg.f(Integer(1))
>>> mg.f(Integer(2))
({}, [[-2, -2]], [[1], [3]])

>>> sorted(K.highest_weight_vectors(), key=str)
[({-e[-1]+e[3]}, [[-2, -1]], [[1], [2]]),
 ({-e[-1]+e[3]}, [[-2, -2]], [[1], [2]]),
 ({}, [[-2, -2]], [[1], [2]])]
```

```
sage: K = crystals.KacModule(['A', [1,1]], [2], [1])
sage: K.cardinality()
96
sage: K.cardinality().factor()
2^5 * 3
sage: len(K.cartan_type().root_system().ambient_space().positive_odd_roots())
4

sage: sorted(K.highest_weight_vectors(), key=str)
[({-e[-1]+e[2]}, [[-2, -1]], [[1]]),
 ({-e[-1]+e[2]}, [[-2, -2]], [[1]]),
 ({}, [[-2, -2]], [[1]])]
sage: K.genuine_lowest_weight_vectors()
(({-e[-2]+e[1], -e[-2]+e[2], -e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[2]]),)
sage: sorted(K.lowest_weight_vectors(), key=str)
[({-e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[2]]),
 ({-e[-2]+e[1], -e[-2]+e[2], -e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[2]]),
 ({-e[-2]+e[2], -e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[1]]),
 ({-e[-2]+e[2], -e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[2]])]
```

```
>>> from sage.all import *
>>> K = crystals.KacModule(['A', [Integer(1),Integer(1)]], [Integer(2)],
↪[Integer(1)])
>>> K.cardinality()
96
>>> K.cardinality().factor()
2^5 * 3
>>> len(K.cartan_type().root_system().ambient_space().positive_odd_roots())
4

>>> sorted(K.highest_weight_vectors(), key=str)
[({-e[-1]+e[2]}, [[-2, -1]], [[1]]),
 ({-e[-1]+e[2]}, [[-2, -2]], [[1]]),
 ({}, [[-2, -2]], [[1]])]
>>> K.genuine_lowest_weight_vectors()
(({-e[-2]+e[1], -e[-2]+e[2], -e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[2]]),)
>>> sorted(K.lowest_weight_vectors(), key=str)
[({-e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[2]]),
 ({-e[-2]+e[1], -e[-2]+e[2], -e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[2]]),
 ({-e[-2]+e[2], -e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[1]]),
 ({-e[-2]+e[2], -e[-1]+e[1], -e[-1]+e[2]}, [[-1, -1]], [[2]])]
```

REFERENCES:

- [Kwon2012]

class Element

Bases: `ElementWrapper`

An element of a Kac module crystal.

e(i)

Return the action of the crystal operator e_i on `self`.

EXAMPLES:

```
sage: K = crystals.KacModule(['A', [2,2]], [2,1], [1])
sage: mg = K.module_generator()
sage: mg.e(0)
sage: mg.e(1)
sage: mg.e(-1)
sage: b = mg.f_string([1,0,1,-1,-2,0,1,2,0,-2,-1,-1,-1]); b
({-e[-3]+e[2], -e[-2]+e[1], -e[-2]+e[2]}, [[-3, -1], [-2]], [[3]])
sage: b.e(-2)
sage: b.e(-1)
({-e[-3]+e[2], -e[-2]+e[1], -e[-2]+e[2]}, [[-3, -2], [-2]], [[3]])
sage: b.e(0)
sage: b.e(1)
({-e[-3]+e[1], -e[-2]+e[1], -e[-2]+e[2]}, [[-3, -1], [-2]], [[3]])
sage: b.e(2)
({-e[-3]+e[2], -e[-2]+e[1], -e[-2]+e[2]}, [[-3, -1], [-2]], [[2]])
```

```
>>> from sage.all import *
>>> K = crystals.KacModule(['A', [Integer(2),Integer(2)]], [Integer(2),
↪Integer(1)], [Integer(1)])
>>> mg = K.module_generator()
>>> mg.e(Integer(0))
>>> mg.e(Integer(1))
>>> mg.e(-Integer(1))
>>> b = mg.f_string([Integer(1),Integer(0),Integer(1),-Integer(1),-
↪Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),-Integer(2),-
↪Integer(1),-Integer(1),-Integer(1)]); b
({-e[-3]+e[2], -e[-2]+e[1], -e[-2]+e[2]}, [[-3, -1], [-2]], [[3]])
>>> b.e(-Integer(2))
>>> b.e(-Integer(1))
({-e[-3]+e[2], -e[-2]+e[1], -e[-2]+e[2]}, [[-3, -2], [-2]], [[3]])
>>> b.e(Integer(0))
>>> b.e(Integer(1))
({-e[-3]+e[1], -e[-2]+e[1], -e[-2]+e[2]}, [[-3, -1], [-2]], [[3]])
>>> b.e(Integer(2))
({-e[-3]+e[2], -e[-2]+e[1], -e[-2]+e[2]}, [[-3, -1], [-2]], [[2]])
```

f(*i*)

Return the action of the crystal operator f_i on `self`.

EXAMPLES:

```
sage: K = crystals.KacModule(['A', [2,2]], [2,1], [1])
sage: mg = K.module_generator()
sage: mg.f(-2)
({}, [[-3, -2], [-2]], [[1]])
sage: mg.f(-1)
({}, [[-3, -3], [-1]], [[1]])
sage: mg.f(0)
({-e[-1]+e[1]}, [[-3, -3], [-2]], [[1]])
sage: mg.f(1)
({}, [[-3, -3], [-2]], [[2]])
sage: mg.f(2)
sage: b = mg.f_string([1,0,1,-1,-2,0,1,2,0,-2,-1,2,0]); b
```

(continues on next page)

(continued from previous page)

```
({-e[-3]+e[3], -e[-2]+e[1], -e[-1]+e[1], -e[-1]+e[2]},
 [[-3, -2], [-2]], [[3]])
```

```
>>> from sage.all import *
>>> K = crystals.KacModule(['A', [Integer(2),Integer(2)]], [Integer(2),
↪Integer(1)], [Integer(1)])
>>> mg = K.module_generator()
>>> mg.f(-Integer(2))
({}, [[-3, -2], [-2]], [[1]])
>>> mg.f(-Integer(1))
({}, [[-3, -3], [-1]], [[1]])
>>> mg.f(Integer(0))
({-e[-1]+e[1]}, [[-3, -3], [-2]], [[1]])
>>> mg.f(Integer(1))
({}, [[-3, -3], [-2]], [[2]])
>>> mg.f(Integer(2))
>>> b = mg.f_string([Integer(1),Integer(0),Integer(1),-Integer(1),-
↪Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),-Integer(2),-
↪Integer(1),Integer(2),Integer(0)]); b
({-e[-3]+e[3], -e[-2]+e[1], -e[-1]+e[1], -e[-1]+e[2]},
 [[-3, -2], [-2]], [[3]])
```

weight()

Return weight of `self`.

EXAMPLES:

```
sage: K = crystals.KacModule(['A', [3,2]], [2,1], [5,1])
sage: mg = K.module_generator()
sage: mg.weight()
(2, 1, 0, 0, 5, 1, 0)
sage: mg.weight().is_dominant()
True
sage: mg.f(0).weight()
(2, 1, 0, -1, 6, 1, 0)
sage: b = mg.f_string([2,1,-3,-2,-1,1,1,0,-2,-1,2,1,1,1,0,2,-3,-2,-1])
sage: b.weight()
(0, 0, 0, 1, 1, 4, 3)
```

```
>>> from sage.all import *
>>> K = crystals.KacModule(['A', [Integer(3),Integer(2)]], [Integer(2),
↪Integer(1)], [Integer(5),Integer(1)])
>>> mg = K.module_generator()
>>> mg.weight()
(2, 1, 0, 0, 5, 1, 0)
>>> mg.weight().is_dominant()
True
>>> mg.f(Integer(0)).weight()
(2, 1, 0, -1, 6, 1, 0)
>>> b = mg.f_string([Integer(2),Integer(1),-Integer(3),-Integer(2),-
↪Integer(1),Integer(1),Integer(1),Integer(0),-Integer(2),-Integer(1),
↪Integer(2),Integer(1),Integer(1),Integer(1),Integer(0),Integer(2),-
```

(continues on next page)

(continued from previous page)

```
↪Integer(3),-Integer(2),-Integer(1)])
>>> b.weight()
(0, 0, 0, 1, 1, 4, 3)
```

module_generator()

Return the module generator of `self`.

EXAMPLES:

```
sage: K = crystals.KacModule(['A', [2,1]], [2,1], [1])
sage: K.module_generator()
({}, [[-3, -3], [-2]], [[1]])
```

```
>>> from sage.all import *
>>> K = crystals.KacModule(['A', [Integer(2),Integer(1)]], [Integer(2),
↪Integer(1)], [Integer(1)])
>>> K.module_generator()
({}, [[-3, -3], [-2]], [[1]])
```

class sage.combinat.crystals.kac_modules.**CrystalOfOddNegativeRoots**(*cartan_type*)

Bases: `UniqueRepresentation`, `Parent`

Crystal of the set of odd negative roots.

Let $\mathfrak{g}$ be the general-linear Lie superalgebra $\mathfrak{gl}(m|n)$. This is the crystal structure on the set of negative roots as given by [Kwon2012].

More specifically, this is the crystal basis of the subalgebra of $U_q^-(\mathfrak{g})$ generated by f_α, where α ranges over all odd positive roots. As $\mathbf{Q}(q)$-modules, we have

$$U_q^-(\mathfrak{g}) \cong K \otimes U_q^-(\mathfrak{gl}_m \oplus \mathfrak{gl}_n).$$

EXAMPLES:

```
sage: S = crystals.OddNegativeRoots(['A', [2,1]])
sage: mg = S.module_generator(); mg
{}
sage: mg.f(0)
{-e[-1]+e[1]}
sage: mg.f_string([0,-1,0,1,2,1,0])
{-e[-2]+e[3], -e[-1]+e[1], -e[-1]+e[2]}
```

```
>>> from sage.all import *
>>> S = crystals.OddNegativeRoots(['A', [Integer(2),Integer(1)]])
>>> mg = S.module_generator(); mg
{}
>>> mg.f(Integer(0))
{-e[-1]+e[1]}
>>> mg.f_string([Integer(0),-Integer(1),Integer(0),Integer(1),Integer(2),
↪Integer(1),Integer(0)])
{-e[-2]+e[3], -e[-1]+e[1], -e[-1]+e[2]}
```

class Element

Bases: `ElementWrapper`

An element of the crystal of odd negative roots.

e(*i*)

Return the action of the crystal operator e_i on `self`.

EXAMPLES:

```
sage: S = crystals.OddNegativeRoots(['A', [2,2]])
sage: mg = S.module_generator()
sage: mg.e(0)
sage: mg.e(1)
sage: b = mg.f_string([0,1,2,-1,0])
sage: b.e(-1)
sage: b.e(0)
{-e[-2]+e[3]}
sage: b.e(1)
sage: b.e(2)
{-e[-2]+e[2], -e[-1]+e[1]}
sage: b.e_string([2,1,0,-1,0])
{}
```

```
>>> from sage.all import *
>>> S = crystals.OddNegativeRoots(['A', [Integer(2),Integer(2)]])
>>> mg = S.module_generator()
>>> mg.e(Integer(0))
>>> mg.e(Integer(1))
>>> b = mg.f_string([Integer(0),Integer(1),Integer(2),-Integer(1),
↪Integer(0)])
>>> b.e(-Integer(1))
>>> b.e(Integer(0))
{-e[-2]+e[3]}
>>> b.e(Integer(1))
>>> b.e(Integer(2))
{-e[-2]+e[2], -e[-1]+e[1]}
>>> b.e_string([Integer(2),Integer(1),Integer(0),-Integer(1),Integer(0)])
{}
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: S = crystals.OddNegativeRoots(['A', [2,2]])
sage: mg = S.module_generator()
sage: [mg.epsilon(i) for i in S.index_set()]
[0, 0, 0, 0, 0]
sage: b = mg.f_string([0,1,0,-1,0,-1,-2,-2]); b
{-e[-3]+e[1], -e[-3]+e[2], -e[-1]+e[1]}
sage: [b.epsilon(i) for i in S.index_set()]
[2, 0, 1, 0, 0]
sage: b = mg.f_string([0,1,0,-1,0,-1,-2,-2,2,-1,0]); b
{-e[-3]+e[1], -e[-3]+e[3], -e[-2]+e[1], -e[-1]+e[1]}
sage: [b.epsilon(i) for i in S.index_set()]
[1, 0, 1, 0, 1]
```

```
>>> from sage.all import *
>>> S = crystals.OddNegativeRoots(['A', [Integer(2),Integer(2)]])
>>> mg = S.module_generator()
>>> [mg.epsilon(i) for i in S.index_set()]
[0, 0, 0, 0, 0]
>>> b = mg.f_string([Integer(0),Integer(1),Integer(0),-Integer(1),
↪Integer(0),-Integer(1),-Integer(2),-Integer(2)]); b
{-e[-3]+e[1], -e[-3]+e[2], -e[-1]+e[1]}
>>> [b.epsilon(i) for i in S.index_set()]
[2, 0, 1, 0, 0]
>>> b = mg.f_string([Integer(0),Integer(1),Integer(0),-Integer(1),
↪Integer(0),-Integer(1),-Integer(2),-Integer(2),Integer(2),-Integer(1),
↪Integer(0)]); b
{-e[-3]+e[1], -e[-3]+e[3], -e[-2]+e[1], -e[-1]+e[1]}
>>> [b.epsilon(i) for i in S.index_set()]
[1, 0, 1, 0, 1]
```

f(*i*)

Return the action of the crystal operator f_i on `self`.

EXAMPLES:

```
sage: S = crystals.OddNegativeRoots(['A', [2,2]])
sage: mg = S.module_generator()
sage: mg.f(0)
{-e[-1]+e[1]}
sage: mg.f(1)
sage: b = mg.f_string([0,1,2,-1,0]); b
{-e[-2]+e[3], -e[-1]+e[1]}
sage: b.f(-2)
{-e[-3]+e[3], -e[-1]+e[1]}
sage: b.f(-1)
sage: b.f(0)
sage: b.f(1)
{-e[-2]+e[3], -e[-1]+e[2]}
```

```
>>> from sage.all import *
>>> S = crystals.OddNegativeRoots(['A', [Integer(2),Integer(2)]])
>>> mg = S.module_generator()
>>> mg.f(Integer(0))
{-e[-1]+e[1]}
>>> mg.f(Integer(1))
>>> b = mg.f_string([Integer(0),Integer(1),Integer(2),-Integer(1),
↪Integer(0)]); b
{-e[-2]+e[3], -e[-1]+e[1]}
>>> b.f(-Integer(2))
{-e[-3]+e[3], -e[-1]+e[1]}
>>> b.f(-Integer(1))
>>> b.f(Integer(0))
>>> b.f(Integer(1))
{-e[-2]+e[3], -e[-1]+e[2]}
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: S = crystals.OddNegativeRoots(['A', [2,2]])
sage: mg = S.module_generator()
sage: [mg.phi(i) for i in S.index_set()]
[0, 0, 1, 0, 0]
sage: b = mg.f(0)
sage: [b.phi(i) for i in S.index_set()]
[0, 1, 0, 1, 0]
sage: b = mg.f_string([0,1,0,-1,0,-1]); b
{-e[-2]+e[1], -e[-2]+e[2], -e[-1]+e[1]}
sage: [b.phi(i) for i in S.index_set()]
[2, 0, 0, 1, 1]
```

```
>>> from sage.all import *
>>> S = crystals.OddNegativeRoots(['A', [Integer(2),Integer(2)]])
>>> mg = S.module_generator()
>>> [mg.phi(i) for i in S.index_set()]
[0, 0, 1, 0, 0]
>>> b = mg.f(Integer(0))
>>> [b.phi(i) for i in S.index_set()]
[0, 1, 0, 1, 0]
>>> b = mg.f_string([Integer(0),Integer(1),Integer(0),-Integer(1),
↪Integer(0),-Integer(1)]); b
{-e[-2]+e[1], -e[-2]+e[2], -e[-1]+e[1]}
>>> [b.phi(i) for i in S.index_set()]
[2, 0, 0, 1, 1]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: S = crystals.OddNegativeRoots(['A', [2,2]])
sage: mg = S.module_generator()
sage: mg.weight()
(0, 0, 0, 0, 0, 0)
sage: mg.f_string([0,1,2,-1,-2]).weight()
(-1, 0, 0, 0, 0, 1)
sage: mg.f_string([0,1,2,-1,-2,0,1,0,2]).weight()
(-1, 0, -2, 1, 0, 2)
```

```
>>> from sage.all import *
>>> S = crystals.OddNegativeRoots(['A', [Integer(2),Integer(2)]])
>>> mg = S.module_generator()
>>> mg.weight()
(0, 0, 0, 0, 0, 0)
>>> mg.f_string([Integer(0),Integer(1),Integer(2),-Integer(1),-
↪Integer(2)]).weight()
(-1, 0, 0, 0, 0, 1)
>>> mg.f_string([Integer(0),Integer(1),Integer(2),-Integer(1),-Integer(2),
↪Integer(0),Integer(1),Integer(0),Integer(2)]).weight()
(-1, 0, -2, 1, 0, 2)
```

module_generator()

Return the module generator of `self`.

EXAMPLES:

```
sage: S = crystals.OddNegativeRoots(['A', [2,1]])
sage: S.module_generator()
{}
```

```
>>> from sage.all import *
>>> S = crystals.OddNegativeRoots(['A', [Integer(2),Integer(1)]])
>>> S.module_generator()
{}
```

sage.combinat.crystals.kac_modules.**latex_dual**(*elt*)

Return a latex representation of a type A_n crystal tableau `elt` expressed in terms of dual letters.

The dual letter of k is expressed as $\overline{n+2-k}$.

EXAMPLES:

```
sage: from sage.combinat.crystals.kac_modules import latex_dual
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: print(latex_dual(T[0]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{2}c}\cline{1-2}
\lr{\overline{3}}&\lr{\overline{3}}\\\cline{1-2}
\lr{\overline{2}}\\\cline{1-1}
\end{array}$}
}
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kac_modules import latex_dual
>>> T = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2),Integer(1)])
>>> print(latex_dual(T[Integer(0)]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{2}c}\cline{1-2}
\lr{\overline{3}}&\lr{\overline{3}}\\\cline{1-2}
\lr{\overline{2}}\\\cline{1-1}
\end{array}$}
}
```

sage.combinat.crystals.kac_modules.**to_dual_tableau**(*elt*)

Return a type A_n crystal tableau `elt` as a tableau expressed in terms of dual letters.

The dual letter of k is expressed as $\overline{n+2-k}$ represented as $-(n+2-k)$.

EXAMPLES:

```
sage: from sage.combinat.crystals.kac_modules import to_dual_tableau
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: ascii_art([to_dual_tableau(t) for t in T])
[  -3 -3    -3 -2    -3 -1    -3 -1    -2 -1    -3 -3    -3 -2    -2 -2 ]
[  -2   ,   -2   ,   -2   ,   -1   ,   -1   ,   -1   ,   -1   ,   -1     ]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kac_modules import to_dual_tableau
>>> T = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2),Integer(1)])
>>> ascii_art([to_dual_tableau(t) for t in T])
[  -3 -3   -3 -2   -3 -1   -3 -1   -2 -1   -3 -3   -3 -2   -2 -2 ]
[  -2    , -2    , -2    , -1    , -1    , -1    , -1    , -1     ]
```

5.1.55 Kirillov-Reshetikhin Crystals

class sage.combinat.crystals.kirillov_reshetikhin.**AmbientRetractMap**(*base*, *ambient*, *pdict_inv*, *index_set*, *similarity_factor_domain=None*, *automorphism=None*)

Bases: Map

The retraction map from the ambient crystal.

Consider a crystal embedding $\phi : X \to Y$, then the elements X can be considered as a subcrystal of the ambient crystal Y. The ambient retract is the partial map $\tilde{\phi} : Y \to X$ such that $\tilde{\phi} \circ \phi$ is the identity on X.

class sage.combinat.crystals.kirillov_reshetikhin.**CrystalDiagramAutomorphism**(*C*, *on_hw*, *index_set=None*, *automorphism=None*, *cache=True*)

Bases: CrystalMorphism

The crystal automorphism induced from the diagram automorphism.

For example, in type $A_n^{(1)}$ this is the promotion operator and in type $D_n^{(1)}$, this corresponds to the automorphism induced from interchanging the 0 and 1 nodes in the Dynkin diagram.

INPUT:

- C – a crystal
- on_hw – a function for the images of the index_set-highest weight elements
- index_set – (default: the empty set) the index set
- automorphism – (default: the identity) the twisting automorphism
- cache – boolean (default: True); cache the result

is_embedding()

Return True as self is a crystal isomorphism.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.promotion().is_isomorphism()
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.promotion().is_isomorphism()
True
```

is_isomorphism()

Return `True` as `self` is a crystal isomorphism.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.promotion().is_isomorphism()
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.promotion().is_isomorphism()
True
```

is_strict()

Return `True` as `self` is a crystal isomorphism.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.promotion().is_isomorphism()
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.promotion().is_isomorphism()
True
```

is_surjective()

Return `True` as `self` is a crystal isomorphism.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.promotion().is_isomorphism()
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.promotion().is_isomorphism()
True
```

class sage.combinat.crystals.kirillov_reshetikhin.**CrystalOfTableaux_E7**(*cartan_type*, *shapes*)

Bases: *CrystalOfTableaux*

The type E_7 crystal $B(s\Lambda_7)$.

This is a helper class for the corresponding:class:$KRcrystal < sage.combinat.crystals.kirillov_reshetikhin.KR_type_E7 >$ $B^{7,s}$.

module_generator(*shape*)

Return the module generator of `self` with shape `shape`.

Note

Only implemented for single rows (i.e., highest weight $s\Lambda_7$).

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import
↪CrystalOfTableaux_E7
sage: T = CrystalOfTableaux_E7(CartanType(['E',7]), shapes=(Partition([5]),))
sage: T.module_generator([5])
[[(7,), (7,), (7,), (7,), (7,)]]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kirillov_reshetikhin import CrystalOfTableaux_
↪E7
>>> T = CrystalOfTableaux_E7(CartanType(['E',Integer(7)]),
↪shapes=(Partition([Integer(5)]),))
>>> T.module_generator([Integer(5)])
[[(7,), (7,), (7,), (7,), (7,)]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_A**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinCrystalFromPromotion*

Class of Kirillov-Reshetikhin crystals of type $A_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: b = K(rows=[[1,2],[2,4]])
sage: b.f(0)
[[1, 1], [2, 2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> b = K(rows=[[Integer(1),Integer(2)],[Integer(2),Integer(4)]])
>>> b.f(Integer(0))
[[1, 1], [2, 2]]
```

classical_decomposition()

Specifies the classical crystal underlying the KR crystal of type A.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['A', 3] and shape(s) [[2, 2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['A', 3] and shape(s) [[2, 2]]
```

dynkin_diagram_automorphism(i)

Specifies the Dynkin diagram automorphism underlying the promotion action on the crystal elements. The automorphism needs to map node 0 to some other Dynkin node.

For type A we use the Dynkin diagram automorphism which $i \mapsto i+1 \mod n+1$, where n is the rank.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.dynkin_diagram_automorphism(0)
1
sage: K.dynkin_diagram_automorphism(3)
0
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.dynkin_diagram_automorphism(Integer(0))
1
>>> K.dynkin_diagram_automorphism(Integer(3))
0
```

promotion()

Specifies the promotion operator used to construct the affine type A crystal.

For type A this corresponds to the Dynkin diagram automorphism which $i \mapsto i+1 \mod n+1$, where n is the rank.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: b = K.classical_decomposition()(rows=[[1,2],[3,4]])
sage: K.promotion()(b)
[[1, 3], [2, 4]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> b = K.classical_decomposition()(rows=[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]])
>>> K.promotion()(b)
[[1, 3], [2, 4]]
```

promotion_inverse()

Specifies the inverse promotion operator used to construct the affine type A crystal.

For type A this corresponds to the Dynkin diagram automorphism which $i \mapsto i-1 \mod n+1$, where n is the rank.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: b = K.classical_decomposition()(rows=[[1,3],[2,4]])
sage: K.promotion_inverse()(b)
[[1, 2], [3, 4]]
```

(continues on next page)

(continued from previous page)

```
sage: b = K.classical_decomposition()(rows=[[1,2],[3,3]])
sage: K.promotion_inverse()(K.promotion()(b))
[[1, 2], [3, 3]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> b = K.classical_decomposition()(rows=[[Integer(1),Integer(3)],[Integer(2),
↪Integer(4)]])
>>> K.promotion_inverse()(b)
[[1, 2], [3, 4]]
>>> b = K.classical_decomposition()(rows=[[Integer(1),Integer(2)],[Integer(3),
↪Integer(3)]])
>>> K.promotion_inverse()(K.promotion()(b))
[[1, 2], [3, 3]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_A2**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: *KirillovReshetikhinGenericCrystal*

Class of Kirillov-Reshetikhin crystals $B^{r,s}$ of type $A_{2n}^{(2)}$ for $1 \leq r \leq n$ in the realization with classical subalgebra B_n. The Cartan type in this case is inputted as the dual of $A_{2n}^{(2)}$.

This is an alternative implementation to *KR_type_box* that uses the classical decomposition into type C_n crystals.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: K
Kirillov-Reshetikhin crystal of type ['BC', 2, 2]^* with (r,s)=(1,1)
sage: b = K(rows=[[-1]])
sage: b.f(0)
[[1]]
sage: b.e(0)
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),␣
↪Integer(1))
>>> K
Kirillov-Reshetikhin crystal of type ['BC', 2, 2]^* with (r,s)=(1,1)
>>> b = K(rows=[[-Integer(1)]])
>>> b.f(Integer(0))
[[1]]
>>> b.e(Integer(0))
```

We can now check whether the two KR crystals of type $A_4^{(2)}$ (namely the KR crystal and its dual construction) are isomorphic up to relabelling of the edges:

```
sage: C = CartanType(['A',4,2])
sage: K = crystals.KirillovReshetikhin(C,1,1)
sage: Kdual = crystals.KirillovReshetikhin(C.dual(),1,1)
sage: G = K.digraph()
```

(continues on next page)

(continued from previous page)

```
sage: Gdual = Kdual.digraph()
sage: f = {0:2, 1:1, 2:0}
sage: Gnew = DiGraph(); Gnew.add_vertices(Gdual.vertices(sort=True)); Gnew.add_
↪edges([(u,v,f[i]) for (u,v,i) in Gdual.edges(sort=True)])
sage: G.is_isomorphic(Gnew, edge_labels = True)
True
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)])
>>> K = crystals.KirillovReshetikhin(C,Integer(1),Integer(1))
>>> Kdual = crystals.KirillovReshetikhin(C.dual(),Integer(1),Integer(1))
>>> G = K.digraph()
>>> Gdual = Kdual.digraph()
>>> f = {Integer(0):Integer(2), Integer(1):Integer(1), Integer(2):Integer(0)}
>>> Gnew = DiGraph(); Gnew.add_vertices(Gdual.vertices(sort=True)); Gnew.add_
↪edges([(u,v,f[i]) for (u,v,i) in Gdual.edges(sort=True)])
>>> G.is_isomorphic(Gnew, edge_labels = True)
True
```

Element

alias of *KR_type_A2Element*

ambient_crystal()

Return the ambient crystal $B^{r,s}$ of type $B_{n+1}^{(1)}$ associated to the Kirillov-Reshetikhin crystal of type $A_{2n}^{(2)}$ dual.

This ambient crystal is used to construct the zero arrows.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 2, 3)
sage: K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(2,3)
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(2),␣
↪Integer(3))
>>> K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(2,3)
```

ambient_dict_pm_diagrams()

Return a dictionary of all self-dual $\pm$ diagrams for the ambient crystal whose keys are their inner shape.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: K.ambient_dict_pm_diagrams()
{[1]: [[0, 0], [1]]}
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: K.ambient_dict_pm_diagrams()
```

(continues on next page)

(continued from previous page)

```
{[]: [[1, 1], [0]], [2]: [[0, 0], [2]]}
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 2, 2)
sage: K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0, 0], [0]],
 [2]: [[0, 0], [1, 1], [0]],
 [2, 2]: [[0, 0], [0, 0], [2]]}
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),
↪Integer(1))
>>> K.ambient_dict_pm_diagrams()
{[1]: [[0, 0], [1]]}
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),
↪Integer(2))
>>> K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0]], [2]: [[0, 0], [2]]}
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(2),
↪Integer(2))
>>> K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0, 0], [0]],
 [2]: [[0, 0], [1, 1], [0]],
 [2, 2]: [[0, 0], [0, 0], [2]]}
```

ambient_highest_weight_dict()

Return a dictionary of all $\{2, \ldots, n+1\}$-highest weight vectors in the ambient crystal.

The key is the inner shape of their corresponding $\pm$ diagram, or equivalently, their $\{2, \ldots, n+1\}$ weight.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: K.ambient_highest_weight_dict()
{[]: [[1, -1]], [2]: [[2, 2]]}
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),
↪Integer(2))
>>> K.ambient_highest_weight_dict()
{[]: [[1, -1]], [2]: [[2, 2]]}
```

classical_decomposition()

Return the classical crystal underlying the Kirillov-Reshetikhin crystal of type $A_{2n}^{(2)}$ with B_n as classical subdiagram.

It is given by $B^{r,s} \cong \bigoplus_\Lambda B(\Lambda)$, where $B(\Lambda)$ is a highest weight crystal of type B_n of highest weight Λ. The sum is over all weights Λ obtained from a rectangle of width s and height r by removing horizontal dominoes. Here we identify the fundamental weight Λ_i with a column of height i.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 2, 2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 2] and shape(s) [[], [2], [2, 2]]
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(2),␣
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['B', 2] and shape(s) [[], [2], [2, 2]]
```

from_ambient_crystal()

Return a map from the ambient crystal of type $B_{n+1}^{(1)}$ to the Kirillov-Reshetikhin crystal of type $A_{2n}^{(2)}$.

Note that this map is only well-defined on type $A_{2n}^{(2)}$ elements that are in the image under *to_ambient_crystal()*.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: b = K.ambient_crystal()(rows=[[2,2]])
sage: K.from_ambient_crystal()(b)
[[1, 1]]
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),␣
↪Integer(2))
>>> b = K.ambient_crystal()(rows=[[Integer(2),Integer(2)]])
>>> K.from_ambient_crystal()(b)
[[1, 1]]
```

highest_weight_dict()

Return a dictionary of the classical highest weight vectors of `self` whose keys are their shape.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: K.highest_weight_dict()
{[]: [], [2]: [[1, 1]]}
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),␣
↪Integer(2))
>>> K.highest_weight_dict()
{[]: [], [2]: [[1, 1]]}
```

module_generator()

Return the unique module generator of classical weight $s\Lambda_r$ of a Kirillov-Reshetikhin crystal $B^{r,s}$.

EXAMPLES:

```
sage: ct = CartanType(['A',8,2]).dual()
sage: K = crystals.KirillovReshetikhin(ct, 3, 5)
sage: K.module_generator()
[[1, 1, 1, 1, 1], [2, 2, 2, 2, 2], [3, 3, 3, 3, 3]]
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(8),Integer(2)]).dual()
>>> K = crystals.KirillovReshetikhin(ct, Integer(3), Integer(5))
>>> K.module_generator()
[[1, 1, 1, 1, 1], [2, 2, 2, 2, 2], [3, 3, 3, 3, 3]]
```

to_ambient_crystal()

Return a map from the Kirillov-Reshetikhin crystal of type $A_{2n}^{(2)}$ to the ambient crystal of type $B_{n+1}^{(1)}$.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: b=K(rows=[[1,1]])
sage: K.to_ambient_crystal()(b)
[[2, 2]]
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 2, 2)
sage: b=K(rows=[[1,1]])
sage: K.to_ambient_crystal()(b)
[[1, 2], [2, -1]]
sage: K.to_ambient_crystal()(b).parent()
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(2,2)
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),␣
↪Integer(2))
>>> b=K(rows=[[Integer(1),Integer(1)]])
>>> K.to_ambient_crystal()(b)
[[2, 2]]
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(2),␣
↪Integer(2))
>>> b=K(rows=[[Integer(1),Integer(1)]])
>>> K.to_ambient_crystal()(b)
[[1, 2], [2, -1]]
>>> K.to_ambient_crystal()(b).parent()
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(2,2)
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_A2Element**

Bases: *KirillovReshetikhinGenericCrystalElement*

Class for the elements in the Kirillov-Reshetikhin crystals $B^{r,s}$ of type $A_{2n}^{(2)}$ for $r < n$ with underlying classical algebra B_n.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2_with_category.
↪element_class'>
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),␣
↪Integer(2))
>>> type(K.module_generators[Integer(0)])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2_with_category.
↪element_class'>
```

e0()

Return e_0 on `self` by mapping `self` to the ambient crystal, calculating $e_1 e_0$ there and pulling the element back.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: b = K(rows=[[1]])
sage: b.e(0) # indirect doctest
[[-1]]
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),␣
↪Integer(1))
>>> b = K(rows=[[Integer(1)]])
>>> b.e(Integer(0)) # indirect doctest
[[-1]]
```

epsilon0()

Calculate ε_0 of `self` by mapping the element to the ambient crystal and calculating `\varepsilon_1` there.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: b=K(rows=[[1]])
sage: b.epsilon(0) # indirect doctest
1
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),␣
↪Integer(1))
>>> b=K(rows=[[Integer(1)]])
>>> b.epsilon(Integer(0)) # indirect doctest
1
```

f0()

Return f_0 on `self` by mapping `self` to the ambient crystal, calculating $f_1 f_0$ there and pulling the element back.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: b = K(rows=[[-1]])
sage: b.f(0) # indirect doctest
[[1]]
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),
↪Integer(1))
>>> b = K(rows=[[-Integer(1)]])
>>> b.f(Integer(0)) # indirect doctest
[[1]]
```

phi0()

Calculate φ_0 of `self` by mapping the element to the ambient crystal and calculating φ_1 there.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: b = K(rows=[[-1]])
sage: b.phi(0) # indirect doctest
1
```

```
>>> from sage.all import *
>>> C = CartanType(['A',Integer(4),Integer(2)]).dual()
>>> K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, Integer(1),
↪Integer(1))
>>> b = K(rows=[[-Integer(1)]])
>>> b.phi(Integer(0)) # indirect doctest
1
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_Bn**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: *KirillovReshetikhinGenericCrystal*

Class of Kirillov-Reshetikhin crystals $B^{n,s}$ of type $B_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(3,2)
sage: b = K(rows=[[1],[2],[3]])
sage: b.f(0)
sage: b.e(0)
[[3]]
```

(continues on next page)

(continued from previous page)

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: [b.weight() for b in K if b.is_highest_weight([1,2,3])]
[-Lambda[0] + Lambda[1], -2*Lambda[0] + 2*Lambda[3]]
sage: [b.weight() for b in K if b.is_highest_weight([0,2,3])]
[Lambda[0] - Lambda[1], -2*Lambda[1] + 2*Lambda[3]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(2))
>>> K
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(3,2)
>>> b = K(rows=[[Integer(1)],[Integer(2)],[Integer(3)]])
>>> b.f(Integer(0))
>>> b.e(Integer(0))
[[3]]

>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(2))
>>> [b.weight() for b in K if b.is_highest_weight([Integer(1),Integer(2),
↪Integer(3)])]
[-Lambda[0] + Lambda[1], -2*Lambda[0] + 2*Lambda[3]]
>>> [b.weight() for b in K if b.is_highest_weight([Integer(0),Integer(2),
↪Integer(3)])]
[Lambda[0] - Lambda[1], -2*Lambda[1] + 2*Lambda[3]]
```

Element

alias of *KR_type_BnElement*

ambient_crystal()

Return the ambient crystal $B^{n,s}$ of type $A_{2n-1}^{(2)}$ associated to the Kirillov-Reshetikhin crystal.

The ambient crystal is used to construct the zero arrows.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 3, 1]^* with (r,s)=(3,2)
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(2))
>>> K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 3, 1]^* with (r,s)=(3,2)
```

ambient_highest_weight_dict()

Return a dictionary of the classical highest weight vectors of the ambient crystal of `self` whose keys are their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K.ambient_highest_weight_dict()
```

(continues on next page)

(continued from previous page)

```
{(2,): [[1, 1]], (2, 1, 1): [[1, 1], [2], [3]], (2, 2, 2): [[1, 1], [2, 2],␣
↪[3, 3]]}

sage: K = crystals.KirillovReshetikhin(['B',3,1],3,3)
sage: K.ambient_highest_weight_dict()
{(3,): [[1, 1, 1]],
 (3, 1, 1): [[1, 1, 1], [2], [3]],
 (3, 2, 2): [[1, 1, 1], [2, 2], [3, 3]],
 (3, 3, 3): [[1, 1, 1], [2, 2, 2], [3, 3, 3]]}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(2))
>>> K.ambient_highest_weight_dict()
{(2,): [[1, 1]], (2, 1, 1): [[1, 1], [2], [3]], (2, 2, 2): [[1, 1], [2, 2],␣
↪[3, 3]]}

>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(3))
>>> K.ambient_highest_weight_dict()
{(3,): [[1, 1, 1]],
 (3, 1, 1): [[1, 1, 1], [2], [3]],
 (3, 2, 2): [[1, 1, 1], [2, 2], [3, 3]],
 (3, 3, 3): [[1, 1, 1], [2, 2, 2], [3, 3, 3]]}
```

classical_decomposition()

Return the classical crystal underlying the Kirillov-Reshetikhin crystal $B^{n,s}$ of type $B_n^{(1)}$.

It is the same as for $r < n$, given by $B^{n,s} \cong \bigoplus_\Lambda B(\Lambda)$, where Λ are weights obtained from a rectangle of width $s/2$ and height n by removing horizontal dominoes. Here we identify the fundamental weight Λ_i with a column of height i for $i < n$ and a column of width $1/2$ for $i = n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1], 3, 2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1], [1, 1, 1]]
sage: K = crystals.KirillovReshetikhin(['B',3,1], 3, 3)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[3/2, 1/2, 1/2], [3/2,␣
↪3/2, 3/2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)], Integer(3),␣
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1], [1, 1, 1]]
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)], Integer(3),␣
↪Integer(3))
>>> K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[3/2, 1/2, 1/2], [3/2,␣
↪3/2, 3/2]]
```

from_ambient_crystal()

Return a map from the ambient crystal of type $A_{2n-1}^{(2)}$ to the Kirillov-Reshetikhin crystal `self`.

Note that this map is only well-defined on elements that are in the image under *to_ambient_crystal()*.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: [b == K.from_ambient_crystal()(K.to_ambient_crystal()(b)) for b in K]
[True, True, True, True, True, True, True, True]
sage: b = K.ambient_crystal()(rows=[[1],[2],[-3]])
sage: K.from_ambient_crystal()(b)
[++-, []]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(1))
>>> [b == K.from_ambient_crystal()(K.to_ambient_crystal()(b)) for b in K]
[True, True, True, True, True, True, True, True]
>>> b = K.ambient_crystal()(rows=[[Integer(1)],[Integer(2)],[-Integer(3)]])
>>> K.from_ambient_crystal()(b)
[++-, []]
```

highest_weight_dict()

Return a dictionary of the classical highest weight vectors of `self` whose keys are 2 times their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K.highest_weight_dict()
{(2,): [[1]], (2, 2, 2): [[1], [2], [3]]}
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,3)
sage: K.highest_weight_dict()
{(3, 1, 1): [+++, [[1]]], (3, 3, 3): [+++, [[1], [2], [3]]]}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(2))
>>> K.highest_weight_dict()
{(2,): [[1]], (2, 2, 2): [[1], [2], [3]]}
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(3))
>>> K.highest_weight_dict()
{(3, 1, 1): [+++, [[1]]], (3, 3, 3): [+++, [[1], [2], [3]]]}
```

similarity_factor()

Set the similarity factor used to map to the ambient crystal.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K.similarity_factor()
{1: 2, 2: 2, 3: 1}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(2))
>>> K.similarity_factor()
{1: 2, 2: 2, 3: 1}
```

to_ambient_crystal()

Return a map from `self` to the ambient crystal of type $A_{2n-1}^{(2)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: [K.to_ambient_crystal()(b) for b in K]
[[[1], [2], [3]], [[1], [2], [-3]], [[1], [3], [-2]], [[2], [3], [-1]], [[1],␣
↪[-3], [-2]],
[[2], [-3], [-1]], [[3], [-2], [-1]], [[-3], [-2], [-1]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(1))
>>> [K.to_ambient_crystal()(b) for b in K]
[[[1], [2], [3]], [[1], [2], [-3]], [[1], [3], [-2]], [[2], [3], [-1]], [[1],␣
↪[-3], [-2]],
[[2], [-3], [-1]], [[3], [-2], [-1]], [[-3], [-2], [-1]]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_BnElement**

Bases: *KirillovReshetikhinGenericCrystalElement*

Class for the elements in the Kirillov-Reshetikhin crystals $B^{n,s}$ of type $B_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B', 3, 1], 3, 2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Bn_with_category.
↪element_class'>
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B', Integer(3), Integer(1)], Integer(3),␣
↪Integer(2))
>>> type(K.module_generators[Integer(0)])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Bn_with_category.
↪element_class'>
```

e0()

Return e_0 on `self` by mapping `self` to the ambient crystal, calculating e_0 there and pulling the element back.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: b = K.module_generators[0]
sage: b.e(0) # indirect doctest
[--+, []]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(3),
↪Integer(1))
>>> b = K.module_generators[Integer(0)]
>>> b.e(Integer(0)) # indirect doctest
[--+, []]
```

epsilon0()

Calculate ε_0 of `self` by mapping the element to the ambient crystal and calculating ε_0 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B', 3, 1], 3, 1)
sage: b = K.module_generators[0]
sage: b.epsilon(0) # indirect doctest
1
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B', Integer(3), Integer(1)],␣
↪Integer(3), Integer(1))
>>> b = K.module_generators[Integer(0)]
>>> b.epsilon(Integer(0)) # indirect doctest
1
```

f0()

Return f_0 on `self` by mapping `self` to the ambient crystal, calculating f_0 there and pulling the element back.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B', 3, 1], 3, 1)
sage: b = K.module_generators[0]
sage: b.f(0) # indirect doctest
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B', Integer(3), Integer(1)],␣
↪Integer(3), Integer(1))
>>> b = K.module_generators[Integer(0)]
>>> b.f(Integer(0)) # indirect doctest
```

phi0()

Calculate φ_0 of `self` by mapping the element to the ambient crystal and calculating φ_0 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B', 3, 1], 3, 1)
sage: b = K.module_generators[0]
sage: b.phi(0) # indirect doctest
0
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['B', Integer(3), Integer(1)],␣
↪Integer(3), Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> b = K.module_generators[Integer(0)]
>>> b.phi(Integer(0)) # indirect doctest
0
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_C**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: *KirillovReshetikhinGenericCrystal*

Class of Kirillov-Reshetikhin crystals $B^{r,s}$ of type $C_n^{(1)}$ for $r < n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: K
Kirillov-Reshetikhin crystal of type ['C', 2, 1] with (r,s)=(1,2)
sage: b = K(rows=[])
sage: b.f(0)
[[1, 1]]
sage: b.e(0)
[[-1, -1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(2),Integer(1)], Integer(1),
↪Integer(2))
>>> K
Kirillov-Reshetikhin crystal of type ['C', 2, 1] with (r,s)=(1,2)
>>> b = K(rows=[])
>>> b.f(Integer(0))
[[1, 1]]
>>> b.e(Integer(0))
[[-1, -1]]
```

Element

alias of *KR_type_CElement*

ambient_crystal()

Return the ambient crystal $B^{r,s}$ of type $A_{2n+1}^{(2)}$ associated to the Kirillov-Reshetikhin crystal of type $C_n^{(1)}$.

This ambient crystal is used to construct the zero arrows.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,3)
sage: K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 4, 1]^* with (r,s)=(2,3)
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)], Integer(2),
↪Integer(3))
>>> K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 4, 1]^* with (r,s)=(2,3)
```

ambient_dict_pm_diagrams()

Return a dictionary of all self-dual $\pm$ diagrams for the ambient crystal whose keys are their inner shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0]], [2]: [[0, 0], [2]]}
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0, 0], [0]],
 [2]: [[0, 0], [1, 1], [0]],
 [2, 2]: [[0, 0], [0, 0], [2]]}
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,3)
sage: K.ambient_dict_pm_diagrams()
{[1, 1]: [[1, 1], [0, 0], [1]],
 [3, 1]: [[0, 0], [1, 1], [1]],
 [3, 3]: [[0, 0], [0, 0], [3]]}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(2),Integer(1)], Integer(1),
↪Integer(2))
>>> K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0]], [2]: [[0, 0], [2]]}
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0, 0], [0]],
 [2]: [[0, 0], [1, 1], [0]],
 [2, 2]: [[0, 0], [0, 0], [2]]}
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)], Integer(2),
↪Integer(3))
>>> K.ambient_dict_pm_diagrams()
{[1, 1]: [[1, 1], [0, 0], [1]],
 [3, 1]: [[0, 0], [1, 1], [1]],
 [3, 3]: [[0, 0], [0, 0], [3]]}
```

ambient_highest_weight_dict()

Return a dictionary of all $\{2, \ldots, n+1\}$-highest weight vectors in the ambient crystal.

The key is the inner shape of their corresponding $\pm$ diagram, or equivalently, their $\{2, \ldots, n+1\}$ weight.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: K.ambient_highest_weight_dict()
{[]: [[2], [-2]], [2]: [[1, 2], [2, -1]], [2, 2]: [[2, 2], [3, 3]]}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.ambient_highest_weight_dict()
{[]: [[2], [-2]], [2]: [[1, 2], [2, -1]], [2, 2]: [[2, 2], [3, 3]]}
```

classical_decomposition()

Return the classical crystal underlying the Kirillov-Reshetikhin crystal of type $C_n^{(1)}$.

It is given by $B^{r,s} \cong \bigoplus_\Lambda B(\Lambda)$, where Λ are weights obtained from a rectangle of width s and height r by removing horizontal dominoes. Here we identify the fundamental weight Λ_i with a column of height i.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 3] and shape(s) [[], [2], [2, 2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['C', 3] and shape(s) [[], [2], [2, 2]]
```

from_ambient_crystal()

Return a map from the ambient crystal of type $A_{2n+1}^{(2)}$ to the Kirillov-Reshetikhin crystal of type $C_n^{(1)}$.

Note that this map is only well-defined on type $C_n^{(1)}$ elements that are in the image under `to_ambient_crystal()`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: b = K.ambient_crystal()(rows=[[2,2],[3,3]])
sage: K.from_ambient_crystal()(b)
[[1, 1], [2, 2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> b = K.ambient_crystal()(rows=[[Integer(2),Integer(2)],[Integer(3),
↪Integer(3)]])
>>> K.from_ambient_crystal()(b)
[[1, 1], [2, 2]]
```

highest_weight_dict()

Return a dictionary of the classical highest weight vectors of `self` whose keys are their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: K.highest_weight_dict()
{[]: [], [2]: [[1, 1]], [2, 2]: [[1, 1], [2, 2]]}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.highest_weight_dict()
{[]: [], [2]: [[1, 1]], [2, 2]: [[1, 1], [2, 2]]}
```

to_ambient_crystal()

Return a map from the Kirillov-Reshetikhin crystal of type $C_n^{(1)}$ to the ambient crystal of type $A_{2n+1}^{(2)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: b=K(rows=[[1,1]])
sage: K.to_ambient_crystal()(b)
[[1, 2], [2, -1]]
sage: b=K(rows=[])
sage: K.to_ambient_crystal()(b)
[[2], [-2]]
sage: K.to_ambient_crystal()(b).parent()
Kirillov-Reshetikhin crystal of type ['B', 4, 1]^* with (r,s)=(2,2)
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> b=K(rows=[[Integer(1),Integer(1)]])
>>> K.to_ambient_crystal()(b)
[[1, 2], [2, -1]]
>>> b=K(rows=[])
>>> K.to_ambient_crystal()(b)
[[2], [-2]]
>>> K.to_ambient_crystal()(b).parent()
Kirillov-Reshetikhin crystal of type ['B', 4, 1]^* with (r,s)=(2,2)
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_CElement**

Bases: *KirillovReshetikhinGenericCrystalElement*

Class for the elements in the Kirillov-Reshetikhin crystals $B^{r,s}$ of type $C_n^{(1)}$ for $r < n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C', 3, 1], 1, 2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_C_with_category.
↪element_class'>
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)], Integer(1),␣
↪Integer(2))
>>> type(K.module_generators[Integer(0)])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_C_with_category.
↪element_class'>
```

e0()

Return e_0 on `self` by mapping `self` to the ambient crystal, calculating $e_1 e_0$ there and pulling the element back.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C', 3, 1], 1, 2)
sage: b = K(rows=[])
sage: b.e(0) # indirect doctest
[[-1, -1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)],␣
↪Integer(1), Integer(2))
>>> b = K(rows=[])
>>> b.e(Integer(0)) # indirect doctest
[[-1, -1]]
```

epsilon0()

Calculate ε_0 of `self` by mapping the element to the ambient crystal and calculating ε_1 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: b=K(rows=[[1,1]])
sage: b.epsilon(0) # indirect doctest
2
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(2),Integer(1)], Integer(1),
↪Integer(2))
>>> b=K(rows=[[Integer(1),Integer(1)]])
>>> b.epsilon(Integer(0)) # indirect doctest
2
```

f0()

Return f_0 on `self` by mapping `self` to the ambient crystal, calculating $f_1 f_0$ there and pulling the element back.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C', 3, 1], 1, 2)
sage: b = K(rows=[])
sage: b.f(0) # indirect doctest
[[1, 1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)],␣
↪Integer(1), Integer(2))
>>> b = K(rows=[])
>>> b.f(Integer(0)) # indirect doctest
[[1, 1]]
```

phi0()

Calculate φ_0 of `self` by mapping the element to the ambient crystal and calculating φ_1 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: b=K(rows=[[-1,-1]])
sage: b.phi(0) # indirect doctest
2
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(2),Integer(1)], Integer(1),
↪Integer(2))
>>> b=K(rows=[[-Integer(1),-Integer(1)]])
>>> b.phi(Integer(0)) # indirect doctest
2
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_Cn**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: *KirillovReshetikhinGenericCrystal*

Class of Kirillov-Reshetikhin crystals $B^{n,s}$ of type $C_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,1)
sage: [[b,b.f(0)] for b in K]
[[[[1], [2], [3]], None], [[[1], [2], [-3]], None],
 [[[1], [3], [-3]], None], [[[2], [3], [-3]], None],
 [[[1], [3], [-2]], None], [[[2], [3], [-2]], None],
 [[[2], [3], [-1]], [[1], [2], [3]]], [[[1], [-3], [-2]], None],
 [[[2], [-3], [-2]], None], [[[2], [-3], [-1]], [[1], [2], [-3]]],
 [[[3], [-3], [-2]], None], [[[3], [-3], [-1]], [[1], [3], [-3]]],
 [[[3], [-2], [-1]], [[1], [3], [-2]]],
 [[[-3], [-2], [-1]], [[1], [-3], [-2]]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)],Integer(3),
↪Integer(1))
>>> [[b,b.f(Integer(0))] for b in K]
[[[[1], [2], [3]], None], [[[1], [2], [-3]], None],
 [[[1], [3], [-3]], None], [[[2], [3], [-3]], None],
 [[[1], [3], [-2]], None], [[[2], [3], [-2]], None],
 [[[2], [3], [-1]], [[1], [2], [3]]], [[[1], [-3], [-2]], None],
 [[[2], [-3], [-2]], None], [[[2], [-3], [-1]], [[1], [2], [-3]]],
 [[[3], [-3], [-2]], None], [[[3], [-3], [-1]], [[1], [3], [-3]]],
 [[[3], [-2], [-1]], [[1], [3], [-2]]],
 [[[-3], [-2], [-1]], [[1], [-3], [-2]]]]
```

Element

alias of *KR_type_CnElement*

classical_decomposition()

Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal $B^{n,s}$ of type $C_n^{(1)}$.

The classical decomposition is given by $B^{n,s} \cong B(s\Lambda_n)$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 3] and shape(s) [[2, 2, 2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)],Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['C', 3] and shape(s) [[2, 2, 2]]
```

from_highest_weight_vector_to_pm_diagram(*b*)

This gives the bijection between an element b in the classical decomposition of the KR crystal that is $2, 3, .., n$-highest weight and $\pm$ diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,2)
sage: T = K.classical_decomposition()
sage: b = T(rows=[[2, 2], [3, 3], [-3, -1]])
sage: pm = K.from_highest_weight_vector_to_pm_diagram(b); pm
[[0, 0], [1, 0], [0, 1], [0]]
sage: pm.pp()
.  .
.  +
-  -

sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3]) ]
sage: all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_
↪vector_to_pm_diagram(b)) == b for b in hw)
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)],Integer(3),
↪Integer(2))
>>> T = K.classical_decomposition()
>>> b = T(rows=[[Integer(2), Integer(2)], [Integer(3), Integer(3)], [-
↪Integer(3), -Integer(1)]])
>>> pm = K.from_highest_weight_vector_to_pm_diagram(b); pm
[[0, 0], [1, 0], [0, 1], [0]]
>>> pm.pp()
.  .
.  +
-  -

>>> hw = [ b for b in T if all(b.epsilon(i)==Integer(0) for i in [Integer(2),
↪Integer(3)]) ]
>>> all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_
↪vector_to_pm_diagram(b)) == b for b in hw)
True
```

from_pm_diagram_to_highest_weight_vector(*pm*)

This gives the bijection between a $\pm$ diagram and an element b in the classical decomposition of the KR crystal that is $\{2, 3, .., n\}$-highest weight.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,2)
sage: pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[0, 0], [1,␣
↪0], [0, 1], [0]])
```

(continues on next page)

(continued from previous page)

```
sage: K.from_pm_diagram_to_highest_weight_vector(pm)
[[2, 2], [3, 3], [-3, -1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)],Integer(3),
↪Integer(2))
>>> pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[Integer(0),
↪Integer(0)], [Integer(1), Integer(0)], [Integer(0), Integer(1)],
↪[Integer(0)]])
>>> K.from_pm_diagram_to_highest_weight_vector(pm)
[[2, 2], [3, 3], [-3, -1]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_CnElement**

Bases: *KirillovReshetikhinGenericCrystalElement*

Class for the elements in the Kirillov-Reshetikhin crystals $B^{n,s}$ of type $C_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C', 3, 1], 3, 2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Cn_with_category.
↪element_class'>
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)], Integer(3),
↪Integer(2))
>>> type(K.module_generators[Integer(0)])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Cn_with_category.
↪element_class'>
```

e0()

Return e_0 on `self` by going to the $\pm$-diagram corresponding to the $\{2, ..., n\}$-highest weight vector in the component of `self`, then applying [Definition 6.1, 4], and pulling back from $\pm$-diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C', 3, 1], 3, 2)
sage: b = K.module_generators[0]
sage: b.e(0) # indirect doctest
[[1, 2], [2, 3], [3, -1]]
sage: b = K(rows=[[1,2],[2,3],[3,-1]])
sage: b.e(0)
[[2, 2], [3, 3], [-1, -1]]
sage: b=K(rows=[[1, -3], [3, -2], [-3, -1]])
sage: b.e(0)
[[3, -3], [-3, -2], [-1, -1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)],
↪Integer(3), Integer(2))
>>> b = K.module_generators[Integer(0)]
```

(continues on next page)

(continued from previous page)

```
>>> b.e(Integer(0)) # indirect doctest
[[1, 2], [2, 3], [3, -1]]
>>> b = K(rows=[[Integer(1),Integer(2)],[Integer(2),Integer(3)],[Integer(3),-
↪Integer(1)]])
>>> b.e(Integer(0))
[[2, 2], [3, 3], [-1, -1]]
>>> b=K(rows=[[Integer(1), -Integer(3)], [Integer(3), -Integer(2)], [-
↪Integer(3), -Integer(1)]])
>>> b.e(Integer(0))
[[3, -3], [-3, -2], [-1, -1]]
```

epsilon0()

Calculate ε_0 of `self` using Lemma 6.1 of [4].

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C', 3, 1], 3, 1)
sage: b = K.module_generators[0]
sage: b.epsilon(0) # indirect doctest
1
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)],␣
↪Integer(3), Integer(1))
>>> b = K.module_generators[Integer(0)]
>>> b.epsilon(Integer(0)) # indirect doctest
1
```

f0()

Return e_0 on `self` by going to the $\pm$-diagram corresponding to the $\{2, ..., n\}$-highest weight vector in the component of `self`, then applying [Definition 6.1, 4], and pulling back from $\pm$-diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,1)
sage: b = K.module_generators[0]
sage: b.f(0) # indirect doctest
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)],Integer(3),
↪Integer(1))
>>> b = K.module_generators[Integer(0)]
>>> b.f(Integer(0)) # indirect doctest
```

phi0()

Calculate φ_0 of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C', 3, 1], 3, 1)
sage: b = K.module_generators[0]
sage: b.phi(0) # indirect doctest
0
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)],␣
↪Integer(3), Integer(1))
>>> b = K.module_generators[Integer(0)]
>>> b.phi(Integer(0)) # indirect doctest
0
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_D_tri1**(*ct*, *s*)

Bases: *KirillovReshetikhinGenericCrystal*

Class of Kirillov-Reshetikhin crystals $B^{1,s}$ of type $D_4^{(3)}$.

The crystal structure was defined in Section 4 of [KMOY2007] using the coordinate representation.

class Element

Bases: *KirillovReshetikhinGenericCrystalElement*

coordinates()

Return `self` as coordinates.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 3)
sage: all(K.from_coordinates(x.coordinates()) == x for x in K)
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],␣
↪Integer(1), Integer(3))
>>> all(K.from_coordinates(x.coordinates()) == x for x in K)
True
```

e0()

Return the action of e_0 on `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1,1)
sage: [x.e0() for x in K]
[[[-1]], [], [[-3]], [[-2]], None, None, None, None]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],␣
↪Integer(1),Integer(1))
>>> [x.e0() for x in K]
[[[-1]], [], [[-3]], [[-2]], None, None, None, None]
```

epsilon0()

Return ε_0 of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 5)
sage: [mg.epsilon0() for mg in K.module_generators]
[5, 6, 7, 8, 9, 10]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],␣
↪Integer(1), Integer(5))
>>> [mg.epsilon0() for mg in K.module_generators]
[5, 6, 7, 8, 9, 10]
```

f0()

Return the action of f_0 on `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1,1)
sage: [x.f0() for x in K]
[[[1]], None, None, None, None, [[2]], [[3]], []]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],␣
↪Integer(1),Integer(1))
>>> [x.f0() for x in K]
[[[1]], None, None, None, None, [[2]], [[3]], []]
```

phi0()

Return φ_0 of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 5)
sage: [mg.phi0() for mg in K.module_generators]
[5, 4, 3, 2, 1, 0]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],␣
↪Integer(1), Integer(5))
>>> [mg.phi0() for mg in K.module_generators]
[5, 4, 3, 2, 1, 0]
```

classical_decomposition()

Return the classical decomposition of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 5)
sage: K.classical_decomposition()
The crystal of tableaux of type ['G', 2]
 and shape(s) [[], [1], [2], [3], [4], [5]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)], Integer(1),␣
↪Integer(5))
>>> K.classical_decomposition()
The crystal of tableaux of type ['G', 2]
 and shape(s) [[], [1], [2], [3], [4], [5]]
```

from_coordinates(*coords*)

Return an element of `self` from the coordinates `coords`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 5)
sage: K.from_coordinates((0, 2, 3, 1, 0, 1))
[[2, 2, 3, 0, -1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)], Integer(1),␣
↪Integer(5))
>>> K.from_coordinates((Integer(0), Integer(2), Integer(3), Integer(1),␣
↪Integer(0), Integer(1)))
[[2, 2, 3, 0, -1]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_Dn_twisted**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: *KirillovReshetikhinGenericCrystal*

Class of Kirillov-Reshetikhin crystals $B^{n,s}$ of type $D_{n+1}^{(2)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: [[b,b.f(0)] for b in K]
[[[+++, []], None], [[++-, []], None], [[+-+, []], None], [[-++, []],
[+++, []]], [[+--, []], None], [[-+-, []], [++-, []]], [[--+, []], [+-+, []]],
[[---, []], [+--, []]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)],Integer(3),
↪Integer(1))
>>> [[b,b.f(Integer(0))] for b in K]
[[[+++, []], None], [[++-, []], None], [[+-+, []], None], [[-++, []],
[+++, []]], [[+--, []], None], [[-+-, []], [++-, []]], [[--+, []], [+-+, []]],
[[---, []], [+--, []]]]
```

Element

alias of *KR_type_Dn_twistedElement*

classical_decomposition()

Return the classical crystal underlying the Kirillov-Reshetikhin crystal $B^{n,s}$ of type $D_{n+1}^{(2)}$.

The classical decomposition is given by $B^{n,s} \cong B(s\Lambda_n)$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1/2, 1/2, 1/2]]
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1, 1, 1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)],Integer(3),
↪Integer(1))
>>> K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1/2, 1/2, 1/2]]
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)],Integer(3),
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1, 1, 1]]
```

from_highest_weight_vector_to_pm_diagram(b)

This gives the bijection between an element b in the classical decomposition of the KR crystal that is $\{2, 3, \ldots, n\}$-highest weight and $\pm$ diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: T = K.classical_decomposition()
sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3]) ]
sage: [K.from_highest_weight_vector_to_pm_diagram(b) for b in hw]
[[[0, 0], [0, 0], [1, 0], [0]], [[0, 0], [0, 0], [0, 1], [0]]]

sage: K = crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: T = K.classical_decomposition()
sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3]) ]
sage: [K.from_highest_weight_vector_to_pm_diagram(b) for b in hw]
[[[0, 0], [0, 0], [2, 0], [0]], [[0, 0], [0, 0], [0, 0], [2]],
 [[0, 0], [2, 0], [0, 0], [0]], [[0, 0], [0, 0], [0, 2], [0]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)],Integer(3),
↪Integer(1))
>>> T = K.classical_decomposition()
>>> hw = [ b for b in T if all(b.epsilon(i)==Integer(0) for i in [Integer(2),
↪Integer(3)]) ]
>>> [K.from_highest_weight_vector_to_pm_diagram(b) for b in hw]
[[[0, 0], [0, 0], [1, 0], [0]], [[0, 0], [0, 0], [0, 1], [0]]]

>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)],Integer(3),
↪Integer(2))
>>> T = K.classical_decomposition()
>>> hw = [ b for b in T if all(b.epsilon(i)==Integer(0) for i in [Integer(2),
↪Integer(3)]) ]
>>> [K.from_highest_weight_vector_to_pm_diagram(b) for b in hw]
[[[0, 0], [0, 0], [2, 0], [0]], [[0, 0], [0, 0], [0, 0], [2]],
 [[0, 0], [2, 0], [0, 0], [0]], [[0, 0], [0, 0], [0, 2], [0]]]
```

Note that, since the classical decomposition of this crystal is of type B_n, there can be (at most one) entry 0 in the $\{2, 3, \ldots, n\}$-highest weight elements at height n. In the following implementation this is realized as an empty column of height n since this uniquely specifies the existence of the 0.

EXAMPLES:

```
sage: b = hw[1]
sage: pm = K.from_highest_weight_vector_to_pm_diagram(b)
sage: pm.pp()
.  .
.  .
.  .
```

```
>>> from sage.all import *
>>> b = hw[Integer(1)]
>>> pm = K.from_highest_weight_vector_to_pm_diagram(b)
>>> pm.pp()
.  .
.  .
.  .
```

from_pm_diagram_to_highest_weight_vector(*pm*)

This gives the bijection between a $\pm$ diagram and an element `b` in the classical decomposition of the KR crystal that is $\{2, 3, \ldots, n\}$-highest weight.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[0, 0], [0,␣
↪0], [0, 0], [2]])
sage: K.from_pm_diagram_to_highest_weight_vector(pm)
[[2], [3], [0]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)],Integer(3),
↪Integer(2))
>>> pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[Integer(0),␣
↪Integer(0)], [Integer(0), Integer(0)], [Integer(0), Integer(0)],␣
↪[Integer(2)]])
>>> K.from_pm_diagram_to_highest_weight_vector(pm)
[[2], [3], [0]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_Dn_twistedElement**

Bases: *KirillovReshetikhinGenericCrystalElement*

Class for the elements in the Kirillov-Reshetikhin crystals $B^{n,s}$ of type $D_{n+1}^{(2)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D', 4, 2], 3, 2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Dn_twisted_with_
↪category.element_class'>
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)], Integer(3),␣
↪Integer(2))
>>> type(K.module_generators[Integer(0)])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Dn_twisted_with_
↪category.element_class'>
```

e0()

Return e_0 on `self` by going to the $\pm$-diagram corresponding to the $\{2, \ldots, n\}$-highest weight vector in the component of `self`, then applying [Definition 6.2, 4], and pulling back from $\pm$-diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D', 4, 2], 3, 3)
sage: b = K.module_generators[0]
sage: b.e(0) # indirect doctest
[+++, [[2], [3], [0]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)],␣
↪Integer(3), Integer(3))
>>> b = K.module_generators[Integer(0)]
>>> b.e(Integer(0)) # indirect doctest
[+++, [[2], [3], [0]]]
```

epsilon0()

Calculate ε_0 of `self` using Lemma 6.2 of [4].

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D', 4, 2], 3, 1)
sage: b = K.module_generators[0]
sage: b.epsilon(0) # indirect doctest
1
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)],␣
↪Integer(3), Integer(1))
>>> b = K.module_generators[Integer(0)]
>>> b.epsilon(Integer(0)) # indirect doctest
1
```

f0()

Return e_0 on `self` by going to the $\pm$-diagram corresponding to the $\{2, \ldots, n\}$-highest weight vector in the component of `self`, then applying [Definition 6.2, 4], and pulling back from $\pm$-diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: b = K.module_generators[0]
sage: b.f(0) # indirect doctest
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)],Integer(3),
↪Integer(2))
>>> b = K.module_generators[Integer(0)]
>>> b.f(Integer(0)) # indirect doctest
```

phi0()

Calculate φ_0 of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: b = K.module_generators[0]
sage: b.phi(0) # indirect doctest
0
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)],Integer(3),
↪Integer(1))
>>> b = K.module_generators[Integer(0)]
>>> b.phi(Integer(0)) # indirect doctest
0
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_E6**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinCrystalFromPromotion*

Class of Kirillov-Reshetikhin crystals of type $E_6^{(1)}$ for $r = 1, 2, 6$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.module_generator().e(0)
[]
sage: K.module_generator().e(0).f(0)
[[(2, -1), (1,)]]
sage: K = crystals.KirillovReshetikhin(['E',6,1], 1,1)
sage: b = K.module_generator()
sage: b
[(1,)]
sage: b.e(0)
[(-2, 1)]
sage: b = next(t for t in K if t.epsilon(1) == 1 and t.phi(3) == 1 and t.phi(2)␣
↪== 0 and t.epsilon(2) == 0)
sage: b
[(-1, 3)]
sage: b.e(0)
[(-1, -2, 3)]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(2),
↪Integer(1))
>>> K.module_generator().e(Integer(0))
[]
>>> K.module_generator().e(Integer(0)).f(Integer(0))
[[(2, -1), (1,)]]
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)], Integer(1),
↪Integer(1))
>>> b = K.module_generator()
>>> b
[(1,)]
>>> b.e(Integer(0))
[(-2, 1)]
>>> b = next(t for t in K if t.epsilon(Integer(1)) == Integer(1) and t.
↪phi(Integer(3)) == Integer(1) and t.phi(Integer(2)) == Integer(0) and t.
```

(continues on next page)

(continued from previous page)

```
↪epsilon(Integer(2)) == Integer(0))
>>> b
[(-1, 3)]
>>> b.e(Integer(0))
[(-1, -2, 3)]
```

The elements of the Kirillov-Reshetikhin crystals can be constructed from a classical crystal element using `retract()`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: La = K.cartan_type().classical().root_system().weight_lattice().fundamental_
↪weights()
sage: H = crystals.HighestWeight(La[2])
sage: t = H.module_generator()
sage: t
[[(2, -1), (1,)]]
sage: type(K.retract(t))
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_E6_with_category.
↪element_class'>
sage: K.retract(t).e(0)
[]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(2),
↪Integer(1))
>>> La = K.cartan_type().classical().root_system().weight_lattice().fundamental_
↪weights()
>>> H = crystals.HighestWeight(La[Integer(2)])
>>> t = H.module_generator()
>>> t
[[(2, -1), (1,)]]
>>> type(K.retract(t))
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_E6_with_category.
↪element_class'>
>>> K.retract(t).e(Integer(0))
[]
```

affine_weight(*b*)

Return the affine level zero weight corresponding to the element `b` of the classical crystal underlying `self`.

For the coefficients to calculate the level, see Table Aff 1 in [Ka1990].

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: [K.affine_weight(x.lift()) for x in K
....:  if all(x.epsilon(i) == 0 for i in [2,3,4,5])]
[(0, 0, 0, 0, 0, 0, 0),
 (-2, 0, 1, 0, 0, 0, 0),
 (-1, -1, 0, 0, 0, 1, 0),
 (0, 0, 0, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 0, 0, 0, 0, 1, -2),
 (0, -1, 1, 0, 0, 0, -1),
 (-1, 0, 0, 1, 0, 0, -1),
 (-1, -1, 0, 0, 1, 0, -1),
 (0, 0, 0, 0, 0, 0, 0),
 (0, -2, 0, 1, 0, 0, 0)]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(2),
↪Integer(1))
>>> [K.affine_weight(x.lift()) for x in K
...  if all(x.epsilon(i) == Integer(0) for i in [Integer(2),Integer(3),
↪Integer(4),Integer(5)])]
[(0, 0, 0, 0, 0, 0, 0),
 (-2, 0, 1, 0, 0, 0, 0),
 (-1, -1, 0, 0, 0, 1, 0),
 (0, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 1, -2),
 (0, -1, 1, 0, 0, 0, -1),
 (-1, 0, 0, 1, 0, 0, -1),
 (-1, -1, 0, 0, 1, 0, -1),
 (0, 0, 0, 0, 0, 0, 0),
 (0, -2, 0, 1, 0, 0, 0)]
```

automorphism_on_affine_weight(*weight*)

Act with the Dynkin diagram automorphism on affine weights as outputted by the `affine_weight` method.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: sorted([x[0], K.automorphism_on_affine_weight(x[0])]
....:  for x in K.highest_weight_dict().values())
[[(-2, 0, 1, 0, 0, 0, 0), (0, -2, 0, 1, 0, 0, 0)],
 [(-1, 0, 0, 1, 0, 0, -1), (-1, -1, 0, 0, 0, 1, 0)],
 [(0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0)],
 [(0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0)],
 [(0, 0, 0, 0, 0, 1, -2), (-2, 0, 1, 0, 0, 0, 0)]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(2),
↪Integer(1))
>>> sorted([x[Integer(0)], K.automorphism_on_affine_weight(x[Integer(0)])]
...  for x in K.highest_weight_dict().values())
[[(-2, 0, 1, 0, 0, 0, 0), (0, -2, 0, 1, 0, 0, 0)],
 [(-1, 0, 0, 1, 0, 0, -1), (-1, -1, 0, 0, 0, 1, 0)],
 [(0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0)],
 [(0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0)],
 [(0, 0, 0, 0, 0, 1, -2), (-2, 0, 1, 0, 0, 0, 0)]]
```

classical_decomposition()

Specifies the classical crystal underlying the KR crystal of type $E_6^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2,2)
sage: K.classical_decomposition()
Direct sum of the crystals Family
 (Finite dimensional highest weight crystal of type ['E', 6] and highest␣
↪weight 0,
  Finite dimensional highest weight crystal of type ['E', 6] and highest␣
↪weight Lambda[2],
  Finite dimensional highest weight crystal of type ['E', 6] and highest␣
↪weight 2*Lambda[2])
sage: K = crystals.KirillovReshetikhin(['E',6,1], 1,2)
sage: K.classical_decomposition()
Direct sum of the crystals Family
 (Finite dimensional highest weight crystal of type ['E', 6] and highest␣
↪weight 2*Lambda[1],)
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)], Integer(2),
↪Integer(2))
>>> K.classical_decomposition()
Direct sum of the crystals Family
 (Finite dimensional highest weight crystal of type ['E', 6] and highest␣
↪weight 0,
  Finite dimensional highest weight crystal of type ['E', 6] and highest␣
↪weight Lambda[2],
  Finite dimensional highest weight crystal of type ['E', 6] and highest␣
↪weight 2*Lambda[2])
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)], Integer(1),
↪Integer(2))
>>> K.classical_decomposition()
Direct sum of the crystals Family
 (Finite dimensional highest weight crystal of type ['E', 6] and highest␣
↪weight 2*Lambda[1],)
```

dynkin_diagram_automorphism(*i*)

Specifies the Dynkin diagram automorphism underlying the promotion action on the crystal elements.

Here we use the Dynkin diagram automorphism of order 3 which maps node 0 to node 1.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: [K.dynkin_diagram_automorphism(i) for i in K.index_set()]
[1, 6, 3, 5, 4, 2, 0]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(2),
↪Integer(1))
>>> [K.dynkin_diagram_automorphism(i) for i in K.index_set()]
[1, 6, 3, 5, 4, 2, 0]
```

highest_weight_dict()

Return a dictionary between $\{1, 2, 3, 4, 5\}$-highest weight elements, and a tuple of affine weights and its classical component.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: sorted(K.highest_weight_dict().items(), key=str)
[([[(2, -1), (1,)]], ((-2, 0, 1, 0, 0, 0, 0), 1)),
 ([[(3, -1, -6), (1,)]], ((-1, 0, 0, 1, 0, 0, -1), 1)),
 ([[(5, -2, -6), (-6, 2)]], ((0, 0, 0, 0, 0, 1, -2), 1)),
 ([[(6, -2), (-6, 2)]], ((0, 0, 0, 0, 0, 0, 0), 1)),
 ([], ((0, 0, 0, 0, 0, 0, 0), 0))]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(2),
↪Integer(1))
>>> sorted(K.highest_weight_dict().items(), key=str)
[([[(2, -1), (1,)]], ((-2, 0, 1, 0, 0, 0, 0), 1)),
 ([[(3, -1, -6), (1,)]], ((-1, 0, 0, 1, 0, 0, -1), 1)),
 ([[(5, -2, -6), (-6, 2)]], ((0, 0, 0, 0, 0, 1, -2), 1)),
 ([[(6, -2), (-6, 2)]], ((0, 0, 0, 0, 0, 0, 0), 1)),
 ([], ((0, 0, 0, 0, 0, 0, 0), 0))]
```

highest_weight_dict_inv()

Return a dictionary between a tuple of affine weights and a classical component, and $\{2, 3, 4, 5, 6\}$-highest weight elements.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.highest_weight_dict_inv()
{((-2, 0, 1, 0, 0, 0, 0), 1): [[(2, -1), (1,)]],
 ((-1, -1, 0, 0, 0, 1, 0), 1): [[(5, -3), (-1, 3)]],
 ((0, -2, 0, 1, 0, 0, 0), 1): [[(-1,), (-1, 3)]],
 ((0, 0, 0, 0, 0, 0, 0), 0): [],
 ((0, 0, 0, 0, 0, 0, 0), 1): [[(1, -3), (-1, 3)]]}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(2),
↪Integer(1))
>>> K.highest_weight_dict_inv()
{((-2, 0, 1, 0, 0, 0, 0), 1): [[(2, -1), (1,)]],
 ((-1, -1, 0, 0, 0, 1, 0), 1): [[(5, -3), (-1, 3)]],
 ((0, -2, 0, 1, 0, 0, 0), 1): [[(-1,), (-1, 3)]],
 ((0, 0, 0, 0, 0, 0, 0), 0): [],
 ((0, 0, 0, 0, 0, 0, 0), 1): [[(1, -3), (-1, 3)]]}
```

hw_auxiliary()

Return the $2, 3, 4, 5$ highest weight elements of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.hw_auxiliary()
([], [[(2, -1), (1,)]],
 [[(5, -3), (-1, 3)]],
 [[(6, -2), (-6, 2)]],
```

(continues on next page)

(continued from previous page)

```
 [[(5, -2, -6), (-6, 2)]],
 [[(-1,), (-6, 2)]],
 [[(3, -1, -6), (1,)]],
 [[(4, -3, -6), (-1, 3)]],
 [[(1, -3), (-1, 3)]],
 [[(-1,), (-1, 3)]])
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(2),
↪Integer(1))
>>> K.hw_auxiliary()
([], [[(2, -1), (1,)]],
 [[(5, -3), (-1, 3)]],
 [[(6, -2), (-6, 2)]],
 [[(5, -2, -6), (-6, 2)]],
 [[(-1,), (-6, 2)]],
 [[(3, -1, -6), (1,)]],
 [[(4, -3, -6), (-1, 3)]],
 [[(1, -3), (-1, 3)]],
 [[(-1,), (-1, 3)]])
```

promotion()

Specifies the promotion operator used to construct the affine type $E_6^{(1)}$ crystal.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2,1)
sage: promotion = K.promotion()
sage: all(promotion(promotion(promotion(b))) == b for b in K.classical_
↪decomposition())
True
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: promotion = K.promotion()
sage: all(promotion(promotion(promotion(b))) == b for b in K.classical_
↪decomposition())
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)], Integer(2),
↪Integer(1))
>>> promotion = K.promotion()
>>> all(promotion(promotion(promotion(b))) == b for b in K.classical_
↪decomposition())
True
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(1),
↪Integer(1))
>>> promotion = K.promotion()
>>> all(promotion(promotion(promotion(b))) == b for b in K.classical_
↪decomposition())
True
```

promotion_inverse()

Return the inverse promotion. Since promotion is of order 3, the inverse promotion is the same as promotion applied twice.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2,1)
sage: p = K.promotion()
sage: p_inv = K.promotion_inverse()
sage: all(p_inv(p(b)) == b for b in K.classical_decomposition())
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)], Integer(2),
↪Integer(1))
>>> p = K.promotion()
>>> p_inv = K.promotion_inverse()
>>> all(p_inv(p(b)) == b for b in K.classical_decomposition())
True
```

promotion_on_highest_weight_vectors()

Return a dictionary of the promotion map on $\{1, 2, 3, 4, 5\}$-highest weight elements to $\{2, 3, 4, 5, 6\}$-highest weight elements in `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2, 1)
sage: dic = K.promotion_on_highest_weight_vectors()
sage: sorted(dic.items(), key=str)
[([[(2, -1), (1,)]], [[(-1,), (-1, 3)]]),
 ([[(3, -1, -6), (1,)]], [[(5, -3), (-1, 3)]]),
 ([[(5, -2, -6), (-6, 2)]], [[(2, -1), (1,)]]),
 ([[(6, -2), (-6, 2)]], []),
 ([], [[(1, -3), (-1, 3)]])]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)], Integer(2),␣
↪Integer(1))
>>> dic = K.promotion_on_highest_weight_vectors()
>>> sorted(dic.items(), key=str)
[([[(2, -1), (1,)]], [[(-1,), (-1, 3)]]),
 ([[(3, -1, -6), (1,)]], [[(5, -3), (-1, 3)]]),
 ([[(5, -2, -6), (-6, 2)]], [[(2, -1), (1,)]]),
 ([[(6, -2), (-6, 2)]], []),
 ([], [[(1, -3), (-1, 3)]])]
```

promotion_on_highest_weight_vectors_function()

Return a lambda function on `x` defined by `self.promotion_on_highest_weight_vectors()[x]`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2, 1)
sage: f = K.promotion_on_highest_weight_vectors_function()
sage: f(K.module_generator().lift())
[[(-1,), (-1, 3)]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)], Integer(2),
↪Integer(1))
>>> f = K.promotion_on_highest_weight_vectors_function()
>>> f(K.module_generator().lift())
[[(-1,), (-1, 3)]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_E7**(*ct*, *r*, *s*)

Bases: *KirillovReshetikhinGenericCrystal*

The Kirillov-Reshetikhin crystal $B^{7,s}$ of type $E_7^{(1)}$.

A7_decomposition()

Return the decomposition of `self` into A_7 highest weight crystals.

The A_7 decomposition of $B^{7,s}$ is given by the parameters $m_4, m_5, m_6, m_7 \geq 0$ such that $m_4 + m_5 \leq m_7$ and $s = m_4 + m_5 + m_6 + m_7$. The corresponding A_7 highest weight crystal has highest weight $\lambda = (m_7 - m_4 - m_5)\Lambda_6 + m_5\Lambda_4 + m_6\Lambda_2$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',7,1], 7, 3)
sage: K.A7_decomposition()
The crystal of tableaux of type ['A', 7] and shape(s)
 [[3, 3, 3, 3, 3, 3], [3, 3, 2, 2, 2, 2], [3, 3, 1, 1, 1, 1], [3, 3],
  [2, 2, 2, 2, 1, 1], [2, 2, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(7),Integer(1)], Integer(7),
↪Integer(3))
>>> K.A7_decomposition()
The crystal of tableaux of type ['A', 7] and shape(s)
 [[3, 3, 3, 3, 3, 3], [3, 3, 2, 2, 2, 2], [3, 3, 1, 1, 1, 1], [3, 3],
  [2, 2, 2, 2, 1, 1], [2, 2, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1]]
```

class Element

Bases: *KirillovReshetikhinGenericCrystalElement*

e0()

Return the action of e_0 on `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',7,1], 7, 2)
sage: mg = K.module_generator()
sage: mg.e0()
[[(7,), (-1, 7)]]
sage: mg.e0().e0()
[[(-1, 7), (-1, 7)]]
sage: mg.e_string([0,0,0]) is None
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(7),Integer(1)],
```

(continues on next page)

(continued from previous page)

```
↪Integer(7), Integer(2))
>>> mg = K.module_generator()
>>> mg.e0()
[[(7,), (-1, 7)]]
>>> mg.e0().e0()
[[(-1, 7), (-1, 7)]]
>>> mg.e_string([Integer(0),Integer(0),Integer(0)]) is None
True
```

f0()

Return the action of f_0 on `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',7,1], 7, 2)
sage: mg = K.module_generator()
sage: x = mg.f_string([7,6,5,4,3,2,4,5,6,1,3,4,5,2,4,3,1])
sage: x.f0()
[[(7,), (7,)]]
sage: mg.f0() is None
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(7),Integer(1)],␣
↪Integer(7), Integer(2))
>>> mg = K.module_generator()
>>> x = mg.f_string([Integer(7),Integer(6),Integer(5),Integer(4),
↪Integer(3),Integer(2),Integer(4),Integer(5),Integer(6),Integer(1),
↪Integer(3),Integer(4),Integer(5),Integer(2),Integer(4),Integer(3),
↪Integer(1)])
>>> x.f0()
[[(7,), (7,)]]
>>> mg.f0() is None
True
```

classical_decomposition()

Return the classical decomposition of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',7,1], 7, 4)
sage: K.classical_decomposition()
The crystal of tableaux of type ['E', 7] and shape(s) [[4]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(7),Integer(1)], Integer(7),␣
↪Integer(4))
>>> K.classical_decomposition()
The crystal of tableaux of type ['E', 7] and shape(s) [[4]]
```

from_A7_crystal()

Return the inclusion of the KR crystal $B^{7,s}$ of type $E_7^{(1)}$ into type A_7 highest weight crystals.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',7,1], 7, 2)
sage: K.from_A7_crystal()
['A', 6] -> ['E', 7, 1] Virtual Crystal morphism:
  From: The crystal of tableaux of type ['A', 7] and shape(s)
    [[2, 2, 2, 2, 2, 2], [2, 2, 1, 1, 1, 1], [2, 2], [1, 1, 1, 1], []]
  To:   Kirillov-Reshetikhin crystal of type ['E', 7, 1] with (r,s)=(7,2)
  Defn: ...
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(7),Integer(1)], Integer(7),␣
↪Integer(2))
>>> K.from_A7_crystal()
['A', 6] -> ['E', 7, 1] Virtual Crystal morphism:
  From: The crystal of tableaux of type ['A', 7] and shape(s)
    [[2, 2, 2, 2, 2, 2], [2, 2, 1, 1, 1, 1], [2, 2], [1, 1, 1, 1], []]
  To:   Kirillov-Reshetikhin crystal of type ['E', 7, 1] with (r,s)=(7,2)
  Defn: ...
```

to_A7_crystal()

Return the map decomposing the KR crystal $B^{7,s}$ of type $E_7^{(1)}$ into type A_7 highest weight crystals.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',7,1], 7, 2)
sage: K.to_A7_crystal()
['A', 6] relabelled by {1: 1, 2: 3, 3: 4, 4: 5, 5: 6, 6: 7} -> ['A', 7]␣
↪Virtual Crystal morphism:
  From: Kirillov-Reshetikhin crystal of type ['E', 7, 1] with (r,s)=(7,2)
  To:   The crystal of tableaux of type ['A', 7] and shape(s)
    [[2, 2, 2, 2, 2, 2], [2, 2, 1, 1, 1, 1], [2, 2], [1, 1, 1, 1], []]
  Defn: ...
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['E',Integer(7),Integer(1)], Integer(7),␣
↪Integer(2))
>>> K.to_A7_crystal()
['A', 6] relabelled by {1: 1, 2: 3, 3: 4, 4: 5, 5: 6, 6: 7} -> ['A', 7]␣
↪Virtual Crystal morphism:
  From: Kirillov-Reshetikhin crystal of type ['E', 7, 1] with (r,s)=(7,2)
  To:   The crystal of tableaux of type ['A', 7] and shape(s)
    [[2, 2, 2, 2, 2, 2], [2, 2, 1, 1, 1, 1], [2, 2], [1, 1, 1, 1], []]
  Defn: ...
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_box**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinGenericCrystal*, *AffineCrystalFromClassical*

Class of Kirillov-Reshetikhin crystals $B^{r,s}$ of type $A_{2n}^{(2)}$ for $r \leq n$ and type $D_{n+1}^{(2)}$ for $r < n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2], 1,1)
sage: K
```

(continues on next page)

(continued from previous page)

```
Kirillov-Reshetikhin crystal of type ['BC', 2, 2] with (r,s)=(1,1)
sage: b = K(rows=[])
sage: b.f(0)
[[1]]
sage: b.e(0)
[[-1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)], Integer(1),
↪Integer(1))
>>> K
Kirillov-Reshetikhin crystal of type ['BC', 2, 2] with (r,s)=(1,1)
>>> b = K(rows=[])
>>> b.f(Integer(0))
[[1]]
>>> b.e(Integer(0))
[[-1]]
```

Element

alias of *KR_type_boxElement*

ambient_crystal()

Return the ambient crystal $B^{r,2s}$ of type $C_n^{(1)}$ associated to the Kirillov-Reshetikhin crystal.

The ambient crystal is used to construct the zero arrows.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2], 2,2)
sage: K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['C', 2, 1] with (r,s)=(2,4)
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)], Integer(2),
↪Integer(2))
>>> K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['C', 2, 1] with (r,s)=(2,4)
```

ambient_highest_weight_dict()

Return a dictionary of the classical highest weight vectors of the ambient crystal of `self` whose keys are their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,2)
sage: K.ambient_highest_weight_dict()
{[]: [],
 [2]: [[1, 1]],
 [2, 2]: [[1, 1], [2, 2]],
 [4]: [[1, 1, 1, 1]],
 [4, 2]: [[1, 1, 1, 1], [2, 2]],
 [4, 4]: [[1, 1, 1, 1], [2, 2, 2, 2]]}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(6),Integer(2)], Integer(2),
↪Integer(2))
>>> K.ambient_highest_weight_dict()
{[]: [],
 [2]: [[1, 1]],
 [2, 2]: [[1, 1], [2, 2]],
 [4]: [[1, 1, 1, 1]],
 [4, 2]: [[1, 1, 1, 1], [2, 2]],
 [4, 4]: [[1, 1, 1, 1], [2, 2, 2, 2]]}
```

classical_decomposition()

Return the classical crystal underlying the Kirillov-Reshetikhin crystal of type $A_{2n}^{(2)}$ and $D_{n+1}^{(2)}$.

It is given by $B^{r,s} \cong \bigoplus_{\Lambda} B(\Lambda)$, where Λ are weights obtained from a rectangle of width s and height r by removing boxes. Here we identify the fundamental weight Λ_i with a column of height i.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 2] and shape(s) [[], [1], [2], [1, 1],␣
↪[2, 1], [2, 2]]
sage: K = crystals.KirillovReshetikhin(['D',4,2], 2,3)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[], [1], [2], [1, 1],␣
↪[3], [2, 1], [3, 1], [2, 2], [3, 2], [3, 3]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)], Integer(2),
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['C', 2] and shape(s) [[], [1], [2], [1, 1],␣
↪[2, 1], [2, 2]]
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)], Integer(2),
↪Integer(3))
>>> K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[], [1], [2], [1, 1],␣
↪[3], [2, 1], [3, 1], [2, 2], [3, 2], [3, 3]]
```

from_ambient_crystal()

Return a map from the ambient crystal of type $C_n^{(1)}$ to the Kirillov-Reshetikhin crystal `self`.

Note that this map is only well-defined on elements that are in the image under *to_ambient_crystal()*.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2], 1,1)
sage: b = K.ambient_crystal()(rows=[[3,-3]])
sage: K.from_ambient_crystal()(b)
[[0]]
sage: K = crystals.KirillovReshetikhin(['A',4,2], 1,1)
sage: b = K.ambient_crystal()(rows=[])
```

(continues on next page)

(continued from previous page)

```
sage: K.from_ambient_crystal()(b)
[]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)], Integer(1),
↪Integer(1))
>>> b = K.ambient_crystal()(rows=[[Integer(3),-Integer(3)]])
>>> K.from_ambient_crystal()(b)
[[0]]
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)], Integer(1),
↪Integer(1))
>>> b = K.ambient_crystal()(rows=[])
>>> K.from_ambient_crystal()(b)
[]
```

highest_weight_dict()

Return a dictionary of the classical highest weight vectors of `self` whose keys are 2 times their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,2)
sage: K.highest_weight_dict()
{[]: [],
 [2]: [[1]],
 [2, 2]: [[1], [2]],
 [4]: [[1, 1]],
 [4, 2]: [[1, 1], [2]],
 [4, 4]: [[1, 1], [2, 2]]}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(6),Integer(2)], Integer(2),
↪Integer(2))
>>> K.highest_weight_dict()
{[]: [],
 [2]: [[1]],
 [2, 2]: [[1], [2]],
 [4]: [[1, 1]],
 [4, 2]: [[1, 1], [2]],
 [4, 4]: [[1, 1], [2, 2]]}
```

similarity_factor()

Set the similarity factor used to map to the ambient crystal.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,2)
sage: K.similarity_factor()
{1: 2, 2: 2, 3: 2}
sage: K = crystals.KirillovReshetikhin(['D',5,2], 1,1)
sage: K.similarity_factor()
{1: 2, 2: 2, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(6),Integer(2)], Integer(2),
↪Integer(2))
>>> K.similarity_factor()
{1: 2, 2: 2, 3: 2}
>>> K = crystals.KirillovReshetikhin(['D',Integer(5),Integer(2)], Integer(1),
↪Integer(1))
>>> K.similarity_factor()
{1: 2, 2: 2, 3: 2, 4: 1}
```

to_ambient_crystal()

Return a map from `self` to the ambient crystal of type $C_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2], 1,1)
sage: [K.to_ambient_crystal()(b) for b in K]
[[], [[1, 1]], [[2, 2]], [[3, 3]], [[3, -3]], [[-3, -3]], [[-2, -2]], [[-1, -
↪1]]]
sage: K = crystals.KirillovReshetikhin(['A',4,2], 1,1)
sage: [K.to_ambient_crystal()(b) for b in K]
[[], [[1, 1]], [[2, 2]], [[-2, -2]], [[-1, -1]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(2)], Integer(1),
↪Integer(1))
>>> [K.to_ambient_crystal()(b) for b in K]
[[], [[1, 1]], [[2, 2]], [[3, 3]], [[3, -3]], [[-3, -3]], [[-2, -2]], [[-1, -
↪1]]]
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)], Integer(1),
↪Integer(1))
>>> [K.to_ambient_crystal()(b) for b in K]
[[], [[1, 1]], [[2, 2]], [[-2, -2]], [[-1, -1]]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_boxElement**

Bases: *KirillovReshetikhinGenericCrystalElement*

Class for the elements in the Kirillov-Reshetikhin crystals $B^{r,s}$ of type $A_{2n}^{(2)}$ for $r \leq n$ and type $D_{n+1}^{(2)}$ for $r < n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2],1,2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_box_with_category.
↪element_class'>
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)],Integer(1),
↪Integer(2))
>>> type(K.module_generators[Integer(0)])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_box_with_category.
↪element_class'>
```

e0()

Return e_0 on `self` by mapping `self` to the ambient crystal, calculating e_0 there and pulling the element back.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2],1,1)
sage: b = K(rows=[])
sage: b.e(0) # indirect doctest
[[-1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)],Integer(1),
↪Integer(1))
>>> b = K(rows=[])
>>> b.e(Integer(0)) # indirect doctest
[[-1]]
```

epsilon0()

Return ε_0 of `self` by mapping the element to the ambient crystal and calculating ε_0 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2], 1,1)
sage: b = K(rows=[[1]])
sage: b.epsilon(0) # indirect doctest
2
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)], Integer(1),
↪Integer(1))
>>> b = K(rows=[[Integer(1)]])
>>> b.epsilon(Integer(0)) # indirect doctest
2
```

f0()

Return f_0 on `self` by mapping `self` to the ambient crystal, calculating f_0 there and pulling the element back.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2],1,1)
sage: b = K(rows=[])
sage: b.f(0) # indirect doctest
[[1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(4),Integer(2)],Integer(1),
↪Integer(1))
>>> b = K(rows=[])
>>> b.f(Integer(0)) # indirect doctest
[[1]]
```

phi0()

Return φ_0 of `self` by mapping the element to the ambient crystal and calculating φ_0 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',3,2], 1,1)
sage: b = K(rows=[[-1]])
sage: b.phi(0) # indirect doctest
2
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(3),Integer(2)], Integer(1),
↪Integer(1))
>>> b = K(rows=[[-Integer(1)]])
>>> b.phi(Integer(0)) # indirect doctest
2
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_spin**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinCrystalFromPromotion*

Class of Kirillov-Reshetikhin crystals $B^{n,s}$ of type $D_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],4,1); K
Kirillov-Reshetikhin crystal of type ['D', 4, 1] with (r,s)=(4,1)
sage: [[b,b.f(0)] for b in K]
[[[++++, []], None], [[++--, []], None], [[+-+-, []], None],
 [[-++-, []], None], [[+--+, []], None], [[-+-+, []], None],
 [[--++, []], [++++, []]], [[----, []], [++--, []]]]

sage: K = crystals.KirillovReshetikhin(['D',4,1],4,2); K
Kirillov-Reshetikhin crystal of type ['D', 4, 1] with (r,s)=(4,2)
sage: [[b,b.f(0)] for b in K]
[[[[1], [2], [3], [4]], None], [[[1], [2], [-4], [4]], None],
 [[[1], [3], [-4], [4]], None], [[[2], [3], [-4], [4]], None],
 [[[1], [4], [-4], [4]], None], [[[2], [4], [-4], [4]], None],
 [[[3], [4], [-4], [4]], [[1], [2], [3], [4]]],
 [[[-4], [4], [-4], [4]], [[1], [2], [-4], [4]]],
 [[[-4], [4], [-4], [-3]], [[1], [2], [-4], [-3]]],
 [[[-4], [4], [-4], [-2]], [[1], [3], [-4], [-3]]],
 [[[-4], [4], [-4], [-1]], [[2], [3], [-4], [-3]]],
 [[[-4], [4], [-3], [-2]], [[1], [4], [-4], [-3]]],
 [[[-4], [4], [-3], [-1]], [[2], [4], [-4], [-3]]],
 [[[-4], [4], [-2], [-1]], [[-4], [4], [-4], [4]]],
 [[[-4], [-3], [-2], [-1]], [[-4], [4], [-4], [-3]]],
 [[[1], [2], [-4], [-3]], None], [[[1], [3], [-4], [-3]], None],
 [[[2], [3], [-4], [-3]], None], [[[1], [3], [-4], [-2]], None],
 [[[2], [3], [-4], [-2]], None], [[[2], [3], [-4], [-1]], None],
 [[[1], [4], [-4], [-3]], None], [[[2], [4], [-4], [-3]], None],
 [[[3], [4], [-4], [-3]], None],
 [[[3], [4], [-4], [-2]], [[1], [3], [-4], [4]]],
 [[[3], [4], [-4], [-1]], [[2], [3], [-4], [4]]],
 [[[1], [4], [-4], [-2]], None], [[[2], [4], [-4], [-2]], None],
```

(continues on next page)

(continued from previous page)

```
[[[2], [4], [-4], [-1]], None], [[[1], [4], [-3], [-2]], None],
 [[[2], [4], [-3], [-2]], None], [[[2], [4], [-3], [-1]], None],
 [[[3], [4], [-3], [-2]], [[1], [4], [-4], [4]]],
 [[[3], [4], [-3], [-1]], [[2], [4], [-4], [4]]],
 [[[3], [4], [-2], [-1]], [[3], [4], [-4], [4]]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(4),
↪Integer(1)); K
Kirillov-Reshetikhin crystal of type ['D', 4, 1] with (r,s)=(4,1)
>>> [[b,b.f(Integer(0))] for b in K]
[[[++++, []], None], [[++--, []], None], [[+-+-, []], None],
 [[-++-, []], None], [[+--+, []], None], [[-+-+, []], None],
 [[--++, []], [++++, []]], [[----, []], [++--, []]]]

>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(4),
↪Integer(2)); K
Kirillov-Reshetikhin crystal of type ['D', 4, 1] with (r,s)=(4,2)
>>> [[b,b.f(Integer(0))] for b in K]
[[[[1], [2], [3], [4]], None], [[[1], [2], [-4], [4]], None],
 [[[1], [3], [-4], [4]], None], [[[2], [3], [-4], [4]], None],
 [[[1], [4], [-4], [4]], None], [[[2], [4], [-4], [4]], None],
 [[[3], [4], [-4], [4]], [[1], [2], [3], [4]]],
 [[[-4], [4], [-4], [4]], [[1], [2], [-4], [4]]],
 [[[-4], [4], [-4], [-3]], [[1], [2], [-4], [-3]]],
 [[[-4], [4], [-4], [-2]], [[1], [3], [-4], [-3]]],
 [[[-4], [4], [-4], [-1]], [[2], [3], [-4], [-3]]],
 [[[-4], [4], [-3], [-2]], [[1], [4], [-4], [-3]]],
 [[[-4], [4], [-3], [-1]], [[2], [4], [-4], [-3]]],
 [[[-4], [4], [-2], [-1]], [[-4], [4], [-4], [4]]],
 [[[-4], [-3], [-2], [-1]], [[-4], [4], [-4], [-3]]],
 [[[1], [2], [-4], [-3]], None], [[[1], [3], [-4], [-3]], None],
 [[[2], [3], [-4], [-3]], None], [[[1], [3], [-4], [-2]], None],
 [[[2], [3], [-4], [-2]], None], [[[2], [3], [-4], [-1]], None],
 [[[1], [4], [-4], [-3]], None], [[[2], [4], [-4], [-3]], None],
 [[[3], [4], [-4], [-3]], None],
 [[[3], [4], [-4], [-2]], [[1], [3], [-4], [4]]],
 [[[3], [4], [-4], [-1]], [[2], [3], [-4], [4]]],
 [[[1], [4], [-4], [-2]], None], [[[2], [4], [-4], [-2]], None],
 [[[2], [4], [-4], [-1]], None], [[[1], [4], [-3], [-2]], None],
 [[[2], [4], [-3], [-2]], None], [[[2], [4], [-3], [-1]], None],
 [[[3], [4], [-3], [-2]], [[1], [4], [-4], [4]]],
 [[[3], [4], [-3], [-1]], [[2], [4], [-4], [4]]],
 [[[3], [4], [-2], [-1]], [[3], [4], [-4], [4]]]]
```

classical_decomposition()

Return the classical crystal underlying the Kirillov-Reshetikhin crystal $B^{r,s}$ of type $D_n^{(1)}$ for $r = n-1, n$.

The classical decomposition is given by $B^{n,s} \cong B(s\Lambda_r)$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],4,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, 1/2]]
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, -1/2]]
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1, 1, 1, -1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(4),
↪Integer(1))
>>> K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, 1/2]]
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(3),
↪Integer(1))
>>> K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, -1/2]]
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(3),
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1, 1, 1, -1]]
```

dynkin_diagram_automorphism(*i*)

Specifies the Dynkin diagram automorphism underlying the promotion action on the crystal elements.

Here we use the Dynkin diagram automorphism which interchanges nodes 0 and 1 and leaves all other nodes unchanged.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],4,1)
sage: K.dynkin_diagram_automorphism(0)
1
sage: K.dynkin_diagram_automorphism(1)
0
sage: K.dynkin_diagram_automorphism(4)
4
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(4),
↪Integer(1))
>>> K.dynkin_diagram_automorphism(Integer(0))
1
>>> K.dynkin_diagram_automorphism(Integer(1))
0
>>> K.dynkin_diagram_automorphism(Integer(4))
4
```

promotion()

Return the promotion operator on $B^{r,s}$ of type $D_n^{(1)}$ for $r = n-1, n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: T = K.classical_decomposition()
sage: promotion = K.promotion()
sage: for t in T:
....:     print("{} {}".format(t, promotion(t)))
[+++-, []] [-++-, []]
[++-+, []] [-+-+, []]
[+-++, []] [--++, []]
[-+++, []] [++++, []]
[+---, []] [----, []]
[-+--, []] [++--, []]
[--+-, []] [+-+-, []]
[---+, []] [+--+, []]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(3),
↪Integer(1))
>>> T = K.classical_decomposition()
>>> promotion = K.promotion()
>>> for t in T:
...     print("{} {}".format(t, promotion(t)))
[+++-, []] [-++-, []]
[++-+, []] [-+-+, []]
[+-++, []] [--++, []]
[-+++, []] [++++, []]
[+---, []] [----, []]
[-+--, []] [++--, []]
[--+-, []] [+-+-, []]
[---+, []] [+--+, []]
```

promotion_inverse()

Return the inverse promotion operator on $B^{r,s}$ of type $D_n^{(1)}$ for $r = n-1, n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: T = K.classical_decomposition()
sage: promotion = K.promotion()
sage: promotion_inverse = K.promotion_inverse()
sage: all(promotion_inverse(promotion(t)) == t for t in T)
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(3),
↪Integer(1))
>>> T = K.classical_decomposition()
>>> promotion = K.promotion()
>>> promotion_inverse = K.promotion_inverse()
>>> all(promotion_inverse(promotion(t)) == t for t in T)
True
```

promotion_on_highest_weight_vectors()

Return the promotion operator on $\{2, 3, \ldots, n\}$-highest weight vectors.

A $\{2, 3, \ldots, n\}$-highest weight vector in $B(s\Lambda_n)$ of weight $w = (w_1, \ldots, w_n)$ is mapped to a $\{2, 3, \ldots, n\}$-highest weight vector in $B(s\Lambda_{n-1})$ of weight $(-w_1, w_2, \ldots, w_n)$ and vice versa.

See also:

- *promotion_on_highest_weight_vectors_inverse()*
- *promotion()*

EXAMPLES:

```
sage: KR = crystals.KirillovReshetikhin(['D',4,1],4,2)
sage: prom = KR.promotion_on_highest_weight_vectors()
sage: T = KR.classical_decomposition()
sage: HW = [t for t in T if t.is_highest_weight([2,3,4])]
sage: for t in HW:
....:     print("{} {}".format(t, prom[t]))
[[1], [2], [3], [4]] [[2], [3], [4], [-1]]
[[2], [3], [-4], [4]] [[2], [3], [4], [-4]]
[[2], [3], [-4], [-1]] [[1], [2], [3], [-4]]

sage: KR = crystals.KirillovReshetikhin(['D',4,1],4,1)
sage: prom = KR.promotion_on_highest_weight_vectors()
sage: T = KR.classical_decomposition()
sage: HW = [t for t in T if t.is_highest_weight([2,3,4])]
sage: for t in HW:
....:     print("{} {}".format(t, prom[t]))
[++++, []] [-+++, []]
[-++-, []] [+++-, []]
```

```
>>> from sage.all import *
>>> KR = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(4),
↪Integer(2))
>>> prom = KR.promotion_on_highest_weight_vectors()
>>> T = KR.classical_decomposition()
>>> HW = [t for t in T if t.is_highest_weight([Integer(2),Integer(3),
↪Integer(4)])]
>>> for t in HW:
...     print("{} {}".format(t, prom[t]))
[[1], [2], [3], [4]] [[2], [3], [4], [-1]]
[[2], [3], [-4], [4]] [[2], [3], [4], [-4]]
[[2], [3], [-4], [-1]] [[1], [2], [3], [-4]]

>>> KR = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(4),
↪Integer(1))
>>> prom = KR.promotion_on_highest_weight_vectors()
>>> T = KR.classical_decomposition()
>>> HW = [t for t in T if t.is_highest_weight([Integer(2),Integer(3),
↪Integer(4)])]
>>> for t in HW:
...     print("{} {}".format(t, prom[t]))
[++++, []] [-+++, []]
[-++-, []] [+++-, []]
```

promotion_on_highest_weight_vectors_inverse()

Return the inverse promotion operator on $\{2, 3, \ldots, n\}$-highest weight vectors.

> **See also**
>
> - *promotion_on_highest_weight_vectors()*
> - *promotion_inverse()*

EXAMPLES:

```
sage: KR = crystals.KirillovReshetikhin(['D',4,1],3,2)
sage: prom = KR.promotion_on_highest_weight_vectors()
sage: prom_inv = KR.promotion_on_highest_weight_vectors_inverse()
sage: T = KR.classical_decomposition()
sage: HW = [t for t in T if t.is_highest_weight([2,3,4])]
sage: all(prom_inv[prom[t]] == t for t in HW)
True
```

```
>>> from sage.all import *
>>> KR = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(3),
↪Integer(2))
>>> prom = KR.promotion_on_highest_weight_vectors()
>>> prom_inv = KR.promotion_on_highest_weight_vectors_inverse()
>>> T = KR.classical_decomposition()
>>> HW = [t for t in T if t.is_highest_weight([Integer(2),Integer(3),
↪Integer(4)])]
>>> all(prom_inv[prom[t]] == t for t in HW)
True
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_vertical**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinCrystalFromPromotion*

Class of Kirillov-Reshetikhin crystals $B^{r,s}$ of type $D_n^{(1)}$ for $r \leq n-2$, $B_n^{(1)}$ for $r < n$, and $A_{2n-1}^{(2)}$ for $r \leq n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: b = K(rows=[])
sage: b.f(0)
[[1], [2]]
sage: b.f(0).f(0)
[[1, 1], [2, 2]]
sage: b.e(0)
[[-2], [-1]]
sage: b.e(0).e(0)
[[-2, -2], [-1, -1]]

sage: K = crystals.KirillovReshetikhin(['D',5,1], 3,1)
sage: b = K(rows=[[1]])
sage: b.e(0)
[[3], [-3], [-2]]
```

(continues on next page)

(continued from previous page)

```
sage: K = crystals.KirillovReshetikhin(['B',3,1], 1,1)
sage: [[b,b.f(0)] for b in K]
[[[[1]], None], [[[2]], None], [[[3]], None], [[[0]], None],
 [[[-3]], None], [[[-2]], [[1]]], [[[-1]], [[2]]]]

sage: K = crystals.KirillovReshetikhin(['A',5,2], 1,1)
sage: [[b,b.f(0)] for b in K]
[[[[1]], None], [[[2]], None], [[[3]], None], [[[-3]], None],
 [[[-2]], [[1]]], [[[-1]], [[2]]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(2))
>>> b = K(rows=[])
>>> b.f(Integer(0))
[[1], [2]]
>>> b.f(Integer(0)).f(Integer(0))
[[1, 1], [2, 2]]
>>> b.e(Integer(0))
[[-2], [-1]]
>>> b.e(Integer(0)).e(Integer(0))
[[-2, -2], [-1, -1]]

>>> K = crystals.KirillovReshetikhin(['D',Integer(5),Integer(1)], Integer(3),
↪Integer(1))
>>> b = K(rows=[[Integer(1)]])
>>> b.e(Integer(0))
[[3], [-3], [-2]]

>>> K = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)], Integer(1),
↪Integer(1))
>>> [[b,b.f(Integer(0))] for b in K]
[[[[1]], None], [[[2]], None], [[[3]], None], [[[0]], None],
 [[[-3]], None], [[[-2]], [[1]]], [[[-1]], [[2]]]]

>>> K = crystals.KirillovReshetikhin(['A',Integer(5),Integer(2)], Integer(1),
↪Integer(1))
>>> [[b,b.f(Integer(0))] for b in K]
[[[[1]], None], [[[2]], None], [[[3]], None], [[[-3]], None],
 [[[-2]], [[1]]], [[[-1]], [[2]]]]
```

classical_decomposition()

Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal of type $D_n^{(1)}$, $B_n^{(1)}$, and $A_{2n-1}^{(2)}$.

It is given by $B^{r,s} \cong \bigoplus_\Lambda B(\Lambda)$, where Λ are weights obtained from a rectangle of width s and height r by removing vertical dominoes. Here we identify the fundamental weight Λ_i with a column of height i.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[], [1, 1], [2, 2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(2))
>>> K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[], [1, 1], [2, 2]]
```

dynkin_diagram_automorphism(*i*)

Specifies the Dynkin diagram automorphism underlying the promotion action on the crystal elements. The automorphism needs to map node 0 to some other Dynkin node.

Here we use the Dynkin diagram automorphism which interchanges nodes 0 and 1 and leaves all other nodes unchanged.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],1,1)
sage: K.dynkin_diagram_automorphism(0)
1
sage: K.dynkin_diagram_automorphism(1)
0
sage: K.dynkin_diagram_automorphism(4)
4
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(1),
↪Integer(1))
>>> K.dynkin_diagram_automorphism(Integer(0))
1
>>> K.dynkin_diagram_automorphism(Integer(1))
0
>>> K.dynkin_diagram_automorphism(Integer(4))
4
```

from_highest_weight_vector_to_pm_diagram(*b*)

This gives the bijection between an element b in the classical decomposition of the KR crystal that is $2, 3, \ldots, n$-highest weight and $\pm$ diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: T = K.classical_decomposition()
sage: b = T(rows=[[2],[-2]])
sage: pm = K.from_highest_weight_vector_to_pm_diagram(b); pm
[[1, 1], [0, 0], [0]]
sage: pm.pp()
+
-
sage: b = T(rows=[])
sage: pm=K.from_highest_weight_vector_to_pm_diagram(b); pm
[[0, 2], [0, 0], [0]]
sage: pm.pp()

sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3,4]) ]
sage: all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_
```

(continues on next page)

(continued from previous page)

```
↪vector_to_pm_diagram(b)) == b for b in hw)
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(2))
>>> T = K.classical_decomposition()
>>> b = T(rows=[[Integer(2)],[-Integer(2)]])
>>> pm = K.from_highest_weight_vector_to_pm_diagram(b); pm
[[1, 1], [0, 0], [0]]
>>> pm.pp()
+
-
>>> b = T(rows=[])
>>> pm=K.from_highest_weight_vector_to_pm_diagram(b); pm
[[0, 2], [0, 0], [0]]
>>> pm.pp()

>>> hw = [ b for b in T if all(b.epsilon(i)==Integer(0) for i in [Integer(2),
↪Integer(3),Integer(4)]) ]
>>> all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_
↪vector_to_pm_diagram(b)) == b for b in hw)
True
```

from_pm_diagram_to_highest_weight_vector(*pm*)

This gives the bijection between a $\pm$ diagram and an element `b` in the classical decomposition of the KR crystal that is $2, 3, \ldots, n$-highest weight.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[1, 1], [0,␣
↪0], [0]])
sage: K.from_pm_diagram_to_highest_weight_vector(pm)
[[2], [-2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(2))
>>> pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[Integer(1),␣
↪Integer(1)], [Integer(0), Integer(0)], [Integer(0)]])
>>> K.from_pm_diagram_to_highest_weight_vector(pm)
[[2], [-2]]
```

promotion()

Specifies the promotion operator used to construct the affine type $D_n^{(1)}$ etc. crystal.

This corresponds to the Dynkin diagram automorphism which interchanges nodes 0 and 1, and leaves all other nodes unchanged. On the level of crystals it is constructed using $\pm$ diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: promotion = K.promotion()
sage: b = K.classical_decomposition()(rows=[])
sage: promotion(b)
[[1, 2], [-2, -1]]
sage: b = K.classical_decomposition()(rows=[[1,3],[2,-1]])
sage: promotion(b)
[[1, 3], [2, -1]]
sage: b = K.classical_decomposition()(rows=[[1],[-3]])
sage: promotion(b)
[[2, -3], [-2, -1]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(2))
>>> promotion = K.promotion()
>>> b = K.classical_decomposition()(rows=[])
>>> promotion(b)
[[1, 2], [-2, -1]]
>>> b = K.classical_decomposition()(rows=[[Integer(1),Integer(3)],[Integer(2),
↪-Integer(1)]])
>>> promotion(b)
[[1, 3], [2, -1]]
>>> b = K.classical_decomposition()(rows=[[Integer(1)],[-Integer(3)]])
>>> promotion(b)
[[2, -3], [-2, -1]]
```

promotion_inverse()

Return inverse of promotion.

In this case promotion is an involution, so promotion inverse equals promotion.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: promotion = K.promotion()
sage: promotion_inverse = K.promotion_inverse()
sage: all( promotion_inverse(promotion(b.lift())) == b.lift() for b in K )
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(2))
>>> promotion = K.promotion()
>>> promotion_inverse = K.promotion_inverse()
>>> all( promotion_inverse(promotion(b.lift())) == b.lift() for b in K )
True
```

promotion_on_highest_weight_vector(b)

Calculate promotion on a $2, 3, \ldots, n$ highest weight vector b.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: T = K.classical_decomposition()
sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3,4]) ]
sage: [K.promotion_on_highest_weight_vector(b) for b in hw]
[[[1, 2], [-2, -1]], [[2, 2], [-2, -1]], [[1, 2], [3, -1]],
 [[2], [-2]], [[1, 2], [2, -2]], [[2, 2], [-1, -1]],
 [[2, 2], [3, -1]], [[2, 2], [3, 3]], [], [[1], [2]],
 [[1, 1], [2, 2]], [[2], [-1]], [[1, 2], [2, -1]],
 [[2], [3]], [[1, 2], [2, 3]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(2))
>>> T = K.classical_decomposition()
>>> hw = [ b for b in T if all(b.epsilon(i)==Integer(0) for i in [Integer(2),
↪Integer(3),Integer(4)]) ]
>>> [K.promotion_on_highest_weight_vector(b) for b in hw]
[[[1, 2], [-2, -1]], [[2, 2], [-2, -1]], [[1, 2], [3, -1]],
 [[2], [-2]], [[1, 2], [2, -2]], [[2, 2], [-1, -1]],
 [[2, 2], [3, -1]], [[2, 2], [3, 3]], [], [[1], [2]],
 [[1, 1], [2, 2]], [[2], [-1]], [[1, 2], [2, -1]],
 [[2], [3]], [[1, 2], [2, 3]]]
```

sage.combinat.crystals.kirillov_reshetikhin.**KashiwaraNakashimaTableaux**(*cartan_type*, *r*, *s*)

Return the Kashiwara-Nakashima model for the Kirillov-Reshetikhin crystal $B^{r,s}$ in the given type.

The Kashiwara-Nakashima (KN) model constructs the KR crystal from the KN tableaux model for the corresponding classical crystals. This model is named for the underlying KN tableaux.

Many Kirillov-Reshetikhin crystals are constructed from a classical crystal together with an automorphism p on the level of crystals which corresponds to a Dynkin diagram automorphism mapping node 0 to some other node i. The action of f_0 and e_0 is then constructed using $f_0 = p^{-1} \circ f_i \circ p$.

For example, for type $A_n^{(1)}$ the Kirillov-Reshetikhin crystal $B^{r,s}$ is obtained from the classical crystal $B(s\omega_r)$ using the promotion operator. For other types, see [Shi2002], [Sch2008], and [JS2010].

Other Kirillov-Reshetikhin crystals are constructed using similarity methods. See Section 4 of [FOS2009].

For more information on Kirillov-Reshetikhin crystals, see `KirillovReshetikhinCrystal()`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2, 1)
sage: K2 = crystals.kirillov_reshetikhin.KashiwaraNakashimaTableaux(['A',3,1], 2,␣
↪1)
sage: K is K2
True
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),␣
↪Integer(1))
>>> K2 = crystals.kirillov_reshetikhin.KashiwaraNakashimaTableaux(['A',Integer(3),
↪Integer(1)], Integer(2), Integer(1))
>>> K is K2
True
```

sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinCrystal**(*cartan_type*, *r*, *s*, *model='KN'*)

Return the Kirillov-Reshetikhin crystal $B^{r,s}$ of the given type in the given model.

For more information about general crystals see *sage.combinat.crystals.crystals*.

There are a variety of models for Kirillov-Reshetikhin crystals. There is one using the classical crystal with *Kashiwara-Nakashima tableaux*. There is one using *rigged configurations*. Another tableaux model comes from the bijection between rigged configurations and tensor products of tableaux called *Kirillov-Reshetikhin tableaux* Lastly there is a model of Kirillov-Reshetikhin crystals for $s = 1$ from crystals of *LS paths*.

INPUT:

- `cartan_type` – an affine Cartan type
- `r` – a label of finite Dynkin diagram
- `s` – positive integer
- `model` – (default: `'KN'`) can be one of the following:
  - `'KN'` or `'KashiwaraNakashimaTableaux'` – use the Kashiwara-Nakashima tableaux model
  - `'KR'` or `'KirillovReshetkihinTableaux'` – use the Kirillov-Reshetkihin tableaux model
  - `'RC'` or `'RiggedConfiguration'` – use the rigged configuration model
  - `'LSPaths'` – use the LS path model

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2, 1)
sage: K.index_set()
(0, 1, 2, 3)
sage: K.list()
[[[1], [2]], [[1], [3]], [[2], [3]], [[1], [4]], [[2], [4]], [[3], [4]]]
sage: b=K(rows=[[1],[2]])
sage: b.weight()
-Lambda[0] + Lambda[2]

sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.automorphism(K.module_generators[0])
[[2, 2], [3, 3]]
sage: K.module_generators[0].e(0)
[[1, 2], [2, 4]]
sage: K.module_generators[0].f(2)
[[1, 1], [2, 3]]
sage: K.module_generators[0].f(1)
sage: K.module_generators[0].phi(0)
0
sage: K.module_generators[0].phi(1)
0
sage: K.module_generators[0].phi(2)
2
sage: K.module_generators[0].epsilon(0)
2
sage: K.module_generators[0].epsilon(1)
0
sage: K.module_generators[0].epsilon(2)
```

(continues on next page)

(continued from previous page)

```
0
sage: b = K(rows=[[1,2],[2,3]])
sage: b
[[1, 2], [2, 3]]
sage: b.f(2)
[[1, 2], [3, 3]]

sage: K = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: K.cartan_type()
['D', 4, 1]
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_vertical_with_
↪category.element_class'>
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),␣
↪Integer(1))
>>> K.index_set()
(0, 1, 2, 3)
>>> K.list()
[[[1], [2]], [[1], [3]], [[2], [3]], [[1], [4]], [[2], [4]], [[3], [4]]]
>>> b=K(rows=[[Integer(1)],[Integer(2)]])
>>> b.weight()
-Lambda[0] + Lambda[2]

>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> K.automorphism(K.module_generators[Integer(0)])
[[2, 2], [3, 3]]
>>> K.module_generators[Integer(0)].e(Integer(0))
[[1, 2], [2, 4]]
>>> K.module_generators[Integer(0)].f(Integer(2))
[[1, 1], [2, 3]]
>>> K.module_generators[Integer(0)].f(Integer(1))
>>> K.module_generators[Integer(0)].phi(Integer(0))
0
>>> K.module_generators[Integer(0)].phi(Integer(1))
0
>>> K.module_generators[Integer(0)].phi(Integer(2))
2
>>> K.module_generators[Integer(0)].epsilon(Integer(0))
2
>>> K.module_generators[Integer(0)].epsilon(Integer(1))
0
>>> K.module_generators[Integer(0)].epsilon(Integer(2))
0
>>> b = K(rows=[[Integer(1),Integer(2)],[Integer(2),Integer(3)]])
>>> b
[[1, 2], [2, 3]]
>>> b.f(Integer(2))
[[1, 2], [3, 3]]
```

(continues on next page)

(continued from previous page)

```
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),␣
↪Integer(1))
>>> K.cartan_type()
['D', 4, 1]
>>> type(K.module_generators[Integer(0)])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_vertical_with_
↪category.element_class'>
```

The following gives some tests with regards to Lemma 3.11 in [LOS2012].

REFERENCES:

- [Shi2002]
- [Sch2008]
- [JS2010]
- [FOS2009]
- [LOS2012]

`sage.combinat.crystals.kirillov_reshetikhin.`**`KirillovReshetikhinCrystalFromLSPaths`**(*cartan_type*, *r*, *s=1*)

Single column Kirillov-Reshetikhin crystals.

This yields the single column Kirillov-Reshetikhin crystals from the projected level zero LS paths, see *CrystalOfLSPaths*. This works for all types (even exceptional types). The weight of the canonical element in this crystal is Λ_r. For other implementation see *KirillovReshetikhinCrystal()*.

EXAMPLES:

```
sage: K = crystals.kirillov_reshetikhin.LSPaths(['A',2,1],2) # indirect doctest
sage: KR = crystals.KirillovReshetikhin(['A',2,1],2,1)
sage: G = K.digraph()
sage: GR = KR.digraph()
sage: G.is_isomorphic(GR, edge_labels = True)
True

sage: K = crystals.kirillov_reshetikhin.LSPaths(['C',3,1],2)
sage: KR = crystals.KirillovReshetikhin(['C',3,1],2,1)
sage: G = K.digraph()
sage: GR = KR.digraph()
sage: G.is_isomorphic(GR, edge_labels = True)
True

sage: K = crystals.kirillov_reshetikhin.LSPaths(['E',6,1],1)
sage: KR = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: G = K.digraph()
sage: GR = KR.digraph()
sage: G.is_isomorphic(GR, edge_labels = True)
True
sage: K.cardinality()
27
```

(continues on next page)

(continued from previous page)

```
sage: K = crystals.kirillov_reshetikhin.LSPaths(['G',2,1],1)
sage: K.cardinality()
7

sage: K = crystals.kirillov_reshetikhin.LSPaths(['B',3,1],2)
sage: KR = crystals.KirillovReshetikhin(['B',3,1],2,1)
sage: KR.cardinality()
22
sage: K.cardinality()
22
sage: G = K.digraph()
sage: GR = KR.digraph()
sage: G.is_isomorphic(GR, edge_labels = True)
True
```

```
>>> from sage.all import *
>>> K = crystals.kirillov_reshetikhin.LSPaths(['A',Integer(2),Integer(1)],
↪Integer(2)) # indirect doctest
>>> KR = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],Integer(2),
↪Integer(1))
>>> G = K.digraph()
>>> GR = KR.digraph()
>>> G.is_isomorphic(GR, edge_labels = True)
True

>>> K = crystals.kirillov_reshetikhin.LSPaths(['C',Integer(3),Integer(1)],
↪Integer(2))
>>> KR = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)],Integer(2),
↪Integer(1))
>>> G = K.digraph()
>>> GR = KR.digraph()
>>> G.is_isomorphic(GR, edge_labels = True)
True

>>> K = crystals.kirillov_reshetikhin.LSPaths(['E',Integer(6),Integer(1)],
↪Integer(1))
>>> KR = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(1),
↪Integer(1))
>>> G = K.digraph()
>>> GR = KR.digraph()
>>> G.is_isomorphic(GR, edge_labels = True)
True
>>> K.cardinality()
27

>>> K = crystals.kirillov_reshetikhin.LSPaths(['G',Integer(2),Integer(1)],
↪Integer(1))
>>> K.cardinality()
7

>>> K = crystals.kirillov_reshetikhin.LSPaths(['B',Integer(3),Integer(1)],
```

(continues on next page)

(continued from previous page)

```
↪Integer(2))
>>> KR = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(2),
↪Integer(1))
>>> KR.cardinality()
22
>>> K.cardinality()
22
>>> G = K.digraph()
>>> GR = KR.digraph()
>>> G.is_isomorphic(GR, edge_labels = True)
True
```

class sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinCrystalFromPromotion**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinGenericCrystal*, *AffineCrystalFromClassicalAndPromotion*

This generic class assumes that the Kirillov-Reshetikhin crystal is constructed from a classical crystal using the `classical_decomposition` and an automorphism `promotion` and its inverse, which corresponds to a Dynkin diagram automorphism `dynkin_diagram_automorphism`.

Each instance using this class needs to implement the methods:

- `classical_decomposition`
- `promotion`
- `promotion_inverse`
- `dynkin_diagram_automorphism`

Element

alias of *KirillovReshetikhinCrystalFromPromotionElement*

class sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinCrystalFromPromotionElement**

Bases: *AffineCrystalFromClassicalAndPromotionElement*, *KirillovReshetikhinGenericCrystalElement*

Element for a Kirillov-Reshetikhin crystal from promotion.

class sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinGenericCrystal**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: *AffineCrystalFromClassical*

Generic class for Kirillov-Reshetikhin crystal $B^{r,s}$ of the given type.

Input is a Dynkin node `r`, a positive integer `s`, and a Cartan type `cartan_type`.

Element

alias of *KirillovReshetikhinGenericCrystalElement*

classically_highest_weight_vectors()

Return the classically highest weight vectors of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2)
sage: K.classically_highest_weight_vectors()
([], [[1], [2]], [[1, 1], [2, 2]])
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2))
>>> K.classically_highest_weight_vectors()
([], [[1], [2]], [[1, 1], [2, 2]])
```

kirillov_reshetikhin_tableaux()

Return the corresponding set of *KirillovReshetikhinTableaux*.

EXAMPLES:

```
sage: KRC = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2)
sage: KRC.kirillov_reshetikhin_tableaux()
Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and shape (2, 2)
```

```
>>> from sage.all import *
>>> KRC = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2))
>>> KRC.kirillov_reshetikhin_tableaux()
Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and shape (2, 2)
```

module_generator()

Return the unique module generator of classical weight $s\Lambda_r$ of a Kirillov-Reshetikhin crystal $B^{r,s}$

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1],1,2)
sage: K.module_generator()
[[1, 1]]
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.module_generator()
[(1,)]

sage: K = crystals.KirillovReshetikhin(['D',4,1],2,1)
sage: K.module_generator()
[[1], [2]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['C',Integer(2),Integer(1)],Integer(1),
↪Integer(2))
>>> K.module_generator()
[[1, 1]]
>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],Integer(1),
↪Integer(1))
>>> K.module_generator()
[(1,)]

>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],Integer(2),
↪Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> K.module_generator()
[[1], [2]]
```

r()

Return r of the underlying Kirillov-Reshetikhin crystal $B^{r,s}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: K.r()
2
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),␣
↪Integer(1))
>>> K.r()
2
```

s()

Return s of the underlying Kirillov-Reshetikhin crystal $B^{r,s}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: K.s()
1
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),␣
↪Integer(1))
>>> K.s()
1
```

class
sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinGenericCrystalElement**

Bases: *AffineCrystalFromClassicalElement*

Abstract class for all Kirillov-Reshetikhin crystal elements.

lusztig_involution()

Return the classical Lusztig involution on `self`.

EXAMPLES:

```
sage: KRC = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: elt = KRC(-1,2); elt
[[2], [-1]]
sage: elt.lusztig_involution()
[[1], [-2]]
```

```
>>> from sage.all import *
>>> KRC = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(2),Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> elt = KRC(-Integer(1),Integer(2)); elt
[[2], [-1]]
>>> elt.lusztig_involution()
[[1], [-2]]
```

pp()

Pretty print self.

EXAMPLES:

```
sage: C = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: C(2,1).pp()
  1
  2
sage: C = crystals.KirillovReshetikhin(['B',3,1], 3,3)
sage: C.module_generators[0].pp()
+ (X)   1
+
+
```

```
>>> from sage.all import *
>>> C = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(1))
>>> C(Integer(2),Integer(1)).pp()
  1
  2
>>> C = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)], Integer(3),
↪Integer(3))
>>> C.module_generators[Integer(0)].pp()
+ (X)   1
+
+
```

to_kirillov_reshetikhin_tableau()

Construct the corresponding *KirillovReshetikhinTableauxElement* from self.

We construct the Kirillov-Reshetikhin tableau element as follows:

1. Let λ be the shape of self.
2. Determine a path $e_{i_1} e_{i_2} \cdots e_{i_k}$ to the highest weight.
3. Apply $f_{i_k} \cdots f_{i_2} f_{i_1}$ to a highest weight KR tableau from filling the shape λ.

EXAMPLES:

```
sage: KRC = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1)
sage: KRC(columns=[[2,1]]).to_kirillov_reshetikhin_tableau()
[[1], [2]]
sage: KRC = crystals.KirillovReshetikhin(['D', 4, 1], 2, 1)
sage: KRC(rows=[]).to_kirillov_reshetikhin_tableau()
[[1], [-1]]
```

```
>>> from sage.all import *
>>> KRC = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)],␣
↪Integer(2), Integer(1))
>>> KRC(columns=[[Integer(2),Integer(1)]]).to_kirillov_reshetikhin_tableau()
[[1], [2]]
>>> KRC = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(2), Integer(1))
>>> KRC(rows=[]).to_kirillov_reshetikhin_tableau()
[[1], [-1]]
```

to_tableau()

Return the *Tableau* corresponding to `self`.

EXAMPLES:

```
sage: C = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: t = C(2,1).to_tableau(); t
[[1], [2]]
sage: type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
```

```
>>> from sage.all import *
>>> C = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪Integer(1))
>>> t = C(Integer(2),Integer(1)).to_tableau(); t
[[1], [2]]
>>> type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
```

class sage.combinat.crystals.kirillov_reshetikhin.**PMDiagram**(*pm_diagram*, *from_shapes=None*)

Bases: *CombinatorialObject*

Class of $\pm$ diagrams. These diagrams are in one-to-one bijection with X_{n-1} highest weight vectors in an X_n highest weight crystal $X = B, C, D$. See Section 4.1 of [Sch2008].

The input is a list $pm = [[a_0, b_0], [a_1, b_1], ..., [a_{n-1}, b_{n-1}], [b_n]]$ of pairs and a last 1-tuple (or list of length 1). The pair $[a_i, b_i]$ specifies the number of a_i $+$ and b_i $-$ in the i-th row of the $\pm$ diagram if $n - i$ is odd and the number of a_i $\pm$ pairs above row i and b_i columns of height i not containing any $+$ or $-$ if $n - i$ is even.

Setting the option `from_shapes = True` one can also input a $\pm$ diagram in terms of its outer, intermediate, and inner shape by specifying a list `[n, s, outer, intermediate, inner]` where `s` is the width of the $\pm$ diagram, and `outer`, `intermediate`, and `inner` are the outer, intermediate, and inner shapes, respectively.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[0,1],[1,2],[1]])
sage: pm.pm_diagram
[[0, 1], [1, 2], [1]]
sage: pm._list
[1, 1, 2, 0, 1]
sage: pm.n
2
sage: pm.width
5
```

(continues on next page)

(continued from previous page)

```
sage: pm.pp()
.   .   .   .
.   +   -   -
sage: PMDiagram([2,5,[4,4],[4,2],[4,1]], from_shapes=True)
[[0, 1], [1, 2], [1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
>>> pm = PMDiagram([[Integer(0),Integer(1)],[Integer(1),Integer(2)],[Integer(1)]])
>>> pm.pm_diagram
[[0, 1], [1, 2], [1]]
>>> pm._list
[1, 1, 2, 0, 1]
>>> pm.n
2
>>> pm.width
5
>>> pm.pp()
.   .   .   .
.   +   -   -
>>> PMDiagram([Integer(2),Integer(5),[Integer(4),Integer(4)],[Integer(4),
↪Integer(2)],[Integer(4),Integer(1)]], from_shapes=True)
[[0, 1], [1, 2], [1]]
```

heights_of_addable_plus()

Return a list with the heights of all addable plus in the $\pm$ diagram.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.heights_of_addable_plus()
[1, 1, 2, 3, 4, 5]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.heights_of_addable_plus()
[1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(2)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1),Integer(1)],
↪[Integer(1)]])
>>> pm.heights_of_addable_plus()
[1, 1, 2, 3, 4, 5]
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(1)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1)]])
>>> pm.heights_of_addable_plus()
[1, 2, 3, 4]
```

heights_of_minus()

Return a list with the heights of all minus in the $\pm$ diagram.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.heights_of_minus()
[5, 5, 3, 3, 1, 1]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.heights_of_minus()
[4, 4, 2, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(2)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1),Integer(1)],
↪[Integer(1)]])
>>> pm.heights_of_minus()
[5, 5, 3, 3, 1, 1]
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(1)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1)]])
>>> pm.heights_of_minus()
[4, 4, 2, 2]
```

inner_shape()

Return the inner shape of the pm diagram.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[0,1],[1,2],[1]])
sage: pm.inner_shape()
[4, 1]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.inner_shape()
[7, 5, 3, 1]
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.inner_shape()
[10, 7, 5, 3, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
>>> pm = PMDiagram([[Integer(0),Integer(1)],[Integer(1),Integer(2)],
↪[Integer(1)]])
>>> pm.inner_shape()
[4, 1]
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(1)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1)]])
>>> pm.inner_shape()
[7, 5, 3, 1]
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(2)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1),Integer(1)],
↪[Integer(1)]])
>>> pm.inner_shape()
[10, 7, 5, 3, 1]
```

intermediate_shape()

Return the intermediate shape of the pm diagram (inner shape plus positions of plusses).

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[0,1],[1,2],[1]])
sage: pm.intermediate_shape()
[4, 2]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.intermediate_shape()
[8, 6, 4, 2]
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.intermediate_shape()
[11, 8, 6, 4, 2]
sage: pm = PMDiagram([[1,0],[0,1],[2,0],[0,0],[0]])
sage: pm.intermediate_shape()
[4, 2, 2]
sage: pm = PMDiagram([[1, 0], [0, 0], [0, 0], [0, 0], [0]])
sage: pm.intermediate_shape()
[1]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
>>> pm = PMDiagram([[Integer(0),Integer(1)],[Integer(1),Integer(2)],
↪[Integer(1)]])
>>> pm.intermediate_shape()
[4, 2]
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(1)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1)]])
>>> pm.intermediate_shape()
[8, 6, 4, 2]
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(2)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1),Integer(1)],
↪[Integer(1)]])
>>> pm.intermediate_shape()
[11, 8, 6, 4, 2]
>>> pm = PMDiagram([[Integer(1),Integer(0)],[Integer(0),Integer(1)],
↪[Integer(2),Integer(0)],[Integer(0),Integer(0)],[Integer(0)]])
>>> pm.intermediate_shape()
[4, 2, 2]
>>> pm = PMDiagram([[Integer(1), Integer(0)], [Integer(0), Integer(0)],␣
↪[Integer(0), Integer(0)], [Integer(0), Integer(0)], [Integer(0)]])
>>> pm.intermediate_shape()
[1]
```

outer_shape()

Return the outer shape of the $\pm$ diagram.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[0,1],[1,2],[1]])
sage: pm.outer_shape()
[4, 4]
```

(continues on next page)

(continued from previous page)

```
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.outer_shape()
[8, 8, 4, 4]
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.outer_shape()
[13, 8, 8, 4, 4]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
>>> pm = PMDiagram([[Integer(0),Integer(1)],[Integer(1),Integer(2)],
↪[Integer(1)]])
>>> pm.outer_shape()
[4, 4]
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(1)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1)]])
>>> pm.outer_shape()
[8, 8, 4, 4]
>>> pm = PMDiagram([[Integer(1),Integer(2)],[Integer(1),Integer(2)],
↪[Integer(1),Integer(1)],[Integer(1),Integer(1)],[Integer(1),Integer(1)],
↪[Integer(1)]])
>>> pm.outer_shape()
[13, 8, 8, 4, 4]
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[1,0],[0,1],[2,0],[0,0],[0]])
sage: pm.pp()
.  .  .  +
.  .  -  -
+  +
-  -
sage: pm = PMDiagram([[0,2], [0,0], [0]])
sage: pm.pp()
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
>>> pm = PMDiagram([[Integer(1),Integer(0)],[Integer(0),Integer(1)],
↪[Integer(2),Integer(0)],[Integer(0),Integer(0)],[Integer(0)]])
>>> pm.pp()
.  .  .  +
.  .  -  -
+  +
-  -
>>> pm = PMDiagram([[Integer(0),Integer(2)], [Integer(0),Integer(0)],␣
↪[Integer(0)]])
>>> pm.pp()
```

sigma()

Return sigma on pm diagrams as needed for the analogue of the Dynkin diagram automorphism that interchanges nodes 0 and 1 for type $D_n(1)$, $B_n(1)$, $A_{2n-1}(2)$ for Kirillov-Reshetikhin crystals.

EXAMPLES:

```
sage: pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[0,1],[1,2],
↪[1]])
sage: pm.sigma()
[[1, 0], [2, 1], [1]]
```

```
>>> from sage.all import *
>>> pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[Integer(0),
↪Integer(1)],[Integer(1),Integer(2)],[Integer(1)]])
>>> pm.sigma()
[[1, 0], [2, 1], [1]]
```

`sage.combinat.crystals.kirillov_reshetikhin.`**`horizontal_dominoes_removed`**`(r, s)`

Return all partitions obtained from a rectangle of width s and height r by removing horizontal dominoes.

EXAMPLES:

```
sage: sage.combinat.crystals.kirillov_reshetikhin.horizontal_dominoes_removed(2,2)
[[], [2], [2, 2]]
sage: sage.combinat.crystals.kirillov_reshetikhin.horizontal_dominoes_removed(3,2)
[[], [2], [2, 2], [2, 2, 2]]
```

```
>>> from sage.all import *
>>> sage.combinat.crystals.kirillov_reshetikhin.horizontal_dominoes_
↪removed(Integer(2),Integer(2))
[[], [2], [2, 2]]
>>> sage.combinat.crystals.kirillov_reshetikhin.horizontal_dominoes_
↪removed(Integer(3),Integer(2))
[[], [2], [2, 2], [2, 2, 2]]
```

`sage.combinat.crystals.kirillov_reshetikhin.`**`partitions_in_box`**`(r, s)`

Return all partitions in a box of width s and height r.

EXAMPLES:

```
sage: sage.combinat.crystals.kirillov_reshetikhin.partitions_in_box(3,2)
[[], [1], [2], [1, 1], [2, 1], [1, 1, 1], [2, 2], [2, 1, 1],
[2, 2, 1], [2, 2, 2]]
```

```
>>> from sage.all import *
>>> sage.combinat.crystals.kirillov_reshetikhin.partitions_in_box(Integer(3),
↪Integer(2))
[[], [1], [2], [1, 1], [2, 1], [1, 1, 1], [2, 2], [2, 1, 1],
[2, 2, 1], [2, 2, 2]]
```

`sage.combinat.crystals.kirillov_reshetikhin.`**`vertical_dominoes_removed`**`(r, s)`

Return all partitions obtained from a rectangle of width s and height r by removing vertical dominoes.

EXAMPLES:

```
sage: sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_removed(2,2)
[[], [1, 1], [2, 2]]
sage: sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_removed(3,2)
[[2], [2, 1, 1], [2, 2, 2]]
sage: sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_removed(4,2)
[[], [1, 1], [1, 1, 1, 1], [2, 2], [2, 2, 1, 1], [2, 2, 2, 2]]
```

```
>>> from sage.all import *
>>> sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_
↪removed(Integer(2),Integer(2))
[[], [1, 1], [2, 2]]
>>> sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_
↪removed(Integer(3),Integer(2))
[[2], [2, 1, 1], [2, 2, 2]]
>>> sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_
↪removed(Integer(4),Integer(2))
[[], [1, 1], [1, 1, 1, 1], [2, 2], [2, 2, 1, 1], [2, 2, 2, 2]]
```

5.1.56 Kyoto Path Model for Affine Highest Weight Crystals

class sage.combinat.crystals.kyoto_path_model.**KyotoPathModel**(*crystals*, *weight*, *P*)

Bases: *TensorProductOfCrystals*

The Kyoto path model for an affine highest weight crystal.

Note

Here we are using anti-Kashiwara notation and might differ from some of the literature.

Consider a Kac–Moody algebra $\mathfrak{g}$ of affine Cartan type X, and we want to model the $U_q'(\mathfrak{g})$-crystal $B(\lambda)$. First we consider the set of fundamental weights $\{\Lambda_i\}_{i\in I}$ of $\mathfrak{g}$ and let $\{\overline{\Lambda}_i\}_{i\in I_0}$ be the corresponding fundamental weights of the corresponding classical Lie algebra $\mathfrak{g}_0$. To model $B(\lambda)$, we start with a sequence of perfect $U_q'(\mathfrak{g})$-crystals $(B^{(i)})_i$ of level l such that

$$\lambda \in \overline{P}_l^+ = \left\{\mu \in \overline{P}^+ \mid \langle c, \mu\rangle = l\right\}$$

where c is the canonical central element of $U_q'(\mathfrak{g})$ and $\overline{P}^+$ is the nonnegative weight lattice spanned by $\{\overline{\Lambda}_i \mid i \in I\}$.

Next we consider the crystal isomorphism $\Phi_0 : B(\lambda_0) \to B^{(0)} \otimes B(\lambda_1)$ defined by $u_{\lambda_0} \mapsto b_{\lambda_0}^{(0)} \otimes u_{\lambda_1}$ where $b_{\lambda_0}^{(0)}$ is the unique element in $B^{(0)}$ such that $\varphi\left(b_{\lambda_0}^{(0)}\right) = \lambda_0$ and $\lambda_1 = \varepsilon\left(b_{\lambda_0}^{(0)}\right)$ and u_μ is the highest weight element in $B(\mu)$. Iterating this, we obtain the following isomorphism:

$$\Phi_n : B(\lambda) \to B^{(0)} \otimes B^{(1)} \otimes \cdots \otimes B^{(N)} \otimes B(\lambda_{N+1}).$$

We note by Lemma 10.6.2 in [HK2002] that for any $b \in B(\lambda)$ there exists a finite N such that

$$\Phi_N(b) = \left(\bigotimes_{k=0}^{N-1} b^{(k)}\right) \otimes u_{\lambda_N}.$$

Therefore we can model elements $b \in B(\lambda)$ as a $U_q'(\mathfrak{g})$-crystal by considering an infinite list of elements $b^{(k)} \in$

$B^{(k)}$ and defining the crystal structure by:

$$\begin{aligned}\overline{\mathrm{wt}}(b) &= \lambda_N + \sum_{k=0}^{N-1} \overline{\mathrm{wt}}\left(b^{(k)}\right)\\ e_i(b) &= e_i\left(b' \otimes b^{(N)}\right) \otimes u_{\lambda_N},\\ f_i(b) &= f_i\left(b' \otimes b^{(N)}\right) \otimes u_{\lambda_N},\\ \varepsilon_i(b) &= \max\left(\varepsilon_i(b') - \varphi_i\left(b^{(N)}\right), 0\right),\\ \varphi_i(b) &= \varphi_i(b') + \max\left(\varphi_i\left(b^{(N)}\right) - \varepsilon_i(b'), 0\right),\end{aligned}$$

where $b' = b^{(0)} \otimes \cdots \otimes b^{(N-1)}$. To translate this into a finite list, we consider a finite sequence $b^{(0)} \otimes \cdots \otimes b^{(N-1)} \otimes b^{(N)}_{\lambda_N}$ and if

$$f_i\left(b^{(0)} \otimes \cdots b^{(N-1)} \otimes b^{(N)}_{\lambda_N}\right) = b_0 \otimes \cdots \otimes b^{(N-1)} \otimes f_i\left(b^{(N)}_{\lambda_N}\right),$$

then we take the image as $b^{(0)} \otimes \cdots \otimes f_i\left(b^{(N)}_{\lambda_N}\right) \otimes b^{(N+1)}_{\lambda_{N+1}}$. Similarly we remove $b^{(N)}_{\lambda_N}$ if we have $b_0 \otimes \cdots \otimes b^{(N-1)} \otimes b^{(N-1)}_{\lambda_{N-1}} \otimes b^{(N)}_{\lambda_N}$. Additionally if

$$e_i\left(b^{(0)} \otimes \cdots \otimes b^{(N-1)} \otimes b^{(N)}_{\lambda_N}\right) = b^{(0)} \otimes \cdots \otimes b^{(N-1)} \otimes e_i\left(b^{(N)}_{\lambda_N}\right),$$

then we consider this to be 0.

We can then lift the $U_q'(\mathfrak{g})$-crystal structure to a $U_q(\mathfrak{g})$-crystal structure by using a tensor product of the `affinization` of the of crystals $B^{(i)}$ for all i.

INPUT:

- `B` – a single or list of U_q' perfect crystal(s) of level l
- `weight` – a weight in $\overline{P}^+_l$

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]; mg
[[[3]]]
sage: mg.f_string([0,1,2,2])
[[[3]], [[3]], [[1]]]
sage: x = mg.f_string([0,1,2]); x
[[[2]], [[3]], [[1]]]
sage: x.weight()
Lambda[0]
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)], Integer(1),
↪Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().fundamental_
↪weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> mg = C.module_generators[Integer(0)]; mg
[[[3]]]
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2)])
[[[3]], [[3]], [[1]]]
>>> x = mg.f_string([Integer(0),Integer(1),Integer(2)]); x
[[[2]], [[3]], [[1]]]
>>> x.weight()
Lambda[0]
```

An example of type $A_5^{(2)}$:

```
sage: B = crystals.KirillovReshetikhin(['A',5,2], 1,1)
sage: La = RootSystem(['A',5,2]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]; mg
[[[-1]]]
sage: mg.f_string([0,2,1,3])
[[[-3]], [[2]], [[-1]]]
sage: mg.f_string([0,2,3,1])
[[[-3]], [[2]], [[-1]]]
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(5),Integer(2)], Integer(1),
↪Integer(1))
>>> La = RootSystem(['A',Integer(5),Integer(2)]).weight_lattice().fundamental_
↪weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]; mg
[[[-1]]]
>>> mg.f_string([Integer(0),Integer(2),Integer(1),Integer(3)])
[[[-3]], [[2]], [[-1]]]
>>> mg.f_string([Integer(0),Integer(2),Integer(3),Integer(1)])
[[[-3]], [[2]], [[-1]]]
```

An example of type $D_3^{(2)}$:

```
sage: B = crystals.KirillovReshetikhin(['D',3,2], 1,1)
sage: La = RootSystem(['D',3,2]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]; mg
[[]]
sage: mg.f_string([0,1,2,0])
[[[0]], [[1]], []]
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['D',Integer(3),Integer(2)], Integer(1),
↪Integer(1))
>>> La = RootSystem(['D',Integer(3),Integer(2)]).weight_lattice().fundamental_
↪weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]; mg
[[]]
```

(continues on next page)

(continued from previous page)

```
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(0)])
[[[0]], [[1]], []]
```

An example using multiple crystals of the same level:

```
sage: B1 = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: B2 = crystals.KirillovReshetikhin(['A',2,1], 2,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel([B1, B2, B1], La[0])
sage: mg = C.module_generators[0]; mg
[[[3]]]
sage: mg.f_string([0,1,2,2])
[[[3]], [[1], [3]], [[3]]]
sage: mg.f_string([0,1,2,2,2])
sage: mg.f_string([0,1,2,2,1,0])
[[[3]], [[2], [3]], [[1]], [[2]]]
sage: mg.f_string([0,1,2,2,1,0,0,2])
[[[3]], [[1], [2]], [[1]], [[3]], [[1], [3]]]
```

```
>>> from sage.all import *
>>> B1 = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)], Integer(1),
↪Integer(1))
>>> B2 = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)], Integer(2),
↪Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().fundamental_
↪weights()
>>> C = crystals.KyotoPathModel([B1, B2, B1], La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]; mg
[[[3]]]
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2)])
[[[3]], [[1], [3]], [[3]]]
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2),Integer(2)])
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2),Integer(1),
↪Integer(0)])
[[[3]], [[2], [3]], [[1]], [[2]]]
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2),Integer(1),
↪Integer(0),Integer(0),Integer(2)])
[[[3]], [[1], [2]], [[1]], [[3]], [[1], [3]]]
```

By using the extended weight lattice, the Kyoto path model lifts the perfect crystals to their affinizations:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: P = RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: La = P.fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]; mg
[[[3]](0)]
sage: x = mg.f_string([0,1,2]); x
[[[2]](-1), [[3]](0), [[1]](0)]
sage: x.weight()
Lambda[0] - delta
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)], Integer(1),
↪Integer(1))
>>> P = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice(extended=True)
>>> La = P.fundamental_weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]; mg
[[[3]](0)]
>>> x = mg.f_string([Integer(0),Integer(1),Integer(2)]); x
[[[2]](-1), [[3]](0), [[1]](0)]
>>> x.weight()
Lambda[0] - delta
```

class Element

Bases: *TensorProductOfRegularCrystalsElement*

An element in the Kyoto path model.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: all(mg.e(i) is None for i in C.index_set())
True
sage: mg.f(0).e(0) == mg
True
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(1),Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().
↪fundamental_weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]
>>> all(mg.e(i) is None for i in C.index_set())
True
>>> mg.f(Integer(0)).e(Integer(0)) == mg
True
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: [mg.epsilon(i) for i in C.index_set()]
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0]
sage: elt = mg.f(0)
sage: [elt.epsilon(i) for i in C.index_set()]
[1, 0, 0]
sage: elt = mg.f_string([0,1,2])
sage: [elt.epsilon(i) for i in C.index_set()]
[0, 0, 1]
sage: elt = mg.f_string([0,1,2,2])
sage: [elt.epsilon(i) for i in C.index_set()]
[0, 0, 2]
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(1),Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().
↪fundamental_weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]
>>> [mg.epsilon(i) for i in C.index_set()]
[0, 0, 0]
>>> elt = mg.f(Integer(0))
>>> [elt.epsilon(i) for i in C.index_set()]
[1, 0, 0]
>>> elt = mg.f_string([Integer(0),Integer(1),Integer(2)])
>>> [elt.epsilon(i) for i in C.index_set()]
[0, 0, 1]
>>> elt = mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2)])
>>> [elt.epsilon(i) for i in C.index_set()]
[0, 0, 2]
```

f(i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: mg.f(2)
sage: mg.f(0)
[[[1]], [[2]]]
sage: mg.f_string([0,1,2])
[[[2]], [[3]], [[1]]]
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(1),Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().
↪fundamental_weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]
```

(continues on next page)

(continued from previous page)

```
>>> mg.f(Integer(2))
>>> mg.f(Integer(0))
[[[1]], [[2]]]
>>> mg.f_string([Integer(0),Integer(1),Integer(2)])
[[[2]], [[3]], [[1]]]
```

phi (*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: [mg.phi(i) for i in C.index_set()]
[1, 0, 0]
sage: elt = mg.f(0)
sage: [elt.phi(i) for i in C.index_set()]
[0, 1, 1]
sage: elt = mg.f_string([0,1])
sage: [elt.phi(i) for i in C.index_set()]
[0, 0, 2]
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(1),Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().
↪fundamental_weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]
>>> [mg.phi(i) for i in C.index_set()]
[1, 0, 0]
>>> elt = mg.f(Integer(0))
>>> [elt.phi(i) for i in C.index_set()]
[0, 1, 1]
>>> elt = mg.f_string([Integer(0),Integer(1)])
>>> [elt.phi(i) for i in C.index_set()]
[0, 0, 2]
```

truncate (*k=None*)

Truncate `self` to have length `k` and return as an element in a (finite) tensor product of crystals.

INPUT:

- `k` – (optional) the length to truncate to; if not specified, then returns one more than the current non-ground-state elements (i.e. the current list in `self`)

EXAMPLES:

```
sage: B1 = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: B2 = crystals.KirillovReshetikhin(['A',2,1], 2,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel([B1,B2,B1], La[0])
sage: mg = C.highest_weight_vector()
```

(continues on next page)

(continued from previous page)

```
sage: elt = mg.f_string([0,1,2,2,1,0]); elt
[[[3]], [[2], [3]], [[1]], [[2]]]
sage: t = elt.truncate(); t
[[[3]], [[2], [3]], [[1]], [[2]]]
sage: t.parent() is C.finite_tensor_product(4)
True
sage: elt.truncate(2)
[[[3]], [[2], [3]]]
sage: elt.truncate(10)
[[[3]], [[2], [3]], [[1]], [[2]], [[1], [3]],
 [[2]], [[1]], [[2], [3]], [[1]], [[3]]]
```

```
>>> from sage.all import *
>>> B1 = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(1),Integer(1))
>>> B2 = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(2),Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().
↪fundamental_weights()
>>> C = crystals.KyotoPathModel([B1,B2,B1], La[Integer(0)])
>>> mg = C.highest_weight_vector()
>>> elt = mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2),
↪Integer(1),Integer(0)]); elt
[[[3]], [[2], [3]], [[1]], [[2]]]
>>> t = elt.truncate(); t
[[[3]], [[2], [3]], [[1]], [[2]]]
>>> t.parent() is C.finite_tensor_product(Integer(4))
True
>>> elt.truncate(Integer(2))
[[[3]], [[2], [3]]]
>>> elt.truncate(Integer(10))
[[[3]], [[2], [3]], [[1]], [[2]], [[1], [3]],
 [[2]], [[1]], [[2], [3]], [[1]], [[3]]]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: P = RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: La = P.fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: mg.weight()
Lambda[0]
sage: mg.f_string([0,1,2]).weight()
Lambda[0] - delta
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],␣
↪Integer(1),Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> P = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True)
>>> La = P.fundamental_weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> mg = C.module_generators[Integer(0)]
>>> mg.weight()
Lambda[0]
>>> mg.f_string([Integer(0),Integer(1),Integer(2)]).weight()
Lambda[0] - delta
```

finite_tensor_product(*k*)

Return the finite tensor product of crystals of length `k` from truncating `self`.

EXAMPLES:

```
sage: B1 = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: B2 = crystals.KirillovReshetikhin(['A',2,1], 2,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel([B1,B2,B1], La[0])
sage: C.finite_tensor_product(5)
Full tensor product of the crystals
 [Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
  Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(2,1),
  Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
  Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
  Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(2,1)]
```

```
>>> from sage.all import *
>>> B1 = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)], Integer(1),
↪Integer(1))
>>> B2 = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)], Integer(2),
↪Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().fundamental_
↪weights()
>>> C = crystals.KyotoPathModel([B1,B2,B1], La[Integer(0)])
>>> C.finite_tensor_product(Integer(5))
Full tensor product of the crystals
 [Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
  Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(2,1),
  Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
  Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
  Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(2,1)]
```

weight_lattice_realization()

Return the weight lattice realization used to express weights.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: C.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: P = RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: C = crystals.KyotoPathModel(B, P.fundamental_weight(0))
sage: C.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

```
>>> from sage.all import *
>>> B = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)], Integer(1),
↪Integer(1))
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice().fundamental_
↪weights()
>>> C = crystals.KyotoPathModel(B, La[Integer(0)])
>>> C.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]

>>> P = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice(extended=True)
>>> C = crystals.KyotoPathModel(B, P.fundamental_weight(Integer(0)))
>>> C.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

5.1.57 Crystals of letters

class sage.combinat.crystals.letters.**BKKLetter**

Bases: *Letter*

e(*i*)

Return the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['A', [2, 1]])
sage: c = C(-2)
sage: c.e(-2)
-3
sage: c = C(1)
sage: c.e(0)
-1
sage: c = C(2)
sage: c.e(1)
1
sage: c.e(-2)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', [Integer(2), Integer(1)]])
>>> c = C(-Integer(2))
>>> c.e(-Integer(2))
-3
>>> c = C(Integer(1))
>>> c.e(Integer(0))
-1
>>> c = C(Integer(2))
>>> c.e(Integer(1))
```

(continues on next page)

(continued from previous page)

```
1
>>> c.e(-Integer(2))
```

f(i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A', [2, 1]])
sage: c = C.an_element()
sage: c.f(-2)
-2
sage: c = C(-1)
sage: c.f(0)
1
sage: c = C(1)
sage: c.f(1)
2
sage: c.f(-2)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', [Integer(2), Integer(1)]])
>>> c = C.an_element()
>>> c.f(-Integer(2))
-2
>>> c = C(-Integer(1))
>>> c.f(Integer(0))
1
>>> c = C(Integer(1))
>>> c.f(Integer(1))
2
>>> c.f(-Integer(2))
```

weight()

Return weight of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A', [2, 1]])
sage: c = C(-1)
sage: c.weight()
(0, 0, 1, 0, 0)
sage: c = C(2)
sage: c.weight()
(0, 0, 0, 0, 1)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', [Integer(2), Integer(1)]])
>>> c = C(-Integer(1))
>>> c.weight()
(0, 0, 1, 0, 0)
>>> c = C(Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> c.weight()
(0, 0, 0, 0, 1)
```

class sage.combinat.crystals.letters.**ClassicalCrystalOfLetters**(*cartan_type*, *element_class*, *element_print_style=None*, *dual=None*)

Bases: UniqueRepresentation, Parent

A generic class for classical crystals of letters.

All classical crystals of letters should be instances of this class or of subclasses. To define a new crystal of letters, one only needs to implement a class for the elements (which subclasses Letter), with appropriate e_i and f_i operations. If the module generator is not 1, one also needs to define the subclass *ClassicalCrystalOfLetters* for the crystal itself.

The basic assumption is that crystals of letters are small, but used intensively as building blocks. Therefore, we explicitly build in memory the list of all elements, the crystal graph and its transitive closure, so as to make the following operations constant time: list, cmp, (todo: phi, epsilon, e, and f with caching)

list()

Return a list of the elements of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: C.list()
[1, 2, 3, 4, 5, 6]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> C.list()
[1, 2, 3, 4, 5, 6]
```

lt_elements(*x*, *y*)

Return True if and only if there is a path from x to y in the crystal graph, when x is not equal to y.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = crystals.Letters(['A', 5])
sage: x = C(1)
sage: y = C(2)
sage: C.lt_elements(x,y)
True
sage: C.lt_elements(y,x)
False
sage: C.lt_elements(x,x)
False
sage: C = crystals.Letters(['D', 4])
sage: C.lt_elements(C(4),C(-4))
False
sage: C.lt_elements(C(-4),C(4))
False
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', Integer(5)])
>>> x = C(Integer(1))
>>> y = C(Integer(2))
>>> C.lt_elements(x,y)
True
>>> C.lt_elements(y,x)
False
>>> C.lt_elements(x,x)
False
>>> C = crystals.Letters(['D', Integer(4)])
>>> C.lt_elements(C(Integer(4)),C(-Integer(4)))
False
>>> C.lt_elements(C(-Integer(4)),C(Integer(4)))
False
```

class sage.combinat.crystals.letters.**ClassicalCrystalOfLettersWrapped**(*cartan_type*)

Bases: *ClassicalCrystalOfLetters*

Crystal of letters by wrapping another crystal.

This is used for a crystal of letters of type E_8 and F_4.

This class follows the same output as the other crystal of letters, where b is represented by the "letter" with $\varphi_i(b)$ (resp., ε_i) number of i's (resp., $-i$'s or $\bar{i}$'s). However, this uses an auxiliary crystal to construct these letters to avoid hardcoding the crystal elements and the corresponding edges; in particular, the 248 nodes of E_8.

class sage.combinat.crystals.letters.**CrystalOfBKKLetters**(*ct*, *dual*)

Bases: *ClassicalCrystalOfLetters*

Crystal of letters for Benkart-Kang-Kashiwara supercrystals.

This implements the $\mathfrak{gl}(m|n)$ crystal of Benkart, Kang and Kashiwara [BKK2000].

EXAMPLES:

```
sage: C = crystals.Letters(['A', [1, 1]]); C
The crystal of letters for type ['A', [1, 1]]

sage: C = crystals.Letters(['A', [2,4]], dual=True); C
The crystal of letters for type ['A', [2, 4]] (dual)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', [Integer(1), Integer(1)]]); C
The crystal of letters for type ['A', [1, 1]]

>>> C = crystals.Letters(['A', [Integer(2),Integer(4)]], dual=True); C
The crystal of letters for type ['A', [2, 4]] (dual)
```

Element

alias of *BKKLetter*

sage.combinat.crystals.letters.**CrystalOfLetters**(*cartan_type*, *element_print_style=None*, *dual=None*)

Return the crystal of letters of the given type.

For classical types, this is a combinatorial model for the crystal with highest weight Λ_1 (the first fundamental weight).

Any irreducible classical crystal appears as the irreducible component of the tensor product of several copies of this crystal (plus possibly one copy of the spin crystal, see `CrystalOfSpins`). See [KN1994]. Elements of this irreducible component have a fixed shape, and can be fit inside a tableau shape. Otherwise said, any irreducible classical crystal is isomorphic to a crystal of tableaux with cells filled by elements of the crystal of letters (possibly tensored with the crystal of spins).

We also have the crystal of fundamental representation of the general linear Lie superalgebra, which are used as letters inside of tableaux following [BKK2000]. Similarly, all of these crystals appear as a subcrystal of a sufficiently large tensor power of this crystal.

INPUT:

- `T` – a Cartan type

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: C.list()
[1, 2, 3, 4, 5, 6]
sage: C.cartan_type()
['A', 5]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> C.list()
[1, 2, 3, 4, 5, 6]
>>> C.cartan_type()
['A', 5]
```

For type E_6, one can also specify how elements are printed. This option is usually set to None and the default representation is used. If one chooses the option 'compact', the elements are printed in the more compact convention with 27 letters `+abcdefghijklmnopqrstuvwxyz` and the 27 letters `-ABCDEFGHIJKLMNOPQRSTUVWXYZ` for the dual crystal.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], element_print_style = 'compact')
sage: C
The crystal of letters for type ['E', 6]
sage: C.list()
[+, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
sage: C = crystals.Letters(['E',6], element_print_style = 'compact', dual = True)
sage: C
The crystal of letters for type ['E', 6] (dual)
sage: C.list()
[-, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)], element_print_style = 'compact')
>>> C
The crystal of letters for type ['E', 6]
>>> C.list()
[+, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
>>> C = crystals.Letters(['E',Integer(6)], element_print_style = 'compact', dual␣
↪= True)
>>> C
```

(continues on next page)

(continued from previous page)

```
The crystal of letters for type ['E', 6] (dual)
>>> C.list()
[-, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]
```

class sage.combinat.crystals.letters.**CrystalOfQueerLetters**(*ct*)

Bases: *ClassicalCrystalOfLetters*

Queer crystal of letters elements.

The index set is of the form $\{-n, \ldots, -1, 1, \ldots, n\}$. For $1 < i \leq n$, the operators e_{-i} and f_{-i} are defined as

$$f_{-i} = s_{w_i^{-1}} f_{-1} s_{w_i}, \quad e_{-i} = s_{w_i^{-1}} e_{-1} s_{w_i},$$

where $w_i = s_2 \cdots s_i s_1 \cdots s_{i-1}$ and s_i is the reflection along the i-string in the crystal. See [GJK+2014].

Element

alias of *QueerLetter_element*

index_set()

Return index set of `self`.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q',3])
sage: Q.index_set()
(1, 2, -2, -1)
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q',Integer(3)])
>>> Q.index_set()
(1, 2, -2, -1)
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_A_element**

Bases: *Letter*

Type A crystal of letters elements.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',4])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1), (3, 2, 2), (4, 3, 3), (5, 4, 4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(4)])
>>> [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1), (3, 2, 2), (4, 3, 3), (5, 4, 4)]
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',4])
sage: [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1), (3, 2), (4, 3), (5, 4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(4)])
>>> [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != Integer(0)]
[(2, 1), (3, 2), (4, 3), (5, 4)]
```

f(i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',4])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2), (2, 2, 3), (3, 3, 4), (4, 4, 5)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(4)])
>>> [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2), (2, 2, 3), (3, 3, 4), (4, 4, 5)]
```

phi(i)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',4])
sage: [(c,i) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1), (2, 2), (3, 3), (4, 4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(4)])
>>> [(c,i) for i in C.index_set() for c in C if c.phi(i) != Integer(0)]
[(1, 1), (2, 2), (3, 3), (4, 4)]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['A',3])]
[(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Letters(['A',Integer(3)])]
[(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_B_element**

Bases: *Letter*

Type B crystal of letters elements.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['B',4])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (0, 4, 4),
 (-4, 4, 0)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['B',Integer(4)])
>>> [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (0, 4, 4),
 (-4, 4, 0)]
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['B',3])
sage: [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (0, 3), (-3, 3)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['B',Integer(3)])
>>> [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != Integer(0)]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (0, 3), (-3, 3)]
```

f(*i*)

Return the actions of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['B',4])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2),
 (3, 3, 4),
```

(continues on next page)

(continued from previous page)

```
 (-4, 3, -3),
 (4, 4, 0),
 (0, 4, -4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['B',Integer(4)])
>>> [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2),
 (3, 3, 4),
 (-4, 3, -3),
 (4, 4, 0),
 (0, 4, -4)]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['B',3])
sage: [(c,i) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3), (0, 3)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['B',Integer(3)])
>>> [(c,i) for i in C.index_set() for c in C if c.phi(i) != Integer(0)]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3), (0, 3)]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['B',3])]
[(1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
 (0, 0, 0),
 (0, 0, -1),
 (0, -1, 0),
 (-1, 0, 0)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Letters(['B',Integer(3)])]
[(1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
 (0, 0, 0),
 (0, 0, -1),
 (0, -1, 0),
 (-1, 0, 0)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_C_element**

Bases: *Letter*

Type C crystal of letters elements.

e(i)

Return the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['C',4])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (-4, 4, 4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C',Integer(4)])
>>> [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (-4, 4, 4)]
```

epsilon(i)

Return ε_i of self.

EXAMPLES:

```
sage: C = crystals.Letters(['C',3])
sage: [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (-3, 3)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C',Integer(3)])
>>> [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != Integer(0)]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (-3, 3)]
```

f(i)

Return the action of f_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['C',4])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2), (-2, 1, -1), (2, 2, 3),
 (-3, 2, -2), (3, 3, 4), (-4, 3, -3), (4, 4, -4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C',Integer(4)])
>>> [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2), (-2, 1, -1), (2, 2, 3),
 (-3, 2, -2), (3, 3, 4), (-4, 3, -3), (4, 4, -4)]
```

phi(i)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['C',3])
sage: [(c,i) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C',Integer(3)])
>>> [(c,i) for i in C.index_set() for c in C if c.phi(i) != Integer(0)]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3)]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['C',3])]
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 0, -1), (0, -1, 0), (-1, 0, 0)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Letters(['C',Integer(3)])]
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 0, -1), (0, -1, 0), (-1, 0, 0)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_D_element**

Bases: *Letter*

Type D crystal of letters elements.

e(i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['D',5])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (5, 4, 4),
 (-4, 4, -5),
 (-5, 5, 4),
 (-4, 5, 5)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['D',Integer(5)])
>>> [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (5, 4, 4),
 (-4, 4, -5),
 (-5, 5, 4),
 (-4, 5, 5)]
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['D',4])
sage: [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (4, 3), (-3, 3), (-4, 4), (-3, 4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['D',Integer(4)])
>>> [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != Integer(0)]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (4, 3), (-3, 3), (-4, 4), (-3, 4)]
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['D',5])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2),
 (3, 3, 4),
 (-4, 3, -3),
 (4, 4, 5),
 (-5, 4, -4),
 (4, 5, -5),
 (5, 5, -4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['D',Integer(5)])
>>> [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2),
```

(continues on next page)

(continued from previous page)

```
 (3, 3, 4),
 (-4, 3, -3),
 (4, 4, 5),
 (-5, 4, -4),
 (4, 5, -5),
 (5, 5, -4)]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['D',4])
sage: [(c,i) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3), (-4, 3), (3, 4), (4, 4)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['D',Integer(4)])
>>> [(c,i) for i in C.index_set() for c in C if c.phi(i) != Integer(0)]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3), (-4, 3), (3, 4), (4, 4)]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['D',4])]
[(1, 0, 0, 0),
 (0, 1, 0, 0),
 (0, 0, 1, 0),
 (0, 0, 0, 1),
 (0, 0, 0, -1),
 (0, 0, -1, 0),
 (0, -1, 0, 0),
 (-1, 0, 0, 0)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Letters(['D',Integer(4)])]
[(1, 0, 0, 0),
 (0, 1, 0, 0),
 (0, 0, 1, 0),
 (0, 0, 0, 1),
 (0, 0, 0, -1),
 (0, 0, -1, 0),
 (0, -1, 0, 0),
 (-1, 0, 0, 0)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_E6_element**

Bases: *LetterTuple*

Type E_6 crystal of letters elements. This crystal corresponds to the highest weight crystal $B(\Lambda_1)$.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: C((-1,3)).e(1)
(1,)
sage: C((-2,-3,4)).e(2)
(-3, 2)
sage: C((1,)).e(1)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)])
>>> C((-Integer(1),Integer(3))).e(Integer(1))
(1,)
>>> C((-Integer(2),-Integer(3),Integer(4))).e(Integer(2))
(-3, 2)
>>> C((Integer(1),)).e(Integer(1))
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: C((1,)).f(1)
(-1, 3)
sage: C((-6,)).f(1)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)])
>>> C((Integer(1),)).f(Integer(1))
(-1, 3)
>>> C((-Integer(6),)).f(Integer(1))
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['E',6])]
[(0, 0, 0, 0, 0, -2/3, -2/3, 2/3),
 (-1/2, 1/2, 1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, 1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, 1/2, -1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, -1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, 1/2, 1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, 1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, 1/2, -1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, -1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (0, 0, 0, 0, 1, 1/3, 1/3, -1/3),
 (1/2, 1/2, 1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, 1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (-1/2, 1/2, -1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, -1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 0, 0, 1, 0, 1/3, 1/3, -1/3),
```

(continues on next page)

(continued from previous page)

```
 (-1/2, 1/2, 1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, 1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 0, 1, 0, 0, 1/3, 1/3, -1/3),
 (1/2, 1/2, -1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 1, 0, 0, 0, 1/3, 1/3, -1/3),
 (1, 0, 0, 0, 0, 1/3, 1/3, -1/3),
 (0, -1, 0, 0, 0, 1/3, 1/3, -1/3),
 (0, 0, -1, 0, 0, 1/3, 1/3, -1/3),
 (0, 0, 0, -1, 0, 1/3, 1/3, -1/3),
 (0, 0, 0, 0, -1, 1/3, 1/3, -1/3),
 (-1/2, -1/2, -1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (-1, 0, 0, 0, 0, 1/3, 1/3, -1/3)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Letters(['E',Integer(6)])]
[(0, 0, 0, 0, 0, -2/3, -2/3, 2/3),
 (-1/2, 1/2, 1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, 1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, 1/2, -1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, -1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, 1/2, 1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, 1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, 1/2, -1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, -1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (0, 0, 0, 0, 1, 1/3, 1/3, -1/3),
 (1/2, 1/2, 1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, 1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (-1/2, 1/2, -1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, -1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 0, 0, 1, 0, 1/3, 1/3, -1/3),
 (-1/2, 1/2, 1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, 1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 0, 1, 0, 0, 1/3, 1/3, -1/3),
 (1/2, 1/2, -1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 1, 0, 0, 0, 1/3, 1/3, -1/3),
 (1, 0, 0, 0, 0, 1/3, 1/3, -1/3),
 (0, -1, 0, 0, 0, 1/3, 1/3, -1/3),
 (0, 0, -1, 0, 0, 1/3, 1/3, -1/3),
 (0, 0, 0, -1, 0, 1/3, 1/3, -1/3),
 (0, 0, 0, 0, -1, 1/3, 1/3, -1/3),
 (-1/2, -1/2, -1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (-1, 0, 0, 0, 0, 1/3, 1/3, -1/3)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_E6_element_dual**

Bases: *LetterTuple*

Type E_6 crystal of letters elements. This crystal corresponds to the highest weight crystal $B(\Lambda_6)$. This crystal is dual to $B(\Lambda_1)$ of type E_6.

e(*i*)

Return the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: C((-1,)).e(1)
(1, -3)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)], dual = True)
>>> C((-Integer(1),)).e(Integer(1))
(1, -3)
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: C((6,)).f(6)
(5, -6)
sage: C((6,)).f(1)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)], dual = True)
>>> C((Integer(6),)).f(Integer(6))
(5, -6)
>>> C((Integer(6),)).f(Integer(1))
```

lift()

Lift an element of `self` to the crystal of letters `crystals.Letters(['E',6])` by taking its inverse weight.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: b = C.module_generators[0]
sage: b.lift()
(-6,)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)], dual = True)
>>> b = C.module_generators[Integer(0)]
>>> b.lift()
(-6,)
```

retract(*p*)

Retract element `p`, which is an element in `crystals.Letters(['E',6])` to an element in `crystals.Letters(['E',6], dual=True)` by taking its inverse weight.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: Cd = crystals.Letters(['E',6], dual = True)
sage: b = Cd.module_generators[0]
sage: p = C((-1,3))
sage: b.retract(p)
(1, -3)
sage: b.retract(None)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)])
>>> Cd = crystals.Letters(['E',Integer(6)], dual = True)
>>> b = Cd.module_generators[Integer(0)]
>>> p = C((-Integer(1),Integer(3)))
>>> b.retract(p)
(1, -3)
>>> b.retract(None)
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: b=C.module_generators[0]
sage: b.weight()
(0, 0, 0, 0, 1, -1/3, -1/3, 1/3)
sage: [v.weight() for v in C]
[(0, 0, 0, 0, 1, -1/3, -1/3, 1/3),
(0, 0, 0, 1, 0, -1/3, -1/3, 1/3),
(0, 0, 1, 0, 0, -1/3, -1/3, 1/3),
(0, 1, 0, 0, 0, -1/3, -1/3, 1/3),
(-1, 0, 0, 0, 0, -1/3, -1/3, 1/3),
(1, 0, 0, 0, 0, -1/3, -1/3, 1/3),
(1/2, 1/2, 1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, -1, 0, 0, 0, -1/3, -1/3, 1/3),
(-1/2, -1/2, 1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, -1, 0, 0, -1/3, -1/3, 1/3),
(-1/2, 1/2, -1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, -1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, 0, -1, 0, -1/3, -1/3, 1/3),
(-1/2, 1/2, 1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, 1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, -1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, -1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, 0, 0, -1, -1/3, -1/3, 1/3),
(-1/2, 1/2, 1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, 1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, -1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, -1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, 1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, 1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, 1/2, -1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, -1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(0, 0, 0, 0, 0, 2/3, 2/3, -2/3)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)], dual = True)
>>> b=C.module_generators[Integer(0)]
>>> b.weight()
(0, 0, 0, 0, 1, -1/3, -1/3, 1/3)
>>> [v.weight() for v in C]
```

(continues on next page)

(continued from previous page)

```
[(0, 0, 0, 0, 1, -1/3, -1/3, 1/3),
(0, 0, 0, 1, 0, -1/3, -1/3, 1/3),
(0, 0, 1, 0, 0, -1/3, -1/3, 1/3),
(0, 1, 0, 0, 0, -1/3, -1/3, 1/3),
(-1, 0, 0, 0, 0, -1/3, -1/3, 1/3),
(1, 0, 0, 0, 0, -1/3, -1/3, 1/3),
(1/2, 1/2, 1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, -1, 0, 0, 0, -1/3, -1/3, 1/3),
(-1/2, -1/2, 1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, -1, 0, 0, -1/3, -1/3, 1/3),
(-1/2, 1/2, -1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, -1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, 0, -1, 0, -1/3, -1/3, 1/3),
(-1/2, 1/2, 1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, 1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, -1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, -1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, 0, 0, -1, -1/3, -1/3, 1/3),
(-1/2, 1/2, 1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, 1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, -1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, -1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, 1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, 1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, 1/2, -1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, -1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(0, 0, 0, 0, 0, 2/3, 2/3, -2/3)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_E7_element**

Bases: *LetterTuple*

Type E_7 crystal of letters elements. This crystal corresponds to the highest weight crystal $B(\Lambda_7)$.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',7])
sage: C((7,)).e(7)
sage: C((-7,6)).e(7)
(7,)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(7)])
>>> C((Integer(7),)).e(Integer(7))
>>> C((-Integer(7),Integer(6))).e(Integer(7))
(7,)
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',7])
sage: C((-7,)).f(7)
sage: C((7,)).f(7)
(-7, 6)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(7)])
>>> C((-Integer(7),)).f(Integer(7))
>>> C((Integer(7),)).f(Integer(7))
(-7, 6)
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['E',7])]
[(0, 0, 0, 0, 0, 1, -1/2, 1/2), (0, 0, 0, 0, 1, 0, -1/2, 1/2), (0, 0, 0,
1, 0, 0, -1/2, 1/2), (0, 0, 1, 0, 0, 0, -1/2, 1/2), (0, 1, 0, 0, 0, 0,
-1/2, 1/2), (-1, 0, 0, 0, 0, 0, -1/2, 1/2), (1, 0, 0, 0, 0, 0, -1/2,
1/2), (1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 0, 0), (0, -1, 0, 0, 0, 0, -1/2,
1/2), (-1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 0, 0), (0, 0, -1, 0, 0, 0, -1/2,
1/2), (-1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 0, 0), (1/2, -1/2, -1/2, 1/2,
1/2, 1/2, 0, 0), (0, 0, 0, -1, 0, 0, -1/2, 1/2), (-1/2, 1/2, 1/2, -1/2,
1/2, 1/2, 0, 0), (1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 0, 0), (1/2, 1/2,
-1/2, -1/2, 1/2, 1/2, 0, 0), (-1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 0, 0),
(0, 0, 0, 0, -1, 0, -1/2, 1/2), (-1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 0, 0),
(1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 0, 0), (1/2, 1/2, -1/2, 1/2, -1/2, 1/2,
0, 0), (-1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 0, 0), (1/2, 1/2, 1/2, -1/2,
-1/2, 1/2, 0, 0), (-1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 0, 0), (-1/2, 1/2,
-1/2, -1/2, -1/2, 1/2, 0, 0), (1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 0, 0),
(0, 0, 0, 0, 0, 1, 1/2, -1/2), (0, 0, 0, 0, 0, -1, -1/2, 1/2), (-1/2,
1/2, 1/2, 1/2, 1/2, -1/2, 0, 0), (1/2, -1/2, 1/2, 1/2, 1/2, -1/2, 0, 0),
(1/2, 1/2, -1/2, 1/2, 1/2, -1/2, 0, 0), (-1/2, -1/2, -1/2, 1/2, 1/2,
-1/2, 0, 0), (1/2, 1/2, 1/2, -1/2, 1/2, -1/2, 0, 0), (-1/2, -1/2, 1/2,
-1/2, 1/2, -1/2, 0, 0), (-1/2, 1/2, -1/2, -1/2, 1/2, -1/2, 0, 0), (1/2,
-1/2, -1/2, -1/2, 1/2, -1/2, 0, 0), (0, 0, 0, 0, 1, 0, 1/2, -1/2), (1/2,
1/2, 1/2, 1/2, -1/2, -1/2, 0, 0), (-1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 0,
0), (-1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 0, 0), (1/2, -1/2, -1/2, 1/2,
-1/2, -1/2, 0, 0), (0, 0, 0, 1, 0, 0, 1/2, -1/2), (-1/2, 1/2, 1/2, -1/2,
-1/2, -1/2, 0, 0), (1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 0, 0), (0, 0, 1,
0, 0, 0, 1/2, -1/2), (1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 0, 0), (0, 1, 0,
0, 0, 0, 1/2, -1/2), (1, 0, 0, 0, 0, 0, 1/2, -1/2), (0, -1, 0, 0, 0, 0,
1/2, -1/2), (0, 0, -1, 0, 0, 0, 1/2, -1/2), (0, 0, 0, -1, 0, 0, 1/2,
-1/2), (0, 0, 0, 0, -1, 0, 1/2, -1/2), (0, 0, 0, 0, 0, -1, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 0, 0), (-1, 0, 0, 0, 0, 0, 1/2,
-1/2)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Letters(['E',Integer(7)])]
[(0, 0, 0, 0, 0, 1, -1/2, 1/2), (0, 0, 0, 0, 1, 0, -1/2, 1/2), (0, 0, 0,
1, 0, 0, -1/2, 1/2), (0, 0, 1, 0, 0, 0, -1/2, 1/2), (0, 1, 0, 0, 0, 0,
-1/2, 1/2), (-1, 0, 0, 0, 0, 0, -1/2, 1/2), (1, 0, 0, 0, 0, 0, -1/2,
```

(continues on next page)

(continued from previous page)

```
1/2), (1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 0, 0), (0, -1, 0, 0, 0, 0, -1/2,
1/2), (-1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 0, 0), (0, 0, -1, 0, 0, 0, -1/2,
1/2), (-1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 0, 0), (1/2, -1/2, -1/2, 1/2,
1/2, 1/2, 0, 0), (0, 0, 0, -1, 0, 0, -1/2, 1/2), (-1/2, 1/2, 1/2, -1/2,
1/2, 1/2, 0, 0), (1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 0, 0), (1/2, 1/2,
-1/2, -1/2, 1/2, 1/2, 0, 0), (-1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 0, 0),
(0, 0, 0, 0, -1, 0, -1/2, 1/2), (-1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 0, 0),
(1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 0, 0), (1/2, 1/2, -1/2, 1/2, -1/2, 1/2,
0, 0), (-1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 0, 0), (1/2, 1/2, 1/2, -1/2,
-1/2, 1/2, 0, 0), (-1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 0, 0), (-1/2, 1/2,
-1/2, -1/2, -1/2, 1/2, 0, 0), (1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 0, 0),
(0, 0, 0, 0, 0, 1, 1/2, -1/2), (0, 0, 0, 0, 0, -1, -1/2, 1/2), (-1/2,
1/2, 1/2, 1/2, 1/2, -1/2, 0, 0), (1/2, -1/2, 1/2, 1/2, 1/2, -1/2, 0, 0),
(1/2, 1/2, -1/2, 1/2, 1/2, -1/2, 0, 0), (-1/2, -1/2, -1/2, 1/2, 1/2,
-1/2, 0, 0), (1/2, 1/2, 1/2, -1/2, 1/2, -1/2, 0, 0), (-1/2, -1/2, 1/2,
-1/2, 1/2, -1/2, 0, 0), (-1/2, 1/2, -1/2, -1/2, 1/2, -1/2, 0, 0), (1/2,
-1/2, -1/2, -1/2, 1/2, -1/2, 0, 0), (0, 0, 0, 0, 1, 0, 1/2, -1/2), (1/2,
1/2, 1/2, 1/2, -1/2, -1/2, 0, 0), (-1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 0,
0), (-1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 0, 0), (1/2, -1/2, -1/2, 1/2,
-1/2, -1/2, 0, 0), (0, 0, 0, 1, 0, 0, 1/2, -1/2), (-1/2, 1/2, 1/2, -1/2,
-1/2, -1/2, 0, 0), (1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 0, 0), (0, 0, 1,
0, 0, 0, 1/2, -1/2), (1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 0, 0), (0, 1, 0,
0, 0, 0, 1/2, -1/2), (1, 0, 0, 0, 0, 0, 1/2, -1/2), (0, -1, 0, 0, 0, 0,
1/2, -1/2), (0, 0, -1, 0, 0, 0, 1/2, -1/2), (0, 0, 0, -1, 0, 0, 1/2,
-1/2), (0, 0, 0, 0, -1, 0, 1/2, -1/2), (0, 0, 0, 0, 0, -1, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 0, 0), (-1, 0, 0, 0, 0, 0, 1/2,
-1/2)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_G_element**

Bases: *Letter*

Type G_2 crystal of letters elements.

e(i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['G',2])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (0, 1, 3),
 (-3, 1, 0),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['G',Integer(2)])
>>> [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (0, 1, 3),
 (-3, 1, 0),
```

(continues on next page)

(continued from previous page)

```
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3)]
```

epsilon(i)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['G',2])
sage: [(c,i,c.epsilon(i)) for i in C.index_set() for c in C if c.epsilon(i) !
↪= 0]
[(2, 1, 1), (0, 1, 1), (-3, 1, 2), (-1, 1, 1), (3, 2, 1), (-2, 2, 1)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['G',Integer(2)])
>>> [(c,i,c.epsilon(i)) for i in C.index_set() for c in C if c.epsilon(i) !=␣
↪Integer(0)]
[(2, 1, 1), (0, 1, 1), (-3, 1, 2), (-1, 1, 1), (3, 2, 1), (-2, 2, 1)]
```

f(i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['G',2])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (3, 1, 0),
 (0, 1, -3),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['G',Integer(2)])
>>> [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (3, 1, 0),
 (0, 1, -3),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2)]
```

phi(i)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['G',2])
sage: [(c,i,c.phi(i)) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1, 1), (3, 1, 2), (0, 1, 1), (-2, 1, 1), (2, 2, 1), (-3, 2, 1)]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['G',Integer(2)])
>>> [(c,i,c.phi(i)) for i in C.index_set() for c in C if c.phi(i) !=␣
↪Integer(0)]
[(1, 1, 1), (3, 1, 2), (0, 1, 1), (-2, 1, 1), (2, 2, 1), (-3, 2, 1)]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['G',2])]
[(1, 0, -1), (1, -1, 0), (0, 1, -1), (0, 0, 0), (0, -1, 1), (-1, 1, 0), (-1,␣
↪0, 1)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Letters(['G',Integer(2)])]
[(1, 0, -1), (1, -1, 0), (0, 1, -1), (0, 0, 0), (0, -1, 1), (-1, 1, 0), (-1,␣
↪0, 1)]
```

class sage.combinat.crystals.letters.**EmptyLetter**

Bases: `Element`

The affine letter $\emptyset$ thought of as a classical crystal letter in classical type B_n and C_n.

> **Warning**
>
> This is not a classical letter.

Used in the rigged configuration bijections.

e(i)

Return e_i of `self` which is `None`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').e(1)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C', Integer(3)])
>>> C('E').e(Integer(1))
```

epsilon(i)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').epsilon(1)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C', Integer(3)])
>>> C('E').epsilon(Integer(1))
0
```

f(*i*)

Return f_i of `self` which is `None`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').f(1)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C', Integer(3)])
>>> C('E').f(Integer(1))
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').phi(1)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C', Integer(3)])
>>> C('E').phi(Integer(1))
0
```

value

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').weight()
(0, 0, 0)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['C', Integer(3)])
>>> C('E').weight()
(0, 0, 0)
```

class sage.combinat.crystals.letters.**Letter**

Bases: `Element`

A class for letters.

Like `ElementWrapper`, plus delegates `__lt__` (comparison) to the parent.

EXAMPLES:

```
sage: from sage.combinat.crystals.letters import Letter
sage: a = Letter(ZZ, 1)
sage: Letter(ZZ, 1).parent()
Integer Ring

sage: Letter(ZZ, 1)._repr_()
'1'

sage: parent1 = ZZ  # Any fake value ...
sage: parent2 = QQ  # Any fake value ...
sage: l11 = Letter(parent1, 1)
sage: l12 = Letter(parent1, 2)
sage: l21 = Letter(parent2, 1)
sage: l22 = Letter(parent2, 2)
sage: l11 == l11
True
sage: l11 == l12
False
sage: l11 == l21 # not tested
False

sage: C = crystals.Letters(['B', 3])
sage: C(0) != C(0)
False
sage: C(1) != C(-1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.letters import Letter
>>> a = Letter(ZZ, Integer(1))
>>> Letter(ZZ, Integer(1)).parent()
Integer Ring

>>> Letter(ZZ, Integer(1))._repr_()
'1'

>>> parent1 = ZZ  # Any fake value ...
>>> parent2 = QQ  # Any fake value ...
>>> l11 = Letter(parent1, Integer(1))
>>> l12 = Letter(parent1, Integer(2))
>>> l21 = Letter(parent2, Integer(1))
>>> l22 = Letter(parent2, Integer(2))
>>> l11 == l11
True
>>> l11 == l12
False
>>> l11 == l21 # not tested
False

>>> C = crystals.Letters(['B', Integer(3)])
>>> C(Integer(0)) != C(Integer(0))
False
```

(continues on next page)

(continued from previous page)

```
>>> C(Integer(1)) != C(-Integer(1))
True
```

value

class sage.combinat.crystals.letters.**LetterTuple**

Bases: Element

Abstract class for type E letters.

epsilon(i)

Return ε_i of self.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: C((-6,)).epsilon(1)
0
sage: C((-6,)).epsilon(6)
1
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)])
>>> C((-Integer(6),)).epsilon(Integer(1))
0
>>> C((-Integer(6),)).epsilon(Integer(6))
1
```

phi(i)

Return φ_i of self.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: C((1,)).phi(1)
1
sage: C((1,)).phi(6)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E',Integer(6)])
>>> C((Integer(1),)).phi(Integer(1))
1
>>> C((Integer(1),)).phi(Integer(6))
0
```

value

class sage.combinat.crystals.letters.**LetterWrapped**

Bases: Element

Element which uses another crystal implementation and converts those elements to a tuple with $\pm i$.

e(*i*)

Return e_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E', 8])
sage: C((-8,)).e(1)
sage: C((-8,)).e(8)
(-7, 8)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E', Integer(8)])
>>> C((-Integer(8),)).e(Integer(1))
>>> C((-Integer(8),)).e(Integer(8))
(-7, 8)
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E', 8])
sage: C((-8,)).epsilon(1)
0
sage: C((-8,)).epsilon(8)
1
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E', Integer(8)])
>>> C((-Integer(8),)).epsilon(Integer(1))
0
>>> C((-Integer(8),)).epsilon(Integer(8))
1
```

f(*i*)

Return f_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E', 8])
sage: C((8,)).f(6)
sage: C((8,)).f(8)
(7, -8)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E', Integer(8)])
>>> C((Integer(8),)).f(Integer(6))
>>> C((Integer(8),)).f(Integer(8))
(7, -8)
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E', 8])
sage: C((8,)).phi(8)
1
sage: C((8,)).phi(6)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['E', Integer(8)])
>>> C((Integer(8),)).phi(Integer(8))
1
>>> C((Integer(8),)).phi(Integer(6))
0
```

value

class sage.combinat.crystals.letters.**QueerLetter_element**

Bases: *Letter*

Queer supercrystal letters elements.

e(*i*)

Return the action of e_i on self.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q',3])
sage: [(c,i,c.e(i)) for i in Q.index_set() for c in Q if c.e(i) is not None]
[(2, 1, 1), (3, 2, 2), (3, -2, 2), (2, -1, 1)]
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q',Integer(3)])
>>> [(c,i,c.e(i)) for i in Q.index_set() for c in Q if c.e(i) is not None]
[(2, 1, 1), (3, 2, 2), (3, -2, 2), (2, -1, 1)]
```

epsilon(*i*)

Return ε_i of self.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q',3])
sage: [(c,i) for i in Q.index_set() for c in Q if c.epsilon(i) != 0]
[(2, 1), (3, 2), (3, -2), (2, -1)]
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q',Integer(3)])
>>> [(c,i) for i in Q.index_set() for c in Q if c.epsilon(i) != Integer(0)]
[(2, 1), (3, 2), (3, -2), (2, -1)]
```

f(*i*)

Return the action of f_i on self.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q',3])
sage: [(c,i,c.f(i)) for i in Q.index_set() for c in Q if c.f(i) is not None]
[(1, 1, 2), (2, 2, 3), (2, -2, 3), (1, -1, 2)]
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q',Integer(3)])
>>> [(c,i,c.f(i)) for i in Q.index_set() for c in Q if c.f(i) is not None]
[(1, 1, 2), (2, 2, 3), (2, -2, 3), (1, -1, 2)]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q',3])
sage: [(c,i) for i in Q.index_set() for c in Q if c.phi(i) != 0]
[(1, 1), (2, 2), (2, -2), (1, -1)]
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q',Integer(3)])
>>> [(c,i) for i in Q.index_set() for c in Q if c.phi(i) != Integer(0)]
[(1, 1), (2, 2), (2, -2), (1, -1)]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['Q',4])]
[(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Letters(['Q',Integer(4)])]
[(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)]
```

5.1.58 Littelmann paths

AUTHORS:

- Mark Shimozono, Anne Schilling (2012): Initial version
- Anne Schilling (2013): Implemented `CrystalOfProjectedLevelZeroLSPaths`
- Travis Scrimshaw (2016): Implemented `InfinityCrystalOfLSPaths`

class sage.combinat.crystals.littelmann_path.**CrystalOfLSPaths**(*starting_weight*, *starting_weight_parent*)

Bases: `UniqueRepresentation`, `Parent`

Crystal graph of LS paths generated from the straight-line path to a given weight.

INPUT:

- `cartan_type` – (optional) the Cartan type

- `starting_weight` – a weight; if `cartan_type` is given, then the weight should be given as a list of coefficients of the fundamental weights, otherwise it should be given in the `weight_space` basis; for affine highest weight crystals, one needs to use the extended weight space

The crystal class of piecewise linear paths in the weight space, generated from a straight-line path from the origin to a given element of the weight lattice.

EXAMPLES:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space(extended = True).basis()
sage: B = crystals.LSPaths(La[2]-La[0]); B
The crystal of LS paths of type ['A', 2, 1] and weight -Lambda[0] + Lambda[2]

sage: C = crystals.LSPaths(['A',2,1],[-1,0,1]); C
The crystal of LS paths of type ['A', 2, 1] and weight -Lambda[0] + Lambda[2]
sage: B == C
True
sage: c = C.module_generators[0]; c
(-Lambda[0] + Lambda[2],)
sage: [c.f(i) for i in C.index_set()]
[None, None, (Lambda[1] - Lambda[2],)]

sage: R = C.R; R
Root system of type ['A', 2, 1]
sage: Lambda = R.weight_space().basis(); Lambda
Finite family {0: Lambda[0], 1: Lambda[1], 2: Lambda[2]}
sage: b=C(tuple([-Lambda[0]+Lambda[2]]))
sage: b==c
True
sage: b.f(2)
(Lambda[1] - Lambda[2],)
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(2),Integer(1)])
>>> La = R.weight_space(extended = True).basis()
>>> B = crystals.LSPaths(La[Integer(2)]-La[Integer(0)]); B
The crystal of LS paths of type ['A', 2, 1] and weight -Lambda[0] + Lambda[2]

>>> C = crystals.LSPaths(['A',Integer(2),Integer(1)],[-Integer(1),Integer(0),
↪Integer(1)]); C
The crystal of LS paths of type ['A', 2, 1] and weight -Lambda[0] + Lambda[2]
>>> B == C
True
>>> c = C.module_generators[Integer(0)]; c
(-Lambda[0] + Lambda[2],)
>>> [c.f(i) for i in C.index_set()]
[None, None, (Lambda[1] - Lambda[2],)]

>>> R = C.R; R
Root system of type ['A', 2, 1]
>>> Lambda = R.weight_space().basis(); Lambda
Finite family {0: Lambda[0], 1: Lambda[1], 2: Lambda[2]}
>>> b=C(tuple([-Lambda[Integer(0)]+Lambda[Integer(2)]]))
```

(continues on next page)

(continued from previous page)

```
>>> b==c
True
>>> b.f(Integer(2))
(Lambda[1] - Lambda[2],)
```

For classical highest weight crystals, we can also compare the results with the tableaux implementation:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: sorted(C, key=str)
[(-2*Lambda[1] + Lambda[2],), (-Lambda[1] + 1/2*Lambda[2], Lambda[1] - 1/
↪2*Lambda[2]),
 (-Lambda[1] + 2*Lambda[2],), (-Lambda[1] - Lambda[2],),
 (1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2]), (2*Lambda[1] -␣
↪Lambda[2],),
 (Lambda[1] + Lambda[2],), (Lambda[1] - 2*Lambda[2],)]
sage: C.cardinality()
8
sage: B = crystals.Tableaux(['A',2],shape=[2,1])
sage: B.cardinality()
8
sage: B.digraph().is_isomorphic(C.digraph())
True
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> sorted(C, key=str)
[(-2*Lambda[1] + Lambda[2],), (-Lambda[1] + 1/2*Lambda[2], Lambda[1] - 1/
↪2*Lambda[2]),
 (-Lambda[1] + 2*Lambda[2],), (-Lambda[1] - Lambda[2],),
 (1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2]), (2*Lambda[1] -␣
↪Lambda[2],),
 (Lambda[1] + Lambda[2],), (Lambda[1] - 2*Lambda[2],)]
>>> C.cardinality()
8
>>> B = crystals.Tableaux(['A',Integer(2)],shape=[Integer(2),Integer(1)])
>>> B.cardinality()
8
>>> B.digraph().is_isomorphic(C.digraph())
True
```

Make sure you use the weight space and not the weight lattice for your weights:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_lattice(extended = True).basis()
sage: B = crystals.LSPaths(La[2]); B
Traceback (most recent call last):
...
ValueError: use the weight space, rather than weight lattice for your weights
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(2),Integer(1)])
>>> La = R.weight_lattice(extended = True).basis()
```

(continues on next page)

(continued from previous page)

```
>>> B = crystals.LSPaths(La[Integer(2)]); B
Traceback (most recent call last):
...
ValueError: use the weight space, rather than weight lattice for your weights
```

REFERENCES:

- [Li1995b]

class Element

Bases: ElementWrapper

A Littelmann path (crystal element).

compress()

Merge consecutive positively parallel steps present in `self`.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: Lambda = C.R.weight_space().fundamental_weights(); Lambda
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: c = C(tuple([1/2*Lambda[1]+1/2*Lambda[2], 1/2*Lambda[1]+1/
↪2*Lambda[2]]))
sage: c.compress()
(Lambda[1] + Lambda[2],)
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> Lambda = C.R.weight_space().fundamental_weights(); Lambda
Finite family {1: Lambda[1], 2: Lambda[2]}
>>> c = C(tuple([Integer(1)/Integer(2)*Lambda[Integer(1)]+Integer(1)/
↪Integer(2)*Lambda[Integer(2)], Integer(1)/
↪Integer(2)*Lambda[Integer(1)]+Integer(1)/
↪Integer(2)*Lambda[Integer(2)]]))
>>> c.compress()
(Lambda[1] + Lambda[2],)
```

dualize()

Return the dualized path of `self`.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: for c in C:
....:     print("{} {}".format(c, c.dualize()))
(Lambda[1] + Lambda[2],) (-Lambda[1] - Lambda[2],)
(-Lambda[1] + 2*Lambda[2],) (Lambda[1] - 2*Lambda[2],)
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2]) (1/2*Lambda[1] -␣
↪Lambda[2], -1/2*Lambda[1] + Lambda[2])
(Lambda[1] - 2*Lambda[2],) (-Lambda[1] + 2*Lambda[2],)
(-Lambda[1] - Lambda[2],) (Lambda[1] + Lambda[2],)
(2*Lambda[1] - Lambda[2],) (-2*Lambda[1] + Lambda[2],)
(-Lambda[1] + 1/2*Lambda[2], Lambda[1] - 1/2*Lambda[2]) (-Lambda[1] + 1/
```

(continues on next page)

(continued from previous page)

```
↪2*Lambda[2], Lambda[1] - 1/2*Lambda[2])
(-2*Lambda[1] + Lambda[2],) (2*Lambda[1] - Lambda[2],)
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> for c in C:
...     print("{} {}".format(c, c.dualize()))
(Lambda[1] + Lambda[2],) (-Lambda[1] - Lambda[2],)
(-Lambda[1] + 2*Lambda[2],) (Lambda[1] - 2*Lambda[2],)
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2]) (1/2*Lambda[1] -␣
↪Lambda[2], -1/2*Lambda[1] + Lambda[2])
(Lambda[1] - 2*Lambda[2],) (-Lambda[1] + 2*Lambda[2],)
(-Lambda[1] - Lambda[2],) (Lambda[1] + Lambda[2],)
(2*Lambda[1] - Lambda[2],) (-2*Lambda[1] + Lambda[2],)
(-Lambda[1] + 1/2*Lambda[2], Lambda[1] - 1/2*Lambda[2]) (-Lambda[1] + 1/
↪2*Lambda[2], Lambda[1] - 1/2*Lambda[2])
(-2*Lambda[1] + Lambda[2],) (2*Lambda[1] - Lambda[2],)
```

e(*i*, *power=1*, *to_string_end=False*, *length_only=False*)

Return the i-th crystal raising operator on `self`.

INPUT:

- `i` – element of the index set of the underlying root system
- `power` – positive integer (default: 1); specifies the power of the raising operator to be applied
- `to_string_end` – boolean (default: `False`); if `True`, returns the dominant end of the i-string of `self`
- `length_only` – boolean; if `True`, returns the distance to the dominant end of the i-string of `self`

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: c = C[2]; c
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
sage: c.e(1)
sage: c.e(2)
(-Lambda[1] + 2*Lambda[2],)
sage: c.e(2,to_string_end=True)
(-Lambda[1] + 2*Lambda[2],)
sage: c.e(1,to_string_end=True)
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
sage: c.e(1,length_only=True)
0
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> c = C[Integer(2)]; c
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
>>> c.e(Integer(1))
>>> c.e(Integer(2))
(-Lambda[1] + 2*Lambda[2],)
>>> c.e(Integer(2),to_string_end=True)
(-Lambda[1] + 2*Lambda[2],)
>>> c.e(Integer(1),to_string_end=True)
```

(continues on next page)

(continued from previous page)

```
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
>>> c.e(Integer(1),length_only=True)
0
```

endpoint()

Compute the endpoint of `self`.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: b = C.module_generators[0]
sage: b.endpoint()
Lambda[1] + Lambda[2]
sage: b.f_string([1,2,2,1])
(-Lambda[1] - Lambda[2],)
sage: b.f_string([1,2,2,1]).endpoint()
-Lambda[1] - Lambda[2]
sage: b.f_string([1,2])
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
sage: b.f_string([1,2]).endpoint()
0
sage: b = C([])
sage: b.endpoint()
0
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> b = C.module_generators[Integer(0)]
>>> b.endpoint()
Lambda[1] + Lambda[2]
>>> b.f_string([Integer(1),Integer(2),Integer(2),Integer(1)])
(-Lambda[1] - Lambda[2],)
>>> b.f_string([Integer(1),Integer(2),Integer(2),Integer(1)]).endpoint()
-Lambda[1] - Lambda[2]
>>> b.f_string([Integer(1),Integer(2)])
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
>>> b.f_string([Integer(1),Integer(2)]).endpoint()
0
>>> b = C([])
>>> b.endpoint()
0
```

epsilon(i)

Return the distance to the beginning of the i-string.

This method overrides the generic implementation in the category of crystals since this computation is more efficient.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: [c.epsilon(1) for c in C]
[0, 1, 0, 0, 1, 0, 1, 2]
```

(continues on next page)

(continued from previous page)

```
sage: [c.epsilon(2) for c in C]
[0, 0, 1, 2, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> [c.epsilon(Integer(1)) for c in C]
[0, 1, 0, 0, 1, 0, 1, 2]
>>> [c.epsilon(Integer(2)) for c in C]
[0, 0, 1, 2, 1, 1, 0, 0]
```

f(*i*, *power=1*, *to_string_end=False*, *length_only=False*)

Return the i-th crystal lowering operator on `self`.

INPUT:

- `i` – element of the index set of the underlying root system
- `power` – positive integer (default: 1); specifies the power of the lowering operator to be applied
- `to_string_end` – boolean (default: `False`); if `True`, returns the anti-dominant end of the i-string of `self`
- `length_only` – boolean; if `True`, returns the distance to the anti-dominant end of the i-string of `self`

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: c = C.module_generators[0]
sage: c.f(1)
(-Lambda[1] + 2*Lambda[2],)
sage: c.f(1,power=2)
sage: c.f(2)
(2*Lambda[1] - Lambda[2],)
sage: c.f(2,to_string_end=True)
(2*Lambda[1] - Lambda[2],)
sage: c.f(2,length_only=True)
1

sage: C = crystals.LSPaths(['A',2,1],[-1,-1,2])
sage: c = C.module_generators[0]
sage: c.f(2,power=2)
(Lambda[0] + Lambda[1] - 2*Lambda[2],)
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> c = C.module_generators[Integer(0)]
>>> c.f(Integer(1))
(-Lambda[1] + 2*Lambda[2],)
>>> c.f(Integer(1),power=Integer(2))
>>> c.f(Integer(2))
(2*Lambda[1] - Lambda[2],)
>>> c.f(Integer(2),to_string_end=True)
(2*Lambda[1] - Lambda[2],)
>>> c.f(Integer(2),length_only=True)
1
```

(continues on next page)

(continued from previous page)

```
>>> C = crystals.LSPaths(['A',Integer(2),Integer(1)],[-Integer(1),-
↪Integer(1),Integer(2)])
>>> c = C.module_generators[Integer(0)]
>>> c.f(Integer(2),power=Integer(2))
(Lambda[0] + Lambda[1] - 2*Lambda[2],)
```

phi(*i*)

Return the distance to the end of the i-string.

This method overrides the generic implementation in the category of crystals since this computation is more efficient.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: [c.phi(1) for c in C]
[1, 0, 0, 1, 0, 2, 1, 0]
sage: [c.phi(2) for c in C]
[1, 2, 1, 0, 0, 0, 0, 1]
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> [c.phi(Integer(1)) for c in C]
[1, 0, 0, 1, 0, 2, 1, 0]
>>> [c.phi(Integer(2)) for c in C]
[1, 2, 1, 0, 0, 0, 0, 1]
```

reflect_step(*which_step*, *i*)

Apply the i-th simple reflection to the indicated step in `self`.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: b = C.module_generators[0]
sage: b.reflect_step(0,1)
(-Lambda[1] + 2*Lambda[2],)
sage: b.reflect_step(0,2)
(2*Lambda[1] - Lambda[2],)
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> b = C.module_generators[Integer(0)]
>>> b.reflect_step(Integer(0),Integer(1))
(-Lambda[1] + 2*Lambda[2],)
>>> b.reflect_step(Integer(0),Integer(2))
(2*Lambda[1] - Lambda[2],)
```

s(*i*)

Compute the reflection of `self` along the i-string.

This method is more efficient than the generic implementation since it uses powers of e and f in the Littelmann model directly.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: c = C.module_generators[0]
sage: c.s(1)
(-Lambda[1] + 2*Lambda[2],)
sage: c.s(2)
(2*Lambda[1] - Lambda[2],)

sage: C = crystals.LSPaths(['A',2,1],[-1,0,1])
sage: c = C.module_generators[0]; c
(-Lambda[0] + Lambda[2],)
sage: c.s(2)
(Lambda[1] - Lambda[2],)
sage: c.s(1)
(-Lambda[0] + Lambda[2],)
sage: c.f(2).s(1)
(Lambda[0] - Lambda[1],)
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> c = C.module_generators[Integer(0)]
>>> c.s(Integer(1))
(-Lambda[1] + 2*Lambda[2],)
>>> c.s(Integer(2))
(2*Lambda[1] - Lambda[2],)

>>> C = crystals.LSPaths(['A',Integer(2),Integer(1)],[-Integer(1),
↪Integer(0),Integer(1)])
>>> c = C.module_generators[Integer(0)]; c
(-Lambda[0] + Lambda[2],)
>>> c.s(Integer(2))
(Lambda[1] - Lambda[2],)
>>> c.s(Integer(1))
(-Lambda[0] + Lambda[2],)
>>> c.f(Integer(2)).s(Integer(1))
(Lambda[0] - Lambda[1],)
```

split_step(*which_step*, *r*)

Split the indicated step into two parallel steps of relative lengths r and $1 - r$.

INPUT:

- `which_step` – a position in the tuple `self`
- `r` – a rational number between 0 and 1

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: b = C.module_generators[0]
sage: b.split_step(0,1/3)
(1/3*Lambda[1] + 1/3*Lambda[2], 2/3*Lambda[1] + 2/3*Lambda[2])
```

```
>>> from sage.all import *
>>> C = crystals.LSPaths(['A',Integer(2)],[Integer(1),Integer(1)])
>>> b = C.module_generators[Integer(0)]
>>> b.split_step(Integer(0),Integer(1)/Integer(3))
```

(continues on next page)

(continued from previous page)

```
(1/3*Lambda[1] + 1/3*Lambda[2], 2/3*Lambda[1] + 2/3*Lambda[2])
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.LSPaths(['A',1,1],[1,0])
sage: b = B.highest_weight_vector()
sage: b.f(0).weight()
-Lambda[0] + 2*Lambda[1] - delta
```

```
>>> from sage.all import *
>>> B = crystals.LSPaths(['A',Integer(1),Integer(1)],[Integer(1),
↪Integer(0)])
>>> b = B.highest_weight_vector()
>>> b.f(Integer(0)).weight()
-Lambda[0] + 2*Lambda[1] - delta
```

weight_lattice_realization()

Return weight lattice realization of `self`.

EXAMPLES:

```
sage: B = crystals.LSPaths(['B',3],[1,1,0])
sage: B.weight_lattice_realization()
Weight space over the Rational Field of the Root system of type ['B', 3]
sage: B = crystals.LSPaths(['B',3,1],[1,1,1,0])
sage: B.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['B',
↪ 3, 1]
```

```
>>> from sage.all import *
>>> B = crystals.LSPaths(['B',Integer(3)],[Integer(1),Integer(1),Integer(0)])
>>> B.weight_lattice_realization()
Weight space over the Rational Field of the Root system of type ['B', 3]
>>> B = crystals.LSPaths(['B',Integer(3),Integer(1)],[Integer(1),Integer(1),
↪Integer(1),Integer(0)])
>>> B.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['B',
↪ 3, 1]
```

class sage.combinat.crystals.littelmann_path.**CrystalOfProjectedLevelZeroLSPaths**(*starting_weight*, *starting_weight_parent*)

Bases: *CrystalOfLSPaths*

Crystal of projected level zero LS paths.

INPUT:

- `weight` – a dominant weight of the weight space of an affine Kac-Moody root system

When `weight` is just a single fundamental weight Λ_r, this crystal is isomorphic to a Kirillov-Reshetikhin (KR) crystal, see also `sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinFromLSPaths()`. For general weights, it is isomorphic to a tensor product of single-column KR crystals.

EXAMPLES:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[3])
sage: LS.cardinality()
84
sage: GLS = LS.digraph()

sage: K1 = crystals.KirillovReshetikhin(['C',3,1],1,1)
sage: K3 = crystals.KirillovReshetikhin(['C',3,1],3,1)
sage: T = crystals.TensorProduct(K3,K1)
sage: T.cardinality()
84
sage: GT = T.digraph() # long time
sage: GLS.is_isomorphic(GT, edge_labels = True) # long time
True
```

```
>>> from sage.all import *
>>> R = RootSystem(['C',Integer(3),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)]+La[Integer(3)])
>>> LS.cardinality()
84
>>> GLS = LS.digraph()

>>> K1 = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)],Integer(1),
↪Integer(1))
>>> K3 = crystals.KirillovReshetikhin(['C',Integer(3),Integer(1)],Integer(3),
↪Integer(1))
>>> T = crystals.TensorProduct(K3,K1)
>>> T.cardinality()
84
>>> GT = T.digraph() # long time
>>> GLS.is_isomorphic(GT, edge_labels = True) # long time
True
```

class Element

Bases: *Element*

Element of a crystal of projected level zero LS paths.

energy_function()

Return the energy function of `self`.

The energy function $D(\pi)$ of the level zero LS path $\pi \in \mathbb{B}_{\mathrm{cl}}(\lambda)$ requires a series of definitions; for simplicity the root system is assumed to be untwisted affine.

The LS path π is a piecewise linear map from the unit interval $[0,1]$ to the weight lattice. It is specified by "times" $0 = \sigma_0 < \sigma_1 < \cdots < \sigma_s = 1$ and "direction vectors" $x_u\lambda$ where $x_u \in W/W_J$ for $1 \le u \le s$, and W_J is the stabilizer of λ in the finite Weyl group W. Precisely,

$$\pi(t) = \sum_{u'=1}^{u-1} (\sigma_{u'} - \sigma_{u'-1}) x_{u'} \lambda + (t - \sigma_{u-1}) x_u \lambda$$

for $1 \leq u \leq s$ and $\sigma_{u-1} \leq t \leq \sigma_u$.

For any $x, y \in W/W_J$, let

$$d : x = w_0 \stackrel{\beta_1}{\leftarrow} w_1 \stackrel{\beta_2}{\leftarrow} \cdots \stackrel{\beta_n}{\leftarrow} w_n = y$$

be a shortest directed path in the parabolic quantum Bruhat graph. Define

$$\mathrm{wt}(d) := \sum_{\substack{1 \leq k \leq n \\ \ell(w_{k-1}) < \ell(w_k)}} \beta_k^\vee.$$

It can be shown that $\mathrm{wt}(d)$ depends only on x, y; call its value $\mathrm{wt}(x, y)$. The energy function $D(\pi)$ is defined by

$$D(\pi) = -\sum_{u=1}^{s-1} (1 - \sigma_u) \langle \lambda, \mathrm{wt}(x_u, x_{u+1}) \rangle.$$

For more information, see [LNSSS2013].

Note: In the dual-of-untwisted case the parabolic quantum Bruhat graph that is used is obtained by exchanging the roles of roots and coroots. Moreover, in the computation of the pairing the short roots must be doubled (or tripled for type G). This factor is determined by the translation factor of the corresponding root. Type BC is viewed as untwisted type, whereas the dual of BC is viewed as twisted. Except for the untwisted cases, these formulas are currently still conjectural.

EXAMPLES:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[3])
sage: b = LS.module_generators[0]
sage: c = b.f(1).f(3).f(2)
sage: c.energy_function()
0
sage: c=b.e(0)
sage: c.energy_function()
1

sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: b = LS.module_generators[0]
sage: c = b.e(0)
sage: c.energy_function()
1
sage: for c in sorted(LS, key=str):
```

(continues on next page)

(continued from previous page)

```
....:     print("{} {}".format(c,c.energy_function()))
(-2*Lambda[0] + 2*Lambda[1],)                    0
(-2*Lambda[1] + 2*Lambda[2],)                    0
(-Lambda[0] + Lambda[1], -Lambda[1] + Lambda[2]) 1
(-Lambda[0] + Lambda[1], Lambda[0] - Lambda[2])  1
(-Lambda[1] + Lambda[2], -Lambda[0] + Lambda[1]) 0
(-Lambda[1] + Lambda[2], Lambda[0] - Lambda[2])  1
(2*Lambda[0] - 2*Lambda[2],)                     0
(Lambda[0] - Lambda[2], -Lambda[0] + Lambda[1])  0
(Lambda[0] - Lambda[2], -Lambda[1] + Lambda[2])  0
```

```
>>> from sage.all import *
>>> R = RootSystem(['C',Integer(3),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)]+La[Integer(3)])
>>> b = LS.module_generators[Integer(0)]
>>> c = b.f(Integer(1)).f(Integer(3)).f(Integer(2))
>>> c.energy_function()
0
>>> c=b.e(Integer(0))
>>> c.energy_function()
1

>>> R = RootSystem(['A',Integer(2),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])
>>> b = LS.module_generators[Integer(0)]
>>> c = b.e(Integer(0))
>>> c.energy_function()
1
>>> for c in sorted(LS, key=str):
...     print("{} {}".format(c,c.energy_function()))
(-2*Lambda[0] + 2*Lambda[1],)                    0
(-2*Lambda[1] + 2*Lambda[2],)                    0
(-Lambda[0] + Lambda[1], -Lambda[1] + Lambda[2]) 1
(-Lambda[0] + Lambda[1], Lambda[0] - Lambda[2])  1
(-Lambda[1] + Lambda[2], -Lambda[0] + Lambda[1]) 0
(-Lambda[1] + Lambda[2], Lambda[0] - Lambda[2])  1
(2*Lambda[0] - 2*Lambda[2],)                     0
(Lambda[0] - Lambda[2], -Lambda[0] + Lambda[1])  0
(Lambda[0] - Lambda[2], -Lambda[1] + Lambda[2])  0
```

The next test checks that the energy function is constant on classically connected components:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])
sage: G = LS.digraph(index_set=[1,2])
sage: C = G.connected_components(sort=False)
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c␣
↪in C]
[True, True, True, True]
```

(continues on next page)

(continued from previous page)

```
sage: R = RootSystem(['D',4,2])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[2])
sage: J = R.cartan_type().classical().index_set()
sage: hw = [x for x in LS if x.is_highest_weight(J)]
sage: [(x.weight(), x.energy_function()) for x in hw]
[(-2*Lambda[0] + Lambda[2], 0), (-2*Lambda[0] + Lambda[1], 1), (0, 2)]
sage: G = LS.digraph(index_set=J)
sage: C = G.connected_components(sort=False)
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c␣
↪in C]
[True, True, True]

sage: R = RootSystem(CartanType(['G',2,1]).dual())
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[2])
sage: G = LS.digraph(index_set=[1,2])
sage: C = G.connected_components(sort=False)
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c␣
↪in C] # long time
[True, True, True, True, True, True, True, True, True, True, True, True,␣
↪True, True, True, True]

sage: ct = CartanType(['BC',2,2]).dual()
sage: R = RootSystem(ct)
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])
sage: G = LS.digraph(index_set=R.cartan_type().classical().index_set())
sage: C = G.connected_components(sort=False)
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c␣
↪in C] # long time
[True, True, True, True, True, True, True, True, True, True, True]

sage: R = RootSystem(['BC',2,2])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])
sage: G = LS.digraph(index_set=R.cartan_type().classical().index_set())
sage: C = G.connected_components(sort=False)
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c␣
↪in C] # long time
[True, True, True, True, True, True, True, True, True, True, True, True,␣
↪True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,␣
↪True, True, True, True]
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(2),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.
↪ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)]+La[Integer(2)])
>>> G = LS.digraph(index_set=[Integer(1),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> C = G.connected_components(sort=False)
>>> [all(c[Integer(0)].energy_function()==a.energy_function() for a in c)␣
↪for c in C]
[True, True, True, True]

>>> R = RootSystem(['D',Integer(4),Integer(2)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(2)])
>>> J = R.cartan_type().classical().index_set()
>>> hw = [x for x in LS if x.is_highest_weight(J)]
>>> [(x.weight(), x.energy_function()) for x in hw]
[(-2*Lambda[0] + Lambda[2], 0), (-2*Lambda[0] + Lambda[1], 1), (0, 2)]
>>> G = LS.digraph(index_set=J)
>>> C = G.connected_components(sort=False)
>>> [all(c[Integer(0)].energy_function()==a.energy_function() for a in c)␣
↪for c in C]
[True, True, True]

>>> R = RootSystem(CartanType(['G',Integer(2),Integer(1)]).dual())
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)]+La[Integer(2)])
>>> G = LS.digraph(index_set=[Integer(1),Integer(2)])
>>> C = G.connected_components(sort=False)
>>> [all(c[Integer(0)].energy_function()==a.energy_function() for a in c)␣
↪for c in C] # long time
[True, True, True, True, True, True, True, True, True, True, True, True,␣
↪True, True, True, True]

>>> ct = CartanType(['BC',Integer(2),Integer(2)]).dual()
>>> R = RootSystem(ct)
>>> La = R.weight_space().basis()
>>> LS = crystals.
↪ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)]+La[Integer(2)])
>>> G = LS.digraph(index_set=R.cartan_type().classical().index_set())
>>> C = G.connected_components(sort=False)
>>> [all(c[Integer(0)].energy_function()==a.energy_function() for a in c)␣
↪for c in C] # long time
[True, True, True, True, True, True, True, True, True, True, True]

>>> R = RootSystem(['BC',Integer(2),Integer(2)])
>>> La = R.weight_space().basis()
>>> LS = crystals.
↪ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)]+La[Integer(2)])
>>> G = LS.digraph(index_set=R.cartan_type().classical().index_set())
>>> C = G.connected_components(sort=False)
>>> [all(c[Integer(0)].energy_function()==a.energy_function() for a in c)␣
↪for c in C] # long time
[True, True, True, True, True, True, True, True, True, True, True, True,␣
↪True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,␣
↪True, True, True, True]
```

scalar_factors()

Return the scalar factors for `self`.

Each LS path (or `self`) can be written as a piecewise linear map

$$\pi(t) = \sum_{u'=1}^{u-1} (\sigma_{u'} - \sigma_{u'-1})\nu_{u'} + (t - \sigma_{u-1})\nu_u$$

for $0 < \sigma_1 < \sigma_2 < \cdots < \sigma_s = 1$ and $\sigma_{u-1} \leq t \leq \sigma_u$ and $1 \leq u \leq s$. This method returns the tuple of $(\sigma_1, \ldots, \sigma_s)$.

EXAMPLES:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[3])
sage: b = LS.module_generators[0]
sage: b.scalar_factors()
[1]
sage: c = b.f(1).f(3).f(2)
sage: c.scalar_factors()
[1/3, 1]
```

```
>>> from sage.all import *
>>> R = RootSystem(['C',Integer(3),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)]+La[Integer(3)])
>>> b = LS.module_generators[Integer(0)]
>>> b.scalar_factors()
[1]
>>> c = b.f(Integer(1)).f(Integer(3)).f(Integer(2))
>>> c.scalar_factors()
[1/3, 1]
```

weyl_group_representation()

Transform the weights in the LS path `self` to elements in the Weyl group.

Each LS path can be written as the piecewise linear map:

$$\pi(t) = \sum_{u'=1}^{u-1} (\sigma_{u'} - \sigma_{u'-1})\nu_{u'} + (t - \sigma_{u-1})\nu_u$$

for $0 < \sigma_1 < \sigma_2 < \cdots < \sigma_s = 1$ and $\sigma_{u-1} \leq t \leq \sigma_u$ and $1 \leq u \leq s$. Each weight ν_u is also associated to a Weyl group element. This method returns the list of Weyl group elements associated to the ν_u for $1 \leq u \leq s$.

EXAMPLES:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[3])
sage: b = LS.module_generators[0]
sage: c = b.f(1).f(3).f(2)
sage: c.weyl_group_representation()
[s2*s1*s3, s1*s3]
```

```
>>> from sage.all import *
>>> R = RootSystem(['C',Integer(3),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)]+La[Integer(3)])
>>> b = LS.module_generators[Integer(0)]
>>> c = b.f(Integer(1)).f(Integer(3)).f(Integer(2))
>>> c.weyl_group_representation()
[s2*s1*s3, s1*s3]
```

classically_highest_weight_vectors()

Return the classically highest weight vectors of `self`.

EXAMPLES:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: LS.classically_highest_weight_vectors()
((-2*Lambda[0] + 2*Lambda[1],),
 (-Lambda[0] + Lambda[1], -Lambda[1] + Lambda[2]))
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(2),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])
>>> LS.classically_highest_weight_vectors()
((-2*Lambda[0] + 2*Lambda[1],),
 (-Lambda[0] + Lambda[1], -Lambda[1] + Lambda[2]))
```

is_perfect(*level=1*)

Check whether the crystal `self` is perfect (of level `level`).

INPUT:

- `level` – (default: 1) positive integer

A crystal $\mathcal{B}$ is perfect of level ℓ if:

1. $\mathcal{B}$ is isomorphic to the crystal graph of a finite-dimensional $U_q^{'}(\mathfrak{g})$-module.
2. $\mathcal{B} \otimes \mathcal{B}$ is connected.
3. There exists a $\lambda \in X$, such that $\mathrm{wt}(\mathcal{B}) \subset \lambda + \sum_{i\in I} \mathbf{Z}_{\leq 0}\alpha_i$ and there is a unique element in $\mathcal{B}$ of classical weight λ.
4. For all $b \in \mathcal{B}$, $\mathrm{level}(\varepsilon(b)) \geq \ell$.
5. For all Λ dominant weights of level ℓ, there exist unique elements $b_\Lambda, b^\Lambda \in \mathcal{B}$, such that $\varepsilon(b_\Lambda) = \Lambda = \varphi(b^\Lambda)$.

Points (1)-(3) are known to hold. This method checks points (4) and (5).

EXAMPLES:

```
sage: C = CartanType(['C',2,1])
sage: R = RootSystem(C)
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1])
```

(continues on next page)

(continued from previous page)

```
sage: LS.is_perfect()
False
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[2])
sage: LS.is_perfect()
True

sage: C = CartanType(['E',6,1])
sage: R = RootSystem(C)
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1])
sage: LS.is_perfect()
True
sage: LS.is_perfect(2)
False

sage: C = CartanType(['D',4,1])
sage: R = RootSystem(C)
sage: La = R.weight_space().basis()
sage: all(crystals.ProjectedLevelZeroLSPaths(La[i]).is_perfect() for i in [1,
→2,3,4])
True

sage: C = CartanType(['A',6,2])
sage: R = RootSystem(C)
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[2])
sage: LS.is_perfect()
True
sage: LS.is_perfect(2)
False
```

```
>>> from sage.all import *
>>> C = CartanType(['C',Integer(2),Integer(1)])
>>> R = RootSystem(C)
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)])
>>> LS.is_perfect()
False
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(2)])
>>> LS.is_perfect()
True

>>> C = CartanType(['E',Integer(6),Integer(1)])
>>> R = RootSystem(C)
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)])
>>> LS.is_perfect()
True
>>> LS.is_perfect(Integer(2))
False

>>> C = CartanType(['D',Integer(4),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> R = RootSystem(C)
>>> La = R.weight_space().basis()
>>> all(crystals.ProjectedLevelZeroLSPaths(La[i]).is_perfect() for i in␣
↪[Integer(1),Integer(2),Integer(3),Integer(4)])
True

>>> C = CartanType(['A',Integer(6),Integer(2)])
>>> R = RootSystem(C)
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)]+La[Integer(2)])
>>> LS.is_perfect()
True
>>> LS.is_perfect(Integer(2))
False
```

maximal_vector()

Return the maximal vector of `self`.

EXAMPLES:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])
sage: LS.maximal_vector()
(-3*Lambda[0] + 2*Lambda[1] + Lambda[2],)
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(2),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.
↪ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)]+La[Integer(2)])
>>> LS.maximal_vector()
(-3*Lambda[0] + 2*Lambda[1] + Lambda[2],)
```

one_dimensional_configuration_sum(*q=None*, *group_components=True*)

Compute the one-dimensional configuration sum.

INPUT:

- `q` – (default: `None`) a variable or `None`; if `None`, a variable `q` is set in the code
- `group_components` – boolean (default: `True`); if `True`, then the terms are grouped by classical component

The one-dimensional configuration sum is the sum of the weights of all elements in the crystal weighted by the energy function. For untwisted types it uses the parabolic quantum Bruhat graph, see [LNSSS2013]. In the dual-of-untwisted case, the parabolic quantum Bruhat graph is defined by exchanging the roles of roots and coroots (which is still conjectural at this point).

EXAMPLES:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: LS.one_dimensional_configuration_sum() # long time
```

(continues on next page)

(continued from previous page)

```
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
 + (q+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
sage: R.<t> = ZZ[]
sage: LS.one_dimensional_configuration_sum(t, False) # long time
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]]
 + (t+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(2),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])
>>> LS.one_dimensional_configuration_sum() # long time
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
 + (q+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> LS.one_dimensional_configuration_sum(t, False) # long time
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]]
 + (t+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]
```

class sage.combinat.crystals.littelmann_path.**InfinityCrystalOfLSPaths**(*cartan_type*)

Bases: UniqueRepresentation, Parent

LS path model for $\mathcal{B}(\infty)$.

Elements of $\mathcal{B}(\infty)$ are equivalence classes of paths $[\pi]$ in $\mathcal{B}(k\rho)$ for $k \gg 0$, where ρ is the Weyl vector. A canonical representative for an element of $\mathcal{B}(\infty)$ is chosen by taking k to be minimal such that the endpoint of π is strictly dominant but its representative in $\mathcal{B}((k-1)\rho)$ is on the wall of the dominant chamber.

REFERENCES:

- [LZ2011]

class Element

Bases: *Element*

e(*i*, *power=1*, *length_only=False*)

Return the i-th crystal raising operator on `self`.

INPUT:

- `i` – element of the index set
- `power` – (default: 1) positive integer; specifies the power of the lowering operator to be applied
- `length_only` – boolean (default: `False`); if `True`, then return the distance to the anti-dominant end of the i-string of `self`

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['B',3,1])
sage: mg = B.module_generator()
sage: mg.e(0)
sage: mg.e(1)
sage: mg.e(2)
sage: x = mg.f_string([1,0,2,1,0,2,1,1,0])
```

(continues on next page)

(continued from previous page)

```
sage: all(x.f(i).e(i) == x for i in B.index_set())
True
sage: all(x.e(i).f(i) == x for i in B.index_set() if x.epsilon(i) > 0)
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.LSPaths(['B',Integer(3),Integer(1)])
>>> mg = B.module_generator()
>>> mg.e(Integer(0))
>>> mg.e(Integer(1))
>>> mg.e(Integer(2))
>>> x = mg.f_string([Integer(1),Integer(0),Integer(2),Integer(1),
↪Integer(0),Integer(2),Integer(1),Integer(1),Integer(0)])
>>> all(x.f(i).e(i) == x for i in B.index_set())
True
>>> all(x.e(i).f(i) == x for i in B.index_set() if x.epsilon(i) >␣
↪Integer(0))
True
```

f(*i*, *power=1*, *length_only=False*)

Return the i-th crystal lowering operator on `self`.

INPUT:

- `i` – element of the index set
- `power` – (default: 1) positive integer; specifies the power of the lowering operator to be applied
- `length_only` – boolean (default: `False`); if `True`, then return the distance to the anti-dominant end of the i-string of `self`

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['D',3,2])
sage: mg = B.highest_weight_vector()
sage: mg.f(1)
(3*Lambda[0] - Lambda[1] + 3*Lambda[2],
 2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
sage: mg.f(2)
(Lambda[0] + 2*Lambda[1] - Lambda[2],
 2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
sage: mg.f(0)
(-Lambda[0] + 2*Lambda[1] + Lambda[2] - delta,
 2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
```

```
>>> from sage.all import *
>>> B = crystals.infinity.LSPaths(['D',Integer(3),Integer(2)])
>>> mg = B.highest_weight_vector()
>>> mg.f(Integer(1))
(3*Lambda[0] - Lambda[1] + 3*Lambda[2],
 2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
>>> mg.f(Integer(2))
(Lambda[0] + 2*Lambda[1] - Lambda[2],
 2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
>>> mg.f(Integer(0))
(-Lambda[0] + 2*Lambda[1] + Lambda[2] - delta,
```

(continues on next page)

(continued from previous page)

```
2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
```

phi(*i*)

Return φ_i of `self`.

Let $\pi \in \mathcal{B}(\infty)$. Define

$$\varphi_i(\pi) := \varepsilon_i(\pi) + \langle h_i, \mathrm{wt}(\pi) \rangle,$$

where h_i is the i-th simple coroot and $\mathrm{wt}(\pi)$ is the *weight()* of π.

INPUT:

- `i` – element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['D',4])
sage: mg = B.highest_weight_vector()
sage: x = mg.f_string([1,3,4,2,4,3,2,1,4])
sage: [x.phi(i) for i in B.index_set()]
[-1, 4, -2, -3]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.LSPaths(['D',Integer(4)])
>>> mg = B.highest_weight_vector()
>>> x = mg.f_string([Integer(1),Integer(3),Integer(4),Integer(2),
↪Integer(4),Integer(3),Integer(2),Integer(1),Integer(4)])
>>> [x.phi(i) for i in B.index_set()]
[-1, 4, -2, -3]
```

weight()

Return the weight of `self`.

Todo

This is a generic algorithm. We should find a better description and implement it.

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['E',6])
sage: mg = B.highest_weight_vector()
sage: f_seq = [1,4,2,6,4,2,3,1,5,5]
sage: x = mg.f_string(f_seq)
sage: x.weight()
-3*Lambda[1] - 2*Lambda[2] + 2*Lambda[3] + Lambda[4] - Lambda[5]

sage: al = B.cartan_type().root_system().weight_space().simple_roots()
sage: x.weight() == -sum(al[i] for i in f_seq)
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.LSPaths(['E',Integer(6)])
>>> mg = B.highest_weight_vector()
```

(continues on next page)

(continued from previous page)

```
>>> f_seq = [Integer(1),Integer(4),Integer(2),Integer(6),Integer(4),
↪Integer(2),Integer(3),Integer(1),Integer(5),Integer(5)]
>>> x = mg.f_string(f_seq)
>>> x.weight()
-3*Lambda[1] - 2*Lambda[2] + 2*Lambda[3] + Lambda[4] - Lambda[5]

>>> al = B.cartan_type().root_system().weight_space().simple_roots()
>>> x.weight() == -sum(al[i] for i in f_seq)
True
```

module_generator()

Return the module generator (or highest weight element) of `self`.

The module generator is the unique path $\pi_\infty : t \mapsto t\rho$, for $t \in [0, \infty)$.

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['A',6,2])
sage: mg = B.module_generator(); mg
(Lambda[0] + Lambda[1] + Lambda[2] + Lambda[3],)
sage: mg.weight()
0
```

```
>>> from sage.all import *
>>> B = crystals.infinity.LSPaths(['A',Integer(6),Integer(2)])
>>> mg = B.module_generator(); mg
(Lambda[0] + Lambda[1] + Lambda[2] + Lambda[3],)
>>> mg.weight()
0
```

weight_lattice_realization()

Return the weight lattice realization of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['C',4])
sage: B.weight_lattice_realization()
Weight space over the Rational Field of the Root system of type ['C', 4]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.LSPaths(['C',Integer(4)])
>>> B.weight_lattice_realization()
Weight space over the Rational Field of the Root system of type ['C', 4]
```

sage.combinat.crystals.littelmann_path.**positively_parallel_weights**(v, w)

Check whether the vectors `v` and `w` are positive scalar multiples of each other.

EXAMPLES:

```
sage: from sage.combinat.crystals.littelmann_path import positively_parallel_
↪weights
sage: La = RootSystem(['A',5,2]).weight_space(extended=True).fundamental_weights()
sage: rho = sum(La)
```

(continues on next page)

(continued from previous page)

```
sage: positively_parallel_weights(rho, 4*rho)
True
sage: positively_parallel_weights(4*rho, rho)
True
sage: positively_parallel_weights(rho, -rho)
False
sage: positively_parallel_weights(rho, La[1] + La[2])
False
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.littelmann_path import positively_parallel_weights
>>> La = RootSystem(['A',Integer(5),Integer(2)]).weight_space(extended=True).
↪fundamental_weights()
>>> rho = sum(La)
>>> positively_parallel_weights(rho, Integer(4)*rho)
True
>>> positively_parallel_weights(Integer(4)*rho, rho)
True
>>> positively_parallel_weights(rho, -rho)
False
>>> positively_parallel_weights(rho, La[Integer(1)] + La[Integer(2)])
False
```

5.1.59 Crystals of Modified Nakajima Monomials

AUTHORS:

- Arthur Lubovsky: Initial version
- Ben Salisbury: Initial version

Let $Y_{i,k}$, for $i \in I$ and $k \in \mathbf{Z}$, be a commuting set of variables, and let $\mathbf{1}$ be a new variable which commutes with each $Y_{i,k}$. (Here, I represents the index set of a Cartan datum.) One may endow the structure of a crystal on the set $\widehat{\mathcal{M}}$ of monomials of the form

$$M = \prod_{(i,k)\in I\times \mathbf{Z}_{\geq 0}} Y_{i,k}^{y_i(k)}\mathbf{1}.$$

Elements of $\widehat{\mathcal{M}}$ are called *modified Nakajima monomials*. We will omit the $\mathbf{1}$ from the end of a monomial if there exists at least one $y_i(k) \neq 0$. The crystal structure on this set is defined by

$$\begin{aligned}
\mathrm{wt}(M) &= \sum_{i\in I}\Bigl(\sum_{k\geq 0} y_i(k)\Bigr)\Lambda_i,\\
\varphi_i(M) &= \max\Bigl\{\sum_{0\leq j\leq k} y_i(j) : k \geq 0\Bigr\},\\
\varepsilon_i(M) &= \varphi_i(M) - \langle h_i, \mathrm{wt}(M)\rangle,\\
k_f = k_f(M) &= \min\Bigl\{k \geq 0 : \varphi_i(M) = \sum_{0\leq j\leq k} y_i(j)\Bigr\},\\
k_e = k_e(M) &= \max\Bigl\{k \geq 0 : \varphi_i(M) = \sum_{0\leq j\leq k} y_i(j)\Bigr\},
\end{aligned}$$

where $\{h_i : i \in I\}$ and $\{\Lambda_i : i \in I\}$ are the simple coroots and fundamental weights, respectively. With a chosen set of integers $C = (c_{ij})_{i \neq j}$ such that $c_{ij} + c_{ji} = 1$, one defines

$$A_{i,k} = Y_{i,k} Y_{i,k+1} \prod_{j \neq i} Y_{j,k+c_{ji}}^{a_{ji}},$$

where (a_{ij}) is a Cartan matrix. Then

$$e_i M = \begin{cases} 0 & \text{if } \varepsilon_i(M) = 0, \\ A_{i,k_e} M & \text{if } \varepsilon_i(M) > 0, \end{cases}$$
$$f_i M = A_{i,k_f}^{-1} M.$$

It is shown in [KKS2007] that the connected component of $\widehat{\mathcal{M}}$ containing the element $\mathbf{1}$, which we denote by $\mathcal{M}(\infty)$, is crystal isomorphic to the crystal $B(\infty)$.

Let $\widetilde{\mathcal{M}}$ be $\widehat{\mathcal{M}}$ as a set, and with crystal structure defined as on $\widehat{\mathcal{M}}$ with the exception that

$$f_i M = \begin{cases} 0 & \text{if } \varphi_i(M) = 0, \\ A_{i,k_f}^{-1} M & \text{if } \varphi_i(M) > 0. \end{cases}$$

Then Kashiwara [Ka2003] showed that the connected component in $\widetilde{\mathcal{M}}$ containing a monomial M such that $e_i M = 0$, for all $i \in I$, is crystal isomorphic to the irreducible highest weight crystal $B(\mathrm{wt}(M))$.

WARNING:

Monomial crystals depend on the choice of positive integers $C = (c_{ij})_{i \neq j}$ satisfying the condition $c_{ij} + c_{ji} = 1$. We have chosen such integers uniformly such that $c_{ij} = 1$ if $i < j$ and $c_{ij} = 0$ if $i > j$.

class sage.combinat.crystals.monomial_crystals.**CrystalOfNakajimaMonomials**(*ct*, *La*, *c*)

Bases: *InfinityCrystalOfNakajimaMonomials*

Let $\widetilde{\mathcal{M}}$ be $\widehat{\mathcal{M}}$ as a set, and with crystal structure defined as on $\widehat{\mathcal{M}}$ with the exception that

$$f_i M = \begin{cases} 0 & \text{if } \varphi_i(M) = 0, \\ A_{i,k_f}^{-1} M & \text{if } \varphi_i(M) > 0. \end{cases}$$

Then Kashiwara [Ka2003] showed that the connected component in $\widetilde{\mathcal{M}}$ containing a monomial M such that $e_i M = 0$, for all $i \in I$, is crystal isomorphic to the irreducible highest weight crystal $B(\mathrm{wt}(M))$.

INPUT:

- `ct` – a Cartan type
- `La` – an element of the weight lattice

EXAMPLES:

```
sage: La = RootSystem("A2").weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials("A2",La[1]+La[2])
sage: B = crystals.Tableaux("A2",shape=[2,1])
sage: GM = M.digraph()
sage: GB = B.digraph()
sage: GM.is_isomorphic(GB,edge_labels=True)
True

sage: La = RootSystem("G2").weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials("G2",La[1]+La[2])
sage: B = crystals.Tableaux("G2",shape=[2,1])
```

(continues on next page)

(continued from previous page)

```
sage: GM = M.digraph()
sage: GB = B.digraph()
sage: GM.is_isomorphic(GB,edge_labels=True)
True

sage: La = RootSystem("B2").weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(['B',2],La[1]+La[2])
sage: B = crystals.Tableaux("B2",shape=[3/2,1/2])
sage: GM = M.digraph()
sage: GB = B.digraph()
sage: GM.is_isomorphic(GB,edge_labels=True)
True

sage: La = RootSystem(['A',3,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: M = crystals.NakajimaMonomials(['A',3,1],La[0]+La[2])
sage: B = crystals.GeneralizedYoungWalls(3,La[0]+La[2])
sage: SM = M.subcrystal(max_depth=4)
sage: SB = B.subcrystal(max_depth=4)
sage: GM = M.digraph(subset=SM) # long time
sage: GB = B.digraph(subset=SB) # long time
sage: GM.is_isomorphic(GB,edge_labels=True) # long time
True

sage: La = RootSystem(['A',5,2]).weight_lattice(extended=True).fundamental_
↪weights()
sage: LA = RootSystem(['A',5,2]).weight_space().fundamental_weights()
sage: M = crystals.NakajimaMonomials(['A',5,2],3*La[0])
sage: B = crystals.LSPaths(3*LA[0])
sage: SM = M.subcrystal(max_depth=4)
sage: SB = B.subcrystal(max_depth=4)
sage: GM = M.digraph(subset=SM)
sage: GB = B.digraph(subset=SB)
sage: GM.is_isomorphic(GB,edge_labels=True)
True

sage: c = matrix([[0,1,0],[0,0,1],[1,0,0]])
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: M = crystals.NakajimaMonomials(2*La[1], c=c)
sage: sorted(M.subcrystal(max_depth=3), key=str)
[Y(0,0) Y(0,1) Y(1,0) Y(2,1)^-1,
 Y(0,0) Y(0,1)^2 Y(1,1)^-1 Y(2,0) Y(2,1)^-1,
 Y(0,0) Y(0,2)^-1 Y(1,0) Y(1,1) Y(2,1)^-1 Y(2,2),
 Y(0,1) Y(0,2)^-1 Y(1,1)^-1 Y(2,0)^2 Y(2,2),
 Y(0,1) Y(1,0) Y(1,1)^-1 Y(2,0),
 Y(0,1)^2 Y(1,1)^-2 Y(2,0)^2,
 Y(0,2)^-1 Y(1,0) Y(2,0) Y(2,2),
 Y(1,0) Y(1,3) Y(2,0) Y(2,3)^-1,
 Y(1,0)^2]
```

```
>>> from sage.all import *
>>> La = RootSystem("A2").weight_lattice().fundamental_weights()
>>> M = crystals.NakajimaMonomials("A2",La[Integer(1)]+La[Integer(2)])
>>> B = crystals.Tableaux("A2",shape=[Integer(2),Integer(1)])
>>> GM = M.digraph()
>>> GB = B.digraph()
>>> GM.is_isomorphic(GB,edge_labels=True)
True

>>> La = RootSystem("G2").weight_lattice().fundamental_weights()
>>> M = crystals.NakajimaMonomials("G2",La[Integer(1)]+La[Integer(2)])
>>> B = crystals.Tableaux("G2",shape=[Integer(2),Integer(1)])
>>> GM = M.digraph()
>>> GB = B.digraph()
>>> GM.is_isomorphic(GB,edge_labels=True)
True

>>> La = RootSystem("B2").weight_lattice().fundamental_weights()
>>> M = crystals.NakajimaMonomials(['B',Integer(2)],La[Integer(1)]+La[Integer(2)])
>>> B = crystals.Tableaux("B2",shape=[Integer(3)/Integer(2),Integer(1)/
↪Integer(2)])
>>> GM = M.digraph()
>>> GB = B.digraph()
>>> GM.is_isomorphic(GB,edge_labels=True)
True

>>> La = RootSystem(['A',Integer(3),Integer(1)]).weight_lattice(extended=True).
↪fundamental_weights()
>>> M = crystals.NakajimaMonomials(['A',Integer(3),Integer(1)],
↪La[Integer(0)]+La[Integer(2)])
>>> B = crystals.GeneralizedYoungWalls(Integer(3),La[Integer(0)]+La[Integer(2)])
>>> SM = M.subcrystal(max_depth=Integer(4))
>>> SB = B.subcrystal(max_depth=Integer(4))
>>> GM = M.digraph(subset=SM) # long time
>>> GB = B.digraph(subset=SB) # long time
>>> GM.is_isomorphic(GB,edge_labels=True) # long time
True

>>> La = RootSystem(['A',Integer(5),Integer(2)]).weight_lattice(extended=True).
↪fundamental_weights()
>>> LA = RootSystem(['A',Integer(5),Integer(2)]).weight_space().fundamental_
↪weights()
>>> M = crystals.NakajimaMonomials(['A',Integer(5),Integer(2)],
↪Integer(3)*La[Integer(0)])
>>> B = crystals.LSPaths(Integer(3)*LA[Integer(0)])
>>> SM = M.subcrystal(max_depth=Integer(4))
>>> SB = B.subcrystal(max_depth=Integer(4))
>>> GM = M.digraph(subset=SM)
>>> GB = B.digraph(subset=SB)
>>> GM.is_isomorphic(GB,edge_labels=True)
True

>>> c = matrix([[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)],[Integer(1),Integer(0),Integer(0)]])
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice(extended=True).
↪fundamental_weights()
>>> M = crystals.NakajimaMonomials(Integer(2)*La[Integer(1)], c=c)
>>> sorted(M.subcrystal(max_depth=Integer(3)), key=str)
[Y(0,0) Y(0,1) Y(1,0) Y(2,1)^-1,
 Y(0,0) Y(0,1)^2 Y(1,1)^-1 Y(2,0) Y(2,1)^-1,
 Y(0,0) Y(0,2)^-1 Y(1,0) Y(1,1) Y(2,1)^-1 Y(2,2),
 Y(0,1) Y(0,2)^-1 Y(1,1)^-1 Y(2,0)^2 Y(2,2),
 Y(0,1) Y(1,0) Y(1,1)^-1 Y(2,0),
 Y(0,1)^2 Y(1,1)^-2 Y(2,0)^2,
 Y(0,2)^-1 Y(1,0) Y(2,0) Y(2,2),
 Y(1,0) Y(1,3) Y(2,0) Y(2,3)^-1,
 Y(1,0)^2]
```

Element

alias of *CrystalOfNakajimaMonomialsElement*

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: La = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(['A',2], La[1])
sage: M.cardinality()
3

sage: La = RootSystem(['D',4,2]).weight_lattice(extended=True).fundamental_
↪weights()
sage: M = crystals.NakajimaMonomials(['D',4,2], La[1])
sage: M.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2)]).weight_lattice().fundamental_weights()
>>> M = crystals.NakajimaMonomials(['A',Integer(2)], La[Integer(1)])
>>> M.cardinality()
3

>>> La = RootSystem(['D',Integer(4),Integer(2)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> M = crystals.NakajimaMonomials(['D',Integer(4),Integer(2)],␣
↪La[Integer(1)])
>>> M.cardinality()
+Infinity
```

class sage.combinat.crystals.monomial_crystals.**CrystalOfNakajimaMonomialsElement**(*parent*, *Y*, *A*)

Bases: *NakajimaMonomial*

Element class for *CrystalOfNakajimaMonomials*.

The f_i operators need to be modified from the version in `monomial_crystalsNakajimaMonomial` in order to create irreducible highest weight realizations. This modified f_i is defined as

$$f_i M = \begin{cases} 0 & \text{if } \varphi_i(M) = 0, \\ A_{i,k_f}^{-1} M & \text{if } \varphi_i(M) > 0. \end{cases}$$

EXAMPLES:

```
sage: La = RootSystem(['A',5,2]).weight_lattice(extended=True).fundamental_
↪weights()
sage: M = crystals.NakajimaMonomials(['A',5,2],3*La[0])
sage: m = M.module_generators[0].f(0); m
Y(0,0)^2 Y(0,1)^-1 Y(2,0)
sage: TestSuite(m).run()
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(5),Integer(2)]).weight_lattice(extended=True).
↪fundamental_weights()
>>> M = crystals.NakajimaMonomials(['A',Integer(5),Integer(2)],
↪Integer(3)*La[Integer(0)])
>>> m = M.module_generators[Integer(0)].f(Integer(0)); m
Y(0,0)^2 Y(0,1)^-1 Y(2,0)
>>> TestSuite(m).run()
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: La = RootSystem(['A',5,2]).weight_lattice(extended=True).fundamental_
↪weights()
sage: M = crystals.NakajimaMonomials(['A',5,2],3*La[0])
sage: m = M.module_generators[0]
sage: [m.f(i) for i in M.index_set()]
[Y(0,0)^2 Y(0,1)^-1 Y(2,0), None, None, None]
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(5),Integer(2)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> M = crystals.NakajimaMonomials(['A',Integer(5),Integer(2)],
↪Integer(3)*La[Integer(0)])
>>> m = M.module_generators[Integer(0)]
>>> [m.f(i) for i in M.index_set()]
[Y(0,0)^2 Y(0,1)^-1 Y(2,0), None, None, None]
```

```
sage: M = crystals.infinity.NakajimaMonomials("E8")
sage: M.set_variables('A')
sage: m = M.module_generators[0].f_string([4,2,3,8])
sage: m
A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1
sage: [m.f(i) for i in M.index_set()]
```

(continues on next page)

(continued from previous page)

```
[A(1,2)^-1 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1,
 A(2,0)^-1 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,0)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(4,1)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(5,0)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(6,0)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(7,1)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-2]
sage: M.set_variables('Y')
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials("E8")
>>> M.set_variables('A')
>>> m = M.module_generators[Integer(0)].f_string([Integer(4),Integer(2),
↪Integer(3),Integer(8)])
>>> m
A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1
>>> [m.f(i) for i in M.index_set()]
[A(1,2)^-1 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1,
 A(2,0)^-1 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,0)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(4,1)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(5,0)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(6,0)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(7,1)^-1 A(8,0)^-1,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-2]
>>> M.set_variables('Y')
```

weight()

Return the weight of `self` as an element of the weight lattice.

EXAMPLES:

```
sage: La = RootSystem("A2").weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials("A2",La[1]+La[2])
sage: M.module_generators[0].weight()
(2, 1, 0)
```

```
>>> from sage.all import *
>>> La = RootSystem("A2").weight_lattice().fundamental_weights()
>>> M = crystals.NakajimaMonomials("A2",La[Integer(1)]+La[Integer(2)])
>>> M.module_generators[Integer(0)].weight()
(2, 1, 0)
```

class sage.combinat.crystals.monomial_crystals.**InfinityCrystalOfNakajimaMonomials**(*ct*, *c*, *category=None*)

Bases: `UniqueRepresentation`, `Parent`

Crystal $B(\infty)$ in terms of (modified) Nakajima monomials.

Let $Y_{i,k}$, for $i \in I$ and $k \in \mathbf{Z}$, be a commuting set of variables, and let $\mathbf{1}$ be a new variable which commutes with each $Y_{i,k}$. (Here, I represents the index set of a Cartan datum.) One may endow the structure of a crystal on the set $\widehat{\mathcal{M}}$ of monomials of the form

$$M = \prod_{(i,k) \in I \times \mathbf{Z}_{\geq 0}} Y_{i,k}^{y_i(k)} \mathbf{1}.$$

Elements of $\widehat{\mathcal{M}}$ are called *modified Nakajima monomials*. We will omit the $\mathbf{1}$ from the end of a monomial if there exists at least one $y_i(k) \neq 0$. The crystal structure on this set is defined by

$$\begin{aligned}
\mathrm{wt}(M) &= \sum_{i \in I} \Bigl(\sum_{k \geq 0} y_i(k) \Bigr) \Lambda_i, \\
\varphi_i(M) &= \max \Bigl\{ \sum_{0 \leq j \leq k} y_i(j) : k \geq 0 \Bigr\}, \\
\varepsilon_i(M) &= \varphi_i(M) - \langle h_i, \mathrm{wt}(M) \rangle, \\
k_f = k_f(M) &= \min \Bigl\{ k \geq 0 : \varphi_i(M) = \sum_{0 \leq j \leq k} y_i(j) \Bigr\}, \\
k_e = k_e(M) &= \max \Bigl\{ k \geq 0 : \varphi_i(M) = \sum_{0 \leq j \leq k} y_i(j) \Bigr\},
\end{aligned}$$

where $\{h_i : i \in I\}$ and $\{\Lambda_i : i \in I\}$ are the simple coroots and fundamental weights, respectively. With a chosen set of nonnegative integers $C = (c_{ij})_{i \neq j}$ such that $c_{ij} + c_{ji} = 1$, one defines

$$A_{i,k} = Y_{i,k} Y_{i,k+1} \prod_{j \neq i} Y_{j,k+c_{ji}}^{a_{ji}},$$

where $(a_{ij})_{i,j \in I}$ is a Cartan matrix. Then

$$\begin{aligned}
e_i M &= \begin{cases} 0 & \text{if } \varepsilon_i(M) = 0, \\ A_{i,k_e} M & \text{if } \varepsilon_i(M) > 0, \end{cases} \\
f_i M &= A_{i,k_f}^{-1} M.
\end{aligned}$$

It is shown in [KKS2007] that the connected component of $\widehat{\mathcal{M}}$ containing the element $\mathbf{1}$, which we denote by $\mathcal{M}(\infty)$, is crystal isomorphic to the crystal $B(\infty)$.

INPUT:

- `cartan_type` – a Cartan type
- `c` – (optional) the matrix $(c_{ij})_{i,j \in I}$ such that $c_{ii} = 0$ for all $i \in I$, $c_{ij} \in \mathbf{Z}_{>0}$ for all $i, j \in I$, and $c_{ij} + c_{ji} = 1$ for all $i \neq j$; the default is $c_{ij} = 0$ if $i < j$ and 0 otherwise

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("C3")
sage: S = B.subcrystal(max_depth=4)
sage: G = B.digraph(subset=S) # long time
sage: M = crystals.infinity.NakajimaMonomials("C3") # long time
sage: T = M.subcrystal(max_depth=4) # long time
sage: H = M.digraph(subset=T) # long time
sage: G.is_isomorphic(H,edge_labels=True) # long time
True

sage: M = crystals.infinity.NakajimaMonomials(['A',2,1])
sage: T = M.subcrystal(max_depth=3)
sage: H = M.digraph(subset=T) # long time
```

(continues on next page)

(continued from previous page)

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(2)
sage: YS = Y.subcrystal(max_depth=3)
sage: YG = Y.digraph(subset=YS) # long time
sage: YG.is_isomorphic(H,edge_labels=True) # long time
True

sage: M = crystals.infinity.NakajimaMonomials("D4")
sage: B = crystals.infinity.Tableaux("D4")
sage: MS = M.subcrystal(max_depth=3)
sage: BS = B.subcrystal(max_depth=3)
sage: MG = M.digraph(subset=MS) # long time
sage: BG = B.digraph(subset=BS) # long time
sage: BG.is_isomorphic(MG,edge_labels=True) # long time
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("C3")
>>> S = B.subcrystal(max_depth=Integer(4))
>>> G = B.digraph(subset=S) # long time
>>> M = crystals.infinity.NakajimaMonomials("C3") # long time
>>> T = M.subcrystal(max_depth=Integer(4)) # long time
>>> H = M.digraph(subset=T) # long time
>>> G.is_isomorphic(H,edge_labels=True) # long time
True

>>> M = crystals.infinity.NakajimaMonomials(['A',Integer(2),Integer(1)])
>>> T = M.subcrystal(max_depth=Integer(3))
>>> H = M.digraph(subset=T) # long time
>>> Y = crystals.infinity.GeneralizedYoungWalls(Integer(2))
>>> YS = Y.subcrystal(max_depth=Integer(3))
>>> YG = Y.digraph(subset=YS) # long time
>>> YG.is_isomorphic(H,edge_labels=True) # long time
True

>>> M = crystals.infinity.NakajimaMonomials("D4")
>>> B = crystals.infinity.Tableaux("D4")
>>> MS = M.subcrystal(max_depth=Integer(3))
>>> BS = B.subcrystal(max_depth=Integer(3))
>>> MG = M.digraph(subset=MS) # long time
>>> BG = B.digraph(subset=BS) # long time
>>> BG.is_isomorphic(MG,edge_labels=True) # long time
True
```

Element

alias of *NakajimaMonomial*

c()

Return the matrix c_{ij} of `self`.

EXAMPLES:

```
sage: La = RootSystem(['B',3]).weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(La[1]+La[2])
```

(continues on next page)

(continued from previous page)

```
sage: M.c()
[0 1 1]
[0 0 1]
[0 0 0]

sage: c = Matrix([[0,0,1],[1,0,0],[0,1,0]])
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: M = crystals.NakajimaMonomials(2*La[1], c=c)
sage: M.c() == c
True
```

```
>>> from sage.all import *
>>> La = RootSystem(['B',Integer(3)]).weight_lattice().fundamental_weights()
>>> M = crystals.NakajimaMonomials(La[Integer(1)]+La[Integer(2)])
>>> M.c()
[0 1 1]
[0 0 1]
[0 0 0]

>>> c = Matrix([[Integer(0),Integer(0),Integer(1)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0)]])
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> M = crystals.NakajimaMonomials(Integer(2)*La[Integer(1)], c=c)
>>> M.c() == c
True
```

cardinality()

Return the cardinality of `self`, which is always ∞.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['A',5,2])
sage: M.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials(['A',Integer(5),Integer(2)])
>>> M.cardinality()
+Infinity
```

get_variables()

Return the type of monomials to use for the element output.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['A', 4])
sage: M.get_variables()
'Y'
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials(['A', Integer(4)])
>>> M.get_variables()
'Y'
```

set_variables(*letter*)

Set the type of monomials to use for the element output.

If the A variables are used, the output is written as $\prod_{i\in I} Y_{i,0}^{\lambda_i} \prod_{i,k} A_{i,k}^{c_{i,k}}$, where $\sum_{i\in I} \lambda_i \Lambda_i$ is the corresponding dominant weight.

INPUT:

- `letter` – can be one of the following:
  - `'Y'` – use $Y_{i,k}$, corresponds to fundamental weights
  - `'A'` – use $A_{i,k}$, corresponds to simple roots

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['A', 4])
sage: elt = M.highest_weight_vector().f_string([2,1,3,2,3,2,4,3])
sage: elt
Y(1,2) Y(2,0)^-1 Y(2,2)^-1 Y(3,0)^-1 Y(3,2)^-1 Y(4,0)
sage: M.set_variables('A')
sage: elt
A(1,1)^-1 A(2,0)^-1 A(2,1)^-2 A(3,0)^-2 A(3,1)^-1 A(4,0)^-1
sage: M.set_variables('Y')
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials(['A', Integer(4)])
>>> elt = M.highest_weight_vector().f_string([Integer(2),Integer(1),
↪Integer(3),Integer(2),Integer(3),Integer(2),Integer(4),Integer(3)])
>>> elt
Y(1,2) Y(2,0)^-1 Y(2,2)^-1 Y(3,0)^-1 Y(3,2)^-1 Y(4,0)
>>> M.set_variables('A')
>>> elt
A(1,1)^-1 A(2,0)^-1 A(2,1)^-2 A(3,0)^-2 A(3,1)^-1 A(4,0)^-1
>>> M.set_variables('Y')
```

```
sage: La = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(La[1]+La[2])
sage: lw = M.lowest_weight_vectors()[0]
sage: lw
Y(1,2)^-1 Y(2,1)^-1
sage: M.set_variables('A')
sage: lw
Y(1,0) Y(2,0) A(1,0)^-1 A(1,1)^-1 A(2,0)^-2
sage: M.set_variables('Y')
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2)]).weight_lattice().fundamental_weights()
>>> M = crystals.NakajimaMonomials(La[Integer(1)]+La[Integer(2)])
>>> lw = M.lowest_weight_vectors()[Integer(0)]
```

(continues on next page)

(continued from previous page)

```
>>> lw
Y(1,2)^-1 Y(2,1)^-1
>>> M.set_variables('A')
>>> lw
Y(1,0) Y(2,0) A(1,0)^-1 A(1,1)^-1 A(2,0)^-2
>>> M.set_variables('Y')
```

class sage.combinat.crystals.monomial_crystals.**NakajimaMonomial**(*parent*, Y, A)

Bases: Element

An element of the monomial crystal.

Monomials of the form $Y_{i_1,k_1}^{y_1} \cdots Y_{i_t,k_t}^{y_t}$, where $i_1, \ldots, i_t$ are elements of the index set, $k_1, \ldots, k_t$ are nonnegative integers, and $y_1, \ldots, y_t$ are integers.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['B',3,1])
sage: mg = M.module_generators[0]
sage: mg
1
sage: mg.f_string([1,3,2,0,1,2,3,0,0,1])
Y(0,0)^-1 Y(0,1)^-1 Y(0,2)^-1 Y(0,3)^-1 Y(1,0)^-3
 Y(1,1)^-2 Y(1,2) Y(2,0)^3 Y(2,2) Y(3,0) Y(3,2)^-1
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials(['B',Integer(3),Integer(1)])
>>> mg = M.module_generators[Integer(0)]
>>> mg
1
>>> mg.f_string([Integer(1),Integer(3),Integer(2),Integer(0),Integer(1),
↪Integer(2),Integer(3),Integer(0),Integer(0),Integer(1)])
Y(0,0)^-1 Y(0,1)^-1 Y(0,2)^-1 Y(0,3)^-1 Y(1,0)^-3
 Y(1,1)^-2 Y(1,2) Y(2,0)^3 Y(2,2) Y(3,0) Y(3,2)^-1
```

An example using the A variables:

```
sage: M = crystals.infinity.NakajimaMonomials("A3")
sage: M.set_variables('A')
sage: mg = M.module_generators[0]
sage: mg.f_string([1,2,3,2,1])
A(1,0)^-1 A(1,1)^-1 A(2,0)^-2 A(3,0)^-1
sage: mg.f_string([3,2,1])
A(1,2)^-1 A(2,1)^-1 A(3,0)^-1
sage: M.set_variables('Y')
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials("A3")
>>> M.set_variables('A')
>>> mg = M.module_generators[Integer(0)]
>>> mg.f_string([Integer(1),Integer(2),Integer(3),Integer(2),Integer(1)])
A(1,0)^-1 A(1,1)^-1 A(2,0)^-2 A(3,0)^-1
>>> mg.f_string([Integer(3),Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
A(1,2)^-1 A(2,1)^-1 A(3,0)^-1
>>> M.set_variables('Y')
```

e(*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['E',7,1])
sage: m = M.module_generators[0].f_string([0,1,4,3])
sage: [m.e(i) for i in M.index_set()]
[None,
 None,
 None,
 Y(0,0)^-1 Y(1,1)^-1 Y(2,1) Y(3,0) Y(3,1) Y(4,0)^-1 Y(4,1)^-1 Y(5,0),
 None,
 None,
 None,
 None]

sage: M = crystals.infinity.NakajimaMonomials("C5")
sage: m = M.module_generators[0].f_string([1,3])
sage: [m.e(i) for i in M.index_set()]
[Y(2,1) Y(3,0)^-1 Y(3,1)^-1 Y(4,0),
 None,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0),
 None,
 None]

sage: M = crystals.infinity.NakajimaMonomials(['D',4,1])
sage: M.set_variables('A')
sage: m = M.module_generators[0].f_string([4,2,3,0])
sage: [m.e(i) for i in M.index_set()]
[A(2,1)^-1 A(3,1)^-1 A(4,0)^-1,
 None,
 None,
 A(0,2)^-1 A(2,1)^-1 A(4,0)^-1,
 None]
sage: M.set_variables('Y')
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials(['E',Integer(7),Integer(1)])
>>> m = M.module_generators[Integer(0)].f_string([Integer(0),Integer(1),
↪Integer(4),Integer(3)])
>>> [m.e(i) for i in M.index_set()]
[None,
 None,
 None,
 Y(0,0)^-1 Y(1,1)^-1 Y(2,1) Y(3,0) Y(3,1) Y(4,0)^-1 Y(4,1)^-1 Y(5,0),
```

(continues on next page)

(continued from previous page)

```
 None,
 None,
 None,
 None]

>>> M = crystals.infinity.NakajimaMonomials("C5")
>>> m = M.module_generators[Integer(0)].f_string([Integer(1),Integer(3)])
>>> [m.e(i) for i in M.index_set()]
[Y(2,1) Y(3,0)^-1 Y(3,1)^-1 Y(4,0),
 None,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0),
 None,
 None]

>>> M = crystals.infinity.NakajimaMonomials(['D',Integer(4),Integer(1)])
>>> M.set_variables('A')
>>> m = M.module_generators[Integer(0)].f_string([Integer(4),Integer(2),
↪Integer(3),Integer(0)])
>>> [m.e(i) for i in M.index_set()]
[A(2,1)^-1 A(3,1)^-1 A(4,0)^-1,
 None,
 None,
 A(0,2)^-1 A(2,1)^-1 A(4,0)^-1,
 None]
>>> M.set_variables('Y')
```

epsilon(*i*)

Return the value of ε_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['G',2,1])
sage: m = M.module_generators[0].f(2)
sage: [m.epsilon(i) for i in M.index_set()]
[0, 0, 1]

sage: M = crystals.infinity.NakajimaMonomials(['C',4,1])
sage: m = M.module_generators[0].f_string([4,2,3])
sage: [m.epsilon(i) for i in M.index_set()]
[0, 0, 0, 1, 0]
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials(['G',Integer(2),Integer(1)])
>>> m = M.module_generators[Integer(0)].f(Integer(2))
>>> [m.epsilon(i) for i in M.index_set()]
[0, 0, 1]

>>> M = crystals.infinity.NakajimaMonomials(['C',Integer(4),Integer(1)])
>>> m = M.module_generators[Integer(0)].f_string([Integer(4),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)])
>>> [m.epsilon(i) for i in M.index_set()]
[0, 0, 0, 1, 0]
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials("B4")
sage: m = M.module_generators[0].f_string([1,3,4])
sage: [m.f(i) for i in M.index_set()]
[Y(1,0)^-2 Y(1,1)^-2 Y(2,0)^2 Y(2,1) Y(3,0)^-1 Y(4,0) Y(4,1)^-1,
 Y(1,0)^-1 Y(1,1)^-1 Y(1,2) Y(2,0) Y(2,2)^-1 Y(3,0)^-1 Y(3,1) Y(4,0) Y(4,1)^-
↪1,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0) Y(2,1)^2 Y(3,0)^-2 Y(3,1)^-1 Y(4,0)^3 Y(4,1)^-1,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0) Y(2,1) Y(3,0)^-1 Y(3,1) Y(4,1)^-2]
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials("B4")
>>> m = M.module_generators[Integer(0)].f_string([Integer(1),Integer(3),
↪Integer(4)])
>>> [m.f(i) for i in M.index_set()]
[Y(1,0)^-2 Y(1,1)^-2 Y(2,0)^2 Y(2,1) Y(3,0)^-1 Y(4,0) Y(4,1)^-1,
 Y(1,0)^-1 Y(1,1)^-1 Y(1,2) Y(2,0) Y(2,2)^-1 Y(3,0)^-1 Y(3,1) Y(4,0) Y(4,1)^-
↪1,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0) Y(2,1)^2 Y(3,0)^-2 Y(3,1)^-1 Y(4,0)^3 Y(4,1)^-1,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0) Y(2,1) Y(3,0)^-1 Y(3,1) Y(4,1)^-2]
```

phi(*i*)

Return the value of φ_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['D',4,3])
sage: m = M.module_generators[0].f(1)
sage: [m.phi(i) for i in M.index_set()]
[1, -1, 1]

sage: M = crystals.infinity.NakajimaMonomials(['C',4,1])
sage: m = M.module_generators[0].f_string([4,2,3])
sage: [m.phi(i) for i in M.index_set()]
[0, 1, -1, 2, -1]
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials(['D',Integer(4),Integer(3)])
```

(continues on next page)

(continued from previous page)

```
>>> m = M.module_generators[Integer(0)].f(Integer(1))
>>> [m.phi(i) for i in M.index_set()]
[1, -1, 1]

>>> M = crystals.infinity.NakajimaMonomials(['C',Integer(4),Integer(1)])
>>> m = M.module_generators[Integer(0)].f_string([Integer(4),Integer(2),
↪Integer(3)])
>>> [m.phi(i) for i in M.index_set()]
[0, 1, -1, 2, -1]
```

weight()

Return the weight of `self` as an element of the weight lattice.

EXAMPLES:

```
sage: C = crystals.infinity.NakajimaMonomials(['A',1,1])
sage: v = C.highest_weight_vector()
sage: v.f(1).weight() + v.f(0).weight()
-delta

sage: M = crystals.infinity.NakajimaMonomials(['A',4,2])
sage: m = M.highest_weight_vector().f_string([1,2,0,1])
sage: m.weight()
2*Lambda[0] - Lambda[1] - delta
```

```
>>> from sage.all import *
>>> C = crystals.infinity.NakajimaMonomials(['A',Integer(1),Integer(1)])
>>> v = C.highest_weight_vector()
>>> v.f(Integer(1)).weight() + v.f(Integer(0)).weight()
-delta

>>> M = crystals.infinity.NakajimaMonomials(['A',Integer(4),Integer(2)])
>>> m = M.highest_weight_vector().f_string([Integer(1),Integer(2),Integer(0),
↪Integer(1)])
>>> m.weight()
2*Lambda[0] - Lambda[1] - delta
```

weight_in_root_lattice()

Return the weight of `self` as an element of the root lattice.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['F',4])
sage: m = M.module_generators[0].f_string([3,3,1,2,4])
sage: m.weight_in_root_lattice()
-alpha[1] - alpha[2] - 2*alpha[3] - alpha[4]

sage: M = crystals.infinity.NakajimaMonomials(['B',3,1])
sage: mg = M.module_generators[0]
sage: m = mg.f_string([1,3,2,0,1,2,3,0,0,1])
sage: m.weight_in_root_lattice()
-3*alpha[0] - 3*alpha[1] - 2*alpha[2] - 2*alpha[3]
```

(continues on next page)

(continued from previous page)

```
sage: M = crystals.infinity.NakajimaMonomials(['C',3,1])
sage: m = M.module_generators[0].f_string([3,0,1,2,0])
sage: m.weight_in_root_lattice()
-2*alpha[0] - alpha[1] - alpha[2] - alpha[3]
```

```
>>> from sage.all import *
>>> M = crystals.infinity.NakajimaMonomials(['F',Integer(4)])
>>> m = M.module_generators[Integer(0)].f_string([Integer(3),Integer(3),
↪Integer(1),Integer(2),Integer(4)])
>>> m.weight_in_root_lattice()
-alpha[1] - alpha[2] - 2*alpha[3] - alpha[4]

>>> M = crystals.infinity.NakajimaMonomials(['B',Integer(3),Integer(1)])
>>> mg = M.module_generators[Integer(0)]
>>> m = mg.f_string([Integer(1),Integer(3),Integer(2),Integer(0),Integer(1),
↪Integer(2),Integer(3),Integer(0),Integer(0),Integer(1)])
>>> m.weight_in_root_lattice()
-3*alpha[0] - 3*alpha[1] - 2*alpha[2] - 2*alpha[3]

>>> M = crystals.infinity.NakajimaMonomials(['C',Integer(3),Integer(1)])
>>> m = M.module_generators[Integer(0)].f_string([Integer(3),Integer(0),
↪Integer(1),Integer(2),Integer(0)])
>>> m.weight_in_root_lattice()
-2*alpha[0] - alpha[1] - alpha[2] - alpha[3]
```

5.1.60 Crystal of Bernstein-Zelevinsky Multisegments

class sage.combinat.crystals.multisegments.**InfinityCrystalOfMultisegments**(*n*)

Bases: Parent, UniqueRepresentation

The type $A_n^{(1)}$ crystal $B(\infty)$ realized using Bernstein-Zelevinsky (BZ) multisegments.

Using (a modified version of the) notation from [JL2009], for $\ell \in \mathbf{Z}_{>0}$ and $i \in \mathbf{Z}/(n+1)\mathbf{Z}$, a segment of length ℓ and head i is the sequence of consecutive residues $[i, i+1, \dots, i+\ell-1]$. The notation for a segment of length ℓ and head i is simplified to $[i; \ell)$. Similarly, a segment of length ℓ and tail i is the sequence of consecutive residues $[i-\ell+1, \dots, i-1, i]$. The latter is denoted simply by $(\ell; i]$. Finally, a multisegment is a formal linear combination of segments, usually written in the form

$$\psi = \sum_{\substack{i \in \mathbf{Z}/(n+1)\mathbf{Z} \\ \ell \in \mathbf{Z}_{>0}}} m_{(\ell;i]} (\ell; i].$$

Such a multisegment is called aperiodic if, for every $\ell > 0$, there exists some $i \in \mathbf{Z}/(n+1)\mathbf{Z}$ such that $(\ell; i]$ does not appear in ψ. Denote the set of all periodic multisegments, together with the empty multisegment $\varnothing$, by Ψ. We define a crystal structure on multisegments as follows. Set $S_{\ell,i} = \sum_{k \geq \ell} (m_{(k;i-1]} - m_{(k;i]})$ and let ℓ_f be the minimal ℓ that attains the value $\min_{\ell > 0} S_{\ell,i}$. Then we have

$$f_i \psi = \begin{cases} \psi + (1; i] & \text{if } \ell_f = 1, \\ \psi + (\ell_f; i] - (\ell_f - 1; i-1] & \text{if } \ell_f > 1. \end{cases}$$

Similarly, let ℓ_e be the maximal ℓ that attains the value $\min_{\ell > 0} S_{\ell,i}$. Then we have

$$e_i \psi = \begin{cases} 0 & \text{if } \min_{\ell > 0} S_{\ell,i} = 0, \\ \psi + (1; i] & \text{if } \ell_e = 1, \\ \psi - (\ell_e; i] + (\ell_e - 1; i-1] & \text{if } \ell_e > 1. \end{cases}$$

Alternatively, the crystal operators may be defined using a signature rule, as detailed in Section 4 of [JL2009] (following [AJL2011]). For $\psi \in \Psi$ and $i \in \mathbf{Z}/(n+1)\mathbf{Z}$, encode all segments in ψ with tail i by the symbol R and all segments in ψ with tail $i-1$ by A. For $\ell > 0$, set $w_{i,\ell} = R^{m_{(\ell;i]}} A^{m_{(\ell;i-1]}}$ and $w_i = \prod_{\ell \geq 1} w_{i,\ell}$. By successively canceling out as many RA factors as possible, set $\widetilde{w}_i = A^{a_i(\psi)} R^{r_i(\psi)}$. If $a_i(\psi) > 0$, denote by $\ell_f > 0$ the length of the rightmost segment A in $\widetilde{w}_i$. If $a_i(\psi) = 0$, set $\ell_f = 0$. Then

$$f_i \psi = \begin{cases} \psi + (1; i] & \text{if } a_i(\psi) = 0, \\ \psi + (\ell_f; i] - (\ell_f - 1; i-1] & \text{if } a_i(\psi) > 0. \end{cases}$$

The rule for computing $e_i \psi$ is similar.

INPUT:

- `n` – for type $A_n^{(1)}$

EXAMPLES:

```
sage: B = crystals.infinity.Multisegments(2)
sage: x = B([(8,1),(6,0),(5,1),(5,0),(4,0),(4,1),(4,1),(3,0),(3,0),(3,1),(3,1),(1,
↪0),(1,2),(1,2)]); x
(8; 1] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (1; 0] + 2 * (1; 2]
sage: x.f(1)
(8; 1] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (2; 1] + 2 * (1; 2]
sage: x.f(1).f(1)
(8; 1] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (2; 1] + (1; 1] + 2 * (1; 2]
sage: x.e(1)
(7; 0] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (1; 0] + 2 * (1; 2]
sage: x.e(1).e(1)
sage: x.f(0)
(8; 1] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (2; 0] + (1; 0] + (1; 2]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(2))
>>> x = B([(Integer(8),Integer(1)),(Integer(6),Integer(0)),(Integer(5),
↪Integer(1)),(Integer(5),Integer(0)),(Integer(4),Integer(0)),(Integer(4),
↪Integer(1)),(Integer(4),Integer(1)),(Integer(3),Integer(0)),(Integer(3),
↪Integer(0)),(Integer(3),Integer(1)),(Integer(3),Integer(1)),(Integer(1),
↪Integer(0)),(Integer(1),Integer(2)),(Integer(1),Integer(2))]); x
(8; 1] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (1; 0] + 2 * (1; 2]
>>> x.f(Integer(1))
(8; 1] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (2; 1] + 2 * (1; 2]
>>> x.f(Integer(1)).f(Integer(1))
(8; 1] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (2; 1] + (1; 1] + 2 * (1; 2]
>>> x.e(Integer(1))
(7; 0] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (1; 0] + 2 * (1; 2]
>>> x.e(Integer(1)).e(Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> x.f(Integer(0))
(8; 1] + (6; 0] + (5; 0] + (5; 1] + (4; 0] + 2 * (4; 1]
 + 2 * (3; 0] + 2 * (3; 1] + (2; 0] + (1; 0] + (1; 2]
```

We check an $\widehat{\mathfrak{sl}}_2$ example against the generalized Young walls:

```
sage: B = crystals.infinity.Multisegments(1)
sage: G = B.subcrystal(max_depth=4).digraph()
sage: C = crystals.infinity.GeneralizedYoungWalls(1)
sage: GC = C.subcrystal(max_depth=4).digraph()
sage: G.is_isomorphic(GC, edge_labels=True)
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(1))
>>> G = B.subcrystal(max_depth=Integer(4)).digraph()
>>> C = crystals.infinity.GeneralizedYoungWalls(Integer(1))
>>> GC = C.subcrystal(max_depth=Integer(4)).digraph()
>>> G.is_isomorphic(GC, edge_labels=True)
True
```

REFERENCES:

- [AJL2011]
- [JL2009]
- [LTV1999]

class Element (*parent*, *value*)

Bases: `ElementWrapper`

An element in a BZ multisegments crystal.

e (*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Multisegments(2)
sage: b = B([(4,2), (3,0), (3,1), (1,1), (1,0)])
sage: b.e(0)
(4; 2] + (3; 0] + (3; 1] + (1; 1]
sage: b.e(1)
sage: b.e(2)
(3; 0] + 2 * (3; 1] + (1; 0] + (1; 1]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(2))
>>> b = B([(Integer(4),Integer(2)), (Integer(3),Integer(0)), (Integer(3),
↪Integer(1)), (Integer(1),Integer(1)), (Integer(1),Integer(0))])
>>> b.e(Integer(0))
(4; 2] + (3; 0] + (3; 1] + (1; 1]
```

(continues on next page)

(continued from previous page)

```
>>> b.e(Integer(1))
>>> b.e(Integer(2))
(3; 0] + 2 * (3; 1] + (1; 0] + (1; 1]
```

epsilon(*i*)

Return ε_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Multisegments(2)
sage: b = B([(4,2), (3,0), (3,1), (1,1), (1,0)])
sage: b.epsilon(0)
1
sage: b.epsilon(1)
0
sage: b.epsilon(2)
1
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(2))
>>> b = B([(Integer(4),Integer(2)), (Integer(3),Integer(0)), (Integer(3),
↪Integer(1)), (Integer(1),Integer(1)), (Integer(1),Integer(0))])
>>> b.epsilon(Integer(0))
1
>>> b.epsilon(Integer(1))
0
>>> b.epsilon(Integer(2))
1
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Multisegments(2)
sage: b = B([(4,2), (3,0), (3,1), (1,1), (1,0)])
sage: b.f(0)
(4; 2] + (3; 0] + (3; 1] + 2 * (1; 0] + (1; 1]
sage: b.f(1)
(4; 2] + (3; 0] + (3; 1] + (1; 0] + 2 * (1; 1]
sage: b.f(2)
2 * (4; 2] + (3; 0] + (1; 0] + (1; 1]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(2))
>>> b = B([(Integer(4),Integer(2)), (Integer(3),Integer(0)), (Integer(3),
↪Integer(1)), (Integer(1),Integer(1)), (Integer(1),Integer(0))])
>>> b.f(Integer(0))
```

(continues on next page)

(continued from previous page)

```
(4; 2] + (3; 0] + (3; 1] + 2 * (1; 0] + (1; 1]
>>> b.f(Integer(1))
(4; 2] + (3; 0] + (3; 1] + (1; 0] + 2 * (1; 1]
>>> b.f(Integer(2))
2 * (4; 2] + (3; 0] + (1; 0] + (1; 1]
```

phi(*i*)

Return φ_i of `self`.

Let $\psi \in \Psi$. Define $\varphi_i(\psi) := \varepsilon_i(\psi) + \langle h_i, \mathrm{wt}(\psi)\rangle$, where h_i is the i-th simple coroot and $\mathrm{wt}(\psi)$ is the `weight()` of ψ.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Multisegments(2)
sage: b = B([(4,2), (3,0), (3,1), (1,1), (1,0)])
sage: b.phi(0)
1
sage: b.phi(1)
0
sage: mg = B.highest_weight_vector()
sage: mg.f(1).phi(0)
1
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(2))
>>> b = B([(Integer(4),Integer(2)), (Integer(3),Integer(0)), (Integer(3),
↪Integer(1)), (Integer(1),Integer(1)), (Integer(1),Integer(0))])
>>> b.phi(Integer(0))
1
>>> b.phi(Integer(1))
0
>>> mg = B.highest_weight_vector()
>>> mg.f(Integer(1)).phi(Integer(0))
1
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.Multisegments(2)
sage: b = B([(4,2), (3,0), (3,1), (1,1), (1,0)])
sage: b.weight()
-4*delta
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(2))
>>> b = B([(Integer(4),Integer(2)), (Integer(3),Integer(0)), (Integer(3),
↪Integer(1)), (Integer(1),Integer(1)), (Integer(1),Integer(0))])
```

(continues on next page)

(continued from previous page)

```
>>> b.weight()
-4*delta
```

highest_weight_vector()

Return the highest weight vector of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.Multisegments(2)
sage: B.highest_weight_vector()
0
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(2))
>>> B.highest_weight_vector()
0
```

weight_lattice_realization()

Return a realization of the weight lattice of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.Multisegments(2)
sage: B.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Multisegments(Integer(2))
>>> B.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

5.1.61 Crystal Of Mirković-Vilonen (MV) Polytopes

AUTHORS:

- Dinakar Muthiah, Travis Scrimshaw (2015-05-11): initial version

class sage.combinat.crystals.mv_polytopes.**MVPolytope**(*parent*, *lusztig_datum*, *long_word=None*)

Bases: *PBWCrystalElement*

A Mirković-Vilonen (MV) polytope.

EXAMPLES:

We can create an animation showing how the MV polytope changes under a string of crystal operators:

```
sage: MV = crystals.infinity.MVPolytopes(['C', 2])
sage: u = MV.highest_weight_vector()
sage: L = RootSystem(['C',2,1]).ambient_space()
sage: s = [1,2,1,2,2,2,1,1,1,1,2,1,2,2,1,2]
sage: BB = [[-9, 2], [-10, 2]]
sage: p = L.plot(reflection_hyperplanes=False, bounding_box=BB)  # long time
sage: frames = [p + L.plot_mv_polytope(u.f_string(s[:i]),  # long time
....:                                  circle_size=0.1,
```

(continues on next page)

(continued from previous page)

```
....:                                       wireframe='green',
....:                                       fill='purple',
....:                                       bounding_box=BB)
....:             for i in range(len(s))]
sage: for f in frames:  # long time
....:     f.axes(False)
sage: animate(frames).show(delay=60) # optional -- ImageMagick # long time
```

```
>>> from sage.all import *
>>> MV = crystals.infinity.MVPolytopes(['C', Integer(2)])
>>> u = MV.highest_weight_vector()
>>> L = RootSystem(['C',Integer(2),Integer(1)]).ambient_space()
>>> s = [Integer(1),Integer(2),Integer(1),Integer(2),Integer(2),Integer(2),
↪Integer(1),Integer(1),Integer(1),Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(2),Integer(1),Integer(2)]
>>> BB = [[-Integer(9), Integer(2)], [-Integer(10), Integer(2)]]
>>> p = L.plot(reflection_hyperplanes=False, bounding_box=BB)  # long time
>>> frames = [p + L.plot_mv_polytope(u.f_string(s[:i]),  # long time
...                                   circle_size=RealNumber('0.1'),
...                                   wireframe='green',
...                                   fill='purple',
...                                   bounding_box=BB)
...             for i in range(len(s))]
>>> for f in frames:  # long time
...     f.axes(False)
>>> animate(frames).show(delay=Integer(60)) # optional -- ImageMagick # long time
```

plot(*P=None*, ***options*)

Plot `self`.

INPUT:

- `P` – (optional) a space to realize the polytope; default is the weight lattice realization of the crystal

See also

plot_mv_polytope()

EXAMPLES:

```
sage: MV = crystals.infinity.MVPolytopes(['C', 2])
sage: b = MV.highest_weight_vector().f_string([1,2,1,2,2,2,1,1,1,1,2,1])
sage: b.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
```

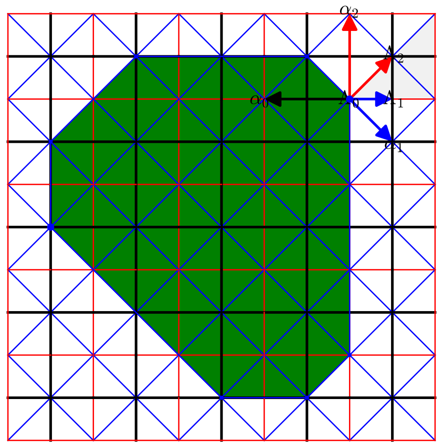
```
>>> from sage.all import *
>>> MV = crystals.infinity.MVPolytopes(['C', Integer(2)])
>>> b = MV.highest_weight_vector().f_string([Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(2),Integer(2),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(2),Integer(1)])
>>> b.plot()                                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
```

Here is the above example placed inside the ambient space of type C_2:



polytope (*P=None*)

Return a polytope of `self`.

INPUT:

- `P` – (optional) a space to realize the polytope; default is the weight lattice realization of the crystal

EXAMPLES:

```
sage: MV = crystals.infinity.MVPolytopes(['C', 3])
sage: b = MV.module_generators[0].f_string([3,2,3,2,1])
sage: P = b.polytope(); P
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 6 vertices
sage: P.vertices()
(A vertex at (0, 0, 0),
 A vertex at (0, 1, -1),
 A vertex at (0, 1, 1),
 A vertex at (1, -1, 0),
 A vertex at (1, 1, -2),
 A vertex at (1, 1, 2))
```

```
>>> from sage.all import *
>>> MV = crystals.infinity.MVPolytopes(['C', Integer(3)])
>>> b = MV.module_generators[Integer(0)].f_string([Integer(3),Integer(2),
↪Integer(3),Integer(2),Integer(1)])
>>> P = b.polytope(); P
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 6 vertices
>>> P.vertices()
(A vertex at (0, 0, 0),
 A vertex at (0, 1, -1),
 A vertex at (0, 1, 1),
 A vertex at (1, -1, 0),
 A vertex at (1, 1, -2),
 A vertex at (1, 1, 2))
```

class sage.combinat.crystals.mv_polytopes.**MVPolytopes**(*cartan_type*)

Bases: *PBWCrystal*

The crystal of Mirković-Vilonen (MV) polytopes.

Let W denote the corresponding Weyl group and $P_{\mathbf{R}} = \mathbf{R} \otimes P$. Let $\Gamma = \{w\Lambda_i \mid w \in W, i \in I\}$. Consider $M = (M_\gamma \in \mathbf{Z})_{\gamma\in\Gamma}$ that satisfy the *tropical Plücker relations* (see Proposition 7.1 of [BZ01]). The *MV polytope* is defined as

$$P(M) = \{\alpha \in P_{\mathbf{R}} \mid \langle \alpha, \gamma \rangle \geq M_\gamma \text{ for all } \gamma \in \Gamma\}.$$

The vertices $\{\mu_w\}_{w\in W}$ are given by

$$\langle \mu_w, \gamma \rangle = M_\gamma$$

and are known as the GGMS datum of the MV polytope.

Each path from μ_e to μ_{w_0} corresponds to a reduced expression $\mathbf{i} = (i_1, \ldots, i_m)$ for w_0 and the corresponding edge lengths $(n_k)_{k=1}^m$ from the Lusztig datum with respect to $\mathbf{i}$. Explicitly, we have

$$n_k = -M_{w_{k-1}\Lambda_{i_k}} - M_{w_k\Lambda_{i_k}} - \sum_{j\neq i} a_{ji} M_{w_k\Lambda_j},$$

$$\mu_{w_k} - \mu_{w_{k-1}} = n_k w_{k-1}\alpha_{i_k},$$

where $w_k = s_{i_1} \cdots s_{i_k}$ and (a_{ji}) is the Cartan matrix.

MV polytopes have a crystal structure that corresponds to the crystal structure, which is isomorphic to $\mathcal{B}(\infty)$ with $\mu_{w_0} = 0$, on *PBW data*. Specifically, we have $f_j P(M)$ as being the unique MV polytope given by shifting μ_e by $-\alpha_j$ and fixing the vertices μ_w when $s_j w < w$ (in Bruhat order) and the weight is given by μ_e. Furthermore, the $*$-involution is given by negating $P(M)$.

INPUT:

- cartan_type – a Cartan type

EXAMPLES:

```
sage: MV = crystals.infinity.MVPolytopes(['B', 3])
sage: hw = MV.highest_weight_vector()
sage: x = hw.f_string([1,2,2,3,3,1,3,3,2,3,2,1,3,1,2,3,1,2,1,3,2]); x
MV polytope with Lusztig datum (1, 1, 1, 3, 1, 0, 0, 1, 1)
```

```
>>> from sage.all import *
>>> MV = crystals.infinity.MVPolytopes(['B', Integer(3)])
>>> hw = MV.highest_weight_vector()
>>> x = hw.f_string([Integer(1),Integer(2),Integer(2),Integer(3),Integer(3),
↪Integer(1),Integer(3),Integer(3),Integer(2),Integer(3),Integer(2),Integer(1),
↪Integer(3),Integer(1),Integer(2),Integer(3),Integer(1),Integer(2),Integer(1),
↪Integer(3),Integer(2)]); x
MV polytope with Lusztig datum (1, 1, 1, 3, 1, 0, 0, 1, 1)
```

Elements are expressed in terms of Lusztig datum for a fixed reduced expression of w_0:

```
sage: MV.default_long_word()
[1, 3, 2, 3, 1, 2, 3, 1, 2]
sage: MV.set_default_long_word([2,1,3,2,1,3,2,3,1])
sage: x
MV polytope with Lusztig datum (3, 1, 1, 0, 1, 0, 1, 3, 4)
sage: MV.set_default_long_word([1, 3, 2, 3, 1, 2, 3, 1, 2])
```

```
>>> from sage.all import *
>>> MV.default_long_word()
[1, 3, 2, 3, 1, 2, 3, 1, 2]
>>> MV.set_default_long_word([Integer(2),Integer(1),Integer(3),Integer(2),
↪Integer(1),Integer(3),Integer(2),Integer(3),Integer(1)])
>>> x
MV polytope with Lusztig datum (3, 1, 1, 0, 1, 0, 1, 3, 4)
>>> MV.set_default_long_word([Integer(1), Integer(3), Integer(2), Integer(3),␣
↪Integer(1), Integer(2), Integer(3), Integer(1), Integer(2)])
```

We can construct elements by giving it Lusztig data (with respect to the default long word reduced expression):

```
sage: MV([1,1,1,3,1,0,0,1,1])
MV polytope with Lusztig datum (1, 1, 1, 3, 1, 0, 0, 1, 1)
```

```
>>> from sage.all import *
>>> MV([Integer(1),Integer(1),Integer(1),Integer(3),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1)])
MV polytope with Lusztig datum (1, 1, 1, 3, 1, 0, 0, 1, 1)
```

We can also construct elements by passing in a reduced expression for a long word:

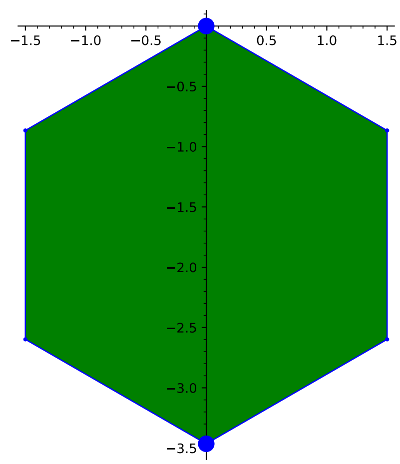
```
sage: x = MV([1,1,1,3,1,0,0,1,1], [3,2,1,3,2,3,2,1,2]); x
MV polytope with Lusztig datum (1, 1, 1, 0, 1, 0, 5, 1, 1)
sage: x.to_highest_weight()[1]
[1, 2, 2, 2, 2, 2, 1, 3, 3, 3, 3, 2, 3, 2, 3, 3, 2, 3, 3, 2, 1, 3]
```

```
>>> from sage.all import *
>>> x = MV([Integer(1),Integer(1),Integer(1),Integer(3),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1)], [Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3),Integer(2),Integer(1),Integer(2)]); x
MV polytope with Lusztig datum (1, 1, 1, 0, 1, 0, 5, 1, 1)
>>> x.to_highest_weight()[Integer(1)]
[1, 2, 2, 2, 2, 2, 1, 3, 3, 3, 3, 2, 3, 2, 3, 3, 2, 3, 3, 2, 1, 3]
```

The highest weight crystal $B(\lambda) \subseteq B(\infty)$ is characterized by the MV polytopes that sit inside of $W\lambda$ (translating $\mu_{w_0} \mapsto \lambda$):

```
sage: MV = crystals.infinity.MVPolytopes(['A',2])
sage: La = MV.weight_lattice_realization().fundamental_weights()
sage: R = crystals.elementary.R(La[1]+La[2])
sage: T = tensor([R, MV])
sage: x = T(R.module_generators[0], MV.highest_weight_vector())
sage: lw = x.to_lowest_weight()[0]; lw
[(2, 1, 0), MV polytope with Lusztig datum (1, 1, 1)]
sage: lw[1].polytope().vertices()
(A vertex at (0, 0, 0),
 A vertex at (0, 1, -1),
 A vertex at (1, -1, 0),
 A vertex at (1, 1, -2),
 A vertex at (2, -1, -1),
 A vertex at (2, 0, -2))
```

```
>>> from sage.all import *
>>> MV = crystals.infinity.MVPolytopes(['A',Integer(2)])
>>> La = MV.weight_lattice_realization().fundamental_weights()
>>> R = crystals.elementary.R(La[Integer(1)]+La[Integer(2)])
>>> T = tensor([R, MV])
>>> x = T(R.module_generators[Integer(0)], MV.highest_weight_vector())
>>> lw = x.to_lowest_weight()[Integer(0)]; lw
[(2, 1, 0), MV polytope with Lusztig datum (1, 1, 1)]
>>> lw[Integer(1)].polytope().vertices()
(A vertex at (0, 0, 0),
 A vertex at (0, 1, -1),
 A vertex at (1, -1, 0),
 A vertex at (1, 1, -2),
 A vertex at (2, -1, -1),
 A vertex at (2, 0, -2))
```



REFERENCES:

- [Kam2007]
- [Kam2010]

Element

alias of *MVPolytope*

latex_options()

Return the latex options of `self`.

EXAMPLES:

```
sage: MV = crystals.infinity.MVPolytopes(['F', 4])
sage: MV.latex_options()
{'P': Ambient space of the Root system of type ['F', 4],
 'circle_size': 0.1,
 'mark_endpoints': True,
 'projection': True}
```

```
>>> from sage.all import *
>>> MV = crystals.infinity.MVPolytopes(['F', Integer(4)])
```

(continues on next page)

(continued from previous page)

```
>>> MV.latex_options()
{'P': Ambient space of the Root system of type ['F', 4],
 'circle_size': 0.1,
 'mark_endpoints': True,
 'projection': True}
```

set_latex_options(***kwds*)

Set the latex options for the elements of `self`.

INPUT:

- `projection` – the projection; set to `True` to use the default projection of the specified weight lattice realization (initial: `True`)
- `P` – the weight lattice realization to use (initial: the weight lattice realization of `self`)
- `mark_endpoints` – whether to mark the endpoints (initial: `True`)
- `circle_size` – the size of the endpoint circles (initial: 0.1)

EXAMPLES:

```
sage: MV = crystals.infinity.MVPolytopes(['C', 2])
sage: P = RootSystem(['C', 2]).weight_lattice()
sage: b = MV.highest_weight_vector().f_string([1,2,1,2])
sage: latex(b)
\begin{tikzpicture}
\draw (0, 0) -- (0, -2) -- (-1, -3) -- (-1, -3) -- (-2, -2);
\draw (0, 0) -- (-1, 1) -- (-1, 1) -- (-2, 0) -- (-2, -2);
\draw[fill=black] (0, 0) circle (0.1);
\draw[fill=black] (-2, -2) circle (0.1);
\end{tikzpicture}
sage: MV.set_latex_options(P=P, circle_size=float(0.2))
sage: latex(b)
\begin{tikzpicture}
\draw (0, 0) -- (2, -2) -- (2, -3) -- (2, -3) -- (0, -2);
\draw (0, 0) -- (-2, 1) -- (-2, 1) -- (-2, 0) -- (0, -2);
\draw[fill=black] (0, 0) circle (0.2);
\draw[fill=black] (0, -2) circle (0.2);
\end{tikzpicture}
sage: MV.set_latex_options(mark_endpoints=False)
sage: latex(b)
\begin{tikzpicture}
\draw (0, 0) -- (2, -2) -- (2, -3) -- (2, -3) -- (0, -2);
\draw (0, 0) -- (-2, 1) -- (-2, 1) -- (-2, 0) -- (0, -2);
\end{tikzpicture}
sage: MV.set_latex_options(P=MV.weight_lattice_realization(),
....:                      circle_size=0.2,
....:                      mark_endpoints=True)
```

```
>>> from sage.all import *
>>> MV = crystals.infinity.MVPolytopes(['C', Integer(2)])
>>> P = RootSystem(['C', Integer(2)]).weight_lattice()
>>> b = MV.highest_weight_vector().f_string([Integer(1),Integer(2),Integer(1),
↪Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> latex(b)
\begin{tikzpicture}
\draw (0, 0) -- (0, -2) -- (-1, -3) -- (-1, -3) -- (-2, -2);
\draw (0, 0) -- (-1, 1) -- (-1, 1) -- (-2, 0) -- (-2, -2);
\draw[fill=black] (0, 0) circle (0.1);
\draw[fill=black] (-2, -2) circle (0.1);
\end{tikzpicture}
>>> MV.set_latex_options(P=P, circle_size=float(RealNumber('0.2')))
>>> latex(b)
\begin{tikzpicture}
\draw (0, 0) -- (2, -2) -- (2, -3) -- (2, -3) -- (0, -2);
\draw (0, 0) -- (-2, 1) -- (-2, 1) -- (-2, 0) -- (0, -2);
\draw[fill=black] (0, 0) circle (0.2);
\draw[fill=black] (0, -2) circle (0.2);
\end{tikzpicture}
>>> MV.set_latex_options(mark_endpoints=False)
>>> latex(b)
\begin{tikzpicture}
\draw (0, 0) -- (2, -2) -- (2, -3) -- (2, -3) -- (0, -2);
\draw (0, 0) -- (-2, 1) -- (-2, 1) -- (-2, 0) -- (0, -2);
\end{tikzpicture}
>>> MV.set_latex_options(P=MV.weight_lattice_realization(),
...                      circle_size=RealNumber('0.2'),
...                      mark_endpoints=True)
```

5.1.62 $\mathcal{B}(\infty)$ Crystal Of PBW Monomials

AUTHORS:

- Dinakar Muthiah (2015-05-11): initial version

See also

For information on PBW datum, see *PBW Data*.

class sage.combinat.crystals.pbw_crystal.**PBWCrystal**(*cartan_type*)

Bases: `Parent`, `UniqueRepresentation`

Crystal of $\mathcal{B}(\infty)$ given by PBW monomials.

A model of the crystal $\mathcal{B}(\infty)$ whose elements are PBW datum up to equivalence by the tropical Plücker relations. The crystal structure on Lusztig data $x = (x_1, \ldots, x_m)$ for the reduced word $s_{i_1} \cdots s_{i_m} = w_0$ is given as follows. Suppose $i_1 = j$, then $f_j x = (x_1 + 1, x_2, \ldots, x_m)$. If $i_1 \neq j$, then we use the tropical Plücker relations to change the reduced expression such that $i'_1 = j$ and then we change back to the original word.

EXAMPLES:

```
sage: PBW = crystals.infinity.PBW(['B', 3])
sage: hw = PBW.highest_weight_vector()
sage: x = hw.f_string([1,2,2,3,3,1,3,3,2,3,2,1,3,1,2,3,1,2,1,3,2]); x
PBW monomial with Lusztig datum (1, 1, 1, 3, 1, 0, 0, 1, 1)
```

```
>>> from sage.all import *
>>> PBW = crystals.infinity.PBW(['B', Integer(3)])
>>> hw = PBW.highest_weight_vector()
>>> x = hw.f_string([Integer(1),Integer(2),Integer(2),Integer(3),Integer(3),
↪Integer(1),Integer(3),Integer(3),Integer(2),Integer(3),Integer(2),Integer(1),
↪Integer(3),Integer(1),Integer(2),Integer(3),Integer(1),Integer(2),Integer(1),
↪Integer(3),Integer(2)]); x
PBW monomial with Lusztig datum (1, 1, 1, 3, 1, 0, 0, 1, 1)
```

Elements are expressed in terms of Lusztig datum for a fixed reduced expression of w_0:

```
sage: PBW.default_long_word()
[1, 3, 2, 3, 1, 2, 3, 1, 2]
sage: PBW.set_default_long_word([2,1,3,2,1,3,2,3,1])
sage: x
PBW monomial with Lusztig datum (3, 1, 1, 0, 1, 0, 1, 3, 4)
sage: PBW.set_default_long_word([1, 3, 2, 3, 1, 2, 3, 1, 2])
```

```
>>> from sage.all import *
>>> PBW.default_long_word()
[1, 3, 2, 3, 1, 2, 3, 1, 2]
>>> PBW.set_default_long_word([Integer(2),Integer(1),Integer(3),Integer(2),
↪Integer(1),Integer(3),Integer(2),Integer(3),Integer(1)])
>>> x
PBW monomial with Lusztig datum (3, 1, 1, 0, 1, 0, 1, 3, 4)
>>> PBW.set_default_long_word([Integer(1), Integer(3), Integer(2), Integer(3),␣
↪Integer(1), Integer(2), Integer(3), Integer(1), Integer(2)])
```

We can construct elements by giving it Lusztig data (with respect to the default long word):

```
sage: PBW([1,1,1,3,1,0,0,1,1])
PBW monomial with Lusztig datum (1, 1, 1, 3, 1, 0, 0, 1, 1)
```

```
>>> from sage.all import *
>>> PBW([Integer(1),Integer(1),Integer(1),Integer(3),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1)])
PBW monomial with Lusztig datum (1, 1, 1, 3, 1, 0, 0, 1, 1)
```

We can also construct elements by passing in a reduced expression for a long word:

```
sage: x = PBW([1,1,1,3,1,0,0,1,1], [3,2,1,3,2,3,2,1,2]); x
PBW monomial with Lusztig datum (1, 1, 1, 0, 1, 0, 5, 1, 1)
sage: x.to_highest_weight()[1]
[1, 2, 2, 2, 2, 2, 1, 3, 3, 3, 3, 2, 3, 2, 3, 3, 2, 3, 3, 2, 1, 3]
```

```
>>> from sage.all import *
>>> x = PBW([Integer(1),Integer(1),Integer(1),Integer(3),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1)], [Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3),Integer(2),Integer(1),Integer(2)]); x
PBW monomial with Lusztig datum (1, 1, 1, 0, 1, 0, 5, 1, 1)
>>> x.to_highest_weight()[Integer(1)]
[1, 2, 2, 2, 2, 2, 1, 3, 3, 3, 3, 2, 3, 2, 3, 3, 2, 3, 3, 2, 1, 3]
```

Element

alias of *PBWCrystalElement*

default_long_word()

Return the default long word used to express elements of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PBW(['E', 6])
sage: B.default_long_word()
[1, 3, 4, 5, 6, 2, 4, 5, 3, 4, 1, 3, 2, 4, 5, 6, 2, 4,
 5, 3, 4, 1, 3, 2, 4, 5, 3, 4, 1, 3, 2, 4, 1, 3, 2, 1]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PBW(['E', Integer(6)])
>>> B.default_long_word()
[1, 3, 4, 5, 6, 2, 4, 5, 3, 4, 1, 3, 2, 4, 5, 6, 2, 4,
 5, 3, 4, 1, 3, 2, 4, 5, 3, 4, 1, 3, 2, 4, 1, 3, 2, 1]
```

set_default_long_word(*word*)

Set the default long word used to express elements of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PBW(['C', 3])
sage: B.default_long_word()
[1, 3, 2, 3, 1, 2, 3, 1, 2]
sage: x = B.highest_weight_vector().f_string([2,1,3,2,3,1,2,3,3,1])
sage: x
PBW monomial with Lusztig datum (1, 2, 2, 0, 0, 0, 0, 0, 1)
sage: B.set_default_long_word([2,1,3,2,1,3,2,3,1])
sage: B.default_long_word()
[2, 1, 3, 2, 1, 3, 2, 3, 1]
sage: x
PBW monomial with Lusztig datum (2, 0, 0, 0, 0, 0, 1, 3, 2)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PBW(['C', Integer(3)])
>>> B.default_long_word()
[1, 3, 2, 3, 1, 2, 3, 1, 2]
>>> x = B.highest_weight_vector().f_string([Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3),Integer(1),Integer(2),Integer(3),Integer(3),
↪Integer(1)])
>>> x
PBW monomial with Lusztig datum (1, 2, 2, 0, 0, 0, 0, 0, 1)
>>> B.set_default_long_word([Integer(2),Integer(1),Integer(3),Integer(2),
↪Integer(1),Integer(3),Integer(2),Integer(3),Integer(1)])
>>> B.default_long_word()
[2, 1, 3, 2, 1, 3, 2, 3, 1]
>>> x
PBW monomial with Lusztig datum (2, 0, 0, 0, 0, 0, 1, 3, 2)
```

class sage.combinat.crystals.pbw_crystal.**PBWCrystalElement**(*parent*, *lusztig_datum*, *long_word=None*)

Bases: `Element`

A crystal element in the PBW model.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PBW(['B', 3])
sage: b = B.highest_weight_vector()
sage: c = b.f_string([2,1,3,2,1,3,2,2]); c
PBW monomial with Lusztig datum (0, 1, 0, 1, 0, 0, 0, 1, 2)
sage: c.e(2)
PBW monomial with Lusztig datum (0, 1, 0, 1, 0, 0, 0, 1, 1)
sage: c.e_string([2,2,1,3,2,1,3,2]) == b
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PBW(['B', Integer(3)])
>>> b = B.highest_weight_vector()
>>> c = b.f_string([Integer(2),Integer(1),Integer(3),Integer(2),Integer(1),
↪Integer(3),Integer(2),Integer(2)]); c
PBW monomial with Lusztig datum (0, 1, 0, 1, 0, 0, 0, 1, 2)
>>> c.e(Integer(2))
PBW monomial with Lusztig datum (0, 1, 0, 1, 0, 0, 0, 1, 1)
>>> c.e_string([Integer(2),Integer(2),Integer(1),Integer(3),Integer(2),
↪Integer(1),Integer(3),Integer(2)]) == b
True
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PBW(["A2"])
sage: s = B((3,0,0), (1,2,1))
sage: s.epsilon(1)
3
sage: s.epsilon(2)
0
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PBW(["A2"])
>>> s = B((Integer(3),Integer(0),Integer(0)), (Integer(1),Integer(2),
↪Integer(1)))
>>> s.epsilon(Integer(1))
3
>>> s.epsilon(Integer(2))
0
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PBW("D4")
sage: b = B.highest_weight_vector()
sage: c = b.f_string([1,2,3,1,2,3,4]); c
PBW monomial with Lusztig datum (0, 1, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0)
sage: c == b.f_string([1,2,4,1,2,3,3])
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PBW("D4")
>>> b = B.highest_weight_vector()
>>> c = b.f_string([Integer(1),Integer(2),Integer(3),Integer(1),Integer(2),
↪Integer(3),Integer(4)]); c
PBW monomial with Lusztig datum (0, 1, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0)
>>> c == b.f_string([Integer(1),Integer(2),Integer(4),Integer(1),Integer(2),
↪Integer(3),Integer(3)])
True
```

lusztig_datum(*word=None*)

Return the Lusztig datum of `self` with respect to the reduced expression of the long word `word`.

EXAMPLES:

```
sage: B = crystals.infinity.PBW(['A', 2])
sage: u = B.highest_weight_vector()
sage: b = u.f_string([2,1,2,2,2,2,1,1,2,1,2,1,2,1,2,2])
sage: b.lusztig_datum()
(6, 0, 10)
sage: b.lusztig_datum(word=[2,1,2])
(4, 6, 0)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PBW(['A', Integer(2)])
>>> u = B.highest_weight_vector()
>>> b = u.f_string([Integer(2),Integer(1),Integer(2),Integer(2),Integer(2),
↪Integer(2),Integer(1),Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1),Integer(2),Integer(1),Integer(2),Integer(2)])
>>> b.lusztig_datum()
(6, 0, 10)
>>> b.lusztig_datum(word=[Integer(2),Integer(1),Integer(2)])
(4, 6, 0)
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PBW(['A', 2])
sage: s = B((3,0,0), (1,2,1))
sage: s.phi(1)
-3
sage: s.phi(2)
3
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PBW(['A', Integer(2)])
>>> s = B((Integer(3),Integer(0),Integer(0)), (Integer(1),Integer(2),
→Integer(1)))
>>> s.phi(Integer(1))
-3
>>> s.phi(Integer(2))
3
```

star()

Return the starred crystal element corresponding to `self`.

Let b be an element of `self` with Lusztig datum $(b_1,\ldots,b_N)$ with respect to $w_0 = s_{i_1}\cdots s_{i_N}$. Then b^* is the element with Lusztig datum $(b_N,\ldots,b_1)$ with respect to $w_0 = s_{i_N^*}\cdots s_{i_1^*}$, where $i_j^* = \omega(i_j)$ with ω being the *automorphism* given by the action of w_0 on the simple roots.

EXAMPLES:

```
sage: P = crystals.infinity.PBW(['A', 2])
sage: P((1,2,3), (1,2,1)).star() == P((3,2,1), (2,1,2))
True

sage: B = crystals.infinity.PBW(['E', 6])
sage: b = B.highest_weight_vector()
sage: c = b.f_string([1,2,6,3,4,2,5,2,3,4,1,6])
sage: c == c.star().star()
True
```

```
>>> from sage.all import *
>>> P = crystals.infinity.PBW(['A', Integer(2)])
>>> P((Integer(1),Integer(2),Integer(3)), (Integer(1),Integer(2),Integer(1))).
→star() == P((Integer(3),Integer(2),Integer(1)), (Integer(2),Integer(1),
→Integer(2)))
True

>>> B = crystals.infinity.PBW(['E', Integer(6)])
>>> b = B.highest_weight_vector()
>>> c = b.f_string([Integer(1),Integer(2),Integer(6),Integer(3),Integer(4),
→Integer(2),Integer(5),Integer(2),Integer(3),Integer(4),Integer(1),
→Integer(6)])
>>> c == c.star().star()
True
```

weight()

Return weight of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PBW(['A', 2])
sage: s = B((2,2,2), (1,2,1))
sage: s.weight()
(-4, 0, 4)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PBW(['A', Integer(2)])
>>> s = B((Integer(2),Integer(2),Integer(2)), (Integer(1),Integer(2),
↪Integer(1)))
>>> s.weight()
(-4, 0, 4)
```

5.1.63 PBW Data

This contains helper classes and functions which encode PBW data in finite type.

AUTHORS:

- Dinakar Muthiah (2015-05): initial version
- Travis Scrimshaw (2016-06): simplified code and converted to Cython

class sage.combinat.crystals.pbw_datum.**PBWData**(*cartan_type*)

Bases: object

Helper class for the set of PBW data.

convert_to_new_long_word(*pbw_datum*, *new_long_word*)

Convert the PBW datum pbw_datum from its long word to new_long_word.

EXAMPLES:

```
sage: from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
sage: P = PBWData("A2")
sage: datum = PBWDatum(P, (1,2,1), (1,0,1))
sage: new_datum = P.convert_to_new_long_word(datum,(2,1,2))
sage: new_datum
PBW Datum element of type ['A', 2] with long word (2, 1, 2)
 and Lusztig datum (0, 1, 0)
sage: new_datum.long_word
(2, 1, 2)
sage: new_datum.lusztig_datum
(0, 1, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
>>> P = PBWData("A2")
>>> datum = PBWDatum(P, (Integer(1),Integer(2),Integer(1)), (Integer(1),
↪Integer(0),Integer(1)))
>>> new_datum = P.convert_to_new_long_word(datum,(Integer(2),Integer(1),
↪Integer(2)))
>>> new_datum
PBW Datum element of type ['A', 2] with long word (2, 1, 2)
 and Lusztig datum (0, 1, 0)
>>> new_datum.long_word
(2, 1, 2)
>>> new_datum.lusztig_datum
(0, 1, 0)
```

class sage.combinat.crystals.pbw_datum.**PBWDatum**(*parent*, *long_word*, *lusztig_datum*)

Bases: object

Helper class which represents a PBW datum.

convert_to_long_word_with_first_letter(*i*)

Return a new PBWDatum equivalent to `self` whose long word begins with `i`.

EXAMPLES:

```
sage: from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
sage: P = PBWData("A3")
sage: datum = PBWDatum(P, (1,2,1,3,2,1), (1,0,1,4,2,3))
sage: datum.convert_to_long_word_with_first_letter(1)
PBW Datum element of type ['A', 3] with long word (1, 2, 3, 1, 2, 1)
 and Lusztig datum (1, 0, 4, 1, 2, 3)
sage: datum.convert_to_long_word_with_first_letter(2)
PBW Datum element of type ['A', 3] with long word (2, 1, 2, 3, 2, 1)
 and Lusztig datum (0, 1, 0, 4, 2, 3)
sage: datum.convert_to_long_word_with_first_letter(3)
PBW Datum element of type ['A', 3] with long word (3, 1, 2, 3, 1, 2)
 and Lusztig datum (8, 1, 0, 4, 1, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
>>> P = PBWData("A3")
>>> datum = PBWDatum(P, (Integer(1),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(1)), (Integer(1),Integer(0),Integer(1),Integer(4),
↪Integer(2),Integer(3)))
>>> datum.convert_to_long_word_with_first_letter(Integer(1))
PBW Datum element of type ['A', 3] with long word (1, 2, 3, 1, 2, 1)
 and Lusztig datum (1, 0, 4, 1, 2, 3)
>>> datum.convert_to_long_word_with_first_letter(Integer(2))
PBW Datum element of type ['A', 3] with long word (2, 1, 2, 3, 2, 1)
 and Lusztig datum (0, 1, 0, 4, 2, 3)
>>> datum.convert_to_long_word_with_first_letter(Integer(3))
PBW Datum element of type ['A', 3] with long word (3, 1, 2, 3, 1, 2)
 and Lusztig datum (8, 1, 0, 4, 1, 2)
```

convert_to_new_long_word(*new_long_word*)

Return a new PBWDatum equivalent to `self` whose long word is `new_long_word`.

EXAMPLES:

```
sage: from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
sage: P = PBWData("A2")
sage: datum = PBWDatum(P, (1,2,1), (1,0,1))
sage: new_datum = datum.convert_to_new_long_word((2,1,2))
sage: new_datum.long_word
(2, 1, 2)
sage: new_datum.lusztig_datum
(0, 1, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
>>> P = PBWData("A2")
>>> datum = PBWDatum(P, (Integer(1),Integer(2),Integer(1)), (Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0),Integer(1)))
>>> new_datum = datum.convert_to_new_long_word((Integer(2),Integer(1),
↪Integer(2)))
>>> new_datum.long_word
(2, 1, 2)
>>> new_datum.lusztig_datum
(0, 1, 0)
```

is_equivalent_to(*other_pbw_datum*)

Return whether `self` is equivalent to `other_pbw_datum`. modulo the tropical Plücker relations.

EXAMPLES:

```
sage: from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
sage: P = PBWData("A2")
sage: L1 = PBWDatum(P, (1,2,1), (1,0,1))
sage: L2 = PBWDatum(P, (2,1,2), (0,1,0))
sage: L1.is_equivalent_to(L2)
True
sage: L1 == L2
False
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
>>> P = PBWData("A2")
>>> L1 = PBWDatum(P, (Integer(1),Integer(2),Integer(1)), (Integer(1),
↪Integer(0),Integer(1)))
>>> L2 = PBWDatum(P, (Integer(2),Integer(1),Integer(2)), (Integer(0),
↪Integer(1),Integer(0)))
>>> L1.is_equivalent_to(L2)
True
>>> L1 == L2
False
```

star()

Return the starred version of `self`, i.e., with reversed `long_word` and `lusztig_datum`

EXAMPLES:

```
sage: from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
sage: P = PBWData("A2")
sage: L1 = PBWDatum(P, (1,2,1), (1,2,3))
sage: L1.star() == PBWDatum(P, (2,1,2), (3,2,1))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
>>> P = PBWData("A2")
>>> L1 = PBWDatum(P, (Integer(1),Integer(2),Integer(1)), (Integer(1),
↪Integer(2),Integer(3)))
>>> L1.star() == PBWDatum(P, (Integer(2),Integer(1),Integer(2)), (Integer(3),
↪Integer(2),Integer(1)))
True
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
sage: P = PBWData("A2")
sage: L = PBWDatum(P, (1,2,1), (1,1,1))
sage: L.weight()
-2*alpha[1] - 2*alpha[2]
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.pbw_datum import PBWData, PBWDatum
>>> P = PBWData("A2")
>>> L = PBWDatum(P, (Integer(1),Integer(2),Integer(1)), (Integer(1),
↪Integer(1),Integer(1)))
>>> L.weight()
-2*alpha[1] - 2*alpha[2]
```

`sage.combinat.crystals.pbw_datum.`**compute_new_lusztig_datum**(*enhanced_braid_chain*, *initial_lusztig_datum*)

Return the Lusztig datum obtained by applying tropical Plücker relations along `enhanced_braid_chain` starting with `initial_lusztig_datum`.

EXAMPLES:

```
sage: from sage.combinat.root_system.braid_move_calculator import␣
↪BraidMoveCalculator
sage: from sage.combinat.crystals.pbw_datum import enhance_braid_move_chain
sage: from sage.combinat.crystals.pbw_datum import compute_new_lusztig_datum
sage: ct = CartanType(['A', 2])
sage: W = CoxeterGroup(ct)
sage: B = BraidMoveCalculator(W)
sage: chain = B.chain_of_reduced_words((1,2,1),(2,1,2))
sage: enhanced_braid_chain = enhance_braid_move_chain(chain, ct)
sage: compute_new_lusztig_datum(enhanced_braid_chain,(1,0,1))
(0, 1, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.braid_move_calculator import␣
↪BraidMoveCalculator
>>> from sage.combinat.crystals.pbw_datum import enhance_braid_move_chain
>>> from sage.combinat.crystals.pbw_datum import compute_new_lusztig_datum
>>> ct = CartanType(['A', Integer(2)])
>>> W = CoxeterGroup(ct)
>>> B = BraidMoveCalculator(W)
>>> chain = B.chain_of_reduced_words((Integer(1),Integer(2),Integer(1)),
↪(Integer(2),Integer(1),Integer(2)))
>>> enhanced_braid_chain = enhance_braid_move_chain(chain, ct)
>>> compute_new_lusztig_datum(enhanced_braid_chain,(Integer(1),Integer(0),
↪Integer(1)))
(0, 1, 0)
```

`sage.combinat.crystals.pbw_datum.`**enhance_braid_move_chain**(*braid_move_chain*, *cartan_type*)

Return a list of tuples that records the data of the long words in `braid_move_chain` plus the data of the intervals where the braid moves occur and the data of the off-diagonal entries of the 2×2 Cartan submatrices of each braid move.

INPUT:

- `braid_move_chain` – a chain of reduced words in the Weyl group of `cartan_type`
- `cartan_type` – a finite Cartan type

OUTPUT:

A list of 2-tuples `(interval_of_change, cartan_sub_matrix)` where

- `interval_of_change` is the (half-open) interval of indices where the braid move occurs; this is `None` for the first tuple
- `cartan_sub_matrix` is the off-diagonal entries of the 2×2 submatrix of the Cartan matrix corresponding to the braid move; this is `None` for the first tuple

For a matrix:

```
[2 a]
[b 2]
```

the `cartan_sub_matrix` is the pair `(a, b)`.

`sage.combinat.crystals.pbw_datum.`**`tropical_plucker_relation`**`(`*a*, *lusztig_datum*`)`

Apply the tropical Plücker relation of type `a` to `lusztig_datum`.

The relations are obtained by tropicalizing the relations in Proposition 7.1 of [BZ01].

INPUT:

- `a` – a pair `(x, y)` of the off-diagonal entries of a 2×2 Cartan matrix

EXAMPLES:

```
sage: from sage.combinat.crystals.pbw_datum import tropical_plucker_relation
sage: tropical_plucker_relation((0,0), (2,3))
(3, 2)
sage: tropical_plucker_relation((-1,-1), (1,2,3))
(4, 1, 2)
sage: tropical_plucker_relation((-1,-2), (1,2,3,4))
(8, 1, 2, 3)
sage: tropical_plucker_relation((-2,-1), (1,2,3,4))
(6, 1, 2, 3)
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.pbw_datum import tropical_plucker_relation
>>> tropical_plucker_relation((Integer(0),Integer(0)), (Integer(2),Integer(3)))
(3, 2)
>>> tropical_plucker_relation((-Integer(1),-Integer(1)), (Integer(1),Integer(2),
↪Integer(3)))
(4, 1, 2)
>>> tropical_plucker_relation((-Integer(1),-Integer(2)), (Integer(1),Integer(2),
↪Integer(3),Integer(4)))
(8, 1, 2, 3)
```

(continues on next page)

(continued from previous page)

```
>>> tropical_plucker_relation((-Integer(2),-Integer(1)), (Integer(1),Integer(2),
↪Integer(3),Integer(4)))
(6, 1, 2, 3)
```

5.1.64 Polyhedral Realization of $B(\infty)$

class sage.combinat.crystals.polyhedral_realization.**InfinityCrystalAsPolyhedralRealization**(*cartan_type*, *seq*)

Bases: *TensorProductOfCrystals*

The polyhedral realization of $B(\infty)$.

Note

Here we are using anti-Kashiwara notation and might differ from some of the literature.

Consider a Kac-Moody algebra $\mathfrak{g}$ of Cartan type X with index set I, and consider a finite sequence $J = (j_1, j_2, \ldots, j_m)$ whose support equals I. We extend this to an infinite sequence by taking $\bar{J} = J \cdot J \cdot J \cdots$, where $\cdot$ denotes concatenation of sequences. Let

$$B_J = B_{j_m} \otimes \cdots \otimes B_{j_2} \otimes B_{j_1},$$

where B_i is an *ElementaryCrystal*.

As given in Theorem 2.1.1 of [Ka1993], there exists a strict crystal embedding $\Psi_i \colon B(\infty) \to B_i \otimes B(\infty)$ defined by $u_\infty \mapsto b_i(0) \otimes u_\infty$, where $b_i(0) \in B_i$ and u_∞ is the (unique) highest weight element in $B(\infty)$. This is sometimes known as the *Kashiwara embedding* [NZ1997] (though, in [NZ1997], the target of this map is denoted by $\mathbf{Z}_J^\infty$). By iterating this embedding by taking $\Psi_J = \Psi_{j_n} \circ \Psi_{j_{n-1}} \circ \cdots \circ \Psi_{j_1}$, we obtain the following strict crystal embedding:

$$\Psi_J^n \colon B(\infty) \to B_J^{\otimes n} \otimes B(\infty).$$

We note there is a natural analog of Lemma 10.6.2 in [HK2002] that for any $b \in B(\infty)$, there exists a positive integer N such that

$$\Psi_J^N(b) = \left(\bigotimes_{k=1}^{N} b^{(k)} \right) \otimes u_\infty.$$

Therefore we can model elements $b \in B(\infty)$ by considering an infinite list of elements $b^{(k)} \in B_J$ and defining the crystal structure by:

$$\begin{aligned}
\mathrm{wt}(b) &= \sum_{k=1}^{N} \mathrm{wt}(b^{(k)}) \\
e_i(b) &= e_i\left(\left(\bigotimes_{k=1}^{N} b^{(k)} \right) \right) \otimes u_\infty, \\
f_i(b) &= f_i\left(\left(\bigotimes_{k=1}^{N} b^{(k)} \right) \right) \otimes u_\infty, \\
\varepsilon_i(b) &= \max_{e_i^k(b) \neq 0} k, \\
\varphi_i(b) &= \varepsilon_i(b) - \langle \mathrm{wt}(b), h_i^\vee \rangle.
\end{aligned}$$

To translate this into a finite list, we consider a finite sequence $b_1 \otimes \cdots \otimes b_N$ and if

$$f_i\left(b^{(1)} \otimes \cdots b^{(N-1)} \otimes b^{(N)}\right) = b^{(1)} \otimes \cdots \otimes b^{(N-1)} \otimes f_i\left(b^{(N)}\right),$$

then we take the image as $b^{(1)} \otimes \cdots \otimes f_i\left(b^{(N)}\right) \otimes b^{(N+1)}$. Similarly we remove $b^{(N)}$ if we have $b^{(N)} = \bigotimes_{k=1}^{m} b_{j_k}(0)$. Additionally if

$$e_i\left(b^{(1)} \otimes \cdots \otimes b^{(N-1)} \otimes b^{(N)}\right) = b^{(1)} \otimes \cdots \otimes b^{(N-1)} \otimes e_i\left(b^{(N)}\right),$$

then we consider this to be 0.

INPUT:

- `cartan_type` – a Cartan type
- `seq` – (default: `None`) a finite sequence whose support equals the index set of the Cartan type; if `None`, then this is the index set

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: mg = B.module_generators[0]; mg
[0, 0]
sage: mg.f_string([2,1,2,2])
[0, -3, -1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['A',Integer(2)])
>>> mg = B.module_generators[Integer(0)]; mg
[0, 0]
>>> mg.f_string([Integer(2),Integer(1),Integer(2),Integer(2)])
[0, -3, -1, 0, 0, 0]
```

An example of type B_2:

```
sage: B = crystals.infinity.PolyhedralRealization(['B',2])
sage: mg = B.module_generators[0]; mg
[0, 0]
sage: mg.f_string([2,1,2,2])
[0, -2, -1, -1, 0, 0]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['B',Integer(2)])
>>> mg = B.module_generators[Integer(0)]; mg
[0, 0]
>>> mg.f_string([Integer(2),Integer(1),Integer(2),Integer(2)])
[0, -2, -1, -1, 0, 0]
```

An example of type G_2:

```
sage: B = crystals.infinity.PolyhedralRealization(['G',2])
sage: mg = B.module_generators[0]; mg
[0, 0]
sage: mg.f_string([2,1,2,2])
[0, -3, -1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['G',Integer(2)])
>>> mg = B.module_generators[Integer(0)]; mg
[0, 0]
>>> mg.f_string([Integer(2),Integer(1),Integer(2),Integer(2)])
[0, -3, -1, 0, 0, 0]
```

class Element

Bases: *TensorProductOfCrystalsElement*

An element in the polyhedral realization of $B(\infty)$.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: mg = B.module_generators[0]
sage: all(mg.e(i) is None for i in B.index_set())
True
sage: mg.f(1).e(1) == mg
True
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['A',Integer(2)])
>>> mg = B.module_generators[Integer(0)]
>>> all(mg.e(i) is None for i in B.index_set())
True
>>> mg.f(Integer(1)).e(Integer(1)) == mg
True
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2,1])
sage: mg = B.module_generators[0]
sage: [mg.epsilon(i) for i in B.index_set()]
[0, 0, 0]
sage: elt = mg.f(0)
sage: [elt.epsilon(i) for i in B.index_set()]
[1, 0, 0]
sage: elt = mg.f_string([0,1,2])
sage: [elt.epsilon(i) for i in B.index_set()]
[0, 0, 1]
sage: elt = mg.f_string([0,1,2,2])
sage: [elt.epsilon(i) for i in B.index_set()]
[0, 0, 2]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['A',Integer(2),
↪Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> mg = B.module_generators[Integer(0)]
>>> [mg.epsilon(i) for i in B.index_set()]
[0, 0, 0]
>>> elt = mg.f(Integer(0))
>>> [elt.epsilon(i) for i in B.index_set()]
[1, 0, 0]
>>> elt = mg.f_string([Integer(0),Integer(1),Integer(2)])
>>> [elt.epsilon(i) for i in B.index_set()]
[0, 0, 1]
>>> elt = mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2)])
>>> [elt.epsilon(i) for i in B.index_set()]
[0, 0, 2]
```

f (*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: mg = B.module_generators[0]
sage: mg.f(1)
[-1, 0, 0, 0]
sage: mg.f_string([1,2,2,1])
[-1, -2, -1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['A',Integer(2)])
>>> mg = B.module_generators[Integer(0)]
>>> mg.f(Integer(1))
[-1, 0, 0, 0]
>>> mg.f_string([Integer(1),Integer(2),Integer(2),Integer(1)])
[-1, -2, -1, 0, 0, 0]
```

phi (*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2,1])
sage: mg = B.module_generators[0]
sage: [mg.phi(i) for i in B.index_set()]
[0, 0, 0]
sage: elt = mg.f(0)
sage: [elt.phi(i) for i in B.index_set()]
[-1, 1, 1]
sage: elt = mg.f_string([0,1])
sage: [elt.phi(i) for i in B.index_set()]
[-1, 0, 2]
sage: elt = mg.f_string([0,1,2,2])
sage: [elt.phi(i) for i in B.index_set()]
[1, 1, 0]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['A',Integer(2),
↪Integer(1)])
>>> mg = B.module_generators[Integer(0)]
>>> [mg.phi(i) for i in B.index_set()]
[0, 0, 0]
>>> elt = mg.f(Integer(0))
>>> [elt.phi(i) for i in B.index_set()]
[-1, 1, 1]
>>> elt = mg.f_string([Integer(0),Integer(1)])
>>> [elt.phi(i) for i in B.index_set()]
[-1, 0, 2]
>>> elt = mg.f_string([Integer(0),Integer(1),Integer(2),Integer(2)])
>>> [elt.phi(i) for i in B.index_set()]
[1, 1, 0]
```

truncate(*k=None*)

Truncate self to have length k and return as an element in a (finite) tensor product of crystals.

INPUT:

- k – (optional) the length of the truncation; if not specified, then returns one more than the current non-ground-state elements (i.e. the current list in self)

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: mg = B.module_generators[0]
sage: elt = mg.f_string([1,2,2,1]); elt
[-1, -2, -1, 0, 0, 0]
sage: t = elt.truncate(); t
[-1, -2, -1, 0, 0, 0]
sage: t.parent() is B.finite_tensor_product(6)
True
sage: elt.truncate(2)
[-1, -2]
sage: elt.truncate(10)
[-1, -2, -1, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['A',Integer(2)])
>>> mg = B.module_generators[Integer(0)]
>>> elt = mg.f_string([Integer(1),Integer(2),Integer(2),Integer(1)]); elt
[-1, -2, -1, 0, 0, 0]
>>> t = elt.truncate(); t
[-1, -2, -1, 0, 0, 0]
>>> t.parent() is B.finite_tensor_product(Integer(6))
True
>>> elt.truncate(Integer(2))
[-1, -2]
>>> elt.truncate(Integer(10))
[-1, -2, -1, 0, 0, 0, 0, 0, 0, 0]
```

finite_tensor_product(*k*)

Return the finite tensor product of crystals of length k by truncating self.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: B.finite_tensor_product(5)
Full tensor product of the crystals
 [The 1-elementary crystal of type ['A', 2],
  The 2-elementary crystal of type ['A', 2],
  The 1-elementary crystal of type ['A', 2],
  The 2-elementary crystal of type ['A', 2],
  The 1-elementary crystal of type ['A', 2]]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.PolyhedralRealization(['A',Integer(2)])
>>> B.finite_tensor_product(Integer(5))
Full tensor product of the crystals
 [The 1-elementary crystal of type ['A', 2],
  The 2-elementary crystal of type ['A', 2],
  The 1-elementary crystal of type ['A', 2],
  The 2-elementary crystal of type ['A', 2],
  The 1-elementary crystal of type ['A', 2]]
```

5.1.65 Spin Crystals

These are the crystals associated with the three spin representations: the spin representations of odd orthogonal groups (or rather their double covers); and the $+$ and $-$ spin representations of the even orthogonal groups.

We follow Kashiwara and Nakashima (Journal of Algebra 165, 1994) in representing the elements of the spin crystal by sequences of signs $\pm$.

`sage.combinat.crystals.spins.`**`CrystalOfSpins`**`(`*ct*`)`

Return the spin crystal of the given type B.

This is a combinatorial model for the crystal with highest weight Λ_n (the n-th fundamental weight). It has 2^n elements, here called Spins. See also *`CrystalOfLetters()`*, *`CrystalOfSpinsPlus()`*, and *`CrystalOfSpinsMinus()`*.

INPUT:

- `['B', n]` – a Cartan type B_n

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: C.list()
[+++, ++-, +-+, -++, +--, -+-, --+, ---]
sage: C.cartan_type()
['B', 3]
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> C.list()
[+++, ++-, +-+, -++, +--, -+-, --+, ---]
>>> C.cartan_type()
['B', 3]
```

```
sage: [x.signature() for x in C]
['+++', '++-', '+-+', '-++', '+--', '-+-', '--+', '---']
```

```
>>> from sage.all import *
>>> [x.signature() for x in C]
['+++', '++-', '+-+', '-++', '+--', '-+-', '--+', '---']
```

sage.combinat.crystals.spins.**CrystalOfSpinsMinus**(*ct*)

Return the minus spin crystal of the given type D.

This is the crystal with highest weight Λ_{n-1} (the $(n-1)$-st fundamental weight).

INPUT:

- `['D', n]` – a Cartan type D_n

EXAMPLES:

```
sage: E = crystals.SpinsMinus(['D',4])
sage: E.list()
[+++-, ++-+, +-++, -+++, +---, -+--, --+-, ---+]
sage: [x.signature() for x in E]
['+++-', '++-+', '+-++', '-+++', '+---', '-+--', '--+-', '---+']
```

```
>>> from sage.all import *
>>> E = crystals.SpinsMinus(['D',Integer(4)])
>>> E.list()
[+++-, ++-+, +-++, -+++, +---, -+--, --+-, ---+]
>>> [x.signature() for x in E]
['+++-', '++-+', '+-++', '-+++', '+---', '-+--', '--+-', '---+']
```

sage.combinat.crystals.spins.**CrystalOfSpinsPlus**(*ct*)

Return the plus spin crystal of the given type D.

This is the crystal with highest weight Λ_n (the n-th fundamental weight).

INPUT:

- `['D', n]` – a Cartan type D_n

EXAMPLES:

```
sage: D = crystals.SpinsPlus(['D',4])
sage: D.list()
[++++, ++--, +-+-, -++-, +--+, -+-+, --++, ----]
```

```
>>> from sage.all import *
>>> D = crystals.SpinsPlus(['D',Integer(4)])
>>> D.list()
[++++, ++--, +-+-, -++-, +--+, -+-+, --++, ----]
```

```
sage: [x.signature() for x in D]
['++++', '++--', '+-+-', '-++-', '+--+', '-+-+', '--++', '----']
```

```
>>> from sage.all import *
>>> [x.signature() for x in D]
['++++', '++--', '+-+-', '-++-', '+--+', '-+-+', '--++', '----']
```

class sage.combinat.crystals.spins.**GenericCrystalOfSpins**(*ct*, *element_class*, *case*)

Bases: `UniqueRepresentation`, `Parent`

A generic crystal of spins.

lt_elements(*x*, *y*)

Return `True` if and only if there is a path from `x` to `y` in the crystal graph.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: x = C([1,1,1])
sage: y = C([-1,-1,-1])
sage: C.lt_elements(x, y)
True
sage: C.lt_elements(y, x)
False
sage: C.lt_elements(x, x)
False
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> x = C([Integer(1),Integer(1),Integer(1)])
>>> y = C([-Integer(1),-Integer(1),-Integer(1)])
>>> C.lt_elements(x, y)
True
>>> C.lt_elements(y, x)
False
>>> C.lt_elements(x, x)
False
```

class sage.combinat.crystals.spins.**Spin**

Bases: `Element`

A spin letter in the crystal of spins.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: c = C([1,1,1])
sage: c
+++
sage: c.parent()
The crystal of spins for type ['B', 3]

sage: D = crystals.Spins(['B',4])
sage: a = C([1,1,1])
sage: b = C([-1,-1,-1])
```

(continues on next page)

(continued from previous page)

```
sage: c = D([1,1,1,1])
sage: a == a
True
sage: a == b
False
sage: b == c
False
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> c = C([Integer(1),Integer(1),Integer(1)])
>>> c
+++
>>> c.parent()
The crystal of spins for type ['B', 3]

>>> D = crystals.Spins(['B',Integer(4)])
>>> a = C([Integer(1),Integer(1),Integer(1)])
>>> b = C([-Integer(1),-Integer(1),-Integer(1)])
>>> c = D([Integer(1),Integer(1),Integer(1),Integer(1)])
>>> a == a
True
>>> a == b
False
>>> b == c
False
```

pp()

Pretty print `self` as a column.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: b = C([1,1,-1])
sage: b.pp()
+
+
-
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> b = C([Integer(1),Integer(1),-Integer(1)])
>>> b.pp()
+
+
-
```

signature()

Return the signature of `self`.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: C([1,1,1]).signature()
'+++'
sage: C([1,1,-1]).signature()
'++-'
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> C([Integer(1),Integer(1),Integer(1)]).signature()
'+++'
>>> C([Integer(1),Integer(1),-Integer(1)]).signature()
'++-'
```

value

Return `self` as a tuple with $+1$ and -1.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: C([1,1,1]).value
(1, 1, 1)
sage: C([1,1,-1]).value
(1, 1, -1)
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> C([Integer(1),Integer(1),Integer(1)]).value
(1, 1, 1)
>>> C([Integer(1),Integer(1),-Integer(1)]).value
(1, 1, -1)
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Spins(['B',3])]
[(1/2, 1/2, 1/2), (1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2), (-1/2, 1/2, 1/2),
 (1/2, -1/2, -1/2), (-1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2), (-1/2, -1/2, -1/2)]
```

```
>>> from sage.all import *
>>> [v.weight() for v in crystals.Spins(['B',Integer(3)])]
[(1/2, 1/2, 1/2), (1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2), (-1/2, 1/2, 1/2),
 (1/2, -1/2, -1/2), (-1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2), (-1/2, -1/2, -1/2)]
```

class sage.combinat.crystals.spins.**Spin_crystal_type_B_element**

Bases: *Spin*

Type B spin representation crystal element

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: [[C[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None], [None, None, +++], [None, ++-, None], [+-+, None, None],
 [None, None, +-+], [+--, None, -++], [None, -+-, None], [None, None, --+]]
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> [[C[m].e(i) for i in range(Integer(1),Integer(4))] for m in␣
↪range(Integer(8))]
[[None, None, None], [None, None, +++], [None, ++-, None], [+-+, None, None],
 [None, None, +-+], [+--, None, -++], [None, -+-, None], [None, None, --+]]
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: [[C[m].epsilon(i) for i in range(1,4)] for m in range(8)]
[[0, 0, 0], [0, 0, 1], [0, 1, 0], [1, 0, 0],
 [0, 0, 1], [1, 0, 1], [0, 1, 0], [0, 0, 1]]
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> [[C[m].epsilon(i) for i in range(Integer(1),Integer(4))] for m in␣
↪range(Integer(8))]
[[0, 0, 0], [0, 0, 1], [0, 1, 0], [1, 0, 0],
 [0, 0, 1], [1, 0, 1], [0, 1, 0], [0, 0, 1]]
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: [[C[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, ++-], [None, +-+, None], [-++, None, +--], [None, None, -+-],
 [-+-, None, None], [None, --+, None], [None, None, ---], [None, None, None]]
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> [[C[m].f(i) for i in range(Integer(1),Integer(4))] for m in␣
↪range(Integer(8))]
[[None, None, ++-], [None, +-+, None], [-++, None, +--], [None, None, -+-],
 [-+-, None, None], [None, --+, None], [None, None, ---], [None, None, None]]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: [[C[m].phi(i) for i in range(1,4)] for m in range(8)]
[[0, 0, 1], [0, 1, 0], [1, 0, 1], [0, 0, 1],
 [1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 0, 0]]
```

```
>>> from sage.all import *
>>> C = crystals.Spins(['B',Integer(3)])
>>> [[C[m].phi(i) for i in range(Integer(1),Integer(4))] for m in␣
↪range(Integer(8))]
[[0, 0, 1], [0, 1, 0], [1, 0, 1], [0, 0, 1],
 [1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 0, 0]]
```

class sage.combinat.crystals.spins.**Spin_crystal_type_D_element**

Bases: *Spin*

Type D spin representation crystal element

e(*i*)

Return the action of e_i on self.

EXAMPLES:

```
sage: D = crystals.SpinsPlus(['D',4])
sage: [[D.list()[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None], [None, None, None], [None, ++--, None], [+-+-, None,␣
↪None],
 [None, None, +-+-], [+--+, None, -++-], [None, -+-+, None], [None, None,␣
↪None]]
```

```
>>> from sage.all import *
>>> D = crystals.SpinsPlus(['D',Integer(4)])
>>> [[D.list()[m].e(i) for i in range(Integer(1),Integer(4))] for m in␣
↪range(Integer(8))]
[[None, None, None], [None, None, None], [None, ++--, None], [+-+-, None,␣
↪None],
 [None, None, +-+-], [+--+, None, -++-], [None, -+-+, None], [None, None,␣
↪None]]
```

```
sage: E = crystals.SpinsMinus(['D',4])
sage: [[E[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None], [None, None, +++-], [None, ++-+, None], [+-++, None,␣
↪None],
 [None, None, None], [+---, None, None], [None, -+--, None], [None, None, --+-
↪]]
```

```
>>> from sage.all import *
>>> E = crystals.SpinsMinus(['D',Integer(4)])
>>> [[E[m].e(i) for i in range(Integer(1),Integer(4))] for m in␣
↪range(Integer(8))]
[[None, None, None], [None, None, +++-], [None, ++-+, None], [+-++, None,␣
↪None],
 [None, None, None], [+---, None, None], [None, -+--, None], [None, None, --+-
↪]]
```

epsilon (i)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.SpinsMinus(['D',4])
sage: [[C[m].epsilon(i) for i in C.index_set()] for m in range(8)]
[[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0],
 [0, 0, 0, 1], [1, 0, 0, 1], [0, 1, 0, 0], [0, 0, 1, 0]]
```

```
>>> from sage.all import *
>>> C = crystals.SpinsMinus(['D',Integer(4)])
>>> [[C[m].epsilon(i) for i in C.index_set()] for m in range(Integer(8))]
[[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0],
 [0, 0, 0, 1], [1, 0, 0, 1], [0, 1, 0, 0], [0, 0, 1, 0]]
```

f (i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: D = crystals.SpinsPlus(['D',4])
sage: [[D.list()[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, None], [None, +-+-, None], [-++-, None, +--+], [None, None, -+-
↪+],
 [-+-+, None, None], [None, --++, None], [None, None, None], [None, None,␣
↪None]]
```

```
>>> from sage.all import *
>>> D = crystals.SpinsPlus(['D',Integer(4)])
>>> [[D.list()[m].f(i) for i in range(Integer(1),Integer(4))] for m in␣
↪range(Integer(8))]
[[None, None, None], [None, +-+-, None], [-++-, None, +--+], [None, None, -+-
↪+],
 [-+-+, None, None], [None, --++, None], [None, None, None], [None, None,␣
↪None]]
```

```
sage: E = crystals.SpinsMinus(['D',4])
sage: [[E[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, ++-+], [None, +-++, None], [-+++, None, None], [None, None,␣
↪None],
 [-+--, None, None], [None, --+-, None], [None, None, ---+], [None, None,␣
↪None]]
```

```
>>> from sage.all import *
>>> E = crystals.SpinsMinus(['D',Integer(4)])
>>> [[E[m].f(i) for i in range(Integer(1),Integer(4))] for m in␣
↪range(Integer(8))]
[[None, None, ++-+], [None, +-++, None], [-+++, None, None], [None, None,␣
↪None],
 [-+--, None, None], [None, --+-, None], [None, None, ---+], [None, None,␣
↪None]]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.SpinsPlus(['D',4])
sage: [[C[m].phi(i) for i in C.index_set()] for m in range(8)]
[[0, 0, 0, 1], [0, 1, 0, 0], [1, 0, 1, 0], [0, 0, 1, 0],
 [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1], [0, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> C = crystals.SpinsPlus(['D',Integer(4)])
>>> [[C[m].phi(i) for i in C.index_set()] for m in range(Integer(8))]
[[0, 0, 0, 1], [0, 1, 0, 0], [1, 0, 1, 0], [0, 0, 1, 0],
 [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1], [0, 0, 0, 0]]
```

5.1.66 Star-Crystal Structure On $B(\infty)$

AUTHORS:

- Ben Salisbury: Initial version
- Travis Scrimshaw: Initial version

class sage.combinat.crystals.star_crystal.**StarCrystal**(*Binf*)

Bases: `UniqueRepresentation`, `Parent`

The star-crystal or $*$-crystal version of a highest weight crystal.

The $*$-crystal structure on $B(\infty)$ is the structure induced by the algebra antiautomorphism $*\colon U_q(\mathfrak{g}) \longrightarrow U_q(\mathfrak{g})$ that stabilizes the negative half $U_q^-(\mathfrak{g})$. It is defined by

$$E_i^* = E_i, \quad F_i^* = F_i, \quad q^* = q, \quad (q^h)^* = q^{-h},$$

where E_i and F_i are the Chevalley generators of $U_q(\mathfrak{g})$ and h is an element of the Cartan subalgebra.

The induced operation on the crystal $B(\infty)$ is called the *Kashiwara involution*. Its implementation here is based on the recursive algorithm from Theorem 2.2.1 of [Ka1993], which states that for any $i \in I$ there is a unique strict crystal embedding

$$\Psi_i\colon B(\infty) \longrightarrow B_i \otimes B(\infty)$$

such that

- $u_\infty \mapsto b_i(0) \otimes u_\infty$, where u_∞ is the highest weight vector in $B(\infty)$;
- if $\Psi_i(b) = f_i^m b_i(0) \otimes b_0$, then $\Psi_i(f_i^* b) = f_i^{m+1} b_i(0) \otimes b_0$ and $\varepsilon_i(b^*) = m$;
- the image of Ψ_i is $\{f_i^m b_i(0) \otimes b : \varepsilon_i(b^*) = 0, \ m \geq 0\}$.

Here, B_i is the i-th elementary crystal. See *`ElementaryCrystal`* for more information.

INPUT:

- `Binf` – a crystal from *`catalog_infinity_crystals`*

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',2])
sage: Bstar = crystals.infinity.Star(B)
sage: mg = Bstar.highest_weight_vector()
sage: mg
[[1, 1], [2]]
sage: mg.f_string([1,2,1,2,2])
[[1, 1, 1, 1, 1, 2, 2], [2, 3, 3, 3]]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['A',Integer(2)])
>>> Bstar = crystals.infinity.Star(B)
>>> mg = Bstar.highest_weight_vector()
>>> mg
[[1, 1], [2]]
>>> mg.f_string([Integer(1),Integer(2),Integer(1),Integer(2),Integer(2)])
[[1, 1, 1, 1, 1, 2, 2], [2, 3, 3, 3]]
```

class Element

Bases: `ElementWrapper`

e(i)

Return the action of e_i^* on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['E',6,1])
sage: RCstar = crystals.infinity.Star(RC)
sage: nuJ = RCstar.module_generators[0].f_string([0,4,6,1,2])
sage: ascii_art(nuJ.e(1))
-1[ ]-1   (/)   0[ ]1   (/)   -1[ ]-1   (/)   -2[ ]-1

sage: M = crystals.infinity.NakajimaMonomials(['B',2,1])
sage: Mstar = crystals.infinity.Star(M)
sage: m = Mstar.module_generators[0].f_string([0,1,2,2,1,0])
sage: m.e(1)
Y(0,0)^-1 Y(0,2)^-1 Y(1,1) Y(1,2)^-1 Y(2,1)^2
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['E',Integer(6),
↪Integer(1)])
>>> RCstar = crystals.infinity.Star(RC)
>>> nuJ = RCstar.module_generators[Integer(0)].f_string([Integer(0),
↪Integer(4),Integer(6),Integer(1),Integer(2)])
>>> ascii_art(nuJ.e(Integer(1)))
-1[ ]-1   (/)   0[ ]1   (/)   -1[ ]-1   (/)   -2[ ]-1

>>> M = crystals.infinity.NakajimaMonomials(['B',Integer(2),Integer(1)])
>>> Mstar = crystals.infinity.Star(M)
>>> m = Mstar.module_generators[Integer(0)].f_string([Integer(0),
↪Integer(1),Integer(2),Integer(2),Integer(1),Integer(0)])
>>> m.e(Integer(1))
```

(continues on next page)

(continued from previous page)

```
Y(0,0)^-1 Y(0,2)^-1 Y(1,1) Y(1,2)^-1 Y(2,1)^2
```

epsilon(*i*)

Return ε_i^* of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(3)
sage: Ystar = crystals.infinity.Star(Y)
sage: y = Ystar.module_generators[0].f_string([0,1,3,2,1,0])
sage: [y.epsilon(i) for i in y.index_set()]
[1, 0, 1, 0]

sage: RC = crystals.infinity.RiggedConfigurations(['E',6,1])
sage: RCstar = crystals.infinity.Star(RC)
sage: nuJ = RCstar.module_generators[0].f_string([0,4,6,1,2])
sage: [nuJ.epsilon(i) for i in nuJ.index_set()]
[0, 1, 1, 0, 0, 0, 1]
```

```
>>> from sage.all import *
>>> Y = crystals.infinity.GeneralizedYoungWalls(Integer(3))
>>> Ystar = crystals.infinity.Star(Y)
>>> y = Ystar.module_generators[Integer(0)].f_string([Integer(0),
↪Integer(1),Integer(3),Integer(2),Integer(1),Integer(0)])
>>> [y.epsilon(i) for i in y.index_set()]
[1, 0, 1, 0]

>>> RC = crystals.infinity.RiggedConfigurations(['E',Integer(6),
↪Integer(1)])
>>> RCstar = crystals.infinity.Star(RC)
>>> nuJ = RCstar.module_generators[Integer(0)].f_string([Integer(0),
↪Integer(4),Integer(6),Integer(1),Integer(2)])
>>> [nuJ.epsilon(i) for i in nuJ.index_set()]
[0, 1, 1, 0, 0, 0, 1]
```

f(*i*)

Return the action of f_i^* on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: T = crystals.infinity.Tableaux("G2")
sage: Tstar = crystals.infinity.Star(T)
sage: t = Tstar.module_generators[0].f_string([1,2,1,1,2])
sage: t
[[1, 1, 1, 2, 0], [2, 3]]

sage: M = crystals.infinity.NakajimaMonomials(['B',2,1])
sage: Mstar = crystals.infinity.Star(M)
sage: m = Mstar.module_generators[0].f_string([0,1,2,2,1,0])
```

(continues on next page)

(continued from previous page)

```
sage: m
Y(0,0)^-1 Y(0,2)^-1 Y(1,0)^-1 Y(1,2)^-1 Y(2,0)^2 Y(2,1)^2
```

```
>>> from sage.all import *
>>> T = crystals.infinity.Tableaux("G2")
>>> Tstar = crystals.infinity.Star(T)
>>> t = Tstar.module_generators[Integer(0)].f_string([Integer(1),
↪Integer(2),Integer(1),Integer(1),Integer(2)])
>>> t
[[1, 1, 1, 2, 0], [2, 3]]

>>> M = crystals.infinity.NakajimaMonomials(['B',Integer(2),Integer(1)])
>>> Mstar = crystals.infinity.Star(M)
>>> m = Mstar.module_generators[Integer(0)].f_string([Integer(0),
↪Integer(1),Integer(2),Integer(2),Integer(1),Integer(0)])
>>> m
Y(0,0)^-1 Y(0,2)^-1 Y(1,0)^-1 Y(1,2)^-1 Y(2,0)^2 Y(2,1)^2
```

jump(*i*)

Return the i-jump of `self`.

For $b \in B(\infty)$,

$$\mathrm{jump}_i(b) = \varepsilon_i(b) + \varepsilon_i^*(b) + \langle h_i, \mathrm{wt}(b) \rangle,$$

where h_i is a simple coroot.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations("D4")
sage: RCstar = crystals.infinity.Star(RC)
sage: nu0star = RCstar.module_generators[0]
sage: nustar = nu0star.f_string([2,1,3,4,2])
sage: [nustar.jump(i) for i in RC.index_set()]
[0, 1, 0, 0]
sage: nustar = nu0star.f_string([2,1,3,4,2,2,1,3,2]) # long time
sage: [nustar.jump(i) for i in RC.index_set()] # long time
[1, 0, 1, 2]
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations("D4")
>>> RCstar = crystals.infinity.Star(RC)
>>> nu0star = RCstar.module_generators[Integer(0)]
>>> nustar = nu0star.f_string([Integer(2),Integer(1),Integer(3),
↪Integer(4),Integer(2)])
>>> [nustar.jump(i) for i in RC.index_set()]
[0, 1, 0, 0]
>>> nustar = nu0star.f_string([Integer(2),Integer(1),Integer(3),
↪Integer(4),Integer(2),Integer(2),Integer(1),Integer(3),Integer(2)]) #␣
↪long time
```

(continues on next page)

(continued from previous page)

```
>>> [nustar.jump(i) for i in RC.index_set()] # long time
[1, 0, 1, 2]
```

phi (i)

Return φ_i^* of `self`.

For $b \in B(\infty)$,

$$\varphi_i^*(b) = \varepsilon_i^*(b) + \langle h_i, \mathrm{wt}(b) \rangle,$$

where h_i is a simple coroot.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: T = crystals.infinity.Tableaux("A2")
sage: Tstar = crystals.infinity.Star(T)
sage: t = Tstar.module_generators[0].f_string([1,2,1,1,2])
sage: [t.phi(i) for i in t.index_set()]
[-3, 1]

sage: M = crystals.infinity.NakajimaMonomials(['B',2,1])
sage: Mstar = crystals.infinity.Star(M)
sage: m = Mstar.module_generators[0].f_string([0,1,2,2,1,0])
sage: [m.phi(i) for i in m.index_set()]
[-1, -1, 4]
```

```
>>> from sage.all import *
>>> T = crystals.infinity.Tableaux("A2")
>>> Tstar = crystals.infinity.Star(T)
>>> t = Tstar.module_generators[Integer(0)].f_string([Integer(1),
↪Integer(2),Integer(1),Integer(1),Integer(2)])
>>> [t.phi(i) for i in t.index_set()]
[-3, 1]

>>> M = crystals.infinity.NakajimaMonomials(['B',Integer(2),Integer(1)])
>>> Mstar = crystals.infinity.Star(M)
>>> m = Mstar.module_generators[Integer(0)].f_string([Integer(0),
↪Integer(1),Integer(2),Integer(2),Integer(1),Integer(0)])
>>> [m.phi(i) for i in m.index_set()]
[-1, -1, 4]
```

weight ()

Return the weight of `self`.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['E',6,1])
sage: RCstar = crystals.infinity.Star(RC)
sage: nuJ = RCstar.module_generators[0].f_string([0,4,6,1,2])
sage: nuJ.weight()
-Lambda[0] - 2*Lambda[1] + 2*Lambda[3] - Lambda[4]
 + 2*Lambda[5] - 2*Lambda[6] - delta
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['E',Integer(6),
↪Integer(1)])
>>> RCstar = crystals.infinity.Star(RC)
>>> nuJ = RCstar.module_generators[Integer(0)].f_string([Integer(0),
↪Integer(4),Integer(6),Integer(1),Integer(2)])
>>> nuJ.weight()
-Lambda[0] - 2*Lambda[1] + 2*Lambda[3] - Lambda[4]
 + 2*Lambda[5] - 2*Lambda[6] - delta
```

5.1.67 Tensor Products of Crystals

Main entry points:

- *TensorProductOfCrystals*
- *CrystalOfTableaux*

AUTHORS:

- Anne Schilling, Nicolas Thiery (2007): Initial version
- Ben Salisbury, Travis Scrimshaw (2013): Refactored tensor products to handle non-regular crystals and created new subclass to take advantage of the regularity
- Travis Scrimshaw (2020): Added queer crystal

class sage.combinat.crystals.tensor_product.**CrystalOfQueerTableaux**(*cartan_type*, *shape*)

Bases: *CrystalOfWords*, *QueerSuperCrystalsMixin*

A queer crystal of the semistandard decomposition tableaux of a given shape.

INPUT:

- `cartan_type` – a Cartan type
- `shape` – a shape

class Element

Bases: *TensorProductOfQueerSuperCrystalsElement*

rows()

Return the list of rows of `self`.

EXAMPLES:

```
sage: B = crystals.Tableaux(['Q',3], shape=[3,2,1])
sage: t = B.an_element()
sage: t.rows()
[[3, 3, 3], [2, 2], [1]]
```

```
>>> from sage.all import *
>>> B = crystals.Tableaux(['Q',Integer(3)], shape=[Integer(3),Integer(2),
↪Integer(1)])
>>> t = B.an_element()
>>> t.rows()
[[3, 3, 3], [2, 2], [1]]
```

class sage.combinat.crystals.tensor_product.**CrystalOfTableaux**(*cartan_type*, *shapes*)

Bases: `CrystalOfWords`

A class for crystals of tableaux with integer valued shapes.

INPUT:

- `cartan_type` – a Cartan type
- `shape` – a partition of length at most `cartan_type.rank()`
- `shapes` – list of such partitions

This constructs a classical crystal with the given Cartan type and highest weight(s) corresponding to the given shape(s).

If the type is D_r, the shape is permitted to have a negative value in the r-th position. Thus if the shape equals $[s_1, \ldots, s_r]$, then s_r may be negative but in any case $s_1 \geq \cdots \geq s_{r-1} \geq |s_r|$. This crystal is related to that of shape $[s_1, \ldots, |s_r|]$ by the outer automorphism of $SO(2r)$.

If the type is D_r or B_r, the shape is permitted to be of length r with all parts of half integer value. This corresponds to having one spin column at the beginning of the tableau. If several shapes are provided, they currently should all or none have this property.

Crystals of tableaux are constructed using an embedding into tensor products following Kashiwara and Nakashima [KN1994]. Sage's tensor product rule for crystals differs from that of Kashiwara and Nakashima by reversing the order of the tensor factors. Sage produces the same crystals of tableaux as Kashiwara and Nakashima. With Sage's convention, the tensor product of crystals is the same as the monoid operation on tableaux and hence the plactic monoid.

See also

`sage.combinat.crystals.crystals` for general help on crystals, and in particular plotting and LaTeX output.

EXAMPLES:

We create the crystal of tableaux for type A_2, with highest weight given by the partition $[2, 1, 1]$:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,1,1])
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A',Integer(3)], shape = [Integer(2),Integer(1),
↪Integer(1)])
```

Here is the list of its elements:

```
sage: T.list()
[[[1, 1], [2], [3]], [[1, 2], [2], [3]], [[1, 3], [2], [3]],
 [[1, 4], [2], [3]], [[1, 4], [2], [4]], [[1, 4], [3], [4]],
 [[2, 4], [3], [4]], [[1, 1], [2], [4]], [[1, 2], [2], [4]],
 [[1, 3], [2], [4]], [[1, 3], [3], [4]], [[2, 3], [3], [4]],
 [[1, 1], [3], [4]], [[1, 2], [3], [4]], [[2, 2], [3], [4]]]
```

```
>>> from sage.all import *
>>> T.list()
[[[1, 1], [2], [3]], [[1, 2], [2], [3]], [[1, 3], [2], [3]],
```

(continues on next page)

(continued from previous page)

```
[[1, 4], [2], [3]], [[1, 4], [2], [4]], [[1, 4], [3], [4]],
[[2, 4], [3], [4]], [[1, 1], [2], [4]], [[1, 2], [2], [4]],
[[1, 3], [2], [4]], [[1, 3], [3], [4]], [[2, 3], [3], [4]],
[[1, 1], [3], [4]], [[1, 2], [3], [4]], [[2, 2], [3], [4]]]
```

Internally, a tableau of a given Cartan type is represented as a tensor product of letters of the same type. The order in which the tensor factors appear is by reading the columns of the tableaux left to right, top to bottom (in French notation). As an example:

```
sage: T = crystals.Tableaux(['A',2], shape = [3,2])
sage: T.module_generators[0]
[[1, 1, 1], [2, 2]]
sage: list(T.module_generators[0])
[2, 1, 2, 1, 1]
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A',Integer(2)], shape = [Integer(3),Integer(2)])
>>> T.module_generators[Integer(0)]
[[1, 1, 1], [2, 2]]
>>> list(T.module_generators[Integer(0)])
[2, 1, 2, 1, 1]
```

To create a tableau, one can use:

```
sage: Tab = crystals.Tableaux(['A',3], shape = [2,2])
sage: Tab(rows=[[1,2],[3,4]])
[[1, 2], [3, 4]]
sage: Tab(columns=[[3,1],[4,2]])
[[1, 2], [3, 4]]
```

```
>>> from sage.all import *
>>> Tab = crystals.Tableaux(['A',Integer(3)], shape = [Integer(2),Integer(2)])
>>> Tab(rows=[[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
[[1, 2], [3, 4]]
>>> Tab(columns=[[Integer(3),Integer(1)],[Integer(4),Integer(2)]])
[[1, 2], [3, 4]]
```

Todo

FIXME:

- Do we want to specify the columns increasingly or decreasingly? That is, should this be `Tab(columns = [[1,3],[2,4]])`?
- Make this fully consistent with `Tableau()`!

We illustrate the use of a shape with a negative last entry in type D:

```
sage: T = crystals.Tableaux(['D',4],shape=[1,1,1,-1])
sage: T.cardinality()
35
sage: TestSuite(T).run()
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['D',Integer(4)],shape=[Integer(1),Integer(1),
↪Integer(1),-Integer(1)])
>>> T.cardinality()
35
>>> TestSuite(T).run()
```

We illustrate the construction of crystals of spin tableaux when the partitions have half integer values in type B and D:

```
sage: T = crystals.Tableaux(['B',3],shape=[3/2,1/2,1/2]); T
The crystal of tableaux of type ['B', 3] and shape(s) [[3/2, 1/2, 1/2]]
sage: T.cardinality()
48
sage: T.module_generators
([+++, [[1]]],)
sage: TestSuite(T).run()

sage: T = crystals.Tableaux(['D',3],shape=[3/2,1/2,-1/2]); T
The crystal of tableaux of type ['D', 3] and shape(s) [[3/2, 1/2, -1/2]]
sage: T.cardinality()
20
sage: T.module_generators
([++-, [[1]]],)
sage: TestSuite(T).run()
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['B',Integer(3)],shape=[Integer(3)/Integer(2),
↪Integer(1)/Integer(2),Integer(1)/Integer(2)]); T
The crystal of tableaux of type ['B', 3] and shape(s) [[3/2, 1/2, 1/2]]
>>> T.cardinality()
48
>>> T.module_generators
([+++, [[1]]],)
>>> TestSuite(T).run()

>>> T = crystals.Tableaux(['D',Integer(3)],shape=[Integer(3)/Integer(2),
↪Integer(1)/Integer(2),-Integer(1)/Integer(2)]); T
The crystal of tableaux of type ['D', 3] and shape(s) [[3/2, 1/2, -1/2]]
>>> T.cardinality()
20
>>> T.module_generators
([++-, [[1]]],)
>>> TestSuite(T).run()
```

We can also construct the tableaux for $\mathfrak{gl}(m|n)$ as given by [BKK2000]:

```
sage: T = crystals.Tableaux(['A', [1,2]], shape=[4,2,1,1,1])
sage: T.cardinality()
1392
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A', [Integer(1),Integer(2)]], shape=[Integer(4),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(1),Integer(1),Integer(1)])
>>> T.cardinality()
1392
```

We can also construct the tableaux for $\mathfrak{q}(n)$ as given by [GJK+2014]:

```
sage: T = crystals.Tableaux(['Q', 3], shape=[3,1])
sage: T.cardinality()
24
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['Q', Integer(3)], shape=[Integer(3),Integer(1)])
>>> T.cardinality()
24
```

class Element

Bases: *CrystalOfTableauxElement*

cartan_type()

Return the Cartan type of the associated crystal.

EXAMPLES:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,2])
sage: T.cartan_type()
['A', 3]
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A',Integer(3)], shape = [Integer(2),Integer(2)])
>>> T.cartan_type()
['A', 3]
```

module_generator(*shape*)

This yields the module generator (or highest weight element) of a classical crystal of given shape. The module generator is the unique tableau with equal shape and content.

EXAMPLES:

```
sage: T = crystals.Tableaux(['D',3], shape = [1,1])
sage: T.module_generator([1,1])
[[1], [2]]

sage: T = crystals.Tableaux(['D',4],shape=[2,2,2,-2])
sage: T.module_generator(tuple([2,2,2,-2]))
[[1, 1], [2, 2], [3, 3], [-4, -4]]
sage: T.cardinality()
294
sage: T = crystals.Tableaux(['D',4],shape=[2,2,2,2])
sage: T.module_generator(tuple([2,2,2,2]))
[[1, 1], [2, 2], [3, 3], [4, 4]]
sage: T.cardinality()
294
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['D',Integer(3)], shape = [Integer(1),Integer(1)])
>>> T.module_generator([Integer(1),Integer(1)])
[[1], [2]]

>>> T = crystals.Tableaux(['D',Integer(4)],shape=[Integer(2),Integer(2),
→Integer(2),-Integer(2)])
>>> T.module_generator(tuple([Integer(2),Integer(2),Integer(2),-Integer(2)]))
[[1, 1], [2, 2], [3, 3], [-4, -4]]
>>> T.cardinality()
294
>>> T = crystals.Tableaux(['D',Integer(4)],shape=[Integer(2),Integer(2),
→Integer(2),Integer(2)])
>>> T.module_generator(tuple([Integer(2),Integer(2),Integer(2),Integer(2)]))
[[1, 1], [2, 2], [3, 3], [4, 4]]
>>> T.cardinality()
294
```

class sage.combinat.crystals.tensor_product.**CrystalOfWords**

Bases: UniqueRepresentation, Parent

Auxiliary class to provide a call method to create tensor product elements. This class is shared with several tensor product classes and is also used in *CrystalOfTableaux* to allow tableaux of different tensor product structures in column-reading (and hence different shapes) to be considered elements in the same crystal.

class Element

Bases: *TensorProductOfCrystalsElement*

class sage.combinat.crystals.tensor_product.**FullTensorProductOfCrystals**(*crystals*, ***options*)

Bases: *TensorProductOfCrystals*

Full tensor product of crystals.

Todo

Merge this into *TensorProductOfCrystals*.

cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C)
sage: T.cardinality()
9
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> T = crystals.TensorProduct(C,C)
>>> T.cardinality()
9
```

weight_lattice_realization()

Return the weight lattice realization used to express weights.

The weight lattice realization is the common parent which all weight lattice realizations of the crystals of `self` coerce into.

EXAMPLES:

```
sage: B = crystals.elementary.B(['A',4], 2)
sage: B.weight_lattice_realization()
Root lattice of the Root system of type ['A', 4]
sage: T = crystals.infinity.Tableaux(['A',4])
sage: T.weight_lattice_realization()
Ambient space of the Root system of type ['A', 4]
sage: TP = crystals.TensorProduct(B, T)
sage: TP.weight_lattice_realization()
Ambient space of the Root system of type ['A', 4]
```

```
>>> from sage.all import *
>>> B = crystals.elementary.B(['A',Integer(4)], Integer(2))
>>> B.weight_lattice_realization()
Root lattice of the Root system of type ['A', 4]
>>> T = crystals.infinity.Tableaux(['A',Integer(4)])
>>> T.weight_lattice_realization()
Ambient space of the Root system of type ['A', 4]
>>> TP = crystals.TensorProduct(B, T)
>>> TP.weight_lattice_realization()
Ambient space of the Root system of type ['A', 4]
```

class sage.combinat.crystals.tensor_product.**FullTensorProductOfQueerSuperCrystals**(*crystals*, ***options*)

Bases: *FullTensorProductOfCrystals*, *QueerSuperCrystalsMixin*

Tensor product of queer super crystals.

class Element

Bases: *TensorProductOfQueerSuperCrystalsElement*

class sage.combinat.crystals.tensor_product.**FullTensorProductOfRegularCrystals**(*crystals*, ***options*)

Bases: *FullTensorProductOfCrystals*

Full tensor product of regular crystals.

class Element

Bases: *TensorProductOfRegularCrystalsElement*

class sage.combinat.crystals.tensor_product.**FullTensorProductOfSuperCrystals**(*crystals*, ***options*)

Bases: *FullTensorProductOfCrystals*

Tensor product of super crystals.

EXAMPLES:

```
sage: L = crystals.Letters(['A', [1,1]])
sage: T = tensor([L,L,L])
sage: T.cardinality()
64
```

```
>>> from sage.all import *
>>> L = crystals.Letters(['A', [Integer(1),Integer(1)]])
>>> T = tensor([L,L,L])
>>> T.cardinality()
64
```

class Element

Bases: *TensorProductOfSuperCrystalsElement*

class sage.combinat.crystals.tensor_product.**QueerSuperCrystalsMixin**

Bases: object

Mixin class with methods for a finite queer supercrystal.

index_set()

Return the enlarged index set.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q',3])
sage: T = tensor([Q,Q])
sage: T.index_set()
(-4, -3, -2, -1, 1, 2)
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q',Integer(3)])
>>> T = tensor([Q,Q])
>>> T.index_set()
(-4, -3, -2, -1, 1, 2)
```

class sage.combinat.crystals.tensor_product.**TensorProductOfCrystals**

Bases: *CrystalOfWords*

Tensor product of crystals.

Given two crystals B and B' of the same Cartan type, one can form the tensor product $B \otimes B'$. As a set $B \otimes B'$ is the Cartesian product $B \times B'$. The crystal operators f_i and e_i act on $b \otimes b' \in B \otimes B'$ as follows:

$$f_i(b \otimes b') = \begin{cases} f_i(b) \otimes b' & \text{if } \varepsilon_i(b) \geq \varphi_i(b') \\ b \otimes f_i(b') & \text{otherwise} \end{cases}$$

and

$$e_i(b \otimes b') = \begin{cases} e_i(b) \otimes b' & \text{if } \varepsilon_i(b) > \varphi_i(b') \\ b \otimes e_i(b') & \text{otherwise.} \end{cases}$$

We also define:

$$\varphi_i(b \otimes b') = \max\left(\varphi_i(b), \varphi_i(b') + \langle \alpha_i^\vee, \mathrm{wt}(b) \rangle\right),$$
$$\varepsilon_i(b \otimes b') = \max\left(\varepsilon_i(b'), \varepsilon_i(b) - \langle \alpha_i^\vee, \mathrm{wt}(b') \rangle\right).$$

Note

This is the opposite of Kashiwara's convention for tensor products of crystals.

Since tensor products are associative $(\mathcal{B} \otimes \mathcal{C}) \otimes \mathcal{D} \cong \mathcal{B} \otimes (\mathcal{C} \otimes \mathcal{D})$ via the natural isomorphism $(b \otimes c) \otimes d \mapsto b \otimes (c \otimes d)$, we can generalizing this to arbitrary tensor products. Thus consider $B_N \otimes \cdots \otimes B_1$, where each B_k is an abstract crystal. The underlying set of the tensor product is $B_N \times \cdots \times B_1$, while the crystal structure is given as follows. Let I be the index set, and fix some $i \in I$ and $b_N \otimes \cdots \otimes b_1 \in B_N \otimes \cdots \otimes B_1$. Define

$$a_i(k) := \varepsilon_i(b_k) - \sum_{j=1}^{k-1} \langle \alpha_i^\vee, \mathrm{wt}(b_j) \rangle.$$

Then

$$\begin{aligned}
\mathrm{wt}(b_N \otimes \cdots \otimes b_1) &= \mathrm{wt}(b_N) + \cdots + \mathrm{wt}(b_1), \\
\varepsilon_i(b_N \otimes \cdots \otimes b_1) &= \max_{1 \leq k \leq n} \left(\sum_{j=1}^{k} \varepsilon_i(b_j) - \sum_{j=1}^{k-1} \varphi_i(b_j) \right) \\
&= \max_{1 \leq k \leq N} \big(a_i(k) \big), \\
\varphi_i(b_N \otimes \cdots \otimes b_1) &= \max_{1 \leq k \leq N} \left(\varphi_i(b_N) + \sum_{j=k}^{N-1} \big(\varphi_i(b_j) - \varepsilon_i(b_{j+1}) \big) \right) \\
&= \max_{1 \leq k \leq N} \big(\lambda_i + a_i(k) \big)
\end{aligned}$$

where $\lambda_i = \langle \alpha_i^\vee, \mathrm{wt}(b_N \otimes \cdots \otimes b_1) \rangle$. Then for $k = 1, \ldots, N$ the action of the Kashiwara operators is determined as follows.

- If $a_i(k) > a_i(j)$ for $1 \leq j < k$ and $a_i(k) \geq a_i(j)$ for $k < j \leq N$:

$$e_i(b_N \otimes \cdots \otimes b_1) = b_N \otimes \cdots \otimes e_i b_k \otimes \cdots \otimes b_1.$$

- If $a_i(k) \geq a_i(j)$ for $1 \leq j < k$ and $a_i(k) > a_i(j)$ for $k < j \leq N$:

$$f_i(b_N \otimes \cdots \otimes b_1) = b_N \otimes \cdots \otimes f_i b_k \otimes \cdots \otimes b_1.$$

Note that this is just recursively applying the definition of the tensor product on two crystals. Recall that $\langle \alpha_i^\vee, \mathrm{wt}(b_j) \rangle = \varphi_i(b_j) - \varepsilon_i(b_j)$ by the definition of a crystal.

Regular crystals

Now if all crystals B_k are regular crystals, all ε_i and φ_i are nonnegative and we can define tensor product by the *signature rule*. We start by writing a word in $+$ and $-$ as follows:

$$\underbrace{- \cdots -}_{\varphi_i(b_N) \text{ times}} \quad \underbrace{+ \cdots +}_{\varepsilon_i(b_N) \text{ times}} \quad \cdots \quad \underbrace{- \cdots -}_{\varphi_i(b_1) \text{ times}} \quad \underbrace{+ \cdots +}_{\varepsilon_i(b_1) \text{ times}},$$

and then canceling ordered pairs of $+-$ until the word is in the reduced form:

$$\underbrace{- \cdots -}_{\varphi_i \text{ times}} \quad \underbrace{+ \cdots +}_{\varepsilon_i \text{ times}}.$$

Here e_i acts on the factor corresponding to the leftmost $+$ and f_i on the factor corresponding to the rightmost $-$. If there is no $+$ or $-$ respectively, then the result is 0 (`None`).

EXAMPLES:

We construct the type A_2-crystal generated by $2 \otimes 1 \otimes 1$:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C,C,generators=[[C(2),C(1),C(1)]])
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> T = crystals.TensorProduct(C,C,C,generators=[[C(Integer(2)),C(Integer(1)),
↪C(Integer(1))]])
```

It has 8 elements:

```
sage: T.list()
[[2, 1, 1], [2, 1, 2], [2, 1, 3], [3, 1, 3],
 [3, 2, 3], [3, 1, 1], [3, 1, 2], [3, 2, 2]]
```

```
>>> from sage.all import *
>>> T.list()
[[2, 1, 1], [2, 1, 2], [2, 1, 3], [3, 1, 3],
 [3, 2, 3], [3, 1, 1], [3, 1, 2], [3, 2, 2]]
```

One can also check the Cartan type of the crystal:

```
sage: T.cartan_type()
['A', 2]
```

```
>>> from sage.all import *
>>> T.cartan_type()
['A', 2]
```

Other examples include crystals of tableaux (which internally are represented as tensor products obtained by reading the tableaux columnwise):

```
sage: C = crystals.Tableaux(['A',3], shape=[1,1,0])
sage: D = crystals.Tableaux(['A',3], shape=[1,0,0])
sage: T = crystals.TensorProduct(C,D, generators=[[C(rows=[[1], [2]]),␣
↪D(rows=[[1]])], [C(rows=[[2], [3]]), D(rows=[[1]])]])
sage: T.cardinality()
24
sage: TestSuite(T).run()
sage: T.module_generators
([[[1], [2]], [[1]]], [[[2], [3]], [[1]]])
sage: [x.weight() for x in T.module_generators]
[(2, 1, 0, 0), (1, 1, 1, 0)]
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',Integer(3)], shape=[Integer(1),Integer(1),
↪Integer(0)])
>>> D = crystals.Tableaux(['A',Integer(3)], shape=[Integer(1),Integer(0),
↪Integer(0)])
>>> T = crystals.TensorProduct(C,D, generators=[[C(rows=[[Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(2)]]), D(rows=[[Integer(1)]])], [C(rows=[[Integer(2)], [Integer(3)]]),␣
↪D(rows=[[Integer(1)]])]])
>>> T.cardinality()
24
>>> TestSuite(T).run()
>>> T.module_generators
([[[1], [2]], [[1]]], [[[2], [3]], [[1]]])
>>> [x.weight() for x in T.module_generators]
[(2, 1, 0, 0), (1, 1, 1, 0)]
```

If no module generators are specified, we obtain the full tensor product:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C)
sage: T.list()
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
sage: T.cardinality()
9
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> T = crystals.TensorProduct(C,C)
>>> T.list()
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
>>> T.cardinality()
9
```

For a tensor product of crystals without module generators, the default implementation of `module_generators` contains all elements in the tensor product of the crystals. If there is a subset of elements in the tensor product that still generates the crystal, this needs to be implemented for the specific crystal separately:

```
sage: T.module_generators.list()
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
```

```
>>> from sage.all import *
>>> T.module_generators.list()
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
```

For classical highest weight crystals, it is also possible to list all highest weight elements:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C,C,generators=[[C(2),C(1),C(1)],[C(1),C(2),
↪C(1)]])
sage: T.highest_weight_vectors()
([2, 1, 1], [1, 2, 1])
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> T = crystals.TensorProduct(C,C,C,generators=[[C(Integer(2)),C(Integer(1)),
↪C(Integer(1))],[C(Integer(1)),C(Integer(2)),C(Integer(1))]])
>>> T.highest_weight_vectors()
([2, 1, 1], [1, 2, 1])
```

Examples with non-regular and infinite crystals (these did not work before Issue #14402):

```
sage: B = crystals.infinity.Tableaux(['D',10])
sage: T = crystals.TensorProduct(B,B)
sage: T
Full tensor product of the crystals
[The infinity crystal of tableaux of type ['D', 10],
 The infinity crystal of tableaux of type ['D', 10]]

sage: B = crystals.infinity.GeneralizedYoungWalls(15)
sage: T = crystals.TensorProduct(B,B,B)
sage: T
Full tensor product of the crystals
[Crystal of generalized Young walls of type ['A', 15, 1],
 Crystal of generalized Young walls of type ['A', 15, 1],
 Crystal of generalized Young walls of type ['A', 15, 1]]

sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
→weights()
sage: B = crystals.GeneralizedYoungWalls(2,La[0]+La[1])
sage: C = crystals.GeneralizedYoungWalls(2,2*La[2])
sage: D = crystals.GeneralizedYoungWalls(2,3*La[0]+La[2])
sage: T = crystals.TensorProduct(B,C,D)
sage: T
Full tensor product of the crystals
[Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and␣
→highest weight Lambda[0] + Lambda[1],
 Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and␣
→highest weight 2*Lambda[2],
 Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and␣
→highest weight 3*Lambda[0] + Lambda[2]]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['D',Integer(10)])
>>> T = crystals.TensorProduct(B,B)
>>> T
Full tensor product of the crystals
[The infinity crystal of tableaux of type ['D', 10],
 The infinity crystal of tableaux of type ['D', 10]]

>>> B = crystals.infinity.GeneralizedYoungWalls(Integer(15))
>>> T = crystals.TensorProduct(B,B,B)
>>> T
Full tensor product of the crystals
[Crystal of generalized Young walls of type ['A', 15, 1],
 Crystal of generalized Young walls of type ['A', 15, 1],
 Crystal of generalized Young walls of type ['A', 15, 1]]

>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice(extended=True).
→fundamental_weights()
>>> B = crystals.GeneralizedYoungWalls(Integer(2),La[Integer(0)]+La[Integer(1)])
>>> C = crystals.GeneralizedYoungWalls(Integer(2),Integer(2)*La[Integer(2)])
>>> D = crystals.GeneralizedYoungWalls(Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)*La[Integer(0)]+La[Integer(2)])
>>> T = crystals.TensorProduct(B,C,D)
>>> T
Full tensor product of the crystals
[Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and␣
↪highest weight Lambda[0] + Lambda[1],
 Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and␣
↪highest weight 2*Lambda[2],
 Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and␣
↪highest weight 3*Lambda[0] + Lambda[2]]
```

There is also a global option for setting the convention (by default Sage uses anti-Kashiwara):

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C)
sage: elt = T(C(1), C(2)); elt
[1, 2]
sage: crystals.TensorProduct.options.convention = "Kashiwara"
sage: elt
[2, 1]
sage: crystals.TensorProduct.options._reset()
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> T = crystals.TensorProduct(C,C)
>>> elt = T(C(Integer(1)), C(Integer(2))); elt
[1, 2]
>>> crystals.TensorProduct.options.convention = "Kashiwara"
>>> elt
[2, 1]
>>> crystals.TensorProduct.options._reset()
```

options = Current options for TensorProductOfCrystals - convention: antiKashiwara

class sage.combinat.crystals.tensor_product.**TensorProductOfCrystalsWithGenerators**(*crystals*, *generators*, *cartan_type*)

Bases: *TensorProductOfCrystals*

Tensor product of crystals with a generating set.

Todo

Deprecate this class in favor of using `subcrystal()`.

class sage.combinat.crystals.tensor_product.**TensorProductOfRegularCrystalsWithGenerators**(*crystals*, *generators*, *cartan_type*)

Bases: *TensorProductOfCrystalsWithGenerators*

Tensor product of regular crystals with a generating set.

class Element

Bases: *TensorProductOfRegularCrystalsElement*

5.1.68 Tensor Products of Crystal Elements

AUTHORS:

- Anne Schilling, Nicolas Thiery (2007): Initial version
- Ben Salisbury, Travis Scrimshaw (2013): Refactored tensor products to handle non-regular crystals and created new subclass to take advantage of the regularity
- Travis Scrimshaw (2017): Cythonized element classes
- Franco Saliola (2017): Tensor products for crystal of super algebras
- Anne Schilling (2018): Tensor products for crystals of queer super algebras

class sage.combinat.crystals.tensor_product_element.**CrystalOfBKKTableauxElement**

Bases: *TensorProductOfSuperCrystalsElement*

Element class for the crystal of tableaux for Lie superalgebras of [BKK2000].

pp()

Pretty print self.

EXAMPLES:

```
sage: C = crystals.Tableaux(['A',[1,2]], shape=[1,1])
sage: c = C.an_element()
sage: c.pp()
-2
-1
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',[Integer(1),Integer(2)]], shape=[Integer(1),
↪Integer(1)])
>>> c = C.an_element()
>>> c.pp()
-2
-1
```

to_tableau()

Return the *Tableau* object corresponding to self.

EXAMPLES:

```
sage: C = crystals.Tableaux(['A',[1,2]], shape=[1,1])
sage: c = C.an_element()
sage: c.to_tableau()
[[-2], [-1]]
sage: type(c.to_tableau())
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
sage: type(c)
<class 'sage.combinat.crystals.bkk_crystals.CrystalOfBKKTableaux_with_
↪category.element_class'>
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',[Integer(1),Integer(2)]], shape=[Integer(1),
↪Integer(1)])
>>> c = C.an_element()
>>> c.to_tableau()
[[-2], [-1]]
>>> type(c.to_tableau())
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
>>> type(c)
<class 'sage.combinat.crystals.bkk_crystals.CrystalOfBKKTableaux_with_
↪category.element_class'>
```

class sage.combinat.crystals.tensor_product_element.**CrystalOfTableauxElement**

Bases: *TensorProductOfRegularCrystalsElement*

Element in a crystal of tableaux.

pp()

EXAMPLES:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,2])
sage: t = T(rows=[[1,2],[3,4]])
sage: t.pp()
1  2
3  4
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A',Integer(3)], shape = [Integer(2),Integer(2)])
>>> t = T(rows=[[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
>>> t.pp()
1  2
3  4
```

promotion()

Return the result of applying promotion on `self`.

Promotion for type A crystals of tableaux of rectangular shape. This method only makes sense in type A with rectangular shapes.

EXAMPLES:

```
sage: C = crystals.Tableaux(["A",3], shape = [3,3,3])
sage: t = C(Tableau([[1,1,1],[2,2,3],[3,4,4]]))
sage: t
```

(continues on next page)

(continued from previous page)

```
[[1, 1, 1], [2, 2, 3], [3, 4, 4]]
sage: t.promotion()
[[1, 1, 2], [2, 2, 3], [3, 4, 4]]
sage: t.promotion().parent()
The crystal of tableaux of type ['A', 3] and shape(s) [[3, 3, 3]]
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(["A",Integer(3)], shape = [Integer(3),Integer(3),
↪Integer(3)])
>>> t = C(Tableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(2),
↪Integer(3)],[Integer(3),Integer(4),Integer(4)]]))
>>> t
[[1, 1, 1], [2, 2, 3], [3, 4, 4]]
>>> t.promotion()
[[1, 1, 2], [2, 2, 3], [3, 4, 4]]
>>> t.promotion().parent()
The crystal of tableaux of type ['A', 3] and shape(s) [[3, 3, 3]]
```

promotion_inverse()

Return the result of applying inverse promotion on `self`.

Inverse promotion for type A crystals of tableaux of rectangular shape. This method only makes sense in type A with rectangular shapes.

EXAMPLES:

```
sage: C = crystals.Tableaux(["A",3], shape = [3,3,3])
sage: t = C(Tableau([[1,1,1],[2,2,3],[3,4,4]]))
sage: t
[[1, 1, 1], [2, 2, 3], [3, 4, 4]]
sage: t.promotion_inverse()
[[1, 1, 2], [2, 3, 3], [4, 4, 4]]
sage: t.promotion_inverse().parent()
The crystal of tableaux of type ['A', 3] and shape(s) [[3, 3, 3]]
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(["A",Integer(3)], shape = [Integer(3),Integer(3),
↪Integer(3)])
>>> t = C(Tableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(2),
↪Integer(3)],[Integer(3),Integer(4),Integer(4)]]))
>>> t
[[1, 1, 1], [2, 2, 3], [3, 4, 4]]
>>> t.promotion_inverse()
[[1, 1, 2], [2, 3, 3], [4, 4, 4]]
>>> t.promotion_inverse().parent()
The crystal of tableaux of type ['A', 3] and shape(s) [[3, 3, 3]]
```

shape()

Return the shape of the tableau corresponding to `self`.

OUTPUT: an instance of *Partition*

See also

to_tableau()

EXAMPLES:

```
sage: C = crystals.Tableaux(["A", 2], shape=[2,1])
sage: x = C.an_element()
sage: x.to_tableau().shape()
[2, 1]
sage: x.shape()
[2, 1]
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(["A", Integer(2)], shape=[Integer(2),Integer(1)])
>>> x = C.an_element()
>>> x.to_tableau().shape()
[2, 1]
>>> x.shape()
[2, 1]
```

to_tableau()

Return the *Tableau* object corresponding to `self`.

EXAMPLES:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,2])
sage: t = T(rows=[[1,2],[3,4]]).to_tableau(); t
[[1, 2], [3, 4]]
sage: type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
sage: type(t[0][0])
<class 'int'>
sage: T = crystals.Tableaux(['D',3], shape = [1,1])
sage: t=T(rows=[[-3],[3]]).to_tableau(); t
[[-3], [3]]
sage: t=T(rows=[[3],[-3]]).to_tableau(); t
[[3], [-3]]
sage: T = crystals.Tableaux(['B',2], shape = [1,1])
sage: t = T(rows=[[0],[0]]).to_tableau(); t
[[0], [0]]
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A',Integer(3)], shape = [Integer(2),Integer(2)])
>>> t = T(rows=[[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).to_
↪tableau(); t
[[1, 2], [3, 4]]
>>> type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
>>> type(t[Integer(0)][Integer(0)])
<class 'int'>
>>> T = crystals.Tableaux(['D',Integer(3)], shape = [Integer(1),Integer(1)])
>>> t=T(rows=[[-Integer(3)],[Integer(3)]]).to_tableau(); t
```

(continues on next page)

(continued from previous page)

```
[[-3], [3]]
>>> t=T(rows=[[Integer(3)],[-Integer(3)]]).to_tableau(); t
[[3], [-3]]
>>> T = crystals.Tableaux(['B',Integer(2)], shape = [Integer(1),Integer(1)])
>>> t = T(rows=[[Integer(0)],[Integer(0)]]).to_tableau(); t
[[0], [0]]
```

class sage.combinat.crystals.tensor_product_element.**ImmutableListWithParent**

Bases: `ClonableArray`

A class for lists having a parent.

Specification: any subclass `C` should implement `__init__` which accepts the following form `C(parent, list=list)`

class sage.combinat.crystals.tensor_product_element.**InfinityCrystalOfTableauxElement**

Bases: *CrystalOfTableauxElement*

e(*i*)

Return the action of $\widetilde{e}_i$ on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['B',3])
sage: b = B(rows=[[1,1,1,1,1,1,1,2,0,-3,-1,-1,-1,-1],[2,2,2,2,-2,-2],[3,-3,-
↪3]])
sage: b.e(3).pp()
1  1  1  1  1  1  1  2  0 -3 -1 -1 -1 -1
2  2  2  2 -2 -2
3  0 -3
sage: b.e(1).pp()
1  1  1  1  1  1  1  0 -3 -1 -1 -1 -1
2  2  2  2 -2 -2
3 -3 -3
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['B',Integer(3)])
>>> b = B(rows=[[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(1),Integer(2),Integer(0),-Integer(3),-Integer(1),-
↪Integer(1),-Integer(1),-Integer(1)],[Integer(2),Integer(2),Integer(2),
↪Integer(2),-Integer(2),-Integer(2)],[Integer(3),-Integer(3),-Integer(3)]])
>>> b.e(Integer(3)).pp()
1  1  1  1  1  1  1  2  0 -3 -1 -1 -1 -1
2  2  2  2 -2 -2
3  0 -3
>>> b.e(Integer(1)).pp()
1  1  1  1  1  1  1  0 -3 -1 -1 -1 -1
2  2  2  2 -2 -2
3 -3 -3
```

f(*i*)

Return the action of $\widetilde{f}_i$ on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['C',4])
sage: b = B.highest_weight_vector()
sage: b.f(1).pp()
1  1  1  1  2
2  2  2
3  3
4
sage: b.f(3).pp()
1  1  1  1  1
2  2  2  2
3  3  4
4
sage: b.f(3).f(4).pp()
1  1  1  1  1
2  2  2  2
3  3 -4
4
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['C',Integer(4)])
>>> b = B.highest_weight_vector()
>>> b.f(Integer(1)).pp()
1  1  1  1  2
2  2  2
3  3
4
>>> b.f(Integer(3)).pp()
1  1  1  1  1
2  2  2  2
3  3  4
4
>>> b.f(Integer(3)).f(Integer(4)).pp()
1  1  1  1  1
2  2  2  2
3  3 -4
4
```

class `sage.combinat.crystals.tensor_product_element.`**InfinityCrystalOfTableauxElementTypeD**

Bases: *InfinityCrystalOfTableauxElement*

e(*i*)

Return the action of $\widetilde{e}_i$ on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['D',4])
sage: b = B.highest_weight_vector().f_string([1,4,3,1,2]); b.pp()
1  1  1  1  2  3
2  2  2
3 -3
sage: b.e(2).pp()
1  1  1  1  2  2
2  2  2
3 -3
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['D',Integer(4)])
>>> b = B.highest_weight_vector().f_string([Integer(1),Integer(4),Integer(3),
↪Integer(1),Integer(2)]); b.pp()
1  1  1  1  2  3
2  2  2
3 -3
>>> b.e(Integer(2)).pp()
1  1  1  1  2  2
2  2  2
3 -3
```

f(i)

Return the action of $\widetilde{f}_i$ on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['D',5])
sage: b = B.highest_weight_vector().f_string([1,4,3,1,5]); b.pp()
1  1  1  1  1  1  2  2
2  2  2  2  2
3  3  3 -5
4  5
sage: b.f(1).pp()
1  1  1  1  1  1  2  2  2
2  2  2  2  2
3  3  3 -5
4  5
sage: b.f(5).pp()
1  1  1  1  1  1  2  2
2  2  2  2  2
3  3  3 -5
4 -4
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['D',Integer(5)])
>>> b = B.highest_weight_vector().f_string([Integer(1),Integer(4),Integer(3),
↪Integer(1),Integer(5)]); b.pp()
```

(continues on next page)

(continued from previous page)

```
1  1  1  1  1  1  2  2
2  2  2  2  2
3  3  3 -5
4  5
>>> b.f(Integer(1)).pp()
1  1  1  1  1  1  2  2  2
2  2  2  2  2
3  3  3 -5
4  5
>>> b.f(Integer(5)).pp()
1  1  1  1  1  1  2  2
2  2  2  2  2
3  3  3 -5
4 -4
```

class sage.combinat.crystals.tensor_product_element.**InfinityQueerCrystalOfTableauxElement**

Bases: *TensorProductOfQueerSuperCrystalsElement*

Initialize `self`.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['Q',4])
sage: t = B([[4,4,4,4,2,1],[3,3,3],[2,2],[1]])
sage: t
[[4, 4, 4, 4, 2, 1], [3, 3, 3], [2, 2], [1]]
sage: TestSuite(t).run()
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['Q',Integer(4)])
>>> t = B([[Integer(4),Integer(4),Integer(4),Integer(4),Integer(2),Integer(1)],
↪[Integer(3),Integer(3),Integer(3)],[Integer(2),Integer(2)],[Integer(1)]])
>>> t
[[4, 4, 4, 4, 2, 1], [3, 3, 3], [2, 2], [1]]
>>> TestSuite(t).run()
```

e(*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['Q',4])
sage: t = B([[4,4,4,4,4,2,1],[3,3,3,3],[2,2,1],[1]])
sage: t.e(1)
[[4, 4, 4, 4, 4, 4, 2, 1], [3, 3, 3, 3, 3], [2, 2, 1, 1], [1]]
sage: t.e(3)
[[4, 4, 4, 4, 4, 3, 2, 1], [3, 3, 3, 3], [2, 2, 1], [1]]
sage: t.e(-1)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['Q',Integer(4)])
>>> t = B([[Integer(4),Integer(4),Integer(4),Integer(4),Integer(4),Integer(2),
↪Integer(1)],[Integer(3),Integer(3),Integer(3),Integer(3)],[Integer(2),
↪Integer(2),Integer(1)],[Integer(1)]])
>>> t.e(Integer(1))
[[4, 4, 4, 4, 4, 4, 2, 1], [3, 3, 3, 3, 3], [2, 2, 1, 1], [1]]
>>> t.e(Integer(3))
[[4, 4, 4, 4, 4, 3, 2, 1], [3, 3, 3, 3], [2, 2, 1], [1]]
>>> t.e(-Integer(1))
```

epsilon(*i*)

Return ε_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['Q',4])
sage: t = B([[4,4,4,4,4,2,1],[3,3,3,3],[2,2,1],[1]])
sage: [t.epsilon(i) for i in B.index_set()]
[-1, 1, -2, 0]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['Q',Integer(4)])
>>> t = B([[Integer(4),Integer(4),Integer(4),Integer(4),Integer(4),Integer(2),
↪Integer(1)],[Integer(3),Integer(3),Integer(3),Integer(3)],[Integer(2),
↪Integer(2),Integer(1)],[Integer(1)]])
>>> [t.epsilon(i) for i in B.index_set()]
[-1, 1, -2, 0]
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['Q',4])
sage: t = B([[4,4,4,4,4,2,1],[3,3,3,3],[2,2,1],[1]])
sage: t.f(1)
[[4, 4, 4, 4, 4, 2, 2], [3, 3, 3, 3], [2, 2, 1], [1]]
sage: t.f(3)
sage: t.f(-1)
[[4, 4, 4, 4, 4, 2, 2], [3, 3, 3, 3], [2, 2, 1], [1]]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['Q',Integer(4)])
>>> t = B([[Integer(4),Integer(4),Integer(4),Integer(4),Integer(4),Integer(2),
↪Integer(1)],[Integer(3),Integer(3),Integer(3),Integer(3)],[Integer(2),
↪Integer(2),Integer(1)],[Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> t.f(Integer(1))
[[4, 4, 4, 4, 4, 2, 2], [3, 3, 3, 3], [2, 2, 1], [1]]
>>> t.f(Integer(3))
>>> t.f(-Integer(1))
[[4, 4, 4, 4, 4, 2, 2], [3, 3, 3, 3], [2, 2, 1], [1]]
```

rows()

Return the list of rows of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['Q',4])
sage: t = B([[4,4,4,4,4,2,1],[3,3,3,3],[2,2,1],[1]])
sage: t.rows()
[[1, 2, 4, 4, 4, 4, 4], [3, 3, 3, 3], [1, 2, 2], [1]]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['Q',Integer(4)])
>>> t = B([[Integer(4),Integer(4),Integer(4),Integer(4),Integer(4),Integer(2),
↪Integer(1)],[Integer(3),Integer(3),Integer(3),Integer(3)],[Integer(2),
↪Integer(2),Integer(1)],[Integer(1)]])
>>> t.rows()
[[1, 2, 4, 4, 4, 4, 4], [3, 3, 3, 3], [1, 2, 2], [1]]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['Q',4])
sage: t = B([[4,4,4,4,4,2,1],[3,3,3,3],[2,2,1],[1]])
sage: t.weight()
(4, 2, 2, 0)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux(['Q',Integer(4)])
>>> t = B([[Integer(4),Integer(4),Integer(4),Integer(4),Integer(4),Integer(2),
↪Integer(1)],[Integer(3),Integer(3),Integer(3),Integer(3)],[Integer(2),
↪Integer(2),Integer(1)],[Integer(1)]])
>>> t.weight()
(4, 2, 2, 0)
```

class sage.combinat.crystals.tensor_product_element.**TensorProductOfCrystalsElement**

Bases: *ImmutableListWithParent*

A class for elements of tensor products of crystals.

e(i)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("D4")
sage: T = crystals.TensorProduct(B,B)
sage: b1 = B.highest_weight_vector().f_string([1,4,3])
sage: b2 = B.highest_weight_vector().f_string([2,2,3,1,4])
sage: t = T(b2, b1)
sage: t.e(1)
[[[1, 1, 1, 1, 1], [2, 2, 3, -3], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -3]]]
sage: t.e(2)
sage: t.e(3)
[[[1, 1, 1, 1, 1, 2], [2, 2, 3, -4], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -
↪3]]]
sage: t.e(4)
[[[1, 1, 1, 1, 1, 2], [2, 2, 3, 4], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -
↪3]]]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("D4")
>>> T = crystals.TensorProduct(B,B)
>>> b1 = B.highest_weight_vector().f_string([Integer(1),Integer(4),
↪Integer(3)])
>>> b2 = B.highest_weight_vector().f_string([Integer(2),Integer(2),Integer(3),
↪Integer(1),Integer(4)])
>>> t = T(b2, b1)
>>> t.e(Integer(1))
[[[1, 1, 1, 1, 1], [2, 2, 3, -3], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -3]]]
>>> t.e(Integer(2))
>>> t.e(Integer(3))
[[[1, 1, 1, 1, 1, 2], [2, 2, 3, -4], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -
↪3]]]
>>> t.e(Integer(4))
[[[1, 1, 1, 1, 1, 2], [2, 2, 3, 4], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -
↪3]]]
```

epsilon(*i*)

Return ε_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("G2")
sage: T = crystals.TensorProduct(B,B)
sage: b1 = B.highest_weight_vector().f(2)
sage: b2 = B.highest_weight_vector().f_string([2,2,1])
sage: t = T(b2, b1)
sage: [t.epsilon(i) for i in B.index_set()]
[0, 3]
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("G2")
>>> T = crystals.TensorProduct(B,B)
>>> b1 = B.highest_weight_vector().f(Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> b2 = B.highest_weight_vector().f_string([Integer(2),Integer(2),
↪Integer(1)])
>>> t = T(b2, b1)
>>> [t.epsilon(i) for i in B.index_set()]
[0, 3]
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: La = RootSystem(['A',3,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: B = crystals.GeneralizedYoungWalls(3,La[0])
sage: T = crystals.TensorProduct(B,B,B)
sage: b1 = B.highest_weight_vector().f_string([0,3])
sage: b2 = B.highest_weight_vector().f_string([0])
sage: b3 = B.highest_weight_vector()
sage: t = T(b3, b2, b1)
sage: t.f(0)
[[[0]], [[0]], [[0, 3]]]
sage: t.f(1)
[[], [[0]], [[0, 3], [1]]]
sage: t.f(2)
[[], [[0]], [[0, 3, 2]]]
sage: t.f(3)
[[], [[0, 3]], [[0, 3]]]
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(3),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> B = crystals.GeneralizedYoungWalls(Integer(3),La[Integer(0)])
>>> T = crystals.TensorProduct(B,B,B)
>>> b1 = B.highest_weight_vector().f_string([Integer(0),Integer(3)])
>>> b2 = B.highest_weight_vector().f_string([Integer(0)])
>>> b3 = B.highest_weight_vector()
>>> t = T(b3, b2, b1)
>>> t.f(Integer(0))
[[[0]], [[0]], [[0, 3]]]
>>> t.f(Integer(1))
[[], [[0]], [[0, 3], [1]]]
>>> t.f(Integer(2))
[[], [[0]], [[0, 3, 2]]]
>>> t.f(Integer(3))
[[], [[0, 3]], [[0, 3]]]
```

phi(*i*)

Return φ_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: B = crystals.GeneralizedYoungWalls(2,La[0]+La[1])
sage: T = crystals.TensorProduct(B,B)
sage: b1 = B.highest_weight_vector().f_string([1,0])
sage: b2 = B.highest_weight_vector().f_string([0,1])
sage: t = T(b2, b1)
sage: [t.phi(i) for i in B.index_set()]
[1, 1, 4]
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> B = crystals.GeneralizedYoungWalls(Integer(2),
↪La[Integer(0)]+La[Integer(1)])
>>> T = crystals.TensorProduct(B,B)
>>> b1 = B.highest_weight_vector().f_string([Integer(1),Integer(0)])
>>> b2 = B.highest_weight_vector().f_string([Integer(0),Integer(1)])
>>> t = T(b2, b1)
>>> [t.phi(i) for i in B.index_set()]
[1, 1, 4]
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: C = crystals.Tableaux(['A',3], shape=[3,1])
sage: D = crystals.Tableaux(['A',3], shape=[1])
sage: E = crystals.Tableaux(['A',3], shape=[2,2,2])
sage: T = crystals.TensorProduct(C,D,E)
sage: T.module_generators[0].pp()
  1  1  1 (X)   1 (X)   1  1
  2                     2  2
                        3  3
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',Integer(3)], shape=[Integer(3),Integer(1)])
>>> D = crystals.Tableaux(['A',Integer(3)], shape=[Integer(1)])
>>> E = crystals.Tableaux(['A',Integer(3)], shape=[Integer(2),Integer(2),
↪Integer(2)])
>>> T = crystals.TensorProduct(C,D,E)
>>> T.module_generators[Integer(0)].pp()
  1  1  1 (X)   1 (X)   1  1
  2                     2  2
                        3  3
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("A3")
sage: T = crystals.TensorProduct(B,B)
sage: b1 = B.highest_weight_vector().f_string([2,1,3])
sage: b2 = B.highest_weight_vector().f(1)
sage: t = T(b2, b1)
sage: t
[[[1, 1, 1, 2], [2, 2], [3]], [[1, 1, 1, 1, 2], [2, 2, 4], [3]]]
sage: t.weight()
(-2, 1, 0, 1)
```

```
>>> from sage.all import *
>>> B = crystals.infinity.Tableaux("A3")
>>> T = crystals.TensorProduct(B,B)
>>> b1 = B.highest_weight_vector().f_string([Integer(2),Integer(1),
↪Integer(3)])
>>> b2 = B.highest_weight_vector().f(Integer(1))
>>> t = T(b2, b1)
>>> t
[[[1, 1, 1, 2], [2, 2], [3]], [[1, 1, 1, 1, 2], [2, 2, 4], [3]]]
>>> t.weight()
(-2, 1, 0, 1)
```

```
sage: C = crystals.Letters(['A',3])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(2)).weight()
(1, 1, 0, 0)
sage: T = crystals.Tableaux(['D',4],shape=[])
sage: T.list()[0].weight()
(0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(3)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(1)),C(Integer(2))).weight()
(1, 1, 0, 0)
>>> T = crystals.Tableaux(['D',Integer(4)],shape=[])
>>> T.list()[Integer(0)].weight()
(0, 0, 0, 0)
```

class sage.combinat.crystals.tensor_product_element.**TensorProductOfQueerSuperCrystalsElement**

Bases: *TensorProductOfRegularCrystalsElement*

Element class for a tensor product of crystals for queer Lie superalgebras.

This implements the tensor product rule for crystals of Grantcharov et al. [GJK+2014]. Given crystals B_1 and B_2 of type $\mathfrak{q}_{n+1}$, we define the tensor product $b_1 \otimes b_2 \in B_1 \otimes B_2$, where $b_1 \in B_1$ and $b_2 \in B_2$, as the following:

In addition to the tensor product rule for type A_n, the tensor product rule for e_{-1} and f_{-1} on $b_1 \otimes b_2$ are given by

$$e_{-1}(b_1 \otimes b_2) = \begin{cases} b_1 \otimes e_{-1} b_2 & \text{if } \operatorname{wt}(b_1)_1 = \operatorname{wt}(b_1)_2 = 0, \\ e_{-1} b_1 \otimes b_2 & \text{otherwise}, \end{cases}$$
$$f_{-1}(b_1 \otimes b_2) = \begin{cases} b_1 \otimes f_{-1} b_2 & \text{if } \operatorname{wt}(b_1)_1 = \operatorname{wt}(b_1)_2 = 0, \\ f_{-1} b_1 \otimes b_2 & \text{otherwise}. \end{cases}$$

For $1 < i \leq n$, the operators e_{-i} and f_{-i} are defined as

$$e_{-i} = s_{w_i^{-1}} e_{-1} s_{w_i}, \quad f_{-i} = s_{w_i^{-1}} f_{-1} s_{w_i}.$$

Here, $w_i = s_2 \cdots s_i s_1 \cdots s_{i-1}$ and s_i is the reflection along the i-string in the crystal. Moreover, for $1 < i \leq n$, we define the operators $e_{-i'}$ and $f_{-i'}$ as

$$e_{-i'} = s_{w_0} f_{-(n+1-i)} s_{w_0}, \quad f_{-i'} = s_{w_0} e_{-(n+1-i)} s_{w_0},$$

where w_0 is the longest element in the symmetric group S_{n+1} generated by $s_1, \ldots, s_n$. In this implementation, we use the integers $-2n, \ldots, -(n+1)$ to respectively denote the indices $-n', \ldots, -1'$.

e (*i*)

Return e_i on `self`.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q', 3])
sage: T = tensor([Q,Q])
sage: t = T(Q(1),Q(1))
sage: t.e(-1)
sage: t = T(Q(2),Q(1))
sage: t.e(-1)
[1, 1]

sage: T = tensor([Q,Q,Q,Q])
sage: t = T(Q(1),Q(3),Q(2),Q(1))
sage: t.e(-2)
[2, 2, 1, 1]
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q', Integer(3)])
>>> T = tensor([Q,Q])
>>> t = T(Q(Integer(1)),Q(Integer(1)))
>>> t.e(-Integer(1))
>>> t = T(Q(Integer(2)),Q(Integer(1)))
>>> t.e(-Integer(1))
[1, 1]

>>> T = tensor([Q,Q,Q,Q])
>>> t = T(Q(Integer(1)),Q(Integer(3)),Q(Integer(2)),Q(Integer(1)))
>>> t.e(-Integer(2))
[2, 2, 1, 1]
```

epsilon (*i*)

Return ε_i on `self`.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q', 3])
sage: T = tensor([Q, Q, Q, Q])
sage: t = T(Q(1), Q(3), Q(2), Q(1))
sage: t.epsilon(-2)
1
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q', Integer(3)])
>>> T = tensor([Q, Q, Q, Q])
>>> t = T(Q(Integer(1)), Q(Integer(3)), Q(Integer(2)), Q(Integer(1)))
>>> t.epsilon(-Integer(2))
1
```

f(i)

Return f_i on `self`.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q', 3])
sage: T = tensor([Q, Q])
sage: t = T(Q(1), Q(1))
sage: t.f(-1)
[2, 1]
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q', Integer(3)])
>>> T = tensor([Q, Q])
>>> t = T(Q(Integer(1)), Q(Integer(1)))
>>> t.f(-Integer(1))
[2, 1]
```

phi(i)

Return φ_i on `self`.

EXAMPLES:

```
sage: Q = crystals.Letters(['Q', 3])
sage: T = tensor([Q, Q, Q, Q])
sage: t = T(Q(1), Q(3), Q(2), Q(1))
sage: t.phi(-2)
0
sage: t.phi(-1)
1
```

```
>>> from sage.all import *
>>> Q = crystals.Letters(['Q', Integer(3)])
>>> T = tensor([Q, Q, Q, Q])
>>> t = T(Q(Integer(1)), Q(Integer(3)), Q(Integer(2)), Q(Integer(1)))
>>> t.phi(-Integer(2))
0
>>> t.phi(-Integer(1))
1
```

class sage.combinat.crystals.tensor_product_element.**TensorProductOfRegularCrystalsElement**

Bases: *TensorProductOfCrystalsElement*

Element class for a tensor product of regular crystals.

e(*i*)

Return the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(2)).e(1) == T(C(1),C(1))
True
sage: T(C(2),C(1)).e(1) is None
True
sage: T(C(2),C(2)).e(1) == T(C(1),C(2))
True
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(1)),C(Integer(2))).e(Integer(1)) == T(C(Integer(1)),
↪C(Integer(1)))
True
>>> T(C(Integer(2)),C(Integer(1))).e(Integer(1)) is None
True
>>> T(C(Integer(2)),C(Integer(2))).e(Integer(1)) == T(C(Integer(1)),
↪C(Integer(2)))
True
```

epsilon(*i*)

Return ε_i of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(1)).epsilon(1)
0
sage: T(C(1),C(2)).epsilon(1)
1
sage: T(C(2),C(1)).epsilon(1)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(1)),C(Integer(1))).epsilon(Integer(1))
0
>>> T(C(Integer(1)),C(Integer(2))).epsilon(Integer(1))
1
>>> T(C(Integer(2)),C(Integer(1))).epsilon(Integer(1))
0
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(1)).f(1)
[1, 2]
sage: T(C(1),C(2)).f(1)
[2, 2]
sage: T(C(2),C(1)).f(1) is None
True
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(1)),C(Integer(1))).f(Integer(1))
[1, 2]
>>> T(C(Integer(1)),C(Integer(2))).f(Integer(1))
[2, 2]
>>> T(C(Integer(2)),C(Integer(1))).f(Integer(1)) is None
True
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(1)).phi(1)
2
sage: T(C(1),C(2)).phi(1)
1
sage: T(C(2),C(1)).phi(1)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(1)),C(Integer(1))).phi(Integer(1))
2
>>> T(C(Integer(1)),C(Integer(2))).phi(Integer(1))
1
>>> T(C(Integer(2)),C(Integer(1))).phi(Integer(1))
0
```

position_of_first_unmatched_plus(*i*)

Return the position of the first unmatched $+$ or `None` if there is no unmatched $+$.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(2),C(1)).position_of_first_unmatched_plus(1)
sage: T(C(1),C(2)).position_of_first_unmatched_plus(1)
1
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(2)),C(Integer(1))).position_of_first_unmatched_
↪plus(Integer(1))
>>> T(C(Integer(1)),C(Integer(2))).position_of_first_unmatched_
↪plus(Integer(1))
1
```

position_of_last_unmatched_minus(*i*)

Return the position of the last unmatched $-$ or `None` if there is no unmatched $-$.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(2),C(1)).position_of_last_unmatched_minus(1)
sage: T(C(1),C(2)).position_of_last_unmatched_minus(1)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(2)),C(Integer(1))).position_of_last_unmatched_
↪minus(Integer(1))
>>> T(C(Integer(1)),C(Integer(2))).position_of_last_unmatched_
↪minus(Integer(1))
0
```

positions_of_unmatched_minus(*i*, *dual=False*, *reverse=False*)

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(2),C(1)).positions_of_unmatched_minus(1)
[]
sage: T(C(1),C(2)).positions_of_unmatched_minus(1)
[0]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(2)),C(Integer(1))).positions_of_unmatched_minus(Integer(1))
[]
>>> T(C(Integer(1)),C(Integer(2))).positions_of_unmatched_minus(Integer(1))
[0]
```

positions_of_unmatched_plus(*i*)

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(2),C(1)).positions_of_unmatched_plus(1)
[]
sage: T(C(1),C(2)).positions_of_unmatched_plus(1)
[1]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> T = crystals.TensorProduct(C,C)
>>> T(C(Integer(2)),C(Integer(1))).positions_of_unmatched_plus(Integer(1))
[]
>>> T(C(Integer(1)),C(Integer(2))).positions_of_unmatched_plus(Integer(1))
[1]
```

class sage.combinat.crystals.tensor_product_element.**TensorProductOfSuperCrystalsElement**

Bases: *TensorProductOfRegularCrystalsElement*

Element class for a tensor product of crystals for Lie superalgebras.

This implements the tensor product rule for crystals of Lie superalgebras of [BKK2000].

e(*i*)

Return e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A', [2, 1]])
sage: T = tensor([C,C])
sage: t = T(C(1),C(1))
sage: t.e(0)
[-1, 1]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', [Integer(2), Integer(1)]])
>>> T = tensor([C,C])
>>> t = T(C(Integer(1)),C(Integer(1)))
>>> t.e(Integer(0))
[-1, 1]
```

epsilon(*i*)

Return ε_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A', [2, 1]])
sage: T = tensor([C,C])
sage: t = T(C(1),C(1))
sage: t.epsilon(0)
1
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', [Integer(2), Integer(1)]])
>>> T = tensor([C,C])
>>> t = T(C(Integer(1)),C(Integer(1)))
>>> t.epsilon(Integer(0))
1
```

f (i)

Return f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A', [2, 1]])
sage: T = tensor([C,C])
sage: t = T(C(1),C(1))
sage: t.f(0)
sage: t.f(1)
[1, 2]
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', [Integer(2), Integer(1)]])
>>> T = tensor([C,C])
>>> t = T(C(Integer(1)),C(Integer(1)))
>>> t.f(Integer(0))
>>> t.f(Integer(1))
[1, 2]
```

phi (i)

Return φ_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A', [2, 1]])
sage: T = tensor([C,C])
sage: t = T(C(1),C(1))
sage: t.phi(0)
0
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A', [Integer(2), Integer(1)]])
>>> T = tensor([C,C])
>>> t = T(C(Integer(1)),C(Integer(1)))
>>> t.phi(Integer(0))
0
```

5.1.69 Cyclic sieving phenomenon

Implementation of the Cyclic Sieving Phenomenon as described by Reiner, Stanton, and White in [RSW2004].

We define the *CyclicSievingPolynomial()* of a finite set S together with cyclic action `cyc_act` (of order n) to be the unique polynomial `P(q)` of order $< n$ such that the triple (S, `cyc_act`, `P(q)`) exhibits the cyclic sieving phenomenon.

AUTHORS:

- Christian Stump

REFERENCES:

`sage.combinat.cyclic_sieving_phenomenon.`**`CyclicSievingCheck`**`(`*L*, *cyc_act*, *f*, *order=None*`)`

Return whether the triple `(L, cyc_act, f)` exhibits the cyclic sieving phenomenon.

If `cyc_act` is None, `L` is expected to contain the orbit lengths.

INPUT:

- `L` – if `cyc_act` is `None`: list of orbit sizes, otherwise list of objects
- `cyc_act` – (default: `None`) bijective function from `L` to `L`
- **`order` – (default: `None`) if set to an integer, this**
  cyclic order of `cyc_act` is used (must be an integer multiple of the order of `cyc_act`) otherwise, the order of `cyc_action` is used

EXAMPLES:

```
sage: from sage.combinat.cyclic_sieving_phenomenon import *
sage: from sage.combinat.q_analogues import q_binomial
sage: S42 = Subsets([1,2,3,4], 2)
sage: def cyc_act(S): return Set(i.mod(4) + 1 for i in S)
sage: cyc_act([1,3])
{2, 4}
sage: cyc_act([1,4])
{1, 2}
sage: p = q_binomial(4,2); p
q^4 + q^3 + 2*q^2 + q + 1
sage: CyclicSievingPolynomial( S42, cyc_act )
q^3 + 2*q^2 + q + 2
sage: CyclicSievingCheck( S42, cyc_act, p )
True
```

```
>>> from sage.all import *
>>> from sage.combinat.cyclic_sieving_phenomenon import *
>>> from sage.combinat.q_analogues import q_binomial
>>> S42 = Subsets([Integer(1),Integer(2),Integer(3),Integer(4)], Integer(2))
>>> def cyc_act(S): return Set(i.mod(Integer(4)) + Integer(1) for i in S)
>>> cyc_act([Integer(1),Integer(3)])
{2, 4}
>>> cyc_act([Integer(1),Integer(4)])
{1, 2}
>>> p = q_binomial(Integer(4),Integer(2)); p
q^4 + q^3 + 2*q^2 + q + 1
>>> CyclicSievingPolynomial( S42, cyc_act )
q^3 + 2*q^2 + q + 2
>>> CyclicSievingCheck( S42, cyc_act, p )
True
```

`sage.combinat.cyclic_sieving_phenomenon.`**`CyclicSievingPolynomial`**`(`*L*, *cyc_act=None*, *order=None*, *get_order=False*`)`

Return the unique polynomial `p` of degree smaller than `order` such that the triple `(L, cyc_act, p)` exhibits the Cyclic Sieving Phenomenon.

If `cyc_act` is None, `L` is expected to contain the orbit lengths.

INPUT:

- L – if cyc_act is None: list of orbit sizes, otherwise list of objects
- cyc_act – (default: None) bijective function from L to L
- **order – (default: None) if set to an integer, this**
  cyclic order of cyc_act is used (must be an integer multiple of the order of cyc_act) otherwise, the order of cyc_action is used
- get_order – (default: False) if True, a tuple [p,n] is returned where p is as above, and n is the order

EXAMPLES:

```
sage: from sage.combinat.cyclic_sieving_phenomenon import CyclicSievingPolynomial
sage: S42 = Subsets([1,2,3,4], 2)
sage: def cyc_act(S): return Set(i.mod(4) + 1 for i in S)
sage: cyc_act([1,3])
{2, 4}
sage: cyc_act([1,4])
{1, 2}
sage: CyclicSievingPolynomial(S42, cyc_act)
q^3 + 2*q^2 + q + 2
sage: CyclicSievingPolynomial(S42, cyc_act, get_order=True)
[q^3 + 2*q^2 + q + 2, 4]
sage: CyclicSievingPolynomial(S42, cyc_act, order=8)
q^6 + 2*q^4 + q^2 + 2
sage: CyclicSievingPolynomial([4,2])
q^3 + 2*q^2 + q + 2
```

```
>>> from sage.all import *
>>> from sage.combinat.cyclic_sieving_phenomenon import CyclicSievingPolynomial
>>> S42 = Subsets([Integer(1),Integer(2),Integer(3),Integer(4)], Integer(2))
>>> def cyc_act(S): return Set(i.mod(Integer(4)) + Integer(1) for i in S)
>>> cyc_act([Integer(1),Integer(3)])
{2, 4}
>>> cyc_act([Integer(1),Integer(4)])
{1, 2}
>>> CyclicSievingPolynomial(S42, cyc_act)
q^3 + 2*q^2 + q + 2
>>> CyclicSievingPolynomial(S42, cyc_act, get_order=True)
[q^3 + 2*q^2 + q + 2, 4]
>>> CyclicSievingPolynomial(S42, cyc_act, order=Integer(8))
q^6 + 2*q^4 + q^2 + 2
>>> CyclicSievingPolynomial([Integer(4),Integer(2)])
q^3 + 2*q^2 + q + 2
```

sage.combinat.cyclic_sieving_phenomenon.**orbit_decomposition**(*L*, *cyc_act*)

Return the orbit decomposition of L by the action of cyc_act.

INPUT:

- L – list
- cyc_act – bijective function from L to L

OUTPUT: list of lists, the orbits under the cyc_act acting on L

EXAMPLES:

```
sage: from sage.combinat.cyclic_sieving_phenomenon import *
sage: S42 = Subsets([1,2,3,4], 2); S42
Subsets of {1, 2, 3, 4} of size 2
sage: def cyc_act(S): return Set(i.mod(4) + 1 for i in S)
sage: cyc_act([1,3])
{2, 4}
sage: cyc_act([1,4])
{1, 2}
sage: orbits = orbit_decomposition(S42, cyc_act)
sage: sorted([sorted(orb, key=sorted) for orb in orbits], key=len)
[[{1, 3}, {2, 4}], [{1, 2}, {1, 4}, {2, 3}, {3, 4}]]
```

```
>>> from sage.all import *
>>> from sage.combinat.cyclic_sieving_phenomenon import *
>>> S42 = Subsets([Integer(1),Integer(2),Integer(3),Integer(4)], Integer(2)); S42
Subsets of {1, 2, 3, 4} of size 2
>>> def cyc_act(S): return Set(i.mod(Integer(4)) + Integer(1) for i in S)
>>> cyc_act([Integer(1),Integer(3)])
{2, 4}
>>> cyc_act([Integer(1),Integer(4)])
{1, 2}
>>> orbits = orbit_decomposition(S42, cyc_act)
>>> sorted([sorted(orb, key=sorted) for orb in orbits], key=len)
[[{1, 3}, {2, 4}], [{1, 2}, {1, 4}, {2, 3}, {3, 4}]]
```

5.1.70 De Bruijn sequences

A De Bruijn sequence is defined as the shortest cyclic sequence that incorporates all substrings of a certain length of an alphabet.

For instance, the $2^3 = 8$ binary strings of length 3 are all included in the following string:

```
sage: DeBruijnSequences(2,3).an_element()
[0, 0, 0, 1, 0, 1, 1, 1]
```

```
>>> from sage.all import *
>>> DeBruijnSequences(Integer(2),Integer(3)).an_element()
[0, 0, 0, 1, 0, 1, 1, 1]
```

They can be obtained as a subsequence of the *cyclic* De Bruijn sequence of parameters $k = 2$ and $n = 3$:

```
sage: seq = DeBruijnSequences(2,3).an_element()
sage: print(Word(seq).string_rep())
00010111
sage: shift = lambda i: [(i+j)%2**3 for j in range(3)]
sage: for i in range(2**3):
....:     w = Word([b if j in shift(i) else '*' for j, b in enumerate(seq)])
....:     print(w.string_rep())
000*****
*001****
**010***
***101**
****011*
```

(continues on next page)

(continued from previous page)

```
*****111
0*****11
00*****1
```

```
>>> from sage.all import *
>>> seq = DeBruijnSequences(Integer(2),Integer(3)).an_element()
>>> print(Word(seq).string_rep())
00010111
>>> shift = lambda i: [(i+j)%Integer(2)**Integer(3) for j in range(Integer(3))]
>>> for i in range(Integer(2)**Integer(3)):
...     w = Word([b if j in shift(i) else '*' for j, b in enumerate(seq)])
...     print(w.string_rep())
000*****
*001****
**010***
***101**
****011*
*****111
0*****11
00*****1
```

This sequence is of length k^n, which is best possible as it is the number of k-ary strings of length n. One can equivalently define a De Bruijn sequence of parameters k and n as a cyclic sequence of length k^n in which all substring of length n are different.

See also Wikipedia article De_Bruijn_sequence.

AUTHOR:

- Eviatar Bach (2011): initial version
- Nathann Cohen (2011): Some work on the documentation and defined the `__contain__` method

class sage.combinat.debruijn_sequence.**DeBruijnSequences**(*k*, *n*)

Bases: UniqueRepresentation, Parent

Represent the De Bruijn sequences of given parameters k and n.

A De Bruijn sequence of parameters k and n is defined as the shortest cyclic sequence that incorporates all substrings of length n a k-ary alphabet.

This class can be used to generate the lexicographically smallest De Bruijn sequence, to count the number of existing De Bruijn sequences or to test whether a given sequence is De Bruijn.

INPUT:

- k – a natural number to define arity; the letters used are the integers $0, \ldots, k-1$
- n – a natural number that defines the length of the substring

EXAMPLES:

Obtaining a De Bruijn sequence:

```
sage: seq = DeBruijnSequences(2, 3).an_element()
sage: seq
[0, 0, 0, 1, 0, 1, 1, 1]
```

```
>>> from sage.all import *
>>> seq = DeBruijnSequences(Integer(2), Integer(3)).an_element()
>>> seq
[0, 0, 0, 1, 0, 1, 1, 1]
```

Testing whether it is indeed one:

```
sage: seq in DeBruijnSequences(2, 3)
True
```

```
>>> from sage.all import *
>>> seq in DeBruijnSequences(Integer(2), Integer(3))
True
```

The total number for these parameters:

```
sage: DeBruijnSequences(2, 3).cardinality()
2
```

```
>>> from sage.all import *
>>> DeBruijnSequences(Integer(2), Integer(3)).cardinality()
2
```

Note

This function only generates one De Bruijn sequence (the smallest lexicographically). Support for generating all possible ones may be added in the future.

an_element()

Return the lexicographically smallest De Bruijn sequence with the given parameters.

ALGORITHM:

The algorithm is described in the book “Combinatorial Generation” by Frank Ruskey. This program is based on a Ruby implementation by Jonas Elfström, which is based on the C program by Joe Sadawa.

EXAMPLES:

```
sage: DeBruijnSequences(2, 3).an_element()
[0, 0, 0, 1, 0, 1, 1, 1]
```

```
>>> from sage.all import *
>>> DeBruijnSequences(Integer(2), Integer(3)).an_element()
[0, 0, 0, 1, 0, 1, 1, 1]
```

cardinality()

Return the number of distinct De Bruijn sequences for the object’s parameters.

EXAMPLES:

```
sage: DeBruijnSequences(2, 5).cardinality()
2048
```

```
>>> from sage.all import *
>>> DeBruijnSequences(Integer(2), Integer(5)).cardinality()
2048
```

ALGORITHM:

The formula for cardinality is $k!^{k^{n-1}}/k^n$ [Ros2002].

sage.combinat.debruijn_sequence.**debruijn_sequence**(*k*, *n*)

The generating function for De Bruijn sequences. This avoids the object creation, so is significantly faster than accessing from DeBruijnSequence. For more information, see the documentation there. The algorithm used is from Frank Ruskey's "Combinatorial Generation".

INPUT:

- `k` – arity; must be an integer
- `n` – substring length; must be an integer

EXAMPLES:

```
sage: from sage.combinat.debruijn_sequence import debruijn_sequence
sage: debruijn_sequence(3, 1)
[0, 1, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.debruijn_sequence import debruijn_sequence
>>> debruijn_sequence(Integer(3), Integer(1))
[0, 1, 2]
```

sage.combinat.debruijn_sequence.**is_debruijn_sequence**(*seq*, *k*, *n*)

Given a sequence of integer elements in $0, \ldots, k-1$, tests whether it corresponds to a De Bruijn sequence of parameters k and n.

INPUT:

- `seq` – sequence of elements in $0, \ldots, k-1$
- `n`, `k` – integers

EXAMPLES:

```
sage: from sage.combinat.debruijn_sequence import is_debruijn_sequence
sage: s = DeBruijnSequences(2, 3).an_element()
sage: is_debruijn_sequence(s, 2, 3)
True
sage: is_debruijn_sequence(s + [0], 2, 3)
False
sage: is_debruijn_sequence([1] + s[1:], 2, 3)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.debruijn_sequence import is_debruijn_sequence
>>> s = DeBruijnSequences(Integer(2), Integer(3)).an_element()
>>> is_debruijn_sequence(s, Integer(2), Integer(3))
True
>>> is_debruijn_sequence(s + [Integer(0)], Integer(2), Integer(3))
```

(continues on next page)

(continued from previous page)

```
False
>>> is_debruijn_sequence([Integer(1)] + s[Integer(1):], Integer(2), Integer(3))
False
```

5.1.71 Degree sequences

The present module implements the `DegreeSequences` class, whose instances represent the integer sequences of length n:

```
sage: DegreeSequences(6)
Degree sequences on 6 elements
```

```
>>> from sage.all import *
>>> DegreeSequences(Integer(6))
Degree sequences on 6 elements
```

With the object `DegreeSequences(n)`, one can:

- Check whether a sequence is indeed a degree sequence:

```
sage: DS = DegreeSequences(5)
sage: [4, 3, 3, 3, 3] in DS
True
sage: [4, 4, 0, 0, 0] in DS
False
```

```
>>> from sage.all import *
>>> DS = DegreeSequences(Integer(5))
>>> [Integer(4), Integer(3), Integer(3), Integer(3), Integer(3)] in DS
True
>>> [Integer(4), Integer(4), Integer(0), Integer(0), Integer(0)] in DS
False
```

- List all the possible degree sequences of length n:

```
sage: for seq in DegreeSequences(4):
....:     print(seq)
[0, 0, 0, 0]
[1, 1, 0, 0]
[2, 1, 1, 0]
[3, 1, 1, 1]
[1, 1, 1, 1]
[2, 2, 1, 1]
[2, 2, 2, 0]
[3, 2, 2, 1]
[2, 2, 2, 2]
[3, 3, 2, 2]
[3, 3, 3, 3]
```

```
>>> from sage.all import *
>>> for seq in DegreeSequences(Integer(4)):
...     print(seq)
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0, 0]
[1, 1, 0, 0]
[2, 1, 1, 0]
[3, 1, 1, 1]
[1, 1, 1, 1]
[2, 2, 1, 1]
[2, 2, 2, 0]
[3, 2, 2, 1]
[2, 2, 2, 2]
[3, 3, 2, 2]
[3, 3, 3, 3]
```

Note

Given a degree sequence, one can obtain a graph realizing it by using `DegreeSequence()`. For instance:

```
sage: ds = [3, 3, 2, 2, 2, 2, 2, 1, 1, 0]
sage: g = graphs.DegreeSequence(ds)                                        #␣
↪needs networkx sage.graphs
sage: g.degree_sequence()                                                  #␣
↪needs networkx sage.graphs
[3, 3, 2, 2, 2, 2, 2, 1, 1, 0]
```

```
>>> from sage.all import *
>>> ds = [Integer(3), Integer(3), Integer(2), Integer(2), Integer(2), Integer(2),␣
↪Integer(2), Integer(1), Integer(1), Integer(0)]
>>> g = graphs.DegreeSequence(ds)                                        #␣
↪needs networkx sage.graphs
>>> g.degree_sequence()                                                  #␣
↪needs networkx sage.graphs
[3, 3, 2, 2, 2, 2, 2, 1, 1, 0]
```

Definitions

A sequence of integers $d_1, ..., d_n$ is said to be a *degree sequence* (or *graphic* sequence) if there exists a graph in which vertex i is of degree d_i. It is often required to be *non-increasing*, i.e. that $d_1 \geq ... \geq d_n$. Finding a graph with given degree sequence is known as *graph realization problem*.

An integer sequence need not necessarily be a degree sequence. Indeed, in a degree sequence of length n no integer can be larger than $n-1$ – the degree of a vertex is at most $n-1$ – and the sum of them is at most $n(n-1)$.

Degree sequences are completely characterized by a result from Erdős and Gallai:

Erdős and Gallai: *The sequence of integers* $d_1 \geq \cdots \geq d_n$ *is a degree sequence if and only if* $\sum_i d_i$ is even and $\forall i$

$$\sum_{j\leq i} d_j \leq j(j-1) + \sum_{j>i} \min(d_j, i).$$

Alternatively, a degree sequence can be defined recursively:

Havel and Hakimi: *The sequence of integers* $d_1 \geq ... \geq d_n$ *is a degree sequence if and only if* $d_2 - 1, ..., d_{d_1+1} - 1, d_{d_1+2}, ..., d_n$ *is also a degree sequence.*

Or equivalently:

Havel and Hakimi (bis): *If there is a realization of an integer sequence as a graph (i.e. if the sequence is a degree sequence), then it can be realized in such a way that the vertex of maximum degree Δ is adjacent to the Δ vertices of highest degree (except itself, of course).*

Algorithms

Checking whether a given sequence is a degree sequence

This is tested using Erdos and Gallai's criterion. It is also checked that the given sequence is non-increasing and has length n.

Iterating through the sequences of length n

From Havel and Hakimi's recursive definition of a degree sequence, one can build an enumeration algorithm as done in [RCES1994]. It consists in trying to **extend** a current degree sequence on n elements into a degree sequence on $n+1$ elements by adding a vertex of degree larger than those already present in the sequence. This can be seen as **reversing** the reduction operation described in Havel and Hakimi's characterization. This operation can appear in several different ways:

- Extensions of a degree sequence that do **not** change the value of the maximum element
  - If the maximum element of a given degree sequence is 0, then one can remove it to reduce the sequence, following Havel and Hakimi's rule. Conversely, if the maximum element of the (current) sequence is 0, then one can always extend it by adding a new element of degree 0 to the sequence.

$$0,0,0 \xrightarrow{Extension} \mathbf{0},0,0,0 \xrightarrow{Extension} \mathbf{0},0,0,...,0,0,0 \xrightarrow{Reduction} 0,0,0,0 \xrightarrow{Reduction} 0,0,0$$

  - If there are at least $\Delta+1$ elements of (maximum) degree Δ in a given degree sequence, then one can reduce it by removing a vertex of degree Δ and decreasing the values of Δ elements of value Δ to $\Delta-1$. Conversely, if the maximum element of the (current) sequence is $d>0$, then one can add a new element of degree d to the sequence if it can be linked to d elements of (current) degree $d-1$. Those d vertices of degree $d-1$ hence become vertices of degree d, and so d elements of degree $d-1$ are removed from the sequence while $d+1$ elements of degree d are added to it.

$$3,2,2,2,1 \xrightarrow{Extension} \mathbf{3},3,(2+1),(2+1),(2+1),1 = \mathbf{3},3,3,3,3,1 \xrightarrow{Reduction} 3,2,2,2,1$$

- Extension of a degree sequence that changes the value of the maximum element:
  - In the general case, i.e. when the number of elements of value $\Delta, \Delta-1$ is small compared to Δ (i.e. the maximum element of a given degree sequence), reducing a sequence strictly decreases the value of the maximum element. According to Havel and Hakimi's characterization there is only **one** way to reduce a sequence, but reversing this operation is more complicated than in the previous cases. Indeed, the following extensions are perfectly valid according to the reduction rule.

$$\begin{aligned}
2,1,1,0,0 &\xrightarrow{Extension} \mathbf{3},(2+1),(1+1),(1+1),0,0 = 3,3,2,2,0,0 \xrightarrow{Reduction} 2,1,1,0,0\\
2,1,1,0,0 &\xrightarrow{Extension} \mathbf{3},(2+1),(1+1),1,(0+1),0 = 3,3,2,1,1,0 \xrightarrow{Reduction} 2,1,1,0,0\\
2,1,1,0,0 &\xrightarrow{Extension} \mathbf{3},(2+1),1,1,(0+1),(0+1) = 3,3,1,1,1,1 \xrightarrow{Reduction} 2,1,1,0,0\\
&\ldots
\end{aligned}$$

  In order to extend a current degree sequence while strictly increasing its maximum degree, it is equivalent to pick a set I of elements of the degree sequence with $|I|>\Delta$ in such a way that the $(d_i+1)_{i\in I}$ are the $|I|$ maximum elements of the sequence $(d_i + {}^{1 \text{ if } i\in I}_{0 \text{ if } i\notin I})_{1\leq i\leq n}$, and to add to this new sequence an element of value $|I|$. The non-increasing sequence containing the elements $|I|$ and $(d_i + {}^{1 \text{ if } i\in I}_{0 \text{ if } i\notin I})_{1\leq i\leq n}$ can be reduced to $(d_i)_{1\leq i\leq n}$ by Havel and Hakimi's rule.

$$...1,1,2,\mathbf{2},\mathbf{2},2,2,3,3,\underline{3},\mathbf{3},\mathbf{3},\mathbf{4},\mathbf{6},... \xrightarrow{Extension} ...1,1,2,2,2,3,3,\underline{3},\mathbf{3},\mathbf{3},\mathbf{4},\mathbf{4},\mathbf{5},\mathbf{7},...$$

The number of possible sets I having this property (i.e. the number of possible extensions of a sequence) is smaller than it seems. Indeed, by definition, if $j \notin I$ then for all $i \in I$ the inequality $d_j \leq d_i + 1$ holds. Hence, each set I is entirely determined by the largest element d_k of the sequence that it does **not** contain (hence I contains $\{1, ..., k-1\}$), and by the cardinalities of $\{i \in I : d_i = d_k\}$ and $\{i \in I : d_i = d_k - 1\}$.

$$I = \{i \in I : d_i = d_k\} \cup \{i \in I : d_i = d_k - 1\} \cup \{i : d_i > d_k\}.$$

The number of possible extensions is hence at most cubic, and is easily enumerated.

About the implementation

In the actual implementation of the enumeration algorithm, the degree sequence is stored differently for reasons of efficiency.

Indeed, when enumerating all the degree sequences of length n, Sage first allocates an array `seq` of $n+1$ integers where `seq[i]` is the number of elements of value `i` in the current sequence. Obviously, `seq[n]=0` holds in permanence : it is useful to allocate a larger array than necessary to simplify the code. The `seq` array is a global variable.

The recursive function `enum(depth, maximum)` is the one building the list of sequences. It builds the list of degree sequences of length n which *extend* the sequence currently stored in `seq[0]...seq[depth-1]`. When it is called, `maximum` must be set to the maximum value of an element in the partial sequence `seq[0]...seq[depth-1]`.

If during its run the function `enum` heavily works on the content of the `seq` array, the value of `seq` is the **same** before and after the run of `enum`.

Extending the current partial sequence

The two cases for which the maximum degree of the partial sequence does not change are easy to detect. It is (slightly) harder to enumerate all the sets I corresponding to possible extensions of the partial sequence. As said previously, to each set I one can associate an integer `current_box` such that I contains all the i satisfying $d_i > current\_box$. The variable `taken` represents the number of all such elements i, so that when enumerating all possible sets I in the algorithm we have the equality

$$I = \text{taken} + \text{number of elements of value } current\_box + \text{number of elements of value } current\_box - 1.$$

REFERENCES:

- [RCES1994]

AUTHORS:

- Nathann Cohen

> **Warning**
>
> For the moment, iterating over all degree sequences involves building the list of them first, then iterate on this list. This is obviously bad, as it requires uselessly a **lot** of memory for large values of n.
>
> This should be changed. Updating the code does not require more than a couple of minutes.

class `sage.combinat.degree_sequences.`**`DegreeSequences`**`(n)`

Bases: `object`

Degree Sequences.

An instance of this class represents the degree sequences of graphs on a given number n of vertices. It can be used to list and count them, as well as to test whether a sequence is a degree sequence. For more information, please refer to the documentation of the *`DegreeSequence`* module.

EXAMPLES:

```
sage: DegreeSequences(8)
Degree sequences on 8 elements
sage: [3,3,2,2,2,2,2,2] in DegreeSequences(8)
True
```

```
>>> from sage.all import *
>>> DegreeSequences(Integer(8))
Degree sequences on 8 elements
>>> [Integer(3),Integer(3),Integer(2),Integer(2),Integer(2),Integer(2),Integer(2),
↪Integer(2)] in DegreeSequences(Integer(8))
True
```

5.1.72 Derangements

AUTHORS:

- Alasdair McAndrew (2010-05): Initial version
- Travis Scrimshaw (2013-03-30): Put derangements into category framework

class sage.combinat.derangements.**Derangement**(*parent*, **args*, ***kwds*)

Bases: *CombinatorialElement*

A derangement.

A derangement on a set S is a permutation σ such that $\sigma(x) \neq x$ for all $x \in S$, i.e. σ is a permutation of S with no fixed points.

EXAMPLES:

```
sage: D = Derangements(4)
sage: elt = D([4,3,2,1])
sage: TestSuite(elt).run()
```

```
>>> from sage.all import *
>>> D = Derangements(Integer(4))
>>> elt = D([Integer(4),Integer(3),Integer(2),Integer(1)])
>>> TestSuite(elt).run()
```

to_permutation()

Return the permutation corresponding to `self`.

EXAMPLES:

```
sage: D = Derangements(4)
sage: p = D([4,3,2,1]).to_permutation(); p
[4, 3, 2, 1]
sage: type(p)
<class 'sage.combinat.permutation.StandardPermutations_all_with_category.
↪element_class'>
sage: D = Derangements([1, 3, 3, 4])
sage: D[0].to_permutation()
Traceback (most recent call last):
...
ValueError: can only convert to a permutation for derangements of [1, 2, ...,␣
↪n]
```

```
>>> from sage.all import *
>>> D = Derangements(Integer(4))
>>> p = D([Integer(4),Integer(3),Integer(2),Integer(1)]).to_permutation(); p
[4, 3, 2, 1]
>>> type(p)
<class 'sage.combinat.permutation.StandardPermutations_all_with_category.
→element_class'>
>>> D = Derangements([Integer(1), Integer(3), Integer(3), Integer(4)])
>>> D[Integer(0)].to_permutation()
Traceback (most recent call last):
...
ValueError: can only convert to a permutation for derangements of [1, 2, ...,␣
→n]
```

class sage.combinat.derangements.**Derangements**(*x*)

Bases: UniqueRepresentation, Parent

The class of all derangements of a set or multiset.

A derangement on a set S is a permutation σ such that $\sigma(x) \neq x$ for all $x \in S$, i.e. σ is a permutation of S with no fixed points.

For an integer, or a list or string with all elements distinct, the derangements are obtained by a standard result described in [BV2004]. For a list or string with repeated elements, the derangements are formed by computing all permutations of the input and discarding all non-derangements.

INPUT:

- x – can be an integer which corresponds to derangements of $\{1, 2, 3, \ldots, x\}$, a list, or a string

REFERENCES:

- [BV2004]
- Wikipedia article Derangement

EXAMPLES:

```
sage: D1 = Derangements([2,3,4,5])
sage: D1.list()
[[3, 4, 5, 2],
 [5, 4, 2, 3],
 [3, 5, 2, 4],
 [4, 5, 3, 2],
 [4, 2, 5, 3],
 [5, 2, 3, 4],
 [5, 4, 3, 2],
 [4, 5, 2, 3],
 [3, 2, 5, 4]]
sage: D1.cardinality()
9
sage: D1.random_element() # random
[4, 2, 5, 3]
sage: D2 = Derangements([1,2,3,1,2,3])
sage: D2.cardinality()
10
sage: D2.list()
```

(continues on next page)

(continued from previous page)

```
[[2, 1, 1, 3, 3, 2],
 [2, 1, 2, 3, 3, 1],
 [2, 3, 1, 2, 3, 1],
 [2, 3, 1, 3, 1, 2],
 [2, 3, 2, 3, 1, 1],
 [3, 1, 1, 2, 3, 2],
 [3, 1, 2, 2, 3, 1],
 [3, 1, 2, 3, 1, 2],
 [3, 3, 1, 2, 1, 2],
 [3, 3, 2, 2, 1, 1]]
sage: D2.random_element() # random
[2, 3, 1, 3, 1, 2]
```

```
>>> from sage.all import *
>>> D1 = Derangements([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> D1.list()
[[3, 4, 5, 2],
 [5, 4, 2, 3],
 [3, 5, 2, 4],
 [4, 5, 3, 2],
 [4, 2, 5, 3],
 [5, 2, 3, 4],
 [5, 4, 3, 2],
 [4, 5, 2, 3],
 [3, 2, 5, 4]]
>>> D1.cardinality()
9
>>> D1.random_element() # random
[4, 2, 5, 3]
>>> D2 = Derangements([Integer(1),Integer(2),Integer(3),Integer(1),Integer(2),
↪Integer(3)])
>>> D2.cardinality()
10
>>> D2.list()
[[2, 1, 1, 3, 3, 2],
 [2, 1, 2, 3, 3, 1],
 [2, 3, 1, 2, 3, 1],
 [2, 3, 1, 3, 1, 2],
 [2, 3, 2, 3, 1, 1],
 [3, 1, 1, 2, 3, 2],
 [3, 1, 2, 2, 3, 1],
 [3, 1, 2, 3, 1, 2],
 [3, 3, 1, 2, 1, 2],
 [3, 3, 2, 2, 1, 1]]
>>> D2.random_element() # random
[2, 3, 1, 3, 1, 2]
```

Element

alias of `Derangement`

cardinality()

Counts the number of derangements of a positive integer, a list, or a string. The list or string may contain repeated elements. If an integer n is given, the value returned is the number of derangements of $[1, 2, 3, \ldots, n]$.

For an integer, or a list or string with all elements distinct, the value is obtained by the standard result $D_2 = 1, D_3 = 2, D_n = (n-1)(D_{n-1} + D_{n-2})$.

For a list or string with repeated elements, the number of derangements is computed by Macmahon's theorem. If the numbers of repeated elements are $a_1, a_2, \ldots, a_k$ then the number of derangements is given by the coefficient of $x_1 x_2 \cdots x_k$ in the expansion of $\prod_{i=0}^{k} (S - s_i)^{a_i}$ where $S = x_1 + x_2 + \cdots + x_k$.

EXAMPLES:

```
sage: D = Derangements(5)
sage: D.cardinality()
44
sage: D = Derangements([1,44,918,67,254])
sage: D.cardinality()
44
sage: D = Derangements(['A','AT','CAT','CATS','CARTS'])
sage: D.cardinality()
44
sage: D = Derangements('UNCOPYRIGHTABLE')
sage: D.cardinality()
481066515734
sage: D = Derangements([1,1,2,2,3,3])
sage: D.cardinality()
10
sage: D = Derangements('SATTAS')
sage: D.cardinality()
10
sage: D = Derangements([1,1,2,2,2])
sage: D.cardinality()
0
```

```
>>> from sage.all import *
>>> D = Derangements(Integer(5))
>>> D.cardinality()
44
>>> D = Derangements([Integer(1),Integer(44),Integer(918),Integer(67),
↪Integer(254)])
>>> D.cardinality()
44
>>> D = Derangements(['A','AT','CAT','CATS','CARTS'])
>>> D.cardinality()
44
>>> D = Derangements('UNCOPYRIGHTABLE')
>>> D.cardinality()
481066515734
>>> D = Derangements([Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
↪Integer(3)])
>>> D.cardinality()
10
>>> D = Derangements('SATTAS')
>>> D.cardinality()
10
>>> D = Derangements([Integer(1),Integer(1),Integer(2),Integer(2),Integer(2)])
>>> D.cardinality()
```

(continues on next page)

(continued from previous page)

```
0
```

random_element()

Produce all derangements of a positive integer, a list, or a string. The list or string may contain repeated elements. If an integer n is given, then a random derangements of $[1, 2, 3, \dots, n]$ is returned

For an integer, or a list or string with all elements distinct, the value is obtained by an algorithm described in [MPP2008]. For a list or string with repeated elements the derangement is formed by choosing an element at random from the list of all possible derangements.

OUTPUT:

A single list or string containing a derangement, or an empty list if there are no derangements.

EXAMPLES:

```
sage: D = Derangements(4)
sage: D.random_element() # random
[2, 3, 4, 1]
sage: D = Derangements(['A','AT','CAT','CATS','CARTS','CARETS'])
sage: D.random_element() # random
['AT', 'CARTS', 'A', 'CAT', 'CARETS', 'CATS']
sage: D = Derangements('UNCOPYRIGHTABLE')
sage: D.random_element() # random
['C', 'U', 'I', 'H', 'O', 'G', 'N', 'B', 'E', 'L', 'A', 'R', 'P', 'Y', 'T']
sage: D = Derangements([1,1,1,1,2,2,2,2,3,3,3,3])
sage: D.random_element() # random
[3, 2, 2, 3, 1, 3, 1, 3, 2, 1, 1, 2]
sage: D = Derangements('ESSENCES')
sage: D.random_element() # random
['N', 'E', 'E', 'C', 'S', 'S', 'S', 'E']
sage: D = Derangements([1,1,2,2,2])
sage: D.random_element()
[]
```

```
>>> from sage.all import *
>>> D = Derangements(Integer(4))
>>> D.random_element() # random
[2, 3, 4, 1]
>>> D = Derangements(['A','AT','CAT','CATS','CARTS','CARETS'])
>>> D.random_element() # random
['AT', 'CARTS', 'A', 'CAT', 'CARETS', 'CATS']
>>> D = Derangements('UNCOPYRIGHTABLE')
>>> D.random_element() # random
['C', 'U', 'I', 'H', 'O', 'G', 'N', 'B', 'E', 'L', 'A', 'R', 'P', 'Y', 'T']
>>> D = Derangements([Integer(1),Integer(1),Integer(1),Integer(1),Integer(2),
↪Integer(2),Integer(2),Integer(2),Integer(3),Integer(3),Integer(3),
↪Integer(3)])
>>> D.random_element() # random
[3, 2, 2, 3, 1, 3, 1, 3, 2, 1, 1, 2]
>>> D = Derangements('ESSENCES')
>>> D.random_element() # random
['N', 'E', 'E', 'C', 'S', 'S', 'S', 'E']
>>> D = Derangements([Integer(1),Integer(1),Integer(2),Integer(2),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> D.random_element()
[]
```

5.1.73 Descent Algebras

AUTHORS:

- Travis Scrimshaw (2013-07-28): Initial version

class sage.combinat.descent_algebra.**DescentAlgebra**(*R*, *n*)

Bases: UniqueRepresentation, Parent

Solomon's descent algebra.

The descent algebra Σ_n over a ring R is a subalgebra of the symmetric group algebra RS_n. (The product in the latter algebra is defined by $(pq)(i) = q(p(i))$ for any two permutations p and q in S_n and every $i \in \{1, 2, \ldots, n\}$. The algebra Σ_n inherits this product.)

There are three bases currently implemented for Σ_n:

- the standard basis D_S of (sums of) descent classes, indexed by subsets S of $\{1, 2, \ldots, n-1\}$,
- the subset basis B_p, indexed by compositions p of n,
- the idempotent basis I_p, indexed by compositions p of n, which is used to construct the mutually orthogonal idempotents of the symmetric group algebra.

The idempotent basis is only defined when R is a $\mathbf{Q}$-algebra.

We follow the notations and conventions in [GR1989], apart from the order of multiplication being different from the one used in that article. Schocker's exposition [Sch2004], in turn, uses the same order of multiplication as we are, but has different notations for the bases.

INPUT:

- R – the base ring
- n – nonnegative integer

REFERENCES:

- [GR1989]
- [At1992]
- [MR1995]
- [Sch2004]

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D(); D
Descent algebra of 4 over Rational Field in the standard basis
sage: B = DA.B(); B
Descent algebra of 4 over Rational Field in the subset basis
sage: I = DA.I(); I
Descent algebra of 4 over Rational Field in the idempotent basis
sage: basis_B = B.basis()
sage: elt = basis_B[Composition([1,2,1])] + 4*basis_B[Composition([1,3])]; elt
B[1, 2, 1] + 4*B[1, 3]
```

(continues on next page)

(continued from previous page)

```
sage: D(elt)
5*D{} + 5*D{1} + D{1, 3} + D{3}
sage: I(elt)
7/6*I[1, 1, 1, 1] + 2*I[1, 1, 2] + 3*I[1, 2, 1] + 4*I[1, 3]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> D = DA.D(); D
Descent algebra of 4 over Rational Field in the standard basis
>>> B = DA.B(); B
Descent algebra of 4 over Rational Field in the subset basis
>>> I = DA.I(); I
Descent algebra of 4 over Rational Field in the idempotent basis
>>> basis_B = B.basis()
>>> elt = basis_B[Composition([Integer(1),Integer(2),Integer(1)])] +␣
↪Integer(4)*basis_B[Composition([Integer(1),Integer(3)])]; elt
B[1, 2, 1] + 4*B[1, 3]
>>> D(elt)
5*D{} + 5*D{1} + D{1, 3} + D{3}
>>> I(elt)
7/6*I[1, 1, 1, 1] + 2*I[1, 1, 2] + 3*I[1, 2, 1] + 4*I[1, 3]
```

As syntactic sugar, one can use the notation `D[i,...,l]` to construct elements of the basis; note that for the empty set one must use `D[[]]` due to Python's syntax:

```
sage: D[[]] + D[2] + 2*D[1,2]
D{} + 2*D{1, 2} + D{2}
```

```
>>> from sage.all import *
>>> D[[]] + D[Integer(2)] + Integer(2)*D[Integer(1),Integer(2)]
D{} + 2*D{1, 2} + D{2}
```

The same syntax works for the other bases:

```
sage: I[1,2,1] + 3*I[4] + 2*I[3,1]
I[1, 2, 1] + 2*I[3, 1] + 3*I[4]
```

```
>>> from sage.all import *
>>> I[Integer(1),Integer(2),Integer(1)] + Integer(3)*I[Integer(4)] +␣
↪Integer(2)*I[Integer(3),Integer(1)]
I[1, 2, 1] + 2*I[3, 1] + 3*I[4]
```

class B(*alg*, *prefix='B'*)

Bases: `CombinatorialFreeModule`, `BindableClass`

The subset basis of a descent algebra (indexed by compositions).

The subset basis $(B_S)_{S\subseteq\{1,2,\ldots,n-1\}}$ of Σ_n is formed by

$$B_S = \sum_{T\subseteq S} D_T,$$

where $(D_S)_{S\subseteq\{1,2,\ldots,n-1\}}$ is the *standard basis*. However it is more natural to index the subset basis by compositions of n under the bijection $\{i_1, i_2, \ldots, i_k\} \mapsto (i_1, i_2 - i_1, i_3 - i_2, \ldots, i_k - i_{k-1}, n - i_k)$ (where $i_1 < i_2 < \cdots < i_k$), which is what Sage uses to index the basis.

The basis element B_p is denoted Ξ^p in [Sch2004].

By using compositions of n, the product $B_p B_q$ becomes a sum over the nonnegative-integer matrices M with row sum p and column sum q. The summand corresponding to M is B_c, where c is the composition obtained by reading M row-by-row from left-to-right and top-to-bottom and removing all zeroes. This multiplication rule is commonly called "Solomon's Mackey formula".

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: list(B.basis())
[B[1, 1, 1, 1], B[1, 1, 2], B[1, 2, 1], B[1, 3],
 B[2, 1, 1], B[2, 2], B[3, 1], B[4]]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> list(B.basis())
[B[1, 1, 1, 1], B[1, 1, 2], B[1, 2, 1], B[1, 3],
 B[2, 1, 1], B[2, 2], B[3, 1], B[4]]
```

one_basis()

Return the identity element which is the composition $[n]$, as per `AlgebrasWithBasis.ParentMethods.one_basis`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).B().one_basis()
[4]
sage: DescentAlgebra(QQ, 0).B().one_basis()
[]

sage: all( U * DescentAlgebra(QQ, 3).B().one() == U
....:      for U in DescentAlgebra(QQ, 3).B().basis() )
True
```

```
>>> from sage.all import *
>>> DescentAlgebra(QQ, Integer(4)).B().one_basis()
[4]
>>> DescentAlgebra(QQ, Integer(0)).B().one_basis()
[]

>>> all( U * DescentAlgebra(QQ, Integer(3)).B().one() == U
...      for U in DescentAlgebra(QQ, Integer(3)).B().basis() )
True
```

product_on_basis(p, q)

Return $B_p B_q$, where p and q are compositions of n.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: p = Composition([1,2,1])
sage: q = Composition([3,1])
sage: B.product_on_basis(p, q)
B[1, 1, 1, 1] + 2*B[1, 2, 1]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> p = Composition([Integer(1),Integer(2),Integer(1)])
>>> q = Composition([Integer(3),Integer(1)])
>>> B.product_on_basis(p, q)
B[1, 1, 1, 1] + 2*B[1, 2, 1]
```

to_D_basis(*p*)

Return B_p as a linear combination of D-basis elements.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: D = DA.D()
sage: list(map(D, B.basis()))  # indirect doctest
[D{} + D{1} + D{1, 2} + D{1, 2, 3}
  + D{1, 3} + D{2} + D{2, 3} + D{3},
 D{} + D{1} + D{1, 2} + D{2},
 D{} + D{1} + D{1, 3} + D{3},
 D{} + D{1},
 D{} + D{2} + D{2, 3} + D{3},
 D{} + D{2},
 D{} + D{3},
 D{}]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> D = DA.D()
>>> list(map(D, B.basis()))  # indirect doctest
[D{} + D{1} + D{1, 2} + D{1, 2, 3}
  + D{1, 3} + D{2} + D{2, 3} + D{3},
 D{} + D{1} + D{1, 2} + D{2},
 D{} + D{1} + D{1, 3} + D{3},
 D{} + D{1},
 D{} + D{2} + D{2, 3} + D{3},
 D{} + D{2},
 D{} + D{3},
 D{}]
```

to_I_basis(*p*)

Return B_p as a linear combination of I-basis elements.

This is done using the formula

$$B_p = \sum_{q \leq p} \frac{1}{\mathbf{k}!(q,p)} I_q,$$

where $\leq$ is the refinement order and $\mathbf{k}!(q,p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i-th entry of p, and then we set $\mathbf{k}!(q,p)$ to be $l(q_{(1)})!l(q_{(2)})! \cdots l(q_{(k)})!$, where $l(r)$ denotes the number of parts of any composition r.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: I = DA.I()
sage: list(map(I, B.basis()))  # indirect doctest
[I[1, 1, 1, 1],
 1/2*I[1, 1, 1, 1] + I[1, 1, 2],
 1/2*I[1, 1, 1, 1] + I[1, 2, 1],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[1, 2, 1] + I[1, 3],
 1/2*I[1, 1, 1, 1] + I[2, 1, 1],
 1/4*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[2, 1, 1] + I[2, 2],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 2, 1] + 1/2*I[2, 1, 1] + I[3, 1],
 1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
  + 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2] + 1/2*I[3, 1] + I[4]]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> I = DA.I()
>>> list(map(I, B.basis()))  # indirect doctest
[I[1, 1, 1, 1],
 1/2*I[1, 1, 1, 1] + I[1, 1, 2],
 1/2*I[1, 1, 1, 1] + I[1, 2, 1],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[1, 2, 1] + I[1, 3],
 1/2*I[1, 1, 1, 1] + I[2, 1, 1],
 1/4*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[2, 1, 1] + I[2, 2],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 2, 1] + 1/2*I[2, 1, 1] + I[3, 1],
 1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
  + 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2] + 1/2*I[3, 1] + I[4]]
```

to_nsym(*p*)

Return B_p as an element in $NSym$, the non-commutative symmetric functions.

This maps B_p to S_p where S denotes the Complete basis of $NSym$.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 4).B()
sage: S = NonCommutativeSymmetricFunctions(QQ).Complete()
sage: list(map(S, B.basis()))  # indirect doctest
[S[1, 1, 1, 1],
 S[1, 1, 2],
 S[1, 2, 1],
 S[1, 3],
 S[2, 1, 1],
```

(continues on next page)

(continued from previous page)

```
 S[2, 2],
 S[3, 1],
 S[4]]
```

```
>>> from sage.all import *
>>> B = DescentAlgebra(QQ, Integer(4)).B()
>>> S = NonCommutativeSymmetricFunctions(QQ).Complete()
>>> list(map(S, B.basis()))   # indirect doctest
[S[1, 1, 1, 1],
 S[1, 1, 2],
 S[1, 2, 1],
 S[1, 3],
 S[2, 1, 1],
 S[2, 2],
 S[3, 1],
 S[4]]
```

class D(*alg*, *prefix='D'*)

Bases: *CombinatorialFreeModule*, BindableClass

The standard basis of a descent algebra.

This basis is indexed by $S \subseteq \{1, 2, \ldots, n-1\}$, and the basis vector indexed by S is the sum of all permutations, taken in the symmetric group algebra RS_n, whose descent set is S. We denote this basis vector by D_S.

Occasionally this basis appears in literature but indexed by compositions of n rather than subsets of $\{1, 2, \ldots, n-1\}$. The equivalence between these two indexings is owed to the bijection from the power set of $\{1, 2, \ldots, n-1\}$ to the set of all compositions of n which sends every subset $\{i_1, i_2, \ldots, i_k\}$ of $\{1, 2, \ldots, n-1\}$ (with $i_1 < i_2 < \cdots < i_k$) to the composition $(i_1, i_2 - i_1, \ldots, i_k - i_{k-1}, n - i_k)$.

The basis element corresponding to a composition p (or to the subset of $\{1, 2, \ldots, n-1\}$) is denoted Δ^p in [Sch2004].

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: list(D.basis())
[D{}, D{1}, D{2}, D{3}, D{1, 2}, D{1, 3}, D{2, 3}, D{1, 2, 3}]

sage: DA = DescentAlgebra(QQ, 0)
sage: D = DA.D()
sage: list(D.basis())
[D{}]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> D = DA.D()
>>> list(D.basis())
[D{}, D{1}, D{2}, D{3}, D{1, 2}, D{1, 3}, D{2, 3}, D{1, 2, 3}]

>>> DA = DescentAlgebra(QQ, Integer(0))
>>> D = DA.D()
```

(continues on next page)

(continued from previous page)

```
>>> list(D.basis())
[D{}]
```

one_basis()

Return the identity element, as per `AlgebrasWithBasis.ParentMethods.one_basis`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).D().one_basis()
()
sage: DescentAlgebra(QQ, 0).D().one_basis()
()

sage: all( U * DescentAlgebra(QQ, 3).D().one() == U
....:      for U in DescentAlgebra(QQ, 3).D().basis() )
True
```

```
>>> from sage.all import *
>>> DescentAlgebra(QQ, Integer(4)).D().one_basis()
()
>>> DescentAlgebra(QQ, Integer(0)).D().one_basis()
()

>>> all( U * DescentAlgebra(QQ, Integer(3)).D().one() == U
...      for U in DescentAlgebra(QQ, Integer(3)).D().basis() )
True
```

product_on_basis(S, T)

Return $D_S D_T$, where S and T are subsets of $[n-1]$.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: D.product_on_basis((1, 3), (2,))
D{} + D{1} + D{1, 2} + 2*D{1, 2, 3} + D{1, 3} + D{2} + D{2, 3} + D{3}
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> D = DA.D()
>>> D.product_on_basis((Integer(1), Integer(3)), (Integer(2),))
D{} + D{1} + D{1, 2} + 2*D{1, 2, 3} + D{1, 3} + D{2} + D{2, 3} + D{3}
```

to_B_basis(S)

Return D_S as a linear combination of B_p-basis elements.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: B = DA.B()
sage: list(map(B, D.basis()))  # indirect doctest
[B[4],
```

(continues on next page)

(continued from previous page)

```
 B[1, 3] - B[4],
 B[2, 2] - B[4],
 B[3, 1] - B[4],
 B[1, 1, 2] - B[1, 3] - B[2, 2] + B[4],
 B[1, 2, 1] - B[1, 3] - B[3, 1] + B[4],
 B[2, 1, 1] - B[2, 2] - B[3, 1] + B[4],
 B[1, 1, 1, 1] - B[1, 1, 2] - B[1, 2, 1] + B[1, 3]
  - B[2, 1, 1] + B[2, 2] + B[3, 1] - B[4]]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> D = DA.D()
>>> B = DA.B()
>>> list(map(B, D.basis()))   # indirect doctest
[B[4],
 B[1, 3] - B[4],
 B[2, 2] - B[4],
 B[3, 1] - B[4],
 B[1, 1, 2] - B[1, 3] - B[2, 2] + B[4],
 B[1, 2, 1] - B[1, 3] - B[3, 1] + B[4],
 B[2, 1, 1] - B[2, 2] - B[3, 1] + B[4],
 B[1, 1, 1, 1] - B[1, 1, 2] - B[1, 2, 1] + B[1, 3]
  - B[2, 1, 1] + B[2, 2] + B[3, 1] - B[4]]
```

to_symmetric_group_algebra_on_basis(S)

Return D_S as a linear combination of basis elements in the symmetric group algebra.

EXAMPLES:

```
sage: D = DescentAlgebra(QQ, 4).D()
sage: [D.to_symmetric_group_algebra_on_basis(tuple(b))
....:  for b in Subsets(3)]
[[1, 2, 3, 4],
 [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3],
 [1, 3, 2, 4] + [1, 4, 2, 3] + [2, 3, 1, 4]
  + [2, 4, 1, 3] + [3, 4, 1, 2],
 [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 3, 4, 1],
 [3, 2, 1, 4] + [4, 2, 1, 3] + [4, 3, 1, 2],
 [2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1]
  + [4, 1, 3, 2] + [4, 2, 3, 1],
 [1, 4, 3, 2] + [2, 4, 3, 1] + [3, 4, 2, 1],
 [4, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> D = DescentAlgebra(QQ, Integer(4)).D()
>>> [D.to_symmetric_group_algebra_on_basis(tuple(b))
...  for b in Subsets(Integer(3))]
[[1, 2, 3, 4],
 [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3],
 [1, 3, 2, 4] + [1, 4, 2, 3] + [2, 3, 1, 4]
  + [2, 4, 1, 3] + [3, 4, 1, 2],
 [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 3, 4, 1],
```

(continues on next page)

(continued from previous page)

```
[3, 2, 1, 4] + [4, 2, 1, 3] + [4, 3, 1, 2],
[2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1]
 + [4, 1, 3, 2] + [4, 2, 3, 1],
[1, 4, 3, 2] + [2, 4, 3, 1] + [3, 4, 2, 1],
[4, 3, 2, 1]]
```

class I(*alg*, *prefix='I'*)

Bases: `CombinatorialFreeModule`, `BindableClass`

The idempotent basis of a descent algebra.

The idempotent basis $(I_p)_{p \models n}$ is a basis for Σ_n whenever the ground ring is a $\mathbf{Q}$-algebra. One way to compute it is using the formula (Theorem 3.3 in [GR1989])

$$I_p = \sum_{q \leq p} \frac{(-1)^{l(q)-l(p)}}{\mathbf{k}(q,p)} B_q,$$

where $\leq$ is the refinement order and $l(r)$ denotes the number of parts of any composition r, and where $\mathbf{k}(q,p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)} q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i-th entry of p, and then we set $\mathbf{k}(q,p)$ to be the product $l(q_{(1)}) l(q_{(2)}) \cdots l(q_{(k)})$.

Let $\lambda(p)$ denote the partition obtained from a composition p by sorting. This basis is called the idempotent basis since for any q such that $\lambda(p) = \lambda(q)$, we have:

$$I_p I_q = s(\lambda) I_p$$

where λ denotes $\lambda(p) = \lambda(q)$, and where $s(\lambda)$ is the stabilizer of λ in S_n. (This is part of Theorem 4.2 in [GR1989].)

It is also straightforward to compute the idempotents E_λ for the symmetric group algebra by the formula (Theorem 3.2 in [GR1989]):

$$E_\lambda = \frac{1}{k!} \sum_{\lambda(p)=\lambda} I_p.$$

Note

The basis elements are not orthogonal idempotents.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: I = DA.I()
sage: list(I.basis())
[I[1, 1, 1, 1], I[1, 1, 2], I[1, 2, 1], I[1, 3], I[2, 1, 1], I[2, 2], I[3, 1],
↪ I[4]]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> I = DA.I()
>>> list(I.basis())
[I[1, 1, 1, 1], I[1, 1, 2], I[1, 2, 1], I[1, 3], I[2, 1, 1], I[2, 2], I[3, 1],
↪ I[4]]
```

idempotent (*la*)

Return the idempotent corresponding to the partition `la` of n.

EXAMPLES:

```
sage: I = DescentAlgebra(QQ, 4).I()
sage: E = I.idempotent([3,1]); E
1/2*I[1, 3] + 1/2*I[3, 1]
sage: E*E == E
True
sage: E2 = I.idempotent([2,1,1]); E2
1/6*I[1, 1, 2] + 1/6*I[1, 2, 1] + 1/6*I[2, 1, 1]
sage: E2*E2 == E2
True
sage: E*E2 == I.zero()
True
```

```
>>> from sage.all import *
>>> I = DescentAlgebra(QQ, Integer(4)).I()
>>> E = I.idempotent([Integer(3),Integer(1)]); E
1/2*I[1, 3] + 1/2*I[3, 1]
>>> E*E == E
True
>>> E2 = I.idempotent([Integer(2),Integer(1),Integer(1)]); E2
1/6*I[1, 1, 2] + 1/6*I[1, 2, 1] + 1/6*I[2, 1, 1]
>>> E2*E2 == E2
True
>>> E*E2 == I.zero()
True
```

one ()

Return the identity element, which is $B_{[n]}$, in the I basis.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).I().one()
1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
 + 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2]
 + 1/2*I[3, 1] + I[4]
sage: DescentAlgebra(QQ, 0).I().one()
I[]
```

```
>>> from sage.all import *
>>> DescentAlgebra(QQ, Integer(4)).I().one()
1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
 + 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2]
 + 1/2*I[3, 1] + I[4]
>>> DescentAlgebra(QQ, Integer(0)).I().one()
I[]
```

one_basis ()

The element 1 is not (generally) a basis vector in the I basis, thus this raises a `TypeError`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).I().one_basis()
Traceback (most recent call last):
...
TypeError: 1 is not a basis element in the I basis
```

```
>>> from sage.all import *
>>> DescentAlgebra(QQ, Integer(4)).I().one_basis()
Traceback (most recent call last):
...
TypeError: 1 is not a basis element in the I basis
```

product_on_basis(*p*, *q*)

Return $I_p I_q$, where p and q are compositions of n.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: I = DA.I()
sage: p = Composition([1,2,1])
sage: q = Composition([3,1])
sage: I.product_on_basis(p, q)
0
sage: I.product_on_basis(p, p)
2*I[1, 2, 1]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> I = DA.I()
>>> p = Composition([Integer(1),Integer(2),Integer(1)])
>>> q = Composition([Integer(3),Integer(1)])
>>> I.product_on_basis(p, q)
0
>>> I.product_on_basis(p, p)
2*I[1, 2, 1]
```

to_B_basis(*p*)

Return I_p as a linear combination of B-basis elements.

This is computed using the formula (Theorem 3.3 in [GR1989])

$$I_p = \sum_{q \leq p} \frac{(-1)^{l(q)-l(p)}}{\mathbf{k}(q,p)} B_q,$$

where $\leq$ is the refinement order and $l(r)$ denotes the number of parts of any composition r, and where $\mathbf{k}(q,p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)} q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i-th entry of p, and then we set $\mathbf{k}(q,p)$ to be $l(q_{(1)}) l(q_{(2)}) \cdots l(q_{(k)})$.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: I = DA.I()
sage: list(map(B, I.basis()))   # indirect doctest
[B[1, 1, 1, 1],
```

(continues on next page)

(continued from previous page)

```
 -1/2*B[1, 1, 1, 1] + B[1, 1, 2],
 -1/2*B[1, 1, 1, 1] + B[1, 2, 1],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[1, 2, 1] + B[1, 3],
 -1/2*B[1, 1, 1, 1] + B[2, 1, 1],
 1/4*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[2, 1, 1] + B[2, 2],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 2, 1] - 1/2*B[2, 1, 1] + B[3, 1],
 -1/4*B[1, 1, 1, 1] + 1/3*B[1, 1, 2] + 1/3*B[1, 2, 1]
  - 1/2*B[1, 3] + 1/3*B[2, 1, 1] - 1/2*B[2, 2]
  - 1/2*B[3, 1] + B[4]]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> I = DA.I()
>>> list(map(B, I.basis()))  # indirect doctest
[B[1, 1, 1, 1],
 -1/2*B[1, 1, 1, 1] + B[1, 1, 2],
 -1/2*B[1, 1, 1, 1] + B[1, 2, 1],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[1, 2, 1] + B[1, 3],
 -1/2*B[1, 1, 1, 1] + B[2, 1, 1],
 1/4*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[2, 1, 1] + B[2, 2],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 2, 1] - 1/2*B[2, 1, 1] + B[3, 1],
 -1/4*B[1, 1, 1, 1] + 1/3*B[1, 1, 2] + 1/3*B[1, 2, 1]
  - 1/2*B[1, 3] + 1/3*B[2, 1, 1] - 1/2*B[2, 2]
  - 1/2*B[3, 1] + B[4]]
```

a_realization()

Return a particular realization of `self` (the B-basis).

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: DA.a_realization()
Descent algebra of 4 over Rational Field in the subset basis
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> DA.a_realization()
Descent algebra of 4 over Rational Field in the subset basis
```

idempotent

alias of `I`

standard

alias of `D`

subset

alias of `B`

class sage.combinat.descent_algebra.**DescentAlgebraBases**(*base*)

Bases: `Category_realization_of_parent`

The category of bases of a descent algebra.

class ElementMethods

Bases: object

to_symmetric_group_algebra()

Return self in the symmetric group algebra.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 4).B()
sage: B[1,3].to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
sage: I = DescentAlgebra(QQ, 4).I()
sage: elt = I(B[1,3])
sage: elt.to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
```

```
>>> from sage.all import *
>>> B = DescentAlgebra(QQ, Integer(4)).B()
>>> B[Integer(1),Integer(3)].to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
>>> I = DescentAlgebra(QQ, Integer(4)).I()
>>> elt = I(B[Integer(1),Integer(3)])
>>> elt.to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
```

class ParentMethods

Bases: object

to_symmetric_group_algebra()

Morphism from self to the symmetric group algebra.

EXAMPLES:

```
sage: D = DescentAlgebra(QQ, 4).D()
sage: D.to_symmetric_group_algebra(D[1,3])
[2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1] + [4, 1, 3, 2] + [4, 2, 3, 1]
sage: B = DescentAlgebra(QQ, 4).B()
sage: B.to_symmetric_group_algebra(B[1,2,1])
[1, 2, 3, 4] + [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 1, 3, 4]
 + [2, 1, 4, 3] + [2, 3, 4, 1] + [3, 1, 2, 4] + [3, 1, 4, 2]
 + [3, 2, 4, 1] + [4, 1, 2, 3] + [4, 1, 3, 2] + [4, 2, 3, 1]
```

```
>>> from sage.all import *
>>> D = DescentAlgebra(QQ, Integer(4)).D()
>>> D.to_symmetric_group_algebra(D[Integer(1),Integer(3)])
[2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1] + [4, 1, 3, 2] + [4, 2, 3, 1]
>>> B = DescentAlgebra(QQ, Integer(4)).B()
>>> B.to_symmetric_group_algebra(B[Integer(1),Integer(2),Integer(1)])
[1, 2, 3, 4] + [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 1, 3, 4]
 + [2, 1, 4, 3] + [2, 3, 4, 1] + [3, 1, 2, 4] + [3, 1, 4, 2]
 + [3, 2, 4, 1] + [4, 1, 2, 3] + [4, 1, 3, 2] + [4, 2, 3, 1]
```

to_symmetric_group_algebra_on_basis(S)

Return the basis element index by S as a linear combination of basis elements in the symmetric group algebra.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 3).B()
sage: [B.to_symmetric_group_algebra_on_basis(c)
....:  for c in Compositions(3)]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3]
  + [2, 3, 1] + [3, 1, 2] + [3, 2, 1],
 [1, 2, 3] + [2, 1, 3] + [3, 1, 2],
 [1, 2, 3] + [1, 3, 2] + [2, 3, 1],
 [1, 2, 3]]
sage: I = DescentAlgebra(QQ, 3).I()
sage: [I.to_symmetric_group_algebra_on_basis(c)
....:  for c in Compositions(3)]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1]
  + [3, 1, 2] + [3, 2, 1],
 1/2*[1, 2, 3] - 1/2*[1, 3, 2] + 1/2*[2, 1, 3]
  - 1/2*[2, 3, 1] + 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
 1/2*[1, 2, 3] + 1/2*[1, 3, 2] - 1/2*[2, 1, 3]
  + 1/2*[2, 3, 1] - 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
 1/3*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3]
  - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] + 1/3*[3, 2, 1]]
```

```
>>> from sage.all import *
>>> B = DescentAlgebra(QQ, Integer(3)).B()
>>> [B.to_symmetric_group_algebra_on_basis(c)
...  for c in Compositions(Integer(3))]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3]
  + [2, 3, 1] + [3, 1, 2] + [3, 2, 1],
 [1, 2, 3] + [2, 1, 3] + [3, 1, 2],
 [1, 2, 3] + [1, 3, 2] + [2, 3, 1],
 [1, 2, 3]]
>>> I = DescentAlgebra(QQ, Integer(3)).I()
>>> [I.to_symmetric_group_algebra_on_basis(c)
...  for c in Compositions(Integer(3))]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1]
  + [3, 1, 2] + [3, 2, 1],
 1/2*[1, 2, 3] - 1/2*[1, 3, 2] + 1/2*[2, 1, 3]
  - 1/2*[2, 3, 1] + 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
 1/2*[1, 2, 3] + 1/2*[1, 3, 2] - 1/2*[2, 1, 3]
  + 1/2*[2, 3, 1] - 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
 1/3*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3]
  - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] + 1/3*[3, 2, 1]]
```

super_categories()

The super categories of `self`.

EXAMPLES:

```
sage: from sage.combinat.descent_algebra import DescentAlgebraBases
sage: DA = DescentAlgebra(QQ, 4)
sage: bases = DescentAlgebraBases(DA)
sage: bases.super_categories()
[Category of finite dimensional algebras with basis over Rational Field,
 Category of realizations of Descent algebra of 4 over Rational Field]
```

```
>>> from sage.all import *
>>> from sage.combinat.descent_algebra import DescentAlgebraBases
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> bases = DescentAlgebraBases(DA)
>>> bases.super_categories()
[Category of finite dimensional algebras with basis over Rational Field,
 Category of realizations of Descent algebra of 4 over Rational Field]
```

5.1.74 Combinatorial designs and incidence structures

All designs can be accessed by `designs.<tab>` and are listed in the design catalog:

- *Catalog of designs*

Design-related classes

- *Incidence structures (i.e. hypergraphs, i.e. set systems)*
- *Covering designs: coverings of t-element subsets of a v-set by k-sets*

Constructions

- *Block designs*
- *Balanced Incomplete Block Designs (BIBD)*
- *Resolvable Balanced Incomplete Block Design (RBIBD)*
- *Group-Divisible Designs (GDD)*
- *Mutually Orthogonal Latin Squares (MOLS)*
- *Orthogonal arrays (OA)*
- *Orthogonal arrays (build recursive constructions)*
- *Orthogonal arrays (find recursive constructions)*
- *Difference families*
- *Difference Matrices*
- *Steiner Quadruple Systems*
- *Two-graphs*
- *Database of small combinatorial designs*
- *Database of generalised quadrangles with spread*

Technical things

- *External Representations of Block Designs*
- *Cython functions for combinatorial designs*
- *Hypergraph isomorphic copy search*
- *Evenly distributed sets in finite fields*

5.1.75 Balanced Incomplete Block Designs (BIBD)

This module gathers everything related to Balanced Incomplete Block Designs. One can build a BIBD (or check that it can be built) with `balanced_incomplete_block_design()`:

```
sage: BIBD = designs.balanced_incomplete_block_design(7,3,1)                  #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> BIBD = designs.balanced_incomplete_block_design(Integer(7),Integer(3),Integer(1))␣
↪                        # needs sage.schemes
```

In particular, Sage can build a $(v, k, 1)$-BIBD when one exists for all $k \leq 5$. The following functions are available:

`balanced_incomplete_block_design()`	Return a BIBD of parameters v, k, λ.
`BIBD_from_TD()`	Return a BIBD through TD-based constructions.
`BIBD_from_difference_family()`	Return the BIBD associated to the difference family `D` on the group `G`.
`BIBD_from_PBD()`	Return a $(v, k, 1)$-BIBD from a (r, K)-PBD where $r = (v-1)/(k-1)$.
`PBD_from_TD()`	Return a $(kt, \{k, t\})$-PBD if $u = 0$ and a $(kt+u, \{k, k+1, t, u\})$-PBD otherwise.
`steiner_triple_system()`	Return a Steiner Triple System.
`v_5_1_BIBD()`	Return a $(v, 5, 1)$-BIBD.
`v_4_1_BIBD()`	Return a $(v, 4, 1)$-BIBD.
`PBD_4_5_8_9_12()`	Return a $(v, \{4, 5, 8, 9, 12\})$-PBD on v elements.
`BIBD_5q_5_for_q_prime_pow`	Return a $(5q, 5, 1)$-BIBD with $q \equiv 1 \pmod 4$ a prime power.

Construction of BIBD when $k = 4$

Decompositions of K_v into K_4 (i.e. $(v, 4, 1)$-BIBD) are built following Douglas Stinson's construction as presented in [Stinson2004] page 167. It is based upon the construction of $(v\{4, 5, 8, 9, 12\})$-PBD (see the doc of `PBD_4_5_8_9_12()`), knowing that a $(v\{4, 5, 8, 9, 12\})$-PBD on v points can always be transformed into a $((k-1)v+1, 4, 1)$-BIBD, which covers all possible cases of $(v, 4, 1)$-BIBD.

Construction of BIBD when $k = 5$

Decompositions of K_v into K_4 (i.e. $(v, 4, 1)$-BIBD) are built following Clayton Smith's construction [ClaytonSmith].

Functions

`sage.combinat.designs.bibd.`**`BIBD`**

alias of `BalancedIncompleteBlockDesign`

`sage.combinat.designs.bibd.`**`BIBD_5q_5_for_q_prime_power`**`(q)`

Return a $(5q, 5, 1)$-BIBD with $q \equiv 1 \pmod 4$ a prime power.

See Theorem 24 [ClaytonSmith].

INPUT:

- `q` – integer; a prime power such that $q \equiv 1 \pmod 4$

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import BIBD_5q_5_for_q_prime_power
sage: for q in [25, 45, 65, 85, 125, 145, 185, 205, 305, 405, 605]: # long time
....:     _ = BIBD_5q_5_for_q_prime_power(q/5)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import BIBD_5q_5_for_q_prime_power
>>> for q in [Integer(25), Integer(45), Integer(65), Integer(85), Integer(125),␣
↪Integer(145), Integer(185), Integer(205), Integer(305), Integer(405),␣
↪Integer(605)]: # long time
...     _ = BIBD_5q_5_for_q_prime_power(q/Integer(5))
```

sage.combinat.designs.bibd.**BIBD_from_PBD**(*PBD*, *v*, *k*, *check=True*, *base_cases=None*)

Return a $(v, k, 1)$-BIBD from a (r, K)-PBD where $r = (v - 1)/(k - 1)$.

This is Theorem 7.20 from [Stinson2004].

INPUT:

- v, k – integers
- PBD – a PBD on $r = (v - 1)/(k - 1)$ points, such that for any block of PBD of size s there must exist a $((k - 1)s + 1, k, 1)$-BIBD
- check – boolean (default: True); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- base_cases – caching system, for internal use

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import PBD_4_5_8_9_12
sage: from sage.combinat.designs.bibd import BIBD_from_PBD
sage: from sage.combinat.designs.bibd import is_pairwise_balanced_design
sage: PBD = PBD_4_5_8_9_12(17)                                              #␣
↪needs sage.schemes
sage: bibd = is_pairwise_balanced_design(BIBD_from_PBD(PBD,52,4),52,[4])    #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import PBD_4_5_8_9_12
>>> from sage.combinat.designs.bibd import BIBD_from_PBD
>>> from sage.combinat.designs.bibd import is_pairwise_balanced_design
>>> PBD = PBD_4_5_8_9_12(Integer(17))                                        ␣
↪     # needs sage.schemes
>>> bibd = is_pairwise_balanced_design(BIBD_from_PBD(PBD,Integer(52),Integer(4)),
↪Integer(52),[Integer(4)])        # needs sage.schemes
```

sage.combinat.designs.bibd.**BIBD_from_TD**(*v*, *k*, *existence=False*)

Return a BIBD through TD-based constructions.

INPUT:

- v, k – integers; computes a $(v, k, 1)$-BIBD
- existence – boolean; instead of building the design, return:
  - True – meaning that Sage knows how to build the design
  - Unknown – meaning that Sage does not know how to build the design, but that the design may exist (see sage.misc.unknown)
  - False – meaning that the design does not exist

This method implements three constructions:

- If there exists a $TD(k, v)$ and a $(v, k, 1)$-BIBD then there exists a $(kv, k, 1)$-BIBD.

  The BIBD is obtained from all blocks of the TD, and from the blocks of the $(v, k, 1)$-BIBDs defined over the k groups of the TD.

- If there exists a $TD(k, v)$ and a $(v + 1, k, 1)$-BIBD then there exists a $(kv + 1, k, 1)$-BIBD.

  The BIBD is obtained from all blocks of the TD, and from the blocks of the $(v + 1, k, 1)$-BIBDs defined over the sets $V_1 \cup \infty, \ldots, V_k \cup \infty$ where the $V_1, \ldots, V_k$ are the groups of the TD.

- If there exists a $TD(k, v)$ and a $(v + k, k, 1)$-BIBD then there exists a $(kv + k, k, 1)$-BIBD.

  The BIBD is obtained from all blocks of the TD, and from the blocks of the $(v + k, k, 1)$-BIBDs defined over the sets $V_1 \cup \{\infty_1, \ldots, \infty_k\}, \ldots, V_k \cup \{\infty_1, \ldots, \infty_k\}$ where the $V_1, \ldots, V_k$ are the groups of the TD. By making sure that all copies of the $(v + k, k, 1)$-BIBD contain the block $\{\infty_1, \ldots, \infty_k\}$, the result is also a BIBD.

These constructions can be found in http://www.argilo.net/files/bibd.pdf.

EXAMPLES:

First construction:

```
sage: from sage.combinat.designs.bibd import BIBD_from_TD
sage: BIBD_from_TD(25,5,existence=True)                                    #␣
↪needs sage.schemes
True
sage: _ = BlockDesign(25,BIBD_from_TD(25,5))                               #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import BIBD_from_TD
>>> BIBD_from_TD(Integer(25),Integer(5),existence=True)                    ␣
↪               # needs sage.schemes
True
>>> _ = BlockDesign(Integer(25),BIBD_from_TD(Integer(25),Integer(5)))      ␣
↪                        # needs sage.schemes
```

Second construction:

```
sage: from sage.combinat.designs.bibd import BIBD_from_TD
sage: BIBD_from_TD(21,5,existence=True)                                    #␣
↪needs sage.schemes
True
sage: _ = BlockDesign(21,BIBD_from_TD(21,5))                               #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import BIBD_from_TD
>>> BIBD_from_TD(Integer(21),Integer(5),existence=True)                    ␣
↪               # needs sage.schemes
True
>>> _ = BlockDesign(Integer(21),BIBD_from_TD(Integer(21),Integer(5)))      ␣
↪                        # needs sage.schemes
```

Third construction:

```
sage: from sage.combinat.designs.bibd import BIBD_from_TD
sage: BIBD_from_TD(85,5,existence=True)                                         #␣
↪needs sage.schemes
True
sage: _ = BlockDesign(85,BIBD_from_TD(85,5))                                    #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import BIBD_from_TD
>>> BIBD_from_TD(Integer(85),Integer(5),existence=True)                         ␣
↪                # needs sage.schemes
True
>>> _ = BlockDesign(Integer(85),BIBD_from_TD(Integer(85),Integer(5)))           ␣
↪                         # needs sage.schemes
```

No idea:

```
sage: from sage.combinat.designs.bibd import BIBD_from_TD
sage: BIBD_from_TD(20,5,existence=True)
Unknown
sage: BIBD_from_TD(20,5)
Traceback (most recent call last):
...
NotImplementedError: I do not know how to build a (20,5,1)-BIBD!
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import BIBD_from_TD
>>> BIBD_from_TD(Integer(20),Integer(5),existence=True)
Unknown
>>> BIBD_from_TD(Integer(20),Integer(5))
Traceback (most recent call last):
...
NotImplementedError: I do not know how to build a (20,5,1)-BIBD!
```

sage.combinat.designs.bibd.**BIBD_from_arc_in_desarguesian_projective_plane**(*n*, *k*, *existence=False*)

Return a $(n,k,1)$-BIBD from a maximal arc in a projective plane.

This function implements a construction from Denniston [Denniston69], who describes a maximal `arc` in a `Desarguesian Projective Plane` of order 2^k. From two powers of two n,q with $n < q$, it produces a $((n-1)(q+1)+1,n,1)$-BIBD.

INPUT:

- `n`, `k` – integers; must be powers of two (among other restrictions)
- `existence` – boolean; whether to return the BIBD obtained through this construction (default), or to merely indicate with a boolean return value whether this method *can* build the requested BIBD.

EXAMPLES:

A $(232,8,1)$-BIBD:

```
sage: from sage.combinat.designs.bibd import BIBD_from_arc_in_desarguesian_
↪projective_plane
```

(continues on next page)

(continued from previous page)

```
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: D = BIBD_from_arc_in_desarguesian_projective_plane(232,8)              #␣
↪needs sage.libs.gap sage.modules sage.rings.finite_rings
sage: BalancedIncompleteBlockDesign(232,D)                                   #␣
↪needs sage.libs.gap sage.modules sage.rings.finite_rings
(232,8,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import BIBD_from_arc_in_desarguesian_
↪projective_plane
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> D = BIBD_from_arc_in_desarguesian_projective_plane(Integer(232),Integer(8))  ␣
↪              # needs sage.libs.gap sage.modules sage.rings.finite_rings
>>> BalancedIncompleteBlockDesign(Integer(232),D)                                ␣
↪     # needs sage.libs.gap sage.modules sage.rings.finite_rings
(232,8,1)-Balanced Incomplete Block Design
```

A $(120, 8, 1)$-BIBD:

```
sage: D = BIBD_from_arc_in_desarguesian_projective_plane(120,8)              #␣
↪needs sage.libs.gap sage.modules sage.rings.finite_rings
sage: BalancedIncompleteBlockDesign(120,D)                                   #␣
↪needs sage.libs.gap sage.modules sage.rings.finite_rings
(120,8,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> D = BIBD_from_arc_in_desarguesian_projective_plane(Integer(120),Integer(8))  ␣
↪              # needs sage.libs.gap sage.modules sage.rings.finite_rings
>>> BalancedIncompleteBlockDesign(Integer(120),D)                                ␣
↪     # needs sage.libs.gap sage.modules sage.rings.finite_rings
(120,8,1)-Balanced Incomplete Block Design
```

Other parameters:

```
sage: all(BIBD_from_arc_in_desarguesian_projective_plane(n,k,existence=True)
....:     for n,k in
....:       [(120, 8), (232, 8), (456, 8), (904, 8), (496, 16),
....:        (976, 16), (1936, 16), (2016, 32), (4000, 32), (8128, 64)])
True
```

```
>>> from sage.all import *
>>> all(BIBD_from_arc_in_desarguesian_projective_plane(n,k,existence=True)
...     for n,k in
...       [(Integer(120), Integer(8)), (Integer(232), Integer(8)), (Integer(456),␣
↪Integer(8)), (Integer(904), Integer(8)), (Integer(496), Integer(16)),
...        (Integer(976), Integer(16)), (Integer(1936), Integer(16)),␣
↪(Integer(2016), Integer(32)), (Integer(4000), Integer(32)), (Integer(8128),␣
↪Integer(64))])
True
```

Of course, not all can be built this way:

```
sage: BIBD_from_arc_in_desarguesian_projective_plane(7,3,existence=True)
False
sage: BIBD_from_arc_in_desarguesian_projective_plane(7,3)
Traceback (most recent call last):
...
ValueError: This function cannot produce a (7,3,1)-BIBD
```

```
>>> from sage.all import *
>>> BIBD_from_arc_in_desarguesian_projective_plane(Integer(7),Integer(3),
↪existence=True)
False
>>> BIBD_from_arc_in_desarguesian_projective_plane(Integer(7),Integer(3))
Traceback (most recent call last):
...
ValueError: This function cannot produce a (7,3,1)-BIBD
```

REFERENCE:

`sage.combinat.designs.bibd.`**`BIBD_from_difference_family`**`(`*G*, *D*, *lambd=None*, *check=True*`)`

Return the BIBD associated to the difference family `D` on the group `G`.

Let G be a group. A (G,k,λ)-*difference family* is a family $B=\{B_1,B_2,\ldots,B_b\}$ of k-subsets of G such that for each element of $G\backslash\{0\}$ there exists exactly λ pairs of elements (x,y), x and y belonging to the same block, such that $x-y=g$ (or x y^{-1} = g` in multiplicative notation).

If $\{B_1,B_2,\ldots,B_b\}$ is a (G,k,λ)-difference family then its set of translates $\{B_i\cdot g; i\in\{1,\ldots,b\}, g\in G\}$ is a (v,k,λ)-BIBD where v is the cardinality of G.

INPUT:

- `G` – a finite additive Abelian group
- `D` – a difference family on `G` (short blocks are allowed)
- `lambd` – the λ parameter (optional, only used if `check` is `True`)
- `check` – boolean (default: `True`); whether or not we check the output

EXAMPLES:

```
sage: G = Zmod(21)
sage: D = [[0,1,4,14,16]]
sage: sorted(G(x-y) for x in D[0] for y in D[0] if x != y)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

sage: from sage.combinat.designs.bibd import BIBD_from_difference_family
sage: BIBD_from_difference_family(G, D)
[[0, 1, 4, 14, 16],
 [1, 2, 5, 15, 17],
 [2, 3, 6, 16, 18],
 [3, 4, 7, 17, 19],
 [4, 5, 8, 18, 20],
 [5, 6, 9, 19, 0],
 [6, 7, 10, 20, 1],
 [7, 8, 11, 0, 2],
 [8, 9, 12, 1, 3],
 [9, 10, 13, 2, 4],
```

(continues on next page)

(continued from previous page)

```
 [10, 11, 14, 3, 5],
 [11, 12, 15, 4, 6],
 [12, 13, 16, 5, 7],
 [13, 14, 17, 6, 8],
 [14, 15, 18, 7, 9],
 [15, 16, 19, 8, 10],
 [16, 17, 20, 9, 11],
 [17, 18, 0, 10, 12],
 [18, 19, 1, 11, 13],
 [19, 20, 2, 12, 14],
 [20, 0, 3, 13, 15]]
```

```
>>> from sage.all import *
>>> G = Zmod(Integer(21))
>>> D = [[Integer(0),Integer(1),Integer(4),Integer(14),Integer(16)]]
>>> sorted(G(x-y) for x in D[Integer(0)] for y in D[Integer(0)] if x != y)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

>>> from sage.combinat.designs.bibd import BIBD_from_difference_family
>>> BIBD_from_difference_family(G, D)
[[0, 1, 4, 14, 16],
 [1, 2, 5, 15, 17],
 [2, 3, 6, 16, 18],
 [3, 4, 7, 17, 19],
 [4, 5, 8, 18, 20],
 [5, 6, 9, 19, 0],
 [6, 7, 10, 20, 1],
 [7, 8, 11, 0, 2],
 [8, 9, 12, 1, 3],
 [9, 10, 13, 2, 4],
 [10, 11, 14, 3, 5],
 [11, 12, 15, 4, 6],
 [12, 13, 16, 5, 7],
 [13, 14, 17, 6, 8],
 [14, 15, 18, 7, 9],
 [15, 16, 19, 8, 10],
 [16, 17, 20, 9, 11],
 [17, 18, 0, 10, 12],
 [18, 19, 1, 11, 13],
 [19, 20, 2, 12, 14],
 [20, 0, 3, 13, 15]]
```

class `sage.combinat.designs.bibd.`**`BalancedIncompleteBlockDesign`**(*points*, *blocks*, *k=None*, *lambd=1*, *check=True*, *copy=True*, ***kwds*)

Bases: *`PairwiseBalancedDesign`*

Balanced Incomplete Block Design (BIBD).

INPUT:

- `points` – the underlying set. If `points` is an integer v, then the set is considered to be $\{0, ..., v-1\}$
- `blocks` – collection of blocks

- `k` – integer; size of the blocks. Set to `None` (automatic guess) by default
- `lambd` – integer; value of λ, set to 1 by default
- `check` – boolean; whether to check that the design is a PBD with the right parameters
- `copy` – (use with caution) if set to `False` then `blocks` must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your `blocks` object will become the instance's internal data.

EXAMPLES:

```
sage: b=designs.balanced_incomplete_block_design(9,3); b
(9,3,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> b=designs.balanced_incomplete_block_design(Integer(9),Integer(3)); b
(9,3,1)-Balanced Incomplete Block Design
```

arc(*s*, *solver=2*, *verbose=None*, *integrality_tolerance=0*)

Return the `s`-arc with maximum cardinality.

A s-arc is a subset of points in a BIBD that intersects each block on at most s points. It is one possible generalization of independent set for graphs.

A simple counting shows that the cardinality of a s-arc is at most $(s-1)*r+1$ where r is the number of blocks incident to any point. A s-arc in a BIBD with cardinality $(s-1)*r+1$ is called maximal and is characterized by the following property: it is not empty and each block either contains 0 or s points of this arc. Equivalently, the trace of the BIBD on these points is again a BIBD (with block size s).

For more informations, see Wikipedia article Arc_(projective_geometry).

INPUT:

- `s` – (default: 2) the maximum number of points from the arc in each block
- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

EXAMPLES:

```
sage: # needs sage.schemes
sage: B = designs.balanced_incomplete_block_design(21, 5)
sage: a2 = B.arc(); a2  # random
[5, 9, 10, 12, 15, 20]
sage: len(a2)
6
sage: a4 = B.arc(4); a4  # random
[0, 1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 20]
sage: len(a4)
16
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> B = designs.balanced_incomplete_block_design(Integer(21), Integer(5))
>>> a2 = B.arc(); a2  # random
[5, 9, 10, 12, 15, 20]
>>> len(a2)
6
>>> a4 = B.arc(Integer(4)); a4  # random
[0, 1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 20]
>>> len(a4)
16
```

The 2-arc and 4-arc above are maximal. One can check that they intersect the blocks in either 0 or s points. Or equivalently that the traces are again BIBD:

```
sage: r = (21-1)//(5-1)
sage: 1 + r*1
6
sage: 1 + r*3
16

sage: B.trace(a2).is_t_design(2, return_parameters=True)                      #␣
↪needs sage.schemes
(True, (2, 6, 2, 1))
sage: B.trace(a4).is_t_design(2, return_parameters=True)                      #␣
↪needs sage.schemes
(True, (2, 16, 4, 1))
```

```
>>> from sage.all import *
>>> r = (Integer(21)-Integer(1))//(Integer(5)-Integer(1))
>>> Integer(1) + r*Integer(1)
6
>>> Integer(1) + r*Integer(3)
16

>>> B.trace(a2).is_t_design(Integer(2), return_parameters=True)                ␣
↪     # needs sage.schemes
(True, (2, 6, 2, 1))
>>> B.trace(a4).is_t_design(Integer(2), return_parameters=True)                ␣
↪     # needs sage.schemes
(True, (2, 16, 4, 1))
```

Some other examples which are not maximal:

```
sage: # needs sage.numerical.mip
sage: B = designs.balanced_incomplete_block_design(25, 4)
sage: a2 = B.arc(2)
sage: r = (25-1)//(4-1)
sage: len(a2), 1 + r
(8, 9)
sage: sa2 = set(a2)
sage: set(len(sa2.intersection(b)) for b in B.blocks())
{0, 1, 2}
```

(continues on next page)

(continued from previous page)

```
sage: B.trace(a2).is_t_design(2)
False

sage: # needs sage.numerical.mip
sage: a3 = B.arc(3)
sage: len(a3), 1 + 2*r
(15, 17)
sage: sa3 = set(a3)
sage: set(len(sa3.intersection(b)) for b in B.blocks()) == set([0,3])
False
sage: B.trace(a3).is_t_design(3)
False
```

```
>>> from sage.all import *
>>> # needs sage.numerical.mip
>>> B = designs.balanced_incomplete_block_design(Integer(25), Integer(4))
>>> a2 = B.arc(Integer(2))
>>> r = (Integer(25)-Integer(1))//(Integer(4)-Integer(1))
>>> len(a2), Integer(1) + r
(8, 9)
>>> sa2 = set(a2)
>>> set(len(sa2.intersection(b)) for b in B.blocks())
{0, 1, 2}
>>> B.trace(a2).is_t_design(Integer(2))
False

>>> # needs sage.numerical.mip
>>> a3 = B.arc(Integer(3))
>>> len(a3), Integer(1) + Integer(2)*r
(15, 17)
>>> sa3 = set(a3)
>>> set(len(sa3.intersection(b)) for b in B.blocks()) == set([Integer(0),
↪Integer(3)])
False
>>> B.trace(a3).is_t_design(Integer(3))
False
```

sage.combinat.designs.bibd.**BruckRyserChowla_check**(*v*, *k*, *lambd*)

Check whether the parameters passed satisfy the Bruck-Ryser-Chowla theorem.

For more information on the theorem, see the corresponding Wikipedia entry.

INPUT:

- v, k, lambd – integers; parameters to check

OUTPUT: True – the parameters satisfy the theorem

- False – the theorem fails for the given parameters
- Unknown – the preconditions of the theorem are not met

EXAMPLES:

sage: from sage.combinat.designs.bibd import BruckRyserChowla_check sage: BruckRyser-Chowla_check(22,7,2) False

Nonexistence of projective planes of order 6 and 14

sage: from sage.combinat.designs.bibd import BruckRyserChowla_check sage: BruckRyserChowla_check(43,7,1) # needs sage.schemes False sage: BruckRyserChowla_check(211,15,1) # needs sage.schemes False

Existence of symmetric BIBDs with parameters $(79, 13, 2)$ and $(56, 11, 2)$

sage: from sage.combinat.designs.bibd import BruckRyserChowla_check sage: BruckRyserChowla_check(79,13,2) # needs sage.schemes True sage: BruckRyserChowla_check(56,11,2) True

sage.combinat.designs.bibd.**PBD_4_5_8_9_12**(*v*, *check=True*)

Return a $(v, \{4, 5, 8, 9, 12\})$-PBD on v elements.

A $(v, \{4, 5, 8, 9, 12\})$-PBD exists if and only if $v \equiv 0, 1 \pmod 4$. The construction implemented here appears page 168 in [Stinson2004].

INPUT:

- `v` – integer congruent to 0 or 1 modulo 4
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(40,4).blocks()   # indirect doctest␣
↪           # needs sage.schemes
[[0, 1, 2, 12], [0, 3, 6, 9], [0, 4, 8, 10],
 [0, 5, 7, 11], [0, 13, 26, 39], [0, 14, 25, 28],
 [0, 15, 27, 38], [0, 16, 22, 32], [0, 17, 23, 34],
...
```

```
>>> from sage.all import *
>>> designs.balanced_incomplete_block_design(Integer(40),Integer(4)).blocks()   #␣
↪indirect doctest           # needs sage.schemes
[[0, 1, 2, 12], [0, 3, 6, 9], [0, 4, 8, 10],
 [0, 5, 7, 11], [0, 13, 26, 39], [0, 14, 25, 28],
 [0, 15, 27, 38], [0, 16, 22, 32], [0, 17, 23, 34],
...
```

Check that Issue #16476 is fixed:

```
sage: from sage.combinat.designs.bibd import PBD_4_5_8_9_12
sage: for v in (0,1,4,5,8,9,12,13,16,17,20,21,24,25):                        #␣
↪needs sage.schemes
....:     _ = PBD_4_5_8_9_12(v)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import PBD_4_5_8_9_12
>>> for v in (Integer(0),Integer(1),Integer(4),Integer(5),Integer(8),Integer(9),
↪Integer(12),Integer(13),Integer(16),Integer(17),Integer(20),Integer(21),
↪Integer(24),Integer(25)):                            # needs sage.schemes
...     _ = PBD_4_5_8_9_12(v)
```

sage.combinat.designs.bibd.**PBD_from_TD**(*k*, *t*, *u*)

Return a $(kt, \{k, t\})$-PBD if $u = 0$ and a $(kt + u, \{k, k + 1, t, u\})$-PBD otherwise.

This is theorem 23 from [ClaytonSmith]. The PBD is obtained from the blocks a truncated $TD(k+1,t)$, to which are added the blocks corresponding to the groups of the TD. When $u=0$, a $TD(k,t)$ is used instead.

INPUT:

- k, t, u – integers such that $0 \leq u \leq t$

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import PBD_from_TD
sage: from sage.combinat.designs.bibd import is_pairwise_balanced_design
sage: PBD = PBD_from_TD(2,2,1); PBD
[[0, 2, 4], [0, 3], [1, 2], [1, 3, 4], [0, 1], [2, 3]]
sage: is_pairwise_balanced_design(PBD,2*2+1,[2,3])
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import PBD_from_TD
>>> from sage.combinat.designs.bibd import is_pairwise_balanced_design
>>> PBD = PBD_from_TD(Integer(2),Integer(2),Integer(1)); PBD
[[0, 2, 4], [0, 3], [1, 2], [1, 3, 4], [0, 1], [2, 3]]
>>> is_pairwise_balanced_design(PBD,Integer(2)*Integer(2)+Integer(1),[Integer(2),
↪Integer(3)])
True
```

class sage.combinat.designs.bibd.**PairwiseBalancedDesign**(*points*, *blocks*, *K=None*, *lambd=1*, *check=True*, *copy=True*, ***kwds*)

Bases: *GroupDivisibleDesign*

Pairwise Balanced Design (PBD).

A Pairwise Balanced Design, or (v,K,λ)-PBD, is a collection $\mathcal{B}$ of blocks defined on a set X of size v, such that any block pair of points $p_1,p_2 \in X$ occurs in exactly λ blocks of $\mathcal{B}$. Besides, for every block $B \in \mathcal{B}$ we must have $|B| \in K$.

INPUT:

- points – the underlying set; if points is an integer v, then the set is considered to be $\{0,...,v-1\}$
- blocks – collection of blocks
- K – list of integers of which the sizes of the blocks must be elements; set to None (automatic guess) by default
- lambd – integer; value of λ, set to 1 by default
- check – boolean; whether to check that the design is a PBD with the right parameters
- copy – (use with caution) if set to False then blocks must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your blocks object will become the instance's internal data.

sage.combinat.designs.bibd.**balanced_incomplete_block_design**(*v*, *k*, *lambd=1*, *existence=False*, *use_LJCR=False*)

Return a BIBD of parameters v,k,λ.

A Balanced Incomplete Block Design of parameters v,k,λ is a collection $\mathcal{C}$ of k-subsets of $V=\{0,\dots,v-1\}$ such that for any two distinct elements $x,y \in V$ there are λ elements $S \in \mathcal{C}$ such that $x,y \in S$.

For more information on BIBD, see the corresponding Wikipedia entry.

INPUT:

- `v`, `k`, `lambd` – integers
- `existence` – boolean; instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`)
  - `False` – meaning that the design does not exist
- `use_LJCR` – boolean; whether to query the La Jolla Covering Repository for the design when Sage does not know how to build it (see *best_known_covering_design_www()*). This requires internet.

See also

- *steiner_triple_system()*
- *v_4_1_BIBD()*
- *v_5_1_BIBD()*

Todo

Implement other constructions from the Handbook of Combinatorial Designs.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(7, 3, 1).blocks()             #␣
↪needs sage.schemes
[[0, 1, 3], [0, 2, 4], [0, 5, 6], [1, 2, 6], [1, 4, 5], [2, 3, 5], [3, 4, 6]]
sage: B = designs.balanced_incomplete_block_design(66, 6, 1,          # optional -␣
↪internet
....:                                                  use_LJCR=True)
sage: B                                                               # optional -␣
↪internet
(66,6,1)-Balanced Incomplete Block Design
sage: B.blocks()                                                      # optional -␣
↪internet
[[0, 1, 2, 3, 4, 65], [0, 5, 22, 32, 38, 58], [0, 6, 21, 30, 43, 48], ...
sage: designs.balanced_incomplete_block_design(216, 6, 1)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build a (216,6,1)-BIBD!
```

```
>>> from sage.all import *
>>> designs.balanced_incomplete_block_design(Integer(7), Integer(3), Integer(1)).
↪blocks()                # needs sage.schemes
[[0, 1, 3], [0, 2, 4], [0, 5, 6], [1, 2, 6], [1, 4, 5], [2, 3, 5], [3, 4, 6]]
>>> B = designs.balanced_incomplete_block_design(Integer(66), Integer(6),␣
↪Integer(1),          # optional - internet
...                                                  use_LJCR=True)
>>> B                                                                 # optional -␣
↪internet
```

(continues on next page)

(continued from previous page)

```
(66,6,1)-Balanced Incomplete Block Design
>>> B.blocks()                                           # optional -␣
↪internet
[[0, 1, 2, 3, 4, 65], [0, 5, 22, 32, 38, 58], [0, 6, 21, 30, 43, 48], ...
>>> designs.balanced_incomplete_block_design(Integer(216), Integer(6), Integer(1))
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build a (216,6,1)-BIBD!
```

sage.combinat.designs.bibd.**biplane**(*n*, *existence=False*)

Return a biplane of order n.

A biplane of order n is a symmetric $(1+\frac{(n+1)(n+2)}{2}, n+2, 2)$-BIBD. A symmetric (or square) (v,k,λ)-BIBD is a (v,k,λ)-BIBD with v blocks.

INPUT:

- n – integer; order of the biplane
- existence – boolean; instead of building the design, return:
  - True – meaning that Sage knows how to build the design
  - Unknown – meaning that Sage does not know how to build the design, but that the design may exist (see sage.misc.unknown)
  - False – meaning that the design does not exist

See also

- *balanced_incomplete_block_design()*

EXAMPLES:

```
sage: designs.biplane(4)                                                    #␣
↪needs sage.rings.finite_rings
(16,6,2)-Balanced Incomplete Block Design
sage: designs.biplane(7, existence=True)                                    #␣
↪needs sage.schemes
True
sage: designs.biplane(11)                                                   #␣
↪needs sage.schemes
(79,13,2)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> designs.biplane(Integer(4))                                             ␣
↪     # needs sage.rings.finite_rings
(16,6,2)-Balanced Incomplete Block Design
>>> designs.biplane(Integer(7), existence=True)                             ␣
↪     # needs sage.schemes
True
>>> designs.biplane(Integer(11))                                            ␣
↪     # needs sage.schemes
(79,13,2)-Balanced Incomplete Block Design
```

sage.combinat.designs.bibd.**steiner_triple_system**(n)

Return a Steiner Triple System.

A Steiner Triple System (STS) of a set $\{0, ..., n-1\}$ is a family S of 3-sets such that for any $i \neq j$ there exists exactly one set of S in which they are both contained.

It can alternatively be thought of as a factorization of the complete graph K_n with triangles.

A Steiner Triple System of a n-set exists if and only if $n \equiv 1 \pmod 6$ or $n \equiv 3 \pmod 6$, in which case one can be found through Bose's and Skolem's constructions, respectively [AndHonk97].

INPUT:

- n – return a Steiner Triple System of $\{0, ..., n-1\}$

EXAMPLES:

A Steiner Triple System on 9 elements

```
sage: sts = designs.steiner_triple_system(9)
sage: sts
(9,3,1)-Balanced Incomplete Block Design
sage: list(sts)
[[0, 1, 5], [0, 2, 4], [0, 3, 6], [0, 7, 8], [1, 2, 3],
 [1, 4, 7], [1, 6, 8], [2, 5, 8], [2, 6, 7], [3, 4, 8],
 [3, 5, 7], [4, 5, 6]]
```

```
>>> from sage.all import *
>>> sts = designs.steiner_triple_system(Integer(9))
>>> sts
(9,3,1)-Balanced Incomplete Block Design
>>> list(sts)
[[0, 1, 5], [0, 2, 4], [0, 3, 6], [0, 7, 8], [1, 2, 3],
 [1, 4, 7], [1, 6, 8], [2, 5, 8], [2, 6, 7], [3, 4, 8],
 [3, 5, 7], [4, 5, 6]]
```

As any pair of vertices is covered once, its parameters are

```
sage: sts.is_t_design(return_parameters=True)
(True, (2, 9, 3, 1))
```

```
>>> from sage.all import *
>>> sts.is_t_design(return_parameters=True)
(True, (2, 9, 3, 1))
```

An exception is raised for invalid values of n

```
sage: designs.steiner_triple_system(10)
Traceback (most recent call last):
...
EmptySetError: Steiner triple systems only exist for n = 1 mod 6 or n = 3 mod 6
```

```
>>> from sage.all import *
>>> designs.steiner_triple_system(Integer(10))
Traceback (most recent call last):
...
EmptySetError: Steiner triple systems only exist for n = 1 mod 6 or n = 3 mod 6
```

REFERENCE:

`sage.combinat.designs.bibd.`**`v_4_1_BIBD`**`(`*v*, *check=True*`)`

Return a $(v, 4, 1)$-BIBD.

A $(v, 4, 1)$-BIBD is an edge-decomposition of the complete graph K_v into copies of K_4. For more information, see `balanced_incomplete_block_design()`. It exists if and only if $v \equiv 1, 4 \pmod{12}$.

See page 167 of [Stinson2004] for the construction details.

See also

- `balanced_incomplete_block_design()`

INPUT:

- `v` – integer; number of points
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import v_4_1_BIBD  # long time
sage: for n in range(13,100):                            # long time
....:    if n%12 in [1,4]:
....:       _ = v_4_1_BIBD(n, check = True)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import v_4_1_BIBD  # long time
>>> for n in range(Integer(13),Integer(100)):                            # long␣
↪time
...    if n%Integer(12) in [Integer(1),Integer(4)]:
...       _ = v_4_1_BIBD(n, check = True)
```

`sage.combinat.designs.bibd.`**`v_5_1_BIBD`**`(`*v*, *check=True*`)`

Return a $(v, 5, 1)$-BIBD.

This method follows the construction from [ClaytonSmith].

INPUT:

- `v` – integer

See also

- `balanced_incomplete_block_design()`

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import v_5_1_BIBD
sage: i = 0
sage: while i<200:                                                    #␣
↪needs sage.libs.pari sage.schemes
....:    i += 20
```

(continues on next page)

(continued from previous page)

```
....:     _ = v_5_1_BIBD(i+1)
....:     _ = v_5_1_BIBD(i+5)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.bibd import v_5_1_BIBD
>>> i = Integer(0)
>>> while i<Integer(200):                                                    ␣
↪    # needs sage.libs.pari sage.schemes
...     i += Integer(20)
...     _ = v_5_1_BIBD(i+Integer(1))
...     _ = v_5_1_BIBD(i+Integer(5))
```

5.1.76 Resolvable Balanced Incomplete Block Design (RBIBD)

This module contains everything related to resolvable Balanced Incomplete Block Designs. The constructions implemented here can be obtained through the `designs.<tab>` object:

```
designs.resolvable_balanced_incomplete_block_design(15,3)
```

For Balanced Incomplete Block Design (BIBD) see the module *bibd*. A BIBD is said to be *resolvable* if its blocks can be partitionned into parallel classes, i.e. partitions of its ground set.

The main function of this module is *resolvable_balanced_incomplete_block_design()*, which calls all others.

resolvable_balanced_incomplete_block_design()	Return a resolvable BIBD of parameters v, k.
kirkman_triple_system()	Return a Kirkman Triple System on v points.
v_4_1_rbibd()	Return a $(v, 4, 1)$-RBIBD
PBD_4_7()	Return a $(v, \{4, 7\})$-PBD
PBD_4_7_from_Y()	Return a $(3v + 1, \{4, 7\})$-PBD from a $(v, \{4, 5, 7\}, \mathbf{N} - \{3, 6, 10\})$-GDD.

References:

Functions

sage.combinat.designs.resolvable_bibd.**PBD_4_7**(*v*, *check=True*, *existence=False*)

Return a $(v, \{4, 7\})$-PBD.

For all v such that $n \equiv 1 \pmod 3$ and $n \neq 10, 19, 31$ there exists a $(v, \{4, 7\})$-PBD. This is proved in Proposition IX.4.5 from [BJL99], which this method implements.

This construction of PBD is used by the construction of Kirkman Triple Systems.

EXAMPLES:

```
sage: from sage.combinat.designs.resolvable_bibd import PBD_4_7
sage: PBD_4_7(22)
Pairwise Balanced Design on 22 points with sets of sizes in [4, 7]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.resolvable_bibd import PBD_4_7
>>> PBD_4_7(Integer(22))
Pairwise Balanced Design on 22 points with sets of sizes in [4, 7]
```

sage.combinat.designs.resolvable_bibd.**PBD_4_7_from_Y**(*gdd*, *check=True*)

Return a $(3v + 1, \{4, 7\})$-PBD from a $(v, \{4, 5, 7\}, \mathbf{N} - \{3, 6, 10\})$-GDD.

This implements Lemma IX.3.11 from [BJL99] (p.625). All points of the GDD are tripled, and a $+\infty$ point is added to the design.

- A group of size $s \in Y = \mathbf{N} - \{3, 6, 10\}$ becomes a set of size $3s$. Adding ∞ to it gives it size $3s + 1$, and this set is then replaced by a $(3s + 1, \{4, 7\})$-PBD.
- A block of size $s \in \{4, 5, 7\}$ becomes a $(3s, \{4, 7\}, \{3\})$-GDD.

This lemma is part of the existence proof of $(v, \{4, 7\})$-PBD as explained in IX.4.5 from [BJL99]).

INPUT:

- `gdd` – a $(v, \{4, 5, 7\}, Y)$-GDD where $Y = \mathbf{N} - \{3, 6, 10\}$
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

EXAMPLES:

```
sage: from sage.combinat.designs.resolvable_bibd import PBD_4_7_from_Y
sage: PBD_4_7_from_Y(designs.transversal_design(7,8))                    #␣
↪needs sage.schemes
Pairwise Balanced Design on 169 points with sets of sizes in [4, 7]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.resolvable_bibd import PBD_4_7_from_Y
>>> PBD_4_7_from_Y(designs.transversal_design(Integer(7),Integer(8)))   ␣
↪             # needs sage.schemes
Pairwise Balanced Design on 169 points with sets of sizes in [4, 7]
```

sage.combinat.designs.resolvable_bibd.**kirkman_triple_system**(*v*, *existence=False*)

Return a Kirkman Triple System on v points.

A Kirkman Triple System $KTS(v)$ is a resolvable Steiner Triple System. It exists if and only if $v \equiv 3 \pmod 6$.

INPUT:

- `n` – integer
- `existence` – boolean (default: `False`); whether to build the $KTS(n)$ or only answer whether it exists

See also

IncidenceStructure.is_resolvable()

EXAMPLES:

A solution to Kirkmman's original problem:

```
sage: kts = designs.kirkman_triple_system(15)
sage: classes = kts.is_resolvable(1)[1]
sage: names = '0123456789abcde'
sage: def to_name(r_s_t):
....:     r, s, t = r_s_t
....:     return ' ' + names[r] + names[s] + names[t] + ' '
```

(continues on next page)

(continued from previous page)

```
sage: rows = ['   '.join(('Day {}'.format(i) for i in range(1,8)))]
sage: rows.extend('   '.join(map(to_name,row)) for row in zip(*classes))
sage: print('\n'.join(rows))
Day 1   Day 2   Day 3   Day 4   Day 5   Day 6   Day 7
 07e     18e     29e     3ae     4be     5ce     6de
 139     24a     35b     46c     05d     167     028
 26b     03c     14d     257     368     049     15a
 458     569     06a     01b     12c     23d     347
 acd     7bd     78c     89d     79a     8ab     9bc
```

```
>>> from sage.all import *
>>> kts = designs.kirkman_triple_system(Integer(15))
>>> classes = kts.is_resolvable(Integer(1))[Integer(1)]
>>> names = '0123456789abcde'
>>> def to_name(r_s_t):
...     r, s, t = r_s_t
...     return ' ' + names[r] + names[s] + names[t] + ' '
>>> rows = ['   '.join(('Day {}'.format(i) for i in range(Integer(1),
↪Integer(8))))]
>>> rows.extend('   '.join(map(to_name,row)) for row in zip(*classes))
>>> print('\n'.join(rows))
Day 1   Day 2   Day 3   Day 4   Day 5   Day 6   Day 7
 07e     18e     29e     3ae     4be     5ce     6de
 139     24a     35b     46c     05d     167     028
 26b     03c     14d     257     368     049     15a
 458     569     06a     01b     12c     23d     347
 acd     7bd     78c     89d     79a     8ab     9bc
```

sage.combinat.designs.resolvable_bibd.**resolvable_balanced_incomplete_block_design**(*v*, *k*, *existence=False*)

Return a resolvable BIBD of parameters v, k.

A BIBD is said to be *resolvable* if its blocks can be partitionned into parallel classes, i.e. partitions of the ground set.

INPUT:

- `v`, `k` – integers
- `existence` – boolean; instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`)
  - `False` – meaning that the design does not exist

See also

- *IncidenceStructure.is_resolvable()*
- *balanced_incomplete_block_design()*

EXAMPLES:

```
sage: KTS15 = designs.resolvable_balanced_incomplete_block_design(15,3)
sage: KTS15
(15,3,1)-Balanced Incomplete Block Design
sage: KTS15.is_resolvable()
True
```

```
>>> from sage.all import *
>>> KTS15 = designs.resolvable_balanced_incomplete_block_design(Integer(15),
↪Integer(3))
>>> KTS15
(15,3,1)-Balanced Incomplete Block Design
>>> KTS15.is_resolvable()
True
```

sage.combinat.designs.resolvable_bibd.**v_4_1_rbibd**(*v*, *existence=False*)

Return a $(v, 4, 1)$-RBIBD.

INPUT:

- `n` – integer
- `existence` – boolean (default: `False`); whether to build the design or only answer whether it exists

See also

- *IncidenceStructure.is_resolvable()*
- *resolvable_balanced_incomplete_block_design()*

Note

A resolvable $(v, 4, 1)$-BIBD exists whenever $1 \equiv 4 \pmod{}12$. This function, however, only implements a construction of $(v, 4, 1)$-BIBD such that $v = 3q + 1 \equiv 1 \pmod 3$ where q is a prime power (see VII.7.5.a from [BJL99]).

EXAMPLES:

```
sage: rBIBD = designs.resolvable_balanced_incomplete_block_design(28,4)
sage: rBIBD.is_resolvable()
True
sage: rBIBD.is_t_design(return_parameters=True)
(True, (2, 28, 4, 1))
```

```
>>> from sage.all import *
>>> rBIBD = designs.resolvable_balanced_incomplete_block_design(Integer(28),
↪Integer(4))
>>> rBIBD.is_resolvable()
True
>>> rBIBD.is_t_design(return_parameters=True)
(True, (2, 28, 4, 1))
```

5.1.77 Group-Divisible Designs (GDD)

This module gathers everything related to Group-Divisible Designs. The constructions defined here can be accessed through `designs.<tab>`:

```
sage: designs.group_divisible_design(14,{4},{2})
Group Divisible Design on 14 points of type 2^7
```

```
>>> from sage.all import *
>>> designs.group_divisible_design(Integer(14),{Integer(4)},{Integer(2)})
Group Divisible Design on 14 points of type 2^7
```

The main function implemented here is *group_divisible_design()* (which calls all others) and the main class is *GroupDivisibleDesign*. The following functions are available:

group_divisible_design()	Return a (v, K, G)-Group Divisible Design.
GDD_4_2()	Return a $(2q, \{4\}, \{2\})$-GDD for q a prime power with $q \equiv 1 \pmod 6$.

Functions

sage.combinat.designs.group_divisible_designs.**GDD_4_2**(*q*, *existence=False*, *check=True*)

Return a $(2q, \{4\}, \{2\})$-GDD for q a prime power with $q \equiv 1 \pmod 6$.

This method implements Lemma VII.5.17 from [BJL99] (p.495).

INPUT:

- `q` – integer
- `existence` – boolean; instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`)
  - `False` – meaning that the design does not exist
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

EXAMPLES:

```
sage: from sage.combinat.designs.group_divisible_designs import GDD_4_2
sage: GDD_4_2(7,existence=True)
True
sage: GDD_4_2(7)
Group Divisible Design on 14 points of type 2^7
sage: GDD_4_2(8,existence=True)
Unknown
sage: GDD_4_2(8)
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.group_divisible_designs import GDD_4_2
>>> GDD_4_2(Integer(7),existence=True)
True
>>> GDD_4_2(Integer(7))
Group Divisible Design on 14 points of type 2^7
>>> GDD_4_2(Integer(8),existence=True)
Unknown
>>> GDD_4_2(Integer(8))
Traceback (most recent call last):
...
NotImplementedError
```

class sage.combinat.designs.group_divisible_designs.**GroupDivisibleDesign**(*points*, *groups*, *blocks*, *G=None*, *K=None*, *lambd=1*, *check=True*, *copy=True*, ***kwds*)

Bases: *IncidenceStructure*

Group Divisible Design (GDD).

Let K and G be sets of positive integers and let λ be a positive integer. A Group Divisible Design of index λ and order v is a triple $(V, \mathcal{G}, \mathcal{B})$ where:

- V is a set of cardinality v
- $\mathcal{G}$ is a partition of V into groups whose size belongs to G
- $\mathcal{B}$ is a family of subsets of V whose size belongs to K such that any two points $p_1, p_2 \in V$ from different groups appear simultaneously in exactly λ elements of $\mathcal{B}$. Besides, a group and a block intersect on at most one point.

If $K = \{k_1, ..., k_l\}$ and G has exactly m_i groups of cardinality k_i then G is said to have type $k_1^{m_1}...k_l^{m_l}$.

INPUT:

- `points` – the underlying set. If `points` is an integer v, then the set is considered to be $\{0, ..., v-1\}$
- `groups` – the groups of the design. Set to `None` for an automatic guess (this triggers `check=True` and can thus cost some time)
- `blocks` – collection of blocks
- `G` – list of integers of which the sizes of the groups must be elements. Set to `None` (automatic guess) by default
- `K` – list of integers of which the sizes of the blocks must be elements. Set to `None` (automatic guess) by default
- `lambd` – integer (default: 1); value of λ
- `check` – boolean (default: `True`); whether to check that the design is indeed a GDD with the right parameters
- `copy` – (use with caution) if set to `False` then `blocks` must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your `blocks` object will become the instance's internal data.

EXAMPLES:

```
sage: from sage.combinat.designs.group_divisible_designs import␣
↪GroupDivisibleDesign
sage: TD = designs.transversal_design(4,10)
sage: groups = [list(range(i*10,(i+1)*10)) for i in range(4)]
sage: GDD = GroupDivisibleDesign(40,groups,TD); GDD
Group Divisible Design on 40 points of type 10^4
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.group_divisible_designs import GroupDivisibleDesign
>>> TD = designs.transversal_design(Integer(4),Integer(10))
>>> groups = [list(range(i*Integer(10),(i+Integer(1))*Integer(10))) for i in␣
↪range(Integer(4))]
>>> GDD = GroupDivisibleDesign(Integer(40),groups,TD); GDD
Group Divisible Design on 40 points of type 10^4
```

With unspecified groups:

```
sage: # needs sage.schemes
sage: D = designs.transversal_design(4,3).relabel(list('abcdefghiklm'),
↪inplace=False).blocks()
sage: GDD = GroupDivisibleDesign('abcdefghiklm',None,D)
sage: sorted(GDD.groups())
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i'], ['k', 'l', 'm']]
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> D = designs.transversal_design(Integer(4),Integer(3)).relabel(list(
↪'abcdefghiklm'),inplace=False).blocks()
>>> GDD = GroupDivisibleDesign('abcdefghiklm',None,D)
>>> sorted(GDD.groups())
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i'], ['k', 'l', 'm']]
```

groups()

Return the groups of the Group-Divisible Design.

EXAMPLES:

```
sage: from sage.combinat.designs.group_divisible_designs import␣
↪GroupDivisibleDesign
sage: TD = designs.transversal_design(4,10)
sage: groups = [list(range(i*10,(i+1)*10)) for i in range(4)]
sage: GDD = GroupDivisibleDesign(40,groups,TD); GDD
Group Divisible Design on 40 points of type 10^4
sage: GDD.groups()
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39]]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.group_divisible_designs import␣
↪GroupDivisibleDesign
```

(continues on next page)

(continued from previous page)

```
>>> TD = designs.transversal_design(Integer(4),Integer(10))
>>> groups = [list(range(i*Integer(10),(i+Integer(1))*Integer(10))) for i in␣
→range(Integer(4))]
>>> GDD = GroupDivisibleDesign(Integer(40),groups,TD); GDD
Group Divisible Design on 40 points of type 10^4
>>> GDD.groups()
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39]]
```

sage.combinat.designs.group_divisible_designs.**group_divisible_design**(*v*, *K*, *G*, *existence=False*, *check=False*)

Return a (v, K, G)-Group Divisible Design.

A (v, K, G)-GDD is a pair $(\mathcal{G}, \mathcal{B})$ where:

- $\mathcal{G}$ is a partition of $X = \bigcup \mathcal{G}$ where $|X| = v$
- $\forall S \in \mathcal{G}, |S| \in G$
- $\forall S \in \mathcal{B}, |S| \in K$
- $\mathcal{G} \cup \mathcal{B}$ is a $(v, K \cup G)$-PBD

For more information, see the documentation of `GroupDivisibleDesign` or *PairwiseBalancedDesign*.

INPUT:

- `v` – integer
- `K`, `G` – sets of integers
- `existence` – boolean; instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`)
  - `False` – meaning that the design does not exist
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

Note

The GDD returned by this function are defined on `range(v)`, and its groups are sets of consecutive integers.

EXAMPLES:

```
sage: designs.group_divisible_design(14,{4},{2})
Group Divisible Design on 14 points of type 2^7
```

```
>>> from sage.all import *
>>> designs.group_divisible_design(Integer(14),{Integer(4)},{Integer(2)})
Group Divisible Design on 14 points of type 2^7
```

5.1.78 Block designs

A *block design* is a set together with a family of subsets (repeated subsets are allowed) whose members are chosen to satisfy some set of properties that are deemed useful for a particular application. See Wikipedia article Block_design.

REFERENCES:

- Block design from wikipedia: Wikipedia article Block_design
- What is a block design?, http://designtheory.org/library/extrep/extrep-1.1-html/node4.html (in 'The External Representation of Block Designs' by Peter J. Cameron, Peter Dobcsanyi, John P. Morgan, Leonard H. Soicher)

AUTHORS:

- Quentin Honoré (2015): construction of Hughes plane Issue #18527
- Vincent Delecroix (2014): rewrite the part on projective planes Issue #16281
- Peter Dobcsanyi and David Joyner (2007-2008)

  This is a significantly modified form of the module block_design.py (version 0.6) written by Peter Dobcsanyi peter@designtheory.org. Thanks go to Robert Miller for lots of good design suggestions.

Todo

Implement more finite non-Desarguesian plane as in [We07] and Wikipedia article Non-Desarguesian_plane.

Functions and methods

`sage.combinat.designs.block_design.`**`AffineGeometryDesign`**`(`*n*, *d*, *F*, *point_coordinates=True*, *check=True*`)`

Return an affine geometry design.

The affine geometry design $AG_d(n,q)$ is the 2-design whose blocks are the d-vector subspaces in $\mathbf{F}_q^n$. It has parameters

$$v = q^n,\ k = q^d,\ \lambda = \binom{n-1}{d-1}_q$$

where the q-binomial coefficient $\binom{m}{r}_q$ is defined by

$$\binom{m}{r}_q = \frac{(q^m-1)(q^{m-1}-1)\cdots(q^{m-r+1}-1)}{(q^r-1)(q^{r-1}-1)\cdots(q-1)}$$

See also

`ProjectiveGeometryDesign()`

INPUT:

- `n` – integer; the Euclidean dimension. The number of points of the design is $v = |\mathbf{F}_q^n|$
- `d` – integer; the dimension of the (affine) subspaces of $\mathbf{F}_q^n$ which make up the blocks
- `F` – a finite field or a prime power
- `point_coordinates` – boolean (default: `True`); whether we use coordinates in $\mathbf{F}_q^n$ or plain integers for the points of the design

- check – boolean (default: True); whether to check the output

EXAMPLES:

```
sage: # needs sage.combinat
sage: BD = designs.AffineGeometryDesign(3, 1, GF(2))
sage: BD.is_t_design(return_parameters=True)
(True, (2, 8, 2, 1))
sage: BD = designs.AffineGeometryDesign(3, 2, GF(4))
sage: BD.is_t_design(return_parameters=True)
(True, (2, 64, 16, 5))
sage: BD = designs.AffineGeometryDesign(4, 2, GF(3))
sage: BD.is_t_design(return_parameters=True)
(True, (2, 81, 9, 13))
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> BD = designs.AffineGeometryDesign(Integer(3), Integer(1), GF(Integer(2)))
>>> BD.is_t_design(return_parameters=True)
(True, (2, 8, 2, 1))
>>> BD = designs.AffineGeometryDesign(Integer(3), Integer(2), GF(Integer(4)))
>>> BD.is_t_design(return_parameters=True)
(True, (2, 64, 16, 5))
>>> BD = designs.AffineGeometryDesign(Integer(4), Integer(2), GF(Integer(3)))
>>> BD.is_t_design(return_parameters=True)
(True, (2, 81, 9, 13))
```

With F an integer instead of a finite field:

```
sage: BD = designs.AffineGeometryDesign(3, 2, 4)
sage: BD.is_t_design(return_parameters=True)
(True, (2, 64, 16, 5))
```

```
>>> from sage.all import *
>>> BD = designs.AffineGeometryDesign(Integer(3), Integer(2), Integer(4))
>>> BD.is_t_design(return_parameters=True)
(True, (2, 64, 16, 5))
```

Testing the option point_coordinates:

```
sage: designs.AffineGeometryDesign(3, 1, GF(2),
....:                              point_coordinates=True).blocks()[0]
[(0, 0, 0), (0, 0, 1)]
sage: designs.AffineGeometryDesign(3, 1, GF(2),
....:                              point_coordinates=False).blocks()[0]
[0, 1]
```

```
>>> from sage.all import *
>>> designs.AffineGeometryDesign(Integer(3), Integer(1), GF(Integer(2)),
...                              point_coordinates=True).blocks()[Integer(0)]
[(0, 0, 0), (0, 0, 1)]
>>> designs.AffineGeometryDesign(Integer(3), Integer(1), GF(Integer(2)),
...                              point_coordinates=False).blocks()[Integer(0)]
[0, 1]
```

sage.combinat.designs.block_design.**CremonaRichmondConfiguration**()

Return the Cremona-Richmond configuration.

The Cremona-Richmond configuration is a set system whose incidence graph is equal to the `TutteCoxeterGraph()`. It is a generalized quadrangle of parameters $(2, 2)$.

For more information, see the Wikipedia article Cremona-Richmond_configuration.

EXAMPLES:

```
sage: H = designs.CremonaRichmondConfiguration(); H                         #␣
↪needs networkx
Incidence structure with 15 points and 15 blocks
sage: g = graphs.TutteCoxeterGraph()                                        #␣
↪needs networkx
sage: H.incidence_graph().is_isomorphic(g)                                  #␣
↪needs networkx
True
```

```
>>> from sage.all import *
>>> H = designs.CremonaRichmondConfiguration(); H                         #␣
↪needs networkx
Incidence structure with 15 points and 15 blocks
>>> g = graphs.TutteCoxeterGraph()                                        #␣
↪needs networkx
>>> H.incidence_graph().is_isomorphic(g)                                  #␣
↪needs networkx
True
```

sage.combinat.designs.block_design.**DesarguesianProjectivePlaneDesign**(*n*, *point_coordinates=True*, *check=True*)

Return the Desarguesian projective plane of order `n` as a 2-design.

The Desarguesian projective plane of order n can also be defined as the projective plane over a field of order n. For more information, have a look at Wikipedia article Projective_plane.

INPUT:

- `n` – integer which must be a power of a prime number
- `point_coordinates` – boolean (default: `True`); whether to label the points with their homogeneous coordinates (default) or with integers
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

See also

ProjectiveGeometryDesign()

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2)
(7,3,1)-Balanced Incomplete Block Design
sage: designs.DesarguesianProjectivePlaneDesign(3)
```

(continues on next page)

(continued from previous page)

```
(13,4,1)-Balanced Incomplete Block Design
sage: designs.DesarguesianProjectivePlaneDesign(4)
(21,5,1)-Balanced Incomplete Block Design
sage: designs.DesarguesianProjectivePlaneDesign(5)
(31,6,1)-Balanced Incomplete Block Design
sage: designs.DesarguesianProjectivePlaneDesign(6)
Traceback (most recent call last):
...
ValueError: the order of a finite field must be a prime power
```

```
>>> from sage.all import *
>>> designs.DesarguesianProjectivePlaneDesign(Integer(2))
(7,3,1)-Balanced Incomplete Block Design
>>> designs.DesarguesianProjectivePlaneDesign(Integer(3))
(13,4,1)-Balanced Incomplete Block Design
>>> designs.DesarguesianProjectivePlaneDesign(Integer(4))
(21,5,1)-Balanced Incomplete Block Design
>>> designs.DesarguesianProjectivePlaneDesign(Integer(5))
(31,6,1)-Balanced Incomplete Block Design
>>> designs.DesarguesianProjectivePlaneDesign(Integer(6))
Traceback (most recent call last):
...
ValueError: the order of a finite field must be a prime power
```

sage.combinat.designs.block_design.**Hadamard3Design**(*n*)

Return the Hadamard 3-design with parameters $3-(n, \frac{n}{2}, \frac{n}{4}-1)$.

This is the unique extension of the Hadamard 2-design (see *HadamardDesign()*). We implement the description from pp. 12 in [CvL].

INPUT:

- n – integer; a multiple of 4 such that $n > 4$

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: designs.Hadamard3Design(12)
Incidence structure with 12 points and 22 blocks
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> designs.Hadamard3Design(Integer(12))
Incidence structure with 12 points and 22 blocks
```

We verify that any two blocks of the Hadamard 3-design $3-(8,4,1)$ design meet in 0 or 2 points. More generally, it is true that any two blocks of a Hadamard 3-design meet in 0 or $\frac{n}{4}$ points (for $n > 4$).

```
sage: # needs sage.combinat sage.modules
sage: D = designs.Hadamard3Design(8)
sage: N = D.incidence_matrix()
sage: N.transpose()*N
[4 2 2 2 2 2 2 2 2 2 2 2 2 0]
[2 4 2 2 2 2 2 2 2 2 2 2 0 2]
```

(continues on next page)

(continued from previous page)

```
[2 2 4 2 2 2 2 2 2 2 2 0 2 2]
[2 2 2 4 2 2 2 2 2 2 0 2 2 2]
[2 2 2 2 4 2 2 2 2 0 2 2 2 2]
[2 2 2 2 2 4 2 2 0 2 2 2 2 2]
[2 2 2 2 2 2 4 0 2 2 2 2 2 2]
[2 2 2 2 2 2 0 4 2 2 2 2 2 2]
[2 2 2 2 2 0 2 2 4 2 2 2 2 2]
[2 2 2 2 0 2 2 2 2 4 2 2 2 2]
[2 2 2 0 2 2 2 2 2 2 4 2 2 2]
[2 2 0 2 2 2 2 2 2 2 2 4 2 2]
[2 0 2 2 2 2 2 2 2 2 2 2 4 2]
[0 2 2 2 2 2 2 2 2 2 2 2 2 4]
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> D = designs.Hadamard3Design(Integer(8))
>>> N = D.incidence_matrix()
>>> N.transpose()*N
[4 2 2 2 2 2 2 2 2 2 2 2 2 0]
[2 4 2 2 2 2 2 2 2 2 2 2 0 2]
[2 2 4 2 2 2 2 2 2 2 2 0 2 2]
[2 2 2 4 2 2 2 2 2 2 0 2 2 2]
[2 2 2 2 4 2 2 2 2 0 2 2 2 2]
[2 2 2 2 2 4 2 2 0 2 2 2 2 2]
[2 2 2 2 2 2 4 0 2 2 2 2 2 2]
[2 2 2 2 2 2 0 4 2 2 2 2 2 2]
[2 2 2 2 2 0 2 2 4 2 2 2 2 2]
[2 2 2 2 0 2 2 2 2 4 2 2 2 2]
[2 2 2 0 2 2 2 2 2 2 4 2 2 2]
[2 2 0 2 2 2 2 2 2 2 2 4 2 2]
[2 0 2 2 2 2 2 2 2 2 2 2 4 2]
[0 2 2 2 2 2 2 2 2 2 2 2 2 4]
```

REFERENCES:

`sage.combinat.designs.block_design.`**`HadamardDesign`**`(n)`

As described in Section 1, p. 10, in [CvL]. The input n must have the property that there is a Hadamard matrix of order $n + 1$ (and that a construction of that Hadamard matrix has been implemented…).

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: designs.HadamardDesign(7)
Incidence structure with 7 points and 7 blocks
sage: print(designs.HadamardDesign(7))
Incidence structure with 7 points and 7 blocks
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> designs.HadamardDesign(Integer(7))
Incidence structure with 7 points and 7 blocks
>>> print(designs.HadamardDesign(Integer(7)))
Incidence structure with 7 points and 7 blocks
```

For example, the Hadamard 2-design with $n = 11$ is a design whose parameters are $2 - (11, 5, 2)$. We verify that $NJ = 5J$ for this design.

```
sage: # needs sage.combinat sage.modules
sage: D = designs.HadamardDesign(11); N = D.incidence_matrix()
sage: J = matrix(ZZ, 11, 11, [1]*11*11); N*J
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> D = designs.HadamardDesign(Integer(11)); N = D.incidence_matrix()
>>> J = matrix(ZZ, Integer(11), Integer(11),
↪[Integer(1)]*Integer(11)*Integer(11)); N*J
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
```

REFERENCES:

- [CvL] P. Cameron, J. H. van Lint, Designs, graphs, codes and their links, London Math. Soc., 1991.

`sage.combinat.designs.block_design.`**`HughesPlane`**`(`*q2*, *check=True*`)`

Return the Hughes projective plane of order `q2`.

Let q be an odd prime, the Hughes plane of order q^2 is a finite projective plane of order q^2 introduced by D. Hughes in [Hu57]. Its construction is as follows.

Let $K = GF(q^2)$ be a finite field with q^2 elements and $F = GF(q) \subset K$ be its unique subfield with q elements. We define a twisted multiplication on K as

$$x \circ y = \begin{cases} x\ y & \text{if y is a square in K} \\ x^q\ y & \text{otherwise} \end{cases}$$

The points of the Hughes plane are the triples (x, y, z) of points in $K^3 \backslash \{0, 0, 0\}$ up to the equivalence relation $(x, y, z) \sim (x \circ k, y \circ k, z \circ k)$ where $k \in K$.

For $a = 1$ or $a \in (K \backslash F)$ we define a block $L(a)$ as the set of triples (x, y, z) so that $x + a \circ y + z = 0$. The rest of the blocks are obtained by letting act the group $GL(3, F)$ by its standard action.

For more information, see Wikipedia article Hughes_plane and [We07].

See also

DesarguesianProjectivePlaneDesign() to build the Desarguesian projective planes

INPUT:

- `q2` – an even power of an odd prime number
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

EXAMPLES:

```
sage: H = designs.HughesPlane(9); H
(91,10,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> H = designs.HughesPlane(Integer(9)); H
(91,10,1)-Balanced Incomplete Block Design
```

We prove in the following computations that the Desarguesian plane `H` is not Desarguesian. Let us consider the two triangles $(0,1,10)$ and $(57,70,59)$. We show that the intersection points $D_{0,1} \cap D_{57,70}$, $D_{1,10} \cap D_{70,59}$ and $D_{10,0} \cap D_{59,57}$ are on the same line while $D_{0,70}$, $D_{1,59}$ and $D_{10,57}$ are not concurrent:

```
sage: blocks = H.blocks()
sage: line = lambda p,q: next(b for b in blocks if p in b and q in b)

sage: b_0_1 = line(0, 1)
sage: b_1_10 = line(1, 10)
sage: b_10_0 = line(10, 0)
sage: b_57_70 = line(57, 70)
sage: b_70_59 = line(70, 59)
sage: b_59_57 = line(59, 57)

sage: set(b_0_1).intersection(b_57_70)
{2}
sage: set(b_1_10).intersection(b_70_59)
{73}
sage: set(b_10_0).intersection(b_59_57)
{60}

sage: line(2, 73) == line(73, 60)
True

sage: b_0_57 = line(0, 57)
sage: b_1_70 = line(1, 70)
sage: b_10_59 = line(10, 59)

sage: p = set(b_0_57).intersection(b_1_70)
sage: q = set(b_1_70).intersection(b_10_59)
sage: p == q
False
```

```
>>> from sage.all import *
>>> blocks = H.blocks()
>>> line = lambda p,q: next(b for b in blocks if p in b and q in b)

>>> b_0_1 = line(Integer(0), Integer(1))
>>> b_1_10 = line(Integer(1), Integer(10))
>>> b_10_0 = line(Integer(10), Integer(0))
>>> b_57_70 = line(Integer(57), Integer(70))
>>> b_70_59 = line(Integer(70), Integer(59))
>>> b_59_57 = line(Integer(59), Integer(57))

>>> set(b_0_1).intersection(b_57_70)
{2}
>>> set(b_1_10).intersection(b_70_59)
{73}
>>> set(b_10_0).intersection(b_59_57)
{60}

>>> line(Integer(2), Integer(73)) == line(Integer(73), Integer(60))
True

>>> b_0_57 = line(Integer(0), Integer(57))
>>> b_1_70 = line(Integer(1), Integer(70))
>>> b_10_59 = line(Integer(10), Integer(59))

>>> p = set(b_0_57).intersection(b_1_70)
>>> q = set(b_1_70).intersection(b_10_59)
>>> p == q
False
```

`sage.combinat.designs.block_design.`**`ProjectiveGeometryDesign`**`(`*n*, *d*, *F*, *algorithm=None*, *point_coordinates=True*, *check=True*`)`

Return a projective geometry design.

The projective geometry design $PG_d(n,q)$ has for points the lines of $\mathbf{F}_q^{n+1}$, and for blocks the $(d+1)$-dimensional subspaces of $\mathbf{F}_q^{n+1}$, each of which contains $\frac{|\mathbf{F}_q|^{d+1}-1}{|\mathbf{F}_q|-1}$ lines. It is a 2-design with parameters

$$v = \binom{n+1}{1}_q,\ k = \binom{d+1}{1}_q,\ \lambda = \binom{n-1}{d-1}_q$$

where the q-binomial coefficient $\binom{m}{r}_q$ is defined by

$$\binom{m}{r}_q = \frac{(q^m-1)(q^{m-1}-1)\cdots(q^{m-r+1}-1)}{(q^r-1)(q^{r-1}-1)\cdots(q-1)}$$

See also

AffineGeometryDesign()

INPUT:

- n – the projective dimension
- d – the dimension of the subspaces which make up the blocks
- F – a finite field or a prime power
- algorithm – set to None by default, which results in using Sage's own implementation. In order to use GAP's implementation instead (i.e. its PGPointFlatBlockDesign function) set algorithm="gap". Note that GAP's "design" package must be available in this case, and that it can be installed with the gap_packages spkg.
- point_coordinates – True by default. Ignored and assumed to be False if algorithm="gap". If True, the ground set is indexed by coordinates in $\mathbf{F}_q^{n+1}$. Otherwise the ground set is indexed by integers.
- check – boolean (default: True); whether to check the output

EXAMPLES:

The set of d-dimensional subspaces in a n-dimensional projective space forms 2-designs (or balanced incomplete block designs):

```
sage: PG = designs.ProjectiveGeometryDesign(4, 2, GF(2)); PG                  #␣
↪needs sage.combinat
Incidence structure with 31 points and 155 blocks
sage: PG.is_t_design(return_parameters=True)                                  #␣
↪needs sage.combinat
(True, (2, 31, 7, 7))

sage: PG = designs.ProjectiveGeometryDesign(3, 1, GF(4))                       #␣
↪needs sage.combinat
sage: PG.is_t_design(return_parameters=True)                                  #␣
↪needs sage.combinat
(True, (2, 85, 5, 1))
```

```
>>> from sage.all import *
>>> PG = designs.ProjectiveGeometryDesign(Integer(4), Integer(2), GF(Integer(2)));
↪ PG                       # needs sage.combinat
Incidence structure with 31 points and 155 blocks
>>> PG.is_t_design(return_parameters=True)                                   #␣
↪needs sage.combinat
(True, (2, 31, 7, 7))

>>> PG = designs.ProjectiveGeometryDesign(Integer(3), Integer(1), GF(Integer(4)))␣
↪                          # needs sage.combinat
>>> PG.is_t_design(return_parameters=True)                                   #␣
↪needs sage.combinat
(True, (2, 85, 5, 1))
```

Check with F being a prime power:

```
sage: PG = designs.ProjectiveGeometryDesign(3, 2, 4); PG
Incidence structure with 85 points and 85 blocks
```

```
>>> from sage.all import *
>>> PG = designs.ProjectiveGeometryDesign(Integer(3), Integer(2), Integer(4)); PG
Incidence structure with 85 points and 85 blocks
```

Use coordinates:

```
sage: PG = designs.ProjectiveGeometryDesign(2, 1, GF(3))
sage: PG.blocks()[0]
[(1, 0, 0), (1, 0, 1), (1, 0, 2), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> PG = designs.ProjectiveGeometryDesign(Integer(2), Integer(1), GF(Integer(3)))
>>> PG.blocks()[Integer(0)]
[(1, 0, 0), (1, 0, 1), (1, 0, 2), (0, 0, 1)]
```

Use indexing by integers:

```
sage: PG = designs.ProjectiveGeometryDesign(2, 1, GF(3), point_coordinates=0)
sage: PG.blocks()[0]
[0, 1, 2, 12]
```

```
>>> from sage.all import *
>>> PG = designs.ProjectiveGeometryDesign(Integer(2), Integer(1), GF(Integer(3)),
↪point_coordinates=Integer(0))
>>> PG.blocks()[Integer(0)]
[0, 1, 2, 12]
```

Check that the constructor using gap also works:

```
sage: BD = designs.ProjectiveGeometryDesign(2, 1, GF(2), algorithm='gap')   #
↪optional - gap_package_design
sage: BD.is_t_design(return_parameters=True)                                #
↪optional - gap_package_design
(True, (2, 7, 3, 1))
```

```
>>> from sage.all import *
>>> BD = designs.ProjectiveGeometryDesign(Integer(2), Integer(1), GF(Integer(2)),
↪algorithm='gap')  # optional - gap_package_design
>>> BD.is_t_design(return_parameters=True)                               #
↪optional - gap_package_design
(True, (2, 7, 3, 1))
```

sage.combinat.designs.block_design.**WittDesign**(*n*)

INPUT:

- n – integer in $9, 10, 11, 12, 21, 22, 23, 24$

Wraps GAP Design's WittDesign. If n=24 then this function returns the large Witt design W_{24}, the unique (up to isomorphism) $5-(24, 8, 1)$ design. If n=12 then this function returns the small Witt design W_{12}, the unique (up to isomorphism) $5-(12, 6, 1)$ design. The other values of n return a block design derived from these.

Note

Requires GAP's Design package (included in the gap_packages Sage spkg).

EXAMPLES:

```
sage: # optional - gap_package_design
sage: BD = designs.WittDesign(9)
sage: BD.is_t_design(return_parameters=True)
(True, (2, 9, 3, 1))
sage: BD
Incidence structure with 9 points and 12 blocks
sage: print(BD)
Incidence structure with 9 points and 12 blocks
```

```
>>> from sage.all import *
>>> # optional - gap_package_design
>>> BD = designs.WittDesign(Integer(9))
>>> BD.is_t_design(return_parameters=True)
(True, (2, 9, 3, 1))
>>> BD
Incidence structure with 9 points and 12 blocks
>>> print(BD)
Incidence structure with 9 points and 12 blocks
```

sage.combinat.designs.block_design.**are_hyperplanes_in_projective_geometry_parameters**(*v*, *k*, *lmbda*, *return_parameters=False*)

Return `True` if the parameters `(v,k,lmbda)` are the one of hyperplanes in a (finite Desarguesian) projective space.

In other words, test whether there exists a prime power `q` and an integer `d` greater than two such that:

- $v = (q^{d+1} - 1)/(q - 1) = q^d + q^{d-1} + ... + 1$
- $k = (q^d - 1)/(q - 1) = q^{d-1} + q^{d-2} + ... + 1$
- $lmbda = (q^{d-1} - 1)/(q - 1) = q^{d-2} + q^{d-3} + ... + 1$

If it exists, such a pair `(q,d)` is unique.

INPUT:

- `v`, `k`, `lmbda` – integers

OUTPUT:

- a boolean or, if `return_parameters` is set to `True` a pair `(True, (q,d))` or `(False, (None,None))`.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import are_hyperplanes_in_
↪projective_geometry_parameters
sage: are_hyperplanes_in_projective_geometry_parameters(40, 13, 4)
True
sage: are_hyperplanes_in_projective_geometry_parameters(40, 13, 4,
....:                                          return_parameters=True)
(True, (3, 3))
```

(continues on next page)

(continued from previous page)

```
sage: PG = designs.ProjectiveGeometryDesign(3, 2, GF(3))                       #␣
↪needs sage.combinat
sage: PG.is_t_design(return_parameters=True)                                   #␣
↪needs sage.combinat
(True, (2, 40, 13, 4))

sage: are_hyperplanes_in_projective_geometry_parameters(15, 3, 1)
False
sage: are_hyperplanes_in_projective_geometry_parameters(15, 3, 1,
....:                                                   return_parameters=True)
(False, (None, None))
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.block_design import are_hyperplanes_in_projective_
↪geometry_parameters
>>> are_hyperplanes_in_projective_geometry_parameters(Integer(40), Integer(13),␣
↪Integer(4))
True
>>> are_hyperplanes_in_projective_geometry_parameters(Integer(40), Integer(13),␣
↪Integer(4),
...                                                   return_parameters=True)
(True, (3, 3))
>>> PG = designs.ProjectiveGeometryDesign(Integer(3), Integer(2), GF(Integer(3)))␣
↪                       # needs sage.combinat
>>> PG.is_t_design(return_parameters=True)                                   #␣
↪needs sage.combinat
(True, (2, 40, 13, 4))

>>> are_hyperplanes_in_projective_geometry_parameters(Integer(15), Integer(3),␣
↪Integer(1))
False
>>> are_hyperplanes_in_projective_geometry_parameters(Integer(15), Integer(3),␣
↪Integer(1),
...                                                   return_parameters=True)
(False, (None, None))
```

sage.combinat.designs.block_design.**normalize_hughes_plane_point**(*p*, *q*)

Return the normalized form of point `p` as a 3-tuple.

In the Hughes projective plane over the finite field K, all triples (xk, yk, zk) with $k \in K$ represent the same point (where the multiplication is over the nearfield built from K). This function chooses a canonical representative among them.

This function is used in `HughesPlane()`.

INPUT:

- `p` – point with the coordinates (x, y, z) (a list, a vector, a tuple…)
- `q` – cardinality of the underlying finite field

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import normalize_hughes_plane_point
sage: K = FiniteField(9,'x')
```

(continues on next page)

(continued from previous page)

```
sage: x = K.gen()
sage: normalize_hughes_plane_point((x, x + 1, x), 9)
(1, x, 1)
sage: normalize_hughes_plane_point(vector((x,x,x)), 9)
(1, 1, 1)
sage: zero = K.zero()
sage: normalize_hughes_plane_point((2*x + 2, zero, zero), 9)
(1, 0, 0)
sage: one = K.one()
sage: normalize_hughes_plane_point((2*x, one, zero), 9)
(2*x, 1, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.block_design import normalize_hughes_plane_point
>>> K = FiniteField(Integer(9),'x')
>>> x = K.gen()
>>> normalize_hughes_plane_point((x, x + Integer(1), x), Integer(9))
(1, x, 1)
>>> normalize_hughes_plane_point(vector((x,x,x)), Integer(9))
(1, 1, 1)
>>> zero = K.zero()
>>> normalize_hughes_plane_point((Integer(2)*x + Integer(2), zero, zero),␣
↪Integer(9))
(1, 0, 0)
>>> one = K.one()
>>> normalize_hughes_plane_point((Integer(2)*x, one, zero), Integer(9))
(2*x, 1, 0)
```

sage.combinat.designs.block_design.**projective_plane**(*n*, *check=True*, *existence=False*)

Return a projective plane of order `n` as a 2-design.

A finite projective plane is a 2-design with n^2+n+1 lines (or blocks) and n^2+n+1 points. For more information on finite projective planes, see the Wikipedia article Projective_plane#Finite_projective_planes.

If no construction is possible, then the function raises a `EmptySetError`, whereas if no construction is available, the function raises a `NotImplementedError`.

INPUT:

- `n` – the finite projective plane's order

EXAMPLES:

```
sage: # needs sage.schemes
sage: designs.projective_plane(2)
(7,3,1)-Balanced Incomplete Block Design
sage: designs.projective_plane(3)
(13,4,1)-Balanced Incomplete Block Design
sage: designs.projective_plane(4)
(21,5,1)-Balanced Incomplete Block Design
sage: designs.projective_plane(5)
(31,6,1)-Balanced Incomplete Block Design
sage: designs.projective_plane(6)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
EmptySetError: By the Bruck-Ryser theorem, no projective plane of order 6 exists.
sage: designs.projective_plane(10)
Traceback (most recent call last):
...
EmptySetError: No projective plane of order 10 exists by
C. Lam, L. Thiel and S. Swiercz "The nonexistence of finite
projective planes of order 10" (1989), Canad. J. Math.
sage: designs.projective_plane(12)
Traceback (most recent call last):
...
NotImplementedError: If such a projective plane exists,
we do not know how to build it.
sage: designs.projective_plane(14)
Traceback (most recent call last):
...
EmptySetError: By the Bruck-Ryser theorem, no projective plane of order 14 exists.
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> designs.projective_plane(Integer(2))
(7,3,1)-Balanced Incomplete Block Design
>>> designs.projective_plane(Integer(3))
(13,4,1)-Balanced Incomplete Block Design
>>> designs.projective_plane(Integer(4))
(21,5,1)-Balanced Incomplete Block Design
>>> designs.projective_plane(Integer(5))
(31,6,1)-Balanced Incomplete Block Design
>>> designs.projective_plane(Integer(6))
Traceback (most recent call last):
...
EmptySetError: By the Bruck-Ryser theorem, no projective plane of order 6 exists.
>>> designs.projective_plane(Integer(10))
Traceback (most recent call last):
...
EmptySetError: No projective plane of order 10 exists by
C. Lam, L. Thiel and S. Swiercz "The nonexistence of finite
projective planes of order 10" (1989), Canad. J. Math.
>>> designs.projective_plane(Integer(12))
Traceback (most recent call last):
...
NotImplementedError: If such a projective plane exists,
we do not know how to build it.
>>> designs.projective_plane(Integer(14))
Traceback (most recent call last):
...
EmptySetError: By the Bruck-Ryser theorem, no projective plane of order 14 exists.
```

sage.combinat.designs.block_design.**projective_plane_to_OA**(*pplane*, *pt=None*, *check=True*)

Return the orthogonal array built from the projective plane `pplane`.

The orthogonal array $OA(n+1,n,2)$ is obtained from the projective plane `pplane` by removing the point `pt` and the $n+1$ lines that pass through it`. These $n+1$ lines form the $n+1$ groups while the remaining n^2+n lines form the transversals.

INPUT:

- `pplane` – a projective plane as a 2-design
- `pt` – a point in the projective plane `pplane`; if it is not provided, then it is set to $n^2 + n$
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import projective_plane_to_OA
sage: p2 = designs.DesarguesianProjectivePlaneDesign(2, point_coordinates=False)
sage: projective_plane_to_OA(p2)
[[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 0]]
sage: p3 = designs.DesarguesianProjectivePlaneDesign(3, point_coordinates=False)
sage: projective_plane_to_OA(p3)
[[0, 0, 0, 0],
 [0, 1, 2, 1],
 [0, 2, 1, 2],
 [1, 0, 2, 2],
 [1, 1, 1, 0],
 [1, 2, 0, 1],
 [2, 0, 1, 1],
 [2, 1, 0, 2],
 [2, 2, 2, 0]]

sage: pp = designs.DesarguesianProjectivePlaneDesign(16, point_coordinates=False)
sage: _ = projective_plane_to_OA(pp, pt=0)
sage: _ = projective_plane_to_OA(pp, pt=3)
sage: _ = projective_plane_to_OA(pp, pt=7)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.block_design import projective_plane_to_OA
>>> p2 = designs.DesarguesianProjectivePlaneDesign(Integer(2), point_
↪coordinates=False)
>>> projective_plane_to_OA(p2)
[[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 0]]
>>> p3 = designs.DesarguesianProjectivePlaneDesign(Integer(3), point_
↪coordinates=False)
>>> projective_plane_to_OA(p3)
[[0, 0, 0, 0],
 [0, 1, 2, 1],
 [0, 2, 1, 2],
 [1, 0, 2, 2],
 [1, 1, 1, 0],
 [1, 2, 0, 1],
 [2, 0, 1, 1],
 [2, 1, 0, 2],
 [2, 2, 2, 0]]

>>> pp = designs.DesarguesianProjectivePlaneDesign(Integer(16), point_
↪coordinates=False)
>>> _ = projective_plane_to_OA(pp, pt=Integer(0))
```

(continues on next page)

(continued from previous page)

```
>>> _ = projective_plane_to_OA(pp, pt=Integer(3))
>>> _ = projective_plane_to_OA(pp, pt=Integer(7))
```

sage.combinat.designs.block_design.**q3_minus_one_matrix**(K)

Return a companion matrix in $GL(3, K)$ whose multiplicative order is $q^3 - 1$.

This function is used in `HughesPlane()`.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import q3_minus_one_matrix
sage: m = q3_minus_one_matrix(GF(3))
sage: m.multiplicative_order() == 3**3 - 1
True

sage: m = q3_minus_one_matrix(GF(4, 'a'))
sage: m.multiplicative_order() == 4**3 - 1
True

sage: m = q3_minus_one_matrix(GF(5))
sage: m.multiplicative_order() == 5**3 - 1
True

sage: m = q3_minus_one_matrix(GF(9, 'a'))
sage: m.multiplicative_order() == 9**3 - 1
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.block_design import q3_minus_one_matrix
>>> m = q3_minus_one_matrix(GF(Integer(3)))
>>> m.multiplicative_order() == Integer(3)**Integer(3) - Integer(1)
True

>>> m = q3_minus_one_matrix(GF(Integer(4), 'a'))
>>> m.multiplicative_order() == Integer(4)**Integer(3) - Integer(1)
True

>>> m = q3_minus_one_matrix(GF(Integer(5)))
>>> m.multiplicative_order() == Integer(5)**Integer(3) - Integer(1)
True

>>> m = q3_minus_one_matrix(GF(Integer(9), 'a'))
>>> m.multiplicative_order() == Integer(9)**Integer(3) - Integer(1)
True
```

sage.combinat.designs.block_design.**tdesign_params**(t, v, k, L)

Return the design's parameters: (t, v, b, r, k, L). Note that t must be given.

EXAMPLES:

```
sage: BD = BlockDesign(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,
↪5]])
sage: from sage.combinat.designs.block_design import tdesign_params
```

(continues on next page)

(continued from previous page)

```
sage: tdesign_params(2,7,3,1)
(2, 7, 7, 3, 3, 1)
```

```
>>> from sage.all import *
>>> BD = BlockDesign(Integer(7), [[Integer(0),Integer(1),Integer(2)],[Integer(0),
↪Integer(3),Integer(4)],[Integer(0),Integer(5),Integer(6)],[Integer(1),
↪Integer(3),Integer(5)],[Integer(1),Integer(4),Integer(6)],[Integer(2),
↪Integer(3),Integer(6)],[Integer(2),Integer(4),Integer(5)]])
>>> from sage.combinat.designs.block_design import tdesign_params
>>> tdesign_params(Integer(2),Integer(7),Integer(3),Integer(1))
(2, 7, 7, 3, 3, 1)
```

5.1.79 Covering Arrays (CA)

A Covering Array, denoted $CA(N;t,k,v)$, is an N by k array with entries from a set of v elements with the property that in every selection of t columns, every sequence of t elements appears in at least one row.

An Orthogonal Array, denoted $OA(N;t,k,v)$ is a covering array with the property that every sequence of t-elements appears in exactly one row. (See *sage.combinat.designs.orthogonal_arrays*).

This module collects methods relating to covering arrays, some of which are inherited from orthogonal array methods. This module defines the following functions:

is_covering_array()	Check that an input list of lists is a $CA(N;t,k,v)$.
CA_relabel()	Return a relabelled version of the CA.
CA_standard_label()	Return a version of the CA relabelled to symbols $(0,\dots,n-1)$.
truncate_columns()	Return an array with k columns from a larger one.
Kleitman_Spencer_Katona()	Return a $CA(N;2,k,2)$ using N as input.
column_Kleitman_Spencer_Katona()	Return a $CA(N;2,k,2)$ using k as input.
database_check()	Check if CA can be made from the database of combinatorial designs.
covering_array()	Return a CA with given parameters.

REFERENCES:

- [Colb2004]
- [SMC2006]
- [WC2007]

AUTHORS:

- Aaron Dwyer and brett stevens (2022): initial version

sage.combinat.designs.covering_array.**Kleitman_Spencer_Katona**(N)

Return a $CA(N;2,k,2)$ where $k = \binom{N-1}{\lceil N/2 \rceil}$.

INPUT:

- N – the number of rows in the array, must be an integer greater than 3 since any smaller would not produce enough columns for a strength 2 array.

This construction is referenced in [Colb2004] from [KS1973] and [Kat1973]

Construction

Take all distinct binary N-tuples of weight $N/2$ that have a 0 in the first position and place them as columns in an array.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_array import Kleitman_Spencer_Katona
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: C = Kleitman_Spencer_Katona(2)
Traceback (most recent call last):
...
ValueError: N must be greater than 3
sage: C = Kleitman_Spencer_Katona(5)
sage: is_covering_array(C,parameters=True)
(True, (5, 2, 4, 2))
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_array import Kleitman_Spencer_Katona
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> C = Kleitman_Spencer_Katona(Integer(2))
Traceback (most recent call last):
...
ValueError: N must be greater than 3
>>> C = Kleitman_Spencer_Katona(Integer(5))
>>> is_covering_array(C,parameters=True)
(True, (5, 2, 4, 2))
```

sage.combinat.designs.covering_array.**column_Kleitman_Spencer_Katona**(k)

Return a covering array with k columns using the Kleitman-Spencer-Katona method.

See *Kleitman_Spencer_Katona()*

INPUT:

- k – the number of columns in the array, must be an integer greater than 3 since any smaller is a trivial array for strength 2.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.covering_array import column_Kleitman_Spencer_
↪Katona
sage: C = column_Kleitman_Spencer_Katona(20)
sage: is_covering_array(C,parameters=True)
(True, (8, 2, 20, 2))
sage: column_Kleitman_Spencer_Katona(25000)
Traceback (most recent call last):
...
NotImplementedError: not implemented for k > 24310
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.covering_array import column_Kleitman_Spencer_
↪Katona
>>> C = column_Kleitman_Spencer_Katona(Integer(20))
>>> is_covering_array(C,parameters=True)
```

(continues on next page)

(continued from previous page)

```
(True, (8, 2, 20, 2))
>>> column_Kleitman_Spencer_Katona(Integer(25000))
Traceback (most recent call last):
...
NotImplementedError: not implemented for k > 24310
```

sage.combinat.designs.covering_array.**covering_array**(*strength*, *number_columns*, *levels*)

Build a $CA(N; t, k, v)$ using direct constructions, where N is the smallest size known.

INPUT:

- strength (integer) – the parameter t of the covering array, such that in any selection of t columns of the array, every t-tuple appears at least once.
- levels (integer) – the parameter v which is the number of unique symbols that appear in the covering array.
- number_columns (integer) – the number of columns desired for the covering array.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.covering_array import covering_array
sage: C1 = covering_array(2, 7, 3)
sage: is_covering_array(C1,parameters=True)
(True, (12, 2, 7, 3))
sage: C2 = covering_array(2, 11, 2)
sage: is_covering_array(C2,parameters=True)
(True, (7, 2, 11, 2))
sage: C3 = covering_array(2, 8, 7)
sage: is_covering_array(C3,parameters=True)
(True, (49, 2, 8, 7))
sage: C4 = covering_array(2, 50, 7)
No direct construction known and/or implemented for a CA(N; 2, 50, 7)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.covering_array import covering_array
>>> C1 = covering_array(Integer(2), Integer(7), Integer(3))
>>> is_covering_array(C1,parameters=True)
(True, (12, 2, 7, 3))
>>> C2 = covering_array(Integer(2), Integer(11), Integer(2))
>>> is_covering_array(C2,parameters=True)
(True, (7, 2, 11, 2))
>>> C3 = covering_array(Integer(2), Integer(8), Integer(7))
>>> is_covering_array(C3,parameters=True)
(True, (49, 2, 8, 7))
>>> C4 = covering_array(Integer(2), Integer(50), Integer(7))
No direct construction known and/or implemented for a CA(N; 2, 50, 7)
```

sage.combinat.designs.covering_array.**database_check**(*number_columns*, *strength*, *levels*)

Check if the database can be used to build a CA with the given parameters. If so return the CA, if not return False.

INPUT:

- strength (integer) – the parameter t of the covering array, such that in any selection of t columns of the array, every t-tuple appears at least once.

- `levels` (integer) – the parameter v which is the number of unique symbols that appear in the covering array.
- `number_columns` (integer) – the number of columns desired for the covering array.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.covering_array import database_check
sage: C = database_check(6, 2, 3)
sage: is_covering_array(C, parameters=True)
(True, (12, 2, 6, 3))
sage: database_check(6, 3, 3)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.covering_array import database_check
>>> C = database_check(Integer(6), Integer(2), Integer(3))
>>> is_covering_array(C, parameters=True)
(True, (12, 2, 6, 3))
>>> database_check(Integer(6), Integer(3), Integer(3))
False
```

`sage.combinat.designs.covering_array.`**`truncate_columns`**(*array*, *k*)

Return a covering array with k columns, obtained by removing excess columns from a larger covering array.

INPUT:

- `array` – the array to be truncated.
- `k` – the number of columns desired. Must be less than the number of columns in `array`.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.covering_array import truncate_columns
sage: from sage.combinat.designs.database import ca_11_2_5_3
sage: C = ca_11_2_5_3()
sage: D = truncate_columns(C,7)
Traceback (most recent call last):
...
ValueError: array only has 5 columns
sage: E = truncate_columns(C,4)
sage: is_covering_array(E,parameters=True)
(True, (11, 2, 4, 3))
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.covering_array import truncate_columns
>>> from sage.combinat.designs.database import ca_11_2_5_3
>>> C = ca_11_2_5_3()
>>> D = truncate_columns(C,Integer(7))
Traceback (most recent call last):
...
ValueError: array only has 5 columns
>>> E = truncate_columns(C,Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> is_covering_array(E,parameters=True)
(True, (11, 2, 4, 3))
```

5.1.80 Covering designs: coverings of t-element subsets of a v-set by k-sets

A (v, k, t) covering design C is an incidence structure consisting of a set of points P of order v, and a set of blocks B, where each block contains k points of P. Every t-element subset of P must be contained in at least one block.

If every t-set is contained in exactly one block of C, then we have a block design. Following the block design implementation, the standard representation of a covering design uses $P = [0, 1, ..., v - 1]$.

In addition to the parameters and incidence structure for a covering design from this database, we include extra information:

- Best known lower bound on the size of a (v, k, t)-covering design
- Name of the person(s) who produced the design
- Method of construction used
- Date when the design was added to the database

REFERENCES:

AUTHORS:

- Daniel M. Gordon (2008-12-22): initial version

Classes and methods

class sage.combinat.designs.covering_design.**CoveringDesign**(*v=0*, *k=0*, *t=0*, *size=0*, *points=None*, *blocks=None*, *low_bd=0*, *method=''*, *creator=''*, *timestamp=''*)

Bases: `SageObject`

Covering design.

INPUT:

- `v`, `k`, `t` – integer parameters of the covering design
- `size` – integer
- `points` – list of points (default: $[0, ..., v - 1]$)
- `blocks`
- `low_bd` – integer; lower bound for such a design
- `method`, `creator`, `timestamp` – database information

creator()

Return the creator of the covering design.

This field is optional, and is used in a database to give attribution for the covering design It can refer to the person who submitted it, or who originally gave a construction

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:     [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6],
....:     [2, 4, 5]],0, 'Projective Plane', 'Gino Fano')
sage: C.creator()
'Gino Fano'
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
↪Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
↪Integer(6)], [Integer(2), Integer(3), Integer(6)],
...     [Integer(2), Integer(4), Integer(5)]],Integer(0), 'Projective Plane',
↪'Gino Fano')
>>> C.creator()
'Gino Fano'
```

incidence_structure()

Return the incidence structure of this design, without extra parameters.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:     [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:     [2, 3, 6], [2, 4, 5]], 0, 'Projective Plane')
sage: D = C.incidence_structure()
sage: D.ground_set()
[0, 1, 2, 3, 4, 5, 6]
sage: D.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5],
[1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
↪Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
↪Integer(6)],
...     [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
↪Integer(5)]], Integer(0), 'Projective Plane')
>>> D = C.incidence_structure()
>>> D.ground_set()
[0, 1, 2, 3, 4, 5, 6]
>>> D.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5],
[1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

is_covering()

Check all t-sets are in fact covered by the blocks of `self`.

Note: This is very slow and wasteful of memory.

EXAMPLES:

```
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:      [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:      [2, 3, 6], [2, 4, 5]], 0, 'Projective Plane')
sage: C.is_covering()
True
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:      [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6],
....:      [2, 4, 6]], 0, 'not a covering')    # last block altered
sage: C.is_covering()
False
```

```
>>> from sage.all import *
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...      [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
↪Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
↪Integer(6)],
...      [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
↪Integer(5)]], Integer(0), 'Projective Plane')
>>> C.is_covering()
True
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...      [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
↪Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
↪Integer(6)], [Integer(2), Integer(3), Integer(6)],
...      [Integer(2), Integer(4), Integer(6)]], Integer(0), 'not a covering') ␣
↪ # last block altered
>>> C.is_covering()
False
```

k()

Return k, the size of blocks of the covering design.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:      [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:      [2, 3, 6], [2, 4, 5]], 0, 'Projective Plane')
sage: C.k()
3
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
```

(continues on next page)

(continued from previous page)

```
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
↪Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
↪Integer(6)],
...     [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
↪Integer(5)]], Integer(0), 'Projective Plane')
>>> C.k()
3
```

low_bd()

Return a lower bound for the number of blocks a covering design with these parameters could have.

Typically this is the Schonheim bound, but for some parameters better bounds have been shown.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:     [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:     [2, 3, 6], [2, 4, 5]], 0, 'Projective Plane')
sage: C.low_bd()
7
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
↪Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
↪Integer(6)],
...     [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
↪Integer(5)]], Integer(0), 'Projective Plane')
>>> C.low_bd()
7
```

method()

Return the method used to create the covering design.

This field is optional, and is used in a database to give information about how coverings were constructed.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:     [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:     [2, 3, 6], [2, 4, 5]], 0, 'Projective Plane')
sage: C.method()
'Projective Plane'
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
```

(continues on next page)

(continued from previous page)

```
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
→Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
→Integer(6)],
...     [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
→Integer(5)]], Integer(0), 'Projective Plane')
>>> C.method()
'Projective Plane'
```

size()

Return the number of blocks in the covering design.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:     [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:     [2, 3, 6], [2, 4, 5]], 0, 'Projective Plane')
sage: C.size()
7
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
→range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
→Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
→Integer(6)],
...     [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
→Integer(5)]], Integer(0), 'Projective Plane')
>>> C.size()
7
```

t()

Return t, the size of sets which must be covered by the blocks of the covering design

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:     [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:     [2, 3, 6], [2, 4, 5]], 0, 'Projective Plane')
sage: C.t()
2
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
→range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
→Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
→Integer(6)],
...     [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(5)]], Integer(0), 'Projective Plane')
>>> C.t()
2
```

timestamp()

Return the time that the covering was submitted to the database.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:     [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:     [2, 3, 6], [2, 4, 5]],0, 'Projective Plane',
....:     'Gino Fano', '1892-01-01 00:00:00')
sage: C.timestamp()  # No exact date known; in Fano's 1892 article
'1892-01-01 00:00:00'
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
↪Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
↪Integer(6)],
...     [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
↪Integer(5)]],Integer(0), 'Projective Plane',
...     'Gino Fano', '1892-01-01 00:00:00')
>>> C.timestamp()  # No exact date known; in Fano's 1892 article
'1892-01-01 00:00:00'
```

v()

Return v, the number of points in the covering design.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C = CoveringDesign(7, 3, 2, 7, range(7), [[0, 1, 2],
....:     [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6],
....:     [2, 3, 6], [2, 4, 5]], 0, 'Projective Plane')
sage: C.v()
7
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import CoveringDesign
>>> C = CoveringDesign(Integer(7), Integer(3), Integer(2), Integer(7),␣
↪range(Integer(7)), [[Integer(0), Integer(1), Integer(2)],
...     [Integer(0), Integer(3), Integer(4)], [Integer(0), Integer(5),␣
↪Integer(6)], [Integer(1), Integer(3), Integer(5)], [Integer(1), Integer(4),␣
↪Integer(6)],
...     [Integer(2), Integer(3), Integer(6)], [Integer(2), Integer(4),␣
↪Integer(5)]], Integer(0), 'Projective Plane')
>>> C.v()
7
```

sage.combinat.designs.covering_design.**best_known_covering_design_www**(*v*, *k*, *t*, *verbose=False*)

Return the best known (v, k, t) covering design from an online database.

This uses the La Jolla Covering Repository, a database available at https://ljcr.dmgordon.org/cover.html

INPUT:

- v – integer; the size of the point set for the design
- k – integer; the number of points per block
- t – integer; the size of sets covered by the blocks
- verbose – boolean (default: False); print verbose message

OUTPUT:

A *CoveringDesign* object representing the (v, k, t)-covering design with smallest number of blocks available in the database.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import (   # optional - internet
....:     best_known_covering_design_www)
sage: C = best_known_covering_design_www(7, 3, 2)   # optional - internet
sage: print(C)                                      # optional - internet
C(7, 3, 2) = 7
Method: lex covering
Submitted on: 1996-12-01 00:00:00
0  1  2
0  3  4
0  5  6
1  3  5
1  4  6
2  3  6
2  4  5
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import (   # optional - internet
...     best_known_covering_design_www)
>>> C = best_known_covering_design_www(Integer(7), Integer(3), Integer(2))   #␣
↪optional - internet
>>> print(C)                                      # optional - internet
C(7, 3, 2) = 7
Method: lex covering
Submitted on: 1996-12-01 00:00:00
0  1  2
0  3  4
0  5  6
1  3  5
1  4  6
2  3  6
2  4  5
```

A ValueError is raised if the (v, k, t) parameters are not found in the database.

sage.combinat.designs.covering_design.**schonheim**(*v*, *k*, *t*)

Return the Schonheim lower bound for the size of such a covering design.

INPUT:

- v – integer; size of point set
- k – integer; cardinality of each block
- t – integer; cardinality of sets being covered

OUTPUT:

The Schonheim lower bound for such a covering design's size: $C(v,k,t) \leq \lceil (\frac{v}{k} \lceil \frac{v-1}{k-1} \cdots \lceil \frac{v-t+1}{k-t+1} \rceil \cdots \rceil \rceil$

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import schonheim
sage: schonheim(10, 3, 2)
17
sage: schonheim(32, 16, 8)
930
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.covering_design import schonheim
>>> schonheim(Integer(10), Integer(3), Integer(2))
17
>>> schonheim(Integer(32), Integer(16), Integer(8))
930
```

sage.combinat.designs.covering_design.**trivial_covering_design**(*v*, *k*, *t*)

Construct a trivial covering design.

INPUT:

- v – integer; size of point set
- k – integer; cardinality of each block
- t – integer; cardinality of sets being covered

OUTPUT:

A trivial (v,k,t) covering design

EXAMPLES:

```
sage: C = trivial_covering_design(8, 3, 1)
sage: print(C)
C(8, 3, 1) = 3
Method: Trivial
0   1   2
0   6   7
3   4   5
sage: C = trivial_covering_design(5, 3, 2)
sage: print(C)
4 <= C(5, 3, 2) <= 10
Method: Trivial
0   1   2
0   1   3
0   1   4
0   2   3
```

(continues on next page)

(continued from previous page)

```
0   2   4
0   3   4
1   2   3
1   2   4
1   3   4
2   3   4
```

```
>>> from sage.all import *
>>> C = trivial_covering_design(Integer(8), Integer(3), Integer(1))
>>> print(C)
C(8, 3, 1) = 3
Method: Trivial
0   1   2
0   6   7
3   4   5
>>> C = trivial_covering_design(Integer(5), Integer(3), Integer(2))
>>> print(C)
4 <= C(5, 3, 2) <= 10
Method: Trivial
0   1   2
0   1   3
0   1   4
0   2   3
0   2   4
0   3   4
1   2   3
1   2   4
1   3   4
2   3   4
```

Note

Cases are:

- $t = 0$: This could be empty, but it's a useful convention to have one block (which is empty if $k = 0$).
- $t = 1$: This contains $\lceil v/k \rceil$ blocks: $[0, ..., k - 1], [k, ..., 2k - 1], ...$. The last block wraps around if k does not divide v.
- anything else: Just use every k-subset of $[0, 1, ..., v - 1]$.

5.1.81 Database of small combinatorial designs

This module implements combinatorial designs that cannot be obtained by more general constructions. Most of them come from the Handbook of Combinatorial Designs [DesignHandbook].

All this would only be a dream without the mathematical knowledge and help of Julian R. Abel.

These functions can all be obtained through the `designs.<tab>` functions.

This module implements:

- `CA(11,2,5,3)`, `CA(12,2,7,3)`, `CA(13,2,9,3)`, `CA(14,2,10,3)`, `CA(15,2,20,3)`, `CA(19,2,6,4)`, `CA(21,2,7,4)`, `CA(29,2,7,5)`, `CA(37,2,4,6)`, `CA(39,2,5,6)`, `CA(41,2,6,6)`

- OA(7,18), OA(9,40), OA(7,66), OA(7,68), OA(8,69), OA(7,74), OA(8,76), OA(11,80), OA(15,112), OA(9,120), OA(9,135), OA(11,160), OA(16,176), OA(11,185), OA(10,205), OA(16,208), OA(15,224), OA(11,254), OA(20,352), OA(20,416), OA(10,469), OA(10,520), OA(12,522), OA(14,524), OA(20,544), OA(17,560), OA(11,640), OA(10,796), OA(15,896), OA(9,1078), OA(25,1262), OA(9,1612), OA(10,1620)
- 2 MOLS of order 10, 5 MOLS of order 12, 4 MOLS of order 14, 4 MOLS of order 15, 3 MOLS of order 18
- $V(m,t)$ **vectors:**
  - $m = 3$ and $t = 2, 4, 6, 10, 12, 14, 20, 24, 26, 32, 34$
  - $m = 4$ and $t = 3, 7, 9, 13, 15, 25$
  - $m = 5$ and $t = 6, 8, 12, 14, 20, 26$
  - $m = 6$ and $t = 5, 7, 11, 13, 17, 21$
  - $m = 7$ and $t = 6, 10, 16, 18$
  - $m = 8$ and $t = 9, 11, 17, 29, 57$
  - $m = 9$ and $t = 12, 14, 18, 20, 22, 30, 34, 42, 44$
  - $m = 10$ and $t = 13, 15, 19, 21, 25, 27, 31, 33, 43, 49, 81, 97, 103, 181, 187, 259, 273, 319, 391, 409$
  - $m = 11$ and $t = 30, 32, 36, 38, 42, 56, 60, 62, 66, 78, 80, 86, 90, 92, 102, 116, 120, 128, 132, 146, 162, 170, 182, 188, 192, 198, 206, 210, 212, 216, 218, 230, 242, 246, 248, 260, 266, 270, 276, 288, 290, 296, 300, 302, 308, 312, 318, 330, 336, 338, 350, 356, 366, 368, 372, 378, 396, 402, 420, 422, 450, 452$
  - $m = 12$ and $t = 33, 35, 45, 51, 55, 59, 61, 63, 69, 71, 73, 83, 85, 89, 91, 93, 101, 103, 115, 119, 121, 129, 133, 135, 139, 141, 145, 149, 155, 161, 169, 171, 185, 189, 191, 195, 199, 203, 213, 223, 229, 233, 243, 253, 255, 259, 265, 269, 271, 275, 281, 289, 293, 295, 301, 303, 309, 311, 321, 323, 335, 341, 355, 363, 379, 383, 385, 399, 401, 405, 409, 411, 413$
- RBIBD(120,8,1)
- (v,k,λ)**-BIBD:**
  - $\lambda = 1$:n $(66,6,1)$, $(76,6,1)$, $(96,6,1)$, $(106,6,1)$, $(111,6,1)$, $(120,8,1)$, $(126,6,1)$, $(136,6,1)$, $(141,6,1)$, $(171,6,1)$, $(196,6,1)$, $(201,6,1)$
  - $\lambda = 2$:n $(56,11,2)$, $(79,13,2)$
  - $\lambda = 8$:n $(45,9,8)$
  - $\lambda = 14$:n $(176,50,14)$
- (v,k,λ)**-difference families:**
  - $\lambda = 1$:n $(15,3,1)$, $(21,3,1)$, $(21,5,1)$, $(25,3,1)$, $(25,4,1)$, $(27,3,1)$, $(33,3,1)$, $(37,4,1)$, $(39,3,1)$, $(40,4,1)$, $(45,3,1)$, $(45,5,1)$, $(49,3,1)$, $(49,4,1)$, $(51,3,1)$, $(52,4,1)$, $(55,3,1)$, $(57,3,1)$, $(63,3,1)$, $(64,4,1)$, $(65,5,1)$, $(69,3,1)$, $(75,3,1)$, $(76,4,1)$, $(81,3,1)$, $(81,5,1)$, $(85,4,1)$, $(91,6,1)$, $(91,7,1)$, $(121,5,1)$, $(121,6,1)$, $(141,5,1)$, $(161,5,1)$, $(175,7,1)$, $(201,5,1)$, $(217,7,1)$, $(221,5,1)$, $(259,7,1)$
  - $\lambda = 2$:n $(16,3,2)$, $(19,4,2)$, $(22,4,2)$, $(28,3,2)$, $(31,4,2)$, $(31,5,2)$, $(34,4,2)$, $(35,5,2)$, $(40,3,2)$, $(43,4,2)$, $(43,7,2)$, $(46,4,2)$, $(46,6,2)$, $(51,5,2)$, $(61,6,2)$, $(64,7,2)$, $(71,5,2)$, $(75,5,2)$, $(85,7,2)$, $(85,8,2)$, $(153,9,2)$, $(181,10,2)$
  - $\lambda = 3$:n $(21,4,3)$, $(21,6,3)$, $(29,7,3)$, $(41,6,3)$, $(43,7,3)$, $(45,12,3)$, $(49,9,3)$, $(51,6,3)$, $(57,7,3)$, $(61,6,3)$, $(61,10,3)$, $(71,7,3)$, $(85,7,3)$, $(97,9,3)$, $(121,10,3)$

  - $\lambda = 4$:n $(22,7,4)$, $(29,8,4)$, $(43,8,4)$, $(46,10,4)$, $(55,9,4)$, $(67,12,4)$, $(71,8,4)$
  - $\lambda = 5$:n $(13,5,5)$, $(17,5,5)$, $(21,6,5)$, $(22,6,5)$, $(28,6,5)$, $(33,5,5)$, $(33,6,5)$, $(37,10,5)$, $(39,6,5)$, $(45,11,5)$, $(46,10,5)$, $(55,10,5)$, $(67,11,5)$, $(73,10,5)$
  - $\lambda = 6$:n $(11,4,6)$, $(15,4,6)$, $(15,5,6)$, $(29,8,6)$, $(46,10,6)$, $(53,13,6)$, $(67,12,6)$
  - $\lambda = 7$:n $(25,7,7)$, $(53,14,7)$, $(61,15,7)$
  - $\lambda = 8$:n $(22,8,8)$, $(34,12,8)$, $(133,33,8)$
  - $\lambda = 9$:n $(21,10,9)$
  - $\lambda = 10$:n $(34,12,10)$, $(43,15,10)$, $(49,21,10)$
  - $\lambda = 12$:n $(22,8,12)$
  - $\lambda = 14$:n $(21,8,14)$
  - $\lambda = 24$:n $(320,88,24)$
  - $\lambda = 30$:n $(144,66,30)$
  - $\lambda = 56$:n $(901,225,56)$

- (v,k,λ)**-difference matrices:**
  - $\lambda = 1$:n $(12,6,1)$, $(21,6,1)$, $(24,8,1)$, $(28,6,1)$, $(33,6,1)$, $(35,6,1)$, $(36,9,1)$, $(39,6,1)$, $(44,6,1)$, $(45,7,1)$, $(48,9,1)$, $(51,6,1)$, $(52,6,1)$, $(55,7,1)$, $(56,8,1)$, $(57,8,1)$, $(60,6,1)$, $(75,8,1)$, $(273,17,1)$, $(993,32,1)$
- $(n,k;\lambda,\mu;u)$**-quasi-difference matrices:**
  $(19,6;1,1;1)$, $(21,5;1,1;1)$, $(21,6;1,1;5)$, $(25,6;1,1;5)$, $(33,6;1,1;1)$, $(35,7;1,1;7)$, $(37,6;1,1;1)$, $(45,7;1,1;9)$, $(54,7;1,1;8)$, $(57,9;1,1;8)$
- (q,k) **evenly distributed sets**
  - $k = 4$: 13, 25, 37, 49, 61, 73, 97, 109, 121, 157, 169, 181, 193, 229, 241, 277, 289, 313, 337, 349, 361, 373, 397, 409, 421, 433, 457, 529, 541, 577, 601, 613, 625, 661, 673, 709, 733, 757, 769, 829, 841, 853, 877, 937, 961, 997, 1009, 1021, 1033, 1069, 1093, 1117, 1129, 1153, 1201, 1213, 1237, 1249, 1297, 1321, 1369, 1381, 1429, 1453, 1489, 1549, 1597, 1609, 1621, 1657, 1669, 1681, 1693, 1741, 1753, 1777, 1789, 1801, 1849, 1861, 1873, 1933, 1993, 2017, 2029, 2053, 2089, 2113, 2137, 2161, 2197, 2209, 2221, 2269, 2281, 2293, 2341, 2377, 2389, 2401, 2437, 2473, 2521, 2557, 2593, 2617, 2677, 2689, 2713, 2749, 2797, 2809
  - $k = 5$: 41, 61, 101, 121, 181, 241, 281, 361, 401, 421, 461, 521, 541, 601, 641, 661, 701, 761, 821, 841, 881, 941, 961, 1021, 1061, 1181, 1201, 1301, 1321, 1361, 1381, 1481, 1601, 1621, 1681, 1721, 1741, 1801, 1861, 1901
  - $k = 6$: 31, 151, 181, 211, 241, 271, 331, 361, 421, 541, 571, 601, 631, 661, 691, 751, 811, 841, 961, 991, 1021, 1051, 1171, 1201, 1231, 1291, 1321, 1381, 1471, 1531, 1621, 1681, 1741, 1801, 1831, 1861, 1951
  - $k = 7$: 169, 337, 379, 421, 463, 547, 631, 673, 757, 841, 883, 967, 1009, 1051, 1093, 1303, 1429, 1471, 1597, 1681, 1723, 1849, 1933
  - $k = 8$: 449, 617, 673, 729, 841, 953, 1009, 1289, 1681, 1849
  - $k = 9$: 73, 433, 937, 1009, 1153, 1297, 1369, 1657, 1801, 1873
  - $k = 10$: 1171, 1531, 1621, 1801

REFERENCES:

Functions

`sage.combinat.designs.database.BIBD_106_6_1()`

Return a (106,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_106_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(106, BIBD_106_6_1())
(106,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_106_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(106), BIBD_106_6_1())
(106,6,1)-Balanced Incomplete Block Design
```

`sage.combinat.designs.database.BIBD_111_6_1()`

Return a (111,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_111_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(111, BIBD_111_6_1())
(111,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_111_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(111), BIBD_111_6_1())
(111,6,1)-Balanced Incomplete Block Design
```

`sage.combinat.designs.database.BIBD_126_6_1()`

Return a (126,6,1)-BIBD.

This constructions appears in VI.16.92 from [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_126_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(126, BIBD_126_6_1())
(126,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_126_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(126), BIBD_126_6_1())
(126,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_136_6_1**()

Return a (136,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_136_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(136, BIBD_136_6_1())
(136,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_136_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(136), BIBD_136_6_1())
(136,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_141_6_1**()

Return a (141,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_141_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(141, BIBD_141_6_1())
(141,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_141_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(141), BIBD_141_6_1())
(141,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_171_6_1**()

Return a (171,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_171_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(171, BIBD_171_6_1())
(171,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_171_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(171), BIBD_171_6_1())
(171,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_196_6_1**()

Return a (196,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_196_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(196, BIBD_196_6_1())
(196,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_196_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(196), BIBD_196_6_1())
(196,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_201_6_1**()

Return a (201,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_201_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(201, BIBD_201_6_1())
(201,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_201_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(201), BIBD_201_6_1())
(201,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_45_9_8**(*from_code=False*)

Return a $(45, 9, 1)$-BIBD.

This BIBD is obtained from the codewords of minimal weight in the `ExtendedQuadraticResidueCode()` of length 48. This construction appears in VII.11.2 from [DesignHandbook], which cites [HT95].

INPUT:

- `from_code` – boolean; whether to build the design from hardcoded data (default) or from the code object (much longer)

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_45_9_8
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: B = BalancedIncompleteBlockDesign(45, BIBD_45_9_8(), lambd=8); B
(45,9,8)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_45_9_8
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> B = BalancedIncompleteBlockDesign(Integer(45), BIBD_45_9_8(),␣
↪lambd=Integer(8)); B
(45,9,8)-Balanced Incomplete Block Design
```

REFERENCE:

`sage.combinat.designs.database.BIBD_56_11_2()`

Return a symmetric $(56, 11, 2)$-BIBD.

The construction implemented is given in [Hall71].

> **Note**
>
> A symmetric (v, k, λ) BIBD is a (v, k, λ) BIBD with v blocks.

EXAMPLES:

sage: from sage.combinat.designs.database import BIBD_56_11_2 sage: D = IncidenceStructure(BIBD_56_11_2()) # needs sage.libs.gap sage: D.is_t_design(t=2, v=56, k=11, l=2) # needs sage.libs.gap True

`sage.combinat.designs.database.BIBD_66_6_1()`

Return a (66,6,1)-BIBD.

This BIBD was obtained from La Jolla covering repository (https://dmgordon.org/cover) where it is attributed to Colin Barker.

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_66_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(66, BIBD_66_6_1())
(66,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_66_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(66), BIBD_66_6_1())
(66,6,1)-Balanced Incomplete Block Design
```

`sage.combinat.designs.database.BIBD_76_6_1()`

Return a (76,6,1)-BIBD.

This BIBD was obtained from La Jolla covering repository (https://dmgordon.org/cover) where it is attributed to Colin Barker.

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_76_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(76, BIBD_76_6_1())
(76,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_76_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(76), BIBD_76_6_1())
(76,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_79_13_2**()

Return a symmetric $(79, 13, 2)$-BIBD.

The construction implemented is the one described in [Aschbacher71]. A typo in that paper was corrected in [Hall71].

Note

A symmetric (v, k, λ) BIBD is a (v, k, λ) BIBD with v blocks.

EXAMPLES:

sage: from sage.combinat.designs.database import BIBD_79_13_2 sage: D = IncidenceStructure(BIBD_79_13_2()) # needs sage.libs.gap sage: D.is_t_design(t=2, v=79, k=13, l=2) # needs sage.libs.gap True

sage.combinat.designs.database.**BIBD_96_6_1**()

Return a (96,6,1)-BIBD.

This BIBD was obtained from La Jolla covering repository (https://dmgordon.org/cover) where it is attributed to Colin Barker.

EXAMPLES:

```
sage: from sage.combinat.designs.database import BIBD_96_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(96, BIBD_96_6_1())
(96,6,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import BIBD_96_6_1
>>> from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
>>> BalancedIncompleteBlockDesign(Integer(96), BIBD_96_6_1())
(96,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**DM_12_6_1**()

Return a $(12, 6, 1)$-difference matrix as built in [Hanani75].

This design is Lemma 3.21 from [Hanani75].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_12_6_1
sage: G,M = DM_12_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_12_6_1
>>> G,M = DM_12_6_1()
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(12,6)
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(12),Integer(6))
```

REFERENCES:

sage.combinat.designs.database.**DM_21_6_1**()

Return a $(21, 6, 1)$-difference matrix.

As explained in the Handbook III.3.50 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_21_6_1
sage: G,M = DM_21_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_21_6_1
>>> G,M = DM_21_6_1()
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(21,6)                                     #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(21),Integer(6))                     ␣
↪               # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_24_8_1**()

Return a $(24, 8, 1)$-difference matrix.

As explained in the Handbook III.3.52 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_24_8_1
sage: G,M = DM_24_8_1()
```

(continues on next page)

(continued from previous page)

```
sage: is_difference_matrix(M,G,8,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_24_8_1
>>> G,M = DM_24_8_1()
>>> is_difference_matrix(M,G,Integer(8),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(24,8)
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(24),Integer(8))
```

sage.combinat.designs.database.**DM_273_17_1**()

Return a $(273, 17, 1)$-difference matrix.

Given by Julian R. Abel.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_273_17_1
sage: G,M = DM_273_17_1()                                                    #␣
↪needs sage.schemes
sage: is_difference_matrix(M,G,17,1)                                         #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_273_17_1
>>> G,M = DM_273_17_1()                                                    #␣
↪needs sage.schemes
>>> is_difference_matrix(M,G,Integer(17),Integer(1))                        ␣
↪               # needs sage.schemes
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(273,17)                                  #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(273),Integer(17))                  ␣
↪               # needs sage.schemes
```

sage.combinat.designs.database.**DM_28_6_1**()

Return a $(28, 6, 1)$-difference matrix.

As explained in the Handbook III.3.54 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_28_6_1
sage: G,M = DM_28_6_1()                                                        #␣
↪needs sage.modules
sage: is_difference_matrix(M,G,6,1)                                            #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_28_6_1
>>> G,M = DM_28_6_1()                                                        #␣
↪needs sage.modules
>>> is_difference_matrix(M,G,Integer(6),Integer(1))                          ␣
↪              # needs sage.modules
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(28,6)                                      #␣
↪needs sage.modules
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(28),Integer(6))                    ␣
↪              # needs sage.modules
```

sage.combinat.designs.database.**DM_33_6_1**()

Return a $(33, 6, 1)$-difference matrix.

As explained in the Handbook III.3.56 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_33_6_1
sage: G,M = DM_33_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_33_6_1
>>> G,M = DM_33_6_1()
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(33,6)                                      #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(33),Integer(6))                    ␣
↪              # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_35_6_1**()

Return a $(35, 6, 1)$-difference matrix.

As explained in the Handbook III.3.58 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_35_6_1
sage: G,M = DM_35_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_35_6_1
>>> G,M = DM_35_6_1()
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(35,6)                                     #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(35),Integer(6))                    ␣
↪              # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_36_9_1**()

Return a $(36, 9, 1)$-difference matrix.

As explained in the Handbook III.3.59 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_36_9_1
sage: G,M = DM_36_9_1()                                                       #␣
↪needs sage.modules
sage: is_difference_matrix(M,G,9,1)                                           #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_36_9_1
>>> G,M = DM_36_9_1()                                                       #␣
↪needs sage.modules
>>> is_difference_matrix(M,G,Integer(9),Integer(1))                          ␣
```

(continues on next page)

(continued from previous page)

```
↪              # needs sage.modules
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(36,9)                                   #␣
↪needs sage.modules
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(36),Integer(9))                    ␣
↪              # needs sage.modules
```

sage.combinat.designs.database.**DM_39_6_1**()

Return a $(39, 6, 1)$-difference matrix.

As explained in the Handbook III.3.61 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_39_6_1
sage: G,M = DM_39_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_39_6_1
>>> G,M = DM_39_6_1()
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

The design is available from the general constructor:

```
sage: designs.difference_matrix(39,6,existence=True)                        #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> designs.difference_matrix(Integer(39),Integer(6),existence=True)         ␣
↪              # needs sage.rings.finite_rings
True
```

sage.combinat.designs.database.**DM_44_6_1**()

Return a $(44, 6, 1)$-difference matrix.

As explained in the Handbook III.3.64 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_44_6_1
sage: G,M = DM_44_6_1()
```

(continues on next page)

(continued from previous page)

```
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_44_6_1
>>> G,M = DM_44_6_1()
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(44,6)
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(44),Integer(6))
```

sage.combinat.designs.database.**DM_45_7_1**()

Return a $(45, 7, 1)$-difference matrix.

As explained in the Handbook III.3.65 [DesignHandbook].

… whose description contained a very deadly typo, kindly fixed by Julian R. Abel.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_45_7_1
sage: G,M = DM_45_7_1()
sage: is_difference_matrix(M,G,7,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_45_7_1
>>> G,M = DM_45_7_1()
>>> is_difference_matrix(M,G,Integer(7),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(45,7)                                   #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(45),Integer(7))                   ␣
↪               # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_48_9_1**()

Return a $(48, 9, 1)$-difference matrix.

As explained in the Handbook III.3.67 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_48_9_1
sage: G,M = DM_48_9_1()                                                    #␣
↪needs sage.rings.finite_rings
sage: is_difference_matrix(M,G,9,1)                                        #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_48_9_1
>>> G,M = DM_48_9_1()                                                    #␣
↪needs sage.rings.finite_rings
>>> is_difference_matrix(M,G,Integer(9),Integer(1))                        ␣
↪               # needs sage.rings.finite_rings
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(48,9)                                  #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(48),Integer(9))                  ␣
↪               # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_51_6_1**()

Return a $(51, 6, 1)$-difference matrix.

As explained in the Handbook III.3.69 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_51_6_1
sage: G,M = DM_51_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_51_6_1
>>> G,M = DM_51_6_1()
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(51,6)                                  #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(51),Integer(6))                    ␣
↪             # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_52_6_1**()

Return a $(52, 6, 1)$-difference matrix.

As explained in the Handbook III.3.70 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_52_6_1
sage: G,M = DM_52_6_1()                                                   #␣
↪needs sage.rings.finite_rings
sage: is_difference_matrix(M,G,6,1)                                       #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_52_6_1
>>> G,M = DM_52_6_1()                                                    #␣
↪needs sage.rings.finite_rings
>>> is_difference_matrix(M,G,Integer(6),Integer(1))                      ␣
↪             # needs sage.rings.finite_rings
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(52,6)                                 #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(52),Integer(6))                    ␣
↪             # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_55_7_1**()

Return a $(55, 7, 1)$-difference matrix.

As explained in the Handbook III.3.72 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_55_7_1
sage: G,M = DM_55_7_1()
sage: is_difference_matrix(M,G,7,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_55_7_1
>>> G,M = DM_55_7_1()
```

(continues on next page)

(continued from previous page)

```
>>> is_difference_matrix(M,G,Integer(7),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(55,7)                              #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(55),Integer(7))              ␣
↪               # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_56_8_1**()

Return a $(56, 8, 1)$-difference matrix.

As explained in the Handbook III.3.73 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_56_8_1
sage: G,M = DM_56_8_1()                                                #␣
↪needs sage.rings.finite_rings
sage: is_difference_matrix(M,G,8,1)                                    #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_56_8_1
>>> G,M = DM_56_8_1()                                                #␣
↪needs sage.rings.finite_rings
>>> is_difference_matrix(M,G,Integer(8),Integer(1))                    ␣
↪               # needs sage.rings.finite_rings
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(56,8)                              #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(56),Integer(8))              ␣
↪               # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_57_8_1**()

Return a $(57, 8, 1)$-difference matrix.

Given by Julian R. Abel.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_57_8_1
sage: G,M = DM_57_8_1()                                                     #␣
↪needs sage.rings.finite_rings sage.schemes
sage: is_difference_matrix(M,G,8,1)                                         #␣
↪needs sage.rings.finite_rings sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_57_8_1
>>> G,M = DM_57_8_1()                                                       #␣
↪needs sage.rings.finite_rings sage.schemes
>>> is_difference_matrix(M,G,Integer(8),Integer(1))                          ␣
↪               # needs sage.rings.finite_rings sage.schemes
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(57,8)                                   #␣
↪needs sage.rings.finite_rings sage.schemes
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(57),Integer(8))                    ␣
↪               # needs sage.rings.finite_rings sage.schemes
```

sage.combinat.designs.database.**DM_60_6_1**()

Return a $(60, 6, 1)$-difference matrix.

As explained in [JulianAbel13].

REFERENCES:

http://onlinelibrary.wiley.com/doi/10.1002/jcd.21384/abstract

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_60_6_1
sage: G,M = DM_60_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_60_6_1
>>> G,M = DM_60_6_1()
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(60,6)
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(60),Integer(6))
```

sage.combinat.designs.database.**DM_75_8_1**()

Return a $(75, 8, 1)$-difference matrix.

As explained in the Handbook III.3.75 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_75_8_1
sage: G,M = DM_75_8_1()
sage: is_difference_matrix(M,G,8,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_75_8_1
>>> G,M = DM_75_8_1()
>>> is_difference_matrix(M,G,Integer(8),Integer(1))
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(75,8)                                    #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(75),Integer(8))                    ␣
↪              # needs sage.rings.finite_rings
```

sage.combinat.designs.database.**DM_993_32_1**()

Return a $(993, 32, 1)$-difference matrix.

Given by Julian R. Abel.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_993_32_1
sage: G,M = DM_993_32_1()                                                    #␣
↪needs sage.schemes
sage: is_difference_matrix(M,G,32,1)                                         #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> from sage.combinat.designs.database import DM_993_32_1
>>> G,M = DM_993_32_1()                                                    #␣
↪needs sage.schemes
>>> is_difference_matrix(M,G,Integer(32),Integer(1))                        ␣
```

(continues on next page)

(continued from previous page)

```
↪                # needs sage.schemes
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(993,32)                                    #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> _ = designs.difference_matrix(Integer(993),Integer(32))                    ␣
↪                # needs sage.schemes
```

sage.combinat.designs.database.**HigmanSimsDesign**()

Return the Higman-Sims designs, which is a $(176, 50, 14)$-BIBD.

This design is built from a from the `WittDesign` W on 24 points. We define two points a, b, and consider:

- The collection W_a of all blocks of W containing a but not containing b.
- The collection W_b of all blocks of W containing b but not containing a.

The design is then obtained from the incidence structure produced by the blocks $A \in W_a$ and $B \in W_b$ whose intersection has cardinality 2. This construction, due to M.Smith, can be found in [KY04] or in 10.A.(v) of [BL1984].

EXAMPLES:

```
sage: H = designs.HigmanSimsDesign(); H                        # optional - gap_
↪package_design
Incidence structure with 176 points and 176 blocks
sage: H.is_t_design(return_parameters=1)                       # optional - gap_
↪package_design
(True, (2, 176, 50, 14))
```

```
>>> from sage.all import *
>>> H = designs.HigmanSimsDesign(); H                        # optional - gap_
↪package_design
Incidence structure with 176 points and 176 blocks
>>> H.is_t_design(return_parameters=Integer(1))                  # optional -␣
↪gap_package_design
(True, (2, 176, 50, 14))
```

Make sure that the automorphism group of this designs is isomorphic to the automorphism group of the `HigmanSimsGraph()`. Note that the first of those permutation groups acts on 176 points, while the second acts on 100:

```
sage: gH = H.automorphism_group()                              # optional - gap_
↪package_design
sage: gG = graphs.HigmanSimsGraph().automorphism_group()     # optional - gap_
↪package_design
sage: gG.is_isomorphic(gG)                          # long time, optional - gap_
↪package_design
True
```

```
>>> from sage.all import *
>>> gH = H.automorphism_group()                          # optional - gap_
↪package_design
>>> gG = graphs.HigmanSimsGraph().automorphism_group()    # optional - gap_
↪package_design
>>> gG.is_isomorphic(gG)                       # long time, optional - gap_
↪package_design
True
```

REFERENCE:

`sage.combinat.designs.database.`**`MOLS_10_2`**`()`

Return a pair of MOLS of order 10.

Data obtained from http://www.cecm.sfu.ca/organics/papers/lam/paper/html/POLS10/POLS10.html

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_
↪latin_squares
sage: from sage.combinat.designs.database import MOLS_10_2
sage: MOLS = MOLS_10_2()                                                    #␣
↪needs sage.modules
sage: print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_
↪squares
>>> from sage.combinat.designs.database import MOLS_10_2
>>> MOLS = MOLS_10_2()                                                    #␣
↪needs sage.modules
>>> print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(2,10)
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(2),Integer(10))
True
```

`sage.combinat.designs.database.`**`MOLS_12_5`**`()`

Return 5 MOLS of order 12.

These MOLS have been found by Brendan McKay.

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_
↪latin_squares
```

(continues on next page)

(continued from previous page)

```
sage: from sage.combinat.designs.database import MOLS_12_5
sage: MOLS = MOLS_12_5()                                                    #␣
↪needs sage.modules
sage: print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_
↪squares
>>> from sage.combinat.designs.database import MOLS_12_5
>>> MOLS = MOLS_12_5()                                                    #␣
↪needs sage.modules
>>> print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

sage.combinat.designs.database.**MOLS_14_4**()

Return four MOLS of order 14.

These MOLS were shared by Ian Wanless. The first proof of existence was given in [Todorov12].

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_
↪latin_squares
sage: from sage.combinat.designs.database import MOLS_14_4
sage: MOLS = MOLS_14_4()                                                    #␣
↪needs sage.modules
sage: print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_
↪squares
>>> from sage.combinat.designs.database import MOLS_14_4
>>> MOLS = MOLS_14_4()                                                    #␣
↪needs sage.modules
>>> print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(4,14)                          #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(4),Integer(14))          ␣
↪                # needs sage.schemes
True
```

REFERENCE:

sage.combinat.designs.database.**MOLS_15_4**()

Return 4 MOLS of order 15.

These MOLS were shared by Ian Wanless.

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_
↪latin_squares
sage: from sage.combinat.designs.database import MOLS_15_4
sage: MOLS = MOLS_15_4()                                                    #␣
↪needs sage.modules
sage: print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_
↪squares
>>> from sage.combinat.designs.database import MOLS_15_4
>>> MOLS = MOLS_15_4()                                                      #␣
↪needs sage.modules
>>> print(are_mutually_orthogonal_latin_squares(MOLS))                      #␣
↪needs sage.modules
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(4,15)                          #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(4),Integer(15))          ␣
↪                # needs sage.schemes
True
```

sage.combinat.designs.database.**MOLS_18_3**()

Return 3 MOLS of order 18.

These MOLS were shared by Ian Wanless.

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_
↪latin_squares
sage: from sage.combinat.designs.database import MOLS_18_3
sage: MOLS = MOLS_18_3()                                                    #␣
↪needs sage.modules
sage: print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_
↪squares
>>> from sage.combinat.designs.database import MOLS_18_3
>>> MOLS = MOLS_18_3()                                                    #␣
↪needs sage.modules
>>> print(are_mutually_orthogonal_latin_squares(MOLS))                    #␣
↪needs sage.modules
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(3,18)
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(3),Integer(18))
True
```

sage.combinat.designs.database.**OA_10_1620**()

Return an OA(10,1620).

This is obtained through the generalized Brouwer-van Rees construction. Indeed, $1620 = 144.11 + (36 = 4.9)$ and there exists an $OA(10, 153) - OA(10, 9)$.

> **Note**
>
> This function should be removed once *find_brouwer_van_rees_with_one_truncated_column()* can handle all incomplete orthogonal arrays obtained through *incomplete_orthogonal_array()*.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_1620
sage: OA = OA_10_1620()                           # not tested -- ~7s
sage: is_orthogonal_array(OA,10,1620,2) # not tested -- ~7s
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_10_1620
>>> OA = OA_10_1620()                           # not tested -- ~7s
>>> is_orthogonal_array(OA,Integer(10),Integer(1620),Integer(2)) # not tested -- ~
↪7s
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,1620)                     #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(10),Integer(1620))                    # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_10_205**()

Return an $OA(10, 205)$.

Julian R. Abel shared the following construction, which originally appeared in Theorem 8.7 of [Greig99], and can in Lemmas 5.14-5.16 of [ColDin01]:

> Consider a $PG(2, 4^2)$ containing a Baer subplane (i.e. a $PG(2, 4)$) B and a point $p \in B$. Among the $4^2 + 1 = 17$ lines of $PG(2, 4^2)$ containing p:
>
> - $4 + 1 = 5$ lines intersect B on 5 points
> - $4^2 - 4 = 12$ lines intersect B on 1 point
>
> As those lines are disjoint outside of B we can use them as groups to build a GDD on $16^2 + 16 + 1 - (4^4 + 4 + 1) = 252$ points. By keeping only 9 lines of the second kind, however, we obtain a $(204, \{9, 13, 17\})$-GDD of type 12^5.16^9.
>
> We complete it into a PBD by adding a block $g \cup \{204\}$ for each group g. We then build an OA from this PBD using the fact that all blocks of size 9 are disjoint.

See also

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_205
sage: OA = OA_10_205()                                                          # needs sage.schemes
sage: is_orthogonal_array(OA,10,205,2)                                          # needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_10_205
>>> OA = OA_10_205()                                                            # needs sage.schemes
>>> is_orthogonal_array(OA,Integer(10),Integer(205),Integer(2))                  # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,205)                            # needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(10),Integer(205))
↪                # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_10_469**()

Return an OA(10,469).

This construction appears in [Brouwer80]. It is based on the same technique used in *brouwer_separable_design()*.

Julian R. Abel's instructions:

> Brouwer notes that a cyclic $PG(2,37)$ (or $(1407,38,1)$-BIBD) can be obtained with a base block containing $13,9$, and 16 points in each residue class mod 3. Thus, by reducing the $PG(2,37)$ to its points congruent to $0 \pmod 3$ one obtains a $(469,\{9,13,16\})$-PBD which consists in 3 symmetric designs, i.e. 469 blocks of size 9, 469 blocks of size 13, and 469 blocks of size 16.
>
> For each block size s, one can build a matrix with size $s\times 469$ in which each block is a row, and such that each point of the PBD appears once per column. By multiplying a row of an $OA(9,s)-s.OA(9,1)$ with the rows of the matrix one obtains a parallel class of a resolvable $OA(9,469)$.
>
> Add to this the parallel class of all blocks $(0,0,...),(1,1,...),...$ to obtain a resolvable $OA(9,469)$ equivalent to an $OA(10,469)$.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_469
sage: OA = OA_10_469()                       # long time                           #
↪needs sage.schemes
sage: is_orthogonal_array(OA,10,469,2)   # long time                           #
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_10_469
>>> OA = OA_10_469()                       # long time                           #
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(10),Integer(469),Integer(2))   # long time
↪                      # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,469)                              #
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(10),Integer(469))
↪                # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_10_520**()

Return an OA(10,520).

This design is built by the slightly more general construction *OA_520_plus_x()*.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_520
sage: OA = OA_10_520()                                                    #␣
↪needs sage.schemes
sage: is_orthogonal_array(OA,10,520,2)                                    #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_10_520
>>> OA = OA_10_520()                                                    #␣
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(10),Integer(520),Integer(2))          ␣
↪                        # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,520)                      #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(10),Integer(520))      ␣
↪                 # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_10_796**()

Return an OA(10,796).

Construction shared by Julian R. Abel, from [AC07]:

> Truncate one block of a $TD(17,47)$ to size 13, then add an extra point. Form a block on each group plus the extra point: we obtain a $(796,\{13,16,17,47,48\})$-PBD in which only the extra point lies in more than one block of size 48 (and each other point lies in exactly 1 such block).
>
> For each block B (of size k say) not containing the extra point, construct a $TD(10,k)-k.TD(k,1)$ on $I(10)XB$. For each block B (of size $k=47$ or 48) containing the extra point, construct a $TD(10,k)-TD(k,1)$ on $I(10)XB$, the size 1 hole being on $I(10)XP$ where P is the extra point. Finally form 1 extra block of size 10 on $I(10)XP$.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_796
sage: OA = OA_10_796()                                                    #␣
↪needs sage.schemes
sage: is_orthogonal_array(OA,10,796,2)                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_10_796
>>> OA = OA_10_796()                                                    #␣
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(10),Integer(796),Integer(2))          ␣
↪                         # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,796)                     #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(10),Integer(796))     ␣
↪                # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_11_160**()

Return an OA(11,160).

Published by Julian R. Abel in [Ab1995]. Uses the fact that $160 = 2^5 \times 5$ is a product of a power of 2 and a prime number.

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_160
sage: OA = OA_11_160()                                                   #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,11,160,2)                                   #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_11_160
>>> OA = OA_11_160()                                                    #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(11),Integer(160),Integer(2))          ␣
↪                         # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,160)                          #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(11),Integer(160))           ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_11_185**()

Return an OA(11,185).

The construction is given in [Greig99]. In Julian R. Abel's words:

> Start with a $PG(2,16)$ with a 7 points Fano subplane; outside this plane there are $7(17-3)=98$ points on a line of the subplane and $273-98-7=168$ other points. Greig notes that the subdesign consisting of these 168 points is a $(168,\{10,12\})-PBD$. Now add the 17 points of a line disjoint from this subdesign (e.g. a line of the Fano subplane). This line will intersect every line of the 168 point subdesign in 1 point. Thus the new line sizes are 11 and 13, plus a unique line of size 17, giving a $(185,\{11,13,17\}$-PBD and an $OA(11,185)$.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_185
sage: OA = OA_11_185()                                                        #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,11,185,2)                                        #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_11_185
>>> OA = OA_11_185()                                                        #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(11),Integer(185),Integer(2))                ␣
↪                         # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,185)                          #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(11),Integer(185))           ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_11_254**()

Return an OA(11,254).

This constructions appears in [Greig99].

From a cyclic $PG(2,19)$ whose base blocks contains 7,9, and 4 points in the congruence classes mod 3, build a $(254,11,13,16)-PBD$ by ignoring the points of a congruence class. There exist $OA(12,11), OA(12,13), OA(12,16)$, which gives the $OA(11,254)$.

See also:

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_254
sage: OA = OA_11_254()                                                    #␣
↪needs sage.schemes
sage: is_orthogonal_array(OA,11,254,2)                                    #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_11_254
>>> OA = OA_11_254()                                                      #␣
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(11),Integer(254),Integer(2))          ␣
↪                        # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,254)                      #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(11),Integer(254))     ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_11_640**()

Return an OA(11,640).

Published by Julian R. Abel in [Ab1995] (uses the fact that $640 = 2^7 \times 5$ is the product of a power of 2 and a prime number).

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_640
sage: OA = OA_11_640()                          # not tested (too long)              #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,11,640,2)  # not tested (too long)              #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_11_640
>>> OA = OA_11_640()                          # not tested (too long)              #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(11),Integer(640),Integer(2))   # not tested␣
↪(too long)                  # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,640)                           #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(11),Integer(640))          ␣
↪              # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_11_80**()

Return an OA(11,80).

As explained in the Handbook III.3.76 [DesignHandbook]. Uses the fact that $80 = 2^4 \times 5$ and that 5 is prime.

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_80
sage: OA = OA_11_80()                                                            #␣
↪needs sage.rings.finite_rings sage.schemes
sage: is_orthogonal_array(OA,11,80,2)                                            #␣
↪needs sage.rings.finite_rings sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_11_80
>>> OA = OA_11_80()                                                            #␣
↪needs sage.rings.finite_rings sage.schemes
```

(continues on next page)

(continued from previous page)

```
>>> is_orthogonal_array(OA,Integer(11),Integer(80),Integer(2))                    # needs sage.rings.finite_rings sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,80)                    # needs sage.rings.finite_rings sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(11),Integer(80))                    # needs sage.rings.finite_rings sage.schemes
True
```

sage.combinat.designs.database.**OA_12_522**()

Return an OA(12,522).

This design is built by the slightly more general construction *OA_520_plus_x()*.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_12_522
sage: OA = OA_12_522()                    # needs sage.schemes
sage: is_orthogonal_array(OA,12,522,2)                    # needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_12_522
>>> OA = OA_12_522()                    # needs sage.schemes
>>> is_orthogonal_array(OA,Integer(12),Integer(522),Integer(2))                    # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(12,522)                    # needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(12),Integer(522))                    # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_14_524**()

Return an OA(14,524).

This design is built by the slightly more general construction *OA_520_plus_x()*.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_14_524
sage: OA = OA_14_524()                                                          #␣
↪needs sage.schemes
sage: is_orthogonal_array(OA,14,524,2)                                          #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_14_524
>>> OA = OA_14_524()                                                        #␣
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(14),Integer(524),Integer(2))               ␣
↪                         # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(14,524)                            #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(14),Integer(524))           ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_15_112**()

Return an OA(15,112).

Published by Julian R. Abel in [Ab1995]. Uses the fact that $112 = 2^4 \times 7$ and that 7 is prime.

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_15_112
sage: OA = OA_15_112()                                                          #␣
↪needs sage.rings.finite_rings sage.schemes
sage: is_orthogonal_array(OA,15,112,2)                                          #␣
↪needs sage.rings.finite_rings sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_15_112
```

(continues on next page)

(continued from previous page)

```
>>> OA = OA_15_112()                                                         #␣
↪needs sage.rings.finite_rings sage.schemes
>>> is_orthogonal_array(OA,Integer(15),Integer(112),Integer(2))              ␣
↪                        # needs sage.rings.finite_rings sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(15,112)                         #␣
↪needs sage.rings.finite_rings sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(15),Integer(112))         ␣
↪               # needs sage.rings.finite_rings sage.schemes
True
```

sage.combinat.designs.database.**OA_15_224**()

Return an OA(15,224).

Published by Julian R. Abel in [Ab1995] (uses the fact that $224 = 2^5 \times 7$ is a product of a power of 2 and a prime number).

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_15_224
sage: OA = OA_15_224()                           # not tested (too long)       #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,15,224,2)   # not tested (too long)               #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_15_224
>>> OA = OA_15_224()                           # not tested (too long)         #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(15),Integer(224),Integer(2))   # not tested␣
↪(too long)                 # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(15,224)                         #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(15),Integer(224))              # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_15_896**()

Return an OA(15,896).

Uses the fact that $896 = 2^7 \times 7$ is the product of a power of 2 and a prime number.

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_15_896
sage: OA = OA_15_896()                       # not tested (too long, ~2min)          # needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,15,896,2)   # not tested (too long)                  # needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_15_896
>>> OA = OA_15_896()                       # not tested (too long, ~2min)          # needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(15),Integer(896),Integer(2))  # not tested (too long)              # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(15,896)                              # needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(15),Integer(896))              # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_16_176**()

Return an OA(16,176).

Published by Julian R. Abel in [Ab1995]. Uses the fact that $176 = 2^4 \times 11$ is a product of a power of 2 and a prime number.

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_16_176
sage: OA = OA_16_176()                                                      #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,16,176,2)                                      #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_16_176
>>> OA = OA_16_176()                                                      #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(16),Integer(176),Integer(2))            ␣
↪                     # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(16,176)                        #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(16),Integer(176))        ␣
↪                # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_16_208**()

Return an OA(16,208).

Published by Julian R. Abel in [Ab1995]. Uses the fact that $208 = 2^4 \times 13$ is a product of 2 and a prime number.

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_16_208
sage: OA = OA_16_208()                  # not tested (too long)             #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,16,208,2)  # not tested (too long)             #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_16_208
>>> OA = OA_16_208()                    # not tested (too long)            #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(16),Integer(208),Integer(2))  # not tested␣
↪(too long)               # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(16,208)                         #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(16),Integer(208))          ␣
↪              # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_17_560**()

Return an OA(17,560).

This OA is built in Corollary 2.2 of [Thwarts].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_17_560
sage: OA = OA_17_560()                                                       #␣
↪needs sage.rings.finite_rings sage.schemes
sage: is_orthogonal_array(OA,17,560,2)                                       #␣
↪needs sage.rings.finite_rings sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_17_560
>>> OA = OA_17_560()                                                       #␣
↪needs sage.rings.finite_rings sage.schemes
>>> is_orthogonal_array(OA,Integer(17),Integer(560),Integer(2))             ␣
↪                       # needs sage.rings.finite_rings sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(17,560)                         #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(17),Integer(560))          ␣
↪              # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_20_352**()

Return an OA(20,352).

Published by Julian R. Abel in [Ab1995] (uses the fact that $352 = 2^5 \times 11$ is the product of a power of 2 and a prime number).

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_20_352
sage: OA = OA_20_352()                          # not tested (~25s)                #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,20,352,2)  # not tested (~25s)                #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_20_352
>>> OA = OA_20_352()                          # not tested (~25s)                #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(20),Integer(352),Integer(2))  # not tested (~
↪25s)                     # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(20,352)                           #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(20),Integer(352))           ␣
↪                # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_20_416**()

Return an OA(20,416).

Published by Julian R. Abel in [Ab1995] (uses the fact that $416 = 2^5 \times 13$ is the product of a power of 2 and a prime number).

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_20_416
sage: OA = OA_20_416()                        # not tested (~35s)                  #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,20,416,2)  # not tested                               #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_20_416
>>> OA = OA_20_416()                        # not tested (~35s)                  #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(20),Integer(416),Integer(2))   # not tested     ␣
↪                         # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(20,416)                               #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(20),Integer(416))               ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_20_544**()

Return an OA(20,544).

Published by Julian R. Abel in [Ab1995] (uses the fact that $544 = 2^5 \times 17$ is the product of a power of 2 and a prime number).

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_20_544
sage: OA = OA_20_544()                        # not tested (too long ~1mn)         #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,20,544,2)  # not tested                               #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_20_544
>>> OA = OA_20_544()                        # not tested (too long ~1mn)         #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(20),Integer(544),Integer(2))  # not tested    ␣
↪                       # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(20,544)                            #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(20),Integer(544))            ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_25_1262**()

Return an OA(25,1262).

The construction is given in [Greig99]. In Julian R. Abel's words:

> Start with a cyclic $PG(2,43)$ or $(1893,44,1)$-BIBD whose base block contains respectively 12, 13 and 19 point in the residue classes mod 3. In the resulting BIBD, remove one of the three classes: the result is a $(1262,\{25,31,32\})$-PBD, from which the $OA(25,1262)$ is obtained.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_25_1262
sage: OA = OA_25_1262()                             # not tested -- too long
sage: is_orthogonal_array(OA,25,1262,2) # not tested -- too long
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_25_1262
>>> OA = OA_25_1262()                             # not tested -- too long
>>> is_orthogonal_array(OA,Integer(25),Integer(1262),Integer(2)) # not tested --␣
↪too long
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(25,1262)                           #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(25),Integer(1262))           ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_520_plus_x**(x)

Return an $OA(10+x,520+x)$.

The construction shared by Julian R. Abel works for `OA(10,520)`, `OA(12,522)`, and `OA(14,524)`.

> Let $n = 520 + x$ and $k = 10 + x$. Build a $TD(17, 31)$. Remove $8 - x$ points contained in a common block, add a new point p and create a block $g_i \cup \{p\}$ for every (possibly truncated) group g_i. The result is a $(520 + x, 9 + x, 16, 17, 31, 32) - PBD$. Note that all blocks of size ≥ 30 only intersect on p, and that the unique block B_9 of size 9 intersects all blocks of size 32 on one point. Now:
>
> - Build an $OA(k, 16) - 16.OA(k, 16)$ for each block of size 16
> - Build an $OA(k, 17) - 17.OA(k, 17)$ for each block of size 17
> - Build an $OA(k, 31) - OA(k, 1)$ for each block of size 31 (with the hole on p).
> - Build an $OA(k, 32) - 2.OA(k, 1)$ for each block B of size 32 (with the holes on p and $B \cap B_9$).
> - Build an $OA(k, 9)$ on B_9.
>
> Only a row $[p, p, ...]$ is missing from the $OA(10 + x, 520 + x)$

This construction is used in `OA(10,520)`, `OA(12,522)`, and `OA(14,524)`.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_520_plus_x
sage: OA = OA_520_plus_x(0)                    # not tested (already tested in OA_
↪10_520)
sage: is_orthogonal_array(OA,10,520,2)  # not tested (already tested in OA_10_520)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_520_plus_x
>>> OA = OA_520_plus_x(Integer(0))                   # not tested (already tested␣
↪in OA_10_520)
>>> is_orthogonal_array(OA,Integer(10),Integer(520),Integer(2))  # not tested␣
↪(already tested in OA_10_520)
True
```

`sage.combinat.designs.database.`**`OA_7_18`**`()`

Return an OA(7,18).

Proved in [JulianAbel13].

See also

sage.combinat.designs.orthogonal_arrays.OA_from_quasi_difference_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_7_18
sage: OA = OA_7_18()
sage: is_orthogonal_array(OA,7,18,2)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_7_18
>>> OA = OA_7_18()
>>> is_orthogonal_array(OA,Integer(7),Integer(18),Integer(2))
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(7,18)                        #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(7),Integer(18))          ␣
↪                # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_7_66**()

Return an OA(7,66).

Construction shared by Julian R. Abel.

See also

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_7_66
sage: OA = OA_7_66()                                                      #␣
↪needs sage.schemes
sage: is_orthogonal_array(OA,7,66,2)                                      #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_7_66
>>> OA = OA_7_66()                                                      #␣
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(7),Integer(66),Integer(2))            ␣
↪                         # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(7,66)                        #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(7),Integer(66))
↪            # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_7_68**()

Return an OA(7,68).

Construction shared by Julian R. Abel.

See also

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_7_68
sage: OA = OA_7_68()                                                          #
↪needs sage.schemes
sage: is_orthogonal_array(OA,7,68,2)                                          #
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_7_68
>>> OA = OA_7_68()                                                          #
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(7),Integer(68),Integer(2))
↪                      # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(7,68)                            #
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(7),Integer(68))
↪            # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_7_74**()

Return an OA(7,74).

Construction shared by Julian R. Abel.

See also

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_7_74
sage: OA = OA_7_74()                                                    #␣
↪needs sage.schemes
sage: is_orthogonal_array(OA,7,74,2)                                    #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_7_74
>>> OA = OA_7_74()                                                    #␣
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(7),Integer(74),Integer(2))          ␣
↪                    # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(7,74)                      #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(7),Integer(74))     ␣
↪            # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_8_69**()

Return an OA(8,69).

Construction shared by Julian R. Abel.

See also

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_8_69
sage: OA = OA_8_69()                                                    #␣
↪needs sage.schemes
sage: is_orthogonal_array(OA,8,69,2)                                    #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_8_69
```

(continues on next page)

(continued from previous page)

```
>>> OA = OA_8_69()                                                          #␣
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(8),Integer(69),Integer(2))                  ␣
↪                       # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(8,69)                            #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(8),Integer(69))             ␣
↪              # needs sage.schemes
True
```

`sage.combinat.designs.database.`**`OA_8_76`**`()`

Return an OA(8,76).

Construction shared by Julian R. Abel.

See also

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_8_76
sage: OA = OA_8_76()                                                          #␣
↪needs sage.schemes
sage: is_orthogonal_array(OA,8,76,2)                                          #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_8_76
>>> OA = OA_8_76()                                                          #␣
↪needs sage.schemes
>>> is_orthogonal_array(OA,Integer(8),Integer(76),Integer(2))                  ␣
↪                       # needs sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(8,76)                            #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(8),Integer(76))              ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_9_1078**()

Return an OA(9,1078).

This is obtained through the generalized Brouwer-van Rees construction. Indeed, $1078 = 89.11 + (99 = 9.11)$ and there exists an $OA(9, 100) - OA(9, 11)$.

Note: This function should be removed once *find_brouwer_van_rees_with_one_truncated_column()* can handle all incomplete orthogonal arrays obtained through *incomplete_orthogonal_array()*.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_1078
sage: OA = OA_9_1078()                              # not tested -- ~3s
sage: is_orthogonal_array(OA,9,1078,2) # not tested -- ~3s
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_9_1078
>>> OA = OA_9_1078()                              # not tested -- ~3s
>>> is_orthogonal_array(OA,Integer(9),Integer(1078),Integer(2)) # not tested -- ~
↪3s
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,1078)                             #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(9),Integer(1078))              ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_9_120**()

Return an OA(9,120).

Construction shared by Julian R. Abel:

> From a resolvable $(120, 8, 1) - BIBD$, one can obtain 7 $MOLS(120)$ or a resolvable $TD(8, 120)$ by forming a resolvable $TD(8, 8) - 8.TD(8, 1)$ on $I_8 \times B$ for each block B in the BIBD. This gives a $TD(8, 120) - 120TD(8, 1)$ (which is resolvable as the BIBD is resolvable).

> **See also**
>
> *RBIBD_120_8_1()*

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_120
sage: OA = OA_9_120()                                                           #␣
↪needs sage.modules sage.schemes
sage: is_orthogonal_array(OA,9,120,2)                                           #␣
↪needs sage.modules sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_9_120
>>> OA = OA_9_120()                                                         #␣
↪needs sage.modules sage.schemes
>>> is_orthogonal_array(OA,Integer(9),Integer(120),Integer(2))                ␣
↪                         # needs sage.modules sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,120)                             #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(9),Integer(120))            ␣
↪               # needs sage.schemes
True
```

`sage.combinat.designs.database.`**`OA_9_135`**`()`

Return an OA(9,135).

Construction shared by Julian R. Abel:

> This design can be built by Wilson's method ($135 = 8.16 + 7$) applied to an Orthogonal Array $OA(9 + 7, 16)$ with 7 groups truncated to size 1 in such a way that a block contain 0, 1 or 3 points of the truncated groups.
>
> This is possible, because $PG(2, 2)$ (the projective plane over $GF(2)$) is a subdesign in $PG(2, 16)$ (the projective plane over $GF(16)$); in a cyclic $PG(2, 16)$ or $BIBD(273, 17, 1)$ the points $\equiv 0 \pmod{39}$ form such a subdesign (note that $273 = 16^2 + 16 + 1$ and $273 = 39 \times 7$ and $7 = 2^2 + 2 + 1$).

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_135
sage: OA = OA_9_135()                                                           #␣
↪needs sage.rings.finite_rings sage.schemes
sage: is_orthogonal_array(OA,9,135,2)                                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_9_135
>>> OA = OA_9_135()                                                          #␣
↪needs sage.rings.finite_rings sage.schemes
>>> is_orthogonal_array(OA,Integer(9),Integer(135),Integer(2))               ␣
↪                         # needs sage.rings.finite_rings sage.schemes
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,135)                          #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(9),Integer(135))          ␣
↪                # needs sage.schemes
True
```

As this orthogonal array requires a $(273, 17, 1)$ cyclic difference set, we check that it is available:

```
sage: G,D = designs.difference_family(273,17,1); G                           #␣
↪needs sage.libs.pari
Ring of integers modulo 273
```

```
>>> from sage.all import *
>>> G,D = designs.difference_family(Integer(273),Integer(17),Integer(1)); G  ␣
↪                         # needs sage.libs.pari
Ring of integers modulo 273
```

sage.combinat.designs.database.**OA_9_1612**()

Return an OA(9,1612).

This is obtained through the generalized Brouwer-van Rees construction. Indeed, $1612 = 89.17 + (99 = 9.11)$ and there exists an $OA(9, 100) - OA(9, 11)$.

> **Note**
>
> This function should be removed once *find_brouwer_van_rees_with_one_truncated_column()* can handle all incomplete orthogonal arrays obtained through *incomplete_orthogonal_array()*.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_1612
sage: OA = OA_9_1612()                              # not tested -- ~6s
```

(continues on next page)

(continued from previous page)

```
sage: is_orthogonal_array(OA,9,1612,2) # not tested -- ~6s
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_9_1612
>>> OA = OA_9_1612()                              # not tested -- ~6s
>>> is_orthogonal_array(OA,Integer(9),Integer(1612),Integer(2)) # not tested -- ~
↪6s
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,1612)                          #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(9),Integer(1612))          ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**OA_9_40**()

Return an OA(9,40).

As explained in the Handbook III.3.62 [DesignHandbook]. Uses the fact that $40 = 2^3 \times 5$ and that 5 is prime.

See also

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_40
sage: OA = OA_9_40()                                                          #␣
↪needs sage.rings.finite_rings
sage: is_orthogonal_array(OA,9,40,2)                                          #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.database import OA_9_40
>>> OA = OA_9_40()                                                          #␣
↪needs sage.rings.finite_rings
>>> is_orthogonal_array(OA,Integer(9),Integer(40),Integer(2))               ␣
↪                         # needs sage.rings.finite_rings
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,40)                              #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(9),Integer(40))              ␣
↪               # needs sage.schemes
True
```

sage.combinat.designs.database.**QDM_19_6_1_1_1**()

Return a $(19, 6; 1, 1; 1)$-quasi-difference matrix.

Used to build an $OA(6, 20)$

Given in the Handbook III.3.49 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_19_6_1_1_1
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_19_6_1_1_1()
sage: is_quasi_difference_matrix(M,G,6,1,1,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_19_6_1_1_1
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_19_6_1_1_1()
>>> is_quasi_difference_matrix(M,G,Integer(6),Integer(1),Integer(1),Integer(1))
True
```

sage.combinat.designs.database.**QDM_21_5_1_1_1**()

Return a $(21, 5; 1, 1; 1)$-quasi-difference matrix.

Used to build an $OA(5, 22)$

Given in the Handbook III.3.51 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_21_5_1_1_1
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_21_5_1_1_1()
sage: is_quasi_difference_matrix(M,G,5,1,1,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_21_5_1_1_1
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_21_5_1_1_1()
>>> is_quasi_difference_matrix(M,G,Integer(5),Integer(1),Integer(1),Integer(1))
True
```

sage.combinat.designs.database.**QDM_21_6_1_1_5**()

Return a $(21, 6; 1, 1; 5)$-quasi-difference matrix.

Used to build an $OA(6, 26)$

Given in the Handbook III.3.53 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_21_6_1_1_5
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_21_6_1_1_5()
sage: is_quasi_difference_matrix(M,G,6,1,1,5)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_21_6_1_1_5
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_21_6_1_1_5()
>>> is_quasi_difference_matrix(M,G,Integer(6),Integer(1),Integer(1),Integer(5))
True
```

sage.combinat.designs.database.**QDM_25_6_1_1_5**()

Return a $(25, 6; 1, 1; 5)$-quasi-difference matrix.

Used to build an $OA(6, 30)$

Given in the Handbook III.3.55 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_25_6_1_1_5
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_25_6_1_1_5()                                                  #␣
↪needs sage.modules
sage: is_quasi_difference_matrix(M,G,6,1,1,5)                                 #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_25_6_1_1_5
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_25_6_1_1_5()                                                  #␣
↪needs sage.modules
>>> is_quasi_difference_matrix(M,G,Integer(6),Integer(1),Integer(1),Integer(5))  ␣
↪                                 # needs sage.modules
True
```

sage.combinat.designs.database.**QDM_33_6_1_1_1**()

Return a $(33, 6; 1, 1; 1)$-quasi-difference matrix.

Used to build an $OA(6, 34)$

Given in the Handbook III.3.57 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_33_6_1_1_1
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_33_6_1_1_1()
```

(continues on next page)

(continued from previous page)

```
sage: is_quasi_difference_matrix(M,G,6,1,1,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_33_6_1_1_1
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_33_6_1_1_1()
>>> is_quasi_difference_matrix(M,G,Integer(6),Integer(1),Integer(1),Integer(1))
True
```

sage.combinat.designs.database.**QDM_35_7_1_1_7**()

Return a $(35,7;1,1;7)$-quasi-difference matrix.

Used to build an $OA(7,42)$

As explained in the Handbook III.3.63 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_35_7_1_1_7
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_35_7_1_1_7()
sage: is_quasi_difference_matrix(M,G,7,1,1,7)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_35_7_1_1_7
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_35_7_1_1_7()
>>> is_quasi_difference_matrix(M,G,Integer(7),Integer(1),Integer(1),Integer(7))
True
```

sage.combinat.designs.database.**QDM_37_6_1_1_1**()

Return a $(37,6;1,1;1)$-quasi-difference matrix.

Used to build an $OA(6,38)$

Given in the Handbook III.3.60 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_37_6_1_1_1
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_37_6_1_1_1()
sage: is_quasi_difference_matrix(M,G,6,1,1,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_37_6_1_1_1
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_37_6_1_1_1()
>>> is_quasi_difference_matrix(M,G,Integer(6),Integer(1),Integer(1),Integer(1))
True
```

sage.combinat.designs.database.**QDM_45_7_1_1_9**()

Return a $(45, 7; 1, 1; 9)$-quasi-difference matrix.

Used to build an $OA(7, 54)$

As explained in the Handbook III.3.71 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_45_7_1_1_9
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_45_7_1_1_9()
sage: is_quasi_difference_matrix(M,G,7,1,1,9)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_45_7_1_1_9
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_45_7_1_1_9()
>>> is_quasi_difference_matrix(M,G,Integer(7),Integer(1),Integer(1),Integer(9))
True
```

sage.combinat.designs.database.**QDM_54_7_1_1_8**()

Return a $(54, 7; 1, 1; 8)$-quasi-difference matrix.

Used to build an $OA(7, 62)$

As explained in the Handbook III.3.74 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_54_7_1_1_8
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_54_7_1_1_8()
sage: is_quasi_difference_matrix(M,G,7,1,1,8)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_54_7_1_1_8
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_54_7_1_1_8()
>>> is_quasi_difference_matrix(M,G,Integer(7),Integer(1),Integer(1),Integer(8))
True
```

sage.combinat.designs.database.**QDM_57_9_1_1_8**()

Return a $(57, 9; 1, 1; 8)$-quasi-difference matrix.

Used to build an $OA(9, 65)$

Construction shared by Julian R. Abel

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_57_9_1_1_8
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_57_9_1_1_8()                                                    #␣
↪needs sage.schemes
```

(continues on next page)

(continued from previous page)

```
sage: is_quasi_difference_matrix(M,G,9,1,1,8)                                   #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_57_9_1_1_8
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_57_9_1_1_8()                                                      #␣
↪needs sage.schemes
>>> is_quasi_difference_matrix(M,G,Integer(9),Integer(1),Integer(1),Integer(8))  ␣
↪                                 # needs sage.schemes
True
```

sage.combinat.designs.database.**RBIBD_120_8_1**()

Return a resolvable $BIBD(120, 8, 1)$.

This function output a list `L` of 17×15 blocks such that `L[i*15:(i+1)*15]` is a partition of 120.

Construction shared by Julian R. Abel:

> Seiden's method: Start with a cyclic $(273, 17, 1) - BIBD$ and let B be an hyperoval, i.e. a set of 18 points which intersects any block of the BIBD in either 0 points (153 blocks) or 2 points (120 blocks). Dualise this design and take these last 120 blocks as points in the design; blocks in the design will correspond to the $273 - 18 = 255$ non-hyperoval points.
>
> The design is also resolvable. In the original $PG(2, 16)$ take any point T in the hyperoval and consider a block $B1$ containing T. The 15 points in $B1$ that do not belong to the hyperoval correspond to 15 blocks forming a parallel class in the dualised design. The other 16 parallel classes come in a similar way, by using point T and the other 16 blocks containing T.

See also

OA_9_120()

EXAMPLES:

```
sage: from sage.combinat.designs.database import RBIBD_120_8_1
sage: from sage.combinat.designs.bibd import is_pairwise_balanced_design
sage: RBIBD = RBIBD_120_8_1()                                                   #␣
↪needs sage.modules
sage: is_pairwise_balanced_design(RBIBD,120,[8])                                #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import RBIBD_120_8_1
>>> from sage.combinat.designs.bibd import is_pairwise_balanced_design
>>> RBIBD = RBIBD_120_8_1()                                                     #␣
↪needs sage.modules
>>> is_pairwise_balanced_design(RBIBD,Integer(120),[Integer(8)])                ␣
↪              # needs sage.modules
True
```

It is indeed resolvable, and the parallel classes are given by 17 slices of consecutive 15 blocks:

```
sage: for i in range(17):                                                    #␣
↪needs sage.modules
....:     assert len(set(sum(RBIBD[i*15:(i+1)*15],[]))) == 120
```

```
>>> from sage.all import *
>>> for i in range(Integer(17)):                                              ␣
↪     # needs sage.modules
...     assert len(set(sum(RBIBD[i*Integer(15):(i+Integer(1))*Integer(15)],[])))␣
↪== Integer(120)
```

The BIBD is available from the constructor:

```
sage: _ = designs.balanced_incomplete_block_design(120,8)                     #␣
↪needs sage.modules
```

```
>>> from sage.all import *
>>> _ = designs.balanced_incomplete_block_design(Integer(120),Integer(8))      ␣
↪              # needs sage.modules
```

sage.combinat.designs.database.**ca_11_2_5_3**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_11_2_5_3
sage: C = ca_11_2_5_3()
sage: is_covering_array(C,2,3)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_11_2_5_3
>>> C = ca_11_2_5_3()
>>> is_covering_array(C,Integer(2),Integer(3))
True
```

sage.combinat.designs.database.**ca_12_2_7_3**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_12_2_7_3
sage: C = ca_12_2_7_3()
sage: is_covering_array(C,2,3)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_12_2_7_3
>>> C = ca_12_2_7_3()
>>> is_covering_array(C,Integer(2),Integer(3))
True
```

sage.combinat.designs.database.**ca_13_2_9_3**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_13_2_9_3
sage: C = ca_13_2_9_3()
sage: is_covering_array(C,2,3)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_13_2_9_3
>>> C = ca_13_2_9_3()
>>> is_covering_array(C,Integer(2),Integer(3))
True
```

sage.combinat.designs.database.**ca_14_2_10_3**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_14_2_10_3
sage: C = ca_14_2_10_3()
sage: is_covering_array(C,2,3)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_14_2_10_3
>>> C = ca_14_2_10_3()
>>> is_covering_array(C,Integer(2),Integer(3))
True
```

sage.combinat.designs.database.**ca_15_2_20_3**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from [Nur2004]

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_15_2_20_3
sage: C = ca_15_2_20_3()
sage: is_covering_array(C,2,3)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_15_2_20_3
>>> C = ca_15_2_20_3()
>>> is_covering_array(C,Integer(2),Integer(3))
True
```

sage.combinat.designs.database.**ca_19_2_6_4**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_19_2_6_4
sage: C = ca_19_2_6_4()
sage: is_covering_array(C,2,4)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_19_2_6_4
>>> C = ca_19_2_6_4()
>>> is_covering_array(C,Integer(2),Integer(4))
True
```

sage.combinat.designs.database.**ca_21_2_7_4**()

Return a CA with the given parameters. This CA is proven to be optimal. This CA is also uniform.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_21_2_7_4
sage: C = ca_21_2_7_4()
sage: is_covering_array(C,2,4)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_21_2_7_4
>>> C = ca_21_2_7_4()
>>> is_covering_array(C,Integer(2),Integer(4))
True
```

sage.combinat.designs.database.**ca_29_2_7_5**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_29_2_7_5
sage: C = ca_29_2_7_5()
sage: is_covering_array(C,2,5)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_29_2_7_5
>>> C = ca_29_2_7_5()
>>> is_covering_array(C,Integer(2),Integer(5))
True
```

sage.combinat.designs.database.**ca_37_2_4_6**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_37_2_4_6
sage: C = ca_37_2_4_6()
sage: is_covering_array(C,2,6)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_37_2_4_6
>>> C = ca_37_2_4_6()
>>> is_covering_array(C,Integer(2),Integer(6))
True
```

sage.combinat.designs.database.**ca_39_2_5_6**()

Return a CA with the given parameters. This CA is proven to be optimal.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_39_2_5_6
sage: C = ca_39_2_5_6()
sage: is_covering_array(C,2,6)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.designs.database import ca_39_2_5_6
>>> C = ca_39_2_5_6()
>>> is_covering_array(C,Integer(2),Integer(6))
True
```

sage.combinat.designs.database.**ca_41_2_6_6**()

Return a CA with the given parameters. This CA is proven to be optimal. This CA is also uniform.

Data obtained from https://zenodo.org/records/1476059

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: from sage.combinat.designs.database import ca_41_2_6_6
sage: C = ca_41_2_6_6()
sage: is_covering_array(C,2,6)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> from sage.combinat.designs.database import ca_41_2_6_6
>>> C = ca_41_2_6_6()
>>> is_covering_array(C,Integer(2),Integer(6))
True
```

sage.combinat.designs.database.**cyclic_shift**(l, i)

sage.combinat.designs.database.**f**()

Return a $(57, 9; 1, 1; 8)$-quasi-difference matrix.

Used to build an $OA(9, 65)$

Construction shared by Julian R. Abel

EXAMPLES:

```
sage: from sage.combinat.designs.database import QDM_57_9_1_1_8
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_57_9_1_1_8()                                                    #␣
↪needs sage.schemes
sage: is_quasi_difference_matrix(M,G,9,1,1,8)                                   #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM_57_9_1_1_8
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> G,M = QDM_57_9_1_1_8()                                                  #␣
↪needs sage.schemes
>>> is_quasi_difference_matrix(M,G,Integer(9),Integer(1),Integer(1),Integer(8))  ␣
↪                                 # needs sage.schemes
True
```

5.1.82 Catalog of designs

This module gathers all designs that can be reached through `designs.<tab>`. Example with the Witt design on 24 points:

```
sage: designs.WittDesign(24)                          # optional - gap_package_
↪design
Incidence structure with 24 points and 759 blocks
```

```
>>> from sage.all import *
>>> designs.WittDesign(Integer(24))                          # optional - gap_
↪package_design
Incidence structure with 24 points and 759 blocks
```

Or a Steiner Triple System on 19 points:

```
sage: designs.steiner_triple_system(19)
(19,3,1)-Balanced Incomplete Block Design
```

```
>>> from sage.all import *
>>> designs.steiner_triple_system(Integer(19))
(19,3,1)-Balanced Incomplete Block Design
```

La Jolla Covering Repository

The La Jolla Covering Repository (LJCR, see[1]) is an online database of covering designs. As it is frequently updated, it is not included in Sage, but one can query it through *designs.best_known_covering_design_from_LJCR*:

```
sage: C = designs.best_known_covering_design_from_LJCR(7, 3, 2)    # optional -␣
↪internet
sage: C                               # optional - internet
(7, 3, 2)-covering design of size 7
Lower bound: 7
Method: lex covering
Submitted on: 1996-12-01 00:00:00
sage: C.incidence_structure()         # optional - internet
Incidence structure with 7 points and 7 blocks
```

```
>>> from sage.all import *
>>> C = designs.best_known_covering_design_from_LJCR(Integer(7), Integer(3),␣
↪Integer(2))    # optional - internet
>>> C                               # optional - internet
(7, 3, 2)-covering design of size 7
Lower bound: 7
Method: lex covering
Submitted on: 1996-12-01 00:00:00
>>> C.incidence_structure()         # optional - internet
Incidence structure with 7 points and 7 blocks
```

Design constructors

This module gathers the following designs:

[1] La Jolla Covering Repository, https://dmgordon.org/cover

ProjectiveGeometryDesign()
DesarguesianProjectivePlaneDesign()
HughesPlane()
HigmanSimsDesign()
balanced_incomplete_block_design()
resolvable_balanced_incomplete_block_design()
kirkman_triple_system()
AffineGeometryDesign()
CremonaRichmondConfiguration()
WittDesign()
HadamardDesign()
Hadamard3Design()
mutually_orthogonal_latin_squares()
transversal_design()
orthogonal_array()
incomplete_orthogonal_array()
difference_family()
difference_matrix()
steiner_triple_system()
steiner_quadruple_system()
projective_plane()
`biplane()`
gen_quadrangles_with_spread()

And the *designs.best_known_covering_design_from_LJCR* function which queries the LJCR.

Todo

Implement DerivedDesign and ComplementaryDesign.

REFERENCES:

5.1.83 Cython functions for combinatorial designs

This module implements the design methods that need to be somewhat efficient.

Functions

`sage.combinat.designs.designs_pyx.`**`is_covering_array`**(*array*, *strength=None*, *levels=None*, *verbose=False*, *parameters=False*)

Check if the input is a covering array with given strength.

See *sage.combinat.designs.covering_array* for a definition.

INPUT:

- `array` – the Covering Array to be tested
- `strength` – integer; the parameter t of the covering array, such that in any selection of t columns of the array, every t -tuple appears at least once. If set to None then all t > 0 are tested to and the maximal strength is used.
- `levels` – the number of symbols that appear in `array` If set to None, then each unique entry in `array` is counted

- `verbose` – boolean; whether to display some information about the covering array
- `parameters` – boolean; whether to return the parameters of the Covering Array. If set to `True`, the function returns a pair `(boolean_answer,(N,t,k,v))`.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_covering_array
sage: C = [[1, 1, 1, 0],
....:      [1, 1, 0, 0],
....:      [0, 0, 0]]
sage: is_covering_array(C)
Traceback (most recent call last):
...
ValueError: Not all rows are the same length, row 2 is not the same length as row␣
↪0

sage: C = [[0, 1, 1],
....:      [1, 1, 0],
....:      [1, 0, 1],
....:      [0, 0, 0,]]
sage: is_covering_array(C,strength=4)
Traceback (most recent call last):
...
ValueError: Strength must be equal or less than number of columns

sage: C = [[0, 1, 1],
....:      [1, 1, 1],
....:      [1, 0, 1]]
sage: is_covering_array(C,verbose=True)
A 3 by 3 Covering Array with strength 0 with entries from a symbol set of size 2
True

sage: C = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....:      [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
....:      [0, 1, 1, 1, 0, 0, 0, 1, 1, 1],
....:      [1, 0, 1, 1, 0, 1, 1, 0, 0, 1],
....:      [1, 1, 0, 1, 1, 0, 1, 0, 1, 0],
....:      [1, 1, 1, 0, 1, 1, 0, 1, 2, 0]]
sage: is_covering_array(C,levels=2)
Traceback (most recent call last):
...
ValueError: Array should contain integer symbols from 0 to 1

sage: C = [[1, 0, 0, 2, 0, 2, 1, 2, 2, 1, 0, 2, 2],
....:      [1, 1, 0, 0, 2, 0, 2, 1, 2, 2, 1, 0, 2],
....:      [1, 1, 1, 0, 0, 2, 0, 2, 1, 2, 2, 1, 0],
....:      [0, 1, 1, 1, 0, 0, 2, 0, 2, 1, 2, 2, 1],
....:      [2, 0, 1, 1, 1, 0, 0, 2, 0, 2, 1, 2, 2],
....:      [1, 2, 0, 1, 1, 1, 0, 0, 2, 0, 2, 1, 2],
....:      [1, 1, 2, 0, 1, 1, 1, 0, 0, 2, 0, 2, 1],
....:      [2, 1, 1, 2, 0, 1, 1, 1, 0, 0, 2, 0, 2],
....:      [1, 2, 1, 1, 2, 0, 1, 1, 1, 0, 0, 2, 0],
....:      [0, 1, 2, 1, 1, 2, 0, 1, 1, 1, 0, 0, 2],
```

(continues on next page)

(continued from previous page)

```
....:         [1, 0, 1, 2, 1, 1, 2, 0, 1, 1, 1, 0, 0],
....:         [0, 1, 0, 1, 2, 1, 1, 2, 0, 1, 1, 1, 0],
....:         [0, 0, 1, 0, 1, 2, 1, 1, 2, 0, 1, 1, 1],
....:         [2, 0, 0, 1, 0, 1, 2, 1, 1, 2, 0, 1, 1],
....:         [2, 2, 0, 0, 1, 0, 1, 2, 1, 1, 2, 0, 1],
....:         [2, 2, 2, 0, 0, 1, 0, 1, 2, 1, 1, 2, 0],
....:         [0, 2, 2, 2, 0, 0, 1, 0, 1, 2, 1, 1, 2],
....:         [1, 0, 2, 2, 2, 0, 0, 1, 0, 1, 2, 1, 1],
....:         [2, 1, 0, 2, 2, 2, 0, 0, 1, 0, 1, 2, 1],
....:         [2, 2, 1, 0, 2, 2, 2, 0, 0, 1, 0, 1, 2],
....:         [1, 2, 2, 1, 0, 2, 2, 2, 0, 0, 1, 0, 1],
....:         [2, 1, 2, 2, 1, 0, 2, 2, 2, 0, 0, 1, 0],
....:         [0, 2, 1, 2, 2, 1, 0, 2, 2, 2, 0, 0, 1],
....:         [2, 0, 2, 1, 2, 2, 1, 0, 2, 2, 2, 0, 0],
....:         [0, 2, 0, 2, 1, 2, 2, 1, 0, 2, 2, 2, 0],
....:         [0, 0, 2, 0, 2, 1, 2, 2, 1, 0, 2, 2, 2],
....:         [1, 1, 0, 2, 1, 1, 2, 1, 0, 1, 0, 0, 2],
....:         [1, 1, 1, 0, 2, 1, 1, 2, 1, 0, 1, 0, 0],
....:         [0, 1, 1, 1, 0, 2, 1, 1, 2, 1, 0, 1, 0],
....:         [0, 0, 1, 1, 1, 0, 2, 1, 1, 2, 1, 0, 1],
....:         [2, 0, 0, 1, 1, 1, 0, 2, 1, 1, 2, 1, 0],
....:         [0, 2, 0, 0, 1, 1, 1, 0, 2, 1, 1, 2, 1],
....:         [2, 0, 2, 0, 0, 1, 1, 1, 0, 2, 1, 1, 2],
....:         [1, 2, 0, 2, 0, 0, 1, 1, 1, 0, 2, 1, 1],
....:         [2, 1, 2, 0, 2, 0, 0, 1, 1, 1, 0, 2, 1],
....:         [2, 2, 1, 2, 0, 2, 0, 0, 1, 1, 1, 0, 2],
....:         [1, 2, 2, 1, 2, 0, 2, 0, 0, 1, 1, 1, 0],
....:         [0, 1, 2, 2, 1, 2, 0, 2, 0, 0, 1, 1, 1],
....:         [2, 0, 1, 2, 2, 1, 2, 0, 2, 0, 0, 1, 1],
....:         [2, 2, 0, 1, 2, 2, 1, 2, 0, 2, 0, 0, 1],
....:         [2, 2, 2, 0, 1, 2, 2, 1, 2, 0, 2, 0, 0],
....:         [0, 2, 2, 2, 0, 1, 2, 2, 1, 2, 0, 2, 0],
....:         [0, 0, 2, 2, 2, 0, 1, 2, 2, 1, 2, 0, 2],
....:         [1, 0, 0, 2, 2, 2, 0, 1, 2, 2, 1, 2, 0],
....:         [0, 1, 0, 0, 2, 2, 2, 0, 1, 2, 2, 1, 2],
....:         [1, 0, 1, 0, 0, 2, 2, 2, 0, 1, 2, 2, 1],
....:         [2, 1, 0, 1, 0, 0, 2, 2, 2, 0, 1, 2, 2],
....:         [1, 2, 1, 0, 1, 0, 0, 2, 2, 2, 0, 1, 2],
....:         [1, 1, 2, 1, 0, 1, 0, 0, 2, 2, 2, 0, 1],
....:         [2, 1, 1, 2, 1, 0, 1, 0, 0, 2, 2, 2, 0],
....:         [0, 2, 1, 1, 2, 1, 0, 1, 0, 0, 2, 2, 2],
....:         [1, 0, 2, 1, 1, 2, 1, 0, 1, 0, 0, 2, 2],
....:         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
sage: is_covering_array(C,parameters=True)
(True, (53, 3, 13, 3))

sage: C = [[1, 0, 1, 1, 2, 0, 2, 2],
....:         [2, 1, 0, 1, 1, 2, 0, 2],
....:         [2, 2, 1, 0, 1, 1, 2, 0],
....:         [0, 2, 2, 1, 0, 1, 1, 2],
....:         [2, 0, 2, 2, 1, 0, 1, 1],
....:         [1, 2, 0, 2, 2, 1, 0, 1],
```

(continues on next page)

(continued from previous page)

```
....:       [1, 1, 2, 0, 2, 2, 1, 0],
....:       [0, 1, 1, 2, 0, 2, 2, 1]]
sage: is_covering_array(C,strength=2,parameters=True)
(False, (8, 0, 8, 3))
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_covering_array
>>> C = [[Integer(1), Integer(1), Integer(1), Integer(0)],
...      [Integer(1), Integer(1), Integer(0), Integer(0)],
...      [Integer(0), Integer(0), Integer(0)]]
>>> is_covering_array(C)
Traceback (most recent call last):
...
ValueError: Not all rows are the same length, row 2 is not the same length as row␣
↪0

>>> C = [[Integer(0), Integer(1), Integer(1)],
...      [Integer(1), Integer(1), Integer(0)],
...      [Integer(1), Integer(0), Integer(1)],
...      [Integer(0), Integer(0), Integer(0),]]
>>> is_covering_array(C,strength=Integer(4))
Traceback (most recent call last):
...
ValueError: Strength must be equal or less than number of columns

>>> C = [[Integer(0), Integer(1), Integer(1)],
...      [Integer(1), Integer(1), Integer(1)],
...      [Integer(1), Integer(0), Integer(1)]]
>>> is_covering_array(C,verbose=True)
A 3 by 3 Covering Array with strength 0 with entries from a symbol set of size 2
True

>>> C = [[Integer(0), Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0), Integer(0)],
...      [Integer(0), Integer(0), Integer(0), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(1), Integer(1), Integer(1)],
...      [Integer(0), Integer(1), Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(1)],
...      [Integer(1), Integer(0), Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(1)],
...      [Integer(1), Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(0)],
...      [Integer(1), Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(0), Integer(1), Integer(2), Integer(0)]]
>>> is_covering_array(C,levels=Integer(2))
Traceback (most recent call last):
...
ValueError: Array should contain integer symbols from 0 to 1

>>> C = [[Integer(1), Integer(0), Integer(0), Integer(2), Integer(0), Integer(2),␣
↪Integer(1), Integer(2), Integer(2), Integer(1), Integer(0), Integer(2),␣
↪Integer(2)],
```

(continues on next page)

(continued from previous page)

```
...       [Integer(1), Integer(1), Integer(0), Integer(0), Integer(2), Integer(0),␣
↪Integer(2), Integer(1), Integer(2), Integer(2), Integer(1), Integer(0),␣
↪Integer(2)],
...       [Integer(1), Integer(1), Integer(1), Integer(0), Integer(0), Integer(2),␣
↪Integer(0), Integer(2), Integer(1), Integer(2), Integer(2), Integer(1),␣
↪Integer(0)],
...       [Integer(0), Integer(1), Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(2), Integer(0), Integer(2), Integer(1), Integer(2), Integer(2),␣
↪Integer(1)],
...       [Integer(2), Integer(0), Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(2), Integer(0), Integer(2), Integer(1), Integer(2),␣
↪Integer(2)],
...       [Integer(1), Integer(2), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(0), Integer(2), Integer(0), Integer(2), Integer(1),␣
↪Integer(2)],
...       [Integer(1), Integer(1), Integer(2), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(2), Integer(0), Integer(2),␣
↪Integer(1)],
...       [Integer(2), Integer(1), Integer(1), Integer(2), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(2), Integer(0),␣
↪Integer(2)],
...       [Integer(1), Integer(2), Integer(1), Integer(1), Integer(2), Integer(0),␣
↪Integer(1), Integer(1), Integer(1), Integer(0), Integer(0), Integer(2),␣
↪Integer(0)],
...       [Integer(0), Integer(1), Integer(2), Integer(1), Integer(1), Integer(2),␣
↪Integer(0), Integer(1), Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(2)],
...       [Integer(1), Integer(0), Integer(1), Integer(2), Integer(1), Integer(1),␣
↪Integer(2), Integer(0), Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)],
...       [Integer(0), Integer(1), Integer(0), Integer(1), Integer(2), Integer(1),␣
↪Integer(1), Integer(2), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0)],
...       [Integer(0), Integer(0), Integer(1), Integer(0), Integer(1), Integer(2),␣
↪Integer(1), Integer(1), Integer(2), Integer(0), Integer(1), Integer(1),␣
↪Integer(1)],
...       [Integer(2), Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(2), Integer(1), Integer(1), Integer(2), Integer(0), Integer(1),␣
↪Integer(1)],
...       [Integer(2), Integer(2), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(1), Integer(1), Integer(2), Integer(0),␣
↪Integer(1)],
...       [Integer(2), Integer(2), Integer(2), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(2), Integer(1), Integer(1), Integer(2),␣
↪Integer(0)],
...       [Integer(0), Integer(2), Integer(2), Integer(2), Integer(0), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(2), Integer(1), Integer(1),␣
↪Integer(2)],
...       [Integer(1), Integer(0), Integer(2), Integer(2), Integer(2), Integer(0),␣
↪Integer(0), Integer(1), Integer(0), Integer(1), Integer(2), Integer(1),␣
↪Integer(1)],
...       [Integer(2), Integer(1), Integer(0), Integer(2), Integer(2), Integer(2),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0), Integer(0), Integer(1), Integer(0), Integer(1), Integer(2),␣
↪Integer(1)],
...       [Integer(2), Integer(2), Integer(1), Integer(0), Integer(2), Integer(2),␣
↪Integer(2), Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(2)],
...       [Integer(1), Integer(2), Integer(2), Integer(1), Integer(0), Integer(2),␣
↪Integer(2), Integer(2), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1)],
...       [Integer(2), Integer(1), Integer(2), Integer(2), Integer(1), Integer(0),␣
↪Integer(2), Integer(2), Integer(2), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)],
...       [Integer(0), Integer(2), Integer(1), Integer(2), Integer(2), Integer(1),␣
↪Integer(0), Integer(2), Integer(2), Integer(2), Integer(0), Integer(0),␣
↪Integer(1)],
...       [Integer(2), Integer(0), Integer(2), Integer(1), Integer(2), Integer(2),␣
↪Integer(1), Integer(0), Integer(2), Integer(2), Integer(2), Integer(0),␣
↪Integer(0)],
...       [Integer(0), Integer(2), Integer(0), Integer(2), Integer(1), Integer(2),␣
↪Integer(2), Integer(1), Integer(0), Integer(2), Integer(2), Integer(2),␣
↪Integer(0)],
...       [Integer(0), Integer(0), Integer(2), Integer(0), Integer(2), Integer(1),␣
↪Integer(2), Integer(2), Integer(1), Integer(0), Integer(2), Integer(2),␣
↪Integer(2)],
...       [Integer(1), Integer(1), Integer(0), Integer(2), Integer(1), Integer(1),␣
↪Integer(2), Integer(1), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(2)],
...       [Integer(1), Integer(1), Integer(1), Integer(0), Integer(2), Integer(1),␣
↪Integer(1), Integer(2), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(0)],
...       [Integer(0), Integer(1), Integer(1), Integer(1), Integer(0), Integer(2),␣
↪Integer(1), Integer(1), Integer(2), Integer(1), Integer(0), Integer(1),␣
↪Integer(0)],
...       [Integer(0), Integer(0), Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(2), Integer(1), Integer(1), Integer(2), Integer(1), Integer(0),␣
↪Integer(1)],
...       [Integer(2), Integer(0), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(2), Integer(1), Integer(1), Integer(2), Integer(1),␣
↪Integer(0)],
...       [Integer(0), Integer(2), Integer(0), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(2), Integer(1), Integer(1), Integer(2),␣
↪Integer(1)],
...       [Integer(2), Integer(0), Integer(2), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(2), Integer(1), Integer(1),␣
↪Integer(2)],
...       [Integer(1), Integer(2), Integer(0), Integer(2), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(1), Integer(0), Integer(2), Integer(1),␣
↪Integer(1)],
...       [Integer(2), Integer(1), Integer(2), Integer(0), Integer(2), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(1), Integer(0), Integer(2),␣
↪Integer(1)],
...       [Integer(2), Integer(2), Integer(1), Integer(2), Integer(0), Integer(2),␣
↪Integer(0), Integer(0), Integer(1), Integer(1), Integer(1), Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)],
...       [Integer(1), Integer(2), Integer(2), Integer(1), Integer(2), Integer(0),␣
↪Integer(2), Integer(0), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0)],
...       [Integer(0), Integer(1), Integer(2), Integer(2), Integer(1), Integer(2),␣
↪Integer(0), Integer(2), Integer(0), Integer(0), Integer(1), Integer(1),␣
↪Integer(1)],
...       [Integer(2), Integer(0), Integer(1), Integer(2), Integer(2), Integer(1),␣
↪Integer(2), Integer(0), Integer(2), Integer(0), Integer(0), Integer(1),␣
↪Integer(1)],
...       [Integer(2), Integer(2), Integer(0), Integer(1), Integer(2), Integer(2),␣
↪Integer(1), Integer(2), Integer(0), Integer(2), Integer(0), Integer(0),␣
↪Integer(1)],
...       [Integer(2), Integer(2), Integer(2), Integer(0), Integer(1), Integer(2),␣
↪Integer(2), Integer(1), Integer(2), Integer(0), Integer(2), Integer(0),␣
↪Integer(0)],
...       [Integer(0), Integer(2), Integer(2), Integer(2), Integer(0), Integer(1),␣
↪Integer(2), Integer(2), Integer(1), Integer(2), Integer(0), Integer(2),␣
↪Integer(0)],
...       [Integer(0), Integer(0), Integer(2), Integer(2), Integer(2), Integer(0),␣
↪Integer(1), Integer(2), Integer(2), Integer(1), Integer(2), Integer(0),␣
↪Integer(2)],
...       [Integer(1), Integer(0), Integer(0), Integer(2), Integer(2), Integer(2),␣
↪Integer(0), Integer(1), Integer(2), Integer(2), Integer(1), Integer(2),␣
↪Integer(0)],
...       [Integer(0), Integer(1), Integer(0), Integer(0), Integer(2), Integer(2),␣
↪Integer(2), Integer(0), Integer(1), Integer(2), Integer(2), Integer(1),␣
↪Integer(2)],
...       [Integer(1), Integer(0), Integer(1), Integer(0), Integer(0), Integer(2),␣
↪Integer(2), Integer(2), Integer(0), Integer(1), Integer(2), Integer(2),␣
↪Integer(1)],
...       [Integer(2), Integer(1), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(2), Integer(2), Integer(2), Integer(0), Integer(1), Integer(2),␣
↪Integer(2)],
...       [Integer(1), Integer(2), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(0), Integer(2), Integer(2), Integer(2), Integer(0), Integer(1),␣
↪Integer(2)],
...       [Integer(1), Integer(1), Integer(2), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(2), Integer(2), Integer(2), Integer(0),␣
↪Integer(1)],
...       [Integer(2), Integer(1), Integer(1), Integer(2), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(2), Integer(2), Integer(2),␣
↪Integer(0)],
...       [Integer(0), Integer(2), Integer(1), Integer(1), Integer(2), Integer(1),␣
↪Integer(0), Integer(1), Integer(0), Integer(0), Integer(2), Integer(2),␣
↪Integer(2)],
...       [Integer(1), Integer(0), Integer(2), Integer(1), Integer(1), Integer(2),␣
↪Integer(1), Integer(0), Integer(1), Integer(0), Integer(0), Integer(2),␣
↪Integer(2)],
...       [Integer(0), Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0)]]
```

(continues on next page)

(continued from previous page)

```
>>> is_covering_array(C,parameters=True)
(True, (53, 3, 13, 3))

>>> C = [[Integer(1), Integer(0), Integer(1), Integer(1), Integer(2), Integer(0),␣
↪Integer(2), Integer(2)],
...      [Integer(2), Integer(1), Integer(0), Integer(1), Integer(1), Integer(2),␣
↪Integer(0), Integer(2)],
...      [Integer(2), Integer(2), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(2), Integer(0)],
...      [Integer(0), Integer(2), Integer(2), Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(2)],
...      [Integer(2), Integer(0), Integer(2), Integer(2), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)],
...      [Integer(1), Integer(2), Integer(0), Integer(2), Integer(2), Integer(1),␣
↪Integer(0), Integer(1)],
...      [Integer(1), Integer(1), Integer(2), Integer(0), Integer(2), Integer(2),␣
↪Integer(1), Integer(0)],
...      [Integer(0), Integer(1), Integer(1), Integer(2), Integer(0), Integer(2),␣
↪Integer(2), Integer(1)]]
>>> is_covering_array(C,strength=Integer(2),parameters=True)
(False, (8, 0, 8, 3))
```

sage.combinat.designs.designs_pyx.**is_difference_matrix**(*M*, *G*, *k*, *lmbda=1*, *verbose=False*)

Test if M is a (G, k, λ)-difference matrix.

A matrix M is a (G, k, λ)-difference matrix if its entries are element of G, and if for any two rows R, R' of M and $x \in G$ there are exactly λ values i such that $R_i - R'_i = x$.

INPUT:

- `M` – a matrix with entries from `G`
- `G` – a group
- `k` – integer
- `lmbda` – integer (default: 1)
- `verbose` – boolean; whether to print some information when the answer is `False`

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: q = 3**3
sage: F = GF(q,'x')                                                          #␣
↪needs sage.rings.finite_rings
sage: M = [[x*y for y in F] for x in F]                                      #␣
↪needs sage.rings.finite_rings
sage: is_difference_matrix(M,F,q,verbose=1)                                  #␣
↪needs sage.rings.finite_rings
True

sage: B = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....:       [0, 1, 2, 3, 4, 2, 3, 4, 0, 1],
....:       [0, 2, 4, 1, 3, 3, 0, 2, 4, 1]]
sage: G = GF(5)
```

(continues on next page)

(continued from previous page)

```
sage: B = [[G(b) for b in R] for R in B]
sage: is_difference_matrix(list(zip(*B)),G,3,2)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_difference_matrix
>>> q = Integer(3)**Integer(3)
>>> F = GF(q,'x')                                                          #␣
↪needs sage.rings.finite_rings
>>> M = [[x*y for y in F] for x in F]                                      #␣
↪needs sage.rings.finite_rings
>>> is_difference_matrix(M,F,q,verbose=Integer(1))                          ␣
↪     # needs sage.rings.finite_rings
True

>>> B = [[Integer(0), Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0), Integer(0)],
...      [Integer(0), Integer(1), Integer(2), Integer(3), Integer(4), Integer(2),␣
↪Integer(3), Integer(4), Integer(0), Integer(1)],
...      [Integer(0), Integer(2), Integer(4), Integer(1), Integer(3), Integer(3),␣
↪Integer(0), Integer(2), Integer(4), Integer(1)]]
>>> G = GF(Integer(5))
>>> B = [[G(b) for b in R] for R in B]
>>> is_difference_matrix(list(zip(*B)),G,Integer(3),Integer(2))
True
```

Bad input:

```
sage: # needs sage.rings.finite_rings
sage: for R in M: R.append(None)
sage: is_difference_matrix(M,F,q,verbose=1)
The matrix has 28 columns but k=27
False
sage: for R in M: _=R.pop(-1)
sage: M.append([None]*3**3)
sage: is_difference_matrix(M,F,q,verbose=1)
The matrix has 28 rows instead of lambda(|G|-1+2u)+mu=1(27-1+2.0)+1=27
False
sage: _= M.pop(-1)
sage: for R in M: R[-1] = 0
sage: is_difference_matrix(M,F,q,verbose=1)
Columns 0 and 26 generate 0 exactly 27 times instead of the expected mu(=1)
False
sage: for R in M: R[-1] = 1
sage: M[-1][-1] = 0
sage: is_difference_matrix(M,F,q,verbose=1)
Columns 0 and 26 do not generate all elements of G exactly lambda(=1) times.
The element x appeared 0 times as a difference.
False
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
>>> for R in M: R.append(None)
>>> is_difference_matrix(M,F,q,verbose=Integer(1))
The matrix has 28 columns but k=27
False
>>> for R in M: _=R.pop(-Integer(1))
>>> M.append([None]*Integer(3)**Integer(3))
>>> is_difference_matrix(M,F,q,verbose=Integer(1))
The matrix has 28 rows instead of lambda(|G|-1+2u)+mu=1(27-1+2.0)+1=27
False
>>> _= M.pop(-Integer(1))
>>> for R in M: R[-Integer(1)] = Integer(0)
>>> is_difference_matrix(M,F,q,verbose=Integer(1))
Columns 0 and 26 generate 0 exactly 27 times instead of the expected mu(=1)
False
>>> for R in M: R[-Integer(1)] = Integer(1)
>>> M[-Integer(1)][-Integer(1)] = Integer(0)
>>> is_difference_matrix(M,F,q,verbose=Integer(1))
Columns 0 and 26 do not generate all elements of G exactly lambda(=1) times.
The element x appeared 0 times as a difference.
False
```

sage.combinat.designs.designs_pyx.**is_group_divisible_design**(*groups*, *blocks*, *v*, *G=None*, *K=None*, *lambd=1*, *verbose=False*)

Check that input is a Group Divisible Design on $\{0, \ldots, v-1\}$.

For more information on Group Divisible Designs, see `GroupDivisibleDesign`.

INPUT:

- `groups` – a partition of X. If set to `None` the groups are guessed automatically, and the function returns `(True, guessed_groups)` instead of `True`
- `blocks` – collection of blocks
- `v` – integers; size of the ground set assumed to be $X = \{0, ..., v-1\}$
- `G` – list of integers of which the sizes of the groups must be elements. Set to `None` (automatic guess) by default
- `K` – list of integers of which the sizes of the blocks must be elements. Set to `None` (automatic guess) by default
- `lambd` – value of λ. Set to 1 by default
- `verbose` – boolean; whether to display some information when the design is not a GDD

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_group_divisible_design
sage: TD = designs.transversal_design(4,10)                                    #␣
↪needs sage.modules
sage: groups = [list(range(i*10,(i+1)*10)) for i in range(4)]
sage: is_group_divisible_design(groups,TD,40,lambd=1)                          #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_group_divisible_design
>>> TD = designs.transversal_design(Integer(4),Integer(10))                    ␣
```

(continues on next page)

(continued from previous page)

```
↪                # needs sage.modules
>>> groups = [list(range(i*Integer(10),(i+Integer(1))*Integer(10))) for i in␣
↪range(Integer(4))]
>>> is_group_divisible_design(groups,TD,Integer(40),lambd=Integer(1))                ␣
↪                # needs sage.modules
True
```

sage.combinat.designs.designs_pyx.**is_orthogonal_array**(*OA*, *k*, *n*, *t=2*, *verbose=False*, *terminology='OA'*)

Check that the integer matrix OA is an $OA(k, n, t)$.

See `orthogonal_array()` for a definition.

INPUT:

- `OA` – the Orthogonal Array to be tested
- `k`, `n`, `t` – integers; only implemented for $t = 2$
- `verbose` – boolean; whether to display some information when `OA` is not an orthogonal array $OA(k, n)$
- `terminology` – string; how to phrase the information when `verbose = True`. Possible values are "OA", "$MOLS$"

EXAMPLES:

```
sage: # needs sage.schemes
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: OA = designs.orthogonal_arrays.build(8,9)
sage: is_orthogonal_array(OA,8,9)
True
sage: is_orthogonal_array(OA,8,10)
False
sage: OA[4][3] = 1
sage: is_orthogonal_array(OA,8,9)
False
sage: is_orthogonal_array(OA,8,9,verbose=True)
Columns 0 and 3 are not orthogonal
False
sage: is_orthogonal_array(OA,8,9, verbose=True, terminology='MOLS')
Squares 0 and 3 are not orthogonal
False
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> OA = designs.orthogonal_arrays.build(Integer(8),Integer(9))
>>> is_orthogonal_array(OA,Integer(8),Integer(9))
True
>>> is_orthogonal_array(OA,Integer(8),Integer(10))
False
>>> OA[Integer(4)][Integer(3)] = Integer(1)
>>> is_orthogonal_array(OA,Integer(8),Integer(9))
False
>>> is_orthogonal_array(OA,Integer(8),Integer(9),verbose=True)
```

(continues on next page)

(continued from previous page)

```
Columns 0 and 3 are not orthogonal
False
>>> is_orthogonal_array(OA,Integer(8),Integer(9), verbose=True, terminology='MOLS
↪')
Squares 0 and 3 are not orthogonal
False
```

sage.combinat.designs.designs_pyx.**is_pairwise_balanced_design**(*blocks*, *v*, *K=None*, *lambd=1*, *verbose=False*)

Check that input is a Pairwise Balanced Design (PBD) on $\{0, \dots, v-1\}$.

For more information on Pairwise Balanced Designs (PBD), see `PairwiseBalancedDesign`.

INPUT:

- `blocks` – collection of blocks
- `v` – integers; size of the ground set assumed to be $X = \{0, ..., v-1\}$
- `K` – list of integers of which the sizes of the blocks must be elements; set to `None` (automatic guess) by default
- `lambd` – value of λ (default: 1)
- `verbose` – boolean; whether to display some information when the design is not a PBD

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_pairwise_balanced_design
sage: sts = designs.steiner_triple_system(9)
sage: is_pairwise_balanced_design(sts,9,[3],1)
True
sage: TD = designs.transversal_design(4,10).blocks()                       #␣
↪needs sage.modules
sage: groups = [list(range(i*10,(i+1)*10)) for i in range(4)]
sage: is_pairwise_balanced_design(TD + groups, 40, [4,10], 1, verbose=True)     #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_pairwise_balanced_design
>>> sts = designs.steiner_triple_system(Integer(9))
>>> is_pairwise_balanced_design(sts,Integer(9),[Integer(3)],Integer(1))
True
>>> TD = designs.transversal_design(Integer(4),Integer(10)).blocks()          ␣
↪              # needs sage.modules
>>> groups = [list(range(i*Integer(10),(i+Integer(1))*Integer(10))) for i in␣
↪range(Integer(4))]
>>> is_pairwise_balanced_design(TD + groups, Integer(40), [Integer(4),
↪Integer(10)], Integer(1), verbose=True)      # needs sage.modules
True
```

sage.combinat.designs.designs_pyx.**is_projective_plane**(*blocks*, *verbose=False*)

Test whether the blocks form a projective plane on $\{0, ..., v-1\}$.

A *projective plane* is an incidence structure that has the following properties:

1. Given any two distinct points, there is exactly one line incident with both of them.

2. Given any two distinct lines, there is exactly one point incident with both of them.
3. There are four points such that no line is incident with more than two of them.

For more informations, see Wikipedia article Projective_plane.

`is_t_design()` can also check if an incidence structure is a projective plane with the parameters $v = k^2 + k + 1$, $t = 2$ and $l = 1$.

INPUT:

- `blocks` – collection of blocks
- `verbose` – whether to print additional information

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_projective_plane
sage: p = designs.projective_plane(4)                                    #␣
↪needs sage.schemes
sage: b = p.blocks()                                                     #␣
↪needs sage.schemes
sage: is_projective_plane(b, verbose=True)                               #␣
↪needs sage.schemes
True

sage: # needs sage.schemes
sage: p = designs.projective_plane(2)
sage: b = p.blocks()
sage: is_projective_plane(b)
True
sage: b[0][2] = 5
sage: is_projective_plane(b, verbose=True)
the pair (0,5) has been seen 2 times but lambda=1
False

sage: is_projective_plane([[0,1,2],[1,2,4]], verbose=True)
the pair (0,3) has been seen 0 times but lambda=1
False

sage: is_projective_plane([[1]], verbose=True)
First block has less than 3 points.
False

sage: # needs sage.schemes
sage: p = designs.projective_plane(2)
sage: b = p.blocks()
sage: b[2].append(4)
sage: is_projective_plane(b, verbose=True)
a block has size 4 while K=[3]
False
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_projective_plane
>>> p = designs.projective_plane(Integer(4))                            ␣
↪     # needs sage.schemes
>>> b = p.blocks()                                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.schemes
>>> is_projective_plane(b, verbose=True)                                    #␣
↪needs sage.schemes
True

>>> # needs sage.schemes
>>> p = designs.projective_plane(Integer(2))
>>> b = p.blocks()
>>> is_projective_plane(b)
True
>>> b[Integer(0)][Integer(2)] = Integer(5)
>>> is_projective_plane(b, verbose=True)
the pair (0,5) has been seen 2 times but lambda=1
False

>>> is_projective_plane([[Integer(0),Integer(1),Integer(2)],[Integer(1),
↪Integer(2),Integer(4)]], verbose=True)
the pair (0,3) has been seen 0 times but lambda=1
False

>>> is_projective_plane([[Integer(1)]], verbose=True)
First block has less than 3 points.
False

>>> # needs sage.schemes
>>> p = designs.projective_plane(Integer(2))
>>> b = p.blocks()
>>> b[Integer(2)].append(Integer(4))
>>> is_projective_plane(b, verbose=True)
a block has size 4 while K=[3]
False
```

sage.combinat.designs.designs_pyx.**is_quasi_difference_matrix**(*M*, *G*, *k*, *lmbda*, *mu*, *u*, *verbose=False*)

Test if the matrix is a $(G, k; \lambda, \mu; u)$-quasi-difference matrix.

Let G be an abelian group of order n. A $(n, k; \lambda, \mu; u)$-quasi-difference matrix (QDM) is a matrix Q_{ij} with $\lambda(n-1+2u)+\mu$ rows and k columns, with each entry either equal to `None` (i.e. the 'missing entries') or to an element of G. Each column contains exactly λu empty entries, and each row contains at most one `None`. Furthermore, for each $1 \leq i < j \leq k$, the multiset

$$\{q_{li} - q_{lj} : 1 \leq l \leq \lambda(n-1+2u)+\mu, \text{ with } q_{li} \text{ and } q_{lj} \text{ not empty}\}$$

contains λ times every nonzero element of G and contains μ times 0.

INPUT:

- `M` – a matrix with entries from `G` (or equal to `None` for missing entries)
- `G` – a group
- `k`, `lmbda`, `mu`, `u` – integers
- `verbose` – boolean; whether to print some information when the answer is `False`

EXAMPLES:

Differences matrices:

```
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: q = 3**3
sage: F = GF(q,'x')                                                     #␣
↪needs sage.rings.finite_rings
sage: M = [[x*y for y in F] for x in F]                                 #␣
↪needs sage.rings.finite_rings
sage: is_quasi_difference_matrix(M,F,q,1,1,0,verbose=1)                 #␣
↪needs sage.rings.finite_rings
True

sage: B = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....:      [0, 1, 2, 3, 4, 2, 3, 4, 0, 1],
....:      [0, 2, 4, 1, 3, 3, 0, 2, 4, 1]]
sage: G = GF(5)
sage: B = [[G(b) for b in R] for R in B]
sage: is_quasi_difference_matrix(list(zip(*B)),G,3,2,2,0)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
>>> q = Integer(3)**Integer(3)
>>> F = GF(q,'x')                                                       #␣
↪needs sage.rings.finite_rings
>>> M = [[x*y for y in F] for x in F]                                   #␣
↪needs sage.rings.finite_rings
>>> is_quasi_difference_matrix(M,F,q,Integer(1),Integer(1),Integer(0),
↪verbose=Integer(1))                            # needs sage.rings.finite_rings
True

>>> B = [[Integer(0), Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0), Integer(0)],
...      [Integer(0), Integer(1), Integer(2), Integer(3), Integer(4), Integer(2),␣
↪Integer(3), Integer(4), Integer(0), Integer(1)],
...      [Integer(0), Integer(2), Integer(4), Integer(1), Integer(3), Integer(3),␣
↪Integer(0), Integer(2), Integer(4), Integer(1)]]
>>> G = GF(Integer(5))
>>> B = [[G(b) for b in R] for R in B]
>>> is_quasi_difference_matrix(list(zip(*B)),G,Integer(3),Integer(2),Integer(2),
↪Integer(0))
True
```

A quasi-difference matrix from the database:

```
sage: from sage.combinat.designs.database import QDM
sage: G,M = QDM[38,1][37,1,1,1][1]()
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=1,u=1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.database import QDM
>>> G,M = QDM[Integer(38),Integer(1)][Integer(37),Integer(1),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)][Integer(1)]()
>>> is_quasi_difference_matrix(M,G,k=Integer(6),lmbda=Integer(1),mu=Integer(1),
↪u=Integer(1))
True
```

Bad input:

```
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=1,u=3,verbose=1)
The matrix has 39 rows instead of lambda(|G|-1+2u)+mu=1(37-1+2.3)+1=43
False
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=2,u=1,verbose=1)
The matrix has 39 rows instead of lambda(|G|-1+2u)+mu=1(37-1+2.1)+2=40
False
sage: M[3][1] = None
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=1,u=1,verbose=1)
Row 3 contains more than one empty entry
False
sage: M[3][1] = 1
sage: M[6][1] = None
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=1,u=1,verbose=1)
Column 1 contains 2 empty entries instead of the expected lambda.u=1.1=1
False
```

```
>>> from sage.all import *
>>> is_quasi_difference_matrix(M,G,k=Integer(6),lmbda=Integer(1),mu=Integer(1),
↪u=Integer(3),verbose=Integer(1))
The matrix has 39 rows instead of lambda(|G|-1+2u)+mu=1(37-1+2.3)+1=43
False
>>> is_quasi_difference_matrix(M,G,k=Integer(6),lmbda=Integer(1),mu=Integer(2),
↪u=Integer(1),verbose=Integer(1))
The matrix has 39 rows instead of lambda(|G|-1+2u)+mu=1(37-1+2.1)+2=40
False
>>> M[Integer(3)][Integer(1)] = None
>>> is_quasi_difference_matrix(M,G,k=Integer(6),lmbda=Integer(1),mu=Integer(1),
↪u=Integer(1),verbose=Integer(1))
Row 3 contains more than one empty entry
False
>>> M[Integer(3)][Integer(1)] = Integer(1)
>>> M[Integer(6)][Integer(1)] = None
>>> is_quasi_difference_matrix(M,G,k=Integer(6),lmbda=Integer(1),mu=Integer(1),
↪u=Integer(1),verbose=Integer(1))
Column 1 contains 2 empty entries instead of the expected lambda.u=1.1=1
False
```

5.1.84 Difference families

This module gathers everything related to difference families. One can build a difference family (or check that it can be built) with *difference_family()*:

```
sage: G,F = designs.difference_family(13,4,1)                              #␣
↪needs sage.libs.pari sage.modules
```

```
>>> from sage.all import *
>>> G,F = designs.difference_family(Integer(13),Integer(4),Integer(1))
↪                        # needs sage.libs.pari sage.modules
```

It defines the following functions:

are_complementary_difference_sets()	Check if `A` and `B` are complementary difference sets over the group `G`.
are_hadamard_difference_set_parameters()	Check whether `(v,k,lmbda)` is of the form `(4N^2, 2N^2 - N, N^2 - N)`.
are_mcfarland_1973_parameters()	Test whether `(v,k,lmbda)` is a triple that can be obtained from the construction from [McF1973].
block_stabilizer()	Compute the left stabilizer of the block `B` under the action of `G`.
complementary_difference_sets()	Compute complementary difference sets over a group of order $n = 2m + 1$.
complementary_difference_setsI()	Construct complementary difference sets in a group of order $n \cong 3 \mod 4$, n a prime power.
complementary_difference_setsII()	Construct complementary difference sets in a group of order $n = p^t$, where $p \cong 5 \mod 8$ and $t \cong 1, 2, 3 \mod 4$.
complementary_difference_setsIII()	Construct complementary difference sets in a group of order $n = 2m + 1$, where $4m + 3$ is a prime power.
df_q_6_1()	Return a $(q, 6, 1)$-difference family over the finite field K.
difference_family()	Return a (`k`, `l`)-difference family on an Abelian group of cardinality `v`.
get_fixed_relative_difference_set()	Construct an equivalent relative difference set fixed by the size of the set.
group_law()	Return a triple `(identity, operation, inverse)` that define the operations on the group `G`.
hadamard_difference_set_product()	Make a product of two Hadamard difference sets.
is_difference_family()	Check whether `D` forms a difference family in the group `G`.
is_fixed_relative_difference_set()	Check if the relative difference set `R` is fixed by `q`.
is_relative_difference_set()	Check if `R` is a difference set of `G` relative to `H`, with the given parameters.
is_supplementary_difference_set()	Check that the sets in `Ks` are $n - \{v; k_1, ..., k_n; \lambda\}$ supplementary difference sets over group `G` of order `v`.
mcfarland_1973_construction()	Return a difference set.
one_cyclic_tiling()	Given a subset `A` of the cyclic additive group $G = Z/nZ$ return another subset B so that $A + B = G$ and $\|A\|\|B\| = n$ (i.e. any element of G is uniquely expressed as a sum $a + b$ with a in A and b in B).
one_radical_difference_family()	Search for a radical difference family on `K` using dancing links algorithm.
radical_difference_family()	Return a `(v,k,l)`-radical difference family.
radical_difference_set()	Return a difference set made of a cyclotomic coset in the finite field `K` and with parameters `k` and `l`.
relative_difference_set_from_homomorphism()	Construct $R((q^N - 1)/(q - 1), n, q^{N-1}, q^{N-2}d)$ where $nd = q - 1$.

continues on next page

Table 1 – continued from previous page

relative_difference_set_from_m_sequence()	Construct $R((q^N-1)/(q-1), q-1, q^{N-1}, q^{N-2})$ where q is a prime power and $N \geq 2$.
singer_difference_set()	Return a difference set associated to the set of hyperplanes in a projective space of dimension d over $GF(q)$.
skew_spin_goethals_seidel_difference_family()	Construct skew spin type Goethals-Seidel difference family with parameters $(n; k_1, k_2, k_3, k_4; \lambda)$.
skew_supplementary_difference_set()	Construct $4-\{n; n_1, n_2, n_3, n_4; \lambda\}$ supplementary difference sets, where S_1 is skew and $n_1+n_2+n_3+n_4 = n+\lambda$.
skew_supplementary_difference_set_over_polynomial_ring()	Construct skew supplementary difference sets over a polynomial ring of order n.
skew_supplementary_difference_set_with_paley_todd()	Construct $4-\{n; n_1, n_2, n_3, n_4; \lambda\}$ skew supplementary difference sets where S_1 is the Paley-Todd difference set.
spin_goethals_seidel_difference_family()	Construct a spin type Goethals-Seidel difference family with parameters $(n; k_1, k_2, k_3, k_4; \lambda)$.
supplementary_difference_set()	Construct $4-\{2v; v, v+1, v, v; 2v\}$ supplementary difference sets where $q = 2v+1$.
supplementary_difference_set_from_rel_diff_se	Construct $4-\{2v; v, v+1, v, v; 2v\}$ supplementary difference sets where $q = 2v+1$.
supplementary_difference_set_hadamard()	Construct $4-\{n; n_1, n_2, n_3, n_4; \lambda\}$ supplementary difference sets, where $n_1 + n_2 + n_3 + n_4 = n+\lambda$.
turyn_1965_3x3xK()	Return a difference set in either $C_3 \times C_3 \times C_4$ or $C_3 \times C_3 \times C_2 \times C_2$ with parameters $v = 36$, $k = 15$, $\lambda = 6$.
twin_prime_powers_difference_set()	Return a difference set on $GF(p) \times GF(p+2)$.

REFERENCES:

Functions

sage.combinat.designs.difference_family.**are_complementary_difference_sets**(*G*, *A*, *B*, *verbose=False*)

Check if A and B are complementary difference sets over the group G.

According to [Sze1971], two sets A, B of size m are complementary difference sets over a group G of size $2m+1$ if:

1. they are $2-\{2m+1; m, m; m-1\}$ supplementary difference sets
2. A is skew, i.e. $a \in A$ implies $-a \notin A$

INPUT:

- G – a group of odd order
- A – set of elements of G
- B – set of elements of G
- verbose – boolean (default: False); if True the function will be verbose when the sets do not satisfy the constraints

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import are_complementary_
↪difference_sets
sage: are_complementary_difference_sets(Zmod(7), [1, 2, 4], [1, 2, 4])
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import are_complementary_
↪difference_sets
>>> are_complementary_difference_sets(Zmod(Integer(7)), [Integer(1), Integer(2),␣
↪Integer(4)], [Integer(1), Integer(2), Integer(4)])
True
```

If `verbose=True`, the function will be verbose:

```
sage: are_complementary_difference_sets(Zmod(7), [1, 2, 5], [1, 2, 4],␣
↪verbose=True)
The sets are not supplementary difference sets with lambda = 2
False
```

```
>>> from sage.all import *
>>> are_complementary_difference_sets(Zmod(Integer(7)), [Integer(1), Integer(2),␣
↪Integer(5)], [Integer(1), Integer(2), Integer(4)], verbose=True)
The sets are not supplementary difference sets with lambda = 2
False
```

See also

is_supplementary_difference_set()

sage.combinat.designs.difference_family.**are_hadamard_difference_set_parameters**(*v*, *k*, *lmbda*)

Check whether `(v,k,lmbda)` is of the form `(4N^2, 2N^2 - N, N^2 - N)`.

INPUT:

- `(v, k, lmbda)` – parameters of a difference set

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import are_hadamard_difference_
↪set_parameters
sage: are_hadamard_difference_set_parameters(36, 15, 6)
True
sage: are_hadamard_difference_set_parameters(60, 13, 5)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import are_hadamard_difference_
↪set_parameters
>>> are_hadamard_difference_set_parameters(Integer(36), Integer(15), Integer(6))
True
```

(continues on next page)

(continued from previous page)

```
>>> are_hadamard_difference_set_parameters(Integer(60), Integer(13), Integer(5))
False
```

sage.combinat.designs.difference_family.**are_mcfarland_1973_parameters**(*v*, *k*, *lmbda*, *return_parameters=False*)

Test whether (v,k,lmbda) is a triple that can be obtained from the construction from [McF1973].

See *mcfarland_1973_construction()*.

INPUT:

- v, k, lmbda – integers; parameters of the difference family
- return_parameters – boolean (default: False); if True, return a pair (True, (q, s)) so that (q,s) can be used in the function *mcfarland_1973_construction()* to actually build a (v,k, lmbda)-difference family. Or (False, None) if the construction is not possible

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.combinat.designs.difference_family import are_mcfarland_1973_
↪parameters
sage: are_mcfarland_1973_parameters(64, 28, 12)
True
sage: are_mcfarland_1973_parameters(64, 28, 12, return_parameters=True)
(True, (2, 2))
sage: are_mcfarland_1973_parameters(60, 13, 5)
False
sage: are_mcfarland_1973_parameters(98125, 19500, 3875)
True
sage: are_mcfarland_1973_parameters(98125, 19500, 3875, True)
(True, (5, 3))

sage: from sage.combinat.designs.difference_family import are_mcfarland_1973_
↪parameters
sage: for v in range(1, 100):                                                #␣
↪needs sage.rings.finite_rings
....:     for k in range(1,30):
....:         for l in range(1,15):
....:             if are_mcfarland_1973_parameters(v,k,l):
....:                 answer, (q,s) = are_mcfarland_1973_parameters(v,k,l,return_
↪parameters=True)
....:                 print("{} {} {} {} {}".format(v,k,l,q,s))
....:                 assert answer is True
....:                 assert designs.difference_family(v,k,l,existence=True) is␣
↪True
....:                 G,D = designs.difference_family(v,k,l)
16 6 2 2 1
45 12 3 3 1
64 28 12 2 2
96 20 4 4 1
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.designs.difference_family import are_mcfarland_1973_
↪parameters
>>> are_mcfarland_1973_parameters(Integer(64), Integer(28), Integer(12))
True
>>> are_mcfarland_1973_parameters(Integer(64), Integer(28), Integer(12), return_
↪parameters=True)
(True, (2, 2))
>>> are_mcfarland_1973_parameters(Integer(60), Integer(13), Integer(5))
False
>>> are_mcfarland_1973_parameters(Integer(98125), Integer(19500), Integer(3875))
True
>>> are_mcfarland_1973_parameters(Integer(98125), Integer(19500), Integer(3875),␣
↪True)
(True, (5, 3))

>>> from sage.combinat.designs.difference_family import are_mcfarland_1973_
↪parameters
>>> for v in range(Integer(1), Integer(100)):                                  ␣
↪               # needs sage.rings.finite_rings
...     for k in range(Integer(1),Integer(30)):
...         for l in range(Integer(1),Integer(15)):
...             if are_mcfarland_1973_parameters(v,k,l):
...                 answer, (q,s) = are_mcfarland_1973_parameters(v,k,l,return_
↪parameters=True)
...                 print("{} {} {} {} {}".format(v,k,l,q,s))
...                 assert answer is True
...                 assert designs.difference_family(v,k,l,existence=True) is True
...                 G,D = designs.difference_family(v,k,l)
16 6 2 2 1
45 12 3 3 1
64 28 12 2 2
96 20 4 4 1
```

`sage.combinat.designs.difference_family.`**`block_stabilizer`**`(`G, B`)`

Compute the left stabilizer of the block `B` under the action of `G`.

This function return the list of all $x \in G$ such that $x \cdot B = B$ (as a set).

INPUT:

- `G` – a group (additive or multiplicative)
- `B` – a subset of `G`

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import block_stabilizer

sage: Z8 = Zmod(8)
sage: block_stabilizer(Z8, [Z8(0),Z8(2),Z8(4),Z8(6)])
[0, 2, 4, 6]
sage: block_stabilizer(Z8, [Z8(0),Z8(2)])
[0]
```

(continues on next page)

(continued from previous page)

```
sage: C = cartesian_product([Zmod(4),Zmod(3)])
sage: block_stabilizer(C, [C((0,0)),C((2,0)),C((0,1)),C((2,1))])
[(0, 0), (2, 0)]

sage: b = list(map(Zmod(45),[1, 3, 7, 10, 22, 25, 30, 35, 37, 38, 44]))
sage: block_stabilizer(Zmod(45),b)
[0]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import block_stabilizer

>>> Z8 = Zmod(Integer(8))
>>> block_stabilizer(Z8, [Z8(Integer(0)),Z8(Integer(2)),Z8(Integer(4)),
↪Z8(Integer(6))])
[0, 2, 4, 6]
>>> block_stabilizer(Z8, [Z8(Integer(0)),Z8(Integer(2))])
[0]

>>> C = cartesian_product([Zmod(Integer(4)),Zmod(Integer(3))])
>>> block_stabilizer(C, [C((Integer(0),Integer(0))),C((Integer(2),Integer(0))),
↪C((Integer(0),Integer(1))),C((Integer(2),Integer(1)))])
[(0, 0), (2, 0)]

>>> b = list(map(Zmod(Integer(45)),[Integer(1), Integer(3), Integer(7),
↪Integer(10), Integer(22), Integer(25), Integer(30), Integer(35), Integer(37),
↪Integer(38), Integer(44)]))
>>> block_stabilizer(Zmod(Integer(45)),b)
[0]
```

sage.combinat.designs.difference_family.**complementary_difference_sets**(*n*, *existence=False*, *check=True*)

Compute complementary difference sets over a group of order $n = 2m + 1$.

According to [Sze1971], two sets A, B of size m are complementary difference sets over a group G of size $n = 2m + 1$ if:

1. they are $2 - \{2m + 1; m, m; m - 1\}$ supplementary difference sets
2. A is skew, i.e. $a \in A$ implies $-a \notin A$

This method tries to call `complementary_difference_setsI()`, `complementary_difference_setsII()` or `complementary_difference_setsIII()` if the parameter n satisfies the requirements of one of these functions.

INPUT:

- `n` – integer; the order of the group over which the sets are constructed
- `existence` – boolean (default: `False`); if `True`, only check whether the supplementary difference sets can be constructed
- `check` – boolean (default: `True`); if `True`, check that the sets are complementary difference sets before returning them; setting this to `False` might speed up the computation for large values of `n`

OUTPUT:

If `existence=False`, the function returns group `G` and two complementary difference sets, or raises an error if data for the given `n` is not available. If `existence=True`, the function returns a boolean representing whether complementary difference sets can be constructed for the given `n`.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import complementary_
↪difference_sets
sage: complementary_difference_sets(15)                                          #␣
↪needs sage.libs.pari
(Ring of integers modulo 15, [1, 2, 4, 6, 7, 10, 12], [0, 1, 2, 6, 9, 13, 14])
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import complementary_difference_
↪sets
>>> complementary_difference_sets(Integer(15))                                     ␣
↪     # needs sage.libs.pari
(Ring of integers modulo 15, [1, 2, 4, 6, 7, 10, 12], [0, 1, 2, 6, 9, 13, 14])
```

If `existence=True`, the function returns a boolean:

```
sage: complementary_difference_sets(15, existence=True)                          #␣
↪needs sage.libs.pari
True
sage: complementary_difference_sets(16, existence=True)
False
```

```
>>> from sage.all import *
>>> complementary_difference_sets(Integer(15), existence=True)                     ␣
↪     # needs sage.libs.pari
True
>>> complementary_difference_sets(Integer(16), existence=True)
False
```

See also

are_complementary_difference_sets()

`sage.combinat.designs.difference_family.`**`complementary_difference_setsI`**`(`*n*, *check=True*`)`

Construct complementary difference sets in a group of order $n \cong 3 \mod 4$, n a prime power.

Let G be a Galois Field of order n, where n satisfies the requirements above. Let A be the set of nonzero quadratic elements in G, and $B = A$. Then A and B are complementary difference sets over a group of order n. This construction is described in [Sze1971].

INPUT:

- `n` – integer; the order of the group G
- `check` – boolean (default: `True`); if `True`, check that the sets are complementary difference sets before returning them

OUTPUT:

The function returns the Galois field of order `n` and the two sets, or raises an error if `n` does not satisfy the requirements of this construction.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import complementary_
↪difference_setsI
sage: complementary_difference_setsI(19)
(Finite Field of size 19,
 [1, 4, 5, 6, 7, 9, 11, 16, 17],
 [1, 4, 5, 6, 7, 9, 11, 16, 17])
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import complementary_difference_
↪setsI
>>> complementary_difference_setsI(Integer(19))
(Finite Field of size 19,
 [1, 4, 5, 6, 7, 9, 11, 16, 17],
 [1, 4, 5, 6, 7, 9, 11, 16, 17])
```

See also

are_complementary_difference_sets() *complementary_difference_sets()*

sage.combinat.designs.difference_family.**complementary_difference_setsII**(*n*, *check=True*)

Construct complementary difference sets in a group of order $n = p^t$, where $p \cong 5 \mod 8$ and $t \cong 1, 2, 3 \mod 4$.

Consider a finite field G of order n and let ρ be the generator of the corresponding multiplicative group. Then, there are two different constructions, depending on whether t is even or odd.

If $t \cong 2 \mod 4$, let C_0 be the set of nonzero octic residues in G, and let $C_i = \rho^i C_0$ for $1 \leq i \leq 7$. Then, $A = C_0 \cup C_1 \cup C_2 \cup C_3$ and $B = C_0 \cup C_1 \cup C_6 \cup C_7$.

If t is odd, let C_0 be the set of nonzero fourth powers in G, and let $C_i = \rho^i C_0$ for $1 \leq i \leq 3$. Then, $A = C_0 \cup C_1$ and $B = C_0 \cup C_3$.

For more details on this construction, see [Sze1971].

INPUT:

- `n` – integer; the order of the group G
- `check` – boolean (default: `True`); if `True`, check that the sets are complementary difference sets before returning them; setting this to `False` might speed up the computation for large values of `n`

OUTPUT:

The function returns the Galois field of order `n` and the two sets, or raises an error if `n` does not satisfy the requirements of this construction.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import complementary_
↪difference_setsII
sage: complementary_difference_setsII(5)                                    #␣
↪needs sage.libs.pari
(Finite Field of size 5, [1, 2], [1, 3])
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import complementary_difference_
↪setsII
>>> complementary_difference_setsII(Integer(5))                                  ␣
↪     # needs sage.libs.pari
(Finite Field of size 5, [1, 2], [1, 3])
```

See also

are_complementary_difference_sets() complementary_difference_sets()

`sage.combinat.designs.difference_family.`**`complementary_difference_setsIII`**(*n*, *check=True*)

Construct complementary difference sets in a group of order $n = 2m + 1$, where $4m + 3$ is a prime power.

Consider a finite field G of order n and let ρ be a primite element of this group. Now let Q be the set of nonzero quadratic residues in G, and let $A = \{a|\rho^{2a} - 1 \in Q\}$, $B' = \{b| - (\rho^{2b} + 1) \in Q\}$. Then A and $B = Q \setminus B'$ are complementary difference sets over the ring of integers modulo n. For more details, see [Sz1969].

INPUT:

- `n` – integer; the order of the group over which the sets are constructed
- `check` – boolean (default: `True`); if `True`, check that the sets are complementary difference sets before returning them; setting this to `False` might speed up the computation for large values of `n`

OUTPUT:

The function returns the Galois field of order `n` and the two sets, or raises an error if `n` does not satisfy the requirements of this construction.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import complementary_
↪difference_setsIII
sage: complementary_difference_setsIII(11)                                      #␣
↪needs sage.libs.pari
(Ring of integers modulo 11, [1, 2, 5, 7, 8], [0, 1, 3, 8, 10])
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import complementary_difference_
↪setsIII
>>> complementary_difference_setsIII(Integer(11))                                ␣
↪     # needs sage.libs.pari
(Ring of integers modulo 11, [1, 2, 5, 7, 8], [0, 1, 3, 8, 10])
```

See also

are_complementary_difference_sets() complementary_difference_sets()

`sage.combinat.designs.difference_family.`**`df_q_6_1`**(*K*, *existence=False*, *check=True*)

Return a $(q, 6, 1)$-difference family over the finite field K.

The construction uses Theorem 11 of [Wi72].

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.combinat.designs.difference_family import is_difference_family,␣
↪df_q_6_1
sage: prime_powers = [v for v in range(31,500,30) if is_prime_power(v)]
sage: parameters = [v for v in prime_powers
....:                if df_q_6_1(GF(v,'a'), existence=True) is True]
sage: parameters
[31, 151, 181, 211, 241, 271, 331, 361, 421]
sage: for v in parameters:
....:     K = GF(v, 'a')
....:     df = df_q_6_1(K, check=True)
....:     assert is_difference_family(K, df, v, 6, 1)
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> from sage.combinat.designs.difference_family import is_difference_family, df_
↪q_6_1
>>> prime_powers = [v for v in range(Integer(31),Integer(500),Integer(30)) if is_
↪prime_power(v)]
>>> parameters = [v for v in prime_powers
...                if df_q_6_1(GF(v,'a'), existence=True) is True]
>>> parameters
[31, 151, 181, 211, 241, 271, 331, 361, 421]
>>> for v in parameters:
...     K = GF(v, 'a')
...     df = df_q_6_1(K, check=True)
...     assert is_difference_family(K, df, v, Integer(6), Integer(1))
```

Todo

Do improvements due to Zhen and Wu 1999.

sage.combinat.designs.difference_family.**difference_family**(*v*, *k*, *l=1*, *existence=False*, *explain_construction=False*, *check=True*)

Return a (k, l)-difference family on an Abelian group of cardinality v.

Let G be a finite Abelian group. For a given subset D of G, we define ΔD to be the multi-set of differences $\Delta D = \{x - y; x \in D, y \in D, x \neq y\}$. A (G, k, λ)-*difference family* is a collection of k-subsets of G, $D = \{D_1, D_2, \ldots, D_b\}$ such that the union of the difference sets ΔD_i for $i = 1, ...b$, seen as a multi-set, contains each element of $G \backslash \{0\}$ exactly λ-times.

When there is only one block, i.e. $\lambda(v-1) = k(k-1)$, then a (G, k, λ)-difference family is also called a *difference set*.

See also Wikipedia article Difference_set.

If there is no such difference family, an EmptySetError is raised and if there is no construction at the moment NotImplementedError is raised.

INPUT:

- v, k, l – parameters of the difference family. If l is not provided it is assumed to be 1

- `existence` – if `True`, then return either `True` if Sage knows how to build such design, `Unknown` if it does not and `False` if it knows that the design does not exist
- `explain_construction` – instead of returning a difference family, returns a string that explains the construction used
- `check` – boolean (default: `True`); if `True`, then the result of the computation is checked before being returned. This should not be needed but ensures that the output is correct

OUTPUT:

A pair `(G,D)` made of a group G and a difference family D on that group. Or, if `existence=True` a troolean or if `explain_construction=True` a string.

EXAMPLES:

```
sage: G,D = designs.difference_family(73,4)                                     #␣
↪needs sage.libs.pari
sage: G                                                                         #␣
↪needs sage.libs.pari
Finite Field of size 73
sage: D                                                                         #␣
↪needs sage.libs.pari
[[0, 1, 5, 18],
 [0, 3, 15, 54],
 [0, 9, 45, 16],
 [0, 27, 62, 48],
 [0, 8, 40, 71],
 [0, 24, 47, 67]]

sage: print(designs.difference_family(73, 4, explain_construction=True))
The database contains a (73,4)-evenly distributed set

sage: # needs sage.libs.pari
sage: G,D = designs.difference_family(15,7,3)
sage: G
Ring of integers modulo 15
sage: D
[[0, 1, 2, 4, 5, 8, 10]]
sage: print(designs.difference_family(15,7,3,explain_construction=True))
Singer difference set

sage: # needs sage.libs.pari
sage: print(designs.difference_family(91,10,1,explain_construction=True))
Singer difference set
sage: print(designs.difference_family(64,28,12, explain_construction=True))
McFarland 1973 construction
sage: print(designs.difference_family(576, 276, 132, explain_construction=True))
Hadamard difference set product from N1=2 and N2=3
```

```
>>> from sage.all import *
>>> G,D = designs.difference_family(Integer(73),Integer(4))                    ␣
↪               # needs sage.libs.pari
>>> G                                                                         #␣
↪needs sage.libs.pari
Finite Field of size 73
```

(continues on next page)

(continued from previous page)

```
>>> D                                                                          #␣
↪needs sage.libs.pari
[[0, 1, 5, 18],
 [0, 3, 15, 54],
 [0, 9, 45, 16],
 [0, 27, 62, 48],
 [0, 8, 40, 71],
 [0, 24, 47, 67]]

>>> print(designs.difference_family(Integer(73), Integer(4), explain_
↪construction=True))
The database contains a (73,4)-evenly distributed set

>>> # needs sage.libs.pari
>>> G,D = designs.difference_family(Integer(15),Integer(7),Integer(3))
>>> G
Ring of integers modulo 15
>>> D
[[0, 1, 2, 4, 5, 8, 10]]
>>> print(designs.difference_family(Integer(15),Integer(7),Integer(3),explain_
↪construction=True))
Singer difference set

>>> # needs sage.libs.pari
>>> print(designs.difference_family(Integer(91),Integer(10),Integer(1),explain_
↪construction=True))
Singer difference set
>>> print(designs.difference_family(Integer(64),Integer(28),Integer(12), explain_
↪construction=True))
McFarland 1973 construction
>>> print(designs.difference_family(Integer(576), Integer(276), Integer(132),␣
↪explain_construction=True))
Hadamard difference set product from N1=2 and N2=3
```

For $k = 6, 7$ we look at the set of small prime powers for which a construction is available:

```
sage: def prime_power_mod(r, m):
....:     k = m+r
....:     while True:
....:         if is_prime_power(k):
....:             yield k
....:         k += m

sage: # needs sage.libs.pari
sage: from itertools import islice
sage: l6 = {True: [], False: [], Unknown: []}
sage: for q in islice(prime_power_mod(1,30), int(60)):
....:     l6[designs.difference_family(q,6,existence=True)].append(q)
sage: l6[True]
[31, 121, 151, 181, 211, ...,  3061, 3121, 3181]
sage: l6[Unknown]
[61]
```

(continues on next page)

(continued from previous page)

```
sage: l6[False]
[]

sage: # needs sage.libs.pari
sage: l7 = {True: [], False: [], Unknown: []}
sage: for q in islice(prime_power_mod(1,42), int(60)):
....:     l7[designs.difference_family(q,7,existence=True)].append(q)
sage: l7[True]
[169, 337, 379, 421, 463, 547, 631, 673, 757, 841, 883, 967, ...,  4621, 4957,␣
↪5167]
sage: l7[Unknown]
[43, 127, 211, 2017, 2143, 2269, 2311, 2437, 2521, 2647, ..., 4999, 5041, 5209]
sage: l7[False]
[]
```

```
>>> from sage.all import *
>>> def prime_power_mod(r, m):
...     k = m+r
...     while True:
...         if is_prime_power(k):
...             yield k
...         k += m

>>> # needs sage.libs.pari
>>> from itertools import islice
>>> l6 = {True: [], False: [], Unknown: []}
>>> for q in islice(prime_power_mod(Integer(1),Integer(30)), int(Integer(60))):
...     l6[designs.difference_family(q,Integer(6),existence=True)].append(q)
>>> l6[True]
[31, 121, 151, 181, 211, ...,  3061, 3121, 3181]
>>> l6[Unknown]
[61]
>>> l6[False]
[]

>>> # needs sage.libs.pari
>>> l7 = {True: [], False: [], Unknown: []}
>>> for q in islice(prime_power_mod(Integer(1),Integer(42)), int(Integer(60))):
...     l7[designs.difference_family(q,Integer(7),existence=True)].append(q)
>>> l7[True]
[169, 337, 379, 421, 463, 547, 631, 673, 757, 841, 883, 967, ...,  4621, 4957,␣
↪5167]
>>> l7[Unknown]
[43, 127, 211, 2017, 2143, 2269, 2311, 2437, 2521, 2647, ..., 4999, 5041, 5209]
>>> l7[False]
[]
```

List available constructions:

```
sage: for v in range(2,100):                                                    #␣
↪needs sage.libs.pari
....:     constructions = []
```

(continues on next page)

(continued from previous page)

```
....:     for k in range(2,10):
....:         for l in range(1,10):
....:             ret = designs.difference_family(v,k,l,existence=True)
....:             if ret is True:
....:                 constructions.append((k,l))
....:                 _ = designs.difference_family(v,k,l)
....:     if constructions:
....:         print("%2d: %s"%(v, ', '.join('(%d,%d)'%(k,l) for k,l in␣
↪constructions)))
 3: (2,1)
 4: (3,2)
 5: (2,1), (4,3)
 6: (5,4)
 7: (2,1), (3,1), (3,2), (4,2), (6,5)
 8: (7,6)
 9: (2,1), (4,3), (8,7)
10: (9,8)
11: (2,1), (4,6), (5,2), (5,4), (6,3)
13: (2,1), (3,1), (3,2), (4,1), (4,3), (5,5), (6,5)
15: (3,1), (4,6), (5,6), (7,3), (7,6)
16: (3,2), (5,4), (6,2)
17: (2,1), (4,3), (5,5), (8,7)
19: (2,1), (3,1), (3,2), (4,2), (6,5), (9,4), (9,8)
21: (3,1), (4,3), (5,1), (6,3), (6,5)
22: (4,2), (6,5), (7,4), (8,8)
23: (2,1)
25: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (7,7), (8,7)
27: (2,1), (3,1)
28: (3,2), (6,5)
29: (2,1), (4,3), (7,3), (7,6), (8,4), (8,6)
31: (2,1), (3,1), (3,2), (4,2), (5,2), (5,4), (6,1), (6,5)
33: (3,1), (5,5), (6,5)
34: (4,2)
35: (5,2)
37: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (9,2), (9,8)
39: (3,1), (6,5)
40: (3,2), (4,1)
41: (2,1), (4,3), (5,1), (5,4), (6,3), (8,7)
43: (2,1), (3,1), (3,2), (4,2), (6,5), (7,2), (7,3), (7,6), (8,4)
45: (3,1), (5,1)
46: (4,2), (6,2)
47: (2,1)
49: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,3)
51: (3,1), (5,2), (6,3)
52: (4,1)
53: (2,1), (4,3)
55: (3,1), (9,4)
57: (3,1), (7,3), (8,1)
59: (2,1)
61: (2,1), (3,1), (3,2), (4,1), (4,3), (5,1), (5,4), (6,2), (6,3), (6,5)
63: (3,1)
64: (3,2), (4,1), (7,2), (7,6), (9,8)
```

(continues on next page)

(continued from previous page)

```
65: (5,1)
67: (2,1), (3,1), (3,2), (6,5)
69: (3,1)
71: (2,1), (5,2), (5,4), (7,3), (7,6), (8,4)
73: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,1), (9,8)
75: (3,1), (5,2)
76: (4,1)
79: (2,1), (3,1), (3,2), (6,5)
81: (2,1), (3,1), (4,3), (5,1), (5,4), (8,7)
83: (2,1)
85: (4,1), (7,2), (7,3), (8,2)
89: (2,1), (4,3), (8,7)
91: (6,1), (7,1)
97: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,3)
```

```
>>> from sage.all import *
>>> for v in range(Integer(2),Integer(100)):                                    ␣
↪             # needs sage.libs.pari
...     constructions = []
...     for k in range(Integer(2),Integer(10)):
...         for l in range(Integer(1),Integer(10)):
...             ret = designs.difference_family(v,k,l,existence=True)
...             if ret is True:
...                 constructions.append((k,l))
...                 _ = designs.difference_family(v,k,l)
...     if constructions:
...         print("%2d: %s"%(v, ', '.join('(%d,%d)'%(k,l) for k,l in␣
↪constructions)))
 3: (2,1)
 4: (3,2)
 5: (2,1), (4,3)
 6: (5,4)
 7: (2,1), (3,1), (3,2), (4,2), (6,5)
 8: (7,6)
 9: (2,1), (4,3), (8,7)
10: (9,8)
11: (2,1), (4,6), (5,2), (5,4), (6,3)
13: (2,1), (3,1), (3,2), (4,1), (4,3), (5,5), (6,5)
15: (3,1), (4,6), (5,6), (7,3), (7,6)
16: (3,2), (5,4), (6,2)
17: (2,1), (4,3), (5,5), (8,7)
19: (2,1), (3,1), (3,2), (4,2), (6,5), (9,4), (9,8)
21: (3,1), (4,3), (5,1), (6,3), (6,5)
22: (4,2), (6,5), (7,4), (8,8)
23: (2,1)
25: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (7,7), (8,7)
27: (2,1), (3,1)
28: (3,2), (6,5)
29: (2,1), (4,3), (7,3), (7,6), (8,4), (8,6)
31: (2,1), (3,1), (3,2), (4,2), (5,2), (5,4), (6,1), (6,5)
33: (3,1), (5,5), (6,5)
34: (4,2)
```

(continues on next page)

(continued from previous page)

```
35: (5,2)
37: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (9,2), (9,8)
39: (3,1), (6,5)
40: (3,2), (4,1)
41: (2,1), (4,3), (5,1), (5,4), (6,3), (8,7)
43: (2,1), (3,1), (3,2), (4,2), (6,5), (7,2), (7,3), (7,6), (8,4)
45: (3,1), (5,1)
46: (4,2), (6,2)
47: (2,1)
49: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,3)
51: (3,1), (5,2), (6,3)
52: (4,1)
53: (2,1), (4,3)
55: (3,1), (9,4)
57: (3,1), (7,3), (8,1)
59: (2,1)
61: (2,1), (3,1), (3,2), (4,1), (4,3), (5,1), (5,4), (6,2), (6,3), (6,5)
63: (3,1)
64: (3,2), (4,1), (7,2), (7,6), (9,8)
65: (5,1)
67: (2,1), (3,1), (3,2), (6,5)
69: (3,1)
71: (2,1), (5,2), (5,4), (7,3), (7,6), (8,4)
73: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,1), (9,8)
75: (3,1), (5,2)
76: (4,1)
79: (2,1), (3,1), (3,2), (6,5)
81: (2,1), (3,1), (4,3), (5,1), (5,4), (8,7)
83: (2,1)
85: (4,1), (7,2), (7,3), (8,2)
89: (2,1), (4,3), (8,7)
91: (6,1), (7,1)
97: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,3)
```

Todo

Implement recursive constructions from Buratti “Recursive for difference matrices and relative difference families” (1998) and Jungnickel “Composition theorems for difference families and regular planes” (1978)

sage.combinat.designs.difference_family.**get_fixed_relative_difference_set**(*G*, *rel_diff_set*, *as_elements=False*)

Construct an equivalent relative difference set fixed by the size of the set.

Given a relative difference set $R(q+1, q-1, q, 1)$, it is possible to find a translation of this set fixed by q (see Section 3 of [Spe1975]). We say that a set is fixed by t if $\{td | d \in R\} = R$.

In addition, the set returned by this function will contain the element 0. This is needed in the construction of supplementary difference sets (see `supplementary_difference_set_from_rel_diff_set()`).

INPUT:

- `G` – a group, of which `rel_diff_set` is a subset

- `rel_diff_set` – the relative difference set
- `as_elements` – boolean (default: `False`); if `True`, the list returned will contain elements of the abelian group (this may slow down the computation considerably)

OUTPUT:

By default, this function returns the set as a list of integers. However, if `as_elements=True` it will return the set as a list containing elements of the abelian group. If no such set can be found, the function will raise an error.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import relative_difference_set_
↪from_m_sequence, get_fixed_relative_difference_set
sage: G, s1 = relative_difference_set_from_m_sequence(5, 2, return_group=True)   #␣
↪needs sage.libs.pari sage.modules
sage: get_fixed_relative_difference_set(G, s1)   # random                         #␣
↪needs sage.libs.pari sage.modules
[2, 10, 19, 23, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import relative_difference_set_
↪from_m_sequence, get_fixed_relative_difference_set
>>> G, s1 = relative_difference_set_from_m_sequence(Integer(5), Integer(2),␣
↪return_group=True)  # needs sage.libs.pari sage.modules
>>> get_fixed_relative_difference_set(G, s1)   # random                         #␣
↪needs sage.libs.pari sage.modules
[2, 10, 19, 23, 0]
```

If `as_elements=True`, the result will contain elements of the group:

```
sage: get_fixed_relative_difference_set(G, s1, as_elements=True)   # random       #␣
↪needs sage.libs.pari sage.modules
[(2), (10), (19), (23), (0)]
```

```
>>> from sage.all import *
>>> get_fixed_relative_difference_set(G, s1, as_elements=True)   # random       #␣
↪needs sage.libs.pari sage.modules
[(2), (10), (19), (23), (0)]
```

`sage.combinat.designs.difference_family.`**`group_law`**`(`G`)`

Return a triple `(identity, operation, inverse)` that define the operations on the group `G`.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import group_law
sage: group_law(Zmod(3))
(0, <built-in function add>, <built-in function neg>)
sage: group_law(SymmetricGroup(5))                                                #␣
↪needs sage.groups
((), <built-in function mul>, <built-in function inv>)
sage: group_law(VectorSpace(QQ, 3))                                               #␣
↪needs sage.modules
((0, 0, 0), <built-in function add>, <built-in function neg>)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import group_law
>>> group_law(Zmod(Integer(3)))
(0, <built-in function add>, <built-in function neg>)
>>> group_law(SymmetricGroup(Integer(5)))                                    ␣
↪     # needs sage.groups
((), <built-in function mul>, <built-in function inv>)
>>> group_law(VectorSpace(QQ, Integer(3)))                                   ␣
↪     # needs sage.modules
((0, 0, 0), <built-in function add>, <built-in function neg>)
```

sage.combinat.designs.difference_family.**hadamard_difference_set_product**(*G1*, *D1*, *G2*, *D2*)

Make a product of two Hadamard difference sets.

This product construction appears in [Tu1984].

INPUT:

- G1, D1, G2, D2 – two Hadamard difference sets

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import hadamard_difference_set_
↪product
sage: from sage.combinat.designs.difference_family import is_difference_family

sage: G1,D1 = designs.difference_family(16,6,2)                              #␣
↪needs sage.rings.finite_rings
sage: G2,D2 = designs.difference_family(36,15,6)                             #␣
↪needs sage.rings.finite_rings

sage: G11,D11 = hadamard_difference_set_product(G1,D1,G1,D1)                 #␣
↪needs sage.rings.finite_rings
sage: assert is_difference_family(G11, D11, 256, 120, 56)                    #␣
↪needs sage.rings.finite_rings
sage: assert designs.difference_family(256, 120, 56, existence=True) is True     #␣
↪needs sage.rings.finite_rings

sage: G12,D12 = hadamard_difference_set_product(G1,D1,G2,D2)                 #␣
↪needs sage.rings.finite_rings
sage: assert is_difference_family(G12, D12, 576, 276, 132)                   #␣
↪needs sage.rings.finite_rings
sage: assert designs.difference_family(576, 276, 132, existence=True) is True    #␣
↪needs sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import hadamard_difference_set_
↪product
>>> from sage.combinat.designs.difference_family import is_difference_family

>>> G1,D1 = designs.difference_family(Integer(16),Integer(6),Integer(2))      ␣
↪                         # needs sage.rings.finite_rings
>>> G2,D2 = designs.difference_family(Integer(36),Integer(15),Integer(6))     ␣
↪                         # needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
>>> G11,D11 = hadamard_difference_set_product(G1,D1,G1,D1)                   #␣
↪needs sage.rings.finite_rings
>>> assert is_difference_family(G11, D11, Integer(256), Integer(120),␣
↪Integer(56))                       # needs sage.rings.finite_rings
>>> assert designs.difference_family(Integer(256), Integer(120), Integer(56),␣
↪existence=True) is True     # needs sage.rings.finite_rings

>>> G12,D12 = hadamard_difference_set_product(G1,D1,G2,D2)                   #␣
↪needs sage.rings.finite_rings
>>> assert is_difference_family(G12, D12, Integer(576), Integer(276),␣
↪Integer(132))                      # needs sage.rings.finite_rings
>>> assert designs.difference_family(Integer(576), Integer(276), Integer(132),␣
↪existence=True) is True    # needs sage.rings.finite_rings
```

sage.combinat.designs.difference_family.**hadamard_difference_set_product_parameters**()

Check whether a product construction is available for Hadamard difference set with parameter `N`.

This function looks for two integers N_1 and N_2 greater than 1 and so that $N = 2N_1N_2$ and there exists Hadamard difference set with parameters $(4N_i^2, 2N_i^2 - N_i, N_i^2 - N_i)$. If such pair exists, the output is the pair `(N_1, N_2)` otherwise it is `None`.

INPUT:

- `N` – positive integer

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import hadamard_difference_set_
↪product_parameters
sage: hadamard_difference_set_product_parameters(8)                          #␣
↪needs sage.rings.finite_rings
(2, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import hadamard_difference_set_
↪product_parameters
>>> hadamard_difference_set_product_parameters(Integer(8))                    ␣
↪     # needs sage.rings.finite_rings
(2, 2)
```

sage.combinat.designs.difference_family.**is_difference_family**(*G*, *D*, *v=None*, *k=None*, *l=None*, *verbose=False*)

Check whether `D` forms a difference family in the group `G`.

INPUT:

- `G` – group of cardinality `v`
- `D` – set of `k`-subsets of `G`
- `v`, `k`, `l` – (optional) parameters of the difference family
- `verbose` – boolean (default: `False`); whether to print additional information

See also

difference_family()

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import is_difference_family
sage: G = Zmod(21)
sage: D = [[0,1,4,14,16]]
sage: is_difference_family(G, D, 21, 5)
True

sage: G = Zmod(41)
sage: D = [[0,1,4,11,29],[0,2,8,17,21]]
sage: is_difference_family(G, D, verbose=True)
Too few:
  5 is obtained 0 times in blocks []
  14 is obtained 0 times in blocks []
  27 is obtained 0 times in blocks []
  36 is obtained 0 times in blocks []
Too much:
  4 is obtained 2 times in blocks [0, 1]
  13 is obtained 2 times in blocks [0, 1]
  28 is obtained 2 times in blocks [0, 1]
  37 is obtained 2 times in blocks [0, 1]
False
sage: D = [[0,1,4,11,29],[0,2,8,17,22]]
sage: is_difference_family(G, D)
True

sage: G = Zmod(61)
sage: D = [[0,1,3,13,34],[0,4,9,23,45],[0,6,17,24,32]]
sage: is_difference_family(G, D)
True

sage: # needs sage.modules
sage: G = AdditiveAbelianGroup([3]*4)
sage: a,b,c,d = G.gens()
sage: D = [[d, -a+d, -c+d, a-b-d, b+c+d],
....:      [c, a+b-d, -b+c, a-b+d, a+b+c],
....:      [-a-b+c+d, a-b-c-d, -a+c-d, b-c+d, a+b],
....:      [-b-d, a+b+d, a-b+c-d, a-b+c, -b+c+d]]
sage: is_difference_family(G, D)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import is_difference_family
>>> G = Zmod(Integer(21))
>>> D = [[Integer(0),Integer(1),Integer(4),Integer(14),Integer(16)]]
>>> is_difference_family(G, D, Integer(21), Integer(5))
True
```

(continues on next page)

(continued from previous page)

```
>>> G = Zmod(Integer(41))
>>> D = [[Integer(0),Integer(1),Integer(4),Integer(11),Integer(29)],[Integer(0),
↪Integer(2),Integer(8),Integer(17),Integer(21)]]
>>> is_difference_family(G, D, verbose=True)
Too few:
  5 is obtained 0 times in blocks []
  14 is obtained 0 times in blocks []
  27 is obtained 0 times in blocks []
  36 is obtained 0 times in blocks []
Too much:
  4 is obtained 2 times in blocks [0, 1]
  13 is obtained 2 times in blocks [0, 1]
  28 is obtained 2 times in blocks [0, 1]
  37 is obtained 2 times in blocks [0, 1]
False
>>> D = [[Integer(0),Integer(1),Integer(4),Integer(11),Integer(29)],[Integer(0),
↪Integer(2),Integer(8),Integer(17),Integer(22)]]
>>> is_difference_family(G, D)
True

>>> G = Zmod(Integer(61))
>>> D = [[Integer(0),Integer(1),Integer(3),Integer(13),Integer(34)],[Integer(0),
↪Integer(4),Integer(9),Integer(23),Integer(45)],[Integer(0),Integer(6),
↪Integer(17),Integer(24),Integer(32)]]
>>> is_difference_family(G, D)
True

>>> # needs sage.modules
>>> G = AdditiveAbelianGroup([Integer(3)]*Integer(4))
>>> a,b,c,d = G.gens()
>>> D = [[d, -a+d, -c+d, a-b-d, b+c+d],
...      [c, a+b-d, -b+c, a-b+d, a+b+c],
...      [-a-b+c+d, a-b-c-d, -a+c-d, b-c+d, a+b],
...      [-b-d, a+b+d, a-b+c-d, a-b+c, -b+c+d]]
>>> is_difference_family(G, D)
True
```

The following example has a third block with a non-trivial stabilizer:

```
sage: G = Zmod(15)
sage: D = [[0,1,4],[0,2,9],[0,5,10]]
sage: is_difference_family(G,D,verbose=True)
It is a (15,3,1)-difference family
True
```

```
>>> from sage.all import *
>>> G = Zmod(Integer(15))
>>> D = [[Integer(0),Integer(1),Integer(4)],[Integer(0),Integer(2),Integer(9)],
↪[Integer(0),Integer(5),Integer(10)]]
>>> is_difference_family(G,D,verbose=True)
It is a (15,3,1)-difference family
True
```

The function also supports multiplicative groups (non necessarily Abelian):

```
sage: # needs sage.groups
sage: G = DihedralGroup(8)
sage: x,y = G.gens()
sage: i = G.one()
sage: D1 = [[i,x,x^4], [i,x^2, y*x], [i,x^5,y], [i,x^6,y*x^2], [i,x^7,y*x^5]]
sage: is_difference_family(G, D1, 16, 3, 2)
True
sage: from sage.combinat.designs.bibd import BIBD_from_difference_family
sage: bibd = BIBD_from_difference_family(G, D1, lambd=2)
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> G = DihedralGroup(Integer(8))
>>> x,y = G.gens()
>>> i = G.one()
>>> D1 = [[i,x,x**Integer(4)], [i,x**Integer(2), y*x], [i,x**Integer(5),y], [i,
↪x**Integer(6),y*x**Integer(2)], [i,x**Integer(7),y*x**Integer(5)]]
>>> is_difference_family(G, D1, Integer(16), Integer(3), Integer(2))
True
>>> from sage.combinat.designs.bibd import BIBD_from_difference_family
>>> bibd = BIBD_from_difference_family(G, D1, lambd=Integer(2))
```

sage.combinat.designs.difference_family.**is_fixed_relative_difference_set**(R, q)

Check if the relative difference set R is fixed by q.

A relative difference set R is fixed by q if $\{qd | d \in R\} = R$ (see Section 3 of [Spe1975]).

INPUT:

- R – list containing elements of an abelian group; the relative difference set
- q – integer

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.combinat.designs.difference_family import relative_difference_set_
↪from_m_sequence, get_fixed_relative_difference_set, is_fixed_relative_
↪difference_set
sage: G, s1 = relative_difference_set_from_m_sequence(7, 2, return_group=True)   #␣
↪needs sage.libs.pari
sage: s2 = get_fixed_relative_difference_set(G, s1, as_elements=True)            #␣
↪needs sage.libs.pari
sage: is_fixed_relative_difference_set(s2, len(s2))                              #␣
↪needs sage.libs.pari
True
sage: G = AdditiveAbelianGroup([15])
sage: s3 = [G[1], G[2], G[3], G[4]]
sage: is_fixed_relative_difference_set(s3, len(s3))
False
```

```
>>> from sage.all import *
>>> # needs sage.modules
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.designs.difference_family import relative_difference_set_
↪from_m_sequence, get_fixed_relative_difference_set, is_fixed_relative_
↪difference_set
>>> G, s1 = relative_difference_set_from_m_sequence(Integer(7), Integer(2),␣
↪return_group=True)  # needs sage.libs.pari
>>> s2 = get_fixed_relative_difference_set(G, s1, as_elements=True)          #␣
↪needs sage.libs.pari
>>> is_fixed_relative_difference_set(s2, len(s2))                            #␣
↪needs sage.libs.pari
True
>>> G = AdditiveAbelianGroup([Integer(15)])
>>> s3 = [G[Integer(1)], G[Integer(2)], G[Integer(3)], G[Integer(4)]]
>>> is_fixed_relative_difference_set(s3, len(s3))
False
```

If the relative difference set does not contain elements of the group, the method returns false:

```
sage: G, s1 = relative_difference_set_from_m_sequence(7, 2, return_group=True)   #␣
↪needs sage.libs.pari sage.modules
sage: s2 = get_fixed_relative_difference_set(G, s1, as_elements=False)          #␣
↪needs sage.libs.pari sage.modules
sage: is_fixed_relative_difference_set(s2, len(s2))                             #␣
↪needs sage.libs.pari sage.modules
False
```

```
>>> from sage.all import *
>>> G, s1 = relative_difference_set_from_m_sequence(Integer(7), Integer(2),␣
↪return_group=True)  # needs sage.libs.pari sage.modules
>>> s2 = get_fixed_relative_difference_set(G, s1, as_elements=False)         #␣
↪needs sage.libs.pari sage.modules
>>> is_fixed_relative_difference_set(s2, len(s2))                            #␣
↪needs sage.libs.pari sage.modules
False
```

sage.combinat.designs.difference_family.**is_relative_difference_set**(*R*, *G*, *H*, *params*, *verbose=False*)

Check if `R` is a difference set of `G` relative to `H`, with the given parameters.

This function checks that G, H and R have the orders specified in the parameters, and that R satisfies the definition of relative difference set (from [EB1966]): the collection of differences $r - s$, $r, s \in R$, $r \neq s$ contains only elements of G which are not in H, and contains every such element exactly d times.

INPUT:

- `R` – list; the relative diffeence set of length k
- `G` – an additive abelian group of order mn
- `H` – list; a submodule of `G` of order n
- `params` – tuple in the form (m, n, k, d)
- `verbose` – boolean (default: `False`); if `True`, the function will be verbose when the sequences do not satisfy the constraints

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import _get_submodule_of_order,
↪ relative_difference_set_from_m_sequence, is_relative_difference_set
sage: q, N = 5, 2
sage: params = ((q^N-1) // (q-1), q - 1, q^(N-1), q^(N-2))
sage: G, R = relative_difference_set_from_m_sequence(q, N, return_group=True)    #␣
↪needs sage.libs.pari sage.modules
sage: H = _get_submodule_of_order(G, q - 1)                                      #␣
↪needs sage.libs.pari sage.modules
sage: is_relative_difference_set(R, G, H, params)                                #␣
↪needs sage.libs.pari sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import _get_submodule_of_order,␣
↪relative_difference_set_from_m_sequence, is_relative_difference_set
>>> q, N = Integer(5), Integer(2)
>>> params = ((q**N-Integer(1)) // (q-Integer(1)), q - Integer(1), q**(N-
↪Integer(1)), q**(N-Integer(2)))
>>> G, R = relative_difference_set_from_m_sequence(q, N, return_group=True)    #␣
↪needs sage.libs.pari sage.modules
>>> H = _get_submodule_of_order(G, q - Integer(1))                               ␣
↪      # needs sage.libs.pari sage.modules
>>> is_relative_difference_set(R, G, H, params)                                #␣
↪needs sage.libs.pari sage.modules
True
```

If we pass the `verbose` argument, the function will explain why it failed:

```
sage: R2 = [G[1], G[2], G[3], G[5], G[6]]                                        #␣
↪needs sage.libs.pari sage.modules
sage: is_relative_difference_set(R2, G, H, params, verbose=True)                 #␣
↪needs sage.libs.pari sage.modules
There is a value in the difference set which is not repeated d times
False
```

```
>>> from sage.all import *
>>> R2 = [G[Integer(1)], G[Integer(2)], G[Integer(3)], G[Integer(5)],␣
↪G[Integer(6)]]                                         # needs sage.libs.pari␣
↪sage.modules
>>> is_relative_difference_set(R2, G, H, params, verbose=True)                 #␣
↪needs sage.libs.pari sage.modules
There is a value in the difference set which is not repeated d times
False
```

sage.combinat.designs.difference_family.**is_supplementary_difference_set**(*Ks*, *v=None*, *lmbda=None*, *G=None*, *verbose=False*)

Check that the sets in `Ks` are $n-\{v; k_1, ..., k_n; \lambda\}$ supplementary difference sets over group `G` of order `v`.

From the definition in [Spe1975]: let $S_1, S_2, ..., S_n$ be n subsets of a group G of order v such that $|S_i| = k_i$. If, for each $g \in G$, $g \neq 0$, the total number of solutions of $a_i - a'_i = g$, with $a_i, a'_i \in S_i$ is λ, then $S_1, S_2, ..., S_n$ are $n-\{v; k_1, ..., k_n; \lambda\}$ supplementary difference sets.

One of the parameters `v` or `G` must always be specified. If `G` is not given, the function will use an `AdditiveAbelianGroup` of order `v`.

INPUT:

- `Ks` – list of sets to be checked
- `v` – integer; the parameter v of the supplementary difference sets
- `lmbda` – integer; the parameter λ of the supplementary difference sets
- `G` – a group of order v
- `verbose` – boolean (default: `False`); if `True`, the function will be verbose when the sets do not satisfy the constraints

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import supplementary_
↪difference_set_from_rel_diff_set, is_supplementary_difference_set
sage: G, [S1, S2, S3, S4] = supplementary_difference_set_from_rel_diff_set(17)   #␣
↪needs sage.modules sage.rings.finite_rings
sage: is_supplementary_difference_set([S1, S2, S3, S4], lmbda=16, G=G)          #␣
↪needs sage.modules sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import supplementary_difference_
↪set_from_rel_diff_set, is_supplementary_difference_set
>>> G, [S1, S2, S3, S4] = supplementary_difference_set_from_rel_diff_
↪set(Integer(17))  # needs sage.modules sage.rings.finite_rings
>>> is_supplementary_difference_set([S1, S2, S3, S4], lmbda=Integer(16), G=G)    ␣
↪      # needs sage.modules sage.rings.finite_rings
True
```

The parameter `v` can be given instead of `G`:

```
sage: is_supplementary_difference_set([S1, S2, S3, S4], v=16, lmbda=16)          #␣
↪needs sage.modules sage.rings.finite_rings
True
sage: is_supplementary_difference_set([S1, S2, S3, S4], v=20, lmbda=16)          #␣
↪needs sage.modules sage.rings.finite_rings
False
```

```
>>> from sage.all import *
>>> is_supplementary_difference_set([S1, S2, S3, S4], v=Integer(16),␣
↪lmbda=Integer(16))          # needs sage.modules sage.rings.finite_rings
True
>>> is_supplementary_difference_set([S1, S2, S3, S4], v=Integer(20),␣
↪lmbda=Integer(16))          # needs sage.modules sage.rings.finite_rings
False
```

If `verbose=True`, the function will be verbose:

```
sage: is_supplementary_difference_set([S1, S2, S3, S4], lmbda=14, G=G,          #␣
↪needs sage.modules sage.rings.finite_rings
....:                                 verbose=True)
```

(continues on next page)

(continued from previous page)

```
Number of pairs with difference (1) is 16, but lambda is 14
False
```

```
>>> from sage.all import *
>>> is_supplementary_difference_set([S1, S2, S3, S4], lmbda=Integer(14), G=G,     ␣
↪     # needs sage.modules sage.rings.finite_rings
...                                 verbose=True)
Number of pairs with difference (1) is 16, but lambda is 14
False
```

See also

supplementary_difference_set_from_rel_diff_set()

sage.combinat.designs.difference_family.**mcfarland_1973_construction**(q, s)

Return a difference set.

The difference set returned has the following parameters

$$v = \frac{q^{s+1}(q^{s+1}+q-2)}{q-1}, k = \frac{q^s(q^{s+1}-1)}{q-1}, \lambda = \frac{q^s(q^s-1)}{q-1}$$

This construction is due to [McF1973].

INPUT:

- q, s – integers; parameters for the difference set (see the above formulas for the expression of v, k, l in terms of q and s)

See also

The function *are_mcfarland_1973_parameters()* makes the translation between the parameters (q, s) corresponding to a given triple (v, k, λ).

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import (
....:     mcfarland_1973_construction, is_difference_family)

sage: G,D = mcfarland_1973_construction(3, 1)                                   #␣
↪needs sage.modules
sage: assert is_difference_family(G, D, 45, 12, 3)                              #␣
↪needs sage.modules

sage: G,D = mcfarland_1973_construction(2, 2)                                   #␣
↪needs sage.modules
sage: assert is_difference_family(G, D, 64, 28, 12)                             #␣
↪needs sage.modules
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import (
...     mcfarland_1973_construction, is_difference_family)

>>> G,D = mcfarland_1973_construction(Integer(3), Integer(1))                   ␣
↪              # needs sage.modules
>>> assert is_difference_family(G, D, Integer(45), Integer(12), Integer(3))     ␣
↪                       # needs sage.modules

>>> G,D = mcfarland_1973_construction(Integer(2), Integer(2))                   ␣
↪              # needs sage.modules
>>> assert is_difference_family(G, D, Integer(64), Integer(28), Integer(12))    ␣
↪                       # needs sage.modules
```

sage.combinat.designs.difference_family.**one_cyclic_tiling**(A, n)

Given a subset A of the cyclic additive group $G = Z/nZ$ return another subset B so that $A + B = G$ and $|A||B| = n$ (i.e. any element of G is uniquely expressed as a sum $a + b$ with a in A and b in B).

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import one_cyclic_tiling
sage: tile = [0,2,4]
sage: m = one_cyclic_tiling(tile,6); m
[0, 3]
sage: sorted((i+j)%6 for i in tile for j in m)
[0, 1, 2, 3, 4, 5]

sage: def print_tiling(tile, translat, n):
....:     for x in translat:
....:         print(''.join('X' if (i-x)%n in tile else '.' for i in range(n)))

sage: tile = [0, 1, 2, 7]
sage: m = one_cyclic_tiling(tile, 12)
sage: print_tiling(tile, m, 12)
XXX....X....
....XXX....X
...X....XXX.

sage: tile = [0, 1, 5]
sage: m = one_cyclic_tiling(tile, 12)
sage: print_tiling(tile, m, 12)
XX...X......
...XX...X...
......XX...X
..X......XX.

sage: tile = [0, 2]
sage: m = one_cyclic_tiling(tile, 8)
sage: print_tiling(tile, m, 8)
X.X.....
....X.X.
.X.X....
.....X.X
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import one_cyclic_tiling
>>> tile = [Integer(0),Integer(2),Integer(4)]
>>> m = one_cyclic_tiling(tile,Integer(6)); m
[0, 3]
>>> sorted((i+j)%Integer(6) for i in tile for j in m)
[0, 1, 2, 3, 4, 5]

>>> def print_tiling(tile, translat, n):
...     for x in translat:
...         print(''.join('X' if (i-x)%n in tile else '.' for i in range(n)))

>>> tile = [Integer(0), Integer(1), Integer(2), Integer(7)]
>>> m = one_cyclic_tiling(tile, Integer(12))
>>> print_tiling(tile, m, Integer(12))
XXX....X....
....XXX....X
...X....XXX.

>>> tile = [Integer(0), Integer(1), Integer(5)]
>>> m = one_cyclic_tiling(tile, Integer(12))
>>> print_tiling(tile, m, Integer(12))
XX...X......
...XX...X...
......XX...X
..X......XX.

>>> tile = [Integer(0), Integer(2)]
>>> m = one_cyclic_tiling(tile, Integer(8))
>>> print_tiling(tile, m, Integer(8))
X.X.....
....X.X.
.X.X....
.....X.X
```

ALGORITHM:

Uses dancing links `sage.combinat.dlx`

`sage.combinat.designs.difference_family.`**`one_radical_difference_family`**(K, k)

Search for a radical difference family on `K` using dancing links algorithm.

For the definition of radical difference family, see `radical_difference_family()`. Here, we consider only radical difference family with $\lambda = 1$.

INPUT:

- `K` – a finite field of cardinality q
- `k` – positive integer so that $k(k-1)$ divides $q-1$

OUTPUT: either a difference family or `None` if it does not exist

ALGORITHM:

The existence of a radical difference family is equivalent to a one dimensional tiling (or packing) problem in a cyclic group. This subsequent problem is solved by a call to the function `one_cyclic_tiling()`.

Let K^* be the multiplicative group of the finite field K. A radical family has the form $\mathcal{B} = \{x_1 B, \ldots, x_k B\}$, where $B = \{x : x^k = 1\}$ (for k odd) or $B = \{x : x^{k-1} = 1\} \cup \{0\}$ (for k even). Equivalently, K^* decomposes as:

$$K^* = \Delta(x_1 B) \cup \cdots \cup \Delta(x_k B) = x_1 \Delta B \cup \cdots \cup x_k \Delta B.$$

We observe that $C = B \backslash 0$ is a subgroup of the (cyclic) group K^*, that can thus be generated by some element r. Furthermore, we observe that ΔB is always a union of cosets of $\pm C$ (which is twice larger than C).

$$\begin{array}{lll} (k \text{ odd}) & \Delta B & = \{r^i - r^j : r^i \neq r^j\} & = \pm C \cdot \{r^i - 1 : 0 < i \leq m\} \\ (k \text{ even}) & \Delta B & = \{r^i - r^j : r^i \neq r^j\} \cup C & = \pm C \cdot \{r^i - 1 : 0 < i < m\} \cup \pm C \end{array}$$

where

$$(k \text{ odd})\ m = (k-1)/2 \quad \text{and} \quad (k \text{ even})\ m = k/2.$$

Consequently, $\mathcal{B} = \{x_1 B, \ldots, x_k B\}$ is a radical difference family if and only if $\{x_1(\Delta B/(\pm C)), \ldots, x_k(\Delta B/(\pm C))\}$ is a partition of the cyclic group $K^*/(\pm C)$.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import (
....:     one_radical_difference_family,
....:     is_difference_family)

sage: one_radical_difference_family(GF(13),4)                              #␣
↪needs sage.rings.finite_rings
[[0, 1, 3, 9]]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import (
...     one_radical_difference_family,
...     is_difference_family)

>>> one_radical_difference_family(GF(Integer(13)),Integer(4))              ␣
↪               # needs sage.rings.finite_rings
[[0, 1, 3, 9]]
```

The parameters that appear in [Bu95]:

```
sage: df = one_radical_difference_family(GF(449), 8); df                    #␣
↪needs sage.rings.finite_rings
[[0, 1, 18, 25, 176, 324, 359, 444],
 [0, 9, 88, 162, 222, 225, 237, 404],
 [0, 11, 140, 198, 275, 357, 394, 421],
 [0, 40, 102, 249, 271, 305, 388, 441],
 [0, 49, 80, 93, 161, 204, 327, 433],
 [0, 70, 99, 197, 230, 362, 403, 435],
 [0, 121, 141, 193, 293, 331, 335, 382],
 [0, 191, 285, 295, 321, 371, 390, 392]]
sage: is_difference_family(GF(449), df, 449, 8, 1)                          #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> df = one_radical_difference_family(GF(Integer(449)), Integer(8)); df                
↪               # needs sage.rings.finite_rings
[[0, 1, 18, 25, 176, 324, 359, 444],
 [0, 9, 88, 162, 222, 225, 237, 404],
 [0, 11, 140, 198, 275, 357, 394, 421],
 [0, 40, 102, 249, 271, 305, 388, 441],
 [0, 49, 80, 93, 161, 204, 327, 433],
 [0, 70, 99, 197, 230, 362, 403, 435],
 [0, 121, 141, 193, 293, 331, 335, 382],
 [0, 191, 285, 295, 321, 371, 390, 392]]
>>> is_difference_family(GF(Integer(449)), df, Integer(449), Integer(8),
↪Integer(1))                                # needs sage.rings.finite_rings
True
```

sage.combinat.designs.difference_family.**radical_difference_family**(*K*, *k*, *l=1*, *existence=False*, *check=True*)

Return a `(v,k,l)`-radical difference family.

Let fix an integer k and a prime power $q = tk(k-1)+1$. Let K be a field of cardinality q. A $(q,k,1)$-difference family is *radical* if its base blocks are either: a coset of the k-th root of unity for k odd or a coset of $k-1$-th root of unity and 0 if k is even (the number t is the number of blocks of that difference family).

The terminology comes from M. Buratti article [Bu95] but the first constructions go back to R. Wilson [Wi72].

INPUT:

- `K` – a finite field
- `k` – positive integer; the size of the blocks
- `l` – integer (default: 1); the λ parameter
- `existence` – if `True`, then return either `True` if Sage knows how to build such design, `Unknown` if it does not and `False` if it knows that the design does not exist
- `check` – boolean (default: `True`); if `True` then the result of the computation is checked before being returned. This should not be needed but ensures that the output is correct

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import radical_difference_
↪family

sage: radical_difference_family(GF(73), 9)                                    #
↪needs sage.rings.finite_rings
[[1, 2, 4, 8, 16, 32, 37, 55, 64]]

sage: radical_difference_family(GF(281), 5)                                   #
↪needs sage.rings.finite_rings
[[1, 86, 90, 153, 232],
 [4, 50, 63, 79, 85],
 [5, 36, 149, 169, 203],
 [7, 40, 68, 219, 228],
 [9, 121, 212, 248, 253],
 [29, 81, 222, 246, 265],
 [31, 137, 167, 247, 261],
```

(continues on next page)

(continued from previous page)

```
 [32, 70, 118, 119, 223],
 [39, 56, 66, 138, 263],
 [43, 45, 116, 141, 217],
 [98, 101, 109, 256, 279],
 [106, 124, 145, 201, 267],
 [111, 123, 155, 181, 273],
 [156, 209, 224, 264, 271]]

sage: for k in range(5,10):                                                     #␣
↪needs sage.rings.finite_rings
....:     print("k = {}".format(k))
....:     list_q = []
....:     for q in range(k*(k-1)+1, 2000, k*(k-1)):
....:         if is_prime_power(q):
....:             K = GF(q,'a')
....:             if radical_difference_family(K, k, existence=True) is True:
....:                 list_q.append(q)
....:                 _ = radical_difference_family(K,k)
....:     print(" ".join(str(p) for p in list_q))
k = 5
41 61 81 241 281 401 421 601 641 661 701 761 821 881 1181 1201 1301 1321
1361 1381 1481 1601 1681 1801 1901
k = 6
181 211 241 631 691 1531 1831 1861
k = 7
337 421 463 883 1723
k = 8
449 1009
k = 9
73 1153 1873
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import radical_difference_family

>>> radical_difference_family(GF(Integer(73)), Integer(9))                      ␣
↪              # needs sage.rings.finite_rings
[[1, 2, 4, 8, 16, 32, 37, 55, 64]]

>>> radical_difference_family(GF(Integer(281)), Integer(5))                     ␣
↪              # needs sage.rings.finite_rings
[[1, 86, 90, 153, 232],
 [4, 50, 63, 79, 85],
 [5, 36, 149, 169, 203],
 [7, 40, 68, 219, 228],
 [9, 121, 212, 248, 253],
 [29, 81, 222, 246, 265],
 [31, 137, 167, 247, 261],
 [32, 70, 118, 119, 223],
 [39, 56, 66, 138, 263],
 [43, 45, 116, 141, 217],
 [98, 101, 109, 256, 279],
 [106, 124, 145, 201, 267],
```

(continues on next page)

(continued from previous page)

```
 [111, 123, 155, 181, 273],
 [156, 209, 224, 264, 271]]

>>> for k in range(Integer(5),Integer(10)):                                    ␣
↪              # needs sage.rings.finite_rings
...     print("k = {}".format(k))
...     list_q = []
...     for q in range(k*(k-Integer(1))+Integer(1), Integer(2000), k*(k-
↪Integer(1))):
...         if is_prime_power(q):
...             K = GF(q,'a')
...             if radical_difference_family(K, k, existence=True) is True:
...                 list_q.append(q)
...                 _ = radical_difference_family(K,k)
...     print(" ".join(str(p) for p in list_q))
k = 5
41 61 81 241 281 401 421 601 641 661 701 761 821 881 1181 1201 1301 1321
1361 1381 1481 1601 1681 1801 1901
k = 6
181 211 241 631 691 1531 1831 1861
k = 7
337 421 463 883 1723
k = 8
449 1009
k = 9
73 1153 1873
```

sage.combinat.designs.difference_family.**radical_difference_set**(*K*, *k*, *l=1*, *existence=False*, *check=True*)

Return a difference set made of a cyclotomic coset in the finite field `K` and with parameters `k` and `l`.

Most of these difference sets appear in chapter VI.18.48 of the Handbook of combinatorial designs.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import radical_difference_set

sage: D = radical_difference_set(GF(7), 3, 1); D                              #␣
↪needs sage.rings.finite_rings
[[1, 2, 4]]
sage: sorted(x-y for x in D[0] for y in D[0] if x != y)                     #␣
↪needs sage.rings.finite_rings
[1, 2, 3, 4, 5, 6]

sage: D = radical_difference_set(GF(16,'a'), 6, 2)                          #␣
↪needs sage.rings.finite_rings
sage: sorted(x-y for x in D[0] for y in D[0] if x != y)                     #␣
↪needs sage.rings.finite_rings
[1,
 1,
 a,
 a,
 a + 1,
```

(continues on next page)

(continued from previous page)

```
 a + 1,
 a^2,
 a^2,
 ...
 a^3 + a^2 + a + 1,
 a^3 + a^2 + a + 1]

sage: for k in range(2,50):                                                    #␣
↪needs sage.rings.finite_rings
....:     for l in reversed(divisors(k*(k-1))):
....:         v = k*(k-1)//l + 1
....:         if is_prime_power(v) and radical_difference_set(GF(v,'a'),k,l,
↪existence=True) is True:
....:             _ = radical_difference_set(GF(v,'a'),k,l)
....:             print("{:3} {:3} {:3}".format(v,k,l))
  3   2   1
  4   3   2
  7   3   1
  5   4   3
  7   4   2
 13   4   1
 11   5   2
  7   6   5
 11   6   3
 16   6   2
  8   7   6
  9   8   7
 19   9   4
 37   9   2
 73   9   1
 11  10   9
 19  10   5
 23  11   5
 13  12  11
 23  12   6
 27  13   6
 27  14   7
 16  15  14
 31  15   7
...
 41  40  39
 79  40  20
 83  41  20
 43  42  41
 83  42  21
 47  46  45
 49  48  47
197  49  12
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import radical_difference_set
```

(continues on next page)

(continued from previous page)

```
>>> D = radical_difference_set(GF(Integer(7)), Integer(3), Integer(1)); D
↪                        # needs sage.rings.finite_rings
[[1, 2, 4]]
>>> sorted(x-y for x in D[Integer(0)] for y in D[Integer(0)] if x != y)
↪               # needs sage.rings.finite_rings
[1, 2, 3, 4, 5, 6]

>>> D = radical_difference_set(GF(Integer(16),'a'), Integer(6), Integer(2))
↪                        # needs sage.rings.finite_rings
>>> sorted(x-y for x in D[Integer(0)] for y in D[Integer(0)] if x != y)
↪               # needs sage.rings.finite_rings
[1,
 1,
 a,
 a,
 a + 1,
 a + 1,
 a^2,
 a^2,
 ...
 a^3 + a^2 + a + 1,
 a^3 + a^2 + a + 1]

>>> for k in range(Integer(2),Integer(50)):
↪               # needs sage.rings.finite_rings
...     for l in reversed(divisors(k*(k-Integer(1)))):
...         v = k*(k-Integer(1))//l + Integer(1)
...         if is_prime_power(v) and radical_difference_set(GF(v,'a'),k,l,
↪existence=True) is True:
...             _ = radical_difference_set(GF(v,'a'),k,l)
...             print("{:3} {:3} {:3}".format(v,k,l))
  3   2   1
  4   3   2
  7   3   1
  5   4   3
  7   4   2
 13   4   1
 11   5   2
  7   6   5
 11   6   3
 16   6   2
  8   7   6
  9   8   7
 19   9   4
 37   9   2
 73   9   1
 11  10   9
 19  10   5
 23  11   5
 13  12  11
 23  12   6
 27  13   6
```

(continues on next page)

(continued from previous page)

```
 27  14   7
 16  15  14
 31  15   7
...
 41  40  39
 79  40  20
 83  41  20
 43  42  41
 83  42  21
 47  46  45
 49  48  47
197  49  12
```

sage.combinat.designs.difference_family.**relative_difference_set_from_homomorphism**(*q*, *N*, *d*, *check=True*, *return_group=False*)

Construct $R((q^N-1)/(q-1), n, q^{N-1}, q^{N-2}d)$ where $nd = q-1$.

Given a prime power q, a number $N \geq 2$ and integers d such that $d|q-1$ we create the relative difference set using the construction from Corollary 5.1.1 of [EB1966].

INPUT:

- `q` – a prime power
- `N` – integer greater than 1
- `d` – integer which divides $q-1$
- `check` – boolean (default: `True`); if `True`, check that the result is a relative difference set before returning it
- `return_group` – boolean (default: `False`); if `True`, the function will also return the group from which the set is created

OUTPUT:

If `return_group=False`, the function return only the relative difference set. Otherwise, it returns a tuple containing the group and the set.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import relative_difference_set_
↪from_homomorphism
sage: relative_difference_set_from_homomorphism(7, 2, 3)        # random        #␣
↪needs sage.modules sage.rings.finite_rings
[(0), (3), (4), (2), (13), (7), (14)]
sage: relative_difference_set_from_homomorphism(9, 2, 4,        # random        #␣
↪needs sage.modules sage.rings.finite_rings
....:                                           check=False, return_group=True)
(Additive abelian group isomorphic to Z/80,
 [(0), (4), (6), (13), (7), (12), (15), (8), (9)])
sage: relative_difference_set_from_homomorphism(9, 2, 5)                        #␣
↪needs sage.modules sage.rings.finite_rings
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: q-1 must be a multiple of d
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import relative_difference_set_
↪from_homomorphism
>>> relative_difference_set_from_homomorphism(Integer(7), Integer(2), Integer(3))␣
↪       # random         # needs sage.modules sage.rings.finite_rings
[(0), (3), (4), (2), (13), (7), (14)]
>>> relative_difference_set_from_homomorphism(Integer(9), Integer(2), Integer(4),␣
↪       # random         # needs sage.modules sage.rings.finite_rings
...                                           check=False, return_group=True)
(Additive abelian group isomorphic to Z/80,
 [(0), (4), (6), (13), (7), (12), (15), (8), (9)])
>>> relative_difference_set_from_homomorphism(Integer(9), Integer(2), Integer(5))␣
↪                        # needs sage.modules sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: q-1 must be a multiple of d
```

sage.combinat.designs.difference_family.**relative_difference_set_from_m_sequence**(*q*, *N*, *check=True*, *return_group=False*)

Construct $R((q^N-1)/(q-1), q-1, q^{N-1}, q^{N-2})$ where `q` is a prime power and $N\geq 2$.

The relative difference set is constructed over the set of additive integers modulo q^N-1, as described in Theorem 5.1 of [EB1966]. Given an m-sequence (a_i) of period q^N-1, the set is: $R=\{i|0\leq i\leq q^{N-1}, a_i=1\}$.

INPUT:

- `q` – a prime power
- `N` – a nonnegative number
- `check` – boolean (default: `True`); if `True`, check that the result is a relative difference set before returning it
- `return_group` – boolean (default: `False`); if `True`, the function will also return the group from which the set is created

OUTPUT:

If `return_group=False`, the function return only the relative difference set. Otherwise, it returns a tuple containing the group and the set.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import relative_difference_set_
↪from_m_sequence
sage: relative_difference_set_from_m_sequence(2, 4,              # random      #␣
↪needs sage.modules sage.rings.finite_rings
....:                                         return_group=True)
(Additive abelian group isomorphic to Z/15,
 [(0), (4), (5), (6), (7), (9), (11), (12)])
sage: relative_difference_set_from_m_sequence(8, 2, check=False)  # random      #␣
↪needs sage.modules sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
[(0), (6), (30), (40), (41), (44), (56), (61)]
sage: relative_difference_set_from_m_sequence(6, 2)                          #␣
↪needs sage.modules
Traceback (most recent call last):
...
ValueError: q must be a prime power
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import relative_difference_set_
↪from_m_sequence
>>> relative_difference_set_from_m_sequence(Integer(2), Integer(4),
↪# random      # needs sage.modules sage.rings.finite_rings
...                                          return_group=True)
(Additive abelian group isomorphic to Z/15,
 [(0), (4), (5), (6), (7), (9), (11), (12)])
>>> relative_difference_set_from_m_sequence(Integer(8), Integer(2), check=False)
↪# random      # needs sage.modules sage.rings.finite_rings
[(0), (6), (30), (40), (41), (44), (56), (61)]
>>> relative_difference_set_from_m_sequence(Integer(6), Integer(2))          ␣
↪              # needs sage.modules
Traceback (most recent call last):
...
ValueError: q must be a prime power
```

sage.combinat.designs.difference_family.**singer_difference_set**(q, d)

Return a difference set associated to the set of hyperplanes in a projective space of dimension d over $GF(q)$.

A Singer difference set has parameters:

$$v = \frac{q^{d+1}-1}{q-1}, \quad k = \frac{q^d-1}{q-1}, \quad \lambda = \frac{q^{d-1}-1}{q-1}.$$

The idea of the construction is as follows. One consider the finite field $GF(q^{d+1})$ as a vector space of dimension $d+1$ over $GF(q)$. The set of $GF(q)$-lines in $GF(q^{d+1})$ is a projective plane and its set of hyperplanes form a balanced incomplete block design.

Now, considering a multiplicative generator z of $GF(q^{d+1})$, we get a transitive action of a cyclic group on our projective plane from which it is possible to build a difference set.

The construction is given in details in [Stinson2004], section 3.3.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import singer_difference_set,␣
↪is_difference_family
sage: G,D = singer_difference_set(3,2)                                       #␣
↪needs sage.rings.finite_rings
sage: is_difference_family(G, D, verbose=True)                               #␣
↪needs sage.rings.finite_rings
It is a (13,4,1)-difference family
True

sage: G,D = singer_difference_set(4,2)                                       #␣
↪needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
sage: is_difference_family(G, D, verbose=True)                               #␣
↪needs sage.rings.finite_rings
It is a (21,5,1)-difference family
True

sage: G,D = singer_difference_set(3,3)                                       #␣
↪needs sage.rings.finite_rings
sage: is_difference_family(G, D, verbose=True)                               #␣
↪needs sage.rings.finite_rings
It is a (40,13,4)-difference family
True

sage: G,D = singer_difference_set(9,3)                                       #␣
↪needs sage.rings.finite_rings
sage: is_difference_family(G, D, verbose=True)                               #␣
↪needs sage.rings.finite_rings
It is a (820,91,10)-difference family
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import singer_difference_set, is_
↪difference_family
>>> G,D = singer_difference_set(Integer(3),Integer(2))                         ␣
↪               # needs sage.rings.finite_rings
>>> is_difference_family(G, D, verbose=True)                                 #␣
↪needs sage.rings.finite_rings
It is a (13,4,1)-difference family
True

>>> G,D = singer_difference_set(Integer(4),Integer(2))                         ␣
↪               # needs sage.rings.finite_rings
>>> is_difference_family(G, D, verbose=True)                                 #␣
↪needs sage.rings.finite_rings
It is a (21,5,1)-difference family
True

>>> G,D = singer_difference_set(Integer(3),Integer(3))                         ␣
↪               # needs sage.rings.finite_rings
>>> is_difference_family(G, D, verbose=True)                                 #␣
↪needs sage.rings.finite_rings
It is a (40,13,4)-difference family
True

>>> G,D = singer_difference_set(Integer(9),Integer(3))                         ␣
↪               # needs sage.rings.finite_rings
>>> is_difference_family(G, D, verbose=True)                                 #␣
↪needs sage.rings.finite_rings
It is a (820,91,10)-difference family
True
```

sage.combinat.designs.difference_family.**skew_spin_goethals_seidel_difference_family**(*n*, *existence=False*, *check=True*)

Construct skew spin type Goethals-Seidel difference family with parameters $(n; k_1, k_2, k_3, k_4; \lambda)$.

The construction is described in [Djo2024]. This function contains, for each value of n, either a full representation of S_1, S_2 together with the multiplier μ, or a subgroup H, two sets of representatives, and the multiplier.

This data is used to construct the difference family using the functions `_construct_gs_difference_family_from_full()` and `_construct_gs_difference_family_from_compact()`.

INPUT:

- `n` – integer; the parameter of the GS difference family
- `existence` – boolean (default: `False`); if `True`, only check whether the skew difference family can be constructed
- `check` – boolean (default: `True`); if `True`, check that the sets are a skew difference family before returning them; setting this parameter to `False` may speed up the computation considerably

OUTPUT:

If `existence=False`, the function returns the group G of integers modulo `n` and a list containing 4 sets, or raises an error if data for the given `n` is not available. If `existence=True`, the function returns a boolean representing whether the skew difference family can be constructed.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import skew_spin_goethals_
↪seidel_difference_family
sage: G, [S1, S2, S3, S4] = skew_spin_goethals_seidel_difference_family(61)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import skew_spin_goethals_seidel_
↪difference_family
>>> G, [S1, S2, S3, S4] = skew_spin_goethals_seidel_difference_family(Integer(61))
```

If existence is `True`, the function returns a boolean:

```
sage: skew_spin_goethals_seidel_difference_family(61, existence=True)
True
sage: skew_spin_goethals_seidel_difference_family(5, existence=True)
False
```

```
>>> from sage.all import *
>>> skew_spin_goethals_seidel_difference_family(Integer(61), existence=True)
True
>>> skew_spin_goethals_seidel_difference_family(Integer(5), existence=True)
False
```

sage.combinat.designs.difference_family.**skew_supplementary_difference_set**(*n*, *existence=False*, *check=True*, *return_group=False*)

Construct $4-\{n; n_1, n_2, n_3, n_4; \lambda\}$ supplementary difference sets, where S_1 is skew and $n_1+n_2+n_3+n_4 = n+\lambda$.

These sets are constructed from available data, as described in [Djo1994a]. The set $S_1 \subset G$ is always skew, i.e. $S_1 \cap (-S_1) = \emptyset$ and $S_1 \cup (-S_1) = G \setminus \{0\}$.

The data is taken from:

- $n = 103, 151$: [Djo1994a]
- $n = 67, 113, 127, 157, 163, 181, 241$: [Djo1992a]
- $n = 37, 43$: [Djo1992b]
- $n = 39, 49, 65, 93, 121, 129, 133, 217, 219, 267$: [Djo1992c]
- $n = 97$: [Djo2008a]
- $n = 109, 145, 247$: [Djo2008b]
- $n = 73$: [Djo2023b]
- $n = 213, 631$: [DGK2014]
- $n = 331$: [DK2016]

Additional skew Supplementary difference sets are built using the function *skew_supplementary_difference_set_over_polynomial_ring()*, and *skew_supplementary_difference_set_with_paley_todd()*.

INPUT:

- `n` – integer; the parameter of the supplementary difference set
- `existence` – boolean (default: `False`); if `True`, only check whether the supplementary difference sets can be constructed
- `check` – boolean (default: `True`); if `True`, check that the sets are supplementary difference sets with S_1 skew before returning them; setting this parameter to `False` may speed up the computation considerably
- `return_group` – boolean (default: `False`); if `True`, the function will also return the group from which the sets are created

OUTPUT:

If `existence=False`, the function returns a list containing 4 sets, or raises an error if data for the given `n` is not available. If `return_group=True` the function will additionally return the group from which the sets are created. If `existence=True`, the function returns a boolean representing whether skew supplementary difference sets can be constructed.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import skew_supplementary_
→difference_set
sage: [S1, S2, S3, S4] = skew_supplementary_difference_set(39)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import skew_supplementary_
→difference_set
>>> [S1, S2, S3, S4] = skew_supplementary_difference_set(Integer(39))
```

If `return_group=True`, the function will also return the group:

```
sage: G, [S1, S2, S3, S4] = skew_supplementary_difference_set(103, return_
→group=True)
```

```
>>> from sage.all import *
>>> G, [S1, S2, S3, S4] = skew_supplementary_difference_set(Integer(103), return_
↪group=True)
```

If existence=True, the function returns a boolean:

```
sage: skew_supplementary_difference_set(103, existence=True)
True
sage: skew_supplementary_difference_set(17, existence=True)
False
```

```
>>> from sage.all import *
>>> skew_supplementary_difference_set(Integer(103), existence=True)
True
>>> skew_supplementary_difference_set(Integer(17), existence=True)
False
```

Note

The data for $n = 247$ in [Djo2008b] contains a typo: the set α_2 should contain 223 instead of 233. This can be verified by checking the resulting sets, which are given explicitly in the paper.

sage.combinat.designs.difference_family.**skew_supplementary_difference_set_over_polynomial_ring**(*n*, *existence=False*, *check=True*)

Construct skew supplementary difference sets over a polynomial ring of order n.

The skew supplementary difference sets for $n = 81, 169$ are taken from [Djo1994a].

INPUT:

- n – integer; the parameter of the supplementary difference sets
- existence – boolean (default: False); if True, only check whether the supplementary difference sets can be constructed
- check – boolean (default: True); if True, check that the sets are supplementary difference sets with S_1 skew before returning them; setting this parameter to False may speed up the computation considerably

OUTPUT:

If existence=False, the function returns a Polynomial Ring of order n and a list containing 4 sets, or raises an error if data for the given n is not available. If existence=True, the function returns a boolean representing whether skew supplementary difference sets can be constructed.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import skew_supplementary_
↪difference_set_over_polynomial_ring
sage: G, [S1, S2, S3, S4] = skew_supplementary_difference_set_over_polynomial_
↪ring(81)      # needs sage.libs.pari
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import skew_supplementary_
↪difference_set_over_polynomial_ring
>>> G, [S1, S2, S3, S4] = skew_supplementary_difference_set_over_polynomial_
↪ring(Integer(81))       # needs sage.libs.pari
```

If existence=True, the function returns a boolean:

```
sage: skew_supplementary_difference_set_over_polynomial_ring(81, existence=True)
True
sage: skew_supplementary_difference_set_over_polynomial_ring(17, existence=True)
False
```

```
>>> from sage.all import *
>>> skew_supplementary_difference_set_over_polynomial_ring(Integer(81),␣
↪existence=True)
True
>>> skew_supplementary_difference_set_over_polynomial_ring(Integer(17),␣
↪existence=True)
False
```

sage.combinat.designs.difference_family.**skew_supplementary_difference_set_with_paley_todd**(*n*, *existence=False*, *check=True*)

Construct $4-\{n;n_1,n_2,n_3,n_4;\lambda\}$ skew supplementary difference sets where S_1 is the Paley-Todd difference set.

The skew SDS returned have the property that $n_1+n_2+n_3+n_4=n+\lambda$.

This construction is described in [DK2016]. The function contains, for each value of n, a set H containing integers modulo n, and four sets J,K,L. Then, these are used to construct $(n;k_2,k_3,k_4;\lambda_2)$ difference family, with $\lambda_2=k_2+k_3+k_4+(3n-1)/4$. Finally, these sets together with the Paley-Todd difference set form a skew supplementary difference set.

INPUT:

- n – integer; the parameter of the supplementary difference set
- existence – boolean (default: False); if True, only check whether the supplementary difference sets can be constructed
- check – boolean (default: True); if True, check that the sets are supplementary difference sets with S_1 skew before returning them; setting this parameter to False may speed up the computation considerably

OUTPUT:

If existence=False, the function returns the group G of integers modulo n and a list containing 4 sets, or raises an error if data for the given n is not available. If existence=True, the function returns a boolean representing whether skew supplementary difference sets can be constructed.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import skew_supplementary_
↪difference_set_with_paley_todd
sage: G, [S1, S2, S3, S4] = skew_supplementary_difference_set_with_paley_todd(239)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import skew_supplementary_
↪difference_set_with_paley_todd
>>> G, [S1, S2, S3, S4] = skew_supplementary_difference_set_with_paley_
↪todd(Integer(239))
```

If existence is `True`, the function returns a boolean:

```
sage: skew_supplementary_difference_set_with_paley_todd(239, existence=True)
True
sage: skew_supplementary_difference_set_with_paley_todd(17, existence=True)
False
```

```
>>> from sage.all import *
>>> skew_supplementary_difference_set_with_paley_todd(Integer(239),␣
↪existence=True)
True
>>> skew_supplementary_difference_set_with_paley_todd(Integer(17), existence=True)
False
```

`sage.combinat.designs.difference_family.`**`spin_goethals_seidel_difference_family`**`(`*n*, *existence=False*, *check=True*`)`

Construct a spin type Goethals-Seidel difference family with parameters $(n; k_1, k_2, k_3, k_4; \lambda)$.

The construction is described in [Djo2024]. This function contains, for each value of n, either a full representation of S_1, S_2 together with the multiplier μ, or a subgroup H, two sets of representatives, and the multiplier. This data is used to construct the difference family using the functions `_construct_gs_difference_family_from_full()` and `_construct_gs_difference_family_from_compact()`.

Additionally, this function also checks if a (skew) difference family can be constructed using *`skew_spin_goethals_seidel_difference_family()`*.

INPUT:

- `n` – integer; the parameter of the GS difference family
- `existence` – boolean (default: `False`); if `True`, only check whether the difference family can be constructed
- `check` – boolean (default: `True`); if `True`, check that the sets are a difference family before returning them; setting this parameter to `False` may speed up the computation considerably

OUTPUT:

If `existence=False`, the function returns the group G of integers modulo `n` and a list containing 4 sets, or raises an error if data for the given `n` is not available. If `existence=True`, the function returns a boolean representing whether the difference family can be constructed.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import spin_goethals_seidel_
↪difference_family
sage: G, [S1, S2, S3, S4] = spin_goethals_seidel_difference_family(73)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import spin_goethals_seidel_
```

(continues on next page)

(continued from previous page)

```
↪difference_family
>>> G, [S1, S2, S3, S4] = spin_goethals_seidel_difference_family(Integer(73))
```

If existence is `True`, the function returns a boolean:

```
sage: spin_goethals_seidel_difference_family(73, existence=True)
True
sage: spin_goethals_seidel_difference_family(5, existence=True)
False
```

```
>>> from sage.all import *
>>> spin_goethals_seidel_difference_family(Integer(73), existence=True)
True
>>> spin_goethals_seidel_difference_family(Integer(5), existence=True)
False
```

`sage.combinat.designs.difference_family.`**`supplementary_difference_set`**(*q*, *existence=False*, *check=True*)

Construct $4-\{2v; v, v+1, v, v; 2v\}$ supplementary difference sets where $q = 2v + 1$.

This is a deprecated version of *`supplementary_difference_set_from_rel_diff_set()`*, please use that instead.

`sage.combinat.designs.difference_family.`**`supplementary_difference_set_from_rel_diff_set`**(*q*, *existence=False*, *check=True*)

Construct $4-\{2v; v, v+1, v, v; 2v\}$ supplementary difference sets where $q = 2v + 1$.

The sets are created from relative difference sets as detailed in Theorem 3.3 of [Spe1975]. this construction requires that q is an odd prime power and that there exists $s \geq 0$ such that $(q-(2^{s+1}+1))/2^{s+1}$ is an odd prime power.

Note that the construction from [Spe1975] states that the resulting sets are $4-\{2v; v+1, v, v, v; 2v\}$ supplementary difference sets. However, the implementation of that construction returns $4-\{2v; v, v+1, v, v; 2v\}$ supplementary difference sets. This is not important, since the supplementary difference sets are not ordered.

INPUT:

- `q` – an odd prime power
- `existence` – boolean (default: `False`); if `True`, only check whether the supplementary difference sets can be constructed
- `check` – boolean (default: `True`); if `True`, check that the sets are supplementary difference sets before returning them

OUTPUT:

If `existence=False`, the function returns the 4 sets (containing integers), or raises an error if `q` does not satisfy the constraints. If `existence=True`, the function returns a boolean representing whether supplementary difference sets can be constructed.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import supplementary_
↪difference_set_from_rel_diff_set
```

(continues on next page)

(continued from previous page)

```
sage: supplementary_difference_set_from_rel_diff_set(17)   #random                    #␣
↪needs sage.libs.pari
(Additive abelian group isomorphic to Z/16,
 [[(1), (5), (6), (7), (9), (13), (14), (15)],
  [(0), (2), (3), (5), (6), (10), (11), (13), (14)],
  [(0), (1), (2), (3), (5), (6), (7), (12)],
  [(0), (2), (3), (5), (6), (7), (9), (12)]])
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import supplementary_difference_
↪set_from_rel_diff_set
>>> supplementary_difference_set_from_rel_diff_set(Integer(17))   #random              ␣
↪     # needs sage.libs.pari
(Additive abelian group isomorphic to Z/16,
 [[(1), (5), (6), (7), (9), (13), (14), (15)],
  [(0), (2), (3), (5), (6), (10), (11), (13), (14)],
  [(0), (1), (2), (3), (5), (6), (7), (12)],
  [(0), (2), (3), (5), (6), (7), (9), (12)]])
```

If `existence=True`, the function returns a boolean:

```
sage: supplementary_difference_set_from_rel_diff_set(7, existence=True)
False
sage: supplementary_difference_set_from_rel_diff_set(17, existence=True)
True
```

```
>>> from sage.all import *
>>> supplementary_difference_set_from_rel_diff_set(Integer(7), existence=True)
False
>>> supplementary_difference_set_from_rel_diff_set(Integer(17), existence=True)
True
```

See also

is_supplementary_difference_set()

`sage.combinat.designs.difference_family.`**`supplementary_difference_set_hadamard`**(*n*, *existence=False*, *check=True*)

Construct $4-\{n; n_1, n_2, n_3, n_4; \lambda\}$ supplementary difference sets, where $n_1 + n_2 + n_3 + n_4 = n + \lambda$.

These sets are constructed from available data, as described in [Djo1994a]. The data is taken from:

- $n = 191$: [Djo2008c]
- $n = 239$: [Djo1994b]
- $n = 251$: [DGK2014]

Additional SDS are constructed using *skew_supplementary_difference_set()*.

INPUT:

- `n` – integer; the parameter of the supplementary difference set

- `existence` – boolean (default: `False`); if `True`, only check whether the supplementary difference sets can be constructed
- `check` – boolean (default: `True`); if `True`, check that the sets are supplementary difference sets before returning them; Setting this parameter to `False` may speed up the computation considerably

OUTPUT:

If `existence=False`, the function returns the ring of integers modulo `n` and a list containing the 4 sets, or raises an error if data for the given `n` is not available. If `existence=True`, the function returns a boolean representing whether skew supplementary difference sets can be constructed.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import supplementary_
↪difference_set_hadamard
sage: G, [S1, S2, S3, S4] = supplementary_difference_set_hadamard(191)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import supplementary_difference_
↪set_hadamard
>>> G, [S1, S2, S3, S4] = supplementary_difference_set_hadamard(Integer(191))
```

If `existence=True`, the function returns a boolean:

```
sage: supplementary_difference_set_hadamard(191, existence=True)
True
sage: supplementary_difference_set_hadamard(17, existence=True)
False
```

```
>>> from sage.all import *
>>> supplementary_difference_set_hadamard(Integer(191), existence=True)
True
>>> supplementary_difference_set_hadamard(Integer(17), existence=True)
False
```

`sage.combinat.designs.difference_family.`**`turyn_1965_3x3xK`**`(k=4)`

Return a difference set in either $C_3 \times C_3 \times C_4$ or $C_3 \times C_3 \times C_2 \times C_2$ with parameters $v = 36$, $k = 15$, $\lambda = 6$.

This example appears in [Tu1965].

INPUT:

- `k` – either `2` (to get a difference set in $C_3 \times C_3 \times C_2 \times C_2$) or `4` (to get a difference set in $C_3 \times C_3 \times C_3 \times C_4$)

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import turyn_1965_3x3xK
sage: from sage.combinat.designs.difference_family import is_difference_family
sage: G,D = turyn_1965_3x3xK(4)
sage: assert is_difference_family(G, D, 36, 15, 6)
sage: G,D = turyn_1965_3x3xK(2)
sage: assert is_difference_family(G, D, 36, 15, 6)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import turyn_1965_3x3xK
>>> from sage.combinat.designs.difference_family import is_difference_family
```

(continues on next page)

(continued from previous page)

```
>>> G,D = turyn_1965_3x3xK(Integer(4))
>>> assert is_difference_family(G, D, Integer(36), Integer(15), Integer(6))
>>> G,D = turyn_1965_3x3xK(Integer(2))
>>> assert is_difference_family(G, D, Integer(36), Integer(15), Integer(6))
```

sage.combinat.designs.difference_family.**twin_prime_powers_difference_set**(*p*, *check=True*)

Return a difference set on $GF(p) \times GF(p+2)$.

The difference set is built from the following element of the Cartesian product of finite fields $GF(p) \times GF(p+2)$:

- $(x, 0)$ with any x
- (x, y) with x and y squares
- (x, y) with x and y non-squares

For more information see Wikipedia article Difference_set.

INPUT:

- `check` – boolean (default: `True`); if `True`, then the result of the computation is checked before being returned. This should not be needed but ensures that the output is correct

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import twin_prime_powers_
↪difference_set
sage: G, D = twin_prime_powers_difference_set(3)
sage: G
The Cartesian product of (Finite Field of size 3, Finite Field of size 5)
sage: D
[[(1, 1), (1, 4), (2, 2), (2, 3), (0, 0), (1, 0), (2, 0)]]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_family import twin_prime_powers_
↪difference_set
>>> G, D = twin_prime_powers_difference_set(Integer(3))
>>> G
The Cartesian product of (Finite Field of size 3, Finite Field of size 5)
>>> D
[[(1, 1), (1, 4), (2, 2), (2, 3), (0, 0), (1, 0), (2, 0)]]
```

5.1.85 Difference Matrices

This module gathers code related to difference matrices. One can build those objects (or know if they can be built) with *difference_matrix()*:

```
sage: G,DM = designs.difference_matrix(9,5,1)
```

```
>>> from sage.all import *
>>> G,DM = designs.difference_matrix(Integer(9),Integer(5),Integer(1))
```

Functions

sage.combinat.designs.difference_matrices.**difference_matrix**(*g*, *k*, *lmbda=1*, *existence=False*, *check=True*)

Return a (g, k, λ)-difference matrix.

A matrix M is a (g, k, λ)-difference matrix if it has size $\lambda g \times k$, its entries belong to the group G of cardinality g, and for any two rows R, R' of M and $x \in G$ there are exactly λ values i such that $R_i - R'_i = x$.

INPUT:

- `k` – integer; number of columns. If `k` is `None` it is set to the largest value available
- `g` – integer; cardinality of the group G
- `lmbda` – integer (default: 1); number of times each element of G appears as a difference
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- `existence` – boolean; instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`)
  - `False` – meaning that the design does not exist

Note

When `k=None` and `existence=True` the function returns an integer, i.e. the largest k such that we can build a (g, k, λ)-DM.

EXAMPLES:

```
sage: G,M = designs.difference_matrix(25,10); G
Finite Field in x of size 5^2
sage: designs.difference_matrix(993,None,existence=1)
32
```

```
>>> from sage.all import *
>>> G,M = designs.difference_matrix(Integer(25),Integer(10)); G
Finite Field in x of size 5^2
>>> designs.difference_matrix(Integer(993),None,existence=Integer(1))
32
```

Here we print for each g the maximum possible k for which Sage knows how to build a $(g, k, 1)$-difference matrix:

```
sage: for g in range(2,30):
....:     k_max = designs.difference_matrix(g=g,k=None,existence=True)
....:     print("{:2} {}".format(g, k_max))
....:     _ = designs.difference_matrix(g,k_max)
 2 2
 3 3
 4 4
 5 5
```

(continues on next page)

(continued from previous page)

```
 6 2
 7 7
 8 8
 9 9
10 2
11 11
12 6
13 13
14 2
15 3
16 16
17 17
18 2
19 19
20 4
21 6
22 2
23 23
24 8
25 25
26 2
27 27
28 6
29 29
```

```
>>> from sage.all import *
>>> for g in range(Integer(2),Integer(30)):
...     k_max = designs.difference_matrix(g=g,k=None,existence=True)
...     print("{:2} {}".format(g, k_max))
...     _ = designs.difference_matrix(g,k_max)
 2 2
 3 3
 4 4
 5 5
 6 2
 7 7
 8 8
 9 9
10 2
11 11
12 6
13 13
14 2
15 3
16 16
17 17
18 2
19 19
20 4
21 6
22 2
23 23
```

(continues on next page)

(continued from previous page)

```
24 8
25 25
26 2
27 27
28 6
29 29
```

sage.combinat.designs.difference_matrices.**difference_matrix_product**(*k*, *M1*, *G1*, *lmbda1*, *M2*, *G2*, *lmbda2*, *check=True*)

Return the product of the `(G1, k, lmbda1)` and `(G2, k, lmbda2)` difference matrices `M1` and `M2`.

The result is a $(G1 \times G2, k, \lambda_1 \lambda_2)$-difference matrix.

INPUT:

- `k`, `lmbda1`, `lmbda2` – positive integers
- `G1`, `G2` – groups
- `M1`, `M2` – `(G1, k, lmbda1)` and `(G, k, lmbda2)` difference matrices
- `check` – boolean (default: `True`); whether to check the output before it is returned

EXAMPLES:

```
sage: from sage.combinat.designs.difference_matrices import (
....:     difference_matrix_product,
....:     is_difference_matrix)
sage: G1,M1 = designs.difference_matrix(11,6)
sage: G2,M2 = designs.difference_matrix(7,6)
sage: G,M = difference_matrix_product(6,M1,G1,1,M2,G2,1)
sage: G1
Finite Field of size 11
sage: G2
Finite Field of size 7
sage: G
The Cartesian product of (Finite Field of size 11, Finite Field of size 7)
sage: is_difference_matrix(M,G,6,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_matrices import (
...     difference_matrix_product,
...     is_difference_matrix)
>>> G1,M1 = designs.difference_matrix(Integer(11),Integer(6))
>>> G2,M2 = designs.difference_matrix(Integer(7),Integer(6))
>>> G,M = difference_matrix_product(Integer(6),M1,G1,Integer(1),M2,G2,Integer(1))
>>> G1
Finite Field of size 11
>>> G2
Finite Field of size 7
>>> G
The Cartesian product of (Finite Field of size 11, Finite Field of size 7)
```

(continues on next page)

(continued from previous page)

```
>>> is_difference_matrix(M,G,Integer(6),Integer(1))
True
```

sage.combinat.designs.difference_matrices.**find_product_decomposition**(*k*, *lmbda=1*)

Try to find a product decomposition construction for difference matrices.

INPUT:

- g, k, lmbda – integers, parameters of the difference matrix

OUTPUT:

A pair of pairs (g1,lmbda),(g2,lmbda2) if Sage knows how to build $(g1, k, lmbda1)$ and $(g2, k, lmbda2)$ difference matrices and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_matrices import find_product_
↪decomposition
sage: find_product_decomposition(77,6)
((7, 1), (11, 1))
sage: find_product_decomposition(616,7)
((7, 1), (88, 1))
sage: find_product_decomposition(24,10)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.difference_matrices import find_product_
↪decomposition
>>> find_product_decomposition(Integer(77),Integer(6))
((7, 1), (11, 1))
>>> find_product_decomposition(Integer(616),Integer(7))
((7, 1), (88, 1))
>>> find_product_decomposition(Integer(24),Integer(10))
False
```

5.1.86 Evenly distributed sets in finite fields

This module consists of a simple class *EvenlyDistributedSetsBacktracker*. Its main purpose is to iterate through the evenly distributed sets of a given finite field.

The naive backtracker implemented here is not directly used to generate difference family as even for small parameters it already takes time to run. Instead, its output has been stored into a database *sage.combinat.designs.database*. If the backtracker is improved, then one might want to update this database with more values.

Classes and methods

class sage.combinat.designs.evenly_distributed_sets.**EvenlyDistributedSetsBacktracker**

Bases: object

Set of evenly distributed subsets in finite fields.

> **Definition:** Let K be a finite field of cardinality q and k an integer so that $k(k-1)$ divides $q-1$. Let $H = K^*$ be the multiplicative group of invertible elements in K. A *k-evenly distributed set* in K is a set $B = \{b_1, b_2, \ldots, b_k\}$ of k elements of K so that the $k(k-1)$ differences $\Delta B = \{b_i - b_j; i \neq j\}$ hit each coset modulo $H^{2(q-1)/(k(k-1))}$ exactly twice.

Evenly distributed sets were introduced by Wilson [Wi72] (see also [BJL99-1], Chapter VII.6). He proved that for any k, and for any prime power q large enough such that $k(k-1)$ divides k there exists a k-evenly distributed set in the field of cardinality q. This existence result based on a counting argument (using Dirichlet theorem) does not provide a simple method to generate them.

From a k-evenly distributed set, it is straightforward to build a $(q, k, 1)$-difference family (see `to_difference_family()`). Another approach to generate a difference family, somehow dual to this one, is via radical difference family (see in particular `radical_difference_family()` from the module `difference_family`).

By default, this backtracker only considers evenly distributed sets up to affine automorphisms, i.e. B is considered equivalent to $sB + t$ for any invertible element s and any element t in the field K. Note that the set of differences is just multiplicatively translated by s as $\Delta(sB + t) = s(\Delta B)$, and so that B is an evenly distributed set if and only if sB is one too. This behaviour can be modified via the argument `up_to_isomorphism` (see the input section and the examples below).

INPUT:

- `K` – a finite field of cardinality q
- `k` – positive integer such that $k(k-1)$ divides $q-1$
- `up_to_isomorphism` – boolean (default: `True`); whether only consider evenly distributed sets up to automorphisms of the field of the form $x \mapsto ax + b$. If set to `False` then the iteration is over all evenly distributed sets that contain `0` and `1`.
- `check` – boolean (default: `False`); whether you want to check intermediate steps of the iterator. This is mainly intended for debugging purpose. Setting it to `True` will considerably slow the iteration.

EXAMPLES:

The main part of the code is contained in the iterator. To get one evenly distributed set just do:

```
sage: from sage.combinat.designs.evenly_distributed_sets import␣
↪EvenlyDistributedSetsBacktracker
sage: E = EvenlyDistributedSetsBacktracker(Zmod(151),6)
sage: B = E.an_element()
sage: B
[0, 1, 69, 36, 57, 89]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.evenly_distributed_sets import␣
↪EvenlyDistributedSetsBacktracker
>>> E = EvenlyDistributedSetsBacktracker(Zmod(Integer(151)),Integer(6))
>>> B = E.an_element()
>>> B
[0, 1, 69, 36, 57, 89]
```

The class has a method to convert it to a difference family:

```
sage: E.to_difference_family(B)
[[0, 1, 69, 36, 57, 89],
 [0, 132, 48, 71, 125, 121],
 [0, 59, 145, 10, 41, 117],
 [0, 87, 114, 112, 127, 42],
 [0, 8, 99, 137, 3, 108]]
```

```
>>> from sage.all import *
>>> E.to_difference_family(B)
[[0, 1, 69, 36, 57, 89],
 [0, 132, 48, 71, 125, 121],
 [0, 59, 145, 10, 41, 117],
 [0, 87, 114, 112, 127, 42],
 [0, 8, 99, 137, 3, 108]]
```

It is also possible to run over all evenly distributed sets:

```
sage: E = EvenlyDistributedSetsBacktracker(Zmod(13), 4, up_to_isomorphism=False)
sage: for B in E: print(B)
[0, 1, 11, 5]
[0, 1, 4, 6]
[0, 1, 9, 3]
[0, 1, 8, 10]

sage: E = EvenlyDistributedSetsBacktracker(Zmod(13), 4, up_to_isomorphism=True)
sage: for B in E: print(B)
[0, 1, 11, 5]
```

```
>>> from sage.all import *
>>> E = EvenlyDistributedSetsBacktracker(Zmod(Integer(13)), Integer(4), up_to_
↪isomorphism=False)
>>> for B in E: print(B)
[0, 1, 11, 5]
[0, 1, 4, 6]
[0, 1, 9, 3]
[0, 1, 8, 10]

>>> E = EvenlyDistributedSetsBacktracker(Zmod(Integer(13)), Integer(4), up_to_
↪isomorphism=True)
>>> for B in E: print(B)
[0, 1, 11, 5]
```

Or only count them:

```
sage: for k in range(13, 200, 12):
....:     if is_prime_power(k):
....:         K = GF(k,'a')
....:         E1 = EvenlyDistributedSetsBacktracker(K, 4, False)
....:         E2 = EvenlyDistributedSetsBacktracker(K, 4, True)
....:         print("{:3} {:3} {:3}".format(k, E1.cardinality(), E2.
↪cardinality()))
 13   4   1
 25  40   4
 37  12   1
 49  24   2
 61  12   1
 73  48   4
 97  64   6
109  72   6
121 240  20
```

(continues on next page)

(continued from previous page)

```
157  96   8
169 240  20
181 204  17
193 336  28
```

```
>>> from sage.all import *
>>> for k in range(Integer(13), Integer(200), Integer(12)):
...     if is_prime_power(k):
...         K = GF(k,'a')
...         E1 = EvenlyDistributedSetsBacktracker(K, Integer(4), False)
...         E2 = EvenlyDistributedSetsBacktracker(K, Integer(4), True)
...         print("{:3} {:3} {:3}".format(k, E1.cardinality(), E2.cardinality()))
 13   4   1
 25  40   4
 37  12   1
 49  24   2
 61  12   1
 73  48   4
 97  64   6
109  72   6
121 240  20
157  96   8
169 240  20
181 204  17
193 336  28
```

Note that by definition, the number of evenly distributed sets up to isomorphisms is at most $k(k-1)$ times smaller than without isomorphisms. But it might not be exactly $k(k-1)$ as some of them might have symmetries:

```
sage: B = EvenlyDistributedSetsBacktracker(Zmod(13), 4).an_element()
sage: B
[0, 1, 11, 5]
sage: [9*x + 5 for x in B]
[5, 1, 0, 11]
sage: [3*x + 11 for x in B]
[11, 1, 5, 0]
```

```
>>> from sage.all import *
>>> B = EvenlyDistributedSetsBacktracker(Zmod(Integer(13)), Integer(4)).an_
↪element()
>>> B
[0, 1, 11, 5]
>>> [Integer(9)*x + Integer(5) for x in B]
[5, 1, 0, 11]
>>> [Integer(3)*x + Integer(11) for x in B]
[11, 1, 5, 0]
```

an_element()

Return an evenly distributed set.

If there is no such subset raise an EmptySetError.

EXAMPLES:

```
sage: from sage.combinat.designs.evenly_distributed_sets import EvenlyDistributedSetsBacktracker

sage: E = EvenlyDistributedSetsBacktracker(Zmod(41),5)
sage: E.an_element()
[0, 1, 13, 38, 31]

sage: E = EvenlyDistributedSetsBacktracker(Zmod(61),6)
sage: E.an_element()
Traceback (most recent call last):
...
EmptySetError: no 6-evenly distributed set in Ring of integers modulo 61
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.evenly_distributed_sets import EvenlyDistributedSetsBacktracker

>>> E = EvenlyDistributedSetsBacktracker(Zmod(Integer(41)),Integer(5))
>>> E.an_element()
[0, 1, 13, 38, 31]

>>> E = EvenlyDistributedSetsBacktracker(Zmod(Integer(61)),Integer(6))
>>> E.an_element()
Traceback (most recent call last):
...
EmptySetError: no 6-evenly distributed set in Ring of integers modulo 61
```

cardinality()

Return the number of evenly distributed sets.

Use with precaution as there can be a lot of such sets and this method might be very long to answer!

EXAMPLES:

```
sage: from sage.combinat.designs.evenly_distributed_sets import EvenlyDistributedSetsBacktracker

sage: E = EvenlyDistributedSetsBacktracker(GF(25,'a'), 4); E
4-evenly distributed sets (up to isomorphism)
 in Finite Field in a of size 5^2
sage: E.cardinality()
4

sage: E = EvenlyDistributedSetsBacktracker(GF(25,'a'), 4,
....:                                      up_to_isomorphism=False)
sage: E.cardinality()
40
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.evenly_distributed_sets import EvenlyDistributedSetsBacktracker

>>> E = EvenlyDistributedSetsBacktracker(GF(Integer(25),'a'), Integer(4)); E
```

(continues on next page)

(continued from previous page)

```
4-evenly distributed sets (up to isomorphism)
 in Finite Field in a of size 5^2
>>> E.cardinality()
4

>>> E = EvenlyDistributedSetsBacktracker(GF(Integer(25),'a'), Integer(4),
...                                      up_to_isomorphism=False)
>>> E.cardinality()
40
```

to_difference_family(*B*, *check=True*)

Given an evenly distributed set `B` convert it to a difference family.

As for any $x \in K^* = H$ we have $|\Delta B \cap x H^{2(q-1)/(k(k-1))}| = 2$ (see *EvenlyDistributedSetsBacktracker*), the difference family is produced as $\{xB : x \in H^{2(q-1)/(k(k-1))}\}$

This method is useful if you want to obtain the difference family from the output of the iterator.

INPUT:

- `B` – an evenly distributed set
- `check` – boolean (default: `True`); whether to check the result

EXAMPLES:

```
sage: from sage.combinat.designs.evenly_distributed_sets import␣
↪EvenlyDistributedSetsBacktracker
sage: E = EvenlyDistributedSetsBacktracker(Zmod(41),5)
sage: B = E.an_element(); B
[0, 1, 13, 38, 31]
sage: D = E.to_difference_family(B); D
[[0, 1, 13, 38, 31], [0, 32, 6, 27, 8]]

sage: from sage.combinat.designs.difference_family import is_difference_family
sage: is_difference_family(Zmod(41),D,41,5,1)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.evenly_distributed_sets import␣
↪EvenlyDistributedSetsBacktracker
>>> E = EvenlyDistributedSetsBacktracker(Zmod(Integer(41)),Integer(5))
>>> B = E.an_element(); B
[0, 1, 13, 38, 31]
>>> D = E.to_difference_family(B); D
[[0, 1, 13, 38, 31], [0, 32, 6, 27, 8]]

>>> from sage.combinat.designs.difference_family import is_difference_family
>>> is_difference_family(Zmod(Integer(41)),D,Integer(41),Integer(5),
↪Integer(1))
True
```

Setting `check` to `False` is much faster:

```
sage: timeit("df = E.to_difference_family(B, check=True)") # random
625 loops, best of 3: 117 µs per loop

sage: timeit("df = E.to_difference_family(B, check=False)")  # random
625 loops, best of 3: 1.83 µs per loop
```

```
>>> from sage.all import *
>>> timeit("df = E.to_difference_family(B, check=True)") # random
625 loops, best of 3: 117 µs per loop

>>> timeit("df = E.to_difference_family(B, check=False)")  # random
625 loops, best of 3: 1.83 µs per loop
```

5.1.87 External Representations of Block Designs

The "ext_rep" module is an API to the abstract tree represented by an XML document containing the External Representation of a list of block designs. The module also provides the related I/O operations for reading/writing ext-rep files or data. The parsing is based on expat.

This is a modified form of the module ext_rep.py (version 0.8) written by Peter Dobcsanyi [Do2009] peter@designtheory.org.

Todo

The XML data from the designtheory.org database contains a wealth of information about things like automorphism groups, transitivity, cycle type representatives, etc, but none of this data is made available through the current implementation.

Functions

class sage.combinat.designs.ext_rep.**XTree**(*node*)

Bases: object

A lazy class to wrap a rooted tree representing an XML document. The tree's nodes are tuples of the structure:

(name, {dictionary of attributes}, [list of children])

Methods and services of an XTree object t:

- t.attribute – attribute named
- t.child – first child named
- t[i] – i-th child
- for child in t: – iterate over t's children
- len(t) – number of t's children

If child is not an empty subtree, return the subtree as an XTree object. If child is an empty subtree, return _name of the subtree. Otherwise return the child itself.

The lazy tree idea originated from a utility class of the pyRXP 0.9 package by Robin Becker at ReportLab.

class sage.combinat.designs.ext_rep.**XTreeProcessor**

Bases: object

An incremental event-driven parser for ext-rep documents. The processing stages:

- `<list_of_designs ...>` opening element. call-back: `list_of_designs_proc`
- `<list_definition>` subtree. call-back: `list_definition_proc`
- `<info>` subtree. call-back: `info_proc`
- iterating over `<designs>` processing each `<block_design>` separately. call-back: `block_design_proc`
- finishing with closing `</designs>` and `</list_of_designs>`.

parse(*xml_source*)

The main parsing function. Given an XML source (either a file handle or a string), parse the entire XML source.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: proc = ext_rep.XTreeProcessor()
sage: proc.save_designs = True
sage: f = ext_rep.open_extrep_file(file_loc)
sage: proc.parse(f)
sage: f.close()
sage: os.remove(file_loc)
sage: proc.list_of_designs[0]
(2, [[0, 1], [0, 1]])
```

```
>>> from sage.all import *
>>> from sage.combinat.designs import ext_rep
>>> file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
>>> proc = ext_rep.XTreeProcessor()
>>> proc.save_designs = True
>>> f = ext_rep.open_extrep_file(file_loc)
>>> proc.parse(f)
>>> f.close()
>>> os.remove(file_loc)
>>> proc.list_of_designs[Integer(0)]
(2, [[0, 1], [0, 1]])
```

sage.combinat.designs.ext_rep.**check_dtrs_protocols**(*input_name*, *input_pv*)

Check that the XML data is in a valid format. We can currently handle version 2.0. For more information see http://designtheory.org/library/extrep/

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: ext_rep.check_dtrs_protocols('source', '2.0')
sage: ext_rep.check_dtrs_protocols('source', '3.0')
Traceback (most recent call last):
...
RuntimeError: Incompatible dtrs_protocols: program: 2.0 source: 3.0
```

```
>>> from sage.all import *
>>> from sage.combinat.designs import ext_rep
>>> ext_rep.check_dtrs_protocols('source', '2.0')
>>> ext_rep.check_dtrs_protocols('source', '3.0')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
RuntimeError: Incompatible dtrs_protocols: program: 2.0 source: 3.0
```

sage.combinat.designs.ext_rep.**designs_from_XML**(*fname*)

Return a list of designs contained in an XML file fname. The list contains tuples of the form (v, bs) where v is the number of points of the design and bs is the list of blocks.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: ext_rep.designs_from_XML(file_loc)[0]
(2, [[0, 1], [0, 1]])
sage: os.remove(file_loc)

sage: from sage.combinat.designs import ext_rep
sage: from sage.combinat.designs.block_design import BlockDesign
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: v, blocks = ext_rep.designs_from_XML(file_loc)[0]
sage: d = BlockDesign(v, blocks)
sage: d.blocks()
[[0, 1], [0, 1]]
sage: d.is_t_design(t=2)
True
sage: d.is_t_design(return_parameters=True)
(True, (2, 2, 2, 2))
```

```
>>> from sage.all import *
>>> from sage.combinat.designs import ext_rep
>>> file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
>>> ext_rep.designs_from_XML(file_loc)[Integer(0)]
(2, [[0, 1], [0, 1]])
>>> os.remove(file_loc)

>>> from sage.combinat.designs import ext_rep
>>> from sage.combinat.designs.block_design import BlockDesign
>>> file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
>>> v, blocks = ext_rep.designs_from_XML(file_loc)[Integer(0)]
>>> d = BlockDesign(v, blocks)
>>> d.blocks()
[[0, 1], [0, 1]]
>>> d.is_t_design(t=Integer(2))
True
>>> d.is_t_design(return_parameters=True)
(True, (2, 2, 2, 2))
```

sage.combinat.designs.ext_rep.**designs_from_XML_url**(*url*)

Return a list of designs contained in an XML file named by a URL. The list contains tuples of the form (v, bs) where v is the number of points of the design and bs is the list of blocks.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: ext_rep.designs_from_XML_url("file://" + file_loc)[0]
(2, [[0, 1], [0, 1]])
sage: os.remove(file_loc)

sage: from sage.combinat.designs import ext_rep
sage: ext_rep.designs_from_XML_url("http://designtheory.org/database/v-b-k/v3-b6-
↪k2.icgsa.txt.bz2") # optional - internet
[(3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 2], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 2], [1, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 2], [0, 2], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 2], [0, 2], [1, 2]]),
 (3, [[0, 1], [0, 1], [0, 2], [0, 2], [1, 2], [1, 2]])]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs import ext_rep
>>> file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
>>> ext_rep.designs_from_XML_url("file://" + file_loc)[Integer(0)]
(2, [[0, 1], [0, 1]])
>>> os.remove(file_loc)

>>> from sage.combinat.designs import ext_rep
>>> ext_rep.designs_from_XML_url("http://designtheory.org/database/v-b-k/v3-b6-k2.
↪icgsa.txt.bz2") # optional - internet
[(3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 2], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 2], [1, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 2], [0, 2], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 2], [0, 2], [1, 2]]),
 (3, [[0, 1], [0, 1], [0, 2], [0, 2], [1, 2], [1, 2]])]
```

sage.combinat.designs.ext_rep.**dump_to_tmpfile**(*s*)

Utility function to dump a string to a temporary file.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile("boo")
sage: os.remove(file_loc)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs import ext_rep
>>> file_loc = ext_rep.dump_to_tmpfile("boo")
>>> os.remove(file_loc)
```

sage.combinat.designs.ext_rep.**open_extrep_file**(*fname*)

Try to guess the compression type from extension and open the extrep file.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: proc = ext_rep.XTreeProcessor()
sage: f = ext_rep.open_extrep_file(file_loc)
sage: proc.parse(f)
sage: f.close()
sage: os.remove(file_loc)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs import ext_rep
>>> file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
>>> proc = ext_rep.XTreeProcessor()
>>> f = ext_rep.open_extrep_file(file_loc)
>>> proc.parse(f)
>>> f.close()
>>> os.remove(file_loc)
```

sage.combinat.designs.ext_rep.**open_extrep_url**(*url*)

Try to guess the compression type from extension and open the extrep file pointed to by the url. This function (unlike open_extrep_file) returns the uncompressed text contained in the file.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: proc = ext_rep.XTreeProcessor()
sage: s = ext_rep.open_extrep_url("file://" + file_loc)
sage: proc.parse(s)
sage: os.remove(file_loc)

sage: from sage.combinat.designs import ext_rep
sage: s = ext_rep.designs_from_XML_url("http://designtheory.org/database/v-b-k/v3-
↪b6-k2.icgsa.txt.bz2") # optional - internet
```

```
>>> from sage.all import *
>>> from sage.combinat.designs import ext_rep
>>> file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
>>> proc = ext_rep.XTreeProcessor()
>>> s = ext_rep.open_extrep_url("file://" + file_loc)
>>> proc.parse(s)
>>> os.remove(file_loc)

>>> from sage.combinat.designs import ext_rep
>>> s = ext_rep.designs_from_XML_url("http://designtheory.org/database/v-b-k/v3-
↪b6-k2.icgsa.txt.bz2") # optional - internet
```

5.1.88 Database of generalised quadrangles with spread

This module implements some construction of generalised quadrangles with spread.

EXAMPLES:

```
sage: GQ, S = designs.generalised_quadrangle_with_spread(4, 16, check=False)
sage: GQ
Incidence structure with 325 points and 1105 blocks
sage: GQ2, O = designs.generalised_quadrangle_hermitian_with_ovoid(4)
sage: GQ2
Incidence structure with 1105 points and 325 blocks
sage: GQ3 = GQ.dual()
sage: set(GQ3._points) == set(GQ2._points)
True
sage: GQ2.is_isomorphic(GQ3) # long time
True
```

```
>>> from sage.all import *
>>> GQ, S = designs.generalised_quadrangle_with_spread(Integer(4), Integer(16),␣
↪check=False)
>>> GQ
Incidence structure with 325 points and 1105 blocks
>>> GQ2, O = designs.generalised_quadrangle_hermitian_with_ovoid(Integer(4))
>>> GQ2
Incidence structure with 1105 points and 325 blocks
>>> GQ3 = GQ.dual()
>>> set(GQ3._points) == set(GQ2._points)
True
>>> GQ2.is_isomorphic(GQ3) # long time
True
```

REFERENCES:

- [PT2009]
- [TP1994]
- Wikipedia article Generalized_quadrangle

AUTHORS:

- Ivo Maffei (2020-07-26): initial version

sage.combinat.designs.gen_quadrangles_with_spread.**dual_GQ_ovoid**(GQ, O)

Compute the dual incidence structure of GQ and return the image of O under the dual map

INPUT:

- `GQ` – IncidenceStructure; the generalised quadrangle we want the dual of
- `O` – iterable; the iterable of blocks we want to compute the dual

OUTPUT: a pair `(D, S)` where `D` is the dual of `GQ` and `S` is the dual of `O`

EXAMPLES:

```
sage: from sage.combinat.designs.gen_quadrangles_with_spread import \
....: dual_GQ_ovoid
sage: t = designs.generalised_quadrangle_hermitian_with_ovoid(3)
sage: t[0].is_generalized_quadrangle(parameters=True)
(9, 3)
sage: t = dual_GQ_ovoid(*t)
sage: t[0].is_generalized_quadrangle(parameters=True)
```

(continues on next page)

(continued from previous page)

```
(3, 9)
sage: all([x in t[0] for x in t[1]])
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.gen_quadrangles_with_spread import dual_GQ_ovoid
>>> t = designs.generalised_quadrangle_hermitian_with_ovoid(Integer(3))
>>> t[Integer(0)].is_generalized_quadrangle(parameters=True)
(9, 3)
>>> t = dual_GQ_ovoid(*t)
>>> t[Integer(0)].is_generalized_quadrangle(parameters=True)
(3, 9)
>>> all([x in t[Integer(0)] for x in t[Integer(1)]])
True
```

sage.combinat.designs.gen_quadrangles_with_spread.**generalised_quadrangle_hermitian_with_ovoid**(q)

Construct the generalised quadrangle $H(3, q^2)$ with an ovoid.

The GQ has order (q^2, q).

INPUT:

- `q` – integer; a prime power

OUTPUT:

A pair `(D, O)` where `D` is an IncidenceStructure representing the generalised quadrangle and `O` is a list of points of `D` which constitute an ovoid of `D`.

EXAMPLES:

```
sage: t = designs.generalised_quadrangle_hermitian_with_ovoid(4)
sage: t[0]
Incidence structure with 1105 points and 325 blocks
sage: len(t[1])
65
sage: G = t[0].intersection_graph([1])  # line graph
sage: G.is_strongly_regular(True)
(325, 68, 3, 17)
sage: set(t[0].block_sizes())
{17}
```

```
>>> from sage.all import *
>>> t = designs.generalised_quadrangle_hermitian_with_ovoid(Integer(4))
>>> t[Integer(0)]
Incidence structure with 1105 points and 325 blocks
>>> len(t[Integer(1)])
65
>>> G = t[Integer(0)].intersection_graph([Integer(1)])  # line graph
>>> G.is_strongly_regular(True)
(325, 68, 3, 17)
>>> set(t[Integer(0)].block_sizes())
{17}
```

REFERENCES:

For more on $H(3, q^2)$ and the construction implemented here see [PT2009] or [TP1994].

sage.combinat.designs.gen_quadrangles_with_spread.**generalised_quadrangle_with_spread**(*s*, *t*, *existence=False*, *check=True*)

Construct a generalised quadrangle GQ of order (s, t) with a spread S.

INPUT:

- `s`, `t` – integers; order of the generalised quadrangle
- `existence` – boolean;
- `check` – boolean; if `True`, then Sage checks that the object built is correct. (default: `True`)

OUTPUT:

A pair (GQ, S) where GQ is a `IncidenceStructure` representing the generalised quadrangle and S is a list of blocks of GQ representing the spread of GQ.

EXAMPLES:

```
sage: t = designs.generalised_quadrangle_with_spread(3, 9)
sage: t[0]
Incidence structure with 112 points and 280 blocks
sage: designs.generalised_quadrangle_with_spread(5, 25, existence=True)
True
sage: (designs.generalised_quadrangle_with_spread(4, 16, check=False))[0]
Incidence structure with 325 points and 1105 blocks
sage: designs.generalised_quadrangle_with_spread(0, 2, existence=True)
False
```

```
>>> from sage.all import *
>>> t = designs.generalised_quadrangle_with_spread(Integer(3), Integer(9))
>>> t[Integer(0)]
Incidence structure with 112 points and 280 blocks
>>> designs.generalised_quadrangle_with_spread(Integer(5), Integer(25),␣
↪existence=True)
True
>>> (designs.generalised_quadrangle_with_spread(Integer(4), Integer(16),␣
↪check=False))[Integer(0)]
Incidence structure with 325 points and 1105 blocks
>>> designs.generalised_quadrangle_with_spread(Integer(0), Integer(2),␣
↪existence=True)
False
```

REFERENCES:

For more on generalised quadrangles and their spread see [PT2009] or [TP1994].

sage.combinat.designs.gen_quadrangles_with_spread.**is_GQ_with_spread**(*GQ*, *S*, *s=None*, *t=None*)

Check if GQ is a generalised quadrangle of order (s, t) and check that S is a spread of GQ

INPUT:

- `GQ` – IncidenceStructure; the incidence structure that is supposed to be a generalised quadrangle

- `S` – iterable; the spread of `GQ` as an iterable of the blocks of `GQ`
- `s`, `t` – integers (optional); if (s,t) are given, then we check that `GQ` has order (s,t)

EXAMPLES:

```
sage: from sage.combinat.designs.gen_quadrangles_with_spread import *
sage: t = generalised_quadrangle_hermitian_with_ovoid(3)
sage: is_GQ_with_spread(*t)
Traceback (most recent call last):
...
TypeError: 'int' object is not iterable
sage: t = dual_GQ_ovoid(*t)
sage: is_GQ_with_spread(*t)
True
sage: is_GQ_with_spread(*t, s=3)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.gen_quadrangles_with_spread import *
>>> t = generalised_quadrangle_hermitian_with_ovoid(Integer(3))
>>> is_GQ_with_spread(*t)
Traceback (most recent call last):
...
TypeError: 'int' object is not iterable
>>> t = dual_GQ_ovoid(*t)
>>> is_GQ_with_spread(*t)
True
>>> is_GQ_with_spread(*t, s=Integer(3))
True
```

5.1.89 Incidence structures (i.e. hypergraphs, i.e. set systems)

An incidence structure is specified by a list of points, blocks, or an incidence matrix ([1],[2]). *IncidenceStructure* instances have the following methods:

automorphism_group()	Return the subgroup of the automorphism group of the incidence graph which respects the P B partition. It is (isomorphic to) the automorphism group of the block design, although the degrees differ.
block_sizes()	Return the set of block sizes.
blocks()	Return the list of blocks.
canonical_label()	Return a canonical label for the incidence structure.
coloring()	Compute a (weak) k-coloring of the hypergraph.
complement()	Return the complement of the incidence structure.
copy()	Return a copy of the incidence structure.
degree()	Return the degree of a point `p` (or a set of points).
degrees()	Return the degree of all sets of given size, or the degree of all points.
dual()	Return the dual of the incidence structure.
edge_coloring()	Compute a proper edge-coloring.
ground_set()	Return the ground set (i.e the list of points).

continues on next page

[1] Block designs and incidence structures from wikipedia, Wikipedia article Block_design Wikipedia article Incidence_structure
[2] E. Assmus, J. Key, Designs and their codes, CUP, 1992.

Table 2 – continued from previous page

`incidence_graph()`	Return the incidence graph of the incidence structure.
`incidence_matrix()`	Return the incidence matrix A of the design. A is a $(v \times b)$ matrix defined by: `A[i,j] = 1` if `i` is in block `B_j` and 0 otherwise.
`induced_substructure()`	Return the substructure induced by a set of points.
`intersection_graph()`	Return the intersection graph of the incidence structure.
`is_berge_cyclic()`	Check whether `self` is a Berge-Cyclic uniform hypergraph.
`is_connected()`	Test whether the design is connected.
`is_generalized_quadrangle()`	Test if the incidence structure is a generalized quadrangle.
`is_isomorphic()`	Return whether the two incidence structures are isomorphic.
`is_regular()`	Test whether the incidence structure is r-regular.
`is_resolvable()`	Test whether the hypergraph is resolvable.
`is_simple()`	Test whether this design is simple (i.e. no repeated block).
`is_spread()`	Check whether the input is a spread for `self`.
`is_t_design()`	Test whether `self` is a $t-(v,k,l)$ design.
`is_uniform()`	Test whether the incidence structure is k-uniform
`isomorphic_substructures_iterator()`	Iterate over all copies of `H2` contained in `self`.
`num_blocks()`	Return the number of blocks.
`num_points()`	Return the size of the ground set.
`packing()`	Return a maximum packing.
`rank()`	Return the rank of the hypergraph (the maximum size of a block).
`relabel()`	Relabel the ground set.
`trace()`	Return the trace of a set of points.

REFERENCES:

AUTHORS:

- Peter Dobcsanyi and David Joyner (2007-2008)

  This is a significantly modified form of part of the module block_design.py (version 0.6) written by Peter Dobcsanyi peter@designtheory.org.

- Vincent Delecroix (2014): major rewrite

Methods

class sage.combinat.designs.incidence_structures.**IncidenceStructure**(*points=None*, *blocks=None*, *incidence_matrix=None*, *name=None*, *check=True*, *copy=True*)

Bases: `object`

A base class for incidence structures (i.e. hypergraphs, i.e. set systems)

An incidence structure (i.e. hypergraph, i.e. set system) can be defined from a collection of blocks (i.e. sets, i.e. edges), optionally with an explicit ground set (i.e. point set, i.e. vertex set). Alternatively they can be defined from a binary incidence matrix.

INPUT:

- `points` – (i.e. ground set, i.e. vertex set) the underlying set. If `points` is an integer v, then the set is considered to be $\{0, ..., v-1\}$.

Note

The following syntax, where `points` is omitted, automatically defines the ground set as the union of the blocks:

```
sage: H = IncidenceStructure([['a','b','c'],['c','d','e']])
sage: sorted(H.ground_set())
['a', 'b', 'c', 'd', 'e']
```

```
>>> from sage.all import *
>>> H = IncidenceStructure([['a','b','c'],['c','d','e']])
>>> sorted(H.ground_set())
['a', 'b', 'c', 'd', 'e']
```

- `blocks` – (i.e. edges, i.e. sets) the blocks defining the incidence structure; can be any iterable
- `incidence_matrix` – a binary incidence matrix; each column represents a set
- `name` – string (such as "Fano plane")
- `check` – whether to check the input
- `copy` – (use with caution) if set to `False` then `blocks` must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your `blocks` object will become the `IncidenceStructure` instance's internal data.

EXAMPLES:

An incidence structure can be constructed by giving the number of points and the list of blocks:

```
sage: IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,
↪5]])
Incidence structure with 7 points and 7 blocks
```

```
>>> from sage.all import *
>>> IncidenceStructure(Integer(7), [[Integer(0),Integer(1),Integer(2)],
↪[Integer(0),Integer(3),Integer(4)],[Integer(0),Integer(5),Integer(6)],
↪[Integer(1),Integer(3),Integer(5)],[Integer(1),Integer(4),Integer(6)],
↪[Integer(2),Integer(3),Integer(6)],[Integer(2),Integer(4),Integer(5)]])
Incidence structure with 7 points and 7 blocks
```

Only providing the set of blocks is sufficient. In this case, the ground set is defined as the union of the blocks:

```
sage: IncidenceStructure([[1,2,3],[2,3,4]])
Incidence structure with 4 points and 2 blocks
```

```
>>> from sage.all import *
>>> IncidenceStructure([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(3),
↪Integer(4)]])
Incidence structure with 4 points and 2 blocks
```

Or by its adjacency matrix (a $\{0,1\}$-matrix in which rows are indexed by points and columns by blocks):

```
sage: m = matrix([[0,1,0],[0,0,1],[1,0,1],[1,1,1]])                          #␣
↪needs sage.modules
sage: IncidenceStructure(m)                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
Incidence structure with 4 points and 3 blocks
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(1)],[Integer(1),Integer(0),Integer(1)],[Integer(1),Integer(1),
↪Integer(1)]])                                    # needs sage.modules
>>> IncidenceStructure(m)                                                    #␣
↪needs sage.modules
Incidence structure with 4 points and 3 blocks
```

The points can be any (hashable) object:

```
sage: V = [(0,'a'),(0,'b'),(1,'a'),(1,'b')]
sage: B = [(V[0],V[1],V[2]), (V[1],V[2]), (V[0],V[2])]
sage: I = IncidenceStructure(V, B)
sage: I.ground_set()
[(0, 'a'), (0, 'b'), (1, 'a'), (1, 'b')]
sage: I.blocks()
[[(0, 'a'), (0, 'b'), (1, 'a')], [(0, 'a'), (1, 'a')], [(0, 'b'), (1, 'a')]]
```

```
>>> from sage.all import *
>>> V = [(Integer(0),'a'),(Integer(0),'b'),(Integer(1),'a'),(Integer(1),'b')]
>>> B = [(V[Integer(0)],V[Integer(1)],V[Integer(2)]), (V[Integer(1)],
↪V[Integer(2)]), (V[Integer(0)],V[Integer(2)])]
>>> I = IncidenceStructure(V, B)
>>> I.ground_set()
[(0, 'a'), (0, 'b'), (1, 'a'), (1, 'b')]
>>> I.blocks()
[[(0, 'a'), (0, 'b'), (1, 'a')], [(0, 'a'), (1, 'a')], [(0, 'b'), (1, 'a')]]
```

The order of the points and blocks does not matter as they are sorted on input (see Issue #11333):

```
sage: A = IncidenceStructure([0,1,2], [[0],[0,2]])
sage: B = IncidenceStructure([1,0,2], [[0],[2,0]])
sage: B == A
True

sage: C = BlockDesign(2, [[0], [1,0]])
sage: D = BlockDesign(2, [[0,1], [0]])
sage: C == D
True
```

```
>>> from sage.all import *
>>> A = IncidenceStructure([Integer(0),Integer(1),Integer(2)], [[Integer(0)],
↪[Integer(0),Integer(2)]])
>>> B = IncidenceStructure([Integer(1),Integer(0),Integer(2)], [[Integer(0)],
↪[Integer(2),Integer(0)]])
>>> B == A
True

>>> C = BlockDesign(Integer(2), [[Integer(0)], [Integer(1),Integer(0)]])
```

(continues on next page)

(continued from previous page)

```
>>> D = BlockDesign(Integer(2), [[Integer(0),Integer(1)], [Integer(0)]])
>>> C == D
True
```

If you care for speed, you can set `copy` to `False`, but in that case, your input must be a list of lists and the ground set must be $0, ..., v-1$:

```
sage: blocks = [[0,1],[2,0],[1,2]]  # a list of lists of integers
sage: I = IncidenceStructure(3, blocks, copy=False)
sage: I._blocks is blocks
True
```

```
>>> from sage.all import *
>>> blocks = [[Integer(0),Integer(1)],[Integer(2),Integer(0)],[Integer(1),
↪Integer(2)]]  # a list of lists of integers
>>> I = IncidenceStructure(Integer(3), blocks, copy=False)
>>> I._blocks is blocks
True
```

automorphism_group()

Return the subgroup of the automorphism group of the incidence graph which respects the P B partition. It is (isomorphic to) the automorphism group of the block design, although the degrees differ.

EXAMPLES:

```
sage: # needs sage.groups sage.rings.finite_rings
sage: P = designs.DesarguesianProjectivePlaneDesign(2); P
(7,3,1)-Balanced Incomplete Block Design
sage: G = P.automorphism_group()
sage: G.is_isomorphic(PGL(3,2))
True
sage: G
Permutation Group with generators [...]
sage: G.cardinality()
168
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.finite_rings
>>> P = designs.DesarguesianProjectivePlaneDesign(Integer(2)); P
(7,3,1)-Balanced Incomplete Block Design
>>> G = P.automorphism_group()
>>> G.is_isomorphic(PGL(Integer(3),Integer(2)))
True
>>> G
Permutation Group with generators [...]
>>> G.cardinality()
168
```

A non self-dual example:

```
sage: IS = IncidenceStructure(list(range(4)), [[0,1,2,3],[1,2,3]])
sage: IS.automorphism_group().cardinality()                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
6
sage: IS.dual().automorphism_group().cardinality()                          #␣
↪needs sage.groups sage.modules
1
```

```
>>> from sage.all import *
>>> IS = IncidenceStructure(list(range(Integer(4))), [[Integer(0),Integer(1),
↪Integer(2),Integer(3)],[Integer(1),Integer(2),Integer(3)]])
>>> IS.automorphism_group().cardinality()                                   #␣
↪needs sage.groups
6
>>> IS.dual().automorphism_group().cardinality()                            #␣
↪needs sage.groups sage.modules
1
```

Examples with non-integer points:

```
sage: I = IncidenceStructure('abc', ('ab','ac','bc'))
sage: I.automorphism_group()                                                #␣
↪needs sage.groups
Permutation Group with generators [('b','c'), ('a','b')]
sage: IncidenceStructure([[(1,2),(3,4)]]).automorphism_group()              #␣
↪needs sage.groups
Permutation Group with generators [((1,2),(3,4))]
```

```
>>> from sage.all import *
>>> I = IncidenceStructure('abc', ('ab','ac','bc'))
>>> I.automorphism_group()                                                  #␣
↪needs sage.groups
Permutation Group with generators [('b','c'), ('a','b')]
>>> IncidenceStructure([[(Integer(1),Integer(2)),(Integer(3),Integer(4))]]).
↪automorphism_group()                 # needs sage.groups
Permutation Group with generators [((1,2),(3,4))]
```

block_sizes()

Return the set of block sizes.

EXAMPLES:

```
sage: BD = IncidenceStructure(8, [[0,1,3],[1,4,5,6],[1,2],[5,6,7]])
sage: BD.block_sizes()
[3, 2, 4, 3]
sage: BD = IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,
↪6],[2,4,5]])
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
```

```
>>> from sage.all import *
>>> BD = IncidenceStructure(Integer(8), [[Integer(0),Integer(1),Integer(3)],
↪[Integer(1),Integer(4),Integer(5),Integer(6)],[Integer(1),Integer(2)],
↪[Integer(5),Integer(6),Integer(7)]])
```

(continues on next page)

(continued from previous page)

```
>>> BD.block_sizes()
[3, 2, 4, 3]
>>> BD = IncidenceStructure(Integer(7),[[Integer(0),Integer(1),Integer(2)],
↪[Integer(0),Integer(3),Integer(4)],[Integer(0),Integer(5),Integer(6)],
↪[Integer(1),Integer(3),Integer(5)],[Integer(1),Integer(4),Integer(6)],
↪[Integer(2),Integer(3),Integer(6)],[Integer(2),Integer(4),Integer(5)]])
>>> BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
```

blocks()

Return the list of blocks.

EXAMPLES:

```
sage: BD = IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,
↪6],[2,4,5]])
sage: BD.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

```
>>> from sage.all import *
>>> BD = IncidenceStructure(Integer(7),[[Integer(0),Integer(1),Integer(2)],
↪[Integer(0),Integer(3),Integer(4)],[Integer(0),Integer(5),Integer(6)],
↪[Integer(1),Integer(3),Integer(5)],[Integer(1),Integer(4),Integer(6)],
↪[Integer(2),Integer(3),Integer(6)],[Integer(2),Integer(4),Integer(5)]])
>>> BD.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

canonical_label()

Return a canonical label for the incidence structure.

A canonical label is relabeling of the points into integers $\{0, ..., n-1\}$ such that isomorphic incidence structures are relabelled to equal objects.

EXAMPLES:

```
sage: # needs sage.schemes
sage: fano1 = designs.balanced_incomplete_block_design(7,3)
sage: fano2 = designs.projective_plane(2)
sage: fano1 == fano2
False
sage: fano1.relabel(fano1.canonical_label())
sage: fano2.relabel(fano2.canonical_label())
sage: fano1 == fano2
True
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> fano1 = designs.balanced_incomplete_block_design(Integer(7),Integer(3))
>>> fano2 = designs.projective_plane(Integer(2))
>>> fano1 == fano2
False
>>> fano1.relabel(fano1.canonical_label())
>>> fano2.relabel(fano2.canonical_label())
```

(continues on next page)

(continued from previous page)

```
>>> fano1 == fano2
True
```

coloring(*k*, *solver=None*, *verbose=None*, *integrality_tolerance=0*)

Compute a (weak) k-coloring of the hypergraph.

A weak coloring of a hypergraph $\mathcal{H}$ is an assignment of colors to its vertices such that no set is monochromatic.

INPUT:

- `k` – integer; compute a coloring with k colors if an integer is provided, otherwise returns an optimal coloring (i.e. with the minimum possible number of colors).
- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – nonnegative integer (default: 0); set the level of verbosity you want from the linear program solver. Since the problem is NP-complete, its solving may take some time depending on the graph. A value of 0 means that there will be no message printed by the solver.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`

EXAMPLES:

The Fano plane has chromatic number 3:

```
sage: len(designs.steiner_triple_system(7).coloring())                         #␣
↪needs sage.numerical.mip
3
```

```
>>> from sage.all import *
>>> len(designs.steiner_triple_system(Integer(7)).coloring())                  ␣
↪     # needs sage.numerical.mip
3
```

One admissible 3-coloring:

```
sage: designs.steiner_triple_system(7).coloring()    # not tested             #␣
↪needs sage.numerical.mip
[[0, 2, 5, 1], [4, 3], [6]]
```

```
>>> from sage.all import *
>>> designs.steiner_triple_system(Integer(7)).coloring()    # not tested      ␣
↪     # needs sage.numerical.mip
[[0, 2, 5, 1], [4, 3], [6]]
```

The chromatic number of a graph is equal to the chromatic number of its 2-uniform corresponding hypergraph:

```
sage: g = graphs.PetersenGraph()
sage: H = IncidenceStructure(g.edges(sort=True, labels=False))
sage: len(g.coloring())
3
sage: len(H.coloring())                                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.numerical.mip
3
```

```
>>> from sage.all import *
>>> g = graphs.PetersenGraph()
>>> H = IncidenceStructure(g.edges(sort=True, labels=False))
>>> len(g.coloring())
3
>>> len(H.coloring())                                                    #␣
↪needs sage.numerical.mip
3
```

complement(*uniform=False*)

Return the complement of the incidence structure.

Two different definitions of "complement" are made available, according to the value of `uniform`.

INPUT:

- `uniform` – boolean
  - if set to `False` (default), returns the incidence structure whose blocks are the complements of all blocks of the incidence structure.
  - If set to `True` and the incidence structure is k-uniform, returns the incidence structure whose blocks are all k-sets of the ground set that do not appear in `self`.

EXAMPLES:

The complement of a *BalancedIncompleteBlockDesign* is also a 2-design:

```
sage: bibd = designs.balanced_incomplete_block_design(13,4)               #␣
↪needs sage.schemes
sage: bibd.is_t_design(return_parameters=True)                            #␣
↪needs sage.schemes
(True, (2, 13, 4, 1))
sage: bibd.complement().is_t_design(return_parameters=True)               #␣
↪needs sage.schemes
(True, (2, 13, 9, 6))
```

```
>>> from sage.all import *
>>> bibd = designs.balanced_incomplete_block_design(Integer(13),Integer(4))  ␣
↪              # needs sage.schemes
>>> bibd.is_t_design(return_parameters=True)                              #␣
↪needs sage.schemes
(True, (2, 13, 4, 1))
>>> bibd.complement().is_t_design(return_parameters=True)                 #␣
↪needs sage.schemes
(True, (2, 13, 9, 6))
```

The "uniform" complement of a graph is a graph:

```
sage: g = graphs.PetersenGraph()
sage: G = IncidenceStructure(g.edges(sort=True, labels=False))
sage: H = G.complement(uniform=True)
```

(continues on next page)

(continued from previous page)

```
sage: h = Graph(H.blocks())
sage: g == h
False
sage: g == h.complement()
True
```

```
>>> from sage.all import *
>>> g = graphs.PetersenGraph()
>>> G = IncidenceStructure(g.edges(sort=True, labels=False))
>>> H = G.complement(uniform=True)
>>> h = Graph(H.blocks())
>>> g == h
False
>>> g == h.complement()
True
```

copy()

Return a copy of the incidence structure.

EXAMPLES:

```
sage: IS = IncidenceStructure([[1,2,3,"e"]], name='Test')
sage: IS
Incidence structure with 4 points and 1 blocks
sage: copy(IS)
Incidence structure with 4 points and 1 blocks
sage: [1, 2, 3, 'e'] in copy(IS)
True
sage: copy(IS)._name
'Test'
```

```
>>> from sage.all import *
>>> IS = IncidenceStructure([[Integer(1),Integer(2),Integer(3),"e"]], name=
↪'Test')
>>> IS
Incidence structure with 4 points and 1 blocks
>>> copy(IS)
Incidence structure with 4 points and 1 blocks
>>> [Integer(1), Integer(2), Integer(3), 'e'] in copy(IS)
True
>>> copy(IS)._name
'Test'
```

degree(*p=None*, *subset=False*)

Return the degree of a point `p` (or a set of points).

The degree of a point (or set of points) is the number of blocks that contain it.

INPUT:

- `p` – a point (or a set of points) of the incidence structure
- `subset` – boolean (default: `False`); whether to interpret the argument as a set of point or as a point (default)

EXAMPLES:

```
sage: designs.steiner_triple_system(9).degree(3)
4
sage: designs.steiner_triple_system(9).degree({1,2},subset=True)
1
```

```
>>> from sage.all import *
>>> designs.steiner_triple_system(Integer(9)).degree(Integer(3))
4
>>> designs.steiner_triple_system(Integer(9)).degree({Integer(1),Integer(2)},
↪subset=True)
1
```

degrees(*size=None*)

Return the degree of all sets of given size, or the degree of all points.

The degree of a point (or set of point) is the number of blocks that contain it.

INPUT:

- `size` – integer; return the degree of all subsets of points of cardinality `size`. When `size=None`, the function outputs the degree of all points.

> **Note**
>
> When `size=None` the output is indexed by the points. When `size=1` it is indexed by tuples of size 1. This is the same information, stored slightly differently.

OUTPUT: a dictionary whose values are degrees and keys are either:

- the points of the incidence structure if `size=None` (default)
- the subsets of size `size` of the points stored as tuples

EXAMPLES:

```
sage: IncidenceStructure([[1,2,3],[1,4]]).degrees(2)
{(1, 2): 1, (1, 3): 1, (1, 4): 1, (2, 3): 1, (2, 4): 0, (3, 4): 0}
```

```
>>> from sage.all import *
>>> IncidenceStructure([[Integer(1),Integer(2),Integer(3)],[Integer(1),
↪Integer(4)]]).degrees(Integer(2))
{(1, 2): 1, (1, 3): 1, (1, 4): 1, (2, 3): 1, (2, 4): 0, (3, 4): 0}
```

In a Steiner triple system, all pairs have degree 1:

```
sage: S13 = designs.steiner_triple_system(13)
sage: all(v == 1 for v in S13.degrees(2).values())
True
```

```
>>> from sage.all import *
>>> S13 = designs.steiner_triple_system(Integer(13))
>>> all(v == Integer(1) for v in S13.degrees(Integer(2)).values())
True
```

dual(*algorithm=None*)

Return the dual of the incidence structure.

INPUT:

- algorithm – whether to use Sage's implementation (algorithm=None, default) or use GAP's (algorithm='gap')

> **Note**
>
> The algorithm='gap' option requires GAP's Design package (included in the gap_packages Sage spkg).

EXAMPLES:

The dual of a projective plane is a projective plane:

```
sage: PP = designs.DesarguesianProjectivePlaneDesign(4)                    #␣
↪needs sage.rings.finite_rings
sage: PP.dual().is_t_design(return_parameters=True)                        #␣
↪needs sage.modules sage.rings.finite_rings
(True, (2, 21, 5, 1))
```

```
>>> from sage.all import *
>>> PP = designs.DesarguesianProjectivePlaneDesign(Integer(4))              ␣
↪     # needs sage.rings.finite_rings
>>> PP.dual().is_t_design(return_parameters=True)                        #␣
↪needs sage.modules sage.rings.finite_rings
(True, (2, 21, 5, 1))
```

REFERENCE:

- Leonard Soicher, Design package manual

edge_coloring()

Compute a proper edge-coloring.

A proper edge-coloring is an assignment of colors to the sets of the incidence structure such that two sets with non-empty intersection receive different colors. The coloring returned minimizes the number of colors.

OUTPUT: a partition of the sets into color classes

EXAMPLES:

```
sage: H = Hypergraph([{1,2,3},{2,3,4},{3,4,5},{4,5,6}]); H
Incidence structure with 6 points and 4 blocks
sage: C = H.edge_coloring()
sage: C # random
[[[3, 4, 5]], [[2, 3, 4]], [[4, 5, 6], [1, 2, 3]]]
sage: Set(map(Set,sum(C,[]))) == Set(map(Set,H.blocks()))
True
```

```
>>> from sage.all import *
>>> H = Hypergraph([{Integer(1),Integer(2),Integer(3)},{Integer(2),Integer(3),
↪Integer(4)},{Integer(3),Integer(4),Integer(5)},{Integer(4),Integer(5),
```

(continues on next page)

(continued from previous page)

```
↪Integer(6)}]); H
Incidence structure with 6 points and 4 blocks
>>> C = H.edge_coloring()
>>> C # random
[[[3, 4, 5]], [[2, 3, 4]], [[4, 5, 6], [1, 2, 3]]]
>>> Set(map(Set,sum(C,[]))) == Set(map(Set,H.blocks()))
True
```

ground_set()

Return the ground set (i.e the list of points).

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[0,2]]).ground_set()
[0, 1, 2]
```

```
>>> from sage.all import *
>>> IncidenceStructure(Integer(3), [[Integer(0),Integer(1)],[Integer(0),
↪Integer(2)]]).ground_set()
[0, 1, 2]
```

incidence_graph(*labels=False*)

Return the incidence graph of the incidence structure.

A point and a block are adjacent in this graph whenever they are incident.

INPUT:

- `labels` – boolean; whether to return a graph whose vertices are integers, or labelled elements
  - `labels is False` – default; in this case the first vertices of the graphs are the elements of `ground_set()`, and appear in the same order. Similarly, the following vertices represent the elements of `blocks()`, and appear in the same order.
  - `labels is True`, the points keep their original labels, and the blocks are `Set` objects.

    Note that the labelled incidence graph can be incorrect when blocks are repeated, and on some (rare) occasions when the elements of `ground_set()` mix `Set()` and non-`Set` objects.

EXAMPLES:

```
sage: BD = IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],
....:                             [1,4,6],[2,3,6],[2,4,5]])
sage: BD.incidence_graph()                                              #␣
↪needs sage.modules
Bipartite graph on 14 vertices
sage: A = BD.incidence_matrix()                                         #␣
↪needs sage.modules
sage: Graph(block_matrix([[A*0, A],                                     #␣
↪needs sage.modules
....:                     [A.transpose(),A*0]])) == BD.incidence_graph()
True
```

```
>>> from sage.all import *
>>> BD = IncidenceStructure(Integer(7), [[Integer(0),Integer(1),Integer(2)],
```

(continues on next page)

(continued from previous page)

```
↪[Integer(0),Integer(3),Integer(4)],[Integer(0),Integer(5),Integer(6)],
↪[Integer(1),Integer(3),Integer(5)],
...                              [Integer(1),Integer(4),Integer(6)],
↪[Integer(2),Integer(3),Integer(6)],[Integer(2),Integer(4),Integer(5)]])
>>> BD.incidence_graph()                                                  #␣
↪needs sage.modules
Bipartite graph on 14 vertices
>>> A = BD.incidence_matrix()                                             #␣
↪needs sage.modules
>>> Graph(block_matrix([[A*Integer(0), A],                                 ␣
↪     # needs sage.modules
...                     [A.transpose(),A*Integer(0)]])) == BD.incidence_
↪graph()
True
```

incidence_matrix()

Return the incidence matrix A of the design. A is a $(v \times b)$ matrix defined by: `A[i,j] = 1` if `i` is in block `B_j` and 0 otherwise.

EXAMPLES:

```
sage: BD = IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],
....:                              [1,4,6],[2,3,6],[2,4,5]])
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
sage: BD.incidence_matrix()                                               #␣
↪needs sage.modules
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[1 0 0 0 0 1 1]
[0 1 0 1 0 1 0]
[0 1 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 0 1 0 1 1 0]

sage: I = IncidenceStructure('abc', ('ab','abc','ac','c'))
sage: I.incidence_matrix()                                                #␣
↪needs sage.modules
[1 1 1 0]
[1 1 0 0]
[0 1 1 1]
```

```
>>> from sage.all import *
>>> BD = IncidenceStructure(Integer(7), [[Integer(0),Integer(1),Integer(2)],
↪[Integer(0),Integer(3),Integer(4)],[Integer(0),Integer(5),Integer(6)],
↪[Integer(1),Integer(3),Integer(5)],
...                              [Integer(1),Integer(4),Integer(6)],
↪[Integer(2),Integer(3),Integer(6)],[Integer(2),Integer(4),Integer(5)]])
>>> BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
>>> BD.incidence_matrix()                                                 #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[1 0 0 0 0 1 1]
[0 1 0 1 0 1 0]
[0 1 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 0 1 0 1 1 0]

>>> I = IncidenceStructure('abc', ('ab','abc','ac','c'))
>>> I.incidence_matrix()                                                    #␣
↪needs sage.modules
[1 1 1 0]
[1 1 0 0]
[0 1 1 1]
```

induced_substructure(*points*)

Return the substructure induced by a set of points.

The substructure induced in $\mathcal{H}$ by a set $X \subseteq V(\mathcal{H})$ of points is the incidence structure $\mathcal{H}_X$ defined on X whose sets are all $S \in \mathcal{H}$ such that $S \subseteq X$.

INPUT:

- `points` – set of points

Note: This method goes over all sets of `self` before building a new *IncidenceStructure* (which involves some relabelling and sorting). It probably should not be called in a performance-critical code.

EXAMPLES:

A Fano plane with one point removed:

```
sage: F = designs.steiner_triple_system(7)
sage: F.induced_substructure([0..5])
Incidence structure with 6 points and 4 blocks
```

```
>>> from sage.all import *
>>> F = designs.steiner_triple_system(Integer(7))
>>> F.induced_substructure((ellipsis_range(Integer(0),Ellipsis,Integer(5))))
Incidence structure with 6 points and 4 blocks
```

intersection_graph(*sizes=None*)

Return the intersection graph of the incidence structure.

The vertices of this graph are the *blocks()* of the incidence structure. Two of them are adjacent if the size of their intersection belongs to the set `sizes`.

INPUT:

- `sizes` – list/set of integers; for convenience, setting `sizes` to `5` has the same effect as `sizes=[5]`. When set to `None` (default), behaves as `sizes=PositiveIntegers()`.

EXAMPLES:

The intersection graph of a *balanced_incomplete_block_design()* is a strongly regular graph (when it is not trivial):

```
sage: BIBD =  designs.balanced_incomplete_block_design(19,3)
sage: G = BIBD.intersection_graph(1)
sage: G.is_strongly_regular(parameters=True)
(57, 24, 11, 9)
```

```
>>> from sage.all import *
>>> BIBD =  designs.balanced_incomplete_block_design(Integer(19),Integer(3))
>>> G = BIBD.intersection_graph(Integer(1))
>>> G.is_strongly_regular(parameters=True)
(57, 24, 11, 9)
```

is_berge_cyclic()

Check whether `self` is a Berge-Cyclic uniform hypergraph.

A k-uniform Berge cycle (named after Claude Berge) of length ℓ is a cyclic list of distinct k-sets $F_1, \ldots, F_\ell$, $\ell > 1$, and distinct vertices $C = \{v_1, \ldots, v_\ell\}$ such that for each $1 \leq i \leq \ell$, F_i contains v_i and v_{i+1} (where $v_{l+1} = v_1$).

A uniform hypergraph is Berge-cyclic if its incidence graph is cyclic. It is called "Berge-acyclic" otherwise.

For more information, see [Fag1983] and Wikipedia article Hypergraph.

EXAMPLES:

```
sage: Hypergraph(5, [[1, 2, 3], [2, 3, 4]]).is_berge_cyclic()              #␣
↪needs sage.modules
True
sage: Hypergraph(6, [[1, 2, 3], [3, 4, 5]]).is_berge_cyclic()              #␣
↪needs sage.modules
False
```

```
>>> from sage.all import *
>>> Hypergraph(Integer(5), [[Integer(1), Integer(2), Integer(3)], [Integer(2),
↪ Integer(3), Integer(4)]]).is_berge_cyclic()              # needs sage.
↪modules
True
>>> Hypergraph(Integer(6), [[Integer(1), Integer(2), Integer(3)], [Integer(3),
↪ Integer(4), Integer(5)]]).is_berge_cyclic()              # needs sage.
↪modules
False
```

is_connected()

Test whether the design is connected.

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[0,2]]).is_connected()
True
sage: IncidenceStructure(4, [[0,1],[2,3]]).is_connected()
False
```

```
>>> from sage.all import *
>>> IncidenceStructure(Integer(3), [[Integer(0),Integer(1)],[Integer(0),
↪Integer(2)]]).is_connected()
True
>>> IncidenceStructure(Integer(4), [[Integer(0),Integer(1)],[Integer(2),
↪Integer(3)]]).is_connected()
False
```

is_generalized_quadrangle(*verbose=False*, *parameters=False*)

Test if the incidence structure is a generalized quadrangle.

An incidence structure is a generalized quadrangle iff (see [BH2012], section 9.6):

- two blocks intersect on at most one point.
- For every point p not in a block B, there is a unique block B' intersecting both $\{p\}$ and B

It is a *regular* generalized quadrangle if furthermore:

- it is $s+1$-`uniform` for some positive integer s.
- it is $t+1$-`regular` for some positive integer t.

For more information, see the Wikipedia article Generalized_quadrangle.

Note

Some references (e.g. [PT2009] or Wikipedia article Generalized_quadrangle) only allow *regular* generalized quadrangles. To use such a definition, see the `parameters` optional argument described below, or the methods `is_regular()` and `is_uniform()`.

INPUT:

- `verbose` – boolean; whether to print an explanation when the instance is not a generalized quadrangle
- `parameters` – (boolean; `False`); if set to `True`, the function returns a pair `(s,t)` instead of `True` answers. In this case, s and t are the integers defined above if they exist (each can be set to `False` otherwise).

EXAMPLES:

```
sage: h = designs.CremonaRichmondConfiguration()                              #␣
↪needs networkx
sage: h.is_generalized_quadrangle()                                           #␣
↪needs networkx
True
```

```
>>> from sage.all import *
>>> h = designs.CremonaRichmondConfiguration()                              #␣
↪needs networkx
>>> h.is_generalized_quadrangle()                                           #␣
↪needs networkx
True
```

This is actually a *regular* generalized quadrangle:

```
sage: h.is_generalized_quadrangle(parameters=True)                    #␣
↪needs networkx
(2, 2)
```

```
>>> from sage.all import *
>>> h.is_generalized_quadrangle(parameters=True)                    #␣
↪needs networkx
(2, 2)
```

is_isomorphic(*other*, *certificate=False*)

Return whether the two incidence structures are isomorphic.

INPUT:

- `other` – an incidence structure
- `certificate` – boolean (default: `False`); whether to return an isomorphism from `self` to `other` instead of a boolean answer

EXAMPLES:

```
sage: # needs sage.schemes
sage: fano1 = designs.balanced_incomplete_block_design(7,3)
sage: fano2 = designs.projective_plane(2)
sage: fano1.is_isomorphic(fano2)
True
sage: fano1.is_isomorphic(fano2,certificate=True)
{0: 0, 1: 1, 2: 2, 3: 6, 4: 4, 5: 3, 6: 5}
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> fano1 = designs.balanced_incomplete_block_design(Integer(7),Integer(3))
>>> fano2 = designs.projective_plane(Integer(2))
>>> fano1.is_isomorphic(fano2)
True
>>> fano1.is_isomorphic(fano2,certificate=True)
{0: 0, 1: 1, 2: 2, 3: 6, 4: 4, 5: 3, 6: 5}
```

is_regular(*r=None*)

Test whether the incidence structure is r-regular.

An incidence structure is said to be r-regular if all its points are incident with exactly r blocks.

INPUT:

- `r` – integer

OUTPUT:

If `r` is defined, a boolean is returned. If `r` is set to `None` (default), the method returns either `False` or the integer `r` such that the incidence structure is r-regular.

Warning

In case of 0-regular incidence structure, beware that `if not H.is_regular()` is a satisfied condition.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(7,3).is_regular()              #␣
↪needs sage.schemes
3
sage: designs.balanced_incomplete_block_design(7,3).is_regular(r=3)           #␣
↪needs sage.schemes
True
sage: designs.balanced_incomplete_block_design(7,3).is_regular(r=4)           #␣
↪needs sage.schemes
False
```

```
>>> from sage.all import *
>>> designs.balanced_incomplete_block_design(Integer(7),Integer(3)).is_
↪regular()             # needs sage.schemes
3
>>> designs.balanced_incomplete_block_design(Integer(7),Integer(3)).is_
↪regular(r=Integer(3))         # needs sage.schemes
True
>>> designs.balanced_incomplete_block_design(Integer(7),Integer(3)).is_
↪regular(r=Integer(4))         # needs sage.schemes
False
```

is_resolvable(*certificate*, *solver=False*, *verbose=None*, *check=0*, *integrality_tolerance=True*)

Test whether the hypergraph is resolvable.

A hypergraph is said to be resolvable if its sets can be partitionned into classes, each of which is a partition of the ground set.

Note

This problem is solved using an Integer Linear Program, and GLPK (the default LP solver) has been reported to be very slow on some instances. If you hit this wall, consider installing a more powerful MILP solver (CPLEX, Gurobi, …).

INPUT:

- `certificate` – boolean; whether to return the classes along with the binary answer (see examples below)
- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`

EXAMPLES:

Some resolvable designs:

```
sage: TD = designs.transversal_design(2,2,resolvable=True)
sage: TD.is_resolvable()
True

sage: AG = designs.AffineGeometryDesign(3,1,GF(2))                          #␣
↪needs sage.combinat
sage: AG.is_resolvable()                                                     #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> TD = designs.transversal_design(Integer(2),Integer(2),resolvable=True)
>>> TD.is_resolvable()
True

>>> AG = designs.AffineGeometryDesign(Integer(3),Integer(1),GF(Integer(2)))  ␣
↪                       # needs sage.combinat
>>> AG.is_resolvable()                                                      #␣
↪needs sage.combinat
True
```

Their classes:

```
sage: b, cls = TD.is_resolvable(True)
sage: b
True
sage: cls # random
[[[0, 3], [1, 2]], [[1, 3], [0, 2]]]

sage: # needs sage.combinat
sage: b, cls = AG.is_resolvable(True)
sage: b
True
sage: cls  # random
[[[6, 7], [4, 5], [0, 1], [2, 3]],
 [[5, 7], [0, 4], [3, 6], [1, 2]],
 [[0, 2], [4, 7], [1, 3], [5, 6]],
 [[3, 4], [0, 7], [1, 5], [2, 6]],
 [[3, 7], [1, 6], [0, 5], [2, 4]],
 [[0, 6], [2, 7], [1, 4], [3, 5]],
 [[4, 6], [0, 3], [2, 5], [1, 7]]]
```

```
>>> from sage.all import *
>>> b, cls = TD.is_resolvable(True)
>>> b
True
>>> cls # random
[[[0, 3], [1, 2]], [[1, 3], [0, 2]]]

>>> # needs sage.combinat
>>> b, cls = AG.is_resolvable(True)
>>> b
```

(continues on next page)

(continued from previous page)

```
True
>>> cls  # random
[[[6, 7], [4, 5], [0, 1], [2, 3]],
 [[5, 7], [0, 4], [3, 6], [1, 2]],
 [[0, 2], [4, 7], [1, 3], [5, 6]],
 [[3, 4], [0, 7], [1, 5], [2, 6]],
 [[3, 7], [1, 6], [0, 5], [2, 4]],
 [[0, 6], [2, 7], [1, 4], [3, 5]],
 [[4, 6], [0, 3], [2, 5], [1, 7]]]
```

A non-resolvable design:

```
sage: Fano = designs.balanced_incomplete_block_design(7,3)                      #␣
↪needs sage.schemes
sage: Fano.is_resolvable()                                                       #␣
↪needs sage.schemes
False
sage: Fano.is_resolvable(True)                                                   #␣
↪needs sage.schemes
(False, [])
```

```
>>> from sage.all import *
>>> Fano = designs.balanced_incomplete_block_design(Integer(7),Integer(3))     ␣
↪              # needs sage.schemes
>>> Fano.is_resolvable()                                                       #␣
↪needs sage.schemes
False
>>> Fano.is_resolvable(True)                                                   #␣
↪needs sage.schemes
(False, [])
```

is_simple()

Test whether this design is simple (i.e. no repeated block).

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[1,2],[0,2]]).is_simple()
True
sage: IncidenceStructure(3, [[0],[0]]).is_simple()
False

sage: V = [(0,'a'),(0,'b'),(1,'a'),(1,'b')]
sage: B = [[V[0],V[1]], [V[1],V[2]]]
sage: I = IncidenceStructure(V, B)
sage: I.is_simple()
True
sage: I2 = IncidenceStructure(V, B*2)
sage: I2.is_simple()
False
```

```
>>> from sage.all import *
>>> IncidenceStructure(Integer(3), [[Integer(0),Integer(1)],[Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)],[Integer(0),Integer(2)]]).is_simple()
True
>>> IncidenceStructure(Integer(3), [[Integer(0)],[Integer(0)]]).is_simple()
False

>>> V = [(Integer(0),'a'),(Integer(0),'b'),(Integer(1),'a'),(Integer(1),'b')]
>>> B = [[V[Integer(0)],V[Integer(1)]], [V[Integer(1)],V[Integer(2)]]]
>>> I = IncidenceStructure(V, B)
>>> I.is_simple()
True
>>> I2 = IncidenceStructure(V, B*Integer(2))
>>> I2.is_simple()
False
```

is_spread(*spread*)

Check whether the input is a spread for `self`.

A spread of an incidence structure (P, B) is a subset of B which forms a partition of P.

INPUT:

- `spread` – iterable; defines the spread

EXAMPLES:

```
sage: E = IncidenceStructure([[1, 2, 3], [4, 5, 6], [1, 5, 6]])
sage: E.is_spread([[1, 2, 3], [4, 5, 6]])
True
sage: E.is_spread([1, 2, 3, 4, 5, 6])
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
sage: E.is_spread([[1, 2, 3, 4], [5, 6]])
False
```

```
>>> from sage.all import *
>>> E = IncidenceStructure([[Integer(1), Integer(2), Integer(3)], [Integer(4),
↪ Integer(5), Integer(6)], [Integer(1), Integer(5), Integer(6)]])
>>> E.is_spread([[Integer(1), Integer(2), Integer(3)], [Integer(4),␣
↪Integer(5), Integer(6)]])
True
>>> E.is_spread([Integer(1), Integer(2), Integer(3), Integer(4), Integer(5),␣
↪Integer(6)])
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
>>> E.is_spread([[Integer(1), Integer(2), Integer(3), Integer(4)],␣
↪[Integer(5), Integer(6)]])
False
```

Order of blocks or of points within each block doesn't matter:

```
sage: E = IncidenceStructure([[1, 2, 3], [4, 5, 6], [1, 5, 6]])
sage: E.is_spread([[5, 6, 4], [3, 1, 2]])
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> E = IncidenceStructure([[Integer(1), Integer(2), Integer(3)], [Integer(4),
↪ Integer(5), Integer(6)], [Integer(1), Integer(5), Integer(6)]])
>>> E.is_spread([[Integer(5), Integer(6), Integer(4)], [Integer(3),␣
↪Integer(1), Integer(2)]])
True
```

is_t_design(*t=None*, *v=None*, *k=None*, *l=None*, *return_parameters=False*)

Test whether `self` is a $t-(v,k,l)$ design.

A $t-(v,k,\lambda)$ (sometimes called t-design for short) is a block design in which:

- the underlying set has cardinality v
- the blocks have size k
- each t-subset of points is covered by λ blocks

INPUT:

- `t`, `v`, `k`, `l` – integers; their value is set to `None` by default. The function tests whether the design is a $t-(v,k,l)$ design using the provided values and guesses the others. Note that `l` cannot be specified if `t` is not.
- `return_parameters` – boolean; whether to return the parameters of the t-design. If set to `True`, the function returns a pair `(boolean_answer, (t,v,k,l))`.

EXAMPLES:

```
sage: fano_blocks = [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]]
sage: BD = IncidenceStructure(7, fano_blocks)
sage: BD.is_t_design()
True
sage: BD.is_t_design(return_parameters=True)
(True, (2, 7, 3, 1))
sage: BD.is_t_design(2, 7, 3, 1)
True
sage: BD.is_t_design(1, 7, 3, 3)
True
sage: BD.is_t_design(0, 7, 3, 7)
True

sage: BD.is_t_design(0,6,3,7) or BD.is_t_design(0,7,4,7) or BD.is_t_design(0,
↪7,3,8)
False

sage: BD = designs.AffineGeometryDesign(3, 1, GF(2))                        #␣
↪needs sage.combinat
sage: BD.is_t_design(1)                                                     #␣
↪needs sage.combinat
True
sage: BD.is_t_design(2)                                                     #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> fano_blocks = [[Integer(0),Integer(1),Integer(2)],[Integer(0),Integer(3),
↪Integer(4)],[Integer(0),Integer(5),Integer(6)],[Integer(1),Integer(3),
↪Integer(5)],[Integer(1),Integer(4),Integer(6)],[Integer(2),Integer(3),
↪Integer(6)],[Integer(2),Integer(4),Integer(5)]]
>>> BD = IncidenceStructure(Integer(7), fano_blocks)
>>> BD.is_t_design()
True
>>> BD.is_t_design(return_parameters=True)
(True, (2, 7, 3, 1))
>>> BD.is_t_design(Integer(2), Integer(7), Integer(3), Integer(1))
True
>>> BD.is_t_design(Integer(1), Integer(7), Integer(3), Integer(3))
True
>>> BD.is_t_design(Integer(0), Integer(7), Integer(3), Integer(7))
True

>>> BD.is_t_design(Integer(0),Integer(6),Integer(3),Integer(7)) or BD.is_t_
↪design(Integer(0),Integer(7),Integer(4),Integer(7)) or BD.is_t_
↪design(Integer(0),Integer(7),Integer(3),Integer(8))
False

>>> BD = designs.AffineGeometryDesign(Integer(3), Integer(1), GF(Integer(2)))␣
↪                         # needs sage.combinat
>>> BD.is_t_design(Integer(1))                                               ␣
↪     # needs sage.combinat
True
>>> BD.is_t_design(Integer(2))                                               ␣
↪     # needs sage.combinat
True
```

Steiner triple and quadruple systems are other names for $2-(v,3,1)$ and $3-(v,4,1)$ designs:

```
sage: S3_9 = designs.steiner_triple_system(9)
sage: S3_9.is_t_design(2,9,3,1)
True

sage: blocks = designs.steiner_quadruple_system(8)
sage: S4_8 = IncidenceStructure(8, blocks)
sage: S4_8.is_t_design(3,8,4,1)
True

sage: blocks = designs.steiner_quadruple_system(14)
sage: S4_14 = IncidenceStructure(14, blocks)
sage: S4_14.is_t_design(3,14,4,1)
True
```

```
>>> from sage.all import *
>>> S3_9 = designs.steiner_triple_system(Integer(9))
>>> S3_9.is_t_design(Integer(2),Integer(9),Integer(3),Integer(1))
True

>>> blocks = designs.steiner_quadruple_system(Integer(8))
```

(continues on next page)

(continued from previous page)

```
>>> S4_8 = IncidenceStructure(Integer(8), blocks)
>>> S4_8.is_t_design(Integer(3),Integer(8),Integer(4),Integer(1))
True

>>> blocks = designs.steiner_quadruple_system(Integer(14))
>>> S4_14 = IncidenceStructure(Integer(14), blocks)
>>> S4_14.is_t_design(Integer(3),Integer(14),Integer(4),Integer(1))
True
```

Some examples of Witt designs that need the gap database:

```
sage: # optional - gap_package_design
sage: BD = designs.WittDesign(9)
sage: BD.is_t_design(2,9,3,1)
True
sage: W12 = designs.WittDesign(12)
sage: W12.is_t_design(5,12,6,1)
True
sage: W12.is_t_design(4)
True
```

```
>>> from sage.all import *
>>> # optional - gap_package_design
>>> BD = designs.WittDesign(Integer(9))
>>> BD.is_t_design(Integer(2),Integer(9),Integer(3),Integer(1))
True
>>> W12 = designs.WittDesign(Integer(12))
>>> W12.is_t_design(Integer(5),Integer(12),Integer(6),Integer(1))
True
>>> W12.is_t_design(Integer(4))
True
```

Further examples:

```
sage: D = IncidenceStructure(4,[[],[]])
sage: D.is_t_design(return_parameters=True)
(True,  (0, 4, 0, 2))

sage: D = IncidenceStructure(4, [[0,1],[0,2],[0,3]])
sage: D.is_t_design(return_parameters=True)
(True, (0, 4, 2, 3))

sage: D = IncidenceStructure(4, [[0],[1],[2],[3]])
sage: D.is_t_design(return_parameters=True)
(True, (1, 4, 1, 1))

sage: D = IncidenceStructure(4,[[0,1],[2,3]])
sage: D.is_t_design(return_parameters=True)
(True, (1, 4, 2, 1))

sage: D = IncidenceStructure(4, [list(range(4))])
sage: D.is_t_design(return_parameters=True)
```

(continues on next page)

(continued from previous page)

```
(True, (4, 4, 4, 1))
```

```
>>> from sage.all import *
>>> D = IncidenceStructure(Integer(4),[[],[]])
>>> D.is_t_design(return_parameters=True)
(True,  (0, 4, 0, 2))

>>> D = IncidenceStructure(Integer(4), [[Integer(0),Integer(1)],[Integer(0),
↪Integer(2)],[Integer(0),Integer(3)]])
>>> D.is_t_design(return_parameters=True)
(True, (0, 4, 2, 3))

>>> D = IncidenceStructure(Integer(4), [[Integer(0)],[Integer(1)],
↪[Integer(2)],[Integer(3)]])
>>> D.is_t_design(return_parameters=True)
(True, (1, 4, 1, 1))

>>> D = IncidenceStructure(Integer(4),[[Integer(0),Integer(1)],[Integer(2),
↪Integer(3)]])
>>> D.is_t_design(return_parameters=True)
(True, (1, 4, 2, 1))

>>> D = IncidenceStructure(Integer(4), [list(range(Integer(4)))])
>>> D.is_t_design(return_parameters=True)
(True, (4, 4, 4, 1))
```

is_uniform(*k=None*)

Test whether the incidence structure is k-uniform

An incidence structure is said to be k-uniform if all its blocks have size k.

INPUT:

- `k` – integer

OUTPUT:

If `k` is defined, a boolean is returned. If `k` is set to `None` (default), the method returns either `False` or the integer `k` such that the incidence structure is k-uniform.

> **Warning**
>
> In case of 0-uniform incidence structure, beware that `if not H.is_uniform()` is a satisfied condition.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(7,3).is_uniform()              #␣
↪needs sage.schemes
3
sage: designs.balanced_incomplete_block_design(7,3).is_uniform(k=3)           #␣
↪needs sage.schemes
True
sage: designs.balanced_incomplete_block_design(7,3).is_uniform(k=4)           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.schemes
False
```

```
>>> from sage.all import *
>>> designs.balanced_incomplete_block_design(Integer(7),Integer(3)).is_
↪uniform()            # needs sage.schemes
3
>>> designs.balanced_incomplete_block_design(Integer(7),Integer(3)).is_
↪uniform(k=Integer(3))          # needs sage.schemes
True
>>> designs.balanced_incomplete_block_design(Integer(7),Integer(3)).is_
↪uniform(k=Integer(4))          # needs sage.schemes
False
```

isomorphic_substructures_iterator(*H2*, *induced=False*)

Iterate over all copies of `H2` contained in `self`.

A hypergraph H_1 contains an isomorphic copy of a hypergraph H_2 if there exists an injection $f : V(H_2) \mapsto V(H_1)$ such that for any set $S_2 \in E(H_2)$ the set $S_1 = f(S2)$ belongs to $E(H_1)$.

It is an *induced* copy if no other set of $E(H_1)$ is contained in $f(V(H_2))$, i.e. $|E(H_2)| = \{S : S \in E(H_1)$ and $f(V(H_2))\}$.

This function lists all such injections. In particular, the number of copies of H in itself is equal to *the size of its automorphism group*.

See *subhypergraph_search* for more information.

INPUT:

- `H2` – an *IncidenceStructure* object
- `induced` – boolean (default: `False`); whether to require the copies to be induced

EXAMPLES:

How many distinct C_5 in Petersen's graph ?

```
sage: P = graphs.PetersenGraph()
sage: C = graphs.CycleGraph(5)
sage: IP = IncidenceStructure(P.edges(sort=True, labels=False))
sage: IC = IncidenceStructure(C.edges(sort=True, labels=False))
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC))
120
```

```
>>> from sage.all import *
>>> P = graphs.PetersenGraph()
>>> C = graphs.CycleGraph(Integer(5))
>>> IP = IncidenceStructure(P.edges(sort=True, labels=False))
>>> IC = IncidenceStructure(C.edges(sort=True, labels=False))
>>> sum(Integer(1) for _ in IP.isomorphic_substructures_iterator(IC))
120
```

As the automorphism group of C_5 has size 10, the number of distinct unlabelled copies is 12. Let us check that all functions returned correspond to an actual C_5 subgraph:

```
sage: for f in IP.isomorphic_substructures_iterator(IC):
....:     assert all(P.has_edge(f[x],f[y]) for x,y in C.edges(sort=True,␣
↪labels=False))
```

```
>>> from sage.all import *
>>> for f in IP.isomorphic_substructures_iterator(IC):
...     assert all(P.has_edge(f[x],f[y]) for x,y in C.edges(sort=True,␣
↪labels=False))
```

The number of induced copies, in this case, is the same:

```
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC,induced=True))
120
```

```
>>> from sage.all import *
>>> sum(Integer(1) for _ in IP.isomorphic_substructures_iterator(IC,
↪induced=True))
120
```

They begin to differ if we make one vertex universal:

```
sage: P.add_edges([(0,x) for x in P], loops=False)
sage: IP = IncidenceStructure(P.edges(sort=True, labels=False))
sage: IC = IncidenceStructure(C.edges(sort=True, labels=False))
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC))
420
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC,induced=True))
60
```

```
>>> from sage.all import *
>>> P.add_edges([(Integer(0),x) for x in P], loops=False)
>>> IP = IncidenceStructure(P.edges(sort=True, labels=False))
>>> IC = IncidenceStructure(C.edges(sort=True, labels=False))
>>> sum(Integer(1) for _ in IP.isomorphic_substructures_iterator(IC))
420
>>> sum(Integer(1) for _ in IP.isomorphic_substructures_iterator(IC,
↪induced=True))
60
```

The number of copies of H in itself is the size of its automorphism group:

```
sage: H = designs.projective_plane(3)                                   #␣
↪needs sage.schemes
sage: sum(1 for _ in H.isomorphic_substructures_iterator(H))            #␣
↪needs sage.schemes
5616
sage: H.automorphism_group().cardinality()                              #␣
↪needs sage.groups sage.schemes
5616
```

```
>>> from sage.all import *
>>> H = designs.projective_plane(Integer(3))                            ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.schemes
>>> sum(Integer(1) for _ in H.isomorphic_substructures_iterator(H))          ␣
↪     # needs sage.schemes
5616
>>> H.automorphism_group().cardinality()                                    #␣
↪needs sage.groups sage.schemes
5616
```

num_blocks()

Return the number of blocks.

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2).num_blocks()
7
sage: B = IncidenceStructure(4, [[0,1],[0,2],[0,3],[1,2], [1,2,3]])
sage: B.num_blocks()
5
```

```
>>> from sage.all import *
>>> designs.DesarguesianProjectivePlaneDesign(Integer(2)).num_blocks()
7
>>> B = IncidenceStructure(Integer(4), [[Integer(0),Integer(1)],[Integer(0),
↪Integer(2)],[Integer(0),Integer(3)],[Integer(1),Integer(2)], [Integer(1),
↪Integer(2),Integer(3)]])
>>> B.num_blocks()
5
```

num_points()

Return the size of the ground set.

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2).num_points()
7
sage: B = IncidenceStructure(4, [[0,1],[0,2],[0,3],[1,2], [1,2,3]])
sage: B.num_points()
4
```

```
>>> from sage.all import *
>>> designs.DesarguesianProjectivePlaneDesign(Integer(2)).num_points()
7
>>> B = IncidenceStructure(Integer(4), [[Integer(0),Integer(1)],[Integer(0),
↪Integer(2)],[Integer(0),Integer(3)],[Integer(1),Integer(2)], [Integer(1),
↪Integer(2),Integer(3)]])
>>> B.num_points()
4
```

packing(*solver*, *verbose=None*, *integrality_tolerance=0*)

Return a maximum packing.

A maximum packing in a hypergraph is collection of disjoint sets/blocks of maximal cardinality. This problem is NP-complete in general, and in particular on 3-uniform hypergraphs. It is solved here with an Integer Linear Program.

For more information, see the Wikipedia article Packing_in_a_hypergraph.

INPUT:

- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

EXAMPLES:

```
sage: P = IncidenceStructure([[1,2],[3,4],[2,3]]).packing()                    #␣
↪needs sage.numerical.mip
sage: sorted(sorted(b) for b in P)                                             #␣
↪needs sage.numerical.mip
[[1, 2], [3, 4]]
sage: len(designs.steiner_triple_system(9).packing())                          #␣
↪needs sage.numerical.mip
3
```

```
>>> from sage.all import *
>>> P = IncidenceStructure([[Integer(1),Integer(2)],[Integer(3),Integer(4)],
↪[Integer(2),Integer(3)]]).packing()                    # needs sage.numerical.
↪mip
>>> sorted(sorted(b) for b in P)                                             #␣
↪needs sage.numerical.mip
[[1, 2], [3, 4]]
>>> len(designs.steiner_triple_system(Integer(9)).packing())                   ␣
↪      # needs sage.numerical.mip
3
```

rank()

Return the rank of the hypergraph (the maximum size of a block).

EXAMPLES:

```
sage: h = Hypergraph(8, [[0,1,3],[1,4,5,6],[1,2]])
sage: h.rank()
4
```

```
>>> from sage.all import *
>>> h = Hypergraph(Integer(8), [[Integer(0),Integer(1),Integer(3)],
↪[Integer(1),Integer(4),Integer(5),Integer(6)],[Integer(1),Integer(2)]])
>>> h.rank()
4
```

relabel(*perm=None*, *inplace=True*)

Relabel the ground set.

INPUT:

- `perm` – can be one of
  - a dictionary – then each point `p` (which should be a key of `d`) is relabeled to `d[p]`

  – a list or a tuple of length n – the first point returned by `ground_set()` is relabeled to l[0], the second to l[1], ...
  – None – the incidence structure is relabeled to be on $\{0, 1, ..., n - 1\}$ in the ordering given by `ground_set()`

- inplace – boolean (default: False); if True then return a relabeled graph and does not touch self

EXAMPLES:

```
sage: # needs sage.schemes
sage: TD = designs.transversal_design(5,5)
sage: TD.relabel({i: chr(97+i) for i in range(25)})
sage: TD.ground_set()
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y']
sage: TD.blocks()[:3]
[['a', 'f', 'k', 'p', 'u'], ['a', 'g', 'm', 's', 'y'], ['a', 'h', 'o', 'q', 'x
↪']]
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> TD = designs.transversal_design(Integer(5),Integer(5))
>>> TD.relabel({i: chr(Integer(97)+i) for i in range(Integer(25))})
>>> TD.ground_set()
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y']
>>> TD.blocks()[:Integer(3)]
[['a', 'f', 'k', 'p', 'u'], ['a', 'g', 'm', 's', 'y'], ['a', 'h', 'o', 'q', 'x
↪']]
```

Relabel to integer points:

```
sage: TD.relabel()                                                      #␣
↪needs sage.schemes
sage: TD.blocks()[:3]                                                   #␣
↪needs sage.schemes
[[0, 5, 10, 15, 20], [0, 6, 12, 18, 24], [0, 7, 14, 16, 23]]
```

```
>>> from sage.all import *
>>> TD.relabel()                                                      #␣
↪needs sage.schemes
>>> TD.blocks()[:Integer(3)]                                            ␣
↪     # needs sage.schemes
[[0, 5, 10, 15, 20], [0, 6, 12, 18, 24], [0, 7, 14, 16, 23]]
```

trace(*points*, *min_size=1*, *multiset=True*)

Return the trace of a set of points.

Given an hypergraph $\mathcal{H}$, the *trace* of a set X of points in $\mathcal{H}$ is the hypergraph whose blocks are all non-empty $S \cap X$ where $S \in \mathcal{H}$.

INPUT:

- points – set of points

- `min_size` – integer (default: 1); minimum size of the sets to keep. By default all empty sets are discarded, i.e. `min_size=1`
- `multiset` – boolean (default: `True`); whether to keep multiple copies of the same set

Note: This method goes over all sets of `self` before building a new `IncidenceStructure` (which involves some relabelling and sorting). It probably should not be called in a performance-critical code.

EXAMPLES:

A Baer subplane of order 2 (i.e. a Fano plane) in a projective plane of order 4:

```
sage: # needs sage.schemes
sage: P4 = designs.projective_plane(4)
sage: F = designs.projective_plane(2)
sage: for x in Subsets(P4.ground_set(),7):
....:     if P4.trace(x,min_size=2).is_isomorphic(F):
....:         break
sage: subplane = P4.trace(x,min_size=2); subplane
Incidence structure with 7 points and 7 blocks
sage: subplane.is_isomorphic(F)
True
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> P4 = designs.projective_plane(Integer(4))
>>> F = designs.projective_plane(Integer(2))
>>> for x in Subsets(P4.ground_set(),Integer(7)):
...     if P4.trace(x,min_size=Integer(2)).is_isomorphic(F):
...         break
>>> subplane = P4.trace(x,min_size=Integer(2)); subplane
Incidence structure with 7 points and 7 blocks
>>> subplane.is_isomorphic(F)
True
```

5.1.90 Mutually Orthogonal Latin Squares (MOLS)

The main function of this module is `mutually_orthogonal_latin_squares()` and can be can be used to generate MOLS (or check that they exist):

```
sage: MOLS = designs.mutually_orthogonal_latin_squares(4,8)                  #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> MOLS = designs.mutually_orthogonal_latin_squares(Integer(4),Integer(8))         ␣
↪            # needs sage.schemes
```

For more information on MOLS, see the Wikipedia entry on MOLS. If you are only interested by latin squares, see `latin`.

The functions defined here are

mutually_orthogonal_latin_squares()	Return k Mutually Orthogonal $n \times n$ Latin Squares.
are_mutually_orthogonal_latin_squares()	Check that the list `l` of matrices in are MOLS.
latin_square_product()	Return the product of two (or more) latin squares.
MOLS_table()	Print the MOLS table.

Table of MOLS

Sage can produce a table of MOLS similar to the one from the Handbook of Combinatorial Designs [DesignHandbook] (available here).

```
sage: from sage.combinat.designs.latin_squares import MOLS_table
sage: MOLS_table(600) # long time
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ________________________________________________________________________________
  0| +oo +oo   1   2   3   4   1   6   7   8   2  10   5  12   4   4  15  16   5  18
 20|   4   5   3  22   7  24   4  26   5  28   4  30  31   5   4   5   8  36   4   5
 40|   7  40   5  42   5   6   4  46   8  48   6   5   5  52   5   6   7   7   5  58
 60|   5  60   5   6  63   7   5  66   5   6   6  70   7  72   5   7   6   6   6  78
 80|   9  80   8  82   6   6   6   6   7  88   6   7   6   6   6   6   7  96   6   8
100|   8 100   6 102   7   7   6 106   6 108   6   6  13 112   6   7   6   8   6   6
120|   7 120   6   6   6 124   6 126 127   7   6 130   6   7   6   7   7 136   6 138
140|   6   7   6  10  10   7   6   7   6 148   6 150   7   8   8   7   6 156   7   6
160|   9   7   6 162   6   7   6 166   7 168   6   8   6 172   6   6  14   9   6 178
180|   6 180   6   6   7   9   6  10   6   8   6 190   7 192   6   7   6 196   6 198
200|   7   7   6   7   6   8   6   8  14  11  10 210   6   7   6   7   7   8   6  10
220|   6  12   6 222  13   8   6 226   6 228   6   7   7 232   6   7   6   7   6 238
240|   7 240   6 242   6   7   6  12   7   7   6 250   6  12   9   7 255 256   6  12
260|   6   8   8 262   7   8   7  10   7 268   7 270  15  16   6  13  10 276   6   9
280|   7 280   6 282   6  12   6   7  15 288   6   6   6 292   6   6   7  10  10  12
300|   7   7   7   7  15  15   6 306   7   7   7 310   7 312   7  10   7 316   7  10
320|  15  15   6  16   8  12   6   7   7   9   6 330   7   8   7   6   7 336   6   7
340|   6  10  10 342   7   7   6 346   6 348   8  12  18 352   6   9   7   9   6 358
360|   7 360   6   7   7   7   6 366  15  15   7  15   7 372   7  15   7  13   7 378
380|   7  12   7 382  15  15   7  15   7 388   7  16   7   7   7   7   8 396   7   7
400|  15 400   7  15  11   8   7  15   8 408   7  13   8  12  10   9  18  15   7 418
420|   7 420   7  15   7  16   6   7   7   7   6 430  15 432   6  15   6  18   7 438
440|   7  15   7 442   7  13   7  11  15 448   7  15   7   7   7  15   7 456   7  16
460|   7 460   7 462  15  15   7 466   8   8   7  15   7  15  10  18   7  15   6 478
480|  15  15   6  15   8   7   6 486   7  15   6 490   6  16   6   7  15  15   6 498
500|   7   8   9 502   7  15   6  15   7 508   6  15 511  18   7  15   8  12   8  15
520|   8 520  10 522  12  15   8  16  15 528   7  15   8  12   7  15   8  15  10  15
540|  12 540   7  15  18   7   7 546   7   8   7  18   7   7   7   7   7 556   7  12
560|  15   7   7 562   7   7   6   7   7 568   6 570   7   7  15  22   8 576   7   7
580|   7   8   7  10   7   8   7 586   7  18  17   7  15 592   8  15   7   7   8 598
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.latin_squares import MOLS_table
>>> MOLS_table(Integer(600)) # long time
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ________________________________________________________________________________
```

(continues on next page)

(continued from previous page)

```
  0| +oo +oo   1   2   3   4   1   6   7   8   2  10   5  12   4   4  15  16   5  18
 20|   4   5   3  22   7  24   4  26   5  28   4  30  31   5   4   5   8  36   4   5
 40|   7  40   5  42   5   6   4  46   8  48   6   5   5  52   5   6   7   7   5  58
 60|   5  60   5   6  63   7   5  66   5   6   6  70   7  72   5   7   6   6   6  78
 80|   9  80   8  82   6   6   6   6   7  88   6   7   6   6   6   6   7  96   6   8
100|   8 100   6 102   7   7   6 106   6 108   6   6  13 112   6   7   6   8   6   6
120|   7 120   6   6   6 124   6 126 127   7   6 130   6   7   6   7   7 136   6 138
140|   6   7   6  10  10   7   6   7   6 148   6 150   7   8   8   7   6 156   7   6
160|   9   7   6 162   6   7   6 166   7 168   6   8   6 172   6   6  14   9   6 178
180|   6 180   6   6   7   9   6  10   6   8   6 190   7 192   6   7   6 196   6 198
200|   7   7   6   7   6   8   6   8  14  11  10 210   6   7   6   7   7   8   6  10
220|   6  12   6 222  13   8   6 226   6 228   6   7   7 232   6   7   6   7   6 238
240|   7 240   6 242   6   7   6  12   7   7   6 250   6  12   9   7 255 256   6  12
260|   6   8   8 262   7   8   7  10   7 268   7 270  15  16   6  13  10 276   6   9
280|   7 280   6 282   6  12   6   7  15 288   6   6   6 292   6   6   7  10  10  12
300|   7   7   7   7  15  15   6 306   7   7   7 310   7 312   7  10   7 316   7  10
320|  15  15   6  16   8  12   6   7   7   9   6 330   7   8   7   6   7 336   6   7
340|   6  10  10 342   7   7   6 346   6 348   8  12  18 352   6   9   7   9   6 358
360|   7 360   6   7   7   7   6 366  15  15   7  15   7 372   7  15   7  13   7 378
380|   7  12   7 382  15  15   7  15   7 388   7  16   7   7   7   7   8 396   7   7
400|  15 400   7  15  11   8   7  15   8 408   7  13   8  12  10   9  18  15   7 418
420|   7 420   7  15   7  16   6   7   7   7   6 430  15 432   6  15   6  18   7 438
440|   7  15   7 442   7  13   7  11  15 448   7  15   7   7   7  15   7 456   7  16
460|   7 460   7 462  15  15   7 466   8   8   7  15   7  15  10  18   7  15   6 478
480|  15  15   6  15   8   7   6 486   7  15   6 490   6  16   6   7  15  15   6 498
500|   7   8   9 502   7  15   6  15   7 508   6  15 511  18   7  15   8  12   8  15
520|   8 520  10 522  12  15   8  16  15 528   7  15   8  12   7  15   8  15  10  15
540|  12 540   7  15  18   7   7 546   7   8   7  18   7   7   7   7   7 556   7  12
560|  15   7   7 562   7   7   6   7   7 568   6 570   7   7  15  22   8 576   7   7
580|   7   8   7  10   7   8   7 586   7  18  17   7  15 592   8  15   7   7   8 598
```

Comparison with the results from the Handbook of Combinatorial Designs (2ed) [DesignHandbook]:

```
sage: MOLS_table(600,compare=True) # long time
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ________________________________________________________________________________
  0|                                                       +               +
 20|
 40|
 60|
 80|
100|
120|
140|
160|
180|
200|       -
220|
240|
260|
280|
300|
```

(continues on next page)

(continued from previous page)

```
320|                                                                   -
340|
360|   -                   -
380|                                                       -
400|
420|                                       -
440|
460|
480|
500|       -
520|
540|
560|
580|
```

```
>>> from sage.all import *
>>> MOLS_table(Integer(600),compare=True) # long time
        0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    _________________________________________________________________________________
  0|                                                           +               +
 20|
 40|
 60|
 80|
100|
120|
140|
160|
180|
200|       -
220|
240|
260|
280|
300|
320|                                                                   -
340|
360|   -                   -
380|                                                       -
400|
420|                                       -
440|
460|
480|
500|       -
520|
540|
560|
580|
```

Todo

Look at [ColDin01].

REFERENCES:

Functions

`sage.combinat.designs.latin_squares.MOLS_table(start, stop=None, compare=False, width=None)`

Print the MOLS table that Sage can produce.

INPUT:

- `start`, `stop` – integers; print the table of MOLS for value of n such that `start<=n<stop`. If only one integer is given as input, it is interpreted as the value of `stop` with `start=0` (same behaviour as `range`).
- `compare` – boolean; if sets to `True` the MOLS displays with $+$ and $-$ entries its difference with the table from the Handbook of Combinatorial Designs (2ed).
- `width` – integer; the width of each column of the table. By default, it is computed from range of values determined by the parameters `start` and `stop`.

EXAMPLES:

```
sage: # needs sage.schemes
sage: from sage.combinat.designs.latin_squares import MOLS_table
sage: MOLS_table(100)
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18 ␣
↪19
    ________________________________________________________________________________
↪__
  0| +oo +oo   1   2   3   4   1   6   7   8   2  10   5  12   4   4  15  16   5 ␣
↪18
 20|   4   5   3  22   7  24   4  26   5  28   4  30  31   5   4   5   8  36   4 ␣
↪ 5
 40|   7  40   5  42   5   6   4  46   8  48   6   5   5  52   5   6   7   7   5 ␣
↪58
 60|   5  60   5   6  63   7   5  66   5   6   6  70   7  72   5   7   6   6   6 ␣
↪78
 80|   9  80   8  82   6   6   6   6   7  88   6   7   6   6   6   6   7  96   6 ␣
↪ 8
sage: MOLS_table(100, width=4)
        0    1    2    3    4    5    6    7    8    9   10   11   12   13   14 ␣
↪ 15   16   17   18   19
     _______________________________________________________________________________
↪_________________________
   0|  +oo  +oo    1    2    3    4    1    6    7    8    2   10    5   12    4 ␣
↪  4   15   16    5   18
  20|    4    5    3   22    7   24    4   26    5   28    4   30   31    5    4 ␣
↪  5    8   36    4    5
  40|    7   40    5   42    5    6    4   46    8   48    6    5    5   52    5 ␣
↪  6    7    7    5   58
  60|    5   60    5    6   63    7    5   66    5    6    6   70    7   72    5 ␣
↪  7    6    6    6   78
  80|    9   80    8   82    6    6    6    6    7   88    6    7    6    6    6 ␣
```

(continues on next page)

(continued from previous page)

```
↪   6   7  96   6   8
sage: MOLS_table(100, compare=True)
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
↪19
    ________________________________________________________________________________
↪__
  0|                                                           +               +
 20|
 40|
 60|
 80|
sage: MOLS_table(50, 100, compare=True)
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
↪19
    ________________________________________________________________________________
↪__
 40|
 60|
 80|
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> from sage.combinat.designs.latin_squares import MOLS_table
>>> MOLS_table(Integer(100))
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
↪19
    ________________________________________________________________________________
↪__
  0| +oo +oo   1   2   3   4   1   6   7   8   2  10   5  12   4   4  15  16   5
↪18
 20|   4   5   3  22   7  24   4  26   5  28   4  30  31   5   4   5   8  36   4
↪ 5
 40|   7  40   5  42   5   6   4  46   8  48   6   5   5  52   5   6   7   7   5
↪58
 60|   5  60   5   6  63   7   5  66   5   6   6  70   7  72   5   7   6   6   6
↪78
 80|   9  80   8  82   6   6   6   6   7  88   6   7   6   6   6   6   7  96   6
↪ 8
>>> MOLS_table(Integer(100), width=Integer(4))
        0    1    2    3    4    5    6    7    8    9   10   11   12   13   14
↪ 15   16   17   18   19
     _____________________________________________________________________________
↪____________________
   0|  +oo  +oo    1    2    3    4    1    6    7    8    2   10    5   12    4
↪  4   15   16    5   18
  20|    4    5    3   22    7   24    4   26    5   28    4   30   31    5    4
↪  5    8   36    4    5
  40|    7   40    5   42    5    6    4   46    8   48    6    5    5   52    5
↪  6    7    7    5   58
  60|    5   60    5    6   63    7    5   66    5    6    6   70    7   72    5
↪  7    6    6    6   78
  80|    9   80    8   82    6    6    6    6    7   88    6    7    6    6    6
```

(continues on next page)

(continued from previous page)

```
↪  6    7   96    6    8
>>> MOLS_table(Integer(100), compare=True)
       0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18 ␣
↪19
    ______________________________________________________________________________________________
↪__
  0|                                                                   +                 +
 20|
 40|
 60|
 80|
>>> MOLS_table(Integer(50), Integer(100), compare=True)
       0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18 ␣
↪19
    ______________________________________________________________________________________________
↪__
 40|
 60|
 80|
```

sage.combinat.designs.latin_squares.**are_mutually_orthogonal_latin_squares**(*l*, *verbose=False*)

Check whether the list of matrices in `l` form mutually orthogonal latin squares.

INPUT:

- `verbose` – if `True` then print why the list of matrices provided are not mutually orthogonal latin squares

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_
↪latin_squares
sage: m1 = matrix([[0,1,2],[2,0,1],[1,2,0]])
sage: m2 = matrix([[0,1,2],[1,2,0],[2,0,1]])
sage: m3 = matrix([[0,1,2],[2,0,1],[1,2,0]])
sage: are_mutually_orthogonal_latin_squares([m1,m2])
True
sage: are_mutually_orthogonal_latin_squares([m1,m3])
False
sage: are_mutually_orthogonal_latin_squares([m2,m3])
True
sage: are_mutually_orthogonal_latin_squares([m1,m2,m3], verbose=True)
Squares 0 and 2 are not orthogonal
False

sage: m = designs.mutually_orthogonal_latin_squares(7,8)                        #␣
↪needs sage.schemes
sage: are_mutually_orthogonal_latin_squares(m)                                  #␣
↪needs sage.schemes
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_
↪squares
```

(continues on next page)

(continued from previous page)

```
>>> m1 = matrix([[Integer(0),Integer(1),Integer(2)],[Integer(2),Integer(0),
↪Integer(1)],[Integer(1),Integer(2),Integer(0)]])
>>> m2 = matrix([[Integer(0),Integer(1),Integer(2)],[Integer(1),Integer(2),
↪Integer(0)],[Integer(2),Integer(0),Integer(1)]])
>>> m3 = matrix([[Integer(0),Integer(1),Integer(2)],[Integer(2),Integer(0),
↪Integer(1)],[Integer(1),Integer(2),Integer(0)]])
>>> are_mutually_orthogonal_latin_squares([m1,m2])
True
>>> are_mutually_orthogonal_latin_squares([m1,m3])
False
>>> are_mutually_orthogonal_latin_squares([m2,m3])
True
>>> are_mutually_orthogonal_latin_squares([m1,m2,m3], verbose=True)
Squares 0 and 2 are not orthogonal
False

>>> m = designs.mutually_orthogonal_latin_squares(Integer(7),Integer(8))
↪               # needs sage.schemes
>>> are_mutually_orthogonal_latin_squares(m)                                #
↪needs sage.schemes
True
```

sage.combinat.designs.latin_squares.**latin_square_product**(*M*, *N*, **others*)

Return the product of two (or more) latin squares.

Given two Latin Squares M, N of respective sizes m, n, the direct product $M \times N$ of size mn is defined by $(M \times N)((i_1, i_2), (j_1, j_2)) = (M(i_1, j_1), N(i_2, j_2))$ where $i_1, j_1 \in [m], i_2, j_2 \in [n]$

Each pair of values $(i, j) \in [m] \times [n]$ is then relabeled to $in + j$.

This is Lemma 6.25 of [Stinson2004].

INPUT:

- `M`, `N`, `*others` – an arbitrary number of latin squares (greater than or equal to 2)

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import latin_square_product
sage: m=designs.mutually_orthogonal_latin_squares(3,4)[0]                     #
↪needs sage.schemes
sage: latin_square_product(m,m,m)                                             #
↪needs sage.schemes
64 x 64 sparse matrix over Integer Ring (use the '.str()' method to see the
↪entries)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.latin_squares import latin_square_product
>>> m=designs.mutually_orthogonal_latin_squares(Integer(3),
↪Integer(4))[Integer(0)]                         # needs sage.schemes
>>> latin_square_product(m,m,m)                                             #
↪needs sage.schemes
64 x 64 sparse matrix over Integer Ring (use the '.str()' method to see the
↪entries)
```

sage.combinat.designs.latin_squares.**mutually_orthogonal_latin_squares**(*k*, *n*, *partitions=False*, *check=True*)

Return k Mutually Orthogonal $n \times n$ Latin Squares (MOLS).

For more information on Mutually Orthogonal Latin Squares, see `latin_squares`.

INPUT:

- `k` – integer; number of MOLS. If `k` is `None` it is set to the largest value available
- `n` – integer; size of the latin square
- `partitions` – boolean; a Latin Square can be seen as 3 partitions of the n^2 cells of the array into n sets of size n, respectively:
  - The partition of rows
  - The partition of columns
  - The partition of number (cells numbered with 0, cells numbered with 1, …)

  These partitions have the additional property that any two sets from different partitions intersect on exactly one element.

  When `partitions` is set to `True`, this function returns a list of $k + 2$ partitions satisfying this intersection property instead of the $k + 2$ MOLS (though the data is exactly the same in both cases).
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

EXAMPLES:

```
sage: designs.mutually_orthogonal_latin_squares(4,5)                        #␣
↪needs sage.schemes
[
[0 1 2 3 4]  [0 1 2 3 4]  [0 1 2 3 4]  [0 1 2 3 4]
[1 2 3 4 0]  [2 3 4 0 1]  [3 4 0 1 2]  [4 0 1 2 3]
[2 3 4 0 1]  [4 0 1 2 3]  [1 2 3 4 0]  [3 4 0 1 2]
[3 4 0 1 2]  [1 2 3 4 0]  [4 0 1 2 3]  [2 3 4 0 1]
[4 0 1 2 3], [3 4 0 1 2], [2 3 4 0 1], [1 2 3 4 0]
]

sage: designs.mutually_orthogonal_latin_squares(3,7)                        #␣
↪needs sage.schemes
[
[0 1 2 3 4 5 6]  [0 1 2 3 4 5 6]  [0 1 2 3 4 5 6]
[1 2 3 4 5 6 0]  [2 3 4 5 6 0 1]  [3 4 5 6 0 1 2]
[2 3 4 5 6 0 1]  [4 5 6 0 1 2 3]  [6 0 1 2 3 4 5]
[3 4 5 6 0 1 2]  [6 0 1 2 3 4 5]  [2 3 4 5 6 0 1]
[4 5 6 0 1 2 3]  [1 2 3 4 5 6 0]  [5 6 0 1 2 3 4]
[5 6 0 1 2 3 4]  [3 4 5 6 0 1 2]  [1 2 3 4 5 6 0]
[6 0 1 2 3 4 5], [5 6 0 1 2 3 4], [4 5 6 0 1 2 3]
]

sage: designs.mutually_orthogonal_latin_squares(2,5,partitions=True)        #␣
↪needs sage.schemes
[[[0, 1, 2, 3, 4],
  [5, 6, 7, 8, 9],
  [10, 11, 12, 13, 14],
```

(continues on next page)

(continued from previous page)

```
  [15, 16, 17, 18, 19],
  [20, 21, 22, 23, 24]],
 [[0, 5, 10, 15, 20],
  [1, 6, 11, 16, 21],
  [2, 7, 12, 17, 22],
  [3, 8, 13, 18, 23],
  [4, 9, 14, 19, 24]],
 [[0, 9, 13, 17, 21],
  [1, 5, 14, 18, 22],
  [2, 6, 10, 19, 23],
  [3, 7, 11, 15, 24],
  [4, 8, 12, 16, 20]],
 [[0, 8, 11, 19, 22],
  [1, 9, 12, 15, 23],
  [2, 5, 13, 16, 24],
  [3, 6, 14, 17, 20],
  [4, 7, 10, 18, 21]]]
```

```
>>> from sage.all import *
>>> designs.mutually_orthogonal_latin_squares(Integer(4),Integer(5))
↪               # needs sage.schemes
[
[0 1 2 3 4]  [0 1 2 3 4]  [0 1 2 3 4]  [0 1 2 3 4]
[1 2 3 4 0]  [2 3 4 0 1]  [3 4 0 1 2]  [4 0 1 2 3]
[2 3 4 0 1]  [4 0 1 2 3]  [1 2 3 4 0]  [3 4 0 1 2]
[3 4 0 1 2]  [1 2 3 4 0]  [4 0 1 2 3]  [2 3 4 0 1]
[4 0 1 2 3], [3 4 0 1 2], [2 3 4 0 1], [1 2 3 4 0]
]

>>> designs.mutually_orthogonal_latin_squares(Integer(3),Integer(7))
↪               # needs sage.schemes
[
[0 1 2 3 4 5 6]  [0 1 2 3 4 5 6]  [0 1 2 3 4 5 6]
[1 2 3 4 5 6 0]  [2 3 4 5 6 0 1]  [3 4 5 6 0 1 2]
[2 3 4 5 6 0 1]  [4 5 6 0 1 2 3]  [6 0 1 2 3 4 5]
[3 4 5 6 0 1 2]  [6 0 1 2 3 4 5]  [2 3 4 5 6 0 1]
[4 5 6 0 1 2 3]  [1 2 3 4 5 6 0]  [5 6 0 1 2 3 4]
[5 6 0 1 2 3 4]  [3 4 5 6 0 1 2]  [1 2 3 4 5 6 0]
[6 0 1 2 3 4 5], [5 6 0 1 2 3 4], [4 5 6 0 1 2 3]
]

>>> designs.mutually_orthogonal_latin_squares(Integer(2),Integer(5),
↪partitions=True)            # needs sage.schemes
[[[0, 1, 2, 3, 4],
  [5, 6, 7, 8, 9],
  [10, 11, 12, 13, 14],
  [15, 16, 17, 18, 19],
  [20, 21, 22, 23, 24]],
 [[0, 5, 10, 15, 20],
  [1, 6, 11, 16, 21],
  [2, 7, 12, 17, 22],
  [3, 8, 13, 18, 23],
```

(continues on next page)

(continued from previous page)

```
  [4, 9, 14, 19, 24]],
 [[0, 9, 13, 17, 21],
  [1, 5, 14, 18, 22],
  [2, 6, 10, 19, 23],
  [3, 7, 11, 15, 24],
  [4, 8, 12, 16, 20]],
 [[0, 8, 11, 19, 22],
  [1, 9, 12, 15, 23],
  [2, 5, 13, 16, 24],
  [3, 6, 14, 17, 20],
  [4, 7, 10, 18, 21]]]
```

What is the maximum number of MOLS of size 8 that Sage knows how to build?:

```
sage: designs.orthogonal_arrays.largest_available_k(8)-2                      #␣
↪needs sage.schemes
7
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.largest_available_k(Integer(8))-Integer(2)        ␣
↪              # needs sage.schemes
7
```

If you only want to know if Sage is able to build a given set of MOLS, query the `orthogonal_arrays.*` functions:

```
sage: designs.orthogonal_arrays.is_available(5+2, 5) # 5 MOLS of order 5
False
sage: designs.orthogonal_arrays.is_available(4+2,6)  # 4 MOLS of order 6        #␣
↪needs sage.schemes
False
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(5)+Integer(2), Integer(5)) # 5␣
↪MOLS of order 5
False
>>> designs.orthogonal_arrays.is_available(Integer(4)+Integer(2),Integer(6))  # 4␣
↪MOLS of order 6        # needs sage.schemes
False
```

Sage, however, is not able to prove that the second MOLS do not exist:

```
sage: designs.orthogonal_arrays.exists(4+2,6)  # 4 MOLS of order 6              #␣
↪needs sage.schemes
Unknown
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.exists(Integer(4)+Integer(2),Integer(6))  # 4 MOLS␣
↪of order 6              # needs sage.schemes
Unknown
```

If you ask for such a MOLS then you will respectively get an informative `EmptySetError` or `NotImplementedError`:

```
sage: designs.mutually_orthogonal_latin_squares(5, 5)
Traceback (most recent call last):
...
EmptySetError: there exist at most n-1 MOLS of size n if n>=2
sage: designs.mutually_orthogonal_latin_squares(4,6)                          #␣
↪needs sage.schemes
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build 4 MOLS of order 6
```

```
>>> from sage.all import *
>>> designs.mutually_orthogonal_latin_squares(Integer(5), Integer(5))
Traceback (most recent call last):
...
EmptySetError: there exist at most n-1 MOLS of size n if n>=2
>>> designs.mutually_orthogonal_latin_squares(Integer(4),Integer(6))           ␣
↪              # needs sage.schemes
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build 4 MOLS of order 6
```

5.1.91 Orthogonal arrays (OA)

This module gathers some construction related to orthogonal arrays (or transversal designs). One can build an $OA(k, n)$ (or check that it can be built) from the Sage console with `designs.orthogonal_arrays.build`:

```
sage: OA = designs.orthogonal_arrays.build(4,8)
```

```
>>> from sage.all import *
>>> OA = designs.orthogonal_arrays.build(Integer(4),Integer(8))
```

See also the modules *orthogonal_arrays_build_recursive* or *orthogonal_arrays_find_recursive* for recursive constructions.

This module defines the following functions:

orthogonal_array()	Return an orthogonal array of parameters k, n, t.
transversal_design()	Return a transversal design of parameters k, n.
incomplete_orthogonal_array()	Return an $OA(k, n) - \sum_{1 \leq i \leq x} OA(k, s_i)$.

is_transversal_design()	Check that a given set of blocks B is a transversal design.
is_orthogonal_array()	Check that the integer matrix OA is an $OA(k, n, t)$.
wilson_construction()	Return a $OA(k, rm+u)$ from a truncated $OA(k+s, r)$ by Wilson's construction.
TD_product()	Return the product of two transversal designs.
OA_find_disjoint_blocks()	Return x disjoint blocks contained in a given $OA(k, n)$.
OA_relabel()	Return a relabelled version of the OA.
OA_standard_label()	Return a version of the OA relabelled to symbols $(0, \dots, n-1)$.
OA_from_quasi_difference_matrix()	Return an Orthogonal Array from a Quasi-Difference matrix
OA_from_Vmt()	Return an Orthogonal Array from a $V(m, t)$
OA_from_PBD()	Return an $OA(k, n)$ from a PBD
OA_n_times_2_pow_c_from_matrix()	Return an $OA(k, \|G\| \cdot 2^c)$ from a constrained $(G, k-1, 2)$-difference matrix.
OA_from_wider_OA()	Return the first k columns of OA.
QDM_from_Vmt()	Return a QDM a $V(m, t)$

REFERENCES:

– [CD1996]

Functions

class sage.combinat.designs.orthogonal_arrays.**OAMainFunctions**(**args*, ***kwds*)

Bases: object

Functions related to orthogonal arrays.

An orthogonal array of parameters k, n, t is a matrix with k columns filled with integers from $[n]$ in such a way that for any t columns, each of the n^t possible rows occurs exactly once. In particular, the matrix has n^t rows.

For more information on orthogonal arrays, see Wikipedia article Orthogonal_array.

From here you have access to:

- build(k,n,t=2): return an orthogonal array with the given parameters.
- is_available(k,n,t=2): answer whether there is a construction available in Sage for a given set of parameters.
- exists(k,n,t=2): answer whether an orthogonal array with these parameters exist.
- largest_available_k(n,t=2): return the largest integer k such that Sage knows how to build an $OA(k, n)$.
- explain_construction(k,n,t=2): return a string that explains the construction that Sage uses to build an $OA(k, n)$.

EXAMPLES:

```
sage: designs.orthogonal_arrays.build(3,2)
[[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 0]]

sage: designs.orthogonal_arrays.build(5,5)
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 1, 3],
 [0, 3, 1, 4, 2], [0, 4, 3, 2, 1], [1, 0, 4, 3, 2],
 [1, 1, 1, 1, 1], [1, 2, 3, 4, 0], [1, 3, 0, 2, 4],
```

(continues on next page)

(continued from previous page)

```
 [1, 4, 2, 0, 3], [2, 0, 3, 1, 4], [2, 1, 0, 4, 3],
 [2, 2, 2, 2, 2], [2, 3, 4, 0, 1], [2, 4, 1, 3, 0],
 [3, 0, 2, 4, 1], [3, 1, 4, 2, 0], [3, 2, 1, 0, 4],
 [3, 3, 3, 3, 3], [3, 4, 0, 1, 2], [4, 0, 1, 2, 3],
 [4, 1, 3, 0, 2], [4, 2, 0, 3, 1], [4, 3, 2, 1, 0],
 [4, 4, 4, 4, 4]]
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.build(Integer(3),Integer(2))
[[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 0]]

>>> designs.orthogonal_arrays.build(Integer(5),Integer(5))
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 1, 3],
 [0, 3, 1, 4, 2], [0, 4, 3, 2, 1], [1, 0, 4, 3, 2],
 [1, 1, 1, 1, 1], [1, 2, 3, 4, 0], [1, 3, 0, 2, 4],
 [1, 4, 2, 0, 3], [2, 0, 3, 1, 4], [2, 1, 0, 4, 3],
 [2, 2, 2, 2, 2], [2, 3, 4, 0, 1], [2, 4, 1, 3, 0],
 [3, 0, 2, 4, 1], [3, 1, 4, 2, 0], [3, 2, 1, 0, 4],
 [3, 3, 3, 3, 3], [3, 4, 0, 1, 2], [4, 0, 1, 2, 3],
 [4, 1, 3, 0, 2], [4, 2, 0, 3, 1], [4, 3, 2, 1, 0],
 [4, 4, 4, 4, 4]]
```

What is the largest value of k for which Sage knows how to compute a $OA(k, 14, 2)$?:

```
sage: designs.orthogonal_arrays.largest_available_k(14)
6
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.largest_available_k(Integer(14))
6
```

If you ask for an orthogonal array that does not exist, then you will either obtain an `EmptySetError` (if it knows that such an orthogonal array does not exist) or a `NotImplementedError`:

```
sage: designs.orthogonal_arrays.build(4,2)
Traceback (most recent call last):
...
EmptySetError: There exists no OA(4,2) as k(=4)>n+t-1=3
sage: designs.orthogonal_arrays.build(12,20)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build an OA(12,20)!
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.build(Integer(4),Integer(2))
Traceback (most recent call last):
...
EmptySetError: There exists no OA(4,2) as k(=4)>n+t-1=3
>>> designs.orthogonal_arrays.build(Integer(12),Integer(20))
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build an OA(12,20)!
```

static build(*k*, *n*, *t=2*, *resolvable=False*)

Return an $OA(k, n)$ of strength t.

An orthogonal array of parameters k, n, t is a matrix with k columns filled with integers from $[n]$ in such a way that for any t columns, each of the n^t possible rows occurs exactly once. In particular, the matrix has n^t rows.

More general definitions sometimes involve a λ parameter, and we assume here that $\lambda = 1$.

For more information on orthogonal arrays, see Wikipedia article Orthogonal_array.

INPUT:

- `k`, `n`, `t` – integers; parameters of the orthogonal array
- `resolvable` – boolean (default: `False`); set to `True` if you want the design to be resolvable. The n classes of the resolvable design are obtained as the first n blocks, then the next n blocks, etc.

EXAMPLES:

```
sage: designs.orthogonal_arrays.build(3,3,resolvable=True) # indirect doctest
[[0, 0, 0],
 [1, 2, 1],
 [2, 1, 2],
 [0, 2, 2],
 [1, 1, 0],
 [2, 0, 1],
 [0, 1, 1],
 [1, 0, 2],
 [2, 2, 0]]
sage: OA_7_50 = designs.orthogonal_arrays.build(7,50)        # indirect doctest
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.build(Integer(3),Integer(3),resolvable=True) #␣
↪indirect doctest
[[0, 0, 0],
 [1, 2, 1],
 [2, 1, 2],
 [0, 2, 2],
 [1, 1, 0],
 [2, 0, 1],
 [0, 1, 1],
 [1, 0, 2],
 [2, 2, 0]]
>>> OA_7_50 = designs.orthogonal_arrays.build(Integer(7),Integer(50))        #␣
↪indirect doctest
```

static exists(*k*, *n*, *t=2*)

Return the existence status of an $OA(k, n)$.

INPUT:

- `k`, `n`, `t` – integers; parameters of the orthogonal array

Warning

The function does not only return booleans, but `True`, `False`, or `Unknown`.

See also

is_available()

EXAMPLES:

```
sage: designs.orthogonal_arrays.exists(3,6) # indirect doctest
True
sage: designs.orthogonal_arrays.exists(4,6) # indirect doctest
Unknown
sage: designs.orthogonal_arrays.exists(7,6) # indirect doctest
False
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.exists(Integer(3),Integer(6)) # indirect doctest
True
>>> designs.orthogonal_arrays.exists(Integer(4),Integer(6)) # indirect doctest
Unknown
>>> designs.orthogonal_arrays.exists(Integer(7),Integer(6)) # indirect doctest
False
```

static explain_construction(*k*, *n*, *t=2*)

Return a string describing how to builds an $OA(k, n)$.

INPUT:

- `k`, `n`, `t` – integers; parameters of the orthogonal array

EXAMPLES:

```
sage: designs.orthogonal_arrays.explain_construction(9,565)
"Wilson's construction n=23.24+13 with master design OA(9+1,23)"
sage: designs.orthogonal_arrays.explain_construction(10,154)
'the database contains a (137,10;1,0;17)-quasi difference matrix'
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.explain_construction(Integer(9),Integer(565))
"Wilson's construction n=23.24+13 with master design OA(9+1,23)"
>>> designs.orthogonal_arrays.explain_construction(Integer(10),Integer(154))
'the database contains a (137,10;1,0;17)-quasi difference matrix'
```

static is_available(*k*, *n*, *t=2*)

Return whether Sage can build an $OA(k, n)$.

INPUT:

- `k`, `n`, `t` – integers; parameters of the orthogonal array

See also

exists()

EXAMPLES:

```
sage: designs.orthogonal_arrays.is_available(3,6) # indirect doctest
True
sage: designs.orthogonal_arrays.is_available(4,6) # indirect doctest
False
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.is_available(Integer(3),Integer(6)) # indirect␣
→doctest
True
>>> designs.orthogonal_arrays.is_available(Integer(4),Integer(6)) # indirect␣
→doctest
False
```

static largest_available_k(*n*, *t=2*)

Return the largest k such that Sage can build an $OA(k, n)$.

INPUT:

- `n` – integer
- `t` – integer (default: 2); strength of the array

EXAMPLES:

```
sage: designs.orthogonal_arrays.largest_available_k(0)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(1)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(10)
4
sage: designs.orthogonal_arrays.largest_available_k(27)
28
sage: designs.orthogonal_arrays.largest_available_k(100)
10
sage: designs.orthogonal_arrays.largest_available_k(-1)
Traceback (most recent call last):
...
ValueError: n(=-1) was expected to be >=0
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.largest_available_k(Integer(0))
+Infinity
>>> designs.orthogonal_arrays.largest_available_k(Integer(1))
+Infinity
>>> designs.orthogonal_arrays.largest_available_k(Integer(10))
4
>>> designs.orthogonal_arrays.largest_available_k(Integer(27))
28
>>> designs.orthogonal_arrays.largest_available_k(Integer(100))
10
>>> designs.orthogonal_arrays.largest_available_k(-Integer(1))
Traceback (most recent call last):
...
ValueError: n(=-1) was expected to be >=0
```

sage.combinat.designs.orthogonal_arrays.**OA_find_disjoint_blocks**(*OA*, *k*, *n*, *x*, *solver*, *integrality_tolerance*)

Return x disjoint blocks contained in a given $OA(k, n)$.

x blocks of an OA are said to be disjoint if they all have different values for a every given index, i.e. if they correspond to disjoint blocks in the TD associated with the OA.

INPUT:

- `OA` – an orthogonal array
- `k`, `n`, `x` – integers
- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`

See also

`incomplete_orthogonal_array()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_find_disjoint_blocks
sage: k=3;n=4;x=3
sage: Bs = OA_find_disjoint_blocks(designs.orthogonal_arrays.build(k,n),k,n,x)
sage: assert len(Bs) == x
sage: for i in range(k):
....:     assert len(set([B[i] for B in Bs])) == x
sage: OA_find_disjoint_blocks(designs.orthogonal_arrays.build(k,n),k,n,5)
Traceback (most recent call last):
...
ValueError: There does not exist 5 disjoint blocks in this OA(3,4)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import OA_find_disjoint_blocks
>>> k=Integer(3);n=Integer(4);x=Integer(3)
>>> Bs = OA_find_disjoint_blocks(designs.orthogonal_arrays.build(k,n),k,n,x)
>>> assert len(Bs) == x
>>> for i in range(k):
...     assert len(set([B[i] for B in Bs])) == x
>>> OA_find_disjoint_blocks(designs.orthogonal_arrays.build(k,n),k,n,Integer(5))
Traceback (most recent call last):
...
ValueError: There does not exist 5 disjoint blocks in this OA(3,4)
```

sage.combinat.designs.orthogonal_arrays.**OA_from_PBD**(*k*, *n*, *PBD*, *check=True*)

Return an $OA(k, n)$ from a PBD.

Construction

Let $\mathcal{B}$ be a $(n, K, 1)$-PBD. If there exists for every $i \in K$ a $TD(k, i) - i \times TD(k, 1)$ (i.e. if there exist k idempotent MOLS), then one can obtain a $OA(k, n)$ by concatenating:

- A $TD(k, i) - i \times TD(k, 1)$ defined over the elements of B for every $B \in \mathcal{B}$.
- The rows $(i, ..., i)$ of length k for every $i \in [n]$.

Note

This function raises an exception when Sage is unable to build the necessary designs.

INPUT:

- k, n – integers
- PBD – a PBD on $0, \ldots, n - 1$

EXAMPLES:

We start from the example VI.1.2 from the [DesignHandbook] to build an $OA(3, 10)$:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_from_PBD
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: pbd = [[0,1,2,3],[0,4,5,6],[0,7,8,9],[1,4,7],[1,5,8],
....: [1,6,9],[2,4,9],[2,5,7],[2,6,8],[3,4,8],[3,5,9],[3,6,7]]
sage: oa = OA_from_PBD(3,10,pbd)
sage: is_orthogonal_array(oa, 3, 10)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import OA_from_PBD
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> pbd = [[Integer(0),Integer(1),Integer(2),Integer(3)],[Integer(0),Integer(4),
↪Integer(5),Integer(6)],[Integer(0),Integer(7),Integer(8),Integer(9)],
↪[Integer(1),Integer(4),Integer(7)],[Integer(1),Integer(5),Integer(8)],
... [Integer(1),Integer(6),Integer(9)],[Integer(2),Integer(4),Integer(9)],
↪[Integer(2),Integer(5),Integer(7)],[Integer(2),Integer(6),Integer(8)],
↪[Integer(3),Integer(4),Integer(8)],[Integer(3),Integer(5),Integer(9)],
↪[Integer(3),Integer(6),Integer(7)]]
>>> oa = OA_from_PBD(Integer(3),Integer(10),pbd)
>>> is_orthogonal_array(oa, Integer(3), Integer(10))
True
```

But we cannot build an $OA(4, 10)$ for this PBD (although there exists an $OA(4, 10)$:

```
sage: OA_from_PBD(4,10,pbd)
Traceback (most recent call last):
...
EmptySetError: There is no OA(n+1,n) - 3.OA(n+1,1)
as all blocks intersect in a projective plane.
```

```
>>> from sage.all import *
>>> OA_from_PBD(Integer(4),Integer(10),pbd)
Traceback (most recent call last):
...
EmptySetError: There is no OA(n+1,n) - 3.OA(n+1,1)
as all blocks intersect in a projective plane.
```

Or an $OA(3, 6)$ (as the PBD has 10 points):

```
sage: _ = OA_from_PBD(3,6,pbd)
Traceback (most recent call last):
...
RuntimeError: PBD is not a valid Pairwise Balanced Design on [0,...,5]
```

```
>>> from sage.all import *
>>> _ = OA_from_PBD(Integer(3),Integer(6),pbd)
Traceback (most recent call last):
...
RuntimeError: PBD is not a valid Pairwise Balanced Design on [0,...,5]
```

sage.combinat.designs.orthogonal_arrays.**OA_from_Vmt**(*m*, *t*, *V*)

Return an Orthogonal Array from a $V(m, t)$.

INPUT:

- m, t – integers
- V – the vector $V(m, t)$

See also

- *QDM_from_Vmt()*
- *OA_from_quasi_difference_matrix()*

EXAMPLES:

```
sage: _ = designs.orthogonal_arrays.build(6,46) # indirect doctest
```

```
>>> from sage.all import *
>>> _ = designs.orthogonal_arrays.build(Integer(6),Integer(46)) # indirect doctest
```

sage.combinat.designs.orthogonal_arrays.**OA_from_quasi_difference_matrix**(*M*, *G*, *add_col=True*, *fill_hole=True*)

Return an Orthogonal Array from a Quasi-Difference matrix.

Difference Matrices

Let G be a group of order g. A *difference matrix* M is a $g \times k$ matrix with entries from G such that for any $1 \leq i < j < k$ the set $\{d_{li} - d_{lj} : 1 \leq l \leq g\}$ is equal to G.

By concatenating the g matrices $M + x$ (where $x \in G$), one obtains a matrix of size $g^2 \times x$ which is also an $OA(k, g)$.

Quasi-difference Matrices

A quasi-difference matrix is a difference matrix with missing entries. The construction above can be applied again in this case, where the missing entries in each column of M are replaced by unique values on which G has a trivial action.

This produces an incomplete orthogonal array with a "hole" (i.e. missing rows) of size 'u' (i.e. the number of missing values per column of M). If there exists an $OA(k, u)$, then adding the rows of this $OA(k, u)$ to the incomplete orthogonal array should lead to an OA…

Formal definition (from the Handbook of Combinatorial Designs [DesignHandbook])

Let G be an abelian group of order n. A $(n, k; \lambda, \mu; u)$-quasi-difference matrix (QDM) is a matrix $Q = (q_{ij})$ with $\lambda(n-1+2u)+\mu$ rows and k columns, with each entry either empty or containing an element of G. Each column contains exactly λu entries, and each row contains at most one empty entry. Furthermore, for each $1 \leq i < j \leq k$ the multiset

$$\{q_{li} - q_{lj} : 1 \leq l \leq \lambda(n-1+2u)+\mu, \text{ with } q_{li} \text{ and } q_{lj} \text{ not empty}\}$$

contains every nonzero element of G exactly λ times, and contains 0 exactly μ times.

Construction

If a $(n, k; \lambda, \mu; u)$-QDM exists and $\mu \leq \lambda$, then an $ITD_\lambda(k, n+u; u)$ exists. Start with a $(n, k; \lambda, \mu; u)$-QDM A over the group G. Append $\lambda - \mu$ rows of zeroes. Then select u elements $\infty_1, \dots, \infty_u$ not in G, and replace the empty entries, each by one of these infinite symbols, so that ∞_i appears exactly once in each column. Develop the resulting matrix over the group G (leaving infinite symbols fixed), to obtain a $\lambda(n^2+2nu) \times k$ matrix T. Then T is an orthogonal array with k columns and index λ, having $n+u$ symbols and one hole of size u.

Adding to T an $OA(k, u)$ with elements $\infty_1, \dots, \infty_u$ yields the $ITD_\lambda(k, n+u; u)$.

For more information, see the Handbook of Combinatorial Designs [DesignHandbook] or http://web.cs.du.edu/~petr/milehigh/2013/Colbourn.pdf.

INPUT:

- `M` – the difference matrix whose entries belong to `G`
- `G` – a group
- `add_col` – boolean; whether to add a column to the final OA equal to $(x_1, \dots, x_g, x_1, \dots, x_g, \dots)$ where $G = \{x_1, \dots, x_g\}$
- `fill_hole` – boolean; whether to return the incomplete orthogonal array, or complete it with the $OA(k, u)$ (default). When `fill_hole is None`, no block of the incomplete OA contains more than one value $\geq |G|$.

EXAMPLES:

```
sage: _ = designs.orthogonal_arrays.build(6,20) # indirect doctest
```

```
>>> from sage.all import *
>>> _ = designs.orthogonal_arrays.build(Integer(6),Integer(20)) # indirect doctest
```

sage.combinat.designs.orthogonal_arrays.**OA_from_wider_OA**(*OA*, *k*)

Return the first k columns of OA.

If OA has k columns, this function returns OA immediately.

INPUT:

- `OA` – an orthogonal array
- `k` – integer

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_from_wider_OA
sage: OA_from_wider_OA(designs.orthogonal_arrays.build(6,20,2),1)[:5]
[(19,), (19,), (19,), (19,), (19,)]
sage: _ = designs.orthogonal_arrays.build(5,46) # indirect doctest
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import OA_from_wider_OA
>>> OA_from_wider_OA(designs.orthogonal_arrays.build(Integer(6),Integer(20),
↪Integer(2)),Integer(1))[:Integer(5)]
[(19,), (19,), (19,), (19,), (19,)]
>>> _ = designs.orthogonal_arrays.build(Integer(5),Integer(46)) # indirect doctest
```

sage.combinat.designs.orthogonal_arrays.**OA_n_times_2_pow_c_from_matrix**(*k*, *c*, *G*, *A*, *Y*, *check=True*)

Return an $OA(k, |G| \cdot 2^c)$ from a constrained $(G, k-1, 2)$-difference matrix.

This construction appears in [AC1994] and [Ab1995].

Let G be an additive Abelian group. We denote by H a $GF(2)$-hyperplane in $GF(2^c)$.

Let A be a $(k-1) \times 2|G|$ array with entries in $G \times GF(2^c)$ and Y be a vector with $k-1$ entries in $GF(2^c)$. Let B and C be respectively the part of the array that belong to G and $GF(2^c)$.

The input A and Y must satisfy the following conditions. For any $i \neq j$ and $g \in G$:

- there are exactly two values of s such that $B_{i,s} - B_{j,s} = g$ (i.e. B is a $(G, k-1, 2)$-difference matrix),
- let s_1 and s_2 denote the two values of s given above, then exactly one of $C_{i,s_1} - C_{j,s_1}$ and $C_{i,s_2} - C_{j,s_2}$ belongs to the $GF(2)$-hyperplane $(Y_i - Y_j) \cdot H$ (we implicitly assumed that $Y_i \neq Y_j$).

Under these conditions, it is easy to check that the array whose $k-1$ rows of length $|G| \cdot 2^c$ indexed by $1 \leq i \leq k-1$ given by $A_{i,s} + (0, Y_i \cdot v)$ where $1 \leq s \leq 2|G|, v \in H$ is a $(G \times GF(2^c), k-1, 1)$-difference matrix.

INPUT:

- `k`, `c` – integers
- `G` – an additive Abelian group
- `A` – a matrix with entries in $G \times GF(2^c)$
- `Y` – a vector with entries in $GF(2^c)$
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

Note

By convention, a multiplicative generator w of $GF(2^c)^*$ is fixed (inside the function). The hyperplane H is the one spanned by $w^0, w^1, \ldots, w^{c-1}$. The $GF(2^c)$ part of the input matrix A and vector Y are given in the following form: the integer i corresponds to the element w^i and `None` corresponds to 0.

See also

Several examples use this construction:

- *OA_9_40()*
- *OA_11_80()*
- *OA_15_112()*
- *OA_11_160()*
- *OA_16_176()*

- *OA_16_208()*
- *OA_15_224()*
- *OA_20_352()*
- *OA_20_416()*
- *OA_20_544()*
- *OA_11_640()*
- *OA_15_896()*

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_n_times_2_pow_c_from_
↪matrix
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: A = [
....: [(0,None),(0,None),(0,None),(0,None),(0,None),(0,None),(0,None),(0,None),(0,
↪None),(0,None)],
....: [(0,None),(1,None),   (2,2),   (3,2),   (4,2),(2,None),(3,None),(4,None),  ␣
↪(0,2),   (1,2)],
....: [(0,None),   (2,5),   (4,5),   (1,2),   (3,6),   (3,4),   (0,0),   (2,1),  ␣
↪(4,1),   (1,6)],
....: [(0,None),   (3,4),   (1,4),   (4,0),   (2,5),(3,None),   (1,0),   (4,1),  ␣
↪(2,2),   (0,3)],
....: ]
sage: Y = [None, 0, 1, 6]
sage: OA = OA_n_times_2_pow_c_from_matrix(5,3,GF(5),A,Y)
sage: is_orthogonal_array(OA,5,40,2)
True

sage: A[0][0] = (1,None)
sage: OA_n_times_2_pow_c_from_matrix(5,3,GF(5),A,Y)
Traceback (most recent call last):
...
ValueError: the first part of the matrix A must be a
(G,k-1,2)-difference matrix

sage: A[0][0] = (0,0)
sage: OA_n_times_2_pow_c_from_matrix(5,3,GF(5),A,Y)
Traceback (most recent call last):
...
ValueError: B_2,0 - B_0,0 = B_2,6 - B_0,6 but the associated part of the
matrix C does not satisfies the required condition
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import OA_n_times_2_pow_c_from_
↪matrix
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> A = [
... [(Integer(0),None),(Integer(0),None),(Integer(0),None),(Integer(0),None),
↪(Integer(0),None),(Integer(0),None),(Integer(0),None),(Integer(0),None),
```

(continues on next page)

(continued from previous page)

```
↪(Integer(0),None),(Integer(0),None)],
... [(Integer(0),None),(Integer(1),None),   (Integer(2),Integer(2)),  ␣
↪(Integer(3),Integer(2)),   (Integer(4),Integer(2)),(Integer(2),None),
↪(Integer(3),None),(Integer(4),None),   (Integer(0),Integer(2)),   (Integer(1),
↪Integer(2))],
... [(Integer(0),None),   (Integer(2),Integer(5)),   (Integer(4),Integer(5)),  ␣
↪(Integer(1),Integer(2)),   (Integer(3),Integer(6)),   (Integer(3),Integer(4)), ␣
↪ (Integer(0),Integer(0)),   (Integer(2),Integer(1)),   (Integer(4),Integer(1)),␣
↪  (Integer(1),Integer(6))],
... [(Integer(0),None),   (Integer(3),Integer(4)),   (Integer(1),Integer(4)),  ␣
↪(Integer(4),Integer(0)),   (Integer(2),Integer(5)),(Integer(3),None),  ␣
↪(Integer(1),Integer(0)),   (Integer(4),Integer(1)),   (Integer(2),Integer(2)), ␣
↪ (Integer(0),Integer(3))],
... ]
>>> Y = [None, Integer(0), Integer(1), Integer(6)]
>>> OA = OA_n_times_2_pow_c_from_matrix(Integer(5),Integer(3),GF(Integer(5)),A,Y)
>>> is_orthogonal_array(OA,Integer(5),Integer(40),Integer(2))
True

>>> A[Integer(0)][Integer(0)] = (Integer(1),None)
>>> OA_n_times_2_pow_c_from_matrix(Integer(5),Integer(3),GF(Integer(5)),A,Y)
Traceback (most recent call last):
...
ValueError: the first part of the matrix A must be a
(G,k-1,2)-difference matrix

>>> A[Integer(0)][Integer(0)] = (Integer(0),Integer(0))
>>> OA_n_times_2_pow_c_from_matrix(Integer(5),Integer(3),GF(Integer(5)),A,Y)
Traceback (most recent call last):
...
ValueError: B_2,0 - B_0,0 = B_2,6 - B_0,6 but the associated part of the
matrix C does not satisfies the required condition
```

sage.combinat.designs.orthogonal_arrays.**OA_relabel**(*OA*, *k*, *n*, *blocks=()*, *matrix=None*, *symbol_list=None*)

Return a relabelled version of the OA.

INPUT:

- `OA` – an OA, or rather a list of blocks of length k, each of which contains integers from 0 to $n-1$
- `k`, `n` – integers
- `blocks` – list of blocks; relabels the integers of the OA from $[0..n-1]$ into $[0..n-1]$ in such a way that the i blocks from `block` are respectively relabeled as `[n-i,...,n-i]`, ..., `[n-1,...,n-1]`. Thus, the blocks from this list are expected to have disjoint values for each coordinate.

  If set to the empty list (default) no such relabelling is performed.
- `matrix` – a matrix of dimensions k, n such that if the i th coordinate of a block is x, this x will be relabelled with `matrix[i][x]`. This is not necessarily an integer between 0 and $n-1$, and it is not necessarily an integer either. This is performed *after* the previous relabelling.

  If set to `None` (default) no such relabelling is performed.
- `symbol_list` – list of the desired symbols for the relabelled OA. If this is not `None`, the same relabelling

is done on all blocks such that the index of an element in symbol_list is its preimage in the relabelling map.

Note: A `None` coordinate in one block remains a `None` coordinate in the final block.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_relabel
sage: OA = designs.orthogonal_arrays.build(3,2)
sage: OA_relabel(OA,3,2,matrix=[["A","B"],["C","D"],["E","F"]])
[['A', 'C', 'E'], ['A', 'D', 'F'], ['B', 'C', 'F'], ['B', 'D', 'E']]

sage: TD = OA_relabel(OA,3,2,matrix=[[0,1],[2,3],[4,5]]); TD
[[0, 2, 4], [0, 3, 5], [1, 2, 5], [1, 3, 4]]
sage: from sage.combinat.designs.orthogonal_arrays import is_transversal_design
sage: is_transversal_design(TD,3,2)
True

sage: OA = designs.orthogonal_arrays.build(3,2)
sage: OA_relabel(OA, 3, 2, symbol_list=['A', 'B'])
[['A', 'A', 'A'], ['A', 'B', 'B'], ['B', 'A', 'B'], ['B', 'B', 'A']]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import OA_relabel
>>> OA = designs.orthogonal_arrays.build(Integer(3),Integer(2))
>>> OA_relabel(OA,Integer(3),Integer(2),matrix=[["A","B"],["C","D"],["E","F"]])
[['A', 'C', 'E'], ['A', 'D', 'F'], ['B', 'C', 'F'], ['B', 'D', 'E']]

>>> TD = OA_relabel(OA,Integer(3),Integer(2),matrix=[[Integer(0),Integer(1)],
↪[Integer(2),Integer(3)],[Integer(4),Integer(5)]]); TD
[[0, 2, 4], [0, 3, 5], [1, 2, 5], [1, 3, 4]]
>>> from sage.combinat.designs.orthogonal_arrays import is_transversal_design
>>> is_transversal_design(TD,Integer(3),Integer(2))
True

>>> OA = designs.orthogonal_arrays.build(Integer(3),Integer(2))
>>> OA_relabel(OA, Integer(3), Integer(2), symbol_list=['A', 'B'])
[['A', 'A', 'A'], ['A', 'B', 'B'], ['B', 'A', 'B'], ['B', 'B', 'A']]
```

Making sure that `[2,2,2,2]` is a block of $OA(4, 3)$. We do this by relabelling block `[0,0,0,0]` which belongs to the design:

```
sage: designs.orthogonal_arrays.build(4,3)
[[0, 0, 0, 0], [0, 1, 2, 1], [0, 2, 1, 2], [1, 0, 2, 2], [1, 1, 1, 0], [1, 2, 0,␣
↪1], [2, 0, 1, 1], [2, 1, 0, 2], [2, 2, 2, 0]]
sage: OA_relabel(designs.orthogonal_arrays.build(4,3),4,3,blocks=[[0,0,0,0]])
[[2, 2, 2, 2], [2, 0, 1, 0], [2, 1, 0, 1], [0, 2, 1, 1], [0, 0, 0, 2], [0, 1, 2,␣
↪0], [1, 2, 0, 0], [1, 0, 2, 1], [1, 1, 1, 2]]
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.build(Integer(4),Integer(3))
```

(continues on next page)

(continued from previous page)

```
[[0, 0, 0, 0], [0, 1, 2, 1], [0, 2, 1, 2], [1, 0, 2, 2], [1, 1, 1, 0], [1, 2, 0,␣
↪1], [2, 0, 1, 1], [2, 1, 0, 2], [2, 2, 2, 0]]
>>> OA_relabel(designs.orthogonal_arrays.build(Integer(4),Integer(3)),Integer(4),
↪Integer(3),blocks=[[Integer(0),Integer(0),Integer(0),Integer(0)]])
[[2, 2, 2, 2], [2, 0, 1, 0], [2, 1, 0, 1], [0, 2, 1, 1], [0, 0, 0, 2], [0, 1, 2,␣
↪0], [1, 2, 0, 0], [1, 0, 2, 1], [1, 1, 1, 2]]
```

sage.combinat.designs.orthogonal_arrays.**OA_standard_label**(*OA*)

Return the inputted OA with entries relabelled as integers [0,…,n-1].

INPUT:

- `OA` – list of lists with symbols as entries that are not necessarily integers

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_standard_label
sage: C = [['a', 'a', 'a', 'b'],
....:      ['a', 'a', 'b', 'a'],
....:      ['a', 'b', 'a', 'a'],
....:      ['b', 'a', 'a', 'a'],
....:      ['b', 'b', 'b', 'b']]
sage: OA_standard_label(C)
[[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import OA_standard_label
>>> C = [['a', 'a', 'a', 'b'],
...      ['a', 'a', 'b', 'a'],
...      ['a', 'b', 'a', 'a'],
...      ['b', 'a', 'a', 'a'],
...      ['b', 'b', 'b', 'b']]
>>> OA_standard_label(C)
[[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0], [1, 1, 1, 1]]
```

sage.combinat.designs.orthogonal_arrays.**QDM_from_Vmt**(*m*, *t*, *V*)

Return a QDM from a $V(m,t)$.

Definition

Let q be a prime power and let $q = mt + 1$ for m, t integers. Let ω be a primitive element of $\mathbf{F}_q$. A $V(m,t)$ vector is a vector $(a_1, \dots, a_{m+1}$ for which, for each $1 \leq k < m$, the differences

$$\{a_{i+k} - a_i : 1 \leq i \leq m+1, i+k \neq m+2\}$$

represent the m cyclotomic classes of $\mathbf{F}_{mt+1}$ (compute subscripts modulo $m+2$). In other words, for fixed k, is $a_{i+k} - a_i = \omega^{mx+\alpha}$ and $a_{j+k} - a_j = \omega^{my+\beta}$ then $\alpha \not\equiv \beta \mod m$

Construction of a quasi-difference matrix from a `V(m,t)` vector

Starting with a $V(m,t)$ vector $(a_1, \dots, a_{m+1})$, form a single row of length $m+2$ whose first entry is empty, and whose remaining entries are $(a_1, \dots, a_{m+1})$. Form t rows by multiplying this row by the t th roots, i.e. the powers of ω^m. From each of these t rows, form $m+2$ rows by taking the $m+2$ cyclic shifts of the row. The result is a $(a, m+2; 1, 0; t) - QDM$.

For more information, refer to the Handbook of Combinatorial Designs [DesignHandbook].

INPUT:

- m, t – integers
- V – the vector $V(m, t)$

See also

OA_from_quasi_difference_matrix()

EXAMPLES:

```
sage: _ = designs.orthogonal_arrays.build(6,46) # indirect doctest
```

```
>>> from sage.all import *
>>> _ = designs.orthogonal_arrays.build(Integer(6),Integer(46)) # indirect doctest
```

sage.combinat.designs.orthogonal_arrays.**TD_product**(*k*, *TD1*, *n1*, *TD2*, *n2*, *check=True*)

Return the product of two transversal designs.

From a transversal design TD_1 of parameters k, n_1 and a transversal design TD_2 of parameters k, n_2, this function returns a transversal design of parameters k, n where $n = n_1 \times n_2$.

Formally, if the groups of TD_1 are $V^1_1, \dots, V^1_k$ and the groups of TD_2 are $V^2_1, \dots, V^2_k$, the groups of the product design are $V^1_1 \times V^2_1, \dots, V^1_k \times V^2_k$ and its blocks are the $\{(x^1_1, x^2_1), \dots, (x^1_k, x^2_k)\}$ where $\{x^1_1, \dots, x^1_k\}$ is a block of TD_1 and $\{x^2_1, \dots, x^2_k\}$ is a block of TD_2.

INPUT:

- TD1, TD2 – transversal designs
- k, n1, n2 – integers
- check – boolean (default: True); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

Note

This function uses transversal designs with $V_1 = \{0, \dots, n-1\}, \dots, V_k = \{(k-1)n, \dots, kn-1\}$ both as input and output.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import TD_product
sage: TD1 = designs.transversal_design(6,7)
sage: TD2 = designs.transversal_design(6,12)
sage: TD6_84 = TD_product(6,TD1,7,TD2,12)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import TD_product
>>> TD1 = designs.transversal_design(Integer(6),Integer(7))
>>> TD2 = designs.transversal_design(Integer(6),Integer(12))
>>> TD6_84 = TD_product(Integer(6),TD1,Integer(7),TD2,Integer(12))
```

class sage.combinat.designs.orthogonal_arrays.**TransversalDesign**(*blocks*, *k=None*, *n=None*, *check=True*, ***kwds*)

Bases: *GroupDivisibleDesign*

Class for Transversal Designs.

INPUT:

- `blocks` – collection of blocks
- `k`, `n` – integers; parameters of the transversal design. They can be set to `None` (default) in which case their value is determined by the blocks.
- `check` – boolean (default: `True`); whether to check that the design is indeed a transversal design with the right parameters

EXAMPLES:

```
sage: designs.transversal_design(None,5)
Transversal Design TD(6,5)
sage: designs.transversal_design(None,30)
Transversal Design TD(6,30)
sage: designs.transversal_design(None,36)
Transversal Design TD(10,36)
```

```
>>> from sage.all import *
>>> designs.transversal_design(None,Integer(5))
Transversal Design TD(6,5)
>>> designs.transversal_design(None,Integer(30))
Transversal Design TD(6,30)
>>> designs.transversal_design(None,Integer(36))
Transversal Design TD(10,36)
```

`sage.combinat.designs.orthogonal_arrays.`**`incomplete_orthogonal_array`**`(`*k*, *n*, *holes*, *resolvable=False*, *existence=False*`)`

Return an $OA(k,n) - \sum_{1\leq i\leq x} OA(k,s_i)$.

An $OA(k,n) - \sum_{1\leq i\leq x} OA(k,s_i)$ is an orthogonal array from which have been removed disjoint $OA(k,s_1),...,OA(k,s_x)$. If there exist $OA(k,s_1),...,OA(k,s_x)$ they can be used to fill the holes and give rise to an $OA(k,n)$.

A very useful particular case (see e.g. the Wilson construction in `wilson_construction()`) is when all $s_i = 1$. In that case the incomplete design is a $OA(k,n) - x.OA(k,1)$. Such design is equivalent to transversal design $TD(k,n)$ from which has been removed x disjoint blocks.

INPUT:

- `k`, `n` – integers
- `holes` – list of integers respective sizes of the holes to be found
- `resolvable` – boolean (default: `False`); set to `True` if you want the design to be resolvable. The classes of the resolvable design are obtained as the first n blocks, then the next n blocks, etc.
- `existence` – boolean; instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`)
  - `False` – meaning that the design does not exist

Note

By convention, the ground set is always $V = \{0, ..., n-1\}$.

If all holes have size 1, in the incomplete orthogonal array returned by this function the holes are $\{n-1, ..., n-s_1\}^k$, $\{n-s_1-1, ..., n-s_1-s_2\}^k$, etc.

More generally, if `holes` is equal to $u1, ..., uk$, the i-th hole is the set of points $\{n-\sum_{j\geq i} u_j, ..., n-\sum_{j\geq i+1} u_j\}^k$.

See also

OA_find_disjoint_blocks()

EXAMPLES:

```
sage: IOA = designs.incomplete_orthogonal_array(3,3,[1,1,1])
sage: IOA
[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
sage: missing_blocks = [[0,0,0],[1,1,1],[2,2,2]]
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: is_orthogonal_array(IOA + missing_blocks,3,3,2)
True
```

```
>>> from sage.all import *
>>> IOA = designs.incomplete_orthogonal_array(Integer(3),Integer(3),[Integer(1),
↪Integer(1),Integer(1)])
>>> IOA
[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
>>> missing_blocks = [[Integer(0),Integer(0),Integer(0)],[Integer(1),Integer(1),
↪Integer(1)],[Integer(2),Integer(2),Integer(2)]]
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> is_orthogonal_array(IOA + missing_blocks,Integer(3),Integer(3),Integer(2))
True
```

`sage.combinat.designs.orthogonal_arrays.`**`is_transversal_design`**(*B*, *k*, *n*, *verbose=False*)

Check that a given set of blocks `B` is a transversal design.

See *transversal_design()* for a definition.

INPUT:

- `B` – the list of blocks
- `k`, `n` – integers
- `verbose` – boolean; whether to display information about what is going wrong

Note

The transversal design must have $\{0, \ldots, kn-1\}$ as a ground set, partitioned as k sets of size n: $\{0, \ldots, k-1\} \sqcup \{k, \ldots, 2k-1\} \sqcup \cdots \sqcup \{k(n-1), \ldots, kn-1\}$.

EXAMPLES:

```
sage: TD = designs.transversal_design(5, 5, check=True) # indirect doctest
sage: from sage.combinat.designs.orthogonal_arrays import is_transversal_design
sage: is_transversal_design(TD, 5, 5)
True
sage: is_transversal_design(TD, 4, 4)
False
```

```
>>> from sage.all import *
>>> TD = designs.transversal_design(Integer(5), Integer(5), check=True) #␣
↪indirect doctest
>>> from sage.combinat.designs.orthogonal_arrays import is_transversal_design
>>> is_transversal_design(TD, Integer(5), Integer(5))
True
>>> is_transversal_design(TD, Integer(4), Integer(4))
False
```

sage.combinat.designs.orthogonal_arrays.**largest_available_k**(*n*, *t=2*)

Return the largest k such that Sage can build an $OA(k, n)$.

INPUT:

- n – integer
- t – integer (default: 2); strength of the array

EXAMPLES:

```
sage: designs.orthogonal_arrays.largest_available_k(0)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(1)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(10)
4
sage: designs.orthogonal_arrays.largest_available_k(27)
28
sage: designs.orthogonal_arrays.largest_available_k(100)
10
sage: designs.orthogonal_arrays.largest_available_k(-1)
Traceback (most recent call last):
...
ValueError: n(=-1) was expected to be >=0
```

```
>>> from sage.all import *
>>> designs.orthogonal_arrays.largest_available_k(Integer(0))
+Infinity
>>> designs.orthogonal_arrays.largest_available_k(Integer(1))
+Infinity
>>> designs.orthogonal_arrays.largest_available_k(Integer(10))
4
>>> designs.orthogonal_arrays.largest_available_k(Integer(27))
28
>>> designs.orthogonal_arrays.largest_available_k(Integer(100))
10
```

(continues on next page)

(continued from previous page)

```
>>> designs.orthogonal_arrays.largest_available_k(-Integer(1))
Traceback (most recent call last):
...
ValueError: n(=-1) was expected to be >=0
```

sage.combinat.designs.orthogonal_arrays.**orthogonal_array**(*k*, *n*, *t=2*, *resolvable=False*, *check=True*, *existence=False*, *explain_construction=False*)

Return an orthogonal array of parameters k, n, t.

An orthogonal array of parameters k, n, t is a matrix with k columns filled with integers from $[n]$ in such a way that for any t columns, each of the n^t possible rows occurs exactly once. In particular, the matrix has n^t rows.

More general definitions sometimes involve a λ parameter, and we assume here that $\lambda = 1$.

An orthogonal array is said to be *resolvable* if it corresponds to a resolvable transversal design (see `sage.combinat.designs.incidence_structures.IncidenceStructure.is_resolvable()`).

For more information on orthogonal arrays, see Wikipedia article Orthogonal_array.

INPUT:

- `k` – integer; number of columns. If `k` is `None` it is set to the largest value available.
- `n` – integer; number of symbols
- `t` – integer (default: 2); strength of the array
- `resolvable` – boolean (default: `False`); set to `True` if you want the design to be resolvable. The n classes of the resolvable design are obtained as the first n blocks, then the next n blocks, etc.
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- `existence` – boolean; instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`)
  - `False` – meaning that the design does not exist

  Note

  When `k=None` and `existence=True` the function returns an integer, i.e. the largest k such that we can build a $OA(k, n)$.

- `explain_construction` – boolean; return a string describing the construction

OUTPUT: the kind of output depends on the input:

- if `existence=False` (the default) then the output is a list of lists that represent an orthogonal array with parameters `k` and `n`
- if `existence=True` and `k` is an integer, then the function returns a troolean: either `True`, `Unknown` or `False`
- if `existence=True` and `k=None` then the output is the largest value of `k` for which Sage knows how to compute a $TD(k, n)$.

Note

This method implements theorems from [Stinson2004]. See the code's documentation for details.

See also

When $t = 2$ an orthogonal array is also a transversal design (see *transversal_design()*) and a family of mutually orthogonal latin squares (see *mutually_orthogonal_latin_squares()*).

`sage.combinat.designs.orthogonal_arrays.`**`transversal_design`**(*k*, *n*, *resolvable=False*, *check=True*, *existence=False*)

Return a transversal design of parameters k, n.

A transversal design of parameters k, n is a collection $\mathcal{S}$ of subsets of $V = V_1 \cup \cdots \cup V_k$ (where the *groups* V_i are disjoint and have cardinality n) such that:

- Any $S \in \mathcal{S}$ has cardinality k and intersects each group on exactly one element.
- Any two elements from distincts groups are contained in exactly one element of $\mathcal{S}$.

More general definitions sometimes involve a λ parameter, and we assume here that $\lambda = 1$.

For more information on transversal designs, see http://mathworld.wolfram.com/TransversalDesign.html.

INPUT:

- `n`, `k` – integers; if `k is None` it is set to the largest value available
- `resolvable` – boolean; set to `True` if you want the design to be resolvable (see *sage.combinat.designs.incidence_structures.IncidenceStructure.is_resolvable()*). The n classes of the resolvable design are obtained as the first n blocks, then the next n blocks, etc … Set to `False` by default.
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- `existence` – boolean; instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`)
  - `False` – meaning that the design does not exist

Note

When `k=None` and `existence=True` the function returns an integer, i.e. the largest k such that we can build a $TD(k, n)$.

OUTPUT: the kind of output depends on the input:

- if `existence=False` (the default) then the output is a list of lists that represent a $TD(k, n)$ with $V_1 = \{0, \dots, n-1\}, \dots, V_k = \{(k-1)n, \dots, kn-1\}$
- if `existence=True` and `k` is an integer, then the function returns a troolean: either `True`, `Unknown` or `False`

- if `existence=True` and `k=None` then the output is the largest value of `k` for which Sage knows how to compute a $TD(k, n)$.

See also

`orthogonal_array()` – a transversal design $TD(k, n)$ is equivalent to an orthogonal array $OA(k, n, 2)$.

EXAMPLES:

```
sage: TD = designs.transversal_design(5,5); TD
Transversal Design TD(5,5)
sage: TD.blocks()
[[0, 5, 10, 15, 20], [0, 6, 12, 18, 24], [0, 7, 14, 16, 23],
 [0, 8, 11, 19, 22], [0, 9, 13, 17, 21], [1, 5, 14, 18, 22],
 [1, 6, 11, 16, 21], [1, 7, 13, 19, 20], [1, 8, 10, 17, 24],
 [1, 9, 12, 15, 23], [2, 5, 13, 16, 24], [2, 6, 10, 19, 23],
 [2, 7, 12, 17, 22], [2, 8, 14, 15, 21], [2, 9, 11, 18, 20],
 [3, 5, 12, 19, 21], [3, 6, 14, 17, 20], [3, 7, 11, 15, 24],
 [3, 8, 13, 18, 23], [3, 9, 10, 16, 22], [4, 5, 11, 17, 23],
 [4, 6, 13, 15, 22], [4, 7, 10, 18, 21], [4, 8, 12, 16, 20],
 [4, 9, 14, 19, 24]]
```

```
>>> from sage.all import *
>>> TD = designs.transversal_design(Integer(5),Integer(5)); TD
Transversal Design TD(5,5)
>>> TD.blocks()
[[0, 5, 10, 15, 20], [0, 6, 12, 18, 24], [0, 7, 14, 16, 23],
 [0, 8, 11, 19, 22], [0, 9, 13, 17, 21], [1, 5, 14, 18, 22],
 [1, 6, 11, 16, 21], [1, 7, 13, 19, 20], [1, 8, 10, 17, 24],
 [1, 9, 12, 15, 23], [2, 5, 13, 16, 24], [2, 6, 10, 19, 23],
 [2, 7, 12, 17, 22], [2, 8, 14, 15, 21], [2, 9, 11, 18, 20],
 [3, 5, 12, 19, 21], [3, 6, 14, 17, 20], [3, 7, 11, 15, 24],
 [3, 8, 13, 18, 23], [3, 9, 10, 16, 22], [4, 5, 11, 17, 23],
 [4, 6, 13, 15, 22], [4, 7, 10, 18, 21], [4, 8, 12, 16, 20],
 [4, 9, 14, 19, 24]]
```

Some examples of the maximal number of transversal Sage is able to build:

```
sage: TD_4_10 = designs.transversal_design(4,10)
sage: designs.transversal_design(5,10,existence=True)
Unknown
```

```
>>> from sage.all import *
>>> TD_4_10 = designs.transversal_design(Integer(4),Integer(10))
>>> designs.transversal_design(Integer(5),Integer(10),existence=True)
Unknown
```

For prime powers, there is an explicit construction which gives a $TD(n + 1, n)$:

```
sage: designs.transversal_design(4, 3, existence=True)
True
sage: designs.transversal_design(674, 673, existence=True)
True
```

```
>>> from sage.all import *
>>> designs.transversal_design(Integer(4), Integer(3), existence=True)
True
>>> designs.transversal_design(Integer(674), Integer(673), existence=True)
True
```

For other values of `n` it depends:

```
sage: designs.transversal_design(7, 6, existence=True)
False
sage: designs.transversal_design(4, 6, existence=True)
Unknown
sage: designs.transversal_design(3, 6, existence=True)
True

sage: designs.transversal_design(11, 10, existence=True)
False
sage: designs.transversal_design(4, 10, existence=True)
True
sage: designs.transversal_design(5, 10, existence=True)
Unknown

sage: designs.transversal_design(7, 20, existence=True)
Unknown
sage: designs.transversal_design(6, 12, existence=True)
True
sage: designs.transversal_design(7, 12, existence=True)
True
sage: designs.transversal_design(8, 12, existence=True)
Unknown

sage: designs.transversal_design(6, 20, existence = True)
True
sage: designs.transversal_design(7, 20, existence = True)
Unknown
```

```
>>> from sage.all import *
>>> designs.transversal_design(Integer(7), Integer(6), existence=True)
False
>>> designs.transversal_design(Integer(4), Integer(6), existence=True)
Unknown
>>> designs.transversal_design(Integer(3), Integer(6), existence=True)
True

>>> designs.transversal_design(Integer(11), Integer(10), existence=True)
False
>>> designs.transversal_design(Integer(4), Integer(10), existence=True)
True
>>> designs.transversal_design(Integer(5), Integer(10), existence=True)
Unknown

>>> designs.transversal_design(Integer(7), Integer(20), existence=True)
Unknown
```

(continues on next page)

(continued from previous page)

```
>>> designs.transversal_design(Integer(6), Integer(12), existence=True)
True
>>> designs.transversal_design(Integer(7), Integer(12), existence=True)
True
>>> designs.transversal_design(Integer(8), Integer(12), existence=True)
Unknown

>>> designs.transversal_design(Integer(6), Integer(20), existence = True)
True
>>> designs.transversal_design(Integer(7), Integer(20), existence = True)
Unknown
```

If you ask for a transversal design that Sage is not able to build then an `EmptySetError` or a `NotImplementedError` is raised:

```
sage: designs.transversal_design(47, 100)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build a TD(47,100)!
sage: designs.transversal_design(55, 54)
Traceback (most recent call last):
...
EmptySetError: There exists no TD(55,54)!
```

```
>>> from sage.all import *
>>> designs.transversal_design(Integer(47), Integer(100))
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build a TD(47,100)!
>>> designs.transversal_design(Integer(55), Integer(54))
Traceback (most recent call last):
...
EmptySetError: There exists no TD(55,54)!
```

Those two errors correspond respectively to the cases where Sage answer `Unknown` or `False` when the parameter `existence` is set to `True`:

```
sage: designs.transversal_design(47, 100, existence=True)
Unknown
sage: designs.transversal_design(55, 54, existence=True)
False
```

```
>>> from sage.all import *
>>> designs.transversal_design(Integer(47), Integer(100), existence=True)
Unknown
>>> designs.transversal_design(Integer(55), Integer(54), existence=True)
False
```

If for a given n you want to know the largest k for which Sage is able to build a $TD(k, n)$ just call the function with k set to `None` and `existence` set to `True` as follows:

```
sage: designs.transversal_design(None, 6, existence=True)
3
sage: designs.transversal_design(None, 20, existence=True)
6
sage: designs.transversal_design(None, 30, existence=True)
6
sage: designs.transversal_design(None, 120, existence=True)
9
```

```
>>> from sage.all import *
>>> designs.transversal_design(None, Integer(6), existence=True)
3
>>> designs.transversal_design(None, Integer(20), existence=True)
6
>>> designs.transversal_design(None, Integer(30), existence=True)
6
>>> designs.transversal_design(None, Integer(120), existence=True)
9
```

sage.combinat.designs.orthogonal_arrays.**wilson_construction**(*OA*, *k*, *r*, *m*, *u*, *check=True*, *explain_construction=False*)

Return a $OA(k, rm + \sum_i u_i)$ from a truncated $OA(k + s, r)$ by Wilson's construction.

Simple form:

Let OA be a truncated $OA(k + s, r)$ with s truncated columns of sizes $u_1, ..., u_s$, whose blocks have sizes in $\{k + b_1, ..., k + b_t\}$. If there exist:

- An $OA(k, m + b_i) - b_i.OA(k, 1)$ for every $1 \leq i \leq t$
- An $OA(k, u_i)$ for every $1 \leq i \leq s$

Then there exists an $OA(k, rm + \sum u_i)$. The construction is a generalization of Lemma 3.16 in [HananiBIBD].

Brouwer-Van Rees form:

Let OA be a truncated $OA(k + s, r)$ with s truncated columns of sizes $u_1, ..., u_s$. Let the set H_i of the u_i points of column $k + i$ be partitionned into $\sum_j H_{ij}$. Let m_{ij} be integers such that:

- For $0 \leq i < l$ there exists an $OA(k, \sum_j m_{ij}|H_{ij}|)$
- For any block $B \in OA$ intersecting the sets $H_{ij(i)}$ there exists an $OA(k, m + \sum_i m_{ij}) - \sum_i OA(k, m_{ij(j)})$.

Then there exists an $OA(k, rm + \sum_{i,j} m_{ij})$. This construction appears in [BvR1982].

INPUT:

- `OA` – an incomplete orthogonal array with $k + s$ columns. The elements of a column of size c must belong to $\{0, ..., c\}$. The missing entries of a block are represented by `None` values. If `OA=None`, it is defined as a truncated orthogonal arrays with $k + s$ columns.
- `k`, `r`, `m` – integers
- `u` – list; two cases depending on the form to use:
  - Simple form: a list of length s such that column `k+i` has size `u[i]`. The untruncated points of column `k+i` are assumed to be `[0,...,u[i]-1]`.
  - Brouwer-Van Rees form: a list of length s such that `u[i]` is the list of pairs $(m_{i0}, |H_{i0}|), ..., (m_{ip_i}, |H_{ip_i}|)$. The untruncated points of column `k+i` are assumed to be $[0, ..., u_i - 1]$

where $u_i = \sum_j |H_{ip_i}|$. Besides, the first $|H_{i0}|$ points represent H_{i0}, the next $|H_{i1}|$ points represent H_{i1}, etc...

- `explain_construction` – boolean; return a string describing the construction
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.

REFERENCE:

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import wilson_construction
sage: from sage.combinat.designs.orthogonal_arrays import OA_relabel
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪wilson_decomposition_with_one_truncated_group
sage: total = 0
sage: for k in range(3,8):
....:     for n in range(1,30):
....:         if find_wilson_decomposition_with_one_truncated_group(k,n):
....:             total += 1
....:             f, args = find_wilson_decomposition_with_one_truncated_group(k,n)
....:             _ = f(*args)
sage: total
41

sage: print(designs.orthogonal_arrays.explain_construction(7,58))
Wilson's construction n=8.7+1+1 with master design OA(7+2,8)
sage: print(designs.orthogonal_arrays.explain_construction(9,115))
Wilson's construction n=13.8+11 with master design OA(9+1,13)
sage: print(wilson_construction(None,5,11,21,[[(5,5)]],explain_construction=True))
Brouwer-van Rees construction n=11.21+(5.5) with master design OA(5+1,11)
sage: print(wilson_construction(None,71,17,21,[[(4,9),(1,1)],[(9,9),(1,1)]],
↪explain_construction=True))
Brouwer-van Rees construction n=17.21+(9.4+1.1)+(9.9+1.1) with master design␣
↪OA(71+2,17)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import wilson_construction
>>> from sage.combinat.designs.orthogonal_arrays import OA_relabel
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪wilson_decomposition_with_one_truncated_group
>>> total = Integer(0)
>>> for k in range(Integer(3),Integer(8)):
...     for n in range(Integer(1),Integer(30)):
...         if find_wilson_decomposition_with_one_truncated_group(k,n):
...             total += Integer(1)
...             f, args = find_wilson_decomposition_with_one_truncated_group(k,n)
...             _ = f(*args)
>>> total
41

>>> print(designs.orthogonal_arrays.explain_construction(Integer(7),Integer(58)))
Wilson's construction n=8.7+1+1 with master design OA(7+2,8)
>>> print(designs.orthogonal_arrays.explain_construction(Integer(9),Integer(115)))
```

(continues on next page)

(continued from previous page)

```
Wilson's construction n=13.8+11 with master design OA(9+1,13)
>>> print(wilson_construction(None,Integer(5),Integer(11),Integer(21),
↪[[(Integer(5),Integer(5))]],explain_construction=True))
Brouwer-van Rees construction n=11.21+(5.5) with master design OA(5+1,11)
>>> print(wilson_construction(None,Integer(71),Integer(17),Integer(21),
↪[[(Integer(4),Integer(9)),(Integer(1),Integer(1))],[(Integer(9),Integer(9)),
↪(Integer(1),Integer(1))]],explain_construction=True))
Brouwer-van Rees construction n=17.21+(9.4+1.1)+(9.9+1.1) with master design␣
↪OA(71+2,17)
```

An example using the Brouwer-van Rees generalization:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.orthogonal_arrays import wilson_construction
sage: OA = designs.orthogonal_arrays.build(6,11)
sage: OA = [[x if (i<5 or x<5) else None for i,x in enumerate(R)] for R in OA]
sage: OAb = wilson_construction(OA,5,11,21,[[(5,5)]])
sage: is_orthogonal_array(OAb,5,256)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> from sage.combinat.designs.orthogonal_arrays import wilson_construction
>>> OA = designs.orthogonal_arrays.build(Integer(6),Integer(11))
>>> OA = [[x if (i<Integer(5) or x<Integer(5)) else None for i,x in enumerate(R)]␣
↪for R in OA]
>>> OAb = wilson_construction(OA,Integer(5),Integer(11),Integer(21),[[(Integer(5),
↪Integer(5))]])
>>> is_orthogonal_array(OAb,Integer(5),Integer(256))
True
```

5.1.92 Orthogonal arrays (build recursive constructions)

This module implements several constructions of *Orthogonal Arrays*. As their input can be complex, they all have a counterpart in the *orthogonal_arrays_find_recursive* module that automatically computes it.

All these constructions are automatically queried when the *orthogonal_array()* function is called.

construction_3_3()	Return an $OA(k, nm + i)$.
construction_3_4()	Return a $OA(k, nm + rs)$.
construction_3_5()	Return an $OA(k, nm + r + s + t)$.
construction_3_6()	Return a $OA(k, nm + i)$.
construction_q_x()	Return an $OA(k, (q-1) * (q-x) + x + 2)$ using the $q - x$ construction.
OA_and_oval()	Return a $OA(q + 1, q)$ whose blocks contains ≤ 2 zeroes in the last q columns.
thwart_lemma_3_5()	Return an $OA(k, nm + a + b + c + d)$.
thwart_lemma_4_1()	Return an $OA(k, nm + 4(n - 2))$.
three_factor_product()	Return an $OA(k + 1, n_1 n_2 n_3)$.
brouwer_separable_design()	Return a $OA(k, t(q^2 + q + 1) + x)$ using Brouwer's result on separable designs.

Functions

sage.combinat.designs.orthogonal_arrays_build_recursive.**OA_and_oval**(*q*, *solver*, *integrality_tolerance*)

Return a $OA(q+1,q)$ whose blocks contains ≤ 2 zeroes in the last q columns.

This OA is build from a projective plane of order q, in which there exists an oval O of size $q+1$ (i.e. a set of $q+1$ points no three of which are [colinear/contained in a common set of the projective plane]).

Removing an element $x\in O$ and all sets that contain it, we obtain a $TD(q+1,q)$ in which O intersects all columns except one. As O is an oval, no block of the TD intersects it more than twice.

INPUT:

- `q` – a prime power
- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`

Note: This function is called by *construction_3_6()*, an implementation of Construction 3.6 from [AC07].

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import OA_and_
↪oval
sage: _ = OA_and_oval
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import OA_and_
↪oval
>>> _ = OA_and_oval
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**brouwer_separable_design**(*k*, *t*, *q*, *x*, *check=False*, *verbose=False*, *explain_construction=False*)

Return a $OA(k,t(q^2+q+1)+x)$ using Brouwer's result on separable designs.

This method is an implementation of Brouwer's construction presented in [Brouwer80]. It consists in a systematic application of the usual transformation from PBD to OA, applied to a specific PBD.

Baer subplanes

When q is a prime power, the projective plane $PG(2,q^2)$ can be partitionned into subplanes $PG(2,q)$ (called Baer subplanes), giving $PG(2,q^2)=B_1\cup\cdots\cup B_{q^2-q+1}$. As a result, every line of the $PG(2,q^2)$ intersects one of the subplane on $q+1$ points and all others on 1 point.

The OA are built by considering $B_1 \cup \dots \cup B_t$, for a total of $t(q^2+q+1)$ points (to which x new points are then added). The blocks of this subdesign belong to two categories:

- The blocks of size t: they come from the lines which intersect a B_i on $q+1$ points for some $i > t$. The blocks of size t can be partitionned into $q^2-q+t-1$ parallel classes according to their associated subplane B_i with $i > t$.
- The blocks of size $q+t$: those blocks form a symmetric design, as every point is incident with $q+t$ of them.

Constructions

In the following, we write $N = t(q^2+q+1)+x$. The code is also heavily commented, and will clear any doubt.

- i) $x = 0$: in that case we build a resolvable $OA(k-1, N)$ that will then be completed into an $OA(k, N)$.
  - *Sets of size* t)

    We take the product of each parallel class with the parallel classes of a resolvable $OA(k-1,t) - t.OA(k-1,t)$, yielding new parallel classes.
  - *Sets of size* $q+t$)

    A $N \times (q+t)$ array is built whose rows are the sets of size $q+t$ such that every value appears once per column. For each block of a $OA(k-1, q+t) - (q+t).OA(k-1,t)$, the product with the rows of the matrix yields a parallel class.
- ii) $x = q + t$
  - *Sets of size* t)

    Each set of size t gives a $OA(k,t) - t.OA(k,1)$, except if there is only one parallel class in which case a $OA(k,t)$ is sufficient.
  - *Sets of size* $q+t$)

    A $(N-x) \times (q+t)$ array M is built whose $N-x$ rows are the sets of size $q+t$ such that every value appears once per column. For each of the new $x = q+t$ points $p_1, \dots, p_{q+t}$ we build a matrix M_i obtained from M by adding a column equal to $(p_i, p_i, p_i \dots)$. We add to the OA the product of all rows of the M_i with the block of the $x = q+t$ parallel classes of a resolvable $OA(k, t+q+1) - (t+q+1).OA(k,1)$.
  - *Set of size* x) An $OA(k,x)$
- iii) $x = q^2 - q + 1 - t$
  - *Sets of size* t)

    All blocks of the i-th parallel class are extended with the i-th new point. The blocks are then replaced by a $OA(k,t+1) - (t+1).OA(k,1)$ or, if there is only one parallel class (i.e. $x=1$) by a $OA(k,t+1) - OA(k,1)$.
  - *Set of size* $q+t$)

    They are replaced by $OA(k, q+t) - (q+t).OA(k,1)$.
  - *Set of size* x) An $OA(k,x)$
- iv) $x = q^2 + 1$
  - *Sets of size* t)

All blocks of the i-th parallel class are extended with the i-th new point (the other $x - q - t$ new points are not touched at this step). The blocks are then replaced by a $OA(k, t + 1) - (t + 1).OA(k, 1)$ or, if there is only one parallel class (i.e. $x = 1$) by a $OA(k, t + 1) - OA(k, 1)$.

- *Sets of size* $q + t$) Same as for ii)
- *Set of size* x) An $OA(k, x)$

- v) $0 < x < q^2 - q + 1 - t$
  - *Sets of size* t)

    The blocks of the first x parallel class are extended with the x new points, and replaced with $OA(k.t + 1) - (t + 1).OA(k, 1)$ or, if $x = 1$, by $OA(k.t + 1) - .OA(k, 1)$

    The blocks of the other parallel classes are replaced by $OA(k, t) - t.OA(k, t)$ or, if there is only one class left, by $OA(k, t) - OA(k, t)$
  - *Sets of size* $q + t$)

    They are replaced with $OA(k, q + t) - (q + t).OA(k, 1)$.
  - *Set of size* x) An $OA(k, x)$

- vi) $t + q < x < q^2 + 1$
  - *Sets of size* t) Same as in v) with an x equal to $x - q + t$.
  - *Sets of size* t) Same as in vii)
  - *Set of size* x) An $OA(k, x)$

INPUT:

- `k`, `t`, `q`, `x` – integers
- `check` – boolean (default: `False`); whether to check that output is correct before returning it
- `verbose` – boolean; whether to print some information on the construction and parameters being used
- `explain_construction` – boolean; return a string describing the construction

See also:

- *find_brouwer_separable_design()*

REFERENCES:

EXAMPLES:

Test all possible cases:

```
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import brouwer_
→separable_design
sage: k,q,t=4,4,3; _=brouwer_separable_design(k,q,t,0,verbose=True)
Case i) with k=4,q=3,t=4,x=0
sage: k,q,t=3,3,3; _=brouwer_separable_design(k,t,q,t+q,verbose=True,check=True)
Case ii) with k=3,q=3,t=3,x=6,e3=1
sage: k,q,t=3,3,6; _=brouwer_separable_design(k,t,q,t+q,verbose=True,check=True)
```

(continues on next page)

(continued from previous page)

```
Case ii) with k=3,q=3,t=6,x=9,e3=0
sage: k,q,t=3,3,6; _=brouwer_separable_design(k,t,q,q**2-q+1-t,verbose=True,
↪check=True)
Case iii) with k=3,q=3,t=6,x=1,e2=0
sage: k,q,t=3,4,6; _=brouwer_separable_design(k,t,q,q**2-q+1-t,verbose=True,
↪check=True)
Case iii) with k=3,q=4,t=6,x=7,e2=1
sage: k,q,t=3,4,6; _=brouwer_separable_design(k,t,q,q**2+1,verbose=True,
↪check=True)
Case iv) with k=3,q=4,t=6,x=17,e4=1
sage: k,q,t=3,2,2; _=brouwer_separable_design(k,t,q,q**2+1,verbose=True,
↪check=True)
Case iv) with k=3,q=2,t=2,x=5,e4=0
sage: k,q,t=3,4,7; _=brouwer_separable_design(k,t,q,3,verbose=True,check=True)
Case v) with k=3,q=4,t=7,x=3,e1=1,e2=1
sage: k,q,t=3,4,7; _=brouwer_separable_design(k,t,q,1,verbose=True,check=True)
Case v) with k=3,q=4,t=7,x=1,e1=1,e2=0
sage: k,q,t=3,4,7; _=brouwer_separable_design(k,t,q,q**2-q-t,verbose=True,
↪check=True)
Case v) with k=3,q=4,t=7,x=5,e1=0,e2=1
sage: k,q,t=5,4,7; _=brouwer_separable_design(k,t,q,t+q+3,verbose=True,check=True)
Case vi) with k=5,q=4,t=7,x=14,e3=1,e4=1
sage: k,q,t=5,4,8; _=brouwer_separable_design(k,t,q,t+q+1,verbose=True,check=True)
Case vi) with k=5,q=4,t=8,x=13,e3=1,e4=0
sage: k,q,t=5,4,8; _=brouwer_separable_design(k,t,q,q**2,verbose=True,check=True)
Case vi) with k=5,q=4,t=8,x=16,e3=0,e4=1

sage: print(designs.orthogonal_arrays.explain_construction(10,189))
Brouwer's separable design construction with t=9,q=4,x=0 from:
   Andries E. Brouwer,
   A series of separable designs with application to pairwise orthogonal Latin␣
↪squares
   Vol. 1, n. 1, pp. 39-41,
   European Journal of Combinatorics, 1980
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import brouwer_
↪separable_design
>>> k,q,t=Integer(4),Integer(4),Integer(3); _=brouwer_separable_design(k,q,t,
↪Integer(0),verbose=True)
Case i) with k=4,q=3,t=4,x=0
>>> k,q,t=Integer(3),Integer(3),Integer(3); _=brouwer_separable_design(k,t,q,t+q,
↪verbose=True,check=True)
Case ii) with k=3,q=3,t=3,x=6,e3=1
>>> k,q,t=Integer(3),Integer(3),Integer(6); _=brouwer_separable_design(k,t,q,t+q,
↪verbose=True,check=True)
Case ii) with k=3,q=3,t=6,x=9,e3=0
>>> k,q,t=Integer(3),Integer(3),Integer(6); _=brouwer_separable_design(k,t,q,
↪q**Integer(2)-q+Integer(1)-t,verbose=True,check=True)
Case iii) with k=3,q=3,t=6,x=1,e2=0
>>> k,q,t=Integer(3),Integer(4),Integer(6); _=brouwer_separable_design(k,t,q,
↪q**Integer(2)-q+Integer(1)-t,verbose=True,check=True)
```

(continues on next page)

(continued from previous page)

```
Case iii) with k=3,q=4,t=6,x=7,e2=1
>>> k,q,t=Integer(3),Integer(4),Integer(6); _=brouwer_separable_design(k,t,q,
↪q**Integer(2)+Integer(1),verbose=True,check=True)
Case iv) with k=3,q=4,t=6,x=17,e4=1
>>> k,q,t=Integer(3),Integer(2),Integer(2); _=brouwer_separable_design(k,t,q,
↪q**Integer(2)+Integer(1),verbose=True,check=True)
Case iv) with k=3,q=2,t=2,x=5,e4=0
>>> k,q,t=Integer(3),Integer(4),Integer(7); _=brouwer_separable_design(k,t,q,
↪Integer(3),verbose=True,check=True)
Case v) with k=3,q=4,t=7,x=3,e1=1,e2=1
>>> k,q,t=Integer(3),Integer(4),Integer(7); _=brouwer_separable_design(k,t,q,
↪Integer(1),verbose=True,check=True)
Case v) with k=3,q=4,t=7,x=1,e1=1,e2=0
>>> k,q,t=Integer(3),Integer(4),Integer(7); _=brouwer_separable_design(k,t,q,
↪q**Integer(2)-q-t,verbose=True,check=True)
Case v) with k=3,q=4,t=7,x=5,e1=0,e2=1
>>> k,q,t=Integer(5),Integer(4),Integer(7); _=brouwer_separable_design(k,t,q,
↪t+q+Integer(3),verbose=True,check=True)
Case vi) with k=5,q=4,t=7,x=14,e3=1,e4=1
>>> k,q,t=Integer(5),Integer(4),Integer(8); _=brouwer_separable_design(k,t,q,
↪t+q+Integer(1),verbose=True,check=True)
Case vi) with k=5,q=4,t=8,x=13,e3=1,e4=0
>>> k,q,t=Integer(5),Integer(4),Integer(8); _=brouwer_separable_design(k,t,q,
↪q**Integer(2),verbose=True,check=True)
Case vi) with k=5,q=4,t=8,x=16,e3=0,e4=1

>>> print(designs.orthogonal_arrays.explain_construction(Integer(10),
↪Integer(189)))
Brouwer's separable design construction with t=9,q=4,x=0 from:
   Andries E. Brouwer,
   A series of separable designs with application to pairwise orthogonal Latin␣
↪squares
   Vol. 1, n. 1, pp. 39-41,
   European Journal of Combinatorics, 1980
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_3_3**(*k*, *n*, *m*, *i*, *explain_construction=False*)

Return an $OA(k, nm + i)$.

This is Wilson's construction with i truncated columns of size 1 and such that a block B_0 of the incomplete OA intersects all truncated columns. As a consequence, all other blocks intersect only 0 or 1 of the last i columns. This allow to consider the block B_0 only up to its first k coordinates and then use a $OA(k, i)$ instead of a $OA(k, m + i) - i.OA(k, 1)$.

This is construction 3.3 from [AC07].

INPUT:

- `k, n, m, i` – integers such that the following designs are available: $OA(k, n)$, $OA(k, m)$, $OA(k, m + 1)$, $OA(k, r)$
- `explain_construction` – boolean; return a string describing the construction

See also

find_construction_3_3()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_3
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_3_3
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k = 11; n = 177
sage: is_orthogonal_array(construction_3_3(*find_construction_3_3(k,n)[1]),k,n,2)␣
↪          # needs sage.schemes
True

sage: print(designs.orthogonal_arrays.explain_construction(9,91))
Construction 3.3 with n=11,m=8,i=3 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_3
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_3_3
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> k = Integer(11); n = Integer(177)
>>> is_orthogonal_array(construction_3_3(*find_construction_3_3(k,n)[Integer(1)]),
↪k,n,Integer(2))           # needs sage.schemes
True

>>> print(designs.orthogonal_arrays.explain_construction(Integer(9),Integer(91)))
Construction 3.3 with n=11,m=8,i=3 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_3_4**(*k*, *n*, *m*, *r*, *s*, *explain_construction=False*)

Return a $OA(k, nm + rs)$.

This is Wilson's construction applied to a truncated $OA(k + r + 1, n)$ with r columns of size 1 and one column of size s.

The unique elements of the r truncated columns are picked so that a block B_0 contains them all.

- If there exists an $OA(k, m + r + 1)$ the column of size s is truncated in order to intersect B_0.
- Otherwise, if there exists an $OA(k, m + r)$, the last column must not intersect B_0

This is construction 3.4 from [AC07].

INPUT:

- k, n, m, r, s – integers; we assume that $s < n$ and $1 \leq r, s$

  The following designs must be available: $OA(k, n)$, $OA(k, m)$, $OA(k, m + 1)$, $OA(k, m + 2)$, $OA(k, s)$. Additionally, it requires either a $OA(k, m + r)$ or a $OA(k, m + r + 1)$.

- explain_construction – boolean; return a string describing the construction

See also

find_construction_3_4()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_4
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_3_4
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k = 8; n = 196
sage: is_orthogonal_array(construction_3_4(*find_construction_3_4(k,n)[1]),k,n,2)␣
↪          # needs sage.schemes
True

sage: print(designs.orthogonal_arrays.explain_construction(8,164))
Construction 3.4 with n=23,m=7,r=2,s=1 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_4
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_3_4
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> k = Integer(8); n = Integer(196)
>>> is_orthogonal_array(construction_3_4(*find_construction_3_4(k,n)[Integer(1)]),
↪k,n,Integer(2))            # needs sage.schemes
True

>>> print(designs.orthogonal_arrays.explain_construction(Integer(8),Integer(164)))
Construction 3.4 with n=23,m=7,r=2,s=1 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_3_5**(*k*, *n*, *m*, *r*, *s*, *t*, *explain_construction=False*)

Return an $OA(k, nm + r + s + t)$.

This is exactly Wilson's construction with three truncated groups except we make sure that all blocks have size $> k$, so we don't need a $OA(k, m + 0)$ but only $OA(k, m + 1)$, $OA(k, m + 2)$,`OA(k,m+3)`.

This is construction 3.5 from [AC07].

INPUT:

- `k`, `n`, `m` – integers
- `r`, `s`, `t` – integers; sizes of the three truncated groups, such that $r \leq s$ and $(q - r - 1)(q - s) \geq (q - s - 1) * (q - r)$
- `explain_construction` – boolean; return a string describing the construction

The following designs must be available : $OA(k, n)$, $OA(k, r)$, $OA(k, s)$, $OA(k, t)$, $OA(k, m+1)$, $OA(k, m+2)$, $OA(k, m + 3)$.

See also

find_construction_3_5()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_5
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_3_5
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k=8;n=111
sage: is_orthogonal_array(construction_3_5(*find_construction_3_5(k,n)[1]),k,n,2)␣
↪           # needs sage.schemes
True

sage: print(designs.orthogonal_arrays.explain_construction(8,90))
Construction 3.5 with n=11,m=6,r=8,s=8,t=8 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_5
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_3_5
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> k=Integer(8);n=Integer(111)
>>> is_orthogonal_array(construction_3_5(*find_construction_3_5(k,n)[Integer(1)]),
↪k,n,Integer(2))            # needs sage.schemes
True

>>> print(designs.orthogonal_arrays.explain_construction(Integer(8),Integer(90)))
Construction 3.5 with n=11,m=6,r=8,s=8,t=8 from:
```

(continues on next page)

(continued from previous page)

```
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_3_6**(*k*, *n*, *m*, *i*, *explain_construction=False*)

Return a $OA(k, nm + i)$.

This is Wilson's construction with r columns of order 1, in which each block intersects at most two truncated columns. Such a design exists when n is a prime power and is returned by `OA_and_oval()`.

INPUT:

- `k`, `n`, `m`, `i` – integers; n must be a prime power. The following designs must be available: $OA(k + r, q)$, $OA(k, m)$, $OA(k, m + 1)$, $OA(k, m + 2)$
- `explain_construction` – boolean; return a string describing the construction

This is construction 3.6 from [AC07].

See also

- `find_construction_3_6()`
- `OA_and_oval()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_6
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_3_6
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k=8;n=95
sage: is_orthogonal_array(construction_3_6(*find_construction_3_6(k,n)[1]),k,n,2)␣
↪           # needs sage.schemes
True

sage: print(designs.orthogonal_arrays.explain_construction(10,756))
Construction 3.6 with n=16,m=47,i=4 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_6
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_3_6
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
```

(continues on next page)

(continued from previous page)

```
>>> k=Integer(8);n=Integer(95)
>>> is_orthogonal_array(construction_3_6(*find_construction_3_6(k,n)[Integer(1)]),
↪k,n,Integer(2))            # needs sage.schemes
True

>>> print(designs.orthogonal_arrays.explain_construction(Integer(10),
↪Integer(756)))
Construction 3.6 with n=16,m=47,i=4 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_q_x**(*k*, *q*, *x*, *check=True*, *explain_construction=False*)

Return an $OA(k,(q-1)*(q-x)+x+2)$ using the $q-x$ construction.

Let $v=(q-1)*(q-x)+x+2$. If there exists a projective plane of order q (e.g. when q is a prime power) and $0<x<q$ then there exists a $(v-1,\{q-x-1,q-x+1\})$-GDD of type $(q-1)^{q-x}(x+1)^1$ (see [Greig99] or Theorem 2.50, section IV.2.3 of [DesignHandbook]). By adding to the ground set one point contained in all groups of the GDD, one obtains a $(v,\{q-x-1,q-x+1,q,x+2\})$-PBD with exactly one set of size $x+2$.

Thus, assuming that we have the following:

- $OA(k,q-x-1)-(q-x-1).OA(k,1)$
- $OA(k,q-x+1)-(q-x+1).OA(k,1)$
- $OA(k,q)-q.OA(k,1)$
- $OA(k,x+2)$

Then we can build from the PBD an $OA(k,v)$.

Construction of the PBD (shared by Julian R. Abel):

> Start with a resolvable $(q^2,q,1)$-BIBD and put the points into a $q\times q$ array so that rows form a parallel class and columns form another.
>
> Now delete:
>
> - All $x(q-1)$ points from the first x columns and not in the first row
> - All $q-x$ points in the last $q-x$ columns AND the first row.
>
> Then add a point p_1 to the blocks that are rows. Add a second point p_2 to the $q-x$ blocks that are columns of size $q-1$, plus the first row of size $x+1$.

INPUT:

- `k`, `q`, `x` – integers such that $0<x<q$ and such that Sage can build:
  - A projective plane of order q
  - $OA(k,q-x-1)-(q-x-1).OA(k,1)$
  - $OA(k,q-x+1)-(q-x+1).OA(k,1)$
  - $OA(k,q)-q.OA(k,1)$

- $OA(k, x+2)$

- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- `explain_construction` – boolean; return a string describing the construction

See also

- *find_q_x()*
- *projective_plane()*
- *orthogonal_array()*
- *OA_from_PBD()*

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_q_x
sage: _ = construction_q_x(9,16,6)                                          ␣
↪          # needs sage.schemes

sage: print(designs.orthogonal_arrays.explain_construction(9,158))
(q-x)-construction with q=16,x=6 from:
   Malcolm Greig,
   Designs from projective planes and PBD bases,
   vol. 7, num. 5, pp. 341--374,
   Journal of Combinatorial Designs, 1999
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import␣
↪construction_q_x
>>> _ = construction_q_x(Integer(9),Integer(16),Integer(6))                 ␣
↪                             # needs sage.schemes

>>> print(designs.orthogonal_arrays.explain_construction(Integer(9),Integer(158)))
(q-x)-construction with q=16,x=6 from:
   Malcolm Greig,
   Designs from projective planes and PBD bases,
   vol. 7, num. 5, pp. 341--374,
   Journal of Combinatorial Designs, 1999
```

REFERENCES:

`sage.combinat.designs.orthogonal_arrays_build_recursive.`**`three_factor_product`**(*k*, *n1*, *n2*, *n3*, *check=False*, *explain_construction=False*)

Return an $OA(k+1, n_1 n_2 n_3)$.

The three factor product construction from [DukesLing14] does the following:

> If $n_1 \leq n_2 \leq n_3$ are such that there exists an $OA(k, n_1)$, $OA(k+1, n_2)$ and $OA(k+1, n_3)$, then there exists a $OA(k+1, n_1n_2n_3)$.

It works with a modified product of orthogonal arrays ([Rees93], [Rees00]) which keeps track of parallel classes in the OA (the definition is given for transversal designs).

> A subset of blocks in an $TD(k, n)$ is called a c-parallel class if every point is covered exactly c times. A 1-parallel class is a parallel class.

The modified product:

> If there exists an $OA(k, n_1)$, and if there exists an $OA(k, n_2)$ whose blocks are partitionned into s n_1-parallel classes and $n_2 - sn_1$ parallel classes, then there exists an $OA(k, n_1n_2)$ whose blocks can be partitionned into sn_1^2 parallel classes and $(n_1n_2 - sn_1^2)/n_1 = n_2 - sn_1$ n_1-parallel classes.
>
> Proof:
>
> - The product of the blocks of a parallel class with an $OA(k, n_1)$ yields an n_1-parallel class of an $OA(k, n_1n_2)$.
> - The product of the blocks of a n_1-parallel class of $OA(k, n_2)$ with an $OA(k, n_1)$ can be done in such a way that it yields n_1n_2 parallel classes of $OA(k, n_1n_2)$. Those classes cover exactly the pairs that would have been covered with the usual product.
>
>   This can be achieved by simple cyclic permutations. Let us build the product of the n_1-parallel class $\mathcal{P} \subseteq OA(k, n_2)$ with $OA(k, n_1)$: when computing the product of $P \in \mathcal{P}$ with $B^1 \in OA(k, n_1)$ the i-th coordinate should not be (B_i^1, P_i) but $(B_i^1 + r, P_i)$ (the sum is mod n_1) where r is the number of blocks of $\mathcal{P}$ we have already processed whose i-th coordinate is equal to P_i (note that $r < n_1$ as $\mathcal{P}$ is n_1-parallel).

With these tools, one can obtain the designs promised by the three factors construction applied to k, n_1, n_2, n_3 (thanks to Julian R. Abel's help):

1) Let s be the largest integer $\leq n_3/n_1$. Apply the product construction to $OA(k, n_1)$ and a resolvable $OA(k, n_3)$ whose blocks are partitionned into s n_1-parallel classes and $n_3 - sn_1$ parallel classes. It results in a $OA(k, n_1n_3)$ partitionned into sn_1^2 parallel classes plus $(n_1n_3 - sn_1^2)/n_1 = n_3 - sn_1$ n_1-parallel classes.
2) Add $n_3 - n_1$ parallel classes to every n_1-parallel class to turn them into n_3-parallel classes. Apply the product construction to this partitionned $OA(k, n_1n_3)$ with a resolvable $OA(k, n_2)$.
3) As $OA(k, n_2)$ is resolvable, the n_2-parallel classes of $OA(k, n_1n_2n_3)$ are actually the union of n_2 parallel classes, thus the $OA(k, n_1n_2n_3)$ is resolvable and can be turned into an $OA(k+1, n_1n_2n_3)$

INPUT:

- `k`, `n1`, `n2`, `n3` – integers
- `check` – boolean; whether to check that everything is going smoothly while the design is being built. It is disabled by default, as the constructor of orthogonal arrays checks the final design anyway.
- `explain_construction` – boolean; return a string describing the construction

See also

- *find_three_factor_product()*

EXAMPLES:

```
sage: # needs sage.schemes
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import three_
↪factor_product
sage: OA = three_factor_product(4,4,4,4)
sage: is_orthogonal_array(OA,5,64)
True
sage: OA = three_factor_product(4,3,4,5)
sage: is_orthogonal_array(OA,5,60)
True
sage: OA = three_factor_product(5,4,5,7)
sage: is_orthogonal_array(OA,6,140)
True
sage: OA = three_factor_product(9,8,9,9) # long time
sage: is_orthogonal_array(OA,10,8*9*9)   # long time
True
sage: print(designs.orthogonal_arrays.explain_construction(10,648))
Three-factor product with n=8.9.9 from:
   Peter J. Dukes, Alan C.H. Ling,
   A three-factor product construction for mutually orthogonal latin squares,
   https://arxiv.org/abs/1401.1466
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import three_
↪factor_product
>>> OA = three_factor_product(Integer(4),Integer(4),Integer(4),Integer(4))
>>> is_orthogonal_array(OA,Integer(5),Integer(64))
True
>>> OA = three_factor_product(Integer(4),Integer(3),Integer(4),Integer(5))
>>> is_orthogonal_array(OA,Integer(5),Integer(60))
True
>>> OA = three_factor_product(Integer(5),Integer(4),Integer(5),Integer(7))
>>> is_orthogonal_array(OA,Integer(6),Integer(140))
True
>>> OA = three_factor_product(Integer(9),Integer(8),Integer(9),Integer(9)) # long␣
↪time
>>> is_orthogonal_array(OA,Integer(10),Integer(8)*Integer(9)*Integer(9))   # long␣
↪time
True
>>> print(designs.orthogonal_arrays.explain_construction(Integer(10),
↪Integer(648)))
Three-factor product with n=8.9.9 from:
   Peter J. Dukes, Alan C.H. Ling,
   A three-factor product construction for mutually orthogonal latin squares,
   https://arxiv.org/abs/1401.1466
```

REFERENCE:

sage.combinat.designs.orthogonal_arrays_build_recursive.**thwart_lemma_3_5**(*k*, *n*, *m*, *a*, *b*, *c*, *d=0*, *complement=False*, *explain_construction=False*)

Return an $OA(k, nm + a + b + c + d)$.

(When `d=0`)

According to [Thwarts] when n is a prime power and $a + b + c \leq n + 1$, one can build an $OA(k + 3, n)$ with three truncated columns of sizes a, b, c in such a way that all blocks have size $\leq k + 2$.

(in order to build a $OA(k, nm + a + b + c)$ the following designs must also exist: $OA(k, a)$, $OA(k, b)$, $OA(k, c)$, $OA(k, m + 0)$, $OA(k, m + 1)$, $OA(k, m + 2)$)

Considering the complement of each truncated column, it is also possible to build an $OA(k + 3, n)$ with three truncated columns of sizes a, b, c in such a way that all blocks have size $> k$ whenever $(n-a)+(n-b)+(n-c) \leq n + 1$.

(in order to build a $OA(k, nm + a + b + c)$ the following designs must also exist: $OA(k, a)$, $OA(k, b)$, $OA(k, c)$, $OA(k, m + 1)$, $OA(k, m + 2)$, $OA(k, m + 3)$)

Here is the proof of Lemma 3.5 from [Thwarts] enriched with explanations from Julian R. Abel:

> For any prime power n one can build $k - 1$ MOLS by associating to every nonzero $x \in \mathbb{F}_n$ the latin square:
>
> $$M_x(i, j) = i + x * j \text{ where } i, j \in \mathbb{F}_n$$
>
> In particular $M_1(i, j) = i + j$, whose n columns and lines are indexed by the elements of $\mathbb{F}_n$. If we order the elements of $\mathbb{F}_n$ as $0, 1, ..., n-1, x+0, ..., x+n-1, x^2+0, ...$ and reorder the columns and lines of M_1 accordingly, the top-left $a \times b$ squares contains at most $a + b - 1$ distinct symbols.

(When $d \neq 0$)

If there exists an $OA(k + 3, n)$ with three truncated columns of sizes a, b, c in such a way that all blocks have size $\leq k + 2$, by truncating arbitrarily another column to size d one obtains an OA with 4 truncated columns whose blocks miss at least one value. Thus, following the proof again one can build an $OA(k + 4)$ with four truncated columns of sizes a, b, c, d with blocks of size $\leq k + 3$.

(in order to build a $OA(k, nm + a + b + c + d)$ the following designs must also exist: $OA(k, a)$, $OA(k, b)$, $OA(k, c)$, $OA(k, d)$, $OA(k, m + 0)$, $OA(k, m + 1)$, $OA(k, m + 2)$, $OA(k, m + 3)$)

As before, this also shows that one can build an $OA(k + 4, n)$ with four truncated columns of sizes a, b, c, d in such a way that all blocks have size $> k$ whenever $(n - a) + (n - b) + (n - c) \leq n + 1$

(in order to build a $OA(k, nm+a+b+c+d)$ the following designs must also exist: $OA(k, n-a)$, $OA(k, n-b)$, $OA(k, n - c)$, $OA(k, d)$, $OA(k, m + 1)$, $OA(k, m + 2)$, $OA(k, m + 3)$, $OA(k, m + 4)$)

INPUT:

- `k`, `n`, `m`, `a`, `b`, `c`, `d` – integers which must satisfy the constraints above. In particular, $a + b + c \leq n + 1$ must hold By default, $d = 0$.
- `complement` – boolean; whether to complement the sets, i.e. follow the $n - a, n - b, n - c$ variant described above
- `explain_construction` – boolean; return a string describing the construction

See also

- *find_thwart_lemma_3_5()*

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import thwart_
↪lemma_3_5
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: OA = thwart_lemma_3_5(6,23,7,5,7,8)                                              ␣
↪          # needs sage.schemes
sage: is_orthogonal_array(OA,6,23*7+5+7+8,2)                                           ␣
↪          # needs sage.schemes
True

sage: print(designs.orthogonal_arrays.explain_construction(10,408))                    ␣
↪          # needs sage.schemes
Lemma 4.1 with n=13,m=28 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_build_recursive import thwart_
↪lemma_3_5
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array
>>> OA = thwart_lemma_3_5(Integer(6),Integer(23),Integer(7),Integer(5),Integer(7),
↪Integer(8))                                                         # needs sage.
↪schemes
>>> is_orthogonal_array(OA,Integer(6),
↪Integer(23)*Integer(7)+Integer(5)+Integer(7)+Integer(8),Integer(2))                   ␣
↪                                       # needs sage.schemes
True

>>> print(designs.orthogonal_arrays.explain_construction(Integer(10),
↪Integer(408)))                                # needs sage.schemes
Lemma 4.1 with n=13,m=28 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

With sets of parameters from [Thwarts]:

```
sage: l = [
....:     [11, 27, 78, 16, 17, 25, 0],
....:     [12, 19, 208, 11, 13, 16, 0],
....:     [12, 19, 208, 13, 13, 16, 0],
....:     [10, 13, 78, 9, 9, 13, 1],
....:     [10, 13, 79, 9, 9, 13, 1]]
sage: for k,n,m,a,b,c,d in l:                                         # not tested -
↪- too long
....:     OA = thwart_lemma_3_5(k,n,m,a,b,c,d,complement=True)
....:     assert is_orthogonal_array(OA,k,n*m+a+b+c+d,verbose=True)
```

(continues on next page)

(continued from previous page)

```
sage: print(designs.orthogonal_arrays.explain_construction(10,1046))
Lemma 3.5 with n=13,m=79,a=9,b=1,c=0,d=9 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

```
>>> from sage.all import *
>>> l = [
...     [Integer(11), Integer(27), Integer(78), Integer(16), Integer(17),␣
↪Integer(25), Integer(0)],
...     [Integer(12), Integer(19), Integer(208), Integer(11), Integer(13),␣
↪Integer(16), Integer(0)],
...     [Integer(12), Integer(19), Integer(208), Integer(13), Integer(13),␣
↪Integer(16), Integer(0)],
...     [Integer(10), Integer(13), Integer(78), Integer(9), Integer(9),␣
↪Integer(13), Integer(1)],
...     [Integer(10), Integer(13), Integer(79), Integer(9), Integer(9),␣
↪Integer(13), Integer(1)]]
>>> for k,n,m,a,b,c,d in l:                                 # not tested --␣
↪too long
...      OA = thwart_lemma_3_5(k,n,m,a,b,c,d,complement=True)
...      assert is_orthogonal_array(OA,k,n*m+a+b+c+d,verbose=True)

>>> print(designs.orthogonal_arrays.explain_construction(Integer(10),
↪Integer(1046)))
Lemma 3.5 with n=13,m=79,a=9,b=1,c=0,d=9 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

REFERENCE:

sage.combinat.designs.orthogonal_arrays_build_recursive.**thwart_lemma_4_1**(*k*, *n*, *m*, *explain_construction=False*)

Return an $OA(k, nm + 4(n - 2))$.

Implements Lemma 4.1 from [Thwarts].

> If $n \equiv 0, 1 \pmod 3$ is a prime power, then there exists a truncated $OA(n + 1, n)$ whose last four columns have size $n - 2$ and intersect every block on $1, 3$ or 4 values. Consequently, if there exists an $OA(k, m + 1)$, $OA(k, m + 3)$, $OA(k, m + 4)$ and a $OA(k, n - 2)$ then there exists an $OA(k, nm + 4(n - 2)$
>
> Proof: form the transversal design by removing one point of the $AG(2, 3)$ (Affine Geometry) contained in the Desarguesian Projective Plane $PG(2, n)$.

The affine geometry on 9 points contained in the projective geometry $PG(2, n)$ is given explicitly in [OS64] (Thanks to Julian R. Abel for finding the reference!).

INPUT:

- `k`, `n`, `m` – integers
- `explain_construction` – boolean; return a string describing the construction

> **See also**
>
> - *find_thwart_lemma_4_1()*

EXAMPLES:

```
sage: print(designs.orthogonal_arrays.explain_construction(10,408))
↪          # needs sage.schemes
Lemma 4.1 with n=13,m=28 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

```
>>> from sage.all import *
>>> print(designs.orthogonal_arrays.explain_construction(Integer(10),
↪Integer(408)))                              # needs sage.schemes
Lemma 4.1 with n=13,m=28 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

REFERENCES:

5.1.93 Orthogonal arrays (find recursive constructions)

This module implements several functions to find recursive constructions of *Orthogonal Arrays*.

The main function of this module, i.e. *find_recursive_construction()*, queries all implemented recursive constructions of designs implemented in *orthogonal_arrays_build_recursive* in order to obtain an $OA(k, n)$.

find_recursive_construction() is called by the *orthogonal_array()* function.

find_recursive_construction()	Find a recursive construction of an $OA(k, n)$ (calls all others `find_*` functions)
find_product_decomposition()	Find $n_1 n_2 = n$ to obtain an $OA(k, n)$ by the product construction
find_wilson_decomposition_with_one_truncated_group()	Find $rm + u = n$ to obtain an $OA(k, n)$ by Wilson's construction with one truncated column.
find_wilson_decomposition_with_two_truncated_groups()	Find $rm + r_1 + r_2 = n$ to obtain an $OA(k, n)$ by Wilson's construction with two truncated columns.
find_construction_3_3()	Find a decomposition for construction 3.3 from [AC07].
find_construction_3_4()	Find a decomposition for construction 3.4 from [AC07].
find_construction_3_5()	Find a decomposition for construction 3.5 from [AC07].
find_construction_3_6()	Find a decomposition for construction 3.6 from [AC07].
find_q_x()	Find integers q, x such that the $q - x$ construction yields an $OA(k, n)$.
find_thwart_lemma_3_5()	Find the values on which Lemma 3.5 from [Thwarts] applies.
find_thwart_lemma_4_1()	Find a decomposition for Lemma 4.1 from [Thwarts].
find_three_factor_product()	Find $n_1 n_2 n_3 = n$ to obtain an $OA(k, n)$ by the three-factor product from [DukesLing14]
find_brouwer_separable_design()	Find $t(q^2 + q + 1) + x = n$ to obtain an $OA(k, n)$ by Brouwer's separable design construction.
find_brouwer_van_rees_with_one_truncated_column()	Find $rm + x_1 + ... + x_c = n$ such that the Brouwer-van Rees constructions yields a $OA(k, n)$.

REFERENCES:

Functions

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_brouwer_separable_design**(*k*, *n*)

Find $t(q^2 + q + 1) + x = n$ to obtain an $OA(k, n)$ by Brouwer's separable design construction.

INPUT:

- `k`, `n` – integers

The assumptions made on the parameters t, q, x are explained in the documentation of *brouwer_separable_design()*.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪brouwer_separable_design
sage: find_brouwer_separable_design(5,13)[1]
(5, 1, 3, 0)
sage: find_brouwer_separable_design(5,14)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪brouwer_separable_design
>>> find_brouwer_separable_design(Integer(5),Integer(13))[Integer(1)]
(5, 1, 3, 0)
```

(continues on next page)

(continued from previous page)

```
>>> find_brouwer_separable_design(Integer(5),Integer(14))
False
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_brouwer_van_rees_with_one_truncated_colu**

Find $rm + x_1 + ... + x_c = n$ such that the Brouwer-van Rees constructions yields a $OA(k, n)$.

Let $n = rm + \sum_{1 \leq i \leq c}$ such that $c \leq r$. The generalization of Wilson's construction found by Brouwer and van Rees (with one truncated column) ensures that an $OA(k, n)$ exists if the following designs exist: $OA(k+1, r)$, $OA(k, m)$, $OA(k, \sum_{1 \leq i \leq c} u_i)$, $OA(k, m+x_1) - OA(k, x_1)$, ..., $OA(k, m+x_c) - OA(k, x_c)$.

For more information, see the documentation of `wilson_construction()`.

INPUT:

- `k`, `n` – integers

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪brouwer_van_rees_with_one_truncated_column
sage: find_brouwer_van_rees_with_one_truncated_column(5,53)[1]
(None, 5, 7, 7, [[(2, 1), (2, 1)]])
sage: find_brouwer_van_rees_with_one_truncated_column(6,96)[1]
(None, 6, 7, 13, [[(3, 1), (1, 1), (1, 1)]])
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪brouwer_van_rees_with_one_truncated_column
>>> find_brouwer_van_rees_with_one_truncated_column(Integer(5),
↪Integer(53))[Integer(1)]
(None, 5, 7, 7, [[(2, 1), (2, 1)]])
>>> find_brouwer_van_rees_with_one_truncated_column(Integer(6),
↪Integer(96))[Integer(1)]
(None, 6, 7, 13, [[(3, 1), (1, 1), (1, 1)]])
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_construction_3_3**(*k*, *n*)

Find a decomposition for construction 3.3 from [AC07].

INPUT:

- `k`, `n` – integers

See also

`construction_3_3()`

OUTPUT:

A pair `f,args` such that `f(*args)` returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_3
```

(continues on next page)

(continued from previous page)

```
sage: find_construction_3_3(11,177)[1]
(11, 11, 16, 1)
sage: find_construction_3_3(12,11)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_3
>>> find_construction_3_3(Integer(11),Integer(177))[Integer(1)]
(11, 11, 16, 1)
>>> find_construction_3_3(Integer(12),Integer(11))
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_construction_3_4**(k, n)

Find a decomposition for construction 3.4 from [AC07].

INPUT:

- k, n – integers

See also

construction_3_4()

OUTPUT:

A pair f,args such that f(*args) returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_4
sage: find_construction_3_4(8,196)[1]
(8, 25, 7, 12, 9)
sage: find_construction_3_4(9,24)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_4
>>> find_construction_3_4(Integer(8),Integer(196))[Integer(1)]
(8, 25, 7, 12, 9)
>>> find_construction_3_4(Integer(9),Integer(24))
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_construction_3_5**(k, n)

Find a decomposition for construction 3.5 from [AC07].

INPUT:

- k, n – integers

See also

construction_3_5()

OUTPUT:

A pair `f,args` such that `f(*args)` returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_5
sage: find_construction_3_5(8,111)[1]
(8, 13, 6, 9, 11, 13)
sage: find_construction_3_5(9,24)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_5
>>> find_construction_3_5(Integer(8),Integer(111))[Integer(1)]
(8, 13, 6, 9, 11, 13)
>>> find_construction_3_5(Integer(9),Integer(24))
```

`sage.combinat.designs.orthogonal_arrays_find_recursive.`**`find_construction_3_6`**`(k, n)`

Find a decomposition for construction 3.6 from [AC07].

INPUT:

- `k`, `n` – integers

See also

`construction_3_6()`

OUTPUT:

A pair `f,args` such that `f(*args)` returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_6
sage: find_construction_3_6(8,95)[1]
(8, 13, 7, 4)
sage: find_construction_3_6(8,98)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪construction_3_6
>>> find_construction_3_6(Integer(8),Integer(95))[Integer(1)]
(8, 13, 7, 4)
>>> find_construction_3_6(Integer(8),Integer(98))
```

`sage.combinat.designs.orthogonal_arrays_find_recursive.`**`find_product_decomposition`**`(k, n)`

Find $n_1 n_2 = n$ to obtain an $OA(k, n)$ by the product construction.

If Sage can build a $OA(k, n_1)$ and a $OA(k, n_2)$ such that $n = n_1 \times n_2$ then a $OA(k, n)$ can be built by a product construction (which correspond to Wilson's construction with no truncated column). This function look for a pair of integers (n_1, n_2) with $n1 \leq n_2, n_1 \times n_2 = n$ and such that both an $OA(k, n_1)$ and an $OA(k, n_2)$ are available.

INPUT:

- `k`, `n` – integers

OUTPUT:

A pair `f,args` such that `f(*args)` is an $OA(k, n)$ or `False` if no product decomposition was found.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪product_decomposition
sage: f,args = find_product_decomposition(6, 84)
sage: args
(None, 6, 7, 12, (), False)
sage: _ = f(*args)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪product_decomposition
>>> f,args = find_product_decomposition(Integer(6), Integer(84))
>>> args
(None, 6, 7, 12, (), False)
>>> _ = f(*args)
```

`sage.combinat.designs.orthogonal_arrays_find_recursive.`**`find_q_x`**`(k, n)`

Find integers q, x such that the $q - x$ construction yields an $OA(k, n)$.

See the documentation of `construction_q_x()` to find out what hypotheses the integers q, x must satisfy.

> **Warning**
>
> For efficiency reasons, this function checks that Sage can build an $OA(k+1, q-x-1)$ and an $OA(k+1, q-x+1)$, which is stronger than checking that Sage can build a $OA(k, q-x-1)-(q-x-1).OA(k, 1)$ and a $OA(k, q-x+1)-(q-x+1).OA(k, 1)$. The latter would trigger a lot of independent set computations in `sage.combinat.designs.orthogonal_arrays.incomplete_orthogonal_array()`.

INPUT:

- `k`, `n` – integers

> **See also**
>
> `construction_q_x()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_q_x
sage: find_q_x(10,9)
False
sage: find_q_x(9,158)[1]
(9, 16, 6)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_q_x
>>> find_q_x(Integer(10),Integer(9))
False
```

(continues on next page)

(continued from previous page)

```
>>> find_q_x(Integer(9),Integer(158))[Integer(1)]
(9, 16, 6)
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_recursive_construction**(n)

Find a recursive construction of an $OA(k, n)$ (calls all others `find_*` functions).

This determines whether an $OA(k, n)$ can be built through the following constructions:

- *wilson_construction()*
- *construction_3_3()*
- *construction_3_4()*
- *construction_3_5()*
- *construction_3_6()*
- *construction_q_x()*
- *thwart_lemma_3_5()*
- *thwart_lemma_4_1()*
- *three_factor_product()*
- *brouwer_separable_design()*

INPUT:

- `k`, `n` – integers

OUTPUT:

Return a pair `f,args` such that `f(*args)` returns the requested OA if possible, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪recursive_construction
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: count = 0
sage: for n in range(10,150):
....:     k = designs.orthogonal_arrays.largest_available_k(n)
....:     if find_recursive_construction(k,n):
....:         count = count + 1
....:         f,args = find_recursive_construction(k,n)
....:         OA = f(*args)
....:         assert is_orthogonal_array(OA,k,n,2,verbose=True)
sage: count
56
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪recursive_construction
>>> from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
>>> count = Integer(0)
>>> for n in range(Integer(10),Integer(150)):
...     k = designs.orthogonal_arrays.largest_available_k(n)
...     if find_recursive_construction(k,n):
```

(continues on next page)

(continued from previous page)

```
...         count = count + Integer(1)
...         f,args = find_recursive_construction(k,n)
...         OA = f(*args)
...         assert is_orthogonal_array(OA,k,n,Integer(2),verbose=True)
>>> count
56
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_three_factor_product**(k, n)

Find $n_1 n_2 n_3 = n$ to obtain an $OA(k, n)$ by the three-factor product from [DukesLing14].

INPUT:

- k, n – integers

See also

three_factor_product()

OUTPUT:

A pair f,args such that f(*args) returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪three_factor_product
sage: find_three_factor_product(10,648)[1]
(9, 8, 9, 9)
sage: find_three_factor_product(10,50)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_three_
↪factor_product
>>> find_three_factor_product(Integer(10),Integer(648))[Integer(1)]
(9, 8, 9, 9)
>>> find_three_factor_product(Integer(10),Integer(50))
False
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_thwart_lemma_3_5**(k, N)

Find the values on which Lemma 3.5 from [Thwarts] applies.

OUTPUT:

A pair (f,args) such that f(*args) returns an $OA(k, n)$ or False if the construction is not available.

See also

thwart_lemma_3_5()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪thwart_lemma_3_5
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array

sage: f,args = find_thwart_lemma_3_5(7,66)
sage: args
(7, 9, 7, 1, 1, 1, 0, False)
sage: OA = f(*args)
sage: is_orthogonal_array(OA,7,66,2)
True

sage: f,args = find_thwart_lemma_3_5(6,100)
sage: args
(6, 8, 10, 8, 7, 5, 0, True)
sage: OA = f(*args)
sage: is_orthogonal_array(OA,6,100,2)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪thwart_lemma_3_5
>>> from sage.combinat.designs.designs_pyx import is_orthogonal_array

>>> f,args = find_thwart_lemma_3_5(Integer(7),Integer(66))
>>> args
(7, 9, 7, 1, 1, 1, 0, False)
>>> OA = f(*args)
>>> is_orthogonal_array(OA,Integer(7),Integer(66),Integer(2))
True

>>> f,args = find_thwart_lemma_3_5(Integer(6),Integer(100))
>>> args
(6, 8, 10, 8, 7, 5, 0, True)
>>> OA = f(*args)
>>> is_orthogonal_array(OA,Integer(6),Integer(100),Integer(2))
True
```

Some values from [Thwarts]:

```
sage: kn = ((10,1046), (10,1048), (10,1059), (11,1524),
....:       (11,2164), (12,3362), (12,3992),  (12,3994))
sage: for k,n in kn:
....:     print("{} {} {}".format(k,n,find_thwart_lemma_3_5(k,n)[1]))
10 1046 (10, 13, 79, 9, 1, 0, 9, False)
10 1048 (10, 13, 79, 9, 1, 0, 11, False)
10 1059 (10, 13, 80, 9, 1, 0, 9, False)
11 1524 (11, 19, 78, 16, 13, 13, 0, True)
11 2164 (11, 27, 78, 23, 19, 16, 0, True)
12 3362 (12, 16, 207, 13, 13, 11, 13, True)
12 3992 (12, 19, 207, 16, 13, 11, 19, True)
12 3994 (12, 19, 207, 16, 13, 13, 19, True)

sage: for k,n in kn:                                                    # not␣
```

(continues on next page)

(continued from previous page)

```
↪tested -- too long
....:     assert designs.orthogonal_array(k,n,existence=True) is True
```

```
>>> from sage.all import *
>>> kn = ((Integer(10),Integer(1046)), (Integer(10),Integer(1048)), (Integer(10),
↪Integer(1059)), (Integer(11),Integer(1524)),
...       (Integer(11),Integer(2164)), (Integer(12),Integer(3362)), (Integer(12),
↪Integer(3992)),  (Integer(12),Integer(3994)))
>>> for k,n in kn:
...     print("{} {} {}".format(k,n,find_thwart_lemma_3_5(k,n)[Integer(1)]))
10 1046 (10, 13, 79, 9, 1, 0, 9, False)
10 1048 (10, 13, 79, 9, 1, 0, 11, False)
10 1059 (10, 13, 80, 9, 1, 0, 9, False)
11 1524 (11, 19, 78, 16, 13, 13, 0, True)
11 2164 (11, 27, 78, 23, 19, 16, 0, True)
12 3362 (12, 16, 207, 13, 13, 11, 13, True)
12 3992 (12, 19, 207, 16, 13, 11, 19, True)
12 3994 (12, 19, 207, 16, 13, 13, 19, True)

>>> for k,n in kn:                                                    # not␣
↪tested -- too long
...     assert designs.orthogonal_array(k,n,existence=True) is True
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_thwart_lemma_4_1**(*k*, *n*)

Find a decomposition for Lemma 4.1 from [Thwarts].

INPUT:

- k, n – integers

See also

thwart_lemma_4_1()

OUTPUT:

A pair f,args such that f(*args) returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪thwart_lemma_4_1
sage: find_thwart_lemma_4_1(10,408)[1]
(10, 13, 28)
sage: find_thwart_lemma_4_1(10,50)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪thwart_lemma_4_1
>>> find_thwart_lemma_4_1(Integer(10),Integer(408))[Integer(1)]
(10, 13, 28)
```

(continues on next page)

(continued from previous page)

```
>>> find_thwart_lemma_4_1(Integer(10),Integer(50))
False
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_wilson_decomposition_with_one_truncated_g**

Find $rm + u = n$ to obtain an $OA(k, n)$ by Wilson's construction with one truncated column.

This function looks for possible integers m, t, u satisfying that $mt + u = n$ and such that Sage knows how to build a $OA(k, m)$, $OA(k, m + 1)$, $OA(k + 1, t)$ and a $OA(k, u)$.

INPUT:

- `k`, `n` – integers

OUTPUT:

A pair `f,args` such that `f(*args)` is an $OA(k, n)$ or `False` if no decomposition with one truncated block was found.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪wilson_decomposition_with_one_truncated_group
sage: f,args = find_wilson_decomposition_with_one_truncated_group(4,38)
sage: args
(None, 4, 5, 7, (3,), False)
sage: _ = f(*args)

sage: find_wilson_decomposition_with_one_truncated_group(4,20)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪wilson_decomposition_with_one_truncated_group
>>> f,args = find_wilson_decomposition_with_one_truncated_group(Integer(4),
↪Integer(38))
>>> args
(None, 4, 5, 7, (3,), False)
>>> _ = f(*args)

>>> find_wilson_decomposition_with_one_truncated_group(Integer(4),Integer(20))
False
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_wilson_decomposition_with_two_truncated_g**

Find $rm + r_1 + r_2 = n$ to obtain an $OA(k, n)$ by Wilson's construction with two truncated columns.

Look for integers r, m, r_1, r_2 satisfying $n = rm + r_1 + r_2$ and $1 \leq r_1, r_2 < r$ and such that the following designs exist : $OA(k + 2, r)$, $OA(k, r1)$, $OA(k, r2)$, $OA(k, m)$, $OA(k, m + 1)$, $OA(k, m + 2)$.

INPUT:

- `k`, `n` – integers

OUTPUT:

A pair `f,args` such that `f(*args)` is an $OA(k, n)$ or `False` if no decomposition with two truncated blocks was found.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪wilson_decomposition_with_two_truncated_groups
sage: f,args = find_wilson_decomposition_with_two_truncated_groups(5,58)
sage: args
(None, 5, 7, 7, (4, 5), False)
sage: _ = f(*args)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import find_
↪wilson_decomposition_with_two_truncated_groups
>>> f,args = find_wilson_decomposition_with_two_truncated_groups(Integer(5),
↪Integer(58))
>>> args
(None, 5, 7, 7, (4, 5), False)
>>> _ = f(*args)
```

`sage.combinat.designs.orthogonal_arrays_find_recursive.`**`int_as_sum`**`(`*value*, *S*, *k_max*`)`

Return a tuple $(s_1, s_2, \ldots, s_k)$ of less then k_max elements of S such that $value = s_1 + s_2 + \ldots + s_k$. If there is no such tuples then the function returns `None`.

INPUT:

- `value` – integer
- `S` – list of integers
- `k_max` – integer

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import int_as_
↪sum
sage: D = int_as_sum(21,[5,12],100)
sage: for k in range(20,40):
....:     print("{} {}".format(k, int_as_sum(k,[5,12],100)))
20 (5, 5, 5, 5)
21 None
22 (12, 5, 5)
23 None
24 (12, 12)
25 (5, 5, 5, 5, 5)
26 None
27 (12, 5, 5, 5)
28 None
29 (12, 12, 5)
30 (5, 5, 5, 5, 5, 5)
31 None
32 (12, 5, 5, 5, 5)
33 None
34 (12, 12, 5, 5)
35 (5, 5, 5, 5, 5, 5, 5)
36 (12, 12, 12)
37 (12, 5, 5, 5, 5, 5)
```

(continues on next page)

(continued from previous page)

```
38 None
39 (12, 12, 5, 5, 5)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.orthogonal_arrays_find_recursive import int_as_sum
>>> D = int_as_sum(Integer(21),[Integer(5),Integer(12)],Integer(100))
>>> for k in range(Integer(20),Integer(40)):
...     print("{} {}".format(k, int_as_sum(k,[Integer(5),Integer(12)],
↪Integer(100))))
20 (5, 5, 5, 5)
21 None
22 (12, 5, 5)
23 None
24 (12, 12)
25 (5, 5, 5, 5, 5)
26 None
27 (12, 5, 5, 5)
28 None
29 (12, 12, 5)
30 (5, 5, 5, 5, 5, 5)
31 None
32 (12, 5, 5, 5, 5)
33 None
34 (12, 12, 5, 5)
35 (5, 5, 5, 5, 5, 5, 5)
36 (12, 12, 12)
37 (12, 5, 5, 5, 5, 5)
38 None
39 (12, 12, 5, 5, 5)
```

5.1.94 Steiner Quadruple Systems

A Steiner Quadruple System on n points is a family $SQS_n \subset \binom{[n]}{4}$ of 4-sets, such that any set $S \subset [n]$ of size three is a subset of exactly one member of SQS_n.

This module implements Haim Hanani's constructive proof that a Steiner Quadruple System exists if and only if $n \equiv 2, 4 \pmod 6$. Hanani's proof consists in 6 different constructions that build a large Steiner Quadruple System from a smaller one, and though it does not give a very clear understanding of why it works (to say the least)… it does !

The constructions have been implemented while reading two papers simultaneously, for one of them sometimes provides the informations that the other one does not. The first one is Haim Hanani's original paper [Han1960], and the other one is a paper from Horan and Hurlbert which goes through all constructions [HH2012].

It can be used through the `designs` object:

```
sage: designs.steiner_quadruple_system(8)
Incidence structure with 8 points and 14 blocks
```

```
>>> from sage.all import *
>>> designs.steiner_quadruple_system(Integer(8))
Incidence structure with 8 points and 14 blocks
```

AUTHORS:

- Nathann Cohen (May 2013, while listening to "*Le Blues Du Pauvre Delahaye*")

Index

This module's main function is the following:

steiner_quadruple_system()	Return a Steiner Quadruple System on n points

This function redistributes its work among 6 constructions:

Construction 1	*two_n()*	Return a Steiner Quadruple System on $2n$ points
Construction 2	*three_n_minus_two()*	Return a Steiner Quadruple System on $3n-2$ points
Construction 3	*three_n_minus_eight()*	Return a Steiner Quadruple System on $3n-8$ points
Construction 4	*three_n_minus_four()*	Return a Steiner Quadruple System on $3n-4$ points
Construction 5	*four_n_minus_six()*	Return a Steiner Quadruple System on $4n-6$ points
Construction 6	*twelve_n_minus_ten()*	Return a Steiner Quadruple System on $12n-10$ points

It also defines two specific Steiner Quadruple Systems that the constructions require, i.e. SQS_{14} and SQS_{38} as well as the systems of pairs $P_\alpha(m)$ and $\overline{P}_\alpha(m)$ (see [Han1960]).

Functions

`sage.combinat.designs.steiner_quadruple_systems.P(alpha, m)`

Return the collection of pairs $P_\alpha(m)$.

For more information on this system, see [Han1960].

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import P
sage: P(3,4)
[(0, 5), (2, 7), (4, 1), (6, 3)]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import P
>>> P(Integer(3),Integer(4))
[(0, 5), (2, 7), (4, 1), (6, 3)]
```

`sage.combinat.designs.steiner_quadruple_systems.barP(eps, m)`

Return the collection of pairs $\overline{P}_\alpha(m)$.

For more information on this system, see [Han1960].

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import barP
sage: barP(3,4)
[(0, 4), (3, 5), (1, 2)]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import barP
>>> barP(Integer(3),Integer(4))
[(0, 4), (3, 5), (1, 2)]
```

sage.combinat.designs.steiner_quadruple_systems.**barP_system**()

Return the 1-factorization of K_{2m} $\overline{P}(m)$.

For more information on this system, see [Han1960].

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import barP_system
sage: barP_system(3)
[[(4, 3), (2, 5)],
[(0, 5), (4, 1)],
[(0, 2), (1, 3)],
[(1, 5), (4, 2), (0, 3)],
[(0, 4), (3, 5), (1, 2)],
[(0, 1), (2, 3), (4, 5)]]
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import barP_system
>>> barP_system(Integer(3))
[[(4, 3), (2, 5)],
[(0, 5), (4, 1)],
[(0, 2), (1, 3)],
[(1, 5), (4, 2), (0, 3)],
[(0, 4), (3, 5), (1, 2)],
[(0, 1), (2, 3), (4, 5)]]
```

sage.combinat.designs.steiner_quadruple_systems.**four_n_minus_six**(B)

Return a Steiner Quadruple System on $4n - 6$ points.

INPUT:

- B – a Steiner Quadruple System on n points

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import four_n_minus_six
sage: for n in range(4, 20):
....:     if (n%6) in [2,4]:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not four_n_minus_six(sqs).is_t_design(3,4*n-6,4,1):
....:             print("Something is wrong !")
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import four_n_minus_six
>>> for n in range(Integer(4), Integer(20)):
...     if (n%Integer(6)) in [Integer(2),Integer(4)]:
...         sqs = designs.steiner_quadruple_system(n)
...         if not four_n_minus_six(sqs).is_t_design(Integer(3),Integer(4)*n-
↪Integer(6),Integer(4),Integer(1)):
...             print("Something is wrong !")
```

sage.combinat.designs.steiner_quadruple_systems.**relabel_system**(*B*)

Relabel the set so that $\{n-4, n-3, n-2, n-1\}$ is in B.

INPUT:

- B – list of 4-uples on $0, ..., n-1$

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import relabel_system
sage: SQS8 = designs.steiner_quadruple_system(8)
sage: relabel_system(SQS8)
Incidence structure with 8 points and 14 blocks
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import relabel_system
>>> SQS8 = designs.steiner_quadruple_system(Integer(8))
>>> relabel_system(SQS8)
Incidence structure with 8 points and 14 blocks
```

sage.combinat.designs.steiner_quadruple_systems.**steiner_quadruple_system**(*check=False*)

Return a Steiner Quadruple System on n points.

INPUT:

- n – integer such that $n \equiv 2, 4 \pmod 6$
- check – boolean (default: False); whether to check that the system is a Steiner Quadruple System before returning it

EXAMPLES:

```
sage: sqs4 = designs.steiner_quadruple_system(4)
sage: sqs4
Incidence structure with 4 points and 1 blocks
sage: sqs4.is_t_design(3,4,4,1)
True

sage: sqs8 = designs.steiner_quadruple_system(8)
sage: sqs8
Incidence structure with 8 points and 14 blocks
sage: sqs8.is_t_design(3,8,4,1)
True
```

```
>>> from sage.all import *
>>> sqs4 = designs.steiner_quadruple_system(Integer(4))
>>> sqs4
Incidence structure with 4 points and 1 blocks
>>> sqs4.is_t_design(Integer(3),Integer(4),Integer(4),Integer(1))
True

>>> sqs8 = designs.steiner_quadruple_system(Integer(8))
>>> sqs8
Incidence structure with 8 points and 14 blocks
>>> sqs8.is_t_design(Integer(3),Integer(8),Integer(4),Integer(1))
True
```

sage.combinat.designs.steiner_quadruple_systems.**three_n_minus_eight**(B)

Return a Steiner Quadruple System on $3n - 8$ points.

INPUT:

- B – a Steiner Quadruple System on n points

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_
↪eight
sage: for n in range(4, 30):
....:     if (n%12) == 2:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not three_n_minus_eight(sqs).is_t_design(3,3*n-8,4,1):
....:             print("Something is wrong !")
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_
↪eight
>>> for n in range(Integer(4), Integer(30)):
...     if (n%Integer(12)) == Integer(2):
...         sqs = designs.steiner_quadruple_system(n)
...         if not three_n_minus_eight(sqs).is_t_design(Integer(3),Integer(3)*n-
↪Integer(8),Integer(4),Integer(1)):
...             print("Something is wrong !")
```

sage.combinat.designs.steiner_quadruple_systems.**three_n_minus_four**(B)

Return a Steiner Quadruple System on $3n - 4$ points.

INPUT:

- B – a Steiner Quadruple System on n points where $n \equiv 10 \pmod{12}$

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_
↪four
sage: for n in range(4, 30):
....:     if n%12 == 10:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not three_n_minus_four(sqs).is_t_design(3,3*n-4,4,1):
....:             print("Something is wrong !")
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_four
>>> for n in range(Integer(4), Integer(30)):
...     if n%Integer(12) == Integer(10):
...         sqs = designs.steiner_quadruple_system(n)
...         if not three_n_minus_four(sqs).is_t_design(Integer(3),Integer(3)*n-
↪Integer(4),Integer(4),Integer(1)):
...             print("Something is wrong !")
```

sage.combinat.designs.steiner_quadruple_systems.**three_n_minus_two**(B)

Return a Steiner Quadruple System on $3n - 2$ points.

INPUT:

- B – a Steiner Quadruple System on n points

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_
↪two
sage: for n in range(4, 30):
....:     if (n%6) in [2,4]:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not three_n_minus_two(sqs).is_t_design(3,3*n-2,4,1):
....:             print("Something is wrong !")
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_two
>>> for n in range(Integer(4), Integer(30)):
...     if (n%Integer(6)) in [Integer(2),Integer(4)]:
...         sqs = designs.steiner_quadruple_system(n)
...         if not three_n_minus_two(sqs).is_t_design(Integer(3),Integer(3)*n-
↪Integer(2),Integer(4),Integer(1)):
...             print("Something is wrong !")
```

sage.combinat.designs.steiner_quadruple_systems.**twelve_n_minus_ten**(B)

Return a Steiner Quadruple System on $12n - 6$ points.

INPUT:

- B – a Steiner Quadruple System on n points

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import twelve_n_minus_
↪ten
sage: for n in range(4, 15):
....:     if (n%6) in [2,4]:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not twelve_n_minus_ten(sqs).is_t_design(3,12*n-10,4,1):
....:             print("Something is wrong !")
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import twelve_n_minus_ten
>>> for n in range(Integer(4), Integer(15)):
...     if (n%Integer(6)) in [Integer(2),Integer(4)]:
...         sqs = designs.steiner_quadruple_system(n)
...         if not twelve_n_minus_ten(sqs).is_t_design(Integer(3),Integer(12)*n-
↪Integer(10),Integer(4),Integer(1)):
...             print("Something is wrong !")
```

sage.combinat.designs.steiner_quadruple_systems.**two_n**(B)

Return a Steiner Quadruple System on $2n$ points.

INPUT:

- B – a Steiner Quadruple System on n points

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import two_n
sage: for n in range(4, 30):
....:     if (n%6) in [2,4]:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not two_n(sqs).is_t_design(3,2*n,4,1):
....:             print("Something is wrong !")
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.steiner_quadruple_systems import two_n
>>> for n in range(Integer(4), Integer(30)):
...     if (n%Integer(6)) in [Integer(2),Integer(4)]:
...         sqs = designs.steiner_quadruple_system(n)
...         if not two_n(sqs).is_t_design(Integer(3),Integer(2)*n,Integer(4),
↪Integer(1)):
...             print("Something is wrong !")
```

5.1.95 Hypergraph isomorphic copy search

This module implements a code for the following problem:

> **INPUT:** two hypergraphs H_1, H_2
>
> **OUTPUT:** a copy of H_2 in H_1

It is also possible to enumerate all such copies, and to require that such copies be induced copies. More formally:

> A copy of H_2 in H_1 is an injection $f : V(H_2) \mapsto V(H_1)$ such that for any set $S_2 \in E(H_2)$ we have $f(S_2) \in E(H_1)$.
>
> It is an *induced* copy if no other set of $E(H_1)$ is contained in $f(V(H_2))$, i.e. $|E(H_2)| = \{S : S \in E(H_1)$ and $S \subseteq f(V(H_2))\}$.

The functions implemented here lists all such injections. In particular, the number of copies of H in itself is equal to $|Aut(H)|$.

The feature is available through `IncidenceStructure.isomorphic_substructures_iterator()`.

Implementation

A hypergraph is stored as a list of edges, each of which is a "dense" bitset over $|V(H_1)|$ points. In particular, two sets of distinct cardinalities require the same memory space. A hypergraph is a C struct with the following fields:

- `n`, `m` – (`int`) number of points and edges
- `limbs` – (`int`) number of 64-bits blocks per set
- `set_space` – (`uint64_t *`) address of the memory used to store the sets
- `sets` – (`uint64_t **`) `sets[i]` points toward the `limbs` blocks encoding set i. Note also that `sets[i][limbs]` is equal to the cardinality of `set[i]`, so that `sets` has length `m*(limbs+1)*sizeof(uint64_t)`.
- `names` – (`int *`) associates an integer 'name' to each of the `n` points

The operations used on this data structure are:

- `void permute(hypergraph * h, int n1, int n2)` – exchanges points $n1$ and $n2$ in the data structure. Note that their names are also exchanged so that we still know which is which.

- `int induced_hypergraph(hypergraph * h1, int n, hypergraph * tmp1)` – stores in `tmp1` the hypergraph induced by the first n points, i.e. all sets S such that $S \subseteq \{0, ..., n-1\}$. The function returns the number of such sets.
- `void trace_hypergraph64(hypergraph * h, int n, hypergraph * tmp)` – stores in `tmp1` the trace of h on the first n points, i.e. all sets of the form $S \cap \{0, \ldots, n-1\}$.

Algorithm

We try all possible assignments of a representant $r_i \in H_1$ for every $i \in H_2$. When we have picked a representant for the first $n <$ points $\{0, \ldots, n-1\} \subsetneq V(H_2)$, we check that:

- The hypergraph induced by the (ordered) list $0, \ldots, n-1$ in H_2 is equal to the one induced by $r_0, \ldots, r_{n-1}$ in H_1.
- If $S \subseteq \{0, ..., n-1\}$ is contained in c sets of size k in H_2, then $\{r_i : i \in S\}$ is contained in $\geq c$ sets of size k in H_1. This is done by comparing the trace of the hypergraphs while remembering the original size of each set.

As we very often need to build the hypergraph obtained by the trace of the first n points (for all possible n), those hypergraphs are cached. The hypergraphs induced by the same points are handled similarly.

Limitations

Number of points For efficiency reason the implementation assumes that H_2 has ≤ 64 points. Making this work for larger values means that calls to `qsort` have to be replaced by calls to `qsort_r` (i.e. to sort the edges you need to know the number of limbs per edge) and that induces a big slowdown for small cases (~50% when this code was implemented). Also, 64 points for H_2 is already very very big considering the problem at hand. Even $|V(H_1)| > 64$ seems too much.

Vertex ordering The order of vertices in H_2 has a huge influence on the performance of the algorithm. If no set of H_2 contains more that one of the first $k < n$ points, then almost all partial assignments of representants are possible for the first k points (though the degree of the vertices is taken into account). For this reason it is best to pick an ordering such that the first vertices are contained in as many sets as possible together. A heuristic is implemented at *relabel_heuristic()*.

AUTHORS:

- Nathann Cohen (November 2014, written in various airports between Nice and Chennai).

Methods

class `sage.combinat.designs.subhypergraph_search.`**SubHypergraphSearch**

Bases: `object`

relabel_heuristic()

Relabel H_2 in order to make the algorithm faster.

Objective: we try to pick an ordering $p_1, ..., p_k$ of the points of H_2 that maximizes the number of sets involving the first points in the ordering. One way to formalize the problems indicates that it may be NP-Hard (generalizes the max clique problem for graphs) so we do not try to solve it exactly: we just need a sufficiently good heuristic.

Assuming that the first points are $p_1, ..., p_k$, we determine p_{k+1} as the point x such that the number of sets S with $x \in S$ and $S \cap \{p_1, ..., p_k\} \neq \emptyset$ is maximal. In case of ties, we take a point with maximum degree.

This function is called when an instance of *SubHypergraphSearch* is created.

EXAMPLES:

```
sage: d = designs.projective_plane(3)                                    #␣
↪needs sage.schemes
```

(continues on next page)

(continued from previous page)

```
sage: d.isomorphic_substructures_iterator(d).relabel_heuristic()          #␣
↪needs sage.schemes
```

```
>>> from sage.all import *
>>> d = designs.projective_plane(Integer(3))                                 ␣
↪     # needs sage.schemes
>>> d.isomorphic_substructures_iterator(d).relabel_heuristic()          #␣
↪needs sage.schemes
```

5.1.96 Two-graphs

A two-graph on n points is a family $T \subset \binom{[n]}{3}$ of 3-sets, such that any 4-set $S \subset [n]$ of size four contains an even number of elements of T. Any graph $([n], E)$ gives rise to a two-graph $T(E) = \{t \in \binom{[n]}{3} : \left|\binom{t}{2} \cap E\right| \ odd\}$, and any two graphs with the same two-graph can be obtained one from the other by `Seidel switching`. This defines an equivalence relation on the graphs on $[n]$, called Seidel switching equivalence. Conversely, given a two-graph T, one can construct a graph Γ in the corresponding Seidel switching class with an isolated vertex w. The graph $\Gamma \setminus w$ is called the *descendant* of T w.r.t. v.

T is called regular if each two-subset of $[n]$ is contained in the same number alpha of triples of T.

This module implements a direct construction of a two-graph from a list of triples, construction of descendant graphs, regularity checking, and other things such as constructing the complement two-graph, cf. [BH2012].

AUTHORS:

- Dima Pasechnik (Aug 2015)

Index

This module's methods are the following:

is_regular_twograph()	Test if `self` is a regular two-graph, i.e. a 2-design
complement()	Return the complement of `self`
descendant()	Return the descendant graph at w

This module's functions are the following:

taylor_twograph()	Construct Taylor's two-graph for $U_3(q)$
is_twograph()	Check that the incidence system is a two-graph
twograph_descendant()	Return the descendant graph w.r.t. a given vertex of the two-graph of a given graph

Methods

class sage.combinat.designs.twographs.**TwoGraph**(*points=None*, *blocks=None*, *incidence_matrix=None*, *name=None*, *check=False*, *copy=True*)

Bases: *IncidenceStructure*

Two-graphs class.

A two-graph on n points is a 3-uniform hypergraph, i.e. a family $T \subset \binom{[n]}{3}$ of 3-sets, such that any 4-set $S \subset [n]$ of size four contains an even number of elements of T. For more information, see the documentation of the *twographs* module.

complement()

The two-graph which is the complement of `self`.

That is, the two-graph consisting exactly of triples not in `self`. Note that this is different from *complement* of the *parent class*.

EXAMPLES:

```
sage: p = graphs.CompleteGraph(8).line_graph().twograph()                    #␣
↪needs sage.modules
sage: pc = p.complement(); pc                                                #␣
↪needs sage.modules
Incidence structure with 28 points and 1260 blocks
```

```
>>> from sage.all import *
>>> p = graphs.CompleteGraph(Integer(8)).line_graph().twograph()              ␣
↪     # needs sage.modules
>>> pc = p.complement(); pc                                                 #␣
↪needs sage.modules
Incidence structure with 28 points and 1260 blocks
```

descendant(*v*)

The descendant `graph` at `v`.

The *switching class of graphs* corresponding to `self` contains a graph `D` with `v` its own connected component; removing `v` from `D`, one obtains the descendant graph of `self` at `v`, which is constructed by this method.

INPUT:

- `v` – an element of `ground_set()`

EXAMPLES:

```
sage: p = graphs.PetersenGraph().twograph().descendant(0)                    #␣
↪needs sage.modules
sage: p.is_strongly_regular(parameters=True)                                 #␣
↪needs sage.modules
(9, 4, 1, 2)
```

```
>>> from sage.all import *
>>> p = graphs.PetersenGraph().twograph().descendant(Integer(0))              ␣
↪     # needs sage.modules
>>> p.is_strongly_regular(parameters=True)                                  #␣
↪needs sage.modules
(9, 4, 1, 2)
```

is_regular_twograph(*alpha=False*)

Test if the *TwoGraph* is regular, i.e. is a 2-design.

Namely, each pair of elements of `ground_set()` is contained in exactly `alpha` triples.

INPUT:

- `alpha` – boolean (default: `False`); return the value of `alpha`, if possible

EXAMPLES:

```
sage: # needs sage.modules
sage: p = graphs.PetersenGraph().twograph()
sage: p.is_regular_twograph(alpha=True)
4
sage: p.is_regular_twograph()
True
sage: p = graphs.PathGraph(5).twograph()
sage: p.is_regular_twograph(alpha=True)
False
sage: p.is_regular_twograph()
False
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> p = graphs.PetersenGraph().twograph()
>>> p.is_regular_twograph(alpha=True)
4
>>> p.is_regular_twograph()
True
>>> p = graphs.PathGraph(Integer(5)).twograph()
>>> p.is_regular_twograph(alpha=True)
False
>>> p.is_regular_twograph()
False
```

sage.combinat.designs.twographs.**is_twograph**(T)

Check whether the incidence system T is a two-graph.

INPUT:

- `T` – an `incidence structure`

EXAMPLES:

a two-graph from a graph:

```
sage: from sage.combinat.designs.twographs import (is_twograph, TwoGraph)
sage: p = graphs.PetersenGraph().twograph()                                    #␣
↪needs sage.modules
sage: is_twograph(p)                                                            #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.twographs import (is_twograph, TwoGraph)
>>> p = graphs.PetersenGraph().twograph()                                   #␣
↪needs sage.modules
>>> is_twograph(p)                                                           #␣
↪needs sage.modules
True
```

a non-regular 2-uniform hypergraph which is a two-graph:

```
sage: is_twograph(TwoGraph([[1,2,3],[1,2,4]]))
True
```

```
>>> from sage.all import *
>>> is_twograph(TwoGraph([[Integer(1),Integer(2),Integer(3)],[Integer(1),
↪Integer(2),Integer(4)]]))
True
```

sage.combinat.designs.twographs.**taylor_twograph**(*q*)

Constructing Taylor's two-graph for $U_3(q)$, q odd prime power.

The Taylor's two-graph T has the q^3+1 points of the projective plane over F_{q^2} singular w.r.t. the non-degenerate Hermitean form S preserved by $U_3(q)$ as its ground set; the triples are $\{x,y,z\}$ satisfying the condition that $S(x,y)S(y,z)S(z,x)$ is square (respectively non-square) if $q \cong 1 \mod 4$ (respectively if $q \cong 3 \mod 4$). See §7E of [BL1984].

There is also a $2-(q^3+1,q+1,1)$-design on these q^3+1 points, known as the unital of order q, also invariant under $U_3(q)$.

INPUT:

- `q` – a power of an odd prime

EXAMPLES:

```
sage: from sage.combinat.designs.twographs import taylor_twograph
sage: T = taylor_twograph(3); T                                         #␣
↪needs sage.rings.finite_rings
Incidence structure with 28 points and 1260 blocks
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.twographs import taylor_twograph
>>> T = taylor_twograph(Integer(3)); T                                  ␣
↪     # needs sage.rings.finite_rings
Incidence structure with 28 points and 1260 blocks
```

sage.combinat.designs.twographs.**twograph_descendant**(*G*, *v*, *name=None*)

Return the descendant graph w.r.t. vertex v of the two-graph of G.

In the *switching class* of G, construct a graph Δ with v an isolated vertex, and return the subgraph $\Delta \setminus v$. It is equivalent to, although much faster than, computing the *TwoGraph.descendant()* of two-graph of G, as the intermediate two-graph is not constructed.

INPUT:

- `G` – a graph
- `v` – a vertex of `G`
- `name` – (default: `None`) no name, otherwise derive from the construction

EXAMPLES:

one of s.r.g.'s from the database:

```
sage: from sage.combinat.designs.twographs import twograph_descendant
sage: A = graphs.strongly_regular_graph(280,135,70)                 # optional -
↪ gap_package_design internet
sage: twograph_descendant(A, 0).is_strongly_regular(parameters=True)   # optional -
↪ gap_package_design internet
(279, 150, 85, 75)
```

```
>>> from sage.all import *
>>> from sage.combinat.designs.twographs import twograph_descendant
>>> A = graphs.strongly_regular_graph(Integer(280),Integer(135),Integer(70))       ␣
↪            # optional - gap_package_design internet
>>> twograph_descendant(A, Integer(0)).is_strongly_regular(parameters=True)   #␣
↪optional - gap_package_design internet
(279, 150, 85, 75)
```

5.1.97 Combinatorial diagrams

A combinatorial diagram is a collection of cells (i, j) indexed by pairs of natural numbers.

For arbitrary diagrams, see `Diagram`. There are also two other specific types of diagrams implemented here. They are northwest diagrams (`NorthwestDiagram`) and Rothe diagrams (`RotheDiagram()`, a special kind of northwest diagram).

AUTHORS:

- Trevor K. Karn (2022-08-01): initial version

class sage.combinat.diagram.**Diagram**(*parent*, *cells*, *n_rows=None*, *n_cols=None*, *check=True*)

Bases: `ClonableArray`

Combinatorial diagrams with positions indexed by rows and columns.

The positions are indexed by rows and columns as in a matrix. For example, a Ferrers diagram is a diagram obtained from a partition $\lambda = (\lambda_0, \lambda_1, \ldots, \lambda_\ell)$, where the cells are in rows i for $0 \leq i \leq \ell$ and the cells in row i consist of (i, j) for $0 \leq j < \lambda_i$. In English notation, the indices are read from top left to bottom right as in a matrix.

Indexing conventions are the same as `Partition`. Printing the diagram of a partition, however, will always be in English notation.

EXAMPLES:

To create an arbitrary diagram, pass a list of all cells:

```
sage: from sage.combinat.diagram import Diagram
sage: cells = [(0,0), (0,1), (1,0), (1,1), (4,4), (4,5), (4,6), (5,4), (7, 6)]
sage: D = Diagram(cells); D
[(0, 0), (0, 1), (1, 0), (1, 1), (4, 4), (4, 5), (4, 6), (5, 4), (7, 6)]
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> cells = [(Integer(0),Integer(0)), (Integer(0),Integer(1)), (Integer(1),
↪Integer(0)), (Integer(1),Integer(1)), (Integer(4),Integer(4)), (Integer(4),
↪Integer(5)), (Integer(4),Integer(6)), (Integer(5),Integer(4)), (Integer(7),␣
↪Integer(6))]
>>> D = Diagram(cells); D
[(0, 0), (0, 1), (1, 0), (1, 1), (4, 4), (4, 5), (4, 6), (5, 4), (7, 6)]
```

We can visualize the diagram by printing `O`'s and `.`'s. `O`'s are present in the cells which are present in the diagram and a `.` represents the absence of a cell in the diagram:

```
sage: D.pp()
O O . . . . .
O O . . . . .
. . . . . . .
```

(continues on next page)

(continued from previous page)

```
. . . . . . .
. . . . O O O
. . . . O . .
. . . . . . .
. . . . . . O
```

```
>>> from sage.all import *
>>> D.pp()
O O . . . . .
O O . . . . .
. . . . . . .
. . . . . . .
. . . . O O O
. . . . O . .
. . . . . . .
. . . . . . O
```

We can also check if certain cells are contained in a given diagram:

```
sage: (1, 0) in D
True
sage: (2, 2) in D
False
```

```
>>> from sage.all import *
>>> (Integer(1), Integer(0)) in D
True
>>> (Integer(2), Integer(2)) in D
False
```

If you know that there are entire empty rows or columns at the end of the diagram, you can manually pass them with keyword arguments `n_rows=` or `n_cols=`:

```
sage: Diagram([(0,0), (0,3), (2,2), (2,4)]).pp()
O . . O .
. . . . .
. . O . O
sage: Diagram([(0,0), (0,3), (2,2), (2,4)], n_rows=6, n_cols=6).pp()
O . . O . .
. . . . . .
. . O . O .
. . . . . .
. . . . . .
. . . . . .
```

```
>>> from sage.all import *
>>> Diagram([(Integer(0),Integer(0)), (Integer(0),Integer(3)), (Integer(2),
↪Integer(2)), (Integer(2),Integer(4))]).pp()
O . . O .
. . . . .
. . O . O
>>> Diagram([(Integer(0),Integer(0)), (Integer(0),Integer(3)), (Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)), (Integer(2),Integer(4))], n_rows=Integer(6), n_cols=Integer(6)).
↪pp()
O . . O . .
. . . . . .
. . O . O .
. . . . . .
. . . . . .
. . . . . .
```

cells()

Return a list of the cells contained in the diagram self.

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagram
sage: D1 = Diagram([(0,2),(0,3),(1,1),(3,2)])
sage: D1.cells()
[(0, 2), (0, 3), (1, 1), (3, 2)]
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D1 = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))])
>>> D1.cells()
[(0, 2), (0, 3), (1, 1), (3, 2)]
```

check()

Check that this is a valid diagram.

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,0), (0,3), (2,2), (2,4)])
sage: D.check()
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(0)), (Integer(0),Integer(3)),␣
↪(Integer(2),Integer(2)), (Integer(2),Integer(4))])
>>> D.check()
```

In the next two examples, a bad diagram is passed. The first example fails because one cell is indexed by negative integers:

```
sage: D = Diagram([(0,0), (0,-3), (2,2), (2,4)])
Traceback (most recent call last):
...
ValueError: diagrams must be indexed by nonnegative integers
```

```
>>> from sage.all import *
>>> D = Diagram([(Integer(0),Integer(0)), (Integer(0),-Integer(3)),␣
↪(Integer(2),Integer(2)), (Integer(2),Integer(4))])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: diagrams must be indexed by nonnegative integers
```

The next example fails because one cell is indexed by rational numbers:

```
sage: D = Diagram([(0,0), (0,3), (2/3,2), (2,4)])
Traceback (most recent call last):
...
ValueError: diagrams must be indexed by nonnegative integers
```

```
>>> from sage.all import *
>>> D = Diagram([(Integer(0),Integer(0)), (Integer(0),Integer(3)),
→(Integer(2)/Integer(3),Integer(2)), (Integer(2),Integer(4))])
Traceback (most recent call last):
...
ValueError: diagrams must be indexed by nonnegative integers
```

essential_set()

Return the essential set of `self` as defined by Fulton.

Let D be a diagram. Then the *essential set* of D are the cells $(i, j) \in D$ such that $(i + 1, j) \notin D$ and $(i, j + 1) \notin D$; that is, the maximally southwest elements in each connected component of D.

EXAMPLES:

```
sage: w = Permutation([2, 1, 5, 4, 3])
sage: D = w.rothe_diagram()
sage: D.essential_set()
((0, 0), (2, 3), (3, 2))
```

```
>>> from sage.all import *
>>> w = Permutation([Integer(2), Integer(1), Integer(5), Integer(4),
→Integer(3)])
>>> D = w.rothe_diagram()
>>> D.essential_set()
((0, 0), (2, 3), (3, 2))
```

n_cells()

Return the total number of cells contained in the diagram `self`.

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagram
sage: D1 = Diagram([(0,2),(0,3),(1,1),(3,2)])
sage: D1.number_of_cells()
4
sage: D1.n_cells()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D1 = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)),(Integer(3),Integer(2))])
>>> D1.number_of_cells()
4
>>> D1.n_cells()
4
```

ncols()

Return the total number of rows of `self`.

EXAMPLES:

The following example has three columns which are filled, but they are contained in rows 0 to 3 (for a total of four):

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,2),(0,3),(1,1),(3,2)])
sage: D.number_of_cols()
4
sage: D.ncols()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))])
>>> D.number_of_cols()
4
>>> D.ncols()
4
```

We can also include empty columns at the end:

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,2),(0,3),(1,1),(3,2)], n_cols=6)
sage: D.number_of_cols()
6
sage: D.pp()
. . O O . .
. O . . . .
. . . . . .
. . O . . .
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))], n_cols=Integer(6))
>>> D.number_of_cols()
6
>>> D.pp()
. . O O . .
. O . . . .
. . . . . .
. . O . . .
```

nrows()

Return the total number of rows of `self`.

EXAMPLES:

The following example has three rows which are filled, but they are contained in rows 0 to 3 (for a total of four):

```
sage: from sage.combinat.diagram import Diagram
sage: D1 = Diagram([(0,2),(0,3),(1,1),(3,2)])
sage: D1.number_of_rows()
4
sage: D1.nrows()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D1 = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))])
>>> D1.number_of_rows()
4
>>> D1.nrows()
4
```

The total number of rows includes including those which are empty. We can also include empty rows at the end:

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,2),(0,3),(1,1),(3,2)], n_rows=6)
sage: D.number_of_rows()
6
sage: D.pp()
. . O O
. O . .
. . . .
. . O .
. . . .
. . . .
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))], n_rows=Integer(6))
>>> D.number_of_rows()
6
>>> D.pp()
. . O O
. O . .
. . . .
. . O .
. . . .
. . . .
```

number_of_cells()

Return the total number of cells contained in the diagram `self`.

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagram
sage: D1 = Diagram([(0,2),(0,3),(1,1),(3,2)])
sage: D1.number_of_cells()
4
sage: D1.n_cells()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D1 = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))])
>>> D1.number_of_cells()
4
>>> D1.n_cells()
4
```

number_of_cols()

Return the total number of rows of `self`.

EXAMPLES:

The following example has three columns which are filled, but they are contained in rows 0 to 3 (for a total of four):

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,2),(0,3),(1,1),(3,2)])
sage: D.number_of_cols()
4
sage: D.ncols()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))])
>>> D.number_of_cols()
4
>>> D.ncols()
4
```

We can also include empty columns at the end:

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,2),(0,3),(1,1),(3,2)], n_cols=6)
sage: D.number_of_cols()
6
sage: D.pp()
. . O O . .
. O . . . .
. . . . . .
. . O . . .
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))], n_cols=Integer(6))
>>> D.number_of_cols()
6
>>> D.pp()
. . O O . .
. O . . . .
. . . . . .
. . O . . .
```

number_of_rows()

Return the total number of rows of `self`.

EXAMPLES:

The following example has three rows which are filled, but they are contained in rows 0 to 3 (for a total of four):

```
sage: from sage.combinat.diagram import Diagram
sage: D1 = Diagram([(0,2),(0,3),(1,1),(3,2)])
sage: D1.number_of_rows()
4
sage: D1.nrows()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D1 = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))])
>>> D1.number_of_rows()
4
>>> D1.nrows()
4
```

The total number of rows includes including those which are empty. We can also include empty rows at the end:

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,2),(0,3),(1,1),(3,2)], n_rows=6)
sage: D.number_of_rows()
6
sage: D.pp()
. . O O
. O . .
. . . .
. . O .
. . . .
. . . .
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
```

(continues on next page)

(continued from previous page)

```
>>> D = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))], n_rows=Integer(6))
>>> D.number_of_rows()
6
>>> D.pp()
. . O O
. O . .
. . . .
. . O .
. . . .
. . . .
```

pp()

Return a visualization of the diagram.

Cells which are present in the diagram are filled with a `O`. Cells which are not present in the diagram are filled with a `.`.

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagram
sage: Diagram([(0,0), (0,3), (2,2), (2,4)]).pp()
O . . O .
. . . . .
. . O . O
sage: Diagram([(0,0), (0,3), (2,2), (2,4)], n_rows=6, n_cols=6).pp()
O . . O . .
. . . . . .
. . O . O .
. . . . . .
. . . . . .
. . . . . .
sage: Diagram([]).pp()
-
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> Diagram([(Integer(0),Integer(0)), (Integer(0),Integer(3)), (Integer(2),
↪Integer(2)), (Integer(2),Integer(4))]).pp()
O . . O .
. . . . .
. . O . O
>>> Diagram([(Integer(0),Integer(0)), (Integer(0),Integer(3)), (Integer(2),
↪Integer(2)), (Integer(2),Integer(4))], n_rows=Integer(6), n_
↪cols=Integer(6)).pp()
O . . O . .
. . . . . .
. . O . O .
. . . . . .
. . . . . .
. . . . . .
>>> Diagram([]).pp()
-
```

size()

Return the total number of cells contained in the diagram `self`.

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagram
sage: D1 = Diagram([(0,2),(0,3),(1,1),(3,2)])
sage: D1.number_of_cells()
4
sage: D1.n_cells()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D1 = Diagram([(Integer(0),Integer(2)),(Integer(0),Integer(3)),(Integer(1),
↪Integer(1)),(Integer(3),Integer(2))])
>>> D1.number_of_cells()
4
>>> D1.n_cells()
4
```

specht_module(*base_ring=None*)

Return the Specht module corresponding to `self`.

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,0), (1,1), (2,2), (2,3)])
sage: SM = D.specht_module(QQ)                                          #␣
↪needs sage.modules
sage: s = SymmetricFunctions(QQ).s()                                    #␣
↪needs sage.modules
sage: s(SM.frobenius_image())                                           #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + 2*s[3, 1] + s[4]
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(2)), (Integer(2),Integer(3))])
>>> SM = D.specht_module(QQ)                                            #␣
↪needs sage.modules
>>> s = SymmetricFunctions(QQ).s()                                      #␣
↪needs sage.modules
>>> s(SM.frobenius_image())                                             #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + 2*s[3, 1] + s[4]
```

specht_module_dimension(*base_ring=None*)

Return the dimension of the Specht module corresponding to `self`.

INPUT:

- `base_ring` – (default: **Q**) the base ring

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,0), (1,1), (2,2), (2,3)])
sage: D.specht_module_dimension()                                          #␣
↪needs sage.modules
12
sage: D.specht_module(QQ).dimension()                                      #␣
↪needs sage.modules
12
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(2)), (Integer(2),Integer(3))])
>>> D.specht_module_dimension()                                          #␣
↪needs sage.modules
12
>>> D.specht_module(QQ).dimension()                                      #␣
↪needs sage.modules
12
```

class sage.combinat.diagram.**Diagrams**(*category=None*)

Bases: UniqueRepresentation, Parent

The class of combinatorial diagrams.

A *combinatorial diagram* is a set of cells indexed by pairs of natural numbers. Calling an instance of *Diagrams* is one way to construct diagrams.

EXAMPLES:

```
sage: from sage.combinat.diagram import Diagrams
sage: Dgms = Diagrams()
sage: D = Dgms([(0,0), (0,3), (2,2), (2,4)])
sage: D.parent()
Combinatorial diagrams
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagrams
>>> Dgms = Diagrams()
>>> D = Dgms([(Integer(0),Integer(0)), (Integer(0),Integer(3)), (Integer(2),
↪Integer(2)), (Integer(2),Integer(4))])
>>> D.parent()
Combinatorial diagrams
```

Element

alias of *Diagram*

from_composition(*alpha*)

Create the diagram corresponding to a weak composition $\alpha \vDash n$.

EXAMPLES:

```
sage: alpha = Composition([3,0,2,1,4,4])
sage: from sage.combinat.diagram import Diagrams
```

(continues on next page)

(continued from previous page)

```
sage: Diagrams()(alpha).pp()
O O O .
. . . .
O O . .
O . . .
O O O O
O O O O
sage: Diagrams().from_composition(alpha).pp()
O O O .
. . . .
O O . .
O . . .
O O O O
O O O O
```

```
>>> from sage.all import *
>>> alpha = Composition([Integer(3),Integer(0),Integer(2),Integer(1),
↪Integer(4),Integer(4)])
>>> from sage.combinat.diagram import Diagrams
>>> Diagrams()(alpha).pp()
O O O .
. . . .
O O . .
O . . .
O O O O
O O O O
>>> Diagrams().from_composition(alpha).pp()
O O O .
. . . .
O O . .
O . . .
O O O O
O O O O
```

from_polyomino(*p*)

Create the diagram corresponding to a 2d `Polyomino`

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino                              #␣
↪needs sage.modules
sage: p = Polyomino([(0,0),(1,0),(1,1),(1,2)])                               #␣
↪needs sage.modules
sage: from sage.combinat.diagram import Diagrams
sage: Diagrams()(p).pp()                                                      #␣
↪needs sage.modules
O . .
O O O
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino                              #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
>>> p = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(1),Integer(1)),(Integer(1),Integer(2))])                    ␣
↪   # needs sage.modules
>>> from sage.combinat.diagram import Diagrams
>>> Diagrams()(p).pp()                                                #␣
↪needs sage.modules
O . .
O O O
```

We can also call this method directly:

```
sage: Diagrams().from_polyomino(p).pp()                               #␣
↪needs sage.modules
O . .
O O O
```

```
>>> from sage.all import *
>>> Diagrams().from_polyomino(p).pp()                                 #␣
↪needs sage.modules
O . .
O O O
```

This only works for a 2d `Polyomino`:

```
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')     #␣
↪needs sage.modules
sage: Diagrams().from_polyomino(p)                                    #␣
↪needs sage.modules
Traceback (most recent call last):
...
ValueError: the polyomino must be 2 dimensional
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(1),Integer(1),Integer(0)), (Integer(1),Integer(1),
↪Integer(1))], color='blue')      # needs sage.modules
>>> Diagrams().from_polyomino(p)                                      #␣
↪needs sage.modules
Traceback (most recent call last):
...
ValueError: the polyomino must be 2 dimensional
```

from_zero_one_matrix(*M*, *check=True*)

Get a diagram from a matrix with entries in $\{0, 1\}$, where positions of cells are indicated by the 1's.

EXAMPLES:

```
sage: M = matrix([[1,0,1,1],[0,1,1,0]])                               #␣
↪needs sage.modules
sage: from sage.combinat.diagram import Diagrams
sage: Diagrams()(M).pp()                                              #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
O . O O
. O O .
sage: Diagrams().from_zero_one_matrix(M).pp()                                 #␣
↪needs sage.modules
O . O O
. O O .

sage: M = matrix([[1, 0, 0], [1, 0, 0], [0, 0, 0]])                           #␣
↪needs sage.modules
sage: Diagrams()(M).pp()                                                       #␣
↪needs sage.modules
O . .
O . .
. . .
```

```
>>> from sage.all import *
>>> M = matrix([[Integer(1),Integer(0),Integer(1),Integer(1)],[Integer(0),
↪Integer(1),Integer(1),Integer(0)]])                                        #␣
↪needs sage.modules
>>> from sage.combinat.diagram import Diagrams
>>> Diagrams()(M).pp()                                                      #␣
↪needs sage.modules
O . O O
. O O .
>>> Diagrams().from_zero_one_matrix(M).pp()                                 #␣
↪needs sage.modules
O . O O
. O O .

>>> M = matrix([[Integer(1), Integer(0), Integer(0)], [Integer(1), Integer(0),
↪ Integer(0)], [Integer(0), Integer(0), Integer(0)]])                        ␣
↪ # needs sage.modules
>>> Diagrams()(M).pp()                                                      #␣
↪needs sage.modules
O . .
O . .
. . .
```

class sage.combinat.diagram.**NorthwestDiagram**(*parent*, *cells*, *n_rows=None*, *n_cols=None*, *check=True*)

Bases: `Diagram`

Diagrams with the northwest property.

A diagram is a set of cells indexed by natural numbers. Such a diagram has the *northwest property* if the presence of cells $(i1, j1)$ and $(i2, j2)$ implies the presence of the cell $(\min(i1, i2), \min(j1, j2))$. Diagrams with the northwest property are called *northwest diagrams*.

For general diagrams see `Diagram`.

EXAMPLES:

```
sage: from sage.combinat.diagram import NorthwestDiagram
sage: N = NorthwestDiagram([(0,0), (0, 2), (2,0)])
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import NorthwestDiagram
>>> N = NorthwestDiagram([(Integer(0),Integer(0)), (Integer(0), Integer(2)),␣
↪(Integer(2),Integer(0))])
```

To visualize them, use the `.pp()` method:

```
sage: N.pp()
O . O
. . .
O . .
```

```
>>> from sage.all import *
>>> N.pp()
O . O
. . .
O . .
```

check()

A diagram has the northwest property if the presence of cells $(i1, j1)$ and $(i2, j2)$ implies the presence of the cell $(min(i1, i2), min(j1, j2))$. This method checks if the northwest property is satisfied for `self`

EXAMPLES:

```
sage: from sage.combinat.diagram import NorthwestDiagram
sage: N = NorthwestDiagram([(0,0), (0,3), (3,0)])
sage: N.check()
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import NorthwestDiagram
>>> N = NorthwestDiagram([(Integer(0),Integer(0)), (Integer(0),Integer(3)),␣
↪(Integer(3),Integer(0))])
>>> N.check()
```

Here is a non-example:

```
sage: notN = NorthwestDiagram([(0,1), (1,0)])  #.check() is implicit
Traceback (most recent call last):
...
ValueError: diagram is not northwest
```

```
>>> from sage.all import *
>>> notN = NorthwestDiagram([(Integer(0),Integer(1)), (Integer(1),
↪Integer(0))])  #.check() is implicit
Traceback (most recent call last):
...
ValueError: diagram is not northwest
```

peelable_tableaux()

Return the set of peelable tableaux whose diagram is `self`.

For a fixed northwest diagram D, we say that a Young tableau T is D-peelable if:

1. the row indices of the cells in the first column of D are the entries in an initial segment in the first column of T and

2. the tableau Q obtained by removing those cells from T and playing jeu de taquin is $(D - C)$-peelable, where $D - C$ is the diagram formed by forgetting the first column of D.

Reiner and Shimozono [RS1995] showed that the number $\mathrm{red}(w)$ of reduced words of a permutation w may be computed using the peelable tableaux of the Rothe diagram $D(w)$. Explicitly,

$$\mathrm{red}(w) = \sum_T f_{\mathrm{shape}\, T},$$

where the sum runs over the $D(w)$-peelable tableaux T and f_λ is the number of standard Young tableaux of shape λ (which may be computed using the hook-length formula).

EXAMPLES:

We can compute the D-peelable diagrams for a northwest diagram D:

```
sage: from sage.combinat.diagram import NorthwestDiagram
sage: cells = [(0,0), (0,1), (0,2), (1,0), (2,0), (2,2), (2,4),
....:          (4,0), (4,2)]
sage: D = NorthwestDiagram(cells); D.pp()
O O O . .
O . . . .
O . O . O
. . . . .
O . O . .
sage: D.peelable_tableaux()
{[[1, 1, 1], [2, 3, 3], [3, 5], [5]],
[[1, 1, 1, 3], [2, 3], [3, 5], [5]]}
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import NorthwestDiagram
>>> cells = [(Integer(0),Integer(0)), (Integer(0),Integer(1)), (Integer(0),
↪Integer(2)), (Integer(1),Integer(0)), (Integer(2),Integer(0)), (Integer(2),
↪Integer(2)), (Integer(2),Integer(4)),
...          (Integer(4),Integer(0)), (Integer(4),Integer(2))]
>>> D = NorthwestDiagram(cells); D.pp()
O O O . .
O . . . .
O . O . O
. . . . .
O . O . .
>>> D.peelable_tableaux()
{[[1, 1, 1], [2, 3, 3], [3, 5], [5]],
[[1, 1, 1, 3], [2, 3], [3, 5], [5]]}
```

EXAMPLES:

If the diagram is only one column, there is only one peelable tableau:

```
sage: from sage.combinat.diagram import NorthwestDiagram
sage: NWD = NorthwestDiagram([(0,0), (2,0)])
sage: NWD.peelable_tableaux()
{[[1], [3]]}
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import NorthwestDiagram
```

(continues on next page)

(continued from previous page)

```
>>> NWD = NorthwestDiagram([(Integer(0),Integer(0)), (Integer(2),Integer(0))])
>>> NWD.peelable_tableaux()
{[[1], [3]]}
```

From [RS1995], we know that there is only one peelable tableau for the Rothe diagram of the permutation (in one line notation) 251643:

```
sage: D = NorthwestDiagram([(1, 2), (1, 3), (3, 2), (3, 3), (4, 2)])
sage: D.pp()
. . . .
. . O O
. . . .
. . O O
. . O .

sage: D.peelable_tableaux()
{[[2, 2], [4, 4], [5]]}
```

```
>>> from sage.all import *
>>> D = NorthwestDiagram([(Integer(1), Integer(2)), (Integer(1), Integer(3)),␣
↪(Integer(3), Integer(2)), (Integer(3), Integer(3)), (Integer(4),␣
↪Integer(2))])
>>> D.pp()
. . . .
. . O O
. . . .
. . O O
. . O .

>>> D.peelable_tableaux()
{[[2, 2], [4, 4], [5]]}
```

Here are all the intermediate steps to compute the peelables for the Rothe diagram of (in one-line notation) 64817235. They are listed from deepest in the recursion to the final step. The recursion has depth five in this case so we will label the intermediate tableaux by D_i where i is the step in the recursion at which they appear.

Start with the one that has a single column:

```
sage: D5 = NorthwestDiagram([(2,0)]); D5.pp()
.
.
O
sage: D5.peelable_tableaux()
{[[3]]}
```

```
>>> from sage.all import *
>>> D5 = NorthwestDiagram([(Integer(2),Integer(0))]); D5.pp()
.
.
O
>>> D5.peelable_tableaux()
{[[3]]}
```

Now we know all of the D_5 peelables, so we can compute the D_4 peelables:

```
sage: D4 = NorthwestDiagram([(0, 0), (2,0), (4, 0), (2, 2)])
sage: D4.pp()
O . .
. . .
O . O
. . .
O . .

sage: D4.peelable_tableaux()
{[[1, 3], [3], [5]]}
```

```
>>> from sage.all import *
>>> D4 = NorthwestDiagram([(Integer(0), Integer(0)), (Integer(2),Integer(0)),␣
↪(Integer(4), Integer(0)), (Integer(2), Integer(2))])
>>> D4.pp()
O . .
. . .
O . O
. . .
O . .

>>> D4.peelable_tableaux()
{[[1, 3], [3], [5]]}
```

There is only one D_4 peelable, so we can compute the D_3 peelables:

```
sage: D3 = NorthwestDiagram([(0,0), (0,1), (2, 1), (2, 3), (4,1)])
sage: D3.pp()
O O . .
. . . .
. O . O
. . . .
. O . .

sage: D3.peelable_tableaux()
{[[1, 1], [3, 3], [5]], [[1, 1, 3], [3], [5]]}
```

```
>>> from sage.all import *
>>> D3 = NorthwestDiagram([(Integer(0),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(2), Integer(1)), (Integer(2), Integer(3)), (Integer(4),
↪Integer(1))])
>>> D3.pp()
O O . .
. . . .
. O . O
. . . .
. O . .

>>> D3.peelable_tableaux()
{[[1, 1], [3, 3], [5]], [[1, 1, 3], [3], [5]]}
```

Now compute the D_2 peelables:

```
sage: cells = [(0,0), (0,1), (0,2), (1,0), (2,0), (2,2), (2,4),
....:           (4,0), (4,2)]
sage: D2 = NorthwestDiagram(cells); D2.pp()
O O O . .
O . . . .
O . O . O
. . . . .
O . O . .

sage: D2.peelable_tableaux()
{[[1, 1, 1], [2, 3, 3], [3, 5], [5]],
[[1, 1, 1, 3], [2, 3], [3, 5], [5]]}
```

```
>>> from sage.all import *
>>> cells = [(Integer(0),Integer(0)), (Integer(0),Integer(1)), (Integer(0),
↪Integer(2)), (Integer(1),Integer(0)), (Integer(2),Integer(0)), (Integer(2),
↪Integer(2)), (Integer(2),Integer(4)),
...           (Integer(4),Integer(0)), (Integer(4),Integer(2))]
>>> D2 = NorthwestDiagram(cells); D2.pp()
O O O . .
O . . . .
O . O . O
. . . . .
O . O . .

>>> D2.peelable_tableaux()
{[[1, 1, 1], [2, 3, 3], [3, 5], [5]],
[[1, 1, 1, 3], [2, 3], [3, 5], [5]]}
```

And the D_1 peelables:

```
sage: cells = [(0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (2,0),
....:           (2,1), (2,3), (2,5), (4,0), (4,1), (4,3)]
sage: D1 = NorthwestDiagram(cells); D1.pp()
O O O O . .
O O . . . .
O O . O . O
. . . . . .
O O . O . .

sage: D1.peelable_tableaux()
{[[1, 1, 1, 1], [2, 2, 3, 3], [3, 3, 5], [5, 5]],
[[1, 1, 1, 1, 3], [2, 2, 3], [3, 3, 5], [5, 5]]}
```

```
>>> from sage.all import *
>>> cells = [(Integer(0),Integer(0)), (Integer(0),Integer(1)), (Integer(0),
↪Integer(2)), (Integer(0),Integer(3)), (Integer(1),Integer(0)), (Integer(1),
↪Integer(1)), (Integer(2),Integer(0)),
...           (Integer(2),Integer(1)), (Integer(2),Integer(3)), (Integer(2),
↪Integer(5)), (Integer(4),Integer(0)), (Integer(4),Integer(1)), (Integer(4),
↪Integer(3))]
>>> D1 = NorthwestDiagram(cells); D1.pp()
```

(continues on next page)

(continued from previous page)

```
O O O O . .
O O . . . .
O O . O . O
. . . . . .
O O . O . .

>>> D1.peelable_tableaux()
{[[1, 1, 1, 1], [2, 2, 3, 3], [3, 3, 5], [5, 5]],
[[1, 1, 1, 1, 3], [2, 2, 3], [3, 3, 5], [5, 5]]}
```

Which we can use to get the D peelables:

```
sage: cells = [(0,0), (0,1), (0,2), (0,3), (0,4),
....:           (1,0), (1,1), (1,2),
....:           (2,0), (2,1), (2,2), (2,4), (2,6),
....:                  (4,1), (4,2), (4,4)]
sage: D = NorthwestDiagram(cells); D.pp()
O O O O O . .
O O O . . . .
O O O . O . O
. . . . . . .
. O O . O . .
sage: D.peelable_tableaux()
{[[1, 1, 1, 1, 1], [2, 2, 2, 3, 3], [3, 3, 3], [5, 5, 5]],
 [[1, 1, 1, 1, 1], [2, 2, 2, 3, 3], [3, 3, 3, 5], [5, 5]],
 [[1, 1, 1, 1, 1, 3], [2, 2, 2, 3], [3, 3, 3], [5, 5, 5]],
 [[1, 1, 1, 1, 1, 3], [2, 2, 2, 3], [3, 3, 3, 5], [5, 5]]}
```

```
>>> from sage.all import *
>>> cells = [(Integer(0),Integer(0)), (Integer(0),Integer(1)), (Integer(0),
↪Integer(2)), (Integer(0),Integer(3)), (Integer(0),Integer(4)),
...           (Integer(1),Integer(0)), (Integer(1),Integer(1)), (Integer(1),
↪Integer(2)),
...           (Integer(2),Integer(0)), (Integer(2),Integer(1)), (Integer(2),
↪Integer(2)), (Integer(2),Integer(4)), (Integer(2),Integer(6)),
...                  (Integer(4),Integer(1)), (Integer(4),Integer(2)),␣
↪(Integer(4),Integer(4))]
>>> D = NorthwestDiagram(cells); D.pp()
O O O O O . .
O O O . . . .
O O O . O . O
. . . . . . .
. O O . O . .
>>> D.peelable_tableaux()
{[[1, 1, 1, 1, 1], [2, 2, 2, 3, 3], [3, 3, 3], [5, 5, 5]],
 [[1, 1, 1, 1, 1], [2, 2, 2, 3, 3], [3, 3, 3, 5], [5, 5]],
 [[1, 1, 1, 1, 1, 3], [2, 2, 2, 3], [3, 3, 3], [5, 5, 5]],
 [[1, 1, 1, 1, 1, 3], [2, 2, 2, 3], [3, 3, 3, 5], [5, 5]]}
```

ALGORITHM:

This implementation uses the algorithm suggested in Remark 25 of [RS1995].

class sage.combinat.diagram.**NorthwestDiagrams**(*category=None*)

Bases: *Diagrams*

Diagrams satisfying the northwest property.

A diagram D is a *northwest diagram* if for every two cells (i_1, j_1) and (i_2, j_2) in D then there exists the cell $(\min(i_1, i_2), \min(j_1, j_2)) \in D$.

EXAMPLES:

```
sage: from sage.combinat.diagram import NorthwestDiagram
sage: N = NorthwestDiagram([(0,0), (0, 10), (5,0)]); N.pp()
O . . . . . . . . . O
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . . . .
O . . . . . . . . . .
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import NorthwestDiagram
>>> N = NorthwestDiagram([(Integer(0),Integer(0)), (Integer(0), Integer(10)),␣
↪(Integer(5),Integer(0))]); N.pp()
O . . . . . . . . . O
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . . . .
O . . . . . . . . . .
```

Note that checking whether or not the northwest property is satisfied is automatically checked. The diagram found by adding the cell $(1, 1)$ to the diagram above is *not* a northwest diagram. The cell $(1, 0)$ should be present due to the presence of $(5, 0)$ and $(1, 1)$:

```
sage: from sage.combinat.diagram import Diagram
sage: Diagram([(0, 0), (0, 10), (5, 0), (1, 1)]).pp()
O . . . . . . . . . O
. O . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . . . .
O . . . . . . . . . .
sage: NorthwestDiagram([(0, 0), (0, 10), (5, 0), (1, 1)])
Traceback (most recent call last):
...
ValueError: diagram is not northwest
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> Diagram([(Integer(0), Integer(0)), (Integer(0), Integer(10)), (Integer(5),␣
↪Integer(0)), (Integer(1), Integer(1))]).pp()
O . . . . . . . . . O
. O . . . . . . . . .
. . . . . . . . . . .
```

(continues on next page)

(continued from previous page)

```
. . . . . . . . . . . .
. . . . . . . . . . . .
O . . . . . . . . . . .
>>> NorthwestDiagram([(Integer(0), Integer(0)), (Integer(0), Integer(10)),␣
↪(Integer(5), Integer(0)), (Integer(1), Integer(1))])
Traceback (most recent call last):
...
ValueError: diagram is not northwest
```

However, this behavior can be turned off if you are confident that you are providing a northwest diagram:

```
sage: N = NorthwestDiagram([(0, 0), (0, 10), (5, 0),
....:                       (1, 1), (0, 1), (1, 0)],
....:                       check=False)
sage: N.pp()
O O . . . . . . . . . O
O O . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
O . . . . . . . . . . .
```

```
>>> from sage.all import *
>>> N = NorthwestDiagram([(Integer(0), Integer(0)), (Integer(0), Integer(10)),␣
↪(Integer(5), Integer(0)),
...                       (Integer(1), Integer(1)), (Integer(0), Integer(1)),␣
↪(Integer(1), Integer(0))],
...                       check=False)
>>> N.pp()
O O . . . . . . . . . O
O O . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
O . . . . . . . . . . .
```

Note that arbitrary diagrams which happen to be northwest diagrams only live in the parent of `Diagrams`:

```
sage: D = Diagram([(0, 0), (0, 10), (5, 0), (1, 1), (0, 1), (1, 0)])
sage: D.pp()
O O . . . . . . . . . O
O O . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
O . . . . . . . . . . .
sage: from sage.combinat.diagram import NorthwestDiagrams
sage: D in NorthwestDiagrams()
False
```

```
>>> from sage.all import *
>>> D = Diagram([(Integer(0), Integer(0)), (Integer(0), Integer(10)), (Integer(5),
```

(continues on next page)

(continued from previous page)

```
↪ Integer(0)), (Integer(1), Integer(1)), (Integer(0), Integer(1)), (Integer(1),␣
↪Integer(0))])
>>> D.pp()
O O . . . . . . . . . O
O O . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
O . . . . . . . . . . .
>>> from sage.combinat.diagram import NorthwestDiagrams
>>> D in NorthwestDiagrams()
False
```

Here are some more examples:

```
sage: from sage.combinat.diagram import NorthwestDiagram, NorthwestDiagrams
sage: D = NorthwestDiagram([(0,1), (0,2), (1,1)]); D.pp()
. O O
. O .
sage: NWDgms = NorthwestDiagrams()
sage: D = NWDgms([(1,1), (1,2), (2,1)]); D.pp()
. . .
. O O
. O .
sage: D.parent()
Combinatorial northwest diagrams
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import NorthwestDiagram, NorthwestDiagrams
>>> D = NorthwestDiagram([(Integer(0),Integer(1)), (Integer(0),Integer(2)),␣
↪(Integer(1),Integer(1))]); D.pp()
. O O
. O .
>>> NWDgms = NorthwestDiagrams()
>>> D = NWDgms([(Integer(1),Integer(1)), (Integer(1),Integer(2)), (Integer(2),
↪Integer(1))]); D.pp()
. . .
. O O
. O .
>>> D.parent()
Combinatorial northwest diagrams
```

Additionally, there are natural constructions of a northwest diagram given the data of a permutation (Rothe diagrams are the prototypical example of northwest diagrams), or the data of a partition of an integer, or a skew partition.

The Rothe diagram $D(\omega)$ of a permutation ω is specified by the cells

$$D(\omega) = \{(\omega_j, i) : i < j,\, \omega_i > \omega_j\}.$$

We can construct one by calling *rothe_diagram()* method on the set of all *NorthwestDiagrams*:

```
sage: w = Permutations(4)([4,3,2,1])
sage: NorthwestDiagrams().rothe_diagram(w).pp()
```

(continues on next page)

(continued from previous page)

```
O O O .
O O . .
O . . .
. . . .
```

```
>>> from sage.all import *
>>> w = Permutations(Integer(4))([Integer(4),Integer(3),Integer(2),Integer(1)])
>>> NorthwestDiagrams().rothe_diagram(w).pp()
O O O .
O O . .
O . . .
. . . .
```

To turn a Ferrers diagram into a northwest diagram, we may call *from_partition()*. This will return a Ferrer's diagram in the set of all northwest diagrams. For many use-cases it is probably better to get Ferrer's diagrams by the corresponding method on partitions, namely `sage.combinat.partitions.Partitions.ferrers_diagram()`:

```
sage: mu = Partition([7,3,1,1])
sage: mu.pp()
*******
***
*
*
sage: NorthwestDiagrams().from_partition(mu).pp()
O O O O O O O
O O O . . . .
O . . . . . .
O . . . . . .
```

```
>>> from sage.all import *
>>> mu = Partition([Integer(7),Integer(3),Integer(1),Integer(1)])
>>> mu.pp()
*******
***
*
*
>>> NorthwestDiagrams().from_partition(mu).pp()
O O O O O O O
O O O . . . .
O . . . . . .
O . . . . . .
```

It is also possible to turn a Ferrers diagram of a skew partition into a northwest diagram, altough it is more subtle than just using the skew diagram itself. One must first reflect the partition about a vertical axis so that the skew partition looks "backwards":

```
sage: mu, nu = Partition([5,4,3,2,1]), Partition([3,2,1])
sage: s = mu/nu; s.pp()
   **
  **
 **
```

(continues on next page)

(continued from previous page)

```
**
*
sage: NorthwestDiagrams().from_skew_partition(s).pp()
O O . . .
. O O . .
. . O O .
. . . O O
. . . . O
```

```
>>> from sage.all import *
>>> mu, nu = Partition([Integer(5),Integer(4),Integer(3),Integer(2),Integer(1)]),␣
↪Partition([Integer(3),Integer(2),Integer(1)])
>>> s = mu/nu; s.pp()
   **
  **
 **
**
*
>>> NorthwestDiagrams().from_skew_partition(s).pp()
O O . . .
. O O . .
. . O O .
. . . O O
. . . . O
```

Element

alias of *NorthwestDiagram*

from_parallelogram_polyomino(*p*)

Create the diagram corresponding to a *ParallelogramPolyomino*.

EXAMPLES:

```
sage: p = ParallelogramPolyomino([[0, 0, 1, 0, 0, 0, 1, 1],                    #␣
↪needs sage.modules
....:                             [1, 1, 0, 1, 0, 0, 0, 0]])
sage: from sage.combinat.diagram import NorthwestDiagrams
sage: NorthwestDiagrams().from_parallelogram_polyomino(p).pp()                 #␣
↪needs sage.modules
O O .
O O O
. O O
. O O
. O O
```

```
>>> from sage.all import *
>>> p = ParallelogramPolyomino([[Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(0), Integer(1), Integer(1)],                  ␣
↪ # needs sage.modules
...                             [Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]])
>>> from sage.combinat.diagram import NorthwestDiagrams
```

(continues on next page)

(continued from previous page)

```
>>> NorthwestDiagrams().from_parallelogram_polyomino(p).pp()              #␣
↪needs sage.modules
O O .
O O O
. O O
. O O
. O O
```

from_partition(*mu*)

Return the Ferrer's diagram of `mu` as a northwest diagram.

EXAMPLES:

```
sage: mu = Partition([5,2,1]); mu.pp()
*****
**
*
sage: mu.parent()
Partitions
sage: from sage.combinat.diagram import NorthwestDiagrams
sage: D = NorthwestDiagrams().from_partition(mu)
sage: D.pp()
O O O O O
O O . . .
O . . . .
sage: D.parent()
Combinatorial northwest diagrams
```

```
>>> from sage.all import *
>>> mu = Partition([Integer(5),Integer(2),Integer(1)]); mu.pp()
*****
**
*
>>> mu.parent()
Partitions
>>> from sage.combinat.diagram import NorthwestDiagrams
>>> D = NorthwestDiagrams().from_partition(mu)
>>> D.pp()
O O O O O
O O . . .
O . . . .
>>> D.parent()
Combinatorial northwest diagrams
```

This will print in English notation even if the notation is set to French for the partition:

```
sage: Partitions.options.convention="french"
sage: mu.pp()
*
**
*****
sage: D.pp()
```

(continues on next page)

(continued from previous page)

```
O O O O O
O O . . .
O . . . .
```

```
>>> from sage.all import *
>>> Partitions.options.convention="french"
>>> mu.pp()
*
**
*****
>>> D.pp()
O O O O O
O O . . .
O . . . .
```

from_permutation(*w*)

Return the Rothe diagram of `w`.

We construct a northwest diagram from a permutation by constructing its Rothe diagram. Formally, if ω is a *Permutation* then the Rothe diagram $D(\omega)$ is the diagram whose cells are

$$D(\omega) = \{(\omega_j, i) : i < j,\, \omega_i > \omega_j\}.$$

Informally, one can construct the Rothe diagram by starting with all n^2 possible cells, and then deleting the cells $(i, \omega(i))$ as well as all cells to the right and below. (These are sometimes called "death rays".)

See also

RotheDiagram()

EXAMPLES:

```
sage: from sage.combinat.diagram import NorthwestDiagrams
sage: w = Permutations(3)([2,1,3])
sage: NorthwestDiagrams().rothe_diagram(w).pp()
O . .
. . .
. . .
sage: NorthwestDiagrams().from_permutation(w).pp()
O . .
. . .
. . .

sage: w = Permutations(8)([2,5,4,1,3,6,7,8])
sage: NorthwestDiagrams().rothe_diagram(w).pp()
O . . . . . . .
O . O O . . . .
O . O . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

(continues on next page)

(continued from previous page)

```
. . . . . . . .
. . . . . . . .
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import NorthwestDiagrams
>>> w = Permutations(Integer(3))([Integer(2),Integer(1),Integer(3)])
>>> NorthwestDiagrams().rothe_diagram(w).pp()
O . .
. . .
. . .
>>> NorthwestDiagrams().from_permutation(w).pp()
O . .
. . .
. . .

>>> w = Permutations(Integer(8))([Integer(2),Integer(5),Integer(4),Integer(1),
→Integer(3),Integer(6),Integer(7),Integer(8)])
>>> NorthwestDiagrams().rothe_diagram(w).pp()
O . . . . . . .
O . O O . . . .
O . O . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

from_skew_partition(*s*)

Get the northwest diagram found by reflecting a skew shape across a vertical plane.

EXAMPLES:

```
sage: mu, nu = Partition([3,2,1]), Partition([2,1])
sage: s = mu/nu; s.pp()
  *
 *
*
sage: from sage.combinat.diagram import NorthwestDiagrams
sage: D = NorthwestDiagrams().from_skew_partition(s)
sage: D.pp()
O . .
. O .
. . O

sage: mu, nu = Partition([3,3,2]), Partition([2,2,2])
sage: s = mu/nu; s.pp()
  *
  *
sage: NorthwestDiagrams().from_skew_partition(s).pp()
O . .
O . .
. . .
```

```
>>> from sage.all import *
>>> mu, nu = Partition([Integer(3),Integer(2),Integer(1)]),␣
↪Partition([Integer(2),Integer(1)])
>>> s = mu/nu; s.pp()
  *
 *
*
>>> from sage.combinat.diagram import NorthwestDiagrams
>>> D = NorthwestDiagrams().from_skew_partition(s)
>>> D.pp()
O . .
. O .
. . O

>>> mu, nu = Partition([Integer(3),Integer(3),Integer(2)]),␣
↪Partition([Integer(2),Integer(2),Integer(2)])
>>> s = mu/nu; s.pp()
  *
  *
>>> NorthwestDiagrams().from_skew_partition(s).pp()
O . .
O . .
. . .
```

rothe_diagram(*w*)

Return the Rothe diagram of `w`.

We construct a northwest diagram from a permutation by constructing its Rothe diagram. Formally, if ω is a `Permutation` then the Rothe diagram $D(\omega)$ is the diagram whose cells are

$$D(\omega) = \{(\omega_j, i) : i < j,\, \omega_i > \omega_j\}.$$

Informally, one can construct the Rothe diagram by starting with all n^2 possible cells, and then deleting the cells $(i, \omega(i))$ as well as all cells to the right and below. (These are sometimes called "death rays".)

See also

`RotheDiagram()`

EXAMPLES:

```
sage: from sage.combinat.diagram import NorthwestDiagrams
sage: w = Permutations(3)([2,1,3])
sage: NorthwestDiagrams().rothe_diagram(w).pp()
O . .
. . .
. . .
sage: NorthwestDiagrams().from_permutation(w).pp()
O . .
. . .
. . .

sage: w = Permutations(8)([2,5,4,1,3,6,7,8])
```

(continues on next page)

(continued from previous page)

```
sage: NorthwestDiagrams().rothe_diagram(w).pp()
O . . . . . . .
O . O O . . . .
O . O . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import NorthwestDiagrams
>>> w = Permutations(Integer(3))([Integer(2),Integer(1),Integer(3)])
>>> NorthwestDiagrams().rothe_diagram(w).pp()
O . .
. . .
. . .
>>> NorthwestDiagrams().from_permutation(w).pp()
O . .
. . .
. . .

>>> w = Permutations(Integer(8))([Integer(2),Integer(5),Integer(4),Integer(1),
↪Integer(3),Integer(6),Integer(7),Integer(8)])
>>> NorthwestDiagrams().rothe_diagram(w).pp()
O . . . . . . .
O . O O . . . .
O . O . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

sage.combinat.diagram.**RotheDiagram**(*w*)

The Rothe diagram of a permutation `w`.

EXAMPLES:

```
sage: w = Permutations(9)([1, 7, 4, 5, 9, 3, 2, 8, 6])
sage: from sage.combinat.diagram import RotheDiagram
sage: D = RotheDiagram(w); D.pp()
. . . . . . . . .
. O O O O O . . .
. O O . . . . . .
. O O . . . . . .
. O O . . O . O .
. O . . . . . . .
. . . . . . . . .
. . . . . O . . .
. . . . . . . . .
```

```
>>> from sage.all import *
>>> w = Permutations(Integer(9))([Integer(1), Integer(7), Integer(4), Integer(5),␣
↪Integer(9), Integer(3), Integer(2), Integer(8), Integer(6)])
>>> from sage.combinat.diagram import RotheDiagram
>>> D = RotheDiagram(w); D.pp()
. . . . . . . . .
. O O O O O . . .
. O O . . . . . .
. O O . . . . . .
. O O . . O . O .
. O . . . . . . .
. . . . . . . . .
. . . . . O . . .
. . . . . . . . .
```

The Rothe diagram is a northwest diagram:

```
sage: D.parent()
Combinatorial northwest diagrams
```

```
>>> from sage.all import *
>>> D.parent()
Combinatorial northwest diagrams
```

Some other examples:

```
sage: RotheDiagram([2, 1, 4, 3]).pp()
O . . .
. . . .
. . O .
. . . .

sage: RotheDiagram([4, 1, 3, 2]).pp()
O O O .
. . . .
. O . .
. . . .
```

```
>>> from sage.all import *
>>> RotheDiagram([Integer(2), Integer(1), Integer(4), Integer(3)]).pp()
O . . .
. . . .
. . O .
. . . .

>>> RotheDiagram([Integer(4), Integer(1), Integer(3), Integer(2)]).pp()
O O O .
. . . .
. O . .
. . . .
```

Currently, only elements of the set of `sage.combinat.permutations.Permutations` are supported. In particular, elements of permutation groups are not supported:

```
sage: w = SymmetricGroup(9).an_element()                                    #␣
↪needs sage.groups
sage: RotheDiagram(w)                                                       #␣
↪needs sage.groups
Traceback (most recent call last):
...
ValueError: w must be a permutation
```

```
>>> from sage.all import *
>>> w = SymmetricGroup(Integer(9)).an_element()                             ␣
↪     # needs sage.groups
>>> RotheDiagram(w)                                                       #␣
↪needs sage.groups
Traceback (most recent call last):
...
ValueError: w must be a permutation
```

5.1.98 Diagram and Partition Algebras

AUTHORS:

- Mike Hansen (2007): Initial version
- Stephen Doty, Aaron Lauve, George H. Seelinger (2012): Implementation of partition, Brauer, Temperley–Lieb, and ideal partition algebras
- Stephen Doty, Aaron Lauve, George H. Seelinger (2015): Implementation of `*Diagram` classes and other methods to improve diagram algebras.
- Mike Zabrocki (2018): Implementation of individual element diagram classes
- Aaron Lauve, Mike Zabrocki (2018): Implementation of orbit basis for Partition algebra.
- Travis Scrimshaw (2024): Implemented the Potts representation of the partition algebra.

class sage.combinat.diagram_algebras.**AbstractPartitionDiagram**(*parent*, *d*, *check=True*)

Bases: *AbstractSetPartition*

Abstract base class for partition diagrams.

This class represents a single partition diagram, that is used as a basis key for a diagram algebra element. A partition diagram should be a partition of the set $\{1, \ldots, k, -1, \ldots, -k\}$. Each such set partition is regarded as a graph on nodes $\{1, \ldots, k, -1, \ldots, -k\}$ arranged in two rows, with nodes $1, \ldots, k$ in the top row from left to right and with nodes $-1, \ldots, -k$ in the bottom row from left to right, and an edge connecting two nodes if and only if the nodes lie in the same subset of the set partition.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: pd1 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]])
sage: pd2 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]])
sage: pd1
{{-2, -1}, {1, 2}}
sage: pd1 == pd2
True
sage: pd1 == [[1,2],[-1,-2]]
```

(continues on next page)

(continued from previous page)

```
True
sage: pd1 == ((-2,-1),(2,1))
True
sage: pd1 == SetPartition([[1,2],[-1,-2]])
True
sage: pd3 = da.AbstractPartitionDiagram(pd, [[1,-2],[-1,2]])
sage: pd1 == pd3
False
sage: pd4 = da.AbstractPartitionDiagram(pd, [[1,2],[3,4]])
Traceback (most recent call last):
...
ValueError: {{1, 2}, {3, 4}} does not represent two rows of vertices of order 2
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> pd1 = da.AbstractPartitionDiagram(pd, [[Integer(1),Integer(2)],[-Integer(1),-
↪Integer(2)]])
>>> pd2 = da.AbstractPartitionDiagram(pd, [[Integer(1),Integer(2)],[-Integer(1),-
↪Integer(2)]])
>>> pd1
{{-2, -1}, {1, 2}}
>>> pd1 == pd2
True
>>> pd1 == [[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]
True
>>> pd1 == ((-Integer(2),-Integer(1)),(Integer(2),Integer(1)))
True
>>> pd1 == SetPartition([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
True
>>> pd3 = da.AbstractPartitionDiagram(pd, [[Integer(1),-Integer(2)],[-Integer(1),
↪Integer(2)]])
>>> pd1 == pd3
False
>>> pd4 = da.AbstractPartitionDiagram(pd, [[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]])
Traceback (most recent call last):
...
ValueError: {{1, 2}, {3, 4}} does not represent two rows of vertices of order 2
```

base_diagram()

Return the underlying implementation of the diagram.

OUTPUT: tuple of tuples of integers

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: pd([[1,2],[-1,-2]]).base_diagram() == ((-2,-1),(1,2))
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]).base_diagram() ==␣
↪((-Integer(2),-Integer(1)),(Integer(1),Integer(2)))
True
```

check()

Check the validity of the input for the diagram.

compose(*other*, *check=True*)

Compose `self` with `other`.

The composition of two diagrams X and Y is given by placing X on top of Y and removing all loops.

OUTPUT:

A tuple where the first entry is the composite diagram and the second entry is how many loop were removed.

> **Note**
>
> This is not really meant to be called directly, but it works to call it this way if desired.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: pd([[1,2],[-1,-2]]).compose(pd([[1,2],[-1,-2]]))
({{-2, -1}, {1, 2}}, 1)
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]).
↪compose(pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]))
({{-2, -1}, {1, 2}}, 1)
```

count_blocks_of_size(*n*)

Count the number of blocks of a given size.

INPUT:

- `n` – positive integer

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PartitionDiagram
sage: pd = PartitionDiagram([[1,-3,-5],[2,4],[3,-1,-2],[5],[-4]])
sage: pd.count_blocks_of_size(1)
2
sage: pd.count_blocks_of_size(2)
1
sage: pd.count_blocks_of_size(3)
2
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagram
>>> pd = PartitionDiagram([[Integer(1),-Integer(3),-Integer(5)],[Integer(2),
↪Integer(4)],[Integer(3),-Integer(1),-Integer(2)],[Integer(5)],[-
↪Integer(4)]])
>>> pd.count_blocks_of_size(Integer(1))
2
>>> pd.count_blocks_of_size(Integer(2))
1
>>> pd.count_blocks_of_size(Integer(3))
2
```

diagram()

Return the underlying implementation of the diagram.

OUTPUT: tuple of tuples of integers

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: pd([[1,2],[-1,-2]]).base_diagram() == ((-2,-1),(1,2))
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]).base_diagram() ==␣
↪((-Integer(2),-Integer(1)),(Integer(1),Integer(2)))
True
```

dual()

Return the dual diagram of `self` by flipping it top-to-bottom.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PartitionDiagram
sage: D = PartitionDiagram([[1,-1],[2,-2,-3],[3]])
sage: D.dual()
{{-3}, {-2, 2, 3}, {-1, 1}}
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagram
>>> D = PartitionDiagram([[Integer(1),-Integer(1)],[Integer(2),-Integer(2),-
↪Integer(3)],[Integer(3)]])
>>> D.dual()
{{-3}, {-2, 2, 3}, {-1, 1}}
```

is_planar()

Test if the diagram `self` is planar.

A diagram element is planar if the graph of the nodes is planar.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import BrauerDiagram
sage: BrauerDiagram([[1,-2],[2,-1]]).is_planar()
False
sage: BrauerDiagram([[1,-1],[2,-2]]).is_planar()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import BrauerDiagram
>>> BrauerDiagram([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]]).is_
→planar()
False
>>> BrauerDiagram([[Integer(1),-Integer(1)],[Integer(2),-Integer(2)]]).is_
→planar()
True
```

order()

Return the maximum entry in the diagram element.

A diagram element will be a partition of the set $\{-1, -2, \ldots, -k, 1, 2, \ldots, k\}$. The order of the diagram element is the value k.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PartitionDiagram
sage: PartitionDiagram([[1,-1],[2,-2,-3],[3]]).order()
3
sage: PartitionDiagram([[1,-1]]).order()
1
sage: PartitionDiagram([[1,-3,-5],[2,4],[3,-1,-2],[5],[-4]]).order()
5
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagram
>>> PartitionDiagram([[Integer(1),-Integer(1)],[Integer(2),-Integer(2),-
→Integer(3)],[Integer(3)]]).order()
3
>>> PartitionDiagram([[Integer(1),-Integer(1)]]).order()
1
>>> PartitionDiagram([[Integer(1),-Integer(3),-Integer(5)],[Integer(2),
→Integer(4)],[Integer(3),-Integer(1),-Integer(2)],[Integer(5)],[-
→Integer(4)]]).order()
5
```

propagating_number()

Return the propagating number of the diagram.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: d1 = pd([[1,-2],[2,-1]])
sage: d1.propagating_number()
```

(continues on next page)

(continued from previous page)

```
2
sage: d2 = pd([[1,2],[-2,-1]])
sage: d2.propagating_number()
0
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> d1 = pd([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> d1.propagating_number()
2
>>> d2 = pd([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])
>>> d2.propagating_number()
0
```

set_partition()

Return the underlying implementation of the diagram as a set of sets.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: X = pd([[1,2],[-1,-2]]).set_partition(); X
{{-2, -1}, {1, 2}}
sage: X.parent()
Set partitions
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> X = pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]).set_
↪partition(); X
{{-2, -1}, {1, 2}}
>>> X.parent()
Set partitions
```

class sage.combinat.diagram_algebras.**AbstractPartitionDiagrams**(*order*, *category=None*)

Bases: `Parent`, `UniqueRepresentation`

This is an abstract base class for partition diagrams.

The primary use of this class is to serve as basis keys for diagram algebras, but diagrams also have properties in their own right. Furthermore, this class is meant to be extended to create more efficient contains methods.

INPUT:

- `order` – integer or integer $+1/2$; the order of the diagrams
- `category` – (default: `FiniteEnumeratedSets()`) the category

All concrete classes should implement attributes

- `_name` – the name of the class
- `_diagram_func` – an iterator function that takes the order as its only input

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(2)
sage: pd
Partition diagrams of order 2
sage: pd.an_element() in pd
True
sage: elm = pd([[1,2],[-1,-2]])
sage: elm in pd
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.PartitionDiagrams(Integer(2))
>>> pd
Partition diagrams of order 2
>>> pd.an_element() in pd
True
>>> elm = pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
>>> elm in pd
True
```

Element

alias of *AbstractPartitionDiagram*

class sage.combinat.diagram_algebras.**BrauerAlgebra**(*k*, *q*, *base_ring*, *prefix*)

Bases: *SubPartitionAlgebra*, *UnitDiagramMixin*

A Brauer algebra.

The Brauer algebra of rank k is an algebra with basis indexed by the collection of set partitions of $\{1, \ldots, k, -1, \ldots, -k\}$ with block size 2.

This algebra is a subalgebra of the partition algebra. For more information, see *PartitionAlgebra*.

INPUT:

- `k` – rank of the algebra
- `q` – the deformation parameter q
- `base_ring` – (default: `None`) a ring containing `q`; if `None` then just takes the parent of `q`
- `prefix` – (default: `'B'`) a label for the basis elements

EXAMPLES:

We now define the Brauer algebra of rank 2 with parameter `x` over $\mathbf{Z}$:

```
sage: R.<x> = ZZ[]
sage: B = BrauerAlgebra(2, x, R)
sage: B
Brauer Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
sage: B.basis()
Lazy family (Term map from Brauer diagrams of order 2 to Brauer Algebra
 of rank 2 with parameter x over Univariate Polynomial Ring in x
 over Integer Ring(i))_{i in Brauer diagrams of order 2}
sage: B.basis().keys()
```

(continues on next page)

(continued from previous page)

```
Brauer diagrams of order 2
sage: B.basis().keys()([[-2, 1], [2, -1]])
{{-2, 1}, {-1, 2}}
sage: b = B.basis().list(); b
[B{{-2, -1}, {1, 2}}, B{{-2, 1}, {-1, 2}}, B{{-2, 2}, {-1, 1}}]
sage: b[0]
B{{-2, -1}, {1, 2}}
sage: b[0]^2
x*B{{-2, -1}, {1, 2}}
sage: b[0]^5
x^4*B{{-2, -1}, {1, 2}}
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> B = BrauerAlgebra(Integer(2), x, R)
>>> B
Brauer Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
>>> B.basis()
Lazy family (Term map from Brauer diagrams of order 2 to Brauer Algebra
 of rank 2 with parameter x over Univariate Polynomial Ring in x
 over Integer Ring(i))_{i in Brauer diagrams of order 2}
>>> B.basis().keys()
Brauer diagrams of order 2
>>> B.basis().keys()([[-Integer(2), Integer(1)], [Integer(2), -Integer(1)]])
{{-2, 1}, {-1, 2}}
>>> b = B.basis().list(); b
[B{{-2, -1}, {1, 2}}, B{{-2, 1}, {-1, 2}}, B{{-2, 2}, {-1, 1}}]
>>> b[Integer(0)]
B{{-2, -1}, {1, 2}}
>>> b[Integer(0)]**Integer(2)
x*B{{-2, -1}, {1, 2}}
>>> b[Integer(0)]**Integer(5)
x^4*B{{-2, -1}, {1, 2}}
```

Note, also that since the symmetric group algebra is contained in the Brauer algebra, there is also a conversion between the two.

```
sage: R.<x> = ZZ[]
sage: B = BrauerAlgebra(2, x, R)
sage: S = SymmetricGroupAlgebra(R, 2)
sage: S([2,1])*B([[1,-1],[2,-2]])
B{{-2, 1}, {-1, 2}}
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> B = BrauerAlgebra(Integer(2), x, R)
>>> S = SymmetricGroupAlgebra(R, Integer(2))
>>> S([Integer(2),Integer(1)])*B([[Integer(1),-Integer(1)],[Integer(2),-
↪Integer(2)]])
B{{-2, 1}, {-1, 2}}
```

jucys_murphy(*j*)

Return the j-th generalized Jucys-Murphy element of `self`.

The j-th Jucys-Murphy element of a Brauer algebra is simply the j-th Jucys-Murphy element of the symmetric group algebra with an extra $(z-1)/2$ term, where `z` is the parameter of the Brauer algebra.

REFERENCES:

EXAMPLES:

```
sage: # needs sage.symbolic
sage: z = var('z')
sage: B = BrauerAlgebra(3,z)
sage: B.jucys_murphy(1)
(1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
sage: B.jucys_murphy(3)
-B{{-3, -2}, {-1, 1}, {2, 3}} - B{{-3, -1}, {-2, 2}, {1, 3}}
 + B{{-3, 1}, {-2, 2}, {-1, 3}} + B{{-3, 2}, {-2, 3}, {-1, 1}}
 + (1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> z = var('z')
>>> B = BrauerAlgebra(Integer(3),z)
>>> B.jucys_murphy(Integer(1))
(1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
>>> B.jucys_murphy(Integer(3))
-B{{-3, -2}, {-1, 1}, {2, 3}} - B{{-3, -1}, {-2, 2}, {1, 3}}
 + B{{-3, 1}, {-2, 2}, {-1, 3}} + B{{-3, 2}, {-2, 3}, {-1, 1}}
 + (1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
```

options = Current options for Brauer diagram - display: normal

class sage.combinat.diagram_algebras.**BrauerDiagram**(*parent*, *d*, *check=True*)

Bases: *AbstractPartitionDiagram*

A Brauer diagram.

A Brauer diagram for an integer k is a partition of the set $\{1,\ldots,k,-1,\ldots,-k\}$ with block size 2.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(2)
sage: bd1 = bd([[1,2],[-1,-2]])
sage: bd2 = bd([[1,2,-1,-2]])
Traceback (most recent call last):
...
ValueError: all blocks of {{-2, -1, 1, 2}} must be of size 2
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(2))
>>> bd1 = bd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
>>> bd2 = bd([[Integer(1),Integer(2),-Integer(1),-Integer(2)]])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: all blocks of {{-2, -1, 1, 2}} must be of size 2
```

bijection_on_free_nodes(*two_line=False*)

Return the induced bijection - as a list of $(x, f(x))$ values - from the free nodes on the top at the Brauer diagram to the free nodes at the bottom of `self`.

OUTPUT:

If `two_line` is `True`, then the output is the induced bijection as a two-row list `(inputs, outputs)`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-2,-3],[3,-1]])
sage: elm.bijection_on_free_nodes()
[[3, -1]]
sage: elm2 = bd([[1,-2],[2,-3],[3,-1]])
sage: elm2.bijection_on_free_nodes(two_line=True)
[[1, 2, 3], [-2, -3, -1]]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> elm = bd([[Integer(1),Integer(2)],[-Integer(2),-Integer(3)],[Integer(3),-
→Integer(1)]])
>>> elm.bijection_on_free_nodes()
[[3, -1]]
>>> elm2 = bd([[Integer(1),-Integer(2)],[Integer(2),-Integer(3)],[Integer(3),-
→Integer(1)]])
>>> elm2.bijection_on_free_nodes(two_line=True)
[[1, 2, 3], [-2, -3, -1]]
```

check()

Check the validity of the input for `self`.

involution_permutation_triple(*curt=True*)

Return the involution permutation triple of `self`.

From Graham-Lehrer (see `BrauerDiagrams`), a Brauer diagram is a triple (D_1, D_2, π), where:

- D_1 is a partition of the top nodes;
- D_2 is a partition of the bottom nodes;
- π is the induced permutation on the free nodes.

INPUT:

- `curt` – boolean (default: `True`); if `True`, then return bijection on free nodes as a one-line notation (standardized to look like a permutation), else, return the honest mapping, a list of pairs $(i, -j)$ describing the bijection on free nodes

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-2,-3],[3,-1]])
sage: elm.involution_permutation_triple()
([(1, 2)], [(-3, -2)], [1])
sage: elm.involution_permutation_triple(curt=False)
([(1, 2)], [(-3, -2)], [[3, -1]])
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> elm = bd([[Integer(1),Integer(2)],[-Integer(2),-Integer(3)],[Integer(3),-
↪Integer(1)]])
>>> elm.involution_permutation_triple()
([(1, 2)], [(-3, -2)], [1])
>>> elm.involution_permutation_triple(curt=False)
([(1, 2)], [(-3, -2)], [[3, -1]])
```

is_elementary_symmetric()

Check if is elementary symmetric.

Let (D_1, D_2, π) be the Graham-Lehrer representation of the Brauer diagram d. We say d is *elementary symmetric* if $D_1 = D_2$ and π is the identity.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-1,-2],[3,-3]])
sage: elm.is_elementary_symmetric()
True
sage: elm2 = bd([[1,2],[-1,-3],[3,-2]])
sage: elm2.is_elementary_symmetric()
False
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> elm = bd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)],[Integer(3),-
↪Integer(3)]])
>>> elm.is_elementary_symmetric()
True
>>> elm2 = bd([[Integer(1),Integer(2)],[-Integer(1),-Integer(3)],[Integer(3),-
↪Integer(2)]])
>>> elm2.is_elementary_symmetric()
False
```

options = Current options for Brauer diagram - display: normal

perm()

Return the induced bijection on the free nodes of `self` in one-line notation, re-indexed and treated as a permutation.

See also

bijection_on_free_nodes()

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-2,-3],[3,-1]])
sage: elm.perm()
[1]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> elm = bd([[Integer(1),Integer(2)],[-Integer(2),-Integer(3)],[Integer(3),-
↪Integer(1)]])
>>> elm.perm()
[1]
```

class sage.combinat.diagram_algebras.**BrauerDiagrams**(*order*, *category=None*)

Bases: *AbstractPartitionDiagrams*

This class represents all Brauer diagrams of integer or integer $+1/2$ order. For more information on Brauer diagrams, see *BrauerAlgebra*.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(2); bd
Brauer diagrams of order 2
sage: bd.list()
[{{-2, -1}, {1, 2}}, {{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]

sage: bd = da.BrauerDiagrams(5/2); bd
Brauer diagrams of order 5/2
sage: bd.list()
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(2)); bd
Brauer diagrams of order 2
>>> bd.list()
[{{-2, -1}, {1, 2}}, {{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]

>>> bd = da.BrauerDiagrams(Integer(5)/Integer(2)); bd
Brauer diagrams of order 5/2
>>> bd.list()
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
```

(continues on next page)

(continued from previous page)

```
{{-3, 3}, {-2, 2}, {-1, 1}}]
```

Element

alias of *BrauerDiagram*

cardinality()

Return the cardinality of `self`.

The number of Brauer diagrams of integer order k is $(2k-1)!!$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: bd.cardinality()
15

sage: bd = da.BrauerDiagrams(7/2)
sage: bd.cardinality()
15
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> bd.cardinality()
15

>>> bd = da.BrauerDiagrams(Integer(7)/Integer(2))
>>> bd.cardinality()
15
```

from_involution_permutation_triple(*D1_D2_pi*)

Construct a Brauer diagram of `self` from an involution permutation triple.

A Brauer diagram can be represented as a triple where the first entry is a list of arcs on the top row of the diagram, the second entry is a list of arcs on the bottom row of the diagram, and the third entry is a permutation on the remaining nodes. This triple is called the *involution permutation triple*. For more information, see [GL1996].

INPUT:

- `D1_D2_pi` – list or tuple where the first entry is a list of arcs on the top of the diagram, the second entry is a list of arcs on the bottom of the diagram, and the third entry is a permutation on the free nodes.

REFERENCES:

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(4)
sage: bd.from_involution_permutation_triple([[[1,2]],[[3,4]],[2,1]])
{{-4, -3}, {-2, 3}, {-1, 4}, {1, 2}}
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
```

(continues on next page)

(continued from previous page)

```
>>> bd = da.BrauerDiagrams(Integer(4))
>>> bd.from_involution_permutation_triple([[[Integer(1),Integer(2)]],
→[[Integer(3),Integer(4)]],[Integer(2),Integer(1)]])
{{-4, -3}, {-2, 3}, {-1, 4}, {1, 2}}
```

options = Current options for Brauer diagram - display: normal

symmetric_diagrams(*l=None*, *perm=None*)

Return the list of Brauer diagrams with symmetric placement of l arcs, and with free nodes permuted according to $perm$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(4)
sage: bd.symmetric_diagrams(l=1, perm=[2,1])
[{{-4, -2}, {-3, 1}, {-1, 3}, {2, 4}},
 {{-4, -3}, {-2, 1}, {-1, 2}, {3, 4}},
 {{-4, -1}, {-3, 2}, {-2, 3}, {1, 4}},
 {{-4, 2}, {-3, -1}, {-2, 4}, {1, 3}},
 {{-4, 3}, {-3, 4}, {-2, -1}, {1, 2}},
 {{-4, 1}, {-3, -2}, {-1, 4}, {2, 3}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(4))
>>> bd.symmetric_diagrams(l=Integer(1), perm=[Integer(2),Integer(1)])
[{{-4, -2}, {-3, 1}, {-1, 3}, {2, 4}},
 {{-4, -3}, {-2, 1}, {-1, 2}, {3, 4}},
 {{-4, -1}, {-3, 2}, {-2, 3}, {1, 4}},
 {{-4, 2}, {-3, -1}, {-2, 4}, {1, 3}},
 {{-4, 3}, {-3, 4}, {-2, -1}, {1, 2}},
 {{-4, 1}, {-3, -2}, {-1, 4}, {2, 3}}]
```

class sage.combinat.diagram_algebras.**DiagramAlgebra**(*k*, *q*, *base_ring*, *prefix*, *diagrams*, *category=None*)

Bases: `CombinatorialFreeModule`

Abstract class for diagram algebras and is not designed to be used directly.

class Element

Bases: `IndexedFreeModuleElement`

An element of a diagram algebra.

This subclass provides a few additional methods for partition algebra elements. Most element methods are already implemented elsewhere.

diagram()

Return the underlying diagram of `self` if `self` is a basis element. Raises an error if `self` is not a basis element.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = 3*P([[1,2],[-2,-1]])
sage: elt.diagram()
{{-2, -1}, {1, 2}}
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> elt = Integer(3)*P([[Integer(1),Integer(2)],[-Integer(2),-
↪Integer(1)]])
>>> elt.diagram()
{{-2, -1}, {1, 2}}
```

diagrams()

Return the diagrams in the support of `self`.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = 3*P([[1,2],[-2,-1]]) + P([[1,2],[-2], [-1]])
sage: sorted(elt.diagrams(), key=str)
[{{-2, -1}, {1, 2}}, {{-2}, {-1}, {1, 2}}]
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> elt = Integer(3)*P([[Integer(1),Integer(2)],[-Integer(2),-
↪Integer(1)]]) + P([[Integer(1),Integer(2)],[-Integer(2)], [-
↪Integer(1)]])
>>> sorted(elt.diagrams(), key=str)
[{{-2, -1}, {1, 2}}, {{-2}, {-1}, {1, 2}}]
```

order()

Return the order of `self`.

The order of a partition algebra is defined as half of the number of nodes in the diagrams.

EXAMPLES:

```
sage: q = var('q')                                                      #␣
↪needs sage.symbolic
sage: PA = PartitionAlgebra(2, q)                                       #␣
↪needs sage.symbolic
sage: PA.order()                                                        #␣
↪needs sage.symbolic
2
```

```
>>> from sage.all import *
>>> q = var('q')                                                      #␣
↪needs sage.symbolic
>>> PA = PartitionAlgebra(Integer(2), q)                               ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.symbolic
>>> PA.order()                                                           #␣
↪needs sage.symbolic
2
```

set_partitions()

Return the collection of underlying set partitions indexing the basis elements of a given diagram algebra.

Todo

Is this really necessary? deprecate?

class sage.combinat.diagram_algebras.**DiagramBasis**(*k*, *q*, *base_ring*, *prefix*, *diagrams*, *category=None*)

Bases: *DiagramAlgebra*

Abstract base class for diagram algebras in the diagram basis.

product_on_basis(*d1*, *d2*)

Return the product $D_{d_1} D_{d_2}$ by two basis diagrams.

class sage.combinat.diagram_algebras.**HalfTemperleyLiebDiagrams**(*order*, *defects*)

Bases: UniqueRepresentation, Parent

Half diagrams for the Temperley-Lieb algebra cell modules.

class Element(*parent*, *d*, *check=True*)

Bases: *AbstractPartitionDiagram*

check()

Check the validity of the input of `self`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: htld = da.HalfTemperleyLiebDiagrams(7, 3)
sage: htld([[1,2], [3,4]])  # indirect doctest
{{1, 2}, {3, 4}}
sage: htld([[1,2], [-1, -2]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: {{-2, -1}, {1, 2}} does not represent a half TL diagram of␣
↪order 7
sage: htld([[1,2,3], [4,5]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: all blocks of {{1, 2, 3}, {4, 5}} must be of size 2
sage: htld([[1,2], [3,4], [5,6]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: {{1, 2}, {3, 4}, {5, 6}} does not have 3 defects
sage: htld([[1,3], [2,4]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: {{1, 3}, {2, 4}} is not planar
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> htld = da.HalfTemperleyLiebDiagrams(Integer(7), Integer(3))
>>> htld([[Integer(1),Integer(2)], [Integer(3),Integer(4)]])  # indirect␣
↪doctest
{{1, 2}, {3, 4}}
>>> htld([[Integer(1),Integer(2)], [-Integer(1), -Integer(2)]])  #␣
↪indirect doctest
Traceback (most recent call last):
...
ValueError: {{-2, -1}, {1, 2}} does not represent a half TL diagram of␣
↪order 7
>>> htld([[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5)]])
↪# indirect doctest
Traceback (most recent call last):
...
ValueError: all blocks of {{1, 2, 3}, {4, 5}} must be of size 2
>>> htld([[Integer(1),Integer(2)], [Integer(3),Integer(4)], [Integer(5),
↪Integer(6)]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: {{1, 2}, {3, 4}, {5, 6}} does not have 3 defects
>>> htld([[Integer(1),Integer(3)], [Integer(2),Integer(4)]])  # indirect␣
↪doctest
Traceback (most recent call last):
...
ValueError: {{1, 3}, {2, 4}} is not planar
```

defects()

Return the defects of `self`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: htld = da.HalfTemperleyLiebDiagrams(7, 3)
sage: d = htld([[1, 2], [4, 5]])
sage: d.defects()
frozenset({3, 6, 7})
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> htld = da.HalfTemperleyLiebDiagrams(Integer(7), Integer(3))
>>> d = htld([[Integer(1), Integer(2)], [Integer(4), Integer(5)]])
>>> d.defects()
frozenset({3, 6, 7})
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: htld = da.HalfTemperleyLiebDiagrams(7, 3)
```

(continues on next page)

(continued from previous page)

```
sage: htld.cardinality()
14
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> htld = da.HalfTemperleyLiebDiagrams(Integer(7), Integer(3))
>>> htld.cardinality()
14
```

class sage.combinat.diagram_algebras.**IdealDiagram**(*parent*, *d*, *check=True*)

Bases: *AbstractPartitionDiagram*

The element class for a ideal diagram.

An ideal diagram for an integer k is a partition of the set $\{1, \ldots, k, -1, \ldots, -k\}$ where the propagating number is strictly smaller than the order.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import IdealDiagrams as IDs
sage: IDs(2)
Ideal diagrams of order 2
sage: IDs(2).list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2}, {-1}, {1, 2}},
 {{-2, -1, 1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1}, {1}, {2}}]

sage: from sage.combinat.diagram_algebras import PartitionDiagrams as PDs
sage: PDs(4).cardinality() == factorial(4) + IDs(4).cardinality()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import IdealDiagrams as IDs
>>> IDs(Integer(2))
Ideal diagrams of order 2
>>> IDs(Integer(2)).list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2}, {-1}, {1, 2}},
 {{-2, -1, 1}, {2}},
 {{-2, 1}, {-1}, {2}},
```

(continues on next page)

(continued from previous page)

```
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1}, {1}, {2}}]

>>> from sage.combinat.diagram_algebras import PartitionDiagrams as PDs
>>> PDs(Integer(4)).cardinality() == factorial(Integer(4)) + IDs(Integer(4)).
↪cardinality()
True
```

check()

Check the validity of the input for `self`.

class sage.combinat.diagram_algebras.**IdealDiagrams**(*order*, *category=None*)

Bases: *AbstractPartitionDiagrams*

All "ideal" diagrams of integer or integer $+1/2$ order.

If k is an integer then an ideal diagram of order k is a partition diagram of order k with propagating number less than k.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: id = da.IdealDiagrams(3)
sage: id.an_element() in id
True
sage: id.cardinality() == len(id.list())
True
sage: da.IdealDiagrams(3/2).list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> id = da.IdealDiagrams(Integer(3))
>>> id.an_element() in id
True
>>> id.cardinality() == len(id.list())
True
>>> da.IdealDiagrams(Integer(3)/Integer(2)).list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]
```

Element

alias of *IdealDiagram*

class sage.combinat.diagram_algebras.**OrbitBasis**(*alg*)

Bases: *DiagramAlgebra*

The orbit basis of the partition algebra.

Let D_π represent the diagram basis element indexed by the partition π, then (see equations (2.14), (2.17) and (2.18) of [BH2017])

$$D_\pi = \sum_{\tau \geq \pi} O_\tau,$$

where the sum is over all partitions τ which are coarser than π and O_τ is the orbit basis element indexed by the partition τ.

If $\mu_{2k}(\pi, \tau)$ represents the Moebius function of the partition lattice, then

$$O_\pi = \sum_{\tau \geq \pi} \mu_{2k}(\pi, \tau) D_\tau.$$

If τ is a partition of ℓ blocks and the i-th block of τ is a union of b_i blocks of π, then

$$\mu_{2k}(\pi, \tau) = \prod_{i=1}^{\ell} (-1)^{b_i - 1} (b_i - 1)!.$$

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P2 = PartitionAlgebra(2, x, R)
sage: O2 = P2.orbit_basis(); O2
Orbit basis of Partition Algebra of rank 2 with parameter x over
 Univariate Polynomial Ring in x over Rational Field
sage: oa = O2([[1],[-1],[2,-2]]); ob = O2([[-1,-2,2],[1]]); oa, ob
(O{{-2, 2}, {-1}, {1}}, O{{-2, -1, 2}, {1}})
sage: oa * ob
(x-2)*O{{-2, -1, 2}, {1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P2 = PartitionAlgebra(Integer(2), x, R)
>>> O2 = P2.orbit_basis(); O2
Orbit basis of Partition Algebra of rank 2 with parameter x over
 Univariate Polynomial Ring in x over Rational Field
>>> oa = O2([[Integer(1)],[-Integer(1)],[Integer(2),-Integer(2)]]); ob = O2([[-
↪Integer(1),-Integer(2),Integer(2)],[Integer(1)]]); oa, ob
(O{{-2, 2}, {-1}, {1}}, O{{-2, -1, 2}, {1}})
>>> oa * ob
(x-2)*O{{-2, -1, 2}, {1}}
```

We can convert between the two bases:

```
sage: pa = P2(oa); pa
2*P{{-2, -1, 1, 2}} - P{{-2, -1, 2}, {1}} - P{{-2, 1, 2}, {-1}}
 + P{{-2, 2}, {-1}, {1}} - P{{-2, 2}, {-1, 1}}
sage: pa * ob
(-x+2)*P{{-2, -1, 1, 2}} + (x-2)*P{{-2, -1, 2}, {1}}
```

(continues on next page)

(continued from previous page)

```
sage: _ == pa * P2(ob)
True
sage: O2(pa * ob)
(x-2)*O{{-2, -1, 2}, {1}}
```

```
>>> from sage.all import *
>>> pa = P2(oa); pa
2*P{{-2, -1, 1, 2}} - P{{-2, -1, 2}, {1}} - P{{-2, 1, 2}, {-1}}
 + P{{-2, 2}, {-1}, {1}} - P{{-2, 2}, {-1, 1}}
>>> pa * ob
(-x+2)*P{{-2, -1, 1, 2}} + (x-2)*P{{-2, -1, 2}, {1}}
>>> _ == pa * P2(ob)
True
>>> O2(pa * ob)
(x-2)*O{{-2, -1, 2}, {1}}
```

Note that the unit in the orbit basis is not a single diagram, in contrast to the natural diagram basis:

```
sage: P2.one()
P{{-2, 2}, {-1, 1}}
sage: O2.one()
O{{-2, -1, 1, 2}} + O{{-2, 2}, {-1, 1}}
sage: O2.one() == P2.one()
True
```

```
>>> from sage.all import *
>>> P2.one()
P{{-2, 2}, {-1, 1}}
>>> O2.one()
O{{-2, -1, 1, 2}} + O{{-2, 2}, {-1, 1}}
>>> O2.one() == P2.one()
True
```

class Element

Bases: *Element*

to_diagram_basis()

Expand `self` in the natural diagram basis of the partition algebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(2, x, R)
sage: O = P.orbit_basis()
sage: elt = O.an_element(); elt
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: elt.to_diagram_basis()
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
sage: pp = P.an_element(); pp
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
sage: op = pp.to_orbit_basis(); op
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
```

(continues on next page)

(continued from previous page)

```
sage: pp == op.to_diagram_basis()
True
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> O = P.orbit_basis()
>>> elt = O.an_element(); elt
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> elt.to_diagram_basis()
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
>>> pp = P.an_element(); pp
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
>>> op = pp.to_orbit_basis(); op
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> pp == op.to_diagram_basis()
True
```

diagram_basis()

Return the associated partition algebra of `self` in the diagram basis.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: O2 = PartitionAlgebra(2, x, R).orbit_basis()
sage: P2 = O2.diagram_basis(); P2
Partition Algebra of rank 2 with parameter x over Univariate
Polynomial Ring in x over Rational Field
sage: o2 = O2.an_element(); o2
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: P2(o2)
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> O2 = PartitionAlgebra(Integer(2), x, R).orbit_basis()
>>> P2 = O2.diagram_basis(); P2
Partition Algebra of rank 2 with parameter x over Univariate
Polynomial Ring in x over Rational Field
>>> o2 = O2.an_element(); o2
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> P2(o2)
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

one()

Return the element 1 of the partition algebra in the orbit basis.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P2 = PartitionAlgebra(2, x, R)
sage: O2 = P2.orbit_basis()
```

(continues on next page)

(continued from previous page)

```
sage: O2.one()
O{{-2, -1, 1, 2}} + O{{-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P2 = PartitionAlgebra(Integer(2), x, R)
>>> O2 = P2.orbit_basis()
>>> O2.one()
O{{-2, -1, 1, 2}} + O{{-2, 2}, {-1, 1}}
```

product_on_basis(*d1*, *d2*)

Return the product $O_{d_1} O_{d_2}$ of two elements in the orbit basis `self`.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: OP = PartitionAlgebra(2, x, R).orbit_basis()
sage: SP = OP.basis().keys(); sp = SP([[-2, -1, 1, 2]])
sage: OP.product_on_basis(sp, sp)
O{{-2, -1, 1, 2}}
sage: o1 = OP.one(); o2 = OP([]); o3 = OP.an_element()
sage: o2 == o1
False
sage: o1 * o1 == o1
True
sage: o3 * o1 == o1 * o3 and o3 * o1 == o3
True
sage: o4 = (3*OP([[-2, -1, 1], [2]]) + 2*OP([[-2, -1, 1, 2]])
....:       + 2*OP([[-2, -1, 2], [1]]))
sage: o4 * o4
6*O{{-2, -1, 1}, {2}} + 4*O{{-2, -1, 1, 2}} + 4*O{{-2, -1, 2}, {1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> OP = PartitionAlgebra(Integer(2), x, R).orbit_basis()
>>> SP = OP.basis().keys(); sp = SP([[-Integer(2), -Integer(1), Integer(1),␣
↪Integer(2)]])
>>> OP.product_on_basis(sp, sp)
O{{-2, -1, 1, 2}}
>>> o1 = OP.one(); o2 = OP([]); o3 = OP.an_element()
>>> o2 == o1
False
>>> o1 * o1 == o1
True
>>> o3 * o1 == o1 * o3 and o3 * o1 == o3
True
>>> o4 = (Integer(3)*OP([[-Integer(2), -Integer(1), Integer(1)],␣
↪[Integer(2)]]) + Integer(2)*OP([[-Integer(2), -Integer(1), Integer(1),␣
↪Integer(2)]])
...       + Integer(2)*OP([[-Integer(2), -Integer(1), Integer(2)],␣
↪[Integer(1)]]))
>>> o4 * o4
```

(continues on next page)

(continued from previous page)

```
6*O{{-2, -1, 1}, {2}} + 4*O{{-2, -1, 1, 2}} + 4*O{{-2, -1, 2}, {1}}
```

We compute Examples 4.5 in [BH2017]:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(3,x); O = P.orbit_basis()
sage: O[[1,2,3],[-1,-2,-3]] * O[[1,2,3],[-1,-2,-3]]
(x-2)*O{{-3, -2, -1}, {1, 2, 3}} + (x-1)*O{{-3, -2, -1, 1, 2, 3}}

sage: P = PartitionAlgebra(4,x); O = P.orbit_basis()
sage: O[[1],[-1],[2,3],[4,-2],[-3,-4]] * O[[1],[2,-2],[3,4],[-1,-3],[-4]]
(x^2-11*x+30)*O{{-4}, {-3, -1}, {-2, 4}, {1}, {2, 3}}
 + (x^2-9*x+20)*O{{-4}, {-3, -1, 1}, {-2, 4}, {2, 3}}
 + (x^2-9*x+20)*O{{-4}, {-3, -1, 2, 3}, {-2, 4}, {1}}
 + (x^2-9*x+20)*O{{-4, 1}, {-3, -1}, {-2, 4}, {2, 3}}
 + (x^2-7*x+12)*O{{-4, 1}, {-3, -1, 2, 3}, {-2, 4}}
 + (x^2-9*x+20)*O{{-4, 2, 3}, {-3, -1}, {-2, 4}, {1}}
 + (x^2-7*x+12)*O{{-4, 2, 3}, {-3, -1, 1}, {-2, 4}}

sage: O[[1,-1],[2,-2],[3],[4,-3],[-4]] * O[[1,-2],[2],[3,-1],[4],[-3],[-4]]
(x-6)*O{{-4}, {-3}, {-2, 1}, {-1, 4}, {2}, {3}}
 + (x-5)*O{{-4}, {-3, 3}, {-2, 1}, {-1, 4}, {2}}
 + (x-5)*O{{-4, 3}, {-3}, {-2, 1}, {-1, 4}, {2}}

sage: P = PartitionAlgebra(6,x); O = P.orbit_basis()
sage: (O[[1,-2,-3],[2,4],[3,5,-6],[6],[-1],[-4,-5]]
....:  * O[[1,-2],[2,3],[4],[5],[6,-4,-5,-6],[-1,-3]])
0

sage: (O[[1,-2],[2,-3],[3,5],[4,-5],[6,-4],[-1],[-6]]
....:  * O[[1,-2],[2,-1],[3,-4],[4,-6],[5,-3],[6,-5]])
O{{-6, 6}, {-5}, {-4, 2}, {-3, 4}, {-2}, {-1, 1}, {3, 5}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(3),x); O = P.orbit_basis()
>>> O[[Integer(1),Integer(2),Integer(3)],[-Integer(1),-Integer(2),-
↪Integer(3)]] * O[[Integer(1),Integer(2),Integer(3)],[-Integer(1),-
↪Integer(2),-Integer(3)]]
(x-2)*O{{-3, -2, -1}, {1, 2, 3}} + (x-1)*O{{-3, -2, -1, 1, 2, 3}}

>>> P = PartitionAlgebra(Integer(4),x); O = P.orbit_basis()
>>> O[[Integer(1)],[-Integer(1)],[Integer(2),Integer(3)],[Integer(4),-
↪Integer(2)],[-Integer(3),-Integer(4)]] * O[[Integer(1)],[Integer(2),-
↪Integer(2)],[Integer(3),Integer(4)],[-Integer(1),-Integer(3)],[-Integer(4)]]
(x^2-11*x+30)*O{{-4}, {-3, -1}, {-2, 4}, {1}, {2, 3}}
 + (x^2-9*x+20)*O{{-4}, {-3, -1, 1}, {-2, 4}, {2, 3}}
 + (x^2-9*x+20)*O{{-4}, {-3, -1, 2, 3}, {-2, 4}, {1}}
 + (x^2-9*x+20)*O{{-4, 1}, {-3, -1}, {-2, 4}, {2, 3}}
 + (x^2-7*x+12)*O{{-4, 1}, {-3, -1, 2, 3}, {-2, 4}}
 + (x^2-9*x+20)*O{{-4, 2, 3}, {-3, -1}, {-2, 4}, {1}}
 + (x^2-7*x+12)*O{{-4, 2, 3}, {-3, -1, 1}, {-2, 4}}
```

(continues on next page)

(continued from previous page)

```
>>> O[[Integer(1),-Integer(1)],[Integer(2),-Integer(2)],[Integer(3)],
↪[Integer(4),-Integer(3)],[-Integer(4)]] * O[[Integer(1),-Integer(2)],
↪[Integer(2)],[Integer(3),-Integer(1)],[Integer(4)],[-Integer(3)],[-
↪Integer(4)]]
(x-6)*O{{-4}, {-3}, {-2, 1}, {-1, 4}, {2}, {3}}
 + (x-5)*O{{-4}, {-3, 3}, {-2, 1}, {-1, 4}, {2}}
 + (x-5)*O{{-4, 3}, {-3}, {-2, 1}, {-1, 4}, {2}}

>>> P = PartitionAlgebra(Integer(6),x); O = P.orbit_basis()
>>> (O[[Integer(1),-Integer(2),-Integer(3)],[Integer(2),Integer(4)],
↪[Integer(3),Integer(5),-Integer(6)],[Integer(6)],[-Integer(1)],[-Integer(4),
↪-Integer(5)]]
...  * O[[Integer(1),-Integer(2)],[Integer(2),Integer(3)],[Integer(4)],
↪[Integer(5)],[Integer(6),-Integer(4),-Integer(5),-Integer(6)],[-Integer(1),-
↪Integer(3)]])
0

>>> (O[[Integer(1),-Integer(2)],[Integer(2),-Integer(3)],[Integer(3),
↪Integer(5)],[Integer(4),-Integer(5)],[Integer(6),-Integer(4)],[-Integer(1)],
↪[-Integer(6)]]
...  * O[[Integer(1),-Integer(2)],[Integer(2),-Integer(1)],[Integer(3),-
↪Integer(4)],[Integer(4),-Integer(6)],[Integer(5),-Integer(3)],[Integer(6),-
↪Integer(5)]])
O{{-6, 6}, {-5}, {-4, 2}, {-3, 4}, {-2}, {-1, 1}, {3, 5}}
```

REFERENCES:

- [BH2017]

class sage.combinat.diagram_algebras.**PartitionAlgebra**(*k*, *q*, *base_ring*, *prefix*)

Bases: *DiagramBasis*, *UnitDiagramMixin*

A partition algebra.

A partition algebra of rank k over a given ground ring R is an algebra with (R-module) basis indexed by the collection of set partitions of $\{1,\ldots,k,-1,\ldots,-k\}$. Each such set partition can be represented by a graph on nodes $\{1,\ldots,k,-1,\ldots,-k\}$ arranged in two rows, with nodes $1,\ldots,k$ in the top row from left to right and with nodes $-1,\ldots,-k$ in the bottom row from left to right, and edges drawn such that the connected components of the graph are precisely the parts of the set partition. (This choice of edges is often not unique, and so there are often many graphs representing one and the same set partition; the representation nevertheless is useful and vivid. We often speak of "diagrams" to mean graphs up to such equivalence of choices of edges; of course, we could just as well speak of set partitions.)

There is not just one partition algebra of given rank over a given ground ring, but rather a whole family of them, indexed by the elements of R. More precisely, for every $q\in R$, the partition algebra of rank k over R with parameter q is defined to be the R-algebra with basis the collection of all set partitions of $\{1,\ldots,k,-1,\ldots,-k\}$, where the product of two basis elements is given by the rule

$$a\cdot b=q^N(a\circ b),$$

where $a\circ b$ is the composite set partition obtained by placing the diagram (i.e., graph) of a above the diagram of b, identifying the bottom row nodes of a with the top row nodes of b, and omitting any closed "loops" in the middle. The number N is the number of connected components formed by the omitted loops.

The parameter q is a deformation parameter. Taking $q = 1$ produces the semigroup algebra (over the base ring) of the partition monoid, in which the product of two set partitions is simply given by their composition.

The partition algebra is regarded as an example of a "diagram algebra" due to the fact that its natural basis is given by certain graphs often called diagrams.

There are a number of predefined elements for the partition algebra. We define the cup/cap pair by `a()`. The simple transpositions are denoted `s()`. Finally, we define elements `e()`, where if $i = (2r + 1)/2$, then `e(i)` contains the blocks $\{r + 1\}$ and $\{-r - 1\}$ and if $i \in \mathbf{Z}$, then e_i contains the block $\{-i, -i - 1, i, i + 1\}$, with all other blocks being $\{-j, j\}$. So we have:

```
sage: P = PartitionAlgebra(4, 0)
sage: P.a(2)
P{{-4, 4}, {-3, -2}, {-1, 1}, {2, 3}}
sage: P.e(3/2)
P{{-4, 4}, {-3, 3}, {-2}, {-1, 1}, {2}}
sage: P.e(2)
P{{-4, 4}, {-3, -2, 2, 3}, {-1, 1}}
sage: P.e(5/2)
P{{-4, 4}, {-3}, {-2, 2}, {-1, 1}, {3}}
sage: P.s(2)
P{{-4, 4}, {-3, 2}, {-2, 3}, {-1, 1}}
```

```
>>> from sage.all import *
>>> P = PartitionAlgebra(Integer(4), Integer(0))
>>> P.a(Integer(2))
P{{-4, 4}, {-3, -2}, {-1, 1}, {2, 3}}
>>> P.e(Integer(3)/Integer(2))
P{{-4, 4}, {-3, 3}, {-2}, {-1, 1}, {2}}
>>> P.e(Integer(2))
P{{-4, 4}, {-3, -2, 2, 3}, {-1, 1}}
>>> P.e(Integer(5)/Integer(2))
P{{-4, 4}, {-3}, {-2, 2}, {-1, 1}, {3}}
>>> P.s(Integer(2))
P{{-4, 4}, {-3, 2}, {-2, 3}, {-1, 1}}
```

An excellent reference for partition algebras and their various subalgebras (Brauer algebra, Temperley–Lieb algebra, etc) is the paper [HR2005].

INPUT:

- `k` – rank of the algebra
- `q` – the deformation parameter q
- `base_ring` – (default: `None`) a ring containing `q`; if `None`, then Sage automatically chooses the parent of `q`
- `prefix` – (default: `'P'`) a label for the basis elements

EXAMPLES:

The following shorthand simultaneously defines the univariate polynomial ring over the rationals as well as the variable `x`:

```
sage: R.<x> = PolynomialRing(QQ)
sage: R
Univariate Polynomial Ring in x over Rational Field
sage: x
```

(continues on next page)

(continued from previous page)

```
x
sage: x.parent() is R
True
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> R
Univariate Polynomial Ring in x over Rational Field
>>> x
x
>>> x.parent() is R
True
```

We now define the partition algebra of rank 2 with parameter x over **Z** in the usual (diagram) basis:

```
sage: R.<x> = ZZ[]
sage: A2 = PartitionAlgebra(2, x, R)
sage: A2
Partition Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
sage: A2.basis().keys()
Partition diagrams of order 2
sage: A2.basis().keys()([[-2, 1, 2], [-1]])
{{-2, 1, 2}, {-1}}
sage: A2.basis().list()
[P{{-2, -1, 1, 2}}, P{{-2, 1, 2}, {-1}},
 P{{-2}, {-1, 1, 2}}, P{{-2, -1}, {1, 2}},
 P{{-2}, {-1}, {1, 2}}, P{{-2, -1, 1}, {2}},
 P{{-2, 1}, {-1, 2}}, P{{-2, 1}, {-1}, {2}},
 P{{-2, 2}, {-1, 1}}, P{{-2, -1, 2}, {1}},
 P{{-2, 2}, {-1}, {1}}, P{{-2}, {-1, 1}, {2}},
 P{{-2}, {-1, 2}, {1}}, P{{-2, -1}, {1}, {2}},
 P{{-2}, {-1}, {1}, {2}}]
sage: E = A2([[1,2],[-2,-1]]); E
P{{-2, -1}, {1, 2}}
sage: E in A2.basis().list()
True
sage: E^2
x*P{{-2, -1}, {1, 2}}
sage: E^5
x^4*P{{-2, -1}, {1, 2}}
sage: (A2([[2,-2],[-1,1]]) - 2*A2([[1,2],[-1,-2]]))^2
(4*x-4)*P{{-2, -1}, {1, 2}} + P{{-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> A2 = PartitionAlgebra(Integer(2), x, R)
>>> A2
Partition Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
>>> A2.basis().keys()
Partition diagrams of order 2
```

(continues on next page)

(continued from previous page)

```
>>> A2.basis().keys()([[-Integer(2), Integer(1), Integer(2)], [-Integer(1)]])
{{-2, 1, 2}, {-1}}
>>> A2.basis().list()
[P{{-2, -1, 1, 2}}, P{{-2, 1, 2}, {-1}},
 P{{-2}, {-1, 1, 2}}, P{{-2, -1}, {1, 2}},
 P{{-2}, {-1}, {1, 2}}, P{{-2, -1, 1}, {2}},
 P{{-2, 1}, {-1, 2}}, P{{-2, 1}, {-1}, {2}},
 P{{-2, 2}, {-1, 1}}, P{{-2, -1, 2}, {1}},
 P{{-2, 2}, {-1}, {1}}, P{{-2}, {-1, 1}, {2}},
 P{{-2}, {-1, 2}, {1}}, P{{-2, -1}, {1}, {2}},
 P{{-2}, {-1}, {1}, {2}}]
>>> E = A2([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]]); E
P{{-2, -1}, {1, 2}}
>>> E in A2.basis().list()
True
>>> E**Integer(2)
x*P{{-2, -1}, {1, 2}}
>>> E**Integer(5)
x^4*P{{-2, -1}, {1, 2}}
>>> (A2([[Integer(2),-Integer(2)],[-Integer(1),Integer(1)]]) -␣
↪Integer(2)*A2([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]))**Integer(2)
(4*x-4)*P{{-2, -1}, {1, 2}} + P{{-2, 2}, {-1, 1}}
```

Next, we construct an element:

```
sage: a2 = A2.an_element(); a2
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

```
>>> from sage.all import *
>>> a2 = A2.an_element(); a2
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

There is a natural embedding into partition algebras on more elements, by adding identity strands:

```
sage: A4 = PartitionAlgebra(4, x, R)
sage: A4(a2)
3*P{{-4, 4}, {-3, 3}, {-2}, {-1, 1, 2}}
 + 2*P{{-4, 4}, {-3, 3}, {-2, -1, 1, 2}}
 + 2*P{{-4, 4}, {-3, 3}, {-2, 1, 2}, {-1}}
```

```
>>> from sage.all import *
>>> A4 = PartitionAlgebra(Integer(4), x, R)
>>> A4(a2)
3*P{{-4, 4}, {-3, 3}, {-2}, {-1, 1, 2}}
 + 2*P{{-4, 4}, {-3, 3}, {-2, -1, 1, 2}}
 + 2*P{{-4, 4}, {-3, 3}, {-2, 1, 2}, {-1}}
```

Thus, the empty partition corresponds to the identity:

```
sage: A4([])
P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
sage: A4(5)
```

(continues on next page)

(continued from previous page)

```
5*P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> A4([])
P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
>>> A4(Integer(5))
5*P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
```

The group algebra of the symmetric group is a subalgebra:

```
sage: S3 = SymmetricGroupAlgebra(ZZ, 3)
sage: s3 = S3.an_element(); s3
[1, 2, 3] + 2*[1, 3, 2] + 3*[2, 1, 3] + [3, 1, 2]
sage: A4(s3)
P{{-4, 4}, {-3, 1}, {-2, 3}, {-1, 2}}
 + 2*P{{-4, 4}, {-3, 2}, {-2, 3}, {-1, 1}}
 + 3*P{{-4, 4}, {-3, 3}, {-2, 1}, {-1, 2}}
 + P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
sage: A4([2,1])
P{{-4, 4}, {-3, 3}, {-2, 1}, {-1, 2}}
```

```
>>> from sage.all import *
>>> S3 = SymmetricGroupAlgebra(ZZ, Integer(3))
>>> s3 = S3.an_element(); s3
[1, 2, 3] + 2*[1, 3, 2] + 3*[2, 1, 3] + [3, 1, 2]
>>> A4(s3)
P{{-4, 4}, {-3, 1}, {-2, 3}, {-1, 2}}
 + 2*P{{-4, 4}, {-3, 2}, {-2, 3}, {-1, 1}}
 + 3*P{{-4, 4}, {-3, 3}, {-2, 1}, {-1, 2}}
 + P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
>>> A4([Integer(2),Integer(1)])
P{{-4, 4}, {-3, 3}, {-2, 1}, {-1, 2}}
```

Be careful not to confuse the embedding of the group algebra of the symmetric group with the embedding of partial set partitions. The latter are embedded by adding the parts $\{i, -i\}$ if possible, and singletons sets for the remaining parts:

```
sage: A4([[2,1]])
P{{-4, 4}, {-3, 3}, {-2}, {-1}, {1, 2}}
sage: A4([[-1,3],[-2,-3,1]])
P{{-4, 4}, {-3, -2, 1}, {-1, 3}, {2}}
```

```
>>> from sage.all import *
>>> A4([[Integer(2),Integer(1)]])
P{{-4, 4}, {-3, 3}, {-2}, {-1}, {1, 2}}
>>> A4([[-Integer(1),Integer(3)],[-Integer(2),-Integer(3),Integer(1)]])
P{{-4, 4}, {-3, -2, 1}, {-1, 3}, {2}}
```

Another subalgebra is the Brauer algebra, which has perfect matchings as basis elements. The group algebra of the symmetric group is in fact a subalgebra of the Brauer algebra:

```
sage: B3 = BrauerAlgebra(3, x, R)
sage: b3 = B3(s3); b3
B{{-3, 1}, {-2, 3}, {-1, 2}} + 2*B{{-3, 2}, {-2, 3}, {-1, 1}}
 + 3*B{{-3, 3}, {-2, 1}, {-1, 2}} + B{{-3, 3}, {-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> B3 = BrauerAlgebra(Integer(3), x, R)
>>> b3 = B3(s3); b3
B{{-3, 1}, {-2, 3}, {-1, 2}} + 2*B{{-3, 2}, {-2, 3}, {-1, 1}}
 + 3*B{{-3, 3}, {-2, 1}, {-1, 2}} + B{{-3, 3}, {-2, 2}, {-1, 1}}
```

An important basis of the partition algebra is the `orbit basis`:

```
sage: O2 = A2.orbit_basis()
sage: o2 = O2([[1,2],[-1,-2]]) + O2([[1,2,-1,-2]]); o2
O{{-2, -1}, {1, 2}} + O{{-2, -1, 1, 2}}
```

```
>>> from sage.all import *
>>> O2 = A2.orbit_basis()
>>> o2 = O2([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]) +␣
→O2([[Integer(1),Integer(2),-Integer(1),-Integer(2)]]); o2
O{{-2, -1}, {1, 2}} + O{{-2, -1, 1, 2}}
```

The diagram basis element corresponds to the sum of all orbit basis elements indexed by coarser set partitions:

```
sage: A2(o2)
P{{-2, -1}, {1, 2}}
```

```
>>> from sage.all import *
>>> A2(o2)
P{{-2, -1}, {1, 2}}
```

We can convert back from the orbit basis to the diagram basis:

```
sage: o2 = O2.an_element(); o2
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: A2(o2)
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

```
>>> from sage.all import *
>>> o2 = O2.an_element(); o2
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> A2(o2)
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

One can work with partition algebras using a symbol for the parameter, leaving the base ring unspecified. This implies that the underlying base ring is Sage's symbolic ring.

```
sage: # needs sage.symbolic
sage: q = var('q')
sage: PA = PartitionAlgebra(2, q); PA
Partition Algebra of rank 2 with parameter q over Symbolic Ring
```

(continues on next page)

(continued from previous page)

```
sage: PA([[1,2],[-2,-1]])^2 == q*PA([[1,2],[-2,-1]])
True
sage: ((PA([[2, -2], [1, -1]]) - 2*PA([[-2, -1], [1, 2]]))^2
....:    == (4*q-4)*PA([[1, 2], [-2, -1]]) + PA([[2, -2], [1, -1]]))
True
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> q = var('q')
>>> PA = PartitionAlgebra(Integer(2), q); PA
Partition Algebra of rank 2 with parameter q over Symbolic Ring
>>> PA([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])**Integer(2) ==␣
↪q*PA([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])
True
>>> ((PA([[Integer(2), -Integer(2)], [Integer(1), -Integer(1)]]) -␣
↪Integer(2)*PA([[-Integer(2), -Integer(1)], [Integer(1),␣
↪Integer(2)]]))**Integer(2)
...    == (Integer(4)*q-Integer(4))*PA([[Integer(1), Integer(2)], [-Integer(2), -
↪Integer(1)]]) + PA([[Integer(2), -Integer(2)], [Integer(1), -Integer(1)]]))
True
```

The identity element of the partition algebra is the set partition $\{\{1,-1\},\{2,-2\},\ldots,\{k,-k\}\}$:

```
sage: # needs sage.symbolic
sage: P = PA.basis().list()
sage: PA.one()
P{{-2, 2}, {-1, 1}}
sage: PA.one() * P[7] == P[7]
True
sage: P[7] * PA.one() == P[7]
True
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> P = PA.basis().list()
>>> PA.one()
P{{-2, 2}, {-1, 1}}
>>> PA.one() * P[Integer(7)] == P[Integer(7)]
True
>>> P[Integer(7)] * PA.one() == P[Integer(7)]
True
```

We now give some further examples of the use of the other arguments. One may wish to "specialize" the parameter to a chosen element of the base ring:

```
sage: R.<q> = RR[]
sage: PA = PartitionAlgebra(2, q, R, prefix='B')
sage: PA
Partition Algebra of rank 2 with parameter q over
 Univariate Polynomial Ring in q over Real Field with 53 bits of precision
sage: PA([[1,2],[-1,-2]])
1.00000000000000*B{{-2, -1}, {1, 2}}
```

(continues on next page)

(continued from previous page)

```
sage: PA = PartitionAlgebra(2, 5, base_ring=ZZ, prefix='B')
sage: PA
Partition Algebra of rank 2 with parameter 5 over Integer Ring
sage: ((PA([[2, -2], [1, -1]]) - 2*PA([[-2, -1], [1, 2]]))^2
....:   == 16*PA([[-2, -1], [1, 2]]) + PA([[2, -2], [1, -1]]))
True
```

```
>>> from sage.all import *
>>> R = RR['q']; (q,) = R._first_ngens(1)
>>> PA = PartitionAlgebra(Integer(2), q, R, prefix='B')
>>> PA
Partition Algebra of rank 2 with parameter q over
 Univariate Polynomial Ring in q over Real Field with 53 bits of precision
>>> PA([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
1.00000000000000*B{{-2, -1}, {1, 2}}
>>> PA = PartitionAlgebra(Integer(2), Integer(5), base_ring=ZZ, prefix='B')
>>> PA
Partition Algebra of rank 2 with parameter 5 over Integer Ring
>>> ((PA([[Integer(2), -Integer(2)], [Integer(1), -Integer(1)]]) -␣
↪Integer(2)*PA([[-Integer(2), -Integer(1)], [Integer(1),␣
↪Integer(2)]]))**Integer(2)
...   == Integer(16)*PA([[-Integer(2), -Integer(1)], [Integer(1), Integer(2)]]) +␣
↪PA([[Integer(2), -Integer(2)], [Integer(1), -Integer(1)]]))
True
```

Symmetric group algebra elements and elements from other subalgebras of the partition algebra (e.g., `BrauerAlgebra` and `TemperleyLiebAlgebra`) can also be coerced into the partition algebra:

```
sage: # needs sage.symbolic
sage: S = SymmetricGroupAlgebra(SR, 2)
sage: B = BrauerAlgebra(2, x, SR)
sage: A = PartitionAlgebra(2, x, SR)
sage: S([2,1]) * A([[1,-1],[2,-2]])
P{{-2, 1}, {-1, 2}}
sage: B([[-1,-2],[2,1]]) * A([[1],[-1],[2,-2]])
P{{-2}, {-1}, {1, 2}}
sage: A([[1],[-1],[2,-2]]) * B([[-1,-2],[2,1]])
P{{-2, -1}, {1}, {2}}
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> S = SymmetricGroupAlgebra(SR, Integer(2))
>>> B = BrauerAlgebra(Integer(2), x, SR)
>>> A = PartitionAlgebra(Integer(2), x, SR)
>>> S([Integer(2),Integer(1)]) * A([[Integer(1),-Integer(1)],[Integer(2),-
↪Integer(2)]])
P{{-2, 1}, {-1, 2}}
>>> B([[-Integer(1),-Integer(2)],[Integer(2),Integer(1)]]) * A([[Integer(1)],[-
↪Integer(1)],[Integer(2),-Integer(2)]])
P{{-2}, {-1}, {1, 2}}
>>> A([[Integer(1)],[-Integer(1)],[Integer(2),-Integer(2)]]) * B([[-Integer(1),-
↪Integer(2)],[Integer(2),Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
P{{-2, -1}, {1}, {2}}
```

The same is true if the elements come from a subalgebra of a partition algebra of smaller order, or if they are defined over a different base ring:

```
sage: R = FractionField(ZZ['q']); q = R.gen()
sage: S = SymmetricGroupAlgebra(ZZ, 2)
sage: B = BrauerAlgebra(2, q, ZZ[q])
sage: A = PartitionAlgebra(3, q, R)
sage: S([2,1]) * A([[1,-1],[2,-3],[3,-2]])
P{{-3, 1}, {-2, 3}, {-1, 2}}
sage: A(B([[-1,-2],[2,1]]))
P{{-3, 3}, {-2, -1}, {1, 2}}
```

```
>>> from sage.all import *
>>> R = FractionField(ZZ['q']); q = R.gen()
>>> S = SymmetricGroupAlgebra(ZZ, Integer(2))
>>> B = BrauerAlgebra(Integer(2), q, ZZ[q])
>>> A = PartitionAlgebra(Integer(3), q, R)
>>> S([Integer(2),Integer(1)]) * A([[Integer(1),-Integer(1)],[Integer(2),-
→Integer(3)],[Integer(3),-Integer(2)]])
P{{-3, 1}, {-2, 3}, {-1, 2}}
>>> A(B([[-Integer(1),-Integer(2)],[Integer(2),Integer(1)]]))
P{{-3, 3}, {-2, -1}, {1, 2}}
```

class Element

Bases: *Element*

dual()

Return the dual of `self`.

The dual of an element in the partition algebra is formed by taking the dual of each diagram in the support.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = P.an_element(); elt
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
sage: elt.dual()
3*P{{-2, -1, 1}, {2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, -1, 2}, {1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> elt = P.an_element(); elt
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
>>> elt.dual()
3*P{{-2, -1, 1}, {2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, -1, 2}, {1}}
```

to_orbit_basis()

Return `self` in the orbit basis of the associated partition algebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(2, x, R)
sage: pp = P.an_element();
sage: pp.to_orbit_basis()
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: pp = (3*P([[-2], [-1, 1, 2]]) + 2*P([[-2, -1, 1, 2]])
....:       + 2*P([[-2, 1, 2], [-1]])); pp
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
sage: pp.to_orbit_basis()
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> pp = P.an_element();
>>> pp.to_orbit_basis()
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> pp = (Integer(3)*P([[-Integer(2)], [-Integer(1), Integer(1),␣
↪Integer(2)]]) + Integer(2)*P([[-Integer(2), -Integer(1), Integer(1),␣
↪Integer(2)]])
...       + Integer(2)*P([[-Integer(2), Integer(1), Integer(2)], [-
↪Integer(1)]])); pp
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
>>> pp.to_orbit_basis()
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
```

L(i)

Return the `i`-th Jucys-Murphy element L_i from [Eny2012].

INPUT:

- `i` – half integer between $1/2$ and k

ALGORITHM:

We use the recursive definition for L_{2i} given in [Cre2020]. See also [Eny2012] and [Eny2013].

Note

$L_{1/2}$ and L_1 differs from [HR2005].

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.jucys_murphy_element(1/2)
0
sage: P3.jucys_murphy_element(1)
P{{-3, 3}, {-2, 2}, {-1}, {1}}
sage: P3.jucys_murphy_element(2)
P{{-3, 3}, {-2}, {-1, 1}, {2}} - P{{-3, 3}, {-2}, {-1, 1, 2}}
 + P{{-3, 3}, {-2, -1}, {1, 2}} - P{{-3, 3}, {-2, -1, 1}, {2}}
```

(continues on next page)

(continued from previous page)

```
 + P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.jucys_murphy_element(3/2)
n*P{{-3, 3}, {-2, -1, 1, 2}} - P{{-3, 3}, {-2, -1, 2}, {1}}
 - P{{-3, 3}, {-2, 1, 2}, {-1}} + P{{-3, 3}, {-2, 2}, {-1, 1}}
sage: P3.L(3/2) * P3.L(2) == P3.L(2) * P3.L(3/2)
True
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.jucys_murphy_element(Integer(1)/Integer(2))
0
>>> P3.jucys_murphy_element(Integer(1))
P{{-3, 3}, {-2, 2}, {-1}, {1}}
>>> P3.jucys_murphy_element(Integer(2))
P{{-3, 3}, {-2}, {-1, 1}, {2}} - P{{-3, 3}, {-2}, {-1, 1, 2}}
 + P{{-3, 3}, {-2, -1}, {1, 2}} - P{{-3, 3}, {-2, -1, 1}, {2}}
 + P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.jucys_murphy_element(Integer(3)/Integer(2))
n*P{{-3, 3}, {-2, -1, 1, 2}} - P{{-3, 3}, {-2, -1, 2}, {1}}
 - P{{-3, 3}, {-2, 1, 2}, {-1}} + P{{-3, 3}, {-2, 2}, {-1, 1}}
>>> P3.L(Integer(3)/Integer(2)) * P3.L(Integer(2)) == P3.L(Integer(2)) * P3.
↪L(Integer(3)/Integer(2))
True
```

We test the relations in Lemma 2.2.3(2) in [Cre2020] (v1):

```
sage: k = 4
sage: R.<n> = QQ[]
sage: P = PartitionAlgebra(k, n)
sage: L = [P.L(i/2) for i in range(1,2*k+1)]
sage: all(x.dual() == x for x in L)
True
sage: all(x * y == y * x for x, y in Subsets(L, 2))  # long time
True
sage: Lsum = sum(L)
sage: gens = [P.s(i) for i in range(1,k)]
sage: gens += [P.e(i/2) for i in range(1,2*k)]
sage: all(x * Lsum == Lsum * x for x in gens)
True
```

```
>>> from sage.all import *
>>> k = Integer(4)
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, n)
>>> L = [P.L(i/Integer(2)) for i in range(Integer(1),Integer(2)*k+Integer(1))]
>>> all(x.dual() == x for x in L)
True
>>> all(x * y == y * x for x, y in Subsets(L, Integer(2)))  # long time
True
>>> Lsum = sum(L)
>>> gens = [P.s(i) for i in range(Integer(1),k)]
```

(continues on next page)

(continued from previous page)

```
>>> gens += [P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k)]
>>> all(x * Lsum == Lsum * x for x in gens)
True
```

Also the relations in Lemma 2.2.3(3) in [Cre2020] (v1):

```
sage: all(P.e((2*i+1)/2) * P.sigma(2*i/2) * P.e((2*i+1)/2)
....:     == (n - P.L((2*i-1)/2)) * P.e((2*i+1)/2) for i in range(1,k))
True
sage: all(P.e(i/2) * (P.L(i/2) + P.L((i+1)/2))
....:     == (P.L(i/2) + P.L((i+1)/2)) * P.e(i/2)
....:     == n * P.e(i/2) for i in range(1,2*k))
True
sage: all(P.sigma(2*i/2) * P.e((2*i-1)/2) * P.e(2*i/2)
....:     == P.L(2*i/2) * P.e(2*i/2) for i in range(1,k))
True
sage: all(P.e(2*i/2) * P.e((2*i-1)/2) * P.sigma(2*i/2)
....:     == P.e(2*i/2) * P.L(2*i/2) for i in range(1,k))
True
sage: all(P.sigma((2*i+1)/2) * P.e((2*i+1)/2) * P.e(2*i/2)
....:     == P.L(2*i/2) * P.e(2*i/2) for i in range(1,k))
True
sage: all(P.e(2*i/2) * P.e((2*i+1)/2) * P.sigma((2*i+1)/2)
....:     == P.e(2*i/2) * P.L(2*i/2) for i in range(1,k))
True
```

```
>>> from sage.all import *
>>> all(P.e((Integer(2)*i+Integer(1))/Integer(2)) * P.sigma(Integer(2)*i/
↪Integer(2)) * P.e((Integer(2)*i+Integer(1))/Integer(2))
...     == (n - P.L((Integer(2)*i-Integer(1))/Integer(2))) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) for i in range(Integer(1),k))
True
>>> all(P.e(i/Integer(2)) * (P.L(i/Integer(2)) + P.L((i+Integer(1))/
↪Integer(2)))
...     == (P.L(i/Integer(2)) + P.L((i+Integer(1))/Integer(2))) * P.e(i/
↪Integer(2))
...     == n * P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.e(Integer(2)*i/Integer(2))
...     == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),k))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.sigma(Integer(2)*i/Integer(2))
...     == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),k))
True
>>> all(P.sigma((Integer(2)*i+Integer(1))/Integer(2)) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) * P.e(Integer(2)*i/Integer(2))
...     == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),k))
```

(continues on next page)

(continued from previous page)

```
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i+Integer(1))/
↪Integer(2)) * P.sigma((Integer(2)*i+Integer(1))/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),k))
True
```

The same tests for a half integer partition algebra:

```
sage: k = 7/2
sage: R.<n> = QQ[]
sage: P = PartitionAlgebra(k, n)
sage: L = [P.L(i/2) for i in range(1,2*k+1)]
sage: all(x.dual() == x for x in L)
True
sage: all(x * y == y * x for x, y in Subsets(L, 2))  # long time
True
sage: Lsum = sum(L)
sage: gens = [P.s(i) for i in range(1,k-1/2)]
sage: gens += [P.e(i/2) for i in range(1,2*k)]
sage: all(x * Lsum == Lsum * x for x in gens)
True
sage: all(P.e((2*i+1)/2) * P.sigma(2*i/2) * P.e((2*i+1)/2)
....:     == (n - P.L((2*i-1)/2)) * P.e((2*i+1)/2) for i in range(1,floor(k)))
True
sage: all(P.e(i/2) * (P.L(i/2) + P.L((i+1)/2))
....:     == (P.L(i/2) + P.L((i+1)/2)) * P.e(i/2)
....:     == n * P.e(i/2) for i in range(1,2*k))
True
sage: all(P.sigma(2*i/2) * P.e((2*i-1)/2) * P.e(2*i/2)
....:     == P.L(2*i/2) * P.e(2*i/2) for i in range(1,ceil(k)))
True
sage: all(P.e(2*i/2) * P.e((2*i-1)/2) * P.sigma(2*i/2)
....:     == P.e(2*i/2) * P.L(2*i/2) for i in range(1,ceil(k)))
True
sage: all(P.sigma((2*i+1)/2) * P.e((2*i+1)/2) * P.e(2*i/2)
....:     == P.L(2*i/2) * P.e(2*i/2) for i in range(1,floor(k)))
True
sage: all(P.e(2*i/2) * P.e((2*i+1)/2) * P.sigma((2*i+1)/2)
....:     == P.e(2*i/2) * P.L(2*i/2) for i in range(1,floor(k)))
True
```

```
>>> from sage.all import *
>>> k = Integer(7)/Integer(2)
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, n)
>>> L = [P.L(i/Integer(2)) for i in range(Integer(1),Integer(2)*k+Integer(1))]
>>> all(x.dual() == x for x in L)
True
>>> all(x * y == y * x for x, y in Subsets(L, Integer(2)))  # long time
True
>>> Lsum = sum(L)
```

(continues on next page)

(continued from previous page)

```
>>> gens = [P.s(i) for i in range(Integer(1),k-Integer(1)/Integer(2))]
>>> gens += [P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k)]
>>> all(x * Lsum == Lsum * x for x in gens)
True
>>> all(P.e((Integer(2)*i+Integer(1))/Integer(2)) * P.sigma(Integer(2)*i/
↪Integer(2)) * P.e((Integer(2)*i+Integer(1))/Integer(2))
...     == (n - P.L((Integer(2)*i-Integer(1))/Integer(2))) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) for i in range(Integer(1),floor(k)))
True
>>> all(P.e(i/Integer(2)) * (P.L(i/Integer(2)) + P.L((i+Integer(1))/
↪Integer(2)))
...     == (P.L(i/Integer(2)) + P.L((i+Integer(1))/Integer(2))) * P.e(i/
↪Integer(2))
...     == n * P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.e(Integer(2)*i/Integer(2))
...     == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),ceil(k)))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.sigma(Integer(2)*i/Integer(2))
...     == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),ceil(k)))
True
>>> all(P.sigma((Integer(2)*i+Integer(1))/Integer(2)) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) * P.e(Integer(2)*i/Integer(2))
...     == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),floor(k)))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i+Integer(1))/
↪Integer(2)) * P.sigma((Integer(2)*i+Integer(1))/Integer(2))
...     == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),floor(k)))
True
```

a(i)

Return the element a_i in `self`.

The element a_i is the cap and cup at $(i, i+1)$, so it contains the blocks $\{i, i+1\}$, $\{-i, -i-1\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- `i` – integer between 1 and $k-1$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.a(1)
P{{-3, 3}, {-2, -1}, {1, 2}}
sage: P3.a(2)
P{{-3, -2}, {-1, 1}, {2, 3}}
```

(continues on next page)

(continued from previous page)

```
sage: P3 = PartitionAlgebra(5/2, n)
sage: P3.a(1)
P{{-3, 3}, {-2, -1}, {1, 2}}
sage: P3.a(2)
Traceback (most recent call last):
...
ValueError: i must be an integer between 1 and 1
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.a(Integer(1))
P{{-3, 3}, {-2, -1}, {1, 2}}
>>> P3.a(Integer(2))
P{{-3, -2}, {-1, 1}, {2, 3}}

>>> P3 = PartitionAlgebra(Integer(5)/Integer(2), n)
>>> P3.a(Integer(1))
P{{-3, 3}, {-2, -1}, {1, 2}}
>>> P3.a(Integer(2))
Traceback (most recent call last):
...
ValueError: i must be an integer between 1 and 1
```

e(*i*)

Return the element e_i in `self`.

If $i = (2r+1)/2$, then e_i contains the blocks $\{r+1\}$ and $\{-r-1\}$. If $i \in \mathbf{Z}$, then e_i contains the block $\{-i, -i-1, i, i+1\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- `i` – half integer between $1/2$ and $k - 1/2$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.e(1)
P{{-3, 3}, {-2, -1, 1, 2}}
sage: P3.e(2)
P{{-3, -2, 2, 3}, {-1, 1}}
sage: P3.e(1/2)
P{{-3, 3}, {-2, 2}, {-1}, {1}}
sage: P3.e(5/2)
P{{-3}, {-2, 2}, {-1, 1}, {3}}
sage: P3.e(0)
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2
sage: P3.e(3)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: i must be an (half) integer between 1/2 and 5/2

sage: P2h = PartitionAlgebra(5/2,n)
sage: [P2h.e(k/2) for k in range(1,5)]
[P{{-3, 3}, {-2, 2}, {-1}, {1}},
 P{{-3, 3}, {-2, -1, 1, 2}},
 P{{-3, 3}, {-2}, {-1, 1}, {2}},
 P{{-3, -2, 2, 3}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.e(Integer(1))
P{{-3, 3}, {-2, -1, 1, 2}}
>>> P3.e(Integer(2))
P{{-3, -2, 2, 3}, {-1, 1}}
>>> P3.e(Integer(1)/Integer(2))
P{{-3, 3}, {-2, 2}, {-1}, {1}}
>>> P3.e(Integer(5)/Integer(2))
P{{-3}, {-2, 2}, {-1, 1}, {3}}
>>> P3.e(Integer(0))
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2
>>> P3.e(Integer(3))
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2

>>> P2h = PartitionAlgebra(Integer(5)/Integer(2),n)
>>> [P2h.e(k/Integer(2)) for k in range(Integer(1),Integer(5))]
[P{{-3, 3}, {-2, 2}, {-1}, {1}},
 P{{-3, 3}, {-2, -1, 1, 2}},
 P{{-3, 3}, {-2}, {-1, 1}, {2}},
 P{{-3, -2, 2, 3}, {-1, 1}}]
```

generator_a(i)

Return the element a_i in `self`.

The element a_i is the cap and cup at $(i, i+1)$, so it contains the blocks $\{i, i+1\}$, $\{-i, -i-1\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- `i` – integer between 1 and $k-1$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.a(1)
P{{-3, 3}, {-2, -1}, {1, 2}}
sage: P3.a(2)
P{{-3, -2}, {-1, 1}, {2, 3}}
```

(continues on next page)

(continued from previous page)

```
sage: P3 = PartitionAlgebra(5/2, n)
sage: P3.a(1)
P{{-3, 3}, {-2, -1}, {1, 2}}
sage: P3.a(2)
Traceback (most recent call last):
...
ValueError: i must be an integer between 1 and 1
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.a(Integer(1))
P{{-3, 3}, {-2, -1}, {1, 2}}
>>> P3.a(Integer(2))
P{{-3, -2}, {-1, 1}, {2, 3}}

>>> P3 = PartitionAlgebra(Integer(5)/Integer(2), n)
>>> P3.a(Integer(1))
P{{-3, 3}, {-2, -1}, {1, 2}}
>>> P3.a(Integer(2))
Traceback (most recent call last):
...
ValueError: i must be an integer between 1 and 1
```

generator_e(*i*)

Return the element e_i in `self`.

If $i = (2r+1)/2$, then e_i contains the blocks $\{r+1\}$ and $\{-r-1\}$. If $i \in \mathbf{Z}$, then e_i contains the block $\{-i, -i-1, i, i+1\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- `i` – half integer between $1/2$ and $k-1/2$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.e(1)
P{{-3, 3}, {-2, -1, 1, 2}}
sage: P3.e(2)
P{{-3, -2, 2, 3}, {-1, 1}}
sage: P3.e(1/2)
P{{-3, 3}, {-2, 2}, {-1}, {1}}
sage: P3.e(5/2)
P{{-3}, {-2, 2}, {-1, 1}, {3}}
sage: P3.e(0)
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2
sage: P3.e(3)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: i must be an (half) integer between 1/2 and 5/2

sage: P2h = PartitionAlgebra(5/2,n)
sage: [P2h.e(k/2) for k in range(1,5)]
[P{{-3, 3}, {-2, 2}, {-1}, {1}},
 P{{-3, 3}, {-2, -1, 1, 2}},
 P{{-3, 3}, {-2}, {-1, 1}, {2}},
 P{{-3, -2, 2, 3}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.e(Integer(1))
P{{-3, 3}, {-2, -1, 1, 2}}
>>> P3.e(Integer(2))
P{{-3, -2, 2, 3}, {-1, 1}}
>>> P3.e(Integer(1)/Integer(2))
P{{-3, 3}, {-2, 2}, {-1}, {1}}
>>> P3.e(Integer(5)/Integer(2))
P{{-3}, {-2, 2}, {-1, 1}, {3}}
>>> P3.e(Integer(0))
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2
>>> P3.e(Integer(3))
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2

>>> P2h = PartitionAlgebra(Integer(5)/Integer(2),n)
>>> [P2h.e(k/Integer(2)) for k in range(Integer(1),Integer(5))]
[P{{-3, 3}, {-2, 2}, {-1}, {1}},
 P{{-3, 3}, {-2, -1, 1, 2}},
 P{{-3, 3}, {-2}, {-1, 1}, {2}},
 P{{-3, -2, 2, 3}, {-1, 1}}]
```

generator_s(i)

Return the i-th simple transposition s_i in self.

Borrowing the notation from the symmetric group, the i-th simple transposition s_i has blocks of the form $\{-i, i+1\}$, $\{-i-1, i\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- i – integer between 1 and $k-1$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.s(1)
P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.s(2)
P{{-3, 2}, {-2, 3}, {-1, 1}}
```

(continues on next page)

(continued from previous page)

```
sage: R.<n> = ZZ[]
sage: P2h = PartitionAlgebra(5/2,n)
sage: P2h.s(1)
P{{-3, 3}, {-2, 1}, {-1, 2}}
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.s(Integer(1))
P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.s(Integer(2))
P{{-3, 2}, {-2, 3}, {-1, 1}}

>>> R = ZZ['n']; (n,) = R._first_ngens(1)
>>> P2h = PartitionAlgebra(Integer(5)/Integer(2),n)
>>> P2h.s(Integer(1))
P{{-3, 3}, {-2, 1}, {-1, 2}}
```

jucys_murphy_element (i)

Return the `i`-th Jucys-Murphy element L_i from [Eny2012].

INPUT:

- `i` – half integer between $1/2$ and k

ALGORITHM:

We use the recursive definition for L_{2i} given in [Cre2020]. See also [Eny2012] and [Eny2013].

> **Note**
>
> $L_{1/2}$ and L_1 differs from [HR2005].

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.jucys_murphy_element(1/2)
0
sage: P3.jucys_murphy_element(1)
P{{-3, 3}, {-2, 2}, {-1}, {1}}
sage: P3.jucys_murphy_element(2)
P{{-3, 3}, {-2}, {-1, 1}, {2}} - P{{-3, 3}, {-2}, {-1, 1, 2}}
 + P{{-3, 3}, {-2, -1}, {1, 2}} - P{{-3, 3}, {-2, -1, 1}, {2}}
 + P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.jucys_murphy_element(3/2)
n*P{{-3, 3}, {-2, -1, 1, 2}} - P{{-3, 3}, {-2, -1, 2}, {1}}
 - P{{-3, 3}, {-2, 1, 2}, {-1}} + P{{-3, 3}, {-2, 2}, {-1, 1}}
sage: P3.L(3/2) * P3.L(2) == P3.L(2) * P3.L(3/2)
True
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.jucys_murphy_element(Integer(1)/Integer(2))
0
>>> P3.jucys_murphy_element(Integer(1))
P{{-3, 3}, {-2, 2}, {-1}, {1}}
>>> P3.jucys_murphy_element(Integer(2))
P{{-3, 3}, {-2}, {-1, 1}, {2}} - P{{-3, 3}, {-2}, {-1, 1, 2}}
 + P{{-3, 3}, {-2, -1}, {1, 2}} - P{{-3, 3}, {-2, -1, 1}, {2}}
 + P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.jucys_murphy_element(Integer(3)/Integer(2))
n*P{{-3, 3}, {-2, -1, 1, 2}} - P{{-3, 3}, {-2, -1, 2}, {1}}
 - P{{-3, 3}, {-2, 1, 2}, {-1}} + P{{-3, 3}, {-2, 2}, {-1, 1}}
>>> P3.L(Integer(3)/Integer(2)) * P3.L(Integer(2)) == P3.L(Integer(2)) * P3.
↪L(Integer(3)/Integer(2))
True
```

We test the relations in Lemma 2.2.3(2) in [Cre2020] (v1):

```
sage: k = 4
sage: R.<n> = QQ[]
sage: P = PartitionAlgebra(k, n)
sage: L = [P.L(i/2) for i in range(1,2*k+1)]
sage: all(x.dual() == x for x in L)
True
sage: all(x * y == y * x for x, y in Subsets(L, 2))  # long time
True
sage: Lsum = sum(L)
sage: gens = [P.s(i) for i in range(1,k)]
sage: gens += [P.e(i/2) for i in range(1,2*k)]
sage: all(x * Lsum == Lsum * x for x in gens)
True
```

```
>>> from sage.all import *
>>> k = Integer(4)
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, n)
>>> L = [P.L(i/Integer(2)) for i in range(Integer(1),Integer(2)*k+Integer(1))]
>>> all(x.dual() == x for x in L)
True
>>> all(x * y == y * x for x, y in Subsets(L, Integer(2)))  # long time
True
>>> Lsum = sum(L)
>>> gens = [P.s(i) for i in range(Integer(1),k)]
>>> gens += [P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k)]
>>> all(x * Lsum == Lsum * x for x in gens)
True
```

Also the relations in Lemma 2.2.3(3) in [Cre2020] (v1):

```
sage: all(P.e((2*i+1)/2) * P.sigma(2*i/2) * P.e((2*i+1)/2)
....:     == (n - P.L((2*i-1)/2)) * P.e((2*i+1)/2) for i in range(1,k))
```

(continues on next page)

(continued from previous page)

```
True
sage: all(P.e(i/2) * (P.L(i/2) + P.L((i+1)/2))
....:     == (P.L(i/2) + P.L((i+1)/2)) * P.e(i/2)
....:     == n * P.e(i/2) for i in range(1,2*k))
True
sage: all(P.sigma(2*i/2) * P.e((2*i-1)/2) * P.e(2*i/2)
....:     == P.L(2*i/2) * P.e(2*i/2) for i in range(1,k))
True
sage: all(P.e(2*i/2) * P.e((2*i-1)/2) * P.sigma(2*i/2)
....:     == P.e(2*i/2) * P.L(2*i/2) for i in range(1,k))
True
sage: all(P.sigma((2*i+1)/2) * P.e((2*i+1)/2) * P.e(2*i/2)
....:     == P.L(2*i/2) * P.e(2*i/2) for i in range(1,k))
True
sage: all(P.e(2*i/2) * P.e((2*i+1)/2) * P.sigma((2*i+1)/2)
....:     == P.e(2*i/2) * P.L(2*i/2) for i in range(1,k))
True
```

```
>>> from sage.all import *
>>> all(P.e((Integer(2)*i+Integer(1))/Integer(2)) * P.sigma(Integer(2)*i/
↪Integer(2)) * P.e((Integer(2)*i+Integer(1))/Integer(2))
...     == (n - P.L((Integer(2)*i-Integer(1))/Integer(2))) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) for i in range(Integer(1),k))
True
>>> all(P.e(i/Integer(2)) * (P.L(i/Integer(2)) + P.L((i+Integer(1))/
↪Integer(2)))
...     == (P.L(i/Integer(2)) + P.L((i+Integer(1))/Integer(2))) * P.e(i/
↪Integer(2))
...     == n * P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.e(Integer(2)*i/Integer(2))
...     == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),k))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.sigma(Integer(2)*i/Integer(2))
...     == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),k))
True
>>> all(P.sigma((Integer(2)*i+Integer(1))/Integer(2)) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) * P.e(Integer(2)*i/Integer(2))
...     == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),k))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i+Integer(1))/
↪Integer(2)) * P.sigma((Integer(2)*i+Integer(1))/Integer(2))
...     == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),k))
True
```

The same tests for a half integer partition algebra:

```
sage: k = 7/2
sage: R.<n> = QQ[]
sage: P = PartitionAlgebra(k, n)
sage: L = [P.L(i/2) for i in range(1,2*k+1)]
sage: all(x.dual() == x for x in L)
True
sage: all(x * y == y * x for x, y in Subsets(L, 2))  # long time
True
sage: Lsum = sum(L)
sage: gens = [P.s(i) for i in range(1,k-1/2)]
sage: gens += [P.e(i/2) for i in range(1,2*k)]
sage: all(x * Lsum == Lsum * x for x in gens)
True
sage: all(P.e((2*i+1)/2) * P.sigma(2*i/2) * P.e((2*i+1)/2)
....:     == (n - P.L((2*i-1)/2)) * P.e((2*i+1)/2) for i in range(1,floor(k)))
True
sage: all(P.e(i/2) * (P.L(i/2) + P.L((i+1)/2))
....:     == (P.L(i/2) + P.L((i+1)/2)) * P.e(i/2)
....:     == n * P.e(i/2) for i in range(1,2*k))
True
sage: all(P.sigma(2*i/2) * P.e((2*i-1)/2) * P.e(2*i/2)
....:     == P.L(2*i/2) * P.e(2*i/2) for i in range(1,ceil(k)))
True
sage: all(P.e(2*i/2) * P.e((2*i-1)/2) * P.sigma(2*i/2)
....:     == P.e(2*i/2) * P.L(2*i/2) for i in range(1,ceil(k)))
True
sage: all(P.sigma((2*i+1)/2) * P.e((2*i+1)/2) * P.e(2*i/2)
....:     == P.L(2*i/2) * P.e(2*i/2) for i in range(1,floor(k)))
True
sage: all(P.e(2*i/2) * P.e((2*i+1)/2) * P.sigma((2*i+1)/2)
....:     == P.e(2*i/2) * P.L(2*i/2) for i in range(1,floor(k)))
True
```

```
>>> from sage.all import *
>>> k = Integer(7)/Integer(2)
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, n)
>>> L = [P.L(i/Integer(2)) for i in range(Integer(1),Integer(2)*k+Integer(1))]
>>> all(x.dual() == x for x in L)
True
>>> all(x * y == y * x for x, y in Subsets(L, Integer(2)))  # long time
True
>>> Lsum = sum(L)
>>> gens = [P.s(i) for i in range(Integer(1),k-Integer(1)/Integer(2))]
>>> gens += [P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k)]
>>> all(x * Lsum == Lsum * x for x in gens)
True
>>> all(P.e((Integer(2)*i+Integer(1))/Integer(2)) * P.sigma(Integer(2)*i/
↪Integer(2)) * P.e((Integer(2)*i+Integer(1))/Integer(2))
...     == (n - P.L((Integer(2)*i-Integer(1))/Integer(2))) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) for i in range(Integer(1),floor(k)))
True
```

(continues on next page)

(continued from previous page)

```
>>> all(P.e(i/Integer(2)) * (P.L(i/Integer(2)) + P.L((i+Integer(1))/
↪Integer(2)))
...     == (P.L(i/Integer(2)) + P.L((i+Integer(1))/Integer(2))) * P.e(i/
↪Integer(2))
...     == n * P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.e(Integer(2)*i/Integer(2))
...     == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),ceil(k)))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.sigma(Integer(2)*i/Integer(2))
...     == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),ceil(k)))
True
>>> all(P.sigma((Integer(2)*i+Integer(1))/Integer(2)) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) * P.e(Integer(2)*i/Integer(2))
...     == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),floor(k)))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i+Integer(1))/
↪Integer(2)) * P.sigma((Integer(2)*i+Integer(1))/Integer(2))
...     == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i␣
↪in range(Integer(1),floor(k)))
True
```

orbit_basis()

Return the orbit basis of `self`.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P2 = PartitionAlgebra(2, x, R)
sage: O2 = P2.orbit_basis(); O2
Orbit basis of Partition Algebra of rank 2 with parameter x over
 Univariate Polynomial Ring in x over Rational Field
sage: pp = 7 * P2[{-1}, {-2, 1, 2}] - 2 * P2[{-2}, {-1, 1}, {2}]; pp
-2*P{{-2}, {-1, 1}, {2}} + 7*P{{-2, 1, 2}, {-1}}
sage: op = pp.to_orbit_basis(); op
-2*O{{-2}, {-1, 1}, {2}} - 2*O{{-2}, {-1, 1, 2}}
 - 2*O{{-2, -1, 1}, {2}} + 5*O{{-2, -1, 1, 2}}
 + 7*O{{-2, 1, 2}, {-1}} - 2*O{{-2, 2}, {-1, 1}}
sage: op == O2(op)
True
sage: pp * op.leading_term()
4*P{{-2}, {-1, 1}, {2}} - 4*P{{-2, -1, 1}, {2}}
 + 14*P{{-2, -1, 1, 2}} - 14*P{{-2, 1, 2}, {-1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P2 = PartitionAlgebra(Integer(2), x, R)
```

(continues on next page)

(continued from previous page)

```
>>> O2 = P2.orbit_basis(); O2
Orbit basis of Partition Algebra of rank 2 with parameter x over
 Univariate Polynomial Ring in x over Rational Field
>>> pp = Integer(7) * P2[{-Integer(1)}, {-Integer(2), Integer(1), Integer(2)}
↪] - Integer(2) * P2[{-Integer(2)}, {-Integer(1), Integer(1)}, {Integer(2)}];
↪ pp
-2*P{{-2}, {-1, 1}, {2}} + 7*P{{-2, 1, 2}, {-1}}
>>> op = pp.to_orbit_basis(); op
-2*O{{-2}, {-1, 1}, {2}} - 2*O{{-2}, {-1, 1, 2}}
 - 2*O{{-2, -1, 1}, {2}} + 5*O{{-2, -1, 1, 2}}
 + 7*O{{-2, 1, 2}, {-1}} - 2*O{{-2, 2}, {-1, 1}}
>>> op == O2(op)
True
>>> pp * op.leading_term()
4*P{{-2}, {-1, 1}, {2}} - 4*P{{-2, -1, 1}, {2}}
 + 14*P{{-2, -1, 1, 2}} - 14*P{{-2, 1, 2}, {-1}}
```

potts_representation(*y=None*)

Return the *PottsRepresentation* with magnetic field direction `y` of `self`.

Note

The deformation parameter d of `self` must be a positive integer.

INPUT:

- `y` – (optional) an integer between 1 and d; ignored if the order of `self` is an integer, otherwise the default is 1

EXAMPLES:

```
sage: PA = algebras.Partition(5/2, QQ(4))
sage: PR = PA.potts_representation()

sage: PA = algebras.Partition(5/2, 3/2)
sage: PA.potts_representation()
Traceback (most recent call last):
...
ValueError: the partition algebra deformation parameter must
 be a positive integer
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(5)/Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation()

>>> PA = algebras.Partition(Integer(5)/Integer(2), Integer(3)/Integer(2))
>>> PA.potts_representation()
Traceback (most recent call last):
...
ValueError: the partition algebra deformation parameter must
 be a positive integer
```

s(i)

Return the i-th simple transposition s_i in self.

Borrowing the notation from the symmetric group, the i-th simple transposition s_i has blocks of the form $\{-i, i+1\}$, $\{-i-1, i\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- i – integer between 1 and $k-1$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.s(1)
P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.s(2)
P{{-3, 2}, {-2, 3}, {-1, 1}}

sage: R.<n> = ZZ[]
sage: P2h = PartitionAlgebra(5/2,n)
sage: P2h.s(1)
P{{-3, 3}, {-2, 1}, {-1, 2}}
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.s(Integer(1))
P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.s(Integer(2))
P{{-3, 2}, {-2, 3}, {-1, 1}}

>>> R = ZZ['n']; (n,) = R._first_ngens(1)
>>> P2h = PartitionAlgebra(Integer(5)/Integer(2),n)
>>> P2h.s(Integer(1))
P{{-3, 3}, {-2, 1}, {-1, 2}}
```

sigma(i)

Return the element σ_i from [Eny2012] of self.

INPUT:

- i – half integer between $1/2$ and $k-1/2$

Note

In [Cre2020] and [Eny2013], these are the elements σ_{2i}.

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.sigma(1)
P{{-3, 3}, {-2, 2}, {-1, 1}}
sage: P3.sigma(3/2)
```

(continues on next page)

(continued from previous page)

```
P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.sigma(2)
-P{{-3, -1, 1, 3}, {-2, 2}} + P{{-3, -1, 3}, {-2, 1, 2}}
 + P{{-3, 1, 3}, {-2, -1, 2}} - P{{-3, 3}, {-2, -1, 1, 2}}
 + P{{-3, 3}, {-2, 2}, {-1, 1}}
sage: P3.sigma(5/2)
-P{{-3, -1, 1, 2}, {-2, 3}} + P{{-3, -1, 2}, {-2, 1, 3}}
 + P{{-3, 1, 2}, {-2, -1, 3}} - P{{-3, 2}, {-2, -1, 1, 3}}
 + P{{-3, 2}, {-2, 3}, {-1, 1}}
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.sigma(Integer(1))
P{{-3, 3}, {-2, 2}, {-1, 1}}
>>> P3.sigma(Integer(3)/Integer(2))
P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.sigma(Integer(2))
-P{{-3, -1, 1, 3}, {-2, 2}} + P{{-3, -1, 3}, {-2, 1, 2}}
 + P{{-3, 1, 3}, {-2, -1, 2}} - P{{-3, 3}, {-2, -1, 1, 2}}
 + P{{-3, 3}, {-2, 2}, {-1, 1}}
>>> P3.sigma(Integer(5)/Integer(2))
-P{{-3, -1, 1, 2}, {-2, 3}} + P{{-3, -1, 2}, {-2, 1, 3}}
 + P{{-3, 1, 2}, {-2, -1, 3}} - P{{-3, 2}, {-2, -1, 1, 3}}
 + P{{-3, 2}, {-2, 3}, {-1, 1}}
```

We test the relations in Lemma 2.2.3(1) in [Cre2020] (v1):

```
sage: k = 4
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(k, x)
sage: all(P.sigma(i/2).dual() == P.sigma(i/2)
....:     for i in range(1,2*k))
True
sage: all(P.sigma(i)*P.sigma(i+1/2) == P.sigma(i+1/2)*P.sigma(i) == P.s(i)
....:     for i in range(1,floor(k)))
True
sage: all(P.sigma(i)*P.e(i) == P.e(i)*P.sigma(i) == P.e(i)
....:     for i in range(1,floor(k)))
True
sage: all(P.sigma(i+1/2)*P.e(i) == P.e(i)*P.sigma(i+1/2) == P.e(i)
....:     for i in range(1,floor(k)))
True

sage: k = 9/2
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(k, x)
sage: all(P.sigma(i/2).dual() == P.sigma(i/2)
....:     for i in range(1,2*k-1))
True
sage: all(P.sigma(i)*P.sigma(i+1/2) == P.sigma(i+1/2)*P.sigma(i) == P.s(i)
....:     for i in range(1,k-1/2))
```

(continues on next page)

(continued from previous page)

```
True
sage: all(P.sigma(i)*P.e(i) == P.e(i)*P.sigma(i) == P.e(i)
....:     for i in range(1,floor(k)))
True
sage: all(P.sigma(i+1/2)*P.e(i) == P.e(i)*P.sigma(i+1/2) == P.e(i)
....:     for i in range(1,floor(k)))
True
```

```
>>> from sage.all import *
>>> k = Integer(4)
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, x)
>>> all(P.sigma(i/Integer(2)).dual() == P.sigma(i/Integer(2))
...     for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(i)*P.sigma(i+Integer(1)/Integer(2)) == P.sigma(i+Integer(1)/
↪Integer(2))*P.sigma(i) == P.s(i)
...     for i in range(Integer(1),floor(k)))
True
>>> all(P.sigma(i)*P.e(i) == P.e(i)*P.sigma(i) == P.e(i)
...     for i in range(Integer(1),floor(k)))
True
>>> all(P.sigma(i+Integer(1)/Integer(2))*P.e(i) == P.e(i)*P.
↪sigma(i+Integer(1)/Integer(2)) == P.e(i)
...     for i in range(Integer(1),floor(k)))
True

>>> k = Integer(9)/Integer(2)
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, x)
>>> all(P.sigma(i/Integer(2)).dual() == P.sigma(i/Integer(2))
...     for i in range(Integer(1),Integer(2)*k-Integer(1)))
True
>>> all(P.sigma(i)*P.sigma(i+Integer(1)/Integer(2)) == P.sigma(i+Integer(1)/
↪Integer(2))*P.sigma(i) == P.s(i)
...     for i in range(Integer(1),k-Integer(1)/Integer(2)))
True
>>> all(P.sigma(i)*P.e(i) == P.e(i)*P.sigma(i) == P.e(i)
...     for i in range(Integer(1),floor(k)))
True
>>> all(P.sigma(i+Integer(1)/Integer(2))*P.e(i) == P.e(i)*P.
↪sigma(i+Integer(1)/Integer(2)) == P.e(i)
...     for i in range(Integer(1),floor(k)))
True
```

class sage.combinat.diagram_algebras.**PartitionDiagram**(*parent*, *d*, *check=True*)

Bases: *AbstractPartitionDiagram*

The element class for a partition diagram.

A partition diagram for an integer k is a partition of the set $\{1, \ldots, k, -1, \ldots, -k\}$

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PartitionDiagram,␣
↪PartitionDiagrams
sage: PartitionDiagrams(1)
Partition diagrams of order 1
sage: PartitionDiagrams(1).list()
[{{-1, 1}}, {{-1}, {1}}]
sage: PartitionDiagram([[1,-1]])
{{-1, 1}}
sage: PartitionDiagram(((1,-2),(2,-1))).parent()
Partition diagrams of order 2
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagram, PartitionDiagrams
>>> PartitionDiagrams(Integer(1))
Partition diagrams of order 1
>>> PartitionDiagrams(Integer(1)).list()
[{{-1, 1}}, {{-1}, {1}}]
>>> PartitionDiagram([[Integer(1),-Integer(1)]])
{{-1, 1}}
>>> PartitionDiagram(((Integer(1),-Integer(2)),(Integer(2),-Integer(1)))).parent()
Partition diagrams of order 2
```

class sage.combinat.diagram_algebras.**PartitionDiagrams**(*order*, *category=None*)

Bases: *AbstractPartitionDiagrams*

This class represents all partition diagrams of integer or integer $+1/2$ order.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(1); pd
Partition diagrams of order 1
sage: pd.list()
[{{-1, 1}}, {{-1}, {1}}]

sage: pd = da.PartitionDiagrams(3/2); pd
Partition diagrams of order 3/2
sage: pd.list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.PartitionDiagrams(Integer(1)); pd
Partition diagrams of order 1
>>> pd.list()
[{{-1, 1}}, {{-1}, {1}}]

>>> pd = da.PartitionDiagrams(Integer(3)/Integer(2)); pd
Partition diagrams of order 3/2
```

(continues on next page)

(continued from previous page)

```
>>> pd.list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]
```

Element

alias of *PartitionDiagram*

cardinality()

The cardinality of partition diagrams of half-integer order n is the $2n$-th Bell number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(3)
sage: pd.cardinality()
203

sage: pd = da.PartitionDiagrams(7/2)
sage: pd.cardinality()
877
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.PartitionDiagrams(Integer(3))
>>> pd.cardinality()
203

>>> pd = da.PartitionDiagrams(Integer(7)/Integer(2))
>>> pd.cardinality()
877
```

class sage.combinat.diagram_algebras.**PlanarAlgebra**(*k*, *q*, *base_ring*, *prefix*)

Bases: *SubPartitionAlgebra*, *UnitDiagramMixin*

A planar algebra.

The planar algebra of rank k is an algebra with basis indexed by the collection of all planar set partitions of $\{1, \ldots, k, -1, \ldots, -k\}$.

This algebra is thus a subalgebra of the partition algebra. For more information, see *PartitionAlgebra*.

INPUT:

- `k` – rank of the algebra
- `q` – the deformation parameter q
- `base_ring` – (default: `None`) a ring containing `q`; if `None` then just takes the parent of `q`
- `prefix` – (default: `'Pl'`) a label for the basis elements

EXAMPLES:

We define the planar algebra of rank 2 with parameter x over $\mathbf{Z}$:

```
sage: R.<x> = ZZ[]
sage: Pl = PlanarAlgebra(2, x, R); Pl
Planar Algebra of rank 2 with parameter x over Univariate Polynomial Ring in x␣
↪over Integer Ring
sage: Pl.basis().keys()
Planar diagrams of order 2
sage: Pl.basis().keys()([[-1, 1], [2, -2]])
{{-2, 2}, {-1, 1}}
sage: Pl.basis().list()
[Pl{{-2}, {-1}, {1, 2}},
 Pl{{-2}, {-1}, {1}, {2}},
 Pl{{-2, 1}, {-1}, {2}},
 Pl{{-2, 2}, {-1}, {1}},
 Pl{{-2, 1, 2}, {-1}},
 Pl{{-2, 2}, {-1, 1}},
 Pl{{-2}, {-1, 1}, {2}},
 Pl{{-2}, {-1, 2}, {1}},
 Pl{{-2}, {-1, 1, 2}},
 Pl{{-2, -1}, {1, 2}},
 Pl{{-2, -1}, {1}, {2}},
 Pl{{-2, -1, 1}, {2}},
 Pl{{-2, -1, 2}, {1}},
 Pl{{-2, -1, 1, 2}}]
sage: E = Pl([[1,2],[-1,-2]])
sage: E^2 == x*E
True
sage: E^5 == x^4*E
True
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> Pl = PlanarAlgebra(Integer(2), x, R); Pl
Planar Algebra of rank 2 with parameter x over Univariate Polynomial Ring in x␣
↪over Integer Ring
>>> Pl.basis().keys()
Planar diagrams of order 2
>>> Pl.basis().keys()([[-Integer(1), Integer(1)], [Integer(2), -Integer(2)]])
{{-2, 2}, {-1, 1}}
>>> Pl.basis().list()
[Pl{{-2}, {-1}, {1, 2}},
 Pl{{-2}, {-1}, {1}, {2}},
 Pl{{-2, 1}, {-1}, {2}},
 Pl{{-2, 2}, {-1}, {1}},
 Pl{{-2, 1, 2}, {-1}},
 Pl{{-2, 2}, {-1, 1}},
 Pl{{-2}, {-1, 1}, {2}},
 Pl{{-2}, {-1, 2}, {1}},
 Pl{{-2}, {-1, 1, 2}},
 Pl{{-2, -1}, {1, 2}},
 Pl{{-2, -1}, {1}, {2}},
 Pl{{-2, -1, 1}, {2}},
 Pl{{-2, -1, 2}, {1}},
```

(continues on next page)

(continued from previous page)

```
 Pl{{-2, -1, 1, 2}}]
>>> E = Pl([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
>>> E**Integer(2) == x*E
True
>>> E**Integer(5) == x**Integer(4)*E
True
```

class sage.combinat.diagram_algebras.**PlanarDiagram**(*parent*, *d*, *check=True*)

Bases: *AbstractPartitionDiagram*

The element class for a planar diagram.

A planar diagram for an integer k is a partition of the set $\{1, \ldots, k, -1, \ldots, -k\}$ so that the diagram is non-crossing.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PlanarDiagrams
sage: PlanarDiagrams(2)
Planar diagrams of order 2
sage: PlanarDiagrams(2).list()
[{{-2}, {-1}, {1, 2}},
 {{-2}, {-1}, {1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, 2}, {-1}, {1}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2, -1}, {1}, {2}},
 {{-2, -1, 1}, {2}},
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1, 2}}]
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PlanarDiagrams
>>> PlanarDiagrams(Integer(2))
Planar diagrams of order 2
>>> PlanarDiagrams(Integer(2)).list()
[{{-2}, {-1}, {1, 2}},
 {{-2}, {-1}, {1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, 2}, {-1}, {1}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2, -1}, {1}, {2}},
 {{-2, -1, 1}, {2}},
```

(continues on next page)

(continued from previous page)

```
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1, 2}}]
```

check()

Check the validity of the input for `self`.

class sage.combinat.diagram_algebras.**PlanarDiagrams**(*order*, *category=None*)

Bases: *AbstractPartitionDiagrams*

All planar diagrams of integer or integer $+1/2$ order.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pld = da.PlanarDiagrams(1); pld
Planar diagrams of order 1
sage: pld.list()
[{{-1, 1}}, {{-1}, {1}}]

sage: pld = da.PlanarDiagrams(3/2); pld
Planar diagrams of order 3/2
sage: pld.list()
[{{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1}, {1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1, 2}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pld = da.PlanarDiagrams(Integer(1)); pld
Planar diagrams of order 1
>>> pld.list()
[{{-1, 1}}, {{-1}, {1}}]

>>> pld = da.PlanarDiagrams(Integer(3)/Integer(2)); pld
Planar diagrams of order 3/2
>>> pld.list()
[{{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1}, {1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1, 2}}]
```

Element

alias of *PlanarDiagram*

cardinality()

Return the cardinality of `self`.

The number of all planar diagrams of order k is the $2k$-th Catalan number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pld = da.PlanarDiagrams(3)
sage: pld.cardinality()
132
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pld = da.PlanarDiagrams(Integer(3))
>>> pld.cardinality()
132
```

class sage.combinat.diagram_algebras.**PottsRepresentation**(*PA*, *y*)

Bases: *CombinatorialFreeModule*

The Potts representation of the partition algebra.

Let $P_n(d)$ be the *PartitionAlgebra* over R with the deformation parameter $d \in \mathbf{Z}_{>0}$ being a positive integer. Recall the multiplication convention of diagrams in $P_n(d)$ computing DD' by placing D above D'.

The *Potts representation* is the right $P_n(d)$-module on $M = V^{\otimes n}$, with $V = R^d$, with the action given as follows. We identify the natural basis vectors in M with words of length n in the alphabet $\{1, \ldots, d\}$ (which we call colors). For a basis vector w and diagram D, define $w \cdot D$ as the sum over all v such that every part in wDv (consider this as coloring the nodes of D) is given by the same color.

If n is a half integer, then there is an extra fixed color for the node $\lceil n \rceil$, which is called the *magnetic field direction* from the physics interpretation of this representation.

EXAMPLES:

In this example, we consider $R = \mathbf{Q}$ and use the Potts representation to construct the centralizer algebra of the left S_{d-1}-action on $V^{\otimes n}$ with $V = \mathbf{Q}^d$ being the permutation action.

```
sage: PA = algebras.Partition(5/2, QQ(2))
sage: PR = PA.potts_representation(2)
sage: mats = [PR.representation_matrix(x) for x in PA.basis()]
sage: MS = mats[0].parent()
sage: CM = MS.submodule(mats)
sage: CM.dimension()
16
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(5)/Integer(2), QQ(Integer(2)))
>>> PR = PA.potts_representation(Integer(2))
>>> mats = [PR.representation_matrix(x) for x in PA.basis()]
>>> MS = mats[Integer(0)].parent()
>>> CM = MS.submodule(mats)
>>> CM.dimension()
16
```

We check that this commutes with the S_{d-1}-action:

```
sage: all((g * v) * x == g * (v * x) for g in PR.symmetric_group()
....:     for v in PR.basis() for x in PA.basis())
True
```

```
>>> from sage.all import *
>>> all((g * v) * x == g * (v * x) for g in PR.symmetric_group()
...     for v in PR.basis() for x in PA.basis())
True
```

Next, we see that the centralizer of the S_d-action is smaller than the semisimple quotient of the partition algebra:

```
sage: PA.dimension()
52
sage: len(PA.radical_basis())
9
sage: SQ = PA.semisimple_quotient()
sage: SQ.dimension()
43
```

```
>>> from sage.all import *
>>> PA.dimension()
52
>>> len(PA.radical_basis())
9
>>> SQ = PA.semisimple_quotient()
>>> SQ.dimension()
43
```

Next, we get orthogonal idempotents that project onto the central orthogonal idempotents in the semisimple quotient and construct the corresponding Peirce summands $e_i P_n(d) e_i$:

```
sage: # long time
sage: orth_idems = PA.orthogonal_idempotents_central_mod_radical()
sage: algs = [PA.peirce_summand(idm, idm) for idm in orth_idems]
sage: [A.dimension() for A in algs]
[16, 2, 1, 25]
```

```
>>> from sage.all import *
>>> # long time
>>> orth_idems = PA.orthogonal_idempotents_central_mod_radical()
>>> algs = [PA.peirce_summand(idm, idm) for idm in orth_idems]
>>> [A.dimension() for A in algs]
[16, 2, 1, 25]
```

We saw that we obtain the entire endomorphism algebra since $d = 2$ and S_{d-1} is the trivial group. Hence, the 16 dimensional Peirce summand computed above is isomorphic to this endomorphism algebra (both are 4×4 matrix algebras over $\mathbf{Q}$). Hence, we have a natural quotient construction of the centralizer algebra from the partition algebra.

Next, we consider a case with a nontrivial S_d-action (now it is S_d since the partition algebra has integer rank). We perform the same computations as before:

```
sage: PA = algebras.Partition(2, QQ(2))
sage: PA.dimension()
15
sage: PA.semisimple_quotient().dimension()
10
```

(continues on next page)

(continued from previous page)

```
sage: orth_idems = PA.orthogonal_idempotents_central_mod_radical()
sage: algs = [PA.peirce_summand(idm, idm) for idm in orth_idems]
sage: [A.dimension() for A in algs]
[4, 2, 4, 1]

sage: PR = PA.potts_representation()
sage: mats = [PR.representation_matrix(x) for x in PA.basis()]
sage: MS = mats[0].parent()
sage: cat = Algebras(QQ).WithBasis().Subobjects()
sage: CM = MS.submodule(mats, category=cat)
sage: CM.dimension()
8
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(2), QQ(Integer(2)))
>>> PA.dimension()
15
>>> PA.semisimple_quotient().dimension()
10
>>> orth_idems = PA.orthogonal_idempotents_central_mod_radical()
>>> algs = [PA.peirce_summand(idm, idm) for idm in orth_idems]
>>> [A.dimension() for A in algs]
[4, 2, 4, 1]

>>> PR = PA.potts_representation()
>>> mats = [PR.representation_matrix(x) for x in PA.basis()]
>>> MS = mats[Integer(0)].parent()
>>> cat = Algebras(QQ).WithBasis().Subobjects()
>>> CM = MS.submodule(mats, category=cat)
>>> CM.dimension()
8
```

To do the remainder of the computation, we need to monkey patch a `product_on_basis` method:

```
sage: CM.product_on_basis
NotImplemented
sage: CM.product_on_basis = lambda x,y: CM.retract(CM.basis()[x].lift() * CM.
↪basis()[y].lift())
sage: CM.orthogonal_idempotents_central_mod_radical()
(1/2*B[0] + 1/2*B[3] + 1/2*B[5] + 1/2*B[6],
 1/2*B[0] - 1/2*B[3] + 1/2*B[5] - 1/2*B[6])
sage: CM.peirce_decomposition()
[[Free module generated by {0, 1, 2, 3} over Rational Field,
  Free module generated by {} over Rational Field],
 [Free module generated by {} over Rational Field,
  Free module generated by {0, 1, 2, 3} over Rational Field]]
```

```
>>> from sage.all import *
>>> CM.product_on_basis
NotImplemented
>>> CM.product_on_basis = lambda x,y: CM.retract(CM.basis()[x].lift() * CM.
↪basis()[y].lift())
```

(continues on next page)

(continued from previous page)

```
>>> CM.orthogonal_idempotents_central_mod_radical()
(1/2*B[0] + 1/2*B[3] + 1/2*B[5] + 1/2*B[6],
 1/2*B[0] - 1/2*B[3] + 1/2*B[5] - 1/2*B[6])
>>> CM.peirce_decomposition()
[[Free module generated by {0, 1, 2, 3} over Rational Field,
  Free module generated by {} over Rational Field],
 [Free module generated by {} over Rational Field,
  Free module generated by {0, 1, 2, 3} over Rational Field]]
```

Hence, we see that the centralizer algebra is isomorphic to a product of two 2×2 matrix algebras (over $\mathbf{Q}$), which are naturally a part of the partition algebra decomposition.

Lastly, we verify the commuting actions:

```
sage: all((g * v) * x == g * (v * x) for g in PR.symmetric_group()
....:     for v in PR.basis() for x in PA.basis())
True
```

```
>>> from sage.all import *
>>> all((g * v) * x == g * (v * x) for g in PR.symmetric_group()
...     for v in PR.basis() for x in PA.basis())
True
```

REFERENCES:

- [MR1998]

class Element

Bases: `IndexedFreeModuleElement`

magnetic_field_direction()

Return the magnetic field direction defining `self`.

EXAMPLES:

```
sage: PA = algebras.Partition(7/2, QQ(4))
sage: PR = PA.potts_representation(2)
sage: PR.magnetic_field_direction()
2
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(7)/Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation(Integer(2))
>>> PR.magnetic_field_direction()
2
```

number_of_colors()

Return the number of colors defining `self`.

EXAMPLES:

```
sage: PA = algebras.Partition(3, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.number_of_colors()
4
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(3), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.number_of_colors()
4
```

number_of_factors()

Return the number of factors defining `self`.

EXAMPLES:

```
sage: PA = algebras.Partition(7/2, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.number_of_factors()
3
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(7)/Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.number_of_factors()
3
```

partition_algebra()

Return the partition algebra that `self` is a representation of.

EXAMPLES:

```
sage: PA = algebras.Partition(3, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.partition_algebra() is PA
True
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(3), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.partition_algebra() is PA
True
```

representation_matrix(*elt*)

Return the representation matrix of `self` in `self`.

EXAMPLES:

```
sage: PA = algebras.Partition(7/2, QQ(2))
sage: PR = PA.potts_representation()
sage: PR.representation_matrix(PA.an_element())
[7 0 3 0 2 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 0 0 0 0]

sage: all(b.to_vector() * PR.representation_matrix(x)   # long time
....:     == (b * x).to_vector()
....:     for b in PR.basis() for x in PA.basis())
True

sage: PA = algebras.Partition(2, QQ(2))
sage: PR = PA.potts_representation()
sage: [PR.representation_matrix(x) for x in PA.basis()]
[
[1 0 0 0]  [1 0 1 0]  [1 1 0 0]  [1 0 0 1]  [1 1 1 1]  [1 0 0 0]
[0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [1 0 0 0]
[0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 1]
[0 0 0 1], [0 1 0 1], [0 0 1 1], [1 0 0 1], [1 1 1 1], [0 0 0 1],

[1 0 0 0]  [1 0 1 0]  [1 0 0 0]  [1 0 0 0]  [1 0 1 0]  [1 1 0 0]
[0 0 1 0]  [1 0 1 0]  [0 1 0 0]  [0 0 0 1]  [0 1 0 1]  [1 1 0 0]
[0 1 0 0]  [0 1 0 1]  [0 0 1 0]  [1 0 0 0]  [1 0 1 0]  [0 0 1 1]
[0 0 0 1], [0 1 0 1], [0 0 0 1], [0 0 0 1], [0 1 0 1], [0 0 1 1],

[1 1 0 0]  [1 0 0 1]  [1 1 1 1]
[0 0 1 1]  [1 0 0 1]  [1 1 1 1]
[1 1 0 0]  [1 0 0 1]  [1 1 1 1]
[0 0 1 1], [1 0 0 1], [1 1 1 1]
]

sage: PA = algebras.Partition(5/2, QQ(2))
sage: PR = PA.potts_representation()
sage: all(PR.representation_matrix(x) * PR.representation_matrix(y)   # long␣
↪time
....:     == PR.representation_matrix(x * y)
....:     for x in PA.basis() for y in PA.basis())
True

sage: PA = algebras.Partition(2, QQ(4))
sage: PR = PA.potts_representation()
sage: all(PR.representation_matrix(x) * PR.representation_matrix(y)
....:     == PR.representation_matrix(x * y)
....:     for x in PA.basis() for y in PA.basis())
True
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(7)/Integer(2), QQ(Integer(2)))
>>> PR = PA.potts_representation()
>>> PR.representation_matrix(PA.an_element())
[7 0 3 0 2 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]

>>> all(b.to_vector() * PR.representation_matrix(x)  # long time
...     == (b * x).to_vector()
...     for b in PR.basis() for x in PA.basis())
True

>>> PA = algebras.Partition(Integer(2), QQ(Integer(2)))
>>> PR = PA.potts_representation()
>>> [PR.representation_matrix(x) for x in PA.basis()]
[
[1 0 0 0]  [1 0 1 0]  [1 1 0 0]  [1 0 0 1]  [1 1 1 1]  [1 0 0 0]
[0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [1 0 0 0]
[0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 0]  [0 0 0 1]
[0 0 0 1], [0 1 0 1], [0 0 1 1], [1 0 0 1], [1 1 1 1], [0 0 0 1],
<BLANKLINE>
[1 0 0 0]  [1 0 1 0]  [1 0 0 0]  [1 0 0 0]  [1 0 1 0]  [1 1 0 0]
[0 0 1 0]  [1 0 1 0]  [0 1 0 0]  [0 0 0 1]  [0 1 0 1]  [1 1 0 0]
[0 1 0 0]  [0 1 0 1]  [0 0 1 0]  [1 0 0 0]  [1 0 1 0]  [0 0 1 1]
[0 0 0 1], [0 1 0 1], [0 0 0 1], [0 0 0 1], [0 1 0 1], [0 0 1 1],
<BLANKLINE>
[1 1 0 0]  [1 0 0 1]  [1 1 1 1]
[0 0 1 1]  [1 0 0 1]  [1 1 1 1]
[1 1 0 0]  [1 0 0 1]  [1 1 1 1]
[0 0 1 1], [1 0 0 1], [1 1 1 1]
]

>>> PA = algebras.Partition(Integer(5)/Integer(2), QQ(Integer(2)))
>>> PR = PA.potts_representation()
>>> all(PR.representation_matrix(x) * PR.representation_matrix(y)  # long time
...     == PR.representation_matrix(x * y)
...     for x in PA.basis() for y in PA.basis())
True

>>> PA = algebras.Partition(Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> all(PR.representation_matrix(x) * PR.representation_matrix(y)
...     == PR.representation_matrix(x * y)
...     for x in PA.basis() for y in PA.basis())
True
```

symmetric_group()

Return the symmetric group that naturally acts on `self`.

EXAMPLES:

```
sage: PA = algebras.Partition(3, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.symmetric_group()
Symmetric group of order 4! as a permutation group
```

(continues on next page)

(continued from previous page)

```
sage: PA = algebras.Partition(7/2, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.symmetric_group().domain()
{2, 3, 4}
sage: PR = PA.potts_representation(2)
sage: PR.symmetric_group().domain()
{1, 3, 4}
sage: PR = PA.potts_representation(4)
sage: PR.symmetric_group().domain()
{1, 2, 3}
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(3), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.symmetric_group()
Symmetric group of order 4! as a permutation group

>>> PA = algebras.Partition(Integer(7)/Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.symmetric_group().domain()
{2, 3, 4}
>>> PR = PA.potts_representation(Integer(2))
>>> PR.symmetric_group().domain()
{1, 3, 4}
>>> PR = PA.potts_representation(Integer(4))
>>> PR.symmetric_group().domain()
{1, 2, 3}
```

class sage.combinat.diagram_algebras.**PropagatingIdeal**(*k*, *q*, *base_ring*, *prefix*)

Bases: *SubPartitionAlgebra*

A propagating ideal.

The propagating ideal of rank k is a non-unital algebra with basis indexed by the collection of ideal set partitions of $\{1, \ldots, k, -1, \ldots, -k\}$. We say a set partition is *ideal* if its propagating number is less than k.

This algebra is a non-unital subalgebra and an ideal of the partition algebra. For more information, see *PartitionAlgebra*.

EXAMPLES:

We now define the propagating ideal of rank 2 with parameter x over $\mathbf{Z}$:

```
sage: R.<x> = QQ[]
sage: I = PropagatingIdeal(2, x, R); I
Propagating Ideal of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Rational Field
sage: I.basis().keys()
Ideal diagrams of order 2
sage: I.basis().list()
[I{{-2, -1, 1, 2}},
 I{{-2, 1, 2}, {-1}},
 I{{-2}, {-1, 1, 2}},
 I{{-2, -1}, {1, 2}},
```

(continues on next page)

(continued from previous page)

```
 I{{-2}, {-1}, {1, 2}},
 I{{-2, -1, 1}, {2}},
 I{{-2, 1}, {-1}, {2}},
 I{{-2, -1, 2}, {1}},
 I{{-2, 2}, {-1}, {1}},
 I{{-2}, {-1, 1}, {2}},
 I{{-2}, {-1, 2}, {1}},
 I{{-2, -1}, {1}, {2}},
 I{{-2}, {-1}, {1}, {2}}]
sage: E = I([[1,2],[-1,-2]])
sage: E^2 == x*E
True
sage: E^5 == x^4*E
True
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> I = PropagatingIdeal(Integer(2), x, R); I
Propagating Ideal of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Rational Field
>>> I.basis().keys()
Ideal diagrams of order 2
>>> I.basis().list()
[I{{-2, -1, 1, 2}},
 I{{-2, 1, 2}, {-1}},
 I{{-2}, {-1, 1, 2}},
 I{{-2, -1}, {1, 2}},
 I{{-2}, {-1}, {1, 2}},
 I{{-2, -1, 1}, {2}},
 I{{-2, 1}, {-1}, {2}},
 I{{-2, -1, 2}, {1}},
 I{{-2, 2}, {-1}, {1}},
 I{{-2}, {-1, 1}, {2}},
 I{{-2}, {-1, 2}, {1}},
 I{{-2, -1}, {1}, {2}},
 I{{-2}, {-1}, {1}, {2}}]
>>> E = I([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
>>> E**Integer(2) == x*E
True
>>> E**Integer(5) == x**Integer(4)*E
True
```

class Element

Bases: *Element*

An element of a propagating ideal.

We need to take care of exponents since we are not unital.

class sage.combinat.diagram_algebras.**SubPartitionAlgebra**(*k*, *q*, *base_ring*, *prefix*, *diagrams*, *category=None*)

Bases: *DiagramBasis*

A subalgebra of the partition algebra in the diagram basis indexed by a subset of the diagrams.

class Element

Bases: `Element`

to_orbit_basis()

Return `self` in the orbit basis of the associated ambient partition algebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: B = BrauerAlgebra(2, x, R)
sage: bb = B([[-2, -1], [1, 2]]); bb
B{{-2, -1}, {1, 2}}
sage: bb.to_orbit_basis()
O{{-2, -1}, {1, 2}} + O{{-2, -1, 1, 2}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> B = BrauerAlgebra(Integer(2), x, R)
>>> bb = B([[-Integer(2), -Integer(1)], [Integer(1), Integer(2)]]); bb
B{{-2, -1}, {1, 2}}
>>> bb.to_orbit_basis()
O{{-2, -1}, {1, 2}} + O{{-2, -1, 1, 2}}
```

ambient()

Return the partition algebra `self` is a sub-algebra of.

EXAMPLES:

```
sage: x = var('x')                                                    #␣
↪needs sage.symbolic
sage: BA = BrauerAlgebra(2, x)                                        #␣
↪needs sage.symbolic
sage: BA.ambient()                                                    #␣
↪needs sage.symbolic
Partition Algebra of rank 2 with parameter x over Symbolic Ring
```

```
>>> from sage.all import *
>>> x = var('x')                                                    #␣
↪needs sage.symbolic
>>> BA = BrauerAlgebra(Integer(2), x)                                ␣
↪      # needs sage.symbolic
>>> BA.ambient()                                                    #␣
↪needs sage.symbolic
Partition Algebra of rank 2 with parameter x over Symbolic Ring
```

lift()

Return the lift map from diagram subalgebra to the ambient space.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: BA = BrauerAlgebra(2, x, R)
sage: E = BA([[1,2],[-1,-2]])
sage: lifted = BA.lift(E); lifted
```

(continues on next page)

(continued from previous page)

```
B{{-2, -1}, {1, 2}}
sage: lifted.parent() is BA.ambient()
True
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> BA = BrauerAlgebra(Integer(2), x, R)
>>> E = BA([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
>>> lifted = BA.lift(E); lifted
B{{-2, -1}, {1, 2}}
>>> lifted.parent() is BA.ambient()
True
```

retract(*x*)

Retract an appropriate partition algebra element to the corresponding element in the partition subalgebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: BA = BrauerAlgebra(2, x, R)
sage: PA = BA.ambient()
sage: E = PA([[1,2], [-1,-2]])
sage: BA.retract(E) in BA
True
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> BA = BrauerAlgebra(Integer(2), x, R)
>>> PA = BA.ambient()
>>> E = PA([[Integer(1),Integer(2)], [-Integer(1),-Integer(2)]])
>>> BA.retract(E) in BA
True
```

sage.combinat.diagram_algebras.**TL_diagram_ascii_art**(*diagram*, *use_unicode=False*, *blobs=[]*)

Return ascii art for a Temperley-Lieb diagram `diagram`.

INPUT:

- `diagram` – list of pairs of matchings of the set $\{-1, \ldots, -n, 1, \ldots, n\}$
- `use_unicode` – boolean (default: `False`); whether or not to use unicode art instead of ascii art
- `blobs` – (optional) a list of matchings with blobs on them

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import TL_diagram_ascii_art
sage: TL = [(-15,-12), (-14,-13), (-11,15), (-10,14), (-9,-6),
....:       (-8,-7), (-5,-4), (-3,1), (-2,-1), (2,3), (4,5),
....:       (6,11), (7, 8), (9,10), (12,13)]
sage: TL_diagram_ascii_art(TL, use_unicode=False)
 o o o o o o o o o o o o o o o
 | `-` `-` | `-` `-` | `-` | |
 |         `---------`     | |
```

(continues on next page)

(continued from previous page)

```
 |                 .-------` |
 `---.             | .-------`
     |     .-----. | | .-----.
 .-. | .-. | .-. | | | | .-. |
 o o o o o o o o o o o o o o o
sage: TL_diagram_ascii_art(TL, use_unicode=True)
 ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬
 │ └─┘ └─┘ │ └─┘ └─┘ │ └─┘ │ │
 │         └─────────┘     │ │
 │                 ┌───────┘ │
 └───┐             │ ┌───────┘
     │     ┌─────┐ │ │ ┌─────┐
 ┌─┐ │ ┌─┐ │ ┌─┐ │ │ │ │ ┌─┐ │
 ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬

sage: TL = [(-20,-9), (-19,-10), (-18,-11), (-17,-16), (-15,-12), (2,3),
....:       (-14,-13), (-8,16), (-7,7), (-6,6), (-5,1), (-4,-3), (-2,-1),
....:       (4,5), (8,15), (9,10), (11,14), (12,13), (17,20), (18,19)]
sage: TL_diagram_ascii_art(TL, use_unicode=False, blobs=[(-2,-1), (-5,1)])
 o o o o o o o o o o o o o o o o o o o o
 | `-` `-` | | | `-` | `-` | | | | `-` |
 |         | | |     `-----` | | `-----`
 |         | | `-------------` |
 `---0---. | | .---------------`
         | | | | .---------------------.
         | | | | | .-----------------. |
         | | | | | | .-------------. | |
         | | | | | | | .-----.     | | |
 .0. .-. | | | | | | | | .-. | .-. | | |
 o o o o o o o o o o o o o o o o o o o o
sage: TL_diagram_ascii_art(TL, use_unicode=True, blobs=[(-2,-1), (-5,1)])
 ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬
 │ └─┘ └─┘ │ │ │ └─┘ │ └─┘ │ │ │ │ └─┘ │
 │         │ │ │     └─────┘ │ │ └─────┘
 │         │ │ └─────────────┘ │
 └───●───┐ │ │ ┌───────────────┘
         │ │ │ │ ┌─────────────────────┐
         │ │ │ │ │ ┌─────────────────┐ │
         │ │ │ │ │ │ ┌─────────────┐ │ │
         │ │ │ │ │ │ │ ┌─────┐     │ │ │
 ┌●┐ ┌─┐ │ │ │ │ │ │ │ │ ┌─┐ │ ┌─┐ │ │ │
 ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import TL_diagram_ascii_art
>>> TL = [(-Integer(15),-Integer(12)), (-Integer(14),-Integer(13)), (-Integer(11),
↪Integer(15)), (-Integer(10),Integer(14)), (-Integer(9),-Integer(6)),
...       (-Integer(8),-Integer(7)), (-Integer(5),-Integer(4)), (-Integer(3),
↪Integer(1)), (-Integer(2),-Integer(1)), (Integer(2),Integer(3)), (Integer(4),
↪Integer(5)),
...       (Integer(6),Integer(11)), (Integer(7), Integer(8)), (Integer(9),
↪Integer(10)), (Integer(12),Integer(13))]
```

(continues on next page)

(continued from previous page)

```
>>> TL_diagram_ascii_art(TL, use_unicode=False)
 o o o o o o o o o o o o o o o
 | `-` `-` | `-` `-` | `-` | |
 |         `---------`     | |
 |                 .-------` |
 `---.             | .-------`
     |     .-----. | | .-----.
 .-. | .-. | .-. | | | | .-. |
 o o o o o o o o o o o o o o o
>>> TL_diagram_ascii_art(TL, use_unicode=True)
 ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬
 │ ╰─╯ ╰─╯ │ ╰─╯ ╰─╯ │ ╰─╯ │ │
 │         ╰─────────╯     │ │
 │                 ╭───────╯ │
 ╰───╮             │ ╭───────╯
     │     ╭─────╮ │ │ ╭─────╮
 ╭─╮ │ ╭─╮ │ ╭─╮ │ │ │ │ ╭─╮ │
 ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬

>>> TL = [(-Integer(20),-Integer(9)), (-Integer(19),-Integer(10)), (-Integer(18),-
↪Integer(11)), (-Integer(17),-Integer(16)), (-Integer(15),-Integer(12)),␣
↪(Integer(2),Integer(3)),
...       (-Integer(14),-Integer(13)), (-Integer(8),Integer(16)), (-Integer(7),
↪Integer(7)), (-Integer(6),Integer(6)), (-Integer(5),Integer(1)), (-Integer(4),-
↪Integer(3)), (-Integer(2),-Integer(1)),
...       (Integer(4),Integer(5)), (Integer(8),Integer(15)), (Integer(9),
↪Integer(10)), (Integer(11),Integer(14)), (Integer(12),Integer(13)),␣
↪(Integer(17),Integer(20)), (Integer(18),Integer(19))]
>>> TL_diagram_ascii_art(TL, use_unicode=False, blobs=[(-Integer(2),-Integer(1)),␣
↪(-Integer(5),Integer(1))])
 o o o o o o o o o o o o o o o o o o o o
 | `-` `-` | | | `-` | `-` | | | | `-` |
 |         | | |     `-----` | | `-----`
 |         | | `-------------` |
 `---0---. | | .---------------`
         | | | | .---------------------.
         | | | | | .-----------------. |
         | | | | | | .-------------. | |
         | | | | | | | .-----.     | | |
 .0. .-. | | | | | | | | .-. | .-. | | |
 o o o o o o o o o o o o o o o o o o o o
>>> TL_diagram_ascii_art(TL, use_unicode=True, blobs=[(-Integer(2),-Integer(1)),␣
↪(-Integer(5),Integer(1))])
 ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬ ⚬
 │ ╰─╯ ╰─╯ │ │ │ ╰─╯ │ ╰─╯ │ │ │ │ ╰─╯ │
 │         │ │ │     ╰─────╯ │ │ ╰─────╯
 │         │ │ ╰─────────────╯ │
 ╰───●───╮ │ │ ╭───────────────╯
         │ │ │ │ ╭─────────────────────╮
         │ │ │ │ │ ╭─────────────────╮ │
         │ │ │ │ │ │ ╭─────────────╮ │ │
         │ │ │ │ │ │ │ ╭─────╮     │ │ │
```

(continues on next page)

(continued from previous page)

```
┌●┐ ┌─┐ │ │ │ │ │ │ │ │ ┌─┐ │ ┌─┐ │ │ │
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
```

class sage.combinat.diagram_algebras.**TemperleyLiebAlgebra**(*k*, *q*, *base_ring*, *prefix*)

Bases: *SubPartitionAlgebra*, *UnitDiagramMixin*

A Temperley–Lieb algebra.

The Temperley–Lieb algebra of rank k is an algebra with basis indexed by the collection of planar set partitions of $\{1, \ldots, k, -1, \ldots, -k\}$ with block size 2.

This algebra is thus a subalgebra of the partition algebra. For more information, see *PartitionAlgebra*.

INPUT:

- `k` – rank of the algebra
- `q` – the deformation parameter q
- `base_ring` – (default: `None`) a ring containing `q`; if `None` then just takes the parent of `q`
- `prefix` – (default: `'T'`) a label for the basis elements

EXAMPLES:

We define the Temperley–Lieb algebra of rank 2 with parameter x over $\mathbf{Z}$:

```
sage: R.<x> = ZZ[]
sage: T = TemperleyLiebAlgebra(2, x, R); T
Temperley-Lieb Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
sage: T.basis()
Lazy family (Term map from Temperley Lieb diagrams of order 2
 to Temperley-Lieb Algebra of rank 2 with parameter x over
 Univariate Polynomial Ring in x over Integer
 Ring(i))_{i in Temperley Lieb diagrams of order 2}
sage: T.basis().keys()
Temperley Lieb diagrams of order 2
sage: T.basis().keys()([[-1, 1], [2, -2]])
{{-2, 2}, {-1, 1}}
sage: b = T.basis().list(); b
[T{{-2, -1}, {1, 2}}, T{{-2, 2}, {-1, 1}}]
sage: b[0]
T{{-2, -1}, {1, 2}}
sage: b[0]^2 == x*b[0]
True
sage: b[0]^5 == x^4*b[0]
True
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> T = TemperleyLiebAlgebra(Integer(2), x, R); T
Temperley-Lieb Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
>>> T.basis()
Lazy family (Term map from Temperley Lieb diagrams of order 2
 to Temperley-Lieb Algebra of rank 2 with parameter x over
```

(continues on next page)

(continued from previous page)

```
 Univariate Polynomial Ring in x over Integer
 Ring(i))_{i in Temperley Lieb diagrams of order 2}
>>> T.basis().keys()
Temperley Lieb diagrams of order 2
>>> T.basis().keys()([[-Integer(1), Integer(1)], [Integer(2), -Integer(2)]])
{{-2, 2}, {-1, 1}}
>>> b = T.basis().list(); b
[T{{-2, -1}, {1, 2}}, T{{-2, 2}, {-1, 1}}]
>>> b[Integer(0)]
T{{-2, -1}, {1, 2}}
>>> b[Integer(0)]**Integer(2) == x*b[Integer(0)]
True
>>> b[Integer(0)]**Integer(5) == x**Integer(4)*b[Integer(0)]
True
```

The Temperley-Lieb algebra is a cellular algebra, and we verify that the dimensions of the simple modules at $q = 0$ is given by OEIS sequence A050166:

```
sage: for k in range(1,5):
....:     TL = TemperleyLiebAlgebra(2*k, 0, QQ)
....:     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension())
....:                   for la in reversed(TL.cell_poset()) if la != 0))
  1
  1  2
  1  4  5
  1  6 14 14
sage: for k in range(1,4):
....:     TL = TemperleyLiebAlgebra(2*k+1, 0, QQ)
....:     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension())
....:                   for la in reversed(TL.cell_poset()) if la != 0))
  1  2
  1  4  5
  1  6 14 14
```

```
>>> from sage.all import *
>>> for k in range(Integer(1),Integer(5)):
...     TL = TemperleyLiebAlgebra(Integer(2)*k, Integer(0), QQ)
...     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension())
...                   for la in reversed(TL.cell_poset()) if la != Integer(0)))
  1
  1  2
  1  4  5
  1  6 14 14
>>> for k in range(Integer(1),Integer(4)):
...     TL = TemperleyLiebAlgebra(Integer(2)*k+Integer(1), Integer(0), QQ)
...     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension())
...                   for la in reversed(TL.cell_poset()) if la != Integer(0)))
  1  2
```

(continues on next page)

(continued from previous page)

```
  1   4   5
  1   6  14  14
```

Additional examples when the Temperley-Lieb algebra is not semisimple:

```
sage: TL = TemperleyLiebAlgebra(8, -1, QQ)
sage: for la in TL.cell_poset():
....:     CM = TL.cell_module(la)
....:     if not CM.nonzero_bilinear_form():
....:         continue
....:     print(la, CM.dimension(), CM.simple_module().dimension())
....:
0 14 1
2 28 28
4 20 13
6 7 7
8 1 1
sage: for k in range(1,5):
....:     TL = TemperleyLiebAlgebra(2*k, -1, QQ)
....:     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension())
....:                   for la in reversed(TL.cell_poset())
....:                    if TL.cell_module(la).nonzero_bilinear_form()))
  1   1
  1   3   1
  1   4   9   1
  1   7  13  28   1
sage: C5.<z5> = CyclotomicField(5)
sage: for k in range(1,5):
....:     TL = TemperleyLiebAlgebra(2*k, z5+~z5, C5)
....:     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension())
....:                   for la in reversed(TL.cell_poset())
....:                    if TL.cell_module(la).nonzero_bilinear_form()))
  1   1
  1   3   2
  1   5   8   5
  1   7  20  21  13
```

```
>>> from sage.all import *
>>> TL = TemperleyLiebAlgebra(Integer(8), -Integer(1), QQ)
>>> for la in TL.cell_poset():
...     CM = TL.cell_module(la)
...     if not CM.nonzero_bilinear_form():
...         continue
...     print(la, CM.dimension(), CM.simple_module().dimension())
....:
0 14 1
2 28 28
4 20 13
6 7 7
8 1 1
```

(continues on next page)

(continued from previous page)

```
>>> for k in range(Integer(1),Integer(5)):
...     TL = TemperleyLiebAlgebra(Integer(2)*k, -Integer(1), QQ)
...     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension())
...                   for la in reversed(TL.cell_poset())
...                    if TL.cell_module(la).nonzero_bilinear_form()))
  1  1
  1  3  1
  1  4  9  1
  1  7 13 28  1
>>> C5 = CyclotomicField(Integer(5), names=('z5',)); (z5,) = C5._first_ngens(1)
>>> for k in range(Integer(1),Integer(5)):
...     TL = TemperleyLiebAlgebra(Integer(2)*k, z5+~z5, C5)
...     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension())
...                   for la in reversed(TL.cell_poset())
...                    if TL.cell_module(la).nonzero_bilinear_form()))
  1  1
  1  3  2
  1  5  8  5
  1  7 20 21 13
```

cell_module_indices(*la*)

Return the indices of the cell module of `self` indexed by `la` .

This is the finite set $M(\lambda)$.

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: TL = TemperleyLiebAlgebra(8, q, R)
sage: TL.cell_module_indices(4)
Half Temperley-Lieb diagrams of order 8 with 4 defects
```

```
>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> TL = TemperleyLiebAlgebra(Integer(8), q, R)
>>> TL.cell_module_indices(Integer(4))
Half Temperley-Lieb diagrams of order 8 with 4 defects
```

cell_poset()

Return the cell poset of `self`.

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: TL = TemperleyLiebAlgebra(7, q, R)
sage: TL.cell_poset().cover_relations()
[[1, 3], [3, 5], [5, 7]]

sage: TL = TemperleyLiebAlgebra(8, q, R)
sage: TL.cell_poset().cover_relations()
[[0, 2], [2, 4], [4, 6], [6, 8]]
```

```
>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> TL = TemperleyLiebAlgebra(Integer(7), q, R)
>>> TL.cell_poset().cover_relations()
[[1, 3], [3, 5], [5, 7]]

>>> TL = TemperleyLiebAlgebra(Integer(8), q, R)
>>> TL.cell_poset().cover_relations()
[[0, 2], [2, 4], [4, 6], [6, 8]]
```

cellular_involution(*x*)

Return the cellular involution of `x` in `self`.

EXAMPLES:

```
sage: TL = TemperleyLiebAlgebra(4, QQ.zero(), QQ)
sage: ascii_art(TL.an_element())
                o o o o       o o o o
   o o o o      | `-` |       | `-` |
2* `-` `-` + 2* `-----`  + 3* `---. |
   .-. .-.      .-. .-.       .-. | |
   o o o o      o o o o       o o o o
sage: ascii_art(TL.cellular_involution(TL.an_element()))
                o o o o       o o o o
   o o o o      `-` `-`       `-` | |
2* `-` `-` + 2* .-----.  + 3* .---` |
   .-. .-.      | .-. |       | .-. |
   o o o o      o o o o       o o o o
```

```
>>> from sage.all import *
>>> TL = TemperleyLiebAlgebra(Integer(4), QQ.zero(), QQ)
>>> ascii_art(TL.an_element())
                o o o o       o o o o
   o o o o      | `-` |       | `-` |
2* `-` `-` + 2* `-----`  + 3* `---. |
   .-. .-.      .-. .-.       .-. | |
   o o o o      o o o o       o o o o
>>> ascii_art(TL.cellular_involution(TL.an_element()))
                o o o o       o o o o
   o o o o      `-` `-`       `-` | |
2* `-` `-` + 2* .-----.  + 3* .---` |
   .-. .-.      | .-. |       | .-. |
   o o o o      o o o o       o o o o
```

class sage.combinat.diagram_algebras.**TemperleyLiebDiagram**(*parent*, *d*, *check=True*)

Bases: *AbstractPartitionDiagram*

The element class for a Temperley-Lieb diagram.

A Temperley-Lieb diagram for an integer k is a partition of the set $\{1, \ldots, k, -1, \ldots, -k\}$ so that the blocks are all of size 2 and the diagram is planar.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import TemperleyLiebDiagrams
sage: TemperleyLiebDiagrams(2)
Temperley Lieb diagrams of order 2
sage: TemperleyLiebDiagrams(2).list()
[{{-2, -1}, {1, 2}}, {{-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import TemperleyLiebDiagrams
>>> TemperleyLiebDiagrams(Integer(2))
Temperley Lieb diagrams of order 2
>>> TemperleyLiebDiagrams(Integer(2)).list()
[{{-2, -1}, {1, 2}}, {{-2, 2}, {-1, 1}}]
```

check()

Check the validity of the input for `self`.

class sage.combinat.diagram_algebras.**TemperleyLiebDiagrams**(*order*, *category=None*)

Bases: *AbstractPartitionDiagrams*

All Temperley-Lieb diagrams of integer or integer $+1/2$ order.

For more information on Temperley-Lieb diagrams, see *TemperleyLiebAlgebra*.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: td = da.TemperleyLiebDiagrams(3); td
Temperley Lieb diagrams of order 3
sage: td.list()
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 1}, {-2, -1}, {2, 3}},
 {{-3, -2}, {-1, 1}, {2, 3}},
 {{-3, -2}, {-1, 3}, {1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]

sage: td = da.TemperleyLiebDiagrams(5/2); td
Temperley Lieb diagrams of order 5/2
sage: td.list()
[{{-3, 3}, {-2, -1}, {1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> td = da.TemperleyLiebDiagrams(Integer(3)); td
Temperley Lieb diagrams of order 3
>>> td.list()
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 1}, {-2, -1}, {2, 3}},
 {{-3, -2}, {-1, 1}, {2, 3}},
 {{-3, -2}, {-1, 3}, {1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]

>>> td = da.TemperleyLiebDiagrams(Integer(5)/Integer(2)); td
Temperley Lieb diagrams of order 5/2
```

(continues on next page)

(continued from previous page)

```
>>> td.list()
[{{-3, 3}, {-2, -1}, {1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}]
```

Element

alias of *TemperleyLiebDiagram*

cardinality()

Return the cardinality of `self`.

The number of Temperley–Lieb diagrams of integer order k is the k-th Catalan number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: td = da.TemperleyLiebDiagrams(3)
sage: td.cardinality()
5
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> td = da.TemperleyLiebDiagrams(Integer(3))
>>> td.cardinality()
5
```

class sage.combinat.diagram_algebras.**UnitDiagramMixin**

Bases: `object`

Mixin class for diagram algebras that have the unit indexed by the *identity_set_partition()*.

one_basis()

The following constructs the identity element of `self`.

It is not called directly; instead one should use `DA.one()` if `DA` is a defined diagram algebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(2, x, R)
sage: P.one_basis()
{{-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> P.one_basis()
{{-2, 2}, {-1, 1}}
```

sage.combinat.diagram_algebras.**brauer_diagrams**(k)

Return a generator of all Brauer diagrams of order `k`.

A Brauer diagram of order k is a partition diagram of order k with block size 2.

INPUT:

- `k` – the order of the Brauer diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.brauer_diagrams(2)]
[{{-2, -1}, {1, 2}}, {{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]
sage: [SetPartition(p) for p in da.brauer_diagrams(5/2)]
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> [SetPartition(p) for p in da.brauer_diagrams(Integer(2))]
[{{-2, -1}, {1, 2}}, {{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]
>>> [SetPartition(p) for p in da.brauer_diagrams(Integer(5)/Integer(2))]
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]
```

sage.combinat.diagram_algebras.**diagram_latex**(*diagram*, *fill=False*, *edge_options=None*, *edge_additions=None*)

Return latex code for the diagram `diagram` using tikz.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PartitionDiagrams, diagram_latex
sage: P = PartitionDiagrams(2)
sage: D = P([[1,2],[-2,-1]])
sage: print(diagram_latex(D)) # indirect doctest
\begin{tikzpicture}[scale = 0.5,thick, baseline={(0,-1ex/2)}]
\tikzstyle{vertex} = [shape = circle, minimum size = 7pt, inner sep = 1pt]
\node[vertex] (G--2) at (1.5, -1) [shape = circle, draw] {};
\node[vertex] (G--1) at (0.0, -1) [shape = circle, draw] {};
\node[vertex] (G-1) at (0.0, 1) [shape = circle, draw] {};
\node[vertex] (G-2) at (1.5, 1) [shape = circle, draw] {};
\draw[] (G--2) .. controls +(-0.5, 0.5) and +(0.5, 0.5) .. (G--1);
\draw[] (G-1) .. controls +(0.5, -0.5) and +(-0.5, -0.5) .. (G-2);
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagrams, diagram_latex
>>> P = PartitionDiagrams(Integer(2))
>>> D = P([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])
>>> print(diagram_latex(D)) # indirect doctest
\begin{tikzpicture}[scale = 0.5,thick, baseline={(0,-1ex/2)}]
\tikzstyle{vertex} = [shape = circle, minimum size = 7pt, inner sep = 1pt]
\node[vertex] (G--2) at (1.5, -1) [shape = circle, draw] {};
\node[vertex] (G--1) at (0.0, -1) [shape = circle, draw] {};
\node[vertex] (G-1) at (0.0, 1) [shape = circle, draw] {};
\node[vertex] (G-2) at (1.5, 1) [shape = circle, draw] {};
\draw[] (G--2) .. controls +(-0.5, 0.5) and +(0.5, 0.5) .. (G--1);
\draw[] (G-1) .. controls +(0.5, -0.5) and +(-0.5, -0.5) .. (G-2);
\end{tikzpicture}
```

sage.combinat.diagram_algebras.**ideal_diagrams**(*k*)

Return a generator of all "ideal" diagrams of order k.

An ideal diagram of order k is a partition diagram of order k with propagating number less than k.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: all_diagrams = da.partition_diagrams(2)
sage: [SetPartition(p) for p in all_diagrams if p not in da.ideal_diagrams(2)]
[{{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]

sage: all_diagrams = da.partition_diagrams(3/2)
sage: [SetPartition(p) for p in all_diagrams if p not in da.ideal_diagrams(3/2)]
[{{-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> all_diagrams = da.partition_diagrams(Integer(2))
>>> [SetPartition(p) for p in all_diagrams if p not in da.ideal_
↪diagrams(Integer(2))]
[{{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]

>>> all_diagrams = da.partition_diagrams(Integer(3)/Integer(2))
>>> [SetPartition(p) for p in all_diagrams if p not in da.ideal_
↪diagrams(Integer(3)/Integer(2))]
[{{-2, 2}, {-1, 1}}]
```

sage.combinat.diagram_algebras.**identity_set_partition**(*k*)

Return the identity set partition $\{\{1,-1\},\ldots,\{k,-k\}\}$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: SetPartition(da.identity_set_partition(2))
{{-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> SetPartition(da.identity_set_partition(Integer(2)))
{{-2, 2}, {-1, 1}}
```

sage.combinat.diagram_algebras.**is_planar**(*sp*)

Return True if the diagram corresponding to the set partition sp is planar; otherwise, return False.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: da.is_planar( da.to_set_partition([[1,-2],[2,-1]]))
False
sage: da.is_planar( da.to_set_partition([[1,-1],[2,-2]]))
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
```

(continues on next page)

(continued from previous page)

```
>>> da.is_planar( da.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-
↪Integer(1)]]))
False
>>> da.is_planar( da.to_set_partition([[Integer(1),-Integer(1)],[Integer(2),-
↪Integer(2)]]))
True
```

sage.combinat.diagram_algebras.**pair_to_graph**(*sp1*, *sp2*)

Return a graph consisting of the disjoint union of the graphs of set partitions `sp1` and `sp2` along with edges joining the bottom row (negative numbers) of `sp1` to the top row (positive numbers) of `sp2`.

The vertices of the graph `sp1` appear in the result as pairs `(k, 1)`, whereas the vertices of the graph `sp2` appear as pairs `(k, 2)`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,-2],[2,-1]])
sage: g = da.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices

sage: g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges(sort=True)
[((-2, 1), (1, 1), None), ((-2, 1), (2, 2), None),
 ((-2, 2), (1, 2), None), ((-1, 1), (1, 2), None),
 ((-1, 1), (2, 1), None), ((-1, 2), (2, 2), None)]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> sp1 = da.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> sp2 = da.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> g = da.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices

>>> g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
>>> g.edges(sort=True)
[((-2, 1), (1, 1), None), ((-2, 1), (2, 2), None),
 ((-2, 2), (1, 2), None), ((-1, 1), (1, 2), None),
 ((-1, 1), (2, 1), None), ((-1, 2), (2, 2), None)]
```

Another example which used to be wrong until Issue #15958:

```
sage: sp3 = da.to_set_partition([[1, -1], [2], [-2]])
sage: sp4 = da.to_set_partition([[1], [-1], [2], [-2]])
sage: g = da.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

sage: g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges(sort=True)
```

(continues on next page)

(continued from previous page)

```
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

```
>>> from sage.all import *
>>> sp3 = da.to_set_partition([[Integer(1), -Integer(1)], [Integer(2)], [-
↪Integer(2)]])
>>> sp4 = da.to_set_partition([[Integer(1)], [-Integer(1)], [Integer(2)], [-
↪Integer(2)]])
>>> g = da.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

>>> g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
>>> g.edges(sort=True)
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

sage.combinat.diagram_algebras.**partition_diagrams**(k)

Return a generator of all partition diagrams of order `k`.

A partition diagram of order $k \in \mathbf{Z}$ to is a set partition of $\{1, \ldots, k, -1, \ldots, -k\}$. If we have $k - 1/2 \in ZZ$, then a partition diagram of order $k \in 1/2\mathbf{Z}$ is a set partition of $\{1, \ldots, k + 1/2, -1, \ldots, -(k + 1/2)\}$ with $k + 1/2$ and $-(k + 1/2)$ in the same block. See [HR2005].

INPUT:

- `k` – the order of the partition diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.partition_diagrams(2)]
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2}, {-1}, {1, 2}},
 {{-2, -1, 1}, {2}},
 {{-2, 1}, {-1, 2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1}, {1}, {2}}]
sage: [SetPartition(p) for p in da.partition_diagrams(3/2)]
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> [SetPartition(p) for p in da.partition_diagrams(Integer(2))]
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2}, {-1}, {1, 2}},
 {{-2, -1, 1}, {2}},
 {{-2, 1}, {-1, 2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1}, {1}, {2}}]
>>> [SetPartition(p) for p in da.partition_diagrams(Integer(3)/Integer(2))]
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]
```

sage.combinat.diagram_algebras.**planar_diagrams**(k)

Return a generator of all planar diagrams of order `k`.

A planar diagram of order k is a partition diagram of order k that has no crossings.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: all_diagrams = [SetPartition(p) for p in da.partition_diagrams(2)]
sage: da2 = [SetPartition(p) for p in da.planar_diagrams(2)]
sage: [p for p in all_diagrams if p not in da2]
[{{-2, 1}, {-1, 2}}]
sage: all_diagrams = [SetPartition(p) for p in da.partition_diagrams(5/2)]
sage: da5o2 = [SetPartition(p) for p in da.planar_diagrams(5/2)]
sage: [p for p in all_diagrams if p not in da5o2]
[{{-3, -1, 3}, {-2, 1, 2}},
 {{-3, -2, 1, 3}, {-1, 2}},
 {{-3, -1, 1, 3}, {-2, 2}},
 {{-3, 1, 3}, {-2, -1, 2}},
 {{-3, 1, 3}, {-2, 2}, {-1}},
 {{-3, 1, 3}, {-2}, {-1, 2}},
 {{-3, -1, 2, 3}, {-2, 1}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, -1, 3}, {-2, 1}, {2}},
 {{-3, -1, 3}, {-2, 2}, {1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
```

(continues on next page)

(continued from previous page)

```
>>> all_diagrams = [SetPartition(p) for p in da.partition_diagrams(Integer(2))]
>>> da2 = [SetPartition(p) for p in da.planar_diagrams(Integer(2))]
>>> [p for p in all_diagrams if p not in da2]
[{{-2, 1}, {-1, 2}}]
>>> all_diagrams = [SetPartition(p) for p in da.partition_diagrams(Integer(5)/
↪Integer(2))]
>>> da5o2 = [SetPartition(p) for p in da.planar_diagrams(Integer(5)/Integer(2))]
>>> [p for p in all_diagrams if p not in da5o2]
[{{-3, -1, 3}, {-2, 1, 2}},
 {{-3, -2, 1, 3}, {-1, 2}},
 {{-3, -1, 1, 3}, {-2, 2}},
 {{-3, 1, 3}, {-2, -1, 2}},
 {{-3, 1, 3}, {-2, 2}, {-1}},
 {{-3, 1, 3}, {-2}, {-1, 2}},
 {{-3, -1, 2, 3}, {-2, 1}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, -1, 3}, {-2, 1}, {2}},
 {{-3, -1, 3}, {-2, 2}, {1}}]
```

sage.combinat.diagram_algebras.**planar_partitions_rec**(*X*)

Iterate over all planar set partitions of X by using a recursive algorithm.

ALGORITHM:

To construct the set partition $\rho = \{\rho_1, \ldots, \rho_k\}$ of $[n]$, we remove the part of the set partition containing the last element of X, which, we consider to be $\rho_k = \{i_1, \ldots, i_m\}$ without loss of generality. The remaining parts come from the planar set partitions of $\{1, \ldots, i_1 - 1\}, \{i_1 + 1, \ldots, i_2 - 1\}, \ldots, \{i_m + 1, \ldots, n\}$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: list(da.planar_partitions_rec([1,2,3]))
[([1, 2], [3]), ([1], [2], [3]), ([2], [1, 3]), ([1], [2, 3]), ([1, 2, 3],)]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> list(da.planar_partitions_rec([Integer(1),Integer(2),Integer(3)]))
[([1, 2], [3]), ([1], [2], [3]), ([2], [1, 3]), ([1], [2, 3]), ([1, 2, 3],)]
```

sage.combinat.diagram_algebras.**propagating_number**(*sp*)

Return the propagating number of the set partition sp.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,2],[-2,-1]])
sage: da.propagating_number(sp1)
2
sage: da.propagating_number(sp2)
0
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> sp1 = da.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> sp2 = da.to_set_partition([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])
>>> da.propagating_number(sp1)
2
>>> da.propagating_number(sp2)
0
```

sage.combinat.diagram_algebras.**temperley_lieb_diagrams**(*k*)

Return a generator of all Temperley–Lieb diagrams of order k.

A Temperley–Lieb diagram of order k is a partition diagram of order k with block size 2 and is planar.

INPUT:

- k – the order of the Temperley–Lieb diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.temperley_lieb_diagrams(2)]
[{{-2, -1}, {1, 2}}, {{-2, 2}, {-1, 1}}]
sage: [SetPartition(p) for p in da.temperley_lieb_diagrams(5/2)]
[{{-3, 3}, {-2, -1}, {1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> [SetPartition(p) for p in da.temperley_lieb_diagrams(Integer(2))]
[{{-2, -1}, {1, 2}}, {{-2, 2}, {-1, 1}}]
>>> [SetPartition(p) for p in da.temperley_lieb_diagrams(Integer(5)/Integer(2))]
[{{-3, 3}, {-2, -1}, {1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}]
```

sage.combinat.diagram_algebras.**to_Brauer_partition**(*l*, *k=None*)

Same as *to_set_partition()* but assumes omitted elements are connected straight through.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: f = lambda sp: SetPartition(da.to_Brauer_partition(sp))
sage: f([[1,2],[-1,-2]]) == SetPartition([[1,2],[-1,-2]])
True
sage: f([[1,3],[-1,-3]]) == SetPartition([[1,3],[-3,-1],[2,-2]])
True
sage: f([[1,-4],[-3,-1],[3,4]]) == SetPartition([[-3,-1],[2,-2],[1,-4],[3,4]])
True
sage: p = SetPartition([[1,2],[-1,-2],[3,-3],[4,-4]])
sage: SetPartition(da.to_Brauer_partition([[1,2],[-1,-2]], k=4)) == p
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> f = lambda sp: SetPartition(da.to_Brauer_partition(sp))
>>> f([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]) ==␣
```

(continues on next page)

(continued from previous page)

```
↪SetPartition([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
True
>>> f([[Integer(1),Integer(3)],[-Integer(1),-Integer(3)]]) ==␣
↪SetPartition([[Integer(1),Integer(3)],[-Integer(3),-Integer(1)],[Integer(2),-
↪Integer(2)]])
True
>>> f([[Integer(1),-Integer(4)],[-Integer(3),-Integer(1)],[Integer(3),
↪Integer(4)]]) == SetPartition([[-Integer(3),-Integer(1)],[Integer(2),-
↪Integer(2)],[Integer(1),-Integer(4)],[Integer(3),Integer(4)]])
True
>>> p = SetPartition([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)],
↪[Integer(3),-Integer(3)],[Integer(4),-Integer(4)]])
>>> SetPartition(da.to_Brauer_partition([[Integer(1),Integer(2)],[-Integer(1),-
↪Integer(2)]], k=Integer(4))) == p
True
```

`sage.combinat.diagram_algebras.`**`to_graph`**`(`*sp*`)`

Return a graph representing the set partition `sp`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: g = da.to_graph( da.to_set_partition([[1,-2],[2,-1]])); g
Graph on 4 vertices

sage: g.vertices(sort=True)
[-2, -1, 1, 2]
sage: g.edges(sort=True)
[(-2, 1, None), (-1, 2, None)]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> g = da.to_graph( da.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-
↪Integer(1)]])); g
Graph on 4 vertices

>>> g.vertices(sort=True)
[-2, -1, 1, 2]
>>> g.edges(sort=True)
[(-2, 1, None), (-1, 2, None)]
```

`sage.combinat.diagram_algebras.`**`to_set_partition`**`(`*l*, *k=None*`)`

Convert input to a set partition of $\{1, \ldots, k, -1, \ldots, -k\}$.

Convert a list of a list of numbers to a set partitions. Each list of numbers in the outer list specifies the numbers contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of $\{1, \ldots, k, -1, \ldots, -k\}$. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

INPUT:

- `l` – list of lists of integers
- `k` – integer (default: `None`)

OUTPUT: list of sets

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: f = lambda sp: SetPartition(da.to_set_partition(sp))
sage: f([[1,-1],[2,-2]]) == SetPartition(da.identity_set_partition(2))
True
sage: da.to_set_partition([[1]])
[{1}, {-1}]
sage: da.to_set_partition([[1,-1],[-2,3]],9/2)
[{-1, 1}, {-2, 3}, {2}, {-4, 4}, {-5, 5}, {-3}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> f = lambda sp: SetPartition(da.to_set_partition(sp))
>>> f([[Integer(1),-Integer(1)],[Integer(2),-Integer(2)]]) == SetPartition(da.
↪identity_set_partition(Integer(2)))
True
>>> da.to_set_partition([[Integer(1)]])
[{1}, {-1}]
>>> da.to_set_partition([[Integer(1),-Integer(1)],[-Integer(2),Integer(3)]],
↪Integer(9)/Integer(2))
[{-1, 1}, {-2, 3}, {2}, {-4, 4}, {-5, 5}, {-3}]
```

5.1.99 Exact Cover Problem via Dancing Links

sage.combinat.dlx.**AllExactCovers**(M)

Use A. Ajanki's DLXMatrix class to solve the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: # needs sage.modules
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])  # no exact covers
sage: for cover in AllExactCovers(M):
....:     print(cover)
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]]) # two exact covers
sage: for cover in AllExactCovers(M):
....:     print(cover)
[(1, 1, 0), (0, 0, 1)]
[(1, 0, 1), (0, 1, 0)]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> M = Matrix([[Integer(1),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(1)],[Integer(0),Integer(1),Integer(1)]])  # no exact covers
>>> for cover in AllExactCovers(M):
...     print(cover)
>>> M = Matrix([[Integer(1),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(1)],[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(1),
↪Integer(0)]]) # two exact covers
>>> for cover in AllExactCovers(M):
```

(continues on next page)

(continued from previous page)

```
...     print(cover)
[(1, 1, 0), (0, 0, 1)]
[(1, 0, 1), (0, 1, 0)]
```

class sage.combinat.dlx.**DLXMatrix**(*ones*, *initialsolution=None*)

Bases: object

Solve the Exact Cover problem by using the Dancing Links algorithm described by Knuth.

Consider a matrix M with entries of 0 and 1, and compute a subset of the rows of this matrix which sum to the vector of all 1s.

The dancing links algorithm works particularly well for sparse matrices, so the input is a list of lists of the form: (note the 1-index!):

```
[
 [1, [i_11,i_12,...,i_1r]]
 ...
 [m,[i_m1,i_m2,...,i_ms]]
]
```

where M[j][i_jk] = 1.

The first example below corresponds to the matrix:

```
1110
1010
0100
0001
```

which is exactly covered by:

```
1110
0001
```

and

```
1010
0100
0001
```

EXAMPLES:

```
sage: from sage.combinat.dlx import *
sage: ones = [[1,[1,2,3]]]
sage: ones+= [[2,[1,3]]]
sage: ones+= [[3,[2]]]
sage: ones+= [[4,[4]]]
sage: DLXM = DLXMatrix(ones,[4])
sage: for C in DLXM:
....:     print(C)
[4, 1]
[4, 2, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.dlx import *
>>> ones = [[Integer(1),[Integer(1),Integer(2),Integer(3)]]]
>>> ones+= [[Integer(2),[Integer(1),Integer(3)]]]
>>> ones+= [[Integer(3),[Integer(2)]]]
>>> ones+= [[Integer(4),[Integer(4)]]]
>>> DLXM = DLXMatrix(ones,[Integer(4)])
>>> for C in DLXM:
...     print(C)
[4, 1]
[4, 2, 3]
```

Note

The 0 entry is reserved internally for headers in the sparse representation, so rows and columns begin their indexing with 1. Apologies for any heartache this causes. Blame the original author, or fix it yourself.

next()

Search for the first solution we can find, and return it.

Knuth describes the Dancing Links algorithm recursively, though actually implementing it as a recursive algorithm is permissible only for highly restricted problems. (for example, the original author implemented this for Sudoku, and it works beautifully there)

What follows is an iterative description of DLX:

```
stack <- [(NULL)]
level <- 0
while level >= 0:
    cur <- stack[level]
    if cur = NULL:
        if R[h] = h:
            level <- level - 1
            yield solution
        else:
            cover(best_column)
            stack[level] = best_column
    else if D[cur] != C[cur]:
        if cur != C[cur]:
            delete solution[level]
            for j in L[cur], L[L[cur]], ... , while j != cur:
                uncover(C[j])
        cur <- D[cur]
        solution[level] <- cur
        stack[level] <- cur
        for j in R[cur], R[R[cur]], ... , while j != cur:
            cover(C[j])
        level <- level + 1
        stack[level] <- (NULL)
    else:
        if C[cur] != cur:
            delete solution[level]
```

(continues on next page)

(continued from previous page)

```
            for j in L[cur], L[L[cur]], ... , while j != cur:
                uncover(C[j])
        uncover(cur)
        level <- level - 1
```

sage.combinat.dlx.**OneExactCover**(*M*)

Use A. Ajanki's DLXMatrix class to solve the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])  # no exact covers              #␣
↪needs sage.modules
sage: OneExactCover(M)                                                       #␣
↪needs sage.modules

sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]])  # two exact covers      #␣
↪needs sage.modules
sage: OneExactCover(M)                                                       #␣
↪needs sage.modules
[(1, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> M = Matrix([[Integer(1),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(1)],[Integer(0),Integer(1),Integer(1)]])  # no exact covers          ␣
↪      # needs sage.modules
>>> OneExactCover(M)                                                         #␣
↪needs sage.modules

>>> M = Matrix([[Integer(1),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(1)],[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(1),
↪Integer(0)]])  # two exact covers         # needs sage.modules
>>> OneExactCover(M)                                                         #␣
↪needs sage.modules
[(1, 1, 0), (0, 0, 1)]
```

5.1.100 Dyck Words

A class of an object enumerated by the `Catalan numbers`, see [Sta-EC2], [StaCat98] for details.

AUTHORS:

- Mike Hansen
- Dan Drake (2008-05-30): DyckWordBacktracker support
- Florent Hivert (2009-02-01): Bijections with NonDecreasingParkingFunctions
- Christian Stump (2011-12): added combinatorial maps and statistics
- Mike Zabrocki:
  - (2012-10): added pretty print, characteristic function, more functions
  - (2013-01): added inverse of area/dinv, bounce/area map
- Jean–Baptiste Priez, Travis Scrimshaw (2013-05-17): Added ASCII art

- Travis Scrimshaw (2013-07-09): Removed `CombinatorialClass` and added global options.

REFERENCES:

class sage.combinat.dyck_word.**CompleteDyckWords**

Bases: *DyckWords*

Abstract base class for all complete Dyck words.

Element

alias of *DyckWord_complete*

from_Catalan_code(*code*)

Return the Dyck word associated to the given Catalan code `code`.

A Catalan code of length n is a sequence $(a_1, a_2, \ldots, a_n)$ of n integers a_i such that:

- $0 \leq a_i \leq n - i$ for every i;
- if $i < j$ and $a_i > 0$ and $a_j > 0$ and $a_{i+1} = a_{i+2} = \cdots = a_{j-1} = 0$, then $a_i - a_j < j - i$.

It turns out that the Catalan codes of length n are in bijection with Dyck words.

The Catalan code of a Dyck word is example (x) in Richard Stanley's exercises on combinatorial interpretations for Catalan objects. The code in this example is the reverse of the description provided there. See [Sta-EC2] and [StaCat98].

EXAMPLES:

```
sage: DyckWords().from_Catalan_code([])
[]
sage: DyckWords().from_Catalan_code([0])
[1, 0]
sage: DyckWords().from_Catalan_code([0, 1])
[1, 1, 0, 0]
sage: DyckWords().from_Catalan_code([0, 0])
[1, 0, 1, 0]
```

```
>>> from sage.all import *
>>> DyckWords().from_Catalan_code([])
[]
>>> DyckWords().from_Catalan_code([Integer(0)])
[1, 0]
>>> DyckWords().from_Catalan_code([Integer(0), Integer(1)])
[1, 1, 0, 0]
>>> DyckWords().from_Catalan_code([Integer(0), Integer(0)])
[1, 0, 1, 0]
```

from_area_sequence(*code*)

Return the Dyck word associated to the given area sequence `code`.

See *to_area_sequence()* for a definition of the area sequence of a Dyck word.

See also

area(), *to_area_sequence()*.

INPUT:

- `code` – list of integers satisfying `code[0] == 0` and `0 <= code[i+1] <= code[i]+1`

EXAMPLES:

```
sage: DyckWords().from_area_sequence([])
[]
sage: DyckWords().from_area_sequence([0])
[1, 0]
sage: DyckWords().from_area_sequence([0, 1])
[1, 1, 0, 0]
sage: DyckWords().from_area_sequence([0, 0])
[1, 0, 1, 0]
```

```
>>> from sage.all import *
>>> DyckWords().from_area_sequence([])
[]
>>> DyckWords().from_area_sequence([Integer(0)])
[1, 0]
>>> DyckWords().from_area_sequence([Integer(0), Integer(1)])
[1, 1, 0, 0]
>>> DyckWords().from_area_sequence([Integer(0), Integer(0)])
[1, 0, 1, 0]
```

from_non_decreasing_parking_function(*pf*)

Bijection from *non-decreasing parking functions*.

See there the method *to_dyck_word()* for more information.

EXAMPLES:

```
sage: D = DyckWords()
sage: D.from_non_decreasing_parking_function([])
[]
sage: D.from_non_decreasing_parking_function([1])
[1, 0]
sage: D.from_non_decreasing_parking_function([1,1])
[1, 1, 0, 0]
sage: D.from_non_decreasing_parking_function([1,2])
[1, 0, 1, 0]
sage: D.from_non_decreasing_parking_function([1,1,1])
[1, 1, 1, 0, 0, 0]
sage: D.from_non_decreasing_parking_function([1,2,3])
[1, 0, 1, 0, 1, 0]
sage: D.from_non_decreasing_parking_function([1,1,3,3,4,6,6])
[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> D = DyckWords()
>>> D.from_non_decreasing_parking_function([])
[]
>>> D.from_non_decreasing_parking_function([Integer(1)])
[1, 0]
>>> D.from_non_decreasing_parking_function([Integer(1),Integer(1)])
[1, 1, 0, 0]
```

(continues on next page)

(continued from previous page)

```
>>> D.from_non_decreasing_parking_function([Integer(1),Integer(2)])
[1, 0, 1, 0]
>>> D.from_non_decreasing_parking_function([Integer(1),Integer(1),Integer(1)])
[1, 1, 1, 0, 0, 0]
>>> D.from_non_decreasing_parking_function([Integer(1),Integer(2),Integer(3)])
[1, 0, 1, 0, 1, 0]
>>> D.from_non_decreasing_parking_function([Integer(1),Integer(1),Integer(3),
↪Integer(3),Integer(4),Integer(6),Integer(6)])
[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0]
```

from_noncrossing_partition(*ncp*)

Convert a noncrossing partition `ncp` to a Dyck word.

EXAMPLES:

```
sage: DyckWord(noncrossing_partition=[[1,2]]) # indirect doctest
[1, 1, 0, 0]
sage: DyckWord(noncrossing_partition=[[1],[2]])
[1, 0, 1, 0]

sage: dws = DyckWords(5).list()
sage: ncps = [x.to_noncrossing_partition() for x in dws]
sage: dws2 = [DyckWord(noncrossing_partition=x) for x in ncps]
sage: dws == dws2
True
```

```
>>> from sage.all import *
>>> DyckWord(noncrossing_partition=[[Integer(1),Integer(2)]]) # indirect␣
↪doctest
[1, 1, 0, 0]
>>> DyckWord(noncrossing_partition=[[Integer(1)],[Integer(2)]])
[1, 0, 1, 0]

>>> dws = DyckWords(Integer(5)).list()
>>> ncps = [x.to_noncrossing_partition() for x in dws]
>>> dws2 = [DyckWord(noncrossing_partition=x) for x in ncps]
>>> dws == dws2
True
```

class sage.combinat.dyck_word.**CompleteDyckWords_all**

Bases: *CompleteDyckWords*, *DyckWords_all*

All complete Dyck words.

class height_poset

Bases: `UniqueRepresentation`, `Parent`

The poset of complete Dyck words compared componentwise by `heights`.

This is, `D` is smaller than or equal to `D'` if it is weakly below `D'`.

This is implemented by comparison of area sequences.

le(*dw1*, *dw2*)

Compare two Dyck words of equal size, and return `True` if all of the heights of `dw1` are less than or equal to the respective heights of `dw2` .

See also

to_area_sequence()

EXAMPLES:

```
sage: poset = DyckWords().height_poset()
sage: poset.le(DyckWord([]), DyckWord([]))
True
sage: poset.le(DyckWord([1,0]), DyckWord([1,0]))
True
sage: poset.le(DyckWord([1,0,1,0]), DyckWord([1,1,0,0]))
True
sage: poset.le(DyckWord([1,1,0,0]), DyckWord([1,0,1,0]))
False
sage: [poset.le(dw1, dw2)
....:     for dw1 in DyckWords(3) for dw2 in DyckWords(3)]
[True, True, True, True, True, False, True, False, True, True,
False, False, True, True, True, False, False, False, True,
True, False, False, False, False, True]
```

```
>>> from sage.all import *
>>> poset = DyckWords().height_poset()
>>> poset.le(DyckWord([]), DyckWord([]))
True
>>> poset.le(DyckWord([Integer(1),Integer(0)]), DyckWord([Integer(1),
→Integer(0)]))
True
>>> poset.le(DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]),␣
→DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]))
True
>>> poset.le(DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]),␣
→DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]))
False
>>> [poset.le(dw1, dw2)
...     for dw1 in DyckWords(Integer(3)) for dw2 in DyckWords(Integer(3))]
[True, True, True, True, True, False, True, False, True, True,
False, False, True, True, True, False, False, False, True,
True, False, False, False, False, True]
```

class sage.combinat.dyck_word.**CompleteDyckWords_size**(*k*)

Bases: *CompleteDyckWords*, *DyckWords_size*

All complete Dyck words of a given size.

cardinality()

Return the number of complete Dyck words of semilength n, i.e. the n-th *Catalan number*.

EXAMPLES:

```
sage: DyckWords(4).cardinality()
14
sage: ns = list(range(9))
sage: dws = [DyckWords(n) for n in ns]
```

(continues on next page)

(continued from previous page)

```
sage: all(dw.cardinality() == len(dw.list()) for dw in dws)
True
```

```
>>> from sage.all import *
>>> DyckWords(Integer(4)).cardinality()
14
>>> ns = list(range(Integer(9)))
>>> dws = [DyckWords(n) for n in ns]
>>> all(dw.cardinality() == len(dw.list()) for dw in dws)
True
```

random_element()

Return a random complete Dyck word of semilength n.

The algorithm is based on a classical combinatorial fact. One chooses at random a word with n 0s and $n+1$ 1s. One then considers every 1 as an ascending step and every 0 as a descending step, and one finds the lowest point of the path (with respect to a slightly tilted slope). One then cuts the path at this point and builds a Dyck word by exchanging the two parts of the word and removing the initial step.

Todo

extend this to m-Dyck words

EXAMPLES:

```
sage: dw = DyckWords(8)
sage: dw.random_element()  # random
[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0]

sage: D = DyckWords(8)
sage: D.random_element() in D
True
```

```
>>> from sage.all import *
>>> dw = DyckWords(Integer(8))
>>> dw.random_element()  # random
[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0]

>>> D = DyckWords(Integer(8))
>>> D.random_element() in D
True
```

class sage.combinat.dyck_word.**DyckWord**(*parent*, *l*, *latex_options={}*)

Bases: *CombinatorialElement*

A Dyck word.

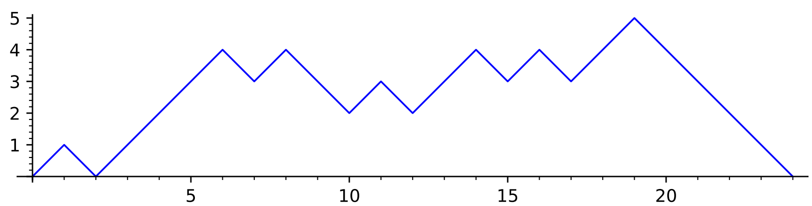
A Dyck word is a sequence of open and close symbols such that every close symbol has a corresponding open symbol preceding it. That is to say, a Dyck word of length n is a list with k entries 1 and $n-k$ entries 0 such that the first i entries always have at least as many 1s among them as 0s. (Here, the 1 serves as the open symbol and the 0 as the close symbol.) Alternatively, the alphabet 1 and 0 can be replaced by other characters such as '(' and ')'.

A Dyck word is *complete* if every open symbol moreover has a corresponding close symbol.

A Dyck word may also be specified by either a noncrossing partition or by an area sequence or the sequence of heights.

A Dyck word may also be thought of as a lattice path in the $\mathbf{Z}^2$ grid, starting at the origin $(0, 0)$, and with steps in the North $N = (0, 1)$ and east $E = (1, 0)$ directions such that it does not pass below the $x = y$ diagonal. The diagonal is referred to as the "main diagonal" in the documentation. A North step is represented by a 1 in the list and an East step is represented by a 0.

Equivalently, the path may be represented with steps in the $NE = (1, 1)$ and the $SE = (1, -1)$ direction such that it does not pass below the horizontal axis.



A path representing a Dyck word (either using N and E steps, or using NE and SE steps) is called a Dyck path.

EXAMPLES:

```
sage: dw = DyckWord([1, 0, 1, 0]); dw
[1, 0, 1, 0]
sage: print(dw)
()()
sage: dw.height()
1
sage: dw.to_noncrossing_partition()
{{1}, {2}}
```

```
>>> from sage.all import *
>>> dw = DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]); dw
[1, 0, 1, 0]
>>> print(dw)
()()
>>> dw.height()
1
>>> dw.to_noncrossing_partition()
{{1}, {2}}
```

```
sage: DyckWord('()()')
[1, 0, 1, 0]
sage: DyckWord('(())')
[1, 1, 0, 0]
sage: DyckWord('((')
[1, 1]
sage: DyckWord('')
[]
```

```
>>> from sage.all import *
>>> DyckWord('()()')
[1, 0, 1, 0]
>>> DyckWord('(())')
[1, 1, 0, 0]
>>> DyckWord('((')
[1, 1]
>>> DyckWord('')
[]
```

```
sage: DyckWord(noncrossing_partition=[[1],[2]])
[1, 0, 1, 0]
sage: DyckWord(noncrossing_partition=[[1,2]])
[1, 1, 0, 0]
sage: DyckWord(noncrossing_partition=[])
[]
```

```
>>> from sage.all import *
>>> DyckWord(noncrossing_partition=[[Integer(1)],[Integer(2)]])
[1, 0, 1, 0]
>>> DyckWord(noncrossing_partition=[[Integer(1),Integer(2)]])
[1, 1, 0, 0]
>>> DyckWord(noncrossing_partition=[])
[]
```

```
sage: DyckWord(area_sequence=[0,0])
[1, 0, 1, 0]
sage: DyckWord(area_sequence=[0,1])
[1, 1, 0, 0]
sage: DyckWord(area_sequence=[0,1,2,2,0,1,1,2])
[1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
sage: DyckWord(area_sequence=[])
[]
```

```
>>> from sage.all import *
>>> DyckWord(area_sequence=[Integer(0),Integer(0)])
[1, 0, 1, 0]
>>> DyckWord(area_sequence=[Integer(0),Integer(1)])
[1, 1, 0, 0]
>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(2),Integer(2),
↪Integer(0),Integer(1),Integer(1),Integer(2)])
[1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
```

(continues on next page)

(continued from previous page)

```
>>> DyckWord(area_sequence=[])
[]
```

```
sage: DyckWord(heights_sequence=(0,1,0,1,0))
[1, 0, 1, 0]
sage: DyckWord(heights_sequence=(0,1,2,1,0))
[1, 1, 0, 0]
sage: DyckWord(heights_sequence=(0,))
[]
```

```
>>> from sage.all import *
>>> DyckWord(heights_sequence=(Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)))
[1, 0, 1, 0]
>>> DyckWord(heights_sequence=(Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(0)))
[1, 1, 0, 0]
>>> DyckWord(heights_sequence=(Integer(0),))
[]
```

```
sage: print(DyckWord([1,0,1,1,0,0]).to_path_string())
   /\
/\/  \
sage: DyckWord([1,0,1,1,0,0]).pretty_print()
   ___
  | x
 _|  .
|  . .
```

```
>>> from sage.all import *
>>> print(DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0)]).to_path_string())
   /\
/\/  \
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),Integer(0)]).
↪pretty_print()
   ___
  | x
 _|  .
|  . .
```

ascent_prime_decomposition()

Decompose this Dyck word into a sequence of ascents and prime Dyck paths.

A Dyck word is *prime* if it is complete and has precisely one return - the final step. In particular, the empty Dyck path is not prime. Thus, the factorization is unique.

This decomposition yields a sequence of odd length: the words with even indices consist of up steps only, the words with odd indices are prime Dyck paths. The concatenation of the result is the original word.

EXAMPLES:

```
sage: D = DyckWord([1,1,1,0,1,0,1,1,1,1,0,1])
sage: D.ascent_prime_decomposition()
[[1, 1], [1, 0], [], [1, 0], [1, 1, 1], [1, 0], [1]]

sage: DyckWord([]).ascent_prime_decomposition()
[[]]

sage: DyckWord([1,1]).ascent_prime_decomposition()
[[1, 1]]

sage: DyckWord([1,0,1,0]).ascent_prime_decomposition()
[[], [1, 0], [], [1, 0], []]
```

```
>>> from sage.all import *
>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(1)])
>>> D.ascent_prime_decomposition()
[[1, 1], [1, 0], [], [1, 0], [1, 1, 1], [1, 0], [1]]

>>> DyckWord([]).ascent_prime_decomposition()
[[]]

>>> DyckWord([Integer(1),Integer(1)]).ascent_prime_decomposition()
[[1, 1]]

>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).ascent_prime_
↪decomposition()
[[], [1, 0], [], [1, 0], []]
```

associated_parenthesis(*pos*)

Report the position for the parenthesis in `self` that matches the one at position `pos`.

The positions in `self` are counted from 0.

INPUT:

- `pos` – the index of the parenthesis in the list

OUTPUT:

- Integer representing the index of the matching parenthesis. If no parenthesis matches, return `None`.

EXAMPLES:

```
sage: DyckWord([1, 0]).associated_parenthesis(0)
1
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(0)
1
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(1)
0
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(2)
3
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(3)
2
```

(continues on next page)

(continued from previous page)

```
sage: DyckWord([1, 1, 0, 0]).associated_parenthesis(0)
3
sage: DyckWord([1, 1, 0, 0]).associated_parenthesis(2)
1
sage: DyckWord([1, 1, 0]).associated_parenthesis(1)
2
sage: DyckWord([1, 1]).associated_parenthesis(0)
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0)]).associated_parenthesis(Integer(0))
1
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).associated_
↪parenthesis(Integer(0))
1
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).associated_
↪parenthesis(Integer(1))
0
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).associated_
↪parenthesis(Integer(2))
3
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).associated_
↪parenthesis(Integer(3))
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).associated_
↪parenthesis(Integer(0))
3
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).associated_
↪parenthesis(Integer(2))
1
>>> DyckWord([Integer(1), Integer(1), Integer(0)]).associated_
↪parenthesis(Integer(1))
2
>>> DyckWord([Integer(1), Integer(1)]).associated_parenthesis(Integer(0))
```

catalan_factorization()

Decompose this Dyck word into a sequence of complete Dyck words.

Each element of the list returned is a (possibly empty) complete Dyck word. The original word is obtained by placing an up step between each of these complete Dyck words. Thus, the number of words returned is one more than the final height.

See Section 1.2 of [CC1982] or Lemma 9.1.1 of [Lot2005].

EXAMPLES:

```
sage: D = DyckWord([1,1,1,0,1,0,1,1,1,1,0,1])
sage: D.catalan_factorization()
[[], [], [1, 0, 1, 0], [], [], [1, 0], []]

sage: DyckWord([]).catalan_factorization()
[[]]

sage: DyckWord([1,1]).catalan_factorization()
```

(continues on next page)

(continued from previous page)

```
[[], [], []]

sage: DyckWord([1,0,1,0]).catalan_factorization()
[[1, 0, 1, 0]]
```

```
>>> from sage.all import *
>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(1)])
>>> D.catalan_factorization()
[[], [], [1, 0, 1, 0], [], [], [1, 0], []]

>>> DyckWord([]).catalan_factorization()
[[]]

>>> DyckWord([Integer(1),Integer(1)]).catalan_factorization()
[[], [], []]

>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).catalan_
↪factorization()
[[1, 0, 1, 0]]
```

height()

Return the height of `self`.

We view the Dyck word as a Dyck path from $(0, 0)$ to $(2n, 0)$ in the first quadrant by letting `1`'s represent steps in the direction $(1, 1)$ and `0`'s represent steps in the direction $(1, -1)$.

The height is the maximum y-coordinate reached.

See also

heights()

EXAMPLES:

```
sage: DyckWord([]).height()
0
sage: DyckWord([1,0]).height()
1
sage: DyckWord([1, 1, 0, 0]).height()
2
sage: DyckWord([1, 1, 0, 1, 0]).height()
2
sage: DyckWord([1, 1, 0, 0, 1, 0]).height()
2
sage: DyckWord([1, 0, 1, 0]).height()
1
sage: DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]).height()
3
```

```
>>> from sage.all import *
>>> DyckWord([]).height()
0
>>> DyckWord([Integer(1),Integer(0)]).height()
1
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).height()
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(0)]).
↪height()
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]).height()
2
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).height()
1
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)]).height()
3
```

heights()

Return the heights of `self`.

We view the Dyck word as a Dyck path from $(0,0)$ to $(2n,0)$ in the first quadrant by letting `1`'s represent steps in the direction $(1,1)$ and `0`'s represent steps in the direction $(1,-1)$.

The heights is the sequence of the y-coordinates of all $2n+1$ lattice points along the path.

See also

from_heights(), *min_from_heights()*

EXAMPLES:

```
sage: DyckWord([]).heights()
(0,)
sage: DyckWord([1,0]).heights()
(0, 1, 0)
sage: DyckWord([1, 1, 0, 0]).heights()
(0, 1, 2, 1, 0)
sage: DyckWord([1, 1, 0, 1, 0]).heights()
(0, 1, 2, 1, 2, 1)
sage: DyckWord([1, 1, 0, 0, 1, 0]).heights()
(0, 1, 2, 1, 0, 1, 0)
sage: DyckWord([1, 0, 1, 0]).heights()
(0, 1, 0, 1, 0)
sage: DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]).heights()
(0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0)
```

```
>>> from sage.all import *
>>> DyckWord([]).heights()
(0,)
>>> DyckWord([Integer(1),Integer(0)]).heights()
```

(continues on next page)

(continued from previous page)

```
(0, 1, 0)
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).heights()
(0, 1, 2, 1, 0)
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(0)]).
↪heights()
(0, 1, 2, 1, 2, 1)
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]).heights()
(0, 1, 2, 1, 0, 1, 0)
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).heights()
(0, 1, 0, 1, 0)
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)]).heights()
(0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0)
```

is_complete()

Return `True` if `self` is complete.

A Dyck word d is complete if d contains as many closers as openers.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).is_complete()
True
sage: DyckWord([1, 0, 1, 1, 0]).is_complete()
False
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).is_complete()
True
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0)]).is_
↪complete()
False
```

latex_options()

Return the latex options for use in the `_latex_` function as a dictionary.

The default values are set using the options.

- `tikz_scale` – (default: 1) scale for use with the tikz package
- `diagonal` – boolean (default: `False`); value to draw the diagonal or not
- `line width` – (default: `2*tikz_scale`) value representing the line width
- `color` – (default: black) the line color
- `bounce path` – boolean (default: `False`); value to indicate if the bounce path should be drawn
- `peaks` – boolean (default: `False`); value to indicate if the peaks should be displayed
- `valleys` – boolean (default: `False`); value to indicate if the valleys should be displayed

EXAMPLES:

```
sage: D = DyckWord([1,0,1,0,1,0])
sage: D.latex_options()
```

(continues on next page)

(continued from previous page)

```
{'bounce path': False,
 'color': black,
 'diagonal': False,
 'line width': 2,
 'peaks': False,
 'tikz_scale': 1,
 'valleys': False}
```

```
>>> from sage.all import *
>>> D = DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)])
>>> D.latex_options()
{'bounce path': False,
 'color': black,
 'diagonal': False,
 'line width': 2,
 'peaks': False,
 'tikz_scale': 1,
 'valleys': False}
```

Todo

This should probably be merged into DyckWord.options.

length()

Return the length of `self`.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).length()
4
sage: DyckWord([1, 0, 1, 1, 0]).length()
5
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).length()
4
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0)]).
↪length()
5
```

number_of_close_symbols()

Return the number of close symbols in `self`.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_close_symbols()
2
sage: DyckWord([1, 0, 1, 1, 0]).number_of_close_symbols()
2
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).number_of_
↪close_symbols()
2
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0)]).
↪number_of_close_symbols()
2
```

number_of_double_rises()

Return a the number of positions in `self` where there are two consecutive 1's.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]).number_of_double_
↪rises()
2
sage: DyckWord([1, 1, 0, 0]).number_of_double_rises()
1
sage: DyckWord([1, 0, 1, 0]).number_of_double_rises()
0
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(1), Integer(0),
↪Integer(0), Integer(1), Integer(0)]).number_of_double_rises()
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).number_of_
↪double_rises()
1
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).number_of_
↪double_rises()
0
```

number_of_initial_rises()

Return the length of the initial run of `self`.

OUTPUT: nonnegative integer indicating the length of the initial rise

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_initial_rises()
1
sage: DyckWord([1, 1, 0, 0]).number_of_initial_rises()
2
sage: DyckWord([1, 1, 0, 0, 1, 0]).number_of_initial_rises()
2
sage: DyckWord([1, 0, 1, 1, 0, 0]).number_of_initial_rises()
1
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).number_of_
↪initial_rises()
1
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).number_of_
```

(continues on next page)

(continued from previous page)

```
↪initial_rises()
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]).number_of_initial_rises()
2
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)]).number_of_initial_rises()
1
```

number_of_open_symbols()

Return the number of open symbols in `self`.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_open_symbols()
2
sage: DyckWord([1, 0, 1, 1, 0]).number_of_open_symbols()
3
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).number_of_open_
↪symbols()
2
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0)]).
↪number_of_open_symbols()
3
```

number_of_peaks()

Return the number of peaks of the Dyck path associated to `self` .

See also

peaks(), *number_of_valleys()*

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_peaks()
2
sage: DyckWord([1, 1, 0, 0]).number_of_peaks()
1
sage: DyckWord([1,1,0,1,0,1,0,0]).number_of_peaks()
3
sage: DyckWord([]).number_of_peaks()
0
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).number_of_
↪peaks()
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).number_of_
↪peaks()
```

(continues on next page)

(continued from previous page)

```
1
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).number_of_peaks()
3
>>> DyckWord([]).number_of_peaks()
0
```

number_of_touch_points()

Return the number of touches of `self` at the main diagonal.

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_touch_points()
2
sage: DyckWord([1, 1, 0, 0]).number_of_touch_points()
1
sage: DyckWord([1, 1, 0, 0, 1, 0]).number_of_touch_points()
2
sage: DyckWord([1, 0, 1, 1, 0, 0]).number_of_touch_points()
2
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).number_of_
↪touch_points()
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).number_of_
↪touch_points()
1
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]).number_of_touch_points()
2
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)]).number_of_touch_points()
2
```

number_of_valleys()

Return the number of valleys of `self`.

> **See also**
>
> *number_of_peaks()*, *valleys()*

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_valleys()
1
sage: DyckWord([1, 1, 0, 0]).number_of_valleys()
0
sage: DyckWord([1, 1, 0, 0, 1, 0]).number_of_valleys()
1
```

(continues on next page)

(continued from previous page)

```
sage: DyckWord([1, 0, 1, 1, 0, 0]).number_of_valleys()
1
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).number_of_
↪valleys()
1
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).number_of_
↪valleys()
0
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]).number_of_valleys()
1
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)]).number_of_valleys()
1
```

peaks()

Return a list of the positions of the peaks of a Dyck word.

A peak is 1 followed by a 0. Note that this does not agree with the definition given in [Hag2008].

See also

valleys(), *number_of_peaks()*

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).peaks()
[0, 2]
sage: DyckWord([1, 1, 0, 0]).peaks()
[1]
sage: DyckWord([1,1,0,1,0,1,0,0]).peaks() # Haglund's def gives 2
[1, 3, 5]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).peaks()
[0, 2]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).peaks()
[1]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).peaks() # Haglund's def gives 2
[1, 3, 5]
```

plot(***kwds*)

Plot a Dyck word as a continuous path.

EXAMPLES:

```
sage: w = DyckWords(100).random_element()
sage: w.plot()                                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> w = DyckWords(Integer(100)).random_element()
>>> w.plot()                                                            #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

position_of_first_return()

Return the number of vertical steps before the Dyck path returns to the main diagonal.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]).position_of_first_
↪return()
1
sage: DyckWord([1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0]).position_of_first_
↪return()
7
sage: DyckWord([1, 1, 0, 0]).position_of_first_return()
2
sage: DyckWord([1, 0, 1, 0]).position_of_first_return()
1
sage: DyckWord([]).position_of_first_return()
0
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0)]).position_of_first_return()
1
>>> DyckWord([Integer(1), Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(0), Integer(0)]).position_of_first_return()
7
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).position_of_
↪first_return()
2
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).position_of_
↪first_return()
1
>>> DyckWord([]).position_of_first_return()
0
```

positions_of_double_rises()

Return a list of positions in `self` where there are two consecutive 1's.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]).positions_of_
↪double_rises()
```

(continues on next page)

(continued from previous page)

```
[2, 5]
sage: DyckWord([1, 1, 0, 0]).positions_of_double_rises()
[0]
sage: DyckWord([1, 0, 1, 0]).positions_of_double_rises()
[]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0)]).positions_of_double_rises()
[2, 5]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).positions_of_
↪double_rises()
[0]
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).positions_of_
↪double_rises()
[]
```

pp(*type=None*, *labelling=None*, *underpath=True*)

Display a DyckWord as a lattice path in the $\mathbf{Z}^2$ grid.

If the `type` is “N-E”, then a cell below the diagonal is indicated by a period, whereas a cell below the path but above the diagonal is indicated by an x. If a list of labels is included, they are displayed along the vertical edges of the Dyck path.

If the `type` is “NE-SE”, then the path is simply printed as up steps and down steps.

INPUT:

- `type` – (default: `None`) can either be:
  - `None` to use the option default
  - “N-E” to show `self` as a path of north and east steps, or
  - “NE-SE” to show `self` as a path of north-east and south-east steps.
- `labelling` – (if type is “N-E”) a list of labels assigned to the up steps in `self`
- `underpath` – (if type is “N-E”, default: `True`) if `True`, the labelling is shown under the path; otherwise, it is shown to the right of the path

EXAMPLES:

```
sage: for D in DyckWords(3): D.pretty_print()
     _
   _|
 _|  .
|  . .
   ___
  | x
 _|  .
|  . .
     _
 ___|
| x  .
|  . .
```

(continues on next page)

(continued from previous page)

```
   ___
 _| x
| x  .
|  . .
 _____
| x x
| x  .
|  . .
```

```
>>> from sage.all import *
>>> for D in DyckWords(Integer(3)): D.pretty_print()
     _
   _|
 _|  .
|  . .
   ___
  | x
 _|  .
|  . .
     _
 ___|
| x  .
|  . .
   ___
 _| x
| x  .
|  . .
 _____
| x x
| x  .
|  . .
```

```
sage: for D in DyckWords(3): D.pretty_print(type="NE-SE")
/\/\/\
   /\
/\/  \
 /\
/  \/\
 /\/\
/    \
  /\
 /  \
/    \
```

```
>>> from sage.all import *
>>> for D in DyckWords(Integer(3)): D.pretty_print(type="NE-SE")
/\/\/\
   /\
/\/  \
 /\
/  \/\
```

(continues on next page)

(continued from previous page)

```
 /\/\
/    \
  /\
 /  \
/    \
```

```
sage: D = DyckWord([1,1,1,0,1,0,0,1,1])
sage: D.pretty_print()
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: D = DyckWord([1,1,1,0,1,0,0,1,1,0])
sage: D.pretty_print()
       _
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: D = DyckWord([1,1,1,0,1,0,0,1,1,0,0])
sage: D.pretty_print()
       ___
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .
```

```
>>> from sage.all import *
>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1)])
>>> D.pretty_print()
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1),Integer(0)])
>>> D.pretty_print()
       _
      | x x
```

(continues on next page)

(continued from previous page)

```
   ___| x   .
 _| x x   . .
| x x   . . .
| x   . . . .
|   . . . . .

>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),Integer(0)])
>>> D.pretty_print()
       ___
      | x x
   ___| x   .
 _| x x   . .
| x x   . . .
| x   . . . .
|   . . . . .
```

```
sage: DyckWord(area_sequence=[0,1,0]).pretty_print(labelling=[1,3,2])
     _
 ___|2
|3x  .
|1 . .

sage: DyckWord(area_sequence=[0,1,0]).pretty_print(labelling=[1,3,2],
↪underpath=False)
     _
 ___|  2
| x  . 3
|  . . 1
```

```
>>> from sage.all import *
>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(0)]).pretty_
↪print(labelling=[Integer(1),Integer(3),Integer(2)])
     _
 ___|2
|3x  .
|1 . .

>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(0)]).pretty_
↪print(labelling=[Integer(1),Integer(3),Integer(2)],underpath=False)
     _
 ___|  2
| x  . 3
|  . . 1
```

```
sage: DyckWord(area_sequence=[0,1,1,2,3,2,3,3,2,0,1,1,2,3,4,2,3]).pretty_
↪print()
                           _______
                          | x x x
                     _____| x x   .
                    | x x x x   . .
```

(continues on next page)

(continued from previous page)

```
                    | x x x  . . .
                    | x x  . . . .
                   _| x  . . . . .
                  | x  . . . . . .
             _____|   . . . . . . .
         ___| x x  . . . . . . . .
       _| x x x  . . . . . . . . .
      | x x x  . . . . . . . . . .
   ___| x x  . . . . . . . . . . .
  | x x x  . . . . . . . . . . . .
  | x x  . . . . . . . . . . . . .
 _| x  . . . . . . . . . . . . . .
| x  . . . . . . . . . . . . . . .
|   . . . . . . . . . . . . . . . .

sage: DyckWord(area_sequence=[0,1,1,2,3,2,3,3,2,0,1,1,2,3,4,2,3]).pretty_
↪print(labelling=list(range(17)),underpath=False)
                           _______
                          | x x x  16
                     _____| x x  . 15
                    | x x x x  . . 14
                    | x x x  . . . 13
                    | x x  . . . . 12
                   _| x  . . . . . 11
                  | x  . . . . . . 10
             _____|   . . . . . . .   9
         ___| x x  . . . . . . . .   8
       _| x x x  . . . . . . . . .   7
      | x x x  . . . . . . . . . .   6
   ___| x x  . . . . . . . . . . .   5
  | x x x  . . . . . . . . . . . .   4
  | x x  . . . . . . . . . . . . .   3
 _| x  . . . . . . . . . . . . . .   2
| x  . . . . . . . . . . . . . . .   1
|   . . . . . . . . . . . . . . . .   0
```

```
>>> from sage.all import *
>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(1),Integer(2),
↪Integer(3),Integer(2),Integer(3),Integer(3),Integer(2),Integer(0),
↪Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(2),
↪Integer(3)]).pretty_print()
                           _______
                          | x x x
                     _____| x x  .
                    | x x x x  . .
                    | x x x  . . .
                    | x x  . . . .
                   _| x  . . . . .
                  | x  . . . . . .
             _____|   . . . . . . .
         ___| x x  . . . . . . . .
       _| x x x  . . . . . . . . .
```

(continues on next page)

(continued from previous page)

```
      | x x x  . . . . . . . . . .
   ___| x x  . . . . . . . . . . .
  | x x x  . . . . . . . . . . . .
  | x x  . . . . . . . . . . . . .
 _| x  . . . . . . . . . . . . . .
| x  . . . . . . . . . . . . . . .
|  . . . . . . . . . . . . . . . .

>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(1),Integer(2),
↪Integer(3),Integer(2),Integer(3),Integer(3),Integer(2),Integer(0),
↪Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(2),
↪Integer(3)]).pretty_print(labelling=list(range(Integer(17))),
↪underpath=False)
                           _______
                          | x x x  16
                     _____| x x  . 15
                    | x x x x  . . 14
                    | x x x  . . . 13
                    | x x  . . . . 12
                   _| x  . . . . . 11
                  | x  . . . . . . 10
             _____|  . . . . . . .   9
         ___| x x  . . . . . . . .   8
       _| x x x  . . . . . . . . .   7
      | x x x  . . . . . . . . . .   6
   ___| x x  . . . . . . . . . . .   5
  | x x x  . . . . . . . . . . . .   4
  | x x  . . . . . . . . . . . . .   3
 _| x  . . . . . . . . . . . . . .   2
| x  . . . . . . . . . . . . . . .   1
|  . . . . . . . . . . . . . . . .   0
```

```
sage: DyckWord([]).pretty_print()
.
```

```
>>> from sage.all import *
>>> DyckWord([]).pretty_print()
.
```

pretty_print(*type=None*, *labelling=None*, *underpath=True*)

Display a DyckWord as a lattice path in the $\mathbf{Z}^2$ grid.

If the `type` is "N-E", then a cell below the diagonal is indicated by a period, whereas a cell below the path but above the diagonal is indicated by an x. If a list of labels is included, they are displayed along the vertical edges of the Dyck path.

If the `type` is "NE-SE", then the path is simply printed as up steps and down steps.

INPUT:

- `type` – (default: `None`) can either be:
  - `None` to use the option default
  - "N-E" to show `self` as a path of north and east steps, or

    – "NE-SE" to show `self` as a path of north-east and south-east steps.

- `labelling` – (if type is "N-E") a list of labels assigned to the up steps in `self`
- `underpath` – (if type is "N-E", default: `True`) if `True`, the labelling is shown under the path; otherwise, it is shown to the right of the path

EXAMPLES:

```
sage: for D in DyckWords(3): D.pretty_print()
     _
   _|
 _|  .
|  . .
   ___
  | x
 _|  .
|  . .
     _
 ___|
| x  .
|  . .
   ___
 _| x
| x  .
|  . .
 _____
| x x
| x  .
|  . .
```

```
>>> from sage.all import *
>>> for D in DyckWords(Integer(3)): D.pretty_print()
     _
   _|
 _|  .
|  . .
   ___
  | x
 _|  .
|  . .
     _
 ___|
| x  .
|  . .
   ___
 _| x
| x  .
|  . .
 _____
| x x
| x  .
|  . .
```

```
sage: for D in DyckWords(3): D.pretty_print(type="NE-SE")
/\/\/\
   /\
/\/  \
 /\
/  \/\
 /\/\
/    \
  /\
 /  \
/    \
```

```
>>> from sage.all import *
>>> for D in DyckWords(Integer(3)): D.pretty_print(type="NE-SE")
/\/\/\
   /\
/\/  \
 /\
/  \/\
 /\/\
/    \
  /\
 /  \
/    \
```

```
sage: D = DyckWord([1,1,1,0,1,0,0,1,1])
sage: D.pretty_print()
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: D = DyckWord([1,1,1,0,1,0,0,1,1,0])
sage: D.pretty_print()
       _
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: D = DyckWord([1,1,1,0,1,0,0,1,1,0,0])
sage: D.pretty_print()
       ___
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
```

(continues on next page)

(continued from previous page)

```
|  . . . . .
```

```
>>> from sage.all import *
>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1)])
>>> D.pretty_print()
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1),Integer(0)])
>>> D.pretty_print()
       _
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),Integer(0)])
>>> D.pretty_print()
       ___
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .
```

```
sage: DyckWord(area_sequence=[0,1,0]).pretty_print(labelling=[1,3,2])
     _
 ___|2
|3x  .
|1 . .

sage: DyckWord(area_sequence=[0,1,0]).pretty_print(labelling=[1,3,2],
↪underpath=False)
     _
 ___|  2
| x  . 3
|  . . 1
```

```
>>> from sage.all import *
>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(0)]).pretty_
↪print(labelling=[Integer(1),Integer(3),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
     _
 ___|2
|3x  .
|1 . .

>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(0)]).pretty_
↪print(labelling=[Integer(1),Integer(3),Integer(2)],underpath=False)
     _
 ___|  2
| x  . 3
|  . . 1
```

```
sage: DyckWord(area_sequence=[0,1,1,2,3,2,3,3,2,0,1,1,2,3,4,2,3]).pretty_
↪print()
                           _______
                          | x x x
                     _____| x x  .
                    | x x x x  . .
                    | x x x  . . .
                    | x x  . . . .
                   _| x  . . . . .
                  | x  . . . . . .
             _____|  . . . . . . .
         ___| x x  . . . . . . . .
       _| x x x  . . . . . . . . .
      | x x x  . . . . . . . . . .
   ___| x x  . . . . . . . . . . .
  | x x x  . . . . . . . . . . . .
  | x x  . . . . . . . . . . . . .
 _| x  . . . . . . . . . . . . . .
| x  . . . . . . . . . . . . . . .
|  . . . . . . . . . . . . . . . .

sage: DyckWord(area_sequence=[0,1,1,2,3,2,3,3,2,0,1,1,2,3,4,2,3]).pretty_
↪print(labelling=list(range(17)),underpath=False)
                           _______
                          | x x x  16
                     _____| x x  . 15
                    | x x x x  . . 14
                    | x x x  . . . 13
                    | x x  . . . . 12
                   _| x  . . . . . 11
                  | x  . . . . . . 10
             _____|  . . . . . . .  9
         ___| x x  . . . . . . . .  8
       _| x x x  . . . . . . . . .  7
      | x x x  . . . . . . . . . .  6
   ___| x x  . . . . . . . . . . .  5
  | x x x  . . . . . . . . . . . .  4
  | x x  . . . . . . . . . . . . .  3
 _| x  . . . . . . . . . . . . . .  2
| x  . . . . . . . . . . . . . . .  1
```

(continues on next page)

(continued from previous page)

```
|  . . . . . . . . . . . . . . . .  0
```

```
>>> from sage.all import *
>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(1),Integer(2),
↪Integer(3),Integer(2),Integer(3),Integer(3),Integer(2),Integer(0),
↪Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(2),
↪Integer(3)]).pretty_print()
                           _______
                          | x x x
                     _____| x x  .
                    | x x x x  . .
                    | x x x  . . .
                    | x x  . . . .
                   _| x  . . . . .
                  | x  . . . . . .
             _____|  . . . . . . .
         ___| x x  . . . . . . . .
       _| x x x  . . . . . . . . .
      | x x x  . . . . . . . . . .
   ___| x x  . . . . . . . . . . .
  | x x x  . . . . . . . . . . . .
  | x x  . . . . . . . . . . . . .
 _| x  . . . . . . . . . . . . . .
| x  . . . . . . . . . . . . . . .
|  . . . . . . . . . . . . . . . .

>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(1),Integer(2),
↪Integer(3),Integer(2),Integer(3),Integer(3),Integer(2),Integer(0),
↪Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(2),
↪Integer(3)]).pretty_print(labelling=list(range(Integer(17))),
↪underpath=False)
                           _______
                          | x x x  16
                     _____| x x  . 15
                    | x x x x  . . 14
                    | x x x  . . . 13
                    | x x  . . . . 12
                   _| x  . . . . . 11
                  | x  . . . . . . 10
             _____|  . . . . . . .  9
         ___| x x  . . . . . . . .  8
       _| x x x  . . . . . . . . .  7
      | x x x  . . . . . . . . . .  6
   ___| x x  . . . . . . . . . . .  5
  | x x x  . . . . . . . . . . . .  4
  | x x  . . . . . . . . . . . . .  3
 _| x  . . . . . . . . . . . . . .  2
| x  . . . . . . . . . . . . . . .  1
|  . . . . . . . . . . . . . . . .  0
```

```
sage: DyckWord([]).pretty_print()
.
```

```
>>> from sage.all import *
>>> DyckWord([]).pretty_print()
.
```

returns_to_zero()

Return a list of positions where `self` has height 0, excluding the position 0.

EXAMPLES:

```
sage: DyckWord([]).returns_to_zero()
[]
sage: DyckWord([1, 0]).returns_to_zero()
[2]
sage: DyckWord([1, 0, 1, 0]).returns_to_zero()
[2, 4]
sage: DyckWord([1, 1, 0, 0]).returns_to_zero()
[4]
```

```
>>> from sage.all import *
>>> DyckWord([]).returns_to_zero()
[]
>>> DyckWord([Integer(1), Integer(0)]).returns_to_zero()
[2]
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).returns_to_
↪zero()
[2, 4]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).returns_to_
↪zero()
[4]
```

rise_composition()

The sequences of lengths of runs of 1's in `self`. Also equal to the sequence of lengths of vertical segments in the Dyck path.

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).pretty_print()
           ___
          | x
   _______|  .
  | x x x  . .
  | x x  . . .
 _| x  . . . .
| x  . . . . .
|  . . . . . .

sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).rise_composition()
[2, 3, 2]
sage: DyckWord([1,1,0,0]).rise_composition()
[2]
sage: DyckWord([1,0,1,0]).rise_composition()
[1, 1]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0)]).pretty_print()
           ___
          | x
   _______|  .
  | x x x  . .
  | x x  . . .
 _| x  . . . .
| x  . . . . .
|  . . . . . .

>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0)]).rise_composition()
[2, 3, 2]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).rise_composition()
[2]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).rise_composition()
[1, 1]
```

set_latex_options(*D*)

Set the latex options for use in the `_latex_` function.

The default values are set in the `__init__` function.

- `tikz_scale` – (default: 1) scale for use with the tikz package
- `diagonal` – boolean (default: `False`); value to draw the diagonal or not
- `line width` – (default: `2*tikz_scale`) value representing the line width
- `color` – (default: black) the line color
- `bounce path` – boolean (default: `False`); value to indicate if the bounce path should be drawn
- `peaks` – boolean (default: `False`); value to indicate if the peaks should be displayed
- `valleys` – boolean (default: `False`); value to indicate if the valleys should be displayed

INPUT:

- `D` – dictionary with a list of latex parameters to change

EXAMPLES:

```
sage: D = DyckWord([1,0,1,0,1,0])
sage: D.set_latex_options({"tikz_scale":2})
sage: D.set_latex_options({"valleys":True, "color":"blue"})
```

```
>>> from sage.all import *
>>> D = DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)])
>>> D.set_latex_options({"tikz_scale":Integer(2)})
>>> D.set_latex_options({"valleys":True, "color":"blue"})
```

Todo

This should probably be merged into DyckWord.options.

tamari_interval(*other*)

Return the Tamari interval between `self` and `other` as a *TamariIntervalPoset*.

A "Tamari interval" means an interval in the Tamari order. The Tamari order on the set of Dyck words of size n is the partial order obtained from the Tamari order on the set of binary trees of size n (see `tamari_lequal()`) by means of the Tamari bijection between Dyck words and binary trees (`to_dyck_word_tamari()`).

INPUT:

- `other` – a Dyck word greater or equal to `self` in the Tamari order

EXAMPLES:

```
sage: # needs sage.graphs
sage: dw = DyckWord([1, 1, 0, 1, 0, 0, 1, 0])
sage: ip = dw.tamari_interval(DyckWord([1, 1, 1, 0, 0, 1, 0, 0])); ip
The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (3, 1),␣
↪(2, 1)]
sage: ip.lower_dyck_word()
[1, 1, 0, 1, 0, 0, 1, 0]
sage: ip.upper_dyck_word()
[1, 1, 1, 0, 0, 1, 0, 0]
sage: ip.interval_cardinality()
4
sage: ip.number_of_tamari_inversions()
2
sage: list(ip.dyck_words())
[[1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 0, 1, 0, 0, 1, 0]]
sage: dw.tamari_interval(DyckWord([1,1,0,0,1,1,0,0]))
Traceback (most recent call last):
...
ValueError: the two Dyck words are not comparable on the Tamari lattice
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> dw = DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),
↪ Integer(0), Integer(1), Integer(0)])
>>> ip = dw.tamari_interval(DyckWord([Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(0), Integer(1), Integer(0), Integer(0)])); ip
The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (3, 1),␣
↪(2, 1)]
>>> ip.lower_dyck_word()
[1, 1, 0, 1, 0, 0, 1, 0]
>>> ip.upper_dyck_word()
[1, 1, 1, 0, 0, 1, 0, 0]
>>> ip.interval_cardinality()
```

(continues on next page)

(continued from previous page)

```
4
>>> ip.number_of_tamari_inversions()
2
>>> list(ip.dyck_words())
[[1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 0, 1, 0, 0, 1, 0]]
>>> dw.tamari_interval(DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(1),Integer(0),Integer(0)]))
Traceback (most recent call last):
...
ValueError: the two Dyck words are not comparable on the Tamari lattice
```

to_area_sequence()

Return the area sequence of the Dyck word `self`.

The area sequence of a Dyck word w is defined as follows: Representing the Dyck word w as a Dyck path from $(0, 0)$ to (n, n) using N and E steps (this involves padding w by E steps until w reaches the main diagonal if w is not already a complete Dyck path), the area sequence of w is the sequence $(a_1, a_2, \ldots, a_n)$, where a_i is the number of full cells in the i-th row of the rectangle $[0, n] \times [0, n]$ which lie completely above the diagonal. (The cells are the regions into which the rectangle is subdivided by the lines $x = i$ with i integer and the lines $y = j$ with j integer. The i-th row consists of all the cells between the lines $y = i - 1$ and $y = i$.)

An alternative definition: Representing the Dyck word w as a Dyck path consisting of NE and SE steps, the area sequence is the sequence of ordinates of all lattice points on the path which are starting points of NE steps.

A list of integers l is the area sequence of some Dyck path if and only if it satisfies $l_0 = 0$ and $0 \leq l_{i+1} \leq l_i + 1$ for $i > 0$.

EXAMPLES:

```
sage: DyckWord([]).to_area_sequence()
[]
sage: DyckWord([1, 0]).to_area_sequence()
[0]
sage: DyckWord([1, 1, 0, 0]).to_area_sequence()
[0, 1]
sage: DyckWord([1, 0, 1, 0]).to_area_sequence()
[0, 0]
sage: all(dw ==
....:     DyckWords().from_area_sequence(dw.to_area_sequence())
....:     for i in range(6) for dw in DyckWords(i))
True
sage: DyckWord([1,0,1,0,1,0,1,0,1,0]).to_area_sequence()
[0, 0, 0, 0, 0]
sage: DyckWord([1,1,1,1,1,0,0,0,0,0]).to_area_sequence()
[0, 1, 2, 3, 4]
sage: DyckWord([1,1,1,1,0,1,0,0,0,0]).to_area_sequence()
[0, 1, 2, 3, 3]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_area_sequence()
[0, 1, 1, 0, 1, 1, 1]
```

```
>>> from sage.all import *
>>> DyckWord([]).to_area_sequence()
[]
>>> DyckWord([Integer(1), Integer(0)]).to_area_sequence()
[0]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).to_area_
↪sequence()
[0, 1]
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).to_area_
↪sequence()
[0, 0]
>>> all(dw ==
...     DyckWords().from_area_sequence(dw.to_area_sequence())
...     for i in range(Integer(6)) for dw in DyckWords(i))
True
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(0),Integer(1),Integer(0)]).to_area_sequence()
[0, 0, 0, 0, 0]
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)]).to_area_sequence()
[0, 1, 2, 3, 4]
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)]).to_area_sequence()
[0, 1, 2, 3, 3]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).to_area_sequence()
[0, 1, 1, 0, 1, 1, 1]
```

to_binary_tree(*usemap='1L0R'*)

Return a binary tree recursively constructed from the Dyck path `self` by the map `usemap`. The default `usemap` is `'1L0R'` which means:

- an empty Dyck word is a leaf,
- a non empty Dyck word reads $1L0R$ where L and R correspond to respectively its left and right subtrees.

INPUT:

- `usemap` – string, either `'1L0R'`, `'1R0L'`, `'L1R0'`, `'R1L0'`

Other valid `usemap` are `'1R0L'`, `'L1R0'`, and `'R1L0'`. These correspond to different maps from Dyck paths to binary trees, whose recursive definitions are hopefully clear from the names.

EXAMPLES:

```
sage: # needs sage.graphs
sage: dw = DyckWord([1,0])
sage: dw.to_binary_tree()
[., .]
sage: dw = DyckWord([])
sage: dw.to_binary_tree()
.
sage: dw = DyckWord([1,0,1,1,0,0])
sage: dw.to_binary_tree()
[., [[., .], .]]
```

(continues on next page)

(continued from previous page)

```
sage: dw.to_binary_tree("L1R0")
[[., .], [., .]]
sage: dw = DyckWord([1,0,1,1,0,0,1,1,1,0,1,0,0,0])
sage: dw.to_binary_tree() == dw.to_binary_tree("1R0L").left_right_symmetry()
True
sage: dw.to_binary_tree() == dw.to_binary_tree("L1R0").left_border_symmetry()
False
sage: dw.to_binary_tree("1R0L") == dw.to_binary_tree("L1R0").left_border_
↪symmetry()
True
sage: dw.to_binary_tree("R1L0") == dw.to_binary_tree("L1R0").left_right_
↪symmetry()
True
sage: dw.to_binary_tree("R10L")
Traceback (most recent call last):
...
ValueError: R10L is not a correct map
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> dw = DyckWord([Integer(1),Integer(0)])
>>> dw.to_binary_tree()
[., .]
>>> dw = DyckWord([])
>>> dw.to_binary_tree()
.
>>> dw = DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0)])
>>> dw.to_binary_tree()
[., [[., .], .]]
>>> dw.to_binary_tree("L1R0")
[[., .], [., .]]
>>> dw = DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)])
>>> dw.to_binary_tree() == dw.to_binary_tree("1R0L").left_right_symmetry()
True
>>> dw.to_binary_tree() == dw.to_binary_tree("L1R0").left_border_symmetry()
False
>>> dw.to_binary_tree("1R0L") == dw.to_binary_tree("L1R0").left_border_
↪symmetry()
True
>>> dw.to_binary_tree("R1L0") == dw.to_binary_tree("L1R0").left_right_
↪symmetry()
True
>>> dw.to_binary_tree("R10L")
Traceback (most recent call last):
...
ValueError: R10L is not a correct map
```

to_binary_tree_tamari()

Return the binary tree corresponding to `self` in a way which is consistent with the Tamari orders on the set of Dyck paths and on the set of binary trees.

This is the 'L1R0' map documented in *to_binary_tree()*.

EXAMPLES:

```
sage: DyckWord([1,0]).to_binary_tree_tamari()                                #␣
↪needs sage.graphs
[., .]
sage: DyckWord([1,0,1,1,0,0]).to_binary_tree_tamari()                        #␣
↪needs sage.graphs
[[., .], [., .]]
sage: DyckWord([1,0,1,0,1,0]).to_binary_tree_tamari()                        #␣
↪needs sage.graphs
[[[., .], .], .]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(0)]).to_binary_tree_tamari()                ␣
↪               # needs sage.graphs
[., .]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0)]).to_binary_tree_tamari()                         # needs sage.
↪graphs
[[., .], [., .]]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)]).to_binary_tree_tamari()                         # needs sage.
↪graphs
[[[., .], .], .]
```

to_path_string(*unicode=False*)

Return a path representation of the Dyck word consisting of steps / and \ .

INPUT:

- unicode – boolean (default: False); whether to use unicode

EXAMPLES:

```
sage: print(DyckWord([1, 0, 1, 0]).to_path_string())
/\/\
sage: print(DyckWord([1, 1, 0, 0]).to_path_string())
 /\
/  \
sage: print(DyckWord([1,1,0,1,1,0,0,1,0,1,0,0]).to_path_string())
    /\
 /\/  \/\/\
/          \
```

```
>>> from sage.all import *
>>> print(DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).to_path_
↪string())
/\/\
>>> print(DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).to_path_
↪string())
 /\
```

(continues on next page)

(continued from previous page)

```
/   \
>>> print(DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0)]).to_path_string())
    /\
 /\/  \/\/\
/          \
```

to_standard_tableau()

Return a standard tableau of shape (a, b) where a is the number of open symbols and b is the number of close symbols in `self`.

EXAMPLES:

```
sage: DyckWord([]).to_standard_tableau()
[]
sage: DyckWord([1, 0]).to_standard_tableau()
[[1], [2]]
sage: DyckWord([1, 1, 0, 0]).to_standard_tableau()
[[1, 2], [3, 4]]
sage: DyckWord([1, 0, 1, 0]).to_standard_tableau()
[[1, 3], [2, 4]]
sage: DyckWord([1]).to_standard_tableau()
[[1]]
sage: DyckWord([1, 0, 1]).to_standard_tableau()
[[1, 3], [2]]
```

```
>>> from sage.all import *
>>> DyckWord([]).to_standard_tableau()
[]
>>> DyckWord([Integer(1), Integer(0)]).to_standard_tableau()
[[1], [2]]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).to_standard_
↪tableau()
[[1, 2], [3, 4]]
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).to_standard_
↪tableau()
[[1, 3], [2, 4]]
>>> DyckWord([Integer(1)]).to_standard_tableau()
[[1]]
>>> DyckWord([Integer(1), Integer(0), Integer(1)]).to_standard_tableau()
[[1, 3], [2]]
```

to_tamari_sorting_tuple()

Convert a Dyck word to a Tamari sorting tuple.

The result is a list of integers, one for every up-step from left to right. To each up-step is associated the distance to the corresponding down step in the Dyck word.

This is useful for a faster conversion to binary trees.

EXAMPLES:

```
sage: DyckWord([]).to_tamari_sorting_tuple()
[]
sage: DyckWord([1, 0]).to_tamari_sorting_tuple()
[0]
sage: DyckWord([1, 1, 0, 0]).to_tamari_sorting_tuple()
[1, 0]
sage: DyckWord([1, 0, 1, 0]).to_tamari_sorting_tuple()
[0, 0]
sage: DyckWord([1, 1, 0, 1, 0, 0]).to_tamari_sorting_tuple()
[2, 0, 0]
```

```
>>> from sage.all import *
>>> DyckWord([]).to_tamari_sorting_tuple()
[]
>>> DyckWord([Integer(1), Integer(0)]).to_tamari_sorting_tuple()
[0]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).to_tamari_
↪sorting_tuple()
[1, 0]
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).to_tamari_
↪sorting_tuple()
[0, 0]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(0)]).to_tamari_sorting_tuple()
[2, 0, 0]
```

See also

to_Catalan_code()

touch_composition()

Return a composition which indicates the positions where `self` returns to the diagonal.

This assumes `self` to be a complete Dyck word.

OUTPUT: a composition of length equal to the length of the Dyck word

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).touch_composition()
[1, 1]
sage: DyckWord([1, 1, 0, 0]).touch_composition()
[2]
sage: DyckWord([1, 1, 0, 0, 1, 0]).touch_composition()
[2, 1]
sage: DyckWord([1, 0, 1, 1, 0, 0]).touch_composition()
[1, 2]
sage: DyckWord([]).touch_composition()
[]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).touch_
```

(continues on next page)

(continued from previous page)

```
↪composition()
[1, 1]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).touch_
↪composition()
[2]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]).touch_composition()
[2, 1]
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)]).touch_composition()
[1, 2]
>>> DyckWord([]).touch_composition()
[]
```

touch_points()

Return the abscissae (or, equivalently, ordinates) of the points where the Dyck path corresponding to `self` (comprising NE and SE steps) touches the main diagonal. This includes the last point (if it is on the main diagonal) but excludes the beginning point.

Note that these abscissae are precisely the entries of *returns_to_zero()* divided by 2.

OUTPUT: list of integers indicating where the path touches the diagonal

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).touch_points()
[1, 2]
sage: DyckWord([1, 1, 0, 0]).touch_points()
[2]
sage: DyckWord([1, 1, 0, 0, 1, 0]).touch_points()
[2, 3]
sage: DyckWord([1, 0, 1, 1, 0, 0]).touch_points()
[1, 3]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).touch_points()
[1, 2]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).touch_points()
[2]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]).touch_points()
[2, 3]
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)]).touch_points()
[1, 3]
```

valleys()

Return a list of the positions of the valleys of a Dyck word.

A valley is 0 followed by a 1.

See also

peaks(), *number_of_valleys()*

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).valleys()
[1]
sage: DyckWord([1, 1, 0, 0]).valleys()
[]
sage: DyckWord([1,1,0,1,0,1,0,0]).valleys()
[2, 4]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).valleys()
[1]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).valleys()
[]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).valleys()
[2, 4]
```

class sage.combinat.dyck_word.**DyckWordBacktracker**(*k1*, *k2*)

Bases: *GenericBacktracker*

This class is an iterator for all Dyck words with n opening parentheses and $n - k$ closing parentheses using the backtracker class. It is used by the *DyckWords_size* class.

This is not really meant to be called directly, partially because it fails in a couple corner cases: `DWB(0)` yields `[0]`, not the empty word, and `DWB(k, k+1)` yields something (it shouldn't yield anything). This could be fixed with a sanity check in `_rec()`, but then we'd be doing the sanity check *every time* we generate new objects; instead, we do *one* sanity check in *DyckWords* and assume here that the sanity check has already been made.

AUTHOR:

- Dan Drake (2008-05-30)

class sage.combinat.dyck_word.**DyckWord_complete**(*parent*, *l*, *latex_options={}*)

Bases: *DyckWord*

The class of complete *Dyck words*. A Dyck word is complete, if it contains as many closers as openers.

For further information on Dyck words, see *DyckWords_class*.

area()

Return the area for `self` corresponding to the area of the Dyck path.

One can view a balanced Dyck word as a lattice path from $(0, 0)$ to (n, n) in the first quadrant by letting '1's represent steps in the direction $(1, 0)$ and '0's represent steps in the direction $(0, 1)$. The resulting path will remain weakly above the diagonal $y = x$.

The area statistic is the number of complete squares in the integer lattice which are below the path and above the line $y = x$. The 'half-squares' directly above the line $y = x$ do not contribute to this statistic.

EXAMPLES:

```
sage: dw = DyckWord([1,0,1,0])
sage: dw.area() # 2 half-squares, 0 complete squares
0
```

```
>>> from sage.all import *
>>> dw = DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)])
>>> dw.area() # 2 half-squares, 0 complete squares
0
```

```
sage: dw = DyckWord([1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,0])
sage: dw.area()
19
```

```
>>> from sage.all import *
>>> dw = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0)])
>>> dw.area()
19
```

```
sage: DyckWord([1,1,1,1,0,0,0,0]).area()
6
sage: DyckWord([1,1,1,0,1,0,0,0]).area()
5
sage: DyckWord([1,1,1,0,0,1,0,0]).area()
4
sage: DyckWord([1,1,1,0,0,0,1,0]).area()
3
sage: DyckWord([1,0,1,1,0,1,0,0]).area()
2
sage: DyckWord([1,1,0,1,1,0,0,0]).area()
4
sage: DyckWord([1,1,0,0,1,1,0,0]).area()
2
sage: DyckWord([1,0,1,1,1,0,0,0]).area()
3
sage: DyckWord([1,0,1,1,0,0,1,0]).area()
1
sage: DyckWord([1,0,1,0,1,1,0,0]).area()
1
sage: DyckWord([1,1,0,0,1,0,1,0]).area()
1
sage: DyckWord([1,1,0,1,0,0,1,0]).area()
2
sage: DyckWord([1,1,0,1,0,1,0,0]).area()
3
sage: DyckWord([1,0,1,0,1,0,1,0]).area()
0
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(0),Integer(0)]).area()
6
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).area()
```

(continues on next page)

(continued from previous page)

```
5
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area()
4
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(1),Integer(0)]).area()
3
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area()
2
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).area()
4
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0)]).area()
2
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).area()
3
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(0)]).area()
1
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0)]).area()
1
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(0)]).area()
1
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(0)]).area()
2
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area()
3
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(0)]).area()
0
```

area_dinv_to_bounce_area_map()

Return the image of `self` under the map which sends a Dyck word with `area` equal to r and `dinv` equal to s to a Dyck word with `bounce` equal to r and `area` equal to s .

The inverse of this map is *bounce_area_to_area_dinv_map()*.

For a definition of this map, see [Hag2008] p. 50 where it is called ζ. However, this map differs from Haglund's map by an application of *reverse()* (as does the definition of the *bounce()* statistic).

EXAMPLES:

```
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area_dinv_to_bounce_area_map()
[1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area()
5
```

(continues on next page)

(continued from previous page)

```
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).dinv()
13
sage: DyckWord([1,1,1,1,1,0,0,0,1,0,0,1,0,0]).area()
13
sage: DyckWord([1,1,1,1,1,0,0,0,1,0,0,1,0,0]).bounce()
5
sage: DyckWord([1,1,1,1,1,0,0,0,1,0,0,1,0,0]).area_dinv_to_bounce_area_map()
[1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]
sage: DyckWord([1,1,0,0]).area_dinv_to_bounce_area_map()
[1, 0, 1, 0]
sage: DyckWord([1,0,1,0]).area_dinv_to_bounce_area_map()
[1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area_dinv_to_bounce_area_map()
[1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area()
5
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).dinv()
13
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area()
13
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).bounce()
5
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area_dinv_to_bounce_area_map()
[1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).area_dinv_to_
↪bounce_area_map()
[1, 0, 1, 0]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).area_dinv_to_
↪bounce_area_map()
[1, 1, 0, 0]
```

bounce()

Return the bounce statistic of `self` due to J. Haglund, see [Hag2008].

One can view a balanced Dyck word as a lattice path from $(0,0)$ to (n,n) in the first quadrant by letting '1's represent steps in the direction $(0,1)$ and '0's represent steps in the direction $(1,0)$. The resulting path will remain weakly above the diagonal $y=x$.

We describe the bounce statistic of such a path in terms of what is known as the "bounce path".

We can think of our bounce path as describing the trail of a billiard ball shot West from (n, n), which "bounces" down whenever it encounters a vertical step and "bounces" left when it encounters the line $y = x$.

The bouncing ball will strike the diagonal at the places

$$(0, 0), (j_1, j_1), (j_2, j_2), \ldots, (j_r - 1, j_r - 1), (j_r, j_r) = (n, n).$$

We define the bounce to be the sum $\sum_{i=1}^{r-1} j_i$.

EXAMPLES:

```
sage: DyckWord([1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,0]).bounce()
7
sage: DyckWord([1,1,1,1,0,0,0,0]).bounce()
0
sage: DyckWord([1,1,1,0,1,0,0,0]).bounce()
1
sage: DyckWord([1,1,1,0,0,1,0,0]).bounce()
2
sage: DyckWord([1,1,1,0,0,0,1,0]).bounce()
3
sage: DyckWord([1,0,1,1,0,1,0,0]).bounce()
3
sage: DyckWord([1,1,0,1,1,0,0,0]).bounce()
1
sage: DyckWord([1,1,0,0,1,1,0,0]).bounce()
2
sage: DyckWord([1,0,1,1,1,0,0,0]).bounce()
1
sage: DyckWord([1,0,1,1,0,0,1,0]).bounce()
4
sage: DyckWord([1,0,1,0,1,1,0,0]).bounce()
3
sage: DyckWord([1,1,0,0,1,0,1,0]).bounce()
5
sage: DyckWord([1,1,0,1,0,0,1,0]).bounce()
4
sage: DyckWord([1,1,0,1,0,1,0,0]).bounce()
2
sage: DyckWord([1,0,1,0,1,0,1,0]).bounce()
6
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0)]).bounce()
7
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(0),Integer(0)]).bounce()
0
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).bounce()
1
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(0),Integer(0)]).bounce()
2
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(1),Integer(0)]).bounce()
3
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).bounce()
3
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).bounce()
1
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0)]).bounce()
2
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).bounce()
1
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(0)]).bounce()
4
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0)]).bounce()
3
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(0)]).bounce()
5
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(0)]).bounce()
4
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).bounce()
2
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(0)]).bounce()
6
```

bounce_area_to_area_dinv_map()

Return the image of the Dyck word under the map which sends a Dyck word with `bounce` equal to r and `area` equal to s to a Dyck word with `area` equal to r and `dinv` equal to s .

This implementation uses a recursive method by saying that the last entry in the area sequence of the Dyck word `self` is equal to the number of touch points of the Dyck path minus 1 of the image of this map.

The inverse of this map is *area_dinv_to_bounce_area_map()*.

For a definition of this map, see [Hag2008] p. 50 where it is called ζ^{-1}. However, this map differs from Haglund's map by an application of *reverse()* (as does the definition of the *bounce()* statistic).

EXAMPLES:

```
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).bounce_area_to_area_dinv_map()
[1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area()
5
```

(continues on next page)

(continued from previous page)

```
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).bounce()
9
sage: DyckWord([1,1,0,0,1,1,1,1,0,0,1,0,0,0]).area()
9
sage: DyckWord([1,1,0,0,1,1,1,1,0,0,1,0,0,0]).dinv()
5
sage: all(D==D.bounce_area_to_area_dinv_map().area_dinv_to_bounce_area_map()␣
↪for D in DyckWords(6))
True
sage: DyckWord([1,1,0,0]).bounce_area_to_area_dinv_map()
[1, 0, 1, 0]
sage: DyckWord([1,0,1,0]).bounce_area_to_area_dinv_map()
[1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).bounce_area_to_area_dinv_map()
[1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area()
5
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).bounce()
9
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).area()
9
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).dinv()
5
>>> all(D==D.bounce_area_to_area_dinv_map().area_dinv_to_bounce_area_map()␣
↪for D in DyckWords(Integer(6)))
True
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).bounce_area_to_
↪area_dinv_map()
[1, 0, 1, 0]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).bounce_area_to_
↪area_dinv_map()
[1, 1, 0, 0]
```

bounce_path()

Return the bounce path of `self` formed by starting at (n, n) and traveling West until encountering the first vertical step of `self`, then South until encountering the diagonal, then West again to hit the path, etc. until the $(0, 0)$ point is reached. The path followed by this walk is the bounce path.

See also

`bounce()`

EXAMPLES:

```
sage: DyckWord([1,1,0,1,0,0]).bounce_path()
[1, 0, 1, 1, 0, 0]
sage: DyckWord([1,1,1,0,0,0]).bounce_path()
[1, 1, 1, 0, 0, 0]
sage: DyckWord([1,0,1,0,1,0]).bounce_path()
[1, 0, 1, 0, 1, 0]
sage: DyckWord([1,1,1,1,0,0,1,0,0,0]).bounce_path()
[1, 1, 0, 0, 1, 1, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0)]).bounce_path()
[1, 0, 1, 1, 0, 0]
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0)]).bounce_path()
[1, 1, 1, 0, 0, 0]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)]).bounce_path()
[1, 0, 1, 0, 1, 0]
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(0),Integer(0),Integer(0)]).bounce_path()
[1, 1, 0, 0, 1, 1, 1, 0, 0, 0]
```

characteristic_symmetric_function(*q=None*, *R=Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field*)

The characteristic function of `self` is the sum of $q^{dinv(D,F)}Q_{ides(read(D,F))}$ over all permutation fillings of the Dyck path representing `self`, where $ides(read(D,F))$ is the descent composition of the inverse of the reading word of the filling.

INPUT:

- `q` – (default: `q = R('q')`) a parameter for the generating function power
- `R` – (default: `R = QQ['q','t'].fraction_field()`) the base ring to do the calculations over

OUTPUT:

- an element of the symmetric functions over the ring `R` (in the Schur basis).

EXAMPLES:

```
sage: R = QQ['q','t'].fraction_field()
sage: (q,t) = R.gens()
sage: f = sum(t**D.area() * D.characteristic_symmetric_function()            #␣
↪needs sage.modules
....:          for D in DyckWords(3)); f
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
sage: f.nabla(power=-1)                                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
s[1, 1, 1]
```

```
>>> from sage.all import *
>>> R = QQ['q','t'].fraction_field()
>>> (q,t) = R.gens()
>>> f = sum(t**D.area() * D.characteristic_symmetric_function()          #␣
↪needs sage.modules
...         for D in DyckWords(Integer(3))); f
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
>>> f.nabla(power=-Integer(1))                                              ␣
↪     # needs sage.modules
s[1, 1, 1]
```

decomposition_reverse()

Return the involution of `self` with a recursive definition.

If a Dyck word D decomposes as $1D_1 0D_2$ where D_1 and D_2 are complete Dyck words then the decomposition reverse is $1\phi(D_2)0\phi(D_1)$.

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0]).decomposition_
↪reverse()
[1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).decomposition_
↪reverse()
[1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0]
sage: DyckWord([1,1,0,0]).decomposition_reverse()
[1, 0, 1, 0]
sage: DyckWord([1,0,1,0]).decomposition_reverse()
[1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0)]).decomposition_reverse()
[1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0)]).decomposition_reverse()
[1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).decomposition_
↪reverse()
[1, 0, 1, 0]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).decomposition_
↪reverse()
[1, 1, 0, 0]
```

dinv(*labeling=None*)

Return the dinv statistic of `self` due to M. Haiman, see [Hag2008].

If a labeling is provided then this function returns the dinv of the labeled Dyck word.

INPUT:

- `labeling` – an optional argument to be viewed as the labelings of the vertical edges of the Dyck path

OUTPUT:

- an integer representing the `dinv` statistic of the Dyck path or the labelled Dyck path.

EXAMPLES:

```
sage: DyckWord([1,0,1,0,1,0,1,0,1,0]).dinv()
10
sage: DyckWord([1,1,1,1,1,0,0,0,0,0]).dinv()
0
sage: DyckWord([1,1,1,1,0,1,0,0,0,0]).dinv()
1
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).dinv()
13
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).dinv([1,2,3,4,5,6,7])
11
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).dinv([6,7,5,3,4,2,1])
2
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(0),Integer(1),Integer(0)]).dinv()
10
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)]).dinv()
0
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)]).dinv()
1
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).dinv()
13
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).dinv([Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(6),Integer(7)])
11
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).dinv([Integer(6),Integer(7),Integer(5),
↪Integer(3),Integer(4),Integer(2),Integer(1)])
2
```

first_return_decomposition()

Decompose a Dyck word into a pair of Dyck words (potentially empty) where the first word consists of the word after the first up step and the corresponding matching closing parenthesis.

EXAMPLES:

```
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).first_return_decomposition()
([1, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0, 0])
```

(continues on next page)

(continued from previous page)

```
sage: DyckWord([1,1,0,0]).first_return_decomposition()
([1, 0], [])
sage: DyckWord([1,0,1,0]).first_return_decomposition()
([], [1, 0])
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).first_return_decomposition()
([1, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0, 0])
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).first_return_
↪decomposition()
([1, 0], [])
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).first_return_
↪decomposition()
([], [1, 0])
```

list_parking_functions()

Return all parking functions whose supporting Dyck path is `self`.

EXAMPLES:

```
sage: DyckWord([1,1,0,0,1,0]).list_parking_functions()
[[1, 1, 3], [1, 3, 1], [3, 1, 1]]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)]).list_parking_functions()
[[1, 1, 3], [1, 3, 1], [3, 1, 1]]
```

major_index()

Return the major index of `self` .

The major index of a Dyck word D is the sum of the positions of the valleys of D (when started counting at position `1`).

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).major_index()
2
sage: DyckWord([1, 1, 0, 0]).major_index()
0
sage: DyckWord([1, 1, 0, 0, 1, 0]).major_index()
4
sage: DyckWord([1, 0, 1, 1, 0, 0]).major_index()
2
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).major_index()
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).major_index()
0
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)]).major_index()
4
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)]).major_index()
2
```

number_of_parking_functions()

Return the number of parking functions with `self` as the supporting Dyck path.

One representation of a parking function is as a pair consisting of a Dyck path and a permutation π such that if $[a_0, a_1, \ldots, a_{n-1}]$ is the area_sequence of the Dyck path (see `to_area_sequence`) then the permutation π satisfies $\pi_i < \pi_{i+1}$ whenever $a_i < a_{i+1}$. This function counts the number of permutations π which satisfy this condition.

EXAMPLES:

```
sage: DyckWord(area_sequence=[0,1,2]).number_of_parking_functions()
1
sage: DyckWord(area_sequence=[0,1,1]).number_of_parking_functions()
3
sage: DyckWord(area_sequence=[0,1,0]).number_of_parking_functions()
3
sage: DyckWord(area_sequence=[0,0,0]).number_of_parking_functions()
6
```

```
>>> from sage.all import *
>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(2)]).number_of_
↪parking_functions()
1
>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(1)]).number_of_
↪parking_functions()
3
>>> DyckWord(area_sequence=[Integer(0),Integer(1),Integer(0)]).number_of_
↪parking_functions()
3
>>> DyckWord(area_sequence=[Integer(0),Integer(0),Integer(0)]).number_of_
↪parking_functions()
6
```

number_of_tunnels(*tunnel_type='centered'*)

Return the number of tunnels of `self` of type `tunnel_type`.

A tunnel is a pair (a, b) where `a` is the position of an open parenthesis and `b` is the position of the matching close parenthesis. If $a + b = n$ then the tunnel is called *centered* . If $a + b < n$ then the tunnel is called *left* and if $a + b > n$, then the tunnel is called *right*.

INPUT:

- `tunnel_type` – (default: `'centered'`) can be one of the following: `'left'`, `'right'`, `'centered'`, or `'all'`

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).number_of_tunnels()
0
```

(continues on next page)

(continued from previous page)

```
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).number_of_tunnels(
↪'left')
5
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).number_of_tunnels(
↪'right')
2
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).number_of_tunnels(
↪'all')
7
sage: DyckWord([1, 1, 0, 0]).number_of_tunnels('centered')
2
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0)]).number_of_tunnels()
0
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0)]).number_of_tunnels('left')
5
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0)]).number_of_tunnels('right')
2
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0)]).number_of_tunnels('all')
7
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).number_of_
↪tunnels('centered')
2
```

parking_functions()

Iterate over parking functions whose supporting Dyck path is `self`.

EXAMPLES:

```
sage: list(DyckWord([1,1,0,1,0,0]).parking_functions())
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

```
>>> from sage.all import *
>>> list(DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0)]).parking_functions())
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

pyramid_weight()

Return the pyramid weight of `self`.

A pyramid of `self` is a subsequence of the form $1^h 0^h$. A pyramid is maximal if it is neither preceded by a 1 nor followed by a 0.

The pyramid weight of a Dyck path is the sum of the lengths of the maximal pyramids and was defined in [DS1992].

EXAMPLES:

```
sage: DyckWord([1,1,0,1,1,1,0,0,1,0,0,0,1,1,0,0]).pyramid_weight()
6
sage: DyckWord([1,1,1,0,0,0]).pyramid_weight()
3
sage: DyckWord([1,0,1,0,1,0]).pyramid_weight()
3
sage: DyckWord([1,1,0,1,0,0]).pyramid_weight()
2
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(1),
↪Integer(1),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(0)]).pyramid_weight()
6
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0)]).pyramid_weight()
3
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)]).pyramid_weight()
3
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0)]).pyramid_weight()
2
```

reading_permutation()

Return the reading permutation of `self`.

This is the permutation formed by taking the reading word of the Dyck path representing `self` (with N and E steps) if the vertical edges of the Dyck path are labeled from bottom to top with 1 through n and the diagonals are read from top to bottom starting with the diagonal furthest from the main diagonal.

EXAMPLES:

```
sage: DyckWord([1,0,1,0]).reading_permutation()
[2, 1]
sage: DyckWord([1,1,0,0]).reading_permutation()
[2, 1]
sage: DyckWord([1,1,0,1,0,0]).reading_permutation()
[3, 2, 1]
sage: DyckWord([1,1,0,0,1,0]).reading_permutation()
[2, 3, 1]
sage: DyckWord([1,0,1,1,0,0,1,0]).reading_permutation()
[3, 4, 2, 1]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).reading_
↪permutation()
[2, 1]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).reading_
↪permutation()
[2, 1]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)]).reading_permutation()
[3, 2, 1]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)]).reading_permutation()
[2, 3, 1]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(0)]).reading_permutation()
[3, 4, 2, 1]
```

reverse()

Return the reverse and complement of `self`.

This operation corresponds to flipping the Dyck path across the $y = -x$ line.

EXAMPLES:

```
sage: DyckWord([1,1,0,0,1,0]).reverse()
[1, 0, 1, 1, 0, 0]
sage: DyckWord([1,1,1,0,0,0]).reverse()
[1, 1, 1, 0, 0, 0]
sage: len([D for D in DyckWords(5) if D.reverse() == D])
10
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)]).reverse()
[1, 0, 1, 1, 0, 0]
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0)]).reverse()
[1, 1, 1, 0, 0, 0]
>>> len([D for D in DyckWords(Integer(5)) if D.reverse() == D])
10
```

semilength()

Return the semilength of `self`.

The semilength of a complete Dyck word d is the number of openers and the number of closers.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).semilength()
2
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).semilength()
2
```

to_132_avoiding_permutation()

Use the bijection by C. Krattenthaler in [Kra2001] to send `self` to a 132-avoiding permutation.

EXAMPLES:

```
sage: DyckWord([1,1,0,0]).to_132_avoiding_permutation()
[1, 2]
```

(continues on next page)

(continued from previous page)

```
sage: DyckWord([1,0,1,0]).to_132_avoiding_permutation()
[2, 1]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_132_avoiding_permutation()
[6, 5, 4, 7, 2, 1, 3]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).to_132_avoiding_
↪permutation()
[1, 2]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).to_132_avoiding_
↪permutation()
[2, 1]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).to_132_avoiding_permutation()
[6, 5, 4, 7, 2, 1, 3]
```

to_312_avoiding_permutation()

Convert `self` to a 312-avoiding permutation using the bijection by Bandlow and Killpatrick in [BK2001].

This sends the area to the inversion number.

EXAMPLES:

```
sage: DyckWord([1,1,0,0]).to_312_avoiding_permutation()
[2, 1]
sage: DyckWord([1,0,1,0]).to_312_avoiding_permutation()
[1, 2]
sage: p = DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_312_avoiding_
↪permutation(); p
[2, 3, 1, 5, 6, 7, 4]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area()
5
sage: p.length()
5
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).to_312_avoiding_
↪permutation()
[2, 1]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).to_312_avoiding_
↪permutation()
[1, 2]
>>> p = DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).to_312_avoiding_permutation(); p
[2, 3, 1, 5, 6, 7, 4]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area()
5
>>> p.length()
5
```

to_321_avoiding_permutation()

Use the bijection (pp. 60-61 of [Knu1973] or section 3.1 of [CK2008]) to send `self` to a 321-avoiding permutation.

It is shown in [EP2004] that it sends the number of centered tunnels to the number of fixed points, the number of right tunnels to the number of excedences, and the semilength plus the height of the middle point to 2 times the length of the longest increasing subsequence.

EXAMPLES:

```
sage: DyckWord([1,0,1,0]).to_321_avoiding_permutation()
[2, 1]
sage: DyckWord([1,1,0,0]).to_321_avoiding_permutation()
[1, 2]
sage: D = DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0])
sage: p = D.to_321_avoiding_permutation()
sage: p
[3, 5, 1, 6, 2, 7, 4]
sage: D.number_of_tunnels()
0
sage: p.number_of_fixed_points()
0
sage: D.number_of_tunnels('right')
4
sage: len(p.weak_excedences())-p.number_of_fixed_points()
4
sage: n = D.semilength()
sage: D.heights()[n] + n
8
sage: 2*p.longest_increasing_subsequence_length()
8
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).to_321_avoiding_
↪permutation()
[2, 1]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).to_321_avoiding_
↪permutation()
[1, 2]
>>> D = DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)])
>>> p = D.to_321_avoiding_permutation()
>>> p
[3, 5, 1, 6, 2, 7, 4]
>>> D.number_of_tunnels()
0
>>> p.number_of_fixed_points()
0
>>> D.number_of_tunnels('right')
4
>>> len(p.weak_excedences())-p.number_of_fixed_points()
4
>>> n = D.semilength()
```

(continues on next page)

(continued from previous page)

```
>>> D.heights()[n] + n
8
>>> Integer(2)*p.longest_increasing_subsequence_length()
8
```

to_Catalan_code()

Return the Catalan code associated to `self`.

A Catalan code of length n is a sequence $(a_1, a_2, \ldots, a_n)$ of n integers a_i such that:

- $0 \leq a_i \leq n - i$ for every i;
- if $i < j$ and $a_i > 0$ and $a_j > 0$ and $a_{i+1} = a_{i+2} = \cdots = a_{j-1} = 0$, then $a_i - a_j < j - i$.

It turns out that the Catalan codes of length n are in bijection with Dyck words.

The Catalan code of a Dyck word is example (x) in Richard Stanley's exercises on combinatorial interpretations for Catalan objects. The code in this example is the reverse of the description provided there. See [Sta-EC2] and [StaCat98].

EXAMPLES:

```
sage: DyckWord([]).to_Catalan_code()
[]
sage: DyckWord([1, 0]).to_Catalan_code()
[0]
sage: DyckWord([1, 1, 0, 0]).to_Catalan_code()
[0, 1]
sage: DyckWord([1, 0, 1, 0]).to_Catalan_code()
[0, 0]
sage: all(dw ==
....:     DyckWords().from_Catalan_code(dw.to_Catalan_code())
....:     for i in range(6) for dw in DyckWords(i))
True
```

```
>>> from sage.all import *
>>> DyckWord([]).to_Catalan_code()
[]
>>> DyckWord([Integer(1), Integer(0)]).to_Catalan_code()
[0]
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).to_Catalan_
↪code()
[0, 1]
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0)]).to_Catalan_
↪code()
[0, 0]
>>> all(dw ==
...     DyckWords().from_Catalan_code(dw.to_Catalan_code())
...     for i in range(Integer(6)) for dw in DyckWords(i))
True
```

See also

`to_tamari_sorting_tuple()`

to_alternating_sign_matrix()

Return self as an alternating sign matrix.

This is an inclusion map from Dyck words of length $2n$ to certain $n \times n$ alternating sign matrices.

EXAMPLES:

```
sage: DyckWord([1,1,1,0,1,0,0,0]).to_alternating_sign_matrix()              #␣
↪needs sage.modules
[ 0  0  1  0]
[ 1  0 -1  1]
[ 0  1  0  0]
[ 0  0  1  0]
sage: DyckWord([1,0,1,0,1,1,0,0]).to_alternating_sign_matrix()              #␣
↪needs sage.modules
[1 0 0 0]
[0 1 0 0]
[0 0 0 1]
[0 0 1 0]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)]).to_alternating_sign_matrix()              ␣
↪ # needs sage.modules
[ 0  0  1  0]
[ 1  0 -1  1]
[ 0  1  0  0]
[ 0  0  1  0]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0)]).to_alternating_sign_matrix()              ␣
↪ # needs sage.modules
[1 0 0 0]
[0 1 0 0]
[0 0 0 1]
[0 0 1 0]
```

to_non_decreasing_parking_function()

Bijection to *non-decreasing parking functions*.

See there the method *to_dyck_word()* for more information.

EXAMPLES:

```
sage: DyckWord([]).to_non_decreasing_parking_function()
[]
sage: DyckWord([1,0]).to_non_decreasing_parking_function()
[1]
sage: DyckWord([1,1,0,0]).to_non_decreasing_parking_function()
[1, 1]
sage: DyckWord([1,0,1,0]).to_non_decreasing_parking_function()
[1, 2]
sage: DyckWord([1,0,1,1,0,1,0,0,1,0]).to_non_decreasing_parking_function()
[1, 2, 2, 3, 5]
```

```
>>> from sage.all import *
>>> DyckWord([]).to_non_decreasing_parking_function()
[]
>>> DyckWord([Integer(1),Integer(0)]).to_non_decreasing_parking_function()
[1]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).to_non_decreasing_
↪parking_function()
[1, 1]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).to_non_decreasing_
↪parking_function()
[1, 2]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0),Integer(1),Integer(0)]).to_non_decreasing_
↪parking_function()
[1, 2, 2, 3, 5]
```

to_noncrossing_partition(*bijection=None*)

Bijection of Biane from `self` to a noncrossing partition.

There is an optional parameter `bijection` that indicates if a different bijection from Dyck words to non-crossing partitions should be used (since there are potentially many).

If the parameter `bijection` is "Stump" then the bijection used is from [Stu2008], see also the method *to_noncrossing_permutation()*.

Thanks to Mathieu Dutour for describing the bijection. See also *from_noncrossing_partition()*.

EXAMPLES:

```
sage: DyckWord([]).to_noncrossing_partition()
{}
sage: DyckWord([1, 0]).to_noncrossing_partition()
{{1}}
sage: DyckWord([1, 1, 0, 0]).to_noncrossing_partition()
{{1, 2}}
sage: DyckWord([1, 1, 1, 0, 0, 0]).to_noncrossing_partition()
{{1, 2, 3}}
sage: DyckWord([1, 0, 1, 0, 1, 0]).to_noncrossing_partition()
{{1}, {2}, {3}}
sage: DyckWord([1, 1, 0, 1, 0, 0]).to_noncrossing_partition()
{{1, 3}, {2}}
sage: DyckWord([]).to_noncrossing_partition("Stump")
{}
sage: DyckWord([1, 0]).to_noncrossing_partition("Stump")
{{1}}
sage: DyckWord([1, 1, 0, 0]).to_noncrossing_partition("Stump")
{{1, 2}}
sage: DyckWord([1, 1, 1, 0, 0, 0]).to_noncrossing_partition("Stump")
{{1, 3}, {2}}
sage: DyckWord([1, 0, 1, 0, 1, 0]).to_noncrossing_partition("Stump")
{{1}, {2}, {3}}
sage: DyckWord([1, 1, 0, 1, 0, 0]).to_noncrossing_partition("Stump")
{{1, 2, 3}}
```

```
>>> from sage.all import *
>>> DyckWord([]).to_noncrossing_partition()
{}
>>> DyckWord([Integer(1), Integer(0)]).to_noncrossing_partition()
{{1}}
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).to_noncrossing_
→partition()
{{1, 2}}
>>> DyckWord([Integer(1), Integer(1), Integer(1), Integer(0), Integer(0),␣
→Integer(0)]).to_noncrossing_partition()
{{1, 2, 3}}
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0), Integer(1),␣
→Integer(0)]).to_noncrossing_partition()
{{1}, {2}, {3}}
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
→Integer(0)]).to_noncrossing_partition()
{{1, 3}, {2}}
>>> DyckWord([]).to_noncrossing_partition("Stump")
{}
>>> DyckWord([Integer(1), Integer(0)]).to_noncrossing_partition("Stump")
{{1}}
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).to_noncrossing_
→partition("Stump")
{{1, 2}}
>>> DyckWord([Integer(1), Integer(1), Integer(1), Integer(0), Integer(0),␣
→Integer(0)]).to_noncrossing_partition("Stump")
{{1, 3}, {2}}
>>> DyckWord([Integer(1), Integer(0), Integer(1), Integer(0), Integer(1),␣
→Integer(0)]).to_noncrossing_partition("Stump")
{{1}, {2}, {3}}
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
→Integer(0)]).to_noncrossing_partition("Stump")
{{1, 2, 3}}
```

to_noncrossing_permutation()

Use the bijection by C. Stump in [Stu2008] to send `self` to a non-crossing permutation.

A non-crossing permutation when written in cyclic notation has cycles which are strictly increasing. Sends the area to the inversion number and `self.major_index()` to $n(n-1) - maj(\sigma) - maj(\sigma^{-1})$. Uses the function *pealing()*

EXAMPLES:

```
sage: DyckWord([1,1,0,0]).to_noncrossing_permutation()
[2, 1]
sage: DyckWord([1,0,1,0]).to_noncrossing_permutation()
[1, 2]
sage: p = DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_noncrossing_
→permutation(); p
[2, 3, 1, 5, 6, 7, 4]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area()
5
sage: p.length()
5
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).to_noncrossing_
↪permutation()
[2, 1]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).to_noncrossing_
↪permutation()
[1, 2]
>>> p = DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).to_noncrossing_permutation(); p
[2, 3, 1, 5, 6, 7, 4]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).area()
5
>>> p.length()
5
```

to_ordered_tree()

Return the ordered tree corresponding to `self` where the depth of the tree is the maximal height of `self`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: D = DyckWord([1,1,0,0])
sage: D.to_ordered_tree()
[[[]]]
sage: D = DyckWord([1,0,1,0])
sage: D.to_ordered_tree()
[[], []]
sage: D = DyckWord([1, 0, 1, 1, 0, 0])
sage: D.to_ordered_tree()
[[], [[]]]
sage: D = DyckWord([1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0])
sage: D.to_ordered_tree()
[[], [[], []], [[], [[]]]]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> D = DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)])
>>> D.to_ordered_tree()
[[[]]]
>>> D = DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)])
>>> D.to_ordered_tree()
[[], []]
>>> D = DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)])
>>> D.to_ordered_tree()
[[], [[]]]
>>> D = DyckWord([Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)])
>>> D.to_ordered_tree()
```

(continues on next page)

(continued from previous page)

```
[[], [[], []], [[], [[]]]]
```

to_pair_of_standard_tableaux()

Convert `self` to a pair of standard tableaux of the same shape and of length less than or equal to two.

EXAMPLES:

```
sage: DyckWord([1,0,1,0]).to_pair_of_standard_tableaux()
([[1], [2]], [[1], [2]])
sage: DyckWord([1,1,0,0]).to_pair_of_standard_tableaux()
([[1, 2]], [[1, 2]])
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_pair_of_standard_tableaux()
([[1, 2, 4, 7], [3, 5, 6]], [[1, 2, 4, 6], [3, 5, 7]])
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).to_pair_of_
↪standard_tableaux()
([[1], [2]], [[1], [2]])
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).to_pair_of_
↪standard_tableaux()
([[1, 2]], [[1, 2]])
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)]).to_pair_of_standard_tableaux()
([[1, 2, 4, 7], [3, 5, 6]], [[1, 2, 4, 6], [3, 5, 7]])
```

to_partition()

Return the partition associated to `self`.

This partition is determined by thinking of `self` as a lattice path and considering the cells which are above the path but within the $n \times n$ grid and the partition is formed by reading the sequence of the number of cells in this collection in each row.

OUTPUT:

- a partition representing the rows of cells in the square lattice and above the path

EXAMPLES:

```
sage: DyckWord([]).to_partition()
[]
sage: DyckWord([1,0]).to_partition()
[]
sage: DyckWord([1,1,0,0]).to_partition()
[]
sage: DyckWord([1,0,1,0]).to_partition()
[1]
sage: DyckWord([1,0,1,0,1,0]).to_partition()
[2, 1]
sage: DyckWord([1,1,0,0,1,0]).to_partition()
[2]
sage: DyckWord([1,0,1,1,0,0]).to_partition()
[1, 1]
```

```
>>> from sage.all import *
>>> DyckWord([]).to_partition()
[]
>>> DyckWord([Integer(1),Integer(0)]).to_partition()
[]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]).to_partition()
[]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]).to_partition()
[1]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)]).to_partition()
[2, 1]
>>> DyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)]).to_partition()
[2]
>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0)]).to_partition()
[1, 1]
```

to_permutation(*map*)

This is simply a method collecting all implemented maps from Dyck words to permutations.

INPUT:

- `map` – defines the map from Dyck words to permutations. These are currently:
  - `Bandlow-Killpatrick`: *to_312_avoiding_permutation()*
  - `Knuth`: *to_321_avoiding_permutation()*
  - `Krattenthaler`: *to_132_avoiding_permutation()*
  - `Stump`: *to_noncrossing_permutation()*

EXAMPLES:

```
sage: D = DyckWord([1,1,1,0,1,0,0,0])
sage: D.pretty_print()
   _____
 _| x x
| x x  .
| x  . .
|  . . .

sage: D.to_permutation(map='Bandlow-Killpatrick')
[3, 4, 2, 1]
sage: D.to_permutation(map='Stump')
[4, 2, 3, 1]
sage: D.to_permutation(map='Knuth')
[1, 2, 4, 3]
sage: D.to_permutation(map='Krattenthaler')
[2, 1, 3, 4]
```

```
>>> from sage.all import *
>>> D = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> D.pretty_print()
   _____
 _| x x
| x x  .
| x  . .
|  . . .

>>> D.to_permutation(map='Bandlow-Killpatrick')
[3, 4, 2, 1]
>>> D.to_permutation(map='Stump')
[4, 2, 3, 1]
>>> D.to_permutation(map='Knuth')
[1, 2, 4, 3]
>>> D.to_permutation(map='Krattenthaler')
[2, 1, 3, 4]
```

to_triangulation()

Map `self` to a triangulation.

The map from complete Dyck words of length $2n$ to triangulations of $n + 2$-gon given by this function is a bijection that can be described as follows.

Consider the Dyck word as a path from $(0, 0)$ to (n, n) staying above the diagonal, where 1 is an up step and 0 is a right step. Then each horizontal step has a co-height (0 at the top and $n - 1$ at most at the bottom). One reads the Dyck word from left to right. At the beginning, all vertices from 0 to $n + 1$ are available. For each horizontal step, one creates an edge from the vertex indexed by the co-height to the next available vertex. This chops out a triangle from the polygon and one removes the middle vertex of this triangle from the list of available vertices.

This bijection has the property that the set of smallest vertices of the edges in a triangulation is an encoding of the co-heights, from which the Dyck word can be easily recovered.

OUTPUT:

a list of pairs (i, j) that are the edges of the triangulations.

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 0]).to_triangulation()
[(0, 2)]

sage: [t.to_triangulation() for t in DyckWords(3)]
[[(2, 4), (1, 4)],
[(2, 4), (0, 2)],
[(1, 3), (1, 4)],
[(1, 3), (0, 3)],
[(0, 2), (0, 3)]]
```

```
>>> from sage.all import *
>>> DyckWord([Integer(1), Integer(1), Integer(0), Integer(0)]).to_
↪triangulation()
[(0, 2)]

>>> [t.to_triangulation() for t in DyckWords(Integer(3))]
[[(2, 4), (1, 4)],
```

(continues on next page)

(continued from previous page)

```
[(2, 4), (0, 2)],
[(1, 3), (1, 4)],
[(1, 3), (0, 3)],
[(0, 2), (0, 3)]]
```

REFERENCES:

- [Cha2005]

to_triangulation_as_graph()

Map `self` to a triangulation and return the result as a graph.

See *to_triangulation()* for the bijection used to map complete Dyck words to triangulations.

OUTPUT:

- a graph containing both the perimeter edges and the inner edges of a triangulation of a regular polygon.

EXAMPLES:

```
sage: g = DyckWord([1, 1, 0, 0, 1, 0]).to_triangulation_as_graph(); g        #␣
↪needs sage.graphs
Graph on 5 vertices
sage: g.edges(sort=True, labels=False)                                       #␣
↪needs sage.graphs
[(0, 1), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (3, 4)]
sage: g.show()                          # not tested                         #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> g = DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]).to_triangulation_as_graph(); g        # needs sage.graphs
Graph on 5 vertices
>>> g.edges(sort=True, labels=False)                                       #␣
↪needs sage.graphs
[(0, 1), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (3, 4)]
>>> g.show()                          # not tested                         #␣
↪needs sage.graphs
```

tunnels()

Return an iterator of ranges of the matching parentheses in the Dyck word `self`.

That is, if `(a,b)` is in `self.tunnels()`, then the matching parenthesis to `self[a]` is `self[b-1]`.

EXAMPLES:

```
sage: list(DyckWord([1, 1, 0, 1, 1, 0, 0, 1, 0, 0]).tunnels())
[(0, 10), (1, 3), (3, 7), (4, 6), (7, 9)]
```

```
>>> from sage.all import *
>>> list(DyckWord([Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),
↪ Integer(0), Integer(0), Integer(1), Integer(0), Integer(0)]).tunnels())
[(0, 10), (1, 3), (3, 7), (4, 6), (7, 9)]
```

class sage.combinat.dyck_word.**DyckWords**

Bases: `UniqueRepresentation`, `Parent`

Dyck words.

A Dyck word is a sequence $(w_1, \ldots, w_n)$ consisting of 1 s and 0 s, with the property that for any i with $1 \leq i \leq n$, the sequence $(w_1, \ldots, w_i)$ contains at least as many 1 s as 0 s.

A Dyck word is balanced if the total number of 1 s is equal to the total number of 0 s. The number of balanced Dyck words of length $2k$ is given by the `Catalan number` C_k.

EXAMPLES:

This class can be called with three keyword parameters `k1`, `k2` and `complete`.

If neither `k1` nor `k2` are specified, then `DyckWords` returns the combinatorial class of all balanced (=complete) Dyck words, unless the keyword `complete` is set to False (in which case it returns the class of all Dyck words).

```
sage: DW = DyckWords(); DW
Complete Dyck words
sage: [] in DW
True
sage: [1, 0, 1, 0] in DW
True
sage: [1, 1, 0] in DW
False
sage: ADW = DyckWords(complete=False); ADW
Dyck words
sage: [] in ADW
True
sage: [1, 0, 1, 0] in ADW
True
sage: [1, 1, 0] in ADW
True
sage: [1, 0, 0] in ADW
False
```

```
>>> from sage.all import *
>>> DW = DyckWords(); DW
Complete Dyck words
>>> [] in DW
True
>>> [Integer(1), Integer(0), Integer(1), Integer(0)] in DW
True
>>> [Integer(1), Integer(1), Integer(0)] in DW
False
>>> ADW = DyckWords(complete=False); ADW
Dyck words
>>> [] in ADW
True
>>> [Integer(1), Integer(0), Integer(1), Integer(0)] in ADW
True
>>> [Integer(1), Integer(1), Integer(0)] in ADW
True
>>> [Integer(1), Integer(0), Integer(0)] in ADW
False
```

If just `k1` is specified, then it returns the balanced Dyck words with `k1` opening parentheses and `k1` closing parentheses.

```
sage: DW2 = DyckWords(2); DW2
Dyck words with 2 opening parentheses and 2 closing parentheses
sage: DW2.first()
[1, 0, 1, 0]
sage: DW2.last()
[1, 1, 0, 0]
sage: DW2.cardinality()
2
sage: DyckWords(100).cardinality() == catalan_number(100)
True
```

```
>>> from sage.all import *
>>> DW2 = DyckWords(Integer(2)); DW2
Dyck words with 2 opening parentheses and 2 closing parentheses
>>> DW2.first()
[1, 0, 1, 0]
>>> DW2.last()
[1, 1, 0, 0]
>>> DW2.cardinality()
2
>>> DyckWords(Integer(100)).cardinality() == catalan_number(Integer(100))
True
```

If `k2` is specified in addition to `k1`, then it returns the Dyck words with `k1` opening parentheses and `k2` closing parentheses.

```
sage: DW32 = DyckWords(3,2); DW32
Dyck words with 3 opening parentheses and 2 closing parentheses
sage: DW32.list()
[[1, 0, 1, 0, 1],
 [1, 0, 1, 1, 0],
 [1, 1, 0, 0, 1],
 [1, 1, 0, 1, 0],
 [1, 1, 1, 0, 0]]
```

```
>>> from sage.all import *
>>> DW32 = DyckWords(Integer(3),Integer(2)); DW32
Dyck words with 3 opening parentheses and 2 closing parentheses
>>> DW32.list()
[[1, 0, 1, 0, 1],
 [1, 0, 1, 1, 0],
 [1, 1, 0, 0, 1],
 [1, 1, 0, 1, 0],
 [1, 1, 1, 0, 0]]
```

Element

alias of *DyckWord*

from_heights(*heights*)

Compute a Dyck word knowing its heights.

We view the Dyck word as a Dyck path from $(0, 0)$ to $(2n, 0)$ in the first quadrant by letting `1`'s represent steps in the direction $(1, 1)$ and `0`'s represent steps in the direction $(1, -1)$.

The `heights()` is the sequence of the y-coordinates of the $2n + 1$ lattice points along this path.

EXAMPLES:

```
sage: from sage.combinat.dyck_word import DyckWord
sage: D = DyckWords(complete=False)
sage: D.from_heights((0,))
[]
sage: D.from_heights((0, 1, 0))
[1, 0]
sage: D.from_heights((0, 1, 2, 1, 0))
[1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.dyck_word import DyckWord
>>> D = DyckWords(complete=False)
>>> D.from_heights((Integer(0),))
[]
>>> D.from_heights((Integer(0), Integer(1), Integer(0)))
[1, 0]
>>> D.from_heights((Integer(0), Integer(1), Integer(2), Integer(1),␣
↪Integer(0)))
[1, 1, 0, 0]
```

This also works for incomplete Dyck words:

```
sage: D.from_heights((0, 1, 2, 1, 2, 1))
[1, 1, 0, 1, 0]
sage: D.from_heights((0, 1, 2, 1))
[1, 1, 0]
```

```
>>> from sage.all import *
>>> D.from_heights((Integer(0), Integer(1), Integer(2), Integer(1),␣
↪Integer(2), Integer(1)))
[1, 1, 0, 1, 0]
>>> D.from_heights((Integer(0), Integer(1), Integer(2), Integer(1)))
[1, 1, 0]
```

See also:

heights(), *min_from_heights()*

min_from_heights(*heights*)

Compute the smallest Dyck word which achieves or surpasses a given sequence of heights.

INPUT:

- `heights` – a nonempty list or iterable consisting of nonnegative integers, the first of which is 0

OUTPUT:

- The smallest Dyck word whose sequence of heights is componentwise higher-or-equal to `heights`. Here, "smaller" can be understood both in the sense of lexicographic order on the Dyck words, and in the sense of every vertex of the path having the smallest possible height.

See also

- *heights()*
- *from_heights()*

EXAMPLES:

```
sage: D = DyckWords(complete=False)
sage: D.min_from_heights((0,))
[]
sage: D.min_from_heights((0, 1, 0))
[1, 0]
sage: D.min_from_heights((0, 0, 2, 0, 0))
[1, 1, 0, 0]
sage: D.min_from_heights((0, 0, 2, 0, 2, 0))
[1, 1, 0, 1, 0]
sage: D.min_from_heights((0, 0, 1, 0, 1, 0))
[1, 1, 0, 1, 0]
```

```
>>> from sage.all import *
>>> D = DyckWords(complete=False)
>>> D.min_from_heights((Integer(0),))
[]
>>> D.min_from_heights((Integer(0), Integer(1), Integer(0)))
[1, 0]
>>> D.min_from_heights((Integer(0), Integer(0), Integer(2), Integer(0),␣
↪Integer(0)))
[1, 1, 0, 0]
>>> D.min_from_heights((Integer(0), Integer(0), Integer(2), Integer(0),␣
↪Integer(2), Integer(0)))
[1, 1, 0, 1, 0]
>>> D.min_from_heights((Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0)))
[1, 1, 0, 1, 0]
```

options = Current options for DyckWords - ascii_art: path - diagram_style: grid - display: list - latex_bounce_path: False - latex_color: black - latex_diagonal: False - latex_line_width_scalar: 2 - latex_peaks: False - latex_tikz_scale: 1 - latex_valleys: False

class sage.combinat.dyck_word.**DyckWords_all**

Bases: *DyckWords*

All Dyck words.

class sage.combinat.dyck_word.**DyckWords_size**(*k1*, *k2*)

Bases: *DyckWords*

Dyck words with k_1 openers and k_2 closers.

cardinality()

Return the number of Dyck words with k_1 openers and k_2 closers.

This number is

$$\frac{k_1 - k_2 + 1}{k_1 + 1} \binom{k_1 + k_2}{k_2} = \binom{k_1 + k_2}{k_2} - \binom{k_1 + k_2}{k_2 - 1}$$

if $k_2 \leq k_1 + 1$, and 0 if $k_2 > k_1$ (these numbers are the same if $k_2 = k_1 + 1$).

EXAMPLES:

```
sage: DyckWords(3, 2).cardinality()
5
sage: all(all(DyckWords(p, q).cardinality()
....:             == len(DyckWords(p, q).list()) for q in range(p + 1))
....:       for p in range(7))
True
```

```
>>> from sage.all import *
>>> DyckWords(Integer(3), Integer(2)).cardinality()
5
>>> all(all(DyckWords(p, q).cardinality()
...             == len(DyckWords(p, q).list()) for q in range(p + Integer(1)))
...       for p in range(Integer(7)))
True
```

sage.combinat.dyck_word.**is_a**(*obj*, *k1=None*, *k2=None*)

Test if `obj` is a Dyck word with exactly `k1` open symbols and exactly `k2` close symbols.

If `k1` is not specified, then the number of open symbols can be arbitrary. If `k1` is specified but `k2` is not, then `k2` is set to be `k1`.

EXAMPLES:

```
sage: from sage.combinat.dyck_word import is_a
sage: is_a([1,1,0,0])
True
sage: is_a([1,0,1,0])
True
sage: is_a([1,1,0,0], 2)
True
sage: is_a([1,1,0,0], 3)
False
sage: is_a([1,1,0,0], 3, 2)
False
sage: is_a([1,1,0])
True
sage: is_a([0,1,0])
False
sage: is_a([1,0,0])
False
sage: is_a([1,1,0],2,1)
True
sage: is_a([1,1,0],2)
False
sage: is_a([1,1,0],1,1)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.dyck_word import is_a
>>> is_a([Integer(1),Integer(1),Integer(0),Integer(0)])
True
>>> is_a([Integer(1),Integer(0),Integer(1),Integer(0)])
True
>>> is_a([Integer(1),Integer(1),Integer(0),Integer(0)], Integer(2))
True
>>> is_a([Integer(1),Integer(1),Integer(0),Integer(0)], Integer(3))
False
>>> is_a([Integer(1),Integer(1),Integer(0),Integer(0)], Integer(3), Integer(2))
False
>>> is_a([Integer(1),Integer(1),Integer(0)])
True
>>> is_a([Integer(0),Integer(1),Integer(0)])
False
>>> is_a([Integer(1),Integer(0),Integer(0)])
False
>>> is_a([Integer(1),Integer(1),Integer(0)],Integer(2),Integer(1))
True
>>> is_a([Integer(1),Integer(1),Integer(0)],Integer(2))
False
>>> is_a([Integer(1),Integer(1),Integer(0)],Integer(1),Integer(1))
False
```

sage.combinat.dyck_word.**is_area_sequence**(*seq*)

Test if a sequence l of integers satisfies $l_0 = 0$ and $0 \leq l_{i+1} \leq l_i + 1$ for $i > 0$.

EXAMPLES:

```
sage: from sage.combinat.dyck_word import is_area_sequence
sage: is_area_sequence([0,2,0])
False
sage: is_area_sequence([1,2,3])
False
sage: is_area_sequence([0,1,0])
True
sage: is_area_sequence([0,1,2])
True
sage: is_area_sequence([])
True
```

```
>>> from sage.all import *
>>> from sage.combinat.dyck_word import is_area_sequence
>>> is_area_sequence([Integer(0),Integer(2),Integer(0)])
False
>>> is_area_sequence([Integer(1),Integer(2),Integer(3)])
False
>>> is_area_sequence([Integer(0),Integer(1),Integer(0)])
True
>>> is_area_sequence([Integer(0),Integer(1),Integer(2)])
True
>>> is_area_sequence([])
```

(continues on next page)

(continued from previous page)

```
True
```

sage.combinat.dyck_word.**pealing**(*D*, *return_touches=False*)

A helper function for computing the bijection from a Dyck word to a 231-avoiding permutation using the bijection "Stump". For details see [Stu2008].

See also

to_noncrossing_partition()

EXAMPLES:

```
sage: from sage.combinat.dyck_word import pealing
sage: pealing(DyckWord([1,1,0,0]))
[1, 0, 1, 0]
sage: pealing(DyckWord([1,0,1,0]))
[1, 0, 1, 0]
sage: pealing(DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]))
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
sage: pealing(DyckWord([1,1,0,0]),return_touches=True)
([1, 0, 1, 0], [[1, 2]])
sage: pealing(DyckWord([1,0,1,0]),return_touches=True)
([1, 0, 1, 0], [])
sage: pealing(DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]),return_touches=True)
([1, 0, 1, 0, 1, 0, 1, 0, 1, 0], [[1, 2], [3, 5]])
```

```
>>> from sage.all import *
>>> from sage.combinat.dyck_word import pealing
>>> pealing(DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]))
[1, 0, 1, 0]
>>> pealing(DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]))
[1, 0, 1, 0]
>>> pealing(DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)]))
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
>>> pealing(DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)]),return_
↪touches=True)
([1, 0, 1, 0], [[1, 2]])
>>> pealing(DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)]),return_
↪touches=True)
([1, 0, 1, 0], [])
>>> pealing(DyckWord([Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)]),return_
↪touches=True)
([1, 0, 1, 0, 1, 0, 1, 0, 1, 0], [[1, 2], [3, 5]])
```

sage.combinat.dyck_word.**replace_parens**(*x*)

A map sending `'('` to `open_symbol` and `')'` to `close_symbol`, and raising an error on any input other than `'('` and `')'`. The values of the constants `open_symbol` and `close_symbol` are subject to change.

This is the inverse map of *replace_symbols()*.

INPUT:

- x – either an opening or closing parenthesis

OUTPUT:

- If x is an opening parenthesis, replace x with the constant open_symbol.
- If x is a closing parenthesis, replace x with the constant close_symbol.
- Raise a ValueError if x is neither an opening nor a closing parenthesis.

See also

replace_symbols()

EXAMPLES:

```
sage: from sage.combinat.dyck_word import replace_parens
sage: replace_parens('(')
1
sage: replace_parens(')')
0
sage: replace_parens(1)
Traceback (most recent call last):
...
ValueError
```

```
>>> from sage.all import *
>>> from sage.combinat.dyck_word import replace_parens
>>> replace_parens('(')
1
>>> replace_parens(')')
0
>>> replace_parens(Integer(1))
Traceback (most recent call last):
...
ValueError
```

sage.combinat.dyck_word.**replace_symbols**(*x*)

A map sending open_symbol to '(' and close_symbol to ')', and raising an error on any input other than open_symbol and close_symbol. The values of the constants open_symbol and close_symbol are subject to change.

This is the inverse map of *replace_parens()*.

INPUT:

- x – either open_symbol or close_symbol

OUTPUT:

- If x is open_symbol, replace x with '('.
- If x is close_symbol, replace x with ')'.
- If x is neither open_symbol nor close_symbol, a ValueError is raised.

See also

replace_parens()

EXAMPLES:

```
sage: from sage.combinat.dyck_word import replace_symbols
sage: replace_symbols(1)
'('
sage: replace_symbols(0)
')'
sage: replace_symbols(3)
Traceback (most recent call last):
...
ValueError
```

```
>>> from sage.all import *
>>> from sage.combinat.dyck_word import replace_symbols
>>> replace_symbols(Integer(1))
'('
>>> replace_symbols(Integer(0))
')'
>>> replace_symbols(Integer(3))
Traceback (most recent call last):
...
ValueError
```

5.1.101 Substitutions over unit cube faces (Rauzy fractals)

This module implements the $E_1^*(\sigma)$ substitution associated with a one-dimensional substitution σ, that acts on unit faces of dimension $(d-1)$ in $\mathbf{R}^d$.

This module defines the following classes and functions:

- `Face` – a class to model a face
- `Patch` – a class to model a finite set of faces
- `E1Star` – a class to model the $E_1^*(\sigma)$ application defined by the substitution sigma

See the documentation of these objects for more information.

The convention for the choice of the unit faces and the definition of $E_1^*(\sigma)$ varies from article to article. Here, unit faces are defined by

$$\begin{array}{rcl} [x,1]^* & = & \{x+\lambda e_2+\mu e_3 : \lambda,\mu\in[0,1]\} \\ [x,2]^* & = & \{x+\lambda e_1+\mu e_3 : \lambda,\mu\in[0,1]\} \\ [x,3]^* & = & \{x+\lambda e_1+\mu e_2 : \lambda,\mu\in[0,1]\} \end{array}$$

and the dual substitution $E_1^*(\sigma)$ is defined by

$$E_1^*(\sigma)([x,i]^*) = \bigcup_{k=1,2,3} \quad \bigcup_{s|\sigma(k)=pis} [M^{-1}(x+\ell(s)),k]^*,$$

where $\ell(s)$ is the abelianized of s, and M is the matrix of σ.

AUTHORS:

- Franco Saliola (2009): initial version
- Vincent Delecroix, Timo Jolivet, Stepan Starosta, Sebastien Labbe (2010-05): redesign
- Timo Jolivet (2010-08, 2010-09, 2011): redesign

REFERENCES:

EXAMPLES:

We start by drawing a simple three-face patch:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: x = [Face((0,0,0),1), Face((0,0,0),2), Face((0,0,0),3)]
sage: P = Patch(x)
sage: P
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]
sage: P.plot()                          #not tested
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import E1Star, Face, Patch
>>> x = [Face((Integer(0),Integer(0),Integer(0)),Integer(1)), Face((Integer(0),
↪Integer(0),Integer(0)),Integer(2)), Face((Integer(0),Integer(0),Integer(0)),
↪Integer(3))]
>>> P = Patch(x)
>>> P
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]
>>> P.plot()                          #not tested
```

We apply a substitution to this patch, and draw the result:

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1, 0, -
↪1), 1]*]
sage: E(P).plot()                          #not tested
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)], Integer(2):[Integer(1),
↪Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1, 0, -
↪1), 1]*]
>>> E(P).plot()                          #not tested
```

Note

- The type of a face is given by an integer in `[1, ..., d]` where `d` is the length of the vector of the face.
- The alphabet of the domain and the codomain of σ must be equal, and they must be of the form `[1, ..., d]`, where `d` is a positive integer corresponding to the length of the vectors of the faces on which $E_1^*(\sigma)$ will act.

```
sage: P = Patch([Face((0,0,0),1), Face((0,0,0),2), Face((0,0,0),3)])
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1, 0, -
↪1), 1]*]
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),Integer(1)), Face((Integer(0),
↪Integer(0),Integer(0)),Integer(2)), Face((Integer(0),Integer(0),Integer(0)),
↪Integer(3))])
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)], Integer(2):[Integer(1),
↪Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1, 0, -
↪1), 1]*]
```

The application of an `E1Star` substitution assigns to each new face the color of its preimage. The `repaint` method allows us to repaint the faces of a patch. A single color can also be assigned to every face, by specifying a list of a single color:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 5)
sage: P.repaint(['green'])
sage: P.plot()                    #not tested
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]])
>>> P = E(P, Integer(5))
>>> P.repaint(['green'])
>>> P.plot()                    #not tested
```

A list of colors allows us to color the faces sequentially:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P)
sage: P.repaint(['red', 'yellow', 'green', 'blue', 'black'])
sage: P = E(P, 3)
sage: P.plot()                    #not tested
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]])
>>> P = E(P)
>>> P.repaint(['red', 'yellow', 'green', 'blue', 'black'])
>>> P = E(P, Integer(3))
>>> P.plot()                    #not tested
```

All the color schemes from `list(matplotlib.cm.datad)` can be used:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.repaint(cmap='summer')
sage: P = E(P, 3)
sage: P.plot()                    #not tested
sage: P.repaint(cmap='hsv')
sage: P = E(P, 2)
sage: P.plot()                    #not tested
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]])
>>> P.repaint(cmap='summer')
>>> P = E(P, Integer(3))
>>> P.plot()                    #not tested
>>> P.repaint(cmap='hsv')
>>> P = E(P, Integer(2))
>>> P.plot()                    #not tested
```

It is also possible to specify a dictionary to color the faces according to their type:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 5)
sage: P.repaint({1:(0.7, 0.7, 0.7), 2:(0.5,0.5,0.5), 3:(0.3,0.3,0.3)})
sage: P.plot()                    #not tested
sage: P.repaint({1:'red', 2:'yellow', 3:'green'})
sage: P.plot()                    #not tested
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]])
>>> P = E(P, Integer(5))
>>> P.repaint({Integer(1):(RealNumber('0.7'), RealNumber('0.7'), RealNumber('0.7')),␣
↪Integer(2):(RealNumber('0.5'),RealNumber('0.5'),RealNumber('0.5')),␣
↪Integer(3):(RealNumber('0.3'),RealNumber('0.3'),RealNumber('0.3'))})
>>> P.plot()                    #not tested
>>> P.repaint({Integer(1):'red', Integer(2):'yellow', Integer(3):'green'})
>>> P.plot()                    #not tested
```

Let us look at a nice big patch in 3D:

```
sage: sigma = WordMorphism({1:[1,2], 2:[3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = P + P.translate([-1,1,0])
sage: P = E(P, 11)
sage: P.plot3d()                    #not tested
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)], Integer(2):[Integer(3)],
↪ Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(3)]])
>>> P = P + P.translate([-Integer(1),Integer(1),Integer(0)])
>>> P = E(P, Integer(11))
>>> P.plot3d()                    #not tested
```

Plotting with TikZ pictures is possible:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: s = P.plot_tikz()
sage: print(s)                        #not tested
\begin{tikzpicture}
[x={(-0.216506cm,-0.125000cm)}, y={(0.216506cm,-0.125000cm)}, z={(0.000000cm,0.
↪250000cm)}]
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]])
>>> s = P.plot_tikz()
>>> print(s)                    #not tested
\begin{tikzpicture}
[x={(-0.216506cm,-0.125000cm)}, y={(0.216506cm,-0.125000cm)}, z={(0.000000cm,0.
↪250000cm)}]
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\end{tikzpicture}
```

Plotting patches made of unit segments instead of unit faces:

```
sage: P = Patch([Face([0,0], 1), Face([0,0], 2)])
sage: E = E1Star(WordMorphism({1:[1,2],2:[1]}))
sage: F = E1Star(WordMorphism({1:[1,1,2],2:[2,1]}))
sage: E(P,5).plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 21 graphics primitives
sage: F(P,3).plot()                                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics object consisting of 34 graphics primitives
```

```
>>> from sage.all import *
>>> P = Patch([Face([Integer(0),Integer(0)], Integer(1)), Face([Integer(0),
↪Integer(0)], Integer(2))])
>>> E = E1Star(WordMorphism({Integer(1):[Integer(1),Integer(2)],
↪Integer(2):[Integer(1)]}))
>>> F = E1Star(WordMorphism({Integer(1):[Integer(1),Integer(1),Integer(2)],
↪Integer(2):[Integer(2),Integer(1)]}))
>>> E(P,Integer(5)).plot()                                                    ␣
↪     # needs sage.plot
Graphics object consisting of 21 graphics primitives
>>> F(P,Integer(3)).plot()                                                    ␣
↪     # needs sage.plot
Graphics object consisting of 34 graphics primitives
```

Everything works in any dimension (except for the plotting features which only work in dimension two or three):

```
sage: P = Patch([Face((0,0,0,0),1), Face((0,0,0,0),4)])
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1,4], 4:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0, 0), 3]*, [(0, 0, 0, 0), 4]*, [(0, 0, 1, -1), 3]*, [(0, 1, 0, -1),␣
↪2]*, [(1, 0, 0, -1), 1]*]
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0),Integer(0)),Integer(1)),␣
↪Face((Integer(0),Integer(0),Integer(0),Integer(0)),Integer(4))])
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)], Integer(2):[Integer(1),
↪Integer(3)], Integer(3):[Integer(1),Integer(4)], Integer(4):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E(P)
Patch: [[(0, 0, 0, 0), 3]*, [(0, 0, 0, 0), 4]*, [(0, 0, 1, -1), 3]*, [(0, 1, 0, -1),␣
↪2]*, [(1, 0, 0, -1), 1]*]
```

```
sage: sigma = WordMorphism({1:[1,2],2:[1,3],3:[1,4],4:[1,5],5:[1,6],6:[1,7],7:[1,8],
↪8:[1,9],9:[1,10],10:[1,11],11:[1,12],12:[1]})
sage: E = E1Star(sigma)
sage: E
E_1^*(1->12, 10->1,11, 11->1,12, 12->1, 2->13, 3->14, 4->15, 5->16, 6->17, 7->18, 8->
↪19, 9->1,10)
sage: P = Patch([Face((0,0,0,0,0,0,0,0,0,0,0,0),t) for t in [1,2,3]])
sage: for x in sorted(E(P), key=lambda x : (x.vector(),x.type())): print(x)
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 2]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 12]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1), 11]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1), 10]*
[(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1), 9]*
[(0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, -1), 8]*
```

(continues on next page)

(continued from previous page)

```
[(0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, -1), 7]*
[(0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, -1), 6]*
[(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1), 5]*
[(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, -1), 4]*
[(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, -1), 3]*
[(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1), 2]*
[(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1), 1]*
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],Integer(2):[Integer(1),
↪Integer(3)],Integer(3):[Integer(1),Integer(4)],Integer(4):[Integer(1),Integer(5)],
↪Integer(5):[Integer(1),Integer(6)],Integer(6):[Integer(1),Integer(7)],
↪Integer(7):[Integer(1),Integer(8)],Integer(8):[Integer(1),Integer(9)],
↪Integer(9):[Integer(1),Integer(10)],Integer(10):[Integer(1),Integer(11)],
↪Integer(11):[Integer(1),Integer(12)],Integer(12):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E
E_1^*(1->12, 10->1,11, 11->1,12, 12->1, 2->13, 3->14, 4->15, 5->16, 6->17, 7->18, 8->
↪19, 9->1,10)
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)),t)␣
↪for t in [Integer(1),Integer(2),Integer(3)]])
>>> for x in sorted(E(P), key=lambda x : (x.vector(),x.type())): print(x)
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 2]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 12]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1), 11]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1), 10]*
[(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1), 9]*
[(0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, -1), 8]*
[(0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, -1), 7]*
[(0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, -1), 6]*
[(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1), 5]*
[(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, -1), 4]*
[(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, -1), 3]*
[(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1), 2]*
[(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1), 1]*
```

class `sage.combinat.e_one_star.`**E1Star**(*sigma*, *method='suffix'*)

Bases: `SageObject`

A class to model the $E_1^*(\sigma)$ map associated with a unimodular substitution σ.

INPUT:

- `sigma` – unimodular `WordMorphism`, i.e. such that its incidence matrix has determinant ± 1
- `method` – 'prefix' or 'suffix' (default: `'suffix'`); enables to use an alternative definition $E_1^*(\sigma)$ substitutions, where the abelianized of the prefix` is used instead of the suffix

Note

The alphabet of the domain and the codomain of σ must be equal, and they must be of the form `[1, ...,` d], where d is a positive integer corresponding to the length of the vectors of the faces on which $E_1^*(\sigma)$ will act.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1,␣
↪0, -1), 1]*]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import E1Star, Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]])
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1,␣
↪0, -1), 1]*]
```

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma, method='prefix')
sage: E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 0, 1), 1]*, [(0,␣
↪0, 1), 2]*]
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]])
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma, method='prefix')
>>> E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 0, 1), 1]*, [(0,␣
↪0, 1), 2]*]
```

```
sage: x = [Face((0,0,0,0),1), Face((0,0,0,0),4)]
sage: P = Patch(x)
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1,4], 4:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0, 0), 3]*, [(0, 0, 0, 0), 4]*, [(0, 0, 1, -1), 3]*, [(0, 1, 0, -
↪1), 2]*, [(1, 0, 0, -1), 1]*]
```

```
>>> from sage.all import *
>>> x = [Face((Integer(0),Integer(0),Integer(0),Integer(0)),Integer(1)),␣
```

(continues on next page)

(continued from previous page)

```
↪Face((Integer(0),Integer(0),Integer(0),Integer(0)),Integer(4))]
>>> P = Patch(x)
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1),Integer(4)],␣
↪Integer(4):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E(P)
Patch: [[(0, 0, 0, 0), 3]*, [(0, 0, 0, 0), 4]*, [(0, 0, 1, -1), 3]*, [(0, 1, 0, -
↪1), 2]*, [(1, 0, 0, -1), 1]*]
```

inverse_matrix()

Return the inverse of the matrix associated with `self`.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E.inverse_matrix()
[ 0  1  0]
[ 0  0  1]
[ 1 -1 -1]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import E1Star, Face, Patch
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E.inverse_matrix()
[ 0  1  0]
[ 0  0  1]
[ 1 -1 -1]
```

matrix()

Return the matrix associated with `self`.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E.matrix()
[1 1 1]
[1 0 0]
[0 1 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import E1Star, Face, Patch
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E.matrix()
```

(continues on next page)

(continued from previous page)

```
[1 1 1]
[1 0 0]
[0 1 0]
```

sigma()

Return the WordMorphism associated with self.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E.sigma()
WordMorphism: 1->12, 2->13, 3->1
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import E1Star, Face, Patch
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> E.sigma()
WordMorphism: 1->12, 2->13, 3->1
```

class sage.combinat.e_one_star.**Face**(*v*, *t*, *color=None*)

Bases: SageObject

A class to model a unit face of arbitrary dimension.

A unit face in dimension d is represented by a d-dimensional vector v and a type t in $\{1, \ldots, d\}$. The type of the face corresponds to the canonical unit vector to which the face is orthogonal. The optional color argument is used in plotting functions.

INPUT:

- v – tuple of integers
- t – integer in [1, ..., len(v)], type of the face. The face of type i is orthogonal to the canonical vector e_i.
- color – color (default: None); color of the face, used for plotting only. If None, its value is guessed from the face type.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face
sage: f = Face((0,2,0), 3)
sage: f.vector()
(0, 2, 0)
sage: f.type()
3
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face
>>> f = Face((Integer(0),Integer(2),Integer(0)), Integer(3))
>>> f.vector()
```

(continues on next page)

(continued from previous page)

```
(0, 2, 0)
>>> f.type()
3
```

```
sage: f = Face((0,2,0), 3, color=(0.5, 0.5, 0.5))
sage: f.color()
RGB color (0.5, 0.5, 0.5)
```

```
>>> from sage.all import *
>>> f = Face((Integer(0),Integer(2),Integer(0)), Integer(3), color=(RealNumber('0.
→5'), RealNumber('0.5'), RealNumber('0.5')))
>>> f.color()
RGB color (0.5, 0.5, 0.5)
```

color (*color=None*)

Return or change the color of the face.

INPUT:

- `color` – string, rgb tuple, color (default: `None`) the new color to assign to the face. If `None`, it returns the color of the face.

OUTPUT: color or None

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face
sage: f = Face((0,2,0), 3)
sage: f.color()
RGB color (0.0, 0.0, 1.0)
sage: f.color('red')
sage: f.color()
RGB color (1.0, 0.0, 0.0)
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face
>>> f = Face((Integer(0),Integer(2),Integer(0)), Integer(3))
>>> f.color()
RGB color (0.0, 0.0, 1.0)
>>> f.color('red')
>>> f.color()
RGB color (1.0, 0.0, 0.0)
```

type ()

Return the type of the face.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face
sage: f = Face((0,2,0), 3)
sage: f.type()
3
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face
>>> f = Face((Integer(0),Integer(2),Integer(0)), Integer(3))
>>> f.type()
3
```

```
sage: f = Face((0,2,0), 3)
sage: f.type()
3
```

```
>>> from sage.all import *
>>> f = Face((Integer(0),Integer(2),Integer(0)), Integer(3))
>>> f.type()
3
```

vector()

Return the vector of the face.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face
sage: f = Face((0,2,0), 3)
sage: f.vector()
(0, 2, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face
>>> f = Face((Integer(0),Integer(2),Integer(0)), Integer(3))
>>> f.vector()
(0, 2, 0)
```

class sage.combinat.e_one_star.**Patch**(*faces*, *face_contour=None*)

Bases: `SageObject`

A class to model a collection of faces. A patch is represented by an immutable set of Faces.

Note

The dimension of a patch is the length of the vectors of the faces in the patch, which is assumed to be the same for every face in the patch.

Note

Since version 4.7.1, Patches are immutable, except for the colors of the faces, which are not taken into account for equality tests and hash functions.

INPUT:

- `faces` – finite iterable of faces
- `face_contour` – dictionary (default: `None`); maps the face type to vectors describing the contour of unit faces. If `None`, defaults contour are assumed for faces of type 1, 2, 3 or 1, 2, 3. Used in plotting methods

only.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]])
>>> P
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]
```

```
sage: face_contour = {}
sage: face_contour[1] = map(vector, [(0,0,0),(0,1,0),(0,1,1),(0,0,1)])
sage: face_contour[2] = map(vector, [(0,0,0),(0,0,1),(1,0,1),(1,0,0)])
sage: face_contour[3] = map(vector, [(0,0,0),(1,0,0),(1,1,0),(0,1,0)])
sage: Patch([Face((0,0,0),t) for t in [1,2,3]], face_contour=face_contour)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]
```

```
>>> from sage.all import *
>>> face_contour = {}
>>> face_contour[Integer(1)] = map(vector, [(Integer(0),Integer(0),Integer(0)),
↪(Integer(0),Integer(1),Integer(0)),(Integer(0),Integer(1),Integer(1)),
↪(Integer(0),Integer(0),Integer(1))])
>>> face_contour[Integer(2)] = map(vector, [(Integer(0),Integer(0),Integer(0)),
↪(Integer(0),Integer(0),Integer(1)),(Integer(1),Integer(0),Integer(1)),
↪(Integer(1),Integer(0),Integer(0))])
>>> face_contour[Integer(3)] = map(vector, [(Integer(0),Integer(0),Integer(0)),
↪(Integer(1),Integer(0),Integer(0)),(Integer(1),Integer(1),Integer(0)),
↪(Integer(0),Integer(1),Integer(0))])
>>> Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in [Integer(1),
↪Integer(2),Integer(3)]], face_contour=face_contour)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]
```

difference(*other*)

Return the difference of `self` and `other`.

INPUT:

- `other` – a finite iterable of faces or a single face

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.difference(Face([0,0,0],2))
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 3]*]
sage: P.difference(P)
Patch: []
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P.difference(Face([Integer(0),Integer(0),Integer(0)],Integer(2)))
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 3]*]
>>> P.difference(P)
Patch: []
```

dimension()

Return the dimension of the vectors of the faces of `self`.

It returns `None` if `self` is the empty patch.

The dimension of a patch is the length of the vectors of the faces in the patch, which is assumed to be the same for every face in the patch.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.dimension()
3
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P.dimension()
3
```

faces_of_color(*color*)

Return a list of the faces that have the given color.

INPUT:

- `color` – color

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1, 'red'), Face((1,2,0),3, 'blue'), Face((1,2,
↪0),1, 'red')])
sage: sorted(P.faces_of_color('red'))
[[(0, 0, 0), 1]*, [(1, 2, 0), 1]*]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),Integer(1), 'red'),␣
↪Face((Integer(1),Integer(2),Integer(0)),Integer(3), 'blue'),␣
↪Face((Integer(1),Integer(2),Integer(0)),Integer(1), 'red')])
>>> sorted(P.faces_of_color('red'))
[[(0, 0, 0), 1]*, [(1, 2, 0), 1]*]
```

faces_of_type(*t*)

Return a list of the faces that have type `t`.

INPUT:

- `t` – integer or any other type

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1), Face((1,2,0),3), Face((1,2,0),1)])
sage: sorted(P.faces_of_type(1))
[[(0, 0, 0), 1]*, [(1, 2, 0), 1]*]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),Integer(1)),␣
↪Face((Integer(1),Integer(2),Integer(0)),Integer(3)), Face((Integer(1),
↪Integer(2),Integer(0)),Integer(1))])
>>> sorted(P.faces_of_type(Integer(1)))
[[(0, 0, 0), 1]*, [(1, 2, 0), 1]*]
```

faces_of_vector(*v*)

Return a list of the faces whose vector is `v`.

INPUT:

- `v` – a vector

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1), Face((1,2,0),3), Face((1,2,0),1)])
sage: sorted(P.faces_of_vector([1,2,0]))
[[(1, 2, 0), 1]*, [(1, 2, 0), 3]*]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),Integer(1)),␣
↪Face((Integer(1),Integer(2),Integer(0)),Integer(3)), Face((Integer(1),
↪Integer(2),Integer(0)),Integer(1))])
>>> sorted(P.faces_of_vector([Integer(1),Integer(2),Integer(0)]))
[[(1, 2, 0), 1]*, [(1, 2, 0), 3]*]
```

occurrences_of(*other*)

Return all positions at which other appears in self, that is, all vectors v such that `set(other.translate(v)) <= set(self)`.

INPUT:

- `other` – a Patch

OUTPUT: list of vectors

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch, E1Star
sage: P = Patch([Face([0,0,0], 1), Face([0,0,0], 2), Face([0,0,0], 3)])
sage: Q = Patch([Face([0,0,0], 1), Face([0,0,0], 2)])
sage: P.occurrences_of(Q)
```

(continues on next page)

(continued from previous page)

```
[(0, 0, 0)]
sage: Q = Q.translate([1,2,3])
sage: P.occurrences_of(Q)
[(-1, -2, -3)]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch, E1Star
>>> P = Patch([Face([Integer(0),Integer(0),Integer(0)], Integer(1)),␣
↪Face([Integer(0),Integer(0),Integer(0)], Integer(2)), Face([Integer(0),
↪Integer(0),Integer(0)], Integer(3))])
>>> Q = Patch([Face([Integer(0),Integer(0),Integer(0)], Integer(1)),␣
↪Face([Integer(0),Integer(0),Integer(0)], Integer(2))])
>>> P.occurrences_of(Q)
[(0, 0, 0)]
>>> Q = Q.translate([Integer(1),Integer(2),Integer(3)])
>>> P.occurrences_of(Q)
[(-1, -2, -3)]
```

```
sage: E = E1Star(WordMorphism({1:[1,2], 2:[1,3], 3:[1]}))
sage: P = Patch([Face([0,0,0], 1), Face([0,0,0], 2), Face([0,0,0], 3)])
sage: P = E(P,4)
sage: Q = Patch([Face([0,0,0], 1), Face([0,0,0], 2)])
sage: L = P.occurrences_of(Q)
sage: sorted(L)
[(0, 0, 0), (0, 0, 1), (0, 1, -1), (1, 0, -1), (1, 1, -3), (1, 1, -2)]
```

```
>>> from sage.all import *
>>> E = E1Star(WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]}))
>>> P = Patch([Face([Integer(0),Integer(0),Integer(0)], Integer(1)),␣
↪Face([Integer(0),Integer(0),Integer(0)], Integer(2)), Face([Integer(0),
↪Integer(0),Integer(0)], Integer(3))])
>>> P = E(P,Integer(4))
>>> Q = Patch([Face([Integer(0),Integer(0),Integer(0)], Integer(1)),␣
↪Face([Integer(0),Integer(0),Integer(0)], Integer(2))])
>>> L = P.occurrences_of(Q)
>>> sorted(L)
[(0, 0, 0), (0, 0, 1), (0, 1, -1), (1, 0, -1), (1, 1, -3), (1, 1, -2)]
```

plot(*projmat=None*, *opacity=0.75*)

Return a 2D graphic object depicting the patch.

INPUT:

- `projmat` – matrix (default: `None`); the projection matrix. Its number of lines must be two. Its number of columns must equal the dimension of the ambient space of the faces. If `None`, the isometric projection is used by default.
- `opacity` – float between `0` and `1` (default: `0.75`) opacity of the face

Warning

Plotting is implemented only for patches in two or three dimensions.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.plot()                                                           #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import E1Star, Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P.plot()                                                           #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 5)
sage: P.plot()                                                           #␣
↪needs sage.plot
Graphics object consisting of 57 graphics primitives
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P = E(P, Integer(5))
>>> P.plot()                                                           #␣
↪needs sage.plot
Graphics object consisting of 57 graphics primitives
```

Plot with a different projection matrix:

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: M = matrix(2, 3, [1,0,-1,0.3,1,-3])
sage: P = E(P, 3)
sage: P.plot(projmat=M)                                                  #␣
↪needs sage.plot
Graphics object consisting of 17 graphics primitives
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
```

(continues on next page)

(continued from previous page)

```
>>> E = E1Star(sigma)
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> M = matrix(Integer(2), Integer(3), [Integer(1),Integer(0),-Integer(1),
↪RealNumber('0.3'),Integer(1),-Integer(3)])
>>> P = E(P, Integer(3))
>>> P.plot(projmat=M)                                                    #␣
↪needs sage.plot
Graphics object consisting of 17 graphics primitives
```

Plot patches made of unit segments:

```
sage: P = Patch([Face([0,0], 1), Face([0,0], 2)])
sage: E = E1Star(WordMorphism({1:[1,2],2:[1]}))
sage: F = E1Star(WordMorphism({1:[1,1,2],2:[2,1]}))
sage: E(P,5).plot()                                                       #␣
↪needs sage.plot
Graphics object consisting of 21 graphics primitives
sage: F(P,3).plot()                                                       #␣
↪needs sage.plot
Graphics object consisting of 34 graphics primitives
```

```
>>> from sage.all import *
>>> P = Patch([Face([Integer(0),Integer(0)], Integer(1)), Face([Integer(0),
↪Integer(0)], Integer(2))])
>>> E = E1Star(WordMorphism({Integer(1):[Integer(1),Integer(2)],
↪Integer(2):[Integer(1)]}))
>>> F = E1Star(WordMorphism({Integer(1):[Integer(1),Integer(1),Integer(2)],
↪Integer(2):[Integer(2),Integer(1)]}))
>>> E(P,Integer(5)).plot()                                                 ␣
↪     # needs sage.plot
Graphics object consisting of 21 graphics primitives
>>> F(P,Integer(3)).plot()                                                 ␣
↪     # needs sage.plot
Graphics object consisting of 34 graphics primitives
```

plot3d()

Return a 3D graphics object depicting the patch.

> **Warning**
>
> 3D plotting is implemented only for patches in three dimensions.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.plot3d()                              #not tested
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import E1Star, Face, Patch
```

(continues on next page)

(continued from previous page)

```
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P.plot3d()                  #not tested
```

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 5)
sage: P.repaint()
sage: P.plot3d()                  #not tested
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P = E(P, Integer(5))
>>> P.repaint()
>>> P.plot3d()                  #not tested
```

plot_tikz(*projmat=None*, *print_tikz_env=True*, *edgecolor='black'*, *scale=0.25*, *drawzero=False*, *extra_code_before=''*, *extra_code_after=''*)

Return a string containing some TikZ code to be included into a LaTeX document, depicting the patch.

> **Warning**
>
> Tikz Plotting is implemented only for patches in three dimensions.

INPUT:

- `projmat` – matrix (default: `None`); the projection matrix. Its number of lines must be two. Its number of columns must equal the dimension of the ambient space of the faces. If `None`, the isometric projection is used by default.
- `print_tikz_env` – boolean (default: `True`); if `True`, the tikzpicture environment are printed
- `edgecolor` – string (default: `'black'`); either `'black'` or `'facecolor'` (color of unit face edges)
- `scale` – real number (default: `0.25`) scaling constant for the whole figure
- `drawzero` – boolean (default: `False`); if `True`, mark the origin by a black dot
- `extra_code_before` – string (default: `''`); extra code to include in the tikz picture
- `extra_code_after` – string (default: `''`); extra code to include in the tikz picture

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: s = P.plot_tikz()
sage: len(s)
602
```

(continues on next page)

(continued from previous page)

```
sage: print(s)       #not tested
\begin{tikzpicture}
[x={(-0.216506cm,-0.125000cm)}, y={(0.216506cm,-0.125000cm)}, z={(0.000000cm,
↪0.250000cm)}]
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import E1Star, Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> s = P.plot_tikz()
>>> len(s)
602
>>> print(s)       #not tested
\begin{tikzpicture}
[x={(-0.216506cm,-0.125000cm)}, y={(0.216506cm,-0.125000cm)}, z={(0.000000cm,
↪0.250000cm)}]
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\end{tikzpicture}
```

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 4)
sage: from sage.misc.latex import latex              #not tested
sage: latex.add_to_preamble('\\usepackage{tikz}')    #not tested
sage: view(P)                              #not tested
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
```

(continues on next page)

(continued from previous page)

```
>>> P = E(P, Integer(4))
>>> from sage.misc.latex import latex              #not tested
>>> latex.add_to_preamble('\\usepackage{tikz}')    #not tested
>>> view(P)                            #not tested
```

Plot using shades of gray (useful for article figures):

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.repaint([(0.9, 0.9, 0.9), (0.65,0.65,0.65), (0.4,0.4,0.4)])
sage: P = E(P, 4)
sage: s = P.plot_tikz()
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P.repaint([(RealNumber('0.9'), RealNumber('0.9'), RealNumber('0.9')),␣
↪(RealNumber('0.65'),RealNumber('0.65'),RealNumber('0.65')), (RealNumber('0.4
↪'),RealNumber('0.4'),RealNumber('0.4'))])
>>> P = E(P, Integer(4))
>>> s = P.plot_tikz()
```

Plotting with various options:

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: M = matrix(2,3,[float(u) for u in [1,0,-0.7071,0,1,-0.7071]])
sage: P = E(P, 3)
sage: s = P.plot_tikz(projmat=M, edgecolor='facecolor', scale=0.6,␣
↪drawzero=True)
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(1),Integer(3)], Integer(3):[Integer(1)]})
>>> E = E1Star(sigma)
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> M = matrix(Integer(2),Integer(3),[float(u) for u in [Integer(1),
↪Integer(0),-RealNumber('0.7071'),Integer(0),Integer(1),-RealNumber('0.7071
↪')]])
>>> P = E(P, Integer(3))
>>> s = P.plot_tikz(projmat=M, edgecolor='facecolor', scale=RealNumber('0.6'),
↪ drawzero=True)
```

Adding X, Y, Z axes using the extra code feature:

```
sage: length = 1.5
sage: space = 0.3
sage: axes = ''
sage: axes += "\\draw[->, thick, black] (0,0,0) -- (%s, 0, 0);\n" % length
sage: axes += "\\draw[->, thick, black] (0,0,0) -- (0, %s, 0);\n" % length
sage: axes += "\\node at (%s,0,0) {$x$};\n" % (length + space)
sage: axes += "\\node at (0,%s,0) {$y$};\n" % (length + space)
sage: axes += "\\node at (0,0,%s) {$z$};\n" % (length + space)
sage: axes += "\\draw[->, thick, black] (0,0,0) -- (0, 0, %s);\n" % length
sage: cube = Patch([Face((0,0,0),1), Face((0,0,0),2), Face((0,0,0),3)])
sage: options = dict(scale=0.5,drawzero=True,extra_code_before=axes)
sage: s = cube.plot_tikz(**options)
sage: len(s)
986
sage: print(s)    #not tested
\begin{tikzpicture}
[x={(-0.433013cm,-0.250000cm)}, y={(0.433013cm,-0.250000cm)}, z={(0.000000cm,
↪0.500000cm)}]
\draw[->, thick, black] (0,0,0) -- (1.50000000000000, 0, 0);
\draw[->, thick, black] (0,0,0) -- (0, 1.50000000000000, 0);
\node at (1.80000000000000,0,0) {$x$};
\node at (0,1.80000000000000,0) {$y$};
\node at (0,0,1.80000000000000) {$z$};
\draw[->, thick, black] (0,0,0) -- (0, 0, 1.50000000000000);
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\node[circle,fill=black,draw=black,minimum size=1.5mm,inner sep=0pt] at (0,0,
↪0) {};
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> length = RealNumber('1.5')
>>> space = RealNumber('0.3')
>>> axes = ''
>>> axes += "\\draw[->, thick, black] (0,0,0) -- (%s, 0, 0);\n" % length
>>> axes += "\\draw[->, thick, black] (0,0,0) -- (0, %s, 0);\n" % length
>>> axes += "\\node at (%s,0,0) {$x$};\n" % (length + space)
>>> axes += "\\node at (0,%s,0) {$y$};\n" % (length + space)
>>> axes += "\\node at (0,0,%s) {$z$};\n" % (length + space)
>>> axes += "\\draw[->, thick, black] (0,0,0) -- (0, 0, %s);\n" % length
>>> cube = Patch([Face((Integer(0),Integer(0),Integer(0)),Integer(1)),␣
↪Face((Integer(0),Integer(0),Integer(0)),Integer(2)), Face((Integer(0),
↪Integer(0),Integer(0)),Integer(3))])
>>> options = dict(scale=RealNumber('0.5'),drawzero=True,extra_code_
↪before=axes)
```

(continues on next page)

(continued from previous page)

```
>>> s = cube.plot_tikz(**options)
>>> len(s)
986
>>> print(s)    #not tested
\begin{tikzpicture}
[x={(-0.433013cm,-0.250000cm)}, y={(0.433013cm,-0.250000cm)}, z={(0.000000cm,
↪0.500000cm)}]
\draw[->, thick, black] (0,0,0) -- (1.50000000000000, 0, 0);
\draw[->, thick, black] (0,0,0) -- (0, 1.50000000000000, 0);
\node at (1.80000000000000,0,0) {$x$};
\node at (0,1.80000000000000,0) {$y$};
\node at (0,0,1.80000000000000) {$z$};
\draw[->, thick, black] (0,0,0) -- (0, 0, 1.50000000000000);
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\node[circle,fill=black,draw=black,minimum size=1.5mm,inner sep=0pt] at (0,0,
↪0) {};
\end{tikzpicture}
```

repaint(*cmap='Set1'*)

Repaint all the faces of `self` from the given color map.

This only changes the colors of the faces of `self`.

INPUT:

- `cmap` – color map (default: `'Set1'`). It can be one of the following:
  - `string` – a coloring map; for available coloring map names type: `sorted(colormaps)`
  - `list` – list of colors to assign cyclically to the faces A list of a single color colors all the faces with the same color
  - `dict` – dictionary of face types mapped to colors, to color the faces according to their type
  - `{}`, the empty dict – shortcut for `{1:'red', 2:'green', 3:'blue'}`

EXAMPLES:

Using a color map:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: color = (0, 0, 0)
sage: P = Patch([Face((0,0,0),t,color) for t in [1,2,3]])
sage: for f in P: f.color()
RGB color (0.0, 0.0, 0.0)
RGB color (0.0, 0.0, 0.0)
RGB color (0.0, 0.0, 0.0)
sage: P.repaint()
```

(continues on next page)

(continued from previous page)

```
sage: next(iter(P)).color()     #random
RGB color (0.498..., 0.432..., 0.522...)
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> color = (Integer(0), Integer(0), Integer(0))
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t,color) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> for f in P: f.color()
RGB color (0.0, 0.0, 0.0)
RGB color (0.0, 0.0, 0.0)
RGB color (0.0, 0.0, 0.0)
>>> P.repaint()
>>> next(iter(P)).color()     #random
RGB color (0.498..., 0.432..., 0.522...)
```

Using a list of colors:

```
sage: P = Patch([Face((0,0,0),t,color) for t in [1,2,3]])
sage: P.repaint([(0.9, 0.9, 0.9), (0.65,0.65,0.65), (0.4,0.4,0.4)])
sage: for f in P: f.color()
RGB color (0.9, 0.9, 0.9)
RGB color (0.65, 0.65, 0.65)
RGB color (0.4, 0.4, 0.4)
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t,color) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P.repaint([(RealNumber('0.9'), RealNumber('0.9'), RealNumber('0.9')),␣
↪(RealNumber('0.65'),RealNumber('0.65'),RealNumber('0.65')), (RealNumber('0.4
↪'),RealNumber('0.4'),RealNumber('0.4'))])
>>> for f in P: f.color()
RGB color (0.9, 0.9, 0.9)
RGB color (0.65, 0.65, 0.65)
RGB color (0.4, 0.4, 0.4)
```

Using a dictionary to color faces according to their type:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.repaint({1:'black', 2:'yellow', 3:'green'})
sage: P.plot()                    #not tested
sage: P.repaint({})
sage: P.plot()                    #not tested
```

```
>>> from sage.all import *
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),t) for t in␣
↪[Integer(1),Integer(2),Integer(3)]])
>>> P.repaint({Integer(1):'black', Integer(2):'yellow', Integer(3):'green'})
>>> P.plot()                    #not tested
>>> P.repaint({})
>>> P.plot()                    #not tested
```

translate(*v*)

Return a translated copy of `self` by vector v.

INPUT:

- `v` – vector or tuple

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1), Face((1,2,0),3), Face((1,2,0),1)])
sage: P.translate([-1,-2,0])
Patch: [[(-1, -2, 0), 1]*, [(0, 0, 0), 1]*, [(0, 0, 0), 3]*]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),Integer(1)),␣
↪Face((Integer(1),Integer(2),Integer(0)),Integer(3)), Face((Integer(1),
↪Integer(2),Integer(0)),Integer(1))])
>>> P.translate([-Integer(1),-Integer(2),Integer(0)])
Patch: [[(-1, -2, 0), 1]*, [(0, 0, 0), 1]*, [(0, 0, 0), 3]*]
```

union(*other*)

Return a Patch consisting of the union of `self` and `other`.

INPUT:

- `other` – a Patch or a Face or a finite iterable of faces

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1), Face((0,0,0),2)])
sage: P.union(Face((1,2,3), 3))
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(1, 2, 3), 3]*]
sage: P.union([Face((1,2,3), 3), Face((2,3,3), 2)])
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(1, 2, 3), 3]*, [(2, 3, 3), 2]*]
```

```
>>> from sage.all import *
>>> from sage.combinat.e_one_star import Face, Patch
>>> P = Patch([Face((Integer(0),Integer(0),Integer(0)),Integer(1)),␣
↪Face((Integer(0),Integer(0),Integer(0)),Integer(2))])
>>> P.union(Face((Integer(1),Integer(2),Integer(3)), Integer(3)))
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(1, 2, 3), 3]*]
>>> P.union([Face((Integer(1),Integer(2),Integer(3)), Integer(3)),␣
↪Face((Integer(2),Integer(3),Integer(3)), Integer(2))])
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(1, 2, 3), 3]*, [(2, 3, 3), 2]*]
```

5.1.102 Enumerated sets and combinatorial objects

Todo

Proofread / point to the main classes rather than the modules

Categories

- `EnumeratedSets`, `FiniteEnumeratedSets`

Basic enumerated sets

- `Subsets`, `Combinations`
- `Arrangements`, `Tuples`
- `FiniteEnumeratedSet`
- `DisjointUnionEnumeratedSets`

Integer lists

- *Integer partitions* (see also: *Enumerated sets of partitions, tableaux, …*)
- *Integer compositions*
- `SignedCompositions`
- `IntegerListsLex`
- *Super Partitions*
- `IntegerVectors`
- `WeightedIntegerVectors()`
- `IntegerVectorsModPermutationGroup`
- *Parking Functions*
- *Non-Decreasing Parking Functions*
- *Sidon sets and their generalizations, Sidon g-sets*

Words

- `Words`
- *Subwords*
- *Necklaces*
- *Lyndon words*
- *Dyck Words*
- *De Bruijn sequences*
- *Shuffle product of iterables*

Permutations, …

- *Permutations*
- *Permutations (Cython file)*
- *Affine Permutations*
- `Arrangements`
- *Derangements*
- *Baxter permutations*

See also

- `SymmetricGroup`, `PermutationGroup()`, Catalog of permutation groups
- `FiniteSetMaps`
- *Integer vectors modulo the action of a permutation group*
- *Robinson-Schensted-Knuth correspondence*

Partitions, tableaux, ...

See: *Enumerated sets of partitions, tableaux, …*

Polyominoes

See: *Parallelogram Polyominoes*

Integer matrices, ...

- *Counting, generating, and manipulating nonnegative integer matrices*
- *Hadamard matrices*
- *Latin Squares*
- *Alternating Sign Matrices*
- *Six Vertex Model*
- *Similarity class types of matrices with entries in a finite field*
- *Restricted growth arrays*
- *Vector Partitions*

See also

- `MatrixSpace`
- Library of Interesting Groups

Subsets and set partitions

- `Subsets`, `Combinations`
- `PairwiseCompatibleSubsets`
- *Subsets satisfying a hereditary property*
- *Ordered Set Partitions*
- *Set Partitions*
- *Diagram and Partition Algebras*
- `OrderedMultisetPartitionsIntoSets`, `OrderedMultisetPartitionIntoSets`

Trees

- *Abstract Recursive Trees*
- *Ordered Rooted Trees*
- *Binary Trees*
- *Rooted (Unordered) Trees*

Enumerated sets related to graphs

- *Degree sequences*
- *Paths in Directed Acyclic Graphs*
- *Perfect matchings*

Backtracking solvers and generic enumerated sets

Todo

Do we want a separate section, possibly more proeminent, for backtracking solvers?

- `RecursivelyEnumeratedSet()`
- *`GenericBacktracker`*
- `sage.parallel.map_reduce`
- *Tiling Solver*
- *Exact Cover Problem via Dancing Links*
- *Dancing links C++ wrapper*
- *Combinatorial species*
- `IntegerListsLex`
- *`IntegerVectorsModPermutationGroup`*

Low level enumerated sets

- *Gray codes*

Misc enumerated sets

- *`GelfandTsetlinPattern`*, *`GelfandTsetlinPatterns`*
- *`KnutsonTaoPuzzleSolver`*
- `LatticePolytope()`

5.1.103 Tools for enumeration modulo the action of a permutation group

`sage.combinat.enumeration_mod_permgroup.`**`all_children`**`(`*v*, *max_part*`)`

Return all the children of an integer vector (`ClonableIntArray`) `v` in the tree of enumeration by lexicographic order. The children of an integer vector `v` whose entries have the sum n are all integer vectors of sum $n + 1$ which follow `v` in the lexicographic order.

That means this function adds 1 on the last nonzero entries and the following ones. For an integer vector v such that

$$v = [\dots, a, 0, 0] \text{ with } a \neq 0,$$

then, the list of children is

$$[[\dots, a+1, 0, 0], [\dots, a, 1, 0], [\dots, a, 0, 1]].$$

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import all_children
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: v = IncreasingIntArrays()([1,2,3,4])
sage: all_children(v, -1)
[[1, 2, 3, 5]]
```

```
>>> from sage.all import *
>>> from sage.combinat.enumeration_mod_permgroup import all_children
>>> from sage.structure.list_clone_demo import IncreasingIntArrays
>>> v = IncreasingIntArrays()([Integer(1),Integer(2),Integer(3),Integer(4)])
>>> all_children(v, -Integer(1))
[[1, 2, 3, 5]]
```

sage.combinat.enumeration_mod_permgroup.**canonical_children**(*sgs*, *v*, *max_part*)

Return the canonical children of the integer vector `v`. This function computes all children of the integer vector `v` via the function *all_children()* and returns from this list only these which are canonicals identified via the function *is_canonical()*.

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import canonical_children
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: sgs = G.strong_generating_system()
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: canonical_children(sgs, IA([1,2,3,5]), -1)
[]
```

```
>>> from sage.all import *
>>> from sage.combinat.enumeration_mod_permgroup import canonical_children
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> sgs = G.strong_generating_system()
>>> from sage.structure.list_clone_demo import IncreasingIntArrays
>>> IA = IncreasingIntArrays()
>>> canonical_children(sgs, IA([Integer(1),Integer(2),Integer(3),Integer(5)]), -
↪Integer(1))
[]
```

sage.combinat.enumeration_mod_permgroup.**canonical_representative_of_orbit_of**(*sgs*, *v*)

Return the maximal vector for the lexicographic order living in the orbit of v under the action of the permutation group whose strong generating system is `sgs`. The maximal vector is also called "canonical". Hence, this method returns the canonical vector inside the orbit of v. If v is already canonical, the method returns v.

Let G to be the permutation group which admits `sgs` as a strong generating system. An integer vector v is said to be canonical under the action of G if and only if:

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import canonical_
↪representative_of_orbit_of
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: sgs = G.strong_generating_system()
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: canonical_representative_of_orbit_of(sgs, IA([1,2,3,5]))
[5, 1, 2, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.enumeration_mod_permgroup import canonical_representative_
↪of_orbit_of
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> sgs = G.strong_generating_system()
>>> from sage.structure.list_clone_demo import IncreasingIntArrays
>>> IA = IncreasingIntArrays()
>>> canonical_representative_of_orbit_of(sgs, IA([Integer(1),Integer(2),
↪Integer(3),Integer(5)]))
[5, 1, 2, 3]
```

`sage.combinat.enumeration_mod_permgroup.`**`is_canonical`**`(`*sgs*, *v*`)`

Return `True` if the integer vector v is maximal with respect to the lexicographic order in its orbit under the action of the permutation group whose strong generating system is `sgs`. Such vectors are said to be canonical.

Let G to be the permutation group which admit `sgs` as a strong generating system. An integer vector v is said to be canonical under the action of G if and only if:

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import is_canonical
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: sgs = G.strong_generating_system()
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: is_canonical(sgs, IA([1,2,3,6]))
False
```

```
>>> from sage.all import *
>>> from sage.combinat.enumeration_mod_permgroup import is_canonical
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> sgs = G.strong_generating_system()
>>> from sage.structure.list_clone_demo import IncreasingIntArrays
>>> IA = IncreasingIntArrays()
>>> is_canonical(sgs, IA([Integer(1),Integer(2),Integer(3),Integer(6)]))
False
```

sage.combinat.enumeration_mod_permgroup.**lex_cmp**(*v1*, *v2*)

Lexicographic comparison of `ClonableIntArray`.

INPUT:

- `v1`, `v2` – two instances of `ClonableIntArray`

OUTPUT:

`-1,0,1`, depending on whether v_1 is lexicographically smaller, equal, or greater than v_2.

EXAMPLES:

```
sage: I =  IntegerVectorsModPermutationGroup(SymmetricGroup(5),5)
sage: I =  IntegerVectorsModPermutationGroup(SymmetricGroup(5))
sage: J =  IntegerVectorsModPermutationGroup(SymmetricGroup(6))
sage: v1 = I([2,3,1,2,3], check=False)
sage: v2 = I([2,3,2,3,2], check=False)
sage: v3 = J([2,3,1,2,3,1], check=False)
sage: from sage.combinat.enumeration_mod_permgroup import lex_cmp
sage: lex_cmp(v1, v1)
0
sage: lex_cmp(v1, v2)
-1
sage: lex_cmp(v2, v1)
1
sage: lex_cmp(v1, v3)
-1
sage: lex_cmp(v3, v1)
1
```

```
>>> from sage.all import *
>>> I =  IntegerVectorsModPermutationGroup(SymmetricGroup(Integer(5)),Integer(5))
>>> I =  IntegerVectorsModPermutationGroup(SymmetricGroup(Integer(5)))
>>> J =  IntegerVectorsModPermutationGroup(SymmetricGroup(Integer(6)))
>>> v1 = I([Integer(2),Integer(3),Integer(1),Integer(2),Integer(3)], check=False)
>>> v2 = I([Integer(2),Integer(3),Integer(2),Integer(3),Integer(2)], check=False)
>>> v3 = J([Integer(2),Integer(3),Integer(1),Integer(2),Integer(3),Integer(1)],␣
↪check=False)
>>> from sage.combinat.enumeration_mod_permgroup import lex_cmp
>>> lex_cmp(v1, v1)
0
>>> lex_cmp(v1, v2)
-1
>>> lex_cmp(v2, v1)
1
>>> lex_cmp(v1, v3)
-1
>>> lex_cmp(v3, v1)
1
```

sage.combinat.enumeration_mod_permgroup.**lex_cmp_partial**(*v1*, *v2*, *step*)

Partial comparison of the two lists according the lexicographic order. It compares the `step`-th first entries.

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import lex_cmp_partial
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: lex_cmp_partial(IA([0,1,2,3]),IA([0,1,2,4]),3)
0
sage: lex_cmp_partial(IA([0,1,2,3]),IA([0,1,2,4]),4)
-1
```

```
>>> from sage.all import *
>>> from sage.combinat.enumeration_mod_permgroup import lex_cmp_partial
>>> from sage.structure.list_clone_demo import IncreasingIntArrays
>>> IA = IncreasingIntArrays()
>>> lex_cmp_partial(IA([Integer(0),Integer(1),Integer(2),Integer(3)]),
↪IA([Integer(0),Integer(1),Integer(2),Integer(4)]),Integer(3))
0
>>> lex_cmp_partial(IA([Integer(0),Integer(1),Integer(2),Integer(3)]),
↪IA([Integer(0),Integer(1),Integer(2),Integer(4)]),Integer(4))
-1
```

sage.combinat.enumeration_mod_permgroup.**orbit**(*sgs*, *v*)

Return the orbit of the integer vector `v` under the action of the permutation group whose strong generating system is `sgs`.

NOTE:

The returned orbit is a set. In the doctests, we convert it into a sorted list.

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import orbit
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: sgs = G.strong_generating_system()
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: sorted(orbit(sgs, IA([1,2,3,4])))
[[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.enumeration_mod_permgroup import orbit
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> sgs = G.strong_generating_system()
>>> from sage.structure.list_clone_demo import IncreasingIntArrays
>>> IA = IncreasingIntArrays()
>>> sorted(orbit(sgs, IA([Integer(1),Integer(2),Integer(3),Integer(4)])))
[[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]]
```

5.1.104 Compute Bell and Uppuluri-Carpenter numbers

AUTHORS:

- Nick Alexander

sage.combinat.expnums.**expnums**(*n*, *aa*)

Compute the first n exponential numbers around aa, starting with the zero-th.

INPUT:

- `n` – C machine int
- `aa` – C machine int

OUTPUT: list of length n

ALGORITHM: We use the same integer addition algorithm as GAP. This is an extension of Bell's triangle to the general case of exponential numbers. The recursion performs $O(n^2)$ additions, but the running time is dominated by the cost of the last integer addition, because the growth of the integer results of partial computations is exponential in n. The algorithm stores $O(n)$ integers, but each is exponential in n.

EXAMPLES:

```
sage: expnums(10, 1)
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

```
>>> from sage.all import *
>>> expnums(Integer(10), Integer(1))
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

```
sage: expnums(10, -1)
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]
```

```
>>> from sage.all import *
>>> expnums(Integer(10), -Integer(1))
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]
```

```
sage: expnums(1, 1)
[1]
sage: expnums(0, 1)
[]
sage: expnums(-1, 0)
[]
```

```
>>> from sage.all import *
>>> expnums(Integer(1), Integer(1))
[1]
>>> expnums(Integer(0), Integer(1))
[]
>>> expnums(-Integer(1), Integer(0))
[]
```

AUTHORS:

- Nick Alexander

`sage.combinat.expnums.`**`expnums2`**`(`*n*, *aa*`)`

A vanilla python (but compiled via Cython) implementation of expnums.

We Compute the first n exponential numbers around aa, starting with the zero-th.

EXAMPLES:

```
sage: from sage.combinat.expnums import expnums2
sage: expnums2(10, 1)
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

```
>>> from sage.all import *
>>> from sage.combinat.expnums import expnums2
>>> expnums2(Integer(10), Integer(1))
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

5.1.105 Families

This is a backward compatibility stub. Use `sage.sets.family` instead.

5.1.106 Brent Yorgey's fast algorithm for integer vector (multiset) partitions.

ALGORITHM:

Brent Yorgey, Generating Multiset Partitions, The Monad Reader, Issue 8, September 2007, p. 5.

https://wiki.haskell.org/The_Monad.Reader/Previous_issues

AUTHORS:

- D. K. Sunko (2020-02-19): initial version
- F. Chapoton (2020-02-22): conversion to iterators and shorter doctests and doc tweaks
- T. Scrimshaw (2020-03-06): Cython optimizations and doc tweaks

`sage.combinat.fast_vector_partitions.`**`fast_vector_partitions`**`(`*v*, *min_vals=None*`)`

Brent Yorgey's fast algorithm for integer vector (multiset) partitions.

INPUT:

- `v` – list of nonnegative integers, understood as the vector to be partitioned
- `min_vals` – (optional) list of nonnegative integers, of same length as `v`

OUTPUT: list of lists, each representing a vector partition of `v`

If `min_vals` is given, only partitions with parts `p >= min_vals` in the lexicographic ordering will appear.

If `min_vals` is given and `len(min_vals) != len(v)`, an error is raised.

EXAMPLES:

The older the computer, the more impressive the comparison:

```
sage: from sage.combinat.fast_vector_partitions import fast_vector_partitions
sage: fastvparts = list(fast_vector_partitions([3, 3, 3]))
sage: vparts = list(VectorPartitions([3, 3, 3]))
sage: vparts == fastvparts[::-1]
True
sage: len(fastvparts)
686
sage: list(fast_vector_partitions([1, 2, 3], min_vals=[0, 1, 1]))
[[[1, 2, 3]],
 [[0, 2, 3], [1, 0, 0]],
 [[0, 2, 2], [1, 0, 1]],
```

(continues on next page)

(continued from previous page)

```
 [[0, 2, 1], [1, 0, 2]],
 [[0, 2, 0], [1, 0, 3]],
 [[0, 1, 3], [1, 1, 0]],
 [[0, 1, 2], [1, 1, 1]],
 [[0, 1, 1], [1, 1, 2]],
 [[0, 1, 1], [0, 1, 2], [1, 0, 0]],
 [[0, 1, 1], [0, 1, 1], [1, 0, 1]]]
sage: L1 = list(fast_vector_partitions([5, 7, 6], min_vals=[1, 3, 2]))
sage: L1 == list(VectorPartitions([5, 7, 6], min=[1, 3, 2]))[::-1]
True
```

```
>>> from sage.all import *
>>> from sage.combinat.fast_vector_partitions import fast_vector_partitions
>>> fastvparts = list(fast_vector_partitions([Integer(3), Integer(3),␣
↪Integer(3)]))
>>> vparts = list(VectorPartitions([Integer(3), Integer(3), Integer(3)]))
>>> vparts == fastvparts[::-Integer(1)]
True
>>> len(fastvparts)
686
>>> list(fast_vector_partitions([Integer(1), Integer(2), Integer(3)], min_
↪vals=[Integer(0), Integer(1), Integer(1)]))
[[[1, 2, 3]],
 [[0, 2, 3], [1, 0, 0]],
 [[0, 2, 2], [1, 0, 1]],
 [[0, 2, 1], [1, 0, 2]],
 [[0, 2, 0], [1, 0, 3]],
 [[0, 1, 3], [1, 1, 0]],
 [[0, 1, 2], [1, 1, 1]],
 [[0, 1, 1], [1, 1, 2]],
 [[0, 1, 1], [0, 1, 2], [1, 0, 0]],
 [[0, 1, 1], [0, 1, 1], [1, 0, 1]]]
>>> L1 = list(fast_vector_partitions([Integer(5), Integer(7), Integer(6)], min_
↪vals=[Integer(1), Integer(3), Integer(2)]))
>>> L1 == list(VectorPartitions([Integer(5), Integer(7), Integer(6)],␣
↪min=[Integer(1), Integer(3), Integer(2)]))[::-Integer(1)]
True
```

Note

The partitions are returned as an iterator.

In this documentation, `a <|= b` means `a[i] <= b[i]` for all `i` (notation following B. Yorgey's paper). It is the monomial partial ordering in Dickson's lemma: `a <|= b` iff `x^a` divides `x^b` as monomials.

Warning

The ordering of the partitions is reversed with respect to the output of Sage class *VectorPartitions*.

`sage.combinat.fast_vector_partitions.`**`recursive_vector_partitions`**`(v, vL)`

Iterate over a lexicographically ordered list of lists, each list representing a vector partition of `v`, such that no part of any partition is lexicographically smaller than `vL`.

Internal part of *fast_vector_partitions()*.

INPUT:

- `v` – list of nonnegative integers, understood as a vector
- `vL` – list of nonnegative integers, understood as a vector

EXAMPLES:

```
sage: from sage.combinat.fast_vector_partitions import recursive_vector_partitions
sage: list(recursive_vector_partitions([2, 2, 2],[1, 1, 1]))
[[[2, 2, 2]], [[1, 1, 1], [1, 1, 1]]]
sage: list(recursive_vector_partitions([2, 2, 2],[1, 1, 0]))
[[[2, 2, 2]], [[1, 1, 1], [1, 1, 1]], [[1, 1, 0], [1, 1, 2]]]
sage: list(recursive_vector_partitions([2, 2, 2],[1, 0, 1]))
[[[2, 2, 2]],
 [[1, 1, 1], [1, 1, 1]],
 [[1, 1, 0], [1, 1, 2]],
 [[1, 0, 2], [1, 2, 0]],
 [[1, 0, 1], [1, 2, 1]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.fast_vector_partitions import recursive_vector_partitions
>>> list(recursive_vector_partitions([Integer(2), Integer(2), Integer(2)],
↪[Integer(1), Integer(1), Integer(1)]))
[[[2, 2, 2]], [[1, 1, 1], [1, 1, 1]]]
>>> list(recursive_vector_partitions([Integer(2), Integer(2), Integer(2)],
↪[Integer(1), Integer(1), Integer(0)]))
[[[2, 2, 2]], [[1, 1, 1], [1, 1, 1]], [[1, 1, 0], [1, 1, 2]]]
>>> list(recursive_vector_partitions([Integer(2), Integer(2), Integer(2)],
↪[Integer(1), Integer(0), Integer(1)]))
[[[2, 2, 2]],
 [[1, 1, 1], [1, 1, 1]],
 [[1, 1, 0], [1, 1, 2]],
 [[1, 0, 2], [1, 2, 0]],
 [[1, 0, 1], [1, 2, 1]]]
```

`sage.combinat.fast_vector_partitions.`**`recursive_within_from_to`**(*m*, *s*, *e*, *useS*, *useE*)

Iterate over a lexicographically ordered list of lists `v` satisfying `e <= v <= s` and `v <|= m` as vectors.

Internal part of *fast_vector_partitions()*.

INPUT:

- `m` – list of nonnegative integers, understood as a vector
- `s` – list of nonnegative integers, understood as a vector
- `e` – list of nonnegative integers, understood as a vector
- `useS` – boolean
- `useE` – boolean

EXAMPLES:

```
sage: from sage.combinat.fast_vector_partitions import recursive_within_from_to
sage: list(recursive_within_from_to([1, 2, 3],[1, 2, 2],[1, 1, 1],True,True))
[[1, 2, 2], [1, 2, 1], [1, 2, 0], [1, 1, 3], [1, 1, 2], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.fast_vector_partitions import recursive_within_from_to
>>> list(recursive_within_from_to([Integer(1), Integer(2), Integer(3)],
↪[Integer(1), Integer(2), Integer(2)],[Integer(1), Integer(1), Integer(1)],True,
↪True))
[[1, 2, 2], [1, 2, 1], [1, 2, 0], [1, 1, 3], [1, 1, 2], [1, 1, 1]]
```

Note: The flags `useS` and `useE` are used to implement the condition efficiently. Because testing it loops over the vector, re-testing at each step as the vector is parsed is inefficient: all but the last comparison have been done cumulatively already. This code tests only for the last one, using the flags to accumulate information from previous calls.

Warning: Expects to be called with `s <|= m`.

Expects to be called first with `useS == useE == True`.

`sage.combinat.fast_vector_partitions.`**`within_from_to`**`(m, s, e)`

Iterate over a lexicographically ordered list of lists `v` satisfying `e <= v <= s` and `v <|= m` as vectors.

Internal part of `fast_vector_partitions()`.

INPUT:

- `m` – list of nonnegative integers, understood as a vector
- `s` – list of nonnegative integers, understood as a vector
- `e` – list of nonnegative integers, understood as a vector

EXAMPLES:

```
sage: from sage.combinat.fast_vector_partitions import within_from_to
sage: list(within_from_to([1, 2, 3], [1, 2, 2], [1, 1, 1]))
[[1, 2, 2], [1, 2, 1], [1, 2, 0], [1, 1, 3], [1, 1, 2], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.fast_vector_partitions import within_from_to
>>> list(within_from_to([Integer(1), Integer(2), Integer(3)], [Integer(1),
↪Integer(2), Integer(2)], [Integer(1), Integer(1), Integer(1)]))
[[1, 2, 2], [1, 2, 1], [1, 2, 0], [1, 1, 3], [1, 1, 2], [1, 1, 1]]
```

Note: The input `s` will be "clipped" internally if it does not satisfy the condition `s <|= m`.

To understand the input check, some line art is helpful. Assume that `(a,b)` are the two least significant coordinates of some vector. Say:

```
e = (2,3), s = (7,6), m = (9,8).
```

In the figure, these values are denoted by `E`, `S`, and `M`, while the letter `X` stands for all other allowed values of `v = (a,b)`:

```
b ^
  |
8 --------X---X---X---X---X-----------M
  |                                   |
7 -       X   X   X   X   X           |
  |                                   |
6 -       X   X   X   X   X   S       |
  |                                   |
5 -       X   X   X   X   X   X       |
  |                                   |
4 -       X   X   X   X   X   X       |
  |                                   |
3 -       E   X   X   X   X   X       |
  |                                   |
2 -           X   X   X   X   X       |
  |                                   |
1 -           X   X   X   X   X       |
  |                                   |
0 ----|---|---X---X---X---X---X---|---|--->
  0   1   2   3   4   5   6   7   8   9   a
```

If `S` moves horizontally, the full-height columns fill the box in until `S` reaches `M`, at which point it remains the limit in the b-direction as it moves out of the box, while M takes over as the limit in the a-direction, so the `M`-column remains filled only up to `S`, no matter how much `S` moves further to the right.

If `S` moves vertically, its column will be filled to the top of the box, but it remains the relevant limit in the a-direction, while `M` takes over in the b-direction as `S` goes out of the box upwards.

Both behaviors are captured by using the smaller coordinate of `S` and `M`, whenever `S` is outside the box defined by M. The input will be "clipped" accordingly in that case.

Warning

The "clipping" behavior is transparent to the user, but may be puzzling when comparing outputs with the function *recursive_within_from_to()* which has no input protection.

5.1.107 Fully commutative elements of Coxeter groups

An element w in a Coxeter system (W,S) is fully commutative (FC) if every two reduced words of w can be related by a sequence of only commutation relations, i.e., relations of the form $st = ts$ where s, t are commuting generators in S. See [Ste1996].

Authors:

- Chase Meadors, Tianyuan Xu (2020): Initial version

Acknowledgements

A draft of this code was written during an REU project at University of Colorado Boulder. We thank Rachel Castro, Joel Courtney, Thomas Magnuson and Natalie Schoenhals for their contribution to the project and the code.

class sage.combinat.fully_commutative_elements.**FullyCommutativeElement**

Bases: NormalizedClonableList

A fully commutative (FC) element in a Coxeter system.

An element w in a Coxeter system (W,S) is fully commutative (FC) if every two reduced word of w can be related by a sequence of only commutation relations, i.e., relations of the form $st = ts$ where s, t are commuting generators in S.

Every FC element has a canonical reduced word called its Cartier–Foata form. See [Gre2006]. We will normalize each FC element to this form.

check()

Called automatically when an element is created.

EXAMPLES:

```
sage: CoxeterGroup(['A', 3]).fully_commutative_elements()([1, 2]) # indirect␣
↪doctest
[1, 2]
sage: CoxeterGroup(['A', 3]).fully_commutative_elements()([1, 2, 1]) #␣
↪indirect doctest
Traceback (most recent call last):
...
ValueError: the input is not a reduced word of a fully commutative element
```

```
>>> from sage.all import *
>>> CoxeterGroup(['A', Integer(3)]).fully_commutative_elements()([Integer(1),␣
↪Integer(2)]) # indirect doctest
[1, 2]
>>> CoxeterGroup(['A', Integer(3)]).fully_commutative_elements()([Integer(1),␣
↪Integer(2), Integer(1)]) # indirect doctest
Traceback (most recent call last):
...
ValueError: the input is not a reduced word of a fully commutative element
```

coset_decomposition(*J*, *side='left'*)

Return the coset decomposition of `self` with respect to the parabolic subgroup generated by `J`.

INPUT:

- `J` – subset of the generating set S of the Coxeter system
- `side` – string (default: `'left'`); if the value is set to 'right', then the function returns the tuple (w'^J, w'_J) from the coset decomposition $w = w'^J \cdot w'_J$ of w with respect to J

OUTPUT:

The tuple of elements (w_J, w^J) such that $w = w_J \cdot w^J$, w_J is generated by the elements in J, and w^J has no left descent from J. This tuple is unique and satisfies the equation $\ell(w) = \ell(w_J) + \ell(w^J)$, where ℓ denotes Coxeter length, by general theory; see Proposition 2.4.4 of [BB2005].

EXAMPLES:

```
sage: FC = CoxeterGroup(['B', 6]).fully_commutative_elements()
sage: w = FC([1, 6, 2, 5, 4, 6, 5])
sage: w.coset_decomposition({1})
([1], [6, 2, 5, 4, 6, 5])
sage: w.coset_decomposition({1}, side = 'right')
([1, 6, 2, 5, 4, 6, 5], [])
sage: w.coset_decomposition({5, 6})
([6, 5, 6], [1, 2, 4, 5])
sage: w.coset_decomposition({5, 6}, side='right')
([1, 6, 2, 5, 4], [6, 5])
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['B', Integer(6)]).fully_commutative_elements()
>>> w = FC([Integer(1), Integer(6), Integer(2), Integer(5), Integer(4),␣
→Integer(6), Integer(5)])
>>> w.coset_decomposition({Integer(1)})
([1], [6, 2, 5, 4, 6, 5])
>>> w.coset_decomposition({Integer(1)}, side = 'right')
([1, 6, 2, 5, 4, 6, 5], [])
>>> w.coset_decomposition({Integer(5), Integer(6)})
([6, 5, 6], [1, 2, 4, 5])
>>> w.coset_decomposition({Integer(5), Integer(6)}, side='right')
([1, 6, 2, 5, 4], [6, 5])
```

Note

The factor w_J of the coset decomposition $w = w_J \cdot w^J$ can be obtained by greedily "pulling left descents of w that are in J to the left"; see the proof of [BB2005]. This greedy algorithm works for all elements in Coxeter group, but it becomes especially simple for FC elements because descents are easier to find for FC elements.

descents(*side='left'*)

Obtain the set of descents on the appropriate side of `self`.

INPUT:

- `side` – string (default: `'left'`); if set to 'right', find the right descents

A generator s is called a left or right descent of an element w if $l(sw)$ or $l(ws)$ is smaller than $l(w)$, respectively. If w is FC, then s is a left descent of w if and only if s appears to in the word and every generator to the left of the leftmost s in the word commutes with s. A similar characterization exists for right descents of FC elements.

EXAMPLES:

```
sage: FC = CoxeterGroup(['B', 5]).fully_commutative_elements()
sage: w = FC([1, 4, 3, 5, 2, 4, 3])
sage: sorted(w.descents())
[1, 4]
sage: w.descents(side='right')
{3}
sage: FC = CoxeterGroup(['A', 5]).fully_commutative_elements()
```

(continues on next page)

(continued from previous page)

```
sage: sorted(FC([1, 4, 3, 5, 2, 4, 3]).descents())
[1, 4]
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['B', Integer(5)]).fully_commutative_elements()
>>> w = FC([Integer(1), Integer(4), Integer(3), Integer(5), Integer(2),␣
↪Integer(4), Integer(3)])
>>> sorted(w.descents())
[1, 4]
>>> w.descents(side='right')
{3}
>>> FC = CoxeterGroup(['A', Integer(5)]).fully_commutative_elements()
>>> sorted(FC([Integer(1), Integer(4), Integer(3), Integer(5), Integer(2),␣
↪Integer(4), Integer(3)]).descents())
[1, 4]
```

See also

find_descent()

find_descent(*s*, *side='left'*)

Check if s is a descent of self and find its position if so.

A generator s is called a left or right descent of an element w if $l(sw)$ or $l(ws)$ is smaller than $l(w)$, respectively. If w is FC, then s is a left descent of w if and only if s appears to in the word and every generator to the left of the leftmost s in the word commutes with s. A similar characterization exists for right descents of FC elements.

INPUT:

- s – integer representing a generator of the Coxeter system
- side – string (default: 'left'); if the argument is set to 'right', the function checks if s is a right descent of self and finds the index of the rightmost occurrence of s if so

OUTPUT:

Determine if the generator s is a left descent of self; return the index of the leftmost occurrence of s in self if so and return None if not.

EXAMPLES:

```
sage: FC = CoxeterGroup(['B', 5]).fully_commutative_elements()
sage: w = FC([1, 4, 3, 5, 2, 4, 3])
sage: w.find_descent(1)
0
sage: w.find_descent(1, side='right')

sage: w.find_descent(4)
1
sage: w.find_descent(4, side='right')

sage: w.find_descent(3)
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['B', Integer(5)]).fully_commutative_elements()
>>> w = FC([Integer(1), Integer(4), Integer(3), Integer(5), Integer(2),␣
→Integer(4), Integer(3)])
>>> w.find_descent(Integer(1))
0
>>> w.find_descent(Integer(1), side='right')
<BLANKLINE>
>>> w.find_descent(Integer(4))
1
>>> w.find_descent(Integer(4), side='right')
<BLANKLINE>
>>> w.find_descent(Integer(3))
<BLANKLINE>
```

group_element()

Get the actual element of the Coxeter group associated with `self.parent()` corresponding to `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A', 3])
sage: FC = W.fully_commutative_elements()
sage: x = FC([1, 2])
sage: x.group_element()
[ 0 -1  1]
[ 1 -1  1]
[ 0  0  1]
sage: x.group_element() in W
True
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A', Integer(3)])
>>> FC = W.fully_commutative_elements()
>>> x = FC([Integer(1), Integer(2)])
>>> x.group_element()
[ 0 -1  1]
[ 1 -1  1]
[ 0  0  1]
>>> x.group_element() in W
True
```

has_descent(*s*, *side='left'*)

Determine if `s` is a descent on the appropriate side of `self`.

INPUT:

- `side` – string (default: `'left'`); if set to ‘right’, determine if `self` has `s` as a right descent

OUTPUT: boolean

EXAMPLES:

```
sage: FC = CoxeterGroup(['B', 5]).fully_commutative_elements()
sage: w = FC([1, 4, 3, 5, 2, 4, 3])
sage: w.has_descent(1)
```

(continues on next page)

(continued from previous page)

```
True
sage: w.has_descent(1, side='right')
False
sage: w.has_descent(4)
True
sage: w.has_descent(4, side='right')
False
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['B', Integer(5)]).fully_commutative_elements()
>>> w = FC([Integer(1), Integer(4), Integer(3), Integer(5), Integer(2),␣
↪Integer(4), Integer(3)])
>>> w.has_descent(Integer(1))
True
>>> w.has_descent(Integer(1), side='right')
False
>>> w.has_descent(Integer(4))
True
>>> w.has_descent(Integer(4), side='right')
False
```

See also

find_descent()

heap(***kargs*)

Create the heap poset of `self`.

The heap of an FC element w is a labeled poset that can be defined from any reduced word of w. Different reduced words yield isomorphic labeled posets, so the heap is well defined.

Heaps are very useful for visualizing and studying FC elements; see, for example, [Ste1996] and [GX2020].

INPUT:

- `self` – list; a reduced word $w = s_0 ... s_{k-1}$ of an FC element
- `one_index` – boolean (default: `False`); setting the value to True will change the underlying set of the poset to $\{1, 2, \dots, n\}$
- `display_labeling` – boolean (default: `False`); setting the value to `True` will display the label s_i for each element i of the poset

OUTPUT:

A labeled poset where the underlying set is $\{0, 1, ..., k - 1\}$ and where each element i carries s_i as its label. The partial order $\prec$ on the poset is defined by declaring $i \prec j$ if $i < j$ and $m(s_i, s_j) \neq 2$.

EXAMPLES:

```
sage: FC = CoxeterGroup(['A', 5]).fully_commutative_elements()
sage: FC([1, 4, 3, 5, 2, 4]).heap().cover_relations()
[[1, 2], [1, 3], [2, 5], [2, 4], [3, 5], [0, 4]]
sage: FC([1, 4, 3, 5, 4, 2]).heap(one_index=True).cover_relations()
[[2, 3], [2, 4], [3, 6], [3, 5], [4, 6], [1, 5]]
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['A', Integer(5)]).fully_commutative_elements()
>>> FC([Integer(1), Integer(4), Integer(3), Integer(5), Integer(2),␣
→Integer(4)]).heap().cover_relations()
[[1, 2], [1, 3], [2, 5], [2, 4], [3, 5], [0, 4]]
>>> FC([Integer(1), Integer(4), Integer(3), Integer(5), Integer(4),␣
→Integer(2)]).heap(one_index=True).cover_relations()
[[2, 3], [2, 4], [3, 6], [3, 5], [4, 6], [1, 5]]
```

is_fully_commutative()

Check if `self` is the reduced word of an FC element.

To check if `self` is FC, we use the well-known characterization that an element w in a Coxeter system (W, S) is FC if and only if for every pair of generators $s, t \in S$ for which $m(s, t) > 2$, no reduced word of w contains the 'braid' word $sts...$ of length $m(s, t)$ as a contiguous subword. See [Ste1996].

`check()` is an alias of this method, and is called automatically when an element is created.

EXAMPLES:

```
sage: FC = CoxeterGroup(['A', 3]).fully_commutative_elements()
sage: x = FC([1, 2]); x.is_fully_commutative()
True
sage: x = FC.element_class(FC, [1, 2, 1], check=False); x.is_fully_
→commutative()
False
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['A', Integer(3)]).fully_commutative_elements()
>>> x = FC([Integer(1), Integer(2)]); x.is_fully_commutative()
True
>>> x = FC.element_class(FC, [Integer(1), Integer(2), Integer(1)],␣
→check=False); x.is_fully_commutative()
False
```

n_value()

Calculate the n-value of `self`.

The *n-value* of a fully commutative element is the *width* (length of any longest antichain) of its heap. The n-value is important as it coincides with Lusztig's a-value for FC elements in all Weyl and affine Weyl groups as well as so-called star-reducible groups; see [GX2020].

EXAMPLES:

```
sage: FC = CoxeterGroup(['A', 5]).fully_commutative_elements()
sage: FC([1,3]).n_value()
2
sage: FC([1,2,3]).n_value()
1
sage: FC([1,3,2]).n_value()
2
sage: FC([1,3,2,5]).n_value()
3
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['A', Integer(5)]).fully_commutative_elements()
>>> FC([Integer(1),Integer(3)]).n_value()
2
>>> FC([Integer(1),Integer(2),Integer(3)]).n_value()
1
>>> FC([Integer(1),Integer(3),Integer(2)]).n_value()
2
>>> FC([Integer(1),Integer(3),Integer(2),Integer(5)]).n_value()
3
```

normalize()

Mutate `self` into Cartier-Foata normal form.

EXAMPLES:

The following reduced words express the same FC elements in B_5:

```
sage: FC = CoxeterGroup(['B', 5]).fully_commutative_elements()
sage: FC([1, 4, 3, 5, 2, 4, 3]) # indirect doctest
[1, 4, 3, 5, 2, 4, 3]
sage: FC([4, 1, 3, 5, 2, 4, 3]) # indirect doctest
[1, 4, 3, 5, 2, 4, 3]
sage: FC([4, 3, 1, 5, 4, 2, 3]) # indirect doctest
[1, 4, 3, 5, 2, 4, 3]
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['B', Integer(5)]).fully_commutative_elements()
>>> FC([Integer(1), Integer(4), Integer(3), Integer(5), Integer(2),
↪Integer(4), Integer(3)]) # indirect doctest
[1, 4, 3, 5, 2, 4, 3]
>>> FC([Integer(4), Integer(1), Integer(3), Integer(5), Integer(2),
↪Integer(4), Integer(3)]) # indirect doctest
[1, 4, 3, 5, 2, 4, 3]
>>> FC([Integer(4), Integer(3), Integer(1), Integer(5), Integer(4),
↪Integer(2), Integer(3)]) # indirect doctest
[1, 4, 3, 5, 2, 4, 3]
```

Note

The Cartier–Foata form of a reduced word of an FC element w can be found recursively by repeatedly moving left descents of elements to the left and ordering the left descents from small to large. In the above example, the left descents of the element are 4 and 1, therefore the Cartier–Foata form of the element is the concatenation of [1,4] with the Cartier–Foata form of the remaining part of the word. See [Gre2006].

See also

descents()

plot_heap()

Display the Hasse diagram of the heap of `self`.

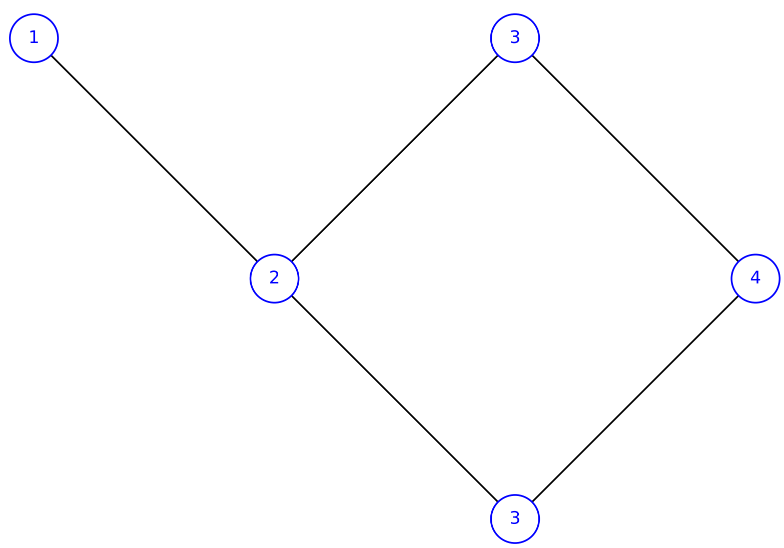
The Hasse diagram is rendered in the lattice $S \times \mathbf{N}$, with every element i in the poset drawn as a point labelled by its label s_i. Every point is placed in the column for its label at a certain level. The levels start at 0 and the level k of an element i is the maximal number k such that the heap contains a chain $i_0 \prec i_1 \prec ... \prec i_k$ where $i_k = i$. See [Ste1996] and [GX2020].

OUTPUT: GraphicsObject

EXAMPLES:

```
sage: FC = CoxeterGroup(['B', 5]).fully_commutative_elements()
sage: FC([3,2,4,3,1]).plot_heap()                                     #␣
↪needs sage.plot
Graphics object consisting of 15 graphics primitives
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['B', Integer(5)]).fully_commutative_elements()
>>> FC([Integer(3),Integer(2),Integer(4),Integer(3),Integer(1)]).plot_heap() ␣
↪                                   # needs sage.plot
Graphics object consisting of 15 graphics primitives
```



star_operation(*J*, *direction*, *side='left'*)

Apply a star operation on `self` relative to two noncommuting generators.

Star operations were first defined on elements of Coxeter groups by Kazhdan and Lusztig in [KL1979] with respect to pair of generators s, t such that $m(s, t) = 3$. Later, Lusztig generalized the definition in [Lus1985], via coset decompositions, to allow star operations with respect to any pair of generators s, t such that $m(s, t) \geq 3$. Given such a pair, we can potentially perform four types of star operations corresponding to all combinations of a 'direction' and a 'side': upper left, lower left, upper right and lower right; see [Gre2006].

Let w be an element in W and let J be any pair $\{s, t\}$ of noncommuting generators in S. Consider the coset decomposition $w = w_J \cdot {}^J w$ of w relative to J. Then an upper left star operation is defined on w if and only if $1 \leq l(w_J) \leq m(s, t) - 2$; when this is the case, the operation returns $x \cdot w_J \cdot w^J$ where x is the letter J different from the leftmost letter of w_J. A lower left star operation is defined on w if and only if $2 \leq l(w_J) \leq m(s, t) - 1$; when this is the case, the operation removes the leftmost letter of w_J from w. Similar facts hold for right star operations. See [Gre2006].

The facts of the previous paragraph hold in general, even if w is not FC. Note that if f is a star operation of any kind, then for every element $w \in W$, the elements w and $f(w)$ are either both FC or both not FC.

INPUT:

- `J` – set of two integers representing two noncommuting generators of the Coxeter system
- `direction` – string, `'upper'` or `'lower'`; the function performs an upper or lower star operation according to `direction`
- `side` – string (default: `'left'`); if this is set to 'right', the function performs a right star operation

OUTPUT:

The Cartier–Foata form of the result of the star operation if the operation is defined on `self`, `None` otherwise.

EXAMPLES:

We will compute all star operations on the following FC element in type B_6 relative to $J = \{5, 6\}$:

```
sage: FC = CoxeterGroup(['B', 6]).fully_commutative_elements()
sage: w = FC([1, 6, 2, 5, 4, 6, 5])
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['B', Integer(6)]).fully_commutative_elements()
>>> w = FC([Integer(1), Integer(6), Integer(2), Integer(5), Integer(4),␣
→Integer(6), Integer(5)])
```

Whether and how a left star operations can be applied depend on the coset decomposition $w = w_J \cdot w^J$:

```
sage: w.coset_decomposition({5, 6})
([6, 5, 6], [1, 2, 4, 5])
```

```
>>> from sage.all import *
>>> w.coset_decomposition({Integer(5), Integer(6)})
([6, 5, 6], [1, 2, 4, 5])
```

Only the lower star operation is defined on the left on w:

```
sage: w.star_operation({5,6}, 'upper')

sage: w.star_operation({5,6}, 'lower')
[1, 5, 2, 4, 6, 5]
```

```
>>> from sage.all import *
>>> w.star_operation({Integer(5),Integer(6)}, 'upper')
<BLANKLINE>
>>> w.star_operation({Integer(5),Integer(6)}, 'lower')
[1, 5, 2, 4, 6, 5]
```

Whether and how a right star operations can be applied depend on the coset decomposition $w = w^J \cdot w_J$:

```
sage: w.coset_decomposition({5, 6}, side='right')
([1, 6, 2, 5, 4], [6, 5])
```

```
>>> from sage.all import *
>>> w.coset_decomposition({Integer(5), Integer(6)}, side='right')
([1, 6, 2, 5, 4], [6, 5])
```

Both types of right star operations on defined for this example:

```
sage: w.star_operation({5, 6}, 'upper', side='right')
[1, 6, 2, 5, 4, 6, 5, 6]

sage: w.star_operation({5, 6}, 'lower', side='right')
[1, 6, 2, 5, 4, 6]
```

```
>>> from sage.all import *
>>> w.star_operation({Integer(5), Integer(6)}, 'upper', side='right')
[1, 6, 2, 5, 4, 6, 5, 6]

>>> w.star_operation({Integer(5), Integer(6)}, 'lower', side='right')
[1, 6, 2, 5, 4, 6]
```

class `sage.combinat.fully_commutative_elements.`**`FullyCommutativeElements`**`(`*coxeter_group*`)`

Bases: `UniqueRepresentation`, `Parent`

Class for the set of fully commutative (FC) elements of a Coxeter system.

Coxeter systems with finitely many FC elements, or *FC-finite* Coxeter systems, are classfied by Stembridge in [Ste1996]. They fall into seven families, namely the groups of types $A_n, B_n, D_n, E_n, F_n, H_n$ and $I_2(m)$.

INPUT:

- `data` – CoxeterMatrix, CartanType, or the usual datum that can is taken in the constructors for these classes (see *`sage.combinat.root_system.coxeter_group.CoxeterGroup()`*)

OUTPUT:

The class of fully commutative elements in the Coxeter group constructed from `data`. This will belong to the category of enumerated sets. If the Coxeter data corresponds to a Cartan type, the category is further refined to either finite enumerated sets or infinite enumerated sets depending on i whether the Coxeter group is FC-finite; the refinement is not carried out if `data` is a Coxeter matrix not corresponding to a Cartan type.

Todo

It would be ideal to implement the aforementioned refinement to finite and infinite enumerated sets for all possible `data`, regardless of whether it corresponds to a Cartan type. Doing so requires determining if an arbitrary Coxeter matrix corresponds to a Cartan type. It may be best to address this issue in `sage.combinat.root_system`. On the other hand, the refinement in the general case may be unnecessary in light of the fact that Stembridge's classification of FC-finite groups contains a very small number of easily-recognizable families.

EXAMPLES:

Create the enumerate set of fully commutative elements in B_3:

```
sage: FC = CoxeterGroup(['B', 3]).fully_commutative_elements(); FC
Fully commutative elements of Finite Coxeter group over Number Field in a with␣
↪defining polynomial x^2 - 2 with a = 1.414213562373095? with Coxeter matrix:
[1 3 2]
```

(continues on next page)

(continued from previous page)

```
[3 1 4]
[2 4 1]
```

```
>>> from sage.all import *
>>> FC = CoxeterGroup(['B', Integer(3)]).fully_commutative_elements(); FC
Fully commutative elements of Finite Coxeter group over Number Field in a with
↪defining polynomial x^2 - 2 with a = 1.414213562373095? with Coxeter matrix:
[1 3 2]
[3 1 4]
[2 4 1]
```

Construct elements:

```
sage: FC([])
[]
sage: FC([1,2])
[1, 2]
sage: FC([2,3,2])
[2, 3, 2]
sage: FC([3,2,3])
[3, 2, 3]
```

```
>>> from sage.all import *
>>> FC([])
[]
>>> FC([Integer(1),Integer(2)])
[1, 2]
>>> FC([Integer(2),Integer(3),Integer(2)])
[2, 3, 2]
>>> FC([Integer(3),Integer(2),Integer(3)])
[3, 2, 3]
```

Elements are normalized to Cartier–Foata normal form upon construction:

```
sage: FC([3,1])
[1, 3]
sage: FC([2,3,1])
[2, 1, 3]
sage: FC([1,3]) == FC([3,1])
True
```

```
>>> from sage.all import *
>>> FC([Integer(3),Integer(1)])
[1, 3]
>>> FC([Integer(2),Integer(3),Integer(1)])
[2, 1, 3]
>>> FC([Integer(1),Integer(3)]) == FC([Integer(3),Integer(1)])
True
```

Attempting to create an element from an input that is not the reduced word of a fully commutative element throws a `ValueError`:

```
sage: FC([1,2,1])
Traceback (most recent call last):
...
ValueError: the input is not a reduced word of a fully commutative element
sage: FC([2,3,2,3])
Traceback (most recent call last):
...
ValueError: the input is not a reduced word of a fully commutative element
```

```
>>> from sage.all import *
>>> FC([Integer(1),Integer(2),Integer(1)])
Traceback (most recent call last):
...
ValueError: the input is not a reduced word of a fully commutative element
>>> FC([Integer(2),Integer(3),Integer(2),Integer(3)])
Traceback (most recent call last):
...
ValueError: the input is not a reduced word of a fully commutative element
```

Enumerate the FC elements in A_3:

```
sage: FCA3 = CoxeterGroup(['A', 3]).fully_commutative_elements()
sage: FCA3.category()
Category of finite enumerated sets
sage: FCA3.list()
[[],
 [1],
 [2],
 [3],
 [2, 1],
 [1, 3],
 [1, 2],
 [3, 2],
 [2, 3],
 [3, 2, 1],
 [2, 1, 3],
 [1, 3, 2],
 [1, 2, 3],
 [2, 1, 3, 2]]
```

```
>>> from sage.all import *
>>> FCA3 = CoxeterGroup(['A', Integer(3)]).fully_commutative_elements()
>>> FCA3.category()
Category of finite enumerated sets
>>> FCA3.list()
[[],
 [1],
 [2],
 [3],
 [2, 1],
 [1, 3],
 [1, 2],
```

(continues on next page)

(continued from previous page)

```
 [3, 2],
 [2, 3],
 [3, 2, 1],
 [2, 1, 3],
 [1, 3, 2],
 [1, 2, 3],
 [2, 1, 3, 2]]
```

Count the FC elements in B_8:

```
sage: FCB8 = CoxeterGroup(['B', 8]).fully_commutative_elements()
sage: len(FCB8)     # long time (7 seconds)
14299
```

```
>>> from sage.all import *
>>> FCB8 = CoxeterGroup(['B', Integer(8)]).fully_commutative_elements()
>>> len(FCB8)     # long time (7 seconds)
14299
```

Iterate through the FC elements of length up to 2 in the non-FC-finite group affine A_2:

```
sage: FCAffineA2 = CoxeterGroup(['A', 2, 1]).fully_commutative_elements()
sage: FCAffineA2.category()
Category of infinite enumerated sets
sage: list(FCAffineA2.iterate_to_length(2))
[[], [0], [1], [2], [1, 0], [2, 0], [0, 1], [2, 1], [0, 2], [1, 2]]
```

```
>>> from sage.all import *
>>> FCAffineA2 = CoxeterGroup(['A', Integer(2), Integer(1)]).fully_commutative_
↪elements()
>>> FCAffineA2.category()
Category of infinite enumerated sets
>>> list(FCAffineA2.iterate_to_length(Integer(2)))
[[], [0], [1], [2], [1, 0], [2, 0], [0, 1], [2, 1], [0, 2], [1, 2]]
```

The cardinality of the set is determined from the classification of FC-finite Coxeter groups:

```
sage: CoxeterGroup('A2').fully_commutative_elements().category()
Category of finite enumerated sets
sage: CoxeterGroup('B7').fully_commutative_elements().category()
Category of finite enumerated sets
sage: CoxeterGroup('A3~').fully_commutative_elements().category()
Category of infinite enumerated sets
sage: CoxeterGroup('F4~').fully_commutative_elements().category()
Category of finite enumerated sets
sage: CoxeterGroup('E8~').fully_commutative_elements().category()
Category of finite enumerated sets
sage: CoxeterGroup('F4~xE8~').fully_commutative_elements().category()
Category of finite enumerated sets
sage: CoxeterGroup('B4~xE8~').fully_commutative_elements().category()
Category of infinite enumerated sets
```

```
>>> from sage.all import *
>>> CoxeterGroup('A2').fully_commutative_elements().category()
Category of finite enumerated sets
>>> CoxeterGroup('B7').fully_commutative_elements().category()
Category of finite enumerated sets
>>> CoxeterGroup('A3~').fully_commutative_elements().category()
Category of infinite enumerated sets
>>> CoxeterGroup('F4~').fully_commutative_elements().category()
Category of finite enumerated sets
>>> CoxeterGroup('E8~').fully_commutative_elements().category()
Category of finite enumerated sets
>>> CoxeterGroup('F4~xE8~').fully_commutative_elements().category()
Category of finite enumerated sets
>>> CoxeterGroup('B4~xE8~').fully_commutative_elements().category()
Category of infinite enumerated sets
```

Element

alias of *FullyCommutativeElement*

coxeter_group()

Obtain the Coxeter group associated with `self`.

EXAMPLES:

```
sage: FCA3 = CoxeterGroup(['A', 3]).fully_commutative_elements()
sage: FCA3.coxeter_group()
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2]
[3 1 3]
[2 3 1]
```

```
>>> from sage.all import *
>>> FCA3 = CoxeterGroup(['A', Integer(3)]).fully_commutative_elements()
>>> FCA3.coxeter_group()
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2]
[3 1 3]
[2 3 1]
```

iterate_to_length(*length*)

Iterate through the elements of this class up to a maximum length.

INPUT:

- `length` – integer; maximum length of element to generate

OUTPUT: generator for elements of `self` of length up to `length`

EXAMPLES:

The following example produces all FC elements of length up to 2 in the group A_3:

```
sage: FCA3 = CoxeterGroup(['A', 3]).fully_commutative_elements()
sage: list(FCA3.iterate_to_length(2))
[[], [1], [2], [3], [2, 1], [1, 3], [1, 2], [3, 2], [2, 3]]
```

```
>>> from sage.all import *
>>> FCA3 = CoxeterGroup(['A', Integer(3)]).fully_commutative_elements()
>>> list(FCA3.iterate_to_length(Integer(2)))
[[], [1], [2], [3], [2, 1], [1, 3], [1, 2], [3, 2], [2, 3]]
```

The lists for length 4 and 5 are the same since 4 is the maximum length of an FC element in A_3:

```
sage: list(FCA3.iterate_to_length(4))
[[], [1], [2], [3], [2, 1], [1, 3], [1, 2], [3, 2], [2, 3],
 [3, 2, 1], [2, 1, 3], [1, 3, 2], [1, 2, 3], [2, 1, 3, 2]]
sage: list(FCA3.iterate_to_length(5))
[[], [1], [2], [3], [2, 1], [1, 3], [1, 2], [3, 2], [2, 3],
 [3, 2, 1], [2, 1, 3], [1, 3, 2], [1, 2, 3], [2, 1, 3, 2]]
sage: list(FCA3.iterate_to_length(4)) == list(FCA3)
True
```

```
>>> from sage.all import *
>>> list(FCA3.iterate_to_length(Integer(4)))
[[], [1], [2], [3], [2, 1], [1, 3], [1, 2], [3, 2], [2, 3],
 [3, 2, 1], [2, 1, 3], [1, 3, 2], [1, 2, 3], [2, 1, 3, 2]]
>>> list(FCA3.iterate_to_length(Integer(5)))
[[], [1], [2], [3], [2, 1], [1, 3], [1, 2], [3, 2], [2, 3],
 [3, 2, 1], [2, 1, 3], [1, 3, 2], [1, 2, 3], [2, 1, 3, 2]]
>>> list(FCA3.iterate_to_length(Integer(4))) == list(FCA3)
True
```

The following example produces all FC elements of length up to 4 in the affine Weyl group $\tilde{A}_2$:

```
sage: FCAffineA2 = CoxeterGroup(['A', 2, 1]).fully_commutative_elements()
sage: FCAffineA2.category()
Category of infinite enumerated sets
sage: list(FCAffineA2.iterate_to_length(4))
[[], [0], [1], [2], [1, 0], [2, 0], [0, 1], [2, 1], [0, 2],
 [1, 2], [2, 1, 0], [1, 2, 0], [2, 0, 1], [0, 2, 1], [1, 0, 2],
 [0, 1, 2], [0, 2, 1, 0], [0, 1, 2, 0], [1, 2, 0, 1],
 [1, 0, 2, 1], [2, 1, 0, 2], [2, 0, 1, 2]]
```

```
>>> from sage.all import *
>>> FCAffineA2 = CoxeterGroup(['A', Integer(2), Integer(1)]).fully_
↪commutative_elements()
>>> FCAffineA2.category()
Category of infinite enumerated sets
>>> list(FCAffineA2.iterate_to_length(Integer(4)))
[[], [0], [1], [2], [1, 0], [2, 0], [0, 1], [2, 1], [0, 2],
 [1, 2], [2, 1, 0], [1, 2, 0], [2, 0, 1], [0, 2, 1], [1, 0, 2],
 [0, 1, 2], [0, 2, 1, 0], [0, 1, 2, 0], [1, 2, 0, 1],
 [1, 0, 2, 1], [2, 1, 0, 2], [2, 0, 1, 2]]
```

5.1.108 Finite state machines, automata, transducers

This module adds support for finite state machines, automata and transducers.

For creating automata and transducers you can use classes

- *Automaton* and *Transducer* (or the more general class *FiniteStateMachine*)

or the generators

- *automata* and *transducers*

which contain *preconstructed and commonly used automata and transducers*. See also the *examples* below.

Contents

FiniteStateMachine and derived classes Transducer and Automaton

Accessing parts of a finite state machine

state()	Get a state by its label
states()	List of states
iter_states()	Iterator over the states
initial_states()	List of initial states
iter_initial_states()	Iterator over initial states
final_states()	List of final states
iter_final_states()	Iterator over final states
transition()	Get a transition by its states and labels
transitions()	List of transitions
iter_transitions()	Iterator over the transitions
predecessors()	List of predecessors of a state
induced_sub_finite_state_machine()	Induced sub-machine
accessible_components()	Accessible components
coaccessible_components()	Coaccessible components
final_components()	Final components (connected components which cannot be left again)

(Modified) Copies

empty_copy()	Return an empty deep copy
deepcopy()	Return a deep copy
relabeled()	Return a relabeled deep copy
Automaton.with_output()	Extend an automaton to a transducer

Manipulation

`add_state()`	Add a state
`add_states()`	Add states
`delete_state()`	Delete a state
`add_transition()`	Add a transition
`add_transitions_from_function()`	Add transitions
`input_alphabet`	Input alphabet
`output_alphabet`	Output alphabet
`on_duplicate_transition`	Hook for handling duplicate transitions
`add_from_transition_function()`	Add transitions by a transition function
`delete_transition()`	Delete a transition
`remove_epsilon_transitions()`	Remove epsilon transitions (not implemented)
`split_transitions()`	Split transitions with input words of length > `1`
`determine_alphabets()`	Determine input and output alphabets
`determine_input_alphabet()`	Determine input alphabet
`determine_output_alphabet()`	Determine output alphabet
`construct_final_word_out()`	Construct final output by implicitly reading trailing letters; cf. `with_final_word_out()`

Properties

`has_state()`	Check for a state
`has_initial_state()`	Check for an initial state
`has_initial_states()`	Check for initial states
`has_final_state()`	Check for a final state
`has_final_states()`	Check for final states
`has_transition()`	Check for a transition
`is_deterministic()`	Check for a deterministic machine
`is_complete()`	Check for a complete machine
`is_connected()`	Check for a connected machine
`Automaton.is_equivalent()`	Check for equivalent automata
`is_Markov_chain()`	Check for a Markov chain
`is_monochromatic()`	Check whether the colors of all states are equal
`number_of_words()`	Determine the number of successful paths
`asymptotic_moments()`	Main terms of expectation and variance of sums of labels
`moments_waiting_time()`	Moments of the waiting time for first true output
`epsilon_successors()`	Epsilon successors of a state
`Automaton.shannon_parry_markov_chai`	Compute Markov chain with Parry measure

Operations

disjoint_union()	Disjoint union
concatenation()	Concatenation
kleene_star()	Kleene star
Automaton.complement()	Complement of an automaton
Automaton.intersection()	Intersection of automata
Transducer.intersection()	Intersection of transducers
Transducer.cartesian_product()	Cartesian product of a transducer with another finite state machine
product_FiniteStateMachine()	Product of finite state machines
composition()	Composition (output of other is input of self)
input_projection()	Input projection (output is deleted)
output_projection()	Output projection (old output is new input)
projection()	Input or output projection
transposition()	Transposition (all transitions are reversed)
with_final_word_out()	Machine with final output constructed by implicitly reading trailing letters, cf. *construct_final_word_out()* for inplace version
Automaton.determinisation()	Determinisation of an automaton
completion()	Completion of a finite state machine
process()	Process input
__call__()	Process input with shortened output
Automaton.process()	Process input of an automaton (output differs from general case)
Transducer.process()	Process input of a transducer (output differs from general case)
iter_process()	Return process iterator
language()	Return all possible output words
Automaton.language()	Return all possible accepted words

Simplification

prepone_output()	Prepone output where possible
equivalence_classes()	List of equivalent states
quotient()	Quotient with respect to equivalence classes
merged_transitions()	Merge transitions while adding input
markov_chain_simplification()	Simplification of a Markov chain
Automaton.minimization()	Minimization of an automaton
Transducer.simplification()	Simplification of a transducer

Conversion

adjacency_matrix()	(Weighted) adjacency `matrix()`
graph()	Underlying `DiGraph`
plot()	Plot

LaTeX output

latex_options()	Set options
set_coordinates()	Set coordinates of the states
default_format_transition_label()	Default formatting of words in transition labels
format_letter_negative()	Format negative numbers as overlined number
format_transition_label_reversed()	Format words in transition labels in reversed order

See also:

LaTeX output

FSMState

final_word_out	Final output of a state
is_final	Describe whether a state is final or not
is_initial	Describe whether a state is initial or not
initial_probability	Probability of starting in this state as part of a Markov chain
label()	Label of a state
relabeled()	Return a relabeled deep copy of a state
fully_equal()	Check whether two states are fully equal (including all attributes)

FSMTransition

from_state	State in which transition starts
to_state	State in which transition ends
word_in	Input word of the transition
word_out	Output word of the transition
deepcopy()	Return a deep copy of the transition

FSMProcessIterator

next()	Make one step in processing the input tape
preview_word()	Read a word from the input tape
result()	Return the finished branches during process

Helper Functions

equal()	Check whether all elements of `iterator` are equal
full_group_by()	Group iterable by values of some key
startswith()	Determine whether list starts with the given prefix
FSMLetterSymbol()	Return a string associated to the input letter
FSMWordSymbol()	Return a string associated to a word
is_FSMState()	Test whether an object inherits from *FSMState*
is_FSMTransition()	Test whether an object inherits from *FSMTransition*
is_FiniteStateMachine()	Test whether an object inherits from *FiniteStateMachine*
duplicate_transition_ignore()	Default function for handling duplicate transitions
duplicate_transition_raise_error()	Raise error when inserting a duplicate transition
duplicate_transition_add_input()	Add input when inserting a duplicate transition

Examples

We start with a general *FiniteStateMachine*. Later there will be also an *Automaton* and a *Transducer*.

A simple finite state machine

We can easily create a finite state machine by

```
sage: fsm = FiniteStateMachine()
sage: fsm
Empty finite state machine
```

```
>>> from sage.all import *
>>> fsm = FiniteStateMachine()
>>> fsm
Empty finite state machine
```

By default this is the empty finite state machine, so not very interesting. Let's create and add some states and transitions:

```
sage: day = fsm.add_state('day')
sage: night = fsm.add_state('night')
sage: sunrise = fsm.add_transition(night, day)
sage: sunset = fsm.add_transition(day, night)
```

```
>>> from sage.all import *
>>> day = fsm.add_state('day')
>>> night = fsm.add_state('night')
>>> sunrise = fsm.add_transition(night, day)
>>> sunset = fsm.add_transition(day, night)
```

Let us look at `sunset` more closely:

```
sage: sunset
Transition from 'day' to 'night': -|-
```

```
>>> from sage.all import *
>>> sunset
Transition from 'day' to 'night': -|-
```

Note that could also have created and added the transitions directly by:

```
sage: fsm.add_transition('day', 'night')
Transition from 'day' to 'night': -|-
```

```
>>> from sage.all import *
>>> fsm.add_transition('day', 'night')
Transition from 'day' to 'night': -|-
```

This would have had added the states automatically, since they are present in the transitions.

Anyhow, we got the following finite state machine:

```
sage: fsm
Finite state machine with 2 states
```

```
>>> from sage.all import *
>>> fsm
Finite state machine with 2 states
```

We can also obtain the underlying `directed graph` by

```
sage: fsm.graph()
Looped multi-digraph on 2 vertices
```

```
>>> from sage.all import *
>>> fsm.graph()
Looped multi-digraph on 2 vertices
```

To visualize a finite state machine, we can use `latex()` and run the result through LaTeX, see the section on *LaTeX output* below.

Alternatively, we could have created the finite state machine above simply by

```
sage: FiniteStateMachine([('night', 'day'), ('day', 'night')])
Finite state machine with 2 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine([('night', 'day'), ('day', 'night')])
Finite state machine with 2 states
```

See *`FiniteStateMachine`* for a lot of possibilities to create finite state machines.

A simple Automaton (recognizing NAFs)

We want to build an automaton which recognizes non-adjacent forms (NAFs), i.e., sequences which have no adjacent nonzeros. We use 0, 1, and -1 as digits:

```
sage: NAF = Automaton(
....:     {'A': [('A', 0), ('B', 1), ('B', -1)], 'B': [('A', 0)]})
sage: NAF.state('A').is_initial = True
sage: NAF.state('A').is_final = True
sage: NAF.state('B').is_final = True
sage: NAF
Automaton with 2 states
```

```
>>> from sage.all import *
>>> NAF = Automaton(
...     {'A': [('A', Integer(0)), ('B', Integer(1)), ('B', -Integer(1))], 'B': [('A',␣
↪Integer(0))]})
>>> NAF.state('A').is_initial = True
>>> NAF.state('A').is_final = True
>>> NAF.state('B').is_final = True
>>> NAF
Automaton with 2 states
```

Of course, we could have specified the initial and final states directly in the definition of `NAF` by `initial_states=['A']` and `final_states=['A', 'B']`.

So let's test the automaton with some input:

```
sage: NAF([0])
True
sage: NAF([0, 1])
True
sage: NAF([1, -1])
False
sage: NAF([0, -1, 0, 1])
True
sage: NAF([0, -1, -1, -1, 0])
False
sage: NAF([-1, 0, 0, 1, 1])
False
```

```
>>> from sage.all import *
>>> NAF([Integer(0)])
True
>>> NAF([Integer(0), Integer(1)])
True
>>> NAF([Integer(1), -Integer(1)])
False
>>> NAF([Integer(0), -Integer(1), Integer(0), Integer(1)])
True
>>> NAF([Integer(0), -Integer(1), -Integer(1), -Integer(1), Integer(0)])
False
>>> NAF([-Integer(1), Integer(0), Integer(0), Integer(1), Integer(1)])
False
```

Alternatively, we could call that by

```
sage: NAF.process([0, -1, 0, 1])
(True, 'B')
```

```
>>> from sage.all import *
>>> NAF.process([Integer(0), -Integer(1), Integer(0), Integer(1)])
(True, 'B')
```

which gives additionally the state in which we arrived.

We can also let an automaton act on a *word*:

```
sage: # needs sage.combinat
sage: W = Words([-1, 0, 1]); W
Finite and infinite words over {-1, 0, 1}
sage: w = W([1, 0, 1, 0, -1]); w
word: 1,0,1,0,-1
sage: NAF(w)
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> W = Words([-Integer(1), Integer(0), Integer(1)]); W
Finite and infinite words over {-1, 0, 1}
>>> w = W([Integer(1), Integer(0), Integer(1), Integer(0), -Integer(1)]); w
word: 1,0,1,0,-1
>>> NAF(w)
True
```

Recognizing NAFs via Automata Operations

Alternatively, we can use automata operations to recognize NAFs; for simplicity, we only use the input alphabet `[0, 1]`. On the one hand, we can construct such an automaton by forbidding the word `11`:

```
sage: forbidden = automata.ContainsWord([1, 1], input_alphabet=[0, 1])
sage: NAF_negative = forbidden.complement()
sage: NAF_negative([1, 1, 0, 1])
False
sage: NAF_negative([1, 0, 1, 0, 1])
True
```

```
>>> from sage.all import *
>>> forbidden = automata.ContainsWord([Integer(1), Integer(1)], input_
↪alphabet=[Integer(0), Integer(1)])
>>> NAF_negative = forbidden.complement()
>>> NAF_negative([Integer(1), Integer(1), Integer(0), Integer(1)])
False
>>> NAF_negative([Integer(1), Integer(0), Integer(1), Integer(0), Integer(1)])
True
```

On the other hand, we can write this as a regular expression and translate that into automata operations:

```
sage: zero = automata.Word([0])
sage: one = automata.Word([1])
sage: epsilon = automata.EmptyWord(input_alphabet=[0, 1])
sage: NAF_positive = (zero + one*zero).kleene_star() * (epsilon + one)
```

```
>>> from sage.all import *
>>> zero = automata.Word([Integer(0)])
>>> one = automata.Word([Integer(1)])
>>> epsilon = automata.EmptyWord(input_alphabet=[Integer(0), Integer(1)])
>>> NAF_positive = (zero + one*zero).kleene_star() * (epsilon + one)
```

We check that the two approaches are equivalent:

```
sage: NAF_negative.is_equivalent(NAF_positive)
True
```

```
>>> from sage.all import *
>>> NAF_negative.is_equivalent(NAF_positive)
True
```

See also

ContainsWord(), *Word()*, *complement()*, *kleene_star()*, *EmptyWord()*, *is_equivalent()*.

LaTeX output

We can visualize a finite state machine by converting it to LaTeX by using the usual function `latex()`. Within LaTeX, TikZ is used for typesetting the graphics, see the Wikipedia article PGF/TikZ.

```
sage: print(latex(NAF))  # abs tol 1e-3
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, accepting, initial] (v0) at (3.000000, 0.000000) {$\text{\texttt{A}}$};
\node[state, accepting] (v1) at (-3.000000, 0.000000) {$\text{\texttt{B}}$};
\path[->] (v0) edge[loop above] node {$0$} ();
\path[->] (v0.185.00) edge node[rotate=360.00, anchor=north] {$1, -1$} (v1.355.00);
\path[->] (v1.5.00) edge node[rotate=0.00, anchor=south] {$0$} (v0.175.00);
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> print(latex(NAF))  # abs tol 1e-3
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, accepting, initial] (v0) at (3.000000, 0.000000) {$\text{\texttt{A}}$};
\node[state, accepting] (v1) at (-3.000000, 0.000000) {$\text{\texttt{B}}$};
\path[->] (v0) edge[loop above] node {$0$} ();
\path[->] (v0.185.00) edge node[rotate=360.00, anchor=north] {$1, -1$} (v1.355.00);
\path[->] (v1.5.00) edge node[rotate=0.00, anchor=south] {$0$} (v0.175.00);
\end{tikzpicture}
```

We can turn this into a graphical representation.

```
sage: view(NAF) # not tested
```

```
>>> from sage.all import *
>>> view(NAF) # not tested
```

To actually see this, use the live documentation in the Sage notebook and execute the cells in this and the previous section.

Several options can be set to customize the output, see `latex_options()` for details. In particular, we use `format_letter_negative()` to format -1 as $\bar{1}$.

```
sage: NAF.latex_options(
....:     coordinates={'A': (0, 0),
....:                  'B': (6, 0)},
....:     initial_where={'A': 'below'},
....:     format_letter=NAF.format_letter_negative,
....:     format_state_label=lambda x:
....:         r'\mathcal{%s}' % x.label()
....: )
sage: print(latex(NAF))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, accepting, initial, initial where=below] (v0) at (0.000000, 0.000000) {$\
↪mathcal{A}$};
\node[state, accepting] (v1) at (6.000000, 0.000000) {$\mathcal{B}$};
\path[->] (v0) edge[loop above] node {$0$} ();
\path[->] (v0.5.00) edge node[rotate=0.00, anchor=south] {$1, \overline{1}$} (v1.175.
↪00);
\path[->] (v1.185.00) edge node[rotate=360.00, anchor=north] {$0$} (v0.355.00);
\end{tikzpicture}
sage: view(NAF) # not tested
```

```
>>> from sage.all import *
>>> NAF.latex_options(
...     coordinates={'A': (Integer(0), Integer(0)),
...                  'B': (Integer(6), Integer(0))},
...     initial_where={'A': 'below'},
...     format_letter=NAF.format_letter_negative,
...     format_state_label=lambda x:
...         r'\mathcal{%s}' % x.label()
... )
>>> print(latex(NAF))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, accepting, initial, initial where=below] (v0) at (0.000000, 0.000000) {$\
↪mathcal{A}$};
\node[state, accepting] (v1) at (6.000000, 0.000000) {$\mathcal{B}$};
\path[->] (v0) edge[loop above] node {$0$} ();
\path[->] (v0.5.00) edge node[rotate=0.00, anchor=south] {$1, \overline{1}$} (v1.175.
↪00);
\path[->] (v1.185.00) edge node[rotate=360.00, anchor=north] {$0$} (v0.355.00);
\end{tikzpicture}
>>> view(NAF) # not tested
```

To use the output of `latex()` in your own LaTeX file, you have to include

```
\usepackage{tikz}
\usetikzlibrary{automata}
```

into the preamble of your file.

A simple transducer (binary inverter)

Let's build a simple transducer, which rewrites a binary word by inverting each bit:

```
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
```

```
>>> from sage.all import *
>>> inverter = Transducer({'A': [('A', Integer(0), Integer(1)), ('A', Integer(1),
↪Integer(0))]},
...     initial_states=['A'], final_states=['A'])
```

We can look at the states and transitions:

```
sage: inverter.states()
['A']
sage: for t in inverter.transitions():
....:     print(t)
Transition from 'A' to 'A': 0|1
Transition from 'A' to 'A': 1|0
```

```
>>> from sage.all import *
>>> inverter.states()
['A']
>>> for t in inverter.transitions():
...     print(t)
Transition from 'A' to 'A': 0|1
Transition from 'A' to 'A': 1|0
```

Now we apply a word to it and see what the transducer does:

```
sage: inverter([0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1])
[1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> inverter([Integer(0), Integer(1), Integer(0), Integer(0), Integer(1), Integer(1),
↪Integer(0), Integer(0), Integer(0), Integer(1), Integer(1), Integer(1)])
[1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0]
```

`True` means, that we landed in a final state, that state is labeled `'A'`, and we also got an output.

Transducers and (in)finite Words

A transducer can also act on everything iterable, in particular, on Sage's *words*.

```
sage: W = Words([0, 1]); W
Finite and infinite words over {0, 1}
```

```
>>> from sage.all import *
>>> W = Words([Integer(0), Integer(1)]); W
Finite and infinite words over {0, 1}
```

Let us take the inverter from the previous section and feed some finite word into it:

```
sage: w = W([1, 1, 0, 1]); w
word: 1101
sage: inverter(w)
word: 0010
```

```
>>> from sage.all import *
>>> w = W([Integer(1), Integer(1), Integer(0), Integer(1)]); w
word: 1101
>>> inverter(w)
word: 0010
```

We see that the output is again a word (this is a consequence of calling `process()` with `automatic_output_type`).

We can even input something infinite like an infinite word:

```
sage: tm = words.ThueMorseWord(); tm
word: 0110100110010110100101100110100110010110...
sage: inverter(tm)
word: 1001011001101001011010011001011001101001...
```

```
>>> from sage.all import *
>>> tm = words.ThueMorseWord(); tm
word: 0110100110010110100101100110100110010110...
>>> inverter(tm)
word: 1001011001101001011010011001011001101001...
```

A transducer which performs division by 3 in binary

Now we build a transducer, which divides a binary number by 3. The labels of the states are the remainder of the division. The transition function is

```
sage: def f(state_from, read):
....:     if state_from + read <= 1:
....:         state_to = 2*state_from + read
....:         write = 0
....:     else:
....:         state_to = 2*state_from + read - 3
....:         write = 1
....:     return (state_to, write)
```

```
>>> from sage.all import *
>>> def f(state_from, read):
...     if state_from + read <= Integer(1):
...         state_to = Integer(2)*state_from + read
...         write = Integer(0)
...     else:
```

(continues on next page)

(continued from previous page)

```
...         state_to = Integer(2)*state_from + read - Integer(3)
...         write = Integer(1)
...     return (state_to, write)
```

which assumes reading a binary number from left to right. We get the transducer with

```
sage: D = Transducer(f, initial_states=[0], final_states=[0],
....:                input_alphabet=[0, 1])
```

```
>>> from sage.all import *
>>> D = Transducer(f, initial_states=[Integer(0)], final_states=[Integer(0)],
...                input_alphabet=[Integer(0), Integer(1)])
```

Let us try to divide 12 by 3:

```
sage: D([1, 1, 0, 0])
[0, 1, 0, 0]
```

```
>>> from sage.all import *
>>> D([Integer(1), Integer(1), Integer(0), Integer(0)])
[0, 1, 0, 0]
```

Now we want to divide 13 by 3:

```
sage: D([1, 1, 0, 1])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

```
>>> from sage.all import *
>>> D([Integer(1), Integer(1), Integer(0), Integer(1)])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

The raised `ValueError` means 13 is not divisible by 3.

Gray Code

The Gray code is a binary numeral system where two successive values differ in only one bit, cf. the Wikipedia article Gray_code. The Gray code of an integer n is obtained by a bitwise xor between the binary expansion of n and the binary expansion of $\lfloor n/2\rfloor$; the latter corresponds to a shift by one position in binary.

The purpose of this example is to construct a transducer converting the standard binary expansion to the Gray code by translating this construction into operations with transducers.

For this construction, the least significant digit is at the left-most position. Note that it is easier to shift everything to the right first, i.e., multiply by 2 instead of building $\lfloor n/2\rfloor$. Then, we take the input xor with the right shift of the input and forget the first letter.

We first construct a transducer shifting the binary expansion to the right. This requires storing the previously read digit in a state.

```
sage: def shift_right_transition(state, digit):
....:     if state == 'I':
....:         return (digit, None)
....:     else:
....:         return (digit, state)
sage: shift_right_transducer = Transducer(
....:     shift_right_transition,
....:     initial_states=['I'],
....:     input_alphabet=[0, 1],
....:     final_states=[0])
sage: shift_right_transducer.transitions()
[Transition from 'I' to 0: 0|-,
 Transition from 'I' to 1: 1|-,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|1,
 Transition from 1 to 1: 1|1]
sage: shift_right_transducer([0, 1, 1, 0])
[0, 1, 1]
sage: shift_right_transducer([1, 0, 0])
[1, 0]
```

```
>>> from sage.all import *
>>> def shift_right_transition(state, digit):
...     if state == 'I':
...         return (digit, None)
...     else:
...         return (digit, state)
>>> shift_right_transducer = Transducer(
...     shift_right_transition,
...     initial_states=['I'],
...     input_alphabet=[Integer(0), Integer(1)],
...     final_states=[Integer(0)])
>>> shift_right_transducer.transitions()
[Transition from 'I' to 0: 0|-,
 Transition from 'I' to 1: 1|-,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|1,
 Transition from 1 to 1: 1|1]
>>> shift_right_transducer([Integer(0), Integer(1), Integer(1), Integer(0)])
[0, 1, 1]
>>> shift_right_transducer([Integer(1), Integer(0), Integer(0)])
[1, 0]
```

The output of the shifts above look a bit weird (from a right-shift transducer, we would expect, for example, that `[1, 0, 0]` was mapped to `[0, 1, 0]`), since we write `None` instead of the zero at the left. Further, note that only 0 is listed as a final state as we have to enforce that a most significant zero is read as the last input letter in order to flush the last digit:

```
sage: shift_right_transducer([0, 1, 0, 1])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

```
>>> from sage.all import *
>>> shift_right_transducer([Integer(0), Integer(1), Integer(0), Integer(1)])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

Next, we construct the transducer performing the xor operation. We also have to take `None` into account as our `shift_right_transducer` waits one iteration until it starts writing output. This corresponds with our intention to forget the first letter.

```
sage: def xor_transition(state, digits):
....:     if digits[0] is None or digits[1] is None:
....:         return (0, None)
....:     else:
....:         return (0, digits[0].__xor__(digits[1]))
sage: from itertools import product
sage: xor_transducer = Transducer(
....:     xor_transition,
....:     initial_states=[0],
....:     final_states=[0],
....:     input_alphabet=list(product([None, 0, 1], [0, 1])))
sage: xor_transducer.transitions()
[Transition from 0 to 0: (None, 0)|-,
 Transition from 0 to 0: (None, 1)|-,
 Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|0]
sage: xor_transducer([(None, 0), (None, 1), (0, 0), (0, 1), (1, 0), (1, 1)])
[0, 1, 1, 0]
sage: xor_transducer([(0, None)])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

```
>>> from sage.all import *
>>> def xor_transition(state, digits):
...     if digits[Integer(0)] is None or digits[Integer(1)] is None:
...         return (Integer(0), None)
...     else:
...         return (Integer(0), digits[Integer(0)].__xor__(digits[Integer(1)]))
>>> from itertools import product
>>> xor_transducer = Transducer(
...     xor_transition,
...     initial_states=[Integer(0)],
...     final_states=[Integer(0)],
...     input_alphabet=list(product([None, Integer(0), Integer(1)], [Integer(0),
↪Integer(1)])))
>>> xor_transducer.transitions()
[Transition from 0 to 0: (None, 0)|-,
 Transition from 0 to 0: (None, 1)|-,
 Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
```

(continues on next page)

(continued from previous page)

```
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|0]
>>> xor_transducer([(None, Integer(0)), (None, Integer(1)), (Integer(0), Integer(0)),
↪(Integer(0), Integer(1)), (Integer(1), Integer(0)), (Integer(1), Integer(1))])
[0, 1, 1, 0]
>>> xor_transducer([(Integer(0), None)])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

The transducer computing the Gray code is then constructed as a *Cartesian product* between the shifted version and the original input (represented here by the `shift_right_transducer` and the *identity transducer*, respectively). This Cartesian product is then fed into the `xor_transducer` as a *composition* of transducers.

```
sage: product_transducer = shift_right_transducer.cartesian_product(transducers.
↪Identity([0, 1]))
sage: Gray_transducer = xor_transducer(product_transducer)
```

```
>>> from sage.all import *
>>> product_transducer = shift_right_transducer.cartesian_product(transducers.
↪Identity([Integer(0), Integer(1)]))
>>> Gray_transducer = xor_transducer(product_transducer)
```

We use *construct_final_word_out()* to make sure that all output is written; otherwise, we would have to make sure that a sufficient number of trailing zeros is read.

```
sage: Gray_transducer.construct_final_word_out([0])
sage: Gray_transducer.transitions()
[Transition from (('I', 0), 0) to ((0, 0), 0): 0|-,
 Transition from (('I', 0), 0) to ((1, 0), 0): 1|-,
 Transition from ((0, 0), 0) to ((0, 0), 0): 0|0,
 Transition from ((0, 0), 0) to ((1, 0), 0): 1|1,
 Transition from ((1, 0), 0) to ((0, 0), 0): 0|1,
 Transition from ((1, 0), 0) to ((1, 0), 0): 1|0]
```

```
>>> from sage.all import *
>>> Gray_transducer.construct_final_word_out([Integer(0)])
>>> Gray_transducer.transitions()
[Transition from (('I', 0), 0) to ((0, 0), 0): 0|-,
 Transition from (('I', 0), 0) to ((1, 0), 0): 1|-,
 Transition from ((0, 0), 0) to ((0, 0), 0): 0|0,
 Transition from ((0, 0), 0) to ((1, 0), 0): 1|1,
 Transition from ((1, 0), 0) to ((0, 0), 0): 0|1,
 Transition from ((1, 0), 0) to ((1, 0), 0): 1|0]
```

There is a *prepackaged transducer* for Gray code, let's see whether they agree. We have to use *relabeled()* to relabel our states with integers.

```
sage: constructed = Gray_transducer.relabeled()
sage: packaged = transducers.GrayCode()
sage: constructed == packaged
True
```

```
>>> from sage.all import *
>>> constructed = Gray_transducer.relabeled()
>>> packaged = transducers.GrayCode()
>>> constructed == packaged
True
```

Finally, we check that this indeed computes the Gray code of the first 10 nonnegative integers.

```
sage: for n in srange(10):
....:     Gray_transducer(n.bits())
[]
[1]
[1, 1]
[0, 1]
[0, 1, 1]
[1, 1, 1]
[1, 0, 1]
[0, 0, 1]
[0, 0, 1, 1]
[1, 0, 1, 1]
```

```
>>> from sage.all import *
>>> for n in srange(Integer(10)):
...     Gray_transducer(n.bits())
[]
[1]
[1, 1]
[0, 1]
[0, 1, 1]
[1, 1, 1]
[1, 0, 1]
[0, 0, 1]
[0, 0, 1, 1]
[1, 0, 1, 1]
```

Using the hook-functions

Let's use the *previous example "division by 3"* to demonstrate the optional state and transition parameters `hook`.

First, we define what those functions should do. In our case, this is just saying in which state we are and which transition we take

```
sage: def state_hook(process, state, output):
....:     print("We are now in State %s." % (state.label(),))
sage: from sage.combinat.finite_state_machine import FSMWordSymbol
sage: def transition_hook(transition, process):
....:     print("Currently we go from %s to %s, "
....:           "reading %s and writing %s." % (
....:                transition.from_state, transition.to_state,
....:                FSMWordSymbol(transition.word_in),
....:                FSMWordSymbol(transition.word_out)))
```

```
>>> from sage.all import *
>>> def state_hook(process, state, output):
...     print("We are now in State %s." % (state.label(),))
>>> from sage.combinat.finite_state_machine import FSMWordSymbol
>>> def transition_hook(transition, process):
...     print("Currently we go from %s to %s, "
...           "reading %s and writing %s." % (
...               transition.from_state, transition.to_state,
...               FSMWordSymbol(transition.word_in),
...               FSMWordSymbol(transition.word_out)))
```

Now, let's add these hook-functions to the existing transducer:

```
sage: for s in D.iter_states():
....:     s.hook = state_hook
sage: for t in D.iter_transitions():
....:     t.hook = transition_hook
```

```
>>> from sage.all import *
>>> for s in D.iter_states():
...     s.hook = state_hook
>>> for t in D.iter_transitions():
...     t.hook = transition_hook
```

Rerunning the process again now gives the following output:

```
sage: D.process([1, 1, 0, 1], check_epsilon_transitions=False)
We are now in State 0.
Currently we go from 0 to 1, reading 1 and writing 0.
We are now in State 1.
Currently we go from 1 to 0, reading 1 and writing 1.
We are now in State 0.
Currently we go from 0 to 0, reading 0 and writing 0.
We are now in State 0.
Currently we go from 0 to 1, reading 1 and writing 0.
We are now in State 1.
(False, 1, [0, 1, 0, 0])
```

```
>>> from sage.all import *
>>> D.process([Integer(1), Integer(1), Integer(0), Integer(1)], check_epsilon_
↪transitions=False)
We are now in State 0.
Currently we go from 0 to 1, reading 1 and writing 0.
We are now in State 1.
Currently we go from 1 to 0, reading 1 and writing 1.
We are now in State 0.
Currently we go from 0 to 0, reading 0 and writing 0.
We are now in State 0.
Currently we go from 0 to 1, reading 1 and writing 0.
We are now in State 1.
(False, 1, [0, 1, 0, 0])
```

The example above just explains the basic idea of using hook-functions. In the following, we will use those hooks more seriously.

> **Warning**
>
> The hooks of the states are also called while exploring the epsilon successors of a state (during processing). In the example above, we used `check_epsilon_transitions=False` to avoid this (and also therefore got a cleaner output).

> **Warning**
>
> The arguments used when calling a hook have changed in Issue #16538 from `hook(state, process)` to `hook(process, state, output)`.

Detecting sequences with same number of 0 and 1

Suppose we have a binary input and want to accept all sequences with the same number of 0 and 1. This cannot be done with a finite automaton. Anyhow, we can make usage of the hook functions to extend our finite automaton by a counter:

```
sage: from sage.combinat.finite_state_machine import FSMState, FSMTransition
sage: C = FiniteStateMachine()
sage: def update_counter(process, state, output):
....:     try:
....:         l = process.preview_word()
....:     except RuntimeError:
....:         raise StopIteration
....:     process.fsm.counter += 1 if l == 1 else -1
....:     if process.fsm.counter > 0:
....:         next_state = 'positive'
....:     elif process.fsm.counter < 0:
....:         next_state = 'negative'
....:     else:
....:         next_state = 'zero'
....:     return FSMTransition(state, process.fsm.state(next_state),
....:                          l, process.fsm.counter)
sage: C.add_state(FSMState('zero', hook=update_counter,
....:             is_initial=True, is_final=True))
'zero'
sage: C.add_state(FSMState('positive', hook=update_counter))
'positive'
sage: C.add_state(FSMState('negative', hook=update_counter))
'negative'
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState, FSMTransition
>>> C = FiniteStateMachine()
>>> def update_counter(process, state, output):
...     try:
...         l = process.preview_word()
...     except RuntimeError:
...         raise StopIteration
```

(continues on next page)

(continued from previous page)

```
...     process.fsm.counter += Integer(1) if l == Integer(1) else -Integer(1)
...     if process.fsm.counter > Integer(0):
...         next_state = 'positive'
...     elif process.fsm.counter < Integer(0):
...         next_state = 'negative'
...     else:
...         next_state = 'zero'
...     return FSMTransition(state, process.fsm.state(next_state),
...                          l, process.fsm.counter)
>>> C.add_state(FSMState('zero', hook=update_counter,
...             is_initial=True, is_final=True))
'zero'
>>> C.add_state(FSMState('positive', hook=update_counter))
'positive'
>>> C.add_state(FSMState('negative', hook=update_counter))
'negative'
```

Now, let's input some sequence:

```
sage: C.counter = 0; C([1, 1, 1, 1, 0, 0])
(False, 'positive', [1, 2, 3, 4, 3, 2])
```

```
>>> from sage.all import *
>>> C.counter = Integer(0); C([Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(0)])
(False, 'positive', [1, 2, 3, 4, 3, 2])
```

The result is False, since there are four 1 but only two 0. We land in the state `positive` and we can also see the values of the counter in each step.

Let's try some other examples:

```
sage: C.counter = 0; C([1, 1, 0, 0])
(True, 'zero', [1, 2, 1, 0])
sage: C.counter = 0; C([0, 1, 0, 0])
(False, 'negative', [-1, 0, -1, -2])
```

```
>>> from sage.all import *
>>> C.counter = Integer(0); C([Integer(1), Integer(1), Integer(0), Integer(0)])
(True, 'zero', [1, 2, 1, 0])
>>> C.counter = Integer(0); C([Integer(0), Integer(1), Integer(0), Integer(0)])
(False, 'negative', [-1, 0, -1, -2])
```

See also methods `Automaton.process()` and `Transducer.process()` (or even `FiniteStateMachine.process()`), the explanation of the parameter `hook` and the examples in `FSMState` and `FSMTransition`, and the description and examples in `FSMProcessIterator` for more information on processing and hooks.

REFERENCES:

AUTHORS:

- Daniel Krenn (2012-03-27): initial version
- Clemens Heuberger (2012-04-05): initial version
- Sara Kropf (2012-04-17): initial version

- Clemens Heuberger (2013-08-21): release candidate for Sage patch
- Daniel Krenn (2013-08-21): release candidate for Sage patch
- Sara Kropf (2013-08-21): release candidate for Sage patch
- Clemens Heuberger (2013-09-02): documentation improved
- Daniel Krenn (2013-09-13): comments from trac worked in
- **Clemens Heuberger (2013-11-03): output (labels) of determinisation,**
  product, composition, etc. changed (for consistency), representation of state changed, documentation improved
- Daniel Krenn (2013-11-04): whitespaces in documentation corrected
- Clemens Heuberger (2013-11-04): full_group_by added
- Daniel Krenn (2013-11-04): next release candidate for Sage patch
- Sara Kropf (2013-11-08): fix for adjacency matrix
- Clemens Heuberger (2013-11-11): fix for prepone_output
- **Daniel Krenn (2013-11-11): comments from Issue #15078 included:**
  docstring of FiniteStateMachine rewritten, Automaton and Transducer inherited from FiniteStateMachine
- **Daniel Krenn (2013-11-25): documentation improved according to**
  comments from Issue #15078
- Clemens Heuberger, Daniel Krenn, Sara Kropf (2014-02-21–2014-07-18): A huge bunch of improvements. Details see Issue #15841, Issue #15847, Issue #15848, Issue #15849, Issue #15850, Issue #15922, Issue #15923, Issue #15924, Issue #15925, Issue #15928, Issue #15960, Issue #15961, Issue #15962, Issue #15963, Issue #15975, Issue #16016, Issue #16024, Issue #16061, Issue #16128, Issue #16132, Issue #16138, Issue #16139, Issue #16140, Issue #16143, Issue #16144, Issue #16145, Issue #16146, Issue #16191, Issue #16200, Issue #16205, Issue #16206, Issue #16207, Issue #16229, Issue #16253, Issue #16254, Issue #16255, Issue #16266, Issue #16355, Issue #16357, Issue #16387, Issue #16425, Issue #16539, Issue #16555, Issue #16557, Issue #16588, Issue #16589, Issue #16666, Issue #16668, Issue #16674, Issue #16675, Issue #16677.
- Daniel Krenn (2015-09-14): cleanup Issue #18227

ACKNOWLEDGEMENT:

- Clemens Heuberger, Daniel Krenn and Sara Kropf are supported by the Austrian Science Fund (FWF): P 24644-N26.

Methods

class `sage.combinat.finite_state_machine.`**`Automaton`**`(*args, **kwargs)`

Bases: `FiniteStateMachine`

This creates an automaton, which is a finite state machine, whose transitions have input labels.

An automaton has additional features like creating a deterministic and a minimized automaton.

See class `FiniteStateMachine` for more information.

EXAMPLES:

We can create an automaton recognizing even numbers (given in binary and read from left to right) in the following way:

```
sage: A = Automaton([('P', 'Q', 0), ('P', 'P', 1),
....:                ('Q', 'P', 1), ('Q', 'Q', 0)],
....:               initial_states=['P'], final_states=['Q'])
sage: A
Automaton with 2 states
sage: A([0])
True
sage: A([1, 1, 0])
True
sage: A([1, 0, 1])
False
```

```
>>> from sage.all import *
>>> A = Automaton([('P', 'Q', Integer(0)), ('P', 'P', Integer(1)),
...                ('Q', 'P', Integer(1)), ('Q', 'Q', Integer(0))],
...               initial_states=['P'], final_states=['Q'])
>>> A
Automaton with 2 states
>>> A([Integer(0)])
True
>>> A([Integer(1), Integer(1), Integer(0)])
True
>>> A([Integer(1), Integer(0), Integer(1)])
False
```

Note that the full output of the commands can be obtained by calling *process()* and looks like this:

```
sage: A.process([1, 0, 1])
(False, 'P')
```

```
>>> from sage.all import *
>>> A.process([Integer(1), Integer(0), Integer(1)])
(False, 'P')
```

cartesian_product(*other*, *only_accessible_components=True*)

Return a new automaton which accepts an input if it is accepted by both given automata.

INPUT:

- `other` – an automaton
- `only_accessible_components` – if `True` (default), then the result is piped through *accessible_components()*. If no `new_input_alphabet` is given, it is determined by *determine_alphabets()*.

OUTPUT:

A new automaton which computes the intersection (see below) of the languages of `self` and `other`.

The set of states of the new automaton is the Cartesian product of the set of states of both given automata. There is a transition $((A, B), (C, D), a)$ in the new automaton if there are transitions (A, C, a) and (B, D, a) in the old automata.

The methods *intersection()* and *cartesian_product()* are the same (for automata).

EXAMPLES:

```
sage: aut1 = Automaton([('1', '2', 1),
....:                    ('2', '2', 1),
....:                    ('2', '2', 0)],
....:                   initial_states=['1'],
....:                   final_states=['2'],
....:                   determine_alphabets=True)
sage: aut2 = Automaton([('A', 'A', 1),
....:                    ('A', 'B', 0),
....:                    ('B', 'B', 0),
....:                    ('B', 'A', 1)],
....:                   initial_states=['A'],
....:                   final_states=['B'],
....:                   determine_alphabets=True)
sage: res = aut1.intersection(aut2)
sage: (aut1([1, 1]), aut2([1, 1]), res([1, 1]))
(True, False, False)
sage: (aut1([1, 0]), aut2([1, 0]), res([1, 0]))
(True, True, True)
sage: res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'A') to ('2', 'B'): 0|-,
 Transition from ('2', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'B') to ('2', 'B'): 0|-,
 Transition from ('2', 'B') to ('2', 'A'): 1|-]
```

```
>>> from sage.all import *
>>> aut1 = Automaton([('1', '2', Integer(1)),
...                    ('2', '2', Integer(1)),
...                    ('2', '2', Integer(0))],
...                   initial_states=['1'],
...                   final_states=['2'],
...                   determine_alphabets=True)
>>> aut2 = Automaton([('A', 'A', Integer(1)),
...                    ('A', 'B', Integer(0)),
...                    ('B', 'B', Integer(0)),
...                    ('B', 'A', Integer(1))],
...                   initial_states=['A'],
...                   final_states=['B'],
...                   determine_alphabets=True)
>>> res = aut1.intersection(aut2)
>>> (aut1([Integer(1), Integer(1)]), aut2([Integer(1), Integer(1)]),␣
↪res([Integer(1), Integer(1)]))
(True, False, False)
>>> (aut1([Integer(1), Integer(0)]), aut2([Integer(1), Integer(0)]),␣
↪res([Integer(1), Integer(0)]))
(True, True, True)
>>> res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'A') to ('2', 'B'): 0|-,
 Transition from ('2', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'B') to ('2', 'B'): 0|-,
 Transition from ('2', 'B') to ('2', 'A'): 1|-]
```

For automata with epsilon-transitions, intersection is not well defined. But for any finite state machine, epsilon-transitions can be removed by `remove_epsilon_transitions()`.

```
sage: a1 = Automaton([(0, 0, 0),
....:                 (0, 1, None),
....:                 (1, 1, 1),
....:                 (1, 2, 1)],
....:                 initial_states=[0],
....:                 final_states=[1],
....:                 determine_alphabets=True)
sage: a2 = Automaton([(0, 0, 0), (0, 1, 1), (1, 1, 1)],
....:                 initial_states=[0],
....:                 final_states=[1],
....:                 determine_alphabets=True)
sage: a1.intersection(a2)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input)
was found.
sage: a1.remove_epsilon_transitions()  # not tested (since not implemented␣
↪yet)
sage: a1.intersection(a2)  # not tested
```

```
>>> from sage.all import *
>>> a1 = Automaton([(Integer(0), Integer(0), Integer(0)),
...                 (Integer(0), Integer(1), None),
...                 (Integer(1), Integer(1), Integer(1)),
...                 (Integer(1), Integer(2), Integer(1))],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(1)],
...                 determine_alphabets=True)
>>> a2 = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)), (Integer(1), Integer(1), Integer(1))],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(1)],
...                 determine_alphabets=True)
>>> a1.intersection(a2)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input)
was found.
>>> a1.remove_epsilon_transitions()  # not tested (since not implemented yet)
>>> a1.intersection(a2)  # not tested
```

complement()

Return the complement of this automaton.

OUTPUT: an `Automaton`

If this automaton recognizes language $\mathcal{L}$ over an input alphabet $\mathcal{A}$, then the complement recognizes $\mathcal{A} \setminus \mathcal{L}$.

EXAMPLES:

```
sage: A = automata.Word([0, 1])
sage: [w for w in ([], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1])
```

(continues on next page)

(continued from previous page)

```
....:  if A(w)]
[[0, 1]]
sage: Ac = A.complement()
sage: Ac.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 0 to 3: 1|-,
 Transition from 2 to 3: 0|-,
 Transition from 2 to 3: 1|-,
 Transition from 1 to 2: 1|-,
 Transition from 1 to 3: 0|-,
 Transition from 3 to 3: 0|-,
 Transition from 3 to 3: 1|-]
sage: [w for w in ([], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1])
....:  if Ac(w)]
[[], [0], [1], [0, 0], [1, 0], [1, 1]]
```

```
>>> from sage.all import *
>>> A = automata.Word([Integer(0), Integer(1)])
>>> [w for w in ([], [Integer(0)], [Integer(1)], [Integer(0), Integer(0)],␣
↪[Integer(0), Integer(1)], [Integer(1), Integer(0)], [Integer(1),␣
↪Integer(1)])
...  if A(w)]
[[0, 1]]
>>> Ac = A.complement()
>>> Ac.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 0 to 3: 1|-,
 Transition from 2 to 3: 0|-,
 Transition from 2 to 3: 1|-,
 Transition from 1 to 2: 1|-,
 Transition from 1 to 3: 0|-,
 Transition from 3 to 3: 0|-,
 Transition from 3 to 3: 1|-]
>>> [w for w in ([], [Integer(0)], [Integer(1)], [Integer(0), Integer(0)],␣
↪[Integer(0), Integer(1)], [Integer(1), Integer(0)], [Integer(1),␣
↪Integer(1)])
...  if Ac(w)]
[[], [0], [1], [0, 0], [1, 0], [1, 1]]
```

The automaton must be deterministic:

```
sage: A = automata.Word([0]) * automata.Word([1])
sage: A.complement()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
sage: Ac = A.determinisation().complement()
sage: [w for w in ([], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1])
....:  if Ac(w)]
[[], [0], [1], [0, 0], [1, 0], [1, 1]]
```

```
>>> from sage.all import *
>>> A = automata.Word([Integer(0)]) * automata.Word([Integer(1)])
>>> A.complement()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
>>> Ac = A.determinisation().complement()
>>> [w for w in ([], [Integer(0)], [Integer(1)], [Integer(0), Integer(0)],
↪[Integer(0), Integer(1)], [Integer(1), Integer(0)], [Integer(1),
↪Integer(1)])
...  if Ac(w)]
[[], [0], [1], [0, 0], [1, 0], [1, 1]]
```

determinisation()

Return a deterministic automaton which accepts the same input words as the original one.

OUTPUT: a new automaton, which is deterministic

The labels of the states of the new automaton are frozensets of states of `self`. The color of a new state is the frozenset of colors of the constituent states of `self`. Therefore, the colors of the constituent states have to be hashable. However, if all constituent states have color `None`, then the resulting color is `None`, too.

The input alphabet must be specified.

EXAMPLES:

```
sage: aut = Automaton([('A', 'A', 0), ('A', 'B', 1), ('B', 'B', 1)],
....:                 initial_states=['A'], final_states=['B'])
sage: aut.determinisation().transitions()
[Transition from frozenset({'A'}) to frozenset({'A'}): 0|-,
 Transition from frozenset({'A'}) to frozenset({'B'}): 1|-,
 Transition from frozenset({'B'}) to frozenset(): 0|-,
 Transition from frozenset({'B'}) to frozenset({'B'}): 1|-,
 Transition from frozenset() to frozenset(): 0|-,
 Transition from frozenset() to frozenset(): 1|-]
```

```
>>> from sage.all import *
>>> aut = Automaton([('A', 'A', Integer(0)), ('A', 'B', Integer(1)), ('B', 'B
↪', Integer(1))],
...                 initial_states=['A'], final_states=['B'])
>>> aut.determinisation().transitions()
[Transition from frozenset({'A'}) to frozenset({'A'}): 0|-,
 Transition from frozenset({'A'}) to frozenset({'B'}): 1|-,
 Transition from frozenset({'B'}) to frozenset(): 0|-,
 Transition from frozenset({'B'}) to frozenset({'B'}): 1|-,
 Transition from frozenset() to frozenset(): 0|-,
 Transition from frozenset() to frozenset(): 1|-]
```

```
sage: A = Automaton([('A', 'A', 1), ('A', 'A', 0), ('A', 'B', 1),
....:                ('B', 'C', 0), ('C', 'C', 1), ('C', 'C', 0)],
....:               initial_states=['A'], final_states=['C'])
sage: A.determinisation().states()
[frozenset({'A'}),
 frozenset({'A', 'B'}),
```

(continues on next page)

(continued from previous page)

```
 frozenset({'A', 'C'}),
 frozenset({'A', 'B', 'C'})]
```

```
>>> from sage.all import *
>>> A = Automaton([('A', 'A', Integer(1)), ('A', 'A', Integer(0)), ('A', 'B',
↪Integer(1)),
...                    ('B', 'C', Integer(0)), ('C', 'C', Integer(1)), ('C', 'C',
↪Integer(0))],
...                   initial_states=['A'], final_states=['C'])
>>> A.determinisation().states()
[frozenset({'A'}),
 frozenset({'A', 'B'}),
 frozenset({'A', 'C'}),
 frozenset({'A', 'B', 'C'})]
```

```
sage: A = Automaton([(0, 1, 1), (0, 2, [1, 1]), (0, 3, [1, 1, 1]),
....:                    (1, 0, -1), (2, 0, -2), (3, 0, -3)],
....:                   initial_states=[0], final_states=[0, 1, 2, 3])
sage: B = A.determinisation().relabeled().coaccessible_components()
sage: sorted(B.transitions())
[Transition from 0 to 1: 1|-,
 Transition from 1 to 0: -1|-,
 Transition from 1 to 3: 1|-,
 Transition from 3 to 0: -2|-,
 Transition from 3 to 4: 1|-,
 Transition from 4 to 0: -3|-]
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(1), Integer(1)), (Integer(0),
↪Integer(2), [Integer(1), Integer(1)]), (Integer(0), Integer(3), [Integer(1),
↪ Integer(1), Integer(1)]),
...                    (Integer(1), Integer(0), -Integer(1)), (Integer(2),
↪Integer(0), -Integer(2)), (Integer(3), Integer(0), -Integer(3))],
...                   initial_states=[Integer(0)], final_states=[Integer(0),
↪Integer(1), Integer(2), Integer(3)])
>>> B = A.determinisation().relabeled().coaccessible_components()
>>> sorted(B.transitions())
[Transition from 0 to 1: 1|-,
 Transition from 1 to 0: -1|-,
 Transition from 1 to 3: 1|-,
 Transition from 3 to 0: -2|-,
 Transition from 3 to 4: 1|-,
 Transition from 4 to 0: -3|-]
```

Note that colors of states have to be hashable:

```
sage: A = Automaton([[0, 0, 0]], initial_states=[0])
sage: A.state(0).color = []
sage: A.determinisation()
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
TypeError: unhashable type: 'list'
sage: A.state(0).color = ()
sage: A.determinisation()
Automaton with 1 state
```

```
>>> from sage.all import *
>>> A = Automaton([[Integer(0), Integer(0), Integer(0)]], initial_
↪states=[Integer(0)])
>>> A.state(Integer(0)).color = []
>>> A.determinisation()
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
>>> A.state(Integer(0)).color = ()
>>> A.determinisation()
Automaton with 1 state
```

If the colors of all constituent states are `None`, the resulting color is `None`, too (Issue #19199):

```
sage: A = Automaton([(0, 0, 0)],
....:               initial_states=[0],
....:               final_states=[0])
sage: [s.color for s in A.determinisation().iter_states()]
[None]
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(0), Integer(0))],
...               initial_states=[Integer(0)],
...               final_states=[Integer(0)])
>>> [s.color for s in A.determinisation().iter_states()]
[None]
```

intersection(*other*, *only_accessible_components=True*)

Return a new automaton which accepts an input if it is accepted by both given automata.

INPUT:

- `other` – an automaton
- `only_accessible_components` – if `True` (default), then the result is piped through *accessible_components()*. If no `new_input_alphabet` is given, it is determined by *determine_alphabets()*.

OUTPUT:

A new automaton which computes the intersection (see below) of the languages of `self` and `other`.

The set of states of the new automaton is the Cartesian product of the set of states of both given automata. There is a transition $((A, B), (C, D), a)$ in the new automaton if there are transitions (A, C, a) and (B, D, a) in the old automata.

The methods *intersection()* and *cartesian_product()* are the same (for automata).

EXAMPLES:

```
sage: aut1 = Automaton([('1', '2', 1),
....:                   ('2', '2', 1),
....:                   ('2', '2', 0)],
....:                  initial_states=['1'],
....:                  final_states=['2'],
....:                  determine_alphabets=True)
sage: aut2 = Automaton([('A', 'A', 1),
....:                   ('A', 'B', 0),
....:                   ('B', 'B', 0),
....:                   ('B', 'A', 1)],
....:                  initial_states=['A'],
....:                  final_states=['B'],
....:                  determine_alphabets=True)
sage: res = aut1.intersection(aut2)
sage: (aut1([1, 1]), aut2([1, 1]), res([1, 1]))
(True, False, False)
sage: (aut1([1, 0]), aut2([1, 0]), res([1, 0]))
(True, True, True)
sage: res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'A') to ('2', 'B'): 0|-,
 Transition from ('2', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'B') to ('2', 'B'): 0|-,
 Transition from ('2', 'B') to ('2', 'A'): 1|-]
```

```
>>> from sage.all import *
>>> aut1 = Automaton([('1', '2', Integer(1)),
...                   ('2', '2', Integer(1)),
...                   ('2', '2', Integer(0))],
...                  initial_states=['1'],
...                  final_states=['2'],
...                  determine_alphabets=True)
>>> aut2 = Automaton([('A', 'A', Integer(1)),
...                   ('A', 'B', Integer(0)),
...                   ('B', 'B', Integer(0)),
...                   ('B', 'A', Integer(1))],
...                  initial_states=['A'],
...                  final_states=['B'],
...                  determine_alphabets=True)
>>> res = aut1.intersection(aut2)
>>> (aut1([Integer(1), Integer(1)]), aut2([Integer(1), Integer(1)]),␣
↪res([Integer(1), Integer(1)]))
(True, False, False)
>>> (aut1([Integer(1), Integer(0)]), aut2([Integer(1), Integer(0)]),␣
↪res([Integer(1), Integer(0)]))
(True, True, True)
>>> res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'A') to ('2', 'B'): 0|-,
 Transition from ('2', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'B') to ('2', 'B'): 0|-,
 Transition from ('2', 'B') to ('2', 'A'): 1|-]
```

For automata with epsilon-transitions, intersection is not well defined. But for any finite state machine, epsilon-transitions can be removed by `remove_epsilon_transitions()`.

```
sage: a1 = Automaton([(0, 0, 0),
....:                 (0, 1, None),
....:                 (1, 1, 1),
....:                 (1, 2, 1)],
....:                 initial_states=[0],
....:                 final_states=[1],
....:                 determine_alphabets=True)
sage: a2 = Automaton([(0, 0, 0), (0, 1, 1), (1, 1, 1)],
....:                 initial_states=[0],
....:                 final_states=[1],
....:                 determine_alphabets=True)
sage: a1.intersection(a2)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input)
was found.
sage: a1.remove_epsilon_transitions()  # not tested (since not implemented␣
↪yet)
sage: a1.intersection(a2)  # not tested
```

```
>>> from sage.all import *
>>> a1 = Automaton([(Integer(0), Integer(0), Integer(0)),
...                 (Integer(0), Integer(1), None),
...                 (Integer(1), Integer(1), Integer(1)),
...                 (Integer(1), Integer(2), Integer(1))],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(1)],
...                 determine_alphabets=True)
>>> a2 = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)), (Integer(1), Integer(1), Integer(1))],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(1)],
...                 determine_alphabets=True)
>>> a1.intersection(a2)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input)
was found.
>>> a1.remove_epsilon_transitions()  # not tested (since not implemented yet)
>>> a1.intersection(a2)  # not tested
```

is_equivalent(*other*)

Test whether two automata are equivalent, i.e., accept the same language.

INPUT:

- `other` – an `Automaton`

EXAMPLES:

```
sage: A = Automaton([(0, 0, 0), (0, 1, 1), (1, 0, 1)],
....:                 initial_states=[0],
```

(continues on next page)

(continued from previous page)

```
....:                final_states=[0])
sage: B = Automaton([('a', 'a', 0), ('a', 'b', 1), ('b', 'a', 1)],
....:                initial_states=['a'],
....:                final_states=['a'])
sage: A.is_equivalent(B)
True
sage: B.add_transition('b', 'a', 0)
Transition from 'b' to 'a': 0|-
sage: A.is_equivalent(B)
False
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)), (Integer(1), Integer(0), Integer(1))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(0)])
>>> B = Automaton([('a', 'a', Integer(0)), ('a', 'b', Integer(1)), ('b', 'a',␣
↪Integer(1))],
...                initial_states=['a'],
...                final_states=['a'])
>>> A.is_equivalent(B)
True
>>> B.add_transition('b', 'a', Integer(0))
Transition from 'b' to 'a': 0|-
>>> A.is_equivalent(B)
False
```

language(*max_length=None*, ***kwargs*)

Return all words accepted by this automaton.

INPUT:

- `max_length` – integer or `None` (default). Only inputs of length at most `max_length` will be considered. If `None`, then this iterates over all possible words without length restrictions.
- `kwargs` – will be passed on to the *process iterator*. See *process()* for a description.

OUTPUT: an iterator

EXAMPLES:

```
sage: NAF = Automaton(
....:     {'A': [('A', 0), ('B', 1), ('B', -1)],
....:      'B': [('A', 0)]},
....:     initial_states=['A'], final_states=['A', 'B'])
sage: list(NAF.language(3))
[[],
 [0], [-1], [1],
 [-1, 0], [0, 0], [1, 0], [0, -1], [0, 1],
 [-1, 0, 0], [0, -1, 0], [0, 0, 0], [0, 1, 0], [1, 0, 0],
 [-1, 0, -1], [-1, 0, 1], [0, 0, -1],
 [0, 0, 1], [1, 0, -1], [1, 0, 1]]
```

```
>>> from sage.all import *
>>> NAF = Automaton(
...     {'A': [('A', Integer(0)), ('B', Integer(1)), ('B', -Integer(1))],
...      'B': [('A', Integer(0))]},
...     initial_states=['A'], final_states=['A', 'B'])
>>> list(NAF.language(Integer(3)))
[[],
 [0], [-1], [1],
 [-1, 0], [0, 0], [1, 0], [0, -1], [0, 1],
 [-1, 0, 0], [0, -1, 0], [0, 0, 0], [0, 1, 0], [1, 0, 0],
 [-1, 0, -1], [-1, 0, 1], [0, 0, -1],
 [0, 0, 1], [1, 0, -1], [1, 0, 1]]
```

See also

FiniteStateMachine.language(), *process()*.

minimization (*algorithm=None*)

Return the minimization of the input automaton as a new automaton.

INPUT:

- `algorithm` – either Moore's algorithm (by `algorithm='Moore'` or as default for deterministic automata) or Brzozowski's algorithm (when `algorithm='Brzozowski'` or when the automaton is not deterministic) is used.

OUTPUT: a new automaton

The resulting automaton is deterministic and has a minimal number of states.

EXAMPLES:

```
sage: A = Automaton([('A', 'A', 1), ('A', 'A', 0), ('A', 'B', 1),
....:                ('B', 'C', 0), ('C', 'C', 1), ('C', 'C', 0)],
....:               initial_states=['A'], final_states=['C'])
sage: B = A.minimization(algorithm='Brzozowski')
sage: B_trans = B.transitions(B.states()[1])
sage: B_trans # random
[Transition from frozenset({frozenset({'B', 'C'}),
                            frozenset({'A', 'C'}),
                            frozenset({'A', 'B', 'C'})})
            to frozenset({frozenset({'C'}),
                          frozenset({'B', 'C'}),
                          frozenset({'A', 'C'}),
                          frozenset({'A', 'B', 'C'})}):
            0|-,
 Transition from frozenset({frozenset({'B', 'C'}),
                            frozenset({'A', 'C'}),
                            frozenset({'A', 'B', 'C'})})
            to frozenset({frozenset({'B', 'C'}),
                          frozenset({'A', 'C'}),
                          frozenset({'A', 'B', 'C'})}):
            1|-]
sage: len(B.states())
```

(continues on next page)

(continued from previous page)

```
3
sage: C = A.minimization(algorithm='Brzozowski')
sage: C_trans = C.transitions(C.states()[1])
sage: B_trans == C_trans
True
sage: len(C.states())
3
```

```
>>> from sage.all import *
>>> A = Automaton([('A', 'A', Integer(1)), ('A', 'A', Integer(0)), ('A', 'B',␣
↪Integer(1)),
...                 ('B', 'C', Integer(0)), ('C', 'C', Integer(1)), ('C', 'C',␣
↪Integer(0))],
...                initial_states=['A'], final_states=['C'])
>>> B = A.minimization(algorithm='Brzozowski')
>>> B_trans = B.transitions(B.states()[Integer(1)])
>>> B_trans # random
[Transition from frozenset({frozenset({'B', 'C'}),
                            frozenset({'A', 'C'}),
                            frozenset({'A', 'B', 'C'})})
            to frozenset({frozenset({'C'}),
                          frozenset({'B', 'C'}),
                          frozenset({'A', 'C'}),
                          frozenset({'A', 'B', 'C'})}):
            0|-,
 Transition from frozenset({frozenset({'B', 'C'}),
                            frozenset({'A', 'C'}),
                            frozenset({'A', 'B', 'C'})})
            to frozenset({frozenset({'B', 'C'}),
                          frozenset({'A', 'C'}),
                          frozenset({'A', 'B', 'C'})}):
            1|-]
>>> len(B.states())
3
>>> C = A.minimization(algorithm='Brzozowski')
>>> C_trans = C.transitions(C.states()[Integer(1)])
>>> B_trans == C_trans
True
>>> len(C.states())
3
```

```
sage: aut = Automaton([('1', '2', 'a'), ('2', '3', 'b'),
....:                  ('3', '2', 'a'), ('2', '1', 'b'),
....:                  ('3', '4', 'a'), ('4', '3', 'b')],
....:                 initial_states=['1'], final_states=['1'])
sage: min = aut.minimization(algorithm='Brzozowski')
sage: [len(min.states()), len(aut.states())]
[3, 4]
sage: min = aut.minimization(algorithm='Moore')
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
NotImplementedError: Minimization via Moore's Algorithm is only
implemented for deterministic finite state machines
```

```
>>> from sage.all import *
>>> aut = Automaton([('1', '2', 'a'), ('2', '3', 'b'),
...                  ('3', '2', 'a'), ('2', '1', 'b'),
...                  ('3', '4', 'a'), ('4', '3', 'b')],
...                 initial_states=['1'], final_states=['1'])
>>> min = aut.minimization(algorithm='Brzozowski')
>>> [len(min.states()), len(aut.states())]
[3, 4]
>>> min = aut.minimization(algorithm='Moore')
Traceback (most recent call last):
...
NotImplementedError: Minimization via Moore's Algorithm is only
implemented for deterministic finite state machines
```

process(**args*, ***kwargs*)

Return whether the automaton accepts the input and the state where the computation stops.

INPUT:

- `input_tape` – the input tape can be a list or an iterable with entries from the input alphabet. If we are working with a multi-tape machine (see parameter `use_multitape_input` and notes below), then the tape is a list or tuple of tracks, each of which can be a list or an iterable with entries from the input alphabet.
- `initial_state` or `initial_states` – the initial state(s) in which the machine starts. Either specify a single one with `initial_state` or a list of them with `initial_states`. If both are given, `initial_state` will be appended to `initial_states`. If neither is specified, the initial states of the finite state machine are taken.
- `list_of_outputs` – (default: `None`) a boolean or `None`. If `True`, then the outputs are given in list form (even if we have no or only one single output). If `False`, then the result is never a list (an exception is raised if the result cannot be returned). If `list_of_outputs=None` the method determines automatically what to do (e.g. if a non-deterministic machine returns more than one path, then the output is returned in list form).
- `only_accepted` – boolean (default: `False`); if set, then the first argument in the output is guaranteed to be `True` (if the output is a list, then the first argument of each element will be `True`).
- `full_output` – boolean (default: `True`); if set, then the full output is given, otherwise only whether the sequence is accepted or not (the first entry below only).
- `always_include_output` – if set (not by default), always return a triple containing the (non-existing) output. This is in order to obtain output compatible with that of `FiniteStateMachine.process()`. If this parameter is set, `full_output` has no effect.
- `format_output` – a function that translates the written output (which is in form of a list) to something more readable. By default (`None`) identity is used here.
- `check_epsilon_transitions` – boolean (default: `True`); a boolean. If `False`, then epsilon transitions are not taken into consideration during process.
- `write_final_word_out` – boolean (default: `True`); whether the final output words should be written or not

- `use_multitape_input` – boolean (default: `False`); if `True`, then the multi-tape mode of the process iterator is activated. See also the notes below for multi-tape machines.
- `process_all_prefixes_of_input` – boolean (default: `False`); if `True`, then each prefix of the input word is processed (instead of processing the whole input word at once). Consequently, there is an output generated for each of these prefixes.
- `process_iterator_class` – (default: `None`) a class inherited from *FSMProcessIterator*. If `None`, then *FSMProcessIterator* is taken. An instance of this class is created and is used during the processing.

OUTPUT:

The full output is a pair (or a list of pairs, cf. parameter `list_of_outputs`), where

- the first entry is `True` if the input string is accepted and
- the second gives the state reached after processing the input tape (This is a state with label `None` if the input could not be processed, i.e., if at one point no transition to go on could be found.).

If `full_output` is `False`, then only the first entry is returned.

If `always_include_output` is set, an additional third entry `[]` is included.

Note that in the case the automaton is not deterministic, all possible paths are taken into account. You can use *determinisation()* to get a deterministic automaton machine.

This function uses an iterator which, in its simplest form, goes from one state to another in each step. To decide which way to go, it uses the input words of the outgoing transitions and compares them to the input tape. More precisely, in each step, the iterator takes an outgoing transition of the current state, whose input label equals the input letter of the tape.

If the choice of the outgoing transition is not unique (i.e., we have a non-deterministic finite state machine), all possibilities are followed. This is done by splitting the process into several branches, one for each of the possible outgoing transitions.

The process (iteration) stops if all branches are finished, i.e., for no branch, there is any transition whose input word coincides with the processed input tape. This can simply happen when the entire tape was read.

Also see *__call__()* for a version of *process()* with shortened output.

Internally this function creates and works with an instance of *FSMProcessIterator*. This iterator can also be obtained with *iter_process()*.

If working with multi-tape finite state machines, all input words of transitions are words of k-tuples of letters. Moreover, the input tape has to consist of k tracks, i.e., be a list or tuple of k iterators, one for each track.

> **Warning**
>
> Working with multi-tape finite state machines is still experimental and can lead to wrong outputs.

EXAMPLES:

In the following examples, we construct an automaton which accepts non-adjacent forms (see also the example on *non-adjacent forms* in the documentation of the module *Finite state machines, automata, transducers*) and then test it by feeding it with several binary digit expansions.

```
sage: NAF = Automaton(
....:     {'_': [('_', 0), ('1', 1)], '1': [('_', 0)]},
....:     initial_states=['_'], final_states=['_', '1'])
```

(continues on next page)

(continued from previous page)

```
sage: [NAF.process(w) for w in [[0], [0, 1], [1, 1], [0, 1, 0, 1],
....:                           [0, 1, 1, 1, 0], [1, 0, 0, 1, 1]]]
[(True, '_'), (True, '1'), (False, None),
 (True, '1'), (False, None), (False, None)]
```

```
>>> from sage.all import *
>>> NAF = Automaton(
...     {'_': [('_', Integer(0)), ('1', Integer(1))], '1': [('_',␣
↪Integer(0))]},
...     initial_states=['_'], final_states=['_', '1'])
>>> [NAF.process(w) for w in [[Integer(0)], [Integer(0), Integer(1)],␣
↪[Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(0), Integer(1)],
...                           [Integer(0), Integer(1), Integer(1), Integer(1),
↪ Integer(0)], [Integer(1), Integer(0), Integer(0), Integer(1), Integer(1)]]]
[(True, '_'), (True, '1'), (False, None),
 (True, '1'), (False, None), (False, None)]
```

If we just want a condensed output, we use:

```
sage: [NAF.process(w, full_output=False)
....:     for w in [[0], [0, 1], [1, 1], [0, 1, 0, 1],
....:               [0, 1, 1, 1, 0], [1, 0, 0, 1, 1]]]
[True, True, False, True, False, False]
```

```
>>> from sage.all import *
>>> [NAF.process(w, full_output=False)
...     for w in [[Integer(0)], [Integer(0), Integer(1)], [Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(0), Integer(1)],
...               [Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0)], [Integer(1), Integer(0), Integer(0), Integer(1), Integer(1)]]]
[True, True, False, True, False, False]
```

It is equivalent to:

```
sage: [NAF(w) for w in [[0], [0, 1], [1, 1], [0, 1, 0, 1],
....:                   [0, 1, 1, 1, 0], [1, 0, 0, 1, 1]]]
[True, True, False, True, False, False]
```

```
>>> from sage.all import *
>>> [NAF(w) for w in [[Integer(0)], [Integer(0), Integer(1)], [Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(0), Integer(1)],
...                   [Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0)], [Integer(1), Integer(0), Integer(0), Integer(1), Integer(1)]]]
[True, True, False, True, False, False]
```

The following example illustrates the difference between non-existing paths and reaching a non-final state:

```
sage: NAF.process([2])
(False, None)
sage: NAF.add_transition(('_', 's', 2))
Transition from '_' to 's': 2|-
```

(continues on next page)

(continued from previous page)

```
sage: NAF.process([2])
(False, 's')
```

```
>>> from sage.all import *
>>> NAF.process([Integer(2)])
(False, None)
>>> NAF.add_transition(('_', 's', Integer(2)))
Transition from '_' to 's': 2|-
>>> NAF.process([Integer(2)])
(False, 's')
```

A simple example of a (non-deterministic) multi-tape automaton is the following: It checks whether the two input tapes have the same number of ones:

```
sage: M = Automaton([('=', '=', (1, 1)),
....:                ('=', '=', (None, 0)),
....:                ('=', '=', (0, None)),
....:                ('=', '<', (None, 1)),
....:                ('<', '<', (None, 1)),
....:                ('<', '<', (None, 0)),
....:                ('=', '>', (1, None)),
....:                ('>', '>', (1, None)),
....:                ('>', '>', (0, None))],
....:               initial_states=['='],
....:               final_states=['='])
sage: M.process(([1, 0, 1], [1, 0]), use_multitape_input=True)
(False, '>')
sage: M.process(([0, 1, 0], [0, 1, 1]), use_multitape_input=True)
(False, '<')
sage: M.process(([1, 1, 0, 1], [0, 0, 1, 0, 1, 1]),
....:           use_multitape_input=True)
(True, '=')
```

```
>>> from sage.all import *
>>> M = Automaton([('=', '=', (Integer(1), Integer(1))),
...                ('=', '=', (None, Integer(0))),
...                ('=', '=', (Integer(0), None)),
...                ('=', '<', (None, Integer(1))),
...                ('<', '<', (None, Integer(1))),
...                ('<', '<', (None, Integer(0))),
...                ('=', '>', (Integer(1), None)),
...                ('>', '>', (Integer(1), None)),
...                ('>', '>', (Integer(0), None))],
...               initial_states=['='],
...               final_states=['='])
>>> M.process(([Integer(1), Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]), use_multitape_input=True)
(False, '>')
>>> M.process(([Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(1),␣
↪Integer(1)]), use_multitape_input=True)
(False, '<')
>>> M.process(([Integer(1), Integer(1), Integer(0), Integer(1)], [Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0), Integer(1), Integer(0), Integer(1), Integer(1)]),
...            use_multitape_input=True)
(True, '=')
```

Alternatively, we can use the following (non-deterministic) multi-tape automaton for the same check:

```
sage: N = Automaton([('=', '=', (0, 0)),
....:                  ('=', '<', (None, 1)),
....:                  ('<', '<', (0, None)),
....:                  ('<', '=', (1, None)),
....:                  ('=', '>', (1, None)),
....:                  ('>', '>', (None, 0)),
....:                  ('>', '=', (None, 1))],
....:                 initial_states=['='],
....:                 final_states=['='])
sage: N.process(([1, 0, 1], [1, 0]), use_multitape_input=True)
(False, '>')
sage: N.process(([0, 1, 0], [0, 1, 1]), use_multitape_input=True)
(False, '<')
sage: N.process(([1, 1, 0, 1], [0, 0, 1, 0, 1, 1]),
....:           use_multitape_input=True)
(True, '=')
```

```
>>> from sage.all import *
>>> N = Automaton([('=', '=', (Integer(0), Integer(0))),
...                  ('=', '<', (None, Integer(1))),
...                  ('<', '<', (Integer(0), None)),
...                  ('<', '=', (Integer(1), None)),
...                  ('=', '>', (Integer(1), None)),
...                  ('>', '>', (None, Integer(0))),
...                  ('>', '=', (None, Integer(1)))],
...                 initial_states=['='],
...                 final_states=['='])
>>> N.process(([Integer(1), Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]), use_multitape_input=True)
(False, '>')
>>> N.process(([Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(1),␣
↪Integer(1)]), use_multitape_input=True)
(False, '<')
>>> N.process(([Integer(1), Integer(1), Integer(0), Integer(1)], [Integer(0),␣
↪Integer(0), Integer(1), Integer(0), Integer(1), Integer(1)]),
...            use_multitape_input=True)
(True, '=')
```

See also

FiniteStateMachine.process(), *Transducer.process()*, *iter_process()*, *__call__()*, *FSMProcessIterator*.

shannon_parry_markov_chain()

Compute a time homogeneous Markov chain such that all words of a given length recognized by the original automaton occur as the output with the same weight; the transition probabilities correspond to the Parry measure.

OUTPUT:

A Markov chain. Its input labels are the transition probabilities, the output labels the labels of the original automaton. In order to obtain equal weight for all words of the same length, an "exit weight" is needed. It is stored in the attribute `color` of the states of the Markov chain. The weights of the words of the same length sum up to one up to an exponentially small error.

The stationary distribution of this Markov chain is saved as the initial probabilities of the states.

The transition probabilities correspond to the Parry measure (see [S1948] and [P1964]).

The automaton is assumed to be deterministic, irreducible and aperiodic. All states must be final.

EXAMPLES:

```
sage: NAF = Automaton([(0, 0, 0), (0, 1, 1), (0, 1, -1),
....:                  (1, 0, 0)], initial_states=[0],
....:                 final_states=[0, 1])
sage: P_NAF = NAF.shannon_parry_markov_chain()                          #␣
↪needs sage.symbolic
sage: P_NAF.transitions()                                               #␣
↪needs sage.symbolic
[Transition from 0 to 0: 1/2|0,
 Transition from 0 to 1: 1/4|1,
 Transition from 0 to 1: 1/4|-1,
 Transition from 1 to 0: 1|0]
sage: for s in P_NAF.iter_states():                                     #␣
↪needs sage.symbolic
....:     print(s.color)
3/4
3/2
```

```
>>> from sage.all import *
>>> NAF = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)), (Integer(0), Integer(1), -Integer(1)),
...                  (Integer(1), Integer(0), Integer(0))], initial_
↪states=[Integer(0)],
...                 final_states=[Integer(0), Integer(1)])
>>> P_NAF = NAF.shannon_parry_markov_chain()                            #␣
↪needs sage.symbolic
>>> P_NAF.transitions()                                                 #␣
↪needs sage.symbolic
[Transition from 0 to 0: 1/2|0,
 Transition from 0 to 1: 1/4|1,
 Transition from 0 to 1: 1/4|-1,
 Transition from 1 to 0: 1|0]
>>> for s in P_NAF.iter_states():                                       #␣
↪needs sage.symbolic
...     print(s.color)
3/4
3/2
```

The stationary distribution is also computed and saved as the initial probabilities of the returned Markov chain:

```
sage: for s in P_NAF.states():                                          #␣
↪needs sage.symbolic
....:     print("{} {}".format(s, s.initial_probability))
0 2/3
1 1/3
```

```
>>> from sage.all import *
>>> for s in P_NAF.states():                                          #␣
↪needs sage.symbolic
...     print("{} {}".format(s, s.initial_probability))
0 2/3
1 1/3
```

The automaton is assumed to be deterministic, irreducible and aperiodic:

```
sage: A = Automaton([(0, 0, 0), (0, 1, 1), (1, 1, 1), (1, 1, 0)],
....:               initial_states=[0])
sage: A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: Automaton must be strongly connected.
sage: A = Automaton([(0, 0, 0), (0, 1, 0)],
....:               initial_states=[0])
sage: A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: Automaton must be deterministic.
sage: A = Automaton([(0, 1, 0), (1, 0, 0)],
....:               initial_states=[0])
sage: A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: Automaton must be aperiodic.
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)), (Integer(1), Integer(1), Integer(1)), (Integer(1),␣
↪Integer(1), Integer(0))],
...               initial_states=[Integer(0)])
>>> A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: Automaton must be strongly connected.
>>> A = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(0))],
...               initial_states=[Integer(0)])
>>> A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: Automaton must be deterministic.
>>> A = Automaton([(Integer(0), Integer(1), Integer(0)), (Integer(1),␣
↪Integer(0), Integer(0))],
```

(continues on next page)

(continued from previous page)

```
...                 initial_states=[Integer(0)])
>>> A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: Automaton must be aperiodic.
```

All states must be final:

```
sage: A = Automaton([(0, 1, 0), (0, 0, 1), (1, 0, 0)],
....:                 initial_states=[0])
sage: A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: All states must be final.
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(1), Integer(0)), (Integer(0),␣
↪Integer(0), Integer(1)), (Integer(1), Integer(0), Integer(0))],
...                 initial_states=[Integer(0)])
>>> A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: All states must be final.
```

ALGORITHM:

See [HKP2015a], Lemma 4.1.

REFERENCES:

with_output(*word_out_function=None*)

Construct a transducer out of this automaton.

INPUT:

- `word_out_function` – (default: `None`) a function. It transforms a *transition* to the output word for this transition.

  If this is `None`, then the output word will be equal to the input word of each transition.

OUTPUT: a transducer

EXAMPLES:

```
sage: A = Automaton([(0, 0, 'A'), (0, 1, 'B'), (1, 2, 'C')])
sage: T = A.with_output(); T
Transducer with 3 states
sage: T.transitions()
[Transition from 0 to 0: 'A'|'A',
 Transition from 0 to 1: 'B'|'B',
 Transition from 1 to 2: 'C'|'C']
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(0), 'A'), (Integer(0), Integer(1), 'B
↪'), (Integer(1), Integer(2), 'C')])
>>> T = A.with_output(); T
```

(continues on next page)

(continued from previous page)

```
Transducer with 3 states
>>> T.transitions()
[Transition from 0 to 0: 'A'|'A',
 Transition from 0 to 1: 'B'|'B',
 Transition from 1 to 2: 'C'|'C']
```

This result is in contrast to:

```
sage: Transducer(A).transitions()
[Transition from 0 to 0: 'A'|-,
 Transition from 0 to 1: 'B'|-,
 Transition from 1 to 2: 'C'|-]
```

```
>>> from sage.all import *
>>> Transducer(A).transitions()
[Transition from 0 to 0: 'A'|-,
 Transition from 0 to 1: 'B'|-,
 Transition from 1 to 2: 'C'|-]
```

where no output labels are created.

Here is another example:

```
sage: T2 = A.with_output(lambda t: [c.lower() for c in t.word_in])
sage: T2.transitions()
[Transition from 0 to 0: 'A'|'a',
 Transition from 0 to 1: 'B'|'b',
 Transition from 1 to 2: 'C'|'c']
```

```
>>> from sage.all import *
>>> T2 = A.with_output(lambda t: [c.lower() for c in t.word_in])
>>> T2.transitions()
[Transition from 0 to 0: 'A'|'a',
 Transition from 0 to 1: 'B'|'b',
 Transition from 1 to 2: 'C'|'c']
```

We can obtain the same result by composing two transducers. As inner transducer of the composition, we use *with_output()* without the optional argument `word_out_function` (which makes the output of each transition equal to its input); as outer transducer we use a *map-transducer* (for converting to lower case). This gives

```
sage: L = transducers.map(lambda x: x.lower(), ['A', 'B', 'C'])
sage: L.composition(A.with_output()).relabeled().transitions()
[Transition from 0 to 0: 'A'|'a',
 Transition from 0 to 1: 'B'|'b',
 Transition from 1 to 2: 'C'|'c']
```

```
>>> from sage.all import *
>>> L = transducers.map(lambda x: x.lower(), ['A', 'B', 'C'])
>>> L.composition(A.with_output()).relabeled().transitions()
[Transition from 0 to 0: 'A'|'a',
 Transition from 0 to 1: 'B'|'b',
 Transition from 1 to 2: 'C'|'c']
```

See also

input_projection(), *output_projection()*, *Transducer*, *transducers.map()*.

sage.combinat.finite_state_machine.**FSMLetterSymbol**(*letter*)

Return a string associated to the input letter.

INPUT:

- `letter` – the input letter or `None` (representing the empty word)

OUTPUT:

If `letter` is `None` the symbol for the empty word `FSMEmptyWordSymbol` is returned, otherwise the string associated to the letter.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMLetterSymbol
sage: FSMLetterSymbol(0)
'0'
sage: FSMLetterSymbol(None)
'-'
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMLetterSymbol
>>> FSMLetterSymbol(Integer(0))
'0'
>>> FSMLetterSymbol(None)
'-'
```

class sage.combinat.finite_state_machine.**FSMProcessIterator**(*fsm*, *input_tape=None*, *initial_state=None*, *initial_states=[]*, *use_multitape_input=False*, *check_epsilon_transitions=True*, *write_final_word_out=True*, *format_output=None*, *process_all_prefixes_of_input=False*, ***kwargs*)

Bases: `SageObject`, `Iterator`

This class takes an input, feeds it into a finite state machine (automaton or transducer, in particular), tests whether this was successful and calculates the written output.

INPUT:

- `fsm` – the finite state machine on which the input should be processed
- `input_tape` – the input tape can be a list or an iterable with entries from the input alphabet. If we are working with a multi-tape machine (see parameter `use_multitape_input` and notes below), then the tape is a list or tuple of tracks, each of which can be a list or an iterable with entries from the input alphabet.
- `initial_state` or `initial_states` – the initial state(s) in which the machine starts. Either specify a single one with `initial_state` or a list of them with `initial_states`. If both are given, `initial_state` will be appended to `initial_states`. If neither is specified, the initial states of the finite state machine are taken.

- `format_output` – a function that translates the written output (which is in form of a list) to something more readable. By default (`None`) identity is used here.
- `check_epsilon_transitions` – boolean (default: `True`); if `False`, then epsilon transitions are not taken into consideration during process
- `write_final_word_out` – boolean (default: `True`); whether the final output words should be written or not
- `use_multitape_input` – boolean (default: `False`); if `True`, then the multi-tape mode of the process iterator is activated. See also the notes below for multi-tape machines.
- `process_all_prefixes_of_input` – boolean (default: `False`); if `True`, then each prefix of the input word is processed (instead of processing the whole input word at once). Consequently, there is an output generated for each of these prefixes.

OUTPUT: an iterator

In its simplest form, it behaves like an iterator which, in each step, goes from one state to another. To decide which way to go, it uses the input words of the outgoing transitions and compares them to the input tape. More precisely, in each step, the process iterator takes an outgoing transition of the current state, whose input label equals the input letter of the tape. The output label of the transition, if present, is written on the output tape.

If the choice of the outgoing transition is not unique (i.e., we have a non-deterministic finite state machine), all possibilities are followed. This is done by splitting the process into several branches, one for each of the possible outgoing transitions.

The process (iteration) stops if all branches are finished, i.e., for no branch, there is any transition whose input word coincides with the processed input tape. This can simply happen when the entire tape was read. When the process stops, a `StopIteration` exception is thrown.

Warning: Processing an input tape of length n usually takes at least $n+1$ iterations, since there will be $n+1$ states visited (in the case the taken transitions have input words consisting of single letters).

An instance of this class is generated when *FiniteStateMachine.process()* or *FiniteStateMachine.iter_process()* of a finite state machine, an automaton, or a transducer is invoked.

When working with multi-tape finite state machines, all input words of transitions are words of k-tuples of letters. Moreover, the input tape has to consist of k tracks, i.e., be a list or tuple of k iterators, one for each track.

Warning: Working with multi-tape finite state machines is still experimental and can lead to wrong outputs.

EXAMPLES:

The following transducer reads binary words and outputs a word, where blocks of ones are replaced by just a single one. Further only words that end with a zero are accepted.

```
sage: T = Transducer({'A': [('A', 0, 0), ('B', 1, None)],
....:                 'B': [('B', 1, None), ('A', 0, [1, 0])]},
....:     initial_states=['A'], final_states=['A'])
sage: input = [1, 1, 0, 0, 1, 0, 1, 1, 1, 0]
```

(continues on next page)

(continued from previous page)

```
sage: T.process(input)
(True, 'A', [1, 0, 0, 1, 0, 1, 0])
```

```
>>> from sage.all import *
>>> T = Transducer({'A': [('A', Integer(0), Integer(0)), ('B', Integer(1), None)],
...                 'B': [('B', Integer(1), None), ('A', Integer(0), [Integer(1),
↪Integer(0)])]},
...     initial_states=['A'], final_states=['A'])
>>> input = [Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),
↪Integer(0), Integer(1), Integer(1), Integer(1), Integer(0)]
>>> T.process(input)
(True, 'A', [1, 0, 0, 1, 0, 1, 0])
```

The function `FiniteStateMachine.process()` (internally) uses a `FSMProcessIterator`. We can do that manually, too, and get full access to the iteration process:

```
sage: from sage.combinat.finite_state_machine import FSMProcessIterator
sage: it = FSMProcessIterator(T, input_tape=input)
sage: for current in it:
....:     print(current)
process (1 branch)
+ at state 'B'
+-- tape at 1, [[]]
process (1 branch)
+ at state 'B'
+-- tape at 2, [[]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 4, [[1, 0, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 5, [[1, 0, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 6, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 7, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 8, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 9, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 10, [[1, 0, 0, 1, 0, 1, 0]]
process (0 branches)
sage: it.result()
```

(continues on next page)

(continued from previous page)

```
[Branch(accept=True, state='A', output=[1, 0, 0, 1, 0, 1, 0])]
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMProcessIterator
>>> it = FSMProcessIterator(T, input_tape=input)
>>> for current in it:
...     print(current)
process (1 branch)
+ at state 'B'
+-- tape at 1, [[]]
process (1 branch)
+ at state 'B'
+-- tape at 2, [[]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 4, [[1, 0, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 5, [[1, 0, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 6, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 7, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 8, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 9, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 10, [[1, 0, 0, 1, 0, 1, 0]]
process (0 branches)
>>> it.result()
[Branch(accept=True, state='A', output=[1, 0, 0, 1, 0, 1, 0])]
```

```
sage: T = Transducer([(0, 0, 0, 'a'), (0, 1, 0, 'b'),
....:                 (1, 2, 1, 'c'), (2, 0, 0, 'd'),
....:                 (2, 1, None, 'd')],
....:                initial_states=[0], final_states=[2])
sage: sorted(T.process([0, 0, 1], format_output=lambda o: ''.join(o)))
[(False, 1, 'abcd'), (True, 2, 'abc')]
sage: it = FSMProcessIterator(T, input_tape=[0, 0, 1],
....:                         format_output=lambda o: ''.join(o))
sage: for current in it:
....:     print(current)
```

(continues on next page)

(continued from previous page)

```
process (2 branches)
+ at state 0
+-- tape at 1, [['a']]
+ at state 1
+-- tape at 1, [['b']]
process (2 branches)
+ at state 0
+-- tape at 2, [['a', 'a']]
+ at state 1
+-- tape at 2, [['a', 'b']]
process (2 branches)
+ at state 1
+-- tape at 3, [['a', 'b', 'c', 'd']]
+ at state 2
+-- tape at 3, [['a', 'b', 'c']]
process (0 branches)
sage: sorted(it.result())
[Branch(accept=False, state=1, output='abcd'),
 Branch(accept=True, state=2, output='abc')]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(0), Integer(0), 'a'), (Integer(0),␣
↪Integer(1), Integer(0), 'b'),
...                 (Integer(1), Integer(2), Integer(1), 'c'), (Integer(2),␣
↪Integer(0), Integer(0), 'd'),
...                 (Integer(2), Integer(1), None, 'd')],
...                initial_states=[Integer(0)], final_states=[Integer(2)])
>>> sorted(T.process([Integer(0), Integer(0), Integer(1)], format_output=lambda␣
↪o: ''.join(o)))
[(False, 1, 'abcd'), (True, 2, 'abc')]
>>> it = FSMProcessIterator(T, input_tape=[Integer(0), Integer(0), Integer(1)],
...                         format_output=lambda o: ''.join(o))
>>> for current in it:
...     print(current)
process (2 branches)
+ at state 0
+-- tape at 1, [['a']]
+ at state 1
+-- tape at 1, [['b']]
process (2 branches)
+ at state 0
+-- tape at 2, [['a', 'a']]
+ at state 1
+-- tape at 2, [['a', 'b']]
process (2 branches)
+ at state 1
+-- tape at 3, [['a', 'b', 'c', 'd']]
+ at state 2
+-- tape at 3, [['a', 'b', 'c']]
process (0 branches)
>>> sorted(it.result())
[Branch(accept=False, state=1, output='abcd'),
```

(continues on next page)

(continued from previous page)

```
Branch(accept=True, state=2, output='abc')]
```

See also

FiniteStateMachine.process(), *Automaton.process()*, *Transducer.process()*, *FiniteStateMachine.iter_process()*, *FiniteStateMachine.__call__()*, *next()*.

class Current

Bases: dict

This class stores the branches which have to be processed during iteration and provides a nicer formatting of them.

This class is derived from dict. It is returned by the next-function during iteration.

EXAMPLES:

In the following example you can see the dict directly and then the nicer output provided by this class:

```
sage: from sage.combinat.finite_state_machine import FSMProcessIterator
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: it = FSMProcessIterator(inverter, input_tape=[0, 1])
sage: for current in it:
....:     print(dict(current))
....:     print(current)
{((1, 0),): {'A': Branch(tape_cache=tape at 1, outputs=[[1]])}}
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
{((2, 0),): {'A': Branch(tape_cache=tape at 2, outputs=[[1, 0]])}}
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
{}
process (0 branches)
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMProcessIterator
>>> inverter = Transducer({'A': [('A', Integer(0), Integer(1)), ('A',␣
↪Integer(1), Integer(0))]},
...     initial_states=['A'], final_states=['A'])
>>> it = FSMProcessIterator(inverter, input_tape=[Integer(0), Integer(1)])
>>> for current in it:
...     print(dict(current))
...     print(current)
{((1, 0),): {'A': Branch(tape_cache=tape at 1, outputs=[[1]])}}
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
{((2, 0),): {'A': Branch(tape_cache=tape at 2, outputs=[[1, 0]])}}
process (1 branch)
```

(continues on next page)

(continued from previous page)

```
+ at state 'A'
+-- tape at 2, [[1, 0]]
{}
process (0 branches)
```

class FinishedBranch(*accept*, *state*, *output*)

Bases: `tuple`

A `named tuple` representing the attributes of a branch, once it is fully processed.

accept

Alias for field number 0

output

Alias for field number 2

state

Alias for field number 1

next()

Makes one step in processing the input tape.

OUTPUT:

It returns the current status of the iterator (see below). A `StopIteration` exception is thrown when there is/was nothing to do (i.e. all branches ended with previous call of `next()`).

The current status is a dictionary (encapsulated into an instance of `Current`). The keys are positions on the tape. The value corresponding to such a position is again a dictionary, where each entry represents a branch of the process. This dictionary maps the current state of a branch to a pair consisting of a tape cache and a list of output words, which were written during reaching this current state.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMProcessIterator
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: it = FSMProcessIterator(inverter, input_tape=[0, 1])
sage: next(it)
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
sage: next(it)
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
sage: next(it)
process (0 branches)
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMProcessIterator
```

(continues on next page)

(continued from previous page)

```
>>> inverter = Transducer({'A': [('A', Integer(0), Integer(1)), ('A',␣
↪Integer(1), Integer(0))]},
...     initial_states=['A'], final_states=['A'])
>>> it = FSMProcessIterator(inverter, input_tape=[Integer(0), Integer(1)])
>>> next(it)
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
>>> next(it)
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
>>> next(it)
process (0 branches)
>>> next(it)
Traceback (most recent call last):
...
StopIteration
```

See also

FiniteStateMachine.process(), *Automaton.process()*, *Transducer.process()*, *FiniteStateMachine.iter_process()*, *FiniteStateMachine.__call__()*, *FSMProcessIterator*.

preview_word(*track_number=None*, *length=1*, *return_word=False*)

Read a word from the input tape.

INPUT:

- `track_number` – integer (default: `None`); if `None`, then a tuple of words (one from each track) is returned
- `length` – (default: `1`) the length of the word(s)
- `return_word` – boolean (default: `False`); if set, then a word is returned, otherwise a single letter (in which case `length` has to be `1`)

OUTPUT: a single letter or a word

An exception `StopIteration` is thrown if the tape (at least one track) has reached its end.

Typically, this method is called from a hook-function of a state.

EXAMPLES:

```
sage: inverter = Transducer({'A': [('A', 0, 'one'),
....:                              ('A', 1, 'zero')]},
....:     initial_states=['A'], final_states=['A'])
sage: def state_hook(process, state, output):
....:     print("We are now in state %s." % (state.label(),))
....:     try:
....:         w = process.preview_word()
....:     except RuntimeError:
....:         raise StopIteration
```

(continues on next page)

(continued from previous page)

```
....:     print("Next on the tape is a %s." % (w,))
sage: inverter.state('A').hook = state_hook
sage: it = inverter.iter_process(
....:     input_tape=[0, 1, 1],
....:     check_epsilon_transitions=False)
sage: for _ in it:
....:     pass
We are now in state A.
Next on the tape is a 0.
We are now in state A.
Next on the tape is a 1.
We are now in state A.
Next on the tape is a 1.
We are now in state A.
sage: it.result()
[Branch(accept=True, state='A', output=['one', 'zero', 'zero'])]
```

```
>>> from sage.all import *
>>> inverter = Transducer({'A': [('A', Integer(0), 'one'),
...                              ('A', Integer(1), 'zero')]},
...     initial_states=['A'], final_states=['A'])
>>> def state_hook(process, state, output):
...     print("We are now in state %s." % (state.label(),))
...     try:
...         w = process.preview_word()
...     except RuntimeError:
...         raise StopIteration
...     print("Next on the tape is a %s." % (w,))
>>> inverter.state('A').hook = state_hook
>>> it = inverter.iter_process(
...     input_tape=[Integer(0), Integer(1), Integer(1)],
...     check_epsilon_transitions=False)
>>> for _ in it:
...     pass
We are now in state A.
Next on the tape is a 0.
We are now in state A.
Next on the tape is a 1.
We are now in state A.
Next on the tape is a 1.
We are now in state A.
>>> it.result()
[Branch(accept=True, state='A', output=['one', 'zero', 'zero'])]
```

result (*format_output=None*)

Return the already finished branches during process.

INPUT:

- `format_output` – a function converting the output from list form to something more readable (default: output the list directly).

OUTPUT:

A list of triples (accepted, state, output).

See also the parameter format_output of *FSMProcessIterator*.

EXAMPLES:

```
sage: inverter = Transducer({'A': [('A', 0, 'one'), ('A', 1, 'zero')]},
....:     initial_states=['A'], final_states=['A'])
sage: it = inverter.iter_process(input_tape=[0, 1, 1])
sage: for _ in it:
....:     pass
sage: it.result()
[Branch(accept=True, state='A', output=['one', 'zero', 'zero'])]
sage: it.result(lambda L: ', '.join(L))
[(True, 'A', 'one, zero, zero')]
```

```
>>> from sage.all import *
>>> inverter = Transducer({'A': [('A', Integer(0), 'one'), ('A', Integer(1),
↪'zero')]},
...     initial_states=['A'], final_states=['A'])
>>> it = inverter.iter_process(input_tape=[Integer(0), Integer(1),␣
↪Integer(1)])
>>> for _ in it:
...     pass
>>> it.result()
[Branch(accept=True, state='A', output=['one', 'zero', 'zero'])]
>>> it.result(lambda L: ', '.join(L))
[(True, 'A', 'one, zero, zero')]
```

Using both the parameter format_output of *FSMProcessIterator* and the parameter format_output of *result()* leads to concatenation of the two functions:

```
sage: it = inverter.iter_process(input_tape=[0, 1, 1],
....:                            format_output=lambda L: ', '.join(L))
sage: for _ in it:
....:     pass
sage: it.result()
[Branch(accept=True, state='A', output='one, zero, zero')]
sage: it.result(lambda L: ', '.join(L))
[(True, 'A', 'o, n, e, ,,  , z, e, r, o, ,,  , z, e, r, o')]
```

```
>>> from sage.all import *
>>> it = inverter.iter_process(input_tape=[Integer(0), Integer(1),␣
↪Integer(1)],
...                            format_output=lambda L: ', '.join(L))
>>> for _ in it:
...     pass
>>> it.result()
[Branch(accept=True, state='A', output='one, zero, zero')]
>>> it.result(lambda L: ', '.join(L))
[(True, 'A', 'o, n, e, ,,  , z, e, r, o, ,,  , z, e, r, o')]
```

class sage.combinat.finite_state_machine.**FSMState**(*label*, *word_out=None*, *is_initial=False*, *is_final=False*, *final_word_out=None*, *initial_probability=None*, *hook=None*, *color=None*, *allow_label_None=False*)

Bases: SageObject

Class for a state of a finite state machine.

INPUT:

- label – the label of the state
- word_out – (default: None) a word that is written when the state is reached
- is_initial – (default: False)
- is_final – (default: False)
- final_word_out – (default: None) a word that is written when the state is reached as the last state of some input; only for final states.
- initial_probability – (default: None) the probability of starting in this state if it is a state of a Markov chain
- hook – (default: None) a function which is called when the state is reached during processing input. It takes two input parameters: the first is the current state (to allow using the same hook for several states), the second is the current process iterator object (to have full access to everything; e.g. the next letter from the input tape can be read in). It can output the next transition, i.e. the transition to take next. If it returns None the process iterator chooses. Moreover, this function can raise a StopIteration exception to stop processing of a finite state machine the input immediately. See also the example below.
- color – (default: None) in order to distinguish states, they can be given an arbitrary "color" (an arbitrary object). This is used in *FiniteStateMachine.equivalence_classes()*: states of different colors are never considered to be equivalent. Note that *Automaton.determinisation()* requires that color is hashable.
- allow_label_None – boolean (default: False); if True allows also None as label. Note that a state with label None is used in *FSMProcessIterator*.

OUTPUT: a state of a finite state machine

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('state 1', word_out=0, is_initial=True)
sage: A
'state 1'
sage: A.label()
'state 1'
sage: B = FSMState('state 2')
sage: A == B
False
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('state 1', word_out=Integer(0), is_initial=True)
>>> A
'state 1'
>>> A.label()
'state 1'
>>> B = FSMState('state 2')
>>> A == B
False
```

We can also define a final output word of a final state which is used if the input of a transducer leads to this state. Such final output words are used in subsequential transducers.

```
sage: C = FSMState('state 3', is_final=True, final_word_out='end')
sage: C.final_word_out
['end']
```

```
>>> from sage.all import *
>>> C = FSMState('state 3', is_final=True, final_word_out='end')
>>> C.final_word_out
['end']
```

The final output word can be a single letter, `None` or a list of letters:

```
sage: A = FSMState('A')
sage: A.is_final = True
sage: A.final_word_out = 2
sage: A.final_word_out
[2]
sage: A.final_word_out = [2, 3]
sage: A.final_word_out
[2, 3]
```

```
>>> from sage.all import *
>>> A = FSMState('A')
>>> A.is_final = True
>>> A.final_word_out = Integer(2)
>>> A.final_word_out
[2]
>>> A.final_word_out = [Integer(2), Integer(3)]
>>> A.final_word_out
[2, 3]
```

Only final states can have a final output word which is not `None`:

```
sage: B = FSMState('B')
sage: B.final_word_out is None
True
sage: B.final_word_out = 2
Traceback (most recent call last):
...
ValueError: Only final states can have a final output word,
but state B is not final.
```

```
>>> from sage.all import *
>>> B = FSMState('B')
>>> B.final_word_out is None
True
>>> B.final_word_out = Integer(2)
Traceback (most recent call last):
...
ValueError: Only final states can have a final output word,
but state B is not final.
```

Setting the `final_word_out` of a final state to `None` is the same as setting it to `[]` and is also the default for a final state:

```
sage: C = FSMState('C', is_final=True)
sage: C.final_word_out
[]
sage: C.final_word_out = None
sage: C.final_word_out
[]
sage: C.final_word_out = []
sage: C.final_word_out
[]
```

```
>>> from sage.all import *
>>> C = FSMState('C', is_final=True)
>>> C.final_word_out
[]
>>> C.final_word_out = None
>>> C.final_word_out
[]
>>> C.final_word_out = []
>>> C.final_word_out
[]
```

It is not allowed to use `None` as a label:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: FSMState(None)
Traceback (most recent call last):
...
ValueError: Label None reserved for a special state,
choose another label.
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> FSMState(None)
Traceback (most recent call last):
...
ValueError: Label None reserved for a special state,
choose another label.
```

This can be overridden by:

```
sage: FSMState(None, allow_label_None=True)
None
```

```
>>> from sage.all import *
>>> FSMState(None, allow_label_None=True)
None
```

Note that *`Automaton.determinisation()`* requires that `color` is hashable:

```
sage: A = Automaton([[0, 0, 0]], initial_states=[0])
sage: A.state(0).color = []
sage: A.determinisation()
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
sage: A.state(0).color = ()
sage: A.determinisation()
Automaton with 1 state
```

```
>>> from sage.all import *
>>> A = Automaton([[Integer(0), Integer(0), Integer(0)]], initial_
↪states=[Integer(0)])
>>> A.state(Integer(0)).color = []
>>> A.determinisation()
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
>>> A.state(Integer(0)).color = ()
>>> A.determinisation()
Automaton with 1 state
```

We can use a hook function of a state to stop processing. This is done by raising a `StopIteration` exception. The following code demonstrates this:

```
sage: T = Transducer([(0, 1, 9, 'a'), (1, 2, 9, 'b'),
....:                  (2, 3, 9, 'c'), (3, 4, 9, 'd')],
....:                 initial_states=[0],
....:                 final_states=[4],
....:                 input_alphabet=[9])
sage: def stop(process, state, output):
....:     raise StopIteration()
sage: T.state(3).hook = stop
sage: T.process([9, 9, 9, 9])
(False, 3, ['a', 'b', 'c'])
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(9), 'a'), (Integer(1),␣
↪Integer(2), Integer(9), 'b'),
...                  (Integer(2), Integer(3), Integer(9), 'c'), (Integer(3),␣
↪Integer(4), Integer(9), 'd')],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(4)],
...                 input_alphabet=[Integer(9)])
>>> def stop(process, state, output):
...     raise StopIteration()
>>> T.state(Integer(3)).hook = stop
>>> T.process([Integer(9), Integer(9), Integer(9), Integer(9)])
(False, 3, ['a', 'b', 'c'])
```

copy()

Return a (shallow) copy of the state.

OUTPUT: a new state

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: A.is_initial = True
sage: A.is_final = True
sage: A.final_word_out = [1]
sage: A.color = 'green'
sage: A.initial_probability = 1/2
sage: B = copy(A)
sage: B.fully_equal(A)
True
sage: A.label() is B.label()
True
sage: A.is_initial is B.is_initial
True
sage: A.is_final is B.is_final
True
sage: A.final_word_out is B.final_word_out
True
sage: A.color is B.color
True
sage: A.initial_probability is B.initial_probability
True
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A')
>>> A.is_initial = True
>>> A.is_final = True
>>> A.final_word_out = [Integer(1)]
>>> A.color = 'green'
>>> A.initial_probability = Integer(1)/Integer(2)
>>> B = copy(A)
>>> B.fully_equal(A)
True
>>> A.label() is B.label()
True
>>> A.is_initial is B.is_initial
True
>>> A.is_final is B.is_final
True
>>> A.final_word_out is B.final_word_out
True
>>> A.color is B.color
True
>>> A.initial_probability is B.initial_probability
True
```

deepcopy (*memo=None*)

Return a deep copy of the state.

INPUT:

- memo – (default: None) a dictionary storing already processed elements

OUTPUT: a new state

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState((1, 3), color=[1, 2],
....:              is_final=True, final_word_out=3,
....:              initial_probability=1/3)
sage: B = deepcopy(A)
sage: B
(1, 3)
sage: B.label() == A.label()
True
sage: B.label is A.label
False
sage: B.color == A.color
True
sage: B.color is A.color
False
sage: B.is_final == A.is_final
True
sage: B.is_final is A.is_final
True
sage: B.final_word_out == A.final_word_out
True
sage: B.final_word_out is A.final_word_out
False
sage: B.initial_probability == A.initial_probability
True
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState((Integer(1), Integer(3)), color=[Integer(1), Integer(2)],
...              is_final=True, final_word_out=Integer(3),
...              initial_probability=Integer(1)/Integer(3))
>>> B = deepcopy(A)
>>> B
(1, 3)
>>> B.label() == A.label()
True
>>> B.label is A.label
False
>>> B.color == A.color
True
>>> B.color is A.color
False
>>> B.is_final == A.is_final
True
>>> B.is_final is A.is_final
True
>>> B.final_word_out == A.final_word_out
True
```

(continues on next page)

(continued from previous page)

```
>>> B.final_word_out is A.final_word_out
False
>>> B.initial_probability == A.initial_probability
True
```

property final_word_out

The final output word of a final state which is written if the state is reached as the last state of the input of the finite state machine. For a non-final state, the value is None.

final_word_out can be a single letter, a list or None, but for a final-state, it is always saved as a list.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_final=True, final_word_out=2)
sage: A.final_word_out
[2]
sage: A.final_word_out = 3
sage: A.final_word_out
[3]
sage: A.final_word_out = [3, 4]
sage: A.final_word_out
[3, 4]
sage: A.final_word_out = None
sage: A.final_word_out
[]
sage: B = FSMState('B')
sage: B.final_word_out is None
True
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A', is_final=True, final_word_out=Integer(2))
>>> A.final_word_out
[2]
>>> A.final_word_out = Integer(3)
>>> A.final_word_out
[3]
>>> A.final_word_out = [Integer(3), Integer(4)]
>>> A.final_word_out
[3, 4]
>>> A.final_word_out = None
>>> A.final_word_out
[]
>>> B = FSMState('B')
>>> B.final_word_out is None
True
```

A non-final state cannot have a final output word:

```
sage: B.final_word_out = [3, 4]
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: Only final states can have a final
output word, but state B is not final.
```

```
>>> from sage.all import *
>>> B.final_word_out = [Integer(3), Integer(4)]
Traceback (most recent call last):
...
ValueError: Only final states can have a final
output word, but state B is not final.
```

fully_equal(*other*, *compare_color=True*)

Check whether two states are fully equal, i.e., including all attributes except `hook`.

INPUT:

- `self` – a state
- `other` – a state
- `compare_color` – boolean (default: `True`); if `True` colors are compared as well, otherwise not

OUTPUT: boolean

Note that usual comparison by == does only compare the labels.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: B = FSMState('A', is_initial=True)
sage: A.fully_equal(B)
False
sage: A == B
True
sage: A.is_initial = True; A.color = 'green'
sage: A.fully_equal(B)
False
sage: A.fully_equal(B, compare_color=False)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A')
>>> B = FSMState('A', is_initial=True)
>>> A.fully_equal(B)
False
>>> A == B
True
>>> A.is_initial = True; A.color = 'green'
>>> A.fully_equal(B)
False
>>> A.fully_equal(B, compare_color=False)
True
```

initial_probability = None

property is_final

Describe whether the state is final or not.

`True` if the state is final and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_final=True, final_word_out=3)
sage: A.is_final
True
sage: A.is_final = False
Traceback (most recent call last):
...
ValueError: State A cannot be non-final, because it has a
final output word. Only final states can have a final output
word.
sage: A.final_word_out = None
sage: A.is_final = False
sage: A.is_final
False
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A', is_final=True, final_word_out=Integer(3))
>>> A.is_final
True
>>> A.is_final = False
Traceback (most recent call last):
...
ValueError: State A cannot be non-final, because it has a
final output word. Only final states can have a final output
word.
>>> A.final_word_out = None
>>> A.is_final = False
>>> A.is_final
False
```

is_initial = False

label()

Return the label of the state.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('state')
sage: A.label()
'state'
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('state')
>>> A.label()
'state'
```

relabeled(*label*, *memo=None*)

Return a deep copy of the state with a new label.

INPUT:

- `label` – the label of new state
- `memo` – (default: `None`) a dictionary storing already processed elements

OUTPUT: a new state

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: A.relabeled('B')
'B'
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A')
>>> A.relabeled('B')
'B'
```

class sage.combinat.finite_state_machine.**FSMTransition**(*from_state*, *to_state*, *word_in=None*, *word_out=None*, *hook=None*)

Bases: `SageObject`

Class for a transition of a finite state machine.

INPUT:

- `from_state` – state from which transition starts
- `to_state` – state in which transition ends
- `word_in` – the input word of the transitions (when the finite state machine is used as automaton)
- `word_out` – the output word of the transitions (when the finite state machine is used as transducer)

OUTPUT: a transition of a finite state machine

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState, FSMTransition
sage: A = FSMState('A')
sage: B = FSMState('B')
sage: S = FSMTransition(A, B, 0, 1)
sage: T = FSMTransition('A', 'B', 0, 1)
sage: T == S
True
sage: U = FSMTransition('A', 'B', 0)
sage: U == T
False
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState, FSMTransition
>>> A = FSMState('A')
>>> B = FSMState('B')
```

(continues on next page)

(continued from previous page)

```
>>> S = FSMTransition(A, B, Integer(0), Integer(1))
>>> T = FSMTransition('A', 'B', Integer(0), Integer(1))
>>> T == S
True
>>> U = FSMTransition('A', 'B', Integer(0))
>>> U == T
False
```

copy()

Return a (shallow) copy of the transition.

OUTPUT: a new transition

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t = FSMTransition('A', 'B', 0)
sage: copy(t)
Transition from 'A' to 'B': 0|-
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMTransition
>>> t = FSMTransition('A', 'B', Integer(0))
>>> copy(t)
Transition from 'A' to 'B': 0|-
```

deepcopy(*memo=None*)

Return a deep copy of the transition.

INPUT:

- `memo` – (default: `None`) a dictionary storing already processed elements

OUTPUT: a new transition

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t = FSMTransition('A', 'B', 0)
sage: deepcopy(t)
Transition from 'A' to 'B': 0|-
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMTransition
>>> t = FSMTransition('A', 'B', Integer(0))
>>> deepcopy(t)
Transition from 'A' to 'B': 0|-
```

from_state = None

State from which the transition starts. Read-only.

to_state = None

State in which the transition ends. Read-only.

word_in = None

Input word of the transition. Read-only.

word_out = None

Output word of the transition. Read-only.

sage.combinat.finite_state_machine.**FSMWordSymbol**(*word*)

Return a string of `word`. It may returns the symbol of the empty word `FSMEmptyWordSymbol`.

INPUT:

- `word` – the input word

OUTPUT: string of `word`

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMWordSymbol
sage: FSMWordSymbol([0, 1, 1])
'0,1,1'
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMWordSymbol
>>> FSMWordSymbol([Integer(0), Integer(1), Integer(1)])
'0,1,1'
```

class sage.combinat.finite_state_machine.**FiniteStateMachine**(*data=None*, *initial_states=None*, *final_states=None*, *input_alphabet=None*, *output_alphabet=None*, *determine_alphabets=None*, *with_final_word_out=None*, *store_states_dict=True*, *on_duplicate_transition=None*)

Bases: `SageObject`

Class for a finite state machine.

A finite state machine is a finite set of states connected by transitions.

INPUT:

- `data` – can be any of the following:
  1. a dictionary of dictionaries (of transitions),
  2. a dictionary of lists (of states or transitions),
  3. a list (of transitions),
  4. a function (transition function),
  5. an other instance of a finite state machine.
- `initial_states` and `final_states` – the initial and final states of this machine
- `input_alphabet` and `output_alphabet` – the input and output alphabets of this machine
- `determine_alphabets` – if `True`, then the function `determine_alphabets()` is called after `data` was read and processed, if `False`, then not. If it is `None`, then it is decided during the construction of the finite state machine whether `determine_alphabets()` should be called.

- `with_final_word_out` – if given (not `None`), then the function `with_final_word_out()` (more precisely, its inplace pendant `construct_final_word_out()`) is called with input `letters=with_final_word_out` at the end of the creation process.
- `store_states_dict` – if `True`, then additionally the states are stored in an internal dictionary for speed up
- `on_duplicate_transition` – a function which is called when a transition is inserted into `self` which already existed (same `from_state`, same `to_state`, same `word_in`, same `word_out`).

  This function is assumed to take two arguments, the first being the already existing transition, the second being the new transition (as an `FSMTransition`). The function must return the (possibly modified) original transition.

  By default, we have `on_duplicate_transition=None`, which is interpreted as `on_duplicate_transition=duplicate_transition_ignore`, where `duplicate_transition_ignore` is a predefined function ignoring the occurrence. Other such predefined functions are `duplicate_transition_raise_error` and `duplicate_transition_add_input`.

OUTPUT: a finite state machine

The object creation of `Automaton` and `Transducer` is the same as the one described here (i.e. just replace the word `FiniteStateMachine` by `Automaton` or `Transducer`).

Each transition of an automaton has an input label. Automata can, for example, be determinised (see `Automaton.determinisation()`) and minimized (see `Automaton.minimization()`). Each transition of a transducer has an input and an output label. Transducers can, for example, be simplified (see `Transducer.simplification()`).

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState, FSMTransition
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState, FSMTransition
```

See documentation for more examples.

We illustrate the different input formats:

1. The input-data can be a dictionary of dictionaries, where

   - the keys of the outer dictionary are state-labels (from-states of transitions),
   - the keys of the inner dictionaries are state-labels (to-states of transitions),
   - the values of the inner dictionaries specify the transition more precisely.

   The easiest is to use a tuple consisting of an input and an output word:

   ```
   sage: FiniteStateMachine({'a':{'b':(0, 1), 'c':(1, 1)}})
   Finite state machine with 3 states
   ```

   ```
   >>> from sage.all import *
   >>> FiniteStateMachine({'a':{'b':(Integer(0), Integer(1)), 'c':(Integer(1),␣
   ↪Integer(1))}})
   Finite state machine with 3 states
   ```

   Instead of the tuple anything iterable (e.g. a list) can be used as well.

   If you want to use the arguments of `FSMTransition` directly, you can use a dictionary:

```
sage: FiniteStateMachine({'a':{'b':{'word_in':0, 'word_out':1},
....:                          'c':{'word_in':1, 'word_out':1}}})
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine({'a':{'b':{'word_in':Integer(0), 'word_out':Integer(1)}
↪,
...                          'c':{'word_in':Integer(1), 'word_out':Integer(1)}
↪}})
Finite state machine with 3 states
```

In the case you already have instances of *FSMTransition*, it is possible to use them directly:

```
sage: FiniteStateMachine({'a':{'b':FSMTransition('a', 'b', 0, 1),
....:                          'c':FSMTransition('a', 'c', 1, 1)}})
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine({'a':{'b':FSMTransition('a', 'b', Integer(0),␣
↪Integer(1)),
...                          'c':FSMTransition('a', 'c', Integer(1),␣
↪Integer(1))}})
Finite state machine with 3 states
```

2. The input-data can be a dictionary of lists, where the keys are states or label of states.

   The list-elements can be states:

```
sage: a = FSMState('a')
sage: b = FSMState('b')
sage: c = FSMState('c')
sage: FiniteStateMachine({a:[b, c]})
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> a = FSMState('a')
>>> b = FSMState('b')
>>> c = FSMState('c')
>>> FiniteStateMachine({a:[b, c]})
Finite state machine with 3 states
```

   Or the list-elements can simply be labels of states:

```
sage: FiniteStateMachine({'a':['b', 'c']})
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine({'a':['b', 'c']})
Finite state machine with 3 states
```

   The list-elements can also be transitions:

```
sage: FiniteStateMachine({'a':[FSMTransition('a', 'b', 0, 1),
....:                          FSMTransition('a', 'c', 1, 1)]})
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine({'a':[FSMTransition('a', 'b', Integer(0), Integer(1)),
...                          FSMTransition('a', 'c', Integer(1), Integer(1))]}
↪)
Finite state machine with 3 states
```

Or they can be tuples of a label, an input word and an output word specifying a transition:

```
sage: FiniteStateMachine({'a':[('b', 0, 1), ('c', 1, 1)]})
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine({'a':[('b', Integer(0), Integer(1)), ('c', Integer(1),␣
↪Integer(1))]})
Finite state machine with 3 states
```

3. The input-data can be a list, where its elements specify transitions:

```
sage: FiniteStateMachine([FSMTransition('a', 'b', 0, 1),
....:                     FSMTransition('a', 'c', 1, 1)])
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine([FSMTransition('a', 'b', Integer(0), Integer(1)),
...                     FSMTransition('a', 'c', Integer(1), Integer(1))])
Finite state machine with 3 states
```

It is possible to skip `FSMTransition` in the example above:

```
sage: FiniteStateMachine([('a', 'b', 0, 1), ('a', 'c', 1, 1)])
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine([('a', 'b', Integer(0), Integer(1)), ('a', 'c',␣
↪Integer(1), Integer(1))])
Finite state machine with 3 states
```

The parameters of the transition are given in tuples. Anyhow, anything iterable (e.g. a list) is possible.

You can also name the parameters of the transition. For this purpose you take a dictionary:

```
sage: FiniteStateMachine([{'from_state':'a', 'to_state':'b',
....:                      'word_in':0, 'word_out':1},
....:                     {'from_state':'a', 'to_state':'c',
....:                      'word_in':1, 'word_out':1}])
Finite state machine with 3 states
```

```
>>> from sage.all import *
>>> FiniteStateMachine([{'from_state':'a', 'to_state':'b',
```

(continues on next page)

(continued from previous page)

```
...                          'word_in':Integer(0), 'word_out':Integer(1)},
...                         {'from_state':'a', 'to_state':'c',
...                          'word_in':Integer(1), 'word_out':Integer(1)}])
Finite state machine with 3 states
```

Other arguments, which `FSMTransition` accepts, can be added, too.

4. The input-data can also be function acting as transition function:

   This function has two input arguments:

   1. a label of a state (from which the transition starts),
   2. a letter of the (input-)alphabet (as input-label of the transition).

   It returns a tuple with the following entries:

   1. a label of a state (to which state the transition goes),
   2. a letter of or a word over the (output-)alphabet (as output-label of the transition).

   It may also output a list of such tuples if several transitions from the from-state and the input letter exist (this means that the finite state machine is non-deterministic).

   If the transition does not exist, the function should raise a `LookupError` or return an empty list.

   When constructing a finite state machine in this way, some initial states and an input alphabet have to be specified.

```
sage: def f(state_from, read):
....:     if int(state_from) + read <= 2:
....:         state_to = 2*int(state_from)+read
....:         write = 0
....:     else:
....:         state_to = 2*int(state_from) + read - 5
....:         write = 1
....:     return (str(state_to), write)
sage: F = FiniteStateMachine(f, input_alphabet=[0, 1],
....:                        initial_states=['0'],
....:                        final_states=['0'])
sage: F([1, 0, 1])
(True, '0', [0, 0, 1])
```

```
>>> from sage.all import *
>>> def f(state_from, read):
...     if int(state_from) + read <= Integer(2):
...         state_to = Integer(2)*int(state_from)+read
...         write = Integer(0)
...     else:
...         state_to = Integer(2)*int(state_from) + read - Integer(5)
...         write = Integer(1)
...     return (str(state_to), write)
>>> F = FiniteStateMachine(f, input_alphabet=[Integer(0), Integer(1)],
...                        initial_states=['0'],
...                        final_states=['0'])
>>> F([Integer(1), Integer(0), Integer(1)])
(True, '0', [0, 0, 1])
```

5. The input-data can be an other instance of a finite state machine:

```
sage: F = FiniteStateMachine()
sage: G = Transducer(F)
sage: G == F
True
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine()
>>> G = Transducer(F)
>>> G == F
True
```

The other parameters cannot be specified in that case. If you want to change these, use the attributes *FSMState.is_initial*, *FSMState.is_final*, *input_alphabet*, *output_alphabet*, *on_duplicate_transition* and methods *determine_alphabets()*, *construct_final_word_out()* on the new machine, respectively.

The following examples demonstrate the use of `on_duplicate_transition`:

```
sage: F = FiniteStateMachine([['a', 'a', 1/2], ['a', 'a', 1/2]])
sage: F.transitions()
[Transition from 'a' to 'a': 1/2|-]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([['a', 'a', Integer(1)/Integer(2)], ['a', 'a',␣
↪Integer(1)/Integer(2)]])
>>> F.transitions()
[Transition from 'a' to 'a': 1/2|-]
```

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_raise_
↪error
sage: F1 = FiniteStateMachine([['a', 'a', 1/2], ['a', 'a', 1/2]],
....:                         on_duplicate_transition=duplicate_transition_raise_
↪error)
Traceback (most recent call last):
...
ValueError: Attempting to re-insert transition Transition from 'a' to 'a': 1/2|-
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_raise_
↪error
>>> F1 = FiniteStateMachine([['a', 'a', Integer(1)/Integer(2)], ['a', 'a',␣
↪Integer(1)/Integer(2)]],
...                         on_duplicate_transition=duplicate_transition_raise_
↪error)
Traceback (most recent call last):
...
ValueError: Attempting to re-insert transition Transition from 'a' to 'a': 1/2|-
```

Use `duplicate_transition_add_input` to emulate a Markov chain, the input labels are considered as transition probabilities:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
sage: F = FiniteStateMachine([['a', 'a', 1/2], ['a', 'a', 1/2]],
....:                        on_duplicate_transition=duplicate_transition_add_
↪input)
sage: F.transitions()
[Transition from 'a' to 'a': 1|-]
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_add_input
>>> F = FiniteStateMachine([['a', 'a', Integer(1)/Integer(2)], ['a', 'a',␣
↪Integer(1)/Integer(2)]],
...                        on_duplicate_transition=duplicate_transition_add_input)
>>> F.transitions()
[Transition from 'a' to 'a': 1|-]
```

Use `with_final_word_out` to construct final output:

```
sage: T = Transducer([(0, 1, 0, 0), (1, 0, 0, 0)],
....:                initial_states=[0],
....:                final_states=[0],
....:                with_final_word_out=0)
sage: for s in T.iter_final_states():
....:     print("{} {}".format(s, s.final_word_out))
0 []
1 [0]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(0), Integer(0)), (Integer(1),
↪ Integer(0), Integer(0), Integer(0))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(0)],
...                with_final_word_out=Integer(0))
>>> for s in T.iter_final_states():
...     print("{} {}".format(s, s.final_word_out))
0 []
1 [0]
```

`__call__`(**args, **kwargs*)

Call either method *`composition()`* or *`process()`* (with `full_output=False`). If the input is not finite (`is_finite` of input is `False`), then *`iter_process()`* (with `iterator_type='simple'`) is called. Moreover, the flag `automatic_output_type` is set (unless `format_output` is specified). See the documentation of these functions for possible parameters.

EXAMPLES:

The following code performs a *`composition()`*:

```
sage: F = Transducer([('A', 'B', 1, 0), ('B', 'B', 1, 1),
....:                 ('B', 'B', 0, 0)],
....:                initial_states=['A'], final_states=['B'])
sage: G = Transducer([(1, 1, 0, 0), (1, 2, 1, 0),
....:                 (2, 2, 0, 1), (2, 1, 1, 1)],
```

(continues on next page)

(continued from previous page)

```
....:                   initial_states=[1], final_states=[1])
sage: H = G(F)
sage: H.states()
[('A', 1), ('B', 1), ('B', 2)]
```

```
>>> from sage.all import *
>>> F = Transducer([('A', 'B', Integer(1), Integer(0)), ('B', 'B', Integer(1),
↪ Integer(1)),
...                  ('B', 'B', Integer(0), Integer(0))],
...                 initial_states=['A'], final_states=['B'])
>>> G = Transducer([(Integer(1), Integer(1), Integer(0), Integer(0)),␣
↪(Integer(1), Integer(2), Integer(1), Integer(0)),
...                  (Integer(2), Integer(2), Integer(0), Integer(1)),␣
↪(Integer(2), Integer(1), Integer(1), Integer(1))],
...                 initial_states=[Integer(1)], final_states=[Integer(1)])
>>> H = G(F)
>>> H.states()
[('A', 1), ('B', 1), ('B', 2)]
```

An automaton or transducer can also act on an input (a list or other iterable of letters):

```
sage: binary_inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:                              initial_states=['A'], final_states=['A'])
sage: binary_inverter([0, 1, 0, 0, 1, 1])
[1, 0, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> binary_inverter = Transducer({'A': [('A', Integer(0), Integer(1)), ('A',␣
↪Integer(1), Integer(0))]},
...                              initial_states=['A'], final_states=['A'])
>>> binary_inverter([Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1)])
[1, 0, 1, 1, 0, 0]
```

We can also let them act on *words*:

```
sage: # needs sage.combinat
sage: W = Words([0, 1]); W
Finite and infinite words over {0, 1}
sage: binary_inverter(W([0, 1, 1, 0, 1, 1]))
word: 100100
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> W = Words([Integer(0), Integer(1)]); W
Finite and infinite words over {0, 1}
>>> binary_inverter(W([Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)]))
word: 100100
```

Infinite words work as well:

```
sage: # needs sage.combinat
sage: words.FibonacciWord()
word: 0100101001001010010100100101001001010010...
sage: binary_inverter(words.FibonacciWord())
word: 1011010110110101101011011010110110101101...
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> words.FibonacciWord()
word: 0100101001001010010100100101001001010010...
>>> binary_inverter(words.FibonacciWord())
word: 1011010110110101101011011010110110101101...
```

When only one successful path is found in a non-deterministic transducer, the result of that path is returned.

```
sage: T = Transducer([(0, 1, 0, 1), (0, 2, 0, 2)],
....:                initial_states=[0], final_states=[1])
sage: T.process([0])
[(False, 2, [2]), (True, 1, [1])]
sage: T([0])
[1]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(0), Integer(1)),␣
→(Integer(0), Integer(2), Integer(0), Integer(2))],
...                initial_states=[Integer(0)], final_states=[Integer(1)])
>>> T.process([Integer(0)])
[(False, 2, [2]), (True, 1, [1])]
>>> T([Integer(0)])
[1]
```

See also

composition(), *process()*, *iter_process()*, *Automaton.process()*, *Transducer.process()*.

accessible_components()

Return a new finite state machine with the accessible states of `self` and all transitions between those states.

OUTPUT:

A finite state machine with the accessible states of `self` and all transitions between those states.

A state is accessible if there is a directed path from an initial state to the state. If `self` has no initial states then a copy of the finite state machine `self` is returned.

EXAMPLES:

```
sage: F = Automaton([(0, 0, 0), (0, 1, 1), (1, 1, 0), (1, 0, 1)],
....:               initial_states=[0])
sage: F.accessible_components()
Automaton with 2 states
```

```
>>> from sage.all import *
>>> F = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)), (Integer(1), Integer(1), Integer(0)), (Integer(1),␣
↪Integer(0), Integer(1))],
...               initial_states=[Integer(0)])
>>> F.accessible_components()
Automaton with 2 states
```

```
sage: F = Automaton([(0, 0, 1), (0, 0, 1), (1, 1, 0), (1, 0, 1)],
....:               initial_states=[0])
sage: F.accessible_components()
Automaton with 1 state
```

```
>>> from sage.all import *
>>> F = Automaton([(Integer(0), Integer(0), Integer(1)), (Integer(0),␣
↪Integer(0), Integer(1)), (Integer(1), Integer(1), Integer(0)), (Integer(1),␣
↪Integer(0), Integer(1))],
...               initial_states=[Integer(0)])
>>> F.accessible_components()
Automaton with 1 state
```

See also

coaccessible_components()

add_from_transition_function(*function*, *initial_states=None*, *explore_existing_states=True*)

Construct a finite state machine from a transition function.

INPUT:

- `function` may return a tuple (new_state, output_word) or a list of such tuples
- `initial_states` – if no initial states are given, the already existing initial states of `self` are taken
- `explore_existing_states` – boolean (default: `True`); if `True` (default), then already existing states in `self` (e.g. already given final states) will also be processed if they are reachable from the initial statess

OUTPUT: nothing

EXAMPLES:

```
sage: F = FiniteStateMachine(initial_states=['A'],
....:                        input_alphabet=[0, 1])
sage: def f(state, input):
....:     return [('A', input), ('B', 1-input)]
sage: F.add_from_transition_function(f)
sage: F.transitions()
[Transition from 'A' to 'A': 0|0,
Transition from 'A' to 'B': 0|1,
Transition from 'A' to 'A': 1|1,
Transition from 'A' to 'B': 1|0,
Transition from 'B' to 'A': 0|0,
Transition from 'B' to 'B': 0|1,
```

(continues on next page)

(continued from previous page)

```
Transition from 'B' to 'A': 1|1,
Transition from 'B' to 'B': 1|0]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine(initial_states=['A'],
...                        input_alphabet=[Integer(0), Integer(1)])
>>> def f(state, input):
...     return [('A', input), ('B', Integer(1)-input)]
>>> F.add_from_transition_function(f)
>>> F.transitions()
[Transition from 'A' to 'A': 0|0,
Transition from 'A' to 'B': 0|1,
Transition from 'A' to 'A': 1|1,
Transition from 'A' to 'B': 1|0,
Transition from 'B' to 'A': 0|0,
Transition from 'B' to 'B': 0|1,
Transition from 'B' to 'A': 1|1,
Transition from 'B' to 'B': 1|0]
```

Initial states can also be given as a parameter:

```
sage: F = FiniteStateMachine(input_alphabet=[0,1])
sage: def f(state, input):
....:     return [('A', input), ('B', 1-input)]
sage: F.add_from_transition_function(f,initial_states=['A'])
sage: F.initial_states()
['A']
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine(input_alphabet=[Integer(0),Integer(1)])
>>> def f(state, input):
...     return [('A', input), ('B', Integer(1)-input)]
>>> F.add_from_transition_function(f,initial_states=['A'])
>>> F.initial_states()
['A']
```

Already existing states in the finite state machine (the final states in the example below) are also explored:

```
sage: F = FiniteStateMachine(initial_states=[0],
....:                        final_states=[1],
....:                        input_alphabet=[0])
sage: def transition_function(state, letter):
....:     return 1 - state, []
sage: F.add_from_transition_function(transition_function)
sage: F.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 1 to 0: 0|-]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine(initial_states=[Integer(0)],
...                        final_states=[Integer(1)],
...                        input_alphabet=[Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> def transition_function(state, letter):
...     return Integer(1) - state, []
>>> F.add_from_transition_function(transition_function)
>>> F.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 1 to 0: 0|-]
```

If `explore_existing_states=False`, however, this behavior is turned off, i.e., already existing states are not explored:

```
sage: F = FiniteStateMachine(initial_states=[0],
....:                        final_states=[1],
....:                        input_alphabet=[0])
sage: def transition_function(state, letter):
....:     return 1 - state, []
sage: F.add_from_transition_function(transition_function,
....:                                explore_existing_states=False)
sage: F.transitions()
[Transition from 0 to 1: 0|-]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine(initial_states=[Integer(0)],
...                        final_states=[Integer(1)],
...                        input_alphabet=[Integer(0)])
>>> def transition_function(state, letter):
...     return Integer(1) - state, []
>>> F.add_from_transition_function(transition_function,
...                                explore_existing_states=False)
>>> F.transitions()
[Transition from 0 to 1: 0|-]
```

add_state(*state*)

Add a state to the finite state machine and returns the new state. If the state already exists, that existing state is returned.

INPUT:

- `state` is either an instance of *FSMState* or, otherwise, a label of a state.

OUTPUT: the new or existing state

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: F = FiniteStateMachine()
sage: A = FSMState('A', is_initial=True)
sage: F.add_state(A)
'A'
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> F = FiniteStateMachine()
>>> A = FSMState('A', is_initial=True)
```

(continues on next page)

(continued from previous page)

```
>>> F.add_state(A)
'A'
```

add_states(*states*)

Add several states. See add_state for more information.

INPUT:

- `states` – list of states or iterator over states

OUTPUT: nothing

EXAMPLES:

```
sage: F = FiniteStateMachine()
sage: F.add_states(['A', 'B'])
sage: F.states()
['A', 'B']
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine()
>>> F.add_states(['A', 'B'])
>>> F.states()
['A', 'B']
```

add_transition(**args*, ***kwargs*)

Add a transition to the finite state machine and returns the new transition.

If the transition already exists, the return value of `self.on_duplicate_transition` is returned. See the documentation of `FiniteStateMachine`.

INPUT:

The following forms are all accepted:

```
sage: from sage.combinat.finite_state_machine import FSMState, FSMTransition
sage: A = FSMState('A')
sage: B = FSMState('B')

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition(FSMTransition(A, B, 0, 1))
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition(A, B, 0, 1)
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition(A, B, word_in=0, word_out=1)
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition('A', 'B', {'word_in': 0, 'word_out': 1})
Transition from 'A' to 'B': {'word_in': 0, 'word_out': 1}|-
```

(continues on next page)

(continued from previous page)

```
sage: FSM = FiniteStateMachine()
sage: FSM.add_transition(from_state=A, to_state=B,
....:                    word_in=0, word_out=1)
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition({'from_state': A, 'to_state': B,
....:                    'word_in': 0, 'word_out': 1})
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition((A, B, 0, 1))
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition([A, B, 0, 1])
Transition from 'A' to 'B': 0|1
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState, FSMTransition
>>> A = FSMState('A')
>>> B = FSMState('B')

>>> FSM = FiniteStateMachine()
>>> FSM.add_transition(FSMTransition(A, B, Integer(0), Integer(1)))
Transition from 'A' to 'B': 0|1

>>> FSM = FiniteStateMachine()
>>> FSM.add_transition(A, B, Integer(0), Integer(1))
Transition from 'A' to 'B': 0|1

>>> FSM = FiniteStateMachine()
>>> FSM.add_transition(A, B, word_in=Integer(0), word_out=Integer(1))
Transition from 'A' to 'B': 0|1

>>> FSM = FiniteStateMachine()
>>> FSM.add_transition('A', 'B', {'word_in': Integer(0), 'word_out':
↪Integer(1)})
Transition from 'A' to 'B': {'word_in': 0, 'word_out': 1}|-

>>> FSM = FiniteStateMachine()
>>> FSM.add_transition(from_state=A, to_state=B,
...                    word_in=Integer(0), word_out=Integer(1))
Transition from 'A' to 'B': 0|1

>>> FSM = FiniteStateMachine()
>>> FSM.add_transition({'from_state': A, 'to_state': B,
...                    'word_in': Integer(0), 'word_out': Integer(1)})
Transition from 'A' to 'B': 0|1

>>> FSM = FiniteStateMachine()
>>> FSM.add_transition((A, B, Integer(0), Integer(1)))
```

(continues on next page)

(continued from previous page)

```
Transition from 'A' to 'B': 0|1

>>> FSM = FiniteStateMachine()
>>> FSM.add_transition([A, B, Integer(0), Integer(1)])
Transition from 'A' to 'B': 0|1
```

If the states `A` and `B` are not instances of `FSMState`, then it is assumed that they are labels of states.

OUTPUT:

The new transition.

add_transitions_from_function(*function*, *labels_as_input=True*)

Add one or more transitions if `function(state, state)` says that there are some.

INPUT:

- `function` – a transition function. Given two states `from_state` and `to_state` (or their labels if `label_as_input` is true), this function shall return a tuple `(word_in, word_out)` to add a transition from `from_state` to `to_state` with input and output labels `word_in` and `word_out`, respectively. If no such addition is to be added, the transition function shall return `None`. The transition function may also return a list of such tuples in order to add multiple transitions between the pair of states.
- `label_as_input` – (default: `True`)

OUTPUT: nothing

EXAMPLES:

```
sage: F = FiniteStateMachine()
sage: F.add_states(['A', 'B', 'C'])
sage: def f(state1, state2):
....:     if state1 == 'C':
....:         return None
....:     return (0, 1)
sage: F.add_transitions_from_function(f)
sage: len(F.transitions())
6
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine()
>>> F.add_states(['A', 'B', 'C'])
>>> def f(state1, state2):
...     if state1 == 'C':
...         return None
...     return (Integer(0), Integer(1))
>>> F.add_transitions_from_function(f)
>>> len(F.transitions())
6
```

Multiple transitions are also possible:

```
sage: F = FiniteStateMachine()
sage: F.add_states([0, 1])
sage: def f(state1, state2):
....:     if state1 != state2:
```

(continues on next page)

(continued from previous page)

```
....:           return [(0, 1), (1, 0)]
....:     else:
....:           return None
sage: F.add_transitions_from_function(f)
sage: F.transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|1,
 Transition from 1 to 0: 1|0]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine()
>>> F.add_states([Integer(0), Integer(1)])
>>> def f(state1, state2):
...     if state1 != state2:
...         return [(Integer(0), Integer(1)), (Integer(1), Integer(0))]
...     else:
...         return None
>>> F.add_transitions_from_function(f)
>>> F.transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|1,
 Transition from 1 to 0: 1|0]
```

adjacency_matrix(*input=None*, *entry=None*)

Return the adjacency matrix of the underlying graph.

INPUT:

- `input` – only transitions with input label `input` are respected
- `entry` – the function `entry` takes a transition and the return value is written in the matrix as the entry `(transition.from_state, transition.to_state)`. The default value (`None`) of entry takes the variable `x` to the power of the sum of the output word of the transition.

OUTPUT: a matrix

If any label of a state is not an integer, the finite state machine is relabeled at the beginning. If there are more than one transitions between two states, then the different return values of `entry` are added up.

EXAMPLES:

```
sage: B = FiniteStateMachine({0:{0:(0, 0), 'a':(1, 0)},
....:                         'a':{2:(0, 0), 3:(1, 0)},
....:                         2:{0:(1, 1), 4:(0, 0)},
....:                         3:{'a':(0, 1), 2:(1, 1)},
....:                         4:{4:(1, 1), 3:(0, 1)}},
....:                        initial_states=[0])
sage: B.adjacency_matrix()                                              #␣
↪needs sage.symbolic
[1 1 0 0 0]
[0 0 1 1 0]
[x 0 0 0 1]
```

(continues on next page)

(continued from previous page)

```
[0 x x 0 0]
[0 0 0 x x]
```

```
>>> from sage.all import *
>>> B = FiniteStateMachine({Integer(0):{Integer(0):(Integer(0), Integer(0)),
↪'a':(Integer(1), Integer(0))},
...                          'a':{Integer(2):(Integer(0), Integer(0)),␣
↪Integer(3):(Integer(1), Integer(0))},
...                          Integer(2):{Integer(0):(Integer(1), Integer(1)),␣
↪Integer(4):(Integer(0), Integer(0))},
...                          Integer(3):{'a':(Integer(0), Integer(1)),␣
↪Integer(2):(Integer(1), Integer(1))},
...                          Integer(4):{Integer(4):(Integer(1), Integer(1)),␣
↪Integer(3):(Integer(0), Integer(1))}},
...                         initial_states=[Integer(0)])
>>> B.adjacency_matrix()                                                    #␣
↪needs sage.symbolic
[1 1 0 0 0]
[0 0 1 1 0]
[x 0 0 0 1]
[0 x x 0 0]
[0 0 0 x x]
```

This is equivalent to:

```
sage: matrix(B)                                                               #␣
↪needs sage.symbolic
[1 1 0 0 0]
[0 0 1 1 0]
[x 0 0 0 1]
[0 x x 0 0]
[0 0 0 x x]
```

```
>>> from sage.all import *
>>> matrix(B)                                                                #␣
↪needs sage.symbolic
[1 1 0 0 0]
[0 0 1 1 0]
[x 0 0 0 1]
[0 x x 0 0]
[0 0 0 x x]
```

It is also possible to use other entries in the adjacency matrix:

```
sage: B.adjacency_matrix(entry=(lambda transition: 1))
[1 1 0 0 0]
[0 0 1 1 0]
[1 0 0 0 1]
[0 1 1 0 0]
[0 0 0 1 1]
sage: var('t')                                                                #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
t
sage: B.adjacency_matrix(1, entry=(lambda transition:                          #␣
↪needs sage.symbolic
....:     exp(I*transition.word_out[0]*t)))
[      0       1       0       0       0]
[      0       0       0       1       0]
[e^(I*t)       0       0       0       0]
[      0       0 e^(I*t)       0       0]
[      0       0       0       0 e^(I*t)]
sage: a = Automaton([(0, 1, 0),
....:                (1, 2, 0),
....:                (2, 0, 1),
....:                (2, 1, 0)],
....:               initial_states=[0],
....:               final_states=[0])
sage: a.adjacency_matrix()                                                     #␣
↪needs sage.symbolic
[0 1 0]
[0 0 1]
[1 1 0]
```

```
>>> from sage.all import *
>>> B.adjacency_matrix(entry=(lambda transition: Integer(1)))
[1 1 0 0 0]
[0 0 1 1 0]
[1 0 0 0 1]
[0 1 1 0 0]
[0 0 0 1 1]
>>> var('t')                                                                  #␣
↪needs sage.symbolic
t
>>> B.adjacency_matrix(Integer(1), entry=(lambda transition:                   ␣
↪     # needs sage.symbolic
...     exp(I*transition.word_out[Integer(0)]*t)))
[      0       1       0       0       0]
[      0       0       0       1       0]
[e^(I*t)       0       0       0       0]
[      0       0 e^(I*t)       0       0]
[      0       0       0       0 e^(I*t)]
>>> a = Automaton([(Integer(0), Integer(1), Integer(0)),
...                (Integer(1), Integer(2), Integer(0)),
...                (Integer(2), Integer(0), Integer(1)),
...                (Integer(2), Integer(1), Integer(0))],
...               initial_states=[Integer(0)],
...               final_states=[Integer(0)])
>>> a.adjacency_matrix()                                                      #␣
↪needs sage.symbolic
[0 1 0]
[0 0 1]
[1 1 0]
```

asymptotic_moments(*variable=None*)

Return the main terms of expectation and variance of the sum of output labels and its covariance with the sum of input labels.

INPUT:

- `variable` – a symbol denoting the length of the input, (default: n)

OUTPUT: a dictionary consisting of

- `expectation` – $en + \mathrm{Order}(1)$,
- `variance` – $vn + \mathrm{Order}(1)$,
- `covariance` – $cn + \mathrm{Order}(1)$

for suitable constants e, v and c.

Assume that all input and output labels are numbers and that `self` is complete and has only one final component. Assume further that this final component is aperiodic. Furthermore, assume that there is exactly one initial state and that all states are final.

Denote by X_n the sum of output labels written by the finite state machine when reading a random input word of length n over the input alphabet (assuming equidistribution).

Then the expectation of X_n is $en + O(1)$, the variance of X_n is $vn + O(1)$ and the covariance of X_n and the sum of input labels is $cn + O(1)$, cf. [HKW2015], Theorem 3.9.

In the case of non-integer input or output labels, performance degrades significantly. For rational input and output labels, consider rescaling to integers. This limitation comes from the fact that determinants over polynomial rings can be computed much more efficiently than over the symbolic ring. In fact, we compute (parts) of a trivariate generating function where the input and output labels are exponents of some indeterminates, see [HKW2015], Theorem 3.9 for details. If those exponents are integers, we can use a polynomial ring.

EXAMPLES:

1. A trivial example: write the negative of the input:

```
sage: T = Transducer([(0, 0, 0, 0), (0, 0, 1, -1)],
....:                initial_states=[0],
....:                final_states=[0])
sage: T([0, 1, 1])
[0, -1, -1]

sage: # needs sage.symbolic
sage: moments = T.asymptotic_moments()
sage: moments['expectation']
-1/2*n + Order(1)
sage: moments['variance']
1/4*n + Order(1)
sage: moments['covariance']
-1/4*n + Order(1)
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(0), Integer(0), Integer(0)),
↪(Integer(0), Integer(0), Integer(1), -Integer(1))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(0)])
>>> T([Integer(0), Integer(1), Integer(1)])
[0, -1, -1]
```

(continues on next page)

(continued from previous page)

```
>>> # needs sage.symbolic
>>> moments = T.asymptotic_moments()
>>> moments['expectation']
-1/2*n + Order(1)
>>> moments['variance']
1/4*n + Order(1)
>>> moments['covariance']
-1/4*n + Order(1)
```

2. For the case of the Hamming weight of the non-adjacent-form (NAF) of integers, cf. the Wikipedia article Non-adjacent_form and the *example on recognizing NAFs*, the following agrees with the results in [HP2007].

   We first use the transducer to convert the standard binary expansion to the NAF given in [HP2007]. We use the parameter `with_final_word_out` such that we do not have to add sufficiently many trailing zeros:

```
sage: NAF = Transducer([(0, 0, 0, 0),
....:                     (0, '.1', 1, None),
....:                     ('.1', 0, 0, [1, 0]),
....:                     ('.1', 1, 1, [-1, 0]),
....:                     (1, 1, 1, 0),
....:                     (1, '.1', 0, None)],
....:                    initial_states=[0],
....:                    final_states=[0],
....:                    with_final_word_out=[0])
```

```
>>> from sage.all import *
>>> NAF = Transducer([(Integer(0), Integer(0), Integer(0), Integer(0)),
...                     (Integer(0), '.1', Integer(1), None),
...                     ('.1', Integer(0), Integer(0), [Integer(1),␣
↪Integer(0)]),
...                     ('.1', Integer(1), Integer(1), [-Integer(1),␣
↪Integer(0)]),
...                     (Integer(1), Integer(1), Integer(1), Integer(0)),
...                     (Integer(1), '.1', Integer(0), None)],
...                    initial_states=[Integer(0)],
...                    final_states=[Integer(0)],
...                    with_final_word_out=[Integer(0)])
```

   As an example, we compute the NAF of 27 by this transducer.

```
sage: binary_27 = 27.bits()
sage: binary_27
[1, 1, 0, 1, 1]
sage: NAF_27 = NAF(binary_27)
sage: NAF_27
[-1, 0, -1, 0, 0, 1, 0]
sage: ZZ(NAF_27, base=2)
27
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> binary_27 = Integer(27).bits()
>>> binary_27
[1, 1, 0, 1, 1]
>>> NAF_27 = NAF(binary_27)
>>> NAF_27
[-1, 0, -1, 0, 0, 1, 0]
>>> ZZ(NAF_27, base=Integer(2))
27
```

Next, we are only interested in the Hamming weight:

```
sage: def weight(state, input):
....:     if input is None:
....:         result = 0
....:     else:
....:         result = ZZ(input != 0)
....:     return (0, result)
sage: weight_transducer = Transducer(weight,
....:                                input_alphabet=[-1, 0, 1],
....:                                initial_states=[0],
....:                                final_states=[0])
sage: NAFweight = weight_transducer.composition(NAF)
sage: NAFweight.transitions()
[Transition from (0, 0) to (0, 0): 0|0,
 Transition from (0, 0) to ('.1', 0): 1|-,
 Transition from ('.1', 0) to (0, 0): 0|1,0,
 Transition from ('.1', 0) to (1, 0): 1|1,0,
 Transition from (1, 0) to ('.1', 0): 0|-,
 Transition from (1, 0) to (1, 0): 1|0]
sage: NAFweight(binary_27)
[1, 0, 1, 0, 0, 1, 0]
```

```
>>> from sage.all import *
>>> def weight(state, input):
...     if input is None:
...         result = Integer(0)
...     else:
...         result = ZZ(input != Integer(0))
...     return (Integer(0), result)
>>> weight_transducer = Transducer(weight,
...                                input_alphabet=[-Integer(1),␣
↪Integer(0), Integer(1)],
...                                initial_states=[Integer(0)],
...                                final_states=[Integer(0)])
>>> NAFweight = weight_transducer.composition(NAF)
>>> NAFweight.transitions()
[Transition from (0, 0) to (0, 0): 0|0,
 Transition from (0, 0) to ('.1', 0): 1|-,
 Transition from ('.1', 0) to (0, 0): 0|1,0,
 Transition from ('.1', 0) to (1, 0): 1|1,0,
 Transition from (1, 0) to ('.1', 0): 0|-,
 Transition from (1, 0) to (1, 0): 1|0]
```

(continues on next page)

(continued from previous page)

```
>>> NAFweight(binary_27)
[1, 0, 1, 0, 0, 1, 0]
```

Now, we actually compute the asymptotic moments:

```
sage: # needs sage.symbolic
sage: moments = NAFweight.asymptotic_moments()
sage: moments['expectation']
1/3*n + Order(1)
sage: moments['variance']
2/27*n + Order(1)
sage: moments['covariance']
Order(1)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> moments = NAFweight.asymptotic_moments()
>>> moments['expectation']
1/3*n + Order(1)
>>> moments['variance']
2/27*n + Order(1)
>>> moments['covariance']
Order(1)
```

3. This is Example 3.16 in [HKW2015], where a transducer with variable output labels is given. There, the aim was to choose the output labels of this very simple transducer such that the input and output sum are asymptotically independent, i.e., the constant c vanishes.

```
sage: # needs sage.symbolic
sage: var('a_1, a_2, a_3, a_4')
(a_1, a_2, a_3, a_4)
sage: T = Transducer([[0, 0, 0, a_1], [0, 1, 1, a_3],
....:                  [1, 0, 0, a_4], [1, 1, 1, a_2]],
....:                 initial_states=[0], final_states=[0, 1])
sage: moments = T.asymptotic_moments()
verbose 0 (...) Non-integer output weights lead to
significant performance degradation.
sage: moments['expectation']
1/4*(a_1 + a_2 + a_3 + a_4)*n + Order(1)
sage: moments['covariance']
-1/4*(a_1 - a_2)*n + Order(1)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> var('a_1, a_2, a_3, a_4')
(a_1, a_2, a_3, a_4)
>>> T = Transducer([[Integer(0), Integer(0), Integer(0), a_1],
↪[Integer(0), Integer(1), Integer(1), a_3],
...                 [Integer(1), Integer(0), Integer(0), a_4],
↪[Integer(1), Integer(1), Integer(1), a_2]],
...                initial_states=[Integer(0)], final_states=[Integer(0),
↪Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> moments = T.asymptotic_moments()
verbose 0 (...) Non-integer output weights lead to
significant performance degradation.
>>> moments['expectation']
1/4*(a_1 + a_2 + a_3 + a_4)*n + Order(1)
>>> moments['covariance']
-1/4*(a_1 - a_2)*n + Order(1)
```

Therefore, the asymptotic covariance vanishes if and only if $a_2 = a_1$.

4. This is Example 4.3 in [HKW2015], dealing with the transducer converting the binary expansion of an integer into Gray code (cf. the Wikipedia article Gray_code and the *example on Gray code*):

```
sage: # needs sage.symbolic
sage: moments = transducers.GrayCode().asymptotic_moments()
sage: moments['expectation']
1/2*n + Order(1)
sage: moments['variance']
1/4*n + Order(1)
sage: moments['covariance']
Order(1)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> moments = transducers.GrayCode().asymptotic_moments()
>>> moments['expectation']
1/2*n + Order(1)
>>> moments['variance']
1/4*n + Order(1)
>>> moments['covariance']
Order(1)
```

5. This is the first part of Example 4.4 in [HKW2015], counting the number of 10 blocks in the standard binary expansion. The least significant digit is at the left-most position:

```
sage: block10 = transducers.CountSubblockOccurrences(
....:     [1, 0],
....:     input_alphabet=[0, 1])
sage: sorted(block10.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|1,
 Transition from (1,) to (1,): 1|0]

sage: # needs sage.symbolic
sage: moments = block10.asymptotic_moments()
sage: moments['expectation']
1/4*n + Order(1)
sage: moments['variance']
1/16*n + Order(1)
sage: moments['covariance']
Order(1)
```

```
>>> from sage.all import *
>>> block10 = transducers.CountSubblockOccurrences(
...     [Integer(1), Integer(0)],
...     input_alphabet=[Integer(0), Integer(1)])
>>> sorted(block10.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|1,
 Transition from (1,) to (1,): 1|0]

>>> # needs sage.symbolic
>>> moments = block10.asymptotic_moments()
>>> moments['expectation']
1/4*n + Order(1)
>>> moments['variance']
1/16*n + Order(1)
>>> moments['covariance']
Order(1)
```

6. This is the second part of Example 4.4 in [HKW2015], counting the number of 11 blocks in the standard binary expansion. The least significant digit is at the left-most position:

```
sage: block11 = transducers.CountSubblockOccurrences(
....:     [1, 1],
....:     input_alphabet=[0, 1])
sage: sorted(block11.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|0,
 Transition from (1,) to (1,): 1|1]

sage: # needs sage.symbolic
sage: var('N')
N
sage: moments = block11.asymptotic_moments(N)
sage: moments['expectation']
1/4*N + Order(1)
sage: moments['variance']
5/16*N + Order(1)
sage: correlation = (moments['covariance'].coefficient(N) /
....:                (1/2 * sqrt(moments['variance'].coefficient(N))))
sage: correlation
2/5*sqrt(5)
```

```
>>> from sage.all import *
>>> block11 = transducers.CountSubblockOccurrences(
...     [Integer(1), Integer(1)],
...     input_alphabet=[Integer(0), Integer(1)])
>>> sorted(block11.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|0,
 Transition from (1,) to (1,): 1|1]
```

(continues on next page)

(continued from previous page)

```
>>> # needs sage.symbolic
>>> var('N')
N
>>> moments = block11.asymptotic_moments(N)
>>> moments['expectation']
1/4*N + Order(1)
>>> moments['variance']
5/16*N + Order(1)
>>> correlation = (moments['covariance'].coefficient(N) /
...                 (Integer(1)/Integer(2) * sqrt(moments['variance'].
↪coefficient(N))))
>>> correlation
2/5*sqrt(5)
```

7. This is Example 4.5 in [HKW2015], counting the number of 01 blocks minus the number of 10 blocks in the standard binary expansion. The least significant digit is at the left-most position:

```
sage: block01 = transducers.CountSubblockOccurrences(
....:     [0, 1],
....:     input_alphabet=[0, 1])
sage: product_01x10 = block01.cartesian_product(block10)
sage: block_difference = transducers.sub([0, 1])(product_01x10)
sage: T = block_difference.simplification().relabeled()
sage: T.transitions()
[Transition from 0 to 2: 0|-1,
 Transition from 0 to 0: 1|0,
 Transition from 1 to 2: 0|0,
 Transition from 1 to 0: 1|0,
 Transition from 2 to 2: 0|0,
 Transition from 2 to 0: 1|1]

sage: # needs sage.symbolic
sage: moments = T.asymptotic_moments()
sage: moments['expectation']
Order(1)
sage: moments['variance']
Order(1)
sage: moments['covariance']
Order(1)
```

```
>>> from sage.all import *
>>> block01 = transducers.CountSubblockOccurrences(
...     [Integer(0), Integer(1)],
...     input_alphabet=[Integer(0), Integer(1)])
>>> product_01x10 = block01.cartesian_product(block10)
>>> block_difference = transducers.sub([Integer(0), Integer(1)])(product_
↪01x10)
>>> T = block_difference.simplification().relabeled()
>>> T.transitions()
[Transition from 0 to 2: 0|-1,
 Transition from 0 to 0: 1|0,
```

(continues on next page)

(continued from previous page)

```
 Transition from 1 to 2: 0|0,
 Transition from 1 to 0: 1|0,
 Transition from 2 to 2: 0|0,
 Transition from 2 to 0: 1|1]

>>> # needs sage.symbolic
>>> moments = T.asymptotic_moments()
>>> moments['expectation']
Order(1)
>>> moments['variance']
Order(1)
>>> moments['covariance']
Order(1)
```

8. The finite state machine must have a unique final component:

```
sage: T = Transducer([(0, -1, -1, -1), (0, 1, 1, 1),
....:                  (-1, -1, -1, -1), (-1, -1, 1, -1),
....:                  (1, 1, -1, 1), (1, 1, 1, 1)],
....:                 initial_states=[0],
....:                 final_states=[0, 1, -1])
sage: T.asymptotic_moments()
Traceback (most recent call last):
...
NotImplementedError: asymptotic_moments is only
implemented for finite state machines with one final
component.
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), -Integer(1), -Integer(1), -Integer(1)), (Integer(0), Integer(1), Integer(1), Integer(1)),
...                 (-Integer(1), -Integer(1), -Integer(1), -Integer(1)), (-Integer(1), -Integer(1), Integer(1), -Integer(1)),
...                 (Integer(1), Integer(1), -Integer(1), Integer(1)), (Integer(1), Integer(1), Integer(1), Integer(1))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(0), Integer(1), -Integer(1)])
>>> T.asymptotic_moments()
Traceback (most recent call last):
...
NotImplementedError: asymptotic_moments is only
implemented for finite state machines with one final
component.
```

In this particular example, the first letter of the input decides whether we reach the loop at -1 or the loop at 1. In the first case, we have $X_n = -n$, while we have $X_n = n$ in the second case. Therefore, the expectation $E(X_n)$ of X_n is $E(X_n) = 0$. We get $(X_n - E(X_n))^2 = n^2$ in all cases, which results in a variance of n^2.

So this example shows that the variance may be non-linear if there is more than one final component.

ALGORITHM:

See [HKW2015], Theorem 3.9.

REFERENCES:

coaccessible_components()

Return the sub-machine induced by the coaccessible states of this finite state machine.

OUTPUT:

A finite state machine of the same type as this finite state machine.

EXAMPLES:

```
sage: A = automata.ContainsWord([1, 1],
....:     input_alphabet=[0, 1]).complement().minimization().relabeled()
sage: A.transitions()
[Transition from 0 to 0: 0|-,
 Transition from 0 to 0: 1|-,
 Transition from 1 to 2: 0|-,
 Transition from 1 to 0: 1|-,
 Transition from 2 to 2: 0|-,
 Transition from 2 to 1: 1|-]
sage: A.initial_states()
[2]
sage: A.final_states()
[1, 2]
sage: C = A.coaccessible_components()
sage: C.transitions()
[Transition from 1 to 2: 0|-,
 Transition from 2 to 2: 0|-,
 Transition from 2 to 1: 1|-]
```

```
>>> from sage.all import *
>>> A = automata.ContainsWord([Integer(1), Integer(1)],
...     input_alphabet=[Integer(0), Integer(1)]).complement().minimization().
↪relabeled()
>>> A.transitions()
[Transition from 0 to 0: 0|-,
 Transition from 0 to 0: 1|-,
 Transition from 1 to 2: 0|-,
 Transition from 1 to 0: 1|-,
 Transition from 2 to 2: 0|-,
 Transition from 2 to 1: 1|-]
>>> A.initial_states()
[2]
>>> A.final_states()
[1, 2]
>>> C = A.coaccessible_components()
>>> C.transitions()
[Transition from 1 to 2: 0|-,
 Transition from 2 to 2: 0|-,
 Transition from 2 to 1: 1|-]
```

See also

accessible_components(), *induced_sub_finite_state_machine()*

completion(*sink=None*)

Return a completion of this finite state machine.

INPUT:

- `sink` – either an instance of `FSMState` or a label for the sink (default: `None`); if `None`, the least available nonzero integer is used

OUTPUT:

A `FiniteStateMachine` of the same type as this finite state machine.

The resulting finite state machine is a complete version of this finite state machine. A finite state machine is considered to be complete if each transition has an input label of length one and for each pair (q, a) where q is a state and a is an element of the input alphabet, there is exactly one transition from q with input label a.

If this finite state machine is already complete, a deep copy is returned. Otherwise, a new non-final state (usually called a sink) is created and transitions to this sink are introduced as appropriate.

EXAMPLES:

```
sage: F = FiniteStateMachine([(0, 0, 0, 0),
....:                         (0, 1, 1, 1),
....:                         (1, 1, 0, 0)])
sage: F.is_complete()
False
sage: G1 = F.completion()
sage: G1.is_complete()
True
sage: G1.transitions()
[Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 2: 1|-,
 Transition from 2 to 2: 0|-,
 Transition from 2 to 2: 1|-]
sage: G2 = F.completion('Sink')
sage: G2.is_complete()
True
sage: G2.transitions()
[Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 'Sink': 1|-,
 Transition from 'Sink' to 'Sink': 0|-,
 Transition from 'Sink' to 'Sink': 1|-]
sage: F.completion(1)
Traceback (most recent call last):
...
ValueError: The finite state machine already contains a state
'1'.
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([(Integer(0), Integer(0), Integer(0), Integer(0)),
...                         (Integer(0), Integer(1), Integer(1), Integer(1)),
...                         (Integer(1), Integer(1), Integer(0), Integer(0))])
```

(continues on next page)

(continued from previous page)

```
>>> F.is_complete()
False
>>> G1 = F.completion()
>>> G1.is_complete()
True
>>> G1.transitions()
[Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 2: 1|-,
 Transition from 2 to 2: 0|-,
 Transition from 2 to 2: 1|-]
>>> G2 = F.completion('Sink')
>>> G2.is_complete()
True
>>> G2.transitions()
[Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 'Sink': 1|-,
 Transition from 'Sink' to 'Sink': 0|-,
 Transition from 'Sink' to 'Sink': 1|-]
>>> F.completion(Integer(1))
Traceback (most recent call last):
...
ValueError: The finite state machine already contains a state
'1'.
```

An input alphabet must be given:

```
sage: F = FiniteStateMachine([(0, 0, 0, 0),
....:                           (0, 1, 1, 1),
....:                           (1, 1, 0, 0)],
....:                          determine_alphabets=False)
sage: F.is_complete()
Traceback (most recent call last):
...
ValueError: No input alphabet is given. Try calling
determine_alphabets().
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([(Integer(0), Integer(0), Integer(0), Integer(0)),
...                           (Integer(0), Integer(1), Integer(1), Integer(1)),
...                           (Integer(1), Integer(1), Integer(0), Integer(0))],
...                          determine_alphabets=False)
>>> F.is_complete()
Traceback (most recent call last):
...
ValueError: No input alphabet is given. Try calling
determine_alphabets().
```

Non-deterministic machines are not allowed.

```
sage: F = FiniteStateMachine([(0, 0, 0, 0), (0, 1, 0, 0)])
sage: F.is_complete()
False
sage: F.completion()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
sage: F = FiniteStateMachine([(0, 0, [0, 0], 0)])
sage: F.is_complete()
False
sage: F.completion()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([(Integer(0), Integer(0), Integer(0), Integer(0)), (Integer(0), Integer(1), Integer(0), Integer(0))])
>>> F.is_complete()
False
>>> F.completion()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
>>> F = FiniteStateMachine([(Integer(0), Integer(0), [Integer(0), Integer(0)], Integer(0))])
>>> F.is_complete()
False
>>> F.completion()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
```

See also

is_complete(), *split_transitions()*, *determine_alphabets()*, *is_deterministic()*.

composition(*other*, *algorithm=None*, *only_accessible_components=True*)

Return a new transducer which is the composition of `self` and `other`.

INPUT:

- `other` – a transducer
- `algorithm` – can be one of the following
  - `direct` – the composition is calculated directly

    There can be arbitrarily many initial and final states, but the input and output labels must have length 1.

Warning

The output of `other` is fed into `self`.

- `explorative` – an explorative algorithm is used

  The input alphabet of `self` has to be specified.

  Warning

  The output of `other` is fed into `self`.

If algorithm is `None`, then the algorithm is chosen automatically (at the moment always `direct`, except when there are output words of `other` or input words of `self` of length greater than 1).

OUTPUT: a new transducer

The labels of the new finite state machine are pairs of states of the original finite state machines. The color of a new state is the tuple of colors of the constituent states.

EXAMPLES:

```
sage: F = Transducer([('A', 'B', 1, 0), ('B', 'A', 0, 1)],
....:                initial_states=['A', 'B'], final_states=['B'],
....:                determine_alphabets=True)
sage: G = Transducer([(1, 1, 1, 0), (1, 2, 0, 1),
....:                 (2, 2, 1, 1), (2, 2, 0, 0)],
....:                initial_states=[1], final_states=[2],
....:                determine_alphabets=True)
sage: Hd = F.composition(G, algorithm='direct')
sage: Hd.initial_states()
[(1, 'B'), (1, 'A')]
sage: Hd.transitions()
[Transition from (1, 'B') to (1, 'A'): 1|1,
 Transition from (1, 'A') to (2, 'B'): 0|0,
 Transition from (2, 'B') to (2, 'A'): 0|1,
 Transition from (2, 'A') to (2, 'B'): 1|0]
sage: He = F.composition(G, algorithm='explorative')
sage: He.initial_states()
[(1, 'A'), (1, 'B')]
sage: He.transitions()
[Transition from (1, 'A') to (2, 'B'): 0|0,
 Transition from (1, 'B') to (1, 'A'): 1|1,
 Transition from (2, 'B') to (2, 'A'): 0|1,
 Transition from (2, 'A') to (2, 'B'): 1|0]
sage: Hd == He
True
```

```
>>> from sage.all import *
>>> F = Transducer([('A', 'B', Integer(1), Integer(0)), ('B', 'A', Integer(0),
↪ Integer(1))],
...                initial_states=['A', 'B'], final_states=['B'],
...                determine_alphabets=True)
>>> G = Transducer([(Integer(1), Integer(1), Integer(1), Integer(0)),␣
↪(Integer(1), Integer(2), Integer(0), Integer(1)),
```

(continues on next page)

(continued from previous page)

```
...                   (Integer(2), Integer(2), Integer(1), Integer(1)),␣
↪(Integer(2), Integer(2), Integer(0), Integer(0))],
...                  initial_states=[Integer(1)], final_states=[Integer(2)],
...                  determine_alphabets=True)
>>> Hd = F.composition(G, algorithm='direct')
>>> Hd.initial_states()
[(1, 'B'), (1, 'A')]
>>> Hd.transitions()
[Transition from (1, 'B') to (1, 'A'): 1|1,
 Transition from (1, 'A') to (2, 'B'): 0|0,
 Transition from (2, 'B') to (2, 'A'): 0|1,
 Transition from (2, 'A') to (2, 'B'): 1|0]
>>> He = F.composition(G, algorithm='explorative')
>>> He.initial_states()
[(1, 'A'), (1, 'B')]
>>> He.transitions()
[Transition from (1, 'A') to (2, 'B'): 0|0,
 Transition from (1, 'B') to (1, 'A'): 1|1,
 Transition from (2, 'B') to (2, 'A'): 0|1,
 Transition from (2, 'A') to (2, 'B'): 1|0]
>>> Hd == He
True
```

The following example has output of length > 1, so the explorative algorithm has to be used (and is selected automatically).

```
sage: F = Transducer([('A', 'B', 1, [1, 0]), ('B', 'B', 1, 1),
....:                  ('B', 'B', 0, 0)],
....:                 initial_states=['A'], final_states=['B'])
sage: G = Transducer([(1, 1, 0, 0), (1, 2, 1, 0),
....:                  (2, 2, 0, 1), (2, 1, 1, 1)],
....:                 initial_states=[1], final_states=[1])
sage: He = G.composition(F, algorithm='explorative')
sage: He.transitions()
[Transition from ('A', 1) to ('B', 2): 1|0,1,
 Transition from ('B', 2) to ('B', 2): 0|1,
 Transition from ('B', 2) to ('B', 1): 1|1,
 Transition from ('B', 1) to ('B', 1): 0|0,
 Transition from ('B', 1) to ('B', 2): 1|0]
sage: Ha = G.composition(F)
sage: Ha == He
True
```

```
>>> from sage.all import *
>>> F = Transducer([('A', 'B', Integer(1), [Integer(1), Integer(0)]), ('B', 'B
↪', Integer(1), Integer(1)),
...                   ('B', 'B', Integer(0), Integer(0))],
...                  initial_states=['A'], final_states=['B'])
>>> G = Transducer([(Integer(1), Integer(1), Integer(0), Integer(0)),␣
↪(Integer(1), Integer(2), Integer(1), Integer(0)),
...                   (Integer(2), Integer(2), Integer(0), Integer(1)),␣
↪(Integer(2), Integer(1), Integer(1), Integer(1))],
```

(continues on next page)

(continued from previous page)

```
...                 initial_states=[Integer(1)], final_states=[Integer(1)])
>>> He = G.composition(F, algorithm='explorative')
>>> He.transitions()
[Transition from ('A', 1) to ('B', 2): 1|0,1,
 Transition from ('B', 2) to ('B', 2): 0|1,
 Transition from ('B', 2) to ('B', 1): 1|1,
 Transition from ('B', 1) to ('B', 1): 0|0,
 Transition from ('B', 1) to ('B', 2): 1|0]
>>> Ha = G.composition(F)
>>> Ha == He
True
```

Final output words are also considered:

```
sage: F = Transducer([('A', 'B', 1, 0), ('B', 'A', 0, 1)],
....:                initial_states=['A', 'B'],
....:                final_states=['A', 'B'])
sage: F.state('A').final_word_out = 0
sage: F.state('B').final_word_out = 1
sage: G = Transducer([(1, 1, 1, 0), (1, 2, 0, 1),
....:                 (2, 2, 1, 1), (2, 2, 0, 0)],
....:                initial_states=[1], final_states=[2])
sage: G.state(2).final_word_out = 0
sage: Hd = F.composition(G, algorithm='direct')
sage: Hd.final_states()
[(2, 'B')]
sage: He = F.composition(G, algorithm='explorative')
sage: He.final_states()
[(2, 'B')]
```

```
>>> from sage.all import *
>>> F = Transducer([('A', 'B', Integer(1), Integer(0)), ('B', 'A', Integer(0),
↪ Integer(1))],
...                 initial_states=['A', 'B'],
...                 final_states=['A', 'B'])
>>> F.state('A').final_word_out = Integer(0)
>>> F.state('B').final_word_out = Integer(1)
>>> G = Transducer([(Integer(1), Integer(1), Integer(1), Integer(0)),␣
↪(Integer(1), Integer(2), Integer(0), Integer(1)),
...                  (Integer(2), Integer(2), Integer(1), Integer(1)),␣
↪(Integer(2), Integer(2), Integer(0), Integer(0))],
...                 initial_states=[Integer(1)], final_states=[Integer(2)])
>>> G.state(Integer(2)).final_word_out = Integer(0)
>>> Hd = F.composition(G, algorithm='direct')
>>> Hd.final_states()
[(2, 'B')]
>>> He = F.composition(G, algorithm='explorative')
>>> He.final_states()
[(2, 'B')]
```

Note that `(2, 'A')` is not final, as the final output 0 of state 2 of G cannot be processed in state `'A'` of F.

```
sage: [s.final_word_out for s in Hd.final_states()]
[[1, 0]]
sage: [s.final_word_out for s in He.final_states()]
[[1, 0]]
sage: Hd == He
True
```

```
>>> from sage.all import *
>>> [s.final_word_out for s in Hd.final_states()]
[[1, 0]]
>>> [s.final_word_out for s in He.final_states()]
[[1, 0]]
>>> Hd == He
True
```

Here is a non-deterministic example with intermediate output length > 1.

```
sage: F = Transducer([(1, 1, 1, ['a', 'a']), (1, 2, 1, 'b'),
....:                 (2, 1, 2, 'a'), (2, 2, 2, 'b')],
....:                initial_states=[1, 2])
sage: G = Transducer([('A', 'A', 'a', 'i'),
....:                 ('A', 'B', 'a', 'l'),
....:                 ('B', 'B', 'b', 'e')],
....:                initial_states=['A', 'B'])
sage: G(F).transitions()
[Transition from (1, 'A') to (1, 'A'): 1|'i','i',
 Transition from (1, 'A') to (1, 'B'): 1|'i','l',
 Transition from (1, 'B') to (2, 'B'): 1|'e',
 Transition from (2, 'A') to (1, 'A'): 2|'i',
 Transition from (2, 'A') to (1, 'B'): 2|'l',
 Transition from (2, 'B') to (2, 'B'): 2|'e']
```

```
>>> from sage.all import *
>>> F = Transducer([(Integer(1), Integer(1), Integer(1), ['a', 'a']),
↪(Integer(1), Integer(2), Integer(1), 'b'),
...                 (Integer(2), Integer(1), Integer(2), 'a'), (Integer(2),
↪Integer(2), Integer(2), 'b')],
...                initial_states=[Integer(1), Integer(2)])
>>> G = Transducer([('A', 'A', 'a', 'i'),
...                 ('A', 'B', 'a', 'l'),
...                 ('B', 'B', 'b', 'e')],
...                initial_states=['A', 'B'])
>>> G(F).transitions()
[Transition from (1, 'A') to (1, 'A'): 1|'i','i',
 Transition from (1, 'A') to (1, 'B'): 1|'i','l',
 Transition from (1, 'B') to (2, 'B'): 1|'e',
 Transition from (2, 'A') to (1, 'A'): 2|'i',
 Transition from (2, 'A') to (1, 'B'): 2|'l',
 Transition from (2, 'B') to (2, 'B'): 2|'e']
```

Be aware that after composition, different transitions may share the same output label (same python object):

```
sage: F = Transducer([ ('A','B',0,0), ('B','A',0,0)],
....:                 initial_states=['A'],
....:                 final_states=['A'])
sage: F.transitions()[0].word_out is F.transitions()[1].word_out
False
sage: G = Transducer([('C','C',0,1)],
....:                 initial_states=['C'],
....:                 final_states=['C'])
sage: H = G.composition(F)
sage: H.transitions()[0].word_out is H.transitions()[1].word_out
True
```

```
>>> from sage.all import *
>>> F = Transducer([ ('A','B',Integer(0),Integer(0)), ('B','A',Integer(0),
↪Integer(0))],
...                 initial_states=['A'],
...                 final_states=['A'])
>>> F.transitions()[Integer(0)].word_out is F.transitions()[Integer(1)].word_
↪out
False
>>> G = Transducer([('C','C',Integer(0),Integer(1))],
...                 initial_states=['C'],
...                 final_states=['C'])
>>> H = G.composition(F)
>>> H.transitions()[Integer(0)].word_out is H.transitions()[Integer(1)].word_
↪out
True
```

concatenation(*other*)

Concatenate this finite state machine with another finite state machine.

INPUT:

- `other` – a *FiniteStateMachine*

OUTPUT:

A *FiniteStateMachine* of the same type as this finite state machine.

Assume that both finite state machines are automata. If $\mathcal{L}_1$ is the language accepted by this automaton and $\mathcal{L}_2$ is the language accepted by the other automaton, then the language accepted by the concatenated automaton is $\{w_1w_2 \mid w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2\}$ where w_1w_2 denotes the concatenation of the words w_1 and w_2.

Assume that both finite state machines are transducers and that this transducer maps words $w_1 \in \mathcal{L}_1$ to words $f_1(w_1)$ and that the other transducer maps words $w_2 \in \mathcal{L}_2$ to words $f_2(w_2)$. Then the concatenated transducer maps words w_1w_2 with $w_1 \in \mathcal{L}_1$ and $w_2 \in \mathcal{L}_2$ to $f_1(w_1)f_2(w_2)$. Here, w_1w_2 and $f_1(w_1)f_2(w_2)$ again denote concatenation of words.

The input alphabet is the union of the input alphabets (if possible) and `None` otherwise. In the latter case, try calling *determine_alphabets()*.

Instead of `A.concatenation(B)`, the notation `A * B` can be used.

EXAMPLES:

Concatenation of two automata:

```
sage: A = automata.Word([0])
sage: B = automata.Word([1])
sage: C = A.concatenation(B)
sage: C.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 1) to (1, 0): -|-,
 Transition from (1, 0) to (1, 1): 1|-]
sage: [w
....:  for w in ([0, 0], [0, 1], [1, 0], [1, 1])
....:  if C(w)]
[[0, 1]]
sage: from sage.combinat.finite_state_machine import (
....:     Automaton, Transducer)
sage: isinstance(C, Automaton)
True
```

```
>>> from sage.all import *
>>> A = automata.Word([Integer(0)])
>>> B = automata.Word([Integer(1)])
>>> C = A.concatenation(B)
>>> C.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 1) to (1, 0): -|-,
 Transition from (1, 0) to (1, 1): 1|-]
>>> [w
...  for w in ([Integer(0), Integer(0)], [Integer(0), Integer(1)],
↪[Integer(1), Integer(0)], [Integer(1), Integer(1)])
...  if C(w)]
[[0, 1]]
>>> from sage.combinat.finite_state_machine import (
...     Automaton, Transducer)
>>> isinstance(C, Automaton)
True
```

Concatenation of two transducers:

```
sage: A = Transducer([(0, 1, 0, 1), (0, 1, 1, 2)],
....:                initial_states=[0],
....:                final_states=[1])
sage: B = Transducer([(0, 1, 0, 1), (0, 1, 1, 0)],
....:                initial_states=[0],
....:                final_states=[1])
sage: C = A.concatenation(B)
sage: C.transitions()
[Transition from (0, 0) to (0, 1): 0|1,
 Transition from (0, 0) to (0, 1): 1|2,
 Transition from (0, 1) to (1, 0): -|-,
 Transition from (1, 0) to (1, 1): 0|1,
 Transition from (1, 0) to (1, 1): 1|0]
sage: [(w, C(w)) for w in ([0, 0], [0, 1], [1, 0], [1, 1])]
[([0, 0], [1, 1]),
 ([0, 1], [1, 0]),
 ([1, 0], [2, 1]),
```

(continues on next page)

(continued from previous page)

```
 ([1, 1], [2, 0])]
sage: isinstance(C, Transducer)
True
```

```
>>> from sage.all import *
>>> A = Transducer([(Integer(0), Integer(1), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(1), Integer(1), Integer(2))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(1)])
>>> B = Transducer([(Integer(0), Integer(1), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(1), Integer(1), Integer(0))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(1)])
>>> C = A.concatenation(B)
>>> C.transitions()
[Transition from (0, 0) to (0, 1): 0|1,
 Transition from (0, 0) to (0, 1): 1|2,
 Transition from (0, 1) to (1, 0): -|-,
 Transition from (1, 0) to (1, 1): 0|1,
 Transition from (1, 0) to (1, 1): 1|0]
>>> [(w, C(w)) for w in ([Integer(0), Integer(0)], [Integer(0), Integer(1)],␣
↪[Integer(1), Integer(0)], [Integer(1), Integer(1)])]
[([0, 0], [1, 1]),
 ([0, 1], [1, 0]),
 ([1, 0], [2, 1]),
 ([1, 1], [2, 0])]
>>> isinstance(C, Transducer)
True
```

Alternative notation as multiplication:

```
sage: C == A * B
True
```

```
>>> from sage.all import *
>>> C == A * B
True
```

Final output words are taken into account:

```
sage: A = Transducer([(0, 1, 0, 1)],
....:                initial_states=[0],
....:                final_states=[1])
sage: A.state(1).final_word_out = 2
sage: B = Transducer([(0, 1, 0, 3)],
....:                initial_states=[0],
....:                final_states=[1])
sage: B.state(1).final_word_out = 4
sage: C = A * B
sage: C([0, 0])
[1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> A = Transducer([(Integer(0), Integer(1), Integer(0), Integer(1))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(1)])
>>> A.state(Integer(1)).final_word_out = Integer(2)
>>> B = Transducer([(Integer(0), Integer(1), Integer(0), Integer(3))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(1)])
>>> B.state(Integer(1)).final_word_out = Integer(4)
>>> C = A * B
>>> C([Integer(0), Integer(0)])
[1, 2, 3, 4]
```

Handling of the input alphabet:

```
sage: A = Automaton([(0, 0, 0)])
sage: B = Automaton([(0, 0, 1)], input_alphabet=[1, 2])
sage: C = Automaton([(0, 0, 2)], determine_alphabets=False)
sage: D = Automaton([(0, 0, [[0, 0]])], input_alphabet=[[0, 0]])
sage: A.input_alphabet
[0]
sage: B.input_alphabet
[1, 2]
sage: C.input_alphabet is None
True
sage: D.input_alphabet
[[0, 0]]
sage: (A * B).input_alphabet
[0, 1, 2]
sage: (A * C).input_alphabet is None
True
sage: (A * D).input_alphabet is None
True
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(0), Integer(0))])
>>> B = Automaton([(Integer(0), Integer(0), Integer(1))], input_
↪alphabet=[Integer(1), Integer(2)])
>>> C = Automaton([(Integer(0), Integer(0), Integer(2))], determine_
↪alphabets=False)
>>> D = Automaton([(Integer(0), Integer(0), [[Integer(0), Integer(0)]])],␣
↪input_alphabet=[[Integer(0), Integer(0)]])
>>> A.input_alphabet
[0]
>>> B.input_alphabet
[1, 2]
>>> C.input_alphabet is None
True
>>> D.input_alphabet
[[0, 0]]
>>> (A * B).input_alphabet
[0, 1, 2]
>>> (A * C).input_alphabet is None
```

(continues on next page)

(continued from previous page)

```
True
>>> (A * D).input_alphabet is None
True
```

See also:

disjoint_union(), determine_alphabets().

construct_final_word_out(*letters*, *allow_non_final=True*)

This is an inplace version of with_final_word_out(). See with_final_word_out() for documentation and examples.

copy()

Return a (shallow) copy of the finite state machine.

OUTPUT: a new finite state machine

deepcopy(*memo=None*)

Return a deep copy of the finite state machine.

INPUT:

- memo – (default: None) a dictionary storing already processed elements

OUTPUT: a new finite state machine

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'A', 0, 1), ('A', 'A', 1, 0)])
sage: deepcopy(F)
Finite state machine with 1 state
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([('A', 'A', Integer(0), Integer(1)), ('A', 'A',
↪Integer(1), Integer(0))])
>>> deepcopy(F)
Finite state machine with 1 state
```

default_format_letter = <sage.misc.latex.Latex object>

default_format_transition_label(*word*)

Default formatting of words in transition labels for LaTeX output.

INPUT:

- word – list of letters

OUTPUT:

String representation of word suitable to be typeset in mathematical mode.

- For a non-empty word: Concatenation of the letters, piped through self.format_letter and separated by blanks.
- For an empty word: sage.combinat.finite_state_machine.EmptyWordLaTeX.

There is also a variant *format_transition_label_reversed()* writing the words in reversed order.

EXAMPLES:

1. Example of a non-empty word:

```
sage: T = Transducer()
sage: print(T.default_format_transition_label(
....:     ['a', 'alpha', 'a_1', '0', 0, (0, 1)]))
\text{\texttt{a}} \text{\texttt{alpha}}
\text{\texttt{a{\char`\_}1}} 0 0 \left(0, 1\right)
```

```
>>> from sage.all import *
>>> T = Transducer()
>>> print(T.default_format_transition_label(
...     ['a', 'alpha', 'a_1', '0', Integer(0), (Integer(0), Integer(1))]))
\text{\texttt{a}} \text{\texttt{alpha}}
\text{\texttt{a{\char`\_}1}} 0 0 \left(0, 1\right)
```

2. In the example above, `'a'` and `'alpha'` should perhaps be symbols:

```
sage: var('a alpha a_1')                                                    #␣
↪needs sage.symbolic
(a, alpha, a_1)
sage: print(T.default_format_transition_label([a, alpha, a_1]))             #␣
↪needs sage.symbolic
a \alpha a_{1}
```

```
>>> from sage.all import *
>>> var('a alpha a_1')                                                    #␣
↪needs sage.symbolic
(a, alpha, a_1)
>>> print(T.default_format_transition_label([a, alpha, a_1]))             #␣
↪needs sage.symbolic
a \alpha a_{1}
```

3. Example of an empty word:

```
sage: print(T.default_format_transition_label([]))
\varepsilon
```

```
>>> from sage.all import *
>>> print(T.default_format_transition_label([]))
\varepsilon
```

We can change this by setting `sage.combinat.finite_state_machine.EmptyWordLaTeX`:

```
sage: sage.combinat.finite_state_machine.EmptyWordLaTeX = ''
sage: T.default_format_transition_label([])
''
```

```
>>> from sage.all import *
>>> sage.combinat.finite_state_machine.EmptyWordLaTeX = ''
```

(continues on next page)

(continued from previous page)

```
>>> T.default_format_transition_label([])
''
```

Finally, we restore the default value:

```
sage: sage.combinat.finite_state_machine.EmptyWordLaTeX = r'\varepsilon'
```

```
>>> from sage.all import *
>>> sage.combinat.finite_state_machine.EmptyWordLaTeX = r'\varepsilon'
```

4. This method is the default value for `FiniteStateMachine.format_transition_label`. That can be changed to be any other function:

```
sage: A = Automaton([(0, 1, 0)])
sage: def custom_format_transition_label(word):
....:     return "t"
sage: A.latex_options(format_transition_label=custom_format_transition_
↪label)
sage: print(latex(A))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\node[state] (v1) at (-3.000000, 0.000000) {$1$};
\path[->] (v0) edge node[rotate=360.00, anchor=south] {$t$} (v1);
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(1), Integer(0))])
>>> def custom_format_transition_label(word):
...     return "t"
>>> A.latex_options(format_transition_label=custom_format_transition_
↪label)
>>> print(latex(A))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\node[state] (v1) at (-3.000000, 0.000000) {$1$};
\path[->] (v0) edge node[rotate=360.00, anchor=south] {$t$} (v1);
\end{tikzpicture}
```

delete_state(*s*)

Delete a state and all transitions coming or going to this state.

INPUT:

- `s` – a label of a state or an *FSMState*

OUTPUT: nothing

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t1 = FSMTransition('A', 'B', 0)
sage: t2 = FSMTransition('B', 'B', 1)
sage: F = FiniteStateMachine([t1, t2])
sage: F.delete_state('A')
```

(continues on next page)

(continued from previous page)

```
sage: F.transitions()
[Transition from 'B' to 'B': 1|-]
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMTransition
>>> t1 = FSMTransition('A', 'B', Integer(0))
>>> t2 = FSMTransition('B', 'B', Integer(1))
>>> F = FiniteStateMachine([t1, t2])
>>> F.delete_state('A')
>>> F.transitions()
[Transition from 'B' to 'B': 1|-]
```

delete_transition(*t*)

Delete a transition by removing it from the list of transitions of the state, where the transition starts.

INPUT:

- `t` – a transition

OUTPUT: nothing

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'B', 0), ('B', 'A', 1)])
sage: F.delete_transition(('A', 'B', 0))
sage: F.transitions()
[Transition from 'B' to 'A': 1|-]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([('A', 'B', Integer(0)), ('B', 'A', Integer(1))])
>>> F.delete_transition(('A', 'B', Integer(0)))
>>> F.transitions()
[Transition from 'B' to 'A': 1|-]
```

determine_alphabets(*reset=True*)

Determine the input and output alphabet according to the transitions in this finite state machine.

INPUT:

- `reset` – if reset is `True`, then the existing input and output alphabets are erased, otherwise new letters are appended to the existing alphabets.

OUTPUT: nothing

After this operation the input alphabet and the output alphabet of this finite state machine are a list of letters.

Todo

At the moment, the letters of the alphabets need to be hashable.

EXAMPLES:

```
sage: T = Transducer([(1, 1, 1, 0), (1, 2, 2, 1),
....:                 (2, 2, 1, 1), (2, 2, 0, 0)],
```

(continues on next page)

(continued from previous page)

```
....:                     final_states=[1],
....:                     determine_alphabets=False)
sage: T.state(1).final_word_out = [1, 4]
sage: (T.input_alphabet, T.output_alphabet)
(None, None)
sage: T.determine_alphabets()
sage: (T.input_alphabet, T.output_alphabet)
([0, 1, 2], [0, 1, 4])
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(1), Integer(1), Integer(1), Integer(0)),␣
↪(Integer(1), Integer(2), Integer(2), Integer(1)),
...                     (Integer(2), Integer(2), Integer(1), Integer(1)),␣
↪(Integer(2), Integer(2), Integer(0), Integer(0))],
...                   final_states=[Integer(1)],
...                   determine_alphabets=False)
>>> T.state(Integer(1)).final_word_out = [Integer(1), Integer(4)]
>>> (T.input_alphabet, T.output_alphabet)
(None, None)
>>> T.determine_alphabets()
>>> (T.input_alphabet, T.output_alphabet)
([0, 1, 2], [0, 1, 4])
```

See also

determine_input_alphabet(), *determine_output_alphabet()*.

determine_input_alphabet(*reset=True*)

Determine the input alphabet according to the transitions of this finite state machine.

INPUT:

- `reset` – boolean (default: `True`); if `True`, then the existing input alphabet is erased, otherwise new letters are appended to the existing alphabet.

OUTPUT: nothing

After this operation the input alphabet of this finite state machine is a list of letters.

Todo

At the moment, the letters of the alphabet need to be hashable.

EXAMPLES:

```
sage: T = Transducer([(1, 1, 1, 0), (1, 2, 2, 1),
....:                      (2, 2, 1, 1), (2, 2, 0, 0)],
....:                     final_states=[1],
....:                     determine_alphabets=False)
sage: (T.input_alphabet, T.output_alphabet)
(None, None)
```

(continues on next page)

(continued from previous page)

```
sage: T.determine_input_alphabet()
sage: (T.input_alphabet, T.output_alphabet)
([0, 1, 2], None)
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(1), Integer(1), Integer(1), Integer(0)),␣
↪(Integer(1), Integer(2), Integer(2), Integer(1)),
...                    (Integer(2), Integer(2), Integer(1), Integer(1)),␣
↪(Integer(2), Integer(2), Integer(0), Integer(0))],
...                   final_states=[Integer(1)],
...                   determine_alphabets=False)
>>> (T.input_alphabet, T.output_alphabet)
(None, None)
>>> T.determine_input_alphabet()
>>> (T.input_alphabet, T.output_alphabet)
([0, 1, 2], None)
```

See also

determine_output_alphabet(), *determine_alphabets()*.

determine_output_alphabet(*reset=True*)

Determine the output alphabet according to the transitions of this finite state machine.

INPUT:

- `reset` – boolean (default: `True`); if `True`, then the existing output alphabet is erased, otherwise new letters are appended to the existing alphabet.

OUTPUT: nothing

After this operation the output alphabet of this finite state machine is a list of letters.

Todo

At the moment, the letters of the alphabet need to be hashable.

EXAMPLES:

```
sage: T = Transducer([(1, 1, 1, 0), (1, 2, 2, 1),
....:                  (2, 2, 1, 1), (2, 2, 0, 0)],
....:                 final_states=[1],
....:                 determine_alphabets=False)
sage: T.state(1).final_word_out = [1, 4]
sage: (T.input_alphabet, T.output_alphabet)
(None, None)
sage: T.determine_output_alphabet()
sage: (T.input_alphabet, T.output_alphabet)
(None, [0, 1, 4])
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(1), Integer(1), Integer(1), Integer(0)),␣
↪(Integer(1), Integer(2), Integer(2), Integer(1)),
...                 (Integer(2), Integer(2), Integer(1), Integer(1)),␣
↪(Integer(2), Integer(2), Integer(0), Integer(0))],
...                final_states=[Integer(1)],
...                determine_alphabets=False)
>>> T.state(Integer(1)).final_word_out = [Integer(1), Integer(4)]
>>> (T.input_alphabet, T.output_alphabet)
(None, None)
>>> T.determine_output_alphabet()
>>> (T.input_alphabet, T.output_alphabet)
(None, [0, 1, 4])
```

See also

determine_input_alphabet(), *determine_alphabets()*.

digraph(*edge_labels='words_in_out'*)

Return the graph of the finite state machine with labeled vertices and labeled edges.

INPUT:

- **edge_label – (default: 'words_in_out') can be**
  - 'words_in_out' (labels will be strings 'i|o')
  - a function with which takes as input a transition and outputs (returns) the label

OUTPUT: a directed graph

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: T = Transducer()
sage: T.graph()
Looped multi-digraph on 0 vertices
sage: T.add_state(A)
'A'
sage: T.graph()
Looped multi-digraph on 1 vertex
sage: T.add_transition(('A', 'A', 0, 1))
Transition from 'A' to 'A': 0|1
sage: T.graph()
Looped multi-digraph on 1 vertex
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A')
>>> T = Transducer()
>>> T.graph()
Looped multi-digraph on 0 vertices
>>> T.add_state(A)
```

(continues on next page)

(continued from previous page)

```
'A'
>>> T.graph()
Looped multi-digraph on 1 vertex
>>> T.add_transition(('A', 'A', Integer(0), Integer(1)))
Transition from 'A' to 'A': 0|1
>>> T.graph()
Looped multi-digraph on 1 vertex
```

See also

`DiGraph`

disjoint_union(*other*)

Return the disjoint union of this and another finite state machine.

INPUT:

- `other` – a *FiniteStateMachine*

OUTPUT:

A finite state machine of the same type as this finite state machine.

In general, the disjoint union of two finite state machines is non-deterministic. In the case of a automata, the language accepted by the disjoint union is the union of the languages accepted by the constituent automata. In the case of transducer, for each successful path in one of the constituent transducers, there will be one successful path with the same input and output labels in the disjoint union.

The labels of the states of the disjoint union are pairs `(i, s)`: for each state `s` of this finite state machine, there is a state `(0, s)` in the disjoint union; for each state `s` of the other finite state machine, there is a state `(1, s)` in the disjoint union.

The input alphabet is the union of the input alphabets (if possible) and `None` otherwise. In the latter case, try calling *determine_alphabets()*.

The disjoint union can also be written as `A + B` or `A | B`.

EXAMPLES:

```
sage: A = Automaton([(0, 1, 0), (1, 0, 1)],
....:               initial_states=[0],
....:               final_states=[0])
sage: A([0, 1, 0, 1])
True
sage: B = Automaton([(0, 1, 0), (1, 2, 0), (2, 0, 1)],
....:               initial_states=[0],
....:               final_states=[0])
sage: B([0, 0, 1])
True
sage: C = A.disjoint_union(B)
sage: C
Automaton with 5 states
sage: C.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 1) to (0, 0): 1|-,
```

(continues on next page)

(continued from previous page)

```
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 1) to (1, 2): 0|-,
 Transition from (1, 2) to (1, 0): 1|-]
sage: C([0, 0, 1])
True
sage: C([0, 1, 0, 1])
True
sage: C([1])
False
sage: C.initial_states()
[(0, 0), (1, 0)]
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(1), Integer(0)), (Integer(1),␣
↪Integer(0), Integer(1))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(0)])
>>> A([Integer(0), Integer(1), Integer(0), Integer(1)])
True
>>> B = Automaton([(Integer(0), Integer(1), Integer(0)), (Integer(1),␣
↪Integer(2), Integer(0)), (Integer(2), Integer(0), Integer(1))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(0)])
>>> B([Integer(0), Integer(0), Integer(1)])
True
>>> C = A.disjoint_union(B)
>>> C
Automaton with 5 states
>>> C.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 1) to (0, 0): 1|-,
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 1) to (1, 2): 0|-,
 Transition from (1, 2) to (1, 0): 1|-]
>>> C([Integer(0), Integer(0), Integer(1)])
True
>>> C([Integer(0), Integer(1), Integer(0), Integer(1)])
True
>>> C([Integer(1)])
False
>>> C.initial_states()
[(0, 0), (1, 0)]
```

Instead of `.disjoint_union`, alternative notations are available:

```
sage: C1 = A + B
sage: C1 == C
True
sage: C2 = A | B
sage: C2 == C
True
```

```
>>> from sage.all import *
>>> C1 = A + B
>>> C1 == C
True
>>> C2 = A | B
>>> C2 == C
True
```

In general, the disjoint union is not deterministic.:

```
sage: C.is_deterministic()
False
sage: D = C.determinisation().minimization()
sage: D.is_equivalent(Automaton([(0, 0, 0), (0, 0, 1),
....:    (1, 7, 0), (1, 0, 1), (2, 6, 0), (2, 0, 1),
....:    (3, 5, 0), (3, 0, 1), (4, 0, 0), (4, 2, 1),
....:    (5, 0, 0), (5, 3, 1), (6, 4, 0), (6, 0, 1),
....:    (7, 4, 0), (7, 3, 1)],
....:    initial_states=[1],
....:    final_states=[1, 2, 3]))
True
```

```
>>> from sage.all import *
>>> C.is_deterministic()
False
>>> D = C.determinisation().minimization()
>>> D.is_equivalent(Automaton([(Integer(0), Integer(0), Integer(0)),␣
↪(Integer(0), Integer(0), Integer(1)),
...    (Integer(1), Integer(7), Integer(0)), (Integer(1), Integer(0),␣
↪Integer(1)), (Integer(2), Integer(6), Integer(0)), (Integer(2), Integer(0),␣
↪Integer(1)),
...    (Integer(3), Integer(5), Integer(0)), (Integer(3), Integer(0),␣
↪Integer(1)), (Integer(4), Integer(0), Integer(0)), (Integer(4), Integer(2),␣
↪Integer(1)),
...    (Integer(5), Integer(0), Integer(0)), (Integer(5), Integer(3),␣
↪Integer(1)), (Integer(6), Integer(4), Integer(0)), (Integer(6), Integer(0),␣
↪Integer(1)),
...    (Integer(7), Integer(4), Integer(0)), (Integer(7), Integer(3),␣
↪Integer(1))],
...    initial_states=[Integer(1)],
...    final_states=[Integer(1), Integer(2), Integer(3)]))
True
```

Disjoint union of transducers:

```
sage: T1 = Transducer([(0, 0, 0, 1)],
....:                 initial_states=[0],
....:                 final_states=[0])
sage: T2 = Transducer([(0, 0, 0, 2)],
....:                 initial_states=[0],
....:                 final_states=[0])
sage: T1([0])
[1]
```

(continues on next page)

(continued from previous page)

```
sage: T2([0])
[2]
sage: T = T1.disjoint_union(T2)
sage: T([0])
Traceback (most recent call last):
...
ValueError: Found more than one accepting path.
sage: T.process([0])
[(True, (0, 0), [1]), (True, (1, 0), [2])]
```

```
>>> from sage.all import *
>>> T1 = Transducer([(Integer(0), Integer(0), Integer(0), Integer(1))],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(0)])
>>> T2 = Transducer([(Integer(0), Integer(0), Integer(0), Integer(2))],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(0)])
>>> T1([Integer(0)])
[1]
>>> T2([Integer(0)])
[2]
>>> T = T1.disjoint_union(T2)
>>> T([Integer(0)])
Traceback (most recent call last):
...
ValueError: Found more than one accepting path.
>>> T.process([Integer(0)])
[(True, (0, 0), [1]), (True, (1, 0), [2])]
```

Handling of the input alphabet (see Issue #18989):

```
sage: A = Automaton([(0, 0, 0)])
sage: B = Automaton([(0, 0, 1)], input_alphabet=[1, 2])
sage: C = Automaton([(0, 0, 2)], determine_alphabets=False)
sage: D = Automaton([(0, 0, [[0, 0]])], input_alphabet=[[0, 0]])
sage: A.input_alphabet
[0]
sage: B.input_alphabet
[1, 2]
sage: C.input_alphabet is None
True
sage: D.input_alphabet
[[0, 0]]
sage: (A + B).input_alphabet
[0, 1, 2]
sage: (A + C).input_alphabet is None
True
sage: (A + D).input_alphabet is None
True
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(0), Integer(0))])
```

(continues on next page)

(continued from previous page)

```
>>> B = Automaton([(Integer(0), Integer(0), Integer(1))], input_
↪alphabet=[Integer(1), Integer(2)])
>>> C = Automaton([(Integer(0), Integer(0), Integer(2))], determine_
↪alphabets=False)
>>> D = Automaton([(Integer(0), Integer(0), [[Integer(0), Integer(0)]])],␣
↪input_alphabet=[[Integer(0), Integer(0)]])
>>> A.input_alphabet
[0]
>>> B.input_alphabet
[1, 2]
>>> C.input_alphabet is None
True
>>> D.input_alphabet
[[0, 0]]
>>> (A + B).input_alphabet
[0, 1, 2]
>>> (A + C).input_alphabet is None
True
>>> (A + D).input_alphabet is None
True
```

See also

Automaton.intersection(), *Transducer.intersection()*, *determine_alphabets()*.

empty_copy (*memo=None*, *new_class=None*)

Return an empty deep copy of the finite state machine, i.e., `input_alphabet`, `output_alphabet`, `on_duplicate_transition` are preserved, but states and transitions are not.

INPUT:

- `memo` – dictionary storing already processed elements
- `new_class` – a class for the copy; by default (`None`), the class of `self` is used

OUTPUT: a new finite state machine

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_
↪raise_error
sage: F = FiniteStateMachine([('A', 'A', 0, 2), ('A', 'A', 1, 3)],
....:                        input_alphabet=[0, 1],
....:                        output_alphabet=[2, 3],
....:                        on_duplicate_transition=duplicate_transition_
↪raise_error)
sage: FE = F.empty_copy(); FE
Empty finite state machine
sage: FE.input_alphabet
[0, 1]
sage: FE.output_alphabet
[2, 3]
```

(continues on next page)

(continued from previous page)

```
sage: FE.on_duplicate_transition == duplicate_transition_raise_error
True
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_raise_
↪error
>>> F = FiniteStateMachine([('A', 'A', Integer(0), Integer(2)), ('A', 'A',␣
↪Integer(1), Integer(3))],
...                        input_alphabet=[Integer(0), Integer(1)],
...                        output_alphabet=[Integer(2), Integer(3)],
...                        on_duplicate_transition=duplicate_transition_raise_
↪error)
>>> FE = F.empty_copy(); FE
Empty finite state machine
>>> FE.input_alphabet
[0, 1]
>>> FE.output_alphabet
[2, 3]
>>> FE.on_duplicate_transition == duplicate_transition_raise_error
True
```

epsilon_successors(*state*)

Return the dictionary with states reachable from `state` without reading anything from an input tape as keys. The values are lists of outputs.

INPUT:

- `state` – the state whose epsilon successors should be determined

OUTPUT: a dictionary mapping states to a list of output words

The states in the output are the epsilon successors of `state`. Each word of the list of output words is a word written when taking a path from `state` to the corresponding state.

EXAMPLES:

```
sage: T = Transducer([(0, 1, None, 'a'), (1, 2, None, 'b')])
sage: T.epsilon_successors(0)
{1: [['a']], 2: [['a', 'b']]}
sage: T.epsilon_successors(1)
{2: [['b']]}
sage: T.epsilon_successors(2)
{}
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), None, 'a'), (Integer(1),␣
↪Integer(2), None, 'b')])
>>> T.epsilon_successors(Integer(0))
{1: [['a']], 2: [['a', 'b']]}
>>> T.epsilon_successors(Integer(1))
{2: [['b']]}
>>> T.epsilon_successors(Integer(2))
{}
```

If there is a cycle with only epsilon transitions, then this cycle is only processed once and there is no infinite loop:

```
sage: S = Transducer([(0, 1, None, 'a'), (1, 0, None, 'b')])
sage: S.epsilon_successors(0)
{0: [['a', 'b']], 1: [['a']]}
sage: S.epsilon_successors(1)
{0: [['b']], 1: [['b', 'a']]}
```

```
>>> from sage.all import *
>>> S = Transducer([(Integer(0), Integer(1), None, 'a'), (Integer(1),␣
↪Integer(0), None, 'b')])
>>> S.epsilon_successors(Integer(0))
{0: [['a', 'b']], 1: [['a']]}
>>> S.epsilon_successors(Integer(1))
{0: [['b']], 1: [['b', 'a']]}
```

equivalence_classes()

Return a list of equivalence classes of states.

Two states a and b are equivalent if and only if there is a bijection φ between paths starting at a and paths starting at b with the following properties: Let p_a be a path from a to a' and p_b a path from b to b' such that $\varphi(p_a) = p_b$, then

- $p_a.word_{in} = p_b.word_{in}$,
- $p_a.word_{out} = p_b.word_{out}$,
- a' and b' have the same output label, and
- a' and b' are both final or both non-final and have the same final output word.

The function *equivalence_classes()* returns a list of the equivalence classes to this equivalence relation.

This is one step of Moore's minimization algorithm.

See also

minimization()

EXAMPLES:

```
sage: fsm = FiniteStateMachine([("A", "B", 0, 1), ("A", "B", 1, 0),
....:                           ("B", "C", 0, 0), ("B", "C", 1, 1),
....:                           ("C", "D", 0, 1), ("C", "D", 1, 0),
....:                           ("D", "A", 0, 0), ("D", "A", 1, 1)])
sage: sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
sage: fsm.state("A").is_final = True
sage: sorted(fsm.equivalence_classes())
[['A'], ['B'], ['C'], ['D']]
sage: fsm.state("C").is_final = True
sage: sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
sage: fsm.state("A").final_word_out = 1
sage: sorted(fsm.equivalence_classes())
```

(continues on next page)

(continued from previous page)

```
[['A'], ['B'], ['C'], ['D']]
sage: fsm.state("C").final_word_out = 1
sage: sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
```

```
>>> from sage.all import *
>>> fsm = FiniteStateMachine([("A", "B", Integer(0), Integer(1)), ("A", "B",
↪Integer(1), Integer(0)),
...                           ("B", "C", Integer(0), Integer(0)), ("B", "C",
↪Integer(1), Integer(1)),
...                           ("C", "D", Integer(0), Integer(1)), ("C", "D",
↪Integer(1), Integer(0)),
...                           ("D", "A", Integer(0), Integer(0)), ("D", "A",
↪Integer(1), Integer(1))])
>>> sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
>>> fsm.state("A").is_final = True
>>> sorted(fsm.equivalence_classes())
[['A'], ['B'], ['C'], ['D']]
>>> fsm.state("C").is_final = True
>>> sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
>>> fsm.state("A").final_word_out = Integer(1)
>>> sorted(fsm.equivalence_classes())
[['A'], ['B'], ['C'], ['D']]
>>> fsm.state("C").final_word_out = Integer(1)
>>> sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
```

final_components()

Return the final components of a finite state machine as finite state machines.

OUTPUT:

A list of finite state machines, each representing a final component of `self`.

A final component of a transducer `T` is a strongly connected component `C` such that there are no transitions of `T` leaving `C`.

The final components are the only parts of a transducer which influence the main terms of the asymptotic behaviour of the sum of output labels of a transducer, see [HKP2015] and [HKW2015].

EXAMPLES:

```
sage: T = Transducer([['A', 'B', 0, 0], ['B', 'C', 0, 1],
....:                 ['C', 'B', 0, 1], ['A', 'D', 1, 0],
....:                 ['D', 'D', 0, 0], ['D', 'B', 1, 0],
....:                 ['A', 'E', 2, 0], ['E', 'E', 0, 0]])
sage: FC = T.final_components()
sage: sorted(FC[0].transitions())
[Transition from 'B' to 'C': 0|1,
 Transition from 'C' to 'B': 0|1]
sage: FC[1].transitions()
[Transition from 'E' to 'E': 0|0]
```

```
>>> from sage.all import *
>>> T = Transducer([['A', 'B', Integer(0), Integer(0)], ['B', 'C', Integer(0),
↪ Integer(1)],
...                 ['C', 'B', Integer(0), Integer(1)], ['A', 'D', Integer(1),
↪ Integer(0)],
...                 ['D', 'D', Integer(0), Integer(0)], ['D', 'B', Integer(1),
↪ Integer(0)],
...                 ['A', 'E', Integer(2), Integer(0)], ['E', 'E', Integer(0),
↪ Integer(0)]])
>>> FC = T.final_components()
>>> sorted(FC[Integer(0)].transitions())
[Transition from 'B' to 'C': 0|1,
 Transition from 'C' to 'B': 0|1]
>>> FC[Integer(1)].transitions()
[Transition from 'E' to 'E': 0|0]
```

Another example (cycle of length 2):

```
sage: T = Automaton([[0, 1, 0], [1, 0, 0]])
sage: len(T.final_components()) == 1
True
sage: T.final_components()[0].transitions()
[Transition from 0 to 1: 0|-,
 Transition from 1 to 0: 0|-]
```

```
>>> from sage.all import *
>>> T = Automaton([[Integer(0), Integer(1), Integer(0)], [Integer(1),␣
↪Integer(0), Integer(0)]])
>>> len(T.final_components()) == Integer(1)
True
>>> T.final_components()[Integer(0)].transitions()
[Transition from 0 to 1: 0|-,
 Transition from 1 to 0: 0|-]
```

final_states()

Return a list of all final states.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_final=True)
sage: B = FSMState('B', is_initial=True)
sage: C = FSMState('C', is_final=True)
sage: F = FiniteStateMachine([(A, B), (A, C)])
sage: F.final_states()
['A', 'C']
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A', is_final=True)
>>> B = FSMState('B', is_initial=True)
>>> C = FSMState('C', is_final=True)
>>> F = FiniteStateMachine([(A, B), (A, C)])
```

(continues on next page)

(continued from previous page)

```
>>> F.final_states()
['A', 'C']
```

format_letter = <sage.misc.latex.Latex object>

format_letter_negative(*letter*)

Format negative numbers as overlined numbers, everything else by standard LaTeX formatting.

INPUT:

- `letter` – anything

OUTPUT:

Overlined absolute value if letter is a negative integer, `latex(letter)` otherwise.

EXAMPLES:

```
sage: A = Automaton([(0, 0, -1)])
sage: list(map(A.format_letter_negative, [-1, 0, 1, 'a', None]))
['\\overline{1}', 0, 1, \text{\texttt{a}}, \mathrm{None}]
sage: A.latex_options(format_letter=A.format_letter_negative)
sage: print(latex(A))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\path[->] (v0) edge[loop above] node {$\overline{1}$} ();
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(0), -Integer(1))])
>>> list(map(A.format_letter_negative, [-Integer(1), Integer(0), Integer(1),
↪'a', None]))
['\\overline{1}', 0, 1, \text{\texttt{a}}, \mathrm{None}]
>>> A.latex_options(format_letter=A.format_letter_negative)
>>> print(latex(A))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\path[->] (v0) edge[loop above] node {$\overline{1}$} ();
\end{tikzpicture}
```

format_transition_label(*word*)

Default formatting of words in transition labels for LaTeX output.

INPUT:

- `word` – list of letters

OUTPUT:

String representation of `word` suitable to be typeset in mathematical mode.

- For a non-empty word: Concatenation of the letters, piped through `self.format_letter` and separated by blanks.
- For an empty word: `sage.combinat.finite_state_machine.EmptyWordLaTeX`.

There is also a variant *format_transition_label_reversed()* writing the words in reversed order.

EXAMPLES:

1. Example of a non-empty word:

```
sage: T = Transducer()
sage: print(T.default_format_transition_label(
....:     ['a', 'alpha', 'a_1', '0', 0, (0, 1)]))
\text{\texttt{a}} \text{\texttt{alpha}}
\text{\texttt{a{\char`\_}1}} 0 0 \left(0, 1\right)
```

```
>>> from sage.all import *
>>> T = Transducer()
>>> print(T.default_format_transition_label(
...     ['a', 'alpha', 'a_1', '0', Integer(0), (Integer(0), Integer(1))]))
\text{\texttt{a}} \text{\texttt{alpha}}
\text{\texttt{a{\char`\_}1}} 0 0 \left(0, 1\right)
```

2. In the example above, `'a'` and `'alpha'` should perhaps be symbols:

```
sage: var('a alpha a_1')                                                    #␣
↪needs sage.symbolic
(a, alpha, a_1)
sage: print(T.default_format_transition_label([a, alpha, a_1]))           #␣
↪needs sage.symbolic
a \alpha a_{1}
```

```
>>> from sage.all import *
>>> var('a alpha a_1')                                                    #␣
↪needs sage.symbolic
(a, alpha, a_1)
>>> print(T.default_format_transition_label([a, alpha, a_1]))           #␣
↪needs sage.symbolic
a \alpha a_{1}
```

3. Example of an empty word:

```
sage: print(T.default_format_transition_label([]))
\varepsilon
```

```
>>> from sage.all import *
>>> print(T.default_format_transition_label([]))
\varepsilon
```

We can change this by setting `sage.combinat.finite_state_machine.EmptyWordLaTeX`:

```
sage: sage.combinat.finite_state_machine.EmptyWordLaTeX = ''
sage: T.default_format_transition_label([])
''
```

```
>>> from sage.all import *
>>> sage.combinat.finite_state_machine.EmptyWordLaTeX = ''
>>> T.default_format_transition_label([])
''
```

Finally, we restore the default value:

```
sage: sage.combinat.finite_state_machine.EmptyWordLaTeX = r'\varepsilon'
```

```
>>> from sage.all import *
>>> sage.combinat.finite_state_machine.EmptyWordLaTeX = r'\varepsilon'
```

4. This method is the default value for `FiniteStateMachine.format_transition_label`. That can be changed to be any other function:

```
sage: A = Automaton([(0, 1, 0)])
sage: def custom_format_transition_label(word):
....:     return "t"
sage: A.latex_options(format_transition_label=custom_format_transition_
↪label)
sage: print(latex(A))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\node[state] (v1) at (-3.000000, 0.000000) {$1$};
\path[->] (v0) edge node[rotate=360.00, anchor=south] {$t$} (v1);
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(1), Integer(0))])
>>> def custom_format_transition_label(word):
...     return "t"
>>> A.latex_options(format_transition_label=custom_format_transition_
↪label)
>>> print(latex(A))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\node[state] (v1) at (-3.000000, 0.000000) {$1$};
\path[->] (v0) edge node[rotate=360.00, anchor=south] {$t$} (v1);
\end{tikzpicture}
```

format_transition_label_reversed(*word*)

Format words in transition labels in reversed order.

INPUT:

- `word` – list of letters

OUTPUT:

String representation of `word` suitable to be typeset in mathematical mode, letters are written in reversed order.

This is the reversed version of *default_format_transition_label()*.

In digit expansions, digits are frequently processed from the least significant to the most significant position, but it is customary to write the least significant digit at the right-most position. Therefore, the labels have to be reversed.

EXAMPLES:

```
sage: T = Transducer([(0, 0, 0, [1, 2, 3])])
sage: T.format_transition_label_reversed([1, 2, 3])
'3 2 1'
```

(continues on next page)

(continued from previous page)

```
sage: T.latex_options(format_transition_label=T.format_transition_label_
↪reversed)
sage: print(latex(T))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\path[->] (v0) edge[loop above] node {$0\mid 3 2 1$} ();
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(0), Integer(0), [Integer(1),␣
↪Integer(2), Integer(3)])])
>>> T.format_transition_label_reversed([Integer(1), Integer(2), Integer(3)])
'3 2 1'
>>> T.latex_options(format_transition_label=T.format_transition_label_
↪reversed)
>>> print(latex(T))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\path[->] (v0) edge[loop above] node {$0\mid 3 2 1$} ();
\end{tikzpicture}
```

graph(*edge_labels='words_in_out'*)

Return the graph of the finite state machine with labeled vertices and labeled edges.

INPUT:

- **edge_label – (default: 'words_in_out') can be**
  - 'words_in_out' (labels will be strings 'i|o')
  - a function with which takes as input a transition and outputs (returns) the label

OUTPUT: a directed graph

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: T = Transducer()
sage: T.graph()
Looped multi-digraph on 0 vertices
sage: T.add_state(A)
'A'
sage: T.graph()
Looped multi-digraph on 1 vertex
sage: T.add_transition(('A', 'A', 0, 1))
Transition from 'A' to 'A': 0|1
sage: T.graph()
Looped multi-digraph on 1 vertex
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A')
>>> T = Transducer()
```

(continues on next page)

(continued from previous page)

```
>>> T.graph()
Looped multi-digraph on 0 vertices
>>> T.add_state(A)
'A'
>>> T.graph()
Looped multi-digraph on 1 vertex
>>> T.add_transition(('A', 'A', Integer(0), Integer(1)))
Transition from 'A' to 'A': 0|1
>>> T.graph()
Looped multi-digraph on 1 vertex
```

See also:

DiGraph

has_final_state(*state*)

Return whether `state` is one of the final states of the finite state machine.

INPUT:

- `state` can be a *FSMState* or a label.

OUTPUT: boolean

EXAMPLES:

```
sage: FiniteStateMachine(final_states=['A']).has_final_state('A')
True
```

```
>>> from sage.all import *
>>> FiniteStateMachine(final_states=['A']).has_final_state('A')
True
```

has_final_states()

Return whether the finite state machine has a final state.

OUTPUT: boolean

EXAMPLES:

```
sage: FiniteStateMachine().has_final_states()
False
```

```
>>> from sage.all import *
>>> FiniteStateMachine().has_final_states()
False
```

has_initial_state(*state*)

Return whether `state` is one of the initial states of the finite state machine.

INPUT:

- `state` can be a *FSMState* or a label.

OUTPUT: boolean

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'A')], initial_states=['A'])
sage: F.has_initial_state('A')
True
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([('A', 'A')], initial_states=['A'])
>>> F.has_initial_state('A')
True
```

has_initial_states()

Return whether the finite state machine has an initial state.

OUTPUT: boolean

EXAMPLES:

```
sage: FiniteStateMachine().has_initial_states()
False
```

```
>>> from sage.all import *
>>> FiniteStateMachine().has_initial_states()
False
```

has_state(*state*)

Return whether `state` is one of the states of the finite state machine.

INPUT:

- `state` can be a *FSMState* or a label of a state.

OUTPUT: boolean

EXAMPLES:

```
sage: FiniteStateMachine().has_state('A')
False
```

```
>>> from sage.all import *
>>> FiniteStateMachine().has_state('A')
False
```

has_transition(*transition*)

Return whether `transition` is one of the transitions of the finite state machine.

INPUT:

- `transition` has to be a *FSMTransition*.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t = FSMTransition('A', 'A', 0, 1)
sage: FiniteStateMachine().has_transition(t)
False
sage: FiniteStateMachine().has_transition(('A', 'A', 0, 1))
Traceback (most recent call last):
...
TypeError: Transition is not an instance of FSMTransition.
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMTransition
>>> t = FSMTransition('A', 'A', Integer(0), Integer(1))
>>> FiniteStateMachine().has_transition(t)
False
>>> FiniteStateMachine().has_transition(('A', 'A', Integer(0), Integer(1)))
Traceback (most recent call last):
...
TypeError: Transition is not an instance of FSMTransition.
```

induced_sub_finite_state_machine(*states*)

Return a sub-finite-state-machine of the finite state machine induced by the given states.

INPUT:

- `states` – list (or an iterator) of states (either labels or instances of *FSMState*) of the sub-finite-state-machine

OUTPUT:

A new finite state machine. It consists (of deep copies) of the given states and (deep copies) of all transitions of `self` between these states.

EXAMPLES:

```
sage: FSM = FiniteStateMachine([(0, 1, 0), (0, 2, 0),
....:                           (1, 2, 0), (2, 0, 0)])
sage: sub_FSM = FSM.induced_sub_finite_state_machine([0, 1])
sage: sub_FSM.states()
[0, 1]
sage: sub_FSM.transitions()
[Transition from 0 to 1: 0|-]
sage: FSM.induced_sub_finite_state_machine([3])
Traceback (most recent call last):
...
ValueError: 3 is not a state of this finite state machine.
```

```
>>> from sage.all import *
>>> FSM = FiniteStateMachine([(Integer(0), Integer(1), Integer(0)),
↪(Integer(0), Integer(2), Integer(0)),
...                           (Integer(1), Integer(2), Integer(0)),
↪(Integer(2), Integer(0), Integer(0))])
>>> sub_FSM = FSM.induced_sub_finite_state_machine([Integer(0), Integer(1)])
>>> sub_FSM.states()
[0, 1]
>>> sub_FSM.transitions()
```

(continues on next page)

(continued from previous page)

```
[Transition from 0 to 1: 0|-]
>>> FSM.induced_sub_finite_state_machine([Integer(3)])
Traceback (most recent call last):
...
ValueError: 3 is not a state of this finite state machine.
```

initial_states()

Return a list of all initial states.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_initial=True)
sage: B = FSMState('B')
sage: F = FiniteStateMachine([(A, B, 1, 0)])
sage: F.initial_states()
['A']
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A', is_initial=True)
>>> B = FSMState('B')
>>> F = FiniteStateMachine([(A, B, Integer(1), Integer(0))])
>>> F.initial_states()
['A']
```

input_alphabet = None

A list of letters representing the input alphabet of the finite state machine.

It can be set by the parameter `input_alphabet` when initializing a finite state machine, see *FiniteStateMachine*.

It can also be set by the method *determine_alphabets()*.

See also

FiniteStateMachine, *determine_alphabets()*, *output_alphabet*.

input_projection()

Return an automaton where the output of each transition of `self` is deleted.

OUTPUT: an automaton

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'B', 0, 1), ('A', 'A', 1, 1),
....:                          ('B', 'B', 1, 0)])
sage: G = F.input_projection()
sage: G.transitions()
[Transition from 'A' to 'B': 0|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 1|-]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([('A', 'B', Integer(0), Integer(1)), ('A', 'A',␣
↪Integer(1), Integer(1)),
...                          ('B', 'B', Integer(1), Integer(0))])
>>> G = F.input_projection()
>>> G.transitions()
[Transition from 'A' to 'B': 0|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 1|-]
```

intersection(*other*)

is_Markov_chain(*is_zero=None*)

Check whether `self` is a Markov chain where the transition probabilities are modeled as input labels.

INPUT:

- `is_zero` – by default (`is_zero=None`), checking for zero is simply done by `is_zero()`. This parameter can be used to provide a more sophisticated check for zero, e.g. in the case of symbolic probabilities, see the examples below.

OUTPUT: boolean

on_duplicate_transition must be *duplicate_transition_add_input()*, the sum of the input weights of the transitions leaving a state must add up to 1 and the sum of initial probabilities must add up to 1 (or all be `None`).

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
sage: F = Transducer([[0, 0, 1/4, 0], [0, 1, 3/4, 1],
....:                 [1, 0, 1/2, 0], [1, 1, 1/2, 1]],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: F.is_Markov_chain()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
>>> F = Transducer([[Integer(0), Integer(0), Integer(1)/Integer(4),␣
↪Integer(0)], [Integer(0), Integer(1), Integer(3)/Integer(4), Integer(1)],
...                 [Integer(1), Integer(0), Integer(1)/Integer(2),␣
↪Integer(0)], [Integer(1), Integer(1), Integer(1)/Integer(2), Integer(1)]],
...                on_duplicate_transition=duplicate_transition_add_input)
>>> F.is_Markov_chain()
True
```

on_duplicate_transition must be *duplicate_transition_add_input()*:

```
sage: F = Transducer([[0, 0, 1/4, 0], [0, 1, 3/4, 1],
....:                 [1, 0, 1/2, 0], [1, 1, 1/2, 1]])
sage: F.is_Markov_chain()
False
```

```
>>> from sage.all import *
>>> F = Transducer([[Integer(0), Integer(0), Integer(1)/Integer(4),␣
↪Integer(0)], [Integer(0), Integer(1), Integer(3)/Integer(4), Integer(1)],
...                 [Integer(1), Integer(0), Integer(1)/Integer(2),␣
↪Integer(0)], [Integer(1), Integer(1), Integer(1)/Integer(2), Integer(1)]])
>>> F.is_Markov_chain()
False
```

Sum of input labels of the transitions leaving states must be 1:

```
sage: F = Transducer([[0, 0, 1/4, 0], [0, 1, 3/4, 1],
....:                 [1, 0, 1/2, 0]],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: F.is_Markov_chain()
False
```

```
>>> from sage.all import *
>>> F = Transducer([[Integer(0), Integer(0), Integer(1)/Integer(4),␣
↪Integer(0)], [Integer(0), Integer(1), Integer(3)/Integer(4), Integer(1)],
...                 [Integer(1), Integer(0), Integer(1)/Integer(2),␣
↪Integer(0)]],
...                on_duplicate_transition=duplicate_transition_add_input)
>>> F.is_Markov_chain()
False
```

The initial probabilities of all states must be `None` or they must sum up to 1. The initial probabilities of all states have to be set in the latter case:

```
sage: F = Transducer([[0, 0, 1/4, 0], [0, 1, 3/4, 1],
....:                 [1, 0, 1, 0]],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: F.is_Markov_chain()
True
sage: F.state(0).initial_probability = 1/4
sage: F.is_Markov_chain()
False
sage: F.state(1).initial_probability = 7
sage: F.is_Markov_chain()
False
sage: F.state(1).initial_probability = 3/4
sage: F.is_Markov_chain()
True
```

```
>>> from sage.all import *
>>> F = Transducer([[Integer(0), Integer(0), Integer(1)/Integer(4),␣
↪Integer(0)], [Integer(0), Integer(1), Integer(3)/Integer(4), Integer(1)],
...                 [Integer(1), Integer(0), Integer(1), Integer(0)]],
...                on_duplicate_transition=duplicate_transition_add_input)
>>> F.is_Markov_chain()
True
>>> F.state(Integer(0)).initial_probability = Integer(1)/Integer(4)
>>> F.is_Markov_chain()
False
```

(continues on next page)

(continued from previous page)

```
>>> F.state(Integer(1)).initial_probability = Integer(7)
>>> F.is_Markov_chain()
False
>>> F.state(Integer(1)).initial_probability = Integer(3)/Integer(4)
>>> F.is_Markov_chain()
True
```

If the probabilities are variables in the symbolic ring, `assume()` will do the trick:

```
sage: # needs sage.symbolic
sage: var('p q')
(p, q)
sage: F = Transducer([(0, 0, p, 1), (0, 0, q, 0)],
....:                 on_duplicate_transition=duplicate_transition_add_input)
sage: assume(p + q == 1)
sage: (p + q - 1).is_zero()
True
sage: F.is_Markov_chain()
True
sage: forget()
sage: del(p, q)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> var('p q')
(p, q)
>>> F = Transducer([(Integer(0), Integer(0), p, Integer(1)), (Integer(0),␣
↪Integer(0), q, Integer(0))],
...                 on_duplicate_transition=duplicate_transition_add_input)
>>> assume(p + q == Integer(1))
>>> (p + q - Integer(1)).is_zero()
True
>>> F.is_Markov_chain()
True
>>> forget()
>>> del(p, q)
```

If the probabilities are variables in some polynomial ring, the parameter `is_zero` can be used:

```
sage: R.<p, q> = PolynomialRing(QQ)
sage: def is_zero_polynomial(polynomial):
....:     return polynomial in (p + q - 1)*R
sage: F = Transducer([(0, 0, p, 1), (0, 0, q, 0)],
....:                 on_duplicate_transition=duplicate_transition_add_input)
sage: F.state(0).initial_probability = p + q
sage: F.is_Markov_chain()
False
sage: F.is_Markov_chain(is_zero_polynomial)                          #␣
↪needs sage.libs.singular
True
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('p', 'q',)); (p, q,) = R._first_ngens(2)
>>> def is_zero_polynomial(polynomial):
...     return polynomial in (p + q - Integer(1))*R
>>> F = Transducer([(Integer(0), Integer(0), p, Integer(1)), (Integer(0),␣
↪Integer(0), q, Integer(0))],
...                on_duplicate_transition=duplicate_transition_add_input)
>>> F.state(Integer(0)).initial_probability = p + q
>>> F.is_Markov_chain()
False
>>> F.is_Markov_chain(is_zero_polynomial)                               #␣
↪needs sage.libs.singular
True
```

is_complete()

Return whether the finite state machine is complete.

OUTPUT: boolean

A finite state machine is considered to be complete if each transition has an input label of length one and for each pair (q, a) where q is a state and a is an element of the input alphabet, there is exactly one transition from q with input label a.

EXAMPLES:

```
sage: fsm = FiniteStateMachine([(0, 0, 0, 0),
....:                           (0, 1, 1, 1),
....:                           (1, 1, 0, 0)],
....:                          determine_alphabets=False)
sage: fsm.is_complete()
Traceback (most recent call last):
...
ValueError: No input alphabet is given. Try calling determine_alphabets().
sage: fsm.input_alphabet = [0, 1]
sage: fsm.is_complete()
False
sage: fsm.add_transition((1, 1, 1, 1))
Transition from 1 to 1: 1|1
sage: fsm.is_complete()
True
sage: fsm.add_transition((0, 0, 1, 0))
Transition from 0 to 0: 1|0
sage: fsm.is_complete()
False
```

```
>>> from sage.all import *
>>> fsm = FiniteStateMachine([(Integer(0), Integer(0), Integer(0),␣
↪Integer(0)),
...                           (Integer(0), Integer(1), Integer(1),␣
↪Integer(1)),
...                           (Integer(1), Integer(1), Integer(0),␣
↪Integer(0))],
...                          determine_alphabets=False)
>>> fsm.is_complete()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: No input alphabet is given. Try calling determine_alphabets().
>>> fsm.input_alphabet = [Integer(0), Integer(1)]
>>> fsm.is_complete()
False
>>> fsm.add_transition((Integer(1), Integer(1), Integer(1), Integer(1)))
Transition from 1 to 1: 1|1
>>> fsm.is_complete()
True
>>> fsm.add_transition((Integer(0), Integer(0), Integer(1), Integer(0)))
Transition from 0 to 0: 1|0
>>> fsm.is_complete()
False
```

is_connected()

is_deterministic()

Return whether the finite finite state machine is deterministic.

OUTPUT: boolean

A finite state machine is considered to be deterministic if each transition has input label of length one and for each pair (q, a) where q is a state and a is an element of the input alphabet, there is at most one transition from q with input label a. Furthermore, the finite state may not have more than one initial state.

EXAMPLES:

```
sage: fsm = FiniteStateMachine()
sage: fsm.add_transition(('A', 'B', 0, []))
Transition from 'A' to 'B': 0|-
sage: fsm.is_deterministic()
True
sage: fsm.add_transition(('A', 'C', 0, []))
Transition from 'A' to 'C': 0|-
sage: fsm.is_deterministic()
False
sage: fsm.add_transition(('A', 'B', [0,1], []))
Transition from 'A' to 'B': 0,1|-
sage: fsm.is_deterministic()
False
```

```
>>> from sage.all import *
>>> fsm = FiniteStateMachine()
>>> fsm.add_transition(('A', 'B', Integer(0), []))
Transition from 'A' to 'B': 0|-
>>> fsm.is_deterministic()
True
>>> fsm.add_transition(('A', 'C', Integer(0), []))
Transition from 'A' to 'C': 0|-
>>> fsm.is_deterministic()
False
>>> fsm.add_transition(('A', 'B', [Integer(0),Integer(1)], []))
```

(continues on next page)

(continued from previous page)

```
Transition from 'A' to 'B': 0,1|-
>>> fsm.is_deterministic()
False
```

Check that Issue #18556 is fixed:

```
sage: Automaton().is_deterministic()
True
sage: Automaton(initial_states=[0]).is_deterministic()
True
sage: Automaton(initial_states=[0, 1]).is_deterministic()
False
```

```
>>> from sage.all import *
>>> Automaton().is_deterministic()
True
>>> Automaton(initial_states=[Integer(0)]).is_deterministic()
True
>>> Automaton(initial_states=[Integer(0), Integer(1)]).is_deterministic()
False
```

is_monochromatic()

Check whether the colors of all states are equal.

OUTPUT: boolean

EXAMPLES:

```
sage: G = transducers.GrayCode()
sage: [s.color for s in G.iter_states()]
[None, None, None]
sage: G.is_monochromatic()
True
sage: G.state(1).color = 'blue'
sage: G.is_monochromatic()
False
```

```
>>> from sage.all import *
>>> G = transducers.GrayCode()
>>> [s.color for s in G.iter_states()]
[None, None, None]
>>> G.is_monochromatic()
True
>>> G.state(Integer(1)).color = 'blue'
>>> G.is_monochromatic()
False
```

iter_final_states()

Return an iterator of the final states.

OUTPUT: an iterator over all initial states

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_final=True)
sage: B = FSMState('B', is_initial=True)
sage: C = FSMState('C', is_final=True)
sage: F = FiniteStateMachine([(A, B), (A, C)])
sage: [s.label() for s in F.iter_final_states()]
['A', 'C']
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A', is_final=True)
>>> B = FSMState('B', is_initial=True)
>>> C = FSMState('C', is_final=True)
>>> F = FiniteStateMachine([(A, B), (A, C)])
>>> [s.label() for s in F.iter_final_states()]
['A', 'C']
```

iter_initial_states()

Return an iterator of the initial states.

OUTPUT: an iterator over all initial states

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_initial=True)
sage: B = FSMState('B')
sage: F = FiniteStateMachine([(A, B, 1, 0)])
sage: [s.label() for s in F.iter_initial_states()]
['A']
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A', is_initial=True)
>>> B = FSMState('B')
>>> F = FiniteStateMachine([(A, B, Integer(1), Integer(0))])
>>> [s.label() for s in F.iter_initial_states()]
['A']
```

iter_process(*input_tape=None*, *initial_state=None*, *process_iterator_class=None*, *iterator_type=None*, *automatic_output_type=False*, ***kwargs*)

This function returns an iterator for processing the input. See `process()` (which runs this iterator until the end) for more information.

INPUT:

- `iterator_type` – if `None` (default), then an instance of `FSMProcessIterator` is returned. If this is `'simple'` only an iterator over one output is returned (an exception is raised if this is not the case, i.e., if the process has branched).

See `process()` for a description of the other parameters.

OUTPUT: an iterator

EXAMPLES:

We can use `iter_process()` to deal with infinite words:

```
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])

sage: # needs sage.combinat
sage: words.FibonacciWord()
word: 0100101001001010010100100101001001010010...
sage: it = inverter.iter_process(
....:     words.FibonacciWord(), iterator_type='simple')
sage: Words([0,1])(it)
word: 1011010110110101101011011010110110101101...
```

```
>>> from sage.all import *
>>> inverter = Transducer({'A': [('A', Integer(0), Integer(1)), ('A',␣
→Integer(1), Integer(0))]},
...     initial_states=['A'], final_states=['A'])

>>> # needs sage.combinat
>>> words.FibonacciWord()
word: 0100101001001010010100100101001001010010...
>>> it = inverter.iter_process(
...     words.FibonacciWord(), iterator_type='simple')
>>> Words([Integer(0),Integer(1)])(it)
word: 1011010110110101101011011010110110101101...
```

This can also be done by:

```
sage: inverter.iter_process(words.FibonacciWord(),                          #␣
→needs sage.combinat
....:                       iterator_type='simple',
....:                       automatic_output_type=True)
word: 1011010110110101101011011010110110101101...
```

```
>>> from sage.all import *
>>> inverter.iter_process(words.FibonacciWord(),                          #␣
→needs sage.combinat
...                       iterator_type='simple',
...                       automatic_output_type=True)
word: 1011010110110101101011011010110110101101...
```

or even simpler by:

```
sage: inverter(words.FibonacciWord())                                       #␣
→needs sage.combinat
word: 1011010110110101101011011010110110101101...
```

```
>>> from sage.all import *
>>> inverter(words.FibonacciWord())                                       #␣
→needs sage.combinat
word: 1011010110110101101011011010110110101101...
```

To see what is going on, we use *iter_process()* without arguments:

```
sage: # needs sage.combinat
sage: from itertools import islice
sage: it = inverter.iter_process(words.FibonacciWord())
sage: for current in islice(it, 4r):
....:     print(current)
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0, 1]]
process (1 branch)
+ at state 'A'
+-- tape at 4, [[1, 0, 1, 1]]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> from itertools import islice
>>> it = inverter.iter_process(words.FibonacciWord())
>>> for current in islice(it, 4):
...     print(current)
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0, 1]]
process (1 branch)
+ at state 'A'
+-- tape at 4, [[1, 0, 1, 1]]
```

The following show the difference between using the `'simple'`-option and not using it. With this option, we have

```
sage: it = inverter.iter_process(input_tape=[0, 1, 1],
....:                            iterator_type='simple')
sage: for i, o in enumerate(it):
....:     print('step %s: output %s' % (i, o))
step 0: output 1
step 1: output 0
step 2: output 0
```

```
>>> from sage.all import *
>>> it = inverter.iter_process(input_tape=[Integer(0), Integer(1),␣
↪Integer(1)],
...                            iterator_type='simple')
```

(continues on next page)

(continued from previous page)

```
>>> for i, o in enumerate(it):
...     print('step %s: output %s' % (i, o))
step 0: output 1
step 1: output 0
step 2: output 0
```

So `iter_process()` is a generator expression which gives a new output letter in each step (and not more). In many cases this is sufficient.

Doing the same without the `'simple'`-option does not give the output directly; it has to be extracted first. On the other hand, additional information is presented:

```
sage: it = inverter.iter_process(input_tape=[0, 1, 1])
sage: for current in it:
....:     print(current)
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0, 0]]
process (0 branches)
sage: it.result()
[Branch(accept=True, state='A', output=[1, 0, 0])]
```

```
>>> from sage.all import *
>>> it = inverter.iter_process(input_tape=[Integer(0), Integer(1),
↪Integer(1)])
>>> for current in it:
...     print(current)
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0, 0]]
process (0 branches)
>>> it.result()
[Branch(accept=True, state='A', output=[1, 0, 0])]
```

One can see the growing of the output (the list of lists at the end of each entry).

Even if the transducer has transitions with empty or multiletter output, the simple iterator returns one new output letter in each step:

```
sage: T = Transducer([(0, 0, 0, []),
....:                 (0, 0, 1, [1]),
```

(continues on next page)

(continued from previous page)

```
....:                  (0, 0, 2, [2, 2])],
....:                  initial_states=[0])
sage: it = T.iter_process(input_tape=[0, 1, 2, 0, 1, 2],
....:                     iterator_type='simple')
sage: for i, o in enumerate(it):
....:     print('step %s: output %s' % (i, o))
step 0: output 1
step 1: output 2
step 2: output 2
step 3: output 1
step 4: output 2
step 5: output 2
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(0), Integer(0), []),
...                  (Integer(0), Integer(0), Integer(1), [Integer(1)]),
...                  (Integer(0), Integer(0), Integer(2), [Integer(2),␣
↪Integer(2)])],
...                 initial_states=[Integer(0)])
>>> it = T.iter_process(input_tape=[Integer(0), Integer(1), Integer(2),␣
↪Integer(0), Integer(1), Integer(2)],
...                     iterator_type='simple')
>>> for i, o in enumerate(it):
...     print('step %s: output %s' % (i, o))
step 0: output 1
step 1: output 2
step 2: output 2
step 3: output 1
step 4: output 2
step 5: output 2
```

See also

FiniteStateMachine.process(), *Automaton.process()*, *Transducer.process()*, *__call__()*, *FSMProcessIterator*.

iter_states()

Return an iterator of the states.

OUTPUT: an iterator of the states of the finite state machine

EXAMPLES:

```
sage: FSM = Automaton([('1', '2', 1), ('2', '2', 0)])
sage: [s.label() for s in FSM.iter_states()]
['1', '2']
```

```
>>> from sage.all import *
>>> FSM = Automaton([('1', '2', Integer(1)), ('2', '2', Integer(0))])
>>> [s.label() for s in FSM.iter_states()]
['1', '2']
```

iter_transitions(*from_state=None*)

Return an iterator of all transitions.

INPUT:

- `from_state` – (default: `None`) if `from_state` is given, then a list of transitions starting there is given

OUTPUT: an iterator of all transitions

EXAMPLES:

```
sage: FSM = Automaton([('1', '2', 1), ('2', '2', 0)])
sage: [(t.from_state.label(), t.to_state.label())
....:      for t in FSM.iter_transitions('1')]
[('1', '2')]
sage: [(t.from_state.label(), t.to_state.label())
....:      for t in FSM.iter_transitions('2')]
[('2', '2')]
sage: [(t.from_state.label(), t.to_state.label())
....:      for t in FSM.iter_transitions()]
[('1', '2'), ('2', '2')]
```

```
>>> from sage.all import *
>>> FSM = Automaton([('1', '2', Integer(1)), ('2', '2', Integer(0))])
>>> [(t.from_state.label(), t.to_state.label())
...      for t in FSM.iter_transitions('1')]
[('1', '2')]
>>> [(t.from_state.label(), t.to_state.label())
...      for t in FSM.iter_transitions('2')]
[('2', '2')]
>>> [(t.from_state.label(), t.to_state.label())
...      for t in FSM.iter_transitions()]
[('1', '2'), ('2', '2')]
```

kleene_star()

Compute the Kleene closure of this finite state machine.

OUTPUT:

A *FiniteStateMachine* of the same type as this finite state machine.

Assume that this finite state machine is an automaton recognizing the language $\mathcal{L}$. Then the Kleene star recognizes the language $\mathcal{L}^* = \{w_1 \dots w_n \mid n \geq 0, w_j \in \mathcal{L} \text{ for all } j\}$.

Assume that this finite state machine is a transducer realizing a function f on some alphabet $\mathcal{L}$. Then the Kleene star realizes a function g on $\mathcal{L}^*$ with $g(w_1 \dots w_n) = f(w_1) \dots f(w_n)$.

EXAMPLES:

Kleene star of an automaton:

```
sage: A = automata.Word([0, 1])
sage: B = A.kleene_star()
sage: B.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 2 to 0: -|-,
 Transition from 1 to 2: 1|-]
sage: from sage.combinat.finite_state_machine import (
```

(continues on next page)

(continued from previous page)

```
....:     is_Automaton, is_Transducer)
sage: isinstance(B, Automaton)
True
sage: [w for w in ([], [0, 1], [0, 1, 0], [0, 1, 0, 1], [0, 1, 1, 1])
....:  if B(w)]
[[],
 [0, 1],
 [0, 1, 0, 1]]
```

```
>>> from sage.all import *
>>> A = automata.Word([Integer(0), Integer(1)])
>>> B = A.kleene_star()
>>> B.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 2 to 0: -|-,
 Transition from 1 to 2: 1|-]
>>> from sage.combinat.finite_state_machine import (
...     is_Automaton, is_Transducer)
>>> isinstance(B, Automaton)
True
>>> [w for w in ([], [Integer(0), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(0)], [Integer(0), Integer(1), Integer(0), Integer(1)], [Integer(0),␣
↪Integer(1), Integer(1), Integer(1)])
...  if B(w)]
[[],
 [0, 1],
 [0, 1, 0, 1]]
```

Kleene star of a transducer:

```
sage: T = Transducer([(0, 1, 0, 1), (0, 1, 1, 0)],
....:                initial_states=[0],
....:                final_states=[1])
sage: S = T.kleene_star()
sage: S.transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: -|-]
sage: isinstance(S, Transducer)
True
sage: for w in ([], [0], [1], [0, 0], [0, 1]):
....:     print("{} {}".format(w, S.process(w)))
[]     (True, 0, [])
[0]    [(True, 0, [1]), (True, 1, [1])]
[1]    [(True, 0, [0]), (True, 1, [0])]
[0, 0] [(True, 0, [1, 1]), (True, 1, [1, 1])]
[0, 1] [(True, 0, [1, 0]), (True, 1, [1, 0])]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(1), Integer(1), Integer(0))],
...                initial_states=[Integer(0)],
```

(continues on next page)

(continued from previous page)

```
...                   final_states=[Integer(1)])
>>> S = T.kleene_star()
>>> S.transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: -|-]
>>> isinstance(S, Transducer)
True
>>> for w in ([], [Integer(0)], [Integer(1)], [Integer(0), Integer(0)],␣
↪[Integer(0), Integer(1)]):
...     print("{} {}".format(w, S.process(w)))
[]      (True, 0, [])
[0]     [(True, 0, [1]), (True, 1, [1])]
[1]     [(True, 0, [0]), (True, 1, [0])]
[0, 0] [(True, 0, [1, 1]), (True, 1, [1, 1])]
[0, 1] [(True, 0, [1, 0]), (True, 1, [1, 0])]
```

Final output words are taken into account:

```
sage: T = Transducer([(0, 1, 0, 1)],
....:                 initial_states=[0],
....:                 final_states=[1])
sage: T.state(1).final_word_out = 2
sage: S = T.kleene_star()
sage: sorted(S.process([0, 0]))
[(True, 0, [1, 2, 1, 2]), (True, 1, [1, 2, 1, 2])]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(0), Integer(1))],
...                   initial_states=[Integer(0)],
...                   final_states=[Integer(1)])
>>> T.state(Integer(1)).final_word_out = Integer(2)
>>> S = T.kleene_star()
>>> sorted(S.process([Integer(0), Integer(0)]))
[(True, 0, [1, 2, 1, 2]), (True, 1, [1, 2, 1, 2])]
```

Final output words may lead to undesirable situations if initial states and final states coincide:

```
sage: T = Transducer(initial_states=[0], final_states=[0])
sage: T.state(0).final_word_out = 1
sage: T([])
[1]
sage: S = T.kleene_star()
sage: S([])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input), but
output is written.
```

```
>>> from sage.all import *
>>> T = Transducer(initial_states=[Integer(0)], final_states=[Integer(0)])
>>> T.state(Integer(0)).final_word_out = Integer(1)
```

(continues on next page)

(continued from previous page)

```
>>> T([])
[1]
>>> S = T.kleene_star()
>>> S([])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input), but
output is written.
```

language (*max_length=None*, ***kwargs*)

Return all words that can be written by this transducer.

INPUT:

- `max_length` – integer or `None` (default). Only output words which come from inputs of length at most `max_length` will be considered. If `None`, then this iterates over all possible words without length restrictions.
- `kwargs` – will be passed on to the *process iterator*. See *process()* for a description.

OUTPUT: an iterator

EXAMPLES:

```
sage: NAF = Transducer([('I', 0, 0, None), ('I', 1, 1, None),
....:                   (0, 0, 0, 0), (0, 1, 1, 0),
....:                   (1, 0, 0, 1), (1, 2, 1, -1),
....:                   (2, 1, 0, 0), (2, 2, 1, 0)],
....:                  initial_states=['I'], final_states=[0],
....:                  input_alphabet=[0, 1])
sage: sorted(NAF.language(4),
....:        key=lambda o: (ZZ(o, base=2), len(o)))
[[], [0], [0, 0], [0, 0, 0],
 [1], [1, 0], [1, 0, 0],
 [0, 1], [0, 1, 0],
 [-1, 0, 1],
 [0, 0, 1],
 [1, 0, 1]]
```

```
>>> from sage.all import *
>>> NAF = Transducer([('I', Integer(0), Integer(0), None), ('I', Integer(1),␣
↪Integer(1), None),
...                   (Integer(0), Integer(0), Integer(0), Integer(0)),␣
↪(Integer(0), Integer(1), Integer(1), Integer(0)),
...                   (Integer(1), Integer(0), Integer(0), Integer(1)),␣
↪(Integer(1), Integer(2), Integer(1), -Integer(1)),
...                   (Integer(2), Integer(1), Integer(0), Integer(0)),␣
↪(Integer(2), Integer(2), Integer(1), Integer(0))],
...                  initial_states=['I'], final_states=[Integer(0)],
...                  input_alphabet=[Integer(0), Integer(1)])
>>> sorted(NAF.language(Integer(4)),
...        key=lambda o: (ZZ(o, base=Integer(2)), len(o)))
[[], [0], [0, 0], [0, 0, 0],
 [1], [1, 0], [1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 1], [0, 1, 0],
[-1, 0, 1],
[0, 0, 1],
[1, 0, 1]]
```

```
sage: iterator = NAF.language()
sage: next(iterator)
[]
sage: next(iterator)
[0]
sage: next(iterator)
[1]
sage: next(iterator)
[0, 0]
sage: next(iterator)
[0, 1]
```

```
>>> from sage.all import *
>>> iterator = NAF.language()
>>> next(iterator)
[]
>>> next(iterator)
[0]
>>> next(iterator)
[1]
>>> next(iterator)
[0, 0]
>>> next(iterator)
[0, 1]
```

See also

Automaton.language(), *process()*.

latex_options(*coordinates=None*, *format_state_label=None*, *format_letter=None*, *format_transition_label=None*, *loop_where=None*, *initial_where=None*, *accepting_style=None*, *accepting_distance=None*, *accepting_where=None*, *accepting_show_empty=None*)

Set options for LaTeX output via `latex()` and therefore `view()`.

INPUT:

- `coordinates` – dictionary or a function mapping labels of states to pairs interpreted as coordinates. If no coordinates are given, states a placed equidistantly on a circle of radius 3. See also *set_coordinates()*.
- `format_state_label` – a function mapping labels of states to a string suitable for typesetting in LaTeX's mathematics mode. If not given, `latex()` is used.
- `format_letter` – a function mapping letters of the input and output alphabets to a string suitable for typesetting in LaTeX's mathematics mode. If not given, *default_format_transition_label()* uses `latex()`.

- `format_transition_label` – a function mapping words over the input and output alphabets to a string suitable for typesetting in LaTeX's mathematics mode. If not given, *`default_format_transition_label()`* is used.
- `loop_where` – dictionary or a function mapping labels of initial states to one of `'above'`, `'left'`, `'below'`, `'right'`. If not given, `'above'` is used.
- `initial_where` – dictionary or a function mapping labels of initial states to one of `'above'`, `'left'`, `'below'`, `'right'`. If not given, TikZ' default (currently `'left'`) is used.
- `accepting_style` – one of `'accepting by double'` and `'accepting by arrow'`. If not given, `'accepting by double'` is used unless there are non-empty final output words.
- `accepting_distance` – string giving a LaTeX length used for the length of the arrow leading from a final state. If not given, TikZ' default (currently `'3ex'`) is used unless there are non-empty final output words, in which case `'7ex'` is used.
- `accepting_where` – dictionary or a function mapping labels of final states to one of `'above'`, `'left'`, `'below'`, `'right'`. If not given, TikZ' default (currently `'right'`) is used. If the final state has a final output word, it is also possible to give an angle in degrees.
- `accepting_show_empty` – if `True` the arrow of an empty final output word is labeled as well. Note that this implicitly implies `accepting_style='accepting by arrow'`. If not given, the default `False` is used.

OUTPUT: nothing

As TikZ (cf. the Wikipedia article PGF/TikZ) is used to typeset the graphics, the syntax is oriented on TikZ' syntax.

This is a convenience function collecting all options for LaTeX output. All of its functionality can also be achieved by directly setting the attributes

- `coordinates`, `format_label`, `loop_where`, `initial_where`, and `accepting_where` of *`FSMState`* (here, `format_label` is a callable without arguments, everything else is a specific value);
- `format_label` of *`FSMTransition`* (`format_label` is a callable without arguments);
- `format_state_label`, `format_letter`, `format_transition_label`, `accepting_style`, `accepting_distance`, and `accepting_show_empty` of *`FiniteStateMachine`*.

This function, however, also (somewhat) checks its input and serves to collect documentation on all these options.

The function can be called several times, only those arguments which are not `None` are taken into account. By the same means, it can be combined with directly setting some attributes as outlined above.

EXAMPLES:

See also the section on *LaTeX output* in the introductory examples of this module.

```
sage: T = Transducer(initial_states=[4],
....:        final_states=[0, 3])
sage: for j in srange(4):
....:     T.add_transition(4, j, 0, [0, j])
....:     T.add_transition(j, 4, 0, [0, -j])
....:     T.add_transition(j, j, 0, 0)
Transition from 4 to 0: 0|0,0
Transition from 0 to 4: 0|0,0
Transition from 0 to 0: 0|0
Transition from 4 to 1: 0|0,1
```

(continues on next page)

(continued from previous page)

```
Transition from 1 to 4: 0|0,-1
Transition from 1 to 1: 0|0
Transition from 4 to 2: 0|0,2
Transition from 2 to 4: 0|0,-2
Transition from 2 to 2: 0|0
Transition from 4 to 3: 0|0,3
Transition from 3 to 4: 0|0,-3
Transition from 3 to 3: 0|0
sage: T.add_transition(4, 4, 0, 0)
Transition from 4 to 4: 0|0
sage: T.state(3).final_word_out = [0, 0]
sage: T.latex_options(
....:     coordinates={4: (0, 0),
....:                  0: (-6, 3),
....:                  1: (-2, 3),
....:                  2: (2, 3),
....:                  3: (6, 3)},
....:     format_state_label=lambda x: r'\mathbf{%s}' % x,
....:     format_letter=lambda x: r'w_{%s}' % x,
....:     format_transition_label=lambda x:
....:         r"{\scriptstyle %s}" % T.default_format_transition_label(x),
....:     loop_where={4: 'below', 0: 'left', 1: 'above',
....:                 2: 'right', 3:'below'},
....:     initial_where=lambda x: 'above',
....:     accepting_style='accepting by double',
....:     accepting_distance='10ex',
....:     accepting_where={0: 'left', 3: 45}
....:     )
sage: T.state(4).format_label=lambda: r'\mathcal{I}'
sage: latex(T)
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, initial, initial where=above] (v0) at (0.000000, 0.000000) {$\
↪mathcal{I}$};
\node[state, accepting, accepting where=left] (v1) at (-6.000000, 3.000000) {
↪$\mathbf{0}$};
\node[state, accepting, accepting where=45] (v2) at (6.000000, 3.000000) {$\
↪mathbf{3}$};
\path[->] (v2.45.00) edge node[rotate=45.00, anchor=south] {$$ \mid {\
↪scriptstyle w_{0} w_{0}}$} ++(45.00:10ex);
\node[state] (v3) at (-2.000000, 3.000000) {$\mathbf{1}$};
\node[state] (v4) at (2.000000, 3.000000) {$\mathbf{2}$};
\path[->] (v1) edge[loop left] node[rotate=90, anchor=south] {${\scriptstyle␣
↪w_{0}}\mid {\scriptstyle w_{0}}$} ();
\path[->] (v1.-21.57) edge node[rotate=-26.57, anchor=south] {${\scriptstyle␣
↪w_{0}}\mid {\scriptstyle w_{0} w_{0}}$} (v0.148.43);
\path[->] (v3) edge[loop above] node {${\scriptstyle w_{0}}\mid {\scriptstyle␣
↪w_{0}}$} ();
\path[->] (v3.-51.31) edge node[rotate=-56.31, anchor=south] {${\scriptstyle␣
↪w_{0}}\mid {\scriptstyle w_{0} w_{-1}}$} (v0.118.69);
\path[->] (v4) edge[loop right] node[rotate=90, anchor=north] {${\scriptstyle␣
↪w_{0}}\mid {\scriptstyle w_{0}}$} ();
\path[->] (v4.-118.69) edge node[rotate=56.31, anchor=north] {${\scriptstyle␣
```

(continues on next page)

(continued from previous page)

```
↪w_{0}}\mid {\scriptstyle w_{0} w_{-2}}$} (v0.51.31);
\path[->] (v2) edge[loop below] node {${\scriptstyle w_{0}}\mid {\scriptstyle␣
↪w_{0}}$} ();
\path[->] (v2.-148.43) edge node[rotate=26.57, anchor=north] {${\scriptstyle␣
↪w_{0}}\mid {\scriptstyle w_{0} w_{-3}}$} (v0.21.57);
\path[->] (v0.158.43) edge node[rotate=333.43, anchor=north] {${\scriptstyle␣
↪w_{0}}\mid {\scriptstyle w_{0} w_{0}}$} (v1.328.43);
\path[->] (v0.128.69) edge node[rotate=303.69, anchor=north] {${\scriptstyle␣
↪w_{0}}\mid {\scriptstyle w_{0} w_{1}}$} (v3.298.69);
\path[->] (v0.61.31) edge node[rotate=56.31, anchor=south] {${\scriptstyle w_
↪{0}}\mid {\scriptstyle w_{0} w_{2}}$} (v4.231.31);
\path[->] (v0.31.57) edge node[rotate=26.57, anchor=south] {${\scriptstyle w_
↪{0}}\mid {\scriptstyle w_{0} w_{3}}$} (v2.201.57);
\path[->] (v0) edge[loop below] node {${\scriptstyle w_{0}}\mid {\scriptstyle␣
↪w_{0}}$} ();
\end{tikzpicture}
sage: view(T) # not tested
```

```
>>> from sage.all import *
>>> T = Transducer(initial_states=[Integer(4)],
...     final_states=[Integer(0), Integer(3)])
>>> for j in srange(Integer(4)):
...     T.add_transition(Integer(4), j, Integer(0), [Integer(0), j])
...     T.add_transition(j, Integer(4), Integer(0), [Integer(0), -j])
...     T.add_transition(j, j, Integer(0), Integer(0))
Transition from 4 to 0: 0|0,0
Transition from 0 to 4: 0|0,0
Transition from 0 to 0: 0|0
Transition from 4 to 1: 0|0,1
Transition from 1 to 4: 0|0,-1
Transition from 1 to 1: 0|0
Transition from 4 to 2: 0|0,2
Transition from 2 to 4: 0|0,-2
Transition from 2 to 2: 0|0
Transition from 4 to 3: 0|0,3
Transition from 3 to 4: 0|0,-3
Transition from 3 to 3: 0|0
>>> T.add_transition(Integer(4), Integer(4), Integer(0), Integer(0))
Transition from 4 to 4: 0|0
>>> T.state(Integer(3)).final_word_out = [Integer(0), Integer(0)]
>>> T.latex_options(
...     coordinates={Integer(4): (Integer(0), Integer(0)),
...                  Integer(0): (-Integer(6), Integer(3)),
...                  Integer(1): (-Integer(2), Integer(3)),
...                  Integer(2): (Integer(2), Integer(3)),
...                  Integer(3): (Integer(6), Integer(3))},
...     format_state_label=lambda x: r'\mathbf{%s}' % x,
...     format_letter=lambda x: r'w_{%s}' % x,
...     format_transition_label=lambda x:
...         r"{\scriptstyle %s}" % T.default_format_transition_label(x),
...     loop_where={Integer(4): 'below', Integer(0): 'left', Integer(1):
↪'above',
```

(continues on next page)

(continued from previous page)

```
...                 Integer(2): 'right', Integer(3):'below'},
...     initial_where=lambda x: 'above',
...     accepting_style='accepting by double',
...     accepting_distance='10ex',
...     accepting_where={Integer(0): 'left', Integer(3): Integer(45)}
...     )
>>> T.state(Integer(4)).format_label=lambda: r'\mathcal{I}'
>>> latex(T)
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, initial, initial where=above] (v0) at (0.000000, 0.000000) {$\
→mathcal{I}$};
\node[state, accepting, accepting where=left] (v1) at (-6.000000, 3.000000) {
→$\mathbf{0}$};
\node[state, accepting, accepting where=45] (v2) at (6.000000, 3.000000) {$\
→mathbf{3}$};
\path[->] (v2.45.00) edge node[rotate=45.00, anchor=south] {$$ \mid {\
→scriptstyle w_{0} w_{0}}$} ++(45.00:10ex);
\node[state] (v3) at (-2.000000, 3.000000) {$\mathbf{1}$};
\node[state] (v4) at (2.000000, 3.000000) {$\mathbf{2}$};
\path[->] (v1) edge[loop left] node[rotate=90, anchor=south] {${\scriptstyle␣
→w_{0}}\mid {\scriptstyle w_{0}}$} ();
\path[->] (v1.-21.57) edge node[rotate=-26.57, anchor=south] {${\scriptstyle␣
→w_{0}}\mid {\scriptstyle w_{0} w_{0}}$} (v0.148.43);
\path[->] (v3) edge[loop above] node {${\scriptstyle w_{0}}\mid {\scriptstyle␣
→w_{0}}$} ();
\path[->] (v3.-51.31) edge node[rotate=-56.31, anchor=south] {${\scriptstyle␣
→w_{0}}\mid {\scriptstyle w_{0} w_{-1}}$} (v0.118.69);
\path[->] (v4) edge[loop right] node[rotate=90, anchor=north] {${\scriptstyle␣
→w_{0}}\mid {\scriptstyle w_{0}}$} ();
\path[->] (v4.-118.69) edge node[rotate=56.31, anchor=north] {${\scriptstyle␣
→w_{0}}\mid {\scriptstyle w_{0} w_{-2}}$} (v0.51.31);
\path[->] (v2) edge[loop below] node {${\scriptstyle w_{0}}\mid {\scriptstyle␣
→w_{0}}$} ();
\path[->] (v2.-148.43) edge node[rotate=26.57, anchor=north] {${\scriptstyle␣
→w_{0}}\mid {\scriptstyle w_{0} w_{-3}}$} (v0.21.57);
\path[->] (v0.158.43) edge node[rotate=333.43, anchor=north] {${\scriptstyle␣
→w_{0}}\mid {\scriptstyle w_{0} w_{0}}$} (v1.328.43);
\path[->] (v0.128.69) edge node[rotate=303.69, anchor=north] {${\scriptstyle␣
→w_{0}}\mid {\scriptstyle w_{0} w_{1}}$} (v3.298.69);
\path[->] (v0.61.31) edge node[rotate=56.31, anchor=south] {${\scriptstyle w_
→{0}}\mid {\scriptstyle w_{0} w_{2}}$} (v4.231.31);
\path[->] (v0.31.57) edge node[rotate=26.57, anchor=south] {${\scriptstyle w_
→{0}}\mid {\scriptstyle w_{0} w_{3}}$} (v2.201.57);
\path[->] (v0) edge[loop below] node {${\scriptstyle w_{0}}\mid {\scriptstyle␣
→w_{0}}$} ();
\end{tikzpicture}
>>> view(T) # not tested
```

To actually see this, use the live documentation in the Sage notebook and execute the cells.

By changing some of the options, we get the following output:

```
sage: T.latex_options(
....:     format_transition_label=T.default_format_transition_label,
....:     accepting_style='accepting by arrow',
....:     accepting_show_empty=True
....:     )
sage: latex(T)
\begin{tikzpicture}[auto, initial text=, >=latex, accepting text=, accepting/.
↪style=accepting by arrow, accepting distance=10ex]
\node[state, initial, initial where=above] (v0) at (0.000000, 0.000000) {$\
↪mathcal{I}$};
\node[state] (v1) at (-6.000000, 3.000000) {$\mathbf{0}$};
\path[->] (v1.180.00) edge node[rotate=360.00, anchor=south] {$$ \mid \
↪varepsilon$} ++(180.00:10ex);
\node[state] (v2) at (6.000000, 3.000000) {$\mathbf{3}$};
\path[->] (v2.45.00) edge node[rotate=45.00, anchor=south] {$$ \mid w_{0} w_
↪{0}$} ++(45.00:10ex);
\node[state] (v3) at (-2.000000, 3.000000) {$\mathbf{1}$};
\node[state] (v4) at (2.000000, 3.000000) {$\mathbf{2}$};
\path[->] (v1) edge[loop left] node[rotate=90, anchor=south] {$w_{0}\mid w_{0}
↪$} ();
\path[->] (v1.-21.57) edge node[rotate=-26.57, anchor=south] {$w_{0}\mid w_{0}
↪ w_{0}$} (v0.148.43);
\path[->] (v3) edge[loop above] node {$w_{0}\mid w_{0}$} ();
\path[->] (v3.-51.31) edge node[rotate=-56.31, anchor=south] {$w_{0}\mid w_{0}
↪ w_{-1}$} (v0.118.69);
\path[->] (v4) edge[loop right] node[rotate=90, anchor=north] {$w_{0}\mid w_
↪{0}$} ();
\path[->] (v4.-118.69) edge node[rotate=56.31, anchor=north] {$w_{0}\mid w_{0}
↪ w_{-2}$} (v0.51.31);
\path[->] (v2) edge[loop below] node {$w_{0}\mid w_{0}$} ();
\path[->] (v2.-148.43) edge node[rotate=26.57, anchor=north] {$w_{0}\mid w_{0}
↪ w_{-3}$} (v0.21.57);
\path[->] (v0.158.43) edge node[rotate=333.43, anchor=north] {$w_{0}\mid w_{0}
↪ w_{0}$} (v1.328.43);
\path[->] (v0.128.69) edge node[rotate=303.69, anchor=north] {$w_{0}\mid w_{0}
↪ w_{1}$} (v3.298.69);
\path[->] (v0.61.31) edge node[rotate=56.31, anchor=south] {$w_{0}\mid w_{0}␣
↪w_{2}$} (v4.231.31);
\path[->] (v0.31.57) edge node[rotate=26.57, anchor=south] {$w_{0}\mid w_{0}␣
↪w_{3}$} (v2.201.57);
\path[->] (v0) edge[loop below] node {$w_{0}\mid w_{0}$} ();
\end{tikzpicture}
sage: view(T) # not tested
```

```
>>> from sage.all import *
>>> T.latex_options(
...     format_transition_label=T.default_format_transition_label,
...     accepting_style='accepting by arrow',
...     accepting_show_empty=True
...     )
>>> latex(T)
\begin{tikzpicture}[auto, initial text=, >=latex, accepting text=, accepting/.
```

(continues on next page)

(continued from previous page)

```
↪style=accepting by arrow, accepting distance=10ex]
\node[state, initial, initial where=above] (v0) at (0.000000, 0.000000) {$\
↪mathcal{I}$};
\node[state] (v1) at (-6.000000, 3.000000) {$\mathbf{0}$};
\path[->] (v1.180.00) edge node[rotate=360.00, anchor=south] {$$ \mid \
↪varepsilon$} ++(180.00:10ex);
\node[state] (v2) at (6.000000, 3.000000) {$\mathbf{3}$};
\path[->] (v2.45.00) edge node[rotate=45.00, anchor=south] {$$ \mid w_{0} w_
↪{0}$} ++(45.00:10ex);
\node[state] (v3) at (-2.000000, 3.000000) {$\mathbf{1}$};
\node[state] (v4) at (2.000000, 3.000000) {$\mathbf{2}$};
\path[->] (v1) edge[loop left] node[rotate=90, anchor=south] {$w_{0}\mid w_{0}
↪$} ();
\path[->] (v1.-21.57) edge node[rotate=-26.57, anchor=south] {$w_{0}\mid w_{0}
↪ w_{0}$} (v0.148.43);
\path[->] (v3) edge[loop above] node {$w_{0}\mid w_{0}$} ();
\path[->] (v3.-51.31) edge node[rotate=-56.31, anchor=south] {$w_{0}\mid w_{0}
↪ w_{-1}$} (v0.118.69);
\path[->] (v4) edge[loop right] node[rotate=90, anchor=north] {$w_{0}\mid w_
↪{0}$} ();
\path[->] (v4.-118.69) edge node[rotate=56.31, anchor=north] {$w_{0}\mid w_{0}
↪ w_{-2}$} (v0.51.31);
\path[->] (v2) edge[loop below] node {$w_{0}\mid w_{0}$} ();
\path[->] (v2.-148.43) edge node[rotate=26.57, anchor=north] {$w_{0}\mid w_{0}
↪ w_{-3}$} (v0.21.57);
\path[->] (v0.158.43) edge node[rotate=333.43, anchor=north] {$w_{0}\mid w_{0}
↪ w_{0}$} (v1.328.43);
\path[->] (v0.128.69) edge node[rotate=303.69, anchor=north] {$w_{0}\mid w_{0}
↪ w_{1}$} (v3.298.69);
\path[->] (v0.61.31) edge node[rotate=56.31, anchor=south] {$w_{0}\mid w_{0}␣
↪w_{2}$} (v4.231.31);
\path[->] (v0.31.57) edge node[rotate=26.57, anchor=south] {$w_{0}\mid w_{0}␣
↪w_{3}$} (v2.201.57);
\path[->] (v0) edge[loop below] node {$w_{0}\mid w_{0}$} ();
\end{tikzpicture}
>>> view(T) # not tested
```

markov_chain_simplification()

Consider `self` as Markov chain with probabilities as input labels and simplify it.

OUTPUT: simplified version of `self`

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
sage: T = Transducer([[1, 2, 1/4, 0], [1, -2, 1/4, 0], [1, -2, 1/2, 0],
....:                  [2, 2, 1/4, 1], [2, -2, 1/4, 1], [-2, -2, 1/4, 1],
....:                  [-2, 2, 1/4, 1], [2, 3, 1/2, 2], [-2, 3, 1/2, 2]],
....:                 initial_states=[1],
....:                 final_states=[3],
....:                 on_duplicate_transition=duplicate_transition_add_input)
sage: T1 = T.markov_chain_simplification()
```

(continues on next page)

(continued from previous page)

```
sage: sorted(T1.transitions())
[Transition from ((1,),) to ((2, -2),): 1|0,
 Transition from ((2, -2),) to ((2, -2),): 1/2|1,
 Transition from ((2, -2),) to ((3,),): 1/2|2]
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
>>> T = Transducer([[Integer(1), Integer(2), Integer(1)/Integer(4),␣
↪Integer(0)], [Integer(1), -Integer(2), Integer(1)/Integer(4), Integer(0)],␣
↪[Integer(1), -Integer(2), Integer(1)/Integer(2), Integer(0)],
...                 [Integer(2), Integer(2), Integer(1)/Integer(4),␣
↪Integer(1)], [Integer(2), -Integer(2), Integer(1)/Integer(4), Integer(1)],␣
↪[-Integer(2), -Integer(2), Integer(1)/Integer(4), Integer(1)],
...                 [-Integer(2), Integer(2), Integer(1)/Integer(4),␣
↪Integer(1)], [Integer(2), Integer(3), Integer(1)/Integer(2), Integer(2)], [-
↪Integer(2), Integer(3), Integer(1)/Integer(2), Integer(2)]],
...                initial_states=[Integer(1)],
...                final_states=[Integer(3)],
...                on_duplicate_transition=duplicate_transition_add_input)
>>> T1 = T.markov_chain_simplification()
>>> sorted(T1.transitions())
[Transition from ((1,),) to ((2, -2),): 1|0,
 Transition from ((2, -2),) to ((2, -2),): 1/2|1,
 Transition from ((2, -2),) to ((3,),): 1/2|2]
```

merged_transitions()

Merges transitions which have the same `from_state`, `to_state` and `word_out` while adding their `word_in`.

OUTPUT:

A finite state machine with merged transitions. If no mergers occur, return `self`.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
sage: T = Transducer([[1, 2, 1/4, 1], [1, -2, 1/4, 1], [1, -2, 1/2, 1],
....:                 [2, 2, 1/4, 1], [2, -2, 1/4, 1], [-2, -2, 1/4, 1],
....:                 [-2, 2, 1/4, 1], [2, 3, 1/2, 1], [-2, 3, 1/2, 1]],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: T1 = T.merged_transitions()
sage: T1 is T
False
sage: sorted(T1.transitions())
[Transition from -2 to -2: 1/4|1,
 Transition from -2 to 2: 1/4|1,
 Transition from -2 to 3: 1/2|1,
 Transition from 1 to 2: 1/4|1,
 Transition from 1 to -2: 3/4|1,
 Transition from 2 to -2: 1/4|1,
 Transition from 2 to 2: 1/4|1,
```

(continues on next page)

(continued from previous page)

```
Transition from 2 to 3: 1/2|1]
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
>>> T = Transducer([[Integer(1), Integer(2), Integer(1)/Integer(4),␣
↪Integer(1)], [Integer(1), -Integer(2), Integer(1)/Integer(4), Integer(1)],␣
↪[Integer(1), -Integer(2), Integer(1)/Integer(2), Integer(1)],
...                 [Integer(2), Integer(2), Integer(1)/Integer(4),␣
↪Integer(1)], [Integer(2), -Integer(2), Integer(1)/Integer(4), Integer(1)],␣
↪[-Integer(2), -Integer(2), Integer(1)/Integer(4), Integer(1)],
...                 [-Integer(2), Integer(2), Integer(1)/Integer(4),␣
↪Integer(1)], [Integer(2), Integer(3), Integer(1)/Integer(2), Integer(1)], [-
↪Integer(2), Integer(3), Integer(1)/Integer(2), Integer(1)]],
...                on_duplicate_transition=duplicate_transition_add_input)
>>> T1 = T.merged_transitions()
>>> T1 is T
False
>>> sorted(T1.transitions())
[Transition from -2 to -2: 1/4|1,
 Transition from -2 to 2: 1/4|1,
 Transition from -2 to 3: 1/2|1,
 Transition from 1 to 2: 1/4|1,
 Transition from 1 to -2: 3/4|1,
 Transition from 2 to -2: 1/4|1,
 Transition from 2 to 2: 1/4|1,
 Transition from 2 to 3: 1/2|1]
```

Applying the function again does not change the result:

```
sage: T2 = T1.merged_transitions()
sage: T2 is T1
True
```

```
>>> from sage.all import *
>>> T2 = T1.merged_transitions()
>>> T2 is T1
True
```

moments_waiting_time(*test=<class 'bool'>*, *is_zero=None*, *expectation_only=False*)

If this finite state machine acts as a Markov chain, return the expectation and variance of the number of steps until first writing `True`.

INPUT:

- `test` – (default: `bool`) a callable deciding whether an output label is to be considered `True`. By default, the standard conversion to boolean is used.
- `is_zero` – (default: `None`) a callable deciding whether an expression for a probability is zero. By default, checking for zero is simply done by `is_zero()`. This parameter can be used to provide a more sophisticated check for zero, e.g. in the case of symbolic probabilities, see the examples below. This parameter is passed on to `is_Markov_chain()`. This parameter only affects the input of the Markov chain.

- `expectation_only` – boolean (default: `False`); if set, the variance is not computed (in order to save time). By default, the variance is computed.

OUTPUT:

A dictionary (if `expectation_only=False`) consisting of

- `expectation`,
- `variance`.

Otherwise, just the expectation is returned (no dictionary for `expectation_only=True`).

Expectation and variance of the number of steps until first writing `True` (as determined by the parameter `test`).

ALGORITHM:

Relies on a (classical and easy) probabilistic argument, cf. [FGT1992], Eqns. (6) and (7).

For the variance, see [FHP2015], Section 2.

EXAMPLES:

1. The simplest example is to wait for the first 1 in a 0-1-string where both digits appear with probability $1/2$. In fact, the waiting time equals k if and only if the string starts with $0^{k-1}1$. This event occurs with probability 2^{-k}. Therefore, the expected waiting time and the variance are $\sum_{k\geq 1} k2^{-k} = 2$ and $\sum_{k\geq 1}(k-2)^2 2^{-k} = 2$:

```
sage: var('k')                                                              # needs sage.symbolic
k
sage: sum(k * 2^(-k), k, 1, infinity)                                       # needs sage.symbolic
2
sage: sum((k-2)^2 * 2^(-k), k, 1, infinity)                                 # needs sage.symbolic
2
```

```
>>> from sage.all import *
>>> var('k')                                                                # needs sage.symbolic
k
>>> sum(k * Integer(2)**(-k), k, Integer(1), infinity)                      # needs sage.symbolic
2
>>> sum((k-Integer(2))**Integer(2) * Integer(2)**(-k), k, Integer(1), infinity)                                           # needs sage.symbolic
2
```

We now compute the same expectation and variance by using a Markov chain:

```
sage: from sage.combinat.finite_state_machine import (
....:     duplicate_transition_add_input)
sage: T = Transducer(
....:     [(0, 0, 1/2, 0), (0, 0, 1/2, 1)],
....:     on_duplicate_transition=\
....:         duplicate_transition_add_input,
....:     initial_states=[0],
```

(continues on next page)

(continued from previous page)

```
....:     final_states=[0])
sage: T.moments_waiting_time()
{'expectation': 2, 'variance': 2}
sage: T.moments_waiting_time(expectation_only=True)
2
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import (
...     duplicate_transition_add_input)
>>> T = Transducer(
...     [(Integer(0), Integer(0), Integer(1)/Integer(2), Integer(0)),␣
↪(Integer(0), Integer(0), Integer(1)/Integer(2), Integer(1))],
...     on_duplicate_transition=        duplicate_transition_add_input,
...     initial_states=[Integer(0)],
...     final_states=[Integer(0)])
>>> T.moments_waiting_time()
{'expectation': 2, 'variance': 2}
>>> T.moments_waiting_time(expectation_only=True)
2
```

In the following, we replace the output `0` by `-1` and demonstrate the use of the parameter `test`:

```
sage: T.delete_transition((0, 0, 1/2, 0))
sage: T.add_transition((0, 0, 1/2, -1))
Transition from 0 to 0: 1/2|-1
sage: T.moments_waiting_time(test=lambda x: x<0)
{'expectation': 2, 'variance': 2}
```

```
>>> from sage.all import *
>>> T.delete_transition((Integer(0), Integer(0), Integer(1)/Integer(2),␣
↪Integer(0)))
>>> T.add_transition((Integer(0), Integer(0), Integer(1)/Integer(2), -
↪Integer(1)))
Transition from 0 to 0: 1/2|-1
>>> T.moments_waiting_time(test=lambda x: x<Integer(0))
{'expectation': 2, 'variance': 2}
```

2. Make sure that the transducer is actually a Markov chain. Although this is checked by the code, unexpected behaviour may still occur if the transducer looks like a Markov chain. In the following example, we 'forget' to assign probabilities, but due to a coincidence, all 'probabilities' add up to one. Nevertheless, 0 is never written, so the expectation is 1.

```
sage: T = Transducer([(0, 0, 0, 0), (0, 0, 1, 1)],
....:                on_duplicate_transition=\
....:                    duplicate_transition_add_input,
....:                initial_states=[0],
....:                final_states=[0])
sage: T.moments_waiting_time()
{'expectation': 1, 'variance': 0}
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(0), Integer(0), Integer(0)),␣
```

(continues on next page)

(continued from previous page)

```
↪(Integer(0), Integer(0), Integer(1), Integer(1))],
...                   on_duplicate_transition=                   duplicate_
↪transition_add_input,
...                   initial_states=[Integer(0)],
...                   final_states=[Integer(0)])
>>> T.moments_waiting_time()
{'expectation': 1, 'variance': 0}
```

3. If `True` is never written, the moments are `+Infinity`:

```
sage: T = Transducer([(0, 0, 1, 0)],
....:                   on_duplicate_transition=\
....:                       duplicate_transition_add_input,
....:                   initial_states=[0],
....:                   final_states=[0])
sage: T.moments_waiting_time()
{'expectation': +Infinity, 'variance': +Infinity}
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(0), Integer(1), Integer(0))],
...                   on_duplicate_transition=                   duplicate_
↪transition_add_input,
...                   initial_states=[Integer(0)],
...                   final_states=[Integer(0)])
>>> T.moments_waiting_time()
{'expectation': +Infinity, 'variance': +Infinity}
```

4. Let h and r be positive integers. We consider random strings of letters $1, \ldots, r$ where the letter j occurs with probability p_j. Let B be the random variable giving the first position of a block of h consecutive identical letters. Then

$$\mathbb{E}(B) = \frac{1}{\displaystyle\sum_{i=1}^{r} \frac{1}{p_i^{-1} + \cdots + p_i^{-h}}},$$

$$\mathbb{V}(B) = \frac{\displaystyle\sum_{i=1}^{r} \left(\frac{p_i + p_i^h}{1 - p_i^h} - 2h \frac{p_i^h (1 - p_i)}{(1 - p_i^h)^2} \right)}{\displaystyle\left(\sum_{i=1}^{r} \frac{1}{p_i^{-1} + \cdots + p_i^{-h}} \right)^2}$$

cf. [S1986], p. 62, or [FHP2015], Theorem 1. We now verify this with a transducer approach.

```
sage: # needs sage.libs.singular
sage: def test(h, r):
....:     R = PolynomialRing(
....:             QQ,
....:             names=['p_%d' % j for j in range(r)])
....:     p = R.gens()
....:     def is_zero(polynomial):
....:         return polynomial in (sum(p) - 1) * R
....:     theory_expectation = 1/(sum(1/sum(p[j]^(-i)
....:                         for i in range(1, h+1))
```

(continues on next page)

(continued from previous page)

```
....:                         for j in range(r)))
....:     theory_variance = sum(
....:         (p[i] + p[i]^h)/(1 - p[i]^h)
....:         - 2*h*p[i]^h * (1 - p[i])/(1 - p[i]^h)^2
....:         for i in range(r)
....:         ) * theory_expectation^2
....:     alphabet = list(range(r))
....:     counters = [
....:         transducers.CountSubblockOccurrences([j]*h,
....:                     alphabet)
....:         for j in alphabet]
....:     all_counter = counters[0].cartesian_product(
....:         counters[1:])
....:     adder = transducers.add(input_alphabet=[0, 1],
....:         number_of_operands=r)
....:     probabilities = Transducer(
....:        [(0, 0, p[j], j) for j in alphabet],
....:        initial_states=[0],
....:        final_states=[0],
....:        on_duplicate_transition=\
....:            duplicate_transition_add_input)
....:     chain = adder(all_counter(probabilities))
....:     result = chain.moments_waiting_time(
....:        is_zero=is_zero)
....:     return is_zero((result['expectation'] -
....:                theory_expectation).numerator()) \
....:            and \
....:            is_zero((result['variance'] -
....:                 theory_variance).numerator())
sage: test(2, 2)
True
sage: test(2, 3)
True
sage: test(3, 3)
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.singular
>>> def test(h, r):
...     R = PolynomialRing(
...             QQ,
...             names=['p_%d' % j for j in range(r)])
...     p = R.gens()
...     def is_zero(polynomial):
...         return polynomial in (sum(p) - Integer(1)) * R
...     theory_expectation = Integer(1)/(sum(Integer(1)/sum(p[j]**(-i)
...                         for i in range(Integer(1), h+Integer(1)))
...                         for j in range(r)))
...     theory_variance = sum(
...         (p[i] + p[i]**h)/(Integer(1) - p[i]**h)
...         - Integer(2)*h*p[i]**h * (Integer(1) - p[i])/(Integer(1) -
↪p[i]**h)**Integer(2)
```

(continues on next page)

(continued from previous page)

```
...             for i in range(r)
...             ) * theory_expectation**Integer(2)
...         alphabet = list(range(r))
...         counters = [
...             transducers.CountSubblockOccurrences([j]*h,
...                         alphabet)
...             for j in alphabet]
...         all_counter = counters[Integer(0)].cartesian_product(
...             counters[Integer(1):])
...         adder = transducers.add(input_alphabet=[Integer(0), Integer(1)],
...             number_of_operands=r)
...         probabilities = Transducer(
...            [(Integer(0), Integer(0), p[j], j) for j in alphabet],
...            initial_states=[Integer(0)],
...            final_states=[Integer(0)],
...            on_duplicate_transition=           duplicate_transition_add_
↪input)
...         chain = adder(all_counter(probabilities))
...         result = chain.moments_waiting_time(
...            is_zero=is_zero)
...         return is_zero((result['expectation'] -
...                     theory_expectation).numerator())            and        ␣
↪    is_zero((result['variance'] -
...                      theory_variance).numerator())
>>> test(Integer(2), Integer(2))
True
>>> test(Integer(2), Integer(3))
True
>>> test(Integer(3), Integer(3))
True
```

5. Consider the alphabet $\{0, \ldots, r-1\}$, some $1 \leq j \leq r$ and some $h \geq 1$. For some probabilities $p_0, \ldots, p_{r-1}$, we consider infinite words where the letters occur independently with the given probabilities. The random variable B_j is the first position n such that there exist j of the r letters having an h-run. The expectation of B_j is given in [FHP2015], Theorem 2. Here, we verify this result by using transducers:

```
sage: # needs sage.libs.singular
sage: def test(h, r, j):
....:     R = PolynomialRing(
....:             QQ,
....:             names=['p_%d' % i for i in range(r)])
....:     p = R.gens()
....:     def is_zero(polynomial):
....:         return polynomial in (sum(p) - 1) * R
....:     alphabet = list(range(r))
....:     counters = [
....:         transducers.Wait([0, 1])(
....:             transducers.CountSubblockOccurrences(
....:                 [i]*h,
....:                 alphabet))
....:         for i in alphabet]
....:     all_counter = counters[0].cartesian_product(
```

(continues on next page)

(continued from previous page)

```
....:         counters[1:])
....:     adder = transducers.add(input_alphabet=[0, 1],
....:         number_of_operands=r)
....:     threshold = transducers.map(
....:         f=lambda x: x >= j,
....:         input_alphabet=srange(r+1))
....:     probabilities = Transducer(
....:         [(0, 0, p[i], i) for i in alphabet],
....:         initial_states=[0],
....:         final_states=[0],
....:         on_duplicate_transition=\
....:             duplicate_transition_add_input)
....:     chain = threshold(adder(all_counter(
....:         probabilities)))
....:     result = chain.moments_waiting_time(
....:         is_zero=is_zero,
....:         expectation_only=True)
....:     R_v = PolynomialRing(
....:             QQ,
....:             names=['p_%d' % i for i in range(r)])
....:     v = R_v.gens()
....:     S = 1/(1 - sum(v[i]/(1+v[i])
....:                    for i in range(r)))
....:     alpha = [(p[i] - p[i]^h)/(1 - p[i])
....:              for i in range(r)]
....:     gamma = [p[i]/(1 - p[i]) for i in range(r)]
....:     alphabet_set = set(alphabet)
....:     expectation = 0
....:     for q in range(j):
....:         for M in Subsets(alphabet_set, q):
....:             summand = S
....:             for i in M:
....:                 summand = summand.subs(
....:                     {v[i]: gamma[i]}) -\
....:                     summand.subs({v[i]: alpha[i]})
....:             for i in alphabet_set - set(M):
....:                 summand = summand.subs(
....:                     {v[i]: alpha[i]})
....:             expectation += summand
....:     return is_zero((result - expectation).\
....:             numerator())
sage: test(2, 3, 2)
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.singular
>>> def test(h, r, j):
...     R = PolynomialRing(
...             QQ,
...             names=['p_%d' % i for i in range(r)])
...     p = R.gens()
...     def is_zero(polynomial):
```

(continues on next page)

(continued from previous page)

```
...         return polynomial in (sum(p) - Integer(1)) * R
...     alphabet = list(range(r))
...     counters = [
...         transducers.Wait([Integer(0), Integer(1)])(
...             transducers.CountSubblockOccurrences(
...                 [i]*h,
...                 alphabet))
...         for i in alphabet]
...     all_counter = counters[Integer(0)].cartesian_product(
...         counters[Integer(1):])
...     adder = transducers.add(input_alphabet=[Integer(0), Integer(1)],
...         number_of_operands=r)
...     threshold = transducers.map(
...         f=lambda x: x >= j,
...         input_alphabet=srange(r+Integer(1)))
...     probabilities = Transducer(
...         [(Integer(0), Integer(0), p[i], i) for i in alphabet],
...         initial_states=[Integer(0)],
...         final_states=[Integer(0)],
...         on_duplicate_transition=            duplicate_transition_add_
↪input)
...     chain = threshold(adder(all_counter(
...         probabilities)))
...     result = chain.moments_waiting_time(
...         is_zero=is_zero,
...         expectation_only=True)
...     R_v = PolynomialRing(
...             QQ,
...             names=['p_%d' % i for i in range(r)])
...     v = R_v.gens()
...     S = Integer(1)/(Integer(1) - sum(v[i]/(Integer(1)+v[i])
...                     for i in range(r)))
...     alpha = [(p[i] - p[i]**h)/(Integer(1) - p[i])
...              for i in range(r)]
...     gamma = [p[i]/(Integer(1) - p[i]) for i in range(r)]
...     alphabet_set = set(alphabet)
...     expectation = Integer(0)
...     for q in range(j):
...         for M in Subsets(alphabet_set, q):
...             summand = S
...             for i in M:
...                 summand = summand.subs(
...                     {v[i]: gamma[i]}) -                    summand.
↪subs({v[i]: alpha[i]})
...             for i in alphabet_set - set(M):
...                 summand = summand.subs(
...                     {v[i]: alpha[i]})
...             expectation += summand
...     return is_zero((result - expectation).            numerator())
>>> test(Integer(2), Integer(3), Integer(2))
True
```

REFERENCES:

number_of_words(*variable=None*, *base_ring=None*)

Return the number of successful input words of given length.

INPUT:

- `variable` – a symbol denoting the length of the words, by default n
- `base_ring` – ring (default: `QQbar`) in which to compute the eigenvalues

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: NAFpm = Automaton([(0, 0, 0), (0, 1, 1),
....:                    (0, 1, -1), (1, 0, 0)],
....:                   initial_states=[0],
....:                   final_states=[0, 1])
sage: N = NAFpm.number_of_words(); N                                      #␣
↪needs sage.symbolic
4/3*2^n - 1/3*(-1)^n
sage: all(len(list(NAFpm.language(s)))                                   #␣
↪needs sage.symbolic
....:     - len(list(NAFpm.language(s-1))) == N.subs(n=s)
....:     for s in srange(1, 6))
True
```

```
>>> from sage.all import *
>>> NAFpm = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)),
...                    (Integer(0), Integer(1), -Integer(1)), (Integer(1),␣
↪Integer(0), Integer(0))],
...                   initial_states=[Integer(0)],
...                   final_states=[Integer(0), Integer(1)])
>>> N = NAFpm.number_of_words(); N                                        #␣
↪needs sage.symbolic
4/3*2^n - 1/3*(-1)^n
>>> all(len(list(NAFpm.language(s)))                                     #␣
↪needs sage.symbolic
...     - len(list(NAFpm.language(s-Integer(1)))) == N.subs(n=s)
...     for s in srange(Integer(1), Integer(6)))
True
```

An example with non-rational eigenvalues. By default, eigenvalues are elements of the `field of algebraic numbers`.

```
sage: NAFp = Automaton([(0, 0, 0), (0, 1, 1),  (1, 0, 0)],
....:                  initial_states=[0],
....:                  final_states=[0, 1])
sage: N = NAFp.number_of_words(); N                                       #␣
↪needs sage.rings.number_field sage.symbolic
1.170820393249937?*1.618033988749895?^n
- 0.1708203932499369?*(-0.618033988749895?)^n
sage: all(len(list(NAFp.language(s)))                                    #␣
↪needs sage.rings.number_field sage.symbolic
....:     - len(list(NAFp.language(s-1))) == N.subs(n=s)
```

(continues on next page)

(continued from previous page)

```
....:     for s in srange(1, 6))
True
```

```
>>> from sage.all import *
>>> NAFp = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)),  (Integer(1), Integer(0), Integer(0))],
...                  initial_states=[Integer(0)],
...                  final_states=[Integer(0), Integer(1)])
>>> N = NAFp.number_of_words(); N                                      #␣
↪needs sage.rings.number_field sage.symbolic
1.170820393249937?*1.618033988749895?^n
- 0.1708203932499369?*(-0.618033988749895?)^n
>>> all(len(list(NAFp.language(s)))                                    #␣
↪needs sage.rings.number_field sage.symbolic
...     - len(list(NAFp.language(s-Integer(1)))) == N.subs(n=s)
...     for s in srange(Integer(1), Integer(6)))
True
```

We specify a suitable `base_ring` to obtain a radical expression. To do so, we first compute the characteristic polynomial and then construct a number field generated by its roots.

```
sage: # needs sage.rings.number_field sage.symbolic
sage: M = NAFp.adjacency_matrix(entry=lambda t: 1)
sage: M.characteristic_polynomial()
x^2 - x - 1
sage: R.<phi> = NumberField(x^2 - x - 1, embedding=1.6)
sage: N = NAFp.number_of_words(base_ring=R); N
1/2*(1/2*sqrt(5) + 1/2)^n*(3*sqrt(1/5) + 1)
- 1/2*(-1/2*sqrt(5) + 1/2)^n*(3*sqrt(1/5) - 1)
sage: all(len(list(NAFp.language(s)))
....:     - len(list(NAFp.language(s-1))) == N.subs(n=s)
....:     for s in srange(1, 6))
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field sage.symbolic
>>> M = NAFp.adjacency_matrix(entry=lambda t: Integer(1))
>>> M.characteristic_polynomial()
x^2 - x - 1
>>> R = NumberField(x**Integer(2) - x - Integer(1), embedding=RealNumber('1.6
↪'), names=('phi',)); (phi,) = R._first_ngens(1)
>>> N = NAFp.number_of_words(base_ring=R); N
1/2*(1/2*sqrt(5) + 1/2)^n*(3*sqrt(1/5) + 1)
- 1/2*(-1/2*sqrt(5) + 1/2)^n*(3*sqrt(1/5) - 1)
>>> all(len(list(NAFp.language(s)))
...     - len(list(NAFp.language(s-Integer(1)))) == N.subs(n=s)
...     for s in srange(Integer(1), Integer(6)))
True
```

In this special case, we might also use the constant `golden_ratio`:

```
sage: # needs sage.rings.number_field sage.symbolic
sage: R.<phi> = NumberField(x^2-x-1, embedding=golden_ratio)
sage: N = NAFp.number_of_words(base_ring=R); N
1/5*(3*golden_ratio + 1)*golden_ratio^n
- 1/5*(3*golden_ratio - 4)*(-golden_ratio + 1)^n
sage: all(len(list(NAFp.language(s)))
....:     - len(list(NAFp.language(s-1))) == N.subs(n=s)
....:     for s in srange(1, 6))
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field sage.symbolic
>>> R = NumberField(x**Integer(2)-x-Integer(1), embedding=golden_ratio,␣
↪names=('phi',)); (phi,) = R._first_ngens(1)
>>> N = NAFp.number_of_words(base_ring=R); N
1/5*(3*golden_ratio + 1)*golden_ratio^n
- 1/5*(3*golden_ratio - 4)*(-golden_ratio + 1)^n
>>> all(len(list(NAFp.language(s)))
...     - len(list(NAFp.language(s-Integer(1)))) == N.subs(n=s)
...     for s in srange(Integer(1), Integer(6)))
True
```

The adjacency matrix of the following example is a Jordan matrix of size 3 to the eigenvalue 4:

```
sage: J3 = Automaton([(0, 1, -1), (1, 2, -1)],
....:     initial_states=[0],
....:     final_states=[0, 1, 2])
sage: for i in range(3):
....:     for j in range(4):
....:         new_transition = J3.add_transition(i, i, j)
sage: J3.adjacency_matrix(entry=lambda t: 1)
[4 1 0]
[0 4 1]
[0 0 4]
sage: N = J3.number_of_words(); N                                          #␣
↪needs sage.symbolic
1/2*4^(n - 2)*(n - 1)*n + 4^(n - 1)*n + 4^n
sage: all(len(list(J3.language(s)))                                        #␣
↪needs sage.symbolic
....:     - len(list(J3.language(s-1))) == N.subs(n=s)
....:     for s in range(1, 6))
True
```

```
>>> from sage.all import *
>>> J3 = Automaton([(Integer(0), Integer(1), -Integer(1)), (Integer(1),␣
↪Integer(2), -Integer(1))],
...     initial_states=[Integer(0)],
...     final_states=[Integer(0), Integer(1), Integer(2)])
>>> for i in range(Integer(3)):
...     for j in range(Integer(4)):
...         new_transition = J3.add_transition(i, i, j)
>>> J3.adjacency_matrix(entry=lambda t: Integer(1))
```

(continues on next page)

(continued from previous page)

```
[4 1 0]
[0 4 1]
[0 0 4]
>>> N = J3.number_of_words(); N                                          #␣
↪needs sage.symbolic
1/2*4^(n - 2)*(n - 1)*n + 4^(n - 1)*n + 4^n
>>> all(len(list(J3.language(s)))                                        #␣
↪needs sage.symbolic
...     - len(list(J3.language(s-Integer(1)))) == N.subs(n=s)
...     for s in range(Integer(1), Integer(6)))
True
```

Here is an automaton without cycles, so with eigenvalue 0.

```
sage: A = Automaton([(j, j+1, 0) for j in range(3)],
....:               initial_states=[0],
....:               final_states=list(range(3)))
sage: A.number_of_words()                                                 #␣
↪needs sage.symbolic
1/2*0^(n - 2)*(n - 1)*n + 0^(n - 1)*n + 0^n
```

```
>>> from sage.all import *
>>> A = Automaton([(j, j+Integer(1), Integer(0)) for j in range(Integer(3))],
...               initial_states=[Integer(0)],
...               final_states=list(range(Integer(3))))
>>> A.number_of_words()                                                  #␣
↪needs sage.symbolic
1/2*0^(n - 2)*(n - 1)*n + 0^(n - 1)*n + 0^n
```

on_duplicate_transition(*old_transition*, *new_transition*)

Which function to call when a duplicate transition is inserted.

It can be set by the parameter `on_duplicate_transition` when initializing a finite state machine, see *FiniteStateMachine*.

See also

FiniteStateMachine, *is_Markov_chain()*, *markov_chain_simplification()*.

output_alphabet = None

A list of letters representing the output alphabet of the finite state machine.

It can be set by the parameter `output_alphabet` when initializing a finite state machine, see *FiniteStateMachine*.

It can also be set by the method *determine_alphabets()*.

See also

FiniteStateMachine, *determine_alphabets()*, *input_alphabet*.

output_projection()

Return a automaton where the input of each transition of self is deleted and the new input is the original output.

OUTPUT: an automaton

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'B', 0, 1), ('A', 'A', 1, 1),
....:                         ('B', 'B', 1, 0)])
sage: G = F.output_projection()
sage: G.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 0|-]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([('A', 'B', Integer(0), Integer(1)), ('A', 'A',
↪Integer(1), Integer(1)),
...                         ('B', 'B', Integer(1), Integer(0))])
>>> G = F.output_projection()
>>> G.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 0|-]
```

Final output words are also considered correctly:

```
sage: H = Transducer([('A', 'B', 0, 1), ('A', 'A', 1, 1),
....:                 ('B', 'B', 1, 0), ('A', ('final', 0), 0, 0)],
....:                final_states=['A', 'B'])
sage: H.state('B').final_word_out = 2
sage: J = H.output_projection()
sage: J.states()
['A', 'B', ('final', 0), ('final', 1)]
sage: J.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'A' to ('final', 0): 0|-,
 Transition from 'B' to 'B': 0|-,
 Transition from 'B' to ('final', 1): 2|-]
sage: J.final_states()
['A', ('final', 1)]
```

```
>>> from sage.all import *
>>> H = Transducer([('A', 'B', Integer(0), Integer(1)), ('A', 'A', Integer(1),
↪ Integer(1)),
...                 ('B', 'B', Integer(1), Integer(0)), ('A', ('final',
↪Integer(0)), Integer(0), Integer(0))],
...                final_states=['A', 'B'])
>>> H.state('B').final_word_out = Integer(2)
>>> J = H.output_projection()
>>> J.states()
['A', 'B', ('final', 0), ('final', 1)]
```

(continues on next page)

(continued from previous page)

```
>>> J.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'A' to ('final', 0): 0|-,
 Transition from 'B' to 'B': 0|-,
 Transition from 'B' to ('final', 1): 2|-]
>>> J.final_states()
['A', ('final', 1)]
```

plot()

Plots a graph of the finite state machine with labeled vertices and labeled edges.

OUTPUT: a plot of the graph of the finite state machine

predecessors(*state*, *valid_input=None*)

List all predecessors of a state.

INPUT:

- `state` – the state from which the predecessors should be listed
- `valid_input` – if `valid_input` is a list, then we only consider transitions whose input labels are contained in `valid_input`. `state` has to be a *FSMState* (not a label of a state). If input labels of length larger than 1 are used, then `valid_input` has to be a list of lists.

OUTPUT: list of states

EXAMPLES:

```
sage: A = Transducer([('I', 'A', 'a', 'b'), ('I', 'B', 'b', 'c'),
....:                 ('I', 'C', 'c', 'a'), ('A', 'F', 'b', 'a'),
....:                 ('B', 'F', ['c', 'b'], 'b'), ('C', 'F', 'a', 'c')],
....:                initial_states=['I'], final_states=['F'])
sage: A.predecessors(A.state('A'))
['A', 'I']
sage: A.predecessors(A.state('F'), valid_input=['b', 'a'])
['F', 'C', 'A', 'I']
sage: A.predecessors(A.state('F'), valid_input=[['c', 'b'], 'a'])
['F', 'C', 'B']
```

```
>>> from sage.all import *
>>> A = Transducer([('I', 'A', 'a', 'b'), ('I', 'B', 'b', 'c'),
...                 ('I', 'C', 'c', 'a'), ('A', 'F', 'b', 'a'),
...                 ('B', 'F', ['c', 'b'], 'b'), ('C', 'F', 'a', 'c')],
...                initial_states=['I'], final_states=['F'])
>>> A.predecessors(A.state('A'))
['A', 'I']
>>> A.predecessors(A.state('F'), valid_input=['b', 'a'])
['F', 'C', 'A', 'I']
>>> A.predecessors(A.state('F'), valid_input=[['c', 'b'], 'a'])
['F', 'C', 'B']
```

prepone_output()

For all paths, shift the output of the path from one transition to the earliest possible preceding transition of the path.

OUTPUT: nothing

Apply the following to each state s (except initial states) of the finite state machine as often as possible:

If the letter a is a prefix of the output label of all transitions from s (including the final output of s), then remove it from all these labels and append it to all output labels of all transitions leading to s.

We assume that the states have no output labels, but final outputs are allowed.

EXAMPLES:

```
sage: A = Transducer([('A', 'B', 1, 1),
....:                 ('B', 'B', 0, 0),
....:                 ('B', 'C', 1, 0)],
....:                initial_states=['A'],
....:                final_states=['C'])
sage: A.prepone_output()
sage: A.transitions()
[Transition from 'A' to 'B': 1|1,0,
 Transition from 'B' to 'B': 0|0,
 Transition from 'B' to 'C': 1|-]
```

```
>>> from sage.all import *
>>> A = Transducer([('A', 'B', Integer(1), Integer(1)),
...                 ('B', 'B', Integer(0), Integer(0)),
...                 ('B', 'C', Integer(1), Integer(0))],
...                initial_states=['A'],
...                final_states=['C'])
>>> A.prepone_output()
>>> A.transitions()
[Transition from 'A' to 'B': 1|1,0,
 Transition from 'B' to 'B': 0|0,
 Transition from 'B' to 'C': 1|-]
```

```
sage: B = Transducer([('A', 'B', 0, 1),
....:                 ('B', 'C', 1, [1, 1]),
....:                 ('B', 'C', 0, 1)],
....:                initial_states=['A'],
....:                final_states=['C'])
sage: B.prepone_output()
sage: B.transitions()
[Transition from 'A' to 'B': 0|1,1,
 Transition from 'B' to 'C': 1|1,
 Transition from 'B' to 'C': 0|-]
```

```
>>> from sage.all import *
>>> B = Transducer([('A', 'B', Integer(0), Integer(1)),
...                 ('B', 'C', Integer(1), [Integer(1), Integer(1)]),
...                 ('B', 'C', Integer(0), Integer(1))],
...                initial_states=['A'],
...                final_states=['C'])
>>> B.prepone_output()
>>> B.transitions()
[Transition from 'A' to 'B': 0|1,1,
```

(continues on next page)

(continued from previous page)

```
 Transition from 'B' to 'C': 1|1,
 Transition from 'B' to 'C': 0|-]
```

If initial states are not labeled as such, unexpected results may be obtained:

```
sage: C = Transducer([(0,1,0,0)])
sage: C.prepone_output()
verbose 0 (...: finite_state_machine.py, prepone_output)
All transitions leaving state 0 have an output label with
prefix 0.  However, there is no inbound transition and it
is not an initial state. This routine (possibly called by
simplification) therefore erased this prefix from all
outbound transitions.
sage: C.transitions()
[Transition from 0 to 1: 0|-]
```

```
>>> from sage.all import *
>>> C = Transducer([(Integer(0),Integer(1),Integer(0),Integer(0))])
>>> C.prepone_output()
verbose 0 (...: finite_state_machine.py, prepone_output)
All transitions leaving state 0 have an output label with
prefix 0.  However, there is no inbound transition and it
is not an initial state. This routine (possibly called by
simplification) therefore erased this prefix from all
outbound transitions.
>>> C.transitions()
[Transition from 0 to 1: 0|-]
```

Also the final output of final states can be changed:

```
sage: T = Transducer([('A', 'B', 0, 1),
....:                  ('B', 'C', 1, [1, 1]),
....:                  ('B', 'C', 0, 1)],
....:                 initial_states=['A'],
....:                 final_states=['B'])
sage: T.state('B').final_word_out = [1]
sage: T.prepone_output()
sage: T.transitions()
[Transition from 'A' to 'B': 0|1,1,
 Transition from 'B' to 'C': 1|1,
 Transition from 'B' to 'C': 0|-]
sage: T.state('B').final_word_out
[]
```

```
>>> from sage.all import *
>>> T = Transducer([('A', 'B', Integer(0), Integer(1)),
...                  ('B', 'C', Integer(1), [Integer(1), Integer(1)]),
...                  ('B', 'C', Integer(0), Integer(1))],
...                 initial_states=['A'],
...                 final_states=['B'])
>>> T.state('B').final_word_out = [Integer(1)]
>>> T.prepone_output()
```

(continues on next page)

(continued from previous page)

```
>>> T.transitions()
[Transition from 'A' to 'B': 0|1,1,
 Transition from 'B' to 'C': 1|1,
 Transition from 'B' to 'C': 0|-]
>>> T.state('B').final_word_out
[]
```

```
sage: S = Transducer([('A', 'B', 0, 1),
....:                  ('B', 'C', 1, [1, 1]),
....:                  ('B', 'C', 0, 1)],
....:                 initial_states=['A'],
....:                 final_states=['B'])
sage: S.state('B').final_word_out = [0]
sage: S.prepone_output()
sage: S.transitions()
[Transition from 'A' to 'B': 0|1,
 Transition from 'B' to 'C': 1|1,1,
 Transition from 'B' to 'C': 0|1]
sage: S.state('B').final_word_out
[0]
```

```
>>> from sage.all import *
>>> S = Transducer([('A', 'B', Integer(0), Integer(1)),
...                  ('B', 'C', Integer(1), [Integer(1), Integer(1)]),
...                  ('B', 'C', Integer(0), Integer(1))],
...                 initial_states=['A'],
...                 final_states=['B'])
>>> S.state('B').final_word_out = [Integer(0)]
>>> S.prepone_output()
>>> S.transitions()
[Transition from 'A' to 'B': 0|1,
 Transition from 'B' to 'C': 1|1,1,
 Transition from 'B' to 'C': 0|1]
>>> S.state('B').final_word_out
[0]
```

Output labels do not have to be hashable:

```
sage: C = Transducer([(0, 1, 0, []),
....:                  (1, 0, 0, [vector([0, 0]), 0]),
....:                  (1, 1, 1, [vector([0, 0]), 1]),
....:                  (0, 0, 1, 0)],
....:                  determine_alphabets=False,
....:                  initial_states=[0])
sage: C.prepone_output()
sage: sorted(C.transitions())
[Transition from 0 to 1: 0|(0, 0),
 Transition from 0 to 0: 1|0,
 Transition from 1 to 0: 0|0,
 Transition from 1 to 1: 1|1,(0, 0)]
```

```
>>> from sage.all import *
>>> C = Transducer([(Integer(0), Integer(1), Integer(0), []),
...                 (Integer(1), Integer(0), Integer(0), [vector([Integer(0),
↪Integer(0)]), Integer(0)]),
...                 (Integer(1), Integer(1), Integer(1), [vector([Integer(0),
↪Integer(0)]), Integer(1)]),
...                 (Integer(0), Integer(0), Integer(1), Integer(0))],
...                determine_alphabets=False,
...                initial_states=[Integer(0)])
>>> C.prepone_output()
>>> sorted(C.transitions())
[Transition from 0 to 1: 0|(0, 0),
 Transition from 0 to 0: 1|0,
 Transition from 1 to 0: 0|0,
 Transition from 1 to 1: 1|1,(0, 0)]
```

process(**args*, ***kwargs*)

Return whether the finite state machine accepts the input, the state where the computation stops and which output is generated.

INPUT:

- `input_tape` – the input tape can be a list or an iterable with entries from the input alphabet. If we are working with a multi-tape machine (see parameter `use_multitape_input` and notes below), then the tape is a list or tuple of tracks, each of which can be a list or an iterable with entries from the input alphabet.
- `initial_state` or `initial_states` – the initial state(s) in which the machine starts. Either specify a single one with `initial_state` or a list of them with `initial_states`. If both are given, `initial_state` will be appended to `initial_states`. If neither is specified, the initial states of the finite state machine are taken.
- `list_of_outputs` – (default: `None`) a boolean or `None`. If `True`, then the outputs are given in list form (even if we have no or only one single output). If `False`, then the result is never a list (an exception is raised if the result cannot be returned). If `list_of_outputs=None`, the method determines automatically what to do (e.g. if a non-deterministic machine returns more than one path, then the output is returned in list form).
- `only_accepted` – boolean (default: `False`); if set, then the first argument in the output is guaranteed to be `True` (if the output is a list, then the first argument of each element will be `True`)
- `always_include_output` – if set (not by default), always include the output. This is inconsequential for a `FiniteStateMachine`, but can be used in derived classes where the output is suppressed by default, cf. `Automaton.process()`.
- `format_output` – a function that translates the written output (which is in form of a list) to something more readable. By default (`None`) identity is used here.
- `check_epsilon_transitions` – boolean (default: `True`); if `False`, then epsilon transitions are not taken into consideration during process
- `write_final_word_out` – boolean (default: `True`); whether the final output words should be written or not
- `use_multitape_input` – boolean (default: `False`); if `True`, then the multi-tape mode of the process iterator is activated. See also the notes below for multi-tape machines.
- `process_all_prefixes_of_input` – boolean (default: `False`); if `True`, then each prefix of the input word is processed (instead of processing the whole input word at once). Consequently, there is an

output generated for each of these prefixes.

- process_iterator_class – (default: None) a class inherited from *FSMProcessIterator*. If None, then *FSMProcessIterator* is taken. An instance of this class is created and is used during the processing.
- automatic_output_type – boolean (default: False); if set and the input has a parent, then the output will have the same parent. If the input does not have a parent, then the output will be of the same type as the input.

OUTPUT:

A triple (or a list of triples, cf. parameter list_of_outputs), where

- the first entry is True if the input string is accepted,
- the second gives the reached state after processing the input tape (This is a state with label None if the input could not be processed, i.e., if at one point no transition to go on could be found.), and
- the third gives a list of the output labels written during processing (in the case the finite state machine runs as transducer).

Note that in the case the finite state machine is not deterministic, all possible paths are taken into account.

This function uses an iterator which, in its simplest form, goes from one state to another in each step. To decide which way to go, it uses the input words of the outgoing transitions and compares them to the input tape. More precisely, in each step, the iterator takes an outgoing transition of the current state, whose input label equals the input letter of the tape. The output label of the transition, if present, is written on the output tape.

If the choice of the outgoing transition is not unique (i.e., we have a non-deterministic finite state machine), all possibilities are followed. This is done by splitting the process into several branches, one for each of the possible outgoing transitions.

The process (iteration) stops if all branches are finished, i.e., for no branch, there is any transition whose input word coincides with the processed input tape. This can simply happen when the entire tape was read.

Also see *__call__()* for a version of *process()* with shortened output.

Internally this function creates and works with an instance of *FSMProcessIterator*. This iterator can also be obtained with *iter_process()*.

If working with multi-tape finite state machines, all input words of transitions are words of k-tuples of letters. Moreover, the input tape has to consist of k tracks, i.e., be a list or tuple of k iterators, one for each track.

Warning

Working with multi-tape finite state machines is still experimental and can lead to wrong outputs.

EXAMPLES:

```
sage: binary_inverter = FiniteStateMachine({'A': [('A', 0, 1), ('A', 1, 0)]},
....:                                      initial_states=['A'], final_
↪states=['A'])
sage: binary_inverter.process([0, 1, 0, 0, 1, 1])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

```
>>> from sage.all import *
>>> binary_inverter = FiniteStateMachine({'A': [('A', Integer(0), Integer(1)),
```

(continues on next page)

(continued from previous page)

```
↪ ('A', Integer(1), Integer(0))]},
...                                     initial_states=['A'], final_states=[
↪'A'])
>>> binary_inverter.process([Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1)])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

Alternatively, we can invoke this function by:

```
sage: binary_inverter([0, 1, 0, 0, 1, 1])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

```
>>> from sage.all import *
>>> binary_inverter([Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1)])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

Below we construct a finite state machine which tests if an input is a non-adjacent form, i.e., no two neighboring letters are both nonzero (see also the example on *non-adjacent forms* in the documentation of the module *Finite state machines, automata, transducers*):

```
sage: NAF = FiniteStateMachine(
....:     {'_': [('_', 0), (1, 1)], 1: [('_', 0)]},
....:     initial_states=['_'], final_states=['_', 1])
sage: [NAF.process(w)[0] for w in [[0], [0, 1], [1, 1], [0, 1, 0, 1],
....:                              [0, 1, 1, 1, 0], [1, 0, 0, 1, 1]]]
[True, True, False, True, False, False]
```

```
>>> from sage.all import *
>>> NAF = FiniteStateMachine(
...     {'_': [('_', Integer(0)), (Integer(1), Integer(1))], Integer(1): [('_
↪', Integer(0))]},
...     initial_states=['_'], final_states=['_', Integer(1)])
>>> [NAF.process(w)[Integer(0)] for w in [[Integer(0)], [Integer(0),␣
↪Integer(1)], [Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(0),␣
↪Integer(1)],
...                              [Integer(0), Integer(1), Integer(1), Integer(1),
↪ Integer(0)], [Integer(1), Integer(0), Integer(0), Integer(1), Integer(1)]]]
[True, True, False, True, False, False]
```

Working only with the first component (i.e., returning whether accepted or not) usually corresponds to using the more specialized class `Automaton`.

Non-deterministic finite state machines can be handled as well.

```
sage: T = Transducer([(0, 1, 0, 0), (0, 2, 0, 0)],
....:     initial_states=[0])
sage: T.process([0])
[(False, 1, [0]), (False, 2, [0])]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(0), Integer(0)),␣
```

(continues on next page)

(continued from previous page)

```
↪(Integer(0), Integer(2), Integer(0), Integer(0))],
...     initial_states=[Integer(0)])
>>> T.process([Integer(0)])
[(False, 1, [0]), (False, 2, [0])]
```

Here is another non-deterministic finite state machine. Note that we use `format_output` (see *FSMProcessIterator*) to convert the written outputs (all characters) to strings.

```
sage: T = Transducer([(0, 1, [0, 0], 'a'), (0, 2, [0, 0, 1], 'b'),
....:                  (0, 1, 1, 'c'), (1, 0, [], 'd'),
....:                  (1, 1, 1, 'e')],
....:                 initial_states=[0], final_states=[0, 1])
sage: T.process([0], format_output=lambda o: ''.join(o))
(False, None, None)
sage: T.process([0, 0], format_output=lambda o: ''.join(o))
[(True, 0, 'ad'), (True, 1, 'a')]
sage: T.process([1], format_output=lambda o: ''.join(o))
[(True, 0, 'cd'), (True, 1, 'c')]
sage: T.process([1, 1], format_output=lambda o: ''.join(o))
[(True, 0, 'cdcd'), (True, 0, 'ced'),
 (True, 1, 'cdc'), (True, 1, 'ce')]
sage: T.process([0, 0, 1], format_output=lambda o: ''.join(o))
[(False, 2, 'b'),
 (True, 0, 'adcd'),
 (True, 0, 'aed'),
 (True, 1, 'adc'),
 (True, 1, 'ae')]
sage: T.process([0, 0, 1], format_output=lambda o: ''.join(o),
....:           only_accepted=True)
[(True, 0, 'adcd'), (True, 0, 'aed'),
 (True, 1, 'adc'), (True, 1, 'ae')]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), [Integer(0), Integer(0)], 'a'),␣
↪(Integer(0), Integer(2), [Integer(0), Integer(0), Integer(1)], 'b'),
...                  (Integer(0), Integer(1), Integer(1), 'c'), (Integer(1),␣
↪Integer(0), [], 'd'),
...                  (Integer(1), Integer(1), Integer(1), 'e')],
...                 initial_states=[Integer(0)], final_states=[Integer(0),␣
↪Integer(1)])
>>> T.process([Integer(0)], format_output=lambda o: ''.join(o))
(False, None, None)
>>> T.process([Integer(0), Integer(0)], format_output=lambda o: ''.join(o))
[(True, 0, 'ad'), (True, 1, 'a')]
>>> T.process([Integer(1)], format_output=lambda o: ''.join(o))
[(True, 0, 'cd'), (True, 1, 'c')]
>>> T.process([Integer(1), Integer(1)], format_output=lambda o: ''.join(o))
[(True, 0, 'cdcd'), (True, 0, 'ced'),
 (True, 1, 'cdc'), (True, 1, 'ce')]
>>> T.process([Integer(0), Integer(0), Integer(1)], format_output=lambda o: '
↪'.join(o))
[(False, 2, 'b'),
```

(continues on next page)

(continued from previous page)

```
 (True, 0, 'adcd'),
 (True, 0, 'aed'),
 (True, 1, 'adc'),
 (True, 1, 'ae')]
>>> T.process([Integer(0), Integer(0), Integer(1)], format_output=lambda o: '
↪'.join(o),
...             only_accepted=True)
[(True, 0, 'adcd'), (True, 0, 'aed'),
 (True, 1, 'adc'), (True, 1, 'ae')]
```

A simple example of a multi-tape finite state machine is the following: It writes the length of the first tape many letters `a` and then the length of the second tape many letters `b`:

```
sage: M = FiniteStateMachine([(0, 0, (1, None), 'a'),
....:                         (0, 1, [], []),
....:                         (1, 1, (None, 1), 'b')],
....:                        initial_states=[0],
....:                        final_states=[1])
sage: M.process(([1, 1], [1]), use_multitape_input=True)
(True, 1, ['a', 'a', 'b'])
```

```
>>> from sage.all import *
>>> M = FiniteStateMachine([(Integer(0), Integer(0), (Integer(1), None), 'a'),
...                         (Integer(0), Integer(1), [], []),
...                         (Integer(1), Integer(1), (None, Integer(1)), 'b
↪')],
...                        initial_states=[Integer(0)],
...                        final_states=[Integer(1)])
>>> M.process(([Integer(1), Integer(1)], [Integer(1)]), use_multitape_
↪input=True)
(True, 1, ['a', 'a', 'b'])
```

See also

`Automaton.process()`, *`Transducer.process()`*, *`iter_process()`*, *`__call__()`*, *`FSMProcessIterator`*.

`product_FiniteStateMachine`(*other*, *function*, *new_input_alphabet=None*, *only_accessible_components=True*, *final_function=None*, *new_class=None*)

Return a new finite state machine whose states are d-tuples of states of the original finite state machines.

INPUT:

- `other` – a finite state machine (for $d = 2$) or a list (or iterable) of $d - 1$ finite state machines
- `function` has to accept d transitions from A_j to B_j for $j \in \{1, \dots, d\}$ and returns a pair `(word_in, word_out)` which is the label of the transition $A = (A_1, \dots, A_d)$ to $B = (B_1, \dots, B_d)$. If there is no transition from A to B, then `function` should raise a `LookupError`.
- `new_input_alphabet` – (optional) the new input alphabet as a list
- `only_accessible_components` – if `True` (default), then the result is piped through *`accessible_components()`*. If no `new_input_alphabet` is given, it is determined by *`determine_alphabets()`*.

- `final_function` – a function mapping d final states of the original finite state machines to the final output of the corresponding state in the new finite state machine. By default, the final output is the empty word if both final outputs of the constituent states are empty; otherwise, a `ValueError` is raised.
- `new_class` – class of the new finite state machine. By default (`None`), the class of `self` is used

OUTPUT:

A finite state machine whose states are d-tuples of states of the original finite state machines. A state is initial or final if all constituent states are initial or final, respectively.

The labels of the transitions are defined by `function`.

The final output of a final state is determined by calling `final_function` on the constituent states.

The color of a new state is the tuple of colors of the constituent states of `self` and `other`. However, if all constituent states have color `None`, then the state has color `None`, too.

EXAMPLES:

```
sage: F = Automaton([('A', 'B', 1), ('A', 'A', 0), ('B', 'A', 2)],
....:               initial_states=['A'], final_states=['B'],
....:               determine_alphabets=True)
sage: G = Automaton([(1, 1, 1)], initial_states=[1], final_states=[1])
sage: def addition(transition1, transition2):
....:     return (transition1.word_in[0] + transition2.word_in[0],
....:             None)
sage: H = F.product_FiniteStateMachine(G, addition, [0, 1, 2, 3], only_
↪accessible_components=False)
sage: H.transitions()
[Transition from ('A', 1) to ('B', 1): 2|-,
 Transition from ('A', 1) to ('A', 1): 1|-,
 Transition from ('B', 1) to ('A', 1): 3|-]
sage: [s.color for s in H.iter_states()]
[None, None]
sage: H1 = F.product_FiniteStateMachine(G, addition, [0, 1, 2, 3], only_
↪accessible_components=False)
sage: H1.states()[0].label()[0] is F.states()[0]
True
sage: H1.states()[0].label()[1] is G.states()[0]
True
```

```
>>> from sage.all import *
>>> F = Automaton([('A', 'B', Integer(1)), ('A', 'A', Integer(0)), ('B', 'A',
↪Integer(2))],
...               initial_states=['A'], final_states=['B'],
...               determine_alphabets=True)
>>> G = Automaton([(Integer(1), Integer(1), Integer(1))], initial_
↪states=[Integer(1)], final_states=[Integer(1)])
>>> def addition(transition1, transition2):
...     return (transition1.word_in[Integer(0)] + transition2.word_
↪in[Integer(0)],
...             None)
>>> H = F.product_FiniteStateMachine(G, addition, [Integer(0), Integer(1),
↪Integer(2), Integer(3)], only_accessible_components=False)
>>> H.transitions()
[Transition from ('A', 1) to ('B', 1): 2|-,
```

(continues on next page)

(continued from previous page)

```
 Transition from ('A', 1) to ('A', 1): 1|-,
 Transition from ('B', 1) to ('A', 1): 3|-]
>>> [s.color for s in H.iter_states()]
[None, None]
>>> H1 = F.product_FiniteStateMachine(G, addition, [Integer(0), Integer(1),␣
↪Integer(2), Integer(3)], only_accessible_components=False)
>>> H1.states()[Integer(0)].label()[Integer(0)] is F.states()[Integer(0)]
True
>>> H1.states()[Integer(0)].label()[Integer(1)] is G.states()[Integer(0)]
True
```

```
sage: F = Automaton([(0,1,1/4), (0,0,3/4), (1,1,3/4), (1,0,1/4)],
....:               initial_states=[0] )
sage: G = Automaton([(0,0,1), (1,1,3/4), (1,0,1/4)],
....:               initial_states=[0] )
sage: H = F.product_FiniteStateMachine(
....:         G, lambda t1,t2: (t1.word_in[0]*t2.word_in[0], None))
sage: H.states()
[(0, 0), (1, 0)]
```

```
>>> from sage.all import *
>>> F = Automaton([(Integer(0),Integer(1),Integer(1)/Integer(4)), (Integer(0),
↪Integer(0),Integer(3)/Integer(4)), (Integer(1),Integer(1),Integer(3)/
↪Integer(4)), (Integer(1),Integer(0),Integer(1)/Integer(4))],
...               initial_states=[Integer(0)] )
>>> G = Automaton([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(1),
↪Integer(3)/Integer(4)), (Integer(1),Integer(0),Integer(1)/Integer(4))],
...               initial_states=[Integer(0)] )
>>> H = F.product_FiniteStateMachine(
...         G, lambda t1,t2: (t1.word_in[Integer(0)]*t2.word_in[Integer(0)],␣
↪None))
>>> H.states()
[(0, 0), (1, 0)]
```

```
sage: F = Automaton([(0,1,1/4), (0,0,3/4), (1,1,3/4), (1,0,1/4)],
....:               initial_states=[0] )
sage: G = Automaton([(0,0,1), (1,1,3/4), (1,0,1/4)],
....:               initial_states=[0] )
sage: H = F.product_FiniteStateMachine(G,
....:                                  lambda t1,t2: (t1.word_in[0]*t2.word_
↪in[0], None),
....:                                  only_accessible_components=False)
sage: H.states()
[(0, 0), (1, 0), (0, 1), (1, 1)]
```

```
>>> from sage.all import *
>>> F = Automaton([(Integer(0),Integer(1),Integer(1)/Integer(4)), (Integer(0),
↪Integer(0),Integer(3)/Integer(4)), (Integer(1),Integer(1),Integer(3)/
↪Integer(4)), (Integer(1),Integer(0),Integer(1)/Integer(4))],
...               initial_states=[Integer(0)] )
>>> G = Automaton([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)/Integer(4)), (Integer(1),Integer(0),Integer(1)/Integer(4))],
...                     initial_states=[Integer(0)] )
>>> H = F.product_FiniteStateMachine(G,
...                                   lambda t1,t2: (t1.word_in[Integer(0)]*t2.
↪word_in[Integer(0)], None),
...                                   only_accessible_components=False)
>>> H.states()
[(0, 0), (1, 0), (0, 1), (1, 1)]
```

Also final output words are considered according to the function `final_function`:

```
sage: F = Transducer([(0, 1, 0, 1), (1, 1, 1, 1), (1, 1, 0, 1)],
....:                final_states=[1])
sage: F.state(1).final_word_out = 1
sage: G = Transducer([(0, 0, 0, 1), (0, 0, 1, 0)], final_states=[0])
sage: G.state(0).final_word_out = 1
sage: def minus(t1, t2):
....:     return (t1.word_in[0] - t2.word_in[0],
....:                t1.word_out[0] - t2.word_out[0])
sage: H = F.product_FiniteStateMachine(G, minus)
Traceback (most recent call last):
...
ValueError: A final function must be given.
sage: def plus(s1, s2):
....:     return s1.final_word_out[0] + s2.final_word_out[0]
sage: H = F.product_FiniteStateMachine(G, minus,
....:                                  final_function=plus)
sage: H.final_states()
[(1, 0)]
sage: H.final_states()[0].final_word_out
[2]
```

```
>>> from sage.all import *
>>> F = Transducer([(Integer(0), Integer(1), Integer(0), Integer(1)),␣
↪(Integer(1), Integer(1), Integer(1), Integer(1)), (Integer(1), Integer(1),␣
↪Integer(0), Integer(1))],
...                     final_states=[Integer(1)])
>>> F.state(Integer(1)).final_word_out = Integer(1)
>>> G = Transducer([(Integer(0), Integer(0), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(0), Integer(1), Integer(0))], final_
↪states=[Integer(0)])
>>> G.state(Integer(0)).final_word_out = Integer(1)
>>> def minus(t1, t2):
...     return (t1.word_in[Integer(0)] - t2.word_in[Integer(0)],
...                t1.word_out[Integer(0)] - t2.word_out[Integer(0)])
>>> H = F.product_FiniteStateMachine(G, minus)
Traceback (most recent call last):
...
ValueError: A final function must be given.
>>> def plus(s1, s2):
...     return s1.final_word_out[Integer(0)] + s2.final_word_out[Integer(0)]
>>> H = F.product_FiniteStateMachine(G, minus,
```

(continues on next page)

(continued from previous page)

```
...                                          final_function=plus)
>>> H.final_states()
[(1, 0)]
>>> H.final_states()[Integer(0)].final_word_out
[2]
```

Products of more than two finite state machines are also possible:

```
sage: def plus(s1, s2, s3):
....:     if s1.word_in == s2.word_in == s3.word_in:
....:          return (s1.word_in,
....:                  sum(s.word_out[0] for s in (s1, s2, s3)))
....:     else:
....:         raise LookupError
sage: T0 = transducers.CountSubblockOccurrences([0, 0], [0, 1, 2])
sage: T1 = transducers.CountSubblockOccurrences([1, 1], [0, 1, 2])
sage: T2 = transducers.CountSubblockOccurrences([2, 2], [0, 1, 2])
sage: T = T0.product_FiniteStateMachine([T1, T2], plus)
sage: T.transitions()
[Transition from ((), (), ()) to ((0,), (), ()): 0|0,
 Transition from ((), (), ()) to ((), (1,), ()): 1|0,
 Transition from ((), (), ()) to ((), (), (2,)): 2|0,
 Transition from ((0,), (), ()) to ((0,), (), ()): 0|1,
 Transition from ((0,), (), ()) to ((), (1,), ()): 1|0,
 Transition from ((0,), (), ()) to ((), (), (2,)): 2|0,
 Transition from ((), (1,), ()) to ((0,), (), ()): 0|0,
 Transition from ((), (1,), ()) to ((), (1,), ()): 1|1,
 Transition from ((), (1,), ()) to ((), (), (2,)): 2|0,
 Transition from ((), (), (2,)) to ((0,), (), ()): 0|0,
 Transition from ((), (), (2,)) to ((), (1,), ()): 1|0,
 Transition from ((), (), (2,)) to ((), (), (2,)): 2|1]
sage: T([0, 0, 1, 1, 2, 2, 0, 1, 2, 2])
[0, 1, 0, 1, 0, 1, 0, 0, 0, 1]
```

```
>>> from sage.all import *
>>> def plus(s1, s2, s3):
...     if s1.word_in == s2.word_in == s3.word_in:
...          return (s1.word_in,
...                  sum(s.word_out[Integer(0)] for s in (s1, s2, s3)))
...     else:
...         raise LookupError
>>> T0 = transducers.CountSubblockOccurrences([Integer(0), Integer(0)],␣
↪[Integer(0), Integer(1), Integer(2)])
>>> T1 = transducers.CountSubblockOccurrences([Integer(1), Integer(1)],␣
↪[Integer(0), Integer(1), Integer(2)])
>>> T2 = transducers.CountSubblockOccurrences([Integer(2), Integer(2)],␣
↪[Integer(0), Integer(1), Integer(2)])
>>> T = T0.product_FiniteStateMachine([T1, T2], plus)
>>> T.transitions()
[Transition from ((), (), ()) to ((0,), (), ()): 0|0,
 Transition from ((), (), ()) to ((), (1,), ()): 1|0,
 Transition from ((), (), ()) to ((), (), (2,)): 2|0,
```

(continues on next page)

(continued from previous page)

```
 Transition from ((0,), (), ()) to ((0,), (), ()): 0|1,
 Transition from ((0,), (), ()) to ((), (1,), ()): 1|0,
 Transition from ((0,), (), ()) to ((), (), (2,)): 2|0,
 Transition from ((), (1,), ()) to ((0,), (), ()): 0|0,
 Transition from ((), (1,), ()) to ((), (1,), ()): 1|1,
 Transition from ((), (1,), ()) to ((), (), (2,)): 2|0,
 Transition from ((), (), (2,)) to ((0,), (), ()): 0|0,
 Transition from ((), (), (2,)) to ((), (1,), ()): 1|0,
 Transition from ((), (), (2,)) to ((), (), (2,)): 2|1]
>>> T([Integer(0), Integer(0), Integer(1), Integer(1), Integer(2), Integer(2),
↪ Integer(0), Integer(1), Integer(2), Integer(2)])
[0, 1, 0, 1, 0, 1, 0, 0, 0, 1]
```

`other` can also be an iterable:

```
sage: T == T0.product_FiniteStateMachine(iter([T1, T2]), plus)
True
```

```
>>> from sage.all import *
>>> T == T0.product_FiniteStateMachine(iter([T1, T2]), plus)
True
```

projection(*what='input'*)

Return an Automaton which transition labels are the projection of the transition labels of the input.

INPUT:

- `what` – (default: `input`) either `input` or `output`

OUTPUT: an automaton

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'B', 0, 1), ('A', 'A', 1, 1),
....:                         ('B', 'B', 1, 0)])
sage: G = F.projection(what='output')
sage: G.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 0|-]
```

```
>>> from sage.all import *
>>> F = FiniteStateMachine([('A', 'B', Integer(0), Integer(1)), ('A', 'A',␣
↪Integer(1), Integer(1)),
...                         ('B', 'B', Integer(1), Integer(0))])
>>> G = F.projection(what='output')
>>> G.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 0|-]
```

quotient(*classes*)

Construct the quotient with respect to the equivalence classes.

INPUT:

- `classes` is a list of equivalence classes of states.

OUTPUT: a finite state machine

The labels of the new states are tuples of states of the `self`, corresponding to `classes`.

Assume that c is a class, and a and b are states in c. Then there is a bijection φ between the transitions from a and the transitions from b with the following properties: if $\varphi(t_a) = t_b$, then

- $t_a.word_{in} = t_b.word_{in}$,
- $t_a.word_{out} = t_b.word_{out}$, and
- t_a and t_b lead to some equivalent states a' and b'.

Non-initial states may be merged with initial states, the resulting state is an initial state.

All states in a class must have the same `is_final`, `final_word_out` and `word_out` values.

EXAMPLES:

```
sage: fsm = FiniteStateMachine([("A", "B", 0, 1), ("A", "B", 1, 0),
....:                           ("B", "C", 0, 0), ("B", "C", 1, 1),
....:                           ("C", "D", 0, 1), ("C", "D", 1, 0),
....:                           ("D", "A", 0, 0), ("D", "A", 1, 1)])
sage: fsmq = fsm.quotient([[fsm.state("A"), fsm.state("C")],
....:                      [fsm.state("B"), fsm.state("D")]])
sage: fsmq.transitions()
[Transition from ('A', 'C')
             to ('B', 'D'): 0|1,
 Transition from ('A', 'C')
             to ('B', 'D'): 1|0,
 Transition from ('B', 'D')
             to ('A', 'C'): 0|0,
 Transition from ('B', 'D')
             to ('A', 'C'): 1|1]
sage: fsmq.relabeled().transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|0,
 Transition from 1 to 0: 1|1]
sage: fsmq1 = fsm.quotient(fsm.equivalence_classes())
sage: fsmq1 == fsmq
True
sage: fsm.quotient([[fsm.state("A"), fsm.state("B"), fsm.state("C"), fsm.
↪state("D")]])
Traceback (most recent call last):
    ...
AssertionError: Transitions of state 'A' and 'B' are incompatible.
```

```
>>> from sage.all import *
>>> fsm = FiniteStateMachine([("A", "B", Integer(0), Integer(1)), ("A", "B",␣
↪Integer(1), Integer(0)),
...                           ("B", "C", Integer(0), Integer(0)), ("B", "C",␣
↪Integer(1), Integer(1)),
...                           ("C", "D", Integer(0), Integer(1)), ("C", "D",␣
↪Integer(1), Integer(0)),
...                           ("D", "A", Integer(0), Integer(0)), ("D", "A",␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1), Integer(1))])
>>> fsmq = fsm.quotient([[fsm.state("A"), fsm.state("C")],
...                      [fsm.state("B"), fsm.state("D")]])
>>> fsmq.transitions()
[Transition from ('A', 'C')
              to ('B', 'D'): 0|1,
 Transition from ('A', 'C')
              to ('B', 'D'): 1|0,
 Transition from ('B', 'D')
              to ('A', 'C'): 0|0,
 Transition from ('B', 'D')
              to ('A', 'C'): 1|1]
>>> fsmq.relabeled().transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|0,
 Transition from 1 to 0: 1|1]
>>> fsmq1 = fsm.quotient(fsm.equivalence_classes())
>>> fsmq1 == fsmq
True
>>> fsm.quotient([[fsm.state("A"), fsm.state("B"), fsm.state("C"), fsm.state(
↪"D")]])
Traceback (most recent call last):
    ...
AssertionError: Transitions of state 'A' and 'B' are incompatible.
```

relabeled(*memo=None*, *labels=None*)

Return a deep copy of the finite state machine, but the states are relabeled.

INPUT:

- `memo` – (default: `None`) a dictionary storing already processed elements
- `labels` – (default: `None`) a dictionary or callable mapping old labels to new labels. If `None`, then the new labels are integers starting with 0.

OUTPUT: a new finite state machine

EXAMPLES:

```
sage: FSM1 = FiniteStateMachine([('A', 'B'), ('B', 'C'), ('C', 'A')])
sage: FSM1.states()
['A', 'B', 'C']
sage: FSM2 = FSM1.relabeled()
sage: FSM2.states()
[0, 1, 2]
sage: FSM3 = FSM1.relabeled(labels={'A': 'a', 'B': 'b', 'C': 'c'})
sage: FSM3.states()
['a', 'b', 'c']
sage: FSM4 = FSM2.relabeled(labels=lambda x: 2*x)
sage: FSM4.states()
[0, 2, 4]
```

```
>>> from sage.all import *
>>> FSM1 = FiniteStateMachine([('A', 'B'), ('B', 'C'), ('C', 'A')])
>>> FSM1.states()
['A', 'B', 'C']
>>> FSM2 = FSM1.relabeled()
>>> FSM2.states()
[0, 1, 2]
>>> FSM3 = FSM1.relabeled(labels={'A': 'a', 'B': 'b', 'C': 'c'})
>>> FSM3.states()
['a', 'b', 'c']
>>> FSM4 = FSM2.relabeled(labels=lambda x: Integer(2)*x)
>>> FSM4.states()
[0, 2, 4]
```

remove_epsilon_transitions()

set_coordinates(*coordinates*, *default=True*)

Set coordinates of the states for the LaTeX representation by a dictionary or a function mapping labels to coordinates.

INPUT:

- `coordinates` – dictionary or a function mapping labels of states to pairs interpreted as coordinates
- `default` – if `True`, then states not given by `coordinates` get a default position on a circle of radius 3.

OUTPUT: nothing

EXAMPLES:

```
sage: F = Automaton([[0, 1, 1], [1, 2, 2], [2, 0, 0]])
sage: F.set_coordinates({0: (0, 0), 1: (2, 0), 2: (1, 1)})
sage: F.state(0).coordinates
(0, 0)
```

```
>>> from sage.all import *
>>> F = Automaton([[Integer(0), Integer(1), Integer(1)], [Integer(1),␣
↪Integer(2), Integer(2)], [Integer(2), Integer(0), Integer(0)]])
>>> F.set_coordinates({Integer(0): (Integer(0), Integer(0)), Integer(1):␣
↪(Integer(2), Integer(0)), Integer(2): (Integer(1), Integer(1))})
>>> F.state(Integer(0)).coordinates
(0, 0)
```

We can also use a function to determine the coordinates:

```
sage: F = Automaton([[0, 1, 1], [1, 2, 2], [2, 0, 0]])
sage: F.set_coordinates(lambda l: (l, 3/(l+1)))
sage: F.state(2).coordinates
(2, 1)
```

```
>>> from sage.all import *
>>> F = Automaton([[Integer(0), Integer(1), Integer(1)], [Integer(1),␣
↪Integer(2), Integer(2)], [Integer(2), Integer(0), Integer(0)]])
>>> F.set_coordinates(lambda l: (l, Integer(3)/(l+Integer(1))))
```

(continues on next page)

(continued from previous page)

```
>>> F.state(Integer(2)).coordinates
(2, 1)
```

split_transitions()

Return a new transducer, where all transitions in `self` with input labels consisting of more than one letter are replaced by a path of the corresponding length.

OUTPUT: a new transducer

EXAMPLES:

```
sage: A = Transducer([('A', 'B', [1, 2, 3], 0)],
....:                initial_states=['A'], final_states=['B'])
sage: A.split_transitions().states()
[('A', ()), ('B', ()),
 ('A', (1,)), ('A', (1, 2))]
```

```
>>> from sage.all import *
>>> A = Transducer([('A', 'B', [Integer(1), Integer(2), Integer(3)],␣
↪Integer(0))],
...                initial_states=['A'], final_states=['B'])
>>> A.split_transitions().states()
[('A', ()), ('B', ()),
 ('A', (1,)), ('A', (1, 2))]
```

state(*state*)

Return the state of the finite state machine.

INPUT:

- `state` – if `state` is not an instance of *FSMState*, then it is assumed that it is the label of a state.

OUTPUT: the state of the finite state machine corresponding to `state`

If no state is found, then a `LookupError` is thrown.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: FSM = FiniteStateMachine([(A, 'B'), ('C', A)])
sage: FSM.state('A') == A
True
sage: FSM.state('xyz')
Traceback (most recent call last):
...
LookupError: No state with label xyz found.
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMState
>>> A = FSMState('A')
>>> FSM = FiniteStateMachine([(A, 'B'), ('C', A)])
>>> FSM.state('A') == A
True
>>> FSM.state('xyz')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
LookupError: No state with label xyz found.
```

states()

Return the states of the finite state machine.

OUTPUT: the states of the finite state machine as list

EXAMPLES:

```
sage: FSM = Automaton([('1', '2', 1), ('2', '2', 0)])
sage: FSM.states()
['1', '2']
```

```
>>> from sage.all import *
>>> FSM = Automaton([('1', '2', Integer(1)), ('2', '2', Integer(0))])
>>> FSM.states()
['1', '2']
```

transition(*transition*)

Return the transition of the finite state machine.

INPUT:

- `transition` – if `transition` is not an instance of `FSMTransition`, then it is assumed that it is a tuple `(from_state, to_state, word_in, word_out)`

OUTPUT: the transition of the finite state machine corresponding to `transition`

If no transition is found, then a `LookupError` is thrown.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t = FSMTransition('A', 'B', 0)
sage: F = FiniteStateMachine([t])
sage: F.transition(('A', 'B', 0))
Transition from 'A' to 'B': 0|-
sage: id(t) == id(F.transition(('A', 'B', 0)))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import FSMTransition
>>> t = FSMTransition('A', 'B', Integer(0))
>>> F = FiniteStateMachine([t])
>>> F.transition(('A', 'B', Integer(0)))
Transition from 'A' to 'B': 0|-
>>> id(t) == id(F.transition(('A', 'B', Integer(0))))
True
```

transitions(*from_state=None*)

Return a list of all transitions.

INPUT:

- `from_state` – (default: `None`) if `from_state` is given, then a list of transitions starting there is given

OUTPUT: list of all transitions

EXAMPLES:

```
sage: FSM = Automaton([('1', '2', 1), ('2', '2', 0)])
sage: FSM.transitions()
[Transition from '1' to '2': 1|-,
 Transition from '2' to '2': 0|-]
```

```
>>> from sage.all import *
>>> FSM = Automaton([('1', '2', Integer(1)), ('2', '2', Integer(0))])
>>> FSM.transitions()
[Transition from '1' to '2': 1|-,
 Transition from '2' to '2': 0|-]
```

transposition(*reverse_output_labels=True*)

Return a new finite state machine, where all transitions of the input finite state machine are reversed.

INPUT:

- `reverse_output_labels` – boolean (default: `True`); whether to reverse output labels

EXAMPLES:

```
sage: aut = Automaton([('A', 'A', 0), ('A', 'A', 1), ('A', 'B', 0)],
....:                 initial_states=['A'], final_states=['B'])
sage: aut.transposition().transitions('B')
[Transition from 'B' to 'A': 0|-]
```

```
>>> from sage.all import *
>>> aut = Automaton([('A', 'A', Integer(0)), ('A', 'A', Integer(1)), ('A', 'B
↪', Integer(0))],
...                 initial_states=['A'], final_states=['B'])
>>> aut.transposition().transitions('B')
[Transition from 'B' to 'A': 0|-]
```

```
sage: aut = Automaton([('1', '1', 1), ('1', '2', 0), ('2', '2', 0)],
....:                 initial_states=['1'], final_states=['1', '2'])
sage: aut.transposition().initial_states()
['1', '2']
```

```
>>> from sage.all import *
>>> aut = Automaton([('1', '1', Integer(1)), ('1', '2', Integer(0)), ('2', '2
↪', Integer(0))],
...                 initial_states=['1'], final_states=['1', '2'])
>>> aut.transposition().initial_states()
['1', '2']
```

```
sage: A = Automaton([(0, 1, [1, 0])],
....:     initial_states=[0],
....:     final_states=[1])
sage: A([1, 0])
```

(continues on next page)

(continued from previous page)

```
True
sage: A.transposition()([0, 1])
True
```

```
>>> from sage.all import *
>>> A = Automaton([(Integer(0), Integer(1), [Integer(1), Integer(0)])],
...     initial_states=[Integer(0)],
...     final_states=[Integer(1)])
>>> A([Integer(1), Integer(0)])
True
>>> A.transposition()([Integer(0), Integer(1)])
True
```

```
sage: T = Transducer([(0, 1, [1, 0], [1, 0])],
....:     initial_states=[0],
....:     final_states=[1])
sage: T([1, 0])
[1, 0]
sage: T.transposition()([0, 1])
[0, 1]
sage: T.transposition(reverse_output_labels=False)([0, 1])
[1, 0]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), [Integer(1), Integer(0)],
↪[Integer(1), Integer(0)])],
...     initial_states=[Integer(0)],
...     final_states=[Integer(1)])
>>> T([Integer(1), Integer(0)])
[1, 0]
>>> T.transposition()([Integer(0), Integer(1)])
[0, 1]
>>> T.transposition(reverse_output_labels=False)([Integer(0), Integer(1)])
[1, 0]
```

with_final_word_out (*letters*, *allow_non_final=True*)

Construct a new finite state machine with final output words for all states by implicitly reading trailing letters until a final state is reached.

INPUT:

- `letters` – either an element of the input alphabet or a list of such elements. This is repeated cyclically when needed.
- `allow_non_final` – boolean (default: `True`); whether we allow that some states may be non-final in the resulting finite state machine. I.e., if `False` then each state has to have a path to a final state with input label matching `letters`.

OUTPUT: a finite state machine

The inplace version of this function is *construct_final_word_out()*.

Suppose for the moment a single element `letter` as input for `letters`. This is equivalent to `letters = [letter]`. We will discuss the general case below.

Let `word_in` be a word over the input alphabet and assume that the original finite state machine transforms `word_in` to `word_out` reaching a possibly non-final state `s`. Let further k be the minimum number of letters `letter` such that there is a path from `s` to some final state `f` whose input label consists of k copies of `letter` and whose output label is `path_word_out`. Then the state `s` of the resulting finite state machine is a final state with final output `path_word_out + f.final_word_out`. Therefore, the new finite state machine transforms `word_in` to `word_out + path_word_out + f.final_word_out`.

This is e.g. useful for finite state machines operating on digit expansions: there, it is sometimes required to read a sufficient number of trailing zeros (at the most significant positions) in order to reach a final state and to flush all carries. In this case, this method constructs an essentially equivalent finite state machine in the sense that it not longer requires adding sufficiently many trailing zeros. However, it is the responsibility of the user to make sure that if adding trailing zeros to the input anyway, the output is equivalent.

If `letters` consists of more than one letter, then it is assumed that (not necessarily complete) cycles of `letters` are appended as trailing input.

See also

example on Gray code

EXAMPLES:

1. A simple transducer transforming 00 blocks to 01 blocks:

```
sage: T = Transducer([(0, 1, 0, 0), (1, 0, 0, 1)],
....:                initial_states=[0],
....:                final_states=[0])
sage: T.process([0, 0, 0])
(False, 1, [0, 1, 0])
sage: T.process([0, 0, 0, 0])
(True, 0, [0, 1, 0, 1])
sage: F = T.with_final_word_out(0)
sage: for f in F.iter_final_states():
....:     print("{} {}".format(f, f.final_word_out))
0 []
1 [1]
sage: F.process([0, 0, 0])
(True, 1, [0, 1, 0, 1])
sage: F.process([0, 0, 0, 0])
(True, 0, [0, 1, 0, 1])
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(0), Integer(0)),␣
↪(Integer(1), Integer(0), Integer(0), Integer(1))],
...                initial_states=[Integer(0)],
...                final_states=[Integer(0)])
>>> T.process([Integer(0), Integer(0), Integer(0)])
(False, 1, [0, 1, 0])
>>> T.process([Integer(0), Integer(0), Integer(0), Integer(0)])
(True, 0, [0, 1, 0, 1])
>>> F = T.with_final_word_out(Integer(0))
>>> for f in F.iter_final_states():
...     print("{} {}".format(f, f.final_word_out))
0 []
```

(continues on next page)

(continued from previous page)

```
1 [1]
>>> F.process([Integer(0), Integer(0), Integer(0)])
(True, 1, [0, 1, 0, 1])
>>> F.process([Integer(0), Integer(0), Integer(0), Integer(0)])
(True, 0, [0, 1, 0, 1])
```

2. A more realistic example: Addition of 1 in binary. We construct a transition function transforming the input to its binary expansion:

```
sage: def binary_transition(carry, input):
....:     value = carry + input
....:     if value.mod(2) == 0:
....:         return (value/2, 0)
....:     else:
....:         return ((value-1)/2, 1)
```

```
>>> from sage.all import *
>>> def binary_transition(carry, input):
...     value = carry + input
...     if value.mod(Integer(2)) == Integer(0):
...         return (value/Integer(2), Integer(0))
...     else:
...         return ((value-Integer(1))/Integer(2), Integer(1))
```

Now, we only have to start with a carry of 1 to get the required transducer:

```
sage: T = Transducer(binary_transition,
....:                input_alphabet=[0, 1],
....:                initial_states=[1],
....:                final_states=[0])
```

```
>>> from sage.all import *
>>> T = Transducer(binary_transition,
...                input_alphabet=[Integer(0), Integer(1)],
...                initial_states=[Integer(1)],
...                final_states=[Integer(0)])
```

We test this for the binary expansion of 7:

```
sage: T.process([1, 1, 1])
(False, 1, [0, 0, 0])
```

```
>>> from sage.all import *
>>> T.process([Integer(1), Integer(1), Integer(1)])
(False, 1, [0, 0, 0])
```

The final carry 1 has not be flushed yet, we have to add a trailing zero:

```
sage: T.process([1, 1, 1, 0])
(True, 0, [0, 0, 0, 1])
```

```
>>> from sage.all import *
>>> T.process([Integer(1), Integer(1), Integer(1), Integer(0)])
(True, 0, [0, 0, 0, 1])
```

We check that with this trailing zero, the transducer performs as advertised:

```
sage: all(ZZ(T(k.bits()+[0]), base=2) == k + 1
....:     for k in srange(16))
True
```

```
>>> from sage.all import *
>>> all(ZZ(T(k.bits()+[Integer(0)]), base=Integer(2)) == k + Integer(1)
...     for k in srange(Integer(16)))
True
```

However, most of the time, we produce superfluous trailing zeros:

```
sage: T(11.bits()+[0])
[0, 0, 1, 1, 0]
```

```
>>> from sage.all import *
>>> T(Integer(11).bits()+[Integer(0)])
[0, 0, 1, 1, 0]
```

We now use this method:

```
sage: F = T.with_final_word_out(0)
sage: for f in F.iter_final_states():
....:     print("{} {}".format(f, f.final_word_out))
1 [1]
0 []
```

```
>>> from sage.all import *
>>> F = T.with_final_word_out(Integer(0))
>>> for f in F.iter_final_states():
...     print("{} {}".format(f, f.final_word_out))
1 [1]
0 []
```

The same tests as above, but we do not have to pad with trailing zeros anymore:

```
sage: F.process([1, 1, 1])
(True, 1, [0, 0, 0, 1])
sage: all(ZZ(F(k.bits()), base=2) == k + 1
....:     for k in srange(16))
True
```

```
>>> from sage.all import *
>>> F.process([Integer(1), Integer(1), Integer(1)])
(True, 1, [0, 0, 0, 1])
>>> all(ZZ(F(k.bits()), base=Integer(2)) == k + Integer(1)
...     for k in srange(Integer(16)))
True
```

No more trailing zero in the output:

```
sage: F(11.bits())
[0, 0, 1, 1]
sage: all(F(k.bits())[-1] == 1
....:     for k in srange(16))
True
```

```
>>> from sage.all import *
>>> F(Integer(11).bits())
[0, 0, 1, 1]
>>> all(F(k.bits())[-Integer(1)] == Integer(1)
...     for k in srange(Integer(16)))
True
```

3. Here is an example, where we allow trailing repeated 10:

```
sage: T = Transducer([(0, 1, 0, 'a'),
....:                 (1, 2, 1, 'b'),
....:                 (2, 0, 0, 'c')],
....:                initial_states=[0],
....:                final_states=[0])
sage: F = T.with_final_word_out([1, 0])
sage: for f in F.iter_final_states():
....:     print(str(f) + ' ' + ''.join(f.final_word_out))
0
1 bc
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(0), 'a'),
...                 (Integer(1), Integer(2), Integer(1), 'b'),
...                 (Integer(2), Integer(0), Integer(0), 'c')],
...                initial_states=[Integer(0)],
...                final_states=[Integer(0)])
>>> F = T.with_final_word_out([Integer(1), Integer(0)])
>>> for f in F.iter_final_states():
...     print(str(f) + ' ' + ''.join(f.final_word_out))
0
1 bc
```

Trying this with trailing repeated 01 does not produce a `final_word_out` for state `1`, but for state `2`:

```
sage: F = T.with_final_word_out([0, 1])
sage: for f in F.iter_final_states():
....:     print(str(f) + ' ' + ''.join(f.final_word_out))
0
2 c
```

```
>>> from sage.all import *
>>> F = T.with_final_word_out([Integer(0), Integer(1)])
>>> for f in F.iter_final_states():
...     print(str(f) + ' ' + ''.join(f.final_word_out))
0
```

(continues on next page)

(continued from previous page)

```
2 c
```

4. Here another example with a more-letter trailing input:

```
sage: T = Transducer([(0, 1, 0, 'a'),
....:                  (1, 2, 0, 'b'), (1, 2, 1, 'b'),
....:                  (2, 3, 0, 'c'), (2, 0, 1, 'e'),
....:                  (3, 1, 0, 'd'), (3, 1, 1, 'd')],
....:                 initial_states=[0],
....:                 final_states=[0],
....:                 with_final_word_out=[0, 0, 1, 1])
sage: for f in T.iter_final_states():
....:     print(str(f) + ' ' + ''.join(f.final_word_out))
0
1 bcdbcdbe
2 cdbe
3 dbe
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), Integer(0), 'a'),
...                  (Integer(1), Integer(2), Integer(0), 'b'),␣
↪(Integer(1), Integer(2), Integer(1), 'b'),
...                  (Integer(2), Integer(3), Integer(0), 'c'),␣
↪(Integer(2), Integer(0), Integer(1), 'e'),
...                  (Integer(3), Integer(1), Integer(0), 'd'),␣
↪(Integer(3), Integer(1), Integer(1), 'd')],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(0)],
...                 with_final_word_out=[Integer(0), Integer(0),␣
↪Integer(1), Integer(1)])
>>> for f in T.iter_final_states():
...     print(str(f) + ' ' + ''.join(f.final_word_out))
0
1 bcdbcdbe
2 cdbe
3 dbe
```

class sage.combinat.finite_state_machine.**Transducer**(*data=None*, *initial_states=None*, *final_states=None*, *input_alphabet=None*, *output_alphabet=None*, *determine_alphabets=None*, *with_final_word_out=None*, *store_states_dict=True*, *on_duplicate_transition=None*)

Bases: `FiniteStateMachine`

This creates a transducer, which is a finite state machine, whose transitions have input and output labels.

A transducer has additional features like creating a simplified transducer.

See class `FiniteStateMachine` for more information.

EXAMPLES:

We can create a transducer performing the addition of 1 (for numbers given in binary and read from right to left) in the following way:

```
sage: T = Transducer([('C', 'C', 1, 0), ('C', 'N', 0, 1),
....:                 ('N', 'N', 0, 0), ('N', 'N', 1, 1)],
....:                initial_states=['C'], final_states=['N'])
sage: T
Transducer with 2 states
sage: T([0])
[1]
sage: T([1,1,0])
[0, 0, 1]
sage: ZZ(T(15.digits(base=2)+[0]), base=2)
16
```

```
>>> from sage.all import *
>>> T = Transducer([('C', 'C', Integer(1), Integer(0)), ('C', 'N', Integer(0),
↪Integer(1)),
...                 ('N', 'N', Integer(0), Integer(0)), ('N', 'N', Integer(1),
↪Integer(1))],
...                initial_states=['C'], final_states=['N'])
>>> T
Transducer with 2 states
>>> T([Integer(0)])
[1]
>>> T([Integer(1),Integer(1),Integer(0)])
[0, 0, 1]
>>> ZZ(T(Integer(15).digits(base=Integer(2))+[Integer(0)]), base=Integer(2))
16
```

Note that we have padded the binary input sequence by a 0 so that the transducer can reach its final state.

cartesian_product(*other*, *only_accessible_components=True*)

Return a new transducer which can simultaneously process an input with the machines `self` and `other` where the output labels are d-tuples of the original output labels.

INPUT:

- `other` – a finite state machine (if $d = 2$) or a list (or other iterable) of $d - 1$ finite state machines
- `only_accessible_components` – if `True` (default), then the result is piped through *accessible_components()*. If no `new_input_alphabet` is given, it is determined by *determine_alphabets()*.

OUTPUT:

A transducer which can simultaneously process an input with `self` and the machine(s) in `other`.

The set of states of the new transducer is the Cartesian product of the set of states of `self` and `other`.

Let (A_j, B_j, a_j, b_j) for $j \in \{1, \dots, d\}$ be transitions in the machines `self` and in `other`. Then there is a transition $((A_1, \dots, A_d), (B_1, \dots, B_d), a, (b_1, \dots, b_d))$ in the new transducer if $a_1 = \cdots = a_d =: a$.

EXAMPLES:

```
sage: transducer1 = Transducer([('A', 'A', 0, 0),
....:                           ('A', 'A', 1, 1)],
....:                          initial_states=['A'],
```

(continues on next page)

(continued from previous page)

```
....:                          final_states=['A'],
....:                          determine_alphabets=True)
sage: transducer2 = Transducer([(0, 1, 0, ['b', 'c']),
....:                           (0, 0, 1, 'b'),
....:                           (1, 1, 0, 'a')],
....:                          initial_states=[0],
....:                          final_states=[1],
....:                          determine_alphabets=True)
sage: result = transducer1.cartesian_product(transducer2)
sage: result
Transducer with 2 states
sage: result.transitions()
[Transition from ('A', 0) to ('A', 1): 0|(0, 'b'),(None, 'c'),
 Transition from ('A', 0) to ('A', 0): 1|(1, 'b'),
 Transition from ('A', 1) to ('A', 1): 0|(0, 'a')]
sage: result([1, 0, 0])
[(1, 'b'), (0, 'b'), (None, 'c'),  (0, 'a')]
sage: (transducer1([1, 0, 0]), transducer2([1, 0, 0]))
([1, 0, 0], ['b', 'b', 'c', 'a'])
```

```
>>> from sage.all import *
>>> transducer1 = Transducer([('A', 'A', Integer(0), Integer(0)),
...                           ('A', 'A', Integer(1), Integer(1))],
...                          initial_states=['A'],
...                          final_states=['A'],
...                          determine_alphabets=True)
>>> transducer2 = Transducer([(Integer(0), Integer(1), Integer(0), ['b', 'c
↪']),
...                           (Integer(0), Integer(0), Integer(1), 'b'),
...                           (Integer(1), Integer(1), Integer(0), 'a')],
...                          initial_states=[Integer(0)],
...                          final_states=[Integer(1)],
...                          determine_alphabets=True)
>>> result = transducer1.cartesian_product(transducer2)
>>> result
Transducer with 2 states
>>> result.transitions()
[Transition from ('A', 0) to ('A', 1): 0|(0, 'b'),(None, 'c'),
 Transition from ('A', 0) to ('A', 0): 1|(1, 'b'),
 Transition from ('A', 1) to ('A', 1): 0|(0, 'a')]
>>> result([Integer(1), Integer(0), Integer(0)])
[(1, 'b'), (0, 'b'), (None, 'c'),  (0, 'a')]
>>> (transducer1([Integer(1), Integer(0), Integer(0)]),
↪transducer2([Integer(1), Integer(0), Integer(0)]))
([1, 0, 0], ['b', 'b', 'c', 'a'])
```

Also final output words are correctly processed:

```
sage: transducer1.state('A').final_word_out = 2
sage: result = transducer1.cartesian_product(transducer2)
sage: result.final_states()[0].final_word_out
[(2, None)]
```

```
>>> from sage.all import *
>>> transducer1.state('A').final_word_out = Integer(2)
>>> result = transducer1.cartesian_product(transducer2)
>>> result.final_states()[Integer(0)].final_word_out
[(2, None)]
```

The following transducer counts the number of 11 blocks minus the number of 10 blocks over the alphabet `[0, 1]`.

```
sage: count_11 = transducers.CountSubblockOccurrences(
....:     [1, 1],
....:     input_alphabet=[0, 1])
sage: count_10 = transducers.CountSubblockOccurrences(
....:     [1, 0],
....:     input_alphabet=[0, 1])
sage: count_11x10 = count_11.cartesian_product(count_10)
sage: difference = transducers.sub([0, 1])(count_11x10)
sage: T = difference.simplification().relabeled()
sage: T.initial_states()
[1]
sage: sorted(T.transitions())
[Transition from 0 to 1: 0|-1,
 Transition from 0 to 0: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 0: 1|0]
sage: input =  [0, 1, 1,  0, 1,  0, 0, 0, 1, 1, 1,  0]
sage: output = [0, 0, 1, -1, 0, -1, 0, 0, 0, 1, 1, -1]
sage: T(input) == output
True
```

```
>>> from sage.all import *
>>> count_11 = transducers.CountSubblockOccurrences(
...     [Integer(1), Integer(1)],
...     input_alphabet=[Integer(0), Integer(1)])
>>> count_10 = transducers.CountSubblockOccurrences(
...     [Integer(1), Integer(0)],
...     input_alphabet=[Integer(0), Integer(1)])
>>> count_11x10 = count_11.cartesian_product(count_10)
>>> difference = transducers.sub([Integer(0), Integer(1)])(count_11x10)
>>> T = difference.simplification().relabeled()
>>> T.initial_states()
[1]
>>> sorted(T.transitions())
[Transition from 0 to 1: 0|-1,
 Transition from 0 to 0: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 0: 1|0]
>>> input =  [Integer(0), Integer(1), Integer(1),  Integer(0), Integer(1), 
↪Integer(0), Integer(0), Integer(0), Integer(1), Integer(1), Integer(1), 
↪Integer(0)]
>>> output = [Integer(0), Integer(0), Integer(1), -Integer(1), Integer(0), -
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(1), Integer(1), -
↪Integer(1)]
```

(continues on next page)

(continued from previous page)

```
>>> T(input) == output
True
```

If `other` is an automaton, then `cartesian_product()` returns `self` where the input is restricted to the input accepted by `other`.

For example, if the transducer transforms the standard binary expansion into the non-adjacent form and the automaton recognizes the binary expansion without adjacent ones, then the Cartesian product of these two is a transducer which does not change the input (except for changing `a` to `(a, None)` and ignoring a leading 0).

```
sage: NAF = Transducer([(0, 1, 0, None),
....:                   (0, 2, 1, None),
....:                   (1, 1, 0, 0),
....:                   (1, 2, 1, 0),
....:                   (2, 1, 0, 1),
....:                   (2, 3, 1, -1),
....:                   (3, 2, 0, 0),
....:                   (3, 3, 1, 0)],
....:                  initial_states=[0],
....:                  final_states=[1],
....:                  determine_alphabets=True)
sage: aut11 = Automaton([(0, 0, 0), (0, 1, 1), (1, 0, 0)],
....:                   initial_states=[0],
....:                   final_states=[0, 1],
....:                   determine_alphabets=True)
sage: res = NAF.cartesian_product(aut11)
sage: res([1, 0, 0, 1, 0, 1, 0])
[(1, None), (0, None), (0, None), (1, None), (0, None), (1, None)]
```

```
>>> from sage.all import *
>>> NAF = Transducer([(Integer(0), Integer(1), Integer(0), None),
...                   (Integer(0), Integer(2), Integer(1), None),
...                   (Integer(1), Integer(1), Integer(0), Integer(0)),
...                   (Integer(1), Integer(2), Integer(1), Integer(0)),
...                   (Integer(2), Integer(1), Integer(0), Integer(1)),
...                   (Integer(2), Integer(3), Integer(1), -Integer(1)),
...                   (Integer(3), Integer(2), Integer(0), Integer(0)),
...                   (Integer(3), Integer(3), Integer(1), Integer(0))],
...                  initial_states=[Integer(0)],
...                  final_states=[Integer(1)],
...                  determine_alphabets=True)
>>> aut11 = Automaton([(Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(1)), (Integer(1), Integer(0), Integer(0))],
...                   initial_states=[Integer(0)],
...                   final_states=[Integer(0), Integer(1)],
...                   determine_alphabets=True)
>>> res = NAF.cartesian_product(aut11)
>>> res([Integer(1), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0)])
[(1, None), (0, None), (0, None), (1, None), (0, None), (1, None)]
```

This is obvious because if the standard binary expansion does not have adjacent ones, then it is the same as the non-adjacent form.

Be aware that *cartesian_product()* is not commutative.

```
sage: aut11.cartesian_product(NAF)
Traceback (most recent call last):
...
TypeError: Only an automaton can be intersected with an automaton.
```

```
>>> from sage.all import *
>>> aut11.cartesian_product(NAF)
Traceback (most recent call last):
...
TypeError: Only an automaton can be intersected with an automaton.
```

The Cartesian product of more than two finite state machines can also be computed:

```
sage: T0 = transducers.CountSubblockOccurrences([0, 0], [0, 1, 2])
sage: T1 = transducers.CountSubblockOccurrences([1, 1], [0, 1, 2])
sage: T2 = transducers.CountSubblockOccurrences([2, 2], [0, 1, 2])
sage: T = T0.cartesian_product([T1, T2])
sage: T.transitions()
[Transition from ((), (), ()) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (), ()) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((), (), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((0,), (), ()) to ((0,), (), ()): 0|(1, 0, 0),
 Transition from ((0,), (), ()) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((0,), (), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((), (1,), ()) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (1,), ()) to ((), (1,), ()): 1|(0, 1, 0),
 Transition from ((), (1,), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((), (), (2,)) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (), (2,)) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((), (), (2,)) to ((), (), (2,)): 2|(0, 0, 1)]
sage: T([0, 0, 1, 1, 2, 2, 0, 1, 2, 2])
[(0, 0, 0),
 (1, 0, 0),
 (0, 0, 0),
 (0, 1, 0),
 (0, 0, 0),
 (0, 0, 1),
 (0, 0, 0),
 (0, 0, 0),
 (0, 0, 0),
 (0, 0, 1)]
```

```
>>> from sage.all import *
>>> T0 = transducers.CountSubblockOccurrences([Integer(0), Integer(0)],␣
↪[Integer(0), Integer(1), Integer(2)])
>>> T1 = transducers.CountSubblockOccurrences([Integer(1), Integer(1)],␣
↪[Integer(0), Integer(1), Integer(2)])
>>> T2 = transducers.CountSubblockOccurrences([Integer(2), Integer(2)],␣
↪[Integer(0), Integer(1), Integer(2)])
>>> T = T0.cartesian_product([T1, T2])
```

(continues on next page)

(continued from previous page)

```
>>> T.transitions()
[Transition from ((), (), ()) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (), ()) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((), (), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((0,), (), ()) to ((0,), (), ()): 0|(1, 0, 0),
 Transition from ((0,), (), ()) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((0,), (), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((), (1,), ()) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (1,), ()) to ((), (1,), ()): 1|(0, 1, 0),
 Transition from ((), (1,), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((), (), (2,)) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (), (2,)) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((), (), (2,)) to ((), (), (2,)): 2|(0, 0, 1)]
>>> T([Integer(0), Integer(0), Integer(1), Integer(1), Integer(2), Integer(2),
↪ Integer(0), Integer(1), Integer(2), Integer(2)])
[(0, 0, 0),
 (1, 0, 0),
 (0, 0, 0),
 (0, 1, 0),
 (0, 0, 0),
 (0, 0, 1),
 (0, 0, 0),
 (0, 0, 0),
 (0, 0, 0),
 (0, 0, 1)]
```

intersection(*other*, *only_accessible_components=True*)

Return a new transducer which accepts an input if it is accepted by both given finite state machines producing the same output.

INPUT:

- `other` – a transducer
- `only_accessible_components` – if `True` (default), then the result is piped through *accessible_components()*. If no `new_input_alphabet` is given, it is determined by *determine_alphabets()*.

OUTPUT:

A new transducer which computes the intersection (see below) of the languages of `self` and `other`.

The set of states of the transducer is the Cartesian product of the set of states of both given transducer. There is a transition $((A, B), (C, D), a, b)$ in the new transducer if there are transitions (A, C, a, b) and (B, D, a, b) in the old transducers.

EXAMPLES:

```
sage: transducer1 = Transducer([('1', '2', 1, 0),
....:                           ('2', '2', 1, 0),
....:                           ('2', '2', 0, 1)],
....:                          initial_states=['1'],
....:                          final_states=['2'])
sage: transducer2 = Transducer([('A', 'A', 1, 0),
....:                           ('A', 'B', 0, 0),
```

(continues on next page)

(continued from previous page)

```
....:                            ('B', 'B', 0, 1),
....:                            ('B', 'A', 1, 1)],
....:                           initial_states=['A'],
....:                           final_states=['B'])
sage: res = transducer1.intersection(transducer2)
sage: res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|0,
 Transition from ('2', 'A') to ('2', 'A'): 1|0]
```

```
>>> from sage.all import *
>>> transducer1 = Transducer([('1', '2', Integer(1), Integer(0)),
...                           ('2', '2', Integer(1), Integer(0)),
...                           ('2', '2', Integer(0), Integer(1))],
...                          initial_states=['1'],
...                          final_states=['2'])
>>> transducer2 = Transducer([('A', 'A', Integer(1), Integer(0)),
...                           ('A', 'B', Integer(0), Integer(0)),
...                           ('B', 'B', Integer(0), Integer(1)),
...                           ('B', 'A', Integer(1), Integer(1))],
...                          initial_states=['A'],
...                          final_states=['B'])
>>> res = transducer1.intersection(transducer2)
>>> res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|0,
 Transition from ('2', 'A') to ('2', 'A'): 1|0]
```

In general, transducers are not closed under intersection. But for transducer which do not have epsilon-transitions, the intersection is well defined (cf. [BaWo2012]). However, in the next example the intersection of the two transducers is not well defined. The intersection of the languages consists of $(a^n, b^n c^n)$. This set is not recognizable by a *finite* transducer.

```
sage: t1 = Transducer([(0, 0, 'a', 'b'),
....:                  (0, 1, None, 'c'),
....:                  (1, 1, None, 'c')],
....:                 initial_states=[0],
....:                 final_states=[0, 1])
sage: t2 = Transducer([('A', 'A', None, 'b'),
....:                  ('A', 'B', 'a', 'c'),
....:                  ('B', 'B', 'a', 'c')],
....:                 initial_states=['A'],
....:                 final_states=['A', 'B'])
sage: t2.intersection(t1)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input or output)
was found.
```

```
>>> from sage.all import *
>>> t1 = Transducer([(Integer(0), Integer(0), 'a', 'b'),
...                  (Integer(0), Integer(1), None, 'c'),
...                  (Integer(1), Integer(1), None, 'c')],
...                 initial_states=[Integer(0)],
```

(continues on next page)

(continued from previous page)

```
...                  final_states=[Integer(0), Integer(1)])
>>> t2 = Transducer([('A', 'A', None, 'b'),
...                   ('A', 'B', 'a', 'c'),
...                   ('B', 'B', 'a', 'c')],
...                  initial_states=['A'],
...                  final_states=['A', 'B'])
>>> t2.intersection(t1)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input or output)
was found.
```

REFERENCES:

process(**args*, ***kwargs*)

Return whether the transducer accepts the input, the state where the computation stops and which output is generated.

INPUT:

- `input_tape` – the input tape can be a list or an iterable with entries from the input alphabet. If we are working with a multi-tape machine (see parameter `use_multitape_input` and notes below), then the tape is a list or tuple of tracks, each of which can be a list or an iterable with entries from the input alphabet.
- `initial_state` or `initial_states` – the initial state(s) in which the machine starts. Either specify a single one with `initial_state` or a list of them with `initial_states`. If both are given, `initial_state` will be appended to `initial_states`. If neither is specified, the initial states of the finite state machine are taken.
- `list_of_outputs` – (default: `None`) a boolean or `None`. If `True`, then the outputs are given in list form (even if we have no or only one single output). If `False`, then the result is never a list (an exception is raised if the result cannot be returned). If `list_of_outputs=None` the method determines automatically what to do (e.g. if a non-deterministic machine returns more than one path, then the output is returned in list form).
- `only_accepted` – boolean (default: `False`); if set, then the first argument in the output is guaranteed to be `True` (if the output is a list, then the first argument of each element will be `True`).
- `full_output` – boolean (default: `True`); if set, then the full output is given, otherwise only the generated output (the third entry below only). If the input is not accepted, a `ValueError` is raised.
- `always_include_output` – if set (not by default), always include the output. This is inconsequential for a *Transducer*, but can be used in other classes derived from *FiniteStateMachine* where the output is suppressed by default, cf. *Automaton.process()*.
- `format_output` – a function that translates the written output (which is in form of a list) to something more readable. By default (`None`) identity is used here.
- `check_epsilon_transitions` – boolean (default: `True`); if `False`, then epsilon transitions are not taken into consideration during process
- `write_final_word_out` – boolean (default: `True`); whether the final output words should be written or not
- `use_multitape_input` – boolean (default: `False`); if `True`, then the multi-tape mode of the process iterator is activated. See also the notes below for multi-tape machines.

- `process_all_prefixes_of_input` – boolean (default: `False`); if `True`, then each prefix of the input word is processed (instead of processing the whole input word at once). Consequently, there is an output generated for each of these prefixes.
- `process_iterator_class` – (default: `None`) a class inherited from `FSMProcessIterator`. If `None`, then `FSMProcessIterator` is taken. An instance of this class is created and is used during the processing.
- `automatic_output_type` – boolean (default: `False`); if set and the input has a parent, then the output will have the same parent. If the input does not have a parent, then the output will be of the same type as the input.

OUTPUT:

The full output is a triple (or a list of triples, cf. parameter `list_of_outputs`), where

- the first entry is `True` if the input string is accepted,
- the second gives the reached state after processing the input tape (This is a state with label `None` if the input could not be processed, i.e., if at one point no transition to go on could be found.), and
- the third gives a list of the output labels written during processing.

If `full_output` is `False`, then only the third entry is returned.

Note that in the case the transducer is not deterministic, all possible paths are taken into account.

This function uses an iterator which, in its simplest form, goes from one state to another in each step. To decide which way to go, it uses the input words of the outgoing transitions and compares them to the input tape. More precisely, in each step, the iterator takes an outgoing transition of the current state, whose input label equals the input letter of the tape. The output label of the transition, if present, is written on the output tape.

If the choice of the outgoing transition is not unique (i.e., we have a non-deterministic finite state machine), all possibilities are followed. This is done by splitting the process into several branches, one for each of the possible outgoing transitions.

The process (iteration) stops if all branches are finished, i.e., for no branch, there is any transition whose input word coincides with the processed input tape. This can simply happen when the entire tape was read.

Also see `__call__()` for a version of `process()` with shortened output.

Internally this function creates and works with an instance of `FSMProcessIterator`. This iterator can also be obtained with `iter_process()`.

If working with multi-tape finite state machines, all input words of transitions are words of k-tuples of letters. Moreover, the input tape has to consist of k tracks, i.e., be a list or tuple of k iterators, one for each track.

Warning: Working with multi-tape finite state machines is still experimental and can lead to wrong outputs.

EXAMPLES:

```
sage: binary_inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:                              initial_states=['A'], final_states=['A'])
sage: binary_inverter.process([0, 1, 0, 0, 1, 1])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

```
>>> from sage.all import *
>>> binary_inverter = Transducer({'A': [('A', Integer(0), Integer(1)), ('A',␣
↪Integer(1), Integer(0))]},
...                               initial_states=['A'], final_states=['A'])
>>> binary_inverter.process([Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1)])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

If we are only interested in the output, we can also use:

```
sage: binary_inverter([0, 1, 0, 0, 1, 1])
[1, 0, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> binary_inverter([Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1)])
[1, 0, 1, 1, 0, 0]
```

This can also be used with words as input:

```
sage: # needs sage.combinat
sage: W = Words([0, 1]); W
Finite and infinite words over {0, 1}
sage: w = W([0, 1, 0, 0, 1, 1]); w
word: 010011
sage: binary_inverter(w)
word: 101100
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> W = Words([Integer(0), Integer(1)]); W
Finite and infinite words over {0, 1}
>>> w = W([Integer(0), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1)]); w
word: 010011
>>> binary_inverter(w)
word: 101100
```

In this case it is automatically determined that the output is a word. The call above is equivalent to:

```
sage: binary_inverter.process(w,                                            #␣
↪needs sage.combinat
....:                          full_output=False,
....:                          list_of_outputs=False,
....:                          automatic_output_type=True)
word: 101100
```

```
>>> from sage.all import *
>>> binary_inverter.process(w,                                            #␣
↪needs sage.combinat
...                          full_output=False,
...                          list_of_outputs=False,
```

(continues on next page)

(continued from previous page)

```
...                             automatic_output_type=True)
word: 101100
```

The following transducer transforms $0^n 1$ to $1^n 2$:

```
sage: T = Transducer([(0, 0, 0, 1), (0, 1, 1, 2)])
sage: T.state(0).is_initial = True
sage: T.state(1).is_final = True
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(0), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(1), Integer(1), Integer(2))])
>>> T.state(Integer(0)).is_initial = True
>>> T.state(Integer(1)).is_final = True
```

We can see the different possibilities of the output by:

```
sage: [T.process(w) for w in [[1], [0, 1], [0, 0, 1], [0, 1, 1],
....:                         [0], [0, 0], [2, 0], [0, 1, 2]]]
[(True, 1, [2]), (True, 1, [1, 2]),
 (True, 1, [1, 1, 2]), (False, None, None),
 (False, 0, [1]), (False, 0, [1, 1]),
 (False, None, None), (False, None, None)]
```

```
>>> from sage.all import *
>>> [T.process(w) for w in [[Integer(1)], [Integer(0), Integer(1)],␣
↪[Integer(0), Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)],
...                         [Integer(0)], [Integer(0), Integer(0)],␣
↪[Integer(2), Integer(0)], [Integer(0), Integer(1), Integer(2)]]]
[(True, 1, [2]), (True, 1, [1, 2]),
 (True, 1, [1, 1, 2]), (False, None, None),
 (False, 0, [1]), (False, 0, [1, 1]),
 (False, None, None), (False, None, None)]
```

If we just want a condensed output, we use:

```
sage: [T.process(w, full_output=False)
....:      for w in [[1], [0, 1], [0, 0, 1]]]
[[2], [1, 2], [1, 1, 2]]
sage: T.process([0], full_output=False)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T.process([0, 1, 2], full_output=False)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

```
>>> from sage.all import *
>>> [T.process(w, full_output=False)
...      for w in [[Integer(1)], [Integer(0), Integer(1)], [Integer(0),␣
↪Integer(0), Integer(1)]]]
```

(continues on next page)

(continued from previous page)

```
[[2], [1, 2], [1, 1, 2]]
>>> T.process([Integer(0)], full_output=False)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
>>> T.process([Integer(0), Integer(1), Integer(2)], full_output=False)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

It is equivalent to:

```
sage: [T(w) for w in [[1], [0, 1], [0, 0, 1]]]
[[2], [1, 2], [1, 1, 2]]
sage: T([0])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T([0, 1, 2])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

```
>>> from sage.all import *
>>> [T(w) for w in [[Integer(1)], [Integer(0), Integer(1)], [Integer(0),␣
↪Integer(0), Integer(1)]]]
[[2], [1, 2], [1, 1, 2]]
>>> T([Integer(0)])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
>>> T([Integer(0), Integer(1), Integer(2)])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

A cycle with empty input and empty output is correctly processed:

```
sage: T = Transducer([(0, 1, None, None), (1, 0, None, None)],
....:                initial_states=[0], final_states=[1])
sage: T.process([])
[(False, 0, []), (True, 1, [])]
sage: _ = T.add_transition(-1, 0, 0, 'r')
sage: T.state(-1).is_initial = True
sage: T.state(0).is_initial = False
sage: T.process([0])
[(False, 0, ['r']), (True, 1, ['r'])]
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), None, None), (Integer(1),␣
↪Integer(0), None, None)],
...                 initial_states=[Integer(0)], final_states=[Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> T.process([])
[(False, 0, []), (True, 1, [])]
>>> _ = T.add_transition(-Integer(1), Integer(0), Integer(0), 'r')
>>> T.state(-Integer(1)).is_initial = True
>>> T.state(Integer(0)).is_initial = False
>>> T.process([Integer(0)])
[(False, 0, ['r']), (True, 1, ['r'])]
```

If there is a cycle with empty input but non-empty output, the possible outputs would be an infinite set:

```
sage: T = Transducer([(0, 1, None, 'z'), (1, 0, None, None)],
....:                initial_states=[0], final_states=[1])
sage: T.process([])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input),
but output is written.
```

```
>>> from sage.all import *
>>> T = Transducer([(Integer(0), Integer(1), None, 'z'), (Integer(1),␣
↪Integer(0), None, None)],
...                initial_states=[Integer(0)], final_states=[Integer(1)])
>>> T.process([])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input),
but output is written.
```

But if this cycle with empty input and non-empty output is not reached, the correct output is produced:

```
sage: _ = T.add_transition(-1, 0, 0, 'r')
sage: T.state(-1).is_initial = True
sage: T.state(0).is_initial = False
sage: T.process([])
(False, -1, [])
sage: T.process([0])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input),
but output is written.
```

```
>>> from sage.all import *
>>> _ = T.add_transition(-Integer(1), Integer(0), Integer(0), 'r')
>>> T.state(-Integer(1)).is_initial = True
>>> T.state(Integer(0)).is_initial = False
>>> T.process([])
(False, -1, [])
>>> T.process([Integer(0)])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input),
but output is written.
```

If we set `check_epsilon_transitions=False`, then no transitions with empty input are considered anymore. Thus cycles with empty input are no problem anymore:

```
sage: T.process([0], check_epsilon_transitions=False)
(False, 0, ['r'])
```

```
>>> from sage.all import *
>>> T.process([Integer(0)], check_epsilon_transitions=False)
(False, 0, ['r'])
```

A simple example of a multi-tape transducer is the following: It writes the length of the first tape many letters `a` and then the length of the second tape many letters `b`:

```
sage: M = Transducer([(0, 0, (1, None), 'a'),
....:                  (0, 1, [], []),
....:                  (1, 1, (None, 1), 'b')],
....:                 initial_states=[0],
....:                 final_states=[1])
sage: M.process(([1, 1], [1]), use_multitape_input=True)
(True, 1, ['a', 'a', 'b'])
```

```
>>> from sage.all import *
>>> M = Transducer([(Integer(0), Integer(0), (Integer(1), None), 'a'),
...                  (Integer(0), Integer(1), [], []),
...                  (Integer(1), Integer(1), (None, Integer(1)), 'b')],
...                 initial_states=[Integer(0)],
...                 final_states=[Integer(1)])
>>> M.process(([Integer(1), Integer(1)], [Integer(1)]), use_multitape_
↪input=True)
(True, 1, ['a', 'a', 'b'])
```

See also

FiniteStateMachine.process(), *Automaton.process()*, *iter_process()*, *__call__()*, *FSMProcessIterator*.

simplification()

Return a simplified transducer.

OUTPUT: a new transducer

This function simplifies a transducer by Moore's algorithm, first moving common output labels of transitions leaving a state to output labels of transitions entering the state (cf. *prepone_output()*).

The resulting transducer implements the same function as the original transducer.

EXAMPLES:

```
sage: fsm = Transducer([("A", "B", 0, 1), ("A", "B", 1, 0),
....:                          ("B", "C", 0, 0), ("B", "C", 1, 1),
....:                          ("C", "D", 0, 1), ("C", "D", 1, 0),
....:                          ("D", "A", 0, 0), ("D", "A", 1, 1)])
sage: fsms = fsm.simplification()
sage: fsms
```

(continues on next page)

(continued from previous page)

```
Transducer with 2 states
sage: fsms.transitions()
[Transition from ('B', 'D') to ('A', 'C'): 0|0,
 Transition from ('B', 'D') to ('A', 'C'): 1|1,
 Transition from ('A', 'C') to ('B', 'D'): 0|1,
 Transition from ('A', 'C') to ('B', 'D'): 1|0]
sage: fsms.relabeled().transitions()
[Transition from 0 to 1: 0|0,
 Transition from 0 to 1: 1|1,
 Transition from 1 to 0: 0|1,
 Transition from 1 to 0: 1|0]
```

```
>>> from sage.all import *
>>> fsm = Transducer([("A", "B", Integer(0), Integer(1)), ("A", "B",␣
↪Integer(1), Integer(0)),
...                               ("B", "C", Integer(0), Integer(0)), ("B", "C",␣
↪Integer(1), Integer(1)),
...                               ("C", "D", Integer(0), Integer(1)), ("C", "D",␣
↪Integer(1), Integer(0)),
...                               ("D", "A", Integer(0), Integer(0)), ("D", "A",␣
↪Integer(1), Integer(1))])
>>> fsms = fsm.simplification()
>>> fsms
Transducer with 2 states
>>> fsms.transitions()
[Transition from ('B', 'D') to ('A', 'C'): 0|0,
 Transition from ('B', 'D') to ('A', 'C'): 1|1,
 Transition from ('A', 'C') to ('B', 'D'): 0|1,
 Transition from ('A', 'C') to ('B', 'D'): 1|0]
>>> fsms.relabeled().transitions()
[Transition from 0 to 1: 0|0,
 Transition from 0 to 1: 1|1,
 Transition from 1 to 0: 0|1,
 Transition from 1 to 0: 1|0]
```

```
sage: fsm = Transducer([("A", "A", 0, 0),
....:                   ("A", "B", 1, 1),
....:                   ("A", "C", 1, -1),
....:                   ("B", "A", 2, 0),
....:                   ("C", "A", 2, 0)])
sage: fsm_simplified = fsm.simplification()
sage: fsm_simplified
Transducer with 2 states
sage: fsm_simplified.transitions()
[Transition from ('A',) to ('A',): 0|0,
 Transition from ('A',) to ('B', 'C'): 1|1,0,
 Transition from ('A',) to ('B', 'C'): 1|-1,0,
 Transition from ('B', 'C') to ('A',): 2|-]
```

```
>>> from sage.all import *
>>> fsm = Transducer([("A", "A", Integer(0), Integer(0)),
```

(continues on next page)

(continued from previous page)

```
...                    ("A", "B", Integer(1), Integer(1)),
...                    ("A", "C", Integer(1), -Integer(1)),
...                    ("B", "A", Integer(2), Integer(0)),
...                    ("C", "A", Integer(2), Integer(0))])
>>> fsm_simplified = fsm.simplification()
>>> fsm_simplified
Transducer with 2 states
>>> fsm_simplified.transitions()
[Transition from ('A',) to ('A',): 0|0,
 Transition from ('A',) to ('B', 'C'): 1|1,0,
 Transition from ('A',) to ('B', 'C'): 1|-1,0,
 Transition from ('B', 'C') to ('A',): 2|-]
```

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
sage: T = Transducer([('A', 'A', 1/2, 0),
....:                 ('A', 'B', 1/4, 1),
....:                 ('A', 'C', 1/4, 1),
....:                 ('B', 'A', 1, 0),
....:                 ('C', 'A', 1, 0)],
....:                initial_states=[0],
....:                final_states=['A', 'B', 'C'],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: sorted(T.simplification().transitions())
[Transition from ('A',) to ('A',): 1/2|0,
 Transition from ('A',) to ('B', 'C'): 1/2|1,
 Transition from ('B', 'C') to ('A',): 1|0]
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
>>> T = Transducer([('A', 'A', Integer(1)/Integer(2), Integer(0)),
...                 ('A', 'B', Integer(1)/Integer(4), Integer(1)),
...                 ('A', 'C', Integer(1)/Integer(4), Integer(1)),
...                 ('B', 'A', Integer(1), Integer(0)),
...                 ('C', 'A', Integer(1), Integer(0))],
...                initial_states=[Integer(0)],
...                final_states=['A', 'B', 'C'],
...                on_duplicate_transition=duplicate_transition_add_input)
>>> sorted(T.simplification().transitions())
[Transition from ('A',) to ('A',): 1/2|0,
 Transition from ('A',) to ('B', 'C'): 1/2|1,
 Transition from ('B', 'C') to ('A',): 1|0]
```

Illustrating the use of colors in order to avoid identification of states:

```
sage: T = Transducer( [[0,0,0,0], [0,1,1,1],
....:                  [1,0,0,0], [1,1,1,1]],
....:                 initial_states=[0],
....:                 final_states=[0,1])
sage: sorted(T.simplification().transitions())
[Transition from (0, 1) to (0, 1): 0|0,
```

(continues on next page)

(continued from previous page)

```
 Transition from (0, 1) to (0, 1): 1|1]
sage: T.state(0).color = 0
sage: T.state(0).color = 1
sage: sorted(T.simplification().transitions())
[Transition from (0,) to (0,): 0|0,
 Transition from (0,) to (1,): 1|1,
 Transition from (1,) to (0,): 0|0,
 Transition from (1,) to (1,): 1|1]
```

```
>>> from sage.all import *
>>> T = Transducer( [[Integer(0),Integer(0),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(1),Integer(1)],
...                      [Integer(1),Integer(0),Integer(0),Integer(0)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1)]],
...                     initial_states=[Integer(0)],
...                     final_states=[Integer(0),Integer(1)])
>>> sorted(T.simplification().transitions())
[Transition from (0, 1) to (0, 1): 0|0,
 Transition from (0, 1) to (0, 1): 1|1]
>>> T.state(Integer(0)).color = Integer(0)
>>> T.state(Integer(0)).color = Integer(1)
>>> sorted(T.simplification().transitions())
[Transition from (0,) to (0,): 0|0,
 Transition from (0,) to (1,): 1|1,
 Transition from (1,) to (0,): 0|0,
 Transition from (1,) to (1,): 1|1]
```

sage.combinat.finite_state_machine.**duplicate_transition_add_input**(*old_transition*, *new_transition*)

Alternative function for handling duplicate transitions in finite state machines. This implementation adds the input label of the new transition to the input label of the old transition. This is intended for the case where a Markov chain is modelled by a finite state machine using the input labels as transition probabilities.

See the documentation of the `on_duplicate_transition` parameter of *FiniteStateMachine*.

INPUT:

- `old_transition` – a transition in a finite state machine
- `new_transition` – a transition, identical to `old_transition`, which is to be inserted into the finite state machine

OUTPUT:

A transition whose input weight is the sum of the input weights of `old_transition` and `new_transition`.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_
↪input
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: duplicate_transition_add_input(FSMTransition('a', 'a', 1/2),
....:                                FSMTransition('a', 'a', 1/2))
Transition from 'a' to 'a': 1|-
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_add_input
>>> from sage.combinat.finite_state_machine import FSMTransition
>>> duplicate_transition_add_input(FSMTransition('a', 'a', Integer(1)/Integer(2)),
...                                FSMTransition('a', 'a', Integer(1)/Integer(2)))
Transition from 'a' to 'a': 1|-
```

Input labels must be lists of length 1:

```
sage: duplicate_transition_add_input(FSMTransition('a', 'a', [1, 1]),
....:                                FSMTransition('a', 'a', [1, 1]))
Traceback (most recent call last):
...
TypeError: Trying to use duplicate_transition_add_input on
"Transition from 'a' to 'a': 1,1|-" and
"Transition from 'a' to 'a': 1,1|-",
but input words are assumed to be lists of length 1
```

```
>>> from sage.all import *
>>> duplicate_transition_add_input(FSMTransition('a', 'a', [Integer(1),
↪Integer(1)]),
...                                FSMTransition('a', 'a', [Integer(1),
↪Integer(1)]))
Traceback (most recent call last):
...
TypeError: Trying to use duplicate_transition_add_input on
"Transition from 'a' to 'a': 1,1|-" and
"Transition from 'a' to 'a': 1,1|-",
but input words are assumed to be lists of length 1
```

sage.combinat.finite_state_machine.**duplicate_transition_ignore**(*old_transition*, *new_transition*)

Default function for handling duplicate transitions in finite state machines. This implementation ignores the occurrence.

See the documentation of the `on_duplicate_transition` parameter of `FiniteStateMachine`.

INPUT:

- `old_transition` – a transition in a finite state machine
- `new_transition` – a transition, identical to `old_transition`, which is to be inserted into the finite state machine

OUTPUT: the same transition, unchanged

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_ignore
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: duplicate_transition_ignore(FSMTransition(0, 0, 1),
....:                             FSMTransition(0, 0, 1))
Transition from 0 to 0: 1|-
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_ignore
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.finite_state_machine import FSMTransition
>>> duplicate_transition_ignore(FSMTransition(Integer(0), Integer(0), Integer(1)),
...                             FSMTransition(Integer(0), Integer(0), Integer(1)))
Transition from 0 to 0: 1|-
```

sage.combinat.finite_state_machine.**duplicate_transition_raise_error**(*old_transition*, *new_transition*)

Alternative function for handling duplicate transitions in finite state machines.

This implementation raises a `ValueError`.

See the documentation of the `on_duplicate_transition` parameter of *FiniteStateMachine*.

INPUT:

- `old_transition` – a transition in a finite state machine
- `new_transition` – a transition, identical to `old_transition`, which is to be inserted into the finite state machine

OUTPUT: nothing. A `ValueError` is raised

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_raise_
↪error
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: duplicate_transition_raise_error(FSMTransition(0, 0, 1),
....:                                  FSMTransition(0, 0, 1))
Traceback (most recent call last):
...
ValueError: Attempting to re-insert transition Transition from 0 to 0: 1|-
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import duplicate_transition_raise_
↪error
>>> from sage.combinat.finite_state_machine import FSMTransition
>>> duplicate_transition_raise_error(FSMTransition(Integer(0), Integer(0),␣
↪Integer(1)),
...                                  FSMTransition(Integer(0), Integer(0),␣
↪Integer(1)))
Traceback (most recent call last):
...
ValueError: Attempting to re-insert transition Transition from 0 to 0: 1|-
```

sage.combinat.finite_state_machine.**equal**(*iterator*)

Check whether all elements of `iterator` are equal.

INPUT:

- `iterator` – an iterator of the elements to check

OUTPUT: boolean

This implements https://stackoverflow.com/a/3844832/1052778.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import equal
sage: equal([0, 0, 0])
True
sage: equal([0, 1, 0])
False
sage: equal([])
True
sage: equal(iter([None, None]))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import equal
>>> equal([Integer(0), Integer(0), Integer(0)])
True
>>> equal([Integer(0), Integer(1), Integer(0)])
False
>>> equal([])
True
>>> equal(iter([None, None]))
True
```

We can test other properties of the elements than the elements themselves. In the following example, we check whether all tuples have the same lengths:

```
sage: equal(len(x) for x in [(1, 2), (2, 3), (3, 1)])
True
sage: equal(len(x) for x in [(1, 2), (1, 2, 3), (3, 1)])
False
```

```
>>> from sage.all import *
>>> equal(len(x) for x in [(Integer(1), Integer(2)), (Integer(2), Integer(3)),
→(Integer(3), Integer(1))])
True
>>> equal(len(x) for x in [(Integer(1), Integer(2)), (Integer(1), Integer(2),
→Integer(3)), (Integer(3), Integer(1))])
False
```

sage.combinat.finite_state_machine.**full_group_by**(*l*, *key=None*)

Group iterable `l` by values of `key`.

INPUT:

- iterable `l`
- key function `key`

OUTPUT:

A list of pairs `(k, elements)` such that `key(e)=k` for all `e` in `elements`.

This is similar to `itertools.groupby()` except that lists are returned instead of iterables and no prior sorting is required.

We do not require

- that the keys are sortable (in contrast to the approach via `sorted()` and `itertools.groupby()`) and

- that the keys are hashable (in contrast to the implementation proposed in https://stackoverflow.com/a/15250161).

However, it is required

- that distinct keys have distinct `str`-representations.

The implementation is inspired by https://stackoverflow.com/a/15250161, but non-hashable keys are allowed.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: from sage.combinat.finite_state_machine import full_group_by
sage: t = [2/x, 1/x, 2/x]
sage: r = full_group_by([0, 1, 2], key=lambda i: t[i])
sage: sorted(r, key=lambda p: p[1])
[(2/x, [0, 2]), (1/x, [1])]
sage: from itertools import groupby
sage: for k, elements in groupby(sorted([0, 1, 2],
....:                              key=lambda i:t[i]),
....:                              key=lambda i:t[i]):
....:     print("{} {}".format(k, list(elements)))
2/x [0]
1/x [1]
2/x [2]
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> from sage.combinat.finite_state_machine import full_group_by
>>> t = [Integer(2)/x, Integer(1)/x, Integer(2)/x]
>>> r = full_group_by([Integer(0), Integer(1), Integer(2)], key=lambda i: t[i])
>>> sorted(r, key=lambda p: p[Integer(1)])
[(2/x, [0, 2]), (1/x, [1])]
>>> from itertools import groupby
>>> for k, elements in groupby(sorted([Integer(0), Integer(1), Integer(2)],
...                              key=lambda i:t[i]),
...                              key=lambda i:t[i]):
...     print("{} {}".format(k, list(elements)))
2/x [0]
1/x [1]
2/x [2]
```

Note that the behavior is different from `itertools.groupby()` because neither $1/x < 2/x$ nor $2/x < 1/x$ does hold.

Here, the result `r` has been sorted in order to guarantee a consistent order for the doctest suite.

`sage.combinat.finite_state_machine.`**`is_Automaton`**`(`*FSM*`)`

Test whether or not `FSM` inherits from *Automaton*.

`sage.combinat.finite_state_machine.`**`is_FSMProcessIterator`**`(`*PI*`)`

Test whether or not `PI` inherits from *FSMProcessIterator*.

`sage.combinat.finite_state_machine.`**`is_FSMState`**`(`*S*`)`

Test whether or not `S` inherits from *FSMState*.

`sage.combinat.finite_state_machine.`**`is_FSMTransition`**`(`*T*`)`

Test whether or not `T` inherits from *FSMTransition*.

sage.combinat.finite_state_machine.**is_FiniteStateMachine**(*FSM*)

Test whether or not FSM inherits from *FiniteStateMachine*.

sage.combinat.finite_state_machine.**is_Transducer**(*FSM*)

Test whether or not FSM inherits from *Transducer*.

sage.combinat.finite_state_machine.**setup_latex_preamble**()

This function adds the package tikz with support for automata to the preamble of Latex so that the finite state machines can be drawn nicely.

See the section on *LaTeX output* in the introductory examples of this module.

sage.combinat.finite_state_machine.**startswith**(*list_*, *prefix*)

Determine whether list starts with the given prefix.

INPUT:

- list_ – list
- prefix – list representing the prefix

OUTPUT: boolean

Similar to str.startswith().

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import startswith
sage: startswith([1, 2, 3], [1, 2])
True
sage: startswith([1], [1, 2])
False
sage: startswith([1, 3, 2], [1, 2])
False
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import startswith
>>> startswith([Integer(1), Integer(2), Integer(3)], [Integer(1), Integer(2)])
True
>>> startswith([Integer(1)], [Integer(1), Integer(2)])
False
>>> startswith([Integer(1), Integer(3), Integer(2)], [Integer(1), Integer(2)])
False
```

sage.combinat.finite_state_machine.**tupleofwords_to_wordoftuples**(*tupleofwords*)

Transposes a tuple of words over the alphabet to a word of tuples.

INPUT:

- tupleofwords – tuple of a list of letters

OUTPUT: list of tuples

Missing letters in the words are padded with the letter None (from the empty word).

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import (
....:     tupleofwords_to_wordoftuples)
sage: tupleofwords_to_wordoftuples(
```

(continues on next page)

(continued from previous page)

```
....:     ([1, 2], [3, 4, 5, 6], [7]))
[(1, 3, 7), (2, 4, None), (None, 5, None), (None, 6, None)]
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import (
...     tupleofwords_to_wordoftuples)
>>> tupleofwords_to_wordoftuples(
...     ([Integer(1), Integer(2)], [Integer(3), Integer(4), Integer(5),␣
↪Integer(6)], [Integer(7)]))
[(1, 3, 7), (2, 4, None), (None, 5, None), (None, 6, None)]
```

sage.combinat.finite_state_machine.**wordoftuples_to_tupleofwords**(*wordoftuples*)

Transposes a word of tuples to a tuple of words over the alphabet.

INPUT:

- `wordoftuples` – list of tuples of letters

OUTPUT: a tuple of lists

Letters `None` (empty word) are removed from each word in the output.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import (
....:     wordoftuples_to_tupleofwords)
sage: wordoftuples_to_tupleofwords(
....:     [(1, 2), (1, None), (1, None), (1, 2), (None, 2)])
([1, 1, 1, 1], [2, 2, 2])
```

```
>>> from sage.all import *
>>> from sage.combinat.finite_state_machine import (
...     wordoftuples_to_tupleofwords)
>>> wordoftuples_to_tupleofwords(
...     [(Integer(1), Integer(2)), (Integer(1), None), (Integer(1), None),␣
↪(Integer(1), Integer(2)), (None, Integer(2))])
([1, 1, 1, 1], [2, 2, 2])
```

5.1.109 Common Automata and Transducers (Finite State Machines Generators)

Automata and Transducers in Sage can be built through the `automata` and `transducers` objects, respectively. It contains generators for common finite state machines. For example,

```
sage: I = transducers.Identity([0, 1, 2])
```

```
>>> from sage.all import *
>>> I = transducers.Identity([Integer(0), Integer(1), Integer(2)])
```

generates an identity transducer on the alphabet $\{0, 1, 2\}$.

To construct automata and transducers manually, you can use the classes `Automaton` and `Transducer`, respectively. See *Finite state machines, automata, transducers* for more details and a lot of *examples*.

Automata

`AnyLetter()`	Return an automaton recognizing any letter.
`AnyWord()`	Return an automaton recognizing any word.
`EmptyWord()`	Return an automaton recognizing the empty word.
`Word()`	Return an automaton recognizing the given word.
`ContainsWord()`	Return an automaton recognizing words containing the given word.

Transducers

`Identity()`	Return a transducer realizing the identity map.
`abs()`	Return a transducer realizing absolute value.
`map()`	Return a transducer realizing a function.
`operator()`	Return a transducer realizing a binary operation.
`all()`	Return a transducer realizing logical `and`.
`any()`	Return a transducer realizing logical `or`.
`add()`	Return a transducer realizing addition.
`sub()`	Return a transducer realizing subtraction.
`CountSubblockOccurrences()`	Return a transducer counting the occurrences of a subblock.
`Wait()`	Return a transducer writing `False` until first (or k-th) true input is read.
`weight()`	Return a transducer realizing the Hamming weight.
`GrayCode()`	Return a transducer realizing binary Gray code.
`Recursion()`	Return a transducer defined by recursions.

AUTHORS:

- Clemens Heuberger (2014-04-07): initial version
- Sara Kropf (2014-04-10): some changes in TransducerGenerator
- Daniel Krenn (2014-04-15): improved common docstring during review
- Clemens Heuberger, Daniel Krenn, Sara Kropf (2014-04-16–2014-05-02): A couple of improvements. Details see Issue #16141, Issue #16142, Issue #16143, Issue #16186.
- Sara Kropf (2014-04-29): weight transducer
- Clemens Heuberger, Daniel Krenn (2014-07-18): transducers Wait, all, any
- Clemens Heuberger (2014-08-10): transducer Recursion
- Clemens Heuberger (2015-07-31): automaton word
- Daniel Krenn (2015-09-14): cleanup Issue #18227

ACKNOWLEDGEMENT:

- Clemens Heuberger, Daniel Krenn and Sara Kropf are supported by the Austrian Science Fund (FWF): P 24644-N26.

Functions and methods

class `sage.combinat.finite_state_machine_generators.`**`AutomatonGenerators`**

Bases: `object`

A collection of constructors for several common automata.

A list of all automata in this database is available via tab completion. Type "`automata.`" and then hit tab to see which automata are available.

The automata currently in this class include:

- `AnyLetter()`
- `AnyWord()`
- `EmptyWord()`
- `Word()`
- `ContainsWord()`

AnyLetter(*input_alphabet*)

Return an automaton recognizing any letter of the given input alphabet.

INPUT:

- `input_alphabet` – list; the input alphabet

OUTPUT: an `Automaton`

EXAMPLES:

```
sage: A = automata.AnyLetter([0, 1])
sage: A([])
False
sage: A([0])
True
sage: A([1])
True
sage: A([0, 0])
False
```

```
>>> from sage.all import *
>>> A = automata.AnyLetter([Integer(0), Integer(1)])
>>> A([])
False
>>> A([Integer(0)])
True
>>> A([Integer(1)])
True
>>> A([Integer(0), Integer(0)])
False
```

See also

`AnyWord()`

AnyWord(*input_alphabet*)

Return an automaton recognizing any word of the given input alphabet.

INPUT:

- `input_alphabet` – list; the input alphabet

OUTPUT: an `Automaton`

EXAMPLES:

```
sage: A = automata.AnyWord([0, 1])
sage: A([0])
True
sage: A([1])
True
sage: A([0, 1])
True
sage: A([0, 2])
False
```

```
>>> from sage.all import *
>>> A = automata.AnyWord([Integer(0), Integer(1)])
>>> A([Integer(0)])
True
>>> A([Integer(1)])
True
>>> A([Integer(0), Integer(1)])
True
>>> A([Integer(0), Integer(2)])
False
```

This is equivalent to taking the `kleene_star()` of `AnyLetter()` and minimizing the result. This method immediately gives a minimized version:

```
sage: B = automata.AnyLetter([0, 1]).kleene_star().minimization().relabeled()
sage: B == A
True
```

```
>>> from sage.all import *
>>> B = automata.AnyLetter([Integer(0), Integer(1)]).kleene_star().
↪minimization().relabeled()
>>> B == A
True
```

See also:

`AnyLetter()`, `Word()`.

ContainsWord(*word*, *input_alphabet*)

Return an automaton recognizing the words containing the given word as a factor.

INPUT:

- `word` – list (or other iterable) of letters; the word we are looking for
- `input_alphabet` – list or other iterable; the input alphabet

OUTPUT: an `Automaton`

EXAMPLES:

```
sage: A = automata.ContainsWord([0, 1, 0, 1, 1],
....:                           input_alphabet=[0, 1])
```

(continues on next page)

(continued from previous page)

```
sage: A([1, 0, 1, 0, 1, 0, 1, 1, 0, 0])
True
sage: A([1, 0, 1, 0, 1, 0, 1, 0])
False
```

```
>>> from sage.all import *
>>> A = automata.ContainsWord([Integer(0), Integer(1), Integer(0), Integer(1),
↪ Integer(1)],
...                              input_alphabet=[Integer(0), Integer(1)])
>>> A([Integer(1), Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),
↪ Integer(1), Integer(1), Integer(0), Integer(0)])
True
>>> A([Integer(1), Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),
↪ Integer(1), Integer(0)])
False
```

This is equivalent to taking the concatenation of `AnyWord()`, `Word()` and `AnyWord()` and minimizing the result. This method immediately gives a minimized version:

```
sage: B = (automata.AnyWord([0, 1]) *
....:     automata.Word([0, 1, 0, 1, 1], [0, 1]) *
....:     automata.AnyWord([0, 1])).minimization()
sage: B.is_equivalent(A)
True
```

```
>>> from sage.all import *
>>> B = (automata.AnyWord([Integer(0), Integer(1)]) *
...     automata.Word([Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1)]) *
...     automata.AnyWord([Integer(0), Integer(1)])).minimization()
>>> B.is_equivalent(A)
True
```

See also

`CountSubblockOccurrences()`, `AnyWord()`, `Word()`.

EmptyWord(*input_alphabet=None*)

Return an automaton recognizing the empty word.

INPUT:

- `input_alphabet` – iterable or `None` (default: `None`)

OUTPUT: an `Automaton`

EXAMPLES:

```
sage: A = automata.EmptyWord()
sage: A([])
True
sage: A([0])
False
```

```
>>> from sage.all import *
>>> A = automata.EmptyWord()
>>> A([])
True
>>> A([Integer(0)])
False
```

See also

AnyLetter(), *AnyWord()*.

Word(*word*, *input_alphabet=None*)

Return an automaton recognizing the given word.

INPUT:

- `word` – an iterable
- `input_alphabet` – list or `None`; if `None`, then the letters occurring in the word are used

OUTPUT: an *Automaton*

EXAMPLES:

```
sage: A = automata.Word([0])
sage: A.transitions()
[Transition from 0 to 1: 0|-]
sage: [A(w) for w in ([], [0], [1])]
[False, True, False]
sage: A = automata.Word([0, 1, 0])
sage: A.transitions()
[Transition from 0 to 1: 0|-,
Transition from 1 to 2: 1|-,
Transition from 2 to 3: 0|-]
sage: [A(w) for w in ([], [0], [0, 1], [0, 1, 1], [0, 1, 0])]
[False, False, False, False, True]
```

```
>>> from sage.all import *
>>> A = automata.Word([Integer(0)])
>>> A.transitions()
[Transition from 0 to 1: 0|-]
>>> [A(w) for w in ([], [Integer(0)], [Integer(1)])]
[False, True, False]
>>> A = automata.Word([Integer(0), Integer(1), Integer(0)])
>>> A.transitions()
[Transition from 0 to 1: 0|-,
Transition from 1 to 2: 1|-,
Transition from 2 to 3: 0|-]
>>> [A(w) for w in ([], [Integer(0)], [Integer(0), Integer(1)], [Integer(0),
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(0)])]
[False, False, False, False, True]
```

If the input alphabet is not given, it is derived from the given word.

```
sage: A.input_alphabet
[0, 1]
sage: A = automata.Word([0, 1, 0], input_alphabet=[0, 1, 2])
sage: A.input_alphabet
[0, 1, 2]
```

```
>>> from sage.all import *
>>> A.input_alphabet
[0, 1]
>>> A = automata.Word([Integer(0), Integer(1), Integer(0)], input_
↪alphabet=[Integer(0), Integer(1), Integer(2)])
>>> A.input_alphabet
[0, 1, 2]
```

See also

AnyWord(), *ContainsWord()*.

class sage.combinat.finite_state_machine_generators.**TransducerGenerators**

Bases: object

A collection of constructors for several common transducers.

A list of all transducers in this database is available via tab completion. Type "transducers." and then hit tab to see which transducers are available.

The transducers currently in this class include:

- *Identity()*
- *abs()*
- *operator()*
- *all()*
- *any()*
- *add()*
- *sub()*
- *CountSubblockOccurrences()*
- *Wait()*
- *GrayCode()*
- *Recursion()*

CountSubblockOccurrences(*block*, *input_alphabet*)

Return a transducer counting the number of (possibly overlapping) occurrences of a block in the input.

INPUT:

- block – list (or other iterable) of letters
- input_alphabet – list or other iterable

OUTPUT:

A transducer counting (in unary) the number of occurrences of the given block in the input. Overlapping occurrences are counted several times.

Denoting the block by $b_0 \dots b_{k-1}$, the input word by $i_0 \dots i_L$ and the output word by $o_0 \dots o_L$, we have $o_j = 1$ if and only if $i_{j-k+1} \dots i_j = b_0 \dots b_{k-1}$. Otherwise, $o_j = 0$.

EXAMPLES:

1. Counting the number of `10` blocks over the alphabet `[0, 1]`:

```
sage: T = transducers.CountSubblockOccurrences(
....:     [1, 0],
....:     [0, 1])
sage: sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|1,
 Transition from (1,) to (1,): 1|0]
sage: T.input_alphabet
[0, 1]
sage: T.output_alphabet
[0, 1]
sage: T.initial_states()
[()]
sage: T.final_states()
[(), (1,)]
```

```
>>> from sage.all import *
>>> T = transducers.CountSubblockOccurrences(
...     [Integer(1), Integer(0)],
...     [Integer(0), Integer(1)])
>>> sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|1,
 Transition from (1,) to (1,): 1|0]
>>> T.input_alphabet
[0, 1]
>>> T.output_alphabet
[0, 1]
>>> T.initial_states()
[()]
>>> T.final_states()
[(), (1,)]
```

Check some sequence:

```
sage: T([0, 1, 0, 1, 1, 0])
[0, 0, 1, 0, 0, 1]
```

```
>>> from sage.all import *
>>> T([Integer(0), Integer(1), Integer(0), Integer(1), Integer(1),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)])
[0, 0, 1, 0, 0, 1]
```

2. Counting the number of `11` blocks over the alphabet `[0, 1]`:

```
sage: T = transducers.CountSubblockOccurrences(
....:     [1, 1],
....:     [0, 1])
sage: sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|0,
 Transition from (1,) to (1,): 1|1]
```

```
>>> from sage.all import *
>>> T = transducers.CountSubblockOccurrences(
...     [Integer(1), Integer(1)],
...     [Integer(0), Integer(1)])
>>> sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|0,
 Transition from (1,) to (1,): 1|1]
```

Check some sequence:

```
sage: T([0, 1, 0, 1, 1, 0])
[0, 0, 0, 0, 1, 0]
```

```
>>> from sage.all import *
>>> T([Integer(0), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(0)])
[0, 0, 0, 0, 1, 0]
```

3. Counting the number of `1010` blocks over the alphabet `[0, 1, 2]`:

```
sage: T = transducers.CountSubblockOccurrences(
....:     [1, 0, 1, 0],
....:     [0, 1, 2])
sage: sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from () to (): 2|0,
 Transition from (1,) to (1, 0): 0|0,
 Transition from (1,) to (1,): 1|0,
 Transition from (1,) to (): 2|0,
 Transition from (1, 0) to (): 0|0,
 Transition from (1, 0) to (1, 0, 1): 1|0,
 Transition from (1, 0) to (): 2|0,
 Transition from (1, 0, 1) to (1, 0): 0|1,
 Transition from (1, 0, 1) to (1,): 1|0,
 Transition from (1, 0, 1) to (): 2|0]
```

(continues on next page)

(continued from previous page)

```
sage: input =  [0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 2]
sage: output = [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0]
sage: T(input) == output
True
```

```
>>> from sage.all import *
>>> T = transducers.CountSubblockOccurrences(
...     [Integer(1), Integer(0), Integer(1), Integer(0)],
...     [Integer(0), Integer(1), Integer(2)])
>>> sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from () to (): 2|0,
 Transition from (1,) to (1, 0): 0|0,
 Transition from (1,) to (1,): 1|0,
 Transition from (1,) to (): 2|0,
 Transition from (1, 0) to (): 0|0,
 Transition from (1, 0) to (1, 0, 1): 1|0,
 Transition from (1, 0) to (): 2|0,
 Transition from (1, 0, 1) to (1, 0): 0|1,
 Transition from (1, 0, 1) to (1,): 1|0,
 Transition from (1, 0, 1) to (): 2|0]
>>> input =  [Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(2)]
>>> output = [Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(0)]
>>> T(input) == output
True
```

See also:

ContainsWord()

GrayCode()

Return a transducer converting the standard binary expansion to Gray code.

OUTPUT: a transducer

Cf. the Wikipedia article Gray_code for a description of the Gray code.

EXAMPLES:

```
sage: G = transducers.GrayCode()
sage: G
Transducer with 3 states
sage: for v in srange(10):
....:     print("{} {}".format(v, G(v.digits(base=2))))
0 []
1 [1]
2 [1, 1]
```

(continues on next page)

(continued from previous page)

```
3 [0, 1]
4 [0, 1, 1]
5 [1, 1, 1]
6 [1, 0, 1]
7 [0, 0, 1]
8 [0, 0, 1, 1]
9 [1, 0, 1, 1]
```

```
>>> from sage.all import *
>>> G = transducers.GrayCode()
>>> G
Transducer with 3 states
>>> for v in srange(Integer(10)):
...     print("{} {}".format(v, G(v.digits(base=Integer(2)))))
0 []
1 [1]
2 [1, 1]
3 [0, 1]
4 [0, 1, 1]
5 [1, 1, 1]
6 [1, 0, 1]
7 [0, 0, 1]
8 [0, 0, 1, 1]
9 [1, 0, 1, 1]
```

In the example *Gray Code* in the documentation of the *Finite state machines, automata, transducers* module, the Gray code transducer is derived from the algorithm converting the binary expansion to the Gray code. The result is the same as the one given here.

Identity(*input_alphabet*)

Return the identity transducer realizing the identity map.

INPUT:

- `input_alphabet` – list or other iterable

OUTPUT: a transducer mapping each word over `input_alphabet` to itself

EXAMPLES:

```
sage: T = transducers.Identity([0, 1])
sage: sorted(T.transitions())
[Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T.input_alphabet
[0, 1]
sage: T.output_alphabet
[0, 1]
sage: T([0, 1, 0, 1, 1])
[0, 1, 0, 1, 1]
```

```
>>> from sage.all import *
>>> T = transducers.Identity([Integer(0), Integer(1)])
>>> sorted(T.transitions())
[Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
>>> T.initial_states()
[0]
>>> T.final_states()
[0]
>>> T.input_alphabet
[0, 1]
>>> T.output_alphabet
[0, 1]
>>> T([Integer(0), Integer(1), Integer(0), Integer(1), Integer(1)])
[0, 1, 0, 1, 1]
```

Recursion(*recursions, base, function=None, var=None, input_alphabet=None, word_function=None, is_zero=None, output_rings=[Integer Ring, Rational Field]*)

Return a transducer realizing the given recursion when reading the digit expansion with base `base`.

INPUT:

- `recursions` – list or iterable of equations. Each equation has either the form
  - `f(base^K * n + r) == f(base^k * n + s) + t` for some integers `0 <= k < K`, `r` and some `t`—valid for all `n` such that the arguments on both sides are nonnegative—

  or the form
  - `f(r) == t` for some integer `r` and some `t`.

  Alternatively, an equation may be replaced by a `transducers.RecursionRule` with the attributes `K`, `r`, `k`, `s`, `t` as above or a tuple `(r, t)`. Note that `t` *must* be a list in this case.
- `base` – base of the digit expansion
- `function` – symbolic function `f` occurring in the recursions
- `var` – symbolic variable
- `input_alphabet` – (default: `None`) a list of digits to be used as the input alphabet. If `None` and the base is an integer, `input_alphabet` is chosen to be `srange(base.abs())`.
- `word_function` – (default: `None`) a symbolic function. If not `None`, `word_function(arg1, ..., argn)` in a symbolic recurrence relation is interpreted as a transition with output `[arg1, ..., argn]`. This could not be entered in a symbolic recurrence relation because lists do not coerce into the `SymbolicRing`.
- `is_zero` – (default: `None`) a callable. The recursion relations are only well-posed if there is no cycle with nonzero output and input consisting of zeros. This parameter is used to determine whether the output of such a cycle is nonzero. By default, the output must evaluate to `False` as a boolean.
- `output_rings` – (default: `[ZZ, QQ]`) a list of rings. The output labels are converted into the first ring of the list in which they are contained. If they are not contained in any ring, they remain in whatever ring they are after parsing the recursions, typically the symbolic ring.

OUTPUT: a transducer `T`

The transducer is constructed such that `T(expansion) == f(n)` if `expansion` is the digit expansion of `n` to the base `base` with the given input alphabet as set of digits. Here, the `+` on the right hand side of the recurrence relation is interpreted as the concatenation of words.

The formal equations and initial conditions in the recursion have to be selected such that `f` is uniquely defined.

EXAMPLES:

- The following example computes the Hamming weight of the ternary expansion of integers.

```
sage: # needs sage.symbolic
sage: function('f')
f
sage: var('n')
n
sage: T = transducers.Recursion([
....:     f(3*n + 1) == f(n) + 1,
....:     f(3*n + 2) == f(n) + 1,
....:     f(3*n) == f(n),
....:     f(0) == 0],
....:     3, f, n)
sage: T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
 Transition from (0, 0) to (0, 0): 1|1,
 Transition from (0, 0) to (0, 0): 2|1]
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> function('f')
f
>>> var('n')
n
>>> T = transducers.Recursion([
...     f(Integer(3)*n + Integer(1)) == f(n) + Integer(1),
...     f(Integer(3)*n + Integer(2)) == f(n) + Integer(1),
...     f(Integer(3)*n) == f(n),
...     f(Integer(0)) == Integer(0)],
...     Integer(3), f, n)
>>> T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
 Transition from (0, 0) to (0, 0): 1|1,
 Transition from (0, 0) to (0, 0): 2|1]
```

To illustrate what this transducer does, we consider the example of $n = 601$:

```
sage: # needs sage.symbolic
sage: ternary_expansion = 601.digits(base=3)
sage: ternary_expansion
[1, 2, 0, 1, 1, 2]
sage: weight_sequence = T(ternary_expansion)
sage: weight_sequence
[1, 1, 1, 1, 1]
sage: sum(weight_sequence)
5
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> ternary_expansion = Integer(601).digits(base=Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> ternary_expansion
[1, 2, 0, 1, 1, 2]
>>> weight_sequence = T(ternary_expansion)
>>> weight_sequence
[1, 1, 1, 1, 1]
>>> sum(weight_sequence)
5
```

Note that the digit zero does not show up in the output because the equation `f(3*n) == f(n)` means that no output is added to `f(n)`.

- The following example computes the Hamming weight of the non-adjacent form, cf. the Wikipedia article Non-adjacent_form.

```
sage: # needs sage.symbolic
sage: function('f')
f
sage: var('n')
n
sage: T = transducers.Recursion([
....:     f(4*n + 1) == f(n) + 1,
....:     f(4*n - 1) == f(n) + 1,
....:     f(2*n) == f(n),
....:     f(0) == 0],
....:     2, f, n)
sage: T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (0, 0): 0|1,
 Transition from (1, 1) to (1, 0): 1|1,
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 0) to (1, 0): 1|-]
sage: [(s.label(), s.final_word_out)
....:  for s in T.iter_final_states()]
[((0, 0), []),
 ((1, 1), [1]),
 ((1, 0), [1])]
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> function('f')
f
>>> var('n')
n
>>> T = transducers.Recursion([
...     f(Integer(4)*n + Integer(1)) == f(n) + Integer(1),
...     f(Integer(4)*n - Integer(1)) == f(n) + Integer(1),
...     f(Integer(2)*n) == f(n),
...     f(Integer(0)) == Integer(0)],
...     Integer(2), f, n)
>>> T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
```

(continues on next page)

(continued from previous page)

```
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (0, 0): 0|1,
 Transition from (1, 1) to (1, 0): 1|1,
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 0) to (1, 0): 1|-]
>>> [(s.label(), s.final_word_out)
...  for s in T.iter_final_states()]
[((0, 0), []),
 ((1, 1), [1]),
 ((1, 0), [1])]
```

As we are interested in the weight only, we also output 1 for numbers congruent to 3 mod 4. The actual expansion is computed in the next example.

Consider the example of $29 = (100\bar{1}01)_2$ (as usual, the digit -1 is denoted by $\bar{1}$ and digits are written from the most significant digit at the left to the least significant digit at the right; for the transducer, we have to give the digits in the reverse order):

```
sage: NAF = [1, 0, -1, 0, 0, 1]
sage: ZZ(NAF, base=2)
29
sage: binary_expansion = 29.digits(base=2)
sage: binary_expansion
[1, 0, 1, 1, 1]
sage: T(binary_expansion)                                                    #␣
↪needs sage.symbolic
[1, 1, 1]
sage: sum(T(binary_expansion))                                               #␣
↪needs sage.symbolic
3
```

```
>>> from sage.all import *
>>> NAF = [Integer(1), Integer(0), -Integer(1), Integer(0), Integer(0),␣
↪Integer(1)]
>>> ZZ(NAF, base=Integer(2))
29
>>> binary_expansion = Integer(29).digits(base=Integer(2))
>>> binary_expansion
[1, 0, 1, 1, 1]
>>> T(binary_expansion)                                                    #␣
↪needs sage.symbolic
[1, 1, 1]
>>> sum(T(binary_expansion))                                               #␣
↪needs sage.symbolic
3
```

Indeed, the given non-adjacent form has three nonzero digits.

- The following example computes the non-adjacent form from the binary expansion, cf. the Wikipedia article Non-adjacent_form. In contrast to the previous example, we actually compute the expansion, not only the weight.

  We have to write the output 0 when converting an even number. This cannot be encoded directly by an equation in the symbolic ring, because `f(2*n) == f(n) + 0` would be equivalent to `f(2*n) ==`

f(n) and an empty output would be written. Therefore, we wrap the output in the symbolic function w and use the parameter word_function to announce this.

Similarly, we use w(-1, 0) to write an output word of length 2 in one iteration. Finally, we write f(0) == w() to write an empty word upon completion.

Moreover, there is a cycle with output [0] which—from the point of view of this method—is a contradicting recursion. We override this by the parameter is_zero.

```
sage: # needs sage.symbolic
sage: var('n')
n
sage: function('f w')
(f, w)
sage: T = transducers.Recursion([
....:       f(2*n) == f(n) + w(0),
....:       f(4*n + 1) == f(n) + w(1, 0),
....:       f(4*n - 1) == f(n) + w(-1, 0),
....:       f(0) == w()],
....:       2, f, n,
....:       word_function=w,
....:       is_zero=lambda x: sum(x).is_zero())
sage: T.transitions()
[Transition from (0, 0) to (0, 0): 0|0,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (0, 0): 0|1,0,
 Transition from (1, 1) to (1, 0): 1|-1,0,
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 0) to (1, 0): 1|0]
sage: for s in T.iter_states():
....:     print("{} {}".format(s, s.final_word_out))
(0, 0) []
(1, 1) [1, 0]
(1, 0) [1, 0]
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> var('n')
n
>>> function('f w')
(f, w)
>>> T = transducers.Recursion([
...       f(Integer(2)*n) == f(n) + w(Integer(0)),
...       f(Integer(4)*n + Integer(1)) == f(n) + w(Integer(1), Integer(0)),
...       f(Integer(4)*n - Integer(1)) == f(n) + w(-Integer(1),␣
↪Integer(0)),
...       f(Integer(0)) == w()],
...       Integer(2), f, n,
...       word_function=w,
...       is_zero=lambda x: sum(x).is_zero())
>>> T.transitions()
[Transition from (0, 0) to (0, 0): 0|0,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (0, 0): 0|1,0,
```

(continues on next page)

(continued from previous page)

```
 Transition from (1, 1) to (1, 0): 1|-1,0,
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 0) to (1, 0): 1|0]
>>> for s in T.iter_states():
...     print("{} {}".format(s, s.final_word_out))
(0, 0) []
(1, 1) [1, 0]
(1, 0) [1, 0]
```

We again consider the example of $n = 29$:

```
sage: T(29.digits(base=2))                                              #␣
↪needs sage.symbolic
[1, 0, -1, 0, 0, 1, 0]
```

```
>>> from sage.all import *
>>> T(Integer(29).digits(base=Integer(2)))                               ␣
↪              # needs sage.symbolic
[1, 0, -1, 0, 0, 1, 0]
```

The same transducer can also be entered bypassing the symbolic equations:

```
sage: R = transducers.RecursionRule
sage: TR = transducers.Recursion([
....:       R(K=1, r=0, k=0, s=0, t=[0]),
....:       R(K=2, r=1, k=0, s=0, t=[1, 0]),
....:       R(K=2, r=-1, k=0, s=0, t=[-1, 0]),
....:       (0, [])],
....:       2,
....:       is_zero=lambda x: sum(x).is_zero())
sage: TR == T                                                           #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> R = transducers.RecursionRule
>>> TR = transducers.Recursion([
...       R(K=Integer(1), r=Integer(0), k=Integer(0), s=Integer(0),␣
↪t=[Integer(0)]),
...       R(K=Integer(2), r=Integer(1), k=Integer(0), s=Integer(0),␣
↪t=[Integer(1), Integer(0)]),
...       R(K=Integer(2), r=-Integer(1), k=Integer(0), s=Integer(0), t=[-
↪Integer(1), Integer(0)]),
...       (Integer(0), [])],
...       Integer(2),
...       is_zero=lambda x: sum(x).is_zero())
>>> TR == T                                                           #␣
↪needs sage.symbolic
True
```

- Here is an artificial example where some of the s are negative:

```
sage: # needs sage.symbolic
sage: function('f')
f
sage: var('n')
n
sage: T = transducers.Recursion([
....:     f(2*n + 1) == f(n-1) + 1,
....:     f(2*n) == f(n),
....:     f(1) == 1,
....:     f(0) == 0], 2, f, n)
sage: T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (-1, 1): 0|1,
 Transition from (1, 1) to (0, 0): 1|1,
 Transition from (-1, 1) to (-1, 2): 0|-,
 Transition from (-1, 1) to (1, 2): 1|-,
 Transition from (-1, 2) to (-1, 1): 0|1,
 Transition from (-1, 2) to (0, 0): 1|1,
 Transition from (1, 2) to (-1, 2): 0|1,
 Transition from (1, 2) to (1, 2): 1|1]
sage: [(s.label(), s.final_word_out)
....:  for s in T.iter_final_states()]
[((0, 0), []),
 ((1, 1), [1]),
 ((-1, 1), [0]),
 ((-1, 2), [0]),
 ((1, 2), [1])]
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> function('f')
f
>>> var('n')
n
>>> T = transducers.Recursion([
...     f(Integer(2)*n + Integer(1)) == f(n-Integer(1)) + Integer(1),
...     f(Integer(2)*n) == f(n),
...     f(Integer(1)) == Integer(1),
...     f(Integer(0)) == Integer(0)], Integer(2), f, n)
>>> T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (-1, 1): 0|1,
 Transition from (1, 1) to (0, 0): 1|1,
 Transition from (-1, 1) to (-1, 2): 0|-,
 Transition from (-1, 1) to (1, 2): 1|-,
 Transition from (-1, 2) to (-1, 1): 0|1,
 Transition from (-1, 2) to (0, 0): 1|1,
 Transition from (1, 2) to (-1, 2): 0|1,
 Transition from (1, 2) to (1, 2): 1|1]
>>> [(s.label(), s.final_word_out)
```

(continues on next page)

(continued from previous page)

```
...   for s in T.iter_final_states()]
[((0, 0), []),
 ((1, 1), [1]),
 ((-1, 1), [0]),
 ((-1, 2), [0]),
 ((1, 2), [1])]
```

- Abelian complexity of the paperfolding sequence (cf. [HKP2015], Example 2.8):

```
sage: # needs sage.symbolic
sage: T = transducers.Recursion([
....:     f(4*n) == f(2*n),
....:     f(4*n+2) == f(2*n+1)+1,
....:     f(16*n+1) == f(8*n+1),
....:     f(16*n+5) == f(4*n+1)+2,
....:     f(16*n+11) == f(4*n+3)+2,
....:     f(16*n+15) == f(2*n+2)+1,
....:     f(1) == 2, f(0) == 0]
....:     + [f(16*n+jj) == f(2*n+1)+2 for jj in [3,7,9,13]],
....:     2, f, n)
sage: T.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (0, 1) to (0, 1): 0|-,
 Transition from (0, 1) to (1, 1): 1|1,
 Transition from (1, 1) to (1, 2): 0|-,
 Transition from (1, 1) to (3, 2): 1|-,
 Transition from (1, 2) to (1, 3): 0|-,
 Transition from (1, 2) to (5, 3): 1|-,
 Transition from (3, 2) to (3, 3): 0|-,
 Transition from (3, 2) to (7, 3): 1|-,
 Transition from (1, 3) to (1, 3): 0|-,
 Transition from (1, 3) to (1, 1): 1|2,
 Transition from (5, 3) to (1, 2): 0|2,
 Transition from (5, 3) to (1, 1): 1|2,
 Transition from (3, 3) to (1, 1): 0|2,
 Transition from (3, 3) to (3, 2): 1|2,
 Transition from (7, 3) to (1, 1): 0|2,
 Transition from (7, 3) to (2, 1): 1|1,
 Transition from (2, 1) to (1, 1): 0|1,
 Transition from (2, 1) to (2, 1): 1|-]
sage: for s in T.iter_states():
....:     print("{} {}".format(s, s.final_word_out))
(0, 0) []
(0, 1) []
(1, 1) [2]
(1, 2) [2]
(3, 2) [2, 2]
(1, 3) [2]
(5, 3) [2, 2]
(3, 3) [2, 2]
(7, 3) [2, 2]
```

(continues on next page)

(continued from previous page)

```
(2, 1) [1, 2]
sage: list(sum(T(n.bits())) for n in srange(1, 21))
[2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6, 5, 4, 5, 4, 3, 4, 5, 6, 5]
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> T = transducers.Recursion([
...     f(Integer(4)*n) == f(Integer(2)*n),
...     f(Integer(4)*n+Integer(2)) ==␣
↪f(Integer(2)*n+Integer(1))+Integer(1),
...     f(Integer(16)*n+Integer(1)) == f(Integer(8)*n+Integer(1)),
...     f(Integer(16)*n+Integer(5)) ==␣
↪f(Integer(4)*n+Integer(1))+Integer(2),
...     f(Integer(16)*n+Integer(11)) ==␣
↪f(Integer(4)*n+Integer(3))+Integer(2),
...     f(Integer(16)*n+Integer(15)) ==␣
↪f(Integer(2)*n+Integer(2))+Integer(1),
...     f(Integer(1)) == Integer(2), f(Integer(0)) == Integer(0)]
...     + [f(Integer(16)*n+jj) == f(Integer(2)*n+Integer(1))+Integer(2)␣
↪for jj in [Integer(3),Integer(7),Integer(9),Integer(13)]],
...     Integer(2), f, n)
>>> T.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (0, 1) to (0, 1): 0|-,
 Transition from (0, 1) to (1, 1): 1|1,
 Transition from (1, 1) to (1, 2): 0|-,
 Transition from (1, 1) to (3, 2): 1|-,
 Transition from (1, 2) to (1, 3): 0|-,
 Transition from (1, 2) to (5, 3): 1|-,
 Transition from (3, 2) to (3, 3): 0|-,
 Transition from (3, 2) to (7, 3): 1|-,
 Transition from (1, 3) to (1, 3): 0|-,
 Transition from (1, 3) to (1, 1): 1|2,
 Transition from (5, 3) to (1, 2): 0|2,
 Transition from (5, 3) to (1, 1): 1|2,
 Transition from (3, 3) to (1, 1): 0|2,
 Transition from (3, 3) to (3, 2): 1|2,
 Transition from (7, 3) to (1, 1): 0|2,
 Transition from (7, 3) to (2, 1): 1|1,
 Transition from (2, 1) to (1, 1): 0|1,
 Transition from (2, 1) to (2, 1): 1|-]
>>> for s in T.iter_states():
...     print("{} {}".format(s, s.final_word_out))
(0, 0) []
(0, 1) []
(1, 1) [2]
(1, 2) [2]
(3, 2) [2, 2]
(1, 3) [2]
(5, 3) [2, 2]
(3, 3) [2, 2]
```

(continues on next page)

(continued from previous page)

```
(7, 3) [2, 2]
(2, 1) [1, 2]
>>> list(sum(T(n.bits())) for n in srange(Integer(1), Integer(21)))
[2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6, 5, 4, 5, 4, 3, 4, 5, 6, 5]
```

- We now demonstrate the use of the `output_rings` parameter. If no `output_rings` are specified, the output labels are converted into `ZZ`:

```
sage: # needs sage.symbolic
sage: function('f')
f
sage: var('n')
n
sage: T = transducers.Recursion([
....:     f(2*n + 1) == f(n) + 1,
....:     f(2*n) == f(n),
....:     f(0) == 2],
....:     2, f, n)
sage: for t in T.transitions():
....:     print([x.parent() for x in t.word_out])
[]
[Integer Ring]
sage: [x.parent() for x in T.states()[0].final_word_out]
[Integer Ring]
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> function('f')
f
>>> var('n')
n
>>> T = transducers.Recursion([
...     f(Integer(2)*n + Integer(1)) == f(n) + Integer(1),
...     f(Integer(2)*n) == f(n),
...     f(Integer(0)) == Integer(2)],
...     Integer(2), f, n)
>>> for t in T.transitions():
...     print([x.parent() for x in t.word_out])
[]
[Integer Ring]
>>> [x.parent() for x in T.states()[Integer(0)].final_word_out]
[Integer Ring]
```

In contrast, if `output_rings` is set to the empty list, the results are not converted:

```
sage: T = transducers.Recursion([                                        #␣
↪needs sage.symbolic
....:     f(2*n + 1) == f(n) + 1,
....:     f(2*n) == f(n),
....:     f(0) == 2],
....:     2, f, n, output_rings=[])
sage: for t in T.transitions():                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
....:     print([x.parent() for x in t.word_out])
[]
[Symbolic Ring]
sage: [x.parent() for x in T.states()[0].final_word_out]                    #␣
↪needs sage.symbolic
[Symbolic Ring]
```

```
>>> from sage.all import *
>>> T = transducers.Recursion([                                           #␣
↪needs sage.symbolic
...     f(Integer(2)*n + Integer(1)) == f(n) + Integer(1),
...     f(Integer(2)*n) == f(n),
...     f(Integer(0)) == Integer(2)],
...     Integer(2), f, n, output_rings=[])
>>> for t in T.transitions():                                             #␣
↪needs sage.symbolic
...     print([x.parent() for x in t.word_out])
[]
[Symbolic Ring]
>>> [x.parent() for x in T.states()[Integer(0)].final_word_out]              ␣
↪       # needs sage.symbolic
[Symbolic Ring]
```

Finally, we use a somewhat questionable conversion:

```
sage: T = transducers.Recursion([                                           #␣
↪needs sage.rings.finite_rings sage.symbolic
....:     f(2*n + 1) == f(n) + 1,
....:     f(2*n) == f(n),
....:     f(0) == 0],
....:     2, f, n, output_rings=[GF(5)])
sage: for t in T.transitions():                                             #␣
↪needs sage.rings.finite_rings sage.symbolic
....:     print([x.parent() for x in t.word_out])
[]
[Finite Field of size 5]
```

```
>>> from sage.all import *
>>> T = transducers.Recursion([                                           #␣
↪needs sage.rings.finite_rings sage.symbolic
...     f(Integer(2)*n + Integer(1)) == f(n) + Integer(1),
...     f(Integer(2)*n) == f(n),
...     f(Integer(0)) == Integer(0)],
...     Integer(2), f, n, output_rings=[GF(Integer(5))])
>>> for t in T.transitions():                                             #␣
↪needs sage.rings.finite_rings sage.symbolic
...     print([x.parent() for x in t.word_out])
[]
[Finite Field of size 5]
```

Todo

Extend the method to

- non-integral bases,
- higher dimensions.

ALGORITHM:

See [HKP2015], Section 6. However, there are also recursion transitions for states of level $< \kappa$ if the recursion rules allow such a transition. Furthermore, the intermediate step of a non-deterministic transducer is left out by implicitly using recursion transitions. The well-posedness is checked in a truncated version of the recursion digraph.

class RecursionRule(*K*, *r*, *k*, *s*, *t*)

Bases: `tuple`

K

Alias for field number 0

k

Alias for field number 2

r

Alias for field number 1

s

Alias for field number 3

t

Alias for field number 4

Wait(*input_alphabet*, *threshold=1*)

Writes `False` until reading the `threshold`-th occurrence of a true input letter; then writes `True`.

INPUT:

- `input_alphabet` – list or other iterable
- `threshold` – positive integer specifying how many occurrences of `True` inputs are waited for

OUTPUT:

A transducer writing `False` until the `threshold`-th true (Python's standard conversion to boolean is used to convert the actual input to boolean) input is read. Subsequently, the transducer writes `True`.

EXAMPLES:

```
sage: T = transducers.Wait([0, 1])
sage: T([0, 0, 1, 0, 1, 0])
[False, False, True, True, True, True]
sage: T2 = transducers.Wait([0, 1], threshold=2)
sage: T2([0, 0, 1, 0, 1, 0])
[False, False, False, False, True, True]
```

```
>>> from sage.all import *
>>> T = transducers.Wait([Integer(0), Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> T([Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),
↪Integer(0)])
[False, False, True, True, True, True]
>>> T2 = transducers.Wait([Integer(0), Integer(1)], threshold=Integer(2))
>>> T2([Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),
↪Integer(0)])
[False, False, False, False, True, True]
```

abs(*input_alphabet*)

Return a transducer which realizes the letter-wise absolute value of an input word over the given input alphabet.

INPUT:

- `input_alphabet` – list or other iterable

OUTPUT:

A transducer mapping $i_0 \dots i_k$ to $|i_0| \dots |i_k|$.

EXAMPLES:

The following transducer realizes letter-wise absolute value:

```
sage: T = transducers.abs([-1, 0, 1])
sage: T.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([-1, -1, 0, 1])
[1, 1, 0, 1]
```

```
>>> from sage.all import *
>>> T = transducers.abs([-Integer(1), Integer(0), Integer(1)])
>>> T.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
>>> T.initial_states()
[0]
>>> T.final_states()
[0]
>>> T([-Integer(1), -Integer(1), Integer(0), Integer(1)])
[1, 1, 0, 1]
```

add(*input_alphabet*, *number_of_operands=2*)

Return a transducer which realizes addition on pairs over the given input alphabet.

INPUT:

- `input_alphabet` – list or other iterable
- `number_of_operands` – (default: 2) it specifies the number of input arguments the operator takes

OUTPUT:

A transducer mapping an input word $(i_{01}, \ldots, i_{0d}) \ldots (i_{k1}, \ldots, i_{kd})$ to the word $(i_{01} + \cdots + i_{0d}) \ldots (i_{k1} + \cdots + i_{kd})$.

The input alphabet of the generated transducer is the Cartesian product of `number_of_operands` copies of `input_alphabet`.

EXAMPLES:

The following transducer realizes letter-wise addition:

```
sage: T = transducers.add([0, 1])
sage: T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|2]
sage: T.input_alphabet
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(0, 0), (0, 1), (1, 0), (1, 1)])
[0, 1, 1, 2]
```

```
>>> from sage.all import *
>>> T = transducers.add([Integer(0), Integer(1)])
>>> T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|2]
>>> T.input_alphabet
[(0, 0), (0, 1), (1, 0), (1, 1)]
>>> T.initial_states()
[0]
>>> T.final_states()
[0]
>>> T([(Integer(0), Integer(0)), (Integer(0), Integer(1)), (Integer(1),
↪Integer(0)), (Integer(1), Integer(1))])
[0, 1, 1, 2]
```

More than two operands can also be handled:

```
sage: T3 = transducers.add([0, 1], number_of_operands=3)
sage: T3.input_alphabet
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),
 (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
sage: T3([(0, 0, 0), (0, 1, 0), (0, 1, 1), (1, 1, 1)])
[0, 1, 2, 3]
```

```
>>> from sage.all import *
>>> T3 = transducers.add([Integer(0), Integer(1)], number_of_
```

(continues on next page)

(continued from previous page)

```
↪operands=Integer(3))
>>> T3.input_alphabet
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),
 (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> T3([(Integer(0), Integer(0), Integer(0)), (Integer(0), Integer(1),␣
↪Integer(0)), (Integer(0), Integer(1), Integer(1)), (Integer(1), Integer(1),␣
↪Integer(1))])
[0, 1, 2, 3]
```

all(*input_alphabet*, *number_of_operands=2*)

Return a transducer which realizes logical `and` over the given input alphabet.

INPUT:

- `input_alphabet` – list or other iterable
- `number_of_operands` – (default: 2) specifies the number of input arguments for the `and` operation

OUTPUT:

A transducer mapping an input word $(i_{01}, \dots, i_{0d}) \dots (i_{k1}, \dots, i_{kd})$ to the word $(i_{01} \wedge \dots \wedge i_{0d}) \dots (i_{k1} \wedge \dots \wedge i_{kd})$.

The input alphabet of the generated transducer is the Cartesian product of `number_of_operands` copies of `input_alphabet`.

EXAMPLES:

The following transducer realizes letter-wise logical `and`:

```
sage: T = transducers.all([False, True])
sage: T.transitions()
[Transition from 0 to 0: (False, False)|False,
 Transition from 0 to 0: (False, True)|False,
 Transition from 0 to 0: (True, False)|False,
 Transition from 0 to 0: (True, True)|True]
sage: T.input_alphabet
[(False, False), (False, True), (True, False), (True, True)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(False, False), (False, True), (True, False), (True, True)])
[False, False, False, True]
```

```
>>> from sage.all import *
>>> T = transducers.all([False, True])
>>> T.transitions()
[Transition from 0 to 0: (False, False)|False,
 Transition from 0 to 0: (False, True)|False,
 Transition from 0 to 0: (True, False)|False,
 Transition from 0 to 0: (True, True)|True]
>>> T.input_alphabet
[(False, False), (False, True), (True, False), (True, True)]
>>> T.initial_states()
[0]
```

(continues on next page)

(continued from previous page)

```
>>> T.final_states()
[0]
>>> T([(False, False), (False, True), (True, False), (True, True)])
[False, False, False, True]
```

More than two operands and other input alphabets (with conversion to boolean) are also possible:

```
sage: T3 = transducers.all([0, 1], number_of_operands=3)
sage: T3([(0, 0, 0), (1, 0, 0), (1, 1, 1)])
[False, False, True]
```

```
>>> from sage.all import *
>>> T3 = transducers.all([Integer(0), Integer(1)], number_of_
↪operands=Integer(3))
>>> T3([(Integer(0), Integer(0), Integer(0)), (Integer(1), Integer(0),␣
↪Integer(0)), (Integer(1), Integer(1), Integer(1))])
[False, False, True]
```

any(*input_alphabet*, *number_of_operands=2*)

Return a transducer which realizes logical `or` over the given input alphabet.

INPUT:

- `input_alphabet` – list or other iterable
- `number_of_operands` – (default: 2) specifies the number of input arguments for the `or` operation

OUTPUT:

A transducer mapping an input word $(i_{01},\ldots,i_{0d})\ldots(i_{k1},\ldots,i_{kd})$ to the word $(i_{01}\vee\cdots\vee i_{0d})\ldots(i_{k1}\vee\cdots\vee i_{kd})$.

The input alphabet of the generated transducer is the Cartesian product of `number_of_operands` copies of `input_alphabet`.

EXAMPLES:

The following transducer realizes letter-wise logical `or`:

```
sage: T = transducers.any([False, True])
sage: T.transitions()
[Transition from 0 to 0: (False, False)|False,
 Transition from 0 to 0: (False, True)|True,
 Transition from 0 to 0: (True, False)|True,
 Transition from 0 to 0: (True, True)|True]
sage: T.input_alphabet
[(False, False), (False, True), (True, False), (True, True)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(False, False), (False, True), (True, False), (True, True)])
[False, True, True, True]
```

```
>>> from sage.all import *
>>> T = transducers.any([False, True])
```

(continues on next page)

(continued from previous page)

```
>>> T.transitions()
[Transition from 0 to 0: (False, False)|False,
 Transition from 0 to 0: (False, True)|True,
 Transition from 0 to 0: (True, False)|True,
 Transition from 0 to 0: (True, True)|True]
>>> T.input_alphabet
[(False, False), (False, True), (True, False), (True, True)]
>>> T.initial_states()
[0]
>>> T.final_states()
[0]
>>> T([(False, False), (False, True), (True, False), (True, True)])
[False, True, True, True]
```

More than two operands and other input alphabets (with conversion to boolean) are also possible:

```
sage: T3 = transducers.any([0, 1], number_of_operands=3)
sage: T3([(0, 0, 0), (1, 0, 0), (1, 1, 1)])
[False, True, True]
```

```
>>> from sage.all import *
>>> T3 = transducers.any([Integer(0), Integer(1)], number_of_
↪operands=Integer(3))
>>> T3([(Integer(0), Integer(0), Integer(0)), (Integer(1), Integer(0),␣
↪Integer(0)), (Integer(1), Integer(1), Integer(1))])
[False, True, True]
```

map(*f*, *input_alphabet*)

Return a transducer which realizes a function on the alphabet.

INPUT:

- `f` – function to realize
- `input_alphabet` – list or other iterable

OUTPUT:

A transducer mapping an input letter x to $f(x)$.

EXAMPLES:

The following binary transducer realizes component-wise absolute value (this transducer is also available as `abs()`):

```
sage: T = transducers.map(abs, [-1, 0, 1])
sage: T.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
sage: T.input_alphabet
[-1, 0, 1]
sage: T.initial_states()
[0]
sage: T.final_states()
```

(continues on next page)

(continued from previous page)

```
[0]
sage: T([-1, 1, 0, 1])
[1, 1, 0, 1]
```

```
>>> from sage.all import *
>>> T = transducers.map(abs, [-Integer(1), Integer(0), Integer(1)])
>>> T.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
>>> T.input_alphabet
[-1, 0, 1]
>>> T.initial_states()
[0]
>>> T.final_states()
[0]
>>> T([-Integer(1), Integer(1), Integer(0), Integer(1)])
[1, 1, 0, 1]
```

See also:

Automaton.with_output().

operator(*operator*, *input_alphabet*, *number_of_operands=2*)

Return a transducer which realizes an operation on tuples over the given input alphabet.

INPUT:

- `operator` – operator to realize; it is a function which takes `number_of_operands` input arguments (each out of `input_alphabet`)
- `input_alphabet` – list or other iterable
- `number_of_operands` – (default: 2) it specifies the number of input arguments the operator takes

OUTPUT:

A transducer mapping an input letter $(i_1, \ldots, i_n)$ to operator$(i_1, \ldots, i_n)$. Here, n equals `number_of_operands`.

The input alphabet of the generated transducer is the Cartesian product of `number_of_operands` copies of `input_alphabet`.

EXAMPLES:

The following binary transducer realizes component-wise addition (this transducer is also available as *add()*):

```
sage: import operator
sage: T = transducers.operator(operator.add, [0, 1])
sage: T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|2]
sage: T.input_alphabet
```

(continues on next page)

(continued from previous page)

```
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(0, 0), (0, 1), (1, 0), (1, 1)])
[0, 1, 1, 2]
```

```
>>> from sage.all import *
>>> import operator
>>> T = transducers.operator(operator.add, [Integer(0), Integer(1)])
>>> T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|2]
>>> T.input_alphabet
[(0, 0), (0, 1), (1, 0), (1, 1)]
>>> T.initial_states()
[0]
>>> T.final_states()
[0]
>>> T([(Integer(0), Integer(0)), (Integer(0), Integer(1)), (Integer(1),␣
→Integer(0)), (Integer(1), Integer(1))])
[0, 1, 1, 2]
```

Note that for a unary operator the input letters of the new transducer are tuples of length 1:

```
sage: T = transducers.operator(abs,
....:                          [-1, 0, 1],
....:                          number_of_operands=1)
sage: T([-1, 1, 0])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T([(-1,), (1,), (0,)])
[1, 1, 0]
```

```
>>> from sage.all import *
>>> T = transducers.operator(abs,
...                          [-Integer(1), Integer(0), Integer(1)],
...                          number_of_operands=Integer(1))
>>> T([-Integer(1), Integer(1), Integer(0)])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
>>> T([(-Integer(1),), (Integer(1),), (Integer(0),)])
[1, 1, 0]
```

Compare this with the transducer generated by *map()*:

```
sage: T = transducers.map(abs,
....:                     [-1, 0, 1])
sage: T([-1, 1, 0])
[1, 1, 0]
```

```
>>> from sage.all import *
>>> T = transducers.map(abs,
...                     [-Integer(1), Integer(0), Integer(1)])
>>> T([-Integer(1), Integer(1), Integer(0)])
[1, 1, 0]
```

In fact, this transducer is also available as *abs()*:

```
sage: T = transducers.abs([-1, 0, 1])
sage: T([-1, 1, 0])
[1, 1, 0]
```

```
>>> from sage.all import *
>>> T = transducers.abs([-Integer(1), Integer(0), Integer(1)])
>>> T([-Integer(1), Integer(1), Integer(0)])
[1, 1, 0]
```

sub(*input_alphabet*)

Return a transducer which realizes subtraction on pairs over the given input alphabet.

INPUT:

- `input_alphabet` – list or other iterable

OUTPUT:

A transducer mapping an input word $(i_0, i'_0) \ldots (i_k, i'_k)$ to the word $(i_0 - i'_0) \ldots (i_k - i'_k)$.

The input alphabet of the generated transducer is the Cartesian product of two copies of `input_alphabet`.

EXAMPLES:

The following transducer realizes letter-wise subtraction:

```
sage: T = transducers.sub([0, 1])
sage: T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|-1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|0]
sage: T.input_alphabet
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(0, 0), (0, 1), (1, 0), (1, 1)])
[0, -1, 1, 0]
```

```
>>> from sage.all import *
>>> T = transducers.sub([Integer(0), Integer(1)])
>>> T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|-1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|0]
>>> T.input_alphabet
[(0, 0), (0, 1), (1, 0), (1, 1)]
>>> T.initial_states()
[0]
>>> T.final_states()
[0]
>>> T([(Integer(0), Integer(0)), (Integer(0), Integer(1)), (Integer(1),␣
↪Integer(0)), (Integer(1), Integer(1))])
[0, -1, 1, 0]
```

weight(*input_alphabet*, *zero=0*)

Return a transducer which realizes the Hamming weight of the input over the given input alphabet.

INPUT:

- `input_alphabet` – list or other iterable
- `zero` – the zero symbol in the alphabet used

OUTPUT:

A transducer mapping $i_0 \ldots i_k$ to $(i_0 \neq 0) \ldots (i_k \neq 0)$.

The Hamming weight is defined as the number of nonzero digits in the input sequence over the alphabet `input_alphabet` (see Wikipedia article Hamming_weight). The output sequence of the transducer is a unary encoding of the Hamming weight. Thus the sum of the output sequence is the Hamming weight of the input.

EXAMPLES:

```
sage: W = transducers.weight([-1, 0, 2])
sage: W.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 2|1]
sage: unary_weight = W([-1, 0, 0, 2, -1])
sage: unary_weight
[1, 0, 0, 1, 1]
sage: weight = add(unary_weight)
sage: weight
3
```

```
>>> from sage.all import *
>>> W = transducers.weight([-Integer(1), Integer(0), Integer(2)])
>>> W.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 2|1]
>>> unary_weight = W([-Integer(1), Integer(0), Integer(0), Integer(2), -
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)])
>>> unary_weight
[1, 0, 0, 1, 1]
>>> weight = add(unary_weight)
>>> weight
3
```

Also the joint Hamming weight can be computed:

```
sage: v1 = vector([-1, 0])
sage: v0 = vector([0, 0])
sage: W = transducers.weight([v1, v0])
sage: unary_weight = W([v1, v0, v1, v0])
sage: add(unary_weight)
2
```

```
>>> from sage.all import *
>>> v1 = vector([-Integer(1), Integer(0)])
>>> v0 = vector([Integer(0), Integer(0)])
>>> W = transducers.weight([v1, v0])
>>> unary_weight = W([v1, v0, v1, v0])
>>> add(unary_weight)
2
```

For the input alphabet `[-1, 0, 1]` the weight transducer is the same as the absolute value transducer *abs()*:

```
sage: W = transducers.weight([-1, 0, 1])
sage: A = transducers.abs([-1, 0, 1])
sage: W == A
True
```

```
>>> from sage.all import *
>>> W = transducers.weight([-Integer(1), Integer(0), Integer(1)])
>>> A = transducers.abs([-Integer(1), Integer(0), Integer(1)])
>>> W == A
True
```

For other input alphabets, we can specify the zero symbol:

```
sage: W = transducers.weight(['a', 'b'], zero='a')
sage: add(W(['a', 'b', 'b']))
2
```

```
>>> from sage.all import *
>>> W = transducers.weight(['a', 'b'], zero='a')
>>> add(W(['a', 'b', 'b']))
2
```

5.1.110 Free Quasi-symmetric functions

AUTHORS:

- Frédéric Chapoton, Darij Grinberg (2017)

class sage.combinat.fqsym.**FQSymBases**(*base*)

Bases: Category_realization_of_parent

The category of graded bases of $FQSym$ indexed by permutations.

class ElementMethods

Bases: object

omega_involution()

Return the image of the element self of $FQSym$ under the omega involution.

The ω involution is defined as the linear map $FQSym \to FQSym$ that sends each basis element F_u of the F-basis of $FQSym$ to the basis element $F_{u \circ w_0}$, where w_0 is the longest word (i.e., $w_0(i) = n+1-i$) in the symmetric group S_n that contains u. The ω involution is a graded algebra automorphism and a coalgebra anti-automorphism of $FQSym$. Every permutation $u \in S_n$ satisfies

$$\omega(F_u) = F_{u \circ w_0}, \qquad \omega(G_u) = G_{w_0 \circ u},$$

where standard notations for classical bases of $FQSym$ are being used (that is, F for the F-basis, and G for the G-basis). In other words, writing permutations in one-line notation, we have

$$\omega(F_{(u_1,u_2,\ldots,u_n)}) = F_{(u_n,u_{n-1},\ldots,u_1)}, \qquad \omega(G_{(u_1,u_2,\ldots,u_n)}) = G_{(n+1-u_1,n+1-u_2,\ldots,n+1-u_n)}.$$

If we also consider the ω involution (*omega_involution()*) of the quasisymmetric functions (by slight abuse of notation), and if we let π be the canonical projection $FQSym \to QSym$, then $\pi \circ \omega = \omega \circ \pi$.

Additionally, consider the ψ involution (*psi_involution()*) of the noncommutative symmetric functions, and if we let ι be the canonical inclusion $NSym \to FQSym$, then $\omega \circ \iota = \iota \circ \psi$.

Todo

Duality?

See also

psi_involution(), *star_involution()*

EXAMPLES:

```
sage: FQSym = algebras.FQSym(ZZ)
sage: F = FQSym.F()
sage: F[[2,3,1]].omega_involution()
F[1, 3, 2]
sage: (3*F[[1]] - 4*F[[]] + 5*F[[1,2]]).omega_involution()
-4*F[] + 3*F[1] + 5*F[2, 1]
sage: G = FQSym.G()
sage: G[[2,3,1]].omega_involution()
G[2, 1, 3]
sage: M = FQSym.M()
```

(continues on next page)

(continued from previous page)

```
sage: M[[2,3,1]].omega_involution()
-M[1, 2, 3] - M[2, 1, 3] - M[3, 1, 2]
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(ZZ)
>>> F = FQSym.F()
>>> F[[Integer(2),Integer(3),Integer(1)]].omega_involution()
F[1, 3, 2]
>>> (Integer(3)*F[[Integer(1)]] - Integer(4)*F[[]] +
↪Integer(5)*F[[Integer(1),Integer(2)]]).omega_involution()
-4*F[] + 3*F[1] + 5*F[2, 1]
>>> G = FQSym.G()
>>> G[[Integer(2),Integer(3),Integer(1)]].omega_involution()
G[2, 1, 3]
>>> M = FQSym.M()
>>> M[[Integer(2),Integer(3),Integer(1)]].omega_involution()
-M[1, 2, 3] - M[2, 1, 3] - M[3, 1, 2]
```

The omega involution is an algebra homomorphism:

```
sage: (F[1,2] * F[1]).omega_involution()
F[2, 1, 3] + F[2, 3, 1] + F[3, 2, 1]
sage: F[1,2].omega_involution() * F[1].omega_involution()
F[2, 1, 3] + F[2, 3, 1] + F[3, 2, 1]
```

```
>>> from sage.all import *
>>> (F[Integer(1),Integer(2)] * F[Integer(1)]).omega_involution()
F[2, 1, 3] + F[2, 3, 1] + F[3, 2, 1]
>>> F[Integer(1),Integer(2)].omega_involution() * F[Integer(1)].omega_
↪involution()
F[2, 1, 3] + F[2, 3, 1] + F[3, 2, 1]
```

The omega involution intertwines the antipode and the inverse of the antipode:

```
sage: all( F(I).antipode().omega_involution().antipode()
....:      == F(I).omega_involution()
....:      for I in Permutations(4) )
True
```

```
>>> from sage.all import *
>>> all( F(I).antipode().omega_involution().antipode()
...      == F(I).omega_involution()
...      for I in Permutations(Integer(4)) )
True
```

Testing the $\pi \circ \omega = \omega \circ \pi$ relation noticed above:

```
sage: all( M[I].omega_involution().to_qsym()
....:      == M[I].to_qsym().omega_involution()
....:      for I in Permutations(4) )
True
```

```
>>> from sage.all import *
>>> all( M[I].omega_involution().to_qsym()
...      == M[I].to_qsym().omega_involution()
...      for I in Permutations(Integer(4)) )
True
```

Testing the $\omega \circ \iota = \iota \circ \psi$ relation:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: all( S[I].psi_involution().to_fqsym() == S[I].to_fqsym().omega_
↪involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> all( S[I].psi_involution().to_fqsym() == S[I].to_fqsym().omega_
↪involution()
...      for I in Compositions(Integer(4)) )
True
```

Todo

Check further commutative squares.

psi_involution()

Return the image of the element `self` of $FQSym$ under the psi involution.

The ψ involution is defined as the linear map $FQSym \to FQSym$ that sends each basis element F_u of the F-basis of $FQSym$ to the basis element $F_{w_0 \circ u}$, where w_0 is the longest word (i.e., $w_0(i) = n+1-i$) in the symmetric group S_n that contains u. The ψ involution is a graded coalgebra automorphism and an algebra anti-automorphism of $FQSym$. Every permutation $u \in S_n$ satisfies

$$\psi(F_u) = F_{w_0 \circ u}, \qquad \psi(G_u) = G_{u \circ w_0},$$

where standard notations for classical bases of $FQSym$ are being used (that is, F for the F-basis, and G for the G-basis). In other words, writing permutations in one-line notation, we have

$$\psi(F_{(u_1,u_2,\ldots,u_n)}) = F_{(n+1-u_1,n+1-u_2,\ldots,n+1-u_n)}, \qquad \psi(G_{(u_1,u_2,\ldots,u_n)}) = G_{(u_n,u_{n-1},\ldots,u_1)}.$$

If we also consider the ψ involution (`psi_involution()`) of the quasisymmetric functions (by slight abuse of notation), and if we let π be the canonical projection $FQSym \to QSym$, then $\pi \circ \psi = \psi \circ \pi$.

Additionally, consider the ω involution (`omega_involution()`) of the noncommutative symmetric functions, and if we let ι be the canonical inclusion $NSym \to FQSym$, then $\psi \circ \iota = \iota \circ \omega$.

Todo

Duality?

See also

omega_involution(), *star_involution()*

EXAMPLES:

```
sage: FQSym = algebras.FQSym(ZZ)
sage: F = FQSym.F()
sage: F[[2,3,1]].psi_involution()
F[2, 1, 3]
sage: (3*F[[1]] - 4*F[[]] + 5*F[[1,2]]).psi_involution()
-4*F[] + 3*F[1] + 5*F[2, 1]
sage: G = FQSym.G()
sage: G[[2,3,1]].psi_involution()
G[1, 3, 2]
sage: M = FQSym.M()
sage: M[[2,3,1]].psi_involution()
-M[1, 2, 3] - M[1, 3, 2] - M[2, 3, 1]
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(ZZ)
>>> F = FQSym.F()
>>> F[[Integer(2),Integer(3),Integer(1)]].psi_involution()
F[2, 1, 3]
>>> (Integer(3)*F[[Integer(1)]] - Integer(4)*F[[]] +␣
↪Integer(5)*F[[Integer(1),Integer(2)]]).psi_involution()
-4*F[] + 3*F[1] + 5*F[2, 1]
>>> G = FQSym.G()
>>> G[[Integer(2),Integer(3),Integer(1)]].psi_involution()
G[1, 3, 2]
>>> M = FQSym.M()
>>> M[[Integer(2),Integer(3),Integer(1)]].psi_involution()
-M[1, 2, 3] - M[1, 3, 2] - M[2, 3, 1]
```

The ψ involution intertwines the antipode and the inverse of the antipode:

```
sage: all( F(I).antipode().psi_involution().antipode()
....:      == F(I).psi_involution()
....:      for I in Permutations(4) )
True
```

```
>>> from sage.all import *
>>> all( F(I).antipode().psi_involution().antipode()
...      == F(I).psi_involution()
...      for I in Permutations(Integer(4)) )
True
```

Testing the $\pi \circ \psi = \psi \circ \pi$ relation above:

```
sage: all( M[I].psi_involution().to_qsym()
....:      == M[I].to_qsym().psi_involution()
....:      for I in Permutations(4) )
True
```

```
>>> from sage.all import *
>>> all( M[I].psi_involution().to_qsym()
...      == M[I].to_qsym().psi_involution()
...      for I in Permutations(Integer(4)) )
True
```

Testing the $\psi \circ \iota = \iota \circ \omega$ relation:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: all( S[I].omega_involution().to_fqsym() == S[I].to_fqsym().psi_
↪involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> all( S[I].omega_involution().to_fqsym() == S[I].to_fqsym().psi_
↪involution()
...      for I in Compositions(Integer(4)) )
True
```

Todo

Check further commutative squares.

star_involution()

Return the image of the element `self` of $FQSym$ under the star involution.

The star involution is defined as the linear map $FQSym \to FQSym$ that sends each basis element F_u of the F-basis of $FQSym$ to the basis element $F_{w_0 \circ u \circ w_0}$, where w_0 is the longest word (i.e., $w_0(i) = n + 1 - i$) in the symmetric group S_n that contains u. The star involution is a graded Hopf algebra anti-automorphism of $FQSym$. It is denoted by $f \mapsto f^*$. Every permutation $u \in S_n$ satisfies

$$(F_u)^* = F_{w_0 \circ u \circ w_0}, \qquad (G_u)^* = G_{w_0 \circ u \circ w_0}, \qquad (\mathcal{M}_u)^* = \mathcal{M}_{w_0 \circ u \circ w_0},$$

where standard notations for classical bases of $FQSym$ are being used (that is, F for the F-basis, G for the G-basis, and $\mathcal{M}$ for the Monomial basis). In other words, writing permutations in one-line notation, we have

$$(F_{(u_1,u_2,\ldots,u_n)})^* = F_{(n+1-u_n,n+1-u_{n-1},\ldots,n+1-u_1)}, \qquad (G_{(u_1,u_2,\ldots,u_n)})^* = G_{(n+1-u_n,n+1-u_{n-1},\ldots,n+1-u_1)},$$

and

$$(\mathcal{M}_{(u_1,u_2,\ldots,u_n)})^* = \mathcal{M}_{(n+1-u_n,n+1-u_{n-1},\ldots,n+1-u_1)}.$$

Let us denote the star involution by $(*)$ as well.

If we also denote by $(*)$ the star involution of of the quasisymmetric functions (*star_involution()*) and if we let $\pi : FQSym \to QSym$ be the canonical projection then $\pi \circ (*) = (*) \circ \pi$. Similar for the noncommutative symmetric functions (*star_involution()*) with $\pi : NSym \to FQSym$ being the canonical inclusion and the word quasisymmetric functions (`star_involution()`) with $\pi : FQSym \to WQSym$ the canonical inclusion.

Todo

Duality?

See also

omega_involution(), *psi_involution()*

EXAMPLES:

```
sage: FQSym = algebras.FQSym(ZZ)
sage: F = FQSym.F()
sage: F[[2,3,1]].star_involution()
F[3, 1, 2]
sage: (3*F[[1]] - 4*F[[]] + 5*F[[1,2]]).star_involution()
-4*F[] + 3*F[1] + 5*F[1, 2]
sage: G = FQSym.G()
sage: G[[2,3,1]].star_involution()
G[3, 1, 2]
sage: M = FQSym.M()
sage: M[[2,3,1]].star_involution()
M[3, 1, 2]
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(ZZ)
>>> F = FQSym.F()
>>> F[[Integer(2),Integer(3),Integer(1)]].star_involution()
F[3, 1, 2]
>>> (Integer(3)*F[[Integer(1)]] - Integer(4)*F[[]] +␣
↪Integer(5)*F[[Integer(1),Integer(2)]]).star_involution()
-4*F[] + 3*F[1] + 5*F[1, 2]
>>> G = FQSym.G()
>>> G[[Integer(2),Integer(3),Integer(1)]].star_involution()
G[3, 1, 2]
>>> M = FQSym.M()
>>> M[[Integer(2),Integer(3),Integer(1)]].star_involution()
M[3, 1, 2]
```

The star involution commutes with the antipode:

```
sage: all( F(I).antipode().star_involution()
....:      == F(I).star_involution().antipode()
....:      for I in Permutations(4) )
True
```

```
>>> from sage.all import *
>>> all( F(I).antipode().star_involution()
...      == F(I).star_involution().antipode()
...      for I in Permutations(Integer(4)) )
True
```

Testing the $\pi \circ (*) = (*) \circ \pi$ relation:

```
sage: all( M[I].star_involution().to_qsym()
....:      == M[I].to_qsym().star_involution()
....:      for I in Permutations(4) )
True
```

```
>>> from sage.all import *
>>> all( M[I].star_involution().to_qsym()
...      == M[I].to_qsym().star_involution()
...      for I in Permutations(Integer(4)) )
True
```

Similar for $NSym$:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: all( S[I].star_involution().to_fqsym() == S[I].to_fqsym().star_
↪involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> all( S[I].star_involution().to_fqsym() == S[I].to_fqsym().star_
↪involution()
...      for I in Compositions(Integer(4)) )
True
```

Similar for $WQSym$:

```
sage: WQSym = algebras.WQSym(ZZ)
sage: all( F(I).to_wqsym().star_involution()
....:      == F(I).star_involution().to_wqsym()
....:      for I in Permutations(4) )
True
```

```
>>> from sage.all import *
>>> WQSym = algebras.WQSym(ZZ)
>>> all( F(I).to_wqsym().star_involution()
...      == F(I).star_involution().to_wqsym()
...      for I in Permutations(Integer(4)) )
True
```

Todo

Check further commutative squares.

to_qsym()

Return the image of `self` under the canonical projection $FQSym \to QSym$.

The canonical projection $FQSym \to QSym$ is a surjective homomorphism of Hopf algebras. It sends a basis element F_w of $FQSym$ to the basis element $F_{\operatorname{Comp} w}$ of the fundamental basis of $QSym$, where Comp w stands for the descent composition (`sage.combinat.permutation.Permutation.descents_composition()`) of the permutation w.

See also

`QuasiSymmetricFunctions` for a definition of $QSym$.

EXAMPLES:

```
sage: G = algebras.FQSym(QQ).G()
sage: x = G[1, 3, 2]
sage: x.to_qsym()
F[2, 1]
sage: G[2, 3, 1].to_qsym()
F[1, 2]
sage: F = algebras.FQSym(QQ).F()
sage: F[2, 3, 1].to_qsym()
F[2, 1]
sage: (F[2, 3, 1] + F[1, 3, 2] + F[1, 2, 3]).to_qsym()
2*F[2, 1] + F[3]
sage: F2 = algebras.FQSym(GF(2)).F()
sage: F2[2, 3, 1].to_qsym()
F[2, 1]
sage: (F2[2, 3, 1] + F2[1, 3, 2] + F2[1, 2, 3]).to_qsym()
F[3]
```

```
>>> from sage.all import *
>>> G = algebras.FQSym(QQ).G()
>>> x = G[Integer(1), Integer(3), Integer(2)]
>>> x.to_qsym()
F[2, 1]
>>> G[Integer(2), Integer(3), Integer(1)].to_qsym()
F[1, 2]
>>> F = algebras.FQSym(QQ).F()
>>> F[Integer(2), Integer(3), Integer(1)].to_qsym()
F[2, 1]
>>> (F[Integer(2), Integer(3), Integer(1)] + F[Integer(1), Integer(3),␣
↪Integer(2)] + F[Integer(1), Integer(2), Integer(3)]).to_qsym()
2*F[2, 1] + F[3]
>>> F2 = algebras.FQSym(GF(Integer(2))).F()
>>> F2[Integer(2), Integer(3), Integer(1)].to_qsym()
F[2, 1]
>>> (F2[Integer(2), Integer(3), Integer(1)] + F2[Integer(1), Integer(3),␣
↪Integer(2)] + F2[Integer(1), Integer(2), Integer(3)]).to_qsym()
F[3]
```

to_symmetric_group_algebra(*n=None*)

Return the element of a symmetric group algebra corresponding to the element `self` of $FQSym$.

INPUT:

- `n` – integer (default: the maximal degree of `self`); the rank of the target symmetric group algebra

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).G()
sage: x = A([1,3,2,4]) + 5/2 * A([2,3,4,1])
sage: x.to_symmetric_group_algebra()
[1, 3, 2, 4] + 5/2*[4, 1, 2, 3]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).G()
>>> x = A([Integer(1),Integer(3),Integer(2),Integer(4)]) + Integer(5)/
↪Integer(2) * A([Integer(2),Integer(3),Integer(4),Integer(1)])
>>> x.to_symmetric_group_algebra()
[1, 3, 2, 4] + 5/2*[4, 1, 2, 3]
```

to_wqsym()

Return the image of `self` under the canonical inclusion map $FQSym \to WQSym$.

The canonical inclusion map $FQSym \to WQSym$ is an injective homomorphism of Hopf algebras. It sends a basis element G_w of $FQSym$ to the sum of basis elements $\mathbf{M}_u$ of $WQSym$, where u ranges over all packed words whose standardization is w.

See also

WordQuasiSymmetricFunctions for a definition of $WQSym$.

EXAMPLES:

```
sage: G = algebras.FQSym(QQ).G()
sage: x = G[1, 3, 2]
sage: x.to_wqsym()
M[{1}, {3}, {2}] + M[{1, 3}, {2}]
sage: G[1, 2].to_wqsym()
M[{1}, {2}] + M[{1, 2}]
sage: F = algebras.FQSym(QQ).F()
sage: F[3, 1, 2].to_wqsym()
M[{3}, {1}, {2}] + M[{3}, {1, 2}]
sage: G[2, 3, 1].to_wqsym()
M[{3}, {1}, {2}] + M[{3}, {1, 2}]
```

```
>>> from sage.all import *
>>> G = algebras.FQSym(QQ).G()
>>> x = G[Integer(1), Integer(3), Integer(2)]
>>> x.to_wqsym()
M[{1}, {3}, {2}] + M[{1, 3}, {2}]
>>> G[Integer(1), Integer(2)].to_wqsym()
M[{1}, {2}] + M[{1, 2}]
>>> F = algebras.FQSym(QQ).F()
>>> F[Integer(3), Integer(1), Integer(2)].to_wqsym()
M[{3}, {1}, {2}] + M[{3}, {1, 2}]
>>> G[Integer(2), Integer(3), Integer(1)].to_wqsym()
M[{3}, {1}, {2}] + M[{3}, {1, 2}]
```

class ParentMethods

Bases: `object`

basis(*degree=None*)

The basis elements (optionally: of the specified degree).

OUTPUT: Family

EXAMPLES:

```
sage: FQSym = algebras.FQSym(QQ)
sage: G = FQSym.G()
sage: G.basis()
Lazy family (Term map from Standard permutations to Free Quasi-symmetric␣
↪functions over Rational Field in the G basis(i))_{i in Standard␣
↪permutations}
sage: G.basis().keys()
Standard permutations
sage: G.basis(degree=3).keys()
Standard permutations of 3
sage: G.basis(degree=3).list()
[G[1, 2, 3], G[1, 3, 2], G[2, 1, 3], G[2, 3, 1], G[3, 1, 2], G[3, 2, 1]]
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(QQ)
>>> G = FQSym.G()
>>> G.basis()
Lazy family (Term map from Standard permutations to Free Quasi-symmetric␣
↪functions over Rational Field in the G basis(i))_{i in Standard␣
↪permutations}
>>> G.basis().keys()
Standard permutations
>>> G.basis(degree=Integer(3)).keys()
Standard permutations of 3
>>> G.basis(degree=Integer(3)).list()
[G[1, 2, 3], G[1, 3, 2], G[2, 1, 3], G[2, 3, 1], G[3, 1, 2], G[3, 2, 1]]
```

from_symmetric_group_algebra(*x*)

Return the element of $FQSym$ corresponding to the element x of a symmetric group algebra.

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: SGA4 = SymmetricGroupAlgebra(QQ, 4)
sage: x = SGA4([1,3,2,4]) + 5/2 * SGA4([1,2,4,3])
sage: A.from_symmetric_group_algebra(x)
5/2*F[1, 2, 4, 3] + F[1, 3, 2, 4]
sage: A.from_symmetric_group_algebra(SGA4.zero())
0
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> SGA4 = SymmetricGroupAlgebra(QQ, Integer(4))
>>> x = SGA4([Integer(1),Integer(3),Integer(2),Integer(4)]) + Integer(5)/
↪Integer(2) * SGA4([Integer(1),Integer(2),Integer(4),Integer(3)])
>>> A.from_symmetric_group_algebra(x)
5/2*F[1, 2, 4, 3] + F[1, 3, 2, 4]
```

(continues on next page)

(continued from previous page)

```
>>> A.from_symmetric_group_algebra(SGA4.zero())
0
```

is_commutative()

Return whether this $FQSym$ is commutative.

EXAMPLES:

```
sage: F = algebras.FQSym(ZZ).F()
sage: F.is_commutative()
False
```

```
>>> from sage.all import *
>>> F = algebras.FQSym(ZZ).F()
>>> F.is_commutative()
False
```

is_field(*proof=True*)

Return whether this $FQSym$ is a field.

EXAMPLES:

```
sage: F = algebras.FQSym(QQ).F()
sage: F.is_field()
False
```

```
>>> from sage.all import *
>>> F = algebras.FQSym(QQ).F()
>>> F.is_field()
False
```

one_basis()

Return the index of the unit.

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: A.one_basis()
[]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> A.one_basis()
[]
```

prec()

Return the $\prec$ product.

On the F-basis of `FQSym`, this product is determined by $F_x \prec F_y = \sum F_z$, where the sum ranges over all z in the shifted shuffle of x and y with the additional condition that the first letter of the result comes from x.

The usual symbol for this operation is $\prec$.

See also

product(), succ()

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: x = A([2,1])
sage: A.prec(x, x)
F[2, 1, 4, 3] + F[2, 4, 1, 3] + F[2, 4, 3, 1]
sage: y = A([2,1,3])
sage: A.prec(x, y)
F[2, 1, 4, 3, 5] + F[2, 4, 1, 3, 5] + F[2, 4, 3, 1, 5]
 + F[2, 4, 3, 5, 1]
sage: A.prec(y, x)
F[2, 1, 3, 5, 4] + F[2, 1, 5, 3, 4] + F[2, 1, 5, 4, 3]
 + F[2, 5, 1, 3, 4] + F[2, 5, 1, 4, 3] + F[2, 5, 4, 1, 3]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> x = A([Integer(2),Integer(1)])
>>> A.prec(x, x)
F[2, 1, 4, 3] + F[2, 4, 1, 3] + F[2, 4, 3, 1]
>>> y = A([Integer(2),Integer(1),Integer(3)])
>>> A.prec(x, y)
F[2, 1, 4, 3, 5] + F[2, 4, 1, 3, 5] + F[2, 4, 3, 1, 5]
 + F[2, 4, 3, 5, 1]
>>> A.prec(y, x)
F[2, 1, 3, 5, 4] + F[2, 1, 5, 3, 4] + F[2, 1, 5, 4, 3]
 + F[2, 5, 1, 3, 4] + F[2, 5, 1, 4, 3] + F[2, 5, 4, 1, 3]
```

prec_by_coercion(x, y)

Return $x \prec y$, computed using coercion to the F-basis.

See *prec()* for the definition of the objects involved.

EXAMPLES:

```
sage: G = algebras.FQSym(ZZ).G()
sage: a = G([1])
sage: b = G([2, 3, 1])
sage: G.prec(a, b) + G.succ(a, b) == a * b # indirect doctest
True
```

```
>>> from sage.all import *
>>> G = algebras.FQSym(ZZ).G()
>>> a = G([Integer(1)])
>>> b = G([Integer(2), Integer(3), Integer(1)])
>>> G.prec(a, b) + G.succ(a, b) == a * b # indirect doctest
True
```

some_elements()

Return some elements of the free quasi-symmetric functions.

EXAMPLES:

```
sage: A = algebras.FQSym(QQ)
sage: F = A.F()
sage: F.some_elements()
[F[], F[1], F[1, 2] + F[2, 1], F[] + F[1, 2] + F[2, 1]]
sage: G = A.G()
sage: G.some_elements()
[G[], G[1], G[1, 2] + G[2, 1], G[] + G[1, 2] + G[2, 1]]
sage: M = A.M()
sage: M.some_elements()
[M[], M[1], M[1, 2] + 2*M[2, 1], M[] + M[1, 2] + 2*M[2, 1]]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ)
>>> F = A.F()
>>> F.some_elements()
[F[], F[1], F[1, 2] + F[2, 1], F[] + F[1, 2] + F[2, 1]]
>>> G = A.G()
>>> G.some_elements()
[G[], G[1], G[1, 2] + G[2, 1], G[] + G[1, 2] + G[2, 1]]
>>> M = A.M()
>>> M.some_elements()
[M[], M[1], M[1, 2] + 2*M[2, 1], M[] + M[1, 2] + 2*M[2, 1]]
```

succ()

Return the $\succ$ product.

On the F-basis of `FQSym`, this product is determined by $F_x \succ F_y = \sum F_z$, where the sum ranges over all z in the shifted shuffle of x and y with the additional condition that the first letter of the result comes from y.

The usual symbol for this operation is $\succ$.

See also

`product()`, *prec()*

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: x = A([1])
sage: A.succ(x, x)
F[2, 1]
sage: y = A([3,1,2])
sage: A.succ(x, y)
F[4, 1, 2, 3] + F[4, 2, 1, 3] + F[4, 2, 3, 1]
sage: A.succ(y, x)
F[4, 3, 1, 2]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> x = A([Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> A.succ(x, x)
F[2, 1]
>>> y = A([Integer(3),Integer(1),Integer(2)])
>>> A.succ(x, y)
F[4, 1, 2, 3] + F[4, 2, 1, 3] + F[4, 2, 3, 1]
>>> A.succ(y, x)
F[4, 3, 1, 2]
```

succ_by_coercion(*x*, *y*)

Return $x \succ y$, computed using coercion to the F-basis.

See *succ()* for the definition of the objects involved.

EXAMPLES:

```
sage: G = algebras.FQSym(ZZ).G()
sage: G.succ(G([1]), G([2, 3, 1])) # indirect doctest
G[2, 3, 4, 1] + G[3, 2, 4, 1] + G[4, 2, 3, 1]
```

```
>>> from sage.all import *
>>> G = algebras.FQSym(ZZ).G()
>>> G.succ(G([Integer(1)]), G([Integer(2), Integer(3), Integer(1)])) #␣
↪indirect doctest
G[2, 3, 4, 1] + G[3, 2, 4, 1] + G[4, 2, 3, 1]
```

super_categories()

The super categories of `self`.

EXAMPLES:

```
sage: from sage.combinat.fqsym import FQSymBases
sage: FQSym = algebras.FQSym(ZZ)
sage: bases = FQSymBases(FQSym)
sage: bases.super_categories()
[Category of realizations of Free Quasi-symmetric functions over Integer Ring,
 Join of Category of realizations of Hopf algebras over Integer Ring
   and Category of graded algebras over Integer Ring
   and Category of graded coalgebras over Integer Ring,
 Category of graded connected Hopf algebras with basis over Integer Ring]
```

```
>>> from sage.all import *
>>> from sage.combinat.fqsym import FQSymBases
>>> FQSym = algebras.FQSym(ZZ)
>>> bases = FQSymBases(FQSym)
>>> bases.super_categories()
[Category of realizations of Free Quasi-symmetric functions over Integer Ring,
 Join of Category of realizations of Hopf algebras over Integer Ring
   and Category of graded algebras over Integer Ring
   and Category of graded coalgebras over Integer Ring,
 Category of graded connected Hopf algebras with basis over Integer Ring]
```

class sage.combinat.fqsym.**FQSymBasis_abstract**(*alg*)

Bases: *CombinatorialFreeModule*, `BindableClass`

Abstract base class for bases of FQSym.

This must define two attributes:

- `_prefix` – the basis prefix
- `_basis_name` – the name of the basis and must match one of the names that the basis can be constructed from FQSym

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: A = algebras.FQSym(QQ)
sage: F = A.F()
sage: F.an_element()
F[1] + 2*F[1, 2] + 2*F[2, 1]
sage: G = A.G()
sage: G.an_element()
G[1] + 2*G[1, 2] + 2*G[2, 1]
sage: M = A.M()
sage: M.an_element()
M[1] + 2*M[1, 2] + 4*M[2, 1]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ)
>>> F = A.F()
>>> F.an_element()
F[1] + 2*F[1, 2] + 2*F[2, 1]
>>> G = A.G()
>>> G.an_element()
G[1] + 2*G[1, 2] + 2*G[2, 1]
>>> M = A.M()
>>> M.an_element()
M[1] + 2*M[1, 2] + 4*M[2, 1]
```

class sage.combinat.fqsym.**FreeQuasisymmetricFunctions**(*R*)

Bases: `UniqueRepresentation`, `Parent`

The free quasi-symmetric functions.

The Hopf algebra $FQSym$ of free quasi-symmetric functions over a commutative ring R is the free R-module with basis indexed by all permutations (i.e., the indexing set is the disjoint union of all symmetric groups). Its product is determined by the shifted shuffles of two permutations, whereas its coproduct is given by splitting a permutation (regarded as a word) into two (at every possible point) and standardizing the two pieces. This Hopf algebra was introduced in [MR]. See [GriRei18] (Chapter 8) for a treatment using modern notations.

In more detail: For each $n \geq 0$, consider the symmetric group S_n. Let S be the disjoint union of the S_n over all $n \geq 0$. Then, $FQSym$ is the free R-module with basis $(F_w)_{w \in S}$. This R-module is graded, with the n-th graded component being spanned by all F_w for $w \in S_n$. A multiplication is defined on $FQSym$ as follows: For any two permutations $u \in S_k$ and $v \in S_l$, we set

$$F_u F_v = \sum F_w,$$

where the sum is over all shuffles of u with $v[k]$. Here, the permutations u and v are regarded as words (by writing them in one-line notation), and $v[k]$ means the word obtained from v by increasing each letter by k (for example,

$(1, 4, 2, 3)[5] = (6, 9, 7, 8)$); and the shuffles w are translated back into permutations. This defines an associative multiplication on $FQSym$; its unity is F_e, where e is the identity permutation in S_0.

In Section 1.3 of [AguSot05], Aguiar and Sottile construct a different basis of $FQSym$. Their basis, called the *monomial basis* and denoted by $(\mathcal{M}_u)$, is also indexed by permutations. It is connected to the above F-basis by the relation

$$F_u = \sum_v \mathcal{M}_v,$$

where the sum ranges over all permutations v such that each inversion of u is an inversion of v. (An *inversion* of a permutation w means a pair (i, j) of positions satisfying $i < j$ and $w(i) > w(j)$.) The above relation yields a unitriangular change-of-basis matrix, and thus can be used to compute the $\mathcal{M}_u$ by Mobius inversion.

Another classical basis of $FQSym$ is $(G_w)_{w\in S}$, where $G_w = F_{w^{-1}}$. This is just a relabeling of the basis $(F_w)_{w\in S}$, but is a more natural choice from some viewpoints.

The algebra $FQSym$ is often identified with ("realized as") a subring of the ring of all bounded-degree noncommutative power series in countably many indeterminates (i.e., elements in $R\langle\langle x_1, x_2, x_3, \ldots\rangle\rangle$ of bounded degree). Namely, consider words over the alphabet $\{1, 2, 3, \ldots\}$; every noncommutative power series is an infinite R-linear combination of these words. Consider the R-linear map that sends each G_u to the sum of all words whose standardization (also known as "standard permutation"; see `standard_permutation()`) is u. This map is an injective R-algebra homomorphism, and thus embeds $FQSym$ into the latter ring.

As an associative algebra, $FQSym$ has the richer structure of a dendriform algebra. This means that the associative product $*$ is decomposed as a sum of two binary operations

$$xy = x \succ y + x \prec y$$

that satisfy the axioms:

$$(x \succ y) \prec z = x \succ (y \prec z),$$

$$(x \prec y) \prec z = x \prec (yz),$$

$$(xy) \succ z = x \succ (y \succ z).$$

These two binary operations are defined similarly to the (associative) product above: We set

$$F_u \prec F_v = \sum F_w,$$

where the sum is now over all shuffles of u with $v[k]$ whose first letter is taken from u (rather than from $v[k]$). Similarly,

$$F_u \succ F_v = \sum F_w,$$

where the sum is over all remaining shuffles of u with $v[k]$.

Todo

Decide what $1 \prec 1$ and $1 \succ 1$ are.

Note

The usual binary operator `*` is used for the associative product.

EXAMPLES:

```
sage: F = algebras.FQSym(ZZ).F()
sage: x,y,z = F([1]), F([1,2]), F([1,3,2])
sage: (x * y) * z
F[1, 2, 3, 4, 6, 5] + ...
```

```
>>> from sage.all import *
>>> F = algebras.FQSym(ZZ).F()
>>> x,y,z = F([Integer(1)]), F([Integer(1),Integer(2)]), F([Integer(1),Integer(3),
↪Integer(2)])
>>> (x * y) * z
F[1, 2, 3, 4, 6, 5] + ...
```

The product of $FQSym$ is associative:

```
sage: x * (y * z) == (x * y) * z
True
```

```
>>> from sage.all import *
>>> x * (y * z) == (x * y) * z
True
```

The associative product decomposes into two parts:

```
sage: x * y == F.prec(x, y) + F.succ(x, y)
True
```

```
>>> from sage.all import *
>>> x * y == F.prec(x, y) + F.succ(x, y)
True
```

The axioms of a dendriform algebra hold:

```
sage: F.prec(F.succ(x, y), z) == F.succ(x, F.prec(y, z))
True
sage: F.prec(F.prec(x, y), z) == F.prec(x, y * z)
True
sage: F.succ(x * y, z) == F.succ(x, F.succ(y, z))
True
```

```
>>> from sage.all import *
>>> F.prec(F.succ(x, y), z) == F.succ(x, F.prec(y, z))
True
>>> F.prec(F.prec(x, y), z) == F.prec(x, y * z)
True
>>> F.succ(x * y, z) == F.succ(x, F.succ(y, z))
True
```

$FQSym$ is also known as the Malvenuto-Reutenauer algebra:

```
sage: algebras.MalvenutoReutenauer(ZZ)
Free Quasi-symmetric functions over Integer Ring
```

```
>>> from sage.all import *
>>> algebras.MalvenutoReutenauer(ZZ)
Free Quasi-symmetric functions over Integer Ring
```

REFERENCES:

- [MR]
- [LR1998]
- [GriRei18]

class F(*alg*)

Bases: `FQSymBasis_abstract`

The F-basis of $FQSym$.

This is the basis (F_w), with w ranging over all permutations. See the documentation of `FreeQuasisymmetricFunctions` for details.

EXAMPLES:

```
sage: FQSym = algebras.FQSym(QQ)
sage: FQSym.F()
Free Quasi-symmetric functions over Rational Field in the F basis
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(QQ)
>>> FQSym.F()
Free Quasi-symmetric functions over Rational Field in the F basis
```

class Element

Bases: `IndexedFreeModuleElement`

to_symmetric_group_algebra(*n=None*)

Return the element of a symmetric group algebra corresponding to the element `self` of $FQSym$.

INPUT:

- `n` – integer (default: the maximal degree of `self`); the rank of the target symmetric group algebra

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: x = A([1,3,2,4]) + 5/2 * A([1,2,4,3])
sage: x.to_symmetric_group_algebra()
5/2*[1, 2, 4, 3] + [1, 3, 2, 4]
sage: x.to_symmetric_group_algebra(n=7)
5/2*[1, 2, 4, 3, 5, 6, 7] + [1, 3, 2, 4, 5, 6, 7]
sage: a = A.zero().to_symmetric_group_algebra(); a
0
sage: parent(a)
Symmetric group algebra of order 0 over Rational Field

sage: y = A([1,3,2,4]) + 5/2 * A([2,1])
sage: y.to_symmetric_group_algebra()
[1, 3, 2, 4] + 5/2*[2, 1, 3, 4]
sage: y.to_symmetric_group_algebra(6)
[1, 3, 2, 4, 5, 6] + 5/2*[2, 1, 3, 4, 5, 6]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> x = A([Integer(1),Integer(3),Integer(2),Integer(4)]) + Integer(5)/
↪Integer(2) * A([Integer(1),Integer(2),Integer(4),Integer(3)])
>>> x.to_symmetric_group_algebra()
5/2*[1, 2, 4, 3] + [1, 3, 2, 4]
>>> x.to_symmetric_group_algebra(n=Integer(7))
5/2*[1, 2, 4, 3, 5, 6, 7] + [1, 3, 2, 4, 5, 6, 7]
>>> a = A.zero().to_symmetric_group_algebra(); a
0
>>> parent(a)
Symmetric group algebra of order 0 over Rational Field

>>> y = A([Integer(1),Integer(3),Integer(2),Integer(4)]) + Integer(5)/
↪Integer(2) * A([Integer(2),Integer(1)])
>>> y.to_symmetric_group_algebra()
[1, 3, 2, 4] + 5/2*[2, 1, 3, 4]
>>> y.to_symmetric_group_algebra(Integer(6))
[1, 3, 2, 4, 5, 6] + 5/2*[2, 1, 3, 4, 5, 6]
```

coproduct_on_basis(*x*)

Return the coproduct of F_σ for σ a permutation (here, σ is x).

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: x = A([1])
sage: ascii_art(A.coproduct(A.one()))  # indirect doctest
1 # 1

sage: ascii_art(A.coproduct(x))  # indirect doctest
1 # F    + F    # 1
     [1]    [1]

sage: A = algebras.FQSym(QQ).F()
sage: x, y, z = A([1]), A([2,1]), A([3,2,1])
sage: A.coproduct(z)
F[] # F[3, 2, 1] + F[1] # F[2, 1] + F[2, 1] # F[1]
+ F[3, 2, 1] # F[]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> x = A([Integer(1)])
>>> ascii_art(A.coproduct(A.one()))  # indirect doctest
1 # 1

>>> ascii_art(A.coproduct(x))  # indirect doctest
1 # F    + F    # 1
     [1]    [1]

>>> A = algebras.FQSym(QQ).F()
>>> x, y, z = A([Integer(1)]), A([Integer(2),Integer(1)]), A([Integer(3),
↪Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> A.coproduct(z)
F[] # F[3, 2, 1] + F[1] # F[2, 1] + F[2, 1] # F[1]
+ F[3, 2, 1] # F[]
```

degree_on_basis(*t*)

Return the degree of a permutation in the algebra of free quasi-symmetric functions.

This is the size of the permutation (i.e., the n for which the permutation belongs to S_n).

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: u = Permutation([2,1])
sage: A.degree_on_basis(u)
2
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> u = Permutation([Integer(2),Integer(1)])
>>> A.degree_on_basis(u)
2
```

prec_product_on_basis(*x*, *y*)

Return the $\prec$ product of two permutations.

This is the shifted shuffle of x and y with the additional condition that the first letter of the result comes from x.

The usual symbol for this operation is $\prec$.

See also

product_on_basis(), *succ_product_on_basis()*

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: x = Permutation([1,2])
sage: A.prec_product_on_basis(x, x)
F[1, 2, 3, 4] + F[1, 3, 2, 4] + F[1, 3, 4, 2]
sage: y = Permutation([])
sage: A.prec_product_on_basis(x, y) == A(x)
True
sage: A.prec_product_on_basis(y, x) == 0
True
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> x = Permutation([Integer(1),Integer(2)])
>>> A.prec_product_on_basis(x, x)
F[1, 2, 3, 4] + F[1, 3, 2, 4] + F[1, 3, 4, 2]
>>> y = Permutation([])
```

(continues on next page)

(continued from previous page)

```
>>> A.prec_product_on_basis(x, y) == A(x)
True
>>> A.prec_product_on_basis(y, x) == Integer(0)
True
```

product_on_basis(*x*, *y*)

Return the $*$ associative product of two permutations.

This is the shifted shuffle of x and y.

See also

succ_product_on_basis(), *prec_product_on_basis()*

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: x = Permutation([1])
sage: A.product_on_basis(x, x)
F[1, 2] + F[2, 1]
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> x = Permutation([Integer(1)])
>>> A.product_on_basis(x, x)
F[1, 2] + F[2, 1]
```

succ_product_on_basis(*x*, *y*)

Return the $\succ$ product of two permutations.

This is the shifted shuffle of x and y with the additional condition that the first letter of the result comes from y.

The usual symbol for this operation is $\succ$.

See also

- *product_on_basis()*, *prec_product_on_basis()*

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).F()
sage: x = Permutation([1,2])
sage: A.succ_product_on_basis(x, x)
F[3, 1, 2, 4] + F[3, 1, 4, 2] + F[3, 4, 1, 2]
sage: y = Permutation([])
sage: A.succ_product_on_basis(x, y) == 0
True
sage: A.succ_product_on_basis(y, x) == A(x)
True
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).F()
>>> x = Permutation([Integer(1),Integer(2)])
>>> A.succ_product_on_basis(x, x)
F[3, 1, 2, 4] + F[3, 1, 4, 2] + F[3, 4, 1, 2]
>>> y = Permutation([])
>>> A.succ_product_on_basis(x, y) == Integer(0)
True
>>> A.succ_product_on_basis(y, x) == A(x)
True
```

class G(*alg*)

Bases: *FQSymBasis_abstract*

The G-basis of $FQSym$.

This is the basis (G_w), with w ranging over all permutations. See the documentation of *FreeQuasisymmetricFunctions* for details.

EXAMPLES:

```
sage: FQSym = algebras.FQSym(QQ)
sage: G = FQSym.G(); G
Free Quasi-symmetric functions over Rational Field in the G basis

sage: G([3, 1, 2]).coproduct()
G[] # G[3, 1, 2] + G[1] # G[2, 1] + G[1, 2] # G[1]
 + G[3, 1, 2] # G[]

sage: G([3, 1, 2]) * G([2, 1])
G[3, 1, 2, 5, 4] + G[4, 1, 2, 5, 3] + G[4, 1, 3, 5, 2]
 + G[4, 2, 3, 5, 1] + G[5, 1, 2, 4, 3] + G[5, 1, 3, 4, 2]
 + G[5, 1, 4, 3, 2] + G[5, 2, 3, 4, 1] + G[5, 2, 4, 3, 1]
 + G[5, 3, 4, 2, 1]
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(QQ)
>>> G = FQSym.G(); G
Free Quasi-symmetric functions over Rational Field in the G basis

>>> G([Integer(3), Integer(1), Integer(2)]).coproduct()
G[] # G[3, 1, 2] + G[1] # G[2, 1] + G[1, 2] # G[1]
 + G[3, 1, 2] # G[]

>>> G([Integer(3), Integer(1), Integer(2)]) * G([Integer(2), Integer(1)])
G[3, 1, 2, 5, 4] + G[4, 1, 2, 5, 3] + G[4, 1, 3, 5, 2]
 + G[4, 2, 3, 5, 1] + G[5, 1, 2, 4, 3] + G[5, 1, 3, 4, 2]
 + G[5, 1, 4, 3, 2] + G[5, 2, 3, 4, 1] + G[5, 2, 4, 3, 1]
 + G[5, 3, 4, 2, 1]
```

degree_on_basis(*t*)

Return the degree of a permutation in the algebra of free quasi-symmetric functions.

This is the size of the permutation (i.e., the n for which the permutation belongs to S_n).

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).G()
sage: u = Permutation([2,1])
sage: A.degree_on_basis(u)
2
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).G()
>>> u = Permutation([Integer(2),Integer(1)])
>>> A.degree_on_basis(u)
2
```

class M(*alg*)

Bases: *FQSymBasis_abstract*

The M-basis of $FQSym$.

This is the Monomial basis $(\mathcal{M}_w)$, with w ranging over all permutations. See the documentation of `FQSym` for details.

EXAMPLES:

```
sage: FQSym = algebras.FQSym(QQ)
sage: M = FQSym.M(); M
Free Quasi-symmetric functions over Rational Field in the Monomial basis

sage: M([3, 1, 2]).coproduct()
M[] # M[3, 1, 2] + M[1] # M[1, 2] + M[3, 1, 2] # M[]
sage: M([3, 2, 1]).coproduct()
M[] # M[3, 2, 1] + M[1] # M[2, 1] + M[2, 1] # M[1]
 + M[3, 2, 1] # M[]

sage: M([1, 2]) * M([1])
M[1, 2, 3] + 2*M[1, 3, 2] + M[2, 3, 1] + M[3, 1, 2]
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(QQ)
>>> M = FQSym.M(); M
Free Quasi-symmetric functions over Rational Field in the Monomial basis

>>> M([Integer(3), Integer(1), Integer(2)]).coproduct()
M[] # M[3, 1, 2] + M[1] # M[1, 2] + M[3, 1, 2] # M[]
>>> M([Integer(3), Integer(2), Integer(1)]).coproduct()
M[] # M[3, 2, 1] + M[1] # M[2, 1] + M[2, 1] # M[1]
 + M[3, 2, 1] # M[]

>>> M([Integer(1), Integer(2)]) * M([Integer(1)])
M[1, 2, 3] + 2*M[1, 3, 2] + M[2, 3, 1] + M[3, 1, 2]
```

class Element

Bases: `IndexedFreeModuleElement`

star_involution()

Return the image of the element `self` of $FQSym$ under the star involution.

See `FQSymBases.ElementMethods.star_involution()` for a definition of the involution and for examples.

See also:

`omega_involution()`, `psi_involution()`

EXAMPLES:

```
sage: FQSym = algebras.FQSym(ZZ)
sage: M = FQSym.M()
sage: M[[2,3,1]].star_involution()
M[3, 1, 2]
sage: M[[]].star_involution()
M[]
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(ZZ)
>>> M = FQSym.M()
>>> M[[Integer(2),Integer(3),Integer(1)]].star_involution()
M[3, 1, 2]
>>> M[[]].star_involution()
M[]
```

coproduct_on_basis(*x*)

Return the coproduct of $\mathcal{M}_\sigma$ for σ a permutation (here, σ is `x`).

This uses Theorem 3.1 in [AguSot05].

EXAMPLES:

```
sage: M = algebras.FQSym(QQ).M()
sage: x = M([1])
sage: ascii_art(M.coproduct(M.one()))  # indirect doctest
1 # 1

sage: ascii_art(M.coproduct(x))  # indirect doctest
1 # M    + M    # 1
     [1]    [1]

sage: M.coproduct(M([2, 1, 3]))
M[] # M[2, 1, 3] + M[2, 1, 3] # M[]
sage: M.coproduct(M([2, 3, 1]))
M[] # M[2, 3, 1] + M[1, 2] # M[1] + M[2, 3, 1] # M[]
sage: M.coproduct(M([3, 2, 1]))
M[] # M[3, 2, 1] + M[1] # M[2, 1] + M[2, 1] # M[1]
+ M[3, 2, 1] # M[]
sage: M.coproduct(M([3, 4, 2, 1]))
M[] # M[3, 4, 2, 1] + M[1, 2] # M[2, 1] + M[2, 3, 1] # M[1]
 + M[3, 4, 2, 1] # M[]
sage: M.coproduct(M([3, 4, 1, 2]))
M[] # M[3, 4, 1, 2] + M[1, 2] # M[1, 2] + M[3, 4, 1, 2] # M[]
```

```
>>> from sage.all import *
>>> M = algebras.FQSym(QQ).M()
>>> x = M([Integer(1)])
>>> ascii_art(M.coproduct(M.one()))  # indirect doctest
1 # 1

>>> ascii_art(M.coproduct(x))  # indirect doctest
1 # M    + M    # 1
     [1]    [1]

>>> M.coproduct(M([Integer(2), Integer(1), Integer(3)]))
M[] # M[2, 1, 3] + M[2, 1, 3] # M[]
>>> M.coproduct(M([Integer(2), Integer(3), Integer(1)]))
M[] # M[2, 3, 1] + M[1, 2] # M[1] + M[2, 3, 1] # M[]
>>> M.coproduct(M([Integer(3), Integer(2), Integer(1)]))
M[] # M[3, 2, 1] + M[1] # M[2, 1] + M[2, 1] # M[1]
+ M[3, 2, 1] # M[]
>>> M.coproduct(M([Integer(3), Integer(4), Integer(2), Integer(1)]))
M[] # M[3, 4, 2, 1] + M[1, 2] # M[2, 1] + M[2, 3, 1] # M[1]
 + M[3, 4, 2, 1] # M[]
>>> M.coproduct(M([Integer(3), Integer(4), Integer(1), Integer(2)]))
M[] # M[3, 4, 1, 2] + M[1, 2] # M[1, 2] + M[3, 4, 1, 2] # M[]
```

degree_on_basis(*t*)

Return the degree of a permutation in the algebra of free quasi-symmetric functions.

This is the size of the permutation (i.e., the n for which the permutation belongs to S_n).

EXAMPLES:

```
sage: A = algebras.FQSym(QQ).M()
sage: u = Permutation([2,1])
sage: A.degree_on_basis(u)
2
```

```
>>> from sage.all import *
>>> A = algebras.FQSym(QQ).M()
>>> u = Permutation([Integer(2),Integer(1)])
>>> A.degree_on_basis(u)
2
```

a_realization()

Return a particular realization of `self` (the F-basis).

EXAMPLES:

```
sage: FQSym = algebras.FQSym(QQ)
sage: FQSym.a_realization()
Free Quasi-symmetric functions over Rational Field in the F basis
```

```
>>> from sage.all import *
>>> FQSym = algebras.FQSym(QQ)
>>> FQSym.a_realization()
Free Quasi-symmetric functions over Rational Field in the F basis
```

5.1.111 Free modules

class sage.combinat.free_module.**CartesianProductWithFlattening**(*flatten*)

Bases: object

A class for Cartesian product constructor, with partial flattening

class sage.combinat.free_module.**CombinatorialFreeModule**(*R*, *basis_keys=None*, *element_class=None*, *category=None*, *prefix=None*, *names=None*, ***kwds*)

Bases: UniqueRepresentation, Module, IndexedGenerators

Class for free modules with a named basis.

INPUT:

- R – base ring
- basis_keys – list; tuple, family, set, etc. defining the indexing set for the basis of this module
- element_class – the class of which elements of this module should be instances (default: None, in which case the elements are instances of IndexedFreeModuleElement)
- category – the category in which this module lies (optional, default None, in which case use the "category of modules with basis" over the base ring R); this should be a subcategory of ModulesWithBasis

For the options controlling the printing of elements, see IndexedGenerators.

Note: These print options may also be accessed and modified using the print_options() method, after the module has been defined.

EXAMPLES:

We construct a free module whose basis is indexed by the letters a, b, c:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: F
Free module generated by {'a', 'b', 'c'} over Rational Field
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a','b','c'])
>>> F
Free module generated by {'a', 'b', 'c'} over Rational Field
```

Its basis is a family, indexed by a, b, c:

```
sage: e = F.basis()
sage: e
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
```

```
>>> from sage.all import *
>>> e = F.basis()
>>> e
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
```

```
sage: [x for x in e]
[B['a'], B['b'], B['c']]
sage: [k for k in e.keys()]
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> [x for x in e]
[B['a'], B['b'], B['c']]
>>> [k for k in e.keys()]
['a', 'b', 'c']
```

Let us construct some elements, and compute with them:

```
sage: e['a']
B['a']
sage: 2*e['a']
2*B['a']
sage: e['a'] + 3*e['b']
B['a'] + 3*B['b']
```

```
>>> from sage.all import *
>>> e['a']
B['a']
>>> Integer(2)*e['a']
2*B['a']
>>> e['a'] + Integer(3)*e['b']
B['a'] + 3*B['b']
```

Some uses of `sage.categories.commutative_additive_semigroups.CommutativeAdditiveSemigroups.ParentMethods.summation()` and *sum()*:

```
sage: F = CombinatorialFreeModule(QQ, [1,2,3,4])
sage: F.summation(F.monomial(1), F.monomial(3))
B[1] + B[3]

sage: F = CombinatorialFreeModule(QQ, [1,2,3,4])
sage: F.sum(F.monomial(i) for i in [1,2,3])
B[1] + B[2] + B[3]
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),Integer(4)])
>>> F.summation(F.monomial(Integer(1)), F.monomial(Integer(3)))
B[1] + B[3]

>>> F = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),Integer(4)])
>>> F.sum(F.monomial(i) for i in [Integer(1),Integer(2),Integer(3)])
B[1] + B[2] + B[3]
```

Note that free modules with a given basis and parameters are unique:

```
sage: F1 = CombinatorialFreeModule(QQ, (1,2,3,4))
sage: F1 is F
True
```

```
>>> from sage.all import *
>>> F1 = CombinatorialFreeModule(QQ, (Integer(1),Integer(2),Integer(3),
↪Integer(4)))
>>> F1 is F
True
```

The identity of the constructed free module depends on the order of the basis and on the other parameters, like the prefix. Note that *CombinatorialFreeModule* is a UniqueRepresentation. Hence, two combinatorial free modules evaluate equal if and only if they are identical:

```
sage: F1 = CombinatorialFreeModule(QQ, (1,2,3,4))
sage: F1 is F
True
sage: F1 = CombinatorialFreeModule(QQ, [4,3,2,1])
sage: F1 == F
False
sage: F2 = CombinatorialFreeModule(QQ, [1,2,3,4], prefix='F')
sage: F2 == F
False
```

```
>>> from sage.all import *
>>> F1 = CombinatorialFreeModule(QQ, (Integer(1),Integer(2),Integer(3),
↪Integer(4)))
>>> F1 is F
True
>>> F1 = CombinatorialFreeModule(QQ, [Integer(4),Integer(3),Integer(2),
↪Integer(1)])
>>> F1 == F
False
>>> F2 = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
↪Integer(4)], prefix='F')
>>> F2 == F
False
```

Because of this, if you create a free module with certain parameters and then modify its prefix or other print options, this affects all modules which were defined using the same parameters.

```
sage: F2.print_options(prefix='x')
sage: F2.prefix()
'x'
sage: F3 = CombinatorialFreeModule(QQ, [1,2,3,4], prefix='F')
sage: F3 is F2   # F3 was defined just like F2
True
sage: F3.prefix()
'x'
sage: F4 = CombinatorialFreeModule(QQ, [1,2,3,4], prefix='F', bracket=True)
sage: F4 == F2   # F4 was NOT defined just like F2
False
sage: F4.prefix()
'F'

sage: F2.print_options(prefix='F') #reset for following doctests
```

```
>>> from sage.all import *
>>> F2.print_options(prefix='x')
>>> F2.prefix()
'x'
>>> F3 = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
↪Integer(4)], prefix='F')
>>> F3 is F2   # F3 was defined just like F2
True
>>> F3.prefix()
'x'
>>> F4 = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
↪Integer(4)], prefix='F', bracket=True)
>>> F4 == F2   # F4 was NOT defined just like F2
False
>>> F4.prefix()
'F'

>>> F2.print_options(prefix='F') #reset for following doctests
```

The constructed module is in the category of modules with basis over the base ring:

```
sage: CombinatorialFreeModule(QQ, Partitions()).category()                      #␣
↪needs sage.combinat
Category of vector spaces with basis over Rational Field
```

```
>>> from sage.all import *
>>> CombinatorialFreeModule(QQ, Partitions()).category()                      #␣
↪needs sage.combinat
Category of vector spaces with basis over Rational Field
```

If furthermore the index set is finite (i.e. in the category `Sets().Finite()`), then the module is declared as being finite dimensional:

```
sage: CombinatorialFreeModule(QQ, [1,2,3,4]).category()
Category of finite dimensional vector spaces with basis over Rational Field
sage: CombinatorialFreeModule(QQ, Partitions(3),                                #␣
↪needs sage.combinat
....:                         category=Algebras(QQ).WithBasis()).category()
Category of finite dimensional algebras with basis over Rational Field
```

```
>>> from sage.all import *
>>> CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),Integer(4)]).
↪category()
Category of finite dimensional vector spaces with basis over Rational Field
>>> CombinatorialFreeModule(QQ, Partitions(Integer(3)),                         ␣
↪     # needs sage.combinat
...                         category=Algebras(QQ).WithBasis()).category()
Category of finite dimensional algebras with basis over Rational Field
```

See `sage.categories.examples.algebras_with_basis` and `sage.categories.examples.hopf_algebras_with_basis` for illustrations of the use of the `category` keyword, and see *`sage.combinat.root_system.weight_space.WeightSpace`* for an example of the use of `element_class`.

Customizing print and LaTeX representations of elements:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b'], prefix='x')
sage: original_print_options = F.print_options()
sage: sorted(original_print_options.items())
[('bracket', None),
 ('iterate_key', False),
 ('latex_bracket', False), ('latex_names', None),
 ('latex_prefix', None), ('latex_scalar_mult', None),
 ('names', None), ('prefix', 'x'),
 ('scalar_mult', '*'),
 ('sorting_key', <function ...<lambda> at ...>),
 ('sorting_reverse', False), ('string_quotes', True),
 ('tensor_symbol', None)]

sage: e = F.basis()
sage: e['a'] - 3 * e['b']
x['a'] - 3*x['b']

sage: F.print_options(prefix='x', scalar_mult=' ', bracket='{')
sage: e['a'] - 3 * e['b']
x{'a'} - 3 x{'b'}
sage: latex(e['a'] - 3 * e['b'])
x_{a} - 3 x_{b}

sage: F.print_options(latex_prefix='y')
sage: latex(e['a'] - 3 * e['b'])
y_{a} - 3  y_{b}

sage: F.print_options(sorting_reverse=True)
sage: e['a'] - 3 * e['b']
-3 x{'b'} + x{'a'}
sage: F.print_options(**original_print_options) # reset print options

sage: F = CombinatorialFreeModule(QQ, [(1,2), (3,4)])
sage: e = F.basis()
sage: e[(1,2)] - 3 * e[(3,4)]
B[(1, 2)] - 3*B[(3, 4)]

sage: F.print_options(bracket=['_{', '}'])
sage: e[(1,2)] - 3 * e[(3,4)]
B_{(1, 2)} - 3*B_{(3, 4)}

sage: F.print_options(prefix='', bracket=False)
sage: e[(1,2)] - 3 * e[(3,4)]
(1, 2) - 3*(3, 4)
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a','b'], prefix='x')
>>> original_print_options = F.print_options()
>>> sorted(original_print_options.items())
[('bracket', None),
 ('iterate_key', False),
 ('latex_bracket', False), ('latex_names', None),
```

(continues on next page)

(continued from previous page)

```
 ('latex_prefix', None), ('latex_scalar_mult', None),
 ('names', None), ('prefix', 'x'),
 ('scalar_mult', '*'),
 ('sorting_key', <function ...<lambda> at ...>),
 ('sorting_reverse', False), ('string_quotes', True),
 ('tensor_symbol', None)]

>>> e = F.basis()
>>> e['a'] - Integer(3) * e['b']
x['a'] - 3*x['b']

>>> F.print_options(prefix='x', scalar_mult=' ', bracket='{')
>>> e['a'] - Integer(3) * e['b']
x{'a'} - 3 x{'b'}
>>> latex(e['a'] - Integer(3) * e['b'])
x_{a} - 3 x_{b}

>>> F.print_options(latex_prefix='y')
>>> latex(e['a'] - Integer(3) * e['b'])
y_{a} - 3  y_{b}

>>> F.print_options(sorting_reverse=True)
>>> e['a'] - Integer(3) * e['b']
-3 x{'b'} + x{'a'}
>>> F.print_options(**original_print_options) # reset print options

>>> F = CombinatorialFreeModule(QQ, [(Integer(1),Integer(2)), (Integer(3),
↪Integer(4))])
>>> e = F.basis()
>>> e[(Integer(1),Integer(2))] - Integer(3) * e[(Integer(3),Integer(4))]
B[(1, 2)] - 3*B[(3, 4)]

>>> F.print_options(bracket=['_{', '}'])
>>> e[(Integer(1),Integer(2))] - Integer(3) * e[(Integer(3),Integer(4))]
B_{(1, 2)} - 3*B_{(3, 4)}

>>> F.print_options(prefix='', bracket=False)
>>> e[(Integer(1),Integer(2))] - Integer(3) * e[(Integer(3),Integer(4))]
(1, 2) - 3*(3, 4)
```

CartesianProduct

alias of *CombinatorialFreeModule_CartesianProduct*

Element

alias of IndexedFreeModuleElement

Tensor

alias of *CombinatorialFreeModule_Tensor*

change_ring(R)

Return the base change of self to R.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, ['a','b','c']); F
Free module generated by {'a', 'b', 'c'} over Integer Ring
sage: F_QQ = F.change_ring(QQ); F_QQ
Free module generated by {'a', 'b', 'c'} over Rational Field
sage: F_QQ.change_ring(ZZ) == F
True

sage: T = F.tensor(F); T
Free module generated by {'a', 'b', 'c'} over Integer Ring
 # Free module generated by {'a', 'b', 'c'} over Integer Ring
sage: T.change_ring(QQ)
Free module generated by {'a', 'b', 'c'} over Rational Field
 # Free module generated by {'a', 'b', 'c'} over Rational Field

sage: G = CombinatorialFreeModule(ZZ, ['x','y']); G
Free module generated by {'x', 'y'} over Integer Ring
sage: T = F.cartesian_product(G); T
Free module generated by {'a', 'b', 'c'} over Integer Ring
 (+) Free module generated by {'x', 'y'} over Integer Ring
sage: T.change_ring(QQ)
Free module generated by {'a', 'b', 'c'} over Rational Field
 (+) Free module generated by {'x', 'y'} over Rational Field
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, ['a','b','c']); F
Free module generated by {'a', 'b', 'c'} over Integer Ring
>>> F_QQ = F.change_ring(QQ); F_QQ
Free module generated by {'a', 'b', 'c'} over Rational Field
>>> F_QQ.change_ring(ZZ) == F
True

>>> T = F.tensor(F); T
Free module generated by {'a', 'b', 'c'} over Integer Ring
 # Free module generated by {'a', 'b', 'c'} over Integer Ring
>>> T.change_ring(QQ)
Free module generated by {'a', 'b', 'c'} over Rational Field
 # Free module generated by {'a', 'b', 'c'} over Rational Field

>>> G = CombinatorialFreeModule(ZZ, ['x','y']); G
Free module generated by {'x', 'y'} over Integer Ring
>>> T = F.cartesian_product(G); T
Free module generated by {'a', 'b', 'c'} over Integer Ring
 (+) Free module generated by {'x', 'y'} over Integer Ring
>>> T.change_ring(QQ)
Free module generated by {'a', 'b', 'c'} over Rational Field
 (+) Free module generated by {'x', 'y'} over Rational Field
```

construction()

The construction functor and base ring for `self`.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'],␣
↪category=AlgebrasWithBasis(QQ))
sage: F.construction()
(VectorFunctor, Rational Field)
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a','b','c'],␣
↪category=AlgebrasWithBasis(QQ))
>>> F.construction()
(VectorFunctor, Rational Field)
```

dimension()

Return the dimension of the free module (which is given by the number of elements in the basis).

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.dimension()
3
sage: F.basis().cardinality()
3
sage: F.basis().keys().cardinality()
3
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> F.dimension()
3
>>> F.basis().cardinality()
3
>>> F.basis().keys().cardinality()
3
```

Rank is available as a synonym:

```
sage: F.rank()
3
```

```
>>> from sage.all import *
>>> F.rank()
3
```

```
sage: s = SymmetricFunctions(QQ).schur()                                    #␣
↪needs sage.combinat
sage: s.dimension()                                                         #␣
↪needs sage.combinat
+Infinity
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).schur()                                  #␣
↪needs sage.combinat
>>> s.dimension()                                                       #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
+Infinity
```

element_class()

The (default) class for the elements of this parent.

Overrides `Parent.element_class()` to force the construction of Python class. This is currently needed to inherit really all the features from categories, and in particular the initialization of `_mul_` in `Magmas.ParentMethods.__init_extra__()`.

EXAMPLES:

```
sage: # needs sage.combinat
sage: A = Algebras(QQ).WithBasis().example(); A
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field
sage: A.element_class.mro()
[<class 'sage.categories.examples.algebras_with_basis.FreeAlgebra_with_
↪category.element_class'>,
 <class 'sage.modules.with_basis.indexed_element.IndexedFreeModuleElement'>,
 ...]
sage: a,b,c = A.algebra_generators()
sage: a * b
B[word: ab]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = Algebras(QQ).WithBasis().example(); A
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field
>>> A.element_class.mro()
[<class 'sage.categories.examples.algebras_with_basis.FreeAlgebra_with_
↪category.element_class'>,
 <class 'sage.modules.with_basis.indexed_element.IndexedFreeModuleElement'>,
 ...]
>>> a,b,c = A.algebra_generators()
>>> a * b
B[word: ab]
```

from_vector(*vector*, *order=None*, *coerce=True*)

Build an element of `self` from a (sparse) vector.

See also

`get_order()`, `CombinatorialFreeModule.Element._vector_()`

EXAMPLES:

```
sage: # needs sage.combinat
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: b = QS3.from_vector(vector((2, 0, 0, 0, 0, 4))); b
2*[1, 2, 3] + 4*[3, 2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: a = 2*QS3([1,2,3]) + 4*QS3([3,2,1])
sage: a == b
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> b = QS3.from_vector(vector((Integer(2), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(4)))); b
2*[1, 2, 3] + 4*[3, 2, 1]
>>> a = Integer(2)*QS3([Integer(1),Integer(2),Integer(3)]) +␣
↪Integer(4)*QS3([Integer(3),Integer(2),Integer(1)])
>>> a == b
True
```

get_order()

Return the order of the elements in the basis.

EXAMPLES:

```
sage: QS2 = SymmetricGroupAlgebra(QQ,2)                                 #␣
↪needs sage.combinat
sage: QS2.get_order()  # note: order changed on 2009-03-13              #␣
↪needs sage.combinat
[[2, 1], [1, 2]]
```

```
>>> from sage.all import *
>>> QS2 = SymmetricGroupAlgebra(QQ,Integer(2))                          ␣
↪     # needs sage.combinat
>>> QS2.get_order()  # note: order changed on 2009-03-13              #␣
↪needs sage.combinat
[[2, 1], [1, 2]]
```

get_order_key()

Return a comparison key on the basis indices that is compatible with the current term order.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
sage: A.set_order(['x', 'y', 'a', 'b'])
sage: Akey = A.get_order_key()
sage: sorted(A.basis().keys(), key=Akey)
['x', 'y', 'a', 'b']
sage: A.set_order(list(reversed(A.basis().keys())))
sage: Akey = A.get_order_key()
sage: sorted(A.basis().keys(), key=Akey)
['b', 'a', 'y', 'x']
```

```
>>> from sage.all import *
>>> A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
>>> A.set_order(['x', 'y', 'a', 'b'])
>>> Akey = A.get_order_key()
```

(continues on next page)

(continued from previous page)

```
>>> sorted(A.basis().keys(), key=Akey)
['x', 'y', 'a', 'b']
>>> A.set_order(list(reversed(A.basis().keys())))
>>> Akey = A.get_order_key()
>>> sorted(A.basis().keys(), key=Akey)
['b', 'a', 'y', 'x']
```

is_exact()

Return `True` if elements of `self` have exact representations, which is true of `self` if and only if it is true of `self.basis().keys()` and `self.base_ring()`.

EXAMPLES:

```
sage: GroupAlgebra(GL(3, GF(7))).is_exact()                                  #␣
↪needs sage.groups sage.rings.finite_rings
True
sage: GroupAlgebra(GL(3, GF(7)), RR).is_exact()                              #␣
↪needs sage.groups sage.rings.finite_rings
False
sage: GroupAlgebra(GL(3, pAdicRing(7))).is_exact()   # not implemented, needs␣
↪sage.groups sage.rings.padics
False
```

```
>>> from sage.all import *
>>> GroupAlgebra(GL(Integer(3), GF(Integer(7)))).is_exact()                   ␣
↪                # needs sage.groups sage.rings.finite_rings
True
>>> GroupAlgebra(GL(Integer(3), GF(Integer(7))), RR).is_exact()               ␣
↪                # needs sage.groups sage.rings.finite_rings
False
>>> GroupAlgebra(GL(Integer(3), pAdicRing(Integer(7)))).is_exact()   # not␣
↪implemented, needs sage.groups sage.rings.padics
False
```

linear_combination(*iter_of_elements_coeff*, *factor_on_left=True*)

Return the linear combination $\lambda_1 v_1 + \cdots + \lambda_k v_k$ (resp. the linear combination $v_1 \lambda_1 + \cdots + v_k \lambda_k$) where `iter_of_elements_coeff` iterates through the sequence $((v_1, \lambda_1), ..., (v_k, \lambda_k))$.

INPUT:

- `iter_of_elements_coeff` – iterator of pairs `(element, coeff)` with `element` in `self` and `coeff` in `self.base_ring()`
- `factor_on_left` – (optional) if `True`, the coefficients are multiplied from the left if `False`, the coefficients are multiplied from the right

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, [1,2])
sage: f = F.an_element(); f
2*B[1] + 2*B[2]
sage: F.linear_combination( (f,i) for i in range(5) )
20*B[1] + 20*B[2]
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, [Integer(1),Integer(2)])
>>> f = F.an_element(); f
2*B[1] + 2*B[2]
>>> F.linear_combination( (f,i) for i in range(Integer(5)) )
20*B[1] + 20*B[2]
```

monomial()

Return the basis element indexed by `i`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.monomial('a')
B['a']
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> F.monomial('a')
B['a']
```

`F.monomial` is in fact (almost) a map:

```
sage: F.monomial
Term map from {'a', 'b', 'c'} to Free module generated by {'a', 'b', 'c'}␣
↪over Rational Field
```

```
>>> from sage.all import *
>>> F.monomial
Term map from {'a', 'b', 'c'} to Free module generated by {'a', 'b', 'c'}␣
↪over Rational Field
```

rank()

Return the dimension of the free module (which is given by the number of elements in the basis).

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.dimension()
3
sage: F.basis().cardinality()
3
sage: F.basis().keys().cardinality()
3
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> F.dimension()
3
>>> F.basis().cardinality()
```

(continues on next page)

(continued from previous page)

```
3
>>> F.basis().keys().cardinality()
3
```

Rank is available as a synonym:

```
sage: F.rank()
3
```

```
>>> from sage.all import *
>>> F.rank()
3
```

```
sage: s = SymmetricFunctions(QQ).schur()                                    #␣
↪needs sage.combinat
sage: s.dimension()                                                         #␣
↪needs sage.combinat
+Infinity
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).schur()                                    #␣
↪needs sage.combinat
>>> s.dimension()                                                         #␣
↪needs sage.combinat
+Infinity
```

set_order(*order*)

Set the order of the elements of the basis.

If `set_order()` has not been called, then the ordering is the one used in the generation of the elements of `self`'s associated enumerated set.

> **Warning**
>
> Many cached methods depend on this order, in particular for constructing subspaces and quotients. Changing the order after some computations have been cached does not invalidate the cache, and is likely to introduce inconsistencies.

EXAMPLES:

```
sage: # needs sage.combinat
sage: QS2 = SymmetricGroupAlgebra(QQ,2)
sage: b = list(QS2.basis().keys())
sage: b.reverse()
sage: QS2.set_order(b)
sage: QS2.get_order()
[[2, 1], [1, 2]]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> QS2 = SymmetricGroupAlgebra(QQ,Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> b = list(QS2.basis().keys())
>>> b.reverse()
>>> QS2.set_order(b)
>>> QS2.get_order()
[[2, 1], [1, 2]]
```

sum(*iter_of_elements*)

Return the sum of all elements in `iter_of_elements`.

Overrides method inherited from commutative additive monoid as it is much faster on dicts directly.

INPUT:

- `iter_of_elements` – iterator of elements of `self`

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ,[1,2])
sage: f = F.an_element(); f
2*B[1] + 2*B[2]
sage: F.sum( f for _ in range(5) )
10*B[1] + 10*B[2]
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ,[Integer(1),Integer(2)])
>>> f = F.an_element(); f
2*B[1] + 2*B[2]
>>> F.sum( f for _ in range(Integer(5)) )
10*B[1] + 10*B[2]
```

sum_of_terms(*terms*, *distinct=False*)

Construct a sum of terms of `self`.

INPUT:

- `terms` – list (or iterable) of pairs `(index, coeff)`
- `distinct` – boolean (default: `False`); whether the indices are guaranteed to be distinct

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.sum_of_terms([('a',2), ('c',3)])
2*B['a'] + 3*B['c']
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> F.sum_of_terms([('a',Integer(2)), ('c',Integer(3))])
2*B['a'] + 3*B['c']
```

If `distinct` is True, then the construction is optimized:

```
sage: F.sum_of_terms([('a',2), ('c',3)], distinct = True)
2*B['a'] + 3*B['c']
```

```
>>> from sage.all import *
>>> F.sum_of_terms([('a',Integer(2)), ('c',Integer(3))], distinct = True)
2*B['a'] + 3*B['c']
```

Warning

Use `distinct=True` only if you are sure that the indices are indeed distinct:

```
sage: F.sum_of_terms([('a',2), ('a',3)], distinct = True)
3*B['a']
```

```
>>> from sage.all import *
>>> F.sum_of_terms([('a',Integer(2)), ('a',Integer(3))], distinct = True)
3*B['a']
```

Extreme case:

```
sage: F.sum_of_terms([])
0
```

```
>>> from sage.all import *
>>> F.sum_of_terms([])
0
```

term(*index*, *coeff=None*)

Construct a term in `self`.

INPUT:

- `index` – the index of a basis element
- `coeff` – an element of the coefficient ring (default: one)

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.term('a',3)
3*B['a']
sage: F.term('a')
B['a']
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> F.term('a',Integer(3))
3*B['a']
>>> F.term('a')
B['a']
```

Design: should this do coercion on the coefficient ring?

zero()

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.zero()
0
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> F.zero()
0
```

class sage.combinat.free_module.**CombinatorialFreeModule_CartesianProduct**(*modules*, ***options*)

Bases: *CombinatorialFreeModule*

An implementation of Cartesian products of modules with basis.

EXAMPLES:

We construct two free modules, assign them short names, and construct their Cartesian product:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename('G')
sage: H = CombinatorialFreeModule(ZZ, [4,7]); H.rename('H')
sage: S = cartesian_product([F, G])
sage: S
F (+) G
sage: S.basis()
Lazy family (Term map
 from Disjoint union of Family ({4, 5}, {4, 6})
   to F (+) G(i))_{i in Disjoint union of Family ({4, 5}, {4, 6})}
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(4),Integer(5)]); F.rename('F')
>>> G = CombinatorialFreeModule(ZZ, [Integer(4),Integer(6)]); G.rename('G')
>>> H = CombinatorialFreeModule(ZZ, [Integer(4),Integer(7)]); H.rename('H')
>>> S = cartesian_product([F, G])
>>> S
F (+) G
>>> S.basis()
Lazy family (Term map
 from Disjoint union of Family ({4, 5}, {4, 6})
   to F (+) G(i))_{i in Disjoint union of Family ({4, 5}, {4, 6})}
```

Note that the indices of the basis elements of F and G intersect non trivially. This is handled by forcing the union to be disjoint:

```
sage: list(S.basis())
[B[(0, 4)], B[(0, 5)], B[(1, 4)], B[(1, 6)]]
```

```
>>> from sage.all import *
>>> list(S.basis())
[B[(0, 4)], B[(0, 5)], B[(1, 4)], B[(1, 6)]]
```

We now compute the Cartesian product of elements of free modules:

```
sage: f =   F.monomial(4) + 2*F.monomial(5)
sage: g = 2*G.monomial(4) +   G.monomial(6)
sage: h =   H.monomial(4) +   H.monomial(7)
sage: cartesian_product([f, g])
B[(0, 4)] + 2*B[(0, 5)] + 2*B[(1, 4)] + B[(1, 6)]
sage: cartesian_product([f, g, h])
B[(0, 4)] + 2*B[(0, 5)] + 2*B[(1, 4)] + B[(1, 6)] + B[(2, 4)] + B[(2, 7)]
sage: cartesian_product([f, g, h]).parent()
F (+) G (+) H
```

```
>>> from sage.all import *
>>> f =   F.monomial(Integer(4)) + Integer(2)*F.monomial(Integer(5))
>>> g = Integer(2)*G.monomial(Integer(4)) +   G.monomial(Integer(6))
>>> h =   H.monomial(Integer(4)) +   H.monomial(Integer(7))
>>> cartesian_product([f, g])
B[(0, 4)] + 2*B[(0, 5)] + 2*B[(1, 4)] + B[(1, 6)]
>>> cartesian_product([f, g, h])
B[(0, 4)] + 2*B[(0, 5)] + 2*B[(1, 4)] + B[(1, 6)] + B[(2, 4)] + B[(2, 7)]
>>> cartesian_product([f, g, h]).parent()
F (+) G (+) H
```

TODO: choose an appropriate semantic for Cartesian products of Cartesian products (associativity?):

```
sage: S = cartesian_product([cartesian_product([F, G]), H])  # todo: not␣
↪implemented
F (+) G (+) H
```

```
>>> from sage.all import *
>>> S = cartesian_product([cartesian_product([F, G]), H])  # todo: not implemented
F (+) G (+) H
```

class Element

Bases: `IndexedFreeModuleElement`

cartesian_embedding(*i*)

Return the natural embedding morphism of the `i`-th Cartesian factor (summand) of `self` into `self`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename('G')
sage: S = cartesian_product([F, G])
sage: phi = S.cartesian_embedding(0)
sage: phi(F.monomial(4) + 2 * F.monomial(5))
B[(0, 4)] + 2*B[(0, 5)]
sage: phi(F.monomial(4) + 2 * F.monomial(6)).parent() == S
True
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(4),Integer(5)]); F.rename('F')
```

(continues on next page)

(continued from previous page)

```
>>> G = CombinatorialFreeModule(ZZ, [Integer(4),Integer(6)]); G.rename('G')
>>> S = cartesian_product([F, G])
>>> phi = S.cartesian_embedding(Integer(0))
>>> phi(F.monomial(Integer(4)) + Integer(2) * F.monomial(Integer(5)))
B[(0, 4)] + 2*B[(0, 5)]
>>> phi(F.monomial(Integer(4)) + Integer(2) * F.monomial(Integer(6))).
↪parent() == S
True
```

cartesian_factors()

Return the factors of the Cartesian product.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename('G')
sage: S = cartesian_product([F, G])
sage: S.cartesian_factors()
(F, G)
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(4),Integer(5)]); F.rename('F')
>>> G = CombinatorialFreeModule(ZZ, [Integer(4),Integer(6)]); G.rename('G')
>>> S = cartesian_product([F, G])
>>> S.cartesian_factors()
(F, G)
```

cartesian_projection(i)

Return the natural projection onto the i-th Cartesian factor (summand) of `self`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename('G')
sage: S = cartesian_product([F, G])
sage: x = S.monomial((0,4)) + 2 * S.monomial((0,5)) + 3 * S.monomial((1,6))
sage: S.cartesian_projection(0)(x)
B[4] + 2*B[5]
sage: S.cartesian_projection(1)(x)
3*B[6]
sage: S.cartesian_projection(0)(x).parent() == F
True
sage: S.cartesian_projection(1)(x).parent() == G
True
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(4),Integer(5)]); F.rename('F')
>>> G = CombinatorialFreeModule(ZZ, [Integer(4),Integer(6)]); G.rename('G')
>>> S = cartesian_product([F, G])
```

(continues on next page)

(continued from previous page)

```
>>> x = S.monomial((Integer(0),Integer(4))) + Integer(2) * S.
↪monomial((Integer(0),Integer(5))) + Integer(3) * S.monomial((Integer(1),
↪Integer(6)))
>>> S.cartesian_projection(Integer(0))(x)
B[4] + 2*B[5]
>>> S.cartesian_projection(Integer(1))(x)
3*B[6]
>>> S.cartesian_projection(Integer(0))(x).parent() == F
True
>>> S.cartesian_projection(Integer(1))(x).parent() == G
True
```

summand_embedding(*i*)

Return the natural embedding morphism of the `i`-th Cartesian factor (summand) of `self` into `self`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename('G')
sage: S = cartesian_product([F, G])
sage: phi = S.cartesian_embedding(0)
sage: phi(F.monomial(4) + 2 * F.monomial(5))
B[(0, 4)] + 2*B[(0, 5)]
sage: phi(F.monomial(4) + 2 * F.monomial(6)).parent() == S
True
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(4),Integer(5)]); F.rename('F')
>>> G = CombinatorialFreeModule(ZZ, [Integer(4),Integer(6)]); G.rename('G')
>>> S = cartesian_product([F, G])
>>> phi = S.cartesian_embedding(Integer(0))
>>> phi(F.monomial(Integer(4)) + Integer(2) * F.monomial(Integer(5)))
B[(0, 4)] + 2*B[(0, 5)]
>>> phi(F.monomial(Integer(4)) + Integer(2) * F.monomial(Integer(6))).
↪parent() == S
True
```

summand_projection(*i*)

Return the natural projection onto the i-th Cartesian factor (summand) of `self`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename('G')
sage: S = cartesian_product([F, G])
sage: x = S.monomial((0,4)) + 2 * S.monomial((0,5)) + 3 * S.monomial((1,6))
sage: S.cartesian_projection(0)(x)
```

(continues on next page)

(continued from previous page)

```
B[4] + 2*B[5]
sage: S.cartesian_projection(1)(x)
3*B[6]
sage: S.cartesian_projection(0)(x).parent() == F
True
sage: S.cartesian_projection(1)(x).parent() == G
True
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(4),Integer(5)]); F.rename('F')
>>> G = CombinatorialFreeModule(ZZ, [Integer(4),Integer(6)]); G.rename('G')
>>> S = cartesian_product([F, G])
>>> x = S.monomial((Integer(0),Integer(4))) + Integer(2) * S.
↪monomial((Integer(0),Integer(5))) + Integer(3) * S.monomial((Integer(1),
↪Integer(6)))
>>> S.cartesian_projection(Integer(0))(x)
B[4] + 2*B[5]
>>> S.cartesian_projection(Integer(1))(x)
3*B[6]
>>> S.cartesian_projection(Integer(0))(x).parent() == F
True
>>> S.cartesian_projection(Integer(1))(x).parent() == G
True
```

class sage.combinat.free_module.**CombinatorialFreeModule_Tensor**(*modules*, ***options*)

Bases: *CombinatorialFreeModule*

Tensor Product of Free Modules.

EXAMPLES:

We construct two free modules, assign them short names, and construct their tensor product:

```
sage: F = CombinatorialFreeModule(ZZ, [1,2]); F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [3,4]); G.rename('G')
sage: T = tensor([F, G]); T
F # G

sage: T.category()
Category of tensor products of
 finite dimensional modules with basis over Integer Ring

sage: T.construction() # todo: not implemented
[tensor, ]
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2)]); F.rename('F')
>>> G = CombinatorialFreeModule(ZZ, [Integer(3),Integer(4)]); G.rename('G')
>>> T = tensor([F, G]); T
F # G

>>> T.category()
Category of tensor products of
```

(continues on next page)

(continued from previous page)

```
 finite dimensional modules with basis over Integer Ring

>>> T.construction() # todo: not implemented
[tensor, ]
```

T is a free module, with same base ring as F and G:

```
sage: T.base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> T.base_ring()
Integer Ring
```

The basis of T is indexed by tuples of basis indices of F and G:

```
sage: T.basis().keys()
Image of Cartesian product of {1, 2}, {3, 4}
      by The map <class 'tuple'> from Cartesian product of {1, 2}, {3, 4}
sage: T.basis().keys().list()
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

```
>>> from sage.all import *
>>> T.basis().keys()
Image of Cartesian product of {1, 2}, {3, 4}
      by The map <class 'tuple'> from Cartesian product of {1, 2}, {3, 4}
>>> T.basis().keys().list()
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

FIXME: Should elements of a CartesianProduct be tuples (making them hashable)?

Here are the basis elements themselves:

```
sage: T.basis().cardinality()
4
sage: list(T.basis())
[B[1] # B[3], B[1] # B[4], B[2] # B[3], B[2] # B[4]]
```

```
>>> from sage.all import *
>>> T.basis().cardinality()
4
>>> list(T.basis())
[B[1] # B[3], B[1] # B[4], B[2] # B[3], B[2] # B[4]]
```

The tensor product is associative and flattens sub tensor products:

```
sage: H = CombinatorialFreeModule(ZZ, [5,6]); H.rename('H')
sage: tensor([F, tensor([G, H])])
F # G # H
sage: tensor([tensor([F, G]), H])
F # G # H
sage: tensor([F, G, H])
F # G # H
```

```
>>> from sage.all import *
>>> H = CombinatorialFreeModule(ZZ, [Integer(5),Integer(6)]); H.rename('H')
>>> tensor([F, tensor([G, H])])
F # G # H
>>> tensor([tensor([F, G]), H])
F # G # H
>>> tensor([F, G, H])
F # G # H
```

We now compute the tensor product of elements of free modules:

```
sage: f =   F.monomial(1) + 2 * F.monomial(2)
sage: g = 2*G.monomial(3) +     G.monomial(4)
sage: h =   H.monomial(5) +     H.monomial(6)
sage: tensor([f, g])
2*B[1] # B[3] + B[1] # B[4] + 4*B[2] # B[3] + 2*B[2] # B[4]
```

```
>>> from sage.all import *
>>> f =   F.monomial(Integer(1)) + Integer(2) * F.monomial(Integer(2))
>>> g = Integer(2)*G.monomial(Integer(3)) +     G.monomial(Integer(4))
>>> h =   H.monomial(Integer(5)) +     H.monomial(Integer(6))
>>> tensor([f, g])
2*B[1] # B[3] + B[1] # B[4] + 4*B[2] # B[3] + 2*B[2] # B[4]
```

Again, the tensor product is associative on elements:

```
sage: tensor([f, tensor([g, h])]) == tensor([f, g, h])
True
sage: tensor([tensor([f, g]), h]) == tensor([f, g, h])
True
```

```
>>> from sage.all import *
>>> tensor([f, tensor([g, h])]) == tensor([f, g, h])
True
>>> tensor([tensor([f, g]), h]) == tensor([f, g, h])
True
```

Note further that the tensor product spaces need not preexist:

```
sage: t = tensor([f, g, h])
sage: t.parent()
F # G # H
```

```
>>> from sage.all import *
>>> t = tensor([f, g, h])
>>> t.parent()
F # G # H
```

tensor_constructor(*modules*)

INPUT:

- `modules` – tuple $(F_1, \ldots, F_n)$ of free modules whose tensor product is self

Returns the canonical multilinear morphism from $F_1 \times \cdots \times F_n$ to $F_1 \otimes \cdots \otimes F_n$

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [1,2]); F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [3,4]); G.rename('G')
sage: H = CombinatorialFreeModule(ZZ, [5,6]); H.rename('H')

sage: f =   F.monomial(1) + 2*F.monomial(2)
sage: g = 2*G.monomial(3) +   G.monomial(4)
sage: h =   H.monomial(5) +   H.monomial(6)
sage: FG = tensor([F, G])
sage: phi_fg = FG.tensor_constructor((F, G))
sage: phi_fg(f, g)
2*B[1] # B[3] + B[1] # B[4] + 4*B[2] # B[3] + 2*B[2] # B[4]

sage: FGH = tensor([F, G, H])
sage: phi_fgh = FGH.tensor_constructor((F, G, H))
sage: phi_fgh(f, g, h)
2*B[1] # B[3] # B[5] + 2*B[1] # B[3] # B[6] + B[1] # B[4] # B[5]
+ B[1] # B[4] # B[6] + 4*B[2] # B[3] # B[5] + 4*B[2] # B[3] # B[6]
+ 2*B[2] # B[4] # B[5] + 2*B[2] # B[4] # B[6]

sage: phi_fg_h = FGH.tensor_constructor((FG, H))
sage: phi_fg_h(phi_fg(f, g), h)
2*B[1] # B[3] # B[5] + 2*B[1] # B[3] # B[6] + B[1] # B[4] # B[5]
+ B[1] # B[4] # B[6] + 4*B[2] # B[3] # B[5] + 4*B[2] # B[3] # B[6]
+ 2*B[2] # B[4] # B[5] + 2*B[2] # B[4] # B[6]
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2)]); F.rename('F')
>>> G = CombinatorialFreeModule(ZZ, [Integer(3),Integer(4)]); G.rename('G')
>>> H = CombinatorialFreeModule(ZZ, [Integer(5),Integer(6)]); H.rename('H')

>>> f =   F.monomial(Integer(1)) + Integer(2)*F.monomial(Integer(2))
>>> g = Integer(2)*G.monomial(Integer(3)) +   G.monomial(Integer(4))
>>> h =   H.monomial(Integer(5)) +   H.monomial(Integer(6))
>>> FG = tensor([F, G])
>>> phi_fg = FG.tensor_constructor((F, G))
>>> phi_fg(f, g)
2*B[1] # B[3] + B[1] # B[4] + 4*B[2] # B[3] + 2*B[2] # B[4]

>>> FGH = tensor([F, G, H])
>>> phi_fgh = FGH.tensor_constructor((F, G, H))
>>> phi_fgh(f, g, h)
2*B[1] # B[3] # B[5] + 2*B[1] # B[3] # B[6] + B[1] # B[4] # B[5]
+ B[1] # B[4] # B[6] + 4*B[2] # B[3] # B[5] + 4*B[2] # B[3] # B[6]
+ 2*B[2] # B[4] # B[5] + 2*B[2] # B[4] # B[6]

>>> phi_fg_h = FGH.tensor_constructor((FG, H))
>>> phi_fg_h(phi_fg(f, g), h)
2*B[1] # B[3] # B[5] + 2*B[1] # B[3] # B[6] + B[1] # B[4] # B[5]
+ B[1] # B[4] # B[6] + 4*B[2] # B[3] # B[5] + 4*B[2] # B[3] # B[6]
+ 2*B[2] # B[4] # B[5] + 2*B[2] # B[4] # B[6]
```

tensor_factors()

Return the tensor factors of this tensor product.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [1,2])
sage: F.rename('F')
sage: G = CombinatorialFreeModule(ZZ, [3,4])
sage: G.rename('G')
sage: T = tensor([F, G]); T
F # G
sage: T.tensor_factors()
(F, G)
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2)])
>>> F.rename('F')
>>> G = CombinatorialFreeModule(ZZ, [Integer(3),Integer(4)])
>>> G.rename('G')
>>> T = tensor([F, G]); T
F # G
>>> T.tensor_factors()
(F, G)
```

5.1.112 Free Dendriform Algebras

AUTHORS:

Frédéric Chapoton (2017)

class sage.combinat.free_dendriform_algebra.**DendriformFunctor**(*vars*)

Bases: ConstructionFunctor

A constructor for dendriform algebras.

EXAMPLES:

```
sage: P = algebras.FreeDendriform(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F = P.construction()[0]; F
Dendriform[x,y]

sage: A = GF(5)['a,b']
sage: a, b = A.gens()
sage: F(A)
Free Dendriform algebra on 2 generators ['x', 'y']
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: f = A.hom([a+b,a-b],A)
sage: F(f)
Generic endomorphism of Free Dendriform algebra on 2 generators ['x', 'y']
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: F(f)(a * F(A)(x))
(a+b)*B[x[., .]]
```

```
>>> from sage.all import *
>>> P = algebras.FreeDendriform(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F = P.construction()[Integer(0)]; F
Dendriform[x,y]

>>> A = GF(Integer(5))['a,b']
>>> a, b = A.gens()
>>> F(A)
Free Dendriform algebra on 2 generators ['x', 'y']
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> f = A.hom([a+b,a-b],A)
>>> F(f)
Generic endomorphism of Free Dendriform algebra on 2 generators ['x', 'y']
 over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> F(f)(a * F(A)(x))
(a+b)*B[x[., .]]
```

merge(*other*)

Merge `self` with another construction functor, or return `None`.

EXAMPLES:

```
sage: F = sage.combinat.free_dendriform_algebra.DendriformFunctor(['x','y'])
sage: G = sage.combinat.free_dendriform_algebra.DendriformFunctor(['t'])
sage: F.merge(G)
Dendriform[x,y,t]
sage: F.merge(F)
Dendriform[x,y]
```

```
>>> from sage.all import *
>>> F = sage.combinat.free_dendriform_algebra.DendriformFunctor(['x','y'])
>>> G = sage.combinat.free_dendriform_algebra.DendriformFunctor(['t'])
>>> F.merge(G)
Dendriform[x,y,t]
>>> F.merge(F)
Dendriform[x,y]
```

Now some actual use cases:

```
sage: R = algebras.FreeDendriform(ZZ, 'x,y,z')
sage: x,y,z = R.gens()
sage: 1/2 * x
1/2*B[x[., .]]
sage: parent(1/2 * x)
Free Dendriform algebra on 3 generators ['x', 'y', 'z'] over Rational Field

sage: S = algebras.FreeDendriform(QQ, 'zt')
sage: z,t = S.gens()
sage: x + t
B[t[., .]] + B[x[., .]]
```

(continues on next page)

(continued from previous page)

```
sage: parent(x + t)
Free Dendriform algebra on 4 generators ['z', 't', 'x', 'y'] over Rational
↪Field
```

```
>>> from sage.all import *
>>> R = algebras.FreeDendriform(ZZ, 'x,y,z')
>>> x,y,z = R.gens()
>>> Integer(1)/Integer(2) * x
1/2*B[x[., .]]
>>> parent(Integer(1)/Integer(2) * x)
Free Dendriform algebra on 3 generators ['x', 'y', 'z'] over Rational Field

>>> S = algebras.FreeDendriform(QQ, 'zt')
>>> z,t = S.gens()
>>> x + t
B[t[., .]] + B[x[., .]]
>>> parent(x + t)
Free Dendriform algebra on 4 generators ['z', 't', 'x', 'y'] over Rational
↪Field
```

rank = 9

class sage.combinat.free_dendriform_algebra.**FreeDendriformAlgebra**(*R*, *names=None*)

Bases: *CombinatorialFreeModule*

The free dendriform algebra.

Dendriform algebras are associative algebras, where the associative product $*$ is decomposed as a sum of two binary operations

$$x * y = x \succ y + x \prec y$$

that satisfy the axioms:

$$(x \succ y) \prec z = x \succ (y \prec z),$$

$$(x \prec y) \prec z = x \prec (y * z).$$

$$(x * y) \succ z = x \succ (y \succ z).$$

The free Dendriform algebra on a given set E has an explicit description using (planar) binary trees, just as the free associative algebra can be described using words. The underlying vector space has a basis indexed by finite binary trees endowed with a map from their vertices to E. In this basis, the associative product of two (decorated) binary trees $S * T$ is the sum over all possible ways of identifying (glueing) the rightmost path in S and the leftmost path in T.

The decomposition of the associative product as the sum of two binary operations $\succ$ and $\prec$ is made by separating the terms according to the origin of the root vertex. For $x \succ y$, one keeps the terms where the root vertex comes from y, whereas for $x \prec y$ one keeps the terms where the root vertex comes from x.

The free dendriform algebra can also be considered as the free algebra over the Dendriform operad.

Note

The usual binary operator $*$ is used for the associative product.

EXAMPLES:

```
sage: F = algebras.FreeDendriform(ZZ, 'xyz')
sage: x,y,z = F.gens()
sage: (x * y) * z
B[x[., y[., z[., .]]]] + B[x[., z[y[., .], .]]] + B[y[x[., .], z[., .]]]
 + B[z[x[., y[., .]], .]] + B[z[y[x[., .], .], .]]
```

```
>>> from sage.all import *
>>> F = algebras.FreeDendriform(ZZ, 'xyz')
>>> x,y,z = F.gens()
>>> (x * y) * z
B[x[., y[., z[., .]]]] + B[x[., z[y[., .], .]]] + B[y[x[., .], z[., .]]]
 + B[z[x[., y[., .]], .]] + B[z[y[x[., .], .], .]]
```

The free dendriform algebra is associative:

```
sage: x * (y * z) == (x * y) * z
True
```

```
>>> from sage.all import *
>>> x * (y * z) == (x * y) * z
True
```

The associative product decomposes in two parts:

```
sage: x * y == F.prec(x, y) + F.succ(x, y)
True
```

```
>>> from sage.all import *
>>> x * y == F.prec(x, y) + F.succ(x, y)
True
```

The axioms hold:

```
sage: F.prec(F.succ(x, y), z) == F.succ(x, F.prec(y, z))
True
sage: F.prec(F.prec(x, y), z) == F.prec(x, y * z)
True
sage: F.succ(x * y, z) == F.succ(x, F.succ(y, z))
True
```

```
>>> from sage.all import *
>>> F.prec(F.succ(x, y), z) == F.succ(x, F.prec(y, z))
True
>>> F.prec(F.prec(x, y), z) == F.prec(x, y * z)
True
>>> F.succ(x * y, z) == F.succ(x, F.succ(y, z))
True
```

When there is only one generator, unlabelled trees are used instead:

```
sage: F1 = algebras.FreeDendriform(QQ)
sage: w = F1.gen(0); w
B[[., .]]
sage: w * w * w
B[[., [., [., .]]]] + B[[., [[., .], .]]] + B[[[., .], [., .]]]
 + B[[[., [., .]], .]] + B[[[[., .], .], .]]
```

```
>>> from sage.all import *
>>> F1 = algebras.FreeDendriform(QQ)
>>> w = F1.gen(Integer(0)); w
B[[., .]]
>>> w * w * w
B[[., [., [., .]]]] + B[[., [[., .], .]]] + B[[[., .], [., .]]]
 + B[[[., [., .]], .]] + B[[[[., .], .], .]]
```

The set E can be infinite:

```
sage: F = algebras.FreeDendriform(QQ, ZZ)
sage: w = F.gen(1); w
B[1[., .]]
sage: x = F.gen(2); x
B[-1[., .]]
sage: w*x
B[-1[1[., .], .]] + B[1[., -1[., .]]]
```

```
>>> from sage.all import *
>>> F = algebras.FreeDendriform(QQ, ZZ)
>>> w = F.gen(Integer(1)); w
B[1[., .]]
>>> x = F.gen(Integer(2)); x
B[-1[., .]]
>>> w*x
B[-1[1[., .], .]] + B[1[., -1[., .]]]
```

REFERENCES:

- [LR1998]

algebra_generators()

Return the generators of this algebra.

These are the binary trees with just one vertex.

EXAMPLES:

```
sage: A = algebras.FreeDendriform(ZZ, 'fgh'); A
Free Dendriform algebra on 3 generators ['f', 'g', 'h']
 over Integer Ring
sage: list(A.algebra_generators())
[B[f[., .]], B[g[., .]], B[h[., .]]]

sage: A = algebras.FreeDendriform(QQ, ['x1','x2'])
sage: list(A.algebra_generators())
[B[x1[., .]], B[x2[., .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(ZZ, 'fgh'); A
Free Dendriform algebra on 3 generators ['f', 'g', 'h']
 over Integer Ring
>>> list(A.algebra_generators())
[B[f[., .]], B[g[., .]], B[h[., .]]]

>>> A = algebras.FreeDendriform(QQ, ['x1','x2'])
>>> list(A.algebra_generators())
[B[x1[., .]], B[x2[., .]]]
```

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ, 'xy')
sage: A.an_element()
B[x[., .]] + 2*B[x[., x[., .]]] + 2*B[x[x[., .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ, 'xy')
>>> A.an_element()
B[x[., .]] + 2*B[x[., x[., .]]] + 2*B[x[x[., .], .]]
```

change_ring(R)

Return the free dendriform algebra in the same variables over R.

INPUT:

- `R` – a ring

EXAMPLES:

```
sage: A = algebras.FreeDendriform(ZZ, 'fgh')
sage: A.change_ring(QQ)
Free Dendriform algebra on 3 generators ['f', 'g', 'h'] over
Rational Field
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(ZZ, 'fgh')
>>> A.change_ring(QQ)
Free Dendriform algebra on 3 generators ['f', 'g', 'h'] over
Rational Field
```

construction()

Return a pair `(F, R)`, where `F` is a *DendriformFunctor* and R is a ring, such that `F(R)` returns `self`.

EXAMPLES:

```
sage: P = algebras.FreeDendriform(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F, R = P.construction()
sage: F
```

(continues on next page)

(continued from previous page)

```
Dendriform[x,y]
sage: R
Integer Ring
sage: F(ZZ) is P
True
sage: F(QQ)
Free Dendriform algebra on 2 generators ['x', 'y'] over Rational Field
```

```
>>> from sage.all import *
>>> P = algebras.FreeDendriform(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F, R = P.construction()
>>> F
Dendriform[x,y]
>>> R
Integer Ring
>>> F(ZZ) is P
True
>>> F(QQ)
Free Dendriform algebra on 2 generators ['x', 'y'] over Rational Field
```

coproduct_on_basis(*x*)

Return the coproduct of a binary tree.

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: x = A.gen(0)
sage: ascii_art(A.coproduct(A.one()))  # indirect doctest
1 # 1

sage: ascii_art(A.coproduct(x))  # indirect doctest
1 # B  + B  # 1
     o    o

sage: A = algebras.FreeDendriform(QQ, 'xyz')
sage: x, y, z = A.gens()
sage: w = A.under(z,A.over(x,y))
sage: A.coproduct(z)
B[.] # B[z[., .]] + B[z[., .]] # B[.]
sage: A.coproduct(w)
B[.] # B[x[z[., .], y[., .]]] + B[x[., .]] # B[z[., y[., .]]] +
B[x[., .]] # B[y[z[., .], .]] + B[x[., y[., .]]] # B[z[., .]] +
B[x[z[., .], .]] # B[y[., .]] + B[x[z[., .], y[., .]]] # B[.]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> x = A.gen(Integer(0))
>>> ascii_art(A.coproduct(A.one()))  # indirect doctest
1 # 1

>>> ascii_art(A.coproduct(x))  # indirect doctest
```

(continues on next page)

(continued from previous page)

```
1 # B  + B  # 1
     o     o

>>> A = algebras.FreeDendriform(QQ, 'xyz')
>>> x, y, z = A.gens()
>>> w = A.under(z,A.over(x,y))
>>> A.coproduct(z)
B[.] # B[z[., .]] + B[z[., .]] # B[.]
>>> A.coproduct(w)
B[.] # B[x[z[., .], y[., .]]] + B[x[., .]] # B[z[., y[., .]]] +
B[x[., .]] # B[y[z[., .], .]] + B[x[., y[., .]]] # B[z[., .]] +
B[x[z[., .], .]] # B[y[., .]] + B[x[z[., .], y[., .]]] # B[.]
```

degree_on_basis(*t*)

Return the degree of a binary tree in the free Dendriform algebra.

This is the number of vertices.

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ,'@')
sage: RT = A.basis().keys()
sage: u = RT([], '@')
sage: A.degree_on_basis(u.over(u))
2
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ,'@')
>>> RT = A.basis().keys()
>>> u = RT([], '@')
>>> A.degree_on_basis(u.over(u))
2
```

gen(*i*)

Return the i-th generator of the algebra.

INPUT:

- i – integer

EXAMPLES:

```
sage: F = algebras.FreeDendriform(ZZ, 'xyz')
sage: F.gen(0)
B[x[., .]]

sage: F.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

```
>>> from sage.all import *
>>> F = algebras.FreeDendriform(ZZ, 'xyz')
>>> F.gen(Integer(0))
```

(continues on next page)

(continued from previous page)

```
B[x[., .]]

>>> F.gen(Integer(4))
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

gens()

Return the generators of `self` (as an algebra).

EXAMPLES:

```
sage: A = algebras.FreeDendriform(ZZ, 'fgh')
sage: A.gens()
(B[f[., .]], B[g[., .]], B[h[., .]])
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(ZZ, 'fgh')
>>> A.gens()
(B[f[., .]], B[g[., .]], B[h[., .]])
```

one_basis()

Return the index of the unit.

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ, '@')
sage: A.one_basis()
.
sage: A = algebras.FreeDendriform(QQ, 'xy')
sage: A.one_basis()
.
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ, '@')
>>> A.one_basis()
.
>>> A = algebras.FreeDendriform(QQ, 'xy')
>>> A.one_basis()
.
```

over()

Return the over product.

The over product x/y is the binary tree obtained by grafting the root of y at the rightmost leaf of x.

The usual symbol for this operation is $/$.

See also

`product()`, *succ()*, *prec()*, *under()*

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = A.gen(0)
sage: A.over(x, x)
B[[., [., .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = A.gen(Integer(0))
>>> A.over(x, x)
B[[., [., .]]]
```

prec()

Return the $\prec$ dendriform product.

This is the sum over all possible ways to identify the rightmost path in x and the leftmost path in y, with the additional condition that the root vertex of the result comes from x.

The usual symbol for this operation is $\prec$.

See also

product(), *succ()*, *over()*, *under()*

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = A.gen(0)
sage: A.prec(x, x)
B[[., [., .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = A.gen(Integer(0))
>>> A.prec(x, x)
B[[., [., .]]]
```

prec_product_on_basis(*x*, *y*)

Return the $\prec$ dendriform product of two trees.

This is the sum over all possible ways of identifying the rightmost path in x and the leftmost path in y, with the additional condition that the root vertex of the result comes from x.

The usual symbol for this operation is $\prec$.

See also

- *product_on_basis()*, *succ_product_on_basis()*

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = RT([])
sage: A.prec_product_on_basis(x, x)
B[[., [., .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = RT([])
>>> A.prec_product_on_basis(x, x)
B[[., [., .]]]
```

product_on_basis(x, y)

Return the $*$ associative dendriform product of two trees.

This is the sum over all possible ways of identifying the rightmost path in x and the leftmost path in y. Every term corresponds to a shuffle of the vertices on the rightmost path in x and the vertices on the leftmost path in y.

See also

- *succ_product_on_basis()*, *prec_product_on_basis()*

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = RT([])
sage: A.product_on_basis(x, x)
B[[., [., .]]] + B[[[., .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = RT([])
>>> A.product_on_basis(x, x)
B[[., [., .]]] + B[[[., .], .]]
```

some_elements()

Return some elements of the free dendriform algebra.

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: A.some_elements()
[B[.],
 B[[., .]],
 B[[., [., .]]] + B[[[., .], .]],
 B[.] + B[[., [., .]]] + B[[[., .], .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> A.some_elements()
[B[.],
 B[[., .]],
 B[[., [., .]]] + B[[[., .], .]],
 B[.] + B[[., [., .]]] + B[[[., .], .]]]
```

With several generators:

```
sage: A = algebras.FreeDendriform(QQ, 'xy')
sage: A.some_elements()
[B[.],
 B[x[., .]],
 B[x[., x[., .]]] + B[x[x[., .], .]],
 B[.] + B[x[., x[., .]]] + B[x[x[., .], .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ, 'xy')
>>> A.some_elements()
[B[.],
 B[x[., .]],
 B[x[., x[., .]]] + B[x[x[., .], .]],
 B[.] + B[x[., x[., .]]] + B[x[x[., .], .]]]
```

succ()

Return the $\succ$ dendriform product.

This is the sum over all possible ways of identifying the rightmost path in x and the leftmost path in y, with the additional condition that the root vertex of the result comes from y.

The usual symbol for this operation is $\succ$.

See also

product(), *prec()*, *over()*, *under()*

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = A.gen(0)
sage: A.succ(x, x)
B[[[., .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = A.gen(Integer(0))
>>> A.succ(x, x)
B[[[., .], .]]
```

succ_product_on_basis(*x*, *y*)

Return the $\succ$ dendriform product of two trees.

This is the sum over all possible ways to identify the rightmost path in x and the leftmost path in y, with the additional condition that the root vertex of the result comes from y.

The usual symbol for this operation is $\succ$.

See also

- *product_on_basis()*, *prec_product_on_basis()*

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = RT([])
sage: A.succ_product_on_basis(x, x)
B[[[., .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = RT([])
>>> A.succ_product_on_basis(x, x)
B[[[., .], .]]
```

under()

Return the under product.

The over product $x\backslash y$ is the binary tree obtained by grafting the root of x at the leftmost leaf of y.

The usual symbol for this operation is $\backslash$.

See also

product(), *succ()*, *prec()*, *over()*

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = A.gen(0)
sage: A.under(x, x)
B[[[., .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = A.gen(Integer(0))
>>> A.under(x, x)
B[[[., .], .]]
```

variable_names()

Return the names of the variables.

EXAMPLES:

```
sage: R = algebras.FreeDendriform(QQ, 'xy')
sage: R.variable_names()
{'x', 'y'}
```

```
>>> from sage.all import *
>>> R = algebras.FreeDendriform(QQ, 'xy')
>>> R.variable_names()
{'x', 'y'}
```

5.1.113 Free Pre-Lie Algebras

AUTHORS:

- Florent Hivert, Frédéric Chapoton (2011)

class sage.combinat.free_prelie_algebra.**FreePreLieAlgebra**(*R*, *names=None*)

Bases: *CombinatorialFreeModule*

The free pre-Lie algebra.

Pre-Lie algebras are non-associative algebras, where the product $*$ satisfies

$$(x * y) * z - x * (y * z) = (x * z) * y - x * (z * y).$$

We use here the convention where the associator

$$(x, y, z) := (x * y) * z - x * (y * z)$$

is symmetric in its two rightmost arguments. This is sometimes called a right pre-Lie algebra.

They have appeared in numerical analysis and deformation theory.

The free Pre-Lie algebra on a given set E has an explicit description using rooted trees, just as the free associative algebra can be described using words. The underlying vector space has a basis indexed by finite rooted trees endowed with a map from their vertices to E. In this basis, the product of two (decorated) rooted trees $S * T$ is the sum over vertices of S of the rooted tree obtained by adding one edge from the root of T to the given vertex of S. The root of these trees is taken to be the root of S. The free pre-Lie algebra can also be considered as the free algebra over the PreLie operad.

Warning

The usual binary operator `*` can be used for the pre-Lie product. Beware that it but must be parenthesized properly, as the pre-Lie product is not associative. By default, a multiple product will be taken with left parentheses.

EXAMPLES:

```
sage: F = algebras.FreePreLie(ZZ, 'xyz')
sage: x,y,z = F.gens()
sage: (x * y) * z
```

(continues on next page)

(continued from previous page)

```
B[x[y[z[]]]] + B[x[y[], z[]]]
sage: (x * y) * z - x * (y * z) == (x * z) * y - x * (z * y)
True
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(ZZ, 'xyz')
>>> x,y,z = F.gens()
>>> (x * y) * z
B[x[y[z[]]]] + B[x[y[], z[]]]
>>> (x * y) * z - x * (y * z) == (x * z) * y - x * (z * y)
True
```

The free pre-Lie algebra is non-associative:

```
sage: x * (y * z) == (x * y) * z
False
```

```
>>> from sage.all import *
>>> x * (y * z) == (x * y) * z
False
```

The default product is with left parentheses:

```
sage: x * y * z == (x * y) * z
True
sage: x * y * z * x == ((x * y) * z) * x
True
```

```
>>> from sage.all import *
>>> x * y * z == (x * y) * z
True
>>> x * y * z * x == ((x * y) * z) * x
True
```

The NAP product as defined in [Liv2006] is also implemented on the same vector space:

```
sage: N = F.nap_product
sage: N(x*y,z*z)
B[x[y[], z[z[]]]]
```

```
>>> from sage.all import *
>>> N = F.nap_product
>>> N(x*y,z*z)
B[x[y[], z[z[]]]]
```

When `None` is given as input, unlabelled trees are used instead:

```
sage: F1 = algebras.FreePreLie(QQ, None)
sage: w = F1.gen(0); w
B[[]]
sage: w * w * w * w
B[[[[[]]]]] + B[[[[], []]]] + 3*B[[[], [[]]]] + B[[[], [], []]]
```

```
>>> from sage.all import *
>>> F1 = algebras.FreePreLie(QQ, None)
>>> w = F1.gen(Integer(0)); w
B[[]]
>>> w * w * w * w
B[[[[[]]]]] + B[[[[], []]]] + 3*B[[[], [[]]]] + B[[[], [], []]]
```

However, it is equally possible to use labelled trees instead:

```
sage: F1 = algebras.FreePreLie(QQ, 'q')
sage: w = F1.gen(0); w
B[q[]]
sage: w * w * w * w
B[q[q[q[q[]]]]] + B[q[q[q[], q[]]]] + 3*B[q[q[], q[q[]]]] + B[q[q[], q[], q[]]]
```

```
>>> from sage.all import *
>>> F1 = algebras.FreePreLie(QQ, 'q')
>>> w = F1.gen(Integer(0)); w
B[q[]]
>>> w * w * w * w
B[q[q[q[q[]]]]] + B[q[q[q[], q[]]]] + 3*B[q[q[], q[q[]]]] + B[q[q[], q[], q[]]]
```

The set E can be infinite:

```
sage: F = algebras.FreePreLie(QQ, ZZ)
sage: w = F.gen(1); w
B[1[]]
sage: x = F.gen(2); x
B[-1[]]
sage: y = F.gen(3); y
B[2[]]
sage: w*x
B[1[-1[]]]
sage: (w*x)*y
B[1[-1[2[]]]] + B[1[-1[], 2[]]]
sage: w*(x*y)
B[1[-1[2[]]]]
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(QQ, ZZ)
>>> w = F.gen(Integer(1)); w
B[1[]]
>>> x = F.gen(Integer(2)); x
B[-1[]]
>>> y = F.gen(Integer(3)); y
B[2[]]
>>> w*x
B[1[-1[]]]
>>> (w*x)*y
B[1[-1[2[]]]] + B[1[-1[], 2[]]]
>>> w*(x*y)
B[1[-1[2[]]]]
```

Elements of a free pre-Lie algebra can be lifted to the universal enveloping algebra of the associated Lie algebra. The universal enveloping algebra is the Grossman-Larson Hopf algebra:

```
sage: F = algebras.FreePreLie(QQ,'abc')
sage: a,b,c = F.gens()
sage: (a*b+b*c).lift()
B[#[a[b[]]]] + B[#[b[c[]]]]
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(QQ,'abc')
>>> a,b,c = F.gens()
>>> (a*b+b*c).lift()
B[#[a[b[]]]] + B[#[b[c[]]]]
```

Note

Variables names can be `None`, a list of strings, a string or an integer. When `None` is given, unlabelled rooted trees are used. When a single string is given, each letter is taken as a variable. See *sage.combinat.words.alphabet.build_alphabet()*.

Warning

Beware that the underlying combinatorial free module is based either on `RootedTrees` or on `LabelledRootedTrees`, with no restriction on the labellings. This means that all code calling the `basis()` method would not give meaningful results, since `basis()` returns many "chaff" elements that do not belong to the algebra.

REFERENCES:

- [ChLi]
- [Liv2006]

class Element

Bases: `IndexedFreeModuleElement`

lift()

Lift element to the Grossman-Larson algebra.

EXAMPLES:

```
sage: F = algebras.FreePreLie(QQ,'abc')
sage: elt = F.an_element().lift(); elt
B[#[a[a[a[a[]]]]]] + B[#[a[a[], a[a[]]]]]
sage: parent(elt)
Grossman-Larson Hopf algebra on 3 generators ['a', 'b', 'c']
over Rational Field
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(QQ,'abc')
>>> elt = F.an_element().lift(); elt
B[#[a[a[a[a[]]]]]] + B[#[a[a[], a[a[]]]]]
```

(continues on next page)

(continued from previous page)

```
>>> parent(elt)
Grossman-Larson Hopf algebra on 3 generators ['a', 'b', 'c']
over Rational Field
```

valuation()

Return the valuation of `self`.

EXAMPLES:

```
sage: a = algebras.FreePreLie(QQ).gen(0)
sage: a.valuation()
1
sage: (a*a).valuation()
2

sage: a, b = algebras.FreePreLie(QQ,'ab').gens()
sage: (a+b).valuation()
1
sage: (a*b).valuation()
2
sage: (a*b+a).valuation()
1
```

```
>>> from sage.all import *
>>> a = algebras.FreePreLie(QQ).gen(Integer(0))
>>> a.valuation()
1
>>> (a*a).valuation()
2

>>> a, b = algebras.FreePreLie(QQ,'ab').gens()
>>> (a+b).valuation()
1
>>> (a*b).valuation()
2
>>> (a*b+a).valuation()
1
```

algebra_generators()

Return the generators of this algebra.

These are the rooted trees with just one vertex.

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh'); A
Free PreLie algebra on 3 generators ['f', 'g', 'h']
 over Integer Ring
sage: list(A.algebra_generators())
[B[f[]], B[g[]], B[h[]]]

sage: A = algebras.FreePreLie(QQ, ['x1','x2'])
sage: list(A.algebra_generators())
[B[x1[]], B[x2[]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(ZZ, 'fgh'); A
Free PreLie algebra on 3 generators ['f', 'g', 'h']
 over Integer Ring
>>> list(A.algebra_generators())
[B[f[]], B[g[]], B[h[]]]

>>> A = algebras.FreePreLie(QQ, ['x1','x2'])
>>> list(A.algebra_generators())
[B[x1[]], B[x2[]]]
```

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, 'xy')
sage: A.an_element()
B[x[x[x[x[]]]]] + B[x[x[], x[x[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, 'xy')
>>> A.an_element()
B[x[x[x[x[]]]]] + B[x[x[], x[x[]]]]
```

bracket_on_basis(x, y)

Return the Lie bracket of two trees.

This is the commutator $[x, y] = x * y - y * x$ of the pre-Lie product.

See also

pre_Lie_product_on_basis()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: y = RT([x])
sage: A.bracket_on_basis(x, y)
-B[[[[], [[]]]]] + B[[[], [[[]]]]] - B[[[[]], [[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> y = RT([x])
>>> A.bracket_on_basis(x, y)
-B[[[[], [[]]]]] + B[[[], [[[]]]]] - B[[[[]], [[]]]]
```

change_ring(R)

Return the free pre-Lie algebra in the same variables over R.

INPUT:

- `R` – a ring

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh')
sage: A.change_ring(QQ)
Free PreLie algebra on 3 generators ['f', 'g', 'h'] over
Rational Field
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(ZZ, 'fgh')
>>> A.change_ring(QQ)
Free PreLie algebra on 3 generators ['f', 'g', 'h'] over
Rational Field
```

construction()

Return a pair `(F, R)`, where `F` is a *PreLieFunctor* and R is a ring, such that `F(R)` returns `self`.

EXAMPLES:

```
sage: P = algebras.FreePreLie(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F, R = P.construction()
sage: F
PreLie[x,y]
sage: R
Integer Ring
sage: F(ZZ) is P
True
sage: F(QQ)
Free PreLie algebra on 2 generators ['x', 'y'] over Rational Field
```

```
>>> from sage.all import *
>>> P = algebras.FreePreLie(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F, R = P.construction()
>>> F
PreLie[x,y]
>>> R
Integer Ring
>>> F(ZZ) is P
True
>>> F(QQ)
Free PreLie algebra on 2 generators ['x', 'y'] over Rational Field
```

corolla(x, y, n, N)

Return the corolla obtained with `x` as root and `y` as leaves.

INPUT:

- `x`, `y` – two elements
- `n` – integer; width of the corolla
- `N` – integer; truncation order (up to order `N` included)

OUTPUT:

the sum over all possible ways to graft `n` copies of `y` on top of `x` (with at most `N` vertices in total)

This operation can be defined by induction starting from the pre-Lie product.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ)
sage: a = A.gen(0)
sage: b = A.corolla(a,a,1,4); b
B[[[]]]
sage: A.corolla(b,b,2,7)
B[[[[[]], [[]]]]] + 2*B[[[[]], [[[]]]]] + B[[[], [[]], [[]]]]

sage: A = algebras.FreePreLie(QQ, 'o')
sage: a = A.gen(0)
sage: b = A.corolla(a,a,1,4)

sage: A = algebras.FreePreLie(QQ,'ab')
sage: a, b = A.gens()
sage: A.corolla(a,b,1,4)
B[a[b[]]]
sage: A.corolla(b,a,3,4)
B[b[a[], a[], a[]]]

sage: A.corolla(a+b,a+b,2,4)
B[a[a[], a[]]] + 2*B[a[a[], b[]]] + B[a[b[], b[]]] + B[b[a[], a[]]] +
2*B[b[a[], b[]]] + B[b[b[], b[]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ)
>>> a = A.gen(Integer(0))
>>> b = A.corolla(a,a,Integer(1),Integer(4)); b
B[[[]]]
>>> A.corolla(b,b,Integer(2),Integer(7))
B[[[[[]], [[]]]]] + 2*B[[[[]], [[[]]]]] + B[[[], [[]], [[]]]]

>>> A = algebras.FreePreLie(QQ, 'o')
>>> a = A.gen(Integer(0))
>>> b = A.corolla(a,a,Integer(1),Integer(4))

>>> A = algebras.FreePreLie(QQ,'ab')
>>> a, b = A.gens()
>>> A.corolla(a,b,Integer(1),Integer(4))
B[a[b[]]]
>>> A.corolla(b,a,Integer(3),Integer(4))
B[b[a[], a[], a[]]]

>>> A.corolla(a+b,a+b,Integer(2),Integer(4))
B[a[a[], a[]]] + 2*B[a[a[], b[]]] + B[a[b[], b[]]] + B[b[a[], a[]]] +
2*B[b[a[], b[]]] + B[b[b[], b[]]]
```

degree_on_basis(*t*)

Return the degree of a rooted tree in the free Pre-Lie algebra.

This is the number of vertices.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: A.degree_on_basis(RT([RT([])]))
2
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> A.degree_on_basis(RT([RT([])]))
2
```

gen(*i*)

Return the `i`-th generator of the algebra.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: F = algebras.FreePreLie(ZZ, 'xyz')
sage: F.gen(0)
B[x[]]

sage: F.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(ZZ, 'xyz')
>>> F.gen(Integer(0))
B[x[]]

>>> F.gen(Integer(4))
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

gens()

Return the generators of `self` (as an algebra).

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh')
sage: A.gens()
(B[f[]], B[g[]], B[h[]])
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(ZZ, 'fgh')
>>> A.gens()
(B[f[]], B[g[]], B[h[]])
```

group_product(*x*, *y*, *n*, *N=10*)

Return the truncated group product of `x` and `y`.

This is a weighted sum of all corollas with up to `n` leaves, with `x` as root and `y` as leaves.

The result is computed up to order `N` (included).

When considered with infinitely many terms and infinite precision, this is an analogue of the Baker-Campbell-Hausdorff formula: it defines an associative product on the completed free pre-Lie algebra.

INPUT:

- `x`, `y` – two elements
- `n` – integer; the maximal width of corollas
- `N` – integer (default: 10); truncation order

EXAMPLES:

In the free pre-Lie algebra with one generator:

```
sage: PL = algebras.FreePreLie(QQ)
sage: a = PL.gen(0)
sage: PL.group_product(a, a, 3, 3)
B[[]] + B[[[]]] + 1/2*B[[[], []]]
```

```
>>> from sage.all import *
>>> PL = algebras.FreePreLie(QQ)
>>> a = PL.gen(Integer(0))
>>> PL.group_product(a, a, Integer(3), Integer(3))
B[[]] + B[[[]]] + 1/2*B[[[], []]]
```

In the free pre-Lie algebra with several generators:

```
sage: PL = algebras.FreePreLie(QQ,'@O')
sage: a, b = PL.gens()
sage: PL.group_product(a, b, 3, 3)
B[@[]] + B[@[O[]]] + 1/2*B[@[O[], O[]]]
sage: PL.group_product(a, b, 3, 10)
B[@[]] + B[@[O[]]] + 1/2*B[@[O[], O[]]] + 1/6*B[@[O[], O[], O[]]]
```

```
>>> from sage.all import *
>>> PL = algebras.FreePreLie(QQ,'@O')
>>> a, b = PL.gens()
>>> PL.group_product(a, b, Integer(3), Integer(3))
B[@[]] + B[@[O[]]] + 1/2*B[@[O[], O[]]]
>>> PL.group_product(a, b, Integer(3), Integer(10))
B[@[]] + B[@[O[]]] + 1/2*B[@[O[], O[]]] + 1/6*B[@[O[], O[], O[]]]
```

nap_product()

Return the NAP product.

See also

nap_product_on_basis()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = A(RT([RT([])]))
sage: A.nap_product(x, x)
B[[[], [[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = A(RT([RT([])]))
>>> A.nap_product(x, x)
B[[[], [[]]]]
```

nap_product_on_basis(*x*, *y*)

Return the NAP product of two trees.

This is the grafting of the root of y over the root of x. The root of the resulting tree is the root of x.

See also

nap_product()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.nap_product_on_basis(x, x)
B[[[], [[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> A.nap_product_on_basis(x, x)
B[[[], [[]]]]
```

pre_Lie_product()

Return the pre-Lie product.

See also

pre_Lie_product_on_basis()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = A(RT([RT([])]))
```

(continues on next page)

(continued from previous page)

```
sage: A.pre_Lie_product(x, x)
B[[[[[]]]]] + B[[[], [[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = A(RT([RT([])]))
>>> A.pre_Lie_product(x, x)
B[[[[[]]]]] + B[[[], [[]]]]
```

pre_Lie_product_on_basis(x, y)

Return the pre-Lie product of two trees.

This is the sum over all graftings of the root of y over a vertex of x. The root of the resulting trees is the root of x.

See also

pre_Lie_product()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.product_on_basis(x, x)
B[[[[[]]]]] + B[[[], [[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> A.product_on_basis(x, x)
B[[[[[]]]]] + B[[[], [[]]]]
```

product_on_basis(x, y)

Return the pre-Lie product of two trees.

This is the sum over all graftings of the root of y over a vertex of x. The root of the resulting trees is the root of x.

See also

pre_Lie_product()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
```

(continues on next page)

(continued from previous page)

```
sage: A.product_on_basis(x, x)
B[[[[[]]]]] + B[[[], [[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> A.product_on_basis(x, x)
B[[[[[]]]]] + B[[[], [[]]]]
```

some_elements()

Return some elements of the free pre-Lie algebra.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: A.some_elements()
[B[[]], B[[[]]], B[[[[[]]]]] + B[[[], [[]]]], B[[[[]]]] + B[[[], []]],␣
↪B[[[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> A.some_elements()
[B[[]], B[[[]]], B[[[[[]]]]] + B[[[], [[]]]], B[[[[]]]] + B[[[], []]],␣
↪B[[[]]]]
```

With several generators:

```
sage: A = algebras.FreePreLie(QQ, 'xy')
sage: A.some_elements()
[B[x[]],
 B[x[x[]]],
 B[x[x[x[x[]]]]] + B[x[x[], x[x[]]]],
 B[x[x[x[]]]] + B[x[x[], x[]]],
 B[x[x[y[]]]] + B[x[x[], y[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, 'xy')
>>> A.some_elements()
[B[x[]],
 B[x[x[]]],
 B[x[x[x[x[]]]]] + B[x[x[], x[x[]]]],
 B[x[x[x[]]]] + B[x[x[], x[]]],
 B[x[x[y[]]]] + B[x[x[], y[]]]]
```

variable_names()

Return the names of the variables.

EXAMPLES:

```
sage: R = algebras.FreePreLie(QQ, 'xy')
sage: R.variable_names()
```

(continues on next page)

(continued from previous page)

```
{'x', 'y'}

sage: R = algebras.FreePreLie(QQ, None)
sage: R.variable_names()
{'o'}
```

```
>>> from sage.all import *
>>> R = algebras.FreePreLie(QQ, 'xy')
>>> R.variable_names()
{'x', 'y'}

>>> R = algebras.FreePreLie(QQ, None)
>>> R.variable_names()
{'o'}
```

class sage.combinat.free_prelie_algebra.**PreLieFunctor**(*vars*)

Bases: ConstructionFunctor

A constructor for pre-Lie algebras.

EXAMPLES:

```
sage: P = algebras.FreePreLie(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F = P.construction()[0]; F
PreLie[x,y]

sage: A = GF(5)['a,b']
sage: a, b = A.gens()
sage: F(A)
Free PreLie algebra on 2 generators ['x', 'y'] over Multivariate Polynomial Ring␣
↪in a, b over Finite Field of size 5

sage: f = A.hom([a+b,a-b],A)
sage: F(f)
Generic endomorphism of Free PreLie algebra on 2 generators ['x', 'y']
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: F(f)(a * F(A)(x))
(a+b)*B[x[]]
```

```
>>> from sage.all import *
>>> P = algebras.FreePreLie(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F = P.construction()[Integer(0)]; F
PreLie[x,y]

>>> A = GF(Integer(5))['a,b']
>>> a, b = A.gens()
>>> F(A)
Free PreLie algebra on 2 generators ['x', 'y'] over Multivariate Polynomial Ring␣
↪in a, b over Finite Field of size 5
```

(continues on next page)

(continued from previous page)

```
>>> f = A.hom([a+b,a-b],A)
>>> F(f)
Generic endomorphism of Free PreLie algebra on 2 generators ['x', 'y']
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> F(f)(a * F(A)(x))
(a+b)*B[x[]]
```

merge(*other*)

Merge `self` with another construction functor, or return None.

EXAMPLES:

```
sage: F = sage.combinat.free_prelie_algebra.PreLieFunctor(['x','y'])
sage: G = sage.combinat.free_prelie_algebra.PreLieFunctor(['t'])
sage: F.merge(G)
PreLie[x,y,t]
sage: F.merge(F)
PreLie[x,y]
```

```
>>> from sage.all import *
>>> F = sage.combinat.free_prelie_algebra.PreLieFunctor(['x','y'])
>>> G = sage.combinat.free_prelie_algebra.PreLieFunctor(['t'])
>>> F.merge(G)
PreLie[x,y,t]
>>> F.merge(F)
PreLie[x,y]
```

Now some actual use cases:

```
sage: R = algebras.FreePreLie(ZZ, 'xyz')
sage: x,y,z = R.gens()
sage: 1/2 * x
1/2*B[x[]]
sage: parent(1/2 * x)
Free PreLie algebra on 3 generators ['x', 'y', 'z'] over Rational Field

sage: S = algebras.FreePreLie(QQ, 'zt')
sage: z,t = S.gens()
sage: x + t
B[t[]] + B[x[]]
sage: parent(x + t)
Free PreLie algebra on 4 generators ['z', 't', 'x', 'y'] over Rational Field
```

```
>>> from sage.all import *
>>> R = algebras.FreePreLie(ZZ, 'xyz')
>>> x,y,z = R.gens()
>>> Integer(1)/Integer(2) * x
1/2*B[x[]]
>>> parent(Integer(1)/Integer(2) * x)
Free PreLie algebra on 3 generators ['x', 'y', 'z'] over Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> S = algebras.FreePreLie(QQ, 'zt')
>>> z,t = S.gens()
>>> x + t
B[t[]] + B[x[]]
>>> parent(x + t)
Free PreLie algebra on 4 generators ['z', 't', 'x', 'y'] over Rational Field
```

rank = 9

sage.combinat.free_prelie_algebra.**corolla_gen**(*tx*, *list_ty*, *labels=True*)

Yield the terms in the corolla with given bottom tree and top trees.

These are the possible terms in the simultaneous grafting of the top trees on vertices of the bottom tree.

INPUT:

- `tx` – a tree
- `list_ty` – list of trees

EXAMPLES:

```
sage: from sage.combinat.free_prelie_algebra import corolla_gen
sage: a = algebras.FreePreLie(QQ).gen(0)
sage: ta = a.support()[0]
sage: list(corolla_gen(ta,[ta],False))
[[[]]]

sage: a, b = algebras.FreePreLie(QQ,'ab').gens()
sage: ta = a.support()[0]
sage: tb = b.support()[0]
sage: ab = (a*b).support()[0]
sage: list(corolla_gen(ta,[tb]))
[a[b[]]]
sage: list(corolla_gen(tb,[ta,ta]))
[b[a[], a[]]]
sage: list(corolla_gen(ab,[ab,ta]))
[a[a[], b[], a[b[]]], a[a[b[]], b[a[]]], a[a[], b[a[b[]]]],
a[b[a[], a[b[]]]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.free_prelie_algebra import corolla_gen
>>> a = algebras.FreePreLie(QQ).gen(Integer(0))
>>> ta = a.support()[Integer(0)]
>>> list(corolla_gen(ta,[ta],False))
[[[]]]

>>> a, b = algebras.FreePreLie(QQ,'ab').gens()
>>> ta = a.support()[Integer(0)]
>>> tb = b.support()[Integer(0)]
>>> ab = (a*b).support()[Integer(0)]
>>> list(corolla_gen(ta,[tb]))
[a[b[]]]
```

(continues on next page)

(continued from previous page)

```
>>> list(corolla_gen(tb,[ta,ta]))
[b[a[], a[]]]
>>> list(corolla_gen(ab,[ab,ta]))
[a[a[], b[], a[b[]]], a[a[b[]], b[a[]]], a[a[], b[a[b[]]]],
a[b[a[], a[b[]]]]]
```

sage.combinat.free_prelie_algebra.**tree_from_sortkey**(*ch*, *labels=True*)

Transform a list of `(valence, label)` into a tree and a remainder.

This is like an inverse of the `sort_key` method.

INPUT:

- `ch` – list of pairs `(integer, label)`
- `labels` – boolean (default: `True`); whether to use labelled trees

OUTPUT:

a pair `(tree, remainder of the input)`

EXAMPLES:

```
sage: from sage.combinat.free_prelie_algebra import tree_from_sortkey
sage: a = algebras.FreePreLie(QQ).gen(0)
sage: t = (a*a*a*a).support()
sage: all(tree_from_sortkey(u.sort_key(), False)[0] == u for u in t)
True

sage: a, b = algebras.FreePreLie(QQ,'ab').gens()
sage: t = (a*b*a*b).support()
sage: all(tree_from_sortkey(u.sort_key())[0] == u for u in t)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.free_prelie_algebra import tree_from_sortkey
>>> a = algebras.FreePreLie(QQ).gen(Integer(0))
>>> t = (a*a*a*a).support()
>>> all(tree_from_sortkey(u.sort_key(), False)[Integer(0)] == u for u in t)
True

>>> a, b = algebras.FreePreLie(QQ,'ab').gens()
>>> t = (a*b*a*b).support()
>>> all(tree_from_sortkey(u.sort_key())[Integer(0)] == u for u in t)
True
```

5.1.114 Fully packed loops

AUTHORS:

- Vincent Knight, James Campbell, Kevin Dilks, Emily Gunawan (2015): Initial version
- Vincent Delecroix (2017): cleaning and enhanced plotting function

class sage.combinat.fully_packed_loop.**FullyPackedLoop**(*parent*, *generator*)

Bases: `Element`

A class for fully packed loops.

A fully packed loop is a collection of non-intersecting lattice paths on a square grid such that every vertex is part of some path, and the paths are either closed internal loops or have endpoints corresponding to alternate points on the boundary [Pro2001]. They are known to be in bijection with alternating sign matrices.

See also

AlternatingSignMatrix

To each fully packed loop, we assign a link pattern, which is the non-crossing matching attained by seeing which points on the boundary are connected by open paths in the fully packed loop.

We can create a fully packed loop using the corresponding alternating sign matrix and also extract the link pattern:

```
sage: A = AlternatingSignMatrix([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
sage: fpl = FullyPackedLoop(A)
sage: fpl.link_pattern()
[(1, 4), (2, 3), (5, 6)]
sage: fpl
    |         |
    |         |
    + -- +    +
         |    |
         |    |
 -- +    +    + --
    |    |
    |    |
    +    + -- +
    |         |
    |         |
sage: B = AlternatingSignMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: fplb = FullyPackedLoop(B)
sage: fplb.link_pattern()
[(1, 6), (2, 5), (3, 4)]
sage: fplb
    |         |
    |         |
    +    + -- +
    |    |
    |    |
 -- +    +    + --
         |    |
         |    |
    + -- +    +
    |         |
    |         |
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(0), Integer(0), Integer(1)], [Integer(0),␣
↪Integer(1), Integer(0)], [Integer(1), Integer(0), Integer(0)]])
>>> fpl = FullyPackedLoop(A)
>>> fpl.link_pattern()
```

(continues on next page)

(continued from previous page)

```
[(1, 4), (2, 3), (5, 6)]
>>> fpl
   |         |
   |         |
   + -- +    +
        |    |
        |    |
-- +    +    + --
   |    |
   |    |
   +    + -- +
   |         |
   |         |
>>> B = AlternatingSignMatrix([[Integer(1), Integer(0), Integer(0)], [Integer(0),␣
↪Integer(1), Integer(0)], [Integer(0), Integer(0), Integer(1)]])
>>> fplb = FullyPackedLoop(B)
>>> fplb.link_pattern()
[(1, 6), (2, 5), (3, 4)]
>>> fplb
   |         |
   |         |
   +    + -- +
   |    |
   |    |
-- +    +    + --
        |    |
        |    |
   + -- +    +
   |         |
   |         |
```

The class also has a plot method:

```
sage: fpl.plot()                                                            #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

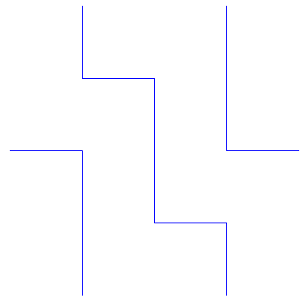
```
>>> from sage.all import *
>>> fpl.plot()                                                            #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

which gives:

Note that we can also create a fully packed loop from a six vertex model configuration:

```
sage: S = SixVertexModel(3, boundary_conditions='ice').from_alternating_sign_
↪matrix(A)
sage: S
    ^    ^    ^
    |    |    |
--> # -> # -> # <--
    ^    ^    |
```

(continues on next page)



(continued from previous page)

```
     |     |     V
--> # -> # <- # <--
     ^     |     |
     |     V     V
--> # <- # <- # <--
     |     |     |
     V     V     V
sage: fpl = FullyPackedLoop(S)
sage: fpl
    │         │
    │         │
    + ── +    +
         │    │
         │    │
── +     +    + ──
    │    │
    │    │
    +    + ── +
    │         │
    │         │
```

```
>>> from sage.all import *
>>> S = SixVertexModel(Integer(3), boundary_conditions='ice').from_alternating_
↪sign_matrix(A)
>>> S
     ^     ^     ^
     |     |     |
--> # -> # -> # <--
     ^     ^     |
     |     |     V
--> # -> # <- # <--
     ^     |     |
     |     V     V
--> # <- # <- # <--
     |     |     |
     V     V     V
>>> fpl = FullyPackedLoop(S)
>>> fpl
    │         │
    │         │
    + ── +    +
```

(continues on next page)

(continued from previous page)

```
         |     |
         |     |
── +     +     + ──
   |     |
   |     |
   +     + ── +
   |          |
   |          |
```

Once we have a fully packed loop we can obtain the corresponding alternating sign matrix:

```
sage: fpl.to_alternating_sign_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> fpl.to_alternating_sign_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
```

Here are some more examples using bigger ASMs:

```
sage: A = AlternatingSignMatrix([[0,1,0,0],[0,0,1,0],[1,-1,0,1],[0,1,0,0]])
sage: S = SixVertexModel(4, boundary_conditions='ice').from_alternating_sign_
↪matrix(A)
sage: fpl = FullyPackedLoop(S)
sage: fpl.link_pattern()
[(1, 2), (3, 6), (4, 5), (7, 8)]
sage: fpl
   |          |
   |          |
   + ── + ── +     + ──
                   |
                   |
── +     + ── + ── +
   |     |
   |     |
   +     +     + ── + ──
   |     |     |
   |     |     |
── +     +     + ── +
         |          |
         |          |

sage: m = AlternatingSignMatrix([[0,0,1,0,0,0],
....:                            [1,0,-1,0,1,0],
....:                            [0,0,0,1,0,0],
....:                            [0,1,0,0,-1,1],
....:                            [0,0,0,0,1,0],
....:                            [0,0,1,0,0,0]])
```

(continues on next page)

(continued from previous page)

```
sage: fpl = FullyPackedLoop(m)
sage: fpl.link_pattern()
[(1, 12), (2, 7), (3, 4), (5, 6), (8, 9), (10, 11)]
sage: fpl
   |         |         |
   |         |         |
   + -- +    +    + -- +    + --
        |    |    |         |
        |    |    |         |
-- + -- +    +    + -- + -- +
             |
             |
   + -- +    + -- + -- +    + --
   |    |              |    |
   |    |              |    |
-- +    +    + -- +    +    +
        |    |    |    |    |
        |    |    |    |    |
   + -- +    + -- +    +    + --
   |                   |
   |                   |
-- +    + -- + -- +    + -- +
        |         |         |
        |         |         |

sage: m = AlternatingSignMatrix([[0,1,0,0,0,0,0],
....:                            [1,-1,0,0,1,0,0],
....:                            [0,0,0,1,0,0,0],
....:                            [0,1,0,0,-1,1,0],
....:                            [0,0,0,0,1,0,0],
....:                            [0,0,1,0,-1,0,1],
....:                            [0,0,0,0,1,0,0]])
sage: fpl = FullyPackedLoop(m)
sage: fpl.link_pattern()
[(1, 2), (3, 4), (5, 6), (7, 8), (9, 14), (10, 11), (12, 13)]
sage: fpl
   |         |         |         |
   |         |         |         |
   + -- + -- +    + -- +    + -- +
                  |         |
                  |         |
-- + -- + -- +    + -- + -- +    + --
             |                   |
             |                   |
   + -- +    + -- + -- +    + -- +
   |    |              |    |
   |    |              |    |
-- +    +    + -- +    +    +    + --
        |    |    |    |    |    |
        |    |    |    |    |    |
   + -- +    + -- +    +    + -- +
   |                   |
```

(continues on next page)

(continued from previous page)

```
   │                   │
── +    + ── + ── +    +    + ── + ──
        │         │    │    │
        │         │    │    │
   + ── +    + ── +    +    + ── +
   │         │         │         │
   │         │         │         │
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(0),Integer(1)],[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> S = SixVertexModel(Integer(4), boundary_conditions='ice').from_alternating_
↪sign_matrix(A)
>>> fpl = FullyPackedLoop(S)
>>> fpl.link_pattern()
[(1, 2), (3, 6), (4, 5), (7, 8)]
>>> fpl
   │         │
   │         │
   + ── + ── +    + ──
                  │
                  │
── +    + ── + ── +
   │    │
   │    │
   +    +    + ── + ──
   │    │    │
   │    │    │
── +    +    + ── +
        │         │
        │         │

>>> m = AlternatingSignMatrix([[Integer(0),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0)],
...                             [Integer(1),Integer(0),-Integer(1),Integer(0),
↪Integer(1),Integer(0)],
...                             [Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(0)],
...                             [Integer(0),Integer(1),Integer(0),Integer(0),-
↪Integer(1),Integer(1)],
...                             [Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1),Integer(0)],
...                             [Integer(0),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0)]])
>>> fpl = FullyPackedLoop(m)
>>> fpl.link_pattern()
[(1, 12), (2, 7), (3, 4), (5, 6), (8, 9), (10, 11)]
>>> fpl
   │         │         │
   │         │         │
   + ── +    +    + ── +    + ──
```

(continues on next page)

(continued from previous page)

```
        │    │    │         │
        │    │    │         │
── + ── +    +    + ── + ── +
             │
             │
   + ── +    + ── + ── +    + ──
   │    │              │    │
   │    │              │    │
── +    +    + ── +    +    +
        │    │    │    │    │
        │    │    │    │    │
   + ── +    + ── +    +    + ──
   │                   │
   │                   │
── +    + ── + ── +    + ── +
        │         │         │
        │         │         │

>>> m = AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0)],
...                             [Integer(1),-Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(0),Integer(0)],
...                             [Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)],
...                             [Integer(0),Integer(1),Integer(0),Integer(0),-
↪Integer(1),Integer(1),Integer(0)],
...                             [Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1),Integer(0),Integer(0)],
...                             [Integer(0),Integer(0),Integer(1),Integer(0),-
↪Integer(1),Integer(0),Integer(1)],
...                             [Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1),Integer(0),Integer(0)]])
>>> fpl = FullyPackedLoop(m)
>>> fpl.link_pattern()
[(1, 2), (3, 4), (5, 6), (7, 8), (9, 14), (10, 11), (12, 13)]
>>> fpl
   │         │         │         │
   │         │         │         │
   + ── + ── +    + ── +    + ── +
                  │         │
                  │         │
── + ── + ── +    + ── + ── +    + ──
             │                   │
             │                   │
   + ── +    + ── + ── +    + ── +
   │    │              │    │
   │    │              │    │
── +    +    + ── +    +    +    + ──
        │    │    │    │    │    │
        │    │    │    │    │    │
   + ── +    + ── +    +    + ── +
   │                   │
```

(continues on next page)

(continued from previous page)

```
   |                   |
── +    + ── + ── +    +    + ── + ──
        |         |    |    |
        |         |    |    |
   + ── +    + ── +    +    + ── +
   |         |         |         |
   |         |         |         |
```

Gyration on an alternating sign matrix/fully packed loop `fpl` of the link pattern corresponding to `fpl`:

```
sage: ASMs = AlternatingSignMatrices(3).list()
sage: ncp = FullyPackedLoop(ASMs[1]).link_pattern() # fpl's gyration orbit size␣
↪is 2
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(5):
....:         a,b=a%6+1,b%6+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(ASMs[1].gyration().to_fully_packed_loop().link_pattern())␣
↪==\
....:     PerfectMatching(rotated_ncp)
True

sage: fpl = FullyPackedLoop(ASMs[0])
sage: ncp = fpl.link_pattern() # fpl's gyration size is 3
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(5):
....:         a,b=a%6+1,b%6+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(ASMs[0].gyration().to_fully_packed_loop().link_pattern())␣
↪==\
....:     PerfectMatching(rotated_ncp)
True

sage: mat = AlternatingSignMatrix([[0,0,1,0,0,0,0],[1,0,-1,0,1,0,0],
....:     [0,0,1,0,0,0,0],[0,1,-1,0,0,1,0],[0,0,1,0,0,0,0],[0,0,0,1,0,0,0],[0,0,0,
↪0,0,0,1]])
sage: fpl = FullyPackedLoop(mat) # n=7
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(13):
....:         a,b=a%14+1,b%14+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==\
....:     PerfectMatching(rotated_ncp)
True

sage: mat = AlternatingSignMatrix([[0,0,0,1,0,0], [0,0,1,-1,1,0],
....:     [0,1,0,0,-1,1], [1,0,-1,1,0,0], [0,0,1,0,0,0], [0,0,0,0,1,0]])
sage: fpl = FullyPackedLoop(mat) # n =6
```

(continues on next page)

(continued from previous page)

```
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(11):
....:         a,b=a%12+1,b%12+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==\
....:     PerfectMatching(rotated_ncp)
True
```

```
>>> from sage.all import *
>>> ASMs = AlternatingSignMatrices(Integer(3)).list()
>>> ncp = FullyPackedLoop(ASMs[Integer(1)]).link_pattern() # fpl's gyration orbit␣
↪size is 2
>>> rotated_ncp=[]
>>> for (a,b) in ncp:
...     for i in range(Integer(5)):
...         a,b=a%Integer(6)+Integer(1),b%Integer(6)+Integer(1);
...     rotated_ncp.append((a,b))
>>> PerfectMatching(ASMs[Integer(1)].gyration().to_fully_packed_loop().link_
↪pattern()) ==    PerfectMatching(rotated_ncp)
True

>>> fpl = FullyPackedLoop(ASMs[Integer(0)])
>>> ncp = fpl.link_pattern() # fpl's gyration size is 3
>>> rotated_ncp=[]
>>> for (a,b) in ncp:
...     for i in range(Integer(5)):
...         a,b=a%Integer(6)+Integer(1),b%Integer(6)+Integer(1);
...     rotated_ncp.append((a,b))
>>> PerfectMatching(ASMs[Integer(0)].gyration().to_fully_packed_loop().link_
↪pattern()) ==    PerfectMatching(rotated_ncp)
True

>>> mat = AlternatingSignMatrix([[Integer(0),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0),Integer(0)],[Integer(1),Integer(0),-Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0)],
...     [Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),-Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1)]])
>>> fpl = FullyPackedLoop(mat) # n=7
>>> ncp = fpl.link_pattern()
>>> rotated_ncp=[]
>>> for (a,b) in ncp:
...     for i in range(Integer(13)):
...         a,b=a%Integer(14)+Integer(1),b%Integer(14)+Integer(1);
...     rotated_ncp.append((a,b))
>>> PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==   ␣
↪PerfectMatching(rotated_ncp)
```

(continues on next page)

(continued from previous page)

```
True

>>> mat = AlternatingSignMatrix([[Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(0)], [Integer(0),Integer(0),Integer(1),-Integer(1),
↪Integer(1),Integer(0)],
...     [Integer(0),Integer(1),Integer(0),Integer(0),-Integer(1),Integer(1)],␣
↪[Integer(1),Integer(0),-Integer(1),Integer(1),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(0),Integer(0),Integer(0),Integer(1),Integer(0)]])
>>> fpl = FullyPackedLoop(mat) # n =6
>>> ncp = fpl.link_pattern()
>>> rotated_ncp=[]
>>> for (a,b) in ncp:
...     for i in range(Integer(11)):
...         a,b=a%Integer(12)+Integer(1),b%Integer(12)+Integer(1);
...     rotated_ncp.append((a,b))
>>> PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==   ␣
↪PerfectMatching(rotated_ncp)
True
```

More examples:

We can initiate a fully packed loop using an alternating sign matrix:

```
sage: A = AlternatingSignMatrix([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
sage: fpl = FullyPackedLoop(A)
sage: fpl
    |         |
    |         |
    + -- +    +
         |    |
         |    |
 -- +    +    + --
    |    |
    |    |
    +    + -- +
    |         |
    |         |
sage: FullyPackedLoops(3)(A) == fpl
True
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(0), Integer(0), Integer(1)], [Integer(0),␣
↪Integer(1), Integer(0)], [Integer(1), Integer(0), Integer(0)]])
>>> fpl = FullyPackedLoop(A)
>>> fpl
    |         |
    |         |
    + -- +    +
         |    |
         |    |
 -- +    +    + --
```

(continues on next page)

(continued from previous page)

```
    |     |
    |     |
    +     + ── +
    |          |
    |          |
>>> FullyPackedLoops(Integer(3))(A) == fpl
True
```

We can also input a matrix:

```
sage: FullyPackedLoop([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
    |          |
    |          |
    + ── +     +
         |     |
         |     |
 ── +    +     + ──
    |    |
    |    |
    +    + ── +
    |          |
    |          |
sage: FullyPackedLoop([[0, 0, 1], [0, 1, 0], [1, 0, 0]]) ==\
....: FullyPackedLoops(3)([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
True
```

```
>>> from sage.all import *
>>> FullyPackedLoop([[Integer(0), Integer(0), Integer(1)], [Integer(0),
↪Integer(1), Integer(0)], [Integer(1), Integer(0), Integer(0)]])
    |          |
    |          |
    + ── +     +
         |     |
         |     |
 ── +    +     + ──
    |    |
    |    |
    +    + ── +
    |          |
    |          |
>>> FullyPackedLoop([[Integer(0), Integer(0), Integer(1)], [Integer(0),
↪Integer(1), Integer(0)], [Integer(1), Integer(0), Integer(0)]])
↪==FullyPackedLoops(Integer(3))([[Integer(0), Integer(0), Integer(1)],
↪[Integer(0), Integer(1), Integer(0)], [Integer(1), Integer(0), Integer(0)]])
True
```

Otherwise we initiate a fully packed loop using a six vertex model:

```
sage: S = SixVertexModel(3, boundary_conditions='ice').from_alternating_sign_
↪matrix(A)
sage: fpl = FullyPackedLoop(S)
sage: fpl
```

(continues on next page)

(continued from previous page)

```
    |         |
    |         |
    + -- +    +
         |    |
         |    |
 -- +    +    + --
    |    |
    |    |
    +    + -- +
    |         |
    |         |

sage: FullyPackedLoops(3)(S) == FullyPackedLoop(S)
True

sage: fpl.six_vertex_model().to_alternating_sign_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> S = SixVertexModel(Integer(3), boundary_conditions='ice').from_alternating_
↪sign_matrix(A)
>>> fpl = FullyPackedLoop(S)
>>> fpl
    |         |
    |         |
    + -- +    +
         |    |
         |    |
 -- +    +    + --
    |    |
    |    |
    +    + -- +
    |         |
    |         |

>>> FullyPackedLoops(Integer(3))(S) == FullyPackedLoop(S)
True

>>> fpl.six_vertex_model().to_alternating_sign_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
```

We can also input the matrix associated to a six vertex model:

```
sage: SixVertexModel(2)([[3,1],[5,3]])
    ^    ^
    |    |
--> # <- # <--
    |    ^
```

(continues on next page)

(continued from previous page)

```
    V     |
--> # -> # <--
    |     |
    V     V

sage: FullyPackedLoop([[3,1],[5,3]])
    |
    |
    +     + ——
    |     |
    |     |
 —— +     +
          |
          |

sage: FullyPackedLoops(2)([[3,1],[5,3]]) == FullyPackedLoop([[3,1],[5,3]])
True
```

```
>>> from sage.all import *
>>> SixVertexModel(Integer(2))([[Integer(3),Integer(1)],[Integer(5),Integer(3)]])
    ^     ^
    |     |
--> # <- # <--
    |     ^
    V     |
--> # -> # <--
    |     |
    V     V

>>> FullyPackedLoop([[Integer(3),Integer(1)],[Integer(5),Integer(3)]])
    |
    |
    +     + ——
    |     |
    |     |
 —— +     +
          |
          |

>>> FullyPackedLoops(Integer(2))([[Integer(3),Integer(1)],[Integer(5),
↪Integer(3)]]) == FullyPackedLoop([[Integer(3),Integer(1)],[Integer(5),
↪Integer(3)]])
True
```

Note that the matrix corresponding to a six vertex model without the ice boundary condition is not allowed:

```
sage: SixVertexModel(2)([[3,1],[5,5]])
    ^     ^
    |     |
--> # <- # <--
    |     ^
    V     V
```

(continues on next page)

(continued from previous page)

```
--> # -> # -->
    |     |
    V     V

sage: FullyPackedLoop([[3,1],[5,5]])
Traceback (most recent call last):
...
ValueError: invalid alternating sign matrix

sage: FullyPackedLoops(2)([[3,1],[5,5]])
Traceback (most recent call last):
...
ValueError: invalid alternating sign matrix
```

```
>>> from sage.all import *
>>> SixVertexModel(Integer(2))([[Integer(3),Integer(1)],[Integer(5),Integer(5)]])
    ^     ^
    |     |
--> # <- # <--
    |     ^
    V     V
--> # -> # -->
    |     |
    V     V

>>> FullyPackedLoop([[Integer(3),Integer(1)],[Integer(5),Integer(5)]])
Traceback (most recent call last):
...
ValueError: invalid alternating sign matrix

>>> FullyPackedLoops(Integer(2))([[Integer(3),Integer(1)],[Integer(5),
↪Integer(5)]])
Traceback (most recent call last):
...
ValueError: invalid alternating sign matrix
```

Note that if anything else is used to generate the fully packed loop an error will occur:

```
sage: fpl = FullyPackedLoop(5)
Traceback (most recent call last):
...
ValueError: invalid alternating sign matrix

sage: fpl = FullyPackedLoop((1, 2, 3))
Traceback (most recent call last):
...
ValueError: the alternating sign matrices must be square

sage: SVM = SixVertexModel(3)[0]
sage: FullyPackedLoop(SVM)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: invalid alternating sign matrix
```

```
>>> from sage.all import *
>>> fpl = FullyPackedLoop(Integer(5))
Traceback (most recent call last):
...
ValueError: invalid alternating sign matrix

>>> fpl = FullyPackedLoop((Integer(1), Integer(2), Integer(3)))
Traceback (most recent call last):
...
ValueError: the alternating sign matrices must be square

>>> SVM = SixVertexModel(Integer(3))[Integer(0)]
>>> FullyPackedLoop(SVM)
Traceback (most recent call last):
...
ValueError: invalid alternating sign matrix
```

REFERENCES:

- [Pro2001]
- [Str2015]

gyration()

Return the fully packed loop obtained by applying gyration to the alternating sign matrix in bijection with `self`.

Gyration was first defined in [Wie2000] as an action on fully-packed loops.

EXAMPLES:

```
sage: A = AlternatingSignMatrix([[1, 0, 0],[0, 1, 0],[0, 0, 1]])
sage: fpl = FullyPackedLoop(A)
sage: fpl.gyration().to_alternating_sign_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm = AlternatingSignMatrix([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: f = FullyPackedLoop(asm)
sage: f.gyration().to_alternating_sign_matrix()
[0 1 0]
[0 0 1]
[1 0 0]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(1), Integer(0), Integer(0)],
↪[Integer(0), Integer(1), Integer(0)],[Integer(0), Integer(0), Integer(1)]])
>>> fpl = FullyPackedLoop(A)
>>> fpl.gyration().to_alternating_sign_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
```

(continues on next page)

(continued from previous page)

```
>>> asm = AlternatingSignMatrix([[Integer(0), Integer(0), Integer(1)],
↪[Integer(1), Integer(0), Integer(0)],[Integer(0), Integer(1), Integer(0)]])
>>> f = FullyPackedLoop(asm)
>>> f.gyration().to_alternating_sign_matrix()
[0 1 0]
[0 0 1]
[1 0 0]
```

link_pattern()

Return a link pattern corresponding to a fully packed loop.

Here we define a link pattern LP to be a partition of the list $[1, ..., 2k]$ into 2-element sets (such a partition is also known as a perfect matching) such that the following non-crossing condition holds: Let the numbers $1, ..., 2k$ be written on the perimeter of a circle. For every 2-element set (a, b) of the partition LP, draw an arc linking the two numbers a and b. We say that LP is non-crossing if every arc can be drawn so that no two arcs intersect.

Since every endpoint of a fully packed loop fpl is connected to a different endpoint, there is a natural surjection from the fully packed loops on an nxn grid onto the link patterns on the list $[1, \dots, 2n]$. The pairs of connected endpoints of a fully packed loop fpl correspond to the 2-element tuples of the corresponding link pattern.

See also

PerfectMatching

Note

by convention, we choose the top left vertex to be even. See [Pro2001] and [Str2015].

EXAMPLES:

We can extract the underlying link pattern (a non-crossing partition) from a fully packed loop:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: fpl = FullyPackedLoop(A)
sage: fpl.link_pattern()
[(1, 2), (3, 6), (4, 5)]

sage: B = AlternatingSignMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: fpl = FullyPackedLoop(B)
sage: fpl.link_pattern()
[(1, 6), (2, 5), (3, 4)]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(0), Integer(1), Integer(0)],␣
↪[Integer(1), -Integer(1), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(0)]])
>>> fpl = FullyPackedLoop(A)
>>> fpl.link_pattern()
[(1, 2), (3, 6), (4, 5)]
```

(continues on next page)

(continued from previous page)

```
>>> B = AlternatingSignMatrix([[Integer(1), Integer(0), Integer(0)],␣
↪[Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0), Integer(1)]])
>>> fpl = FullyPackedLoop(B)
>>> fpl.link_pattern()
[(1, 6), (2, 5), (3, 4)]
```

Gyration on an alternating sign matrix/fully packed loop `fpl` corresponds to a rotation (i.e. a becomes a-1 mod 2n) of the link pattern corresponding to `fpl`:

```
sage: ASMs = AlternatingSignMatrices(3).list()
sage: ncp = FullyPackedLoop(ASMs[1]).link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(5):
....:         a,b=a%6+1,b%6+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(ASMs[1].gyration().to_fully_packed_loop().link_
↪pattern()) ==\
....:     PerfectMatching(rotated_ncp)
True

sage: fpl = FullyPackedLoop(ASMs[0])
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(5):
....:         a,b=a%6+1,b%6+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(ASMs[0].gyration().to_fully_packed_loop().link_
↪pattern()) ==\
....:     PerfectMatching(rotated_ncp)
True

sage: mat = AlternatingSignMatrix([[0,0,1,0,0,0,0],[1,0,-1,0,1,0,0],[0,0,1,0,
↪0,0,0],
....:     [0,1,-1,0,0,1,0],[0,0,1,0,0,0,0],[0,0,0,1,0,0,0],[0,0,0,0,0,0,1]])
sage: fpl = FullyPackedLoop(mat) # n=7
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(13):
....:         a,b=a%14+1,b%14+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern())␣
↪==\
....:     PerfectMatching(rotated_ncp)
True

sage: mat = AlternatingSignMatrix([[0,0,0,1,0,0], [0,0,1,-1,1,0], [0,1,0,0,-1,
↪1], [1,0,-1,1,0,0],
....:     [0,0,1,0,0,0], [0,0,0,0,1,0]])
sage: fpl = FullyPackedLoop(mat)
```

(continues on next page)

(continued from previous page)

```
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(11):
....:         a,b=a%12+1,b%12+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern())␣
↪==\
....:     PerfectMatching(rotated_ncp)
True
```

```
>>> from sage.all import *
>>> ASMs = AlternatingSignMatrices(Integer(3)).list()
>>> ncp = FullyPackedLoop(ASMs[Integer(1)]).link_pattern()
>>> rotated_ncp=[]
>>> for (a,b) in ncp:
...     for i in range(Integer(5)):
...         a,b=a%Integer(6)+Integer(1),b%Integer(6)+Integer(1);
...     rotated_ncp.append((a,b))
>>> PerfectMatching(ASMs[Integer(1)].gyration().to_fully_packed_loop().link_
↪pattern()) ==    PerfectMatching(rotated_ncp)
True

>>> fpl = FullyPackedLoop(ASMs[Integer(0)])
>>> ncp = fpl.link_pattern()
>>> rotated_ncp=[]
>>> for (a,b) in ncp:
...     for i in range(Integer(5)):
...         a,b=a%Integer(6)+Integer(1),b%Integer(6)+Integer(1);
...     rotated_ncp.append((a,b))
>>> PerfectMatching(ASMs[Integer(0)].gyration().to_fully_packed_loop().link_
↪pattern()) ==    PerfectMatching(rotated_ncp)
True

>>> mat = AlternatingSignMatrix([[Integer(0),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0),Integer(0)],[Integer(1),Integer(0),-Integer(1),
↪Integer(0),Integer(1),Integer(0),Integer(0)],[Integer(0),Integer(0),
↪Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)],
...     [Integer(0),Integer(1),-Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(1)]])
>>> fpl = FullyPackedLoop(mat) # n=7
>>> ncp = fpl.link_pattern()
>>> rotated_ncp=[]
>>> for (a,b) in ncp:
...     for i in range(Integer(13)):
...         a,b=a%Integer(14)+Integer(1),b%Integer(14)+Integer(1);
...     rotated_ncp.append((a,b))
>>> PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) == ␣
↪  PerfectMatching(rotated_ncp)
```

(continues on next page)

(continued from previous page)

```
True

>>> mat = AlternatingSignMatrix([[Integer(0),Integer(0),Integer(0),Integer(1),
→Integer(0),Integer(0)], [Integer(0),Integer(0),Integer(1),-Integer(1),
→Integer(1),Integer(0)], [Integer(0),Integer(1),Integer(0),Integer(0),-
→Integer(1),Integer(1)], [Integer(1),Integer(0),-Integer(1),Integer(1),
→Integer(0),Integer(0)],
...       [Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),Integer(0)],
→[Integer(0),Integer(0),Integer(0),Integer(0),Integer(1),Integer(0)]])
>>> fpl = FullyPackedLoop(mat)
>>> ncp = fpl.link_pattern()
>>> rotated_ncp=[]
>>> for (a,b) in ncp:
...     for i in range(Integer(11)):
...         a,b=a%Integer(12)+Integer(1),b%Integer(12)+Integer(1);
...     rotated_ncp.append((a,b))
>>> PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==
→ PerfectMatching(rotated_ncp)
True
```

plot(*link=True*, *loop=True*, *loop_fill=False*, ***options*)

Return a graphical object of the Fully Packed Loop.

Each option can be specified separately for links (the curves that join boundary points) and the loops. In order to do so, you need to prefix its name with either `'link_'` or `'loop_'`. As an example, setting `color='red'` will color both links and loops in red while setting `link_color='red'` will only apply the color option for the links.

INPUT:

- `link`, `loop` – boolean (default: `True`); whether to plot the links or the loops
- `color`, `link_color`, `loop_color` – (optional) string or RGB triple
- `colors`, `link_colors`, `loop_colors` – (optional) list of colors
- `color_map`, `link_color_map`, `loop_color_map` – (string, optional) a name of a matplotlib color map for the link or the loop
- `link_color_randomize` – boolean (default: `False`); when `link_colors` or `link_color_map` is specified it randomizes its order. Setting this option to `True` makes it unlikely to have two neighboring links with the same color.
- `loop_fill` – (boolean, optional) whether to fill the interior of the loops

EXAMPLES:

To plot the fully packed loop associated to the following alternating sign matrix

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

simply do:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: fpl = FullyPackedLoop(A)
```

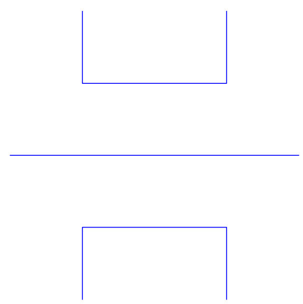
(continues on next page)

(continued from previous page)

```
sage: fpl.plot()                                                  #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(0), Integer(1), Integer(0)],␣
↪[Integer(1), -Integer(1), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(0)]])
>>> fpl = FullyPackedLoop(A)
>>> fpl.plot()                                                   #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

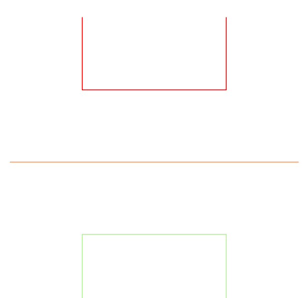
The resulting graphics is as follows



You can also have the three links in different colors with:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: fpl = FullyPackedLoop(A)
sage: fpl.plot(link_color_map='rainbow')                          #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

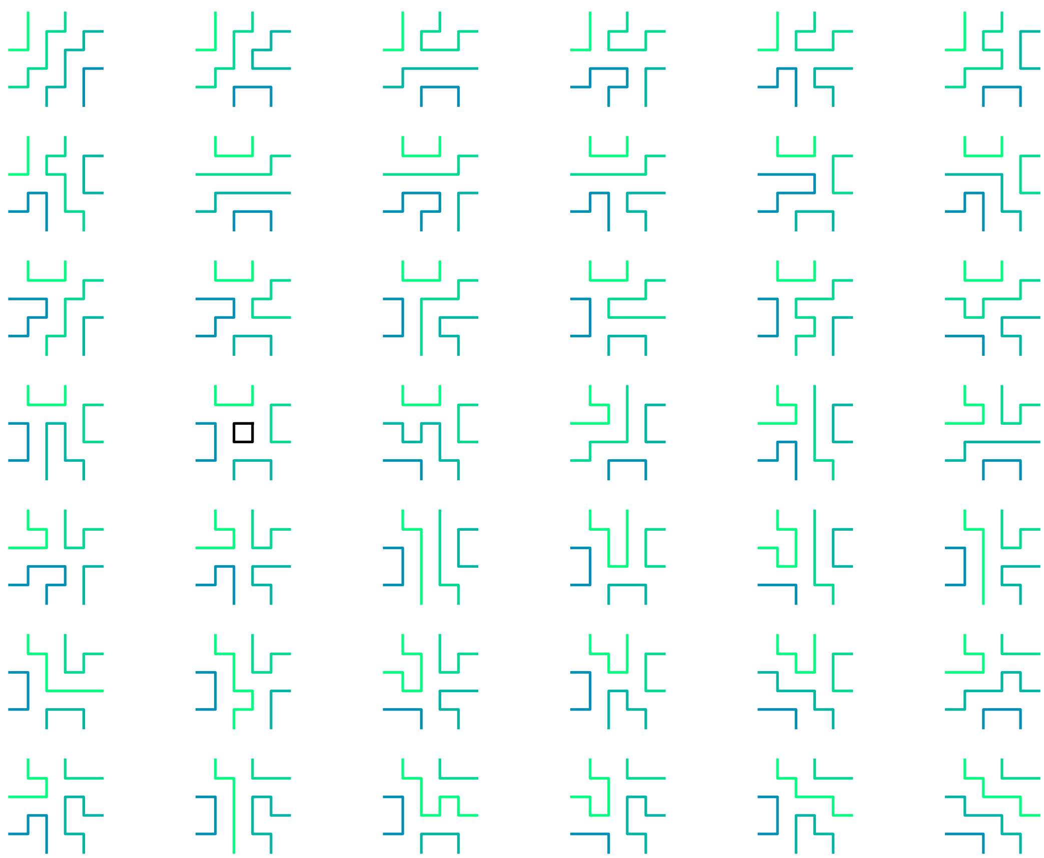
```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(0), Integer(1), Integer(0)],␣
↪[Integer(1), -Integer(1), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(0)]])
>>> fpl = FullyPackedLoop(A)
>>> fpl.plot(link_color_map='rainbow')                           #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```



You can plot the 42 fully packed loops of size 4×4 using:

```
sage: G = [fpl.plot(link_color_map='winter', loop_color='black')            #␣
↪needs sage.plot
....:      for fpl in FullyPackedLoops(4)]
sage: graphics_array(G, 7, 6)                                               #␣
↪needs sage.plot
Graphics Array of size 7 x 6
```

```
>>> from sage.all import *
>>> G = [fpl.plot(link_color_map='winter', loop_color='black')             #␣
↪needs sage.plot
...      for fpl in FullyPackedLoops(Integer(4))]
>>> graphics_array(G, Integer(7), Integer(6))                               ␣
↪              # needs sage.plot
Graphics Array of size 7 x 6
```



Here is an example of a 20×20 fully packed loop:

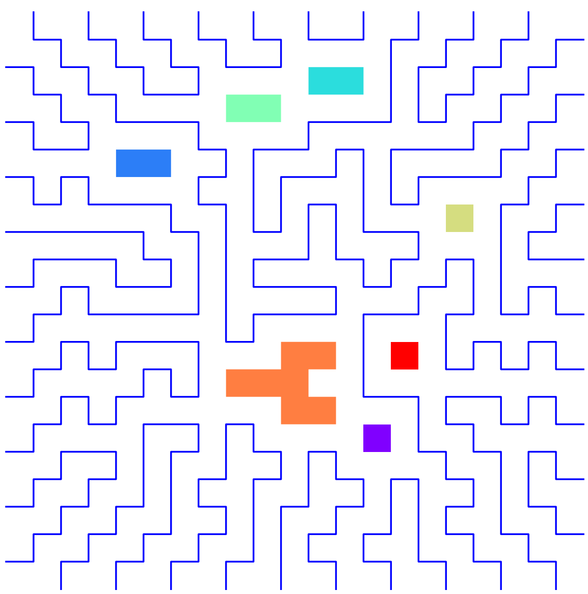
```
sage: s = "00000000000+0000000000000000+00-0+00000000000+00-00+0-+00000\
....: 0000+-00+00-+00000000+00-0000+0000-+00000000+000-0+0-+0-+000\
....: 000+-000+-00+0000000+-+-000+00-+0-000+000+-000+-0+000000-0+\
....: 0000+0-+0-+00000-+00000+-+0-0+-00+0000000000+-0000+0-00+0000\
....: 000000+0-000+000000000000000+0000-00+00000000+0000-000+00000\
```

(continues on next page)

(continued from previous page)

```
....: 00+0-00+000000000000000+-0000+000000-+000000+00-0000+-00+00\
....: 00000000+-0000+00000000000000+0000000000"
sage: a = matrix(20, [{'0':0, '+':1, '-': -1}[i] for i in s])
sage: fpl = FullyPackedLoop(a)
sage: fpl.plot(loop_fill=True, loop_color_map='rainbow')                    #␣
↪needs sage.plot
Graphics object consisting of 27 graphics primitives
```

```
>>> from sage.all import *
>>> s = "00000000000+0000000000000000+00-0+00000000000+00-00+0-+00000\
... 0000+-00+00-+00000000+00-0000+0000-+00000000+000-0+0-+0-+000\
... 000+-000+-00+0000000+-+-000+00-+0-000+000+-000+-0+0000000-0+\
... 0000+0-+0-+00000-+00000+-+0-0+-00+0000000000+-0000+0-00+0000\
... 000000+0-000+000000000000000+0000-00+00000000+0000-000+00000\
... 00+0-00+000000000000000+-0000+000000-+000000+00-0000+-00+00\
... 00000000+-0000+00000000000000+0000000000"
>>> a = matrix(Integer(20), [{'0':Integer(0), '+':Integer(1), '-': -
↪Integer(1)}[i] for i in s])
>>> fpl = FullyPackedLoop(a)
>>> fpl.plot(loop_fill=True, loop_color_map='rainbow')                    #␣
↪needs sage.plot
Graphics object consisting of 27 graphics primitives
```



six_vertex_model()

Return the underlying six vertex model configuration.

EXAMPLES:

```
sage: B = AlternatingSignMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: fpl = FullyPackedLoop(B)
sage: fpl
    |         |
```

(continues on next page)

(continued from previous page)

```
    |           |
    +     + ─── +
    |     |
    |     |
─── +     +     + ───
          |     |
          |     |
    + ─── +     +
    |           |
    |           |
sage: fpl.six_vertex_model()
    ^     ^     ^
    |     |     |
--> # <- # <- # <--
    |     ^     ^
    V     |     |
--> # -> # <- # <--
    |     |     ^
    V     V     |
--> # -> # -> # <--
    |     |     |
    V     V     V
```

```
>>> from sage.all import *
>>> B = AlternatingSignMatrix([[Integer(1), Integer(0), Integer(0)],
↪[Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0), Integer(1)]])
>>> fpl = FullyPackedLoop(B)
>>> fpl
    |           |
    |           |
    +     + ─── +
    |     |
    |     |
─── +     +     + ───
          |     |
          |     |
    + ─── +     +
    |           |
    |           |
>>> fpl.six_vertex_model()
    ^     ^     ^
    |     |     |
--> # <- # <- # <--
    |     ^     ^
    V     |     |
--> # -> # <- # <--
    |     |     ^
    V     V     |
--> # -> # -> # <--
    |     |     |
    V     V     V
```

to_alternating_sign_matrix()

Return the alternating sign matrix corresponding to this class.

> **See also**
>
> *AlternatingSignMatrix*

EXAMPLES:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: S = SixVertexModel(3, boundary_conditions='ice').from_alternating_sign_
↪matrix(A)
sage: fpl = FullyPackedLoop(S)
sage: fpl.to_alternating_sign_matrix()
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
sage: A = AlternatingSignMatrix([[0,1,0,0],[0,0,1,0],[1,-1,0,1],[0,1,0,0]])
sage: S = SixVertexModel(4, boundary_conditions='ice').from_alternating_sign_
↪matrix(A)
sage: fpl = FullyPackedLoop(S)
sage: fpl.to_alternating_sign_matrix()
[ 0  1  0  0]
[ 0  0  1  0]
[ 1 -1  0  1]
[ 0  1  0  0]
```

```
>>> from sage.all import *
>>> A = AlternatingSignMatrix([[Integer(0), Integer(1), Integer(0)],␣
↪[Integer(1), -Integer(1), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(0)]])
>>> S = SixVertexModel(Integer(3), boundary_conditions='ice').from_
↪alternating_sign_matrix(A)
>>> fpl = FullyPackedLoop(S)
>>> fpl.to_alternating_sign_matrix()
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
>>> A = AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(1),-Integer(1),
↪Integer(0),Integer(1)],[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> S = SixVertexModel(Integer(4), boundary_conditions='ice').from_
↪alternating_sign_matrix(A)
>>> fpl = FullyPackedLoop(S)
>>> fpl.to_alternating_sign_matrix()
[ 0  1  0  0]
[ 0  0  1  0]
[ 1 -1  0  1]
[ 0  1  0  0]
```

class sage.combinat.fully_packed_loop.**FullyPackedLoops**(*n*)

Bases: Parent, UniqueRepresentation

Class of all fully packed loops on an $n \times n$ grid.

They are known to be in bijection with alternating sign matrices.

See also

AlternatingSignMatrices

INPUT:

- `n` – the number of row (and column) or grid

EXAMPLES:

This will create an instance to manipulate the fully packed loops of size 3:

```
sage: FPLs = FullyPackedLoops(3)
sage: FPLs
Fully packed loops on a 3x3 grid
sage: FPLs.cardinality()
7
```

```
>>> from sage.all import *
>>> FPLs = FullyPackedLoops(Integer(3))
>>> FPLs
Fully packed loops on a 3x3 grid
>>> FPLs.cardinality()
7
```

When using the square ice model, it is known that the number of configurations is equal to the number of alternating sign matrices:

```
sage: M = FullyPackedLoops(1)
sage: len(M)
1
sage: M = FullyPackedLoops(4)
sage: len(M)
42
sage: all(len(SixVertexModel(n, boundary_conditions='ice'))
....:     == FullyPackedLoops(n).cardinality() for n in range(1, 7))
True
```

```
>>> from sage.all import *
>>> M = FullyPackedLoops(Integer(1))
>>> len(M)
1
>>> M = FullyPackedLoops(Integer(4))
>>> len(M)
42
>>> all(len(SixVertexModel(n, boundary_conditions='ice'))
...     == FullyPackedLoops(n).cardinality() for n in range(Integer(1),
↪Integer(7)))
True
```

Element

alias of *FullyPackedLoop*

cardinality()

Return the cardinality of `self`.

The number of fully packed loops on $n \times n$ grid

$$\prod_{k=0}^{n-1} \frac{(3k+1)!}{(n+k)!} = \frac{1!4!7!10! \cdots (3n-2)!}{n!(n+1)!(n+2)!(n+3)! \cdots (2n-1)!}.$$

EXAMPLES:

```
sage: [AlternatingSignMatrices(n).cardinality() for n in range(10)]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460]
```

```
>>> from sage.all import *
>>> [AlternatingSignMatrices(n).cardinality() for n in range(Integer(10))]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460]
```

size()

Return the size of the matrices in `self`.

5.1.115 Gelfand-Tsetlin Patterns

AUTHORS:

- Travis Scrimshaw (2013-15-03): Initial version

REFERENCES:

class sage.combinat.gelfand_tsetlin_patterns.**GelfandTsetlinPattern**

Bases: `ClonableArray`

A Gelfand-Tsetlin (sometimes written as Gelfand-Zetlin or Gelfand-Cetlin) pattern. They were originally defined in [GC50].

A Gelfand-Tsetlin pattern is a triangular array:

$$\begin{array}{ccccccccc} a_{1,1} & & a_{1,2} & & a_{1,3} & & \cdots & & a_{1,n} \\ & a_{2,2} & & a_{2,3} & & \cdots & & a_{2,n} & \\ & & a_{3,3} & & \cdots & & a_{3,n} & & \\ & & & \ddots & & & & & \\ & & & & a_{n,n} & & & & \end{array}$$

such that $a_{i,j} \geq a_{i+1,j+1} \geq a_{i,j+1}$.

Gelfand-Tsetlin patterns are in bijection with semistandard Young tableaux by the following algorithm. Let G be a Gelfand-Tsetlin pattern with $\lambda^{(k)}$ being the $(n-k+1)$-st row (note that this is a partition). The definition of G implies

$$\lambda^{(0)} \subseteq \lambda^{(1)} \subseteq \cdots \subseteq \lambda^{(n)},$$

where $\lambda^{(0)}$ is the empty partition, and each skew shape $\lambda^{(k)}/\lambda^{(k-1)}$ is a horizontal strip. Thus define $T(G)$ by inserting k into the squares of the skew shape $\lambda^{(k)}/\lambda^{(k-1)}$, for $k = 1, \ldots, n$.

To each entry in a Gelfand-Tsetlin pattern, one may attach a decoration of a circle or a box (or both or neither). These decorations appear in the study of Weyl group multiple Dirichlet series, and are implemented here following the exposition in [BBF].

Note

We use the "right-hand" rule for determining circled and boxed entries.

Warning

The entries in Sage are 0-based and are thought of as flushed to the left in a matrix. In other words, the coordinates of entries in the Gelfand-Tsetlin patterns are thought of as the matrix:

$$\begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & \cdots & g_{0,n-2} & g_{n-1,n-1} \\ g_{1,0} & g_{1,1} & g_{1,2} & \cdots & g_{1,n-2} \\ g_{2,0} & g_{2,1} & g_{2,2} & \cdots \\ \vdots & \vdots & \vdots \\ g_{n-2,0} & g_{n-2,1} \\ g_{n-1,0} \end{bmatrix}.$$

However, in the discussions, we will be using the **standard** numbering system.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3, 2, 1], [2, 1], [1]]); G
[[3, 2, 1], [2, 1], [1]]
sage: G.pp()
  3     2     1
     2     1
        1
sage: G = GelfandTsetlinPattern([[7, 7, 4, 0], [7, 7, 3], [7, 5], [5]]); G.pp()
  7     7     4     0
     7     7     3
        7     5
           5
sage: G.to_tableau().pp()
  1  1  1  1  1  2  2
  2  2  2  2  2  3  3
  3  3  3  4
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(3), Integer(2), Integer(1)], [Integer(2),
↪Integer(1)], [Integer(1)]]); G
[[3, 2, 1], [2, 1], [1]]
>>> G.pp()
  3     2     1
     2     1
        1
>>> G = GelfandTsetlinPattern([[Integer(7), Integer(7), Integer(4), Integer(0)],
↪[Integer(7), Integer(7), Integer(3)], [Integer(7), Integer(5)], [Integer(5)]]);
↪G.pp()
  7     7     4     0
     7     7     3
        7     5
```

(continues on next page)

(continued from previous page)

```
            5
>>> G.to_tableau().pp()
  1  1  1  1  1  2  2
  2  2  2  2  2  3  3
  3  3  3  4
```

Tokuyama_coefficient (*name='t'*)

Return the Tokuyama coefficient attached to `self`.

Following the exposition of [BBF], Tokuyama's formula asserts

$$\sum_G (t+1)^{s(G)} t^{l(G)} z_1^{d_{n+1}} z_2^{d_n - d_{n+1}} \cdots z_{n+1}^{d_1 - d_2} = s_\lambda(z_1, \ldots, z_{n+1}) \prod_{i<j} (z_j + t z_i),$$

where the sum is over all strict Gelfand-Tsetlin patterns with fixed top row $\lambda + \rho$, with λ a partition with at most $n+1$ parts and $\rho = (n, n-1, \ldots, 1, 0)$, and s_λ is a Schur function.

INPUT:

- `name` – (default: `'t'`) an alternative name for the variable t

EXAMPLES:

```
sage: P = GelfandTsetlinPattern([[3,2,1],[2,2],[2]])
sage: P.Tokuyama_coefficient()
0
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[2]])
sage: G.Tokuyama_coefficient()
t^2 + t
sage: G = GelfandTsetlinPattern([[2,1,0],[1,1],[1]])
sage: G.Tokuyama_coefficient()
0
sage: G = GelfandTsetlinPattern([[5,3,2,1,0],[4,3,2,0],[4,2,1],[3,2],[3]])
sage: G.Tokuyama_coefficient()
t^8 + 3*t^7 + 3*t^6 + t^5
```

```
>>> from sage.all import *
>>> P = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(2)]])
>>> P.Tokuyama_coefficient()
0
>>> G = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(2)]])
>>> G.Tokuyama_coefficient()
t^2 + t
>>> G = GelfandTsetlinPattern([[Integer(2),Integer(1),Integer(0)],[Integer(1),
↪Integer(1)],[Integer(1)]])
>>> G.Tokuyama_coefficient()
0
>>> G = GelfandTsetlinPattern([[Integer(5),Integer(3),Integer(2),Integer(1),
↪Integer(0)],[Integer(4),Integer(3),Integer(2),Integer(0)],[Integer(4),
↪Integer(2),Integer(1)],[Integer(3),Integer(2)],[Integer(3)]])
>>> G.Tokuyama_coefficient()
t^8 + 3*t^7 + 3*t^6 + t^5
```

bender_knuth_involution(*i*)

Return the image of `self` under the i-th Bender-Knuth involution.

If the triangle `self` has size n then this is defined for $0 < i < n$.

The entries of `self` can take values in any ordered ring. Usually, this will be the integers but can also be the rationals or the real numbers.

This implements the construction of the Bender-Knuth involution using toggling due to Berenstein-Kirillov.

This agrees with the Bender-Knuth involution on semistandard tableaux.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[5,3,2,1,0],[4,3,2,0],[4,2,1],[3,2],[3]])
sage: G.bender_knuth_involution(2)
[[5, 3, 2, 1, 0], [4, 3, 2, 0], [4, 2, 1], [4, 1], [3]]

sage: G = GelfandTsetlinPattern([[3,2,0],[2.2,0],[2]])
sage: G.bender_knuth_involution(2)
[[3, 2, 0], [2.80000000000000, 2], [2]]
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(5),Integer(3),Integer(2),Integer(1),
↪Integer(0)],[Integer(4),Integer(3),Integer(2),Integer(0)],[Integer(4),
↪Integer(2),Integer(1)],[Integer(3),Integer(2)],[Integer(3)]])
>>> G.bender_knuth_involution(Integer(2))
[[5, 3, 2, 1, 0], [4, 3, 2, 0], [4, 2, 1], [4, 1], [3]]

>>> G = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(0)],[RealNumber(
↪'2.2'),Integer(0)],[Integer(2)]])
>>> G.bender_knuth_involution(Integer(2))
[[3, 2, 0], [2.80000000000000, 2], [2]]
```

boxed_entries()

Return the position of the boxed entries of `self`.

Using the *right-hand* rule, an entry $a_{i,j}$ is boxed if $a_{i,j} = a_{i-1,j-1}$; i.e., $a_{i,j}$ has the same value as its neighbor to the northwest.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.boxed_entries()
((1, 0),)
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(1)]])
>>> G.boxed_entries()
((1, 0),)
```

check()

Check that this is a valid Gelfand-Tsetlin pattern.

EXAMPLES:

```
sage: G = GelfandTsetlinPatterns()
sage: G([[3,2,1],[2,1],[1]]).check()
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPatterns()
>>> G([[Integer(3),Integer(2),Integer(1)],[Integer(2),Integer(1)],
↪[Integer(1)]]).check()
```

circled_entries()

Return the circled entries of `self`.

Using the *right-hand* rule, an entry $a_{i,j}$ is circled if $a_{i,j} = a_{i-1,j}$; i.e., $a_{i,j}$ has the same value as its neighbor to the northeast.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.circled_entries()
((1, 1), (2, 0))
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(1)]])
>>> G.circled_entries()
((1, 1), (2, 0))
```

is_strict()

Return `True` if `self` is a strict Gelfand-Tsetlin pattern.

A Gelfand-Tsetlin pattern is said to be *strict* if every row is strictly decreasing.

EXAMPLES:

```
sage: GelfandTsetlinPattern([[7,3,1],[6,2],[4]]).is_strict()
True
sage: GelfandTsetlinPattern([[3,2,1],[3,1],[1]]).is_strict()
True
sage: GelfandTsetlinPattern([[6,0,0],[3,0],[2]]).is_strict()
False
```

```
>>> from sage.all import *
>>> GelfandTsetlinPattern([[Integer(7),Integer(3),Integer(1)],[Integer(6),
↪Integer(2)],[Integer(4)]]).is_strict()
True
>>> GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(1)]]).is_strict()
True
>>> GelfandTsetlinPattern([[Integer(6),Integer(0),Integer(0)],[Integer(3),
↪Integer(0)],[Integer(2)]]).is_strict()
False
```

number_of_boxes()

Return the number of boxed entries. See `boxed_entries()`.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.number_of_boxes()
1
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(1)]])
>>> G.number_of_boxes()
1
```

number_of_circles()

Return the number of boxed entries. See *circled_entries()*.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.number_of_circles()
2
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(1)]])
>>> G.number_of_circles()
2
```

number_of_special_entries()

Return the number of special entries. See *special_entries()*.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[4,2,1],[4,1],[2]])
sage: G.number_of_special_entries()
1
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(4),Integer(2),Integer(1)],[Integer(4),
↪Integer(1)],[Integer(2)]])
>>> G.number_of_special_entries()
1
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: G = GelfandTsetlinPatterns()
sage: G([[3,2,1],[2,1],[1]]).pp()
  3     2     1
     2     1
        1
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPatterns()
```

(continues on next page)

(continued from previous page)

```
>>> G([[Integer(3),Integer(2),Integer(1)],[Integer(2),Integer(1)],
↪[Integer(1)]]).pp()
  3     2     1
     2     1
        1
```

row_sums()

Return the list of row sums.

For a Gelfand-Tsetlin pattern G, the i-th row sum d_i is

$$d_i = d_i(G) = \sum_{j=i}^{n} a_{i,j}.$$

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[5,3,2,1,0],[4,3,2,0],[4,2,1],[3,2],[3]])
sage: G.row_sums()
[11, 9, 7, 5, 3]
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[2]])
sage: G.row_sums()
[6, 4, 2]
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(5),Integer(3),Integer(2),Integer(1),
↪Integer(0)],[Integer(4),Integer(3),Integer(2),Integer(0)],[Integer(4),
↪Integer(2),Integer(1)],[Integer(3),Integer(2)],[Integer(3)]])
>>> G.row_sums()
[11, 9, 7, 5, 3]
>>> G = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(2)]])
>>> G.row_sums()
[6, 4, 2]
```

special_entries()

Return the special entries.

An entry $a_{i,j}$ is special if $a_{i-1,j-1} > a_{i,j} > a_{i-1,j}$, that is to say, the entry is neither boxed nor circled and is **not** in the first row. The name was coined by [Tok88].

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.special_entries()
()
sage: G = GelfandTsetlinPattern([[4,2,1],[4,1],[2]])
sage: G.special_entries()
((2, 0),)
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(1)]])
>>> G.special_entries()
```

(continues on next page)

(continued from previous page)

```
()
>>> G = GelfandTsetlinPattern([[Integer(4),Integer(2),Integer(1)],[Integer(4),
↪Integer(1)],[Integer(2)]])
>>> G.special_entries()
((2, 0),)
```

to_tableau()

Return `self` as a semistandard Young tableau.

The conversion from a Gelfand-Tsetlin pattern to a semistandard Young tableaux is as follows. Let G be a Gelfand-Tsetlin pattern with $\lambda^{(k)}$ being the $(n-k+1)$-st row (note that this is a partition). The definition of G implies

$$\lambda^{(0)} \subseteq \lambda^{(1)} \subseteq \cdots \subseteq \lambda^{(n)},$$

where $\lambda^{(0)}$ is the empty partition, and each skew shape $\lambda^{(k)}/\lambda^{(k-1)}$ is a horizontal strip. Thus define $T(G)$ by inserting k into the squares of the skew shape $\lambda^{(k)}/\lambda^{(k-1)}$, for $k = 1, \ldots, n$.

EXAMPLES:

```
sage: G = GelfandTsetlinPatterns()
sage: elt = G([[3,2,1],[2,1],[1]])
sage: T = elt.to_tableau(); T
[[1, 2, 3], [2, 3], [3]]
sage: T.pp()
  1  2  3
  2  3
  3
sage: G(T) == elt
True
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPatterns()
>>> elt = G([[Integer(3),Integer(2),Integer(1)],[Integer(2),Integer(1)],
↪[Integer(1)]])
>>> T = elt.to_tableau(); T
[[1, 2, 3], [2, 3], [3]]
>>> T.pp()
  1  2  3
  2  3
  3
>>> G(T) == elt
True
```

weight()

Return the weight of `self`.

Define the weight of G to be the content of the tableau to which G corresponds under the bijection between Gelfand-Tsetlin patterns and semistandard tableaux. More precisely,

$$\mathrm{wt}(G) = (d_n, d_{n-1} - d_n, \ldots, d_1 - d_2),$$

where the d_i are the row sums.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[2,1,0],[1,0],[1]])
sage: G.weight()
(1, 0, 2)
sage: G = GelfandTsetlinPattern([[4,2,1],[3,1],[2]])
sage: G.weight()
(2, 2, 3)
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPattern([[Integer(2),Integer(1),Integer(0)],[Integer(1),
↪Integer(0)],[Integer(1)]])
>>> G.weight()
(1, 0, 2)
>>> G = GelfandTsetlinPattern([[Integer(4),Integer(2),Integer(1)],[Integer(3),
↪Integer(1)],[Integer(2)]])
>>> G.weight()
(2, 2, 3)
```

class sage.combinat.gelfand_tsetlin_patterns.**GelfandTsetlinPatterns**(*n*, *k*, *strict*)

Bases: UniqueRepresentation, Parent

Gelfand-Tsetlin patterns.

INPUT:

- n – the width or depth of the array, also known as the rank
- k – (default: None) if specified, this is the maximum value that can occur in the patterns
- top_row – (default: None) if specified, this is the fixed top row of all patterns
- strict – (default: False) set to True if all patterns are strict patterns

Element

alias of *GelfandTsetlinPattern*

random_element()

Return a uniformly random Gelfand-Tsetlin pattern.

EXAMPLES:

```
sage: g = GelfandTsetlinPatterns(4, 5)
sage: x = g.random_element()
sage: x in g
True
sage: len(x)
4
sage: all(y in range(5+1) for z in x for y in z)
True
sage: x.check()
```

```
>>> from sage.all import *
>>> g = GelfandTsetlinPatterns(Integer(4), Integer(5))
>>> x = g.random_element()
>>> x in g
True
>>> len(x)
```

(continues on next page)

(continued from previous page)

```
4
>>> all(y in range(Integer(5)+Integer(1)) for z in x for y in z)
True
>>> x.check()
```

```
sage: g = GelfandTsetlinPatterns(4, 5, strict=True)
sage: x = g.random_element()
sage: x in g
True
sage: len(x)
4
sage: all(y in range(5+1) for z in x for y in z)
True
sage: x.check()
sage: x.is_strict()
True
```

```
>>> from sage.all import *
>>> g = GelfandTsetlinPatterns(Integer(4), Integer(5), strict=True)
>>> x = g.random_element()
>>> x in g
True
>>> len(x)
4
>>> all(y in range(Integer(5)+Integer(1)) for z in x for y in z)
True
>>> x.check()
>>> x.is_strict()
True
```

class sage.combinat.gelfand_tsetlin_patterns.**GelfandTsetlinPatternsTopRow**(*top_row*, *strict*)

Bases: *GelfandTsetlinPatterns*

Gelfand-Tsetlin patterns with a fixed top row.

Tokuyama_formula(*name='t'*)

Return the Tokuyama formula of `self`.

Following the exposition of [BBF], Tokuyama's formula asserts

$$\sum_G (t+1)^{s(G)} t^{l(G)} z_1^{d_{n+1}} z_2^{d_n - d_{n+1}} \cdots z_{n+1}^{d_1 - d_2} = s_\lambda(z_1, \ldots, z_{n+1}) \prod_{i<j} (z_j + t z_i),$$

where the sum is over all strict Gelfand-Tsetlin patterns with fixed top row $\lambda + \rho$, with λ a partition with at most $n+1$ parts and $\rho = (n, n-1, \ldots, 1, 0)$, and s_λ is a Schur function.

INPUT:

- `name` – (default: `'t'`) an alternative name for the variable t

EXAMPLES:

```
sage: GT = GelfandTsetlinPatterns(top_row=[2,1,0],strict=True)
sage: GT.Tokuyama_formula()
t^3*x1^2*x2 + t^2*x1*x2^2 + t^2*x1^2*x3 + t^2*x1*x2*x3 + t*x1*x2*x3 + t*x2^
```

(continues on next page)

(continued from previous page)

```
↪2*x3 + t*x1*x3^2 + x2*x3^2
sage: GT = GelfandTsetlinPatterns(top_row=[3,2,1],strict=True)
sage: GT.Tokuyama_formula()
t^3*x1^3*x2^2*x3 + t^2*x1^2*x2^3*x3 + t^2*x1^3*x2*x3^2 + t^2*x1^2*x2^2*x3^2 +␣
↪t*x1^2*x2^2*x3^2 + t*x1*x2^3*x3^2 + t*x1^2*x2*x3^3 + x1*x2^2*x3^3
sage: GT = GelfandTsetlinPatterns(top_row=[1,1,1],strict=True)
sage: GT.Tokuyama_formula()
0
```

```
>>> from sage.all import *
>>> GT = GelfandTsetlinPatterns(top_row=[Integer(2),Integer(1),Integer(0)],
↪strict=True)
>>> GT.Tokuyama_formula()
t^3*x1^2*x2 + t^2*x1*x2^2 + t^2*x1^2*x3 + t^2*x1*x2*x3 + t*x1*x2*x3 + t*x2^
↪2*x3 + t*x1*x3^2 + x2*x3^2
>>> GT = GelfandTsetlinPatterns(top_row=[Integer(3),Integer(2),Integer(1)],
↪strict=True)
>>> GT.Tokuyama_formula()
t^3*x1^3*x2^2*x3 + t^2*x1^2*x2^3*x3 + t^2*x1^3*x2*x3^2 + t^2*x1^2*x2^2*x3^2 +␣
↪t*x1^2*x2^2*x3^2 + t*x1*x2^3*x3^2 + t*x1^2*x2*x3^3 + x1*x2^2*x3^3
>>> GT = GelfandTsetlinPatterns(top_row=[Integer(1),Integer(1),Integer(1)],
↪strict=True)
>>> GT.Tokuyama_formula()
0
```

random_element()

Return a uniformly random Gelfand-Tsetlin pattern with specified top row.

EXAMPLES:

```
sage: g = GelfandTsetlinPatterns(top_row = [4, 3, 1, 1])
sage: x = g.random_element()
sage: x in g
True
sage: x[0] == [4, 3, 1, 1]
True
sage: x.check()

sage: g = GelfandTsetlinPatterns(top_row=[4, 3, 2, 1], strict=True)
sage: x = g.random_element()
sage: x in g
True
sage: x[0] == [4, 3, 2, 1]
True
sage: x.is_strict()
True
sage: x.check()
```

```
>>> from sage.all import *
>>> g = GelfandTsetlinPatterns(top_row = [Integer(4), Integer(3), Integer(1),␣
↪Integer(1)])
>>> x = g.random_element()
```

(continues on next page)

(continued from previous page)

```
>>> x in g
True
>>> x[Integer(0)] == [Integer(4), Integer(3), Integer(1), Integer(1)]
True
>>> x.check()

>>> g = GelfandTsetlinPatterns(top_row=[Integer(4), Integer(3), Integer(2),␣
↪Integer(1)], strict=True)
>>> x = g.random_element()
>>> x in g
True
>>> x[Integer(0)] == [Integer(4), Integer(3), Integer(2), Integer(1)]
True
>>> x.is_strict()
True
>>> x.check()
```

top_row()

Return the top row of `self`.

EXAMPLES:

```
sage: G = GelfandTsetlinPatterns(top_row=[4,4,3,1])
sage: G.top_row()
(4, 4, 3, 1)
```

```
>>> from sage.all import *
>>> G = GelfandTsetlinPatterns(top_row=[Integer(4),Integer(4),Integer(3),
↪Integer(1)])
>>> G.top_row()
(4, 4, 3, 1)
```

5.1.116 Paths in Directed Acyclic Graphs

sage.combinat.graph_path.**GraphPaths**(*g*, *source=None*, *target=None*)

Return the combinatorial class of paths in the directed acyclic graph g.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
```

If source and target are not given, then the returned class contains all paths (including trivial paths containing only one vertex).

```
sage: p = GraphPaths(G); p
Paths in Multi-digraph on 5 vertices
sage: p.cardinality()
```

(continues on next page)

(continued from previous page)

```
37
sage: path = p.random_element()
sage: all(G.has_edge(*path[i:i+2]) for i in range(len(path) -1))
True
```

```
>>> from sage.all import *
>>> p = GraphPaths(G); p
Paths in Multi-digraph on 5 vertices
>>> p.cardinality()
37
>>> path = p.random_element()
>>> all(G.has_edge(*path[i:i+Integer(2)]) for i in range(len(path) -Integer(1)))
True
```

If the source is specified, then the returned class contains all of the paths starting at the vertex source (including the trivial path).

```
sage: p = GraphPaths(G, source=3); p
Paths in Multi-digraph on 5 vertices starting at 3
sage: p.list()
[[3], [3, 4], [3, 4, 5], [3, 4, 5]]
```

```
>>> from sage.all import *
>>> p = GraphPaths(G, source=Integer(3)); p
Paths in Multi-digraph on 5 vertices starting at 3
>>> p.list()
[[3], [3, 4], [3, 4, 5], [3, 4, 5]]
```

If the target is specified, then the returned class contains all of the paths ending at the vertex target (including the trivial path).

```
sage: p = GraphPaths(G, target=3); p
Paths in Multi-digraph on 5 vertices ending at 3
sage: p.cardinality()
5
sage: p.list()
[[3], [1, 3], [2, 3], [1, 2, 3], [1, 2, 3]]
```

```
>>> from sage.all import *
>>> p = GraphPaths(G, target=Integer(3)); p
Paths in Multi-digraph on 5 vertices ending at 3
>>> p.cardinality()
5
>>> p.list()
[[3], [1, 3], [2, 3], [1, 2, 3], [1, 2, 3]]
```

If both the target and source are specified, then the returned class contains all of the paths from source to target.

```
sage: p = GraphPaths(G, source=1, target=3); p
Paths in Multi-digraph on 5 vertices starting at 1 and ending at 3
sage: p.cardinality()
3
```

(continues on next page)

(continued from previous page)

```
sage: p.list()
[[1, 2, 3], [1, 2, 3], [1, 3]]
```

```
>>> from sage.all import *
>>> p = GraphPaths(G, source=Integer(1), target=Integer(3)); p
Paths in Multi-digraph on 5 vertices starting at 1 and ending at 3
>>> p.cardinality()
3
>>> p.list()
[[1, 2, 3], [1, 2, 3], [1, 3]]
```

Note that G must be a directed acyclic graph.

```
sage: G = DiGraph({1:[2,2,3,5], 2:[3,4], 3:[4], 4:[2,5,7], 5:[6]},
↪multiedges=True)
sage: GraphPaths(G)
Traceback (most recent call last):
...
TypeError: g must be a directed acyclic graph
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3),Integer(5)],
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],
↪Integer(4):[Integer(2),Integer(5),Integer(7)], Integer(5):[Integer(6)]},
↪multiedges=True)
>>> GraphPaths(G)
Traceback (most recent call last):
...
TypeError: g must be a directed acyclic graph
```

class sage.combinat.graph_path.**GraphPaths_all**(*g*)

Bases: `Parent`, *`GraphPaths_common`*

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.cardinality()
37
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> p = GraphPaths(G)
>>> p.cardinality()
37
```

list()

Return a list of the paths of `self`.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: len(GraphPaths(G).list())
37
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> len(GraphPaths(G).list())
37
```

class sage.combinat.graph_path.**GraphPaths_common**

Bases: object

incoming_edges(*v*)

Return a list of v's incoming edges.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.incoming_edges(2)
[(1, 2, None), (1, 2, None)]
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> p = GraphPaths(G)
>>> p.incoming_edges(Integer(2))
[(1, 2, None), (1, 2, None)]
```

incoming_paths(*v*)

Return a list of paths that end at v.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.incoming_paths(2)
[[2], [1, 2], [1, 2]]
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> gp = GraphPaths(G)
>>> gp.incoming_paths(Integer(2))
[[2], [1, 2], [1, 2]]
```

outgoing_edges(*v*)

Return a list of v's outgoing edges.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.outgoing_edges(2)
[(2, 3, None), (2, 4, None)]
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> p = GraphPaths(G)
>>> p.outgoing_edges(Integer(2))
[(2, 3, None), (2, 4, None)]
```

outgoing_paths(*v*)

Return a list of the paths that start at v.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.outgoing_paths(3)
[[3], [3, 4], [3, 4, 5], [3, 4, 5]]
sage: gp.outgoing_paths(2)
[[2],
 [2, 3],
 [2, 3, 4],
 [2, 3, 4, 5],
 [2, 3, 4, 5],
 [2, 4],
 [2, 4, 5],
 [2, 4, 5]]
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> gp = GraphPaths(G)
>>> gp.outgoing_paths(Integer(3))
[[3], [3, 4], [3, 4, 5], [3, 4, 5]]
>>> gp.outgoing_paths(Integer(2))
[[2],
 [2, 3],
 [2, 3, 4],
 [2, 3, 4, 5],
 [2, 3, 4, 5],
 [2, 4],
 [2, 4, 5],
 [2, 4, 5]]
```

paths()

Return a list of all the paths of `self`.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: len(gp.paths())
37
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> gp = GraphPaths(G)
>>> len(gp.paths())
37
```

paths_from_source_to_target(*source*, *target*)

Return a list of paths from source to target.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.paths_from_source_to_target(2,4)
[[2, 3, 4], [2, 4]]
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> gp = GraphPaths(G)
>>> gp.paths_from_source_to_target(Integer(2),Integer(4))
[[2, 3, 4], [2, 4]]
```

class sage.combinat.graph_path.**GraphPaths_s**(*g*, *source*)

Bases: Parent, *GraphPaths_common*

list()

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G, 4)
sage: p.list()
[[4], [4, 5], [4, 5]]
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> p = GraphPaths(G, Integer(4))
>>> p.list()
[[4], [4, 5], [4, 5]]
```

class sage.combinat.graph_path.**GraphPaths_st**(*g*, *source*, *target*)

Bases: Parent, *GraphPaths_common*

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: GraphPaths(G,1,2).cardinality()
2
sage: GraphPaths(G,1,3).cardinality()
3
sage: GraphPaths(G,1,4).cardinality()
5
sage: GraphPaths(G,1,5).cardinality()
10
sage: GraphPaths(G,2,3).cardinality()
1
sage: GraphPaths(G,2,4).cardinality()
2
sage: GraphPaths(G,2,5).cardinality()
4
sage: GraphPaths(G,3,4).cardinality()
1
sage: GraphPaths(G,3,5).cardinality()
2
sage: GraphPaths(G,4,5).cardinality()
2
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> GraphPaths(G,Integer(1),Integer(2)).cardinality()
2
>>> GraphPaths(G,Integer(1),Integer(3)).cardinality()
3
>>> GraphPaths(G,Integer(1),Integer(4)).cardinality()
5
>>> GraphPaths(G,Integer(1),Integer(5)).cardinality()
10
>>> GraphPaths(G,Integer(2),Integer(3)).cardinality()
1
>>> GraphPaths(G,Integer(2),Integer(4)).cardinality()
2
>>> GraphPaths(G,Integer(2),Integer(5)).cardinality()
4
>>> GraphPaths(G,Integer(3),Integer(4)).cardinality()
1
>>> GraphPaths(G,Integer(3),Integer(5)).cardinality()
2
>>> GraphPaths(G,Integer(4),Integer(5)).cardinality()
2
```

list()

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G,1,2)
```

(continues on next page)

(continued from previous page)

```
sage: p.list()
[[1, 2], [1, 2]]
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> p = GraphPaths(G,Integer(1),Integer(2))
>>> p.list()
[[1, 2], [1, 2]]
```

class sage.combinat.graph_path.**GraphPaths_t**(*g*, *target*)

Bases: `Parent`, *GraphPaths_common*

list()

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G, target=4)
sage: p.list()
[[4],
 [2, 4],
 [1, 2, 4],
 [1, 2, 4],
 [3, 4],
 [1, 3, 4],
 [2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 4]]
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1):[Integer(2),Integer(2),Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5),Integer(5)]}, multiedges=True)
>>> p = GraphPaths(G, target=Integer(4))
>>> p.list()
[[4],
 [2, 4],
 [1, 2, 4],
 [1, 2, 4],
 [3, 4],
 [1, 3, 4],
 [2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 4]]
```

5.1.117 Gray codes

Functions

sage.combinat.gray_codes.**combinations**(*n*, *t*)

Iterator through the switches of the revolving door algorithm.

The revolving door algorithm is a way to generate all combinations of a set (i.e. the subset of given cardinality) in such way that two consecutive subsets differ by one element. At each step, the iterator output a pair (i,j) where the item i has to be removed and j has to be added.

The ground set is always $\{0, 1, ..., n-1\}$. Note that n can be infinity in that algorithm.

See [Knu2011] Section 7.2.1.3, "Generating All Combinations".

INPUT:

- n – integer or Infinity; size of the ground set
- t – integer; size of the subsets

EXAMPLES:

```
sage: from sage.combinat.gray_codes import combinations
sage: b = [1, 1, 1, 0, 0]
sage: for i,j in combinations(5,3):
....:     b[i] = 0; b[j] = 1
....:     print(b)
[1, 0, 1, 1, 0]
[0, 1, 1, 1, 0]
[1, 1, 0, 1, 0]
[1, 0, 0, 1, 1]
[0, 1, 0, 1, 1]
[0, 0, 1, 1, 1]
[1, 0, 1, 0, 1]
[0, 1, 1, 0, 1]
[1, 1, 0, 0, 1]

sage: s = set([0,1])
sage: for i,j in combinations(4,2):
....:     s.remove(i)
....:     s.add(j)
....:     print(sorted(s))
[1, 2]
[0, 2]
[2, 3]
[1, 3]
[0, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.gray_codes import combinations
>>> b = [Integer(1), Integer(1), Integer(1), Integer(0), Integer(0)]
>>> for i,j in combinations(Integer(5),Integer(3)):
...     b[i] = Integer(0); b[j] = Integer(1)
...     print(b)
[1, 0, 1, 1, 0]
[0, 1, 1, 1, 0]
[1, 1, 0, 1, 0]
[1, 0, 0, 1, 1]
[0, 1, 0, 1, 1]
[0, 0, 1, 1, 1]
[1, 0, 1, 0, 1]
[0, 1, 1, 0, 1]
```

(continues on next page)

(continued from previous page)

```
[1, 1, 0, 0, 1]

>>> s = set([Integer(0),Integer(1)])
>>> for i,j in combinations(Integer(4),Integer(2)):
...     s.remove(i)
...     s.add(j)
...     print(sorted(s))
[1, 2]
[0, 2]
[2, 3]
[1, 3]
[0, 3]
```

Note that n can be infinity:

```
sage: c = combinations(Infinity,4)
sage: s = set([0,1,2,3])
sage: for _ in range(10):
....:     i,j = next(c)
....:     s.remove(i); s.add(j)
....:     print(sorted(s))
[0, 1, 3, 4]
[1, 2, 3, 4]
[0, 2, 3, 4]
[0, 1, 2, 4]
[0, 1, 4, 5]
[1, 2, 4, 5]
[0, 2, 4, 5]
[2, 3, 4, 5]
[1, 3, 4, 5]
[0, 3, 4, 5]
sage: for _ in range(1000):
....:     i,j = next(c)
....:     s.remove(i); s.add(j)
sage: sorted(s)
[0, 4, 13, 14]
```

```
>>> from sage.all import *
>>> c = combinations(Infinity,Integer(4))
>>> s = set([Integer(0),Integer(1),Integer(2),Integer(3)])
>>> for _ in range(Integer(10)):
...     i,j = next(c)
...     s.remove(i); s.add(j)
...     print(sorted(s))
[0, 1, 3, 4]
[1, 2, 3, 4]
[0, 2, 3, 4]
[0, 1, 2, 4]
[0, 1, 4, 5]
[1, 2, 4, 5]
[0, 2, 4, 5]
[2, 3, 4, 5]
```

(continues on next page)

(continued from previous page)

```
[1, 3, 4, 5]
[0, 3, 4, 5]
>>> for _ in range(Integer(1000)):
...     i,j = next(c)
...     s.remove(i); s.add(j)
>>> sorted(s)
[0, 4, 13, 14]
```

sage.combinat.gray_codes.**product**(*m*)

Iterator over the switch for the iteration of the product $[m_0] \times [m_1] \ldots \times [m_k]$.

The iterator return at each step a pair `(p,i)` which corresponds to the modification to perform to get the next element. More precisely, one has to apply the increment `i` at the position `p`. By construction, the increment is either `+1` or `-1`.

This is algorithm H in [Knu2011] Section 7.2.1.1, “Generating All n-Tuples”: loopless reflected mixed-radix Gray generation.

INPUT:

- `m` – list or tuple of positive integers that correspond to the size of the sets in the product

EXAMPLES:

```
sage: from sage.combinat.gray_codes import product
sage: l = [0,0,0]
sage: for p,i in product([3,3,3]):
....:     l[p] += i
....:     print(l)
[1, 0, 0]
[2, 0, 0]
[2, 1, 0]
[1, 1, 0]
[0, 1, 0]
[0, 2, 0]
[1, 2, 0]
[2, 2, 0]
[2, 2, 1]
[1, 2, 1]
[0, 2, 1]
[0, 1, 1]
[1, 1, 1]
[2, 1, 1]
[2, 0, 1]
[1, 0, 1]
[0, 0, 1]
[0, 0, 2]
[1, 0, 2]
[2, 0, 2]
[2, 1, 2]
[1, 1, 2]
[0, 1, 2]
[0, 2, 2]
[1, 2, 2]
```

(continues on next page)

(continued from previous page)

```
[2, 2, 2]
sage: l = [0,0]
sage: for i,j in product([2,1]):
....:     l[i] += j
....:     print(l)
[1, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.gray_codes import product
>>> l = [Integer(0),Integer(0),Integer(0)]
>>> for p,i in product([Integer(3),Integer(3),Integer(3)]):
...     l[p] += i
...     print(l)
[1, 0, 0]
[2, 0, 0]
[2, 1, 0]
[1, 1, 0]
[0, 1, 0]
[0, 2, 0]
[1, 2, 0]
[2, 2, 0]
[2, 2, 1]
[1, 2, 1]
[0, 2, 1]
[0, 1, 1]
[1, 1, 1]
[2, 1, 1]
[2, 0, 1]
[1, 0, 1]
[0, 0, 1]
[0, 0, 2]
[1, 0, 2]
[2, 0, 2]
[2, 1, 2]
[1, 1, 2]
[0, 1, 2]
[0, 2, 2]
[1, 2, 2]
[2, 2, 2]
>>> l = [Integer(0),Integer(0)]
>>> for i,j in product([Integer(2),Integer(1)]):
...     l[i] += j
...     print(l)
[1, 0]
```

5.1.118 Growth diagrams and dual graded graphs

AUTHORS:

- Martin Rubey (2016-09): Initial version
- Martin Rubey (2017-09): generalize, more rules, improve documentation
- Travis Scrimshaw (2017-09): switch to rule-based framework

Todo

- provide examples for the P and Q-symbol in the skew case
- implement a method providing a visualization of the growth diagram with all labels, perhaps as LaTeX code
- when shape is given, check that it is compatible with filling or labels
- optimize rules, mainly for `RuleRSK` and `RuleBurge`
- implement backward rules for `GrowthDiagram.rules.Domino`
- implement backward rule from [LLMSSZ2013], [LS2007]
- make semistandard extension generic
- accommodate dual filtered graphs

A guided tour

Growth diagrams, invented by Sergey Fomin [Fom1994], [Fom1995], provide a vast generalization of the Robinson-Schensted-Knuth (RSK) correspondence between matrices with nonnegative integer entries and pairs of semi-standard Young tableaux of the same shape.

The main fact is that many correspondences similar to RSK can be defined by providing a pair of so-called local rules: a 'forward' rule, whose input are three vertices y, t and x of a certain directed graph (in the case of Robinson-Schensted: the directed graph corresponding to Young's lattice) and an integer (in the case of Robinson-Schensted: 0 or 1), and whose output is a fourth vertex z. This rule should be invertible in the following sense: there is a so-called 'backward' rule that recovers the integer and t given y, z and x.

As an example, the growth rules for the classical RSK correspondence are provided by `RuleRSK`. To produce a growth diagram, pass the desired rule and a permutation to `GrowthDiagram`:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: w = [2,3,6,1,4,5]; G = GrowthDiagram(RuleRSK, w); G
0  0  0  1  0  0
1  0  0  0  0  0
0  1  0  0  0  0
0  0  0  0  1  0
0  0  0  0  0  1
0  0  1  0  0  0
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> w = [Integer(2),Integer(3),Integer(6),Integer(1),Integer(4),Integer(5)]; G =␣
↪GrowthDiagram(RuleRSK, w); G
0  0  0  1  0  0
1  0  0  0  0  0
0  1  0  0  0  0
0  0  0  0  1  0
0  0  0  0  0  1
0  0  1  0  0  0
```

The forward rule just mentioned assigns 49 partitions to the corners of each of the 36 cells of this matrix (i.e., 49 the vertices of a $(6 + 1) \times (6 + 1)$ grid graph), with the exception of the corners on the left and top boundary, which are initialized with the empty partition. More precisely, for each cell, the `forward_rule()` computes the partition z labelling the lower right corner, given the content c of a cell and the other three partitions:

```
t --- x
|  c  |
y --- z
```

Warning: Note that a growth diagram is printed with matrix coordinates, the origin being in the top-left corner. Therefore, the growth is from the top left to the bottom right!

The partitions along the boundary opposite of the origin, reading from the bottom left to the top right, are obtained by using the method *out_labels()*:

```
sage: G.out_labels()
[[],
 [1],
 [2],
 [3],
 [3, 1],
 [3, 2],
 [4, 2],
 [4, 1],
 [3, 1],
 [2, 1],
 [1, 1],
 [1],
 []]
```

```
>>> from sage.all import *
>>> G.out_labels()
[[],
 [1],
 [2],
 [3],
 [3, 1],
 [3, 2],
 [4, 2],
 [4, 1],
 [3, 1],
 [2, 1],
 [1, 1],
 [1],
 []]
```

However, in the case of a rectangular filling, it is more practical to split this sequence of labels in two. Interpreting the sequence of partitions along the right boundary as a standard Young tableau, we then obtain the so-called *P_symbol()*, the partitions along the bottom boundary yield the so-called *Q_symbol()*. These coincide with the output of the classical *RSK()* insertion algorithm:

```
sage: ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  3  4  5     1  2  3  6 ]
[   2  6      ,    4  5       ]
```

(continues on next page)

(continued from previous page)

```
sage: ascii_art(RSK(w))
[   1  3  4  5    1  2  3  6 ]
[   2  6       ,  4  5       ]
```

```
>>> from sage.all import *
>>> ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  3  4  5    1  2  3  6 ]
[   2  6       ,  4  5       ]
>>> ascii_art(RSK(w))
[   1  3  4  5    1  2  3  6 ]
[   2  6       ,  4  5       ]
```

The filling can be recovered knowing the partitions labelling the corners of the bottom and the right boundary alone, by repeatedly applying the *backward_rule()*. Therefore, to initialize a *GrowthDiagram*, we can provide these labels instead of the filling:

```
sage: GrowthDiagram(RuleRSK, labels=G.out_labels())
0  0  0  1  0  0
1  0  0  0  0  0
0  1  0  0  0  0
0  0  0  0  1  0
0  0  0  0  0  1
0  0  1  0  0  0
```

```
>>> from sage.all import *
>>> GrowthDiagram(RuleRSK, labels=G.out_labels())
0  0  0  1  0  0
1  0  0  0  0  0
0  1  0  0  0  0
0  0  0  0  1  0
0  0  0  0  0  1
0  0  1  0  0  0
```

Invocation

In general, growth diagrams are defined for $0-1$-fillings of arbitrary skew shapes. In the case of the Robinson-Schensted-Knuth correspondence, even arbitrary nonnegative integers are allowed. In other cases, entries may be either zero or an r-th root of unity - for example, *RuleDomino* insertion is defined for signed permutations, that is, $r=2$. Traditionally, words and permutations are also used to specify a filling in special cases.

To accommodate all this, the filling may be passed in various ways. The most general possibility is to pass a dictionary of coordinates to (signed) entries, where zeros can be omitted. In this case, when the parameter `shape` is not explicitly specified, it is assumed to be the minimal rectangle containing the origin and all coordinates with nonzero entries.

For example, consider the following generalized permutation:

```
1 2 2 2 4 4
4 2 3 3 2 3
```

that we encode as the dictionary:

```
sage: P = {(1-1,4-1): 1, (2-1,2-1): 1, (2-1,3-1): 2, (4-1,2-1): 1, (4-1,3-1): 1}
```

```
>>> from sage.all import *
>>> P = {(Integer(1)-Integer(1),Integer(4)-Integer(1)): Integer(1), (Integer(2)-
↪Integer(1),Integer(2)-Integer(1)): Integer(1), (Integer(2)-Integer(1),Integer(3)-
↪Integer(1)): Integer(2), (Integer(4)-Integer(1),Integer(2)-Integer(1)): Integer(1),␣
↪(Integer(4)-Integer(1),Integer(3)-Integer(1)): Integer(1)}
```

Note that we are subtracting 1 from all entries because of zero-based indexing, we obtain:

```
sage: GrowthDiagram(RuleRSK, P)
0  0  0  0
0  1  0  1
0  2  0  1
1  0  0  0
```

```
>>> from sage.all import *
>>> GrowthDiagram(RuleRSK, P)
0  0  0  0
0  1  0  1
0  2  0  1
1  0  0  0
```

Alternatively, we could create the same growth diagram using a matrix.

Let us also mention that one can pass the arguments specifying a growth diagram directly to the rule:

```
sage: RuleRSK(P)
0  0  0  0
0  1  0  1
0  2  0  1
1  0  0  0
```

```
>>> from sage.all import *
>>> RuleRSK(P)
0  0  0  0
0  1  0  1
0  2  0  1
1  0  0  0
```

In contrast to the classical insertion algorithms, growth diagrams immediately generalize to fillings whose shape is an arbitrary skew partition:

```
sage: GrowthDiagram(RuleRSK, [3,1,2], shape=SkewPartition([[3,3,2],[1,1]]))
.  1  0
.  0  1
1  0
```

```
>>> from sage.all import *
>>> GrowthDiagram(RuleRSK, [Integer(3),Integer(1),Integer(2)],␣
↪shape=SkewPartition([[Integer(3),Integer(3),Integer(2)],[Integer(1),Integer(1)]]))
.  1  0
.  0  1
1  0
```

As an important example, consider the Stanley-Sundaram correspondence between oscillating tableaux and (partial) perfect matchings. Perfect matchings of $\{1, \ldots, 2r\}$ are in bijection with $0-1$-fillings of a triangular shape with $2r-1$ rows, such that for each k there is either exactly one nonzero entry in row k or exactly one nonzero entry in column $2r-k$. Explicitly, if (i, j) is a pair in the perfect matching, the entry in column $i-1$ and row $2r-j$ equals 1. For example:

```
sage: m = [[1,5],[3,4],[2,7],[6,8]]
sage: G = RuleRSK({(i-1, 8-j): 1 for i,j in m}, shape=[7,6,5,4,3,2,1]); G
0  0  0  0  0  1  0
0  1  0  0  0  0
0  0  0  0  0
1  0  0  0
0  0  1
0  0
0
```

```
>>> from sage.all import *
>>> m = [[Integer(1),Integer(5)],[Integer(3),Integer(4)],[Integer(2),Integer(7)],
↪[Integer(6),Integer(8)]]
>>> G = RuleRSK({(i-Integer(1), Integer(8)-j): Integer(1) for i,j in m},␣
↪shape=[Integer(7),Integer(6),Integer(5),Integer(4),Integer(3),Integer(2),
↪Integer(1)]); G
0  0  0  0  0  1  0
0  1  0  0  0  0
0  0  0  0  0
1  0  0  0
0  0  1
0  0
0
```

The partitions labelling the bottom-right corners along the boundary opposite of the origin then form a so-called oscillating tableau - the remaining partitions along the bottom-right boundary are redundant:

```
sage: G.out_labels()[1::2]
[[1], [1, 1], [2, 1], [1, 1], [1], [1, 1], [1]]
```

```
>>> from sage.all import *
>>> G.out_labels()[Integer(1)::Integer(2)]
[[1], [1, 1], [2, 1], [1, 1], [1], [1, 1], [1]]
```

Another great advantage of growth diagrams is that we immediately have access to a skew version of the correspondence, by providing different initialization for the labels on the side of the origin. We reproduce the original example of Bruce Sagan and Richard Stanley, see also Tom Roby's thesis [Rob1991]:

```
sage: w = {(1-1,4-1): 1, (2-1,2-1): 1, (4-1,3-1): 1}
sage: T = SkewTableau([[None, None], [None, 5], [1]])
sage: U = SkewTableau([[None, None], [None, 3], [5]])
sage: labels = T.to_chain()[::-1] + U.to_chain()[1:]
sage: G = GrowthDiagram(RuleRSK, filling=w, shape=[5,5,5,5,5], labels=labels); G
0  0  0  0  0
0  1  0  0  0
0  0  0  1  0
1  0  0  0  0
0  0  0  0  0
```

(continues on next page)

(continued from previous page)

```
sage: ascii_art([G.P_symbol(), G.Q_symbol()])
[   .  .  2  3    .  .  1  4 ]
[   .  .          .  .       ]
[   .  4          .  2       ]
[   1             3          ]
[   5        ,    5          ]
```

```
>>> from sage.all import *
>>> w = {(Integer(1)-Integer(1),Integer(4)-Integer(1)): Integer(1), (Integer(2)-
↪Integer(1),Integer(2)-Integer(1)): Integer(1), (Integer(4)-Integer(1),Integer(3)-
↪Integer(1)): Integer(1)}
>>> T = SkewTableau([[None, None], [None, Integer(5)], [Integer(1)]])
>>> U = SkewTableau([[None, None], [None, Integer(3)], [Integer(5)]])
>>> labels = T.to_chain()[::-Integer(1)] + U.to_chain()[Integer(1):]
>>> G = GrowthDiagram(RuleRSK, filling=w, shape=[Integer(5),Integer(5),Integer(5),
↪Integer(5),Integer(5)], labels=labels); G
0  0  0  0  0
0  1  0  0  0
0  0  0  1  0
1  0  0  0  0
0  0  0  0  0
>>> ascii_art([G.P_symbol(), G.Q_symbol()])
[   .  .  2  3    .  .  1  4 ]
[   .  .          .  .       ]
[   .  4          .  2       ]
[   1             3          ]
[   5        ,    5          ]
```

Similarly, there is a correspondence for skew oscillating tableau. Let us conclude by reproducing Example 4.2.6 from [Rob1991]. The oscillating tableau, as given, is:

```
sage: o = [[2,1],[2,2],[3,2],[4,2],[4,1],[4,1,1],[3,1,1],[3,1],[3,2],[3,1],[2,1]]
```

```
>>> from sage.all import *
>>> o = [[Integer(2),Integer(1)],[Integer(2),Integer(2)],[Integer(3),Integer(2)],
↪[Integer(4),Integer(2)],[Integer(4),Integer(1)],[Integer(4),Integer(1),Integer(1)],
↪[Integer(3),Integer(1),Integer(1)],[Integer(3),Integer(1)],[Integer(3),Integer(2)],
↪[Integer(3),Integer(1)],[Integer(2),Integer(1)]]
```

From this, we have to construct the list of labels of the corners along the bottom-right boundary. The labels with odd indices are given by the oscillating tableau, the other labels are obtained by taking the smaller of the two neighbouring partitions:

```
sage: l = [o[i//2] if is_even(i) else min(o[(i-1)//2],o[(i+1)//2])
....:      for i in range(2*len(o)-1)]
sage: la = list(range(len(o)-2, 0, -1))
sage: G = RuleRSK(labels=l[1:-1], shape=la); G
0  0  0  0  0  0  0  1  0
0  1  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0
0  0  1  0  0
```

(continues on next page)

(continued from previous page)

```
0  0  0  0
0  0  0
0  0
0
```

```
>>> from sage.all import *
>>> l = [o[i//Integer(2)] if is_even(i) else min(o[(i-Integer(1))//Integer(2)],
↪o[(i+Integer(1))//Integer(2)])
...      for i in range(Integer(2)*len(o)-Integer(1))]
>>> la = list(range(len(o)-Integer(2), Integer(0), -Integer(1)))
>>> G = RuleRSK(labels=l[Integer(1):-Integer(1)], shape=la); G
0  0  0  0  0  0  0  1  0
0  1  0  0  0  0  0  0
0  0  0  0  0  0  0
0  0  0  0  0  0
0  0  1  0  0
0  0  0  0
0  0  0
0  0
0
```

The skew tableaux can now be read off the partitions labelling the left and the top boundary. These can be accessed using the method *in_labels()*:

```
sage: ascii_art(SkewTableau(chain=G.in_labels()[len(o)-2:]),
....:           SkewTableau(chain=G.in_labels()[len(o)-2::-1]))
.  1  .  7
5     4
```

```
>>> from sage.all import *
>>> ascii_art(SkewTableau(chain=G.in_labels()[len(o)-Integer(2):]),
...           SkewTableau(chain=G.in_labels()[len(o)-Integer(2)::-Integer(1)]))
.  1  .  7
5     4
```

Rules currently available

As mentioned at the beginning, the Robinson-Schensted-Knuth correspondence is just a special case of growth diagrams. In particular, we have implemented the following local rules:

- RSK (*RuleRSK*).
- A variation of RSK originally due to Burge (*RuleBurge*).
- A correspondence producing binary words originally due to Viennot (*RuleBinaryWord*).
- A correspondence producing domino tableaux (*RuleDomino*) originally due to Barbasch and Vogan.
- A correspondence for shifted shapes (*RuleShiftedShapes*), where the original insertion algorithm is due to Sagan and Worley, and Haiman.
- The Sylvester correspondence, producing binary trees (*RuleSylvester*).
- The Young-Fibonacci correspondence (*RuleYoungFibonacci*).
- LLMS insertion (*RuleLLMS*).

Background

At the heart of Fomin's framework is the notion of dual graded graphs. This is a pair of digraphs P, Q (multiple edges being allowed) on the same set of vertices V, that satisfy the following conditions:

- the graphs are graded, that is, there is a function $\rho : V \to \mathbf{N}$, such that for any edge (v, w) of P and also of Q we have $\rho(w) = \rho(v) + 1$,
- there is a vertex 0 with rank zero, and
- there is a positive integer r such that $DU = UD + rI$ on the free $\mathbf{Z}$-module $\mathbf{Z}[V]$, where D is the down operator of Q, assigning to each vertex the formal sum of its predecessors, U is the up operator of P, assigning to each vertex the formal sum of its successors, and I is the identity operator.

Note that the condition $DU = UD + rI$ is symmetric with respect to the interchange of the graphs P and Q, because the up operator of a graph is the transpose of its down operator.

For example, taking for both P and Q to be Young's lattice and $r = 1$, we obtain the dual graded graphs for classical Schensted insertion.

Given such a pair of graphs, there is a bijection between the r-colored permutations on k letters and pairs (p, q), where p is a path in P from zero to a vertex of rank k and q is a path in Q from zero to the same vertex.

It turns out that - in principle - this bijection can always be described by so-called local forward and backward rules, see [Fom1995] for a detailed description. Knowing at least the forward rules, or the backward rules, you can implement your own growth diagram class.

Implementing your own growth diagrams

The class `GrowthDiagram` is written so that it is easy to implement growth diagrams you come across in your research. Moreover, the class tolerates some deviations from Fomin's definitions. For example, although the general Robinson-Schensted-Knuth correspondence between integer matrices and semistandard tableaux is, strictly speaking, not a growth on dual graded graphs, it is supported by our framework.

For illustration, let us implement a growth diagram class with the backward rule only. Suppose that the vertices of the graph are the nonnegative integers, the rank is given by the integer itself, and the backward rule is $(y, z, x) \mapsto (\min(x, y), 0)$ if $y = z$ or $x = z$ and $(y, z, x) \mapsto (\min(x, y), 1)$ otherwise.

We first need to import the base class for a rule:

```
sage: from sage.combinat.growth import Rule
```

```
>>> from sage.all import *
>>> from sage.combinat.growth import Rule
```

Next, we implement the backward rule and the rank function and provide the bottom element `zero` of the graph. For more information, see `Rule`.

```
sage: class RulePascal(Rule):
....:     zero = 0
....:     def rank(self, v): return v
....:     def backward_rule(self, y, z, x):
....:         return (min(x,y), 0 if y==z or x==z else 1)
```

```
>>> from sage.all import *
>>> class RulePascal(Rule):
...     zero = Integer(0)
...     def rank(self, v): return v
```

(continues on next page)

(continued from previous page)

```
...     def backward_rule(self, y, z, x):
...         return (min(x,y), Integer(0) if y==z or x==z else Integer(1))
```

We can now compute the filling corresponding to a sequence of labels as follows:

```
sage: GrowthDiagram(RulePascal(), labels=[0,1,2,1,2,1,0])
1  0  0
0  0  1
0  1
```

```
>>> from sage.all import *
>>> GrowthDiagram(RulePascal(), labels=[Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(1),Integer(0)])
1  0  0
0  0  1
0  1
```

Of course, since we have not provided the forward rule, we cannot compute the labels belonging to a filling:

```
sage: GrowthDiagram(RulePascal(), [3,1,2])
Traceback (most recent call last):
...
AttributeError: 'RulePascal' object has no attribute 'forward_rule'...
```

```
>>> from sage.all import *
>>> GrowthDiagram(RulePascal(), [Integer(3),Integer(1),Integer(2)])
Traceback (most recent call last):
...
AttributeError: 'RulePascal' object has no attribute 'forward_rule'...
```

We now re-implement the rule where we provide the dual graded graphs:

```
sage: class RulePascal(Rule):
....:     zero = 0
....:     def rank(self, v): return v
....:     def backward_rule(self, y, z, x):
....:         return (min(x,y), 0 if y==z or x==z else 1)
....:     def vertices(self, n): return [n]
....:     def is_P_edge(self, v, w): return w == v + 1
....:     def is_Q_edge(self, v, w): return w == v + 1
```

```
>>> from sage.all import *
>>> class RulePascal(Rule):
...     zero = Integer(0)
...     def rank(self, v): return v
...     def backward_rule(self, y, z, x):
...         return (min(x,y), Integer(0) if y==z or x==z else Integer(1))
...     def vertices(self, n): return [n]
...     def is_P_edge(self, v, w): return w == v + Integer(1)
...     def is_Q_edge(self, v, w): return w == v + Integer(1)
```

Are they really dual?

```
sage: RulePascal()._check_duality(3)
Traceback (most recent call last):
...
ValueError: D U - U D differs from 1 I for vertex 3:
D U = [3]
U D + 1 I = [3, 3]
```

```
>>> from sage.all import *
>>> RulePascal()._check_duality(Integer(3))
Traceback (most recent call last):
...
ValueError: D U - U D differs from 1 I for vertex 3:
D U = [3]
U D + 1 I = [3, 3]
```

With our current definition, duality fails - in fact, there are no dual graded graphs on the integers without multiple edges. Consequently, also the backward rule cannot work as `backward_rule` requires additional information (the edge labels as arguments).

Let us thus continue with the example from Section 4.7 of [Fom1995] instead, which defines dual graded graphs with multiple edges on the integers. The color `self.zero_edge`, which defaults to `0` is reserved for degenerate edges, but may be abused for the unique edge if one of the graphs has no multiple edges. For greater clarity in this example we set it to `None`:

```
sage: class RulePascal(Rule):
....:     zero = 0
....:     has_multiple_edges = True
....:     zero_edge = None
....:     def rank(self, v): return v
....:     def vertices(self, n): return [n]
....:     def is_P_edge(self, v, w): return [0] if w == v + 1 else []
....:     def is_Q_edge(self, v, w): return list(range(w)) if w == v+1 else []
```

```
>>> from sage.all import *
>>> class RulePascal(Rule):
...     zero = Integer(0)
...     has_multiple_edges = True
...     zero_edge = None
...     def rank(self, v): return v
...     def vertices(self, n): return [n]
...     def is_P_edge(self, v, w): return [Integer(0)] if w == v + Integer(1) else []
...     def is_Q_edge(self, v, w): return list(range(w)) if w == v+Integer(1) else []
```

We verify these are 1 dual at level 5:

```
sage: RulePascal()._check_duality(5)
```

```
>>> from sage.all import *
>>> RulePascal()._check_duality(Integer(5))
```

Finally, let us provide the backward rule. The arguments of the rule are vertices together with the edge labels now, specifying the path from the lower left to the upper right of the cell. The horizontal edges come from Q, whereas the vertical edges come from P.

Thus, the definition in Section 4.7 of [Fom1995] translates as follows:

```
sage: class RulePascal(Rule):
....:     zero = 0
....:     has_multiple_edges = True
....:     zero_edge = None
....:     def rank(self, v): return v
....:     def vertices(self, n): return [n]
....:     def is_P_edge(self, v, w): return [0] if w == v + 1 else []
....:     def is_Q_edge(self, v, w): return list(range(w)) if w == v+1 else []
....:     def backward_rule(self, y, g, z, h, x):
....:         if g is None:
....:             return (0, x, None, 0)
....:         if h is None:
....:             return (None, y, g, 0)
....:         if g == 0:
....:             return (None, y, None, 1)
....:         else:
....:             return (0, x-1, g-1, 0)
```

```
>>> from sage.all import *
>>> class RulePascal(Rule):
...     zero = Integer(0)
...     has_multiple_edges = True
...     zero_edge = None
...     def rank(self, v): return v
...     def vertices(self, n): return [n]
...     def is_P_edge(self, v, w): return [Integer(0)] if w == v + Integer(1) else []
...     def is_Q_edge(self, v, w): return list(range(w)) if w == v+Integer(1) else []
...     def backward_rule(self, y, g, z, h, x):
...         if g is None:
...             return (Integer(0), x, None, Integer(0))
...         if h is None:
...             return (None, y, g, Integer(0))
...         if g == Integer(0):
...             return (None, y, None, Integer(1))
...         else:
...             return (Integer(0), x-Integer(1), g-Integer(1), Integer(0))
```

The labels are now alternating between vertices and edge-colors:

```
sage: GrowthDiagram(RulePascal(), labels=[0,0,1,0,2,0,1,0,0])
1  0
0  1

sage: GrowthDiagram(RulePascal(), labels=[0,0,1,1,2,0,1,0,0])
0  1
1  0
```

```
>>> from sage.all import *
>>> GrowthDiagram(RulePascal(), labels=[Integer(0),Integer(0),Integer(1),Integer(0),
↪Integer(2),Integer(0),Integer(1),Integer(0),Integer(0)])
1  0
```

(continues on next page)

(continued from previous page)

```
0  1

>>> GrowthDiagram(RulePascal(), labels=[Integer(0),Integer(0),Integer(1),Integer(1),
↪Integer(2),Integer(0),Integer(1),Integer(0),Integer(0)])
0  1
1  0
```

class sage.combinat.growth.**GrowthDiagram**(*rule*, *filling=None*, *shape=None*, *labels=None*)

Bases: SageObject

A generalized Schensted growth diagram in the sense of Fomin.

Growth diagrams were introduced by Sergey Fomin [Fom1994], [Fom1995] and provide a vast generalization of the Robinson-Schensted-Knuth (RSK) correspondence between matrices with nonnegative integer entries and pairs of semistandard Young tableaux of the same shape.

A growth diagram is based on the notion of *dual graded graphs*, a pair of digraphs P, Q (multiple edges being allowed) on the same set of vertices V, that satisfy the following conditions:

- the graphs are graded, that is, there is a function $\rho : V \to \mathbf{N}$, such that for any edge (v, w) of P and also of Q we have $\rho(w) = \rho(v) + 1$,
- there is a vertex 0 with rank zero, and
- there is a positive integer r such that $DU = UD + rI$ on the free $\mathbf{Z}$-module $\mathbf{Z}[V]$, where D is the down operator of Q, assigning to each vertex the formal sum of its predecessors, U is the up operator of P, assigning to each vertex the formal sum of its successors, and I is the identity operator.

Growth diagrams are defined by providing a pair of local rules: a 'forward' rule, whose input are three vertices y, t and x of the dual graded graphs and an integer, and whose output is a fourth vertex z. This rule should be invertible in the following sense: there is a so-called 'backward' rule that recovers the integer and t given y, z and x.

All implemented growth diagram rules are available by `GrowthDiagram.rules.<tab>`. The current list is:

- *RuleRSK* – RSK
- *RuleBurge* – a variation of RSK originally due to Burge
- *RuleBinaryWord* – a correspondence producing binary words originally due to Viennot
- *RuleDomino* – a correspondence producing domino tableaux originally due to Barbasch and Vogan
- *RuleShiftedShapes* – a correspondence for shifted shapes, where the original insertion algorithm is due to Sagan and Worley, and Haiman.
- *RuleSylvester* – the Sylvester correspondence, producing binary trees
- *RuleYoungFibonacci* – the Young-Fibonacci correspondence
- *RuleLLMS* – LLMS insertion

INPUT:

- `rule` – *Rule*; the growth diagram rule
- `filling` – (optional) a dictionary whose keys are coordinates and values are integers, a list of lists of integers, or a word with integer values; if a word, then negative letters but without repetitions are allowed and interpreted as coloured permutations
- `shape` – (optional) a (possibly skew) partition
- `labels` – (optional) a list that specifies a path whose length in the half-perimeter of the shape; more details given below

If `filling` is not given, then the growth diagram is determined by applying the backward rule to `labels` decorating the boundary opposite of the origin of the `shape`. In this case, `labels` are interpreted as labelling the boundary opposite of the origin.

Otherwise, `shape` is inferred from `filling` or `labels` if possible and `labels` is set to `rule.zero` if not specified. Here, `labels` are labelling the boundary on the side of the origin.

For `labels`, if `rule.has_multiple_edges` is `True`, then the elements should be of the form $(v_1, e_1, \ldots, e_{n-1}, v_n)$, where n is the half-perimeter of `shape`, and (v_{i-1}, e_i, v_i) is an edge in the dual graded graph for all i. Otherwise, it is a list of n vertices.

Note

Coordinates are of the form `(col, row)` where the origin is in the upper left, to be consistent with permutation matrices and skew tableaux (in English convention). This is different from Fomin's convention, who uses a Cartesian coordinate system.

Conventions are chosen such that for permutations, the same growth diagram is constructed when passing the permutation matrix instead.

EXAMPLES:

We create a growth diagram using the forward RSK rule and a permutation:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: pi = Permutation([4, 1, 2, 3])
sage: G = GrowthDiagram(RuleRSK, pi); G
0  1  0  0
0  0  1  0
0  0  0  1
1  0  0  0
sage: G.out_labels()
[[], [1], [1, 1], [2, 1], [3, 1], [3], [2], [1], []]
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> pi = Permutation([Integer(4), Integer(1), Integer(2), Integer(3)])
>>> G = GrowthDiagram(RuleRSK, pi); G
0  1  0  0
0  0  1  0
0  0  0  1
1  0  0  0
>>> G.out_labels()
[[], [1], [1, 1], [2, 1], [3, 1], [3], [2], [1], []]
```

Passing the permutation matrix instead gives the same result:

```
sage: G = GrowthDiagram(RuleRSK, pi.to_matrix())                        #␣
↪needs sage.modules
sage: ascii_art([G.P_symbol(), G.Q_symbol()])                           #␣
↪needs sage.modules
[   1  2  3     1  3  4 ]
[   4       ,   2       ]
```

```
>>> from sage.all import *
>>> G = GrowthDiagram(RuleRSK, pi.to_matrix())                              #␣
↪needs sage.modules
>>> ascii_art([G.P_symbol(), G.Q_symbol()])                                 #␣
↪needs sage.modules
[   1  2  3    1  3  4 ]
[   4      ,   2       ]
```

We give the same example but using a skew shape:

```
sage: shape = SkewPartition([[4,4,4,2],[1,1]])
sage: G = GrowthDiagram(RuleRSK, pi, shape=shape); G
.  1  0  0
.  0  1  0
0  0  0  1
1  0
sage: G.out_labels()
[[], [1], [1, 1], [1], [2], [3], [2], [1], []]
```

```
>>> from sage.all import *
>>> shape = SkewPartition([[Integer(4),Integer(4),Integer(4),Integer(2)],
↪[Integer(1),Integer(1)]])
>>> G = GrowthDiagram(RuleRSK, pi, shape=shape); G
.  1  0  0
.  0  1  0
0  0  0  1
1  0
>>> G.out_labels()
[[], [1], [1, 1], [1], [2], [3], [2], [1], []]
```

We construct a growth diagram using the backwards RSK rule by specifying the labels:

```
sage: GrowthDiagram(RuleRSK, labels=G.out_labels())
0  1  0  0
0  0  1  0
0  0  0  1
1  0
```

```
>>> from sage.all import *
>>> GrowthDiagram(RuleRSK, labels=G.out_labels())
0  1  0  0
0  0  1  0
0  0  0  1
1  0
```

P_chain()

Return the labels along the vertical boundary of a rectangular growth diagram.

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: G = GrowthDiagram(BinaryWord, [4, 1, 2, 3])
```

(continues on next page)

(continued from previous page)

```
sage: G.P_chain()
[word: , word: 1, word: 11, word: 111, word: 1011]
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> G = GrowthDiagram(BinaryWord, [Integer(4), Integer(1), Integer(2),␣
↪Integer(3)])
>>> G.P_chain()
[word: , word: 1, word: 11, word: 111, word: 1011]
```

Check that Issue #25631 is fixed:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: BinaryWord(filling = {}).P_chain()
[word: ]
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> BinaryWord(filling = {}).P_chain()
[word: ]
```

P_symbol()

Return the labels along the vertical boundary of a rectangular growth diagram as a generalized standard tableau.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = GrowthDiagram(RuleRSK, [[0,1,0], [1,0,2]])
sage: ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  2    1  3  3 ]
[   2      ,   2       ]
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = GrowthDiagram(RuleRSK, [[Integer(0),Integer(1),Integer(0)],␣
↪[Integer(1),Integer(0),Integer(2)]])
>>> ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  2    1  3  3 ]
[   2      ,   2       ]
```

Q_chain()

Return the labels along the horizontal boundary of a rectangular growth diagram.

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: G = GrowthDiagram(BinaryWord, [[0,1,0,0], [0,0,1,0], [0,0,0,1], [1,0,0,
↪0]])
sage: G.Q_chain()
[word: , word: 1, word: 10, word: 101, word: 1011]
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> G = GrowthDiagram(BinaryWord, [[Integer(0),Integer(1),Integer(0),
↪Integer(0)], [Integer(0),Integer(0),Integer(1),Integer(0)], [Integer(0),
↪Integer(0),Integer(0),Integer(1)], [Integer(1),Integer(0),Integer(0),
↪Integer(0)]])
>>> G.Q_chain()
[word: , word: 1, word: 10, word: 101, word: 1011]
```

Check that Issue #25631 is fixed:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: BinaryWord(filling = {}).Q_chain()
[word: ]
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> BinaryWord(filling = {}).Q_chain()
[word: ]
```

Q_symbol()

Return the labels along the horizontal boundary of a rectangular growth diagram as a generalized standard tableau.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = GrowthDiagram(RuleRSK, [[0,1,0], [1,0,2]])
sage: ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  2     1  3  3 ]
[   2       ,   2       ]
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = GrowthDiagram(RuleRSK, [[Integer(0),Integer(1),Integer(0)],
↪[Integer(1),Integer(0),Integer(2)]])
>>> ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  2     1  3  3 ]
[   2       ,   2       ]
```

conjugate()

Return the conjugate growth diagram of `self`.

This is the growth diagram with the filling reflected over the main diagonal.

The sequence of labels along the boundary on the side of the origin is the reversal of the corresponding sequence of the original growth diagram.

When the filling is a permutation, the conjugate filling corresponds to its inverse.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = GrowthDiagram(RuleRSK, [[0,1,0], [1,0,2]])
sage: Gc = G.conjugate()
```

(continues on next page)

(continued from previous page)

```
sage: (Gc.P_symbol(), Gc.Q_symbol()) == (G.Q_symbol(), G.P_symbol())
True
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = GrowthDiagram(RuleRSK, [[Integer(0),Integer(1),Integer(0)],␣
↪[Integer(1),Integer(0),Integer(2)]])
>>> Gc = G.conjugate()
>>> (Gc.P_symbol(), Gc.Q_symbol()) == (G.Q_symbol(), G.P_symbol())
True
```

filling()

Return the filling of the diagram as a dictionary.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = GrowthDiagram(RuleRSK, [[0,1,0], [1,0,2]])
sage: G.filling()
{(0, 1): 1, (1, 0): 1, (2, 1): 2}
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = GrowthDiagram(RuleRSK, [[Integer(0),Integer(1),Integer(0)],␣
↪[Integer(1),Integer(0),Integer(2)]])
>>> G.filling()
{(0, 1): 1, (1, 0): 1, (2, 1): 2}
```

half_perimeter()

Return half the perimeter of the shape of the growth diagram.

in_labels()

Return the labels along the boundary on the side of the origin.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = GrowthDiagram(RuleRSK, labels=[[2,2],[3,2],[3,3],[3,2]]); G
1 0
sage: G.in_labels()
[[2, 2], [2, 2], [2, 2], [3, 2]]
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = GrowthDiagram(RuleRSK, labels=[[Integer(2),Integer(2)],[Integer(3),
↪Integer(2)],[Integer(3),Integer(3)],[Integer(3),Integer(2)]]); G
1 0
>>> G.in_labels()
[[2, 2], [2, 2], [2, 2], [3, 2]]
```

is_rectangular()

Return `True` if the shape of the growth diagram is rectangular.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: GrowthDiagram(RuleRSK, [2,3,1]).is_rectangular()
True
sage: GrowthDiagram(RuleRSK, [[1,0,1],[0,1]]).is_rectangular()
False
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> GrowthDiagram(RuleRSK, [Integer(2),Integer(3),Integer(1)]).is_
↪rectangular()
True
>>> GrowthDiagram(RuleRSK, [[Integer(1),Integer(0),Integer(1)],[Integer(0),
↪Integer(1)]]).is_rectangular()
False
```

out_labels()

Return the labels along the boundary opposite of the origin.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = GrowthDiagram(RuleRSK, [[0,1,0], [1,0,2]])
sage: G.out_labels()
[[], [1], [1, 1], [3, 1], [1], []]
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = GrowthDiagram(RuleRSK, [[Integer(0),Integer(1),Integer(0)],
↪[Integer(1),Integer(0),Integer(2)]])
>>> G.out_labels()
[[], [1], [1, 1], [3, 1], [1], []]
```

rotate()

Return the growth diagram with the filling rotated by 180 degrees.

The rotated growth diagram is initialized with `labels=None`, that is, all labels along the boundary on the side of the origin are set to `rule.zero`.

For RSK-growth diagrams and rectangular fillings, this corresponds to evacuation of the P- and the Q-symbol.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = GrowthDiagram(RuleRSK, [[0,1,0], [1,0,2]])
sage: Gc = G.rotate()
sage: ascii_art([Gc.P_symbol(), Gc.Q_symbol()])
[   1  1  1    1  1  2 ]
[   2      ,   3       ]

sage: ascii_art([Tableau(t).evacuation()
....:            for t in [G.P_symbol(), G.Q_symbol()]])
[   1  1  1    1  1  2 ]
[   2      ,   3       ]
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = GrowthDiagram(RuleRSK, [[Integer(0),Integer(1),Integer(0)],
→[Integer(1),Integer(0),Integer(2)]])
>>> Gc = G.rotate()
>>> ascii_art([Gc.P_symbol(), Gc.Q_symbol()])
[   1  1  1     1  1  2 ]
[   2       ,   3       ]

>>> ascii_art([Tableau(t).evacuation()
...            for t in [G.P_symbol(), G.Q_symbol()]])
[   1  1  1     1  1  2 ]
[   2       ,   3       ]
```

rules

alias of *Rules*

shape()

Return the shape of the growth diagram as a skew partition.

> **Warning**
>
> In the literature the label on the corner opposite of the origin of a rectangular filling is often called the shape of the filling. This method returns the shape of the region instead.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: GrowthDiagram(RuleRSK, [1]).shape()
[1] / []
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> GrowthDiagram(RuleRSK, [Integer(1)]).shape()
[1] / []
```

to_biword()

Return the filling as a biword, if the shape is rectangular.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: P = Tableau([[1,2,2],[2]])
sage: Q = Tableau([[1,3,3],[2]])
sage: bw = RSK_inverse(P, Q); bw
[[1, 2, 3, 3], [2, 1, 2, 2]]
sage: G = GrowthDiagram(RuleRSK, labels=Q.to_chain()[:-1]+P.to_chain()[::-1]);
→ G
0  1  0
1  0  2

sage: P = SemistandardTableau([[1, 1, 2], [2]])
```

(continues on next page)

(continued from previous page)

```
sage: Q = SemistandardTableau([[1, 2, 2], [2]])
sage: G = GrowthDiagram(RuleRSK, labels=Q.to_chain()[:-1]+P.to_chain()[::-1]);
↪ G
0  2
1  1
sage: G.to_biword()
([1, 2, 2, 2], [2, 1, 1, 2])
sage: RSK([1, 2, 2, 2], [2, 1, 1, 2])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> P = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(2)]])
>>> Q = Tableau([[Integer(1),Integer(3),Integer(3)],[Integer(2)]])
>>> bw = RSK_inverse(P, Q); bw
[[1, 2, 3, 3], [2, 1, 2, 2]]
>>> G = GrowthDiagram(RuleRSK, labels=Q.to_chain()[:-Integer(1)]+P.to_
↪chain()[::-Integer(1)]); G
0  1  0
1  0  2

>>> P = SemistandardTableau([[Integer(1), Integer(1), Integer(2)],␣
↪[Integer(2)]])
>>> Q = SemistandardTableau([[Integer(1), Integer(2), Integer(2)],␣
↪[Integer(2)]])
>>> G = GrowthDiagram(RuleRSK, labels=Q.to_chain()[:-Integer(1)]+P.to_
↪chain()[::-Integer(1)]); G
0  2
1  1
>>> G.to_biword()
([1, 2, 2, 2], [2, 1, 1, 2])
>>> RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2),␣
↪Integer(1), Integer(1), Integer(2)])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

to_word()

Return the filling as a word, if the shape is rectangular and there is at most one nonzero entry in each column, which must be 1.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: w = [3,3,2,4,1]; G = GrowthDiagram(RuleRSK, w)
sage: G
0  0  0  0  1
0  0  1  0  0
1  1  0  0  0
0  0  0  1  0
sage: G.to_word()
[3, 3, 2, 4, 1]
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> w = [Integer(3),Integer(3),Integer(2),Integer(4),Integer(1)]; G =␣
↪GrowthDiagram(RuleRSK, w)
>>> G
0  0  0  0  1
0  0  1  0  0
1  1  0  0  0
0  0  0  1  0
>>> G.to_word()
[3, 3, 2, 4, 1]
```

class sage.combinat.growth.**Rule**

Bases: UniqueRepresentation

Generic base class for a rule for a growth diagram.

Subclasses may provide the following attributes:

- zero – the zero element of the vertices of the graphs
- r – (default: 1) the parameter in the equation $DU - UD = rI$
- has_multiple_edges – boolean (default: False); if the dual graded graph has multiple edges and therefore edges are triples consisting of two vertices and a label
- zero_edge – (default: 0) the zero label of the edges of the graphs used for degenerate edges. It is allowed to use this label also for other edges.

Subclasses may provide the following methods:

- normalize_vertex – a function that converts its input to a vertex
- vertices – a function that takes a nonnegative integer as input and returns the list of vertices on this rank
- rank – the rank function of the dual graded graphs
- forward_rule – a function with input (y, t, x, content) or (y, e, t, f, x, content) if has_multiple_edges is True. (y, e, t) is an edge in the graph P, (t, f, x) an edge in the graph Q. It should return the fourth vertex z, or, if has_multiple_edges is True, the path (g, z, h) from y to x.
- backward_rule – a function with input (y, z, x) or (y, g, z, h, x) if has_multiple_edges is True. (y, g, z) is an edge in the graph Q, (z, h, x) an edge in the graph P. It should return the fourth vertex and the content (t, content), or, if has_multiple_edges is True, the path from y to x and the content as (e, t, f, content).
- is_P_edge, is_Q_edge – functions that take two vertices as arguments and return True or False, or, if multiple edges are allowed, the list of edge labels of the edges from the first vertex to the second in the respective graded graph. These are only used for checking user input and providing the dual graded graph, and are therefore not mandatory.

Note that the class *GrowthDiagram* is able to use partially implemented subclasses just fine. Suppose that MyRule is such a subclass. Then:

- GrowthDiagram(MyRule, my_filling) requires only an implementation of forward_rule, zero and possibly has_multiple_edges.
- GrowthDiagram(MyRule, labels=my_labels, shape=my_shape) requires only an implementation of backward_rule and possibly has_multiple_edges, provided that the labels my_labels are given as needed by backward_rule.

- `GrowthDiagram(MyRule, labels=my_labels)` additionally needs an implementation of `rank` to deduce the shape.

In particular, this allows to implement rules which do not quite fit Fomin's notion of dual graded graphs. An example would be Bloom and Saracino's variant of the RSK correspondence [BS2012], where a backward rule is not available.

Similarly:

- `MyRule.P_graph` only requires an implementation of `vertices`, `is_P_edge` and possibly `has_multiple_edges` is required, mutatis mutandis for `MyRule.Q_graph`.
- `MyRule._check_duality` requires `P_graph` and `Q_graph`.

In particular, this allows to work with dual graded graphs without local rules.

P_graph(*n*)

Return the first `n` levels of the first dual graded graph.

The non-degenerate edges in the vertical direction come from this graph.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: Domino.P_graph(3)
Finite poset containing 8 elements
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> Domino.P_graph(Integer(3))
Finite poset containing 8 elements
```

Q_graph(*n*)

Return the first `n` levels of the second dual graded graph.

The non-degenerate edges in the horizontal direction come from this graph.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: Q = Domino.Q_graph(3); Q
Finite poset containing 8 elements

sage: Q.upper_covers(Partition([1,1]))
[[1, 1, 1, 1], [3, 1], [2, 2]]
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> Q = Domino.Q_graph(Integer(3)); Q
Finite poset containing 8 elements

>>> Q.upper_covers(Partition([Integer(1),Integer(1)]))
[[1, 1, 1, 1], [3, 1], [2, 2]]
```

has_multiple_edges = False

normalize_vertex(*v*)

Return v as a vertex of the dual graded graph.

This is a default implementation, returning its argument.

EXAMPLES:

```
sage: from sage.combinat.growth import Rule
sage: Rule().normalize_vertex("hello") == "hello"
True
```

```
>>> from sage.all import *
>>> from sage.combinat.growth import Rule
>>> Rule().normalize_vertex("hello") == "hello"
True
```

r = 1

zero_edge = 0

class sage.combinat.growth.**RuleBinaryWord**

Bases: *Rule*

A rule modelling a Schensted-like correspondence for binary words.

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: GrowthDiagram(BinaryWord, [3,1,2])
0  1  0
0  0  1
1  0  0
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> GrowthDiagram(BinaryWord, [Integer(3),Integer(1),Integer(2)])
0  1  0
0  0  1
1  0  0
```

The vertices of the dual graded graph are binary words:

```
sage: BinaryWord.vertices(3)
[word: 100, word: 101, word: 110, word: 111]
```

```
>>> from sage.all import *
>>> BinaryWord.vertices(Integer(3))
[word: 100, word: 101, word: 110, word: 111]
```

Note that, instead of passing the rule to *GrowthDiagram*, we can also use call the rule to create growth diagrams. For example:

```
sage: BinaryWord([2,4,1,3]).P_chain()
[word: , word: 1, word: 10, word: 101, word: 1101]
sage: BinaryWord([2,4,1,3]).Q_chain()
[word: , word: 1, word: 11, word: 110, word: 1101]
```

```
>>> from sage.all import *
>>> BinaryWord([Integer(2),Integer(4),Integer(1),Integer(3)]).P_chain()
[word: , word: 1, word: 10, word: 101, word: 1101]
>>> BinaryWord([Integer(2),Integer(4),Integer(1),Integer(3)]).Q_chain()
[word: , word: 1, word: 11, word: 110, word: 1101]
```

The Kleitman Greene invariant is the descent word, encoded by the positions of the zeros:

```
sage: pi = Permutation([4,1,8,3,6,5,2,7,9])
sage: G = BinaryWord(pi); G
0  1  0  0  0  0  0  0  0
0  0  0  0  0  0  1  0  0
0  0  0  1  0  0  0  0  0
1  0  0  0  0  0  0  0  0
0  0  0  0  0  1  0  0  0
0  0  0  0  1  0  0  0  0
0  0  0  0  0  0  0  1  0
0  0  1  0  0  0  0  0  0
0  0  0  0  0  0  0  0  1
sage: pi.descents()
[1, 3, 5, 6]
```

```
>>> from sage.all import *
>>> pi = Permutation([Integer(4),Integer(1),Integer(8),Integer(3),Integer(6),
↪Integer(5),Integer(2),Integer(7),Integer(9)])
>>> G = BinaryWord(pi); G
0  1  0  0  0  0  0  0  0
0  0  0  0  0  0  1  0  0
0  0  0  1  0  0  0  0  0
1  0  0  0  0  0  0  0  0
0  0  0  0  0  1  0  0  0
0  0  0  0  1  0  0  0  0
0  0  0  0  0  0  0  1  0
0  0  1  0  0  0  0  0  0
0  0  0  0  0  0  0  0  1
>>> pi.descents()
[1, 3, 5, 6]
```

backward_rule(y, z, x)

Return the content and the input shape.

See [Fom1995] Lemma 4.6.1, page 40.

- `y, z, x` – three binary words from a cell in a growth diagram, labelled as:

```
  x
y z
```

OUTPUT:

A pair `(t, content)` consisting of the shape of the fourth word and the content of the cell according to Viennot's bijection [Vie1983].

forward_rule(y, t, x, *content*)

Return the output shape given three shapes and the content.

See [Fom1995] Lemma 4.6.1, page 40.

INPUT:

- y, t, x – three binary words from a cell in a growth diagram, labelled as:

```
t x
y
```

- content – 0 or 1; the content of the cell

OUTPUT:

The fourth binary word z according to Viennot's bijection [Vie1983].

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()

sage: BinaryWord.forward_rule([], [], [], 1)
word: 1

sage: BinaryWord.forward_rule([1], [1], [1], 1)
word: 11
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()

>>> BinaryWord.forward_rule([], [], [], Integer(1))
word: 1

>>> BinaryWord.forward_rule([Integer(1)], [Integer(1)], [Integer(1)],␣
↪Integer(1))
word: 11
```

if x != y append last letter of x to y:

```
sage: BinaryWord.forward_rule([1,0], [1], [1,1], 0)
word: 101
```

```
>>> from sage.all import *
>>> BinaryWord.forward_rule([Integer(1),Integer(0)], [Integer(1)],␣
↪[Integer(1),Integer(1)], Integer(0))
word: 101
```

if x == y != t append 0 to y:

```
sage: BinaryWord.forward_rule([1,1], [1], [1,1], 0)
word: 110
```

```
>>> from sage.all import *
>>> BinaryWord.forward_rule([Integer(1),Integer(1)], [Integer(1)],␣
↪[Integer(1),Integer(1)], Integer(0))
word: 110
```

is_P_edge(*v*, *w*)

Return whether `(v, w)` is a P-edge of `self`.

`(v, w)` is an edge if `v` is obtained from `w` by deleting a letter.

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: v = BinaryWord.vertices(2)[1]; v
word: 11
sage: [w for w in BinaryWord.vertices(3) if BinaryWord.is_P_edge(v, w)]
[word: 101, word: 110, word: 111]
sage: [w for w in BinaryWord.vertices(4) if BinaryWord.is_P_edge(v, w)]
[]
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> v = BinaryWord.vertices(Integer(2))[Integer(1)]; v
word: 11
>>> [w for w in BinaryWord.vertices(Integer(3)) if BinaryWord.is_P_edge(v, w)]
[word: 101, word: 110, word: 111]
>>> [w for w in BinaryWord.vertices(Integer(4)) if BinaryWord.is_P_edge(v, w)]
[]
```

is_Q_edge(*v*, *w*)

Return whether `(v, w)` is a Q-edge of `self`.

`(w, v)` is an edge if `w` is obtained from `v` by appending a letter.

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: v = BinaryWord.vertices(2)[0]; v
word: 10
sage: [w for w in BinaryWord.vertices(3) if BinaryWord.is_Q_edge(v, w)]
[word: 100, word: 101]
sage: [w for w in BinaryWord.vertices(4) if BinaryWord.is_Q_edge(v, w)]
[]
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> v = BinaryWord.vertices(Integer(2))[Integer(0)]; v
word: 10
>>> [w for w in BinaryWord.vertices(Integer(3)) if BinaryWord.is_Q_edge(v, w)]
[word: 100, word: 101]
>>> [w for w in BinaryWord.vertices(Integer(4)) if BinaryWord.is_Q_edge(v, w)]
[]
```

normalize_vertex(*v*)

Return `v` as a binary word.

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: BinaryWord.normalize_vertex([0,1]).parent()
Finite words over {0, 1}
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> BinaryWord.normalize_vertex([Integer(0),Integer(1)]).parent()
Finite words over {0, 1}
```

rank(*v*)

Return the rank of `v`: number of letters of the word.

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: BinaryWord.rank(BinaryWord.vertices(3)[0])
3
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> BinaryWord.rank(BinaryWord.vertices(Integer(3))[Integer(0)])
3
```

vertices(*n*)

Return the vertices of the dual graded graph on level `n`.

EXAMPLES:

```
sage: BinaryWord = GrowthDiagram.rules.BinaryWord()
sage: BinaryWord.vertices(3)
[word: 100, word: 101, word: 110, word: 111]
```

```
>>> from sage.all import *
>>> BinaryWord = GrowthDiagram.rules.BinaryWord()
>>> BinaryWord.vertices(Integer(3))
[word: 100, word: 101, word: 110, word: 111]
```

zero = word:

class sage.combinat.growth.**RuleBurge**

Bases: *RulePartitions*

A rule modelling Burge insertion.

EXAMPLES:

```
sage: Burge = GrowthDiagram.rules.Burge()
sage: GrowthDiagram(Burge, labels=[[],[1,1,1],[2,1,1,1],[2,1,1],[2,1],[1,1],[]])
1  1
0  1
1  0
1  0
```

```
>>> from sage.all import *
>>> Burge = GrowthDiagram.rules.Burge()
>>> GrowthDiagram(Burge, labels=[[],[Integer(1),Integer(1),Integer(1)],
↪[Integer(2),Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(1),
↪Integer(1)],[Integer(2),Integer(1)],[Integer(1),Integer(1)],[]])
```

(continues on next page)

(continued from previous page)

```
1  1
0  1
1  0
1  0
```

The vertices of the dual graded graph are integer partitions:

```
sage: Burge.vertices(3)
Partitions of the integer 3
```

```
>>> from sage.all import *
>>> Burge.vertices(Integer(3))
Partitions of the integer 3
```

The local rules implemented provide Burge's correspondence between matrices with nonnegative integer entries and pairs of semistandard tableaux, the `P_symbol()` and the `Q_symbol()`. For permutations, it reduces to classical Schensted insertion.

Instead of passing the rule to `GrowthDiagram`, we can also call the rule to create growth diagrams. For example:

```
sage: m = matrix([[2,0,0,1,0],[1,1,0,0,0], [0,0,0,0,3]])
sage: G = Burge(m); G
2  0  0  1  0
1  1  0  0  0
0  0  0  0  3

sage: ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  3     1  2  5 ]
[   1  3        1  5    ]
[   1  3        1  5    ]
[   2       ,   4       ]
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(2),Integer(0),Integer(0),Integer(1),Integer(0)],
↪[Integer(1),Integer(1),Integer(0),Integer(0),Integer(0)], [Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(3)]])
>>> G = Burge(m); G
2  0  0  1  0
1  1  0  0  0
0  0  0  0  3

>>> ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  3     1  2  5 ]
[   1  3        1  5    ]
[   1  3        1  5    ]
[   2       ,   4       ]
```

For rectangular fillings, the Kleitman-Greene invariant is the shape of the `P_symbol()`. Put differently, it is the partition labelling the lower right corner of the filling (recall that we are using matrix coordinates). It can be computed alternatively as the transpose of the partition $(\mu_1, \dots, \mu_n)$, where $\mu_1 + \dots + \mu_i$ is the maximal sum of entries in a collection of i pairwise disjoint sequences of cells with weakly decreasing row indices and weakly increasing column indices.

backward_rule(*y*, *z*, *x*)

Return the content and the input shape.

See [Kra2006] $(B^4 0) - (B^4 2)$. (In the arXiv version of the article there is a typo: in the computation of carry in $(B^4 2)$, ρ must be replaced by λ).

INPUT:

- `y`, `z`, `x` – three partitions from a cell in a growth diagram, labelled as:

```
  x
y z
```

OUTPUT:

A pair `(t, content)` consisting of the shape of the fourth partition according to the Burge correspondence and the content of the cell.

EXAMPLES:

```
sage: Burge = GrowthDiagram.rules.Burge()
sage: Burge.backward_rule([1,1,1],[2,1,1,1],[2,1,1])
([1, 1], 0)
```

```
>>> from sage.all import *
>>> Burge = GrowthDiagram.rules.Burge()
>>> Burge.backward_rule([Integer(1),Integer(1),Integer(1)],[Integer(2),
↪Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(1),Integer(1)])
([1, 1], 0)
```

forward_rule(*y*, *t*, *x*, *content*)

Return the output shape given three shapes and the content.

See [Kra2006] $(F^4 0) - (F^4 2)$.

INPUT:

- `y`, `t`, `x` – three from a cell in a growth diagram, labelled as:

```
t x
y
```

- `content` – nonnegative integer; the content of the cell

OUTPUT: the fourth partition according to the Burge correspondence

EXAMPLES:

```
sage: Burge = GrowthDiagram.rules.Burge()
sage: Burge.forward_rule([2,1],[2,1],[2,1],1)
[3, 1]

sage: Burge.forward_rule([1],[],[2],2)
[2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Burge = GrowthDiagram.rules.Burge()
>>> Burge.forward_rule([Integer(2),Integer(1)],[Integer(2),Integer(1)],
```

(continues on next page)

(continued from previous page)

```
↪[Integer(2),Integer(1)],Integer(1))
[3, 1]

>>> Burge.forward_rule([Integer(1)],[],[Integer(2)],Integer(2))
[2, 1, 1, 1]
```

class sage.combinat.growth.**RuleDomino**

Bases: *Rule*

A rule modelling domino insertion.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: GrowthDiagram(Domino, [[1,0,0],[0,0,1],[0,-1,0]])
1  0  0
0  0  1
0 -1  0
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> GrowthDiagram(Domino, [[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(0),-Integer(1),Integer(0)]])
1  0  0
0  0  1
0 -1  0
```

The vertices of the dual graded graph are integer partitions whose Ferrers diagram can be tiled with dominoes:

```
sage: Domino.vertices(2)
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Domino.vertices(Integer(2))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

Instead of passing the rule to *GrowthDiagram*, we can also call the rule to create growth diagrams. For example, let us check Figure 3 in [Lam2004]:

```
sage: G = Domino([[0,0,0,-1],[0,0,1,0],[-1,0,0,0],[0,1,0,0]]); G
 0  0  0 -1
 0  0  1  0
-1  0  0  0
 0  1  0  0

sage: ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  4     1  2  2 ]
[   1  2  4     1  3  3 ]
[   3  3   ,    4  4    ]
```

```
>>> from sage.all import *
>>> G = Domino([[Integer(0),Integer(0),Integer(0),-Integer(1)],[Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0),Integer(1),Integer(0)],[-Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)]]); G
 0  0  0 -1
 0  0  1  0
-1  0  0  0
 0  1  0  0

>>> ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  4     1  2  2 ]
[   1  2  4     1  3  3 ]
[   3  3   ,    4  4    ]
```

The spin of a domino tableau is half the number of vertical dominoes:

```
sage: def spin(T):
....:     return sum(2*len(set(row)) - len(row) for row in T)/4
```

```
>>> from sage.all import *
>>> def spin(T):
...     return sum(Integer(2)*len(set(row)) - len(row) for row in T)/Integer(4)
```

According to [Lam2004], the number of negative entries in the signed permutation equals the sum of the spins of the two associated tableaux:

```
sage: pi = [3,-1,2,4,-5]
sage: G = Domino(pi)
sage: list(G.filling().values()).count(-1) == spin(G.P_symbol()) + spin(G.Q_
↪symbol())
True
```

```
>>> from sage.all import *
>>> pi = [Integer(3),-Integer(1),Integer(2),Integer(4),-Integer(5)]
>>> G = Domino(pi)
>>> list(G.filling().values()).count(-Integer(1)) == spin(G.P_symbol()) + spin(G.
↪Q_symbol())
True
```

Negating all signs transposes all the partitions:

```
sage: G.P_symbol() == Domino([-e for e in pi]).P_symbol().conjugate()
True
```

```
>>> from sage.all import *
>>> G.P_symbol() == Domino([-e for e in pi]).P_symbol().conjugate()
True
```

P_symbol(*P_chain*)

Return the labels along the vertical boundary of a rectangular growth diagram as a (skew) domino tableau.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: G = Domino([[0,1,0],[0,0,-1],[1,0,0]])
sage: G.P_symbol().pp()
1  1
2  3
2  3
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> G = Domino([[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),-
↪Integer(1)],[Integer(1),Integer(0),Integer(0)]])
>>> G.P_symbol().pp()
1  1
2  3
2  3
```

Q_symbol(*P_chain*)

Return the labels along the vertical boundary of a rectangular growth diagram as a (skew) domino tableau.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: G = Domino([[0,1,0],[0,0,-1],[1,0,0]])
sage: G.P_symbol().pp()
1  1
2  3
2  3
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> G = Domino([[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),-
↪Integer(1)],[Integer(1),Integer(0),Integer(0)]])
>>> G.P_symbol().pp()
1  1
2  3
2  3
```

forward_rule(*y*, *t*, *x*, *content*)

Return the output shape given three shapes and the content.

See [Lam2004] Section 3.1.

INPUT:

- `y`, `t`, `x` – three partitions from a cell in a growth diagram, labelled as:

```
t  x
y
```

- `content` – -1, 0 or 1; the content of the cell

OUTPUT: the fourth partition according to domino insertion

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
```

Rule 1:

```
sage: Domino.forward_rule([], [], [], 1)
[2]

sage: Domino.forward_rule([1,1], [1,1], [1,1], 1)
[3, 1]
```

```
>>> from sage.all import *
>>> Domino.forward_rule([], [], [], Integer(1))
[2]

>>> Domino.forward_rule([Integer(1),Integer(1)], [Integer(1),Integer(1)],␣
↪[Integer(1),Integer(1)], Integer(1))
[3, 1]
```

Rule 2:

```
sage: Domino.forward_rule([1,1], [1,1], [1,1], -1)
[1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Domino.forward_rule([Integer(1),Integer(1)], [Integer(1),Integer(1)],␣
↪[Integer(1),Integer(1)], -Integer(1))
[1, 1, 1, 1]
```

Rule 3:

```
sage: Domino.forward_rule([1,1], [1,1], [2,2], 0)
[2, 2]
```

```
>>> from sage.all import *
>>> Domino.forward_rule([Integer(1),Integer(1)], [Integer(1),Integer(1)],␣
↪[Integer(2),Integer(2)], Integer(0))
[2, 2]
```

Rule 4:

```
sage: Domino.forward_rule([2,2,2], [2,2], [3,3], 0)
[3, 3, 2]

sage: Domino.forward_rule([2], [], [1,1], 0)
[2, 2]

sage: Domino.forward_rule([1,1], [], [2], 0)
[2, 2]

```

(continues on next page)

(continued from previous page)

```
sage: Domino.forward_rule([2], [], [2], 0)
[2, 2]

sage: Domino.forward_rule([4], [2], [4], 0)
[4, 2]

sage: Domino.forward_rule([1,1,1,1], [1,1], [1,1,1,1], 0)
[2, 2, 1, 1]

sage: Domino.forward_rule([2,1,1], [2], [4], 0)
[4, 1, 1]
```

```
>>> from sage.all import *
>>> Domino.forward_rule([Integer(2),Integer(2),Integer(2)], [Integer(2),
↪Integer(2)], [Integer(3),Integer(3)], Integer(0))
[3, 3, 2]

>>> Domino.forward_rule([Integer(2)], [], [Integer(1),Integer(1)], Integer(0))
[2, 2]

>>> Domino.forward_rule([Integer(1),Integer(1)], [], [Integer(2)], Integer(0))
[2, 2]

>>> Domino.forward_rule([Integer(2)], [], [Integer(2)], Integer(0))
[2, 2]

>>> Domino.forward_rule([Integer(4)], [Integer(2)], [Integer(4)], Integer(0))
[4, 2]

>>> Domino.forward_rule([Integer(1),Integer(1),Integer(1),Integer(1)],␣
↪[Integer(1),Integer(1)], [Integer(1),Integer(1),Integer(1),Integer(1)],␣
↪Integer(0))
[2, 2, 1, 1]

>>> Domino.forward_rule([Integer(2),Integer(1),Integer(1)], [Integer(2)],␣
↪[Integer(4)], Integer(0))
[4, 1, 1]
```

is_P_edge(*v*, *w*)

Return whether `(v, w)` is a P-edge of `self`.

`(v, w)` is an edge if `v` is obtained from `w` by deleting a domino.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: v = Domino.vertices(2)[1]; ascii_art(v)
***
*
sage: ascii_art([w for w in Domino.vertices(3) if Domino.is_P_edge(v, w)])
[              *** ]
[              *   ]
[ *****   ***  *   ]
```

(continues on next page)

(continued from previous page)

```
[ *     , ***, *    ]
sage: [w for w in Domino.vertices(4) if Domino.is_P_edge(v, w)]
[]
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> v = Domino.vertices(Integer(2))[Integer(1)]; ascii_art(v)
***
*
>>> ascii_art([w for w in Domino.vertices(Integer(3)) if Domino.is_P_edge(v,␣
↪w)])
[               *** ]
[               *   ]
[ *****   ***   *   ]
[ *     , ***, *    ]
>>> [w for w in Domino.vertices(Integer(4)) if Domino.is_P_edge(v, w)]
[]
```

is_Q_edge(*v*, *w*)

Return whether `(v, w)` is a P-edge of `self`.

`(v, w)` is an edge if `v` is obtained from `w` by deleting a domino.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: v = Domino.vertices(2)[1]; ascii_art(v)
***
*
sage: ascii_art([w for w in Domino.vertices(3) if Domino.is_P_edge(v, w)])
[               *** ]
[               *   ]
[ *****   ***   *   ]
[ *     , ***, *    ]
sage: [w for w in Domino.vertices(4) if Domino.is_P_edge(v, w)]
[]
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> v = Domino.vertices(Integer(2))[Integer(1)]; ascii_art(v)
***
*
>>> ascii_art([w for w in Domino.vertices(Integer(3)) if Domino.is_P_edge(v,␣
↪w)])
[               *** ]
[               *   ]
[ *****   ***   *   ]
[ *     , ***, *    ]
>>> [w for w in Domino.vertices(Integer(4)) if Domino.is_P_edge(v, w)]
[]
```

normalize_vertex(*v*)

Return `v` as a partition.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: Domino.normalize_vertex([3,1]).parent()
Partitions
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> Domino.normalize_vertex([Integer(3),Integer(1)]).parent()
Partitions
```

r = 2

rank(*v*)

Return the rank of `v`.

The rank of a vertex is half the size of the partition, which equals the number of dominoes in any filling.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: Domino.rank(Domino.vertices(3)[0])
3
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> Domino.rank(Domino.vertices(Integer(3))[Integer(0)])
3
```

vertices(*n*)

Return the vertices of the dual graded graph on level `n`.

EXAMPLES:

```
sage: Domino = GrowthDiagram.rules.Domino()
sage: Domino.vertices(2)
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Domino = GrowthDiagram.rules.Domino()
>>> Domino.vertices(Integer(2))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

zero = []

class sage.combinat.growth.**RuleLLMS**(*k*)

Bases: `Rule`

A rule modelling the Schensted correspondence for affine permutations.

EXAMPLES:

```
sage: LLMS3 = GrowthDiagram.rules.LLMS(3)
sage: GrowthDiagram(LLMS3, [3,1,2])
0  1  0
```

(continues on next page)

(continued from previous page)

```
0  0  1
1  0  0
```

```
>>> from sage.all import *
>>> LLMS3 = GrowthDiagram.rules.LLMS(Integer(3))
>>> GrowthDiagram(LLMS3, [Integer(3),Integer(1),Integer(2)])
0  1  0
0  0  1
1  0  0
```

The vertices of the dual graded graph are `Cores`:

```
sage: LLMS3.vertices(4)
3-Cores of length 4
```

```
>>> from sage.all import *
>>> LLMS3.vertices(Integer(4))
3-Cores of length 4
```

Let us check example of Figure 1 in [LS2007]. Note that, instead of passing the rule to `GrowthDiagram`, we can also call the rule to create growth diagrams:

```
sage: G = LLMS3([4,1,2,6,3,5]); G
0  1  0  0  0  0
0  0  1  0  0  0
0  0  0  0  1  0
1  0  0  0  0  0
0  0  0  0  0  1
0  0  0  1  0  0
```

```
>>> from sage.all import *
>>> G = LLMS3([Integer(4),Integer(1),Integer(2),Integer(6),Integer(3),
↪Integer(5)]); G
0  1  0  0  0  0
0  0  1  0  0  0
0  0  0  0  1  0
1  0  0  0  0  0
0  0  0  0  0  1
0  0  0  1  0  0
```

The `P_symbol()` is a `StrongTableau`:

```
sage: G.P_symbol().pp()
-1 -2 -3 -5
 3  5
-4 -6
 5
 6
```

```
>>> from sage.all import *
>>> G.P_symbol().pp()
```

(continues on next page)

(continued from previous page)

```
-1 -2 -3 -5
 3  5
-4 -6
 5
 6
```

The `Q_symbol()` is a `WeakTableau`:

```
sage: G.Q_symbol().pp()
1  3  4  5
2  5
3  6
5
6
```

```
>>> from sage.all import *
>>> G.Q_symbol().pp()
1  3  4  5
2  5
3  6
5
6
```

Let us also check Example 6.2 in [LLMSSZ2013]:

```
sage: G = LLMS3([4,1,3,2])
sage: G.P_symbol().pp()
-1 -2  3
-3
-4

sage: G.Q_symbol().pp()
1  3  4
2
3
```

```
>>> from sage.all import *
>>> G = LLMS3([Integer(4),Integer(1),Integer(3),Integer(2)])
>>> G.P_symbol().pp()
-1 -2  3
-3
-4

>>> G.Q_symbol().pp()
1  3  4
2
3
```

P_symbol(*P_chain*)

Return the labels along the vertical boundary of a rectangular growth diagram as a skew `StrongTableau`.

EXAMPLES:

```
sage: LLMS4 = GrowthDiagram.rules.LLMS(4)
sage: G = LLMS4([3,4,1,2])
sage: G.P_symbol().pp()
-1 -2
-3 -4
```

```
>>> from sage.all import *
>>> LLMS4 = GrowthDiagram.rules.LLMS(Integer(4))
>>> G = LLMS4([Integer(3),Integer(4),Integer(1),Integer(2)])
>>> G.P_symbol().pp()
-1 -2
-3 -4
```

Q_symbol(*Q_chain*)

Return the labels along the horizontal boundary of a rectangular growth diagram as a skew `WeakTableau`.

EXAMPLES:

```
sage: LLMS4 = GrowthDiagram.rules.LLMS(4)
sage: G = LLMS4([3,4,1,2])
sage: G.Q_symbol().pp()
1 2
3 4
```

```
>>> from sage.all import *
>>> LLMS4 = GrowthDiagram.rules.LLMS(Integer(4))
>>> G = LLMS4([Integer(3),Integer(4),Integer(1),Integer(2)])
>>> G.Q_symbol().pp()
1 2
3 4
```

forward_rule(*y*, *e*, *t*, *f*, *x*, *content*)

Return the output path given two incident edges and the content.

See [LS2007] Section 3.4 and [LLMSSZ2013] Section 6.3.

INPUT:

- `y, e, t, f, x` – a path of three partitions and two colors from a cell in a growth diagram, labelled as:

```
t f x
e
y
```

- `content` – 0 or 1; the content of the cell

OUTPUT:

The two colors and the fourth partition g, z, h according to LLMS insertion.

EXAMPLES:

```
sage: LLMS3 = GrowthDiagram.rules.LLMS(3)
sage: LLMS4 = GrowthDiagram.rules.LLMS(4)

sage: Z = LLMS3.zero
```

(continues on next page)

(continued from previous page)

```
sage: LLMS3.forward_rule(Z, None, Z, None, Z, 0)
(None, [], None)

sage: LLMS3.forward_rule(Z, None, Z, None, Z, 1)
(None, [1], 0)

sage: Y = Core([3,1,1], 3)
sage: LLMS3.forward_rule(Y, None, Y, None, Y, 1)
(None, [4, 2, 1, 1], 3)
```

```
>>> from sage.all import *
>>> LLMS3 = GrowthDiagram.rules.LLMS(Integer(3))
>>> LLMS4 = GrowthDiagram.rules.LLMS(Integer(4))

>>> Z = LLMS3.zero
>>> LLMS3.forward_rule(Z, None, Z, None, Z, Integer(0))
(None, [], None)

>>> LLMS3.forward_rule(Z, None, Z, None, Z, Integer(1))
(None, [1], 0)

>>> Y = Core([Integer(3),Integer(1),Integer(1)], Integer(3))
>>> LLMS3.forward_rule(Y, None, Y, None, Y, Integer(1))
(None, [4, 2, 1, 1], 3)
```

if `x != y`:

```
sage: Y = Core([1,1], 3); T = Core([1], 3); X = Core([2], 3)
sage: LLMS3.forward_rule(Y, -1, T, None, X, 0)
(None, [2, 1, 1], -1)

sage: Y = Core([2], 4); T = Core([1], 4); X = Core([1,1], 4)
sage: LLMS4.forward_rule(Y, 1, T, None, X, 0)
(None, [2, 1], 1)

sage: Y = Core([2,1,1], 3); T = Core([2], 3); X = Core([3,1], 3)
sage: LLMS3.forward_rule(Y, -1, T, None, X, 0)
(None, [3, 1, 1], -2)
```

```
>>> from sage.all import *
>>> Y = Core([Integer(1),Integer(1)], Integer(3)); T = Core([Integer(1)],␣
↪Integer(3)); X = Core([Integer(2)], Integer(3))
>>> LLMS3.forward_rule(Y, -Integer(1), T, None, X, Integer(0))
(None, [2, 1, 1], -1)

>>> Y = Core([Integer(2)], Integer(4)); T = Core([Integer(1)], Integer(4)); X␣
↪= Core([Integer(1),Integer(1)], Integer(4))
>>> LLMS4.forward_rule(Y, Integer(1), T, None, X, Integer(0))
(None, [2, 1], 1)

>>> Y = Core([Integer(2),Integer(1),Integer(1)], Integer(3)); T =␣
↪Core([Integer(2)], Integer(3)); X = Core([Integer(3),Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(3))
>>> LLMS3.forward_rule(Y, -Integer(1), T, None, X, Integer(0))
(None, [3, 1, 1], -2)
```

if `x == y != t`:

```
sage: Y = Core([1], 3); T = Core([], 3); X = Core([1], 3)
sage: LLMS3.forward_rule(Y, 0, T, None, X, 0)
(None, [1, 1], -1)

sage: Y = Core([1], 4); T = Core([], 4); X = Core([1], 4)
sage: LLMS4.forward_rule(Y, 0, T, None, X, 0)
(None, [1, 1], -1)

sage: Y = Core([2,1], 4); T = Core([1,1], 4); X = Core([2,1], 4)
sage: LLMS4.forward_rule(Y, 1, T, None, X, 0)
(None, [2, 2], 0)
```

```
>>> from sage.all import *
>>> Y = Core([Integer(1)], Integer(3)); T = Core([], Integer(3)); X =␣
↪Core([Integer(1)], Integer(3))
>>> LLMS3.forward_rule(Y, Integer(0), T, None, X, Integer(0))
(None, [1, 1], -1)

>>> Y = Core([Integer(1)], Integer(4)); T = Core([], Integer(4)); X =␣
↪Core([Integer(1)], Integer(4))
>>> LLMS4.forward_rule(Y, Integer(0), T, None, X, Integer(0))
(None, [1, 1], -1)

>>> Y = Core([Integer(2),Integer(1)], Integer(4)); T = Core([Integer(1),
↪Integer(1)], Integer(4)); X = Core([Integer(2),Integer(1)], Integer(4))
>>> LLMS4.forward_rule(Y, Integer(1), T, None, X, Integer(0))
(None, [2, 2], 0)
```

has_multiple_edges = True

is_P_edge(*v*, *w*)

Return whether `(v, w)` is a P-edge of `self`.

For two k-cores v and w containing v, there are as many edges as there are components in the skew partition w/v. These components are ribbons, and therefore contain a unique cell with maximal content. We index the edge with this content.

EXAMPLES:

```
sage: LLMS4 = GrowthDiagram.rules.LLMS(4)
sage: v = LLMS4.vertices(2)[0]; v
[2]
sage: [(w, LLMS4.is_P_edge(v, w)) for w in LLMS4.vertices(3)]
[([3], [2]), ([2, 1], [-1]), ([1, 1, 1], [])]
sage: all(LLMS4.is_P_edge(v, w) == [] for w in LLMS4.vertices(4))
True
```

```
>>> from sage.all import *
>>> LLMS4 = GrowthDiagram.rules.LLMS(Integer(4))
>>> v = LLMS4.vertices(Integer(2))[Integer(0)]; v
[2]
>>> [(w, LLMS4.is_P_edge(v, w)) for w in LLMS4.vertices(Integer(3))]
[([3], [2]), ([2, 1], [-1]), ([1, 1, 1], [])]
>>> all(LLMS4.is_P_edge(v, w) == [] for w in LLMS4.vertices(Integer(4)))
True
```

is_Q_edge(*v*, *w*)

Return whether `(v, w)` is a Q-edge of `self`.

`(v, w)` is an edge if `w` is a weak cover of `v`, see *weak_covers()*.

EXAMPLES:

```
sage: LLMS4 = GrowthDiagram.rules.LLMS(4)
sage: v = LLMS4.vertices(3)[1]; v
[2, 1]
sage: [w for w in LLMS4.vertices(4) if len(LLMS4.is_Q_edge(v, w)) > 0]
[[2, 2], [3, 1, 1]]
sage: all(LLMS4.is_Q_edge(v, w) == [] for w in LLMS4.vertices(5))
True
```

```
>>> from sage.all import *
>>> LLMS4 = GrowthDiagram.rules.LLMS(Integer(4))
>>> v = LLMS4.vertices(Integer(3))[Integer(1)]; v
[2, 1]
>>> [w for w in LLMS4.vertices(Integer(4)) if len(LLMS4.is_Q_edge(v, w)) >␣
↪Integer(0)]
[[2, 2], [3, 1, 1]]
>>> all(LLMS4.is_Q_edge(v, w) == [] for w in LLMS4.vertices(Integer(5)))
True
```

normalize_vertex(*v*)

Convert `v` to a k-core.

EXAMPLES:

```
sage: LLMS3 = GrowthDiagram.rules.LLMS(3)
sage: LLMS3.normalize_vertex([3,1]).parent()
3-Cores of length 3
```

```
>>> from sage.all import *
>>> LLMS3 = GrowthDiagram.rules.LLMS(Integer(3))
>>> LLMS3.normalize_vertex([Integer(3),Integer(1)]).parent()
3-Cores of length 3
```

rank(*v*)

Return the rank of `v`: the length of the core.

EXAMPLES:

```
sage: LLMS3 = GrowthDiagram.rules.LLMS(3)
sage: LLMS3.rank(LLMS3.vertices(3)[0])
3
```

```
>>> from sage.all import *
>>> LLMS3 = GrowthDiagram.rules.LLMS(Integer(3))
>>> LLMS3.rank(LLMS3.vertices(Integer(3))[Integer(0)])
3
```

vertices(*n*)

Return the vertices of the dual graded graph on level `n`.

EXAMPLES:

```
sage: LLMS3 = GrowthDiagram.rules.LLMS(3)
sage: LLMS3.vertices(2)
3-Cores of length 2
```

```
>>> from sage.all import *
>>> LLMS3 = GrowthDiagram.rules.LLMS(Integer(3))
>>> LLMS3.vertices(Integer(2))
3-Cores of length 2
```

zero_edge = None

class sage.combinat.growth.**RulePartitions**

Bases: *Rule*

A rule for growth diagrams on Young's lattice on integer partitions graded by size.

P_symbol(*P_chain*)

Return the labels along the vertical boundary of a rectangular growth diagram as a (skew) tableau.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = RuleRSK([[0,1,0], [1,0,2]])
sage: G.P_symbol().pp()
1  2  2
2
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = RuleRSK([[Integer(0),Integer(1),Integer(0)], [Integer(1),Integer(0),
↪Integer(2)]])
>>> G.P_symbol().pp()
1  2  2
2
```

Q_symbol(*Q_chain*)

Return the labels along the horizontal boundary of a rectangular growth diagram as a skew tableau.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: G = RuleRSK([[0,1,0], [1,0,2]])
sage: G.Q_symbol().pp()
1  3  3
2
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> G = RuleRSK([[Integer(0),Integer(1),Integer(0)], [Integer(1),Integer(0),
↪Integer(2)]])
>>> G.Q_symbol().pp()
1  3  3
2
```

normalize_vertex(*v*)

Return v as a partition.

EXAMPLES:

```
sage: RSK = GrowthDiagram.rules.RSK()
sage: RSK.normalize_vertex([3,1]).parent()
Partitions
```

```
>>> from sage.all import *
>>> RSK = GrowthDiagram.rules.RSK()
>>> RSK.normalize_vertex([Integer(3),Integer(1)]).parent()
Partitions
```

rank(*v*)

Return the rank of v: the size of the partition.

EXAMPLES:

```
sage: RSK = GrowthDiagram.rules.RSK()
sage: RSK.rank(RSK.vertices(3)[0])
3
```

```
>>> from sage.all import *
>>> RSK = GrowthDiagram.rules.RSK()
>>> RSK.rank(RSK.vertices(Integer(3))[Integer(0)])
3
```

vertices(*n*)

Return the vertices of the dual graded graph on level n.

EXAMPLES:

```
sage: RSK = GrowthDiagram.rules.RSK()
sage: RSK.vertices(3)
Partitions of the integer 3
```

```
>>> from sage.all import *
>>> RSK = GrowthDiagram.rules.RSK()
```

(continues on next page)

(continued from previous page)

```
>>> RSK.vertices(Integer(3))
Partitions of the integer 3
```

zero = []

class sage.combinat.growth.**RuleRSK**

Bases: *RulePartitions*

A rule modelling Robinson-Schensted-Knuth insertion.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: GrowthDiagram(RuleRSK, [3,2,1,2,3])
0  0  1  0  0
0  1  0  1  0
1  0  0  0  1
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> GrowthDiagram(RuleRSK, [Integer(3),Integer(2),Integer(1),Integer(2),
↪Integer(3)])
0  0  1  0  0
0  1  0  1  0
1  0  0  0  1
```

The vertices of the dual graded graph are integer partitions:

```
sage: RuleRSK.vertices(3)
Partitions of the integer 3
```

```
>>> from sage.all import *
>>> RuleRSK.vertices(Integer(3))
Partitions of the integer 3
```

The local rules implemented provide the RSK correspondence between matrices with nonnegative integer entries and pairs of semistandard tableaux, the *P_symbol()* and the *Q_symbol()*. For permutations, it reduces to classical Schensted insertion.

Instead of passing the rule to *GrowthDiagram*, we can also call the rule to create growth diagrams. For example:

```
sage: m = matrix([[0,0,0,0,1],[1,1,0,2,0], [0,3,0,0,0]])
sage: G = RuleRSK(m); G
0  0  0  0  1
1  1  0  2  0
0  3  0  0  0

sage: ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  2  2  3    1  2  2  2  2 ]
[   2  3             4  4          ]
[   3           ,    5             ]
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(0),Integer(0),Integer(0),Integer(1)],
↪[Integer(1),Integer(1),Integer(0),Integer(2),Integer(0)], [Integer(0),
↪Integer(3),Integer(0),Integer(0),Integer(0)]])
>>> G = RuleRSK(m); G
0  0  0  0  1
1  1  0  2  0
0  3  0  0  0

>>> ascii_art([G.P_symbol(), G.Q_symbol()])
[   1  2  2  2  3    1  2  2  2  2 ]
[   2  3             4  4          ]
[   3           ,    5             ]
```

For rectangular fillings, the Kleitman-Greene invariant is the shape of the `P_symbol()` (or the `Q_symbol()`). Put differently, it is the partition labelling the lower right corner of the filling (recall that we are using matrix coordinates). It can be computed alternatively as the partition $(\mu_1, \ldots, \mu_n)$, where $\mu_1 + \cdots + \mu_i$ is the maximal sum of entries in a collection of i pairwise disjoint sequences of cells with weakly increasing coordinates.

For rectangular fillings, we could also use the (faster) implementation provided via `RSK()`. Because the of the coordinate conventions in `RSK()`, we have to transpose matrices:

```
sage: [G.P_symbol(), G.Q_symbol()] == RSK(m.transpose())
True

sage: n = 5; l = [(pi, RuleRSK(pi)) for pi in Permutations(n)]
sage: all([G.P_symbol(), G.Q_symbol()] == RSK(pi) for pi, G in l)
True

sage: n = 5; l = [(w, RuleRSK(w)) for w in Words([1,2,3], 5)]
sage: all([G.P_symbol(), G.Q_symbol()] == RSK(pi) for pi, G in l)
True
```

```
>>> from sage.all import *
>>> [G.P_symbol(), G.Q_symbol()] == RSK(m.transpose())
True

>>> n = Integer(5); l = [(pi, RuleRSK(pi)) for pi in Permutations(n)]
>>> all([G.P_symbol(), G.Q_symbol()] == RSK(pi) for pi, G in l)
True

>>> n = Integer(5); l = [(w, RuleRSK(w)) for w in Words([Integer(1),Integer(2),
↪Integer(3)], Integer(5))]
>>> all([G.P_symbol(), G.Q_symbol()] == RSK(pi) for pi, G in l)
True
```

backward_rule(*y*, *z*, *x*)

Return the content and the input shape.

See [Kra2006] $(B^1 0) - (B^1 2)$.

INPUT:

- `y`, `z`, `x` – three partitions from a cell in a growth diagram, labelled as:

```
  x
y z
```

OUTPUT:

A pair `(t, content)` consisting of the shape of the fourth word according to the Robinson-Schensted-Knuth correspondence and the content of the cell.

forward_rule(*y*, *t*, *x*, *content*)

Return the output shape given three shapes and the content.

See [Kra2006] $(F^1 0) - (F^1 2)$.

INPUT:

- `y`, `t`, `x` – three partitions from a cell in a growth diagram, labelled as:

```
t x
y
```

- `content` – nonnegative integer; the content of the cell

OUTPUT:

The fourth partition according to the Robinson-Schensted-Knuth correspondence.

EXAMPLES:

```
sage: RuleRSK = GrowthDiagram.rules.RSK()
sage: RuleRSK.forward_rule([2,1],[2,1],[2,1],1)
[3, 1]

sage: RuleRSK.forward_rule([1],[],[2],2)
[4, 1]
```

```
>>> from sage.all import *
>>> RuleRSK = GrowthDiagram.rules.RSK()
>>> RuleRSK.forward_rule([Integer(2),Integer(1)],[Integer(2),Integer(1)],
↪[Integer(2),Integer(1)],Integer(1))
[3, 1]

>>> RuleRSK.forward_rule([Integer(1)],[],[Integer(2)],Integer(2))
[4, 1]
```

class sage.combinat.growth.**RuleShiftedShapes**

Bases: *Rule*

A class modelling the Schensted correspondence for shifted shapes.

This agrees with Sagan [Sag1987] and Worley's [Wor1984], and Haiman's [Hai1989] insertion algorithms, see Proposition 4.5.2 of [Fom1995].

EXAMPLES:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: GrowthDiagram(Shifted, [3,1,2])
0  1  0
```

(continues on next page)

(continued from previous page)

```
0  0  1
1  0  0
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> GrowthDiagram(Shifted, [Integer(3),Integer(1),Integer(2)])
0  1  0
0  0  1
1  0  0
```

The vertices of the dual graded graph are shifted shapes:

```
sage: Shifted.vertices(3)
Partitions of the integer 3 satisfying constraints max_slope=-1
```

```
>>> from sage.all import *
>>> Shifted.vertices(Integer(3))
Partitions of the integer 3 satisfying constraints max_slope=-1
```

Let us check the example just before Corollary 3.2 in [Sag1987]. Note that, instead of passing the rule to `GrowthDiagram`, we can also call the rule to create growth diagrams:

```
sage: G = Shifted([2,6,5,1,7,4,3])
sage: G.P_chain()
[[], 0, [1], 0, [2], 0, [3], 0, [3, 1], 0, [3, 2], 0, [4, 2], 0, [5, 2]]
sage: G.Q_chain()
[[], 1, [1], 2, [2], 1, [2, 1], 3, [3, 1], 2, [4, 1], 3, [4, 2], 3, [5, 2]]
```

```
>>> from sage.all import *
>>> G = Shifted([Integer(2),Integer(6),Integer(5),Integer(1),Integer(7),
↪Integer(4),Integer(3)])
>>> G.P_chain()
[[], 0, [1], 0, [2], 0, [3], 0, [3, 1], 0, [3, 2], 0, [4, 2], 0, [5, 2]]
>>> G.Q_chain()
[[], 1, [1], 2, [2], 1, [2, 1], 3, [3, 1], 2, [4, 1], 3, [4, 2], 3, [5, 2]]
```

P_symbol(*P_chain*)

Return the labels along the vertical boundary of a rectangular growth diagram as a shifted tableau.

EXAMPLES:

Check the example just before Corollary 3.2 in [Sag1987]:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: G = Shifted([2,6,5,1,7,4,3])
sage: G.P_symbol().pp()
1  2  3  6  7
   4  5
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> G = Shifted([Integer(2),Integer(6),Integer(5),Integer(1),Integer(7),
```

(continues on next page)

(continued from previous page)

```
↪Integer(4),Integer(3)])
>>> G.P_symbol().pp()
1  2  3  6  7
   4  5
```

Check the example just before Corollary 8.2 in [SS1990]:

```
sage: T = ShiftedPrimedTableau([[4],[1],[5]], skew=[3,1])
sage: T.pp()
 .  .  .  4
    .  1
       5
sage: U = ShiftedPrimedTableau([[1],[3.5],[5]], skew=[3,1])
sage: U.pp()
 .  .  .  1
    .  4'
       5
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: labels = [mu if is_even(i) else 0
....:           for i, mu in enumerate(T.to_chain()[::-1])] + U.to_chain()[1:]
sage: G = Shifted({(1,2):1, (2,1):1}, shape=[5,5,5,5,5], labels=labels)
sage: G.P_symbol().pp()
 .  .  .  .  2
    .  .  1  3
       .  4  5
```

```
>>> from sage.all import *
>>> T = ShiftedPrimedTableau([[Integer(4)],[Integer(1)],[Integer(5)]],␣
↪skew=[Integer(3),Integer(1)])
>>> T.pp()
 .  .  .  4
    .  1
       5
>>> U = ShiftedPrimedTableau([[Integer(1)],[RealNumber('3.5')],[Integer(5)]],␣
↪skew=[Integer(3),Integer(1)])
>>> U.pp()
 .  .  .  1
    .  4'
       5
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> labels = [mu if is_even(i) else Integer(0)
...           for i, mu in enumerate(T.to_chain()[::-Integer(1)])] + U.to_
↪chain()[Integer(1):]
>>> G = Shifted({(Integer(1),Integer(2)):Integer(1), (Integer(2),
↪Integer(1)):Integer(1)}, shape=[Integer(5),Integer(5),Integer(5),Integer(5),
↪Integer(5)], labels=labels)
>>> G.P_symbol().pp()
 .  .  .  .  2
    .  .  1  3
       .  4  5
```

Q_symbol(*Q_chain*)

Return the labels along the horizontal boundary of a rectangular growth diagram as a skew tableau.

EXAMPLES:

Check the example just before Corollary 3.2 in [Sag1987]:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: G = Shifted([2,6,5,1,7,4,3])
sage: G.Q_symbol().pp()
1  2  4' 5  7'
   3  6'
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> G = Shifted([Integer(2),Integer(6),Integer(5),Integer(1),Integer(7),
↪Integer(4),Integer(3)])
>>> G.Q_symbol().pp()
1  2  4' 5  7'
   3  6'
```

Check the example just before Corollary 8.2 in [SS1990]:

```
sage: T = ShiftedPrimedTableau([[4],[1],[5]], skew=[3,1])
sage: T.pp()
 .  .  .  4
    .  1
       5
sage: U = ShiftedPrimedTableau([[1],[3.5],[5]], skew=[3,1])
sage: U.pp()
 .  .  .  1
    .  4'
       5
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: labels = [mu if is_even(i) else 0
....:           for i, mu in enumerate(T.to_chain()[::-1])] + U.to_chain()[1:]
sage: G = Shifted({(1,2):1, (2,1):1}, shape=[5,5,5,5,5], labels=labels)
sage: G.Q_symbol().pp()
 .  .  .  .  2
    .  .  1  4'
       .  3' 5'
```

```
>>> from sage.all import *
>>> T = ShiftedPrimedTableau([[Integer(4)],[Integer(1)],[Integer(5)]],␣
↪skew=[Integer(3),Integer(1)])
>>> T.pp()
 .  .  .  4
    .  1
       5
>>> U = ShiftedPrimedTableau([[Integer(1)],[RealNumber('3.5')],[Integer(5)]],␣
↪skew=[Integer(3),Integer(1)])
>>> U.pp()
 .  .  .  1
    .  4'
       5
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> labels = [mu if is_even(i) else Integer(0)
```

(continues on next page)

(continued from previous page)

```
...             for i, mu in enumerate(T.to_chain()[::-Integer(1)])] + U.to_
↪chain()[Integer(1):]
>>> G = Shifted({(Integer(1),Integer(2)):Integer(1), (Integer(2),
↪Integer(1)):Integer(1)}, shape=[Integer(5),Integer(5),Integer(5),Integer(5),
↪Integer(5)], labels=labels)
>>> G.Q_symbol().pp()
 .  .  .  .  2
    .  .  1  4'
       .  3' 5'
```

backward_rule(*y*, *g*, *z*, *h*, *x*)

Return the input path and the content given two incident edges.

See [Fom1995] Lemma 4.5.1, page 38.

INPUT:

- y, g, z, h, x – a path of three partitions and two colors from a cell in a growth diagram, labelled as:

```
    x
    h
y g z
```

OUTPUT:

A tuple (e, t, f, content) consisting of the shape t of the fourth word, the colours of the incident edges and the content of the cell according to Sagan - Worley insertion.

EXAMPLES:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: Shifted.backward_rule([], 1, [1], 0, [])
(0, [], 0, 1)

sage: Shifted.backward_rule([1], 2, [2], 0, [1])
(0, [1], 0, 1)
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> Shifted.backward_rule([], Integer(1), [Integer(1)], Integer(0), [])
(0, [], 0, 1)

>>> Shifted.backward_rule([Integer(1)], Integer(2), [Integer(2)], Integer(0),␣
↪[Integer(1)])
(0, [1], 0, 1)
```

if x != y:

```
sage: Shifted.backward_rule([3], 1, [3, 1], 0, [2,1])
(0, [2], 1, 0)

sage: Shifted.backward_rule([2,1], 2, [3, 1], 0, [3])
(0, [2], 2, 0)
```

```
>>> from sage.all import *
>>> Shifted.backward_rule([Integer(3)], Integer(1), [Integer(3), Integer(1)],␣
↪Integer(0), [Integer(2),Integer(1)])
(0, [2], 1, 0)

>>> Shifted.backward_rule([Integer(2),Integer(1)], Integer(2), [Integer(3),␣
↪Integer(1)], Integer(0), [Integer(3)])
(0, [2], 2, 0)
```

if `x == y != t`:

```
sage: Shifted.backward_rule([3], 1, [3, 1], 0, [3])
(0, [2], 2, 0)

sage: Shifted.backward_rule([3,1], 2, [3, 2], 0, [3,1])
(0, [2, 1], 2, 0)

sage: Shifted.backward_rule([2,1], 3, [3, 1], 0, [2,1])
(0, [2], 1, 0)

sage: Shifted.backward_rule([3], 3, [4], 0, [3])
(0, [2], 3, 0)
```

```
>>> from sage.all import *
>>> Shifted.backward_rule([Integer(3)], Integer(1), [Integer(3), Integer(1)],␣
↪Integer(0), [Integer(3)])
(0, [2], 2, 0)

>>> Shifted.backward_rule([Integer(3),Integer(1)], Integer(2), [Integer(3),␣
↪Integer(2)], Integer(0), [Integer(3),Integer(1)])
(0, [2, 1], 2, 0)

>>> Shifted.backward_rule([Integer(2),Integer(1)], Integer(3), [Integer(3),␣
↪Integer(1)], Integer(0), [Integer(2),Integer(1)])
(0, [2], 1, 0)

>>> Shifted.backward_rule([Integer(3)], Integer(3), [Integer(4)], Integer(0),␣
↪[Integer(3)])
(0, [2], 3, 0)
```

forward_rule(*y*, *e*, *t*, *f*, *x*, *content*)

Return the output path given two incident edges and the content.

See [Fom1995] Lemma 4.5.1, page 38.

INPUT:

- `y`, `e`, `t`, `f`, `x` – a path of three partitions and two colors from a cell in a growth diagram, labelled as:

```
t f x
e
y
```

- `content` – 0 or 1; the content of the cell

OUTPUT:

The two colors and the fourth partition `g`, `z`, `h` according to Sagan-Worley insertion.

EXAMPLES:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: Shifted.forward_rule([], 0, [], 0, [], 1)
(1, [1], 0)

sage: Shifted.forward_rule([1], 0, [1], 0, [1], 1)
(2, [2], 0)
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> Shifted.forward_rule([], Integer(0), [], Integer(0), [], Integer(1))
(1, [1], 0)

>>> Shifted.forward_rule([Integer(1)], Integer(0), [Integer(1)], Integer(0),␣
→[Integer(1)], Integer(1))
(2, [2], 0)
```

if `x != y`:

```
sage: Shifted.forward_rule([3], 0, [2], 1, [2,1], 0)
(1, [3, 1], 0)

sage: Shifted.forward_rule([2,1], 0, [2], 2, [3], 0)
(2, [3, 1], 0)
```

```
>>> from sage.all import *
>>> Shifted.forward_rule([Integer(3)], Integer(0), [Integer(2)], Integer(1),␣
→[Integer(2),Integer(1)], Integer(0))
(1, [3, 1], 0)

>>> Shifted.forward_rule([Integer(2),Integer(1)], Integer(0), [Integer(2)],␣
→Integer(2), [Integer(3)], Integer(0))
(2, [3, 1], 0)
```

if `x == y != t`:

```
sage: Shifted.forward_rule([3], 0, [2], 2, [3], 0)
(1, [3, 1], 0)

sage: Shifted.forward_rule([3,1], 0, [2,1], 2, [3,1], 0)
(2, [3, 2], 0)

sage: Shifted.forward_rule([2,1], 0, [2], 1, [2,1], 0)
(3, [3, 1], 0)

sage: Shifted.forward_rule([3], 0, [2], 3, [3], 0)
(3, [4], 0)
```

```
>>> from sage.all import *
>>> Shifted.forward_rule([Integer(3)], Integer(0), [Integer(2)], Integer(2),␣
↪[Integer(3)], Integer(0))
(1, [3, 1], 0)

>>> Shifted.forward_rule([Integer(3),Integer(1)], Integer(0), [Integer(2),
↪Integer(1)], Integer(2), [Integer(3),Integer(1)], Integer(0))
(2, [3, 2], 0)

>>> Shifted.forward_rule([Integer(2),Integer(1)], Integer(0), [Integer(2)],␣
↪Integer(1), [Integer(2),Integer(1)], Integer(0))
(3, [3, 1], 0)

>>> Shifted.forward_rule([Integer(3)], Integer(0), [Integer(2)], Integer(3),␣
↪[Integer(3)], Integer(0))
(3, [4], 0)
```

has_multiple_edges = True

is_P_edge(*v*, *w*)

Return whether `(v, w)` is a P-edge of `self`.

`(v, w)` is an edge if `w` contains `v`.

EXAMPLES:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: v = Shifted.vertices(2)[0]; v
[2]
sage: [w for w in Shifted.vertices(3) if Shifted.is_P_edge(v, w)]
[[3], [2, 1]]
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> v = Shifted.vertices(Integer(2))[Integer(0)]; v
[2]
>>> [w for w in Shifted.vertices(Integer(3)) if Shifted.is_P_edge(v, w)]
[[3], [2, 1]]
```

is_Q_edge(*v*, *w*)

Return whether `(v, w)` is a Q-edge of `self`.

`(v, w)` is an edge if `w` is obtained from `v` by adding a cell. It is a black (color 1) edge, if the cell is on the diagonal, otherwise it can be blue or red (color 2 or 3).

EXAMPLES:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: v = Shifted.vertices(2)[0]; v
[2]
sage: [(w, Shifted.is_Q_edge(v, w)) for w in Shifted.vertices(3)]
[([3], [2, 3]), ([2, 1], [1])]
sage: all(Shifted.is_Q_edge(v, w) == [] for w in Shifted.vertices(4))
True
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> v = Shifted.vertices(Integer(2))[Integer(0)]; v
[2]
>>> [(w, Shifted.is_Q_edge(v, w)) for w in Shifted.vertices(Integer(3))]
[([3], [2, 3]), ([2, 1], [1])]
>>> all(Shifted.is_Q_edge(v, w) == [] for w in Shifted.vertices(Integer(4)))
True
```

normalize_vertex(*v*)

Return v as a partition.

EXAMPLES:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: Shifted.normalize_vertex([3,1]).parent()
Partitions
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> Shifted.normalize_vertex([Integer(3),Integer(1)]).parent()
Partitions
```

rank(*v*)

Return the rank of v: the size of the shifted partition.

EXAMPLES:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: Shifted.rank(Shifted.vertices(3)[0])
3
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> Shifted.rank(Shifted.vertices(Integer(3))[Integer(0)])
3
```

vertices(*n*)

Return the vertices of the dual graded graph on level n.

EXAMPLES:

```
sage: Shifted = GrowthDiagram.rules.ShiftedShapes()
sage: Shifted.vertices(3)
Partitions of the integer 3 satisfying constraints max_slope=-1
```

```
>>> from sage.all import *
>>> Shifted = GrowthDiagram.rules.ShiftedShapes()
>>> Shifted.vertices(Integer(3))
Partitions of the integer 3 satisfying constraints max_slope=-1
```

zero = []

class sage.combinat.growth.**RuleSylvester**

Bases: *Rule*

A rule modelling a Schensted-like correspondence for binary trees.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: GrowthDiagram(Sylvester, [3,1,2])
0  1  0
0  0  1
1  0  0
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> GrowthDiagram(Sylvester, [Integer(3),Integer(1),Integer(2)])
0  1  0
0  0  1
1  0  0
```

The vertices of the dual graded graph are *BinaryTrees*:

```
sage: Sylvester.vertices(3)
Binary trees of size 3
```

```
>>> from sage.all import *
>>> Sylvester.vertices(Integer(3))
Binary trees of size 3
```

The *P_graph()* is also known as the bracket tree, the *Q_graph()* is the lattice of finite order ideals of the infinite binary tree, see Example 2.4.6 in [Fom1994].

For a permutation, the *P_symbol()* is the binary search tree, the *Q_symbol()* is the increasing tree corresponding to the inverse permutation. Note that, instead of passing the rule to *GrowthDiagram*, we can also call the rule to create growth diagrams. From [Nze2007]:

```
sage: pi = Permutation([3,5,1,4,2,6]); G = Sylvester(pi); G
0  0  1  0  0  0
0  0  0  0  1  0
1  0  0  0  0  0
0  0  0  1  0  0
0  1  0  0  0  0
0  0  0  0  0  1
sage: ascii_art(G.P_symbol())
  __3__
 /     \
1       5
 \     / \
  2   4   6
sage: ascii_art(G.Q_symbol())
  __1__
 /     \
3       2
 \     / \
  5   4   6
```

(continues on next page)

(continued from previous page)

```
sage: all(Sylvester(pi).P_symbol() == pi.binary_search_tree()
....:     for pi in Permutations(5))
True

sage: all(Sylvester(pi).Q_symbol() == pi.inverse().increasing_tree()
....:     for pi in Permutations(5))
True
```

```
>>> from sage.all import *
>>> pi = Permutation([Integer(3),Integer(5),Integer(1),Integer(4),Integer(2),
↪Integer(6)]); G = Sylvester(pi); G
0  0  1  0  0  0
0  0  0  0  1  0
1  0  0  0  0  0
0  0  0  1  0  0
0  1  0  0  0  0
0  0  0  0  0  1
>>> ascii_art(G.P_symbol())
  __3__
 /     \
1       5
 \     / \
  2   4   6
>>> ascii_art(G.Q_symbol())
  __1__
 /     \
3       2
 \     / \
  5   4   6

>>> all(Sylvester(pi).P_symbol() == pi.binary_search_tree()
...     for pi in Permutations(Integer(5)))
True

>>> all(Sylvester(pi).Q_symbol() == pi.inverse().increasing_tree()
...     for pi in Permutations(Integer(5)))
True
```

P_symbol(*P_chain*)

Return the labels along the vertical boundary of a rectangular growth diagram as a labelled binary tree.

For permutations, this coincides with the binary search tree.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: pi = Permutation([2,4,3,1])
sage: ascii_art(Sylvester(pi).P_symbol())
  _2_
 /   \
1     4
     /
```

(continues on next page)

(continued from previous page)

```
    3
sage: Sylvester(pi).P_symbol() == pi.binary_search_tree()
True
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> pi = Permutation([Integer(2),Integer(4),Integer(3),Integer(1)])
>>> ascii_art(Sylvester(pi).P_symbol())
  _2_
 /   \
1     4
     /
    3
>>> Sylvester(pi).P_symbol() == pi.binary_search_tree()
True
```

We can also do the skew version:

```
sage: B = BinaryTree; E = B(); N = B([])
sage: ascii_art(Sylvester([3,2], shape=[3,3,3], labels=[N,N,N,E,E,E,N]).P_
↪symbol())
  __1___
 /      \
None     3
        /
       2
```

```
>>> from sage.all import *
>>> B = BinaryTree; E = B(); N = B([])
>>> ascii_art(Sylvester([Integer(3),Integer(2)], shape=[Integer(3),Integer(3),
↪Integer(3)], labels=[N,N,N,E,E,E,N]).P_symbol())
  __1___
 /      \
None     3
        /
       2
```

Q_symbol(*Q_chain*)

Return the labels along the vertical boundary of a rectangular growth diagram as a labelled binary tree.

For permutations, this coincides with the increasing tree.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: pi = Permutation([2,4,3,1])
sage: ascii_art(Sylvester(pi).Q_symbol())
  _1_
 /   \
4     2
     /
    3
```

(continues on next page)

(continued from previous page)

```
sage: Sylvester(pi).Q_symbol() == pi.inverse().increasing_tree()
True
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> pi = Permutation([Integer(2),Integer(4),Integer(3),Integer(1)])
>>> ascii_art(Sylvester(pi).Q_symbol())
  _1_
 /   \
4     2
     /
    3
>>> Sylvester(pi).Q_symbol() == pi.inverse().increasing_tree()
True
```

We can also do the skew version:

```
sage: B = BinaryTree; E = B(); N = B([])
sage: ascii_art(Sylvester([3,2], shape=[3,3,3], labels=[N,N,N,E,E,E,N]).Q_
↪symbol())
  _None_
 /      \
3        1
        /
       2
```

```
>>> from sage.all import *
>>> B = BinaryTree; E = B(); N = B([])
>>> ascii_art(Sylvester([Integer(3),Integer(2)], shape=[Integer(3),Integer(3),
↪Integer(3)], labels=[N,N,N,E,E,E,N]).Q_symbol())
  _None_
 /      \
3        1
        /
       2
```

backward_rule(*y*, *z*, *x*)

Return the output shape given three shapes and the content.

See [Nze2007], page 9.

INPUT:

- `y, z, x` – three binary trees from a cell in a growth diagram, labelled as:

```
  x
y z
```

OUTPUT:

A pair `(t, content)` consisting of the shape of the fourth binary tree t and the content of the cell.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: B = BinaryTree; E = B(); N = B([]); L = B([[],None])
sage: R = B([None,[]]); T = B([[],[]])

sage: ascii_art(Sylvester.backward_rule(E, E, E))
( , 0 )
sage: ascii_art(Sylvester.backward_rule(N, N, N))
( o, 0 )
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> B = BinaryTree; E = B(); N = B([]); L = B([[],None])
>>> R = B([None,[]]); T = B([[],[]])

>>> ascii_art(Sylvester.backward_rule(E, E, E))
( , 0 )
>>> ascii_art(Sylvester.backward_rule(N, N, N))
( o, 0 )
```

forward_rule(*y*, *t*, *x*, *content*)

Return the output shape given three shapes and the content.

See [Nze2007], page 9.

INPUT:

- `y`, `t`, `x` – three binary trees from a cell in a growth diagram, labelled as:

```
t x
y
```

- `content` – 0 or 1; the content of the cell

OUTPUT: the fourth binary tree `z`

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: B = BinaryTree; E = B(); N = B([]); L = B([[],None])
sage: R = B([None,[]]); T = B([[],[]])

sage: ascii_art(Sylvester.forward_rule(E, E, E, 1))
o
sage: ascii_art(Sylvester.forward_rule(N, N, N, 1))
o
 \
  o
sage: ascii_art(Sylvester.forward_rule(L, L, L, 1))
  o
 / \
o   o
sage: ascii_art(Sylvester.forward_rule(R, R, R, 1))
o
 \
  o
```

(continues on next page)

(continued from previous page)

```
    \
     o
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> B = BinaryTree; E = B(); N = B([]); L = B([[],None])
>>> R = B([None,[]]); T = B([[],[]])

>>> ascii_art(Sylvester.forward_rule(E, E, E, Integer(1)))
o
>>> ascii_art(Sylvester.forward_rule(N, N, N, Integer(1)))
o
 \
  o
>>> ascii_art(Sylvester.forward_rule(L, L, L, Integer(1)))
  o
 / \
o   o
>>> ascii_art(Sylvester.forward_rule(R, R, R, Integer(1)))
o
 \
  o
   \
    o
```

If `y != x`, obtain `z` from `y` adding a node such that deleting the right most gives `x`:

```
sage: ascii_art(Sylvester.forward_rule(R, N, L, 0))
  o
 / \
o   o

sage: ascii_art(Sylvester.forward_rule(L, N, R, 0))
  o
 /
o
 \
  o
```

```
>>> from sage.all import *
>>> ascii_art(Sylvester.forward_rule(R, N, L, Integer(0)))
  o
 / \
o   o

>>> ascii_art(Sylvester.forward_rule(L, N, R, Integer(0)))
  o
 /
o
 \
  o
```

If `y == x != t`, obtain `z` from `x` by adding a node as left child to the right most node:

```
sage: ascii_art(Sylvester.forward_rule(N, E, N, 0))
  o
 /
o
sage: ascii_art(Sylvester.forward_rule(T, L, T, 0))
  _o_
 /   \
o     o
     /
    o
sage: ascii_art(Sylvester.forward_rule(L, N, L, 0))
    o
   /
  o
 /
o
sage: ascii_art(Sylvester.forward_rule(R, N, R, 0))
o
 \
  o
 /
o
```

```
>>> from sage.all import *
>>> ascii_art(Sylvester.forward_rule(N, E, N, Integer(0)))
  o
 /
o
>>> ascii_art(Sylvester.forward_rule(T, L, T, Integer(0)))
  _o_
 /   \
o     o
     /
    o
>>> ascii_art(Sylvester.forward_rule(L, N, L, Integer(0)))
    o
   /
  o
 /
o
>>> ascii_art(Sylvester.forward_rule(R, N, R, Integer(0)))
o
 \
  o
 /
o
```

is_P_edge(*v*, *w*)

Return whether `(v, w)` is a P-edge of `self`.

`(v, w)` is an edge if `v` is obtained from `w` by deleting its right-most node.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: v = Sylvester.vertices(2)[1]; ascii_art(v)
  o
 /
o

sage: ascii_art([w for w in Sylvester.vertices(3) if Sylvester.is_P_edge(v,␣
↪w)])
[    o   ,      o ]
[   / \        /  ]
[ o    o     o    ]
[          /      ]
[         o       ]

sage: [w for w in Sylvester.vertices(4) if Sylvester.is_P_edge(v, w)]
[]
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> v = Sylvester.vertices(Integer(2))[Integer(1)]; ascii_art(v)
  o
 /
o

>>> ascii_art([w for w in Sylvester.vertices(Integer(3)) if Sylvester.is_P_
↪edge(v, w)])
[    o   ,      o ]
[   / \        /  ]
[ o    o     o    ]
[          /      ]
[         o       ]

>>> [w for w in Sylvester.vertices(Integer(4)) if Sylvester.is_P_edge(v, w)]
[]
```

is_Q_edge(*v*, *w*)

Return whether `(v, w)` is a Q-edge of `self`.

`(v, w)` is an edge if `v` is a sub-tree of `w` with one node less.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: v = Sylvester.vertices(2)[1]; ascii_art(v)
  o
 /
o
sage: ascii_art([w for w in Sylvester.vertices(3) if Sylvester.is_Q_edge(v,␣
↪w)])
[    o   ,   o,        o ]
[   / \     /         /  ]
[ o    o  o         o    ]
[          \       /     ]
```

(continues on next page)

(continued from previous page)

```
[           o   o      ]
sage: [w for w in Sylvester.vertices(4) if Sylvester.is_Q_edge(v, w)]
[]
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> v = Sylvester.vertices(Integer(2))[Integer(1)]; ascii_art(v)
  o
 /
o
>>> ascii_art([w for w in Sylvester.vertices(Integer(3)) if Sylvester.is_Q_
↪edge(v, w)])
[   o  ,    o,      o ]
[  / \     /       /  ]
[ o   o  o       o    ]
[          \    /     ]
[           o  o      ]
>>> [w for w in Sylvester.vertices(Integer(4)) if Sylvester.is_Q_edge(v, w)]
[]
```

normalize_vertex(*v*)

Return v as a binary tree.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: Sylvester.normalize_vertex([[],[]]).parent()
Binary trees
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> Sylvester.normalize_vertex([[],[]]).parent()
Binary trees
```

rank(*v*)

Return the rank of v: the number of nodes of the tree.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: Sylvester.rank(Sylvester.vertices(3)[0])
3
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> Sylvester.rank(Sylvester.vertices(Integer(3))[Integer(0)])
3
```

vertices(*n*)

Return the vertices of the dual graded graph on level n.

EXAMPLES:

```
sage: Sylvester = GrowthDiagram.rules.Sylvester()
sage: Sylvester.vertices(3)
Binary trees of size 3
```

```
>>> from sage.all import *
>>> Sylvester = GrowthDiagram.rules.Sylvester()
>>> Sylvester.vertices(Integer(3))
Binary trees of size 3
```

zero = .

class sage.combinat.growth.**RuleYoungFibonacci**

Bases: *Rule*

A rule modelling a Schensted-like correspondence for Young-Fibonacci-tableaux.

EXAMPLES:

```
sage: YF = GrowthDiagram.rules.YoungFibonacci()
sage: GrowthDiagram(YF, [3,1,2])
0  1  0
0  0  1
1  0  0
```

```
>>> from sage.all import *
>>> YF = GrowthDiagram.rules.YoungFibonacci()
>>> GrowthDiagram(YF, [Integer(3),Integer(1),Integer(2)])
0  1  0
0  0  1
1  0  0
```

The vertices of the dual graded graph are Fibonacci words - compositions into parts of size at most two:

```
sage: YF.vertices(4)
[word: 22, word: 211, word: 121, word: 112, word: 1111]
```

```
>>> from sage.all import *
>>> YF.vertices(Integer(4))
[word: 22, word: 211, word: 121, word: 112, word: 1111]
```

Note that, instead of passing the rule to *GrowthDiagram*, we can also use call the rule to create growth diagrams. For example:

```
sage: G = YF([2, 3, 7, 4, 1, 6, 5]); G
0  0  0  0  1  0  0
1  0  0  0  0  0  0
0  1  0  0  0  0  0
0  0  0  1  0  0  0
0  0  0  0  0  0  1
0  0  0  0  0  1  0
0  0  1  0  0  0  0
```

```
>>> from sage.all import *
>>> G = YF([Integer(2), Integer(3), Integer(7), Integer(4), Integer(1),␣
↪Integer(6), Integer(5)]); G
0  0  0  0  1  0  0
1  0  0  0  0  0  0
0  1  0  0  0  0  0
0  0  0  1  0  0  0
0  0  0  0  0  0  1
0  0  0  0  0  1  0
0  0  1  0  0  0  0
```

The Kleitman Greene invariant is: take the last letter and the largest letter of the permutation and remove them. If they coincide write 1, otherwise write 2:

```
sage: G.P_chain()[-1]
word: 21211
```

```
>>> from sage.all import *
>>> G.P_chain()[-Integer(1)]
word: 21211
```

backward_rule(*y*, *z*, *x*)

Return the content and the input shape.

See [Fom1995] Lemma 4.4.1, page 35.

- y, z, x – three Fibonacci words from a cell in a growth diagram, labelled as:

```
  x
y z
```

OUTPUT:

A pair (t, content) consisting of the shape of the fourth word and the content of the cell.

forward_rule(*y*, *t*, *x*, *content*)

Return the output shape given three shapes and the content.

See [Fom1995] Lemma 4.4.1, page 35.

INPUT:

- y, t, x – three Fibonacci words from a cell in a growth diagram, labelled as:

```
t x
y
```

- content – 0 or 1; the content of the cell

OUTPUT: the fourth Fibonacci word

EXAMPLES:

```
sage: YF = GrowthDiagram.rules.YoungFibonacci()

sage: YF.forward_rule([], [], [], 1)
word: 1
```

(continues on next page)

(continued from previous page)

```
sage: YF.forward_rule([1], [1], [1], 1)
word: 11

sage: YF.forward_rule([1,2], [1], [1,1], 0)
word: 21

sage: YF.forward_rule([1,1], [1], [1,1], 0)
word: 21
```

```
>>> from sage.all import *
>>> YF = GrowthDiagram.rules.YoungFibonacci()

>>> YF.forward_rule([], [], [], Integer(1))
word: 1

>>> YF.forward_rule([Integer(1)], [Integer(1)], [Integer(1)], Integer(1))
word: 11

>>> YF.forward_rule([Integer(1),Integer(2)], [Integer(1)], [Integer(1),
↪Integer(1)], Integer(0))
word: 21

>>> YF.forward_rule([Integer(1),Integer(1)], [Integer(1)], [Integer(1),
↪Integer(1)], Integer(0))
word: 21
```

is_P_edge(*v*, *w*)

Return whether `(v, w)` is a P-edge of `self`.

`(v, w)` is an edge if `v` is obtained from `w` by deleting a `1` or replacing the left-most `2` by a `1`.

EXAMPLES:

```
sage: YF = GrowthDiagram.rules.YoungFibonacci()
sage: v = YF.vertices(5)[5]; v
word: 1121
sage: [w for w in YF.vertices(6) if YF.is_P_edge(v, w)]
[word: 2121, word: 11121]
sage: [w for w in YF.vertices(7) if YF.is_P_edge(v, w)]
[]
```

```
>>> from sage.all import *
>>> YF = GrowthDiagram.rules.YoungFibonacci()
>>> v = YF.vertices(Integer(5))[Integer(5)]; v
word: 1121
>>> [w for w in YF.vertices(Integer(6)) if YF.is_P_edge(v, w)]
[word: 2121, word: 11121]
>>> [w for w in YF.vertices(Integer(7)) if YF.is_P_edge(v, w)]
[]
```

is_Q_edge(*v*, *w*)

Return whether `(v, w)` is a P-edge of `self`.

(v, w) is an edge if v is obtained from w by deleting a 1 or replacing the left-most 2 by a 1.

EXAMPLES:

```
sage: YF = GrowthDiagram.rules.YoungFibonacci()
sage: v = YF.vertices(5)[5]; v
word: 1121
sage: [w for w in YF.vertices(6) if YF.is_P_edge(v, w)]
[word: 2121, word: 11121]
sage: [w for w in YF.vertices(7) if YF.is_P_edge(v, w)]
[]
```

```
>>> from sage.all import *
>>> YF = GrowthDiagram.rules.YoungFibonacci()
>>> v = YF.vertices(Integer(5))[Integer(5)]; v
word: 1121
>>> [w for w in YF.vertices(Integer(6)) if YF.is_P_edge(v, w)]
[word: 2121, word: 11121]
>>> [w for w in YF.vertices(Integer(7)) if YF.is_P_edge(v, w)]
[]
```

normalize_vertex(*v*)

Return v as a word with letters 1 and 2.

EXAMPLES:

```
sage: YF = GrowthDiagram.rules.YoungFibonacci()
sage: YF.normalize_vertex([1,2,1]).parent()
Finite words over {1, 2}
```

```
>>> from sage.all import *
>>> YF = GrowthDiagram.rules.YoungFibonacci()
>>> YF.normalize_vertex([Integer(1),Integer(2),Integer(1)]).parent()
Finite words over {1, 2}
```

rank(*v*)

Return the rank of v: the size of the corresponding composition.

EXAMPLES:

```
sage: YF = GrowthDiagram.rules.YoungFibonacci()
sage: YF.rank(YF.vertices(3)[0])
3
```

```
>>> from sage.all import *
>>> YF = GrowthDiagram.rules.YoungFibonacci()
>>> YF.rank(YF.vertices(Integer(3))[Integer(0)])
3
```

vertices(*n*)

Return the vertices of the dual graded graph on level n.

EXAMPLES:

```
sage: YF = GrowthDiagram.rules.YoungFibonacci()
sage: YF.vertices(3)
[word: 21, word: 12, word: 111]
```

```
>>> from sage.all import *
>>> YF = GrowthDiagram.rules.YoungFibonacci()
>>> YF.vertices(Integer(3))
[word: 21, word: 12, word: 111]
```

zero = word:

class sage.combinat.growth.**Rules**

Bases: object

Catalog of rules for growth diagrams.

BinaryWord

alias of *RuleBinaryWord*

Burge

alias of *RuleBurge*

Domino

alias of *RuleDomino*

LLMS

alias of *RuleLLMS*

RSK

alias of *RuleRSK*

ShiftedShapes

alias of *RuleShiftedShapes*

Sylvester

alias of *RuleSylvester*

YoungFibonacci

alias of *RuleYoungFibonacci*

5.1.119 Grossman-Larson Hopf Algebras

AUTHORS:

- Frédéric Chapoton (2017)

class sage.combinat.grossman_larson_algebras.**GrossmanLarsonAlgebra**(*R*, *names=None*)

Bases: *CombinatorialFreeModule*

The Grossman-Larson Hopf Algebra.

The Grossman-Larson Hopf Algebras are Hopf algebras with a basis indexed by forests of decorated rooted trees. They are the universal enveloping algebras of free pre-Lie algebras, seen as Lie algebras.

The Grossman-Larson Hopf algebra on a given set E has an explicit description using rooted forests. The underlying vector space has a basis indexed by finite rooted forests endowed with a map from their vertices to E (called the "labeling"). In this basis, the product of two (decorated) rooted forests $S * T$ is a sum over all maps from the set of roots of T to the union of a singleton $\{\#\}$ and the set of vertices of S. Given such a map, one defines a new forest as follows. Starting from the disjoint union of all rooted trees of S and T, one adds an edge from every root of T to its image when this image is not the fake vertex labelled `#`. The coproduct sends a rooted forest T to the sum of all tensors $T_1 \otimes T_2$ obtained by splitting the connected components of T into two subsets and letting T_1 be the forest formed by the first subset and T_2 the forest formed by the second. This yields a connected graded Hopf algebra (the degree of a forest is its number of vertices).

See [Pana2002] (Section 2) and [GroLar1]. (Note that both references use rooted trees rather than rooted forests, so think of each rooted forest grafted onto a new root. Also, the product is reversed, so they are defining the opposite algebra structure.)

Warning

For technical reasons, instead of using forests as labels for the basis, we use rooted trees. Their root vertex should be considered as a fake vertex. This fake root vertex is labelled `'#'` when labels are present.

EXAMPLES:

```
sage: G = algebras.GrossmanLarson(QQ, 'xy')
sage: x, y = G.single_vertex_all()
sage: ascii_art(x*y)
B  + B
 #      #_
 |     / /
 x    x y
 |
 y

sage: ascii_art(x*x*x)
B  + B      + 3*B     + B
 #      #         #_      _#__
 |      |        / /     / / /
 x      x_      x x     x x x
 |     / /        |
 x    x x         x
 |
 x
```

```
>>> from sage.all import *
>>> G = algebras.GrossmanLarson(QQ, 'xy')
>>> x, y = G.single_vertex_all()
>>> ascii_art(x*y)
B  + B
 #      #_
 |     / /
 x    x y
 |
 y

>>> ascii_art(x*x*x)
B  + B      + 3*B     + B
 #      #         #_      _#__
 |      |        / /     / / /
```

(continues on next page)

(continued from previous page)

```
 x       x_      x x     x x x
 |      / /        |
 x     x x         x
 |
 x
```

The Grossman-Larson algebra is associative:

```
sage: z = x * y
sage: x * (y * z) == (x * y) * z
True
```

```
>>> from sage.all import *
>>> z = x * y
>>> x * (y * z) == (x * y) * z
True
```

It is not commutative:

```
sage: x * y == y * x
False
```

```
>>> from sage.all import *
>>> x * y == y * x
False
```

When `None` is given as input, unlabelled forests are used instead; this corresponds to a 1-element set E:

```
sage: G = algebras.GrossmanLarson(QQ, None)
sage: x = G.single_vertex_all()[0]
sage: ascii_art(x*x)
B  + B
 o       o_
 |      / /
 o     o o
 |
 o
```

```
>>> from sage.all import *
>>> G = algebras.GrossmanLarson(QQ, None)
>>> x = G.single_vertex_all()[Integer(0)]
>>> ascii_art(x*x)
B  + B
 o       o_
 |      / /
 o     o o
 |
 o
```

Note

Variables names can be `None`, a list of strings, a string or an integer. When `None` is given, unlabelled rooted forests are used. When a single string is given, each letter is taken as a variable. See *sage.combinat.words.alphabet.build_alphabet()*.

> **Warning:** Beware that the underlying combinatorial free module is based either on `RootedTrees` or on `LabelledRootedTrees`, with no restriction on the labellings. This means that all code calling the `basis()` method would not give meaningful results, since `basis()` returns many "chaff" elements that do not belong to the algebra.

REFERENCES:

- [Pana2002]
- [GroLar1]

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(QQ, 'xy')
sage: A.an_element()
B[#[x[]]] + 2*B[#[x[x[]]]] + 2*B[#[x[], x[]]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'xy')
>>> A.an_element()
B[#[x[]]] + 2*B[#[x[x[]]]] + 2*B[#[x[], x[]]]
```

antipode_on_basis(*x*)

Return the antipode of a forest.

EXAMPLES:

```
sage: G = algebras.GrossmanLarson(QQ,2)
sage: x, y = G.single_vertex_all()
sage: G.antipode(x)   # indirect doctest
-B[#[0[]]]

sage: G.antipode(y*x)   # indirect doctest
B[#[0[1[]]]] + B[#[0[], 1[]]]
```

```
>>> from sage.all import *
>>> G = algebras.GrossmanLarson(QQ,Integer(2))
>>> x, y = G.single_vertex_all()
>>> G.antipode(x)   # indirect doctest
-B[#[0[]]]

>>> G.antipode(y*x)   # indirect doctest
B[#[0[1[]]]] + B[#[0[], 1[]]]
```

change_ring(*R*)

Return the Grossman-Larson algebra in the same variables over R.

INPUT:

- R – a ring

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(ZZ, 'fgh')
sage: A.change_ring(QQ)
Grossman-Larson Hopf algebra on 3 generators ['f', 'g', 'h']
over Rational Field
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(ZZ, 'fgh')
>>> A.change_ring(QQ)
Grossman-Larson Hopf algebra on 3 generators ['f', 'g', 'h']
over Rational Field
```

coproduct_on_basis(*x*)

Return the coproduct of a forest.

EXAMPLES:

```
sage: G = algebras.GrossmanLarson(QQ,2)
sage: x, y = G.single_vertex_all()
sage: ascii_art(G.coproduct(x))  # indirect doctest
1 # B  + B  # 1
     #    #
     |    |
     0    0

sage: Delta_xy = G.coproduct(y*x)
sage: ascii_art(Delta_xy)  # random indirect doctest
1 # B     + 1 # B  + B  # B  + B     # 1 + B  # B  + B  # 1
       #_         #    #    #      #_         #    #    #
      / /         |    |    |     / /         |    |    |
     0 1          1    0    1    0 1          1    0    1
                  |                                     |
                  0                                     0
```

```
>>> from sage.all import *
>>> G = algebras.GrossmanLarson(QQ,Integer(2))
>>> x, y = G.single_vertex_all()
>>> ascii_art(G.coproduct(x))  # indirect doctest
1 # B  + B  # 1
     #    #
     |    |
     0    0

>>> Delta_xy = G.coproduct(y*x)
>>> ascii_art(Delta_xy)  # random indirect doctest
1 # B     + 1 # B  + B  # B  + B     # 1 + B  # B  + B  # 1
       #_         #    #    #      #_         #    #    #
```

(continues on next page)

(continued from previous page)

```
      / /          |     |     |      / /          |     |     |
     0 1           1     0     1     0 1           1     0     1
                   |                                           |
                   0                                           0
```

counit_on_basis(*x*)

Return the counit on a basis element.

This is zero unless the forest x is empty.

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(QQ, 'xy')
sage: RT = A.basis().keys()
sage: x = RT([RT([],'x')],'#')
sage: A.counit_on_basis(x)
0
sage: A.counit_on_basis(RT([],'#'))
1
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'xy')
>>> RT = A.basis().keys()
>>> x = RT([RT([],'x')],'#')
>>> A.counit_on_basis(x)
0
>>> A.counit_on_basis(RT([],'#'))
1
```

degree_on_basis(*t*)

Return the degree of a rooted forest in the Grossman-Larson algebra.

This is the total number of vertices of the forest.

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(QQ, '@')
sage: RT = A.basis().keys()
sage: A.degree_on_basis(RT([RT([])]))
1
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, '@')
>>> RT = A.basis().keys()
>>> A.degree_on_basis(RT([RT([])]))
1
```

one_basis()

Return the empty rooted forest.

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(QQ, 'ab')
sage: A.one_basis()
```

(continues on next page)

(continued from previous page)

```
#[]

sage: A = algebras.GrossmanLarson(QQ, None)
sage: A.one_basis()
[]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'ab')
>>> A.one_basis()
#[]

>>> A = algebras.GrossmanLarson(QQ, None)
>>> A.one_basis()
[]
```

product_on_basis(*x*, *y*)

Return the product of two forests x and y.

This is the sum over all possible ways for the components of the forest y to either fall side-by-side with components of x or be grafted on a vertex of x.

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.product_on_basis(x, x)
B[[[[]]]] + B[[[], []]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> A.product_on_basis(x, x)
B[[[[]]]] + B[[[], []]]
```

Check that the product is the correct one:

```
sage: A = algebras.GrossmanLarson(QQ, 'uv')
sage: RT = A.basis().keys()
sage: Tu = RT([RT([],'u')],'#')
sage: Tv = RT([RT([],'v')],'#')
sage: A.product_on_basis(Tu, Tv)
B[#[u[v[]]]] + B[#[u[], v[]]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'uv')
>>> RT = A.basis().keys()
>>> Tu = RT([RT([],'u')],'#')
>>> Tv = RT([RT([],'v')],'#')
>>> A.product_on_basis(Tu, Tv)
B[#[u[v[]]]] + B[#[u[], v[]]]
```

single_vertex(*i*)

Return the i-th rooted forest with one vertex.

This is the rooted forest with just one vertex, labelled by the i-th element of the label list.

See also:

single_vertex_all().

INPUT:

- i – nonnegative integer

EXAMPLES:

```
sage: F = algebras.GrossmanLarson(ZZ, 'xyz')
sage: F.single_vertex(0)
B[#[x[]]]

sage: F.single_vertex(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

```
>>> from sage.all import *
>>> F = algebras.GrossmanLarson(ZZ, 'xyz')
>>> F.single_vertex(Integer(0))
B[#[x[]]]

>>> F.single_vertex(Integer(4))
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

single_vertex_all()

Return the rooted forests with one vertex in self.

They freely generate the Lie algebra of primitive elements as a pre-Lie algebra.

See also:

single_vertex().

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(ZZ, 'fgh')
sage: A.single_vertex_all()
(B[#[f[]]], B[#[g[]]], B[#[h[]]])

sage: A = algebras.GrossmanLarson(QQ, ['x1','x2'])
sage: A.single_vertex_all()
(B[#[x1[]]], B[#[x2[]]])
```

(continues on next page)

(continued from previous page)

```
sage: A = algebras.GrossmanLarson(ZZ, None)
sage: A.single_vertex_all()
(B[[[]]],)
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(ZZ, 'fgh')
>>> A.single_vertex_all()
(B[#[f[]]], B[#[g[]]], B[#[h[]]])

>>> A = algebras.GrossmanLarson(QQ, ['x1','x2'])
>>> A.single_vertex_all()
(B[#[x1[]]], B[#[x2[]]])

>>> A = algebras.GrossmanLarson(ZZ, None)
>>> A.single_vertex_all()
(B[[[]]],)
```

some_elements()

Return some elements of the Grossman-Larson Hopf algebra.

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(QQ, None)
sage: A.some_elements()
[B[[[]]], B[[]] + B[[[[]]]] + B[[[], []]],
4*B[[[[]]]] + 4*B[[[], []]]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, None)
>>> A.some_elements()
[B[[[]]], B[[]] + B[[[[]]]] + B[[[], []]],
4*B[[[[]]]] + 4*B[[[], []]]]
```

With several generators:

```
sage: A = algebras.GrossmanLarson(QQ, 'xy')
sage: A.some_elements()
[B[#[x[]]],
 B[#[]] + B[#[x[x[]]]] + B[#[x[], x[]]],
 B[#[x[x[]]]] + 3*B[#[x[y[]]]] + B[#[x[], x[]]] + 3*B[#[x[], y[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'xy')
>>> A.some_elements()
[B[#[x[]]],
 B[#[]] + B[#[x[x[]]]] + B[#[x[], x[]]],
 B[#[x[x[]]]] + 3*B[#[x[y[]]]] + B[#[x[], x[]]] + 3*B[#[x[], y[]]]]
```

variable_names()

Return the names of the variables.

This returns the set E (as a family).

EXAMPLES:

```
sage: R = algebras.GrossmanLarson(QQ, 'xy')
sage: R.variable_names()
{'x', 'y'}

sage: R = algebras.GrossmanLarson(QQ, ['a','b'])
sage: R.variable_names()
{'a', 'b'}

sage: R = algebras.GrossmanLarson(QQ, 2)
sage: R.variable_names()
{0, 1}

sage: R = algebras.GrossmanLarson(QQ, None)
sage: R.variable_names()
{'o'}
```

```
>>> from sage.all import *
>>> R = algebras.GrossmanLarson(QQ, 'xy')
>>> R.variable_names()
{'x', 'y'}

>>> R = algebras.GrossmanLarson(QQ, ['a','b'])
>>> R.variable_names()
{'a', 'b'}

>>> R = algebras.GrossmanLarson(QQ, Integer(2))
>>> R.variable_names()
{0, 1}

>>> R = algebras.GrossmanLarson(QQ, None)
>>> R.variable_names()
{'o'}
```

5.1.120 Hall Polynomials

`sage.combinat.hall_polynomial.`**`hall_polynomial`**(*nu*, *mu*, *la*, *q=None*)

Return the (classical) Hall polynomial $P^{\nu}_{\mu,\lambda}$ (where ν, μ and λ are the inputs `nu`, `mu` and `la`).

Let ν, μ, λ be partitions. The Hall polynomial $P^{\nu}_{\mu,\lambda}(q)$ (in the indeterminate q) is defined as follows: Specialize q to a prime power, and consider the category of $\mathbf{F}_q$-vector spaces with a distinguished nilpotent endomorphism. The morphisms in this category shall be the linear maps commuting with the distinguished endomorphisms. The *type* of an object in the category will be the Jordan type of the distinguished endomorphism; this is a partition. Now, if N is any fixed object of type ν in this category, then the polynomial $P^{\nu}_{\mu,\lambda}(q)$ specialized at the prime power q counts the number of subobjects L of N having type λ such that the quotient object N/L has type μ. This determines the values of the polynomial $P^{\nu}_{\mu,\lambda}$ at infinitely many points (namely, at all prime powers), and hence $P^{\nu}_{\mu,\lambda}$ is uniquely determined. The degree of this polynomial is at most $n(\nu) - n(\lambda) - n(\mu)$, where $n(\kappa) = \sum_i (i-1)\kappa_i$ for every partition κ. (If this is negative, then the polynomial is zero.)

These are the structure coefficients of the `(classical) Hall algebra`.

If $|\nu| \neq |\mu| + |\lambda|$, then we have $P^{\nu}_{\mu,\lambda} = 0$. More generally, if the Littlewood-Richardson coefficient $c^{\nu}_{\mu,\lambda}$ vanishes, then $P^{\nu}_{\mu,\lambda} = 0$.

The Hall polynomials satisfy the symmetry property $P^{\nu}_{\mu,\lambda} = P^{\nu}_{\lambda,\mu}$.

ALGORITHM:

If $\lambda = (1^r)$ and $|\nu| = |\mu| + |\lambda|$, then we compute $P^{\nu}_{\mu,\lambda}$ as follows (cf. Example 2.4 in [Sch2006]):

First, write $\nu = (1^{l_1}, 2^{l_2}, \ldots, n^{l_n})$, and define a sequence $r = r_0 \geq r_1 \geq \cdots \geq r_n$ such that

$$\mu = \left(1^{l_1 - r_0 + 2r_1 - r_2}, 2^{l_2 - r_1 + 2r_2 - r_3}, \ldots, (n-1)^{l_{n-1} - r_{n-2} + 2r_{n-1} - r_n}, n^{l_n - r_{n-1} + r_n}\right).$$

Thus if $\mu = (1^{m_1}, \ldots, n^{m_n})$, we have the following system of equations:

$$\begin{aligned} m_1 &= l_1 - r + 2r_1 - r_2, \\ m_2 &= l_2 - r_1 + 2r_2 - r_3, \\ &\vdots, \\ m_{n-1} &= l_{n-1} - r_{n-2} + 2r_{n-1} - r_n, \\ m_n &= l_n - r_{n-1} + r_n \end{aligned}$$

and solving for r_i and back substituting we obtain the equations:

$$\begin{aligned} r_n &= r_{n-1} + m_n - l_n, \\ r_{n-1} &= r_{n-2} + m_{n-1} - l_{n-1} + m_n - l_n, \\ &\vdots, \\ r_1 &= r + \sum_{k=1}^{n} (m_k - l_k), \end{aligned}$$

or in general we have the recursive equation:

$$r_i = r_{i-1} + \sum_{k=i}^{n} (m_k - l_k).$$

This, combined with the condition that $r_0 = r$, determines the r_i uniquely (recursively). Next we define

$$t = (r_{n-2} - r_{n-1})(l_n - r_{n-1}) + (r_{n-3} - r_{n-2})(l_{n-1} + l_n - r_{n-2}) + \cdots + (r_0 - r_1)(l_2 + \cdots + l_n - r_1),$$

and with these notations we have

$$P^{\nu}_{\mu,(1^r)} = q^t \binom{l_n}{r_{n-1}}_q \binom{l_{n-1}}{r_{n-2} - r_{n-1}}_q \cdots \binom{l_1}{r_0 - r_1}_q.$$

To compute $P^{\nu}_{\mu,\lambda}$ in general, we compute the product $I_\mu I_\lambda$ in the Hall algebra and return the coefficient of I_ν.

EXAMPLES:

```
sage: from sage.combinat.hall_polynomial import hall_polynomial
sage: hall_polynomial([1,1],[1],[1])
q + 1
sage: hall_polynomial([2],[1],[1])
1
sage: hall_polynomial([2,1],[2],[1])
q
sage: hall_polynomial([2,2,1],[2,1],[1,1])
q^2 + q
sage: hall_polynomial([2,2,2,1],[2,2,1],[1,1])
q^4 + q^3 + q^2
```

(continues on next page)

(continued from previous page)

```
sage: hall_polynomial([3,2,2,1], [3,2], [2,1])
q^6 + q^5
sage: hall_polynomial([4,2,1,1], [3,1,1], [2,1])
2*q^3 + q^2 - q - 1
sage: hall_polynomial([4,2], [2,1], [2,1], 0)
1
```

```
>>> from sage.all import *
>>> from sage.combinat.hall_polynomial import hall_polynomial
>>> hall_polynomial([Integer(1),Integer(1)],[Integer(1)],[Integer(1)])
q + 1
>>> hall_polynomial([Integer(2)],[Integer(1)],[Integer(1)])
1
>>> hall_polynomial([Integer(2),Integer(1)],[Integer(2)],[Integer(1)])
q
>>> hall_polynomial([Integer(2),Integer(2),Integer(1)],[Integer(2),Integer(1)],
↪[Integer(1),Integer(1)])
q^2 + q
>>> hall_polynomial([Integer(2),Integer(2),Integer(2),Integer(1)],[Integer(2),
↪Integer(2),Integer(1)],[Integer(1),Integer(1)])
q^4 + q^3 + q^2
>>> hall_polynomial([Integer(3),Integer(2),Integer(2),Integer(1)], [Integer(3),
↪Integer(2)], [Integer(2),Integer(1)])
q^6 + q^5
>>> hall_polynomial([Integer(4),Integer(2),Integer(1),Integer(1)], [Integer(3),
↪Integer(1),Integer(1)], [Integer(2),Integer(1)])
2*q^3 + q^2 - q - 1
>>> hall_polynomial([Integer(4),Integer(2)], [Integer(2),Integer(1)], [Integer(2),
↪Integer(1)], Integer(0))
1
```

5.1.121 The Hillman-Grassl correspondence

This module implements weak reverse plane partitions and four correspondences on them: the Hillman-Grassl correspondence and its inverse, as well as the Sulzgruber correspondence and its inverse (the Pak correspondence).

Fix a partition λ (see `Partition()`). We draw all partitions and tableaux in English notation.

A λ*-array* will mean a tableau of shape λ whose entries are nonnegative integers. (No conditions on the order of these entries are made. Note that 0 is allowed.)

A *weak reverse plane partition of shape* λ (short: λ*-rpp*) will mean a λ-array whose entries weakly increase along each row and weakly increase along each column. (The name "weak reverse plane partition" comes from Stanley in [EnumComb2] Section 7.22; other authors – such as Pak [Sulzgr2017], or Hillman and Grassl in [HilGra1976] – just call it a reverse plane partition.)

The Hillman-Grassl correspondence is a bijection from the set of λ-arrays to the set of λ-rpps. For its definition, see `hillman_grassl()`; for its inverse, see `hillman_grassl_inverse()`.

The Sulzgruber correspondence Φ_λ and the Pak correspondence ξ_λ are two further mutually inverse bijections between the set of λ-arrays and the set of λ-rpps. They appear (sometimes with different definitions, but defining the same maps) in [Pak2002], [Hopkins2017] and [Sulzgr2017]. For their definitions, see `sulzgruber_correspondence()` and `pak_correspondence()`.

EXAMPLES:

We construct a λ-rpp for $\lambda = (3, 3, 1)$ (note that λ needs not be specified explicitly):

```
sage: p = WeakReversePlanePartition([[0, 1, 3], [2, 4, 4], [3]])
sage: p.parent()
Weak Reverse Plane Partitions
```

```
>>> from sage.all import *
>>> p = WeakReversePlanePartition([[Integer(0), Integer(1), Integer(3)], [Integer(2),
→Integer(4), Integer(4)], [Integer(3)]])
>>> p.parent()
Weak Reverse Plane Partitions
```

(This is the example in Section 7.22 of [EnumComb2].)

Next, we apply the inverse of the Hillman-Grassl correspondence to it:

```
sage: HGp = p.hillman_grassl_inverse(); HGp
[[1, 2, 0], [1, 0, 1], [1]]
sage: HGp.parent()
Tableaux
```

```
>>> from sage.all import *
>>> HGp = p.hillman_grassl_inverse(); HGp
[[1, 2, 0], [1, 0, 1], [1]]
>>> HGp.parent()
Tableaux
```

This is a λ-array, encoded as a tableau. We can recover our original λ-rpp from it using the Hillman-Grassl correspondence:

```
sage: HGp.hillman_grassl() == p
True
```

```
>>> from sage.all import *
>>> HGp.hillman_grassl() == p
True
```

We can also apply the Pak correspondence to our rpp:

```
sage: Pp = p.pak_correspondence(); Pp
[[2, 0, 1], [0, 2, 0], [1]]
sage: Pp.parent()
Tableaux
```

```
>>> from sage.all import *
>>> Pp = p.pak_correspondence(); Pp
[[2, 0, 1], [0, 2, 0], [1]]
>>> Pp.parent()
Tableaux
```

This is undone by the Sulzgruber correspondence:

```
sage: Pp.sulzgruber_correspondence() == p
True
```

```
>>> from sage.all import *
>>> Pp.sulzgruber_correspondence() == p
True
```

These four correspondences can also be accessed as standalone functions (`hillman_grassl_inverse()`, `hillman_grassl()`, `pak_correspondence()` and `sulzgruber_correspondence()`) that transform lists of lists into lists of lists; this may be more efficient. For example, the above computation of `HGp` can also be obtained as follows:

```
sage: from sage.combinat.hillman_grassl import hillman_grassl_inverse
sage: HGp_bare = hillman_grassl_inverse([[0, 1, 3], [2, 4, 4], [3]])
sage: HGp_bare
[[1, 2, 0], [1, 0, 1], [1]]
sage: isinstance(HGp_bare, list)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.hillman_grassl import hillman_grassl_inverse
>>> HGp_bare = hillman_grassl_inverse([[Integer(0), Integer(1), Integer(3)],
↪[Integer(2), Integer(4), Integer(4)], [Integer(3)]])
>>> HGp_bare
[[1, 2, 0], [1, 0, 1], [1]]
>>> isinstance(HGp_bare, list)
True
```

REFERENCES:

- [Gans1981]
- [HilGra1976]
- [EnumComb2]
- [Pak2002]
- [Sulzgr2017]
- [Hopkins2017]

AUTHORS:

- Darij Grinberg and Tom Roby (2018): Initial implementation

class sage.combinat.hillman_grassl.**WeakReversePlanePartition**(*parent*, *t*)

Bases: `Tableau`

A weak reverse plane partition (short: rpp).

A weak reverse plane partition is a tableau with nonnegative entries that are weakly increasing in each row and weakly increasing in each column.

EXAMPLES:

```
sage: x = WeakReversePlanePartition([[0, 1, 1], [0, 1, 3], [1, 2, 2], [1, 2, 3],
↪[2]]); x
[[0, 1, 1], [0, 1, 3], [1, 2, 2], [1, 2, 3], [2]]
sage: x.pp()
  0  1  1
  0  1  3
```

(continues on next page)

(continued from previous page)

```
  1  2  2
  1  2  3
  2
sage: x.shape()
[3, 3, 3, 3, 1]
```

```
>>> from sage.all import *
>>> x = WeakReversePlanePartition([[Integer(0), Integer(1), Integer(1)],
↪[Integer(0), Integer(1), Integer(3)], [Integer(1), Integer(2), Integer(2)],
↪[Integer(1), Integer(2), Integer(3)], [Integer(2)]]); x
[[0, 1, 1], [0, 1, 3], [1, 2, 2], [1, 2, 3], [2]]
>>> x.pp()
  0  1  1
  0  1  3
  1  2  2
  1  2  3
  2
>>> x.shape()
[3, 3, 3, 3, 1]
```

conjugate()

Return the conjugate of `self`.

EXAMPLES:

```
sage: c = WeakReversePlanePartition([[1,1],[1,3],[2]]).conjugate(); c
[[1, 1, 2], [1, 3]]
sage: c.parent()
Weak Reverse Plane Partitions
```

```
>>> from sage.all import *
>>> c = WeakReversePlanePartition([[Integer(1),Integer(1)],[Integer(1),
↪Integer(3)],[Integer(2)]]).conjugate(); c
[[1, 1, 2], [1, 3]]
>>> c.parent()
Weak Reverse Plane Partitions
```

hillman_grassl_inverse()

Return the image of the λ-rpp `self` under the inverse of the Hillman-Grassl correspondence (as a `Tableau`).

Fix a partition λ (see `Partition()`). We draw all partitions and tableaux in English notation.

A λ-*array* will mean a tableau of shape λ whose entries are nonnegative integers. (No conditions on the order of these entries are made. Note that 0 is allowed.)

A *weak reverse plane partition of shape* λ (short: λ-*rpp*) will mean a λ-array whose entries weakly increase along each row and weakly increase along each column.

The inverse H^{-1} of the Hillman-Grassl correspondence (see (`hillman_grassl()` for the latter) sends a λ-rpp π to a λ-array $H^{-1}(\pi)$ constructed recursively as follows:

- If all entries of π are 0, then $H^{-1}(\pi) = \pi$.
- Otherwise, let s be the index of the leftmost column of π containing a nonzero entry. Write the λ-array M as $(m_{i,j})$.

- Define a sequence $((i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n))$ of boxes in the diagram of λ (actually a lattice path made of northward and eastward steps) as follows: Let (i_1, j_1) be the bottommost box in the s-th column of π. If (i_k, j_k) is defined for some $k \geq 1$, then (i_{k+1}, j_{k+1}) is constructed as follows: If q_{i_k-1, j_k} is well-defined and equals q_{i_k, j_k}, then we set $(i_{k+1}, j_{k+1}) = (i_k - 1, j_k)$. Otherwise, we set $(i_{k+1}, j_{k+1}) = (i_k, j_k + 1)$ if this is still a box of λ. Otherwise, the sequence ends here.
- Let π' be the λ-rpp obtained from π by subtracting 1 from the (i_k, j_k)-th entry of π for each $k \in \{1, 2, \ldots, n\}$.
- Let N' be the image $H^{-1}(\pi')$ (which is already constructed by recursion). Then, $H^{-1}(\pi)$ is obtained from N' by adding 1 to the (i_n, s)-th entry of N'.

This construction appears in [HilGra1976] Section 6 (where λ-arrays are re-encoded as sequences of "hook number multiplicities") and [EnumComb2] Section 7.22.

See also

hillman_grassl_inverse() for the inverse of the Hillman-Grassl correspondence as a standalone function.

hillman_grassl() for the inverse map.

EXAMPLES:

```
sage: a = WeakReversePlanePartition([[2, 2, 4], [2, 3, 4], [3, 5]])
sage: a.hillman_grassl_inverse()
[[2, 1, 1], [0, 2, 0], [1, 1]]
sage: b = WeakReversePlanePartition([[1, 1, 2, 2], [1, 1, 2, 2], [2, 2, 3, 3],
↪ [2, 2, 3, 3]])
sage: B = b.hillman_grassl_inverse(); B
[[1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1]]
sage: b.parent(), B.parent()
(Weak Reverse Plane Partitions, Tableaux)
```

```
>>> from sage.all import *
>>> a = WeakReversePlanePartition([[Integer(2), Integer(2), Integer(4)],
↪[Integer(2), Integer(3), Integer(4)], [Integer(3), Integer(5)]])
>>> a.hillman_grassl_inverse()
[[2, 1, 1], [0, 2, 0], [1, 1]]
>>> b = WeakReversePlanePartition([[Integer(1), Integer(1), Integer(2),
↪Integer(2)], [Integer(1), Integer(1), Integer(2), Integer(2)], [Integer(2),
↪Integer(2), Integer(3), Integer(3)], [Integer(2), Integer(2), Integer(3),
↪Integer(3)]])
>>> B = b.hillman_grassl_inverse(); B
[[1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1]]
>>> b.parent(), B.parent()
(Weak Reverse Plane Partitions, Tableaux)
```

Applying the inverse of the Hillman-Grassl correspondence to the transpose of a λ-rpp M yields the same result as applying it to M and then transposing the result ([Gans1981] Corollary 3.4):

```
sage: a = WeakReversePlanePartition([[1,3,5],[2,4]])
sage: aic = a.hillman_grassl_inverse().conjugate()
sage: aic == a.conjugate().hillman_grassl_inverse()
True
```

```
>>> from sage.all import *
>>> a = WeakReversePlanePartition([[Integer(1),Integer(3),Integer(5)],
↪[Integer(2),Integer(4)]])
>>> aic = a.hillman_grassl_inverse().conjugate()
>>> aic == a.conjugate().hillman_grassl_inverse()
True
```

pak_correspondence()

Return the image of the λ-rpp `self` under the Pak correspondence (as a *Tableau*).

See *hillman_grassl*.

The Pak correspondence is the map ξ_λ from [Sulzgr2017] Section 7, and is the map ξ_λ from [Pak2002] Section 4. It is the inverse of the Sulzgruber correspondence (*sulzgruber_correspondence()*). The following description of the Pak correspondence follows [Hopkins2017] (which denotes it by $\mathcal{RSK}^{-1}$):

Fix a partition λ (see *Partition()*). We draw all partitions and tableaux in English notation.

A *λ-array* will mean a tableau of shape λ whose entries are nonnegative integers. (No conditions on the order of these entries are made. Note that 0 is allowed.)

A *weak reverse plane partition of shape λ* (short: *λ-rpp*) will mean a λ-array whose entries weakly increase along each row and weakly increase along each column.

We shall also use the following notation: If (u, v) is a cell of λ, and if π is a λ-rpp, then:

- the *lower bound* of π at (u, v) (denoted by $\pi_{<(u,v)}$) is defined to be $\max\{\pi_{u-1,v}, \pi_{u,v-1}\}$ (where $\pi_{0,v}$ and $\pi_{u,0}$ are understood to mean 0).
- the *upper bound* of π at (u, v) (denoted by $\pi_{>(u,v)}$) is defined to be $\min\{\pi_{u+1,v}, \pi_{u,v+1}\}$ (where $\pi_{i,j}$ is understood to mean $+\infty$ if (i, j) is not in λ; thus, the upper bound at a corner cell is $+\infty$).
- *toggling* π at (u, v) means replacing the entry $\pi_{u,v}$ of π at (u, v) by $\pi_{<(u,v)} + \pi_{>(u,v)} - \pi_{u,v}$ (this is well-defined as long as (u, v) is not a corner of λ).

Note that every λ-rpp π and every cell (u, v) of λ satisfy $\pi_{<(u,v)} \leq \pi_{u,v} \leq \pi_{>(u,v)}$. Note that toggling a λ-rpp (at a cell that is not a corner) always results in a λ-rpp. Also, toggling is an involution).

Note also that the lower bound of π at (u, v) is defined (and finite) even when (u, v) is not a cell of λ, as long as both $(u - 1, v)$ and $(u, v - 1)$ are cells of λ.

The Pak correspondence Φ_λ sends a λ-array $M = (m_{i,j})$ to a λ-rpp $\Phi_\lambda(M)$. It is defined by recursion on λ (that is, we assume that Φ_μ is already defined for every partition μ smaller than λ), and its definition proceeds as follows:

- If $\lambda = \varnothing$, then Φ_λ is the obvious bijection sending the only $\varnothing$-array to the only $\varnothing$-rpp.
- Pick any corner $c = (i, j)$ of λ, and let μ be the result of removing this corner c from the partition λ. (The exact choice of c is immaterial.)
- Let M' be what remains of M when the corner cell c is removed.
- Let $\pi' = \Phi_\mu(M')$.
- For each positive integer k such that $(i - k, j - k)$ is a cell of λ, toggle π' at $(i - k, j - k)$. (All these togglings commute, so the order in which they are made is immaterial.)
- Extend the μ-rpp π' to a λ-rpp π by adding the cell c and writing the number $m_{i,j} - \pi'_{<(i,j)}$ into this cell.
- Set $\Phi_\lambda(M) = \pi$.

See also

`pak_correspondence()` for the Pak correspondence as a standalone function.

`sulzgruber_correspondence()` for the inverse map.

EXAMPLES:

```
sage: a = WeakReversePlanePartition([[1, 2, 3], [1, 2, 3], [2, 4, 4]])
sage: A = a.pak_correspondence(); A
[[1, 0, 2], [0, 2, 0], [1, 1, 0]]
sage: a.parent(), A.parent()
(Weak Reverse Plane Partitions, Tableaux)
```

```
>>> from sage.all import *
>>> a = WeakReversePlanePartition([[Integer(1), Integer(2), Integer(3)],␣
↪[Integer(1), Integer(2), Integer(3)], [Integer(2), Integer(4), Integer(4)]])
>>> A = a.pak_correspondence(); A
[[1, 0, 2], [0, 2, 0], [1, 1, 0]]
>>> a.parent(), A.parent()
(Weak Reverse Plane Partitions, Tableaux)
```

Applying the Pak correspondence to the transpose of a λ-rpp M yields the same result as applying it to M and then transposing the result:

```
sage: a = WeakReversePlanePartition([[1,3,5],[2,4]])
sage: acc = a.pak_correspondence().conjugate()
sage: acc == a.conjugate().pak_correspondence()
True
```

```
>>> from sage.all import *
>>> a = WeakReversePlanePartition([[Integer(1),Integer(3),Integer(5)],
↪[Integer(2),Integer(4)]])
>>> acc = a.pak_correspondence().conjugate()
>>> acc == a.conjugate().pak_correspondence()
True
```

class sage.combinat.hillman_grassl.**WeakReversePlanePartitions**

Bases: `Tableaux`

The set of all weak reverse plane partitions.

Element

alias of `WeakReversePlanePartition`

an_element()

Return a particular element of the class.

sage.combinat.hillman_grassl.**hillman_grassl**(M)

Return the image of the λ-array `M` under the Hillman-Grassl correspondence.

The Hillman-Grassl correspondence is a bijection between the tableaux with nonnegative entries (otherwise arbitrary) and the weak reverse plane partitions with nonnegative entries. This bijection preserves the shape of the tableau. See `hillman_grassl`.

See *hillman_grassl()* for a description of this map.

See also:

hillman_grassl_inverse()

EXAMPLES:

```
sage: from sage.combinat.hillman_grassl import hillman_grassl
sage: hillman_grassl([[2, 1, 1], [0, 2, 0], [1, 1]])
[[2, 2, 4], [2, 3, 4], [3, 5]]
sage: hillman_grassl([[1, 2, 0], [1, 0, 1], [1]])
[[0, 1, 3], [2, 4, 4], [3]]
sage: hillman_grassl([])
[]
sage: hillman_grassl([[3, 1, 2]])
[[3, 4, 6]]
sage: hillman_grassl([[2, 2, 0], [1, 1, 1], [1]])
[[1, 2, 4], [3, 5, 5], [4]]
sage: hillman_grassl([[1, 1, 1, 1]]*3)
[[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
```

```
>>> from sage.all import *
>>> from sage.combinat.hillman_grassl import hillman_grassl
>>> hillman_grassl([[Integer(2), Integer(1), Integer(1)], [Integer(0), Integer(2),
↪ Integer(0)], [Integer(1), Integer(1)]])
[[2, 2, 4], [2, 3, 4], [3, 5]]
>>> hillman_grassl([[Integer(1), Integer(2), Integer(0)], [Integer(1), Integer(0),
↪ Integer(1)], [Integer(1)]])
[[0, 1, 3], [2, 4, 4], [3]]
>>> hillman_grassl([])
[]
>>> hillman_grassl([[Integer(3), Integer(1), Integer(2)]])
[[3, 4, 6]]
>>> hillman_grassl([[Integer(2), Integer(2), Integer(0)], [Integer(1), Integer(1),
↪ Integer(1)], [Integer(1)]])
[[1, 2, 4], [3, 5, 5], [4]]
>>> hillman_grassl([[Integer(1), Integer(1), Integer(1), Integer(1)]]*Integer(3))
[[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
```

sage.combinat.hillman_grassl.**hillman_grassl_inverse**(M)

Return the image of the λ-rpp `M` under the inverse of the Hillman-Grassl correspondence.

See *hillman_grassl*.

See *hillman_grassl_inverse()* for a description of this map.

See also:

hillman_grassl()

EXAMPLES:

```
sage: from sage.combinat.hillman_grassl import hillman_grassl_inverse
sage: hillman_grassl_inverse([[2, 2, 4], [2, 3, 4], [3, 5]])
[[2, 1, 1], [0, 2, 0], [1, 1]]
sage: hillman_grassl_inverse([[0, 1, 3], [2, 4, 4], [3]])
[[1, 2, 0], [1, 0, 1], [1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.hillman_grassl import hillman_grassl_inverse
>>> hillman_grassl_inverse([[Integer(2), Integer(2), Integer(4)], [Integer(2),
↪Integer(3), Integer(4)], [Integer(3), Integer(5)]])
[[2, 1, 1], [0, 2, 0], [1, 1]]
>>> hillman_grassl_inverse([[Integer(0), Integer(1), Integer(3)], [Integer(2),
↪Integer(4), Integer(4)], [Integer(3)]])
[[1, 2, 0], [1, 0, 1], [1]]
```

Applying the inverse of the Hillman-Grassl correspondence to the transpose of a λ-rpp M yields the same result as applying it to M and then transposing the result ([Gans1981] Corollary 3.4):

```
sage: hillman_grassl_inverse([[1,3,5],[2,4]])
[[1, 2, 2], [1, 1]]
sage: hillman_grassl_inverse([[1,2],[3,4],[5]])
[[1, 1], [2, 1], [2]]
sage: hillman_grassl_inverse([[1, 2, 3], [1, 2, 3], [2, 4, 4]])
[[1, 2, 0], [0, 1, 1], [1, 0, 1]]
sage: hillman_grassl_inverse([[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]])
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> hillman_grassl_inverse([[Integer(1),Integer(3),Integer(5)],[Integer(2),
↪Integer(4)]])
[[1, 2, 2], [1, 1]]
>>> hillman_grassl_inverse([[Integer(1),Integer(2)],[Integer(3),Integer(4)],
↪[Integer(5)]])
[[1, 1], [2, 1], [2]]
>>> hillman_grassl_inverse([[Integer(1), Integer(2), Integer(3)], [Integer(1),
↪Integer(2), Integer(3)], [Integer(2), Integer(4), Integer(4)]])
[[1, 2, 0], [0, 1, 1], [1, 0, 1]]
>>> hillman_grassl_inverse([[Integer(1), Integer(2), Integer(3), Integer(4)],
↪[Integer(2), Integer(3), Integer(4), Integer(5)], [Integer(3), Integer(4),
↪Integer(5), Integer(6)]])
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
```

sage.combinat.hillman_grassl.**pak_correspondence**(*M*, *copy=True*)

Return the image of a λ-rpp `M` under the Pak correspondence.

The Pak correspondence is the map ξ_λ from [Sulzgr2017] Section 7, and is the map ξ_λ from [Pak2002] Section 4. It is the inverse of the Sulzgruber correspondence (`sulzgruber_correspondence()`).

See `pak_correspondence()` for a description of this map.

INPUT:

- `copy` – boolean (default: `True`); if set to `False`, the algorithm will mutate the input (but be more efficient)

EXAMPLES:

```
sage: from sage.combinat.hillman_grassl import pak_correspondence
sage: pak_correspondence([[1, 2, 3], [1, 2, 3], [2, 4, 4]])
[[1, 0, 2], [0, 2, 0], [1, 1, 0]]
sage: pak_correspondence([[1, 1, 4], [2, 3, 4], [4, 4, 4]])
[[1, 1, 2], [0, 1, 0], [3, 0, 0]]
sage: pak_correspondence([[0, 2, 3], [1, 3, 3], [2, 4]])
[[1, 0, 2], [0, 2, 0], [1, 1]]
sage: pak_correspondence([[1, 2, 4], [1, 3], [3]])
[[0, 2, 2], [1, 1], [2]]
sage: pak_correspondence([[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]])
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.hillman_grassl import pak_correspondence
>>> pak_correspondence([[Integer(1), Integer(2), Integer(3)], [Integer(1),␣
↪Integer(2), Integer(3)], [Integer(2), Integer(4), Integer(4)]])
[[1, 0, 2], [0, 2, 0], [1, 1, 0]]
>>> pak_correspondence([[Integer(1), Integer(1), Integer(4)], [Integer(2),␣
↪Integer(3), Integer(4)], [Integer(4), Integer(4), Integer(4)]])
[[1, 1, 2], [0, 1, 0], [3, 0, 0]]
>>> pak_correspondence([[Integer(0), Integer(2), Integer(3)], [Integer(1),␣
↪Integer(3), Integer(3)], [Integer(2), Integer(4)]])
[[1, 0, 2], [0, 2, 0], [1, 1]]
>>> pak_correspondence([[Integer(1), Integer(2), Integer(4)], [Integer(1),␣
↪Integer(3)], [Integer(3)]])
[[0, 2, 2], [1, 1], [2]]
>>> pak_correspondence([[Integer(1), Integer(2), Integer(3), Integer(4)],␣
↪[Integer(2), Integer(3), Integer(4), Integer(5)], [Integer(3), Integer(4),␣
↪Integer(5), Integer(6)]])
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
```

The Pak correspondence can actually be extended (by the same definition) to "rpps" of nonnegative reals rather than nonnegative integers. This implementation supports this:

```
sage: pak_correspondence([[0, 1, 3/2], [1/2, 3/2, 3/2], [1, 2]])
[[1/2, 0, 1], [0, 1, 0], [1/2, 1/2]]
```

```
>>> from sage.all import *
>>> pak_correspondence([[Integer(0), Integer(1), Integer(3)/Integer(2)],␣
↪[Integer(1)/Integer(2), Integer(3)/Integer(2), Integer(3)/Integer(2)],␣
↪[Integer(1), Integer(2)]])
[[1/2, 0, 1], [0, 1, 0], [1/2, 1/2]]
```

sage.combinat.hillman_grassl.**sulzgruber_correspondence**(M)

Return the image of a λ-array M under the Sulzgruber correspondence.

The Sulzgruber correspondence is the map Φ_λ from [Sulzgr2017] Section 7, and is the map ξ_λ^{-1} from [Pak2002] Section 5. It is denoted by $\mathcal{RSK}$ in [Hopkins2017]. It is the inverse of the Pak correspondence (*pak_correspondence()*).

See *sulzgruber_correspondence()* for a description of this map.

EXAMPLES:

```
sage: from sage.combinat.hillman_grassl import sulzgruber_correspondence
sage: sulzgruber_correspondence([[1, 0, 2], [0, 2, 0], [1, 1, 0]])
[[1, 2, 3], [1, 2, 3], [2, 4, 4]]
sage: sulzgruber_correspondence([[1, 1, 2], [0, 1, 0], [3, 0, 0]])
[[1, 1, 4], [2, 3, 4], [4, 4, 4]]
sage: sulzgruber_correspondence([[1, 0, 2], [0, 2, 0], [1, 1]])
[[0, 2, 3], [1, 3, 3], [2, 4]]
sage: sulzgruber_correspondence([[0, 2, 2], [1, 1], [2]])
[[1, 2, 4], [1, 3], [3]]
sage: sulzgruber_correspondence([[1, 1, 1, 1]]*3)
[[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
```

```
>>> from sage.all import *
>>> from sage.combinat.hillman_grassl import sulzgruber_correspondence
>>> sulzgruber_correspondence([[Integer(1), Integer(0), Integer(2)], [Integer(0),␣
↪Integer(2), Integer(0)], [Integer(1), Integer(1), Integer(0)]])
[[1, 2, 3], [1, 2, 3], [2, 4, 4]]
>>> sulzgruber_correspondence([[Integer(1), Integer(1), Integer(2)], [Integer(0),␣
↪Integer(1), Integer(0)], [Integer(3), Integer(0), Integer(0)]])
[[1, 1, 4], [2, 3, 4], [4, 4, 4]]
>>> sulzgruber_correspondence([[Integer(1), Integer(0), Integer(2)], [Integer(0),␣
↪Integer(2), Integer(0)], [Integer(1), Integer(1)]])
[[0, 2, 3], [1, 3, 3], [2, 4]]
>>> sulzgruber_correspondence([[Integer(0), Integer(2), Integer(2)], [Integer(1),␣
↪Integer(1)], [Integer(2)]])
[[1, 2, 4], [1, 3], [3]]
>>> sulzgruber_correspondence([[Integer(1), Integer(1), Integer(1),␣
↪Integer(1)]]*Integer(3))
[[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
```

The Sulzgruber correspondence can actually be extended (by the same definition) to arrays of nonnegative reals rather than nonnegative integers. This implementation supports this:

```
sage: sulzgruber_correspondence([[1/2, 0, 1], [0, 1, 0], [1/2, 1/2]])
[[0, 1, 3/2], [1/2, 3/2, 3/2], [1, 2]]
```

```
>>> from sage.all import *
>>> sulzgruber_correspondence([[Integer(1)/Integer(2), Integer(0), Integer(1)],␣
↪[Integer(0), Integer(1), Integer(0)], [Integer(1)/Integer(2), Integer(1)/
↪Integer(2)]])
[[0, 1, 3/2], [1/2, 3/2, 3/2], [1, 2]]
```

sage.combinat.hillman_grassl.**transpose**(M)

Return the transpose of a λ-array.

The transpose of a λ-array $(m_{i,j})$ is the λ^t-array $(m_{j,i})$ (where λ^t is the conjugate of the partition λ).

EXAMPLES:

```
sage: from sage.combinat.hillman_grassl import transpose
sage: transpose([[1, 2, 3], [4, 5]])
[[1, 4], [2, 5], [3]]
sage: transpose([[5, 0, 3], [4, 1, 0], [7]])
[[5, 4, 7], [0, 1], [3, 0]]
```

```
>>> from sage.all import *
>>> from sage.combinat.hillman_grassl import transpose
>>> transpose([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5)]])
[[1, 4], [2, 5], [3]]
>>> transpose([[Integer(5), Integer(0), Integer(3)], [Integer(4), Integer(1),␣
↪Integer(0)], [Integer(7)]])
[[5, 4, 7], [0, 1], [3, 0]]
```

5.1.122 Enumerated set of lists of integers with constraints: base classes

- *IntegerListsBackend*: base class for the Cython back-end of an enumerated set of lists of integers with specified constraints.
- *Envelope*: a utility class for upper (lower) envelope of a function under constraints.

class sage.combinat.integer_lists.base.**Envelope**

Bases: object

The (currently approximated) upper (lower) envelope of a function under the specified constraints.

INPUT:

- f – a function, list, or tuple; if f is a list, it is considered as the function f(i)=f[i], completed for larger i with f(i)=max_part.
- min_part, max_part, min_slope, max_slope, … as for *IntegerListsLex* (please consult for details).
- sign – (+1 or -1) multiply the input values with sign and multiply the output with sign. Setting this to -1 can be used to implement a lower envelope.

The *upper envelope* $U(f)$ of f is the (pointwise) largest function which is bounded above by f and satisfies the max_part and max_slope conditions. Furthermore, for i,i+1<min_length, the upper envelope also satisfies the min_slope condition.

Upon computing $U(f)(i)$, all the previous values for $j \leq i$ are computed and cached; in particular $f(i)$ will be computed at most once for each i.

Todo

- This class is a good candidate for Cythonization, especially to get the critical path in __call__ super fast.
- To get full envelopes, we would want both the min_slope and max_slope conditions to always be satisfied. This is only properly defined for the restriction of f to a finite interval $0, .., k$, and depends on k.
- This is the core "data structure" of IntegerListsLex. Improving the lookahead there essentially depends on having functions with a good complexity to compute the area below an envelope; and in particular how it evolves when increasing the length.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import Envelope
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_lists import Envelope
```

Trivial upper and lower envelopes:

```
sage: f = Envelope([3,2,2])
sage: [f(i) for i in range(10)]
[3, 2, 2, inf, inf, inf, inf, inf, inf, inf]
sage: f = Envelope([3,2,2], sign=-1)
sage: [f(i) for i in range(10)]
[3, 2, 2, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> f = Envelope([Integer(3),Integer(2),Integer(2)])
>>> [f(i) for i in range(Integer(10))]
[3, 2, 2, inf, inf, inf, inf, inf, inf, inf]
>>> f = Envelope([Integer(3),Integer(2),Integer(2)], sign=-Integer(1))
>>> [f(i) for i in range(Integer(10))]
[3, 2, 2, 0, 0, 0, 0, 0, 0, 0]
```

A more interesting lower envelope:

```
sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1)
sage: [f(i) for i in range(10)]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]
```

```
>>> from sage.all import *
>>> f = Envelope([Integer(4),Integer(1),Integer(5),Integer(3),Integer(5)], sign=-
↪Integer(1), min_part=Integer(2), min_slope=-Integer(1))
>>> [f(i) for i in range(Integer(10))]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]
```

Currently, adding `max_slope` has no effect:

```
sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1, max_slope=0)
sage: [f(i) for i in range(10)]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]
```

```
>>> from sage.all import *
>>> f = Envelope([Integer(4),Integer(1),Integer(5),Integer(3),Integer(5)], sign=-
↪Integer(1), min_part=Integer(2), min_slope=-Integer(1), max_slope=Integer(0))
>>> [f(i) for i in range(Integer(10))]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]
```

unless `min_length` is large enough:

```
sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1, max_slope=0,␣
↪min_length=2)
sage: [f(i) for i in range(10)]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]

sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1, max_slope=0,␣
↪min_length=4)
sage: [f(i) for i in range(10)]
[5, 5, 5, 4, 5, 4, 3, 2, 2, 2]
```

(continues on next page)

(continued from previous page)

```
sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1, max_slope=0,␣
↪min_length=5)
sage: [f(i) for i in range(10)]
[5, 5, 5, 5, 5, 4, 3, 2, 2, 2]
```

```
>>> from sage.all import *
>>> f = Envelope([Integer(4),Integer(1),Integer(5),Integer(3),Integer(5)], sign=-
↪Integer(1), min_part=Integer(2), min_slope=-Integer(1), max_slope=Integer(0),␣
↪min_length=Integer(2))
>>> [f(i) for i in range(Integer(10))]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]

>>> f = Envelope([Integer(4),Integer(1),Integer(5),Integer(3),Integer(5)], sign=-
↪Integer(1), min_part=Integer(2), min_slope=-Integer(1), max_slope=Integer(0),␣
↪min_length=Integer(4))
>>> [f(i) for i in range(Integer(10))]
[5, 5, 5, 4, 5, 4, 3, 2, 2, 2]

>>> f = Envelope([Integer(4),Integer(1),Integer(5),Integer(3),Integer(5)], sign=-
↪Integer(1), min_part=Integer(2), min_slope=-Integer(1), max_slope=Integer(0),␣
↪min_length=Integer(5))
>>> [f(i) for i in range(Integer(10))]
[5, 5, 5, 5, 5, 4, 3, 2, 2, 2]
```

A non trivial upper envelope:

```
sage: f = Envelope([9,1,5,4], max_part=7, max_slope=2)
sage: [f(i) for i in range(10)]
[7, 1, 3, 4, 6, 7, 7, 7, 7, 7]
```

```
>>> from sage.all import *
>>> f = Envelope([Integer(9),Integer(1),Integer(5),Integer(4)], max_
↪part=Integer(7), max_slope=Integer(2))
>>> [f(i) for i in range(Integer(10))]
[7, 1, 3, 4, 6, 7, 7, 7, 7, 7]
```

adapt (m, j)

Return this envelope adapted to an additional local constraint.

INPUT:

- `m` – nonnegative integer (starting value)
- `j` – nonnegative integer (position)

This method adapts this envelope to the additional local constraint imposed by having a part m at position j. Namely, this returns a function which computes, for any $i > j$, the minimum of the ceiling function and the value restriction given by the slope conditions.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import Envelope
sage: f = Envelope(3)
sage: g = f.adapt(1,1)
```

(continues on next page)

(continued from previous page)

```
sage: g is f
True
sage: [g(i) for i in range(10)]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]

sage: f = Envelope(3, max_slope=1)
sage: g = f.adapt(1,1)
sage: [g(i) for i in range(10)]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_lists import Envelope
>>> f = Envelope(Integer(3))
>>> g = f.adapt(Integer(1),Integer(1))
>>> g is f
True
>>> [g(i) for i in range(Integer(10))]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]

>>> f = Envelope(Integer(3), max_slope=Integer(1))
>>> g = f.adapt(Integer(1),Integer(1))
>>> [g(i) for i in range(Integer(10))]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
```

Note that, in both cases above, the adapted envelope is only guaranteed to be valid for $i > j$! This is to leave potential room in the future for sharing similar adapted envelopes:

```
sage: g = f.adapt(0,0)
sage: [g(i) for i in range(10)]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]

sage: g = f.adapt(2,2)
sage: [g(i) for i in range(10)]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]

sage: g = f.adapt(3,3)
sage: [g(i) for i in range(10)]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
```

```
>>> from sage.all import *
>>> g = f.adapt(Integer(0),Integer(0))
>>> [g(i) for i in range(Integer(10))]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]

>>> g = f.adapt(Integer(2),Integer(2))
>>> [g(i) for i in range(Integer(10))]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]

>>> g = f.adapt(Integer(3),Integer(3))
>>> [g(i) for i in range(Integer(10))]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
```

Now with a lower envelope:

```
sage: f = Envelope(1, sign=-1, min_slope=-1)
sage: g = f.adapt(2,2)
sage: [g(i) for i in range(10)]
[4, 3, 2, 1, 1, 1, 1, 1, 1, 1]
sage: g = f.adapt(1,3)
sage: [g(i) for i in range(10)]
[4, 3, 2, 1, 1, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> f = Envelope(Integer(1), sign=-Integer(1), min_slope=-Integer(1))
>>> g = f.adapt(Integer(2),Integer(2))
>>> [g(i) for i in range(Integer(10))]
[4, 3, 2, 1, 1, 1, 1, 1, 1, 1]
>>> g = f.adapt(Integer(1),Integer(3))
>>> [g(i) for i in range(Integer(10))]
[4, 3, 2, 1, 1, 1, 1, 1, 1, 1]
```

limit()

Return a bound on the limit of `self`.

OUTPUT: nonnegative integer or ∞

This returns some upper bound for the accumulation points of this upper envelope. For a lower envelope, a lower bound is returned instead.

In particular this gives a bound for the value of `self` at i for i large enough. Special case: for a lower envelop, and when the limit is ∞, the envelope is guaranteed to tend to ∞ instead.

When `s=self.limit_start()` is finite, this bound is guaranteed to be valid for $i >= s$.

Sometimes it's better to have a loose bound that starts early; sometimes the converse holds. At this point which specific bound and starting point is returned is not set in stone, in order to leave room for later optimizations.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import Envelope
sage: Envelope([4,1,5]).limit()
inf
sage: Envelope([4,1,5], max_part=2).limit()
2
sage: Envelope([4,1,5], max_slope=0).limit()
1
sage: Envelope(lambda x: 3, max_part=2).limit()
2
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_lists import Envelope
>>> Envelope([Integer(4),Integer(1),Integer(5)]).limit()
inf
>>> Envelope([Integer(4),Integer(1),Integer(5)], max_part=Integer(2)).limit()
2
>>> Envelope([Integer(4),Integer(1),Integer(5)], max_slope=Integer(0)).limit()
1
```

(continues on next page)

(continued from previous page)

```
>>> Envelope(lambda x: Integer(3), max_part=Integer(2)).limit()
2
```

Lower envelopes:

```
sage: Envelope(lambda x: 3, min_part=2, sign=-1).limit()
2
sage: Envelope([4,1,5], min_slope=0, sign=-1).limit()
5
sage: Envelope([4,1,5], sign=-1).limit()
0
```

```
>>> from sage.all import *
>>> Envelope(lambda x: Integer(3), min_part=Integer(2), sign=-Integer(1)).
↪limit()
2
>>> Envelope([Integer(4),Integer(1),Integer(5)], min_slope=Integer(0), sign=-
↪Integer(1)).limit()
5
>>> Envelope([Integer(4),Integer(1),Integer(5)], sign=-Integer(1)).limit()
0
```

See also

limit_start()

limit_start()

Return from which i on the bound returned by `limit` holds.

See also

limit() for the precise specifications.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import Envelope
sage: Envelope([4,1,5]).limit_start()
3
sage: Envelope([4,1,5], sign=-1).limit_start()
3

sage: Envelope([4,1,5], max_part=2).limit_start()
3

sage: Envelope(4).limit_start()
0
sage: Envelope(4, sign=-1).limit_start()
0

sage: Envelope(lambda x: 3).limit_start() == Infinity
```

(continues on next page)

(continued from previous page)

```
True
sage: Envelope(lambda x: 3, max_part=2).limit_start() == Infinity
True

sage: Envelope(lambda x: 3, sign=-1, min_part=2).limit_start() == Infinity
True
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_lists import Envelope
>>> Envelope([Integer(4),Integer(1),Integer(5)]).limit_start()
3
>>> Envelope([Integer(4),Integer(1),Integer(5)], sign=-Integer(1)).limit_
↪start()
3

>>> Envelope([Integer(4),Integer(1),Integer(5)], max_part=Integer(2)).limit_
↪start()
3

>>> Envelope(Integer(4)).limit_start()
0
>>> Envelope(Integer(4), sign=-Integer(1)).limit_start()
0

>>> Envelope(lambda x: Integer(3)).limit_start() == Infinity
True
>>> Envelope(lambda x: Integer(3), max_part=Integer(2)).limit_start() ==␣
↪Infinity
True

>>> Envelope(lambda x: Integer(3), sign=-Integer(1), min_part=Integer(2)).
↪limit_start() == Infinity
True
```

max_part

max_slope

min_slope

sign

class sage.combinat.integer_lists.base.**IntegerListsBackend**

Bases: `object`

Base class for the Cython back-end of an enumerated set of lists of integers with specified constraints.

This base implements the basic operations, including checking for containment using `_contains()`, but not iteration. For iteration, subclass this class and implement an `_iter()` method.

EXAMPLES:

```
sage: from sage.combinat.integer_lists.base import IntegerListsBackend
sage: L = IntegerListsBackend(6, max_slope=-1)
```

(continues on next page)

(continued from previous page)

```
sage: L._contains([3,2,1])
True
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_lists.base import IntegerListsBackend
>>> L = IntegerListsBackend(Integer(6), max_slope=-Integer(1))
>>> L._contains([Integer(3),Integer(2),Integer(1)])
True
```

ceiling

floor

max_length

max_part

max_slope

max_sum

min_length

min_part

min_slope

min_sum

5.1.123 Enumerated set of lists of integers with constraints: front-end

- *IntegerLists*: class which models an enumerated set of lists of integers with certain constraints. This is a Python front-end where all user-accessible functionality should be implemented.

class sage.combinat.integer_lists.lists.**IntegerList**

Bases: ClonableArray

Element class for *IntegerLists*.

check()

Check to make sure this is a valid element in its *IntegerLists* parent.

EXAMPLES:

```
sage: C = IntegerListsLex(4)
sage: C([4]).check()
True
sage: C([5]).check()
False
```

```
>>> from sage.all import *
>>> C = IntegerListsLex(Integer(4))
>>> C([Integer(4)]).check()
True
>>> C([Integer(5)]).check()
False
```

class sage.combinat.integer_lists.lists.**IntegerLists**(**args*, ***kwds*)

Bases: `Parent`

Enumerated set of lists of integers with constraints.

Currently, this is simply an abstract base class which should not be used by itself. See *IntegerListsLex* for a class which can be used by end users.

`IntegerLists` is just a Python front-end, acting as a `Parent`, supporting element classes and so on. The attribute `.backend` which is an instance of *sage.combinat.integer_lists.base.IntegerListsBackend* is the Cython back-end which implements all operations such as iteration.

The front-end (i.e. this class) and the back-end are supposed to be orthogonal: there is no imposed correspondence between front-ends and back-ends.

For example, the set of partitions of 5 and the set of weakly decreasing sequences which sum to 5 might be implemented by the same back-end, but they will be presented to the user by a different front-end.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import IntegerLists
sage: L = IntegerLists(5)
sage: L
Integer lists of sum 5 satisfying certain constraints

sage: IntegerListsLex(2, length=3, name="A given name")
A given name
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_lists import IntegerLists
>>> L = IntegerLists(Integer(5))
>>> L
Integer lists of sum 5 satisfying certain constraints

>>> IntegerListsLex(Integer(2), length=Integer(3), name="A given name")
A given name
```

Element

alias of *IntegerList*

backend = None

backend_class

alias of *IntegerListsBackend*

5.1.124 Enumerated set of lists of integers with constraints, in inverse lexicographic order

- *IntegerListsLex*: the enumerated set of lists of nonnegative integers with specified constraints, in inverse lexicographic order.
- *IntegerListsBackend_invlex*: Cython back-end for *IntegerListsLex*.

HISTORY:

This generic tool was originally written by Hivert and Thiery in MuPAD-Combinat in 2000 and ported over to Sage by Mike Hansen in 2007. It was then completely rewritten in 2015 by Gillespie, Schilling, and Thiery, with the help of many, to deal with limitations and lack of robustness w.r.t. input.

class sage.combinat.integer_lists.invlex.**IntegerListsBackend_invlex**

Bases: *IntegerListsBackend*

Cython back-end of a set of lists of integers with specified constraints enumerated in inverse lexicographic order.

check

class sage.combinat.integer_lists.invlex.**IntegerListsLex**(**args*, ***kwds*)

Bases: *IntegerLists*

Lists of nonnegative integers with constraints, in inverse lexicographic order.

An *integer list* is a list l of nonnegative integers, its *parts*. The slope (at position i) is the difference `l[i+1]-l[i]` between two consecutive parts.

This class allows to construct the set S of all integer lists l satisfying specified bounds on the sum, the length, the slope, and the individual parts, enumerated in *inverse* lexicographic order, that is from largest to smallest in lexicographic order. Note that, to admit such an enumeration, S is almost necessarily finite (see *On finiteness and inverse lexicographic enumeration*).

The main purpose is to provide a generic iteration engine for all the enumerated sets like *Partitions*, *Compositions*, *IntegerVectors*. It can also be used to generate many other combinatorial objects like Dyck paths, Motzkin paths, etc. Mathematically speaking, this is a special case of set of integral points of a polytope (or union thereof, when the length is not fixed).

INPUT:

- `min_sum` – nonnegative integer (default: 0); a lower bound on `sum(l)`
- `max_sum` – nonnegative integer or ∞ (default: ∞); an upper bound on `sum(l)`
- `n` – nonnegative integer (optional); if specified, this overrides `min_sum` and `max_sum`
- `min_length` – nonnegative integer (default: 0); a lower bound on `len(l)`
- `max_length` – nonnegative integer or ∞ (default: ∞); an upper bound on `len(l)`
- `length` – integer (optional); overrides `min_length` and `max_length` if specified
- `min_part` – nonnegative integer; a lower bounds on all parts: `min_part <= l[i]` for `0 <= i < len(l)`
- `floor` – list of nonnegative integers or a function; lower bounds on the individual parts $l[i]$

  If `floor` is a list of integers, then `floor<=l[i]` for `0 <= i < min(len(l), len(floor))`. Similarly, if `floor` is a function, then `floor(i) <= l[i]` for `0 <= i < len(l)`.
- `max_part` – nonnegative integer or ∞; an upper bound on all parts: `l[i] <= max_part` for `0 <= i < len(l)`
- `ceiling` – upper bounds on the individual parts `l[i]`; this takes the same type of input as `floor`, except that ∞ is allowed in addition to integers, and the default value is ∞.
- `min_slope` – integer or $-\infty$ (default: $-\infty$); a lower bound on the slope between consecutive parts: `min_slope <= l[i+1]-l[i]` for `0 <= i < len(l)-1`
- `max_slope` – integer or $+\infty$ (defaults: $+\infty$); an upper bound on the slope between consecutive parts: `l[i+1]-l[i] <= max_slope` for `0 <= i < len(l)-1`
- `category` – a category (default: `FiniteEnumeratedSets`)
- `check` – boolean (default: `True`); whether to display the warnings raised when functions are given as input to `floor` or `ceiling` and the errors raised when there is no proper enumeration.

- `name` – string or `None` (default: `None`); if set, this will be passed down to `Parent.rename()` to specify the name of `self`. It is recommended to use rename method directly because this feature may become deprecated.
- `element_constructor` – a function (or callable) that creates elements of `self` from a list. See also `Parent`
- `element_class` – a class for the elements of `self` (default: $ClonableArray$). This merely sets the attribute `self.Element`. See the examples for details.

Note

When several lists satisfying the constraints differ only by trailing zeroes, only the shortest one is enumerated (and therefore counted). The others are still considered valid. See the examples below.

This feature is questionable. It is recommended not to rely on it, as it may eventually be discontinued.

EXAMPLES:

We create the enumerated set of all lists of nonnegative integers of length 3 and sum 2:

```
sage: C = IntegerListsLex(2, length=3)
sage: C
Integer lists of sum 2 satisfying certain constraints
sage: C.cardinality()
6
sage: [p for p in C]
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]

sage: [2, 0, 0] in C
True
sage: [2, 0, 1] in C
False
sage: "a" in C
False
sage: ["a"] in C
False
sage: C.first()
[2, 0, 0]
```

```
>>> from sage.all import *
>>> C = IntegerListsLex(Integer(2), length=Integer(3))
>>> C
Integer lists of sum 2 satisfying certain constraints
>>> C.cardinality()
6
>>> [p for p in C]
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]

>>> [Integer(2), Integer(0), Integer(0)] in C
True
>>> [Integer(2), Integer(0), Integer(1)] in C
False
>>> "a" in C
```

(continues on next page)

(continued from previous page)

```
False
>>> ["a"] in C
False
>>> C.first()
[2, 0, 0]
```

One can specify lower and upper bounds on each part:

```
sage: list(IntegerListsLex(5, length=3, floor=[1,2,0], ceiling=[3,2,3]))
[[3, 2, 0], [2, 2, 1], [1, 2, 2]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(5), length=Integer(3), floor=[Integer(1),
↪Integer(2),Integer(0)], ceiling=[Integer(3),Integer(2),Integer(3)]))
[[3, 2, 0], [2, 2, 1], [1, 2, 2]]
```

When the length is fixed as above, one can also use *IntegerVectors*:

```
sage: IntegerVectors(2,3).list()
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

```
>>> from sage.all import *
>>> IntegerVectors(Integer(2),Integer(3)).list()
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

Using the slope condition, one can generate integer partitions (but see *Partitions*):

```
sage: list(IntegerListsLex(4, max_slope=0))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(4), max_slope=Integer(0)))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

The following is the list of all partitions of 7 with parts at least 2:

```
sage: list(IntegerListsLex(7, max_slope=0, min_part=2))
[[7], [5, 2], [4, 3], [3, 2, 2]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(7), max_slope=Integer(0), min_part=Integer(2)))
[[7], [5, 2], [4, 3], [3, 2, 2]]
```

floor and ceiling conditions

Next we list all partitions of 5 of length at most 3 which are bounded below by `[2,1,1]`:

```
sage: list(IntegerListsLex(5, max_slope=0, max_length=3, floor=[2,1,1]))
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(5), max_slope=Integer(0), max_length=Integer(3),␣
↪floor=[Integer(2),Integer(1),Integer(1)]))
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1]]
```

Note that `[5]` is considered valid, because the floor constraints only apply to existing positions in the list. To obtain instead the partitions containing `[2,1,1]`, one needs to use `min_length` or `length`:

```
sage: list(IntegerListsLex(5, max_slope=0, length=3, floor=[2,1,1]))
[[3, 1, 1], [2, 2, 1]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(5), max_slope=Integer(0), length=Integer(3),␣
↪floor=[Integer(2),Integer(1),Integer(1)]))
[[3, 1, 1], [2, 2, 1]]
```

Here is the list of all partitions of 5 which are contained in `[3,2,2]`:

```
sage: list(IntegerListsLex(5, max_slope=0, max_length=3, ceiling=[3,2,2]))
[[3, 2], [3, 1, 1], [2, 2, 1]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(5), max_slope=Integer(0), max_length=Integer(3),␣
↪ceiling=[Integer(3),Integer(2),Integer(2)]))
[[3, 2], [3, 1, 1], [2, 2, 1]]
```

This is the list of all compositions of 4 (but see *Compositions*):

```
sage: list(IntegerListsLex(4, min_part=1))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(4), min_part=Integer(1)))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

This is the list of all integer vectors of sum 4 and length 3:

```
sage: list(IntegerListsLex(4, length=3))
[[4, 0, 0], [3, 1, 0], [3, 0, 1], [2, 2, 0], [2, 1, 1],
 [2, 0, 2], [1, 3, 0], [1, 2, 1], [1, 1, 2], [1, 0, 3],
 [0, 4, 0], [0, 3, 1], [0, 2, 2], [0, 1, 3], [0, 0, 4]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(4), length=Integer(3)))
[[4, 0, 0], [3, 1, 0], [3, 0, 1], [2, 2, 0], [2, 1, 1],
 [2, 0, 2], [1, 3, 0], [1, 2, 1], [1, 1, 2], [1, 0, 3],
 [0, 4, 0], [0, 3, 1], [0, 2, 2], [0, 1, 3], [0, 0, 4]]
```

For whatever it is worth, the `floor` and `min_part` constraints can be combined:

```
sage: L = IntegerListsLex(5, floor=[2,0,2], min_part=1)
sage: L.list()
[[5], [4, 1], [3, 2], [2, 3], [2, 1, 2]]
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(Integer(5), floor=[Integer(2),Integer(0),Integer(2)], min_
↪part=Integer(1))
>>> L.list()
[[5], [4, 1], [3, 2], [2, 3], [2, 1, 2]]
```

This is achieved by updating the floor upon constructing `L`:

```
sage: [L.floor(i) for i in range(5)]
[2, 1, 2, 1, 1]
```

```
>>> from sage.all import *
>>> [L.floor(i) for i in range(Integer(5))]
[2, 1, 2, 1, 1]
```

Similarly, the `ceiling` and `max_part` constraints can be combined:

```
sage: L = IntegerListsLex(4, ceiling=[2,3,1], max_part=2, length=3)
sage: L.list()
[[2, 2, 0], [2, 1, 1], [1, 2, 1]]
sage: [L.ceiling(i) for i in range(5)]
[2, 2, 1, 2, 2]
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(Integer(4), ceiling=[Integer(2),Integer(3),Integer(1)],␣
↪max_part=Integer(2), length=Integer(3))
>>> L.list()
[[2, 2, 0], [2, 1, 1], [1, 2, 1]]
>>> [L.ceiling(i) for i in range(Integer(5))]
[2, 2, 1, 2, 2]
```

This can be used to generate Motzkin words (see Wikipedia article Motzkin_number):

```
sage: def motzkin_words(n):
....:     return IntegerListsLex(length=n+1, min_slope=-1, max_slope=1,
....:                 ceiling=[0]+[+oo for i in range(n-1)]+[0])
sage: motzkin_words(4).list()
[[0, 1, 2, 1, 0],
 [0, 1, 1, 1, 0],
 [0, 1, 1, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 1, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0]]
sage: [motzkin_words(n).cardinality() for n in range(8)]
[1, 1, 2, 4, 9, 21, 51, 127]
sage: oeis(_)                  # optional -- internet
0: A001006: Motzkin numbers: number of ways of drawing any number
of nonintersecting chords joining n (labeled) points on a circle.
1: ...
2: ...
```

```
>>> from sage.all import *
>>> def motzkin_words(n):
...     return IntegerListsLex(length=n+Integer(1), min_slope=-Integer(1), max_
↪slope=Integer(1),
...                  ceiling=[Integer(0)]+[+oo for i in range(n-
↪Integer(1))]+[Integer(0)])
>>> motzkin_words(Integer(4)).list()
[[0, 1, 2, 1, 0],
 [0, 1, 1, 1, 0],
 [0, 1, 1, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 1, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0]]
>>> [motzkin_words(n).cardinality() for n in range(Integer(8))]
[1, 1, 2, 4, 9, 21, 51, 127]
>>> oeis(_)                    # optional -- internet
0: A001006: Motzkin numbers: number of ways of drawing any number
of nonintersecting chords joining n (labeled) points on a circle.
1: ...
2: ...
```

or Dyck words (see also *DyckWords*), through the bijection with paths from $(0,0)$ to (n,n) with left and up steps that remain below the diagonal:

```
sage: def dyck_words(n):
....:     return IntegerListsLex(length=n, ceiling=list(range(n+1)), min_slope=0)
sage: [dyck_words(n).cardinality() for n in range(8)]
[1, 1, 2, 5, 14, 42, 132, 429]
sage: dyck_words(3).list()
[[0, 1, 2], [0, 1, 1], [0, 0, 2], [0, 0, 1], [0, 0, 0]]
```

```
>>> from sage.all import *
>>> def dyck_words(n):
...     return IntegerListsLex(length=n, ceiling=list(range(n+Integer(1))), min_
↪slope=Integer(0))
>>> [dyck_words(n).cardinality() for n in range(Integer(8))]
[1, 1, 2, 5, 14, 42, 132, 429]
>>> dyck_words(Integer(3)).list()
[[0, 1, 2], [0, 1, 1], [0, 0, 2], [0, 0, 1], [0, 0, 0]]
```

On finiteness and inverse lexicographic enumeration

The set of all lists of integers cannot be enumerated in inverse lexicographic order, since there is no largest list (take $[n]$ for n as large as desired):

```
sage: IntegerListsLex().first()
Traceback (most recent call last):
...
ValueError: could not prove that the specified constraints yield a finite set
```

```
>>> from sage.all import *
>>> IntegerListsLex().first()
Traceback (most recent call last):
...
ValueError: could not prove that the specified constraints yield a finite set
```

Here is a variant which could be enumerated in lexicographic order but not in inverse lexicographic order:

```
sage: L = IntegerListsLex(length=2, ceiling=[Infinity, 0], floor=[0,1])
sage: for l in L: print(l)
Traceback (most recent call last):
...
ValueError: infinite upper bound for values of m
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(length=Integer(2), ceiling=[Infinity, Integer(0)],␣
↪floor=[Integer(0),Integer(1)])
>>> for l in L: print(l)
Traceback (most recent call last):
...
ValueError: infinite upper bound for values of m
```

Even when the sum is specified, it is not necessarily possible to enumerate *all* elements in inverse lexicographic order. In the following example, the list `[1, 1, 1]` will never appear in the enumeration:

```
sage: IntegerListsLex(3).first()
Traceback (most recent call last):
...
ValueError: could not prove that the specified constraints yield a finite set
```

```
>>> from sage.all import *
>>> IntegerListsLex(Integer(3)).first()
Traceback (most recent call last):
...
ValueError: could not prove that the specified constraints yield a finite set
```

If one wants to proceed anyway, one can sign a waiver by setting `check=False` (again, be warned that some valid lists may never appear):

```
sage: L = IntegerListsLex(3, check=False)
sage: it = iter(L)
sage: [next(it) for i in range(6)]
[[3], [2, 1], [2, 0, 1], [2, 0, 0, 1], [2, 0, 0, 0, 1], [2, 0, 0, 0, 0, 1]]
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(Integer(3), check=False)
>>> it = iter(L)
>>> [next(it) for i in range(Integer(6))]
[[3], [2, 1], [2, 0, 1], [2, 0, 0, 1], [2, 0, 0, 0, 1], [2, 0, 0, 0, 0, 1]]
```

In fact, being inverse lexicographically enumerable is almost equivalent to being finite. The only infinity that can occur would be from a tail of numbers $0, 1$ as in the previous example, where the 1 moves further and further to the right. If there is any list that is inverse lexicographically smaller than such a configuration, the iterator would not reach it and hence would not be considered iterable. Given that the infinite cases are very specific, at this point only the finite cases are supported (without signing the waiver).

The finiteness detection is not complete yet, so some finite cases may not be supported either, at least not without disabling the checks. Practical examples of such are welcome.

On trailing zeroes, and their caveats

As mentioned above, when several lists satisfying the constraints differ only by trailing zeroes, only the shortest one is listed:

```
sage: L = IntegerListsLex(max_length=4, max_part=1)
sage: L.list()
[[1, 1, 1, 1],
 [1, 1, 1],
 [1, 1, 0, 1],
 [1, 1],
 [1, 0, 1, 1],
 [1, 0, 1],
 [1, 0, 0, 1],
 [1],
 [0, 1, 1, 1],
 [0, 1, 1],
 [0, 1, 0, 1],
 [0, 1],
 [0, 0, 1, 1],
 [0, 0, 1],
 [0, 0, 0, 1],
 []]
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(max_length=Integer(4), max_part=Integer(1))
>>> L.list()
[[1, 1, 1, 1],
 [1, 1, 1],
 [1, 1, 0, 1],
 [1, 1],
 [1, 0, 1, 1],
 [1, 0, 1],
 [1, 0, 0, 1],
 [1],
 [0, 1, 1, 1],
 [0, 1, 1],
 [0, 1, 0, 1],
 [0, 1],
 [0, 0, 1, 1],
 [0, 0, 1],
 [0, 0, 0, 1],
 []]
```

and counted:

```
sage: L.cardinality()
16
```

```
>>> from sage.all import *
>>> L.cardinality()
16
```

Still, the others are considered as elements of L:

```
sage: L = IntegerListsLex(4,min_length=3,max_length=4)
sage: L.list()
[..., [2, 2, 0], ...]

sage: [2, 2, 0] in L       # in L.list()
True
sage: [2, 2, 0, 0] in L    # not in L.list() !
True
sage: [2, 2, 0, 0, 0] in L
False
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(Integer(4),min_length=Integer(3),max_length=Integer(4))
>>> L.list()
[..., [2, 2, 0], ...]

>>> [Integer(2), Integer(2), Integer(0)] in L       # in L.list()
True
>>> [Integer(2), Integer(2), Integer(0), Integer(0)] in L    # not in L.list() !
True
>>> [Integer(2), Integer(2), Integer(0), Integer(0), Integer(0)] in L
False
```

Specifying functions as input for the floor or ceiling

We construct all lists of sum 4 and length 4 such that `l[i] <= i`:

```
sage: list(IntegerListsLex(4, length=4, ceiling=lambda i: i, check=False))
[[0, 1, 2, 1], [0, 1, 1, 2], [0, 1, 0, 3], [0, 0, 2, 2], [0, 0, 1, 3]]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(4), length=Integer(4), ceiling=lambda i: i,␣
→check=False))
[[0, 1, 2, 1], [0, 1, 1, 2], [0, 1, 0, 3], [0, 0, 2, 2], [0, 0, 1, 3]]
```

Warning

When passing a function as `floor` or `ceiling`, it may become undecidable to detect improper inverse lexicographic enumeration. For example, the following example has a finite enumeration:

```
sage: L = IntegerListsLex(3, floor=lambda i: 1 if i>=2 else 0, check=False)
sage: L.list()
[[3],
 [2, 1],
 [2, 0, 1],
 [1, 2],
```

```
 [1, 1, 1],
 [1, 0, 2],
 [1, 0, 1, 1],
 [0, 3],
 [0, 2, 1],
 [0, 1, 2],
 [0, 1, 1, 1],
 [0, 0, 3],
 [0, 0, 2, 1],
 [0, 0, 1, 2],
 [0, 0, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(Integer(3), floor=lambda i: Integer(1) if i>
↪=Integer(2) else Integer(0), check=False)
>>> L.list()
[[3],
 [2, 1],
 [2, 0, 1],
 [1, 2],
 [1, 1, 1],
 [1, 0, 2],
 [1, 0, 1, 1],
 [0, 3],
 [0, 2, 1],
 [0, 1, 2],
 [0, 1, 1, 1],
 [0, 0, 3],
 [0, 0, 2, 1],
 [0, 0, 1, 2],
 [0, 0, 1, 1, 1]]
```

but one cannot decide whether the following has an improper inverse lexicographic enumeration without computing the floor all the way to `Infinity`:

```
sage: L = IntegerListsLex(3, floor=lambda i: 0, check=False)
sage: it = iter(L)
sage: [next(it) for i in range(6)]
[[3], [2, 1], [2, 0, 1], [2, 0, 0, 1], [2, 0, 0, 0, 1], [2, 0, 0, 0, 0, 1]]
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(Integer(3), floor=lambda i: Integer(0), check=False)
>>> it = iter(L)
>>> [next(it) for i in range(Integer(6))]
[[3], [2, 1], [2, 0, 1], [2, 0, 0, 1], [2, 0, 0, 0, 1], [2, 0, 0, 0, 0, 1]]
```

Hence a warning is raised when a function is specified as input, unless the waiver is signed by setting `check=False`:

```
sage: L = IntegerListsLex(3, floor=lambda i: 1 if i>=2 else 0)
doctest:...
A function has been given as input of the floor=[...] or ceiling=[...]
arguments of IntegerListsLex. Please see the documentation for the caveats.
If you know what you are doing, you can set check=False to skip this warning.
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(Integer(3), floor=lambda i: Integer(1) if i>
↪=Integer(2) else Integer(0))
doctest:...
A function has been given as input of the floor=[...] or ceiling=[...]
arguments of IntegerListsLex. Please see the documentation for the caveats.
If you know what you are doing, you can set check=False to skip this warning.
```

Similarly, the algorithm may need to search forever for a solution when the ceiling is ultimately zero:

```
sage: L = IntegerListsLex(2,ceiling=lambda i:0, check=False)
sage: L.first()           # not tested: will hang forever
sage: L = IntegerListsLex(2,ceiling=lambda i:0 if i<20 else 1, check=False)
sage: it = iter(L)
sage: next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
sage: next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]
sage: next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1]
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(Integer(2),ceiling=lambda i:Integer(0), check=False)
>>> L.first()           # not tested: will hang forever
>>> L = IntegerListsLex(Integer(2),ceiling=lambda i:Integer(0) if i
↪<Integer(20) else Integer(1), check=False)
>>> it = iter(L)
>>> next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
>>> next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]
>>> next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1]
```

Tip: using disjoint union enumerated sets for additional flexibility

Sometimes, specifying a range for the sum or the length may be too restrictive. One would want instead to specify a list, or iterable L, of acceptable values. This is easy to achieve using a `disjoint union of enumerated sets`. Here we want to accept the values $n = 0, 2, 3$:

```
sage: C = DisjointUnionEnumeratedSets(Family([0,2,3],
....:         lambda n: IntegerListsLex(n, length=2)))
sage: C
Disjoint union of Finite family
{0: Integer lists of sum 0 satisfying certain constraints,
 2: Integer lists of sum 2 satisfying certain constraints,
 3: Integer lists of sum 3 satisfying certain constraints}
sage: C.list()
[[0, 0],
 [2, 0], [1, 1], [0, 2],
 [3, 0], [2, 1], [1, 2], [0, 3]]
```

```
>>> from sage.all import *
>>> C = DisjointUnionEnumeratedSets(Family([Integer(0),Integer(2),Integer(3)],
...         lambda n: IntegerListsLex(n, length=Integer(2))))
>>> C
Disjoint union of Finite family
{0: Integer lists of sum 0 satisfying certain constraints,
 2: Integer lists of sum 2 satisfying certain constraints,
 3: Integer lists of sum 3 satisfying certain constraints}
>>> C.list()
[[0, 0],
 [2, 0], [1, 1], [0, 2],
 [3, 0], [2, 1], [1, 2], [0, 3]]
```

The price to pay is that the enumeration order is now *graded lexicographic* instead of lexicographic: first choose the value according to the order specified by L, and use lexicographic order within each value. Here is we reverse L:

```
sage: DisjointUnionEnumeratedSets(Family([3,2,0],
....:     lambda n: IntegerListsLex(n, length=2))).list()
[[3, 0], [2, 1], [1, 2], [0, 3],
 [2, 0], [1, 1], [0, 2],
 [0, 0]]
```

```
>>> from sage.all import *
>>> DisjointUnionEnumeratedSets(Family([Integer(3),Integer(2),Integer(0)],
...     lambda n: IntegerListsLex(n, length=Integer(2)))).list()
[[3, 0], [2, 1], [1, 2], [0, 3],
 [2, 0], [1, 1], [0, 2],
 [0, 0]]
```

Note that if a given value appears several times, the corresponding elements will be enumerated several times, which may, or not, be what one wants:

```
sage: DisjointUnionEnumeratedSets(Family([2,2],
....:     lambda n: IntegerListsLex(n, length=2))).list()
[[2, 0], [1, 1], [0, 2], [2, 0], [1, 1], [0, 2]]
```

```
>>> from sage.all import *
>>> DisjointUnionEnumeratedSets(Family([Integer(2),Integer(2)],
...     lambda n: IntegerListsLex(n, length=Integer(2)))).list()
[[2, 0], [1, 1], [0, 2], [2, 0], [1, 1], [0, 2]]
```

Here is a variant where we specify acceptable values for the length:

```
sage: DisjointUnionEnumeratedSets(Family([0,1,3],
....:     lambda l: IntegerListsLex(2, length=l))).list()
[[2],
 [2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

```
>>> from sage.all import *
>>> DisjointUnionEnumeratedSets(Family([Integer(0),Integer(1),Integer(3)],
...     lambda l: IntegerListsLex(Integer(2), length=l))).list()
```

(continues on next page)

(continued from previous page)

```
[[2],
 [2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

This technique can also be useful to obtain a proper enumeration on infinite sets by using a graded lexicographic enumeration:

```
sage: C = DisjointUnionEnumeratedSets(Family(NN,
....:             lambda n: IntegerListsLex(n, length=2)))
sage: C
Disjoint union of Lazy family (<lambda>(i))_{i in Non negative integer semiring}
sage: it = iter(C)
sage: [next(it) for i in range(10)]
[[0, 0],
 [1, 0], [0, 1],
 [2, 0], [1, 1], [0, 2],
 [3, 0], [2, 1], [1, 2], [0, 3]]
```

```
>>> from sage.all import *
>>> C = DisjointUnionEnumeratedSets(Family(NN,
...             lambda n: IntegerListsLex(n, length=Integer(2))))
>>> C
Disjoint union of Lazy family (<lambda>(i))_{i in Non negative integer semiring}
>>> it = iter(C)
>>> [next(it) for i in range(Integer(10))]
[[0, 0],
 [1, 0], [0, 1],
 [2, 0], [1, 1], [0, 2],
 [3, 0], [2, 1], [1, 2], [0, 3]]
```

Specifying how to construct elements

This is the list of all monomials of degree 4 which divide the monomial $x^3y^1z^2$ (a monomial being identified with its exponent vector):

```
sage: R.<x,y,z> = QQ[]
sage: m = [3,1,2]
sage: def term(exponents):
....:     return x^exponents[0] * y^exponents[1] * z^exponents[2]
sage: list( IntegerListsLex(4, length=len(m), ceiling=m, element_
↪constructor=term) )
[x^3*y, x^3*z, x^2*y*z, x^2*z^2, x*y*z^2]
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> m = [Integer(3),Integer(1),Integer(2)]
>>> def term(exponents):
...     return x**exponents[Integer(0)] * y**exponents[Integer(1)] *␣
↪z**exponents[Integer(2)]
>>> list( IntegerListsLex(Integer(4), length=len(m), ceiling=m, element_
↪constructor=term) )
[x^3*y, x^3*z, x^2*y*z, x^2*z^2, x*y*z^2]
```

Note the use of the `element_constructor` option to specify how to construct elements from a plain list.

A variant is to specify a class for the elements. With the default element constructor, this class should take as input the parent `self` and a list.

> **Warning**
>
> The protocol for specifying the element class and constructor is subject to changes.

ALGORITHM:

The iteration algorithm uses a depth first search through the prefix tree of the list of integers (see also *Lexicographic generation of lists of integers*). While doing so, it does some lookahead heuristics to attempt to cut dead branches.

In most practical use cases, most dead branches are cut. Then, roughly speaking, the time needed to iterate through all the elements of S is proportional to the number of elements, where the proportion factor is controlled by the length l of the longest element of S. In addition, the memory usage is also controlled by l, which is to say negligible in practice.

Still, there remains much room for efficiency improvements; see Issue #18055, Issue #18056.

> **Note**
>
> The generation algorithm could in principle be extended to deal with non-constant slope constraints and with negative parts.

TESTS from comments on Issue #17979

Comment 191:

```
sage: list(IntegerListsLex(1, min_length=2, min_slope=0, max_slope=0))
[]
```

```
>>> from sage.all import *
>>> list(IntegerListsLex(Integer(1), min_length=Integer(2), min_slope=Integer(0),␣
↪max_slope=Integer(0)))
[]
```

Comment 240:

```
sage: L = IntegerListsLex(min_length=2, max_part=0)
sage: L.list()
[[0, 0]]
```

```
>>> from sage.all import *
>>> L = IntegerListsLex(min_length=Integer(2), max_part=Integer(0))
>>> L.list()
[[0, 0]]
```

Tests on the element constructor feature and mutability

Internally, the iterator works on a single list that is mutated along the way. Therefore, you need to make sure that the `element_constructor` actually **copies** its input. This example shows what can go wrong:

```
sage: P = IntegerListsLex(n=3, max_slope=0, min_part=1, element_
↪constructor=lambda x: x)
sage: list(P)
[[], [], []]
```

```
>>> from sage.all import *
>>> P = IntegerListsLex(n=Integer(3), max_slope=Integer(0), min_part=Integer(1),␣
↪element_constructor=lambda x: x)
>>> list(P)
[[], [], []]
```

However, specifying `list()` as constructor solves this problem:

```
sage: P = IntegerListsLex(n=3, max_slope=0, min_part=1, element_constructor=list)
sage: list(P)
[[3], [2, 1], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> P = IntegerListsLex(n=Integer(3), max_slope=Integer(0), min_part=Integer(1),␣
↪element_constructor=list)
>>> list(P)
[[3], [2, 1], [1, 1, 1]]
```

Same, step by step:

```
sage: it = iter(P)
sage: a = next(it); a
[3]
sage: b = next(it); b
[2, 1]
sage: a
[3]
sage: a is b
False
```

```
>>> from sage.all import *
>>> it = iter(P)
>>> a = next(it); a
[3]
>>> b = next(it); b
[2, 1]
>>> a
[3]
>>> a is b
False
```

Tests from MuPAD-Combinat:

```
sage: IntegerListsLex(7, min_length=2, max_length=6, floor=[0,0,2,0,0,1],␣
↪ceiling=[3,2,3,2,1,2]).cardinality()
83
sage: IntegerListsLex(7, min_length=2, max_length=6, floor=[0,0,2,0,1,1],␣
```

(continues on next page)

(continued from previous page)

```
↪ceiling=[3,2,3,2,1,2]).cardinality()
53
sage: IntegerListsLex(5, min_length=2, max_length=6, floor=[0,0,2,0,0,0],␣
↪ceiling=[2,2,2,2,2,2]).cardinality()
30
sage: IntegerListsLex(5, min_length=2, max_length=6, floor=[0,0,1,1,0,0],␣
↪ceiling=[2,2,2,2,2,2]).cardinality()
43

sage: IntegerListsLex(0, min_length=0, max_length=7, floor=[1,1,0,0,1,0],␣
↪ceiling=[4,3,2,3,2,2,1]).first()
[]

sage: IntegerListsLex(0, min_length=1, max_length=7, floor=[0,1,0,0,1,0],␣
↪ceiling=[4,3,2,3,2,2,1]).first()
[0]
sage: IntegerListsLex(0, min_length=1, max_length=7, floor=[1,1,0,0,1,0],␣
↪ceiling=[4,3,2,3,2,2,1]).cardinality()
0

sage: IntegerListsLex(2, min_length=0, max_length=7, floor=[1,1,0,0,0,0],␣
↪ceiling=[4,3,2,3,2,2,1]).first()  # Was [1,1], due to slightly different specs
[2]
sage: IntegerListsLex(1, min_length=1, max_length=7, floor=[1,1,0,0,0,0],␣
↪ceiling=[4,3,2,3,2,2,1]).first()
[1]
sage: IntegerListsLex(1, min_length=2, max_length=7, floor=[1,1,0,0,0,0],␣
↪ceiling=[4,3,2,3,2,2,1]).cardinality()
0
sage: IntegerListsLex(2, min_length=5, max_length=7, floor=[1,1,0,0,0,0],␣
↪ceiling=[4,3,2,3,2,2,1]).first()
[1, 1, 0, 0, 0]
sage: IntegerListsLex(2, min_length=5, max_length=7, floor=[1,1,0,0,0,1],␣
↪ceiling=[4,3,2,3,2,2,1]).first()
[1, 1, 0, 0, 0]
sage: IntegerListsLex(2, min_length=5, max_length=7, floor=[1,1,0,0,1,0],␣
↪ceiling=[4,3,2,3,2,2,1]).cardinality()
0

sage: IntegerListsLex(4, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1],␣
↪ceiling=[3, 1, 2, 3, 2, 2]).cardinality()
0
sage: IntegerListsLex(5, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1],␣
↪ceiling=[3, 1, 2, 3, 2, 2]).first()
[2, 1, 2]
sage: IntegerListsLex(6, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1],␣
↪ceiling=[3, 1, 2, 3, 2, 2]).first()
[3, 1, 2]
sage: IntegerListsLex(12, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1],␣
↪ceiling=[3, 1, 2, 3, 2, 2]).first()
[3, 1, 2, 3, 2, 1]
sage: IntegerListsLex(13, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1],␣
```

(continues on next page)

(continued from previous page)

```
↪ceiling=[3, 1, 2, 3, 2, 2]).first()
[3, 1, 2, 3, 2, 2]
sage: IntegerListsLex(14, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1],␣
↪ceiling=[3, 1, 2, 3, 2, 2]).cardinality()
0
```

```
>>> from sage.all import *
>>> IntegerListsLex(Integer(7), min_length=Integer(2), max_length=Integer(6),␣
↪floor=[Integer(0),Integer(0),Integer(2),Integer(0),Integer(0),Integer(1)],␣
↪ceiling=[Integer(3),Integer(2),Integer(3),Integer(2),Integer(1),Integer(2)]).
↪cardinality()
83
>>> IntegerListsLex(Integer(7), min_length=Integer(2), max_length=Integer(6),␣
↪floor=[Integer(0),Integer(0),Integer(2),Integer(0),Integer(1),Integer(1)],␣
↪ceiling=[Integer(3),Integer(2),Integer(3),Integer(2),Integer(1),Integer(2)]).
↪cardinality()
53
>>> IntegerListsLex(Integer(5), min_length=Integer(2), max_length=Integer(6),␣
↪floor=[Integer(0),Integer(0),Integer(2),Integer(0),Integer(0),Integer(0)],␣
↪ceiling=[Integer(2),Integer(2),Integer(2),Integer(2),Integer(2),Integer(2)]).
↪cardinality()
30
>>> IntegerListsLex(Integer(5), min_length=Integer(2), max_length=Integer(6),␣
↪floor=[Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),Integer(0)],␣
↪ceiling=[Integer(2),Integer(2),Integer(2),Integer(2),Integer(2),Integer(2)]).
↪cardinality()
43

>>> IntegerListsLex(Integer(0), min_length=Integer(0), max_length=Integer(7),␣
↪floor=[Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),Integer(0)],␣
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).first()
[]

>>> IntegerListsLex(Integer(0), min_length=Integer(1), max_length=Integer(7),␣
↪floor=[Integer(0),Integer(1),Integer(0),Integer(0),Integer(1),Integer(0)],␣
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).first()
[0]
>>> IntegerListsLex(Integer(0), min_length=Integer(1), max_length=Integer(7),␣
↪floor=[Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),Integer(0)],␣
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).cardinality()
0

>>> IntegerListsLex(Integer(2), min_length=Integer(0), max_length=Integer(7),␣
↪floor=[Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)],␣
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).first()  # Was [1,1], due to slightly different specs
[2]
>>> IntegerListsLex(Integer(1), min_length=Integer(1), max_length=Integer(7),␣
↪floor=[Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)],␣
```

(continues on next page)

(continued from previous page)

```
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).first()
[1]
>>> IntegerListsLex(Integer(1), min_length=Integer(2), max_length=Integer(7),␣
↪floor=[Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)],␣
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).cardinality()
0
>>> IntegerListsLex(Integer(2), min_length=Integer(5), max_length=Integer(7),␣
↪floor=[Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)],␣
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).first()
[1, 1, 0, 0, 0]
>>> IntegerListsLex(Integer(2), min_length=Integer(5), max_length=Integer(7),␣
↪floor=[Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(1)],␣
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).first()
[1, 1, 0, 0, 0]
>>> IntegerListsLex(Integer(2), min_length=Integer(5), max_length=Integer(7),␣
↪floor=[Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),Integer(0)],␣
↪ceiling=[Integer(4),Integer(3),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]).cardinality()
0

>>> IntegerListsLex(Integer(4), min_length=Integer(3), max_length=Integer(6),␣
↪floor=[Integer(2), Integer(1), Integer(2), Integer(1), Integer(1), Integer(1)],␣
↪ceiling=[Integer(3), Integer(1), Integer(2), Integer(3), Integer(2),␣
↪Integer(2)]).cardinality()
0
>>> IntegerListsLex(Integer(5), min_length=Integer(3), max_length=Integer(6),␣
↪floor=[Integer(2), Integer(1), Integer(2), Integer(1), Integer(1), Integer(1)],␣
↪ceiling=[Integer(3), Integer(1), Integer(2), Integer(3), Integer(2),␣
↪Integer(2)]).first()
[2, 1, 2]
>>> IntegerListsLex(Integer(6), min_length=Integer(3), max_length=Integer(6),␣
↪floor=[Integer(2), Integer(1), Integer(2), Integer(1), Integer(1), Integer(1)],␣
↪ceiling=[Integer(3), Integer(1), Integer(2), Integer(3), Integer(2),␣
↪Integer(2)]).first()
[3, 1, 2]
>>> IntegerListsLex(Integer(12), min_length=Integer(3), max_length=Integer(6),␣
↪floor=[Integer(2), Integer(1), Integer(2), Integer(1), Integer(1), Integer(1)],␣
↪ceiling=[Integer(3), Integer(1), Integer(2), Integer(3), Integer(2),␣
↪Integer(2)]).first()
[3, 1, 2, 3, 2, 1]
>>> IntegerListsLex(Integer(13), min_length=Integer(3), max_length=Integer(6),␣
↪floor=[Integer(2), Integer(1), Integer(2), Integer(1), Integer(1), Integer(1)],␣
↪ceiling=[Integer(3), Integer(1), Integer(2), Integer(3), Integer(2),␣
↪Integer(2)]).first()
[3, 1, 2, 3, 2, 2]
>>> IntegerListsLex(Integer(14), min_length=Integer(3), max_length=Integer(6),␣
↪floor=[Integer(2), Integer(1), Integer(2), Integer(1), Integer(1), Integer(1)],␣
↪ceiling=[Integer(3), Integer(1), Integer(2), Integer(3), Integer(2),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)]).cardinality()
0
```

This used to hang (see comment 389 and fix in `Envelope.__init__()`):

```
sage: IntegerListsLex(7, max_part=0, ceiling=lambda i:i, check=False).list()
[]
```

```
>>> from sage.all import *
>>> IntegerListsLex(Integer(7), max_part=Integer(0), ceiling=lambda i:i,␣
↪check=False).list()
[]
```

backend_class

alias of *IntegerListsBackend_invlex*

class sage.combinat.integer_lists.invlex.**IntegerListsLexIter**(*backend*)

Bases: `object`

Iterator class for IntegerListsLex.

Let `T` be the prefix tree of all lists of nonnegative integers that satisfy all constraints except possibly for `min_length` and `min_sum`; let the children of a list be sorted decreasingly according to their last part.

The iterator is based on a depth-first search exploration of a subtree of this tree, trying to cut branches that do not contain a valid list. Each call of `next` iterates through the nodes of this tree until it finds a valid list to return.

Here are the attributes describing the current state of the iterator, and their invariants:

- `backend` – the `IntegerListsBackend` object this is iterating on;
- `_current_list` – the list corresponding to the current node of the tree;
- `_j` – the index of the last element of `_current_list`: `self._j == len(self._current_list) - 1`;
- `_current_sum` – the sum of the parts of `_current_list`;
- `_search_ranges` – list of same length as `_current_list`: the range for each part

Furthermore, we assume that there is no obvious contradiction in the constraints:

- `self.backend.min_length <= self.backend.max_length`;
- `self.backend.min_slope <= self.backend.max_slope` unless `self.backend.min_length <= 1`.

Along this iteration, `next` switches between the following states:

- LOOKAHEAD: determine whether the current list could be a prefix of a valid list;
- PUSH: go deeper into the prefix tree by appending the largest possible part to the current list;
- ME: check whether the current list is valid and if yes return it
- DECREASE: decrease the last part;
- POP: pop the last part of the current list;
- STOP: the iteration is finished.

The attribute `_next_state` contains the next state `next` should enter in.

5.1.125 Counting, generating, and manipulating nonnegative integer matrices

Counting, generating, and manipulating nonnegative integer matrices with prescribed row sums and column sums.

AUTHORS:

- Franco Saliola

class sage.combinat.integer_matrices.**IntegerMatrices**(*row_sums*, *column_sums*)

Bases: UniqueRepresentation, Parent

The class of nonnegative integer matrices with prescribed row sums and column sums.

An *integer matrix* m with column sums $c := (c_1, ..., c_k)$ and row sums $l := (l_1, ..., l_n)$ where $c_1 + ... + c_k$ is equal to $l_1 + ... + l_n$, is a $n \times k$ matrix $m = (m_{i,j})$ such that $m_{1,j} + \cdots + m_{n,j} = c_j$, for all j and $m_{i,1} + \cdots + m_{i,k} = l_i$, for all i.

EXAMPLES:

There are 6 integer matrices with row sums $[3, 2, 2]$ and column sums $[2, 5]$:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IM = IntegerMatrices([3,2,2], [2,5]); IM
Non-negative integer matrices with row sums [3, 2, 2] and column sums [2, 5]
sage: IM.list()
[
[2 1]  [1 2]  [1 2]  [0 3]  [0 3]  [0 3]
[0 2]  [1 1]  [0 2]  [2 0]  [1 1]  [0 2]
[0 2], [0 2], [1 1], [0 2], [1 1], [2 0]
]
sage: IM.cardinality()
6
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_matrices import IntegerMatrices
>>> IM = IntegerMatrices([Integer(3),Integer(2),Integer(2)], [Integer(2),
↪Integer(5)]); IM
Non-negative integer matrices with row sums [3, 2, 2] and column sums [2, 5]
>>> IM.list()
[
[2 1]  [1 2]  [1 2]  [0 3]  [0 3]  [0 3]
[0 2]  [1 1]  [0 2]  [2 0]  [1 1]  [0 2]
[0 2], [0 2], [1 1], [0 2], [1 1], [2 0]
]
>>> IM.cardinality()
6
```

cardinality()

The number of integer matrices with the prescribed row sums and columns sums.

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IntegerMatrices([2,5], [3,2,2]).cardinality()
6
sage: IntegerMatrices([1,1,1,1,1], [1,1,1,1,1]).cardinality()
120
```

(continues on next page)

(continued from previous page)

```
sage: IntegerMatrices([2,2,2,2], [2,2,2,2]).cardinality()
282
sage: IntegerMatrices([4], [3]).cardinality()
0
sage: len(IntegerMatrices([0,0], [0]).list())
1
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_matrices import IntegerMatrices
>>> IntegerMatrices([Integer(2),Integer(5)], [Integer(3),Integer(2),
↪Integer(2)]).cardinality()
6
>>> IntegerMatrices([Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)]).cardinality()
120
>>> IntegerMatrices([Integer(2),Integer(2),Integer(2),Integer(2)],␣
↪[Integer(2),Integer(2),Integer(2),Integer(2)]).cardinality()
282
>>> IntegerMatrices([Integer(4)], [Integer(3)]).cardinality()
0
>>> len(IntegerMatrices([Integer(0),Integer(0)], [Integer(0)]).list())
1
```

This method computes the cardinality using symmetric functions. Below are the same examples, but computed by generating the actual matrices:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: len(IntegerMatrices([2,5], [3,2,2]).list())
6
sage: len(IntegerMatrices([1,1,1,1,1], [1,1,1,1,1]).list())
120
sage: len(IntegerMatrices([2,2,2,2], [2,2,2,2]).list())
282
sage: len(IntegerMatrices([4], [3]).list())
0
sage: len(IntegerMatrices([0], [0]).list())
1
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_matrices import IntegerMatrices
>>> len(IntegerMatrices([Integer(2),Integer(5)], [Integer(3),Integer(2),
↪Integer(2)]).list())
6
>>> len(IntegerMatrices([Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1)], [Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)]).
↪list())
120
>>> len(IntegerMatrices([Integer(2),Integer(2),Integer(2),Integer(2)],␣
↪[Integer(2),Integer(2),Integer(2),Integer(2)]).list())
282
>>> len(IntegerMatrices([Integer(4)], [Integer(3)]).list())
0
```

(continues on next page)

(continued from previous page)

```
>>> len(IntegerMatrices([Integer(0)], [Integer(0)]).list())
1
```

column_sums()

The column sums of the integer matrices in `self`.

OUTPUT: Composition

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IM = IntegerMatrices([3,2,2], [2,5])
sage: IM.column_sums()
[2, 5]
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_matrices import IntegerMatrices
>>> IM = IntegerMatrices([Integer(3),Integer(2),Integer(2)], [Integer(2),
↪Integer(5)])
>>> IM.column_sums()
[2, 5]
```

row_sums()

The row sums of the integer matrices in `self`.

OUTPUT: Composition

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IM = IntegerMatrices([3,2,2], [2,5])
sage: IM.row_sums()
[3, 2, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_matrices import IntegerMatrices
>>> IM = IntegerMatrices([Integer(3),Integer(2),Integer(2)], [Integer(2),
↪Integer(5)])
>>> IM.row_sums()
[3, 2, 2]
```

to_composition(x)

The composition corresponding to the integer matrix.

This is the composition obtained by reading the entries of the matrix from left to right along each row, and reading the rows from top to bottom, ignore zeros.

INPUT:

- `x` – matrix

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IM = IntegerMatrices([3,2,2], [2,5]); IM
```

(continues on next page)

(continued from previous page)

```
Non-negative integer matrices with row sums [3, 2, 2] and column sums [2, 5]
sage: IM.list()
[
[2 1]  [1 2]  [1 2]  [0 3]  [0 3]  [0 3]
[0 2]  [1 1]  [0 2]  [2 0]  [1 1]  [0 2]
[0 2], [0 2], [1 1], [0 2], [1 1], [2 0]
]
sage: for m in IM: print(IM.to_composition(m))
[2, 1, 2, 2]
[1, 2, 1, 1, 2]
[1, 2, 2, 1, 1]
[3, 2, 2]
[3, 1, 1, 1, 1]
[3, 2, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_matrices import IntegerMatrices
>>> IM = IntegerMatrices([Integer(3),Integer(2),Integer(2)], [Integer(2),
↪Integer(5)]); IM
Non-negative integer matrices with row sums [3, 2, 2] and column sums [2, 5]
>>> IM.list()
[
[2 1]  [1 2]  [1 2]  [0 3]  [0 3]  [0 3]
[0 2]  [1 1]  [0 2]  [2 0]  [1 1]  [0 2]
[0 2], [0 2], [1 1], [0 2], [1 1], [2 0]
]
>>> for m in IM: print(IM.to_composition(m))
[2, 1, 2, 2]
[1, 2, 1, 1, 2]
[1, 2, 2, 1, 1]
[3, 2, 2]
[3, 1, 1, 1, 1]
[3, 2, 2]
```

sage.combinat.integer_matrices.**integer_matrices_generator**(*row_sums*, *column_sums*)

Recursively generate the integer matrices with the prescribed row sums and column sums.

INPUT:

- `row_sums` – list or tuple
- `column_sums` – list or tuple

OUTPUT: an iterator producing a list of lists

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import integer_matrices_generator
sage: iter = integer_matrices_generator([3,2,2], [2,5]); iter
<generator object ...integer_matrices_generator at ...>
sage: for m in iter: print(m)
[[2, 1], [0, 2], [0, 2]]
[[1, 2], [1, 1], [0, 2]]
[[1, 2], [0, 2], [1, 1]]
```

(continues on next page)

(continued from previous page)

```
[[0, 3], [2, 0], [0, 2]]
[[0, 3], [1, 1], [1, 1]]
[[0, 3], [0, 2], [2, 0]]
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_matrices import integer_matrices_generator
>>> iter = integer_matrices_generator([Integer(3),Integer(2),Integer(2)],␣
↪[Integer(2),Integer(5)]); iter
<generator object ...integer_matrices_generator at ...>
>>> for m in iter: print(m)
[[2, 1], [0, 2], [0, 2]]
[[1, 2], [1, 1], [0, 2]]
[[1, 2], [0, 2], [1, 1]]
[[0, 3], [2, 0], [0, 2]]
[[0, 3], [1, 1], [1, 1]]
[[0, 3], [0, 2], [2, 0]]
```

5.1.126 (Non-negative) Integer vectors

AUTHORS:

- Mike Hansen (2007) - original module
- Nathann Cohen, David Joyner (2009-2010) - Gale-Ryser stuff
- Nathann Cohen, David Joyner (2011) - Gale-Ryser bugfix
- Travis Scrimshaw (2012-05-12) - Updated doc-strings to tell the user of that the class's name is a misnomer (that they only contains non-negative entries).
- Federico Poloni (2013) - specialized `rank()`
- Travis Scrimshaw (2013-02-04) - Refactored to use `ClonableIntArray`

class sage.combinat.integer_vector.**IntegerVector**

Bases: `ClonableArray`

An integer vector.

check()

Check to make sure this is a valid integer vector by making sure all entries are nonnegative.

EXAMPLES:

```
sage: IV = IntegerVectors()
sage: elt = IV([1,2,1])
sage: elt.check()
```

```
>>> from sage.all import *
>>> IV = IntegerVectors()
>>> elt = IV([Integer(1),Integer(2),Integer(1)])
>>> elt.check()
```

Check Issue #34510:

```
sage: IV3 = IntegerVectors(n=3)
sage: IV3([2,2])
Traceback (most recent call last):
...
ValueError: [2, 2] doesn't satisfy correct constraints
sage: IVk3 = IntegerVectors(k=3)
sage: IVk3([2,2])
Traceback (most recent call last):
...
ValueError: [2, 2] doesn't satisfy correct constraints
sage: IV33 = IntegerVectors(n=3, k=3)
sage: IV33([2,2])
Traceback (most recent call last):
...
ValueError: [2, 2] doesn't satisfy correct constraints
```

```
>>> from sage.all import *
>>> IV3 = IntegerVectors(n=Integer(3))
>>> IV3([Integer(2),Integer(2)])
Traceback (most recent call last):
...
ValueError: [2, 2] doesn't satisfy correct constraints
>>> IVk3 = IntegerVectors(k=Integer(3))
>>> IVk3([Integer(2),Integer(2)])
Traceback (most recent call last):
...
ValueError: [2, 2] doesn't satisfy correct constraints
>>> IV33 = IntegerVectors(n=Integer(3), k=Integer(3))
>>> IV33([Integer(2),Integer(2)])
Traceback (most recent call last):
...
ValueError: [2, 2] doesn't satisfy correct constraints
```

specht_module(*base_ring=None*)

Return the Specht module corresponding to `self`.

EXAMPLES:

```
sage: SM = IntegerVectors()([2,0,1,0,2]).specht_module(QQ); SM              #␣
↪needs sage.combinat sage.modules
Specht module of [(0, 0), (0, 1), (2, 0), (4, 0), (4, 1)] over Rational Field
sage: s = SymmetricFunctions(QQ).s()                                        #␣
↪needs sage.combinat sage.modules
sage: s(SM.frobenius_image())                                               #␣
↪needs sage.combinat sage.modules
s[2, 2, 1]
```

```
>>> from sage.all import *
>>> SM = IntegerVectors()([Integer(2),Integer(0),Integer(1),Integer(0),
↪Integer(2)]).specht_module(QQ); SM             # needs sage.combinat sage.
↪modules
Specht module of [(0, 0), (0, 1), (2, 0), (4, 0), (4, 1)] over Rational Field
>>> s = SymmetricFunctions(QQ).s()                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat sage.modules
>>> s(SM.frobenius_image())                                                  #␣
↪needs sage.combinat sage.modules
s[2, 2, 1]
```

specht_module_dimension(*base_ring=None*)

Return the dimension of the Specht module corresponding to `self`.

INPUT:

- `BR` – (default: **Q**) the base ring

EXAMPLES:

```
sage: IntegerVectors()([2,0,1,0,2]).specht_module_dimension()                  #␣
↪needs sage.combinat sage.modules
5
sage: IntegerVectors()([2,0,1,0,2]).specht_module_dimension(GF(2))             #␣
↪needs sage.combinat sage.modules sage.rings.finite_rings
5
```

```
>>> from sage.all import *
>>> IntegerVectors()([Integer(2),Integer(0),Integer(1),Integer(0),
↪Integer(2)]).specht_module_dimension()                  # needs sage.combinat␣
↪sage.modules
5
>>> IntegerVectors()([Integer(2),Integer(0),Integer(1),Integer(0),
↪Integer(2)]).specht_module_dimension(GF(Integer(2)))          # needs sage.
↪combinat sage.modules sage.rings.finite_rings
5
```

trim()

Remove trailing zeros from the integer vector.

EXAMPLES:

```
sage: IV = IntegerVectors()
sage: IV([5,3,5,1,0,0]).trim()
[5, 3, 5, 1]
sage: IV([5,0,5,1,0]).trim()
[5, 0, 5, 1]
sage: IV([4,3,3]).trim()
[4, 3, 3]
sage: IV([0,0,0]).trim()
[]

sage: IV = IntegerVectors(k=4)
sage: v = IV([4,3,2,0]).trim(); v
[4, 3, 2]
sage: v.parent()
Integer vectors
```

```
>>> from sage.all import *
>>> IV = IntegerVectors()
>>> IV([Integer(5),Integer(3),Integer(5),Integer(1),Integer(0),Integer(0)]).
↪trim()
[5, 3, 5, 1]
>>> IV([Integer(5),Integer(0),Integer(5),Integer(1),Integer(0)]).trim()
[5, 0, 5, 1]
>>> IV([Integer(4),Integer(3),Integer(3)]).trim()
[4, 3, 3]
>>> IV([Integer(0),Integer(0),Integer(0)]).trim()
[]

>>> IV = IntegerVectors(k=Integer(4))
>>> v = IV([Integer(4),Integer(3),Integer(2),Integer(0)]).trim(); v
[4, 3, 2]
>>> v.parent()
Integer vectors
```

class sage.combinat.integer_vector.**IntegerVectors**(*category=None*)

Bases: Parent

The class of (nonnegative) integer vectors.

INPUT:

- n – if set to an integer, returns the combinatorial class of integer vectors whose sum is n; if set to None (default), no such constraint is defined
- k – the length of the vectors; set to None (default) if you do not want such a constraint

Note

The entries are nonnegative integers.

EXAMPLES:

If n is not specified, it returns the class of all integer vectors:

```
sage: IntegerVectors()
Integer vectors
sage: [] in IntegerVectors()
True
sage: [1,2,1] in IntegerVectors()
True
sage: [1, 0, 0] in IntegerVectors()
True
```

```
>>> from sage.all import *
>>> IntegerVectors()
Integer vectors
>>> [] in IntegerVectors()
True
>>> [Integer(1),Integer(2),Integer(1)] in IntegerVectors()
True
```

(continues on next page)

(continued from previous page)

```
>>> [Integer(1), Integer(0), Integer(0)] in IntegerVectors()
True
```

Entries are nonnegative:

```
sage: [-1, 2] in IntegerVectors()
False
```

```
>>> from sage.all import *
>>> [-Integer(1), Integer(2)] in IntegerVectors()
False
```

If `n` is specified, then it returns the class of all integer vectors which sum to `n`:

```
sage: IV3 = IntegerVectors(3); IV3
Integer vectors that sum to 3
```

```
>>> from sage.all import *
>>> IV3 = IntegerVectors(Integer(3)); IV3
Integer vectors that sum to 3
```

Note that trailing zeros are ignored so that `[3, 0]` does not show up in the following list (since `[3]` does):

```
sage: IntegerVectors(3, max_length=2).list()
[[3], [2, 1], [1, 2], [0, 3]]
```

```
>>> from sage.all import *
>>> IntegerVectors(Integer(3), max_length=Integer(2)).list()
[[3], [2, 1], [1, 2], [0, 3]]
```

If `n` and `k` are both specified, then it returns the class of integer vectors that sum to `n` and are of length `k`:

```
sage: IV53 = IntegerVectors(5,3); IV53
Integer vectors of length 3 that sum to 5
sage: IV53.cardinality()
21
sage: IV53.first()
[5, 0, 0]
sage: IV53.last()
[0, 0, 5]
sage: IV53.random_element().parent() is IV53
True
```

```
>>> from sage.all import *
>>> IV53 = IntegerVectors(Integer(5),Integer(3)); IV53
Integer vectors of length 3 that sum to 5
>>> IV53.cardinality()
21
>>> IV53.first()
[5, 0, 0]
>>> IV53.last()
[0, 0, 5]
```

(continues on next page)

(continued from previous page)

```
>>> IV53.random_element().parent() is IV53
True
```

Further examples:

```
sage: IntegerVectors(-1, 0, min_part=1).list()
[]
sage: IntegerVectors(-1, 2, min_part=1).list()
[]
sage: IntegerVectors(0, 0, min_part=1).list()
[[]]
sage: IntegerVectors(3, 0, min_part=1).list()
[]
sage: IntegerVectors(0, 1, min_part=1).list()
[]
sage: IntegerVectors(2, 2, min_part=1).list()
[[1, 1]]
sage: IntegerVectors(2, 3, min_part=1).list()
[]
sage: IntegerVectors(4, 2, min_part=1).list()
[[3, 1], [2, 2], [1, 3]]
```

```
>>> from sage.all import *
>>> IntegerVectors(-Integer(1), Integer(0), min_part=Integer(1)).list()
[]
>>> IntegerVectors(-Integer(1), Integer(2), min_part=Integer(1)).list()
[]
>>> IntegerVectors(Integer(0), Integer(0), min_part=Integer(1)).list()
[[]]
>>> IntegerVectors(Integer(3), Integer(0), min_part=Integer(1)).list()
[]
>>> IntegerVectors(Integer(0), Integer(1), min_part=Integer(1)).list()
[]
>>> IntegerVectors(Integer(2), Integer(2), min_part=Integer(1)).list()
[[1, 1]]
>>> IntegerVectors(Integer(2), Integer(3), min_part=Integer(1)).list()
[]
>>> IntegerVectors(Integer(4), Integer(2), min_part=Integer(1)).list()
[[3, 1], [2, 2], [1, 3]]
```

```
sage: IntegerVectors(0, 3, outer=[0,0,0]).list()
[[0, 0, 0]]
sage: IntegerVectors(1, 3, outer=[0,0,0]).list()
[]
sage: IntegerVectors(2, 3, outer=[0,2,0]).list()
[[0, 2, 0]]
sage: IntegerVectors(2, 3, outer=[1,2,1]).list()
[[1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1]]
sage: IntegerVectors(2, 3, outer=[1,1,1]).list()
[[1, 1, 0], [1, 0, 1], [0, 1, 1]]
sage: IntegerVectors(2, 5, outer=[1,1,1,1,1]).list()
[[1, 1, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
 [1, 0, 1, 0, 0],
 [1, 0, 0, 1, 0],
 [1, 0, 0, 0, 1],
 [0, 1, 1, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 1, 0, 0, 1],
 [0, 0, 1, 1, 0],
 [0, 0, 1, 0, 1],
 [0, 0, 0, 1, 1]]
```

```
>>> from sage.all import *
>>> IntegerVectors(Integer(0), Integer(3), outer=[Integer(0),Integer(0),
↪Integer(0)]).list()
[[0, 0, 0]]
>>> IntegerVectors(Integer(1), Integer(3), outer=[Integer(0),Integer(0),
↪Integer(0)]).list()
[]
>>> IntegerVectors(Integer(2), Integer(3), outer=[Integer(0),Integer(2),
↪Integer(0)]).list()
[[0, 2, 0]]
>>> IntegerVectors(Integer(2), Integer(3), outer=[Integer(1),Integer(2),
↪Integer(1)]).list()
[[1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1]]
>>> IntegerVectors(Integer(2), Integer(3), outer=[Integer(1),Integer(1),
↪Integer(1)]).list()
[[1, 1, 0], [1, 0, 1], [0, 1, 1]]
>>> IntegerVectors(Integer(2), Integer(5), outer=[Integer(1),Integer(1),
↪Integer(1),Integer(1),Integer(1)]).list()
[[1, 1, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [1, 0, 0, 1, 0],
 [1, 0, 0, 0, 1],
 [0, 1, 1, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 1, 0, 0, 1],
 [0, 0, 1, 1, 0],
 [0, 0, 1, 0, 1],
 [0, 0, 0, 1, 1]]
```

```
sage: iv = [ IntegerVectors(n,k) for n in range(-2, 7) for k in range(7) ]
sage: all(map(lambda x: x.cardinality() == len(x.list()), iv))
True
sage: essai = [[1,1,1], [2,5,6], [6,5,2]]
sage: iv = [ IntegerVectors(x[0], x[1], max_part = x[2]-1) for x in essai ]
sage: all(map(lambda x: x.cardinality() == len(x.list()), iv))
True
```

```
>>> from sage.all import *
>>> iv = [ IntegerVectors(n,k) for n in range(-Integer(2), Integer(7)) for k in␣
↪range(Integer(7)) ]
>>> all(map(lambda x: x.cardinality() == len(x.list()), iv))
```

(continues on next page)

(continued from previous page)

```
True
>>> essai = [[Integer(1),Integer(1),Integer(1)], [Integer(2),Integer(5),
↪Integer(6)], [Integer(6),Integer(5),Integer(2)]]
>>> iv = [ IntegerVectors(x[Integer(0)], x[Integer(1)], max_part = x[Integer(2)]-
↪Integer(1)) for x in essai ]
>>> all(map(lambda x: x.cardinality() == len(x.list()), iv))
True
```

An example showing the same output by using IntegerListsLex:

```
sage: IntegerVectors(4, max_length=2).list()
[[4], [3, 1], [2, 2], [1, 3], [0, 4]]
sage: list(IntegerListsLex(4, max_length=2))
[[4], [3, 1], [2, 2], [1, 3], [0, 4]]
```

```
>>> from sage.all import *
>>> IntegerVectors(Integer(4), max_length=Integer(2)).list()
[[4], [3, 1], [2, 2], [1, 3], [0, 4]]
>>> list(IntegerListsLex(Integer(4), max_length=Integer(2)))
[[4], [3, 1], [2, 2], [1, 3], [0, 4]]
```

See also

sage.combinat.integer_lists.invlex.IntegerListsLex

Element

alias of *IntegerVector*

class sage.combinat.integer_vector.**IntegerVectorsConstraints**(*n=None*, *k=None*, ***constraints*)

Bases: *IntegerVectors*

Class of integer vectors subject to various constraints.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: IntegerVectors(3, 3, min_part=1).cardinality()
1
sage: IntegerVectors(5, 3, min_part=1).cardinality()
6
sage: IntegerVectors(13, 4, max_part=4).cardinality()
20
sage: IntegerVectors(k=4, max_part=3).cardinality()
256
sage: IntegerVectors(k=3, min_part=2, max_part=4).cardinality()
27
sage: IntegerVectors(13, 4, min_part=2, max_part=4).cardinality()
16
```

```
>>> from sage.all import *
>>> IntegerVectors(Integer(3), Integer(3), min_part=Integer(1)).cardinality()
1
>>> IntegerVectors(Integer(5), Integer(3), min_part=Integer(1)).cardinality()
6
>>> IntegerVectors(Integer(13), Integer(4), max_part=Integer(4)).cardinality()
20
>>> IntegerVectors(k=Integer(4), max_part=Integer(3)).cardinality()
256
>>> IntegerVectors(k=Integer(3), min_part=Integer(2), max_part=Integer(4)).
↪cardinality()
27
>>> IntegerVectors(Integer(13), Integer(4), min_part=Integer(2), max_
↪part=Integer(4)).cardinality()
16
```

class sage.combinat.integer_vector.**IntegerVectors_all**

Bases: UniqueRepresentation, *IntegerVectors*

Class of all integer vectors.

class sage.combinat.integer_vector.**IntegerVectors_k**(*k*)

Bases: UniqueRepresentation, *IntegerVectors*

Integer vectors of length k.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: IntegerVectors(k=0).cardinality()
1
sage: IntegerVectors(k=10).cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> IntegerVectors(k=Integer(0)).cardinality()
1
>>> IntegerVectors(k=Integer(10)).cardinality()
+Infinity
```

rank(*x*)

Return the rank of a given element.

INPUT:

- `x` – list with `len(x) == k`

EXAMPLES:

```
sage: IntegerVectors(k=5).rank([0,0,0,0,0])
0
sage: IntegerVectors(k=5).rank([1,1,0,0,0])
7
```

```
>>> from sage.all import *
>>> IntegerVectors(k=Integer(5)).rank([Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0)])
0
>>> IntegerVectors(k=Integer(5)).rank([Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(0)])
7
```

unrank(*x*)

Return the element at given rank x.

INPUT:

- `x` – integer such that `x < self.cardinality()`

EXAMPLES:

```
sage: IntegerVectors(k=5).unrank(10)
[1, 0, 0, 0, 1]
sage: IntegerVectors(k=5).unrank(15)
[0, 0, 2, 0, 0]
sage: IntegerVectors(k=0).unrank(0)
[]
```

```
>>> from sage.all import *
>>> IntegerVectors(k=Integer(5)).unrank(Integer(10))
[1, 0, 0, 0, 1]
>>> IntegerVectors(k=Integer(5)).unrank(Integer(15))
[0, 0, 2, 0, 0]
>>> IntegerVectors(k=Integer(0)).unrank(Integer(0))
[]
```

class sage.combinat.integer_vector.**IntegerVectors_n**(*n*)

Bases: `UniqueRepresentation`, *`IntegerVectors`*

Integer vectors that sum to n.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: IntegerVectors(n=0).cardinality()
1
sage: IntegerVectors(n=10).cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> IntegerVectors(n=Integer(0)).cardinality()
1
>>> IntegerVectors(n=Integer(10)).cardinality()
+Infinity
```

rank(*x*)

Return the rank of a given element.

INPUT:

- `x` – list with `sum(x) == n`

EXAMPLES:

```
sage: IntegerVectors(n=5).rank([5,0])
1
sage: IntegerVectors(n=5).rank([3,2])
3
```

```
>>> from sage.all import *
>>> IntegerVectors(n=Integer(5)).rank([Integer(5),Integer(0)])
1
>>> IntegerVectors(n=Integer(5)).rank([Integer(3),Integer(2)])
3
```

unrank(*x*)

Return the element at given rank x.

INPUT:

- `x` – integer

EXAMPLES:

```
sage: IntegerVectors(n=5).unrank(2)
[4, 1]
sage: IntegerVectors(n=10).unrank(10)
[1, 9]
```

```
>>> from sage.all import *
>>> IntegerVectors(n=Integer(5)).unrank(Integer(2))
[4, 1]
>>> IntegerVectors(n=Integer(10)).unrank(Integer(10))
[1, 9]
```

class sage.combinat.integer_vector.**IntegerVectors_nk**(*n*, *k*)

Bases: `UniqueRepresentation`, *IntegerVectors*

Integer vectors of length k that sum to n.

AUTHORS:

- Martin Albrecht
- Mike Hansen

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: IntegerVectors(3,5).cardinality()
35
sage: IntegerVectors(99, 3).cardinality()
5050
sage: IntegerVectors(10^9 - 1, 3).cardinality()
500000000500000000
```

```
>>> from sage.all import *
>>> IntegerVectors(Integer(3),Integer(5)).cardinality()
35
>>> IntegerVectors(Integer(99), Integer(3)).cardinality()
5050
>>> IntegerVectors(Integer(10)**Integer(9) - Integer(1), Integer(3)).
↪cardinality()
500000000500000000
```

rank(*x*)

Return the rank of a given element.

INPUT:

- `x` – list with `sum(x) == n` and `len(x) == k`

unrank(*x*)

Return the element at given rank x.

INPUT:

- `x` – integer such that `x < self.cardinality()`

EXAMPLES:

```
sage: IntegerVectors(4,5).unrank(30)
[1, 0, 1, 0, 2]
sage: IntegerVectors(2,3).unrank(5)
[0, 0, 2]
```

```
>>> from sage.all import *
>>> IntegerVectors(Integer(4),Integer(5)).unrank(Integer(30))
[1, 0, 1, 0, 2]
>>> IntegerVectors(Integer(2),Integer(3)).unrank(Integer(5))
[0, 0, 2]
```

class sage.combinat.integer_vector.**IntegerVectors_nnondescents**(*n*, *comp*)

Bases: `UniqueRepresentation`, *IntegerVectors*

Integer vectors graded by two parameters.

The grading parameters on the integer vector v are:

- `n` – the sum of the parts of v
- `c` – the non descents composition of v

In other words: the length of v equals $c_1 + \cdots + c_k$, and v is decreasing in the consecutive blocs of length $c_1, \ldots, c_k$,

INPUT:

- `n` – the positive integer n
- `comp` – the composition c

Those are the integer vectors of sum n that are lexicographically maximal (for the natural left-to-right reading) in their orbit by the Young subgroup $S_{c_1} \times \cdots \times S_{c_k}$. In particular, they form a set of orbit representative of integer vectors with respect to this Young subgroup.

sage.combinat.integer_vector.**gale_ryser_theorem**(*p1*, *p2*, *algorithm*, *solver*, *integrality_tolerance='gale'*)

Return the binary matrix given by the Gale-Ryser theorem.

The Gale Ryser theorem asserts that if p_1, p_2 are two partitions of n of respective lengths k_1, k_2, then there is a binary $k_1 \times k_2$ matrix M such that p_1 is the vector of row sums and p_2 is the vector of column sums of M, if and only if the conjugate of p_2 dominates p_1.

INPUT:

- `p1`, `p2` – list of integers representing the vectors of row/column sums
- `algorithm` – two possible string values:
  - `'ryser'` implements the construction due to Ryser [Ryser63].
  - `'gale'` (default) implements the construction due to Gale [Gale57].
- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`

OUTPUT: a binary matrix if it exists, `None` otherwise

Gale's Algorithm:

(Gale [Gale57]): A matrix satisfying the constraints of its sums can be defined as the solution of the following Linear Program, which Sage knows how to solve.

$$\forall i \sum_{j=1}^{k_2} b_{i,j} = p_{1,j}$$
$$\forall i \sum_{j=1}^{k_1} b_{j,i} = p_{2,j}$$
$$b_{i,j} \text{ is a binary variable}$$

Ryser's Algorithm:

(Ryser [Ryser63]): The construction of an $m \times n$ matrix $A = A_{r,s}$, due to Ryser, is described as follows. The construction works if and only if have $s \preceq r^*$.

- Construct the $m \times n$ matrix B from r by defining the i-th row of B to be the vector whose first r_i entries are 1, and the remainder are 0s, $1 \le i \le m$. This maximal matrix B with row sum r and ones left justified has column sum r^*.
- Shift the last 1 in certain rows of B to column n in order to achieve the sum s_n. Call this B again.
  - The 1's in column n are to appear in those rows in which A has the largest row sums, giving preference to the bottom-most positions in case of ties.
  - Note: When this step automatically "fixes" other columns, one must skip ahead to the first column index with a wrong sum in the step below.
- Proceed inductively to construct columns $n-1$, …, 2, 1. Note: when performing the induction on step k, we consider the row sums of the first k columns.
- Set $A = B$. Return A.

EXAMPLES:

Computing the matrix for $p_1 = p_2 = 2 + 2 + 1$:

```
sage: # needs sage.combinat sage.modules
sage: from sage.combinat.integer_vector import gale_ryser_theorem
sage: p1 = [2,2,1]
sage: p2 = [2,2,1]
sage: print(gale_ryser_theorem(p1, p2))         # not tested
[1 1 0]
[1 0 1]
[0 1 0]
sage: A = gale_ryser_theorem(p1, p2)
sage: rs = [sum(x) for x in A.rows()]
sage: cs = [sum(x) for x in A.columns()]
sage: p1 == rs; p2 == cs
True
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> from sage.combinat.integer_vector import gale_ryser_theorem
>>> p1 = [Integer(2),Integer(2),Integer(1)]
>>> p2 = [Integer(2),Integer(2),Integer(1)]
>>> print(gale_ryser_theorem(p1, p2))         # not tested
[1 1 0]
[1 0 1]
[0 1 0]
>>> A = gale_ryser_theorem(p1, p2)
>>> rs = [sum(x) for x in A.rows()]
>>> cs = [sum(x) for x in A.columns()]
>>> p1 == rs; p2 == cs
True
True
```

Or for a non-square matrix with $p_1 = 3+3+2+1$ and $p_2 = 3+2+2+1+1$, using Ryser's algorithm:

```
sage: # needs sage.combinat sage.modules
sage: from sage.combinat.integer_vector import gale_ryser_theorem
sage: p1 = [3,3,1,1]
sage: p2 = [3,3,1,1]
sage: gale_ryser_theorem(p1, p2, algorithm='ryser')
[1 1 1 0]
[1 1 0 1]
[1 0 0 0]
[0 1 0 0]
sage: p1 = [4,2,2]
sage: p2 = [3,3,1,1]
sage: gale_ryser_theorem(p1, p2, algorithm='ryser')
[1 1 1 1]
[1 1 0 0]
[1 1 0 0]
sage: p1 = [4,2,2,0]
sage: p2 = [3,3,1,1,0,0]
sage: gale_ryser_theorem(p1, p2, algorithm='ryser')
[1 1 1 1 0 0]
[1 1 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[1 1 0 0 0 0]
[0 0 0 0 0 0]
sage: p1 = [3,3,2,1]
sage: p2 = [3,2,2,1,1]
sage: print(gale_ryser_theorem(p1, p2, algorithm='gale'))       # not tested
[1 1 1 0 0]
[1 1 0 0 1]
[1 0 1 0 0]
[0 0 0 1 0]
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> from sage.combinat.integer_vector import gale_ryser_theorem
>>> p1 = [Integer(3),Integer(3),Integer(1),Integer(1)]
>>> p2 = [Integer(3),Integer(3),Integer(1),Integer(1)]
>>> gale_ryser_theorem(p1, p2, algorithm='ryser')
[1 1 1 0]
[1 1 0 1]
[1 0 0 0]
[0 1 0 0]
>>> p1 = [Integer(4),Integer(2),Integer(2)]
>>> p2 = [Integer(3),Integer(3),Integer(1),Integer(1)]
>>> gale_ryser_theorem(p1, p2, algorithm='ryser')
[1 1 1 1]
[1 1 0 0]
[1 1 0 0]
>>> p1 = [Integer(4),Integer(2),Integer(2),Integer(0)]
>>> p2 = [Integer(3),Integer(3),Integer(1),Integer(1),Integer(0),Integer(0)]
>>> gale_ryser_theorem(p1, p2, algorithm='ryser')
[1 1 1 1 0 0]
[1 1 0 0 0 0]
[1 1 0 0 0 0]
[0 0 0 0 0 0]
>>> p1 = [Integer(3),Integer(3),Integer(2),Integer(1)]
>>> p2 = [Integer(3),Integer(2),Integer(2),Integer(1),Integer(1)]
>>> print(gale_ryser_theorem(p1, p2, algorithm='gale'))       # not tested
[1 1 1 0 0]
[1 1 0 0 1]
[1 0 1 0 0]
[0 0 0 1 0]
```

With 0 in the sequences, and with unordered inputs:

```
sage: from sage.combinat.integer_vector import gale_ryser_theorem
sage: gale_ryser_theorem([3,3,0,1,1,0], [3,1,3,1,0], algorithm='ryser')       #␣
↪needs sage.combinat sage.modules
[1 1 1 0 0]
[1 0 1 1 0]
[0 0 0 0 0]
[1 0 0 0 0]
[0 0 1 0 0]
[0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
sage: p1 = [3,1,1,1,1]; p2 = [3,2,2,0]
sage: gale_ryser_theorem(p1, p2, algorithm='ryser')                          #␣
↪needs sage.combinat sage.modules
[1 1 1 0]
[1 0 0 0]
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_vector import gale_ryser_theorem
>>> gale_ryser_theorem([Integer(3),Integer(3),Integer(0),Integer(1),Integer(1),
↪Integer(0)], [Integer(3),Integer(1),Integer(3),Integer(1),Integer(0)],␣
↪algorithm='ryser')         # needs sage.combinat sage.modules
[1 1 1 0 0]
[1 0 1 1 0]
[0 0 0 0 0]
[1 0 0 0 0]
[0 0 1 0 0]
[0 0 0 0 0]
>>> p1 = [Integer(3),Integer(1),Integer(1),Integer(1),Integer(1)]; p2 =␣
↪[Integer(3),Integer(2),Integer(2),Integer(0)]
>>> gale_ryser_theorem(p1, p2, algorithm='ryser')                          #␣
↪needs sage.combinat sage.modules
[1 1 1 0]
[1 0 0 0]
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
```

REFERENCES:

sage.combinat.integer_vector.**integer_vectors_nk_fast_iter**(*n*, *k*)

A fast iterator for integer vectors of `n` of length `k` which yields Python lists filled with Sage Integers.

EXAMPLES:

```
sage: from sage.combinat.integer_vector import integer_vectors_nk_fast_iter
sage: list(integer_vectors_nk_fast_iter(3, 2))
[[3, 0], [2, 1], [1, 2], [0, 3]]
sage: list(integer_vectors_nk_fast_iter(2, 2))
[[2, 0], [1, 1], [0, 2]]
sage: list(integer_vectors_nk_fast_iter(1, 2))
[[1, 0], [0, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_vector import integer_vectors_nk_fast_iter
>>> list(integer_vectors_nk_fast_iter(Integer(3), Integer(2)))
[[3, 0], [2, 1], [1, 2], [0, 3]]
>>> list(integer_vectors_nk_fast_iter(Integer(2), Integer(2)))
[[2, 0], [1, 1], [0, 2]]
>>> list(integer_vectors_nk_fast_iter(Integer(1), Integer(2)))
[[1, 0], [0, 1]]
```

We check some corner cases:

```
sage: list(integer_vectors_nk_fast_iter(5, 1))
[[5]]
sage: list(integer_vectors_nk_fast_iter(1, 1))
[[1]]
sage: list(integer_vectors_nk_fast_iter(2, 0))
[]
sage: list(integer_vectors_nk_fast_iter(0, 2))
[[0, 0]]
sage: list(integer_vectors_nk_fast_iter(0, 0))
[[]]
```

```
>>> from sage.all import *
>>> list(integer_vectors_nk_fast_iter(Integer(5), Integer(1)))
[[5]]
>>> list(integer_vectors_nk_fast_iter(Integer(1), Integer(1)))
[[1]]
>>> list(integer_vectors_nk_fast_iter(Integer(2), Integer(0)))
[]
>>> list(integer_vectors_nk_fast_iter(Integer(0), Integer(2)))
[[0, 0]]
>>> list(integer_vectors_nk_fast_iter(Integer(0), Integer(0)))
[[]]
```

sage.combinat.integer_vector.**is_gale_ryser**(*r*, *s*)

Test whether the given sequences satisfy the condition of the Gale-Ryser theorem.

Given a binary matrix B of dimension $n \times m$, the vector of row sums is defined as the vector whose i^{th} component is equal to the sum of the i^{th} row in A. The vector of column sums is defined similarly.

If, given a binary matrix, these two vectors are easy to compute, the Gale-Ryser theorem lets us decide whether, given two nonnegative vectors r, s, there exists a binary matrix whose row/column sums vectors are r and s.

This functions answers accordingly.

INPUT:

- r, s – lists of nonnegative integers

ALGORITHM:

Without loss of generality, we can assume that:

- The two given sequences do not contain any 0 (which would correspond to an empty column/row)
- The two given sequences are ordered in decreasing order (reordering the sequence of row (resp. column) sums amounts to reordering the rows (resp. columns) themselves in the matrix, which does not alter the columns (resp. rows) sums.

We can then assume that r and s are partitions (see the corresponding class *Partition*)

If r^* denote the conjugate of r, the Gale-Ryser theorem asserts that a binary Matrix satisfying the constraints exists if and only if $s \preceq r^*$, where $\preceq$ denotes the domination order on partitions.

EXAMPLES:

```
sage: from sage.combinat.integer_vector import is_gale_ryser
sage: is_gale_ryser([4,2,2], [3,3,1,1])                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
True
sage: is_gale_ryser([4,2,1,1], [3,3,1,1])                                  #␣
↪needs sage.combinat
True
sage: is_gale_ryser([3,2,1,1], [3,3,1,1])                                  #␣
↪needs sage.combinat
False
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_vector import is_gale_ryser
>>> is_gale_ryser([Integer(4),Integer(2),Integer(2)], [Integer(3),Integer(3),
↪Integer(1),Integer(1)])                                          # needs sage.
↪combinat
True
>>> is_gale_ryser([Integer(4),Integer(2),Integer(1),Integer(1)], [Integer(3),
↪Integer(3),Integer(1),Integer(1)])                                        #␣
↪needs sage.combinat
True
>>> is_gale_ryser([Integer(3),Integer(2),Integer(1),Integer(1)], [Integer(3),
↪Integer(3),Integer(1),Integer(1)])                                        #␣
↪needs sage.combinat
False
```

REMARK: In the literature, what we are calling a Gale-Ryser sequence sometimes goes by the (rather generic-sounding) term "realizable sequence".

sage.combinat.integer_vector.**list2func**(*l*, *default=None*)

Given a list `l`, return a function that takes in a value `i` and return `l[i]`. If default is not `None`, then the function will return the default value for out of range `i`'s.

EXAMPLES:

```
sage: f = sage.combinat.integer_vector.list2func([1,2,3])
sage: f(0)
1
sage: f(1)
2
sage: f(2)
3
sage: f(3)
Traceback (most recent call last):
...
IndexError: list index out of range
```

```
>>> from sage.all import *
>>> f = sage.combinat.integer_vector.list2func([Integer(1),Integer(2),Integer(3)])
>>> f(Integer(0))
1
>>> f(Integer(1))
2
>>> f(Integer(2))
```

(continues on next page)

(continued from previous page)

```
3
>>> f(Integer(3))
Traceback (most recent call last):
...
IndexError: list index out of range
```

```
sage: f = sage.combinat.integer_vector.list2func([1,2,3], 0)
sage: f(2)
3
sage: f(3)
0
```

```
>>> from sage.all import *
>>> f = sage.combinat.integer_vector.list2func([Integer(1),Integer(2),Integer(3)],
↪ Integer(0))
>>> f(Integer(2))
3
>>> f(Integer(3))
0
```

5.1.127 Weighted Integer Vectors

AUTHORS:

- Mike Hansen (2007): initial version, ported from MuPAD-Combinat
- Nicolas M. Thiery (2010-10-30): WeightedIntegerVectors(weights) + cleanup

class sage.combinat.integer_vector_weighted.**WeightedIntegerVectors**(*n*, *weight*)

Bases: Parent, UniqueRepresentation

The class of integer vectors of n weighted by `weight`, that is, the nonnegative integer vectors $(v_1, \ldots, v_\ell)$ satisfying $\sum_{i=1}^{\ell} v_i w_i = n$ where ℓ is `length(weight)` and w_i is `weight[i]`.

INPUT:

- `n` – nonnegative integer (optional)
- `weight` – tuple (or list or iterable) of positive integers

EXAMPLES:

```
sage: WeightedIntegerVectors(8, [1,1,2])
Integer vectors of 8 weighted by [1, 1, 2]
sage: WeightedIntegerVectors(8, [1,1,2]).first()
[0, 0, 4]
sage: WeightedIntegerVectors(8, [1,1,2]).last()
[8, 0, 0]
sage: WeightedIntegerVectors(8, [1,1,2]).cardinality()
25
sage: w = WeightedIntegerVectors(8, [1,1,2]).random_element()
sage: w.parent() is WeightedIntegerVectors(8, [1,1,2])
True

sage: WeightedIntegerVectors([1,1,2])
```

(continues on next page)

(continued from previous page)

```
Integer vectors weighted by [1, 1, 2]
sage: WeightedIntegerVectors([1,1,2]).cardinality()
+Infinity
sage: WeightedIntegerVectors([1,1,2]).first()
[0, 0, 0]
```

```
>>> from sage.all import *
>>> WeightedIntegerVectors(Integer(8), [Integer(1),Integer(1),Integer(2)])
Integer vectors of 8 weighted by [1, 1, 2]
>>> WeightedIntegerVectors(Integer(8), [Integer(1),Integer(1),Integer(2)]).first()
[0, 0, 4]
>>> WeightedIntegerVectors(Integer(8), [Integer(1),Integer(1),Integer(2)]).last()
[8, 0, 0]
>>> WeightedIntegerVectors(Integer(8), [Integer(1),Integer(1),Integer(2)]).
↪cardinality()
25
>>> w = WeightedIntegerVectors(Integer(8), [Integer(1),Integer(1),Integer(2)]).
↪random_element()
>>> w.parent() is WeightedIntegerVectors(Integer(8), [Integer(1),Integer(1),
↪Integer(2)])
True

>>> WeightedIntegerVectors([Integer(1),Integer(1),Integer(2)])
Integer vectors weighted by [1, 1, 2]
>>> WeightedIntegerVectors([Integer(1),Integer(1),Integer(2)]).cardinality()
+Infinity
>>> WeightedIntegerVectors([Integer(1),Integer(1),Integer(2)]).first()
[0, 0, 0]
```

Todo

Should the order of the arguments `n` and `weight` be exchanged to simplify the logic?

`Element`

alias of *`IntegerVector`*

class `sage.combinat.integer_vector_weighted.`**`WeightedIntegerVectors_all`**(*weight*)

Bases: `DisjointUnionEnumeratedSets`

Set of weighted integer vectors.

EXAMPLES:

```
sage: W = WeightedIntegerVectors([3,1,1,2,1]); W
Integer vectors weighted by [3, 1, 1, 2, 1]
sage: W.cardinality()
+Infinity

sage: W12 = W.graded_component(12)
sage: W12.an_element()
[4, 0, 0, 0, 0]
```

(continues on next page)

(continued from previous page)

```
sage: W12.last()
[0, 12, 0, 0, 0]
sage: W12.cardinality()
441
sage: for w in W12: print(w)
[4, 0, 0, 0, 0]
[3, 0, 0, 1, 1]
[3, 0, 1, 1, 0]
...
[0, 11, 1, 0, 0]
[0, 12, 0, 0, 0]
```

```
>>> from sage.all import *
>>> W = WeightedIntegerVectors([Integer(3),Integer(1),Integer(1),Integer(2),
↪Integer(1)]); W
Integer vectors weighted by [3, 1, 1, 2, 1]
>>> W.cardinality()
+Infinity

>>> W12 = W.graded_component(Integer(12))
>>> W12.an_element()
[4, 0, 0, 0, 0]
>>> W12.last()
[0, 12, 0, 0, 0]
>>> W12.cardinality()
441
>>> for w in W12: print(w)
[4, 0, 0, 0, 0]
[3, 0, 0, 1, 1]
[3, 0, 1, 1, 0]
...
[0, 11, 1, 0, 0]
[0, 12, 0, 0, 0]
```

grading(*x*)

EXAMPLES:

```
sage: C = WeightedIntegerVectors([2,1,3])
sage: C.grading((2,1,1))
8
```

```
>>> from sage.all import *
>>> C = WeightedIntegerVectors([Integer(2),Integer(1),Integer(3)])
>>> C.grading((Integer(2),Integer(1),Integer(1)))
8
```

subset(*size=None*)

EXAMPLES:

```
sage: C = WeightedIntegerVectors([2,1,3])
sage: C.subset(4)
Integer vectors of 4 weighted by [2, 1, 3]
```

```
>>> from sage.all import *
>>> C = WeightedIntegerVectors([Integer(2),Integer(1),Integer(3)])
>>> C.subset(Integer(4))
Integer vectors of 4 weighted by [2, 1, 3]
```

sage.combinat.integer_vector_weighted.**iterator_fast**(*n*, *l*)

Iterate over all `l` weighted integer vectors with total weight `n`.

INPUT:

- `n` – integer
- `l` – the weights in weakly decreasing order

EXAMPLES:

```
sage: from sage.combinat.integer_vector_weighted import iterator_fast
sage: list(iterator_fast(3, [2,1,1]))
[[1, 1, 0], [1, 0, 1], [0, 3, 0], [0, 2, 1], [0, 1, 2], [0, 0, 3]]
sage: list(iterator_fast(2, [2]))
[[1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.integer_vector_weighted import iterator_fast
>>> list(iterator_fast(Integer(3), [Integer(2),Integer(1),Integer(1)]))
[[1, 1, 0], [1, 0, 1], [0, 3, 0], [0, 2, 1], [0, 1, 2], [0, 0, 3]]
>>> list(iterator_fast(Integer(2), [Integer(2)]))
[[1]]
```

Test that Issue #20491 is fixed:

```
sage: type(list(iterator_fast(2, [2]))[0][0])
<class 'sage.rings.integer.Integer'>
```

```
>>> from sage.all import *
>>> type(list(iterator_fast(Integer(2), [Integer(2)]))[Integer(0)][Integer(0)])
<class 'sage.rings.integer.Integer'>
```

5.1.128 Integer vectors modulo the action of a permutation group

AUTHORS:

- Nicolas Borie (2010-2012) - original module
- Jukka Kohonen (2023) - fast cardinality method, Issue #36787, Issue #36681

class sage.combinat.integer_vectors_mod_permgroup.**IntegerVectorsModPermutationGroup**

Bases: `UniqueRepresentation`

Return an enumerated set containing integer vectors which are maximal in their orbit under the action of the permutation group `G` for the lexicographic order.

In Sage, a permutation group G is viewed as a subgroup of the symmetric group S_n of degree n and n is said to be the degree of G. Any integer vector v is said to be canonical if it is maximal in its orbit under the action of G. v is canonical if and only if

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

The action of G is on position. This means for example that the simple transposition $s_1 = (1, 2)$ swaps the first and the second entries of any integer vector $v = [a_1, a_2, a_3, \dots, a_n]$

$$s_1 \cdot v = [a_2, a_1, a_3, \dots, a_n]$$

This function returns a parent which contains, from each orbit orbit under the action of the permutation group G, a single canonical vector. The canonical vector is the one that is maximal within the orbit according to lexicographic order.

INPUT:

- `G` – a permutation group
- `sum` – (default: `None`) - a nonnegative integer
- `max_part` – (default: `None`) - a nonnegative integer setting the maximum value for every element
- `sgs` – (default: `None`) - a strong generating system of the group G. If you do not provide it, it will be calculated at the creation of the parent

OUTPUT:

- If `sum` and `max_part` are None, it returns the infinite enumerated set of all integer vectors (lists of integers) maximal in their orbit for the lexicographic order. Exceptionally, if the domain of `G` is empty, the result is a finite enumerated set that contains one element, namely the empty vector.
- If `sum` is an integer, it returns a finite enumerated set containing all integer vectors maximal in their orbit for the lexicographic order and whose entries sum to `sum`.

EXAMPLES:

Here is the set enumerating integer vectors modulo the action of the cyclic group of 3 elements:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3)]]))
sage: I.category()
Category of infinite enumerated quotients of sets
sage: I.cardinality()
+Infinity
sage: I.list()
Traceback (most recent call last):
...
NotImplementedError: cannot list an infinite set
sage: p = iter(I)
sage: for i in range(10): next(p)
[0, 0, 0]
[1, 0, 0]
[2, 0, 0]
[1, 1, 0]
[3, 0, 0]
[2, 1, 0]
[2, 0, 1]
[1, 1, 1]
[4, 0, 0]
[3, 1, 0]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3))]]))
```

(continues on next page)

(continued from previous page)

```
>>> I.category()
Category of infinite enumerated quotients of sets
>>> I.cardinality()
+Infinity
>>> I.list()
Traceback (most recent call last):
...
NotImplementedError: cannot list an infinite set
>>> p = iter(I)
>>> for i in range(Integer(10)): next(p)
[0, 0, 0]
[1, 0, 0]
[2, 0, 0]
[1, 1, 0]
[3, 0, 0]
[2, 1, 0]
[2, 0, 1]
[1, 1, 1]
[4, 0, 0]
[3, 1, 0]
```

The method `is_canonical()` tests if an integer vector is maximal in its orbit. This method is also used in the containment test:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.is_canonical([5,2,0,4])
True
sage: I.is_canonical([5,0,6,4])
False
sage: I.is_canonical([1,1,1,1])
True
sage: [2,3,1,0] in I
False
sage: [5,0,5,0] in I
True
sage: 'Bla' in I
False
sage: I.is_canonical('bla')
Traceback (most recent call last):
...
AssertionError: bla should be a list or an integer vector
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> I.is_canonical([Integer(5),Integer(2),Integer(0),Integer(4)])
True
>>> I.is_canonical([Integer(5),Integer(0),Integer(6),Integer(4)])
False
>>> I.is_canonical([Integer(1),Integer(1),Integer(1),Integer(1)])
True
>>> [Integer(2),Integer(3),Integer(1),Integer(0)] in I
```

(continues on next page)

(continued from previous page)

```
False
>>> [Integer(5),Integer(0),Integer(5),Integer(0)] in I
True
>>> 'Bla' in I
False
>>> I.is_canonical('bla')
Traceback (most recent call last):
...
AssertionError: bla should be a list or an integer vector
```

If you give a value to the extra argument `sum`, the set returned will be a finite set containing only canonical vectors whose entries sum to `sum`.:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3)]]), sum=6)
sage: I.cardinality()
10
sage: I.list()
[[6, 0, 0], [5, 1, 0], [5, 0, 1], [4, 2, 0], [4, 1, 1],
 [4, 0, 2], [3, 3, 0], [3, 2, 1], [3, 1, 2], [2, 2, 2]]
sage: I.category()
Join of Category of finite enumerated sets
    and Category of subquotients of finite sets
    and Category of quotients of sets
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3))]]), sum=Integer(6))
>>> I.cardinality()
10
>>> I.list()
[[6, 0, 0], [5, 1, 0], [5, 0, 1], [4, 2, 0], [4, 1, 1],
 [4, 0, 2], [3, 3, 0], [3, 2, 1], [3, 1, 2], [2, 2, 2]]
>>> I.category()
Join of Category of finite enumerated sets
    and Category of subquotients of finite sets
    and Category of quotients of sets
```

To get the orbit of any integer vector v under the action of the group, use the method `orbit()`; we convert the returned set of vectors into a list in increasing lexicographic order, to get a reproducible test:

```
sage: sorted(I.orbit([6,0,0]))
[[0, 0, 6], [0, 6, 0], [6, 0, 0]]
sage: sorted(I.orbit([5,1,0]))
[[0, 5, 1], [1, 0, 5], [5, 1, 0]]
sage: I.orbit([2,2,2])
{[2, 2, 2]}
```

```
>>> from sage.all import *
>>> sorted(I.orbit([Integer(6),Integer(0),Integer(0)]))
[[0, 0, 6], [0, 6, 0], [6, 0, 0]]
>>> sorted(I.orbit([Integer(5),Integer(1),Integer(0)]))
[[0, 5, 1], [1, 0, 5], [5, 1, 0]]
```

(continues on next page)

(continued from previous page)

```
>>> I.orbit([Integer(2),Integer(2),Integer(2)])
{[2, 2, 2]}
```

Even without constraints, for an empty domain the result is a singleton set:

```
sage: G = PermutationGroup([], domain=[])
sage: sgs = tuple(tuple(s) for s in G.strong_generating_system())
sage: list(IntegerVectorsModPermutationGroup(G, sgs=sgs))
[[]]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([], domain=[])
>>> sgs = tuple(tuple(s) for s in G.strong_generating_system())
>>> list(IntegerVectorsModPermutationGroup(G, sgs=sgs))
[[]]
```

Warning

Because of Issue #36527, permutation groups that have different domains but similar generators can be erroneously treated as the same group. This will silently produce erroneous results. To avoid this issue, compute a strong generating system for the group as:

```
sgs = tuple(tuple(s) for s in G.strong_generating_system())
```

and provide it as the optional `sgs` argument to the constructor.

class sage.combinat.integer_vectors_mod_permgroup.**IntegerVectorsModPermutationGroup_All**(*G*, *sgs=None*)

Bases: `UniqueRepresentation`, `RecursivelyEnumeratedSet_forest`

A class for integer vectors enumerated up to the action of a permutation group.

A Sage permutation group is viewed as a subgroup of the symmetric group S_n for a certain n. This group has a natural action by position on vectors of length n. This class implements a set which keeps a single vector for each orbit. We say that a vector is canonical if it is the maximum in its orbit under the action of the permutation group for the lexicographic order.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]])); I
Integer vectors of length 4 enumerated up to the action of
 Permutation Group with generators [(1,2,3,4)]
sage: I.cardinality()
+Infinity
sage: TestSuite(I).run()
sage: it = iter(I)
sage: [next(it), next(it), next(it), next(it), next(it)]
[[0, 0, 0, 0],
 [1, 0, 0, 0],
 [2, 0, 0, 0],
 [1, 1, 0, 0],
 [1, 0, 1, 0]]
```

(continues on next page)

(continued from previous page)

```
sage: x = next(it); x
[3, 0, 0, 0]
sage: I.first()
[0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]])); I
Integer vectors of length 4 enumerated up to the action of
 Permutation Group with generators [(1,2,3,4)]
>>> I.cardinality()
+Infinity
>>> TestSuite(I).run()
>>> it = iter(I)
>>> [next(it), next(it), next(it), next(it), next(it)]
[[0, 0, 0, 0],
 [1, 0, 0, 0],
 [2, 0, 0, 0],
 [1, 1, 0, 0],
 [1, 0, 1, 0]]
>>> x = next(it); x
[3, 0, 0, 0]
>>> I.first()
[0, 0, 0, 0]
```

class Element

Bases: `ClonableIntArray`

Element class for the set of integer vectors of given sum enumerated modulo the action of a permutation group. These vectors are clonable lists of integers which must satisfy conditions coming from the parent appearing in the method `check()`.

check()

Check that `self` verify the invariants needed for living in `self.parent()`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,
↪4)]]))
sage: v = I.an_element()
sage: v.check()
sage: w = I([0,4,0,0], check=False); w
[0, 4, 0, 0]
sage: w.check()
Traceback (most recent call last):
...
AssertionError
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> v = I.an_element()
>>> v.check()
```

(continues on next page)

(continued from previous page)

```
>>> w = I([Integer(0),Integer(4),Integer(0),Integer(0)], check=False); w
[0, 4, 0, 0]
>>> w.check()
Traceback (most recent call last):
...
AssertionError
```

ambient()

Return the ambient space from which `self` is a quotient.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: S.ambient()
Integer vectors of length 4
```

```
>>> from sage.all import *
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> S.ambient()
Integer vectors of length 4
```

children(*x*)

Return the list of children of the element `x`. This method is required to build the tree structure of `self` which inherits from the class `RecursivelyEnumeratedSet_forest`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.children(I([2,1,0,0], check=False))
[[2, 2, 0, 0], [2, 1, 1, 0], [2, 1, 0, 1]]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> I.children(I([Integer(2),Integer(1),Integer(0),Integer(0)], check=False))
[[2, 2, 0, 0], [2, 1, 1, 0], [2, 1, 0, 1]]
```

is_canonical(*v*, *check=True*)

Return `True` if the integer list `v` is maximal in its orbit under the action of the permutation group given to define `self`. Such integer vectors are said to be canonical. A vector v is canonical if and only if

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.is_canonical([4,3,2,1])
True
sage: I.is_canonical([4,0,0,1])
True
sage: I.is_canonical([4,0,3,3])
True
```

(continues on next page)

(continued from previous page)

```
sage: I.is_canonical([4,0,4,4])
False
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> I.is_canonical([Integer(4),Integer(3),Integer(2),Integer(1)])
True
>>> I.is_canonical([Integer(4),Integer(0),Integer(0),Integer(1)])
True
>>> I.is_canonical([Integer(4),Integer(0),Integer(3),Integer(3)])
True
>>> I.is_canonical([Integer(4),Integer(0),Integer(4),Integer(4)])
False
```

lift (*elt*)

Lift the element `elt` inside the ambient space from which `self` is a quotient.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: v = S.lift(S([4,3,0,1])); v
[4, 3, 0, 1]
sage: type(v)
<class 'list'>
```

```
>>> from sage.all import *
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> v = S.lift(S([Integer(4),Integer(3),Integer(0),Integer(1)])); v
[4, 3, 0, 1]
>>> type(v)
<class 'list'>
```

orbit (*v*)

Return the orbit of the integer vector `v` under the action of the permutation group defining `self`. The result is a set.

EXAMPLES:

In order to get reproducible doctests, we convert the returned sets into lists in increasing lexicographic order:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: sorted(I.orbit([2,2,0,0]))
[[0, 0, 2, 2], [0, 2, 2, 0], [2, 0, 0, 2], [2, 2, 0, 0]]
sage: sorted(I.orbit([2,1,0,0]))
[[0, 0, 2, 1], [0, 2, 1, 0], [1, 0, 0, 2], [2, 1, 0, 0]]
sage: sorted(I.orbit([2,0,1,0]))
[[0, 1, 0, 2], [0, 2, 0, 1], [1, 0, 2, 0], [2, 0, 1, 0]]
sage: sorted(I.orbit([2,0,2,0]))
[[0, 2, 0, 2], [2, 0, 2, 0]]
sage: I.orbit([1,1,1,1])
{[1, 1, 1, 1]}
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> sorted(I.orbit([Integer(2),Integer(2),Integer(0),Integer(0)]))
[[0, 0, 2, 2], [0, 2, 2, 0], [2, 0, 0, 2], [2, 2, 0, 0]]
>>> sorted(I.orbit([Integer(2),Integer(1),Integer(0),Integer(0)]))
[[0, 0, 2, 1], [0, 2, 1, 0], [1, 0, 0, 2], [2, 1, 0, 0]]
>>> sorted(I.orbit([Integer(2),Integer(0),Integer(1),Integer(0)]))
[[0, 1, 0, 2], [0, 2, 0, 1], [1, 0, 2, 0], [2, 0, 1, 0]]
>>> sorted(I.orbit([Integer(2),Integer(0),Integer(2),Integer(0)]))
[[0, 2, 0, 2], [2, 0, 2, 0]]
>>> I.orbit([Integer(1),Integer(1),Integer(1),Integer(1)])
{[1, 1, 1, 1]}
```

permutation_group()

Return the permutation group given to define `self`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.permutation_group()
Permutation Group with generators [(1,2,3,4)]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> I.permutation_group()
Permutation Group with generators [(1,2,3,4)]
```

retract(*elt*)

Return the canonical representative of the orbit of the integer `elt` under the action of the permutation group defining `self`.

If the element `elt` is already maximal in its orbit for the lexicographic order, `elt` is thus the good representative for its orbit.

EXAMPLES:

```
sage: [0,0,0,0] in IntegerVectors(0,4)
True
sage: [1,0,0,0] in IntegerVectors(1,4)
True
sage: [0,1,0,0] in IntegerVectors(1,4)
True
sage: [1,0,1,0] in IntegerVectors(2,4)
True
sage: [0,1,0,1] in IntegerVectors(2,4)
True
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: S.retract([0,0,0,0])
[0, 0, 0, 0]
sage: S.retract([1,0,0,0])
[1, 0, 0, 0]
sage: S.retract([0,1,0,0])
```

(continues on next page)

(continued from previous page)

```
[1, 0, 0, 0]
sage: S.retract([1,0,1,0])
[1, 0, 1, 0]
sage: S.retract([0,1,0,1])
[1, 0, 1, 0]
```

```
>>> from sage.all import *
>>> [Integer(0),Integer(0),Integer(0),Integer(0)] in␣
↪IntegerVectors(Integer(0),Integer(4))
True
>>> [Integer(1),Integer(0),Integer(0),Integer(0)] in␣
↪IntegerVectors(Integer(1),Integer(4))
True
>>> [Integer(0),Integer(1),Integer(0),Integer(0)] in␣
↪IntegerVectors(Integer(1),Integer(4))
True
>>> [Integer(1),Integer(0),Integer(1),Integer(0)] in␣
↪IntegerVectors(Integer(2),Integer(4))
True
>>> [Integer(0),Integer(1),Integer(0),Integer(1)] in␣
↪IntegerVectors(Integer(2),Integer(4))
True
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> S.retract([Integer(0),Integer(0),Integer(0),Integer(0)])
[0, 0, 0, 0]
>>> S.retract([Integer(1),Integer(0),Integer(0),Integer(0)])
[1, 0, 0, 0]
>>> S.retract([Integer(0),Integer(1),Integer(0),Integer(0)])
[1, 0, 0, 0]
>>> S.retract([Integer(1),Integer(0),Integer(1),Integer(0)])
[1, 0, 1, 0]
>>> S.retract([Integer(0),Integer(1),Integer(0),Integer(1)])
[1, 0, 1, 0]
```

roots()

Return the root of generation of `self`. This method is required to build the tree structure of `self` which inherits from the class `RecursivelyEnumeratedSet_forest`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.roots()
[[0, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> I.roots()
[[0, 0, 0, 0]]
```

subset(*sum=None*, *max_part=None*)

Return the subset of `self` containing integer vectors whose entries sum to `sum`.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: S.subset(4)
Integer vectors of length 4 and of sum 4 enumerated up to
the action of Permutation Group with generators
[(1,2,3,4)]
```

```
>>> from sage.all import *
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> S.subset(Integer(4))
Integer vectors of length 4 and of sum 4 enumerated up to
the action of Permutation Group with generators
[(1,2,3,4)]
```

class sage.combinat.integer_vectors_mod_permgroup.**IntegerVectorsModPermutationGroup_with_constraints**

Bases: UniqueRepresentation, RecursivelyEnumeratedSet_forest

This class models finite enumerated sets of integer vectors with constraint enumerated up to the action of a permutation group. Integer vectors are enumerated modulo the action of the permutation group. To implement that, we keep a single integer vector by orbit under the action of the permutation group. Elements chosen are vectors maximal in their orbit for the lexicographic order.

For more information see *IntegerVectorsModPermutationGroup*.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                       max_part=1)
sage: I.list()
[[0, 0, 0, 0], [1, 0, 0, 0], [1, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 0],
 [1, 1, 1, 1]]
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                       sum=6, max_part=4)
sage: I.list()
[[4, 2, 0, 0], [4, 1, 1, 0], [4, 1, 0, 1], [4, 0, 2, 0], [4, 0, 1, 1],
 [4, 0, 0, 2], [3, 3, 0, 0], [3, 2, 1, 0], [3, 2, 0, 1], [3, 1, 2, 0],
 [3, 1, 1, 1], [3, 1, 0, 2], [3, 0, 3, 0], [3, 0, 2, 1], [3, 0, 1, 2],
 [2, 2, 2, 0], [2, 2, 1, 1], [2, 1, 2, 1]]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                       max_part=Integer(1))
>>> I.list()
[[0, 0, 0, 0], [1, 0, 0, 0], [1, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 0],
 [1, 1, 1, 1]]
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                       sum=Integer(6), max_part=Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> I.list()
[[4, 2, 0, 0], [4, 1, 1, 0], [4, 1, 0, 1], [4, 0, 2, 0], [4, 0, 1, 1],
 [4, 0, 0, 2], [3, 3, 0, 0], [3, 2, 1, 0], [3, 2, 0, 1], [3, 1, 2, 0],
 [3, 1, 1, 1], [3, 1, 0, 2], [3, 0, 3, 0], [3, 0, 2, 1], [3, 0, 1, 2],
 [2, 2, 2, 0], [2, 2, 1, 1], [2, 1, 2, 1]]
```

Here is the enumeration of unlabeled graphs over 5 vertices:

```
sage: G = IntegerVectorsModPermutationGroup(TransitiveGroup(10,12), max_part=1)
sage: G.cardinality()
34
```

```
>>> from sage.all import *
>>> G = IntegerVectorsModPermutationGroup(TransitiveGroup(Integer(10),
↪Integer(12)), max_part=Integer(1))
>>> G.cardinality()
34
```

class Element

Bases: `ClonableIntArray`

Element class for the set of integer vectors with constraints enumerated modulo the action of a permutation group. These vectors are clonable lists of integers which must satisfy conditions coming from the parent as in the method *check()*.

check()

Check that `self` meets the constraints of being an element of `self.parent()`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,
↪4)]]), 4)
sage: v = I.an_element()
sage: v.check()
sage: w = I([0,4,0,0], check=False); w
[0, 4, 0, 0]
sage: w.check()
Traceback (most recent call last):
...
AssertionError
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]), Integer(4))
>>> v = I.an_element()
>>> v.check()
>>> w = I([Integer(0),Integer(4),Integer(0),Integer(0)], check=False); w
[0, 4, 0, 0]
>>> w.check()
Traceback (most recent call last):
...
AssertionError
```

ambient()

Return the ambient space from which `self` is a quotient.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),␣
↪6)
sage: S.ambient()
Integer vectors that sum to 6
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                       6, max_part=12)
sage: S.ambient()
Integer vectors that sum to 6 with constraints: max_part=12
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                       max_part=12)
sage: S.ambient()
Integer vectors with constraints: max_part=12
```

```
>>> from sage.all import *
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]), Integer(6))
>>> S.ambient()
Integer vectors that sum to 6
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                       Integer(6), max_part=Integer(12))
>>> S.ambient()
Integer vectors that sum to 6 with constraints: max_part=12
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                       max_part=Integer(12))
>>> S.ambient()
Integer vectors with constraints: max_part=12
```

an_element()

Return an element of `self`.

Raises an `EmptySetError` when `self` is empty.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                       sum=0, max_part=1)
sage: S.an_element()
[0, 0, 0, 0]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                       sum=1, max_part=1)
sage: S.an_element()
[1, 0, 0, 0]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                       sum=2, max_part=1)
sage: S.an_element()
[1, 1, 0, 0]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
```

(continues on next page)

(continued from previous page)

```
....:                                          sum=3, max_part=1)
sage: S.an_element()
[1, 1, 1, 0]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                          sum=4, max_part=1)
sage: S.an_element()
[1, 1, 1, 1]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                          sum=5, max_part=1)
sage: S.an_element()
Traceback (most recent call last):
...
EmptySetError
```

```
>>> from sage.all import *
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                          sum=Integer(0), max_part=Integer(1))
>>> S.an_element()
[0, 0, 0, 0]
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                          sum=Integer(1), max_part=Integer(1))
>>> S.an_element()
[1, 0, 0, 0]
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                          sum=Integer(2), max_part=Integer(1))
>>> S.an_element()
[1, 1, 0, 0]
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                          sum=Integer(3), max_part=Integer(1))
>>> S.an_element()
[1, 1, 1, 0]
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                          sum=Integer(4), max_part=Integer(1))
>>> S.an_element()
[1, 1, 1, 1]
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                          sum=Integer(5), max_part=Integer(1))
>>> S.an_element()
Traceback (most recent call last):
...
EmptySetError
```

cardinality()

Return the number of integer vectors in the set.

The algorithm utilises Cycle Index Theorem, allowing for a faster than a plain enumeration computation.

EXAMPLES:

With a trivial group all vectors are canonical:

```
sage: G = PermutationGroup([], domain=[1,2,3])
sage: IntegerVectorsModPermutationGroup(G, 5).cardinality()
21
sage: IntegerVectors(5, 3).cardinality()
21
```

```
>>> from sage.all import *
>>> G = PermutationGroup([], domain=[Integer(1),Integer(2),Integer(3)])
>>> IntegerVectorsModPermutationGroup(G, Integer(5)).cardinality()
21
>>> IntegerVectors(Integer(5), Integer(3)).cardinality()
21
```

With two interchangeable elements, the smaller one ranges from zero to `sum//2`:

```
sage: G = PermutationGroup([(1,2)])
sage: IntegerVectorsModPermutationGroup(G, 1000).cardinality()
501
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(1),Integer(2))])
>>> IntegerVectorsModPermutationGroup(G, Integer(1000)).cardinality()
501
```

Binary vectors up to full symmetry are first some ones and then some zeros:

```
sage: G = SymmetricGroup(10)
sage: I = IntegerVectorsModPermutationGroup(G, max_part=1)
sage: I.cardinality()
11
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(10))
>>> I = IntegerVectorsModPermutationGroup(G, max_part=Integer(1))
>>> I.cardinality()
11
```

Binary vectors of constant weight, up to PGL(2,17), which is 3-transitive, but not 4-transitive:

```
sage: G=PGL(2,17)
sage: I = IntegerVectorsModPermutationGroup(G, sum=3, max_part=1)
sage: I.cardinality()
1
sage: I = IntegerVectorsModPermutationGroup(G, sum=4, max_part=1)
sage: I.cardinality()
3
```

```
>>> from sage.all import *
>>> G=PGL(Integer(2),Integer(17))
```

(continues on next page)

(continued from previous page)

```
>>> I = IntegerVectorsModPermutationGroup(G, sum=Integer(3), max_
↪part=Integer(1))
>>> I.cardinality()
1
>>> I = IntegerVectorsModPermutationGroup(G, sum=Integer(4), max_
↪part=Integer(1))
>>> I.cardinality()
3
```

children(*x*)

Return the list of children of the element x.

This method is required to build the tree structure of self which inherits from the class RecursivelyEnumeratedSet_forest.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.children(I([2,1,0,0], check=False))
[[2, 2, 0, 0], [2, 1, 1, 0], [2, 1, 0, 1]]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> I.children(I([Integer(2),Integer(1),Integer(0),Integer(0)], check=False))
[[2, 2, 0, 0], [2, 1, 1, 0], [2, 1, 0, 1]]
```

is_canonical(*v*, *check=True*)

Return True if the integer list v is maximal in its orbit under the action of the permutation group given to define self. Such integer vectors are said to be canonical. A vector v is canonical if and only if

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                        max_part=3)
sage: I.is_canonical([3,0,0,0])
True
sage: I.is_canonical([1,0,2,0])
False
sage: I.is_canonical([2,0,1,0])
True
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                       max_part=Integer(3))
>>> I.is_canonical([Integer(3),Integer(0),Integer(0),Integer(0)])
True
>>> I.is_canonical([Integer(1),Integer(0),Integer(2),Integer(0)])
False
>>> I.is_canonical([Integer(2),Integer(0),Integer(1),Integer(0)])
True
```

lift(*elt*)

Lift the element `elt` inside the ambient space from which `self` is a quotient.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                        max_part=1)
sage: v = S.lift([1,0,1,0]); v
[1, 0, 1, 0]
sage: v in IntegerVectors(2,4,max_part=1)
True
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                        sum=6)
sage: v = S.lift(S.list()[5]); v
[4, 1, 1, 0]
sage: v in IntegerVectors(n=6)
True
```

```
>>> from sage.all import *
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                        max_part=Integer(1))
>>> v = S.lift([Integer(1),Integer(0),Integer(1),Integer(0)]); v
[1, 0, 1, 0]
>>> v in IntegerVectors(Integer(2),Integer(4),max_part=Integer(1))
True
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                        sum=Integer(6))
>>> v = S.lift(S.list()[Integer(5)]); v
[4, 1, 1, 0]
>>> v in IntegerVectors(n=Integer(6))
True
```

orbit(*v*)

Return the orbit of the vector `v` under the action of the permutation group defining `self`. The result is a set.

INPUT:

- `v` – an element of `self` or any list of length the degree of the permutation group

EXAMPLES:

We convert the result in a list in increasing lexicographic order, to get a reproducible doctest:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),␣
↪4)
sage: I.orbit([1,1,1,1])
{[1, 1, 1, 1]}
sage: sorted(I.orbit([3,0,0,1]))
[[0, 0, 1, 3], [0, 1, 3, 0], [1, 3, 0, 0], [3, 0, 0, 1]]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]), Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> I.orbit([Integer(1),Integer(1),Integer(1),Integer(1)])
{[1, 1, 1, 1]}
>>> sorted(I.orbit([Integer(3),Integer(0),Integer(0),Integer(1)]))
[[0, 0, 1, 3], [0, 1, 3, 0], [1, 3, 0, 0], [3, 0, 0, 1]]
```

permutation_group()

Return the permutation group given to define `self`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3)]]), 5)
sage: I.permutation_group()
Permutation Group with generators [(1,2,3)]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3))]]), Integer(5))
>>> I.permutation_group()
Permutation Group with generators [(1,2,3)]
```

retract(*elt*)

Return the canonical representative of the orbit of the integer `elt` under the action of the permutation group defining `self`.

If the element `elt` is already maximal in its orbits for the lexicographic order, `elt` is thus the good representative for its orbit.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),
....:                                       sum=2, max_part=1)
sage: S.retract([1,1,0,0])
[1, 1, 0, 0]
sage: S.retract([1,0,1,0])
[1, 0, 1, 0]
sage: S.retract([1,0,0,1])
[1, 1, 0, 0]
sage: S.retract([0,1,1,0])
[1, 1, 0, 0]
sage: S.retract([0,1,0,1])
[1, 0, 1, 0]
sage: S.retract([0,0,1,1])
[1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> S = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]),
...                                       sum=Integer(2), max_part=Integer(1))
>>> S.retract([Integer(1),Integer(1),Integer(0),Integer(0)])
[1, 1, 0, 0]
>>> S.retract([Integer(1),Integer(0),Integer(1),Integer(0)])
[1, 0, 1, 0]
>>> S.retract([Integer(1),Integer(0),Integer(0),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
[1, 1, 0, 0]
>>> S.retract([Integer(0),Integer(1),Integer(1),Integer(0)])
[1, 1, 0, 0]
>>> S.retract([Integer(0),Integer(1),Integer(0),Integer(1)])
[1, 0, 1, 0]
>>> S.retract([Integer(0),Integer(0),Integer(1),Integer(1)])
[1, 1, 0, 0]
```

roots()

Return the root of generation of `self`.

This method is required to build the tree structure of `self` which inherits from the class `RecursivelyEnumeratedSet_forest`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.roots()
[[0, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> I = IntegerVectorsModPermutationGroup(PermutationGroup([[(Integer(1),
↪Integer(2),Integer(3),Integer(4))]]))
>>> I.roots()
[[0, 0, 0, 0]]
```

5.1.129 Tamari Interval-posets

This module implements Tamari interval-posets: combinatorial objects which represent intervals of the Tamari order. They have been introduced in [CP2015] and allow for many combinatorial operations on Tamari intervals. In particular, they are linked to *DyckWords* and *BinaryTrees*. An introduction into Tamari interval-posets is given in Chapter 7 of [Pons2013].

The Tamari lattice can be defined as a lattice structure on either of several classes of Catalan objects, especially binary trees and Dyck paths [Tam1962] [HT1972] [Sta-EC2]. An interval can be seen as a pair of comparable elements. The number of intervals has been given in [Cha2008].

AUTHORS:

- Viviane Pons 2014: initial implementation
- Frédéric Chapoton 2014: review
- Darij Grinberg 2014: review
- Travis Scrimshaw 2014: review

`sage.combinat.interval_posets.`**TIP**

alias of *TamariIntervalPoset*

class `sage.combinat.interval_posets.`**TamariIntervalPoset**(*parent*, *size*, *relations=None*, *check=True*)

Bases: `Element`

The class of Tamari interval-posets.

An interval-poset is a labelled poset of size n, with labels $1, 2, \ldots, n$, satisfying the following conditions:

- if $a < c$ (as integers) and a precedes c in the poset, then, for all b such that $a < b < c$, b precedes c,

- if $a < c$ (as integers) and c precedes a in the poset, then, for all b such that $a < b < c$, b precedes a.

We use the word "precedes" here to distinguish the poset order and the natural order on numbers. "Precedes" means "is smaller than with respect to the poset structure"; this does not imply a covering relation.

Interval-posets of size n are in bijection with intervals of the Tamari lattice of binary trees of size n. Specifically, if P is an interval-poset of size n, then the set of linear extensions of P (as permutations in S_n) is an interval in the right weak order (see *permutohedron_lequal()*), and is in fact the preimage of an interval in the Tamari lattice (of binary trees of size n) under the operation which sends a permutation to its right-to-left binary search tree (*binary_search_tree()* with the `left_to_right` variable set to `False`) without its labelling.

INPUT:

- `size` – integer; the size of the interval-posets (number of vertices)
- `relations` – list (or tuple) of pairs `(a,b)` (themselves lists or tuples), each representing a relation of the form 'a precedes b' in the poset
- `check` – boolean (default: `True`); whether to check the interval-poset condition or not

Warning

The `relations` input can be a list or tuple, but not an iterator (nor should its entries be iterators).

NOTATION:

Here and in the following, the signs $<$ and $>$ always refer to the natural ordering on integers, whereas the word "precedes" refers to the order of the interval-poset. "Minimal" and "maximal" refer to the natural ordering on integers.

The *increasing relations* of an interval-poset P mean the pairs (a, b) of elements of P such that $a < b$ as integers and a precedes b in P. The *initial forest* of P is the poset obtained by imposing (only) the increasing relations on the ground set of P. It is a sub-interval poset of P, and is a forest with its roots on top. This forest is usually given the structure of a planar forest by ordering brother nodes by their labels; it then has the property that if its nodes are traversed in post-order (see *post_order_traversal()*, and traverse the trees of the forest from left to right as well), then the labels encountered are $1, 2, \ldots, n$ in this order.

The *decreasing relations* of an interval-poset P mean the pairs (a, b) of elements of P such that $b < a$ as integers and a precedes b in P. The *final forest* of P is the poset obtained by imposing (only) the decreasing relations on the ground set of P. It is a sub-interval poset of P, and is a forest with its roots on top. This forest is usually given the structure of a planar forest by ordering brother nodes by their labels; it then has the property that if its nodes are traversed in pre-order (see *pre_order_traversal()*, and traverse the trees of the forest from left to right as well), then the labels encountered are $1, 2, \ldots, n$ in this order.

EXAMPLES:

```
sage: TamariIntervalPoset(0,[])
The Tamari interval of size 0 induced by relations []
sage: TamariIntervalPoset(3,[])
The Tamari interval of size 3 induced by relations []
sage: TamariIntervalPoset(3,[(1,2)])
The Tamari interval of size 3 induced by relations [(1, 2)]
sage: TamariIntervalPoset(3,[(1,2),(2,3)])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
sage: TamariIntervalPoset(3,[(1,2),(2,3),(1,3)])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
sage: TamariIntervalPoset(3,[(1,2),(3,2)])
```

(continues on next page)

(continued from previous page)

```
The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)]
sage: TamariIntervalPoset(3,[[1,2],[2,3]])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
sage: TamariIntervalPoset(3,[[1,2],[2,3],[1,2],[1,3]])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]

sage: TamariIntervalPoset(3,[(3,4)])
Traceback (most recent call last):
...
ValueError: the relations do not correspond to the size of the poset

sage: TamariIntervalPoset(2,[(2,1),(1,2)])
Traceback (most recent call last):
...
ValueError: The graph is not directed acyclic

sage: TamariIntervalPoset(3,[(1,3)])
Traceback (most recent call last):
...
ValueError: this does not satisfy the Tamari interval-poset condition
```

```
>>> from sage.all import *
>>> TamariIntervalPoset(Integer(0),[])
The Tamari interval of size 0 induced by relations []
>>> TamariIntervalPoset(Integer(3),[])
The Tamari interval of size 3 induced by relations []
>>> TamariIntervalPoset(Integer(3),[(Integer(1),Integer(2))])
The Tamari interval of size 3 induced by relations [(1, 2)]
>>> TamariIntervalPoset(Integer(3),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
>>> TamariIntervalPoset(Integer(3),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(1),Integer(3))])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
>>> TamariIntervalPoset(Integer(3),[(Integer(1),Integer(2)),(Integer(3),
↪Integer(2))])
The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)]
>>> TamariIntervalPoset(Integer(3),[[Integer(1),Integer(2)],[Integer(2),
↪Integer(3)]])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
>>> TamariIntervalPoset(Integer(3),[[Integer(1),Integer(2)],[Integer(2),
↪Integer(3)],[Integer(1),Integer(2)],[Integer(1),Integer(3)]])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]

>>> TamariIntervalPoset(Integer(3),[(Integer(3),Integer(4))])
Traceback (most recent call last):
...
ValueError: the relations do not correspond to the size of the poset

>>> TamariIntervalPoset(Integer(2),[(Integer(2),Integer(1)),(Integer(1),
↪Integer(2))])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: The graph is not directed acyclic

>>> TamariIntervalPoset(Integer(3),[(Integer(1),Integer(3))])
Traceback (most recent call last):
...
ValueError: this does not satisfy the Tamari interval-poset condition
```

It is also possible to transform a poset directly into an interval-poset:

```
sage: TIP = TamariIntervalPosets()
sage: p = Poset(([1,2,3], [(1,2)]))
sage: TIP(p)
The Tamari interval of size 3 induced by relations [(1, 2)]
sage: TIP(Poset({1: []}))
The Tamari interval of size 1 induced by relations []
sage: TIP(Poset({}))
The Tamari interval of size 0 induced by relations []
```

```
>>> from sage.all import *
>>> TIP = TamariIntervalPosets()
>>> p = Poset(([Integer(1),Integer(2),Integer(3)], [(Integer(1),Integer(2))]))
>>> TIP(p)
The Tamari interval of size 3 induced by relations [(1, 2)]
>>> TIP(Poset({Integer(1): []}))
The Tamari interval of size 1 induced by relations []
>>> TIP(Poset({}))
The Tamari interval of size 0 induced by relations []
```

binary_trees()

Return an iterator on all the binary trees in the interval represented by `self`.

See also

interval_cardinality()

EXAMPLES:

```
sage: list(TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)]).binary_trees())
[[., [[., [., .]], .]],
 [[., [., [., .]]], .],
 [., [[[., .], .], .]],
 [[., [[., .], .]], .]]
sage: set(TamariIntervalPoset(4,[]).binary_trees()) == set(BinaryTrees(4))
True
```

```
>>> from sage.all import *
>>> list(TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))]).binary_
↪trees())
[[., [[., [., .]], .]],
```

(continues on next page)

(continued from previous page)

```
 [[., [., [., .]]], .],
 [., [[[., .], .], .]],
 [[., [[., .], .]], .]]
>>> set(TamariIntervalPoset(Integer(4),[]).binary_trees()) ==␣
↪set(BinaryTrees(Integer(4)))
True
```

complement()

Return the complement of the interval-poset `self`.

If P is a Tamari interval-poset of size n, then the *complement* of P is defined as the interval-poset Q whose base set is $[n] = \{1, 2, \ldots, n\}$ (just as for P), but whose order relation has a precede b if and only if $n+1-a$ precedes $n+1-b$ in P.

In terms of the Tamari lattice, the *complement* is the symmetric of `self`. It is formed from the left-right symmetrized of the binary trees of the interval (switching left and right subtrees, see `left_right_symmetry()`). In particular, initial intervals are sent to final intervals and vice-versa.

EXAMPLES:

```
sage: TamariIntervalPoset(3, [(2, 1), (3, 1)]).complement()
The Tamari interval of size 3 induced by relations [(1, 3), (2, 3)]
sage: TamariIntervalPoset(0, []).complement()
The Tamari interval of size 0 induced by relations []
sage: ip = TamariIntervalPoset(4, [(1, 2), (2, 4), (3, 4)])
sage: ip.complement() == TamariIntervalPoset(4, [(2, 1), (3, 1), (4, 3)])
True
sage: ip.lower_binary_tree() == ip.complement().upper_binary_tree().left_
↪right_symmetry()
True
sage: ip.upper_binary_tree() == ip.complement().lower_binary_tree().left_
↪right_symmetry()
True
sage: ip.is_initial_interval()
True
sage: ip.complement().is_final_interval()
True
```

```
>>> from sage.all import *
>>> TamariIntervalPoset(Integer(3), [(Integer(2), Integer(1)), (Integer(3),␣
↪Integer(1))]).complement()
The Tamari interval of size 3 induced by relations [(1, 3), (2, 3)]
>>> TamariIntervalPoset(Integer(0), []).complement()
The Tamari interval of size 0 induced by relations []
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(1), Integer(2)),␣
↪(Integer(2), Integer(4)), (Integer(3), Integer(4))])
>>> ip.complement() == TamariIntervalPoset(Integer(4), [(Integer(2),␣
↪Integer(1)), (Integer(3), Integer(1)), (Integer(4), Integer(3))])
True
>>> ip.lower_binary_tree() == ip.complement().upper_binary_tree().left_right_
↪symmetry()
True
>>> ip.upper_binary_tree() == ip.complement().lower_binary_tree().left_right_
```

(continues on next page)

(continued from previous page)

```
↪symmetry()
True
>>> ip.is_initial_interval()
True
>>> ip.complement().is_final_interval()
True
```

contains_binary_tree(*binary_tree*)

Return whether the interval represented by `self` contains the binary tree `binary_tree`.

INPUT:

- `binary_tree` – a binary tree

See also:

contains_dyck_word()

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.contains_binary_tree(BinaryTree([[None,[None,[]]],None]))
True
sage: ip.contains_binary_tree(BinaryTree([None,[[[],None],None]]))
True
sage: ip.contains_binary_tree(BinaryTree([[],[[],None]]))
False
sage: ip.contains_binary_tree(ip.lower_binary_tree())
True
sage: ip.contains_binary_tree(ip.upper_binary_tree())
True
sage: all(ip.contains_binary_tree(bt) for bt in ip.binary_trees())
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> ip.contains_binary_tree(BinaryTree([[None,[None,[]]],None]))
True
>>> ip.contains_binary_tree(BinaryTree([None,[[[],None],None]]))
True
>>> ip.contains_binary_tree(BinaryTree([[],[[],None]]))
False
>>> ip.contains_binary_tree(ip.lower_binary_tree())
True
>>> ip.contains_binary_tree(ip.upper_binary_tree())
True
>>> all(ip.contains_binary_tree(bt) for bt in ip.binary_trees())
True
```

contains_dyck_word(*dyck_word*)

Return whether the interval represented by `self` contains the Dyck word `dyck_word`.

INPUT:

- `dyck_word` – a Dyck word

See also

`contains_binary_tree()`

EXAMPLES:

```
sage: # needs sage.combinat
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.contains_dyck_word(DyckWord([1,1,1,0,0,0,1,0]))
True
sage: ip.contains_dyck_word(DyckWord([1,1,0,1,0,1,0,0]))
True
sage: ip.contains_dyck_word(DyckWord([1,0,1,1,0,1,0,0]))
False
sage: ip.contains_dyck_word(ip.lower_dyck_word())
True
sage: ip.contains_dyck_word(ip.upper_dyck_word())
True
sage: all(ip.contains_dyck_word(bt) for bt in ip.dyck_words())
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> ip.contains_dyck_word(DyckWord([Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0),Integer(1),Integer(0)]))
True
>>> ip.contains_dyck_word(DyckWord([Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(1),Integer(0),Integer(0)]))
True
>>> ip.contains_dyck_word(DyckWord([Integer(1),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(1),Integer(0),Integer(0)]))
False
>>> ip.contains_dyck_word(ip.lower_dyck_word())
True
>>> ip.contains_dyck_word(ip.upper_dyck_word())
True
>>> all(ip.contains_dyck_word(bt) for bt in ip.dyck_words())
True
```

contains_interval(*other*)

Return whether the interval represented by `other` is contained in `self` as an interval of the Tamari lattice.

In terms of interval-posets, it means that all relations of `self` are relations of `other`.

INPUT:

- `other` – an interval-poset

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip2 = TamariIntervalPoset(4,[(2,3)])
sage: ip2.contains_interval(ip1)
True
sage: ip3 = TamariIntervalPoset(4,[(2,1)])
sage: ip2.contains_interval(ip3)
False
sage: ip4 = TamariIntervalPoset(3,[(2,3)])
sage: ip2.contains_interval(ip4)
False
```

```
>>> from sage.all import *
>>> ip1 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
→Integer(3)),(Integer(4),Integer(3))])
>>> ip2 = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(3))])
>>> ip2.contains_interval(ip1)
True
>>> ip3 = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(1))])
>>> ip2.contains_interval(ip3)
False
>>> ip4 = TamariIntervalPoset(Integer(3),[(Integer(2),Integer(3))])
>>> ip2.contains_interval(ip4)
False
```

cubical_coordinates()

Return the cubical coordinates of `self`.

This provides a fast and natural way to order the set of interval-posets of a given size.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.cubical_coordinates()
(-1, -2, 0)
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
→Integer(3))])
>>> ip.cubical_coordinates()
(-1, -2, 0)
```

REFERENCES:

- [Com2019]

decomposition_to_triple()

Decompose an interval-poset into a triple (`left`, `right`, `r`).

For the inverse method, see *TamariIntervalPosets.recomposition_from_triple()*.

OUTPUT:

a triple (`left`, `right`, `r`) where `left` and `right` are interval-posets and `r` (an integer) is the parameter of the decomposition.

EXAMPLES:

```
sage: tip = TamariIntervalPoset(8, [(1,2), (2,4), (3,4), (6,7),
....:                             (3,2), (5,4), (6,4), (8,7)])
sage: tip.decomposition_to_triple()
(The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)],
 The Tamari interval of size 4 induced by relations [(2, 3), (4, 3)],
 2)
sage: tip == TamariIntervalPosets.recomposition_from_triple(
....:           *tip.decomposition_to_triple())
True
```

```
>>> from sage.all import *
>>> tip = TamariIntervalPoset(Integer(8), [(Integer(1),Integer(2)),␣
↪(Integer(2),Integer(4)), (Integer(3),Integer(4)), (Integer(6),Integer(7)),
...                             (Integer(3),Integer(2)), (Integer(5),
↪Integer(4)), (Integer(6),Integer(4)), (Integer(8),Integer(7))])
>>> tip.decomposition_to_triple()
(The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)],
 The Tamari interval of size 4 induced by relations [(2, 3), (4, 3)],
 2)
>>> tip == TamariIntervalPosets.recomposition_from_triple(
...           *tip.decomposition_to_triple())
True
```

REFERENCES:

- [CP2015]

decreasing_children(*v*)

Return the children of `v` in the final forest of `self`.

INPUT:

- `v` – integer representing a vertex of `self` (between 1 and `size`)

OUTPUT:

The list of all children of `v` in the final forest of `self`, in increasing order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]);
↪ ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.decreasing_children(2)
[3, 5]
sage: ip.decreasing_children(3)
[4]
sage: ip.decreasing_children(1)
[]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(3),Integer(5)),(Integer(4),Integer(5))]); ip
```

(continues on next page)

(continued from previous page)

```
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),
↪(6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.decreasing_children(Integer(2))
[3, 5]
>>> ip.decreasing_children(Integer(3))
[4]
>>> ip.decreasing_children(Integer(1))
[]
```

decreasing_cover_relations()

Return the cover relations of the final forest of `self`.

This is the poset formed by keeping only the relations of the form a precedes b with $a > b$.

The final forest of `self` is a forest with its roots being on top. It is also called the decreasing poset of `self`.

Warning

This method computes the cover relations of the final forest. This is not identical with the cover relations of `self` which happen to be decreasing!

See also

final_forest()

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(2,1),(3,2),(3,4),(4,2)]).decreasing_cover_
↪relations()
[(4, 2), (3, 2), (2, 1)]
sage: TamariIntervalPoset(4,[(2,1),(4,3),(2,3)]).decreasing_cover_relations()
[(4, 3), (2, 1)]
sage: TamariIntervalPoset(3,[(2,1),(3,1),(3,2)]).decreasing_cover_relations()
[(3, 2), (2, 1)]
```

```
>>> from sage.all import *
>>> TamariIntervalPoset(Integer(4),[(Integer(2),Integer(1)),(Integer(3),
↪Integer(2)),(Integer(3),Integer(4)),(Integer(4),Integer(2))]).decreasing_
↪cover_relations()
[(4, 2), (3, 2), (2, 1)]
>>> TamariIntervalPoset(Integer(4),[(Integer(2),Integer(1)),(Integer(4),
↪Integer(3)),(Integer(2),Integer(3))]).decreasing_cover_relations()
[(4, 3), (2, 1)]
>>> TamariIntervalPoset(Integer(3),[(Integer(2),Integer(1)),(Integer(3),
↪Integer(1)),(Integer(3),Integer(2))]).decreasing_cover_relations()
[(3, 2), (2, 1)]
```

decreasing_parent(*v*)

Return the vertex parent of `v` in the final forest of `self`.

This is the highest (as integer!) vertex $a < v$ such that `v` precedes `a`. If there is no such vertex (that is, v is a decreasing root), then `None` is returned.

INPUT:

- `v` – integer representing a vertex of `self` (between 1 and `size`)

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]);
↪ ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.decreasing_parent(4)
3
sage: ip.decreasing_parent(3)
2
sage: ip.decreasing_parent(5)
2
sage: ip.decreasing_parent(2) is None
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(3),Integer(5)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.decreasing_parent(Integer(4))
3
>>> ip.decreasing_parent(Integer(3))
2
>>> ip.decreasing_parent(Integer(5))
2
>>> ip.decreasing_parent(Integer(2)) is None
True
```

decreasing_roots()

Return the root vertices of the final forest of `self`.

These are the vertices b such that there is no $a < b$ with b preceding a.

OUTPUT:

The list of all roots of the final forest of `self`, in increasing order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]);
↪ ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.decreasing_roots()
[1, 2]
sage: ip.final_forest().decreasing_roots()
[1, 2]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
→Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
→Integer(2)),(Integer(3),Integer(5)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
→(6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.decreasing_roots()
[1, 2]
>>> ip.final_forest().decreasing_roots()
[1, 2]
```

dyck_words()

Return an iterator on all the Dyck words in the interval represented by `self`.

EXAMPLES:

```
sage: list(TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)]).dyck_words())    #␣
→needs sage.combinat
[[1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 0, 1, 0, 0, 1, 0]]
sage: set(TamariIntervalPoset(4,[]).dyck_words()) == set(DyckWords(4))       #␣
→needs sage.combinat
True
```

```
>>> from sage.all import *
>>> list(TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
→Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))]).dyck_words())␣
→   # needs sage.combinat
[[1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 0, 1, 0, 0, 1, 0]]
>>> set(TamariIntervalPoset(Integer(4),[]).dyck_words()) ==␣
→set(DyckWords(Integer(4)))       # needs sage.combinat
True
```

factor()

Return the unique decomposition as a list of connected components.

EXAMPLES:

```
sage: factor(TamariIntervalPoset(2,[]))  # indirect doctest
[The Tamari interval of size 1 induced by relations [],
 The Tamari interval of size 1 induced by relations []]
```

```
>>> from sage.all import *
>>> factor(TamariIntervalPoset(Integer(2),[]))  # indirect doctest
[The Tamari interval of size 1 induced by relations [],
 The Tamari interval of size 1 induced by relations []]
```

See also

is_connected()

final_forest()

Return the final forest of `self`, i.e., the interval-poset formed with only the decreasing relations of `self`.

See also

initial_forest()

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(2,1),(3,2),(3,4),(4,2)]).final_forest()
The Tamari interval of size 4 induced by relations [(4, 2), (3, 2), (2, 1)]
sage: ip = TamariIntervalPoset(3,[(2,1),(3,1)])
sage: ip.final_forest() == ip
True
```

```
>>> from sage.all import *
>>> TamariIntervalPoset(Integer(4),[(Integer(2),Integer(1)),(Integer(3),
↪Integer(2)),(Integer(3),Integer(4)),(Integer(4),Integer(2))]).final_forest()
The Tamari interval of size 4 induced by relations [(4, 2), (3, 2), (2, 1)]
>>> ip = TamariIntervalPoset(Integer(3),[(Integer(2),Integer(1)),(Integer(3),
↪Integer(1))])
>>> ip.final_forest() == ip
True
```

ge(*e1*, *e2*)

Return whether `e2` precedes or equals `e1` in `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.ge(2,1)
True
sage: ip.ge(3,1)
True
sage: ip.ge(3,2)
True
sage: ip.ge(4,3)
False
sage: ip.ge(1,1)
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
>>> ip.ge(Integer(2),Integer(1))
True
>>> ip.ge(Integer(3),Integer(1))
```

(continues on next page)

(continued from previous page)

```
True
>>> ip.ge(Integer(3),Integer(2))
True
>>> ip.ge(Integer(4),Integer(3))
False
>>> ip.ge(Integer(1),Integer(1))
True
```

grafting_tree()

Return the grafting tree of the interval-poset.

For the inverse method, see *TamariIntervalPosets.from_grafting_tree()*.

EXAMPLES:

```
sage: tip = TamariIntervalPoset(8, [(1,2), (2,4), (3,4), (6,7),
....:                              (3,2), (5,4), (6,4), (8,7)])
sage: tip.grafting_tree()
2[1[0[., .], 0[., .]], 0[., 1[0[., .], 0[., .]]]]
sage: tip == TamariIntervalPosets.from_grafting_tree(tip.grafting_tree())
True
```

```
>>> from sage.all import *
>>> tip = TamariIntervalPoset(Integer(8), [(Integer(1),Integer(2)),␣
↪(Integer(2),Integer(4)), (Integer(3),Integer(4)), (Integer(6),Integer(7)),
...                              (Integer(3),Integer(2)), (Integer(5),
↪Integer(4)), (Integer(6),Integer(4)), (Integer(8),Integer(7))])
>>> tip.grafting_tree()
2[1[0[., .], 0[., .]], 0[., 1[0[., .], 0[., .]]]]
>>> tip == TamariIntervalPosets.from_grafting_tree(tip.grafting_tree())
True
```

REFERENCES:

- [Pons2018]

gt(*e1*, *e2*)

Return whether `e2` strictly precedes `e1` in `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.gt(2,1)
True
sage: ip.gt(3,1)
True
sage: ip.gt(3,2)
True
sage: ip.gt(4,3)
False
sage: ip.gt(1,1)
False
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
>>> ip.gt(Integer(2),Integer(1))
True
>>> ip.gt(Integer(3),Integer(1))
True
>>> ip.gt(Integer(3),Integer(2))
True
>>> ip.gt(Integer(4),Integer(3))
False
>>> ip.gt(Integer(1),Integer(1))
False
```

increasing_children(v)

Return the children of v in the initial forest of self.

INPUT:

- v – integer representing a vertex of self (between 1 and size)

OUTPUT:

The list of all children of v in the initial forest of self, in decreasing order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]);
↪ ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.increasing_children(2)
[1]
sage: ip.increasing_children(5)
[4, 3]
sage: ip.increasing_children(1)
[]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(3),Integer(5)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.increasing_children(Integer(2))
[1]
>>> ip.increasing_children(Integer(5))
[4, 3]
>>> ip.increasing_children(Integer(1))
[]
```

increasing_cover_relations()

Return the cover relations of the initial forest of self.

This is the poset formed by keeping only the relations of the form a precedes b with $a < b$.

The initial forest of `self` is a forest with its roots being on top. It is also called the increasing poset of `self`.

Warning: This method computes the cover relations of the initial forest. This is not identical with the cover relations of `self` which happen to be increasing!

See also:

initial_forest()

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(1,2),(3,2),(2,4),(3,4)]).increasing_cover_
↪relations()
[(1, 2), (2, 4), (3, 4)]
sage: TamariIntervalPoset(3,[(1,2),(1,3),(2,3)]).increasing_cover_relations()
[(1, 2), (2, 3)]
```

```
>>> from sage.all import *
>>> TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(3),
↪Integer(2)),(Integer(2),Integer(4)),(Integer(3),Integer(4))]).increasing_
↪cover_relations()
[(1, 2), (2, 4), (3, 4)]
>>> TamariIntervalPoset(Integer(3),[(Integer(1),Integer(2)),(Integer(1),
↪Integer(3)),(Integer(2),Integer(3))]).increasing_cover_relations()
[(1, 2), (2, 3)]
```

increasing_parent(*v*)

Return the vertex parent of `v` in the initial forest of `self`.

This is the lowest (as integer!) vertex $b > v$ such that v precedes b. If there is no such vertex (that is, v is an increasing root), then `None` is returned.

INPUT:

- `v` – integer representing a vertex of `self` (between 1 and `size`)

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]);
↪ ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.increasing_parent(1)
2
sage: ip.increasing_parent(3)
5
sage: ip.increasing_parent(4)
5
sage: ip.increasing_parent(5) is None
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(3),Integer(5)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.increasing_parent(Integer(1))
2
>>> ip.increasing_parent(Integer(3))
5
>>> ip.increasing_parent(Integer(4))
5
>>> ip.increasing_parent(Integer(5)) is None
True
```

increasing_roots()

Return the root vertices of the initial forest of `self`.

These are the vertices a of `self` such that there is no $b > a$ with a precedes b.

OUTPUT:

The list of all roots of the initial forest of `self`, in decreasing order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]);
↪ ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.increasing_roots()
[6, 5, 2]
sage: ip.initial_forest().increasing_roots()
[6, 5, 2]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(3),Integer(5)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5),␣
↪(6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.increasing_roots()
[6, 5, 2]
>>> ip.initial_forest().increasing_roots()
[6, 5, 2]
```

initial_forest()

Return the initial forest of `self`, i.e., the interval-poset formed from only the increasing relations of `self`.

See also

final_forest()

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(1,2),(3,2),(2,4),(3,4)]).initial_forest()
The Tamari interval of size 4 induced by relations [(1, 2), (2, 4), (3, 4)]
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.initial_forest() == ip
True
```

```
>>> from sage.all import *
>>> TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(3),
↪Integer(2)),(Integer(2),Integer(4)),(Integer(3),Integer(4))]).initial_
↪forest()
The Tamari interval of size 4 induced by relations [(1, 2), (2, 4), (3, 4)]
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
>>> ip.initial_forest() == ip
True
```

insertion(i)

Return the Tamari insertion of an integer i into the interval-poset `self`.

If P is a Tamari interval-poset of size n and i is an integer with $1 \leq i \leq n+1$, then the Tamari insertion of i into P is defined as the Tamari interval-poset of size $n+1$ which corresponds to the interval $[C_1, C_2]$ on the Tamari lattice, where the binary trees C_1 and C_2 are defined as follows: We write the interval-poset P as $[B_1, B_2]$ for two binary trees B_1 and B_2. We label the vertices of each of these two trees with the integers $1, 2, \ldots, i-1, i+1, i+2, \ldots, n+1$ in such a way that the trees are binary search trees (this labelling is unique). Then, we insert i into each of these trees (in the way as explained in *binary_search_insert()*). The shapes of the resulting two trees are denoted C_1 and C_2.

An alternative way to construct the insertion of i into P is by relabeling each vertex u of P satisfying $u \geq i$ (as integers) as $u+1$, and then adding a vertex i which should precede $i-1$ and $i+1$.

Todo

To study this, it would be more natural to define interval-posets on arbitrary ordered sets rather than just on $\{1, 2, \ldots, n\}$.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4, [(2, 3), (4, 3)]); ip
The Tamari interval of size 4 induced by relations [(2, 3), (4, 3)]
sage: ip.insertion(1)
The Tamari interval of size 5 induced by relations [(1, 2), (3, 4), (5, 4)]
sage: ip.insertion(2)
The Tamari interval of size 5 induced by relations [(2, 3), (3, 4), (5, 4),␣
↪(2, 1)]
sage: ip.insertion(3)
The Tamari interval of size 5 induced by relations [(2, 4), (3, 4), (5, 4),␣
↪(3, 2)]
sage: ip.insertion(4)
The Tamari interval of size 5 induced by relations [(2, 3), (4, 5), (5, 3),␣
↪(4, 3)]
sage: ip.insertion(5)
The Tamari interval of size 5 induced by relations [(2, 3), (5, 4), (4, 3)]
```

(continues on next page)

(continued from previous page)

```
sage: ip = TamariIntervalPoset(0, [])
sage: ip.insertion(1)
The Tamari interval of size 1 induced by relations []

sage: ip = TamariIntervalPoset(1, [])
sage: ip.insertion(1)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: ip.insertion(2)
The Tamari interval of size 2 induced by relations [(2, 1)]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(2), Integer(3)),␣
↪(Integer(4), Integer(3))]); ip
The Tamari interval of size 4 induced by relations [(2, 3), (4, 3)]
>>> ip.insertion(Integer(1))
The Tamari interval of size 5 induced by relations [(1, 2), (3, 4), (5, 4)]
>>> ip.insertion(Integer(2))
The Tamari interval of size 5 induced by relations [(2, 3), (3, 4), (5, 4),␣
↪(2, 1)]
>>> ip.insertion(Integer(3))
The Tamari interval of size 5 induced by relations [(2, 4), (3, 4), (5, 4),␣
↪(3, 2)]
>>> ip.insertion(Integer(4))
The Tamari interval of size 5 induced by relations [(2, 3), (4, 5), (5, 3),␣
↪(4, 3)]
>>> ip.insertion(Integer(5))
The Tamari interval of size 5 induced by relations [(2, 3), (5, 4), (4, 3)]

>>> ip = TamariIntervalPoset(Integer(0), [])
>>> ip.insertion(Integer(1))
The Tamari interval of size 1 induced by relations []

>>> ip = TamariIntervalPoset(Integer(1), [])
>>> ip.insertion(Integer(1))
The Tamari interval of size 2 induced by relations [(1, 2)]
>>> ip.insertion(Integer(2))
The Tamari interval of size 2 induced by relations [(2, 1)]
```

intersection(*other*)

Return the interval-poset formed by combining the relations from both `self` and `other`. It corresponds to the intersection of the two corresponding intervals of the Tamari lattice.

INPUT:

- `other` – an interval-poset of the same size as `self`

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip2 = TamariIntervalPoset(4,[(4,3)])
sage: ip1.intersection(ip2)
The Tamari interval of size 4 induced by relations [(1, 2), (2, 3), (4, 3)]
```

(continues on next page)

(continued from previous page)

```
sage: ip3 = TamariIntervalPoset(4,[(2,1)])
sage: ip1.intersection(ip3)
Traceback (most recent call last):
...
ValueError: this intersection is empty, it does not correspond to an interval-
↪poset
sage: ip4 = TamariIntervalPoset(3,[(2,3)])
sage: ip2.intersection(ip4)
Traceback (most recent call last):
...
ValueError: intersections are only possible on interval-posets of the same␣
↪size
```

```
>>> from sage.all import *
>>> ip1 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
>>> ip2 = TamariIntervalPoset(Integer(4),[(Integer(4),Integer(3))])
>>> ip1.intersection(ip2)
The Tamari interval of size 4 induced by relations [(1, 2), (2, 3), (4, 3)]
>>> ip3 = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(1))])
>>> ip1.intersection(ip3)
Traceback (most recent call last):
...
ValueError: this intersection is empty, it does not correspond to an interval-
↪poset
>>> ip4 = TamariIntervalPoset(Integer(3),[(Integer(2),Integer(3))])
>>> ip2.intersection(ip4)
Traceback (most recent call last):
...
ValueError: intersections are only possible on interval-posets of the same␣
↪size
```

interval_cardinality()

Return the cardinality of the interval, i.e., the number of elements (binary trees or Dyck words) in the interval represented by `self`.

Not to be confused with *size()* which is the number of vertices.

See also

binary_trees()

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)]).interval_cardinality()
4
sage: TamariIntervalPoset(4,[]).interval_cardinality()
14
sage: TamariIntervalPoset(4,[(1,2),(2,3),(3,4)]).interval_cardinality()
1
```

```
>>> from sage.all import *
>>> TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))]).interval_
↪cardinality()
4
>>> TamariIntervalPoset(Integer(4),[]).interval_cardinality()
14
>>> TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(3),Integer(4))]).interval_cardinality()
1
```

is_connected()

Return whether `self` is a connected Tamari interval.

This means that the Hasse diagram is connected.

This condition is invariant under complementation.

See also

is_indecomposable(), *factor()*

EXAMPLES:

```
sage: len([T for T in TamariIntervalPosets(3) if T.is_connected()])
8
```

```
>>> from sage.all import *
>>> len([T for T in TamariIntervalPosets(Integer(3)) if T.is_connected()])
8
```

is_dexter()

Return whether `self` is a dexter Tamari interval.

This is defined by an exclusion pattern in the Hasse diagram. See the code for the exact description.

This condition is not invariant under complementation.

EXAMPLES:

```
sage: len([T for T in TamariIntervalPosets(3) if T.is_dexter()])
12
```

```
>>> from sage.all import *
>>> len([T for T in TamariIntervalPosets(Integer(3)) if T.is_dexter()])
12
```

is_exceptional()

Return whether `self` is an exceptional Tamari interval.

This is defined by exclusion of a simple pattern in the Hasse diagram, namely there is no configuration `y <-- x --> z` with $1 \leq y < x < z \leq n$.

This condition is invariant under complementation.

EXAMPLES:

```
sage: len([T for T in TamariIntervalPosets(3)
....:     if T.is_exceptional()])
12
```

```
>>> from sage.all import *
>>> len([T for T in TamariIntervalPosets(Integer(3))
...     if T.is_exceptional()])
12
```

is_final_interval()

Return if `self` corresponds to a final interval of the Tamari lattice.

This means that its upper end is the largest element of the lattice. It consists of checking that `self` does not contain any increasing relations.

See also

is_initial_interval()

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4, [(4, 3), (3, 1), (2, 1)])
sage: ip.is_final_interval()
True
sage: ip.upper_dyck_word()                                                  #␣
↪needs sage.combinat
[1, 1, 1, 1, 0, 0, 0, 0]
sage: ip = TamariIntervalPoset(4, [(4, 3), (3, 1), (2, 1), (2, 3)])
sage: ip.is_final_interval()
False
sage: ip.upper_dyck_word()                                                  #␣
↪needs sage.combinat
[1, 1, 0, 1, 1, 0, 0, 0]
sage: all(dw.tamari_interval(DyckWord([1, 1, 1, 0, 0, 0]))                  #␣
↪needs sage.combinat
....:           .is_final_interval()
....:     for dw in DyckWords(3))
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(4), Integer(3)),␣
↪(Integer(3), Integer(1)), (Integer(2), Integer(1))])
>>> ip.is_final_interval()
True
>>> ip.upper_dyck_word()                                                  #␣
↪needs sage.combinat
[1, 1, 1, 1, 0, 0, 0, 0]
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(4), Integer(3)),␣
↪(Integer(3), Integer(1)), (Integer(2), Integer(1)), (Integer(2),␣
↪Integer(3))])
>>> ip.is_final_interval()
False
```

(continues on next page)

(continued from previous page)

```
>>> ip.upper_dyck_word()                                                    #␣
↪needs sage.combinat
[1, 1, 0, 1, 1, 0, 0, 0]
>>> all(dw.tamari_interval(DyckWord([Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(0), Integer(0)]))                    # needs sage.combinat
...            .is_final_interval()
...     for dw in DyckWords(Integer(3)))
True
```

is_indecomposable()

Return whether `self` is an indecomposable Tamari interval.

This is the terminology of [Cha2008].

This means that the upper binary tree has an empty left subtree.

This condition is not invariant under complementation.

See also

is_connected()

EXAMPLES:

```
sage: len([T for T in TamariIntervalPosets(3)
....:      if T.is_indecomposable()])
8
```

```
>>> from sage.all import *
>>> len([T for T in TamariIntervalPosets(Integer(3))
...      if T.is_indecomposable()])
8
```

is_infinitely_modern()

Return whether `self` is an infinitely-modern Tamari interval.

This is defined by the exclusion of the configuration $i \rightarrow i+1$ and $j+1 \rightarrow j$ with $i < j$.

This condition is invariant under complementation.

See also

is_new(), *is_modern()*

EXAMPLES:

```
sage: len([T for T in TamariIntervalPosets(3)
....:     if T.is_infinitely_modern()])
12
```

```
>>> from sage.all import *
>>> len([T for T in TamariIntervalPosets(Integer(3))
```

(continues on next page)

(continued from previous page)

```
...     if T.is_infinitely_modern()])
12
```

REFERENCES:

- [Rog2018]

is_initial_interval()

Return if `self` corresponds to an initial interval of the Tamari lattice.

This means that its lower end is the smallest element of the lattice. It consists of checking that `self` does not contain any decreasing relations.

See also:

is_final_interval()

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4, [(1, 2), (2, 4), (3, 4)])
sage: ip.is_initial_interval()
True
sage: ip.lower_dyck_word()                                              #␣
↪needs sage.combinat
[1, 0, 1, 0, 1, 0, 1, 0]
sage: ip = TamariIntervalPoset(4, [(1, 2), (2, 4), (3, 4), (3, 2)])
sage: ip.is_initial_interval()
False
sage: ip.lower_dyck_word()                                              #␣
↪needs sage.combinat
[1, 0, 1, 1, 0, 0, 1, 0]
sage: all(DyckWord([1,0,1,0,1,0]).tamari_interval(dw)                  #␣
↪needs sage.combinat
....:         .is_initial_interval()
....:     for dw in DyckWords(3))
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(1), Integer(2)),␣
↪(Integer(2), Integer(4)), (Integer(3), Integer(4))])
>>> ip.is_initial_interval()
True
>>> ip.lower_dyck_word()                                                #␣
↪needs sage.combinat
[1, 0, 1, 0, 1, 0, 1, 0]
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(1), Integer(2)),␣
↪(Integer(2), Integer(4)), (Integer(3), Integer(4)), (Integer(3),␣
↪Integer(2))])
>>> ip.is_initial_interval()
False
>>> ip.lower_dyck_word()                                                #␣
↪needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
[1, 0, 1, 1, 0, 0, 1, 0]
>>> all(DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)]).tamari_interval(dw)                          # needs sage.combinat
...         .is_initial_interval()
...     for dw in DyckWords(Integer(3)))
True
```

is_linear_extension(*perm*)

Return whether the permutation `perm` is a linear extension of `self`.

INPUT:

- `perm` – a permutation of the size of `self`

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip.is_linear_extension([1,4,2,3])
True
sage: ip.is_linear_extension(Permutation([1,4,2,3]))
True
sage: ip.is_linear_extension(Permutation([1,4,3,2]))
False
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(4),Integer(3))])
>>> ip.is_linear_extension([Integer(1),Integer(4),Integer(2),Integer(3)])
True
>>> ip.is_linear_extension(Permutation([Integer(1),Integer(4),Integer(2),
↪Integer(3)]))
True
>>> ip.is_linear_extension(Permutation([Integer(1),Integer(4),Integer(3),
↪Integer(2)]))
False
```

is_modern()

Return whether `self` is a modern Tamari interval.

This is defined by exclusion of a simple pattern in the Hasse diagram, namely there is no configuration $y \to x \leftarrow z$ with $1 \leq y < x < z \leq n$.

This condition is invariant under complementation.

See also

is_new(), *is_infinitely_modern()*

EXAMPLES:

```
sage: len([T for T in TamariIntervalPosets(3) if T.is_modern()])
12
```

```
>>> from sage.all import *
>>> len([T for T in TamariIntervalPosets(Integer(3)) if T.is_modern()])
12
```

REFERENCES:

- [Rog2018]

is_new()

Return whether `self` is a new Tamari interval.

Here 'new' means that the interval is not contained in any facet of the associahedron. This condition is invariant under complementation.

They have been considered in section 9 of [Cha2008].

See also

is_modern()

EXAMPLES:

```
sage: TIP4 = TamariIntervalPosets(4)
sage: len([u for u in TIP4 if u.is_new()])
12

sage: TIP3 = TamariIntervalPosets(3)
sage: len([u for u in TIP3 if u.is_new()])
3
```

```
>>> from sage.all import *
>>> TIP4 = TamariIntervalPosets(Integer(4))
>>> len([u for u in TIP4 if u.is_new()])
12

>>> TIP3 = TamariIntervalPosets(Integer(3))
>>> len([u for u in TIP3 if u.is_new()])
3
```

is_simple()

Return whether `self` is a simple Tamari interval.

Here 'simple' means that the interval contains a unique binary tree.

These intervals define the simple modules over the incidence algebras of the Tamari lattices.

See also

is_final_interval(), *is_initial_interval()*

EXAMPLES:

```
sage: TIP4 = TamariIntervalPosets(4)
sage: len([u for u in TIP4 if u.is_simple()])
```

(continues on next page)

(continued from previous page)

```
14

sage: TIP3 = TamariIntervalPosets(3)
sage: len([u for u in TIP3 if u.is_simple()])
5
```

```
>>> from sage.all import *
>>> TIP4 = TamariIntervalPosets(Integer(4))
>>> len([u for u in TIP4 if u.is_simple()])
14

>>> TIP3 = TamariIntervalPosets(Integer(3))
>>> len([u for u in TIP3 if u.is_simple()])
5
```

is_synchronized()

Return whether `self` is a synchronized Tamari interval.

This means that the upper and lower binary trees have the same canopee. This condition is invariant under complementation.

This has been considered in [FPR2015]. The numbers of synchronized intervals are given by the sequence OEIS sequence A000139.

EXAMPLES:

```
sage: len([T for T in TamariIntervalPosets(3)
....:      if T.is_synchronized()])
6
```

```
>>> from sage.all import *
>>> len([T for T in TamariIntervalPosets(Integer(3))
...      if T.is_synchronized()])
6
```

latex_options()

Return the latex options for use in the `_latex_` function as a dictionary.

The default values are set using the options.

- `tikz_scale` – (default: 1) scale for use with the tikz package
- `line_width` – (default: 1) value representing the line width (additionally scaled by `tikz_scale`)
- `color_decreasing` – (default: `'red'`) the color for decreasing relations
- `color_increasing` – (default: `'blue'`) the color for increasing relations
- `hspace` – (default: 1) the difference between horizontal coordinates of adjacent vertices
- `vspace` – (default: 1) the difference between vertical coordinates of adjacent vertices

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.latex_options()['color_decreasing']
'red'
```

(continues on next page)

(continued from previous page)

```
sage: ip.latex_options()['hspace']
1
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> ip.latex_options()['color_decreasing']
'red'
>>> ip.latex_options()['hspace']
1
```

le(*e1*, *e2*)

Return whether `e1` precedes or equals `e2` in `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.le(1,2)
True
sage: ip.le(1,3)
True
sage: ip.le(2,3)
True
sage: ip.le(3,4)
False
sage: ip.le(1,1)
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
>>> ip.le(Integer(1),Integer(2))
True
>>> ip.le(Integer(1),Integer(3))
True
>>> ip.le(Integer(2),Integer(3))
True
>>> ip.le(Integer(3),Integer(4))
False
>>> ip.le(Integer(1),Integer(1))
True
```

left_branch_involution()

Return the image of `self` by the left-branch involution.

OUTPUT: an interval-poset

See also

rise_contact_involution()

EXAMPLES:

```
sage: tip = TamariIntervalPoset(8, [(1,2), (2,4), (3,4), (6,7), (3,2), (5,4),
↪(6,4), (8,7)])
sage: t = tip.left_branch_involution(); t
The Tamari interval of size 8 induced by relations [(1, 6), (2, 6),
(3, 5), (4, 5), (5, 6), (6, 8), (7, 8), (7, 6), (4, 3), (3, 1),
(2, 1)]
sage: t.left_branch_involution() == tip
True
```

```
>>> from sage.all import *
>>> tip = TamariIntervalPoset(Integer(8), [(Integer(1),Integer(2)),
↪(Integer(2),Integer(4)), (Integer(3),Integer(4)), (Integer(6),Integer(7)),
↪(Integer(3),Integer(2)), (Integer(5),Integer(4)), (Integer(6),Integer(4)),
↪(Integer(8),Integer(7))])
>>> t = tip.left_branch_involution(); t
The Tamari interval of size 8 induced by relations [(1, 6), (2, 6),
(3, 5), (4, 5), (5, 6), (6, 8), (7, 8), (7, 6), (4, 3), (3, 1),
(2, 1)]
>>> t.left_branch_involution() == tip
True
```

REFERENCES:

- [Pons2018]

linear_extensions()

Return an iterator on the permutations which are linear extensions of `self`.

They form an interval of the right weak order (also called the right permutohedron order – see *permutohedron_lequal()* for a definition).

EXAMPLES:

```
sage: ip = TamariIntervalPoset(3, [(1,2),(3,2)])
sage: list(ip.linear_extensions())                                       #
↪needs sage.modules sage.rings.finite_rings
[[3, 1, 2], [1, 3, 2]]
sage: ip = TamariIntervalPoset(4, [(1,2),(2,3),(4,3)])
sage: list(ip.linear_extensions())                                       #
↪needs sage.modules sage.rings.finite_rings
[[4, 1, 2, 3], [1, 2, 4, 3], [1, 4, 2, 3]]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(3), [(Integer(1),Integer(2)),(Integer(3),
↪Integer(2))])
>>> list(ip.linear_extensions())                                       #
↪needs sage.modules sage.rings.finite_rings
[[3, 1, 2], [1, 3, 2]]
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(4),Integer(3))])
>>> list(ip.linear_extensions())                                       #
↪needs sage.modules sage.rings.finite_rings
[[4, 1, 2, 3], [1, 2, 4, 3], [1, 4, 2, 3]]
```

lower_binary_tree()

Return the lowest binary tree in the interval of the Tamari lattice represented by `self`.

This is a binary tree. It is the shape of the unique binary search tree whose left-branch ordered forest (i.e., the result of applying *to_ordered_tree_left_branch()* and cutting off the root) is the final forest of `self`.

See also:

lower_dyck_word()

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6, [(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.lower_binary_tree()
[[., .], [[., [., .]], [., .]]]
sage: ff = TamariIntervalPosets.final_forest(ip.lower_binary_tree())
sage: ff == ip.final_forest()
True
sage: ip == TamariIntervalPosets.from_binary_trees(ip.lower_binary_tree(),
....:                                              ip.upper_binary_tree())
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6), [(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.lower_binary_tree()
[[., .], [[., [., .]], [., .]]]
>>> ff = TamariIntervalPosets.final_forest(ip.lower_binary_tree())
>>> ff == ip.final_forest()
True
>>> ip == TamariIntervalPosets.from_binary_trees(ip.lower_binary_tree(),
...                                              ip.upper_binary_tree())
True
```

lower_contained_intervals()

If `self` represents the interval $[t_1, t_2]$ of the Tamari lattice, return an iterator on all intervals $[t_1, t]$ with $t \leq t_2$ for the Tamari lattice.

In terms of interval-posets, it corresponds to adding all possible relations of the form n precedes m with $n < m$.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4, [(2,4),(3,4),(2,1),(3,1)])
sage: list(ip.lower_contained_intervals())
[The Tamari interval of size 4 induced by relations
  [(2, 4), (3, 4), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations
```

(continues on next page)

(continued from previous page)

```
 [(1, 4), (2, 4), (3, 4), (3, 1), (2, 1)],
The Tamari interval of size 4 induced by relations
 [(2, 3), (3, 4), (3, 1), (2, 1)],
The Tamari interval of size 4 induced by relations
 [(1, 4), (2, 3), (3, 4), (3, 1), (2, 1)]]
sage: ip = TamariIntervalPoset(4,[])
sage: len(list(ip.lower_contained_intervals()))
14
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> list(ip.lower_contained_intervals())
[The Tamari interval of size 4 induced by relations
 [(2, 4), (3, 4), (3, 1), (2, 1)],
The Tamari interval of size 4 induced by relations
 [(1, 4), (2, 4), (3, 4), (3, 1), (2, 1)],
The Tamari interval of size 4 induced by relations
 [(2, 3), (3, 4), (3, 1), (2, 1)],
The Tamari interval of size 4 induced by relations
 [(1, 4), (2, 3), (3, 4), (3, 1), (2, 1)]]
>>> ip = TamariIntervalPoset(Integer(4),[])
>>> len(list(ip.lower_contained_intervals()))
14
```

lower_contains_interval(*other*)

Return whether the interval represented by `other` is contained in `self` as an interval of the Tamari lattice and if they share the same lower bound.

As interval-posets, it means that `other` contains the relations of `self` plus some extra increasing relations.

INPUT:

- `other` – an interval-poset

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip2 = TamariIntervalPoset(4,[(4,3)])
sage: ip2.lower_contains_interval(ip1)
True
sage: ip2.contains_interval(ip1) and ip2.lower_binary_tree() == ip1.lower_
↪binary_tree()
True
sage: ip3 = TamariIntervalPoset(4,[(4,3),(2,1)])
sage: ip2.contains_interval(ip3)
True
sage: ip2.lower_binary_tree() == ip3.lower_binary_tree()
False
sage: ip2.lower_contains_interval(ip3)
False
```

```
>>> from sage.all import *
>>> ip1 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(4),Integer(3))])
>>> ip2 = TamariIntervalPoset(Integer(4),[(Integer(4),Integer(3))])
>>> ip2.lower_contains_interval(ip1)
True
>>> ip2.contains_interval(ip1) and ip2.lower_binary_tree() == ip1.lower_
↪binary_tree()
True
>>> ip3 = TamariIntervalPoset(Integer(4),[(Integer(4),Integer(3)),(Integer(2),
↪Integer(1))])
>>> ip2.contains_interval(ip3)
True
>>> ip2.lower_binary_tree() == ip3.lower_binary_tree()
False
>>> ip2.lower_contains_interval(ip3)
False
```

lower_dyck_word()

Return the lowest Dyck word in the interval of the Tamari lattice represented by `self`.

See also

lower_binary_tree()

EXAMPLES:

```
sage: # needs sage.combinat
sage: ip = TamariIntervalPoset(6, [(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.lower_dyck_word()
[1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0]
sage: ldw_ff = TamariIntervalPosets.final_forest(ip.lower_dyck_word())
sage: ldw_ff == ip.final_forest()
True
sage: ip == TamariIntervalPosets.from_dyck_words(ip.lower_dyck_word(),
....:                                             ip.upper_dyck_word())
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> ip = TamariIntervalPoset(Integer(6), [(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.lower_dyck_word()
[1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0]
>>> ldw_ff = TamariIntervalPosets.final_forest(ip.lower_dyck_word())
>>> ldw_ff == ip.final_forest()
True
```

(continues on next page)

(continued from previous page)

```
>>> ip == TamariIntervalPosets.from_dyck_words(ip.lower_dyck_word(),
...                                              ip.upper_dyck_word())
True
```

lt(*e1*, *e2*)

Return whether `e1` strictly precedes `e2` in `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.lt(1,2)
True
sage: ip.lt(1,3)
True
sage: ip.lt(2,3)
True
sage: ip.lt(3,4)
False
sage: ip.lt(1,1)
False
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
>>> ip.lt(Integer(1),Integer(2))
True
>>> ip.lt(Integer(1),Integer(3))
True
>>> ip.lt(Integer(2),Integer(3))
True
>>> ip.lt(Integer(3),Integer(4))
False
>>> ip.lt(Integer(1),Integer(1))
False
```

max_linear_extension()

Return the maximal permutation for the right weak order which is a linear extension of `self`.

This is also the maximal permutation in the sylvester class of `self.upper_binary_tree()` and is a 132-avoiding permutation.

The right weak order is also known as the right permutohedron order. See `permutohedron_lequal()` for its definition.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip.max_linear_extension()
[4, 1, 2, 3]
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (4, 5), (6, 5),␣
↪(5, 2), (4, 3), (3, 2)]
sage: ip.max_linear_extension()
[6, 4, 5, 3, 1, 2]
```

(continues on next page)

(continued from previous page)

```
sage: ip = TamariIntervalPoset(0,[]); ip
The Tamari interval of size 0 induced by relations []
sage: ip.max_linear_extension()
[]
sage: ip = TamariIntervalPoset(5, [(1, 4), (2, 4), (3, 4), (5, 4)]); ip
The Tamari interval of size 5 induced by relations [(1, 4), (2, 4), (3, 4),␣
↪(5, 4)]
sage: ip.max_linear_extension()
[5, 3, 2, 1, 4]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(4),Integer(3))])
>>> ip.max_linear_extension()
[4, 1, 2, 3]
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (4, 5), (6, 5),␣
↪(5, 2), (4, 3), (3, 2)]
>>> ip.max_linear_extension()
[6, 4, 5, 3, 1, 2]
>>> ip = TamariIntervalPoset(Integer(0),[]); ip
The Tamari interval of size 0 induced by relations []
>>> ip.max_linear_extension()
[]
>>> ip = TamariIntervalPoset(Integer(5), [(Integer(1), Integer(4)),␣
↪(Integer(2), Integer(4)), (Integer(3), Integer(4)), (Integer(5),␣
↪Integer(4))]); ip
The Tamari interval of size 5 induced by relations [(1, 4), (2, 4), (3, 4),␣
↪(5, 4)]
>>> ip.max_linear_extension()
[5, 3, 2, 1, 4]
```

maximal_chain_binary_trees()

Return an iterator on the binary trees forming a longest chain of `self` (regarding `self` as an interval of the Tamari lattice).

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: list(ip.maximal_chain_binary_trees())
[[[., [[., .], .]], .], [., [[[., .], .], .]], [., [[., [., .]], .]]]
sage: ip = TamariIntervalPoset(4,[])
sage: list(ip.maximal_chain_binary_trees())
[[[[[., .], .], .], .],
 [[[., [., .]], .], .],
 [[., [[., .], .]], .],
 [., [[[., .], .], .]],
 [., [[., [., .]], .]],
 [., [., [[., .], .]]],
 [., [., [., [., .]]]]]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> list(ip.maximal_chain_binary_trees())
[[[., [[., .], .]], .], [., [[[., .], .], .]], [., [[., [., .]], .]]]
>>> ip = TamariIntervalPoset(Integer(4),[])
>>> list(ip.maximal_chain_binary_trees())
[[[[[., .], .], .], .],
 [[[., [., .]], .], .],
 [[., [[., .], .]], .],
 [., [[[., .], .], .]],
 [., [[., [., .]], .]],
 [., [., [[., .], .]]],
 [., [., [., [., .]]]]]
```

maximal_chain_dyck_words()

Return an iterator on the Dyck words forming a longest chain of `self` (regarding `self` as an interval of the Tamari lattice).

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: list(ip.maximal_chain_dyck_words())                              #␣
↪needs sage.combinat
[[1, 1, 0, 1, 0, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0, 0], [1, 1, 1, 0, 0, 1, 0, 0]]
sage: ip = TamariIntervalPoset(4,[])
sage: list(ip.maximal_chain_dyck_words())                              #␣
↪needs sage.combinat
[[1, 0, 1, 0, 1, 0, 1, 0],
 [1, 1, 0, 0, 1, 0, 1, 0],
 [1, 1, 0, 1, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 1, 0, 0, 0],
 [1, 1, 1, 1, 0, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> list(ip.maximal_chain_dyck_words())                              #␣
↪needs sage.combinat
[[1, 1, 0, 1, 0, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0, 0], [1, 1, 1, 0, 0, 1, 0, 0]]
>>> ip = TamariIntervalPoset(Integer(4),[])
>>> list(ip.maximal_chain_dyck_words())                              #␣
↪needs sage.combinat
[[1, 0, 1, 0, 1, 0, 1, 0],
 [1, 1, 0, 0, 1, 0, 1, 0],
 [1, 1, 0, 1, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 1, 0, 0, 0],
 [1, 1, 1, 1, 0, 0, 0, 0]]
```

maximal_chain_tamari_intervals()

Return an iterator on the upper contained intervals of one longest chain of the Tamari interval represented by `self`.

If `self` represents the interval $[T_1, T_2]$ of the Tamari lattice, this returns intervals $[T', T_2]$ with T' following one longest chain between T_1 and T_2.

To obtain a longest chain, we use the Tamari inversions of `self`. The elements of the chain are obtained by adding one by one the relations (b, a) from each Tamari inversion (a, b) to `self`, where the Tamari inversions are taken in lexicographic order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4, [(2,4),(3,4),(2,1),(3,1)])
sage: list(ip.maximal_chain_tamari_intervals())
[The Tamari interval of size 4 induced by relations
  [(2, 4), (3, 4), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations
  [(2, 4), (3, 4), (4, 1), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations
  [(2, 4), (3, 4), (4, 1), (3, 2), (2, 1)]]
sage: ip = TamariIntervalPoset(4, [])
sage: list(ip.maximal_chain_tamari_intervals())
[The Tamari interval of size 4 induced by relations [],
 The Tamari interval of size 4 induced by relations [(2, 1)],
 The Tamari interval of size 4 induced by relations [(3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 1), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 1), (3, 2), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 2), (3, 2), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 3), (3, 2), (2, 1)]]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4), [(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> list(ip.maximal_chain_tamari_intervals())
[The Tamari interval of size 4 induced by relations
  [(2, 4), (3, 4), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations
  [(2, 4), (3, 4), (4, 1), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations
  [(2, 4), (3, 4), (4, 1), (3, 2), (2, 1)]]
>>> ip = TamariIntervalPoset(Integer(4), [])
>>> list(ip.maximal_chain_tamari_intervals())
[The Tamari interval of size 4 induced by relations [],
 The Tamari interval of size 4 induced by relations [(2, 1)],
 The Tamari interval of size 4 induced by relations [(3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 1), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 1), (3, 2), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 2), (3, 2), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 3), (3, 2), (2, 1)]]
```

min_linear_extension()

Return the minimal permutation for the right weak order which is a linear extension of `self`.

This is also the minimal permutation in the sylvester class of `self.lower_binary_tree()` and is a 312-avoiding permutation.

The right weak order is also known as the right permutohedron order. See `permutohedron_lequal()` for its definition.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip.min_linear_extension()
[1, 2, 4, 3]
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)])
sage: ip.min_linear_extension()
[1, 4, 3, 6, 5, 2]
sage: ip = TamariIntervalPoset(0,[])
sage: ip.min_linear_extension()
[]
sage: ip = TamariIntervalPoset(5, [(1, 4), (2, 4), (3, 4), (5, 4)]); ip
The Tamari interval of size 5 induced by relations [(1, 4), (2, 4), (3, 4),␣
↪(5, 4)]
sage: ip.min_linear_extension()
[1, 2, 3, 5, 4]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(4),Integer(3))])
>>> ip.min_linear_extension()
[1, 2, 4, 3]
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(4),Integer(5))])
>>> ip.min_linear_extension()
[1, 4, 3, 6, 5, 2]
>>> ip = TamariIntervalPoset(Integer(0),[])
>>> ip.min_linear_extension()
[]
>>> ip = TamariIntervalPoset(Integer(5), [(Integer(1), Integer(4)),␣
↪(Integer(2), Integer(4)), (Integer(3), Integer(4)), (Integer(5),␣
↪Integer(4))]); ip
The Tamari interval of size 5 induced by relations [(1, 4), (2, 4), (3, 4),␣
↪(5, 4)]
>>> ip.min_linear_extension()
[1, 2, 3, 5, 4]
```

new_decomposition()

Return the decomposition of the interval-poset into new interval-posets.

Every interval-poset has a unique decomposition as a planar tree of new interval-posets, as explained in [Cha2008]. This function computes the terms of this decomposition, but not the planar tree.

For the number of terms, you can use instead the method `number_of_new_components()`.

OUTPUT: list of new interval-posets

See also

`number_of_new_components()`, `is_new()`

EXAMPLES:

```
sage: ex = TamariIntervalPosets(4)[11]
sage: ex.number_of_new_components()
3
sage: ex.new_decomposition()                                              #␣
↪needs sage.combinat
[The Tamari interval of size 1 induced by relations [],
 The Tamari interval of size 2 induced by relations [],
 The Tamari interval of size 1 induced by relations []]
```

```
>>> from sage.all import *
>>> ex = TamariIntervalPosets(Integer(4))[Integer(11)]
>>> ex.number_of_new_components()
3
>>> ex.new_decomposition()                                              #␣
↪needs sage.combinat
[The Tamari interval of size 1 induced by relations [],
 The Tamari interval of size 2 induced by relations [],
 The Tamari interval of size 1 induced by relations []]
```

number_of_new_components()

Return the number of terms in the decomposition in new interval-posets.

Every interval-poset has a unique decomposition as a planar tree of new interval-posets, as explained in [Cha2008]. This function just computes the number of terms, not the planar tree nor the terms themselves.

See also:

is_new(), *new_decomposition()*

EXAMPLES:

```
sage: TIP4 = TamariIntervalPosets(4)
sage: nb = [u.number_of_new_components() for u in TIP4]
sage: [nb.count(i) for i in range(1, 5)]
[12, 21, 21, 14]
```

```
>>> from sage.all import *
>>> TIP4 = TamariIntervalPosets(Integer(4))
>>> nb = [u.number_of_new_components() for u in TIP4]
>>> [nb.count(i) for i in range(Integer(1), Integer(5))]
[12, 21, 21, 14]
```

number_of_tamari_inversions()

Return the number of Tamari inversions of `self`.

This is also the length the longest chain of the Tamari interval represented by `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.number_of_tamari_inversions()
```

(continues on next page)

(continued from previous page)

```
2
sage: ip = TamariIntervalPoset(4,[])
sage: ip.number_of_tamari_inversions()
6
sage: ip = TamariIntervalPoset(3,[])
sage: ip.number_of_tamari_inversions()
3
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> ip.number_of_tamari_inversions()
2
>>> ip = TamariIntervalPoset(Integer(4),[])
>>> ip.number_of_tamari_inversions()
6
>>> ip = TamariIntervalPoset(Integer(3),[])
>>> ip.number_of_tamari_inversions()
3
```

plot(***kwds*)

Return a picture.

The picture represents the Hasse diagram, where the covers are colored in blue if they are increasing and in red if they are decreasing.

This uses the same coordinates as the latex view.

EXAMPLES:

```
sage: ti = TamariIntervalPosets(4)[2]
sage: ti.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 6 graphics primitives
```

```
>>> from sage.all import *
>>> ti = TamariIntervalPosets(Integer(4))[Integer(2)]
>>> ti.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 6 graphics primitives
```

poset()

Return `self` as a labelled poset.

An interval-poset is indeed constructed from a labelled poset which is stored internally. This method allows to access the poset and all the associated methods.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(3,2),(2,4),(3,4)])
sage: pos = ip.poset(); pos
Finite poset containing 4 elements
sage: pos.maximal_chains()
[[3, 2, 4], [1, 2, 4]]
```

(continues on next page)

(continued from previous page)

```
sage: pos.maximal_elements()
[4]
sage: pos.is_lattice()
False
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(3),
↪Integer(2)),(Integer(2),Integer(4)),(Integer(3),Integer(4))])
>>> pos = ip.poset(); pos
Finite poset containing 4 elements
>>> pos.maximal_chains()
[[3, 2, 4], [1, 2, 4]]
>>> pos.maximal_elements()
[4]
>>> pos.is_lattice()
False
```

rise_contact_involution()

Return the image of `self` by the rise-contact involution.

OUTPUT: an interval-poset

This is defined by conjugating the complement involution by the left-branch involution.

See also

left_branch_involution(), *complement()*

EXAMPLES:

```
sage: tip = TamariIntervalPoset(8, [(1,2), (2,4), (3,4), (6,7),
....:                               (3,2), (5,4), (6,4), (8,7)])
sage: t = tip.rise_contact_involution(); t
The Tamari interval of size 8 induced by relations [(2, 8), (3, 8),
(4, 5), (5, 7), (6, 7), (7, 8), (8, 1), (7, 2), (6, 2), (5, 3),
(4, 3), (3, 2), (2, 1)]
sage: t.rise_contact_involution() == tip
True
sage: (tip.lower_dyck_word().number_of_touch_points()                    #␣
↪needs sage.combinat
....:     == t.upper_dyck_word().number_of_initial_rises())
True
sage: tip.number_of_tamari_inversions() == t.number_of_tamari_inversions()
True
```

```
>>> from sage.all import *
>>> tip = TamariIntervalPoset(Integer(8), [(Integer(1),Integer(2)),␣
↪(Integer(2),Integer(4)), (Integer(3),Integer(4)), (Integer(6),Integer(7)),
...                               (Integer(3),Integer(2)), (Integer(5),
↪Integer(4)), (Integer(6),Integer(4)), (Integer(8),Integer(7))])
>>> t = tip.rise_contact_involution(); t
The Tamari interval of size 8 induced by relations [(2, 8), (3, 8),
```

(continues on next page)

(continued from previous page)

```
(4, 5), (5, 7), (6, 7), (7, 8), (8, 1), (7, 2), (6, 2), (5, 3),
(4, 3), (3, 2), (2, 1)]
>>> t.rise_contact_involution() == tip
True
>>> (tip.lower_dyck_word().number_of_touch_points()                    #␣
↪needs sage.combinat
...      == t.upper_dyck_word().number_of_initial_rises())
True
>>> tip.number_of_tamari_inversions() == t.number_of_tamari_inversions()
True
```

REFERENCES:

- [Pons2018]

set_latex_options(*D*)

Set the latex options for use in the `_latex_` function.

The default values are set in the `__init__` function.

- `tikz_scale` – (default: 1) scale for use with the tikz package
- `line_width` – (default: 1 * `tikz_scale`) value representing the line width
- `color_decreasing` – (default: red) the color for decreasing relations
- `color_increasing` – (default: blue) the color for increasing relations
- `hspace` – (default: 1) the difference between horizontal coordinates of adjacent vertices
- `vspace` – (default: 1) the difference between vertical coordinates of adjacent vertices

INPUT:

- `D` – dictionary with a list of latex parameters to change

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.latex_options()["color_decreasing"]
'red'
sage: ip.set_latex_options({"color_decreasing":'green'})
sage: ip.latex_options()["color_decreasing"]
'green'
sage: ip.set_latex_options({"color_increasing":'black'})
sage: ip.latex_options()["color_increasing"]
'black'
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> ip.latex_options()["color_decreasing"]
'red'
>>> ip.set_latex_options({"color_decreasing":'green'})
>>> ip.latex_options()["color_decreasing"]
'green'
>>> ip.set_latex_options({"color_increasing":'black'})
```

(continues on next page)

(continued from previous page)

```
>>> ip.latex_options()["color_increasing"]
'black'
```

To change the default options for all interval-posets, use the parent's latex options:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip2 = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.latex_options()["color_decreasing"]
'red'
sage: ip2.latex_options()["color_decreasing"]
'red'
sage: TamariIntervalPosets.options(latex_color_decreasing='green')
sage: ip.latex_options()["color_decreasing"]
'green'
sage: ip2.latex_options()["color_decreasing"]
'green'
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> ip2 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
>>> ip.latex_options()["color_decreasing"]
'red'
>>> ip2.latex_options()["color_decreasing"]
'red'
>>> TamariIntervalPosets.options(latex_color_decreasing='green')
>>> ip.latex_options()["color_decreasing"]
'green'
>>> ip2.latex_options()["color_decreasing"]
'green'
```

Next we set a local latex option and show the global option does not override it:

```
sage: ip.set_latex_options({"color_decreasing": 'black'})
sage: ip.latex_options()["color_decreasing"]
'black'
sage: TamariIntervalPosets.options(latex_color_decreasing='blue')
sage: ip.latex_options()["color_decreasing"]
'black'
sage: ip2.latex_options()["color_decreasing"]
'blue'
sage: TamariIntervalPosets.options._reset()
```

```
>>> from sage.all import *
>>> ip.set_latex_options({"color_decreasing": 'black'})
>>> ip.latex_options()["color_decreasing"]
'black'
>>> TamariIntervalPosets.options(latex_color_decreasing='blue')
>>> ip.latex_options()["color_decreasing"]
'black'
>>> ip2.latex_options()["color_decreasing"]
```

(continues on next page)

(continued from previous page)

```
'blue'
>>> TamariIntervalPosets.options._reset()
```

size()

Return the size (number of vertices) of the interval-poset.

EXAMPLES:

```
sage: TamariIntervalPoset(3,[(2,1),(3,1)]).size()
3
```

```
>>> from sage.all import *
>>> TamariIntervalPoset(Integer(3),[(Integer(2),Integer(1)),(Integer(3),
↪Integer(1))]).size()
3
```

sub_poset(*start*, *end*)

Return the renormalized subposet of `self` consisting solely of integers from `start` (inclusive) to `end` (not inclusive).

"Renormalized" means that these integers are relabelled $1, 2, \ldots, k$ in the obvious way (i.e., by subtracting `start - 1`).

INPUT:

- `start` – integer; the starting vertex (inclusive)
- `end` – integer; the ending vertex (not inclusive)

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6, [(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,
↪5)]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.subposet(1,3)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: ip.subposet(1,4)
The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)]
sage: ip.subposet(1,5)
The Tamari interval of size 4 induced by relations [(1, 2), (4, 3), (3, 2)]
sage: ip.subposet(1,7) == ip
True
sage: ip.subposet(1,1)
The Tamari interval of size 0 induced by relations []
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6), [(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(3),Integer(5)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.subposet(Integer(1),Integer(3))
The Tamari interval of size 2 induced by relations [(1, 2)]
```

(continues on next page)

(continued from previous page)

```
>>> ip.subposet(Integer(1),Integer(4))
The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)]
>>> ip.subposet(Integer(1),Integer(5))
The Tamari interval of size 4 induced by relations [(1, 2), (4, 3), (3, 2)]
>>> ip.subposet(Integer(1),Integer(7)) == ip
True
>>> ip.subposet(Integer(1),Integer(1))
The Tamari interval of size 0 induced by relations []
```

subposet (*start*, *end*)

Return the renormalized subposet of `self` consisting solely of integers from `start` (inclusive) to `end` (not inclusive).

"Renormalized" means that these integers are relabelled $1, 2, \ldots, k$ in the obvious way (i.e., by subtracting `start - 1`).

INPUT:

- `start` – integer; the starting vertex (inclusive)
- `end` – integer; the ending vertex (not inclusive)

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6, [(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,
↪5)]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.subposet(1,3)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: ip.subposet(1,4)
The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)]
sage: ip.subposet(1,5)
The Tamari interval of size 4 induced by relations [(1, 2), (4, 3), (3, 2)]
sage: ip.subposet(1,7) == ip
True
sage: ip.subposet(1,1)
The Tamari interval of size 0 induced by relations []
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6), [(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(3),Integer(5)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.subposet(Integer(1),Integer(3))
The Tamari interval of size 2 induced by relations [(1, 2)]
>>> ip.subposet(Integer(1),Integer(4))
The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)]
>>> ip.subposet(Integer(1),Integer(5))
The Tamari interval of size 4 induced by relations [(1, 2), (4, 3), (3, 2)]
>>> ip.subposet(Integer(1),Integer(7)) == ip
True
>>> ip.subposet(Integer(1),Integer(1))
```

(continues on next page)

(continued from previous page)

```
The Tamari interval of size 0 induced by relations []
```

tamari_inversions()

Return the Tamari inversions of `self`.

A Tamari inversion is a pair of vertices (a, b) with $a < b$ such that:

- the decreasing parent of b is strictly smaller than a (or does not exist), and
- the increasing parent of a is strictly bigger than b (or does not exist).

"Smaller" and "bigger" refer to the numerical values of the elements, not to the poset order.

This method returns the list of all Tamari inversions in lexicographic order.

The number of Tamari inversions is the length of the longest chain of the Tamari interval represented by `self`.

Indeed, when an interval consists of just one binary tree, it has no inversion. One can also prove that if a Tamari interval $I' = [T_1', T_2']$ is a proper subset of a Tamari interval $I = [T_1, T_2]$, then the inversion number of I' is strictly lower than the inversion number of I. And finally, by adding the relation (b, a) to the interval-poset where (a, b) is the first inversion of I in lexicographic order, one reduces the inversion number by exactly 1.

See also

tamari_inversions_iter(), *number_of_tamari_inversions()*

EXAMPLES:

```
sage: ip = TamariIntervalPoset(3,[])
sage: ip.tamari_inversions()
[(1, 2), (1, 3), (2, 3)]
sage: ip = TamariIntervalPoset(3,[(2,1)])
sage: ip.tamari_inversions()
[(1, 3), (2, 3)]
sage: ip = TamariIntervalPoset(3,[(1,2)])
sage: ip.tamari_inversions()
[(2, 3)]
sage: ip = TamariIntervalPoset(3,[(1,2),(3,2)])
sage: ip.tamari_inversions()
[]
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.tamari_inversions()
[(1, 4), (2, 3)]
sage: ip = TamariIntervalPoset(4,[])
sage: ip.tamari_inversions()
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
sage: all(not TamariIntervalPosets.from_binary_trees(bt,bt)                  #␣
↪needs sage.combinat
....:                                   .tamari_inversions()
....:     for bt in BinaryTrees(3))
True
sage: all(not TamariIntervalPosets.from_binary_trees(bt,bt)                  #␣
↪needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
....:                                    .tamari_inversions()
....:     for bt in BinaryTrees(4))
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(3),[])
>>> ip.tamari_inversions()
[(1, 2), (1, 3), (2, 3)]
>>> ip = TamariIntervalPoset(Integer(3),[(Integer(2),Integer(1))])
>>> ip.tamari_inversions()
[(1, 3), (2, 3)]
>>> ip = TamariIntervalPoset(Integer(3),[(Integer(1),Integer(2))])
>>> ip.tamari_inversions()
[(2, 3)]
>>> ip = TamariIntervalPoset(Integer(3),[(Integer(1),Integer(2)),(Integer(3),
↪Integer(2))])
>>> ip.tamari_inversions()
[]
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(2),Integer(4)),(Integer(3),
↪Integer(4)),(Integer(2),Integer(1)),(Integer(3),Integer(1))])
>>> ip.tamari_inversions()
[(1, 4), (2, 3)]
>>> ip = TamariIntervalPoset(Integer(4),[])
>>> ip.tamari_inversions()
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
>>> all(not TamariIntervalPosets.from_binary_trees(bt,bt)                  #␣
↪needs sage.combinat
...                                      .tamari_inversions()
...     for bt in BinaryTrees(Integer(3)))
True
>>> all(not TamariIntervalPosets.from_binary_trees(bt,bt)                  #␣
↪needs sage.combinat
...                                      .tamari_inversions()
...     for bt in BinaryTrees(Integer(4)))
True
```

tamari_inversions_iter()

Iterate over the Tamari inversions of `self`, in lexicographic order.

See *tamari_inversions()* for the definition of the terms involved.

EXAMPLES:

```
sage: T = TamariIntervalPoset(5, [[1,2],[3,4],[3,2],[5,2],[4,2]])
sage: list(T.tamari_inversions_iter())
[(4, 5)]

sage: T = TamariIntervalPoset(8, [(2, 7), (3, 7), (4, 7), (5, 7), (6, 7),
....:                             (8, 7), (6, 4), (5, 4), (4, 3), (3, 2)])
sage: list(T.tamari_inversions_iter())
[(1, 2), (1, 7), (5, 6)]

sage: T = TamariIntervalPoset(1, [])
```

(continues on next page)

(continued from previous page)

```
sage: list(T.tamari_inversions_iter())
[]

sage: T = TamariIntervalPoset(0, [])
sage: list(T.tamari_inversions_iter())
[]
```

```
>>> from sage.all import *
>>> T = TamariIntervalPoset(Integer(5), [[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)],[Integer(3),Integer(2)],[Integer(5),Integer(2)],[Integer(4),
↪Integer(2)]])
>>> list(T.tamari_inversions_iter())
[(4, 5)]

>>> T = TamariIntervalPoset(Integer(8), [(Integer(2), Integer(7)),␣
↪(Integer(3), Integer(7)), (Integer(4), Integer(7)), (Integer(5),␣
↪Integer(7)), (Integer(6), Integer(7)),
...                                  (Integer(8), Integer(7)), (Integer(6),␣
↪Integer(4)), (Integer(5), Integer(4)), (Integer(4), Integer(3)),␣
↪(Integer(3), Integer(2))])
>>> list(T.tamari_inversions_iter())
[(1, 2), (1, 7), (5, 6)]

>>> T = TamariIntervalPoset(Integer(1), [])
>>> list(T.tamari_inversions_iter())
[]

>>> T = TamariIntervalPoset(Integer(0), [])
>>> list(T.tamari_inversions_iter())
[]
```

upper_binary_tree()

Return the highest binary tree in the interval of the Tamari lattice represented by `self`.

This is a binary tree. It is the shape of the unique binary search tree whose right-branch ordered forest (i.e., the result of applying *to_ordered_tree_right_branch()* and cutting off the root) is the initial forest of `self`.

See also

upper_dyck_word()

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.upper_binary_tree()
[[., .], [., [[., .], [., .]]]]

sage: TamariIntervalPosets.initial_forest(ip.upper_binary_tree()) == ip.
```

(continues on next page)

(continued from previous page)

```
↪initial_forest()
True
sage: ip == TamariIntervalPosets.from_binary_trees(ip.lower_binary_tree(),ip.
↪upper_binary_tree())
True
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.upper_binary_tree()
[[., .], [., [[., .], [., .]]]]

>>> TamariIntervalPosets.initial_forest(ip.upper_binary_tree()) == ip.initial_
↪forest()
True
>>> ip == TamariIntervalPosets.from_binary_trees(ip.lower_binary_tree(),ip.
↪upper_binary_tree())
True
```

upper_contains_interval(*other*)

Return whether the interval represented by `other` is contained in `self` as an interval of the Tamari lattice and if they share the same upper bound.

As interval-posets, it means that `other` contains the relations of `self` plus some extra decreasing relations.

INPUT:

- `other` – an interval-poset

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip2 = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip2.upper_contains_interval(ip1)
True
sage: ip2.contains_interval(ip1) and ip2.upper_binary_tree() == ip1.upper_
↪binary_tree()
True
sage: ip3 = TamariIntervalPoset(4,[(1,2),(2,3),(3,4)])
sage: ip2.upper_contains_interval(ip3)
False
sage: ip2.contains_interval(ip3)
True
sage: ip2.upper_binary_tree() == ip3.upper_binary_tree()
False
```

```
>>> from sage.all import *
>>> ip1 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(4),Integer(3))])
>>> ip2 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3))])
>>> ip2.upper_contains_interval(ip1)
True
>>> ip2.contains_interval(ip1) and ip2.upper_binary_tree() == ip1.upper_
↪binary_tree()
True
>>> ip3 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(3),Integer(4))])
>>> ip2.upper_contains_interval(ip3)
False
>>> ip2.contains_interval(ip3)
True
>>> ip2.upper_binary_tree() == ip3.upper_binary_tree()
False
```

upper_dyck_word()

Return the highest Dyck word in the interval of the Tamari lattice represented by `self`.

See also

upper_binary_tree()

EXAMPLES:

```
sage: # needs sage.combinat
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
sage: ip.upper_dyck_word()
[1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0]
sage: udw_if = TamariIntervalPosets.initial_forest(ip.upper_dyck_word())
sage: udw_if == ip.initial_forest()
True
sage: ip == TamariIntervalPosets.from_dyck_words(ip.lower_dyck_word(),
....:                                             ip.upper_dyck_word())
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> ip = TamariIntervalPoset(Integer(6),[(Integer(3),Integer(2)),(Integer(4),
↪Integer(3)),(Integer(5),Integer(2)),(Integer(6),Integer(5)),(Integer(1),
↪Integer(2)),(Integer(4),Integer(5))]); ip
The Tamari interval of size 6 induced by relations
 [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
>>> ip.upper_dyck_word()
[1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0]
>>> udw_if = TamariIntervalPosets.initial_forest(ip.upper_dyck_word())
>>> udw_if == ip.initial_forest()
True
>>> ip == TamariIntervalPosets.from_dyck_words(ip.lower_dyck_word(),
...                                             ip.upper_dyck_word())
```

(continues on next page)

(continued from previous page)

```
True
```

class sage.combinat.interval_posets.**TamariIntervalPosets**

Bases: UniqueRepresentation, Parent

Factory for interval-posets.

INPUT:

- size – integer (optional)

OUTPUT:

- the set of all interval-posets (of the given size if specified)

EXAMPLES:

```
sage: TamariIntervalPosets()
Interval-posets

sage: TamariIntervalPosets(2)
Interval-posets of size 2
```

```
>>> from sage.all import *
>>> TamariIntervalPosets()
Interval-posets

>>> TamariIntervalPosets(Integer(2))
Interval-posets of size 2
```

Note

This is a factory class whose constructor returns instances of subclasses.

static check_poset(*poset*)

Check if the given poset poset is a interval-poset, that is, if it satisfies the following properties:

- Its labels are exactly $1, \ldots, n$ where n is its size.
- If $a < c$ (as numbers) and a precedes c, then b precedes c for all b such that $a < b < c$.
- If $a < c$ (as numbers) and c precedes a, then b precedes a for all b such that $a < b < c$.

INPUT:

- poset – a finite labeled poset

EXAMPLES:

```
sage: p = Poset(([1,2,3],[(1,2),(3,2)]))
sage: TamariIntervalPosets.check_poset(p)
True
sage: p = Poset(([2,3],[(3,2)]))
sage: TamariIntervalPosets.check_poset(p)
False
sage: p = Poset(([1,2,3],[(3,1)]))
```

(continues on next page)

(continued from previous page)

```
sage: TamariIntervalPosets.check_poset(p)
False
sage: p = Poset(([1,2,3],[(1,3)]))
sage: TamariIntervalPosets.check_poset(p)
False
```

```
>>> from sage.all import *
>>> p = Poset(([Integer(1),Integer(2),Integer(3)],[(Integer(1),Integer(2)),
↪(Integer(3),Integer(2))]))
>>> TamariIntervalPosets.check_poset(p)
True
>>> p = Poset(([Integer(2),Integer(3)],[(Integer(3),Integer(2))]))
>>> TamariIntervalPosets.check_poset(p)
False
>>> p = Poset(([Integer(1),Integer(2),Integer(3)],[(Integer(3),Integer(1))]))
>>> TamariIntervalPosets.check_poset(p)
False
>>> p = Poset(([Integer(1),Integer(2),Integer(3)],[(Integer(1),Integer(3))]))
>>> TamariIntervalPosets.check_poset(p)
False
```

static final_forest(*element*)

Return the final forest of a binary tree, an interval-poset or a Dyck word.

A final forest is an interval-poset corresponding to a final interval of the Tamari lattice, i.e., containing only decreasing relations.

It can be constructed from a binary tree by its binary search tree labeling with the rule: b precedes a in the final forest iff b is in the right subtree of a in the binary search tree.

INPUT:

- `element` – a binary tree, a Dyck word or an interval-poset

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: TamariIntervalPosets.final_forest(ip)
The Tamari interval of size 4 induced by relations [(4, 3)]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(4),Integer(3))])
>>> TamariIntervalPosets.final_forest(ip)
The Tamari interval of size 4 induced by relations [(4, 3)]
```

From binary trees:

```
sage: bt = BinaryTree(); bt
.
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 0 induced by relations []
sage: bt = BinaryTree([]); bt
[., .]
```

(continues on next page)

(continued from previous page)

```
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 1 induced by relations []
sage: bt = BinaryTree([[],None]); bt
[[., .], .]
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 2 induced by relations []
sage: bt = BinaryTree([None,[]]); bt
[., [., .]]
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 2 induced by relations [(2, 1)]
sage: bt = BinaryTree([[],[]]); bt
[[., .], [., .]]
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 3 induced by relations [(3, 2)]
sage: bt = BinaryTree([[None,[[],None]],[]]); bt
[[., [[., .], .]], [., .]]
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 5 induced by relations [(5, 4), (3, 1), (2, 1)]
```

```
>>> from sage.all import *
>>> bt = BinaryTree(); bt
.
>>> TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 0 induced by relations []
>>> bt = BinaryTree([]); bt
[., .]
>>> TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 1 induced by relations []
>>> bt = BinaryTree([[],None]); bt
[[., .], .]
>>> TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 2 induced by relations []
>>> bt = BinaryTree([None,[]]); bt
[., [., .]]
>>> TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 2 induced by relations [(2, 1)]
>>> bt = BinaryTree([[],[]]); bt
[[., .], [., .]]
>>> TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 3 induced by relations [(3, 2)]
>>> bt = BinaryTree([[None,[[],None]],[]]); bt
[[., [[., .], .]], [., .]]
>>> TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 5 induced by relations [(5, 4), (3, 1), (2, 1)]
```

From Dyck words:

```
sage: # needs sage.combinat
sage: dw = DyckWord([1,0])
sage: TamariIntervalPosets.final_forest(dw)
The Tamari interval of size 1 induced by relations []
sage: dw = DyckWord([1,1,0,1,0,0,1,1,0,0])
```

(continues on next page)

(continued from previous page)

```
sage: TamariIntervalPosets.final_forest(dw)
The Tamari interval of size 5 induced by relations [(5, 4), (3, 1), (2, 1)]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> dw = DyckWord([Integer(1),Integer(0)])
>>> TamariIntervalPosets.final_forest(dw)
The Tamari interval of size 1 induced by relations []
>>> dw = DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(0)])
>>> TamariIntervalPosets.final_forest(dw)
The Tamari interval of size 5 induced by relations [(5, 4), (3, 1), (2, 1)]
```

static from_binary_trees(*tree1*, *tree2*)

Return the interval-poset corresponding to the interval [`tree1`, `tree2`] of the Tamari lattice.

Raise an exception if `tree1` is not $\leq$ `tree2` in the Tamari lattice.

INPUT:

- `tree1` – a binary tree
- `tree2` – a binary tree greater or equal than `tree1` for the Tamari lattice

EXAMPLES:

```
sage: tree1 = BinaryTree([[],None])
sage: tree2 = BinaryTree([None,[]])
sage: TamariIntervalPosets.from_binary_trees(tree1,tree2)
The Tamari interval of size 2 induced by relations []
sage: TamariIntervalPosets.from_binary_trees(tree1,tree1)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: TamariIntervalPosets.from_binary_trees(tree2,tree2)
The Tamari interval of size 2 induced by relations [(2, 1)]

sage: tree1 = BinaryTree([[],[[None,[]],[]]])
sage: tree2 = BinaryTree([None,[None,[None,[[],[]]]]])
sage: TamariIntervalPosets.from_binary_trees(tree1,tree2)
The Tamari interval of size 6 induced by relations
 [(4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]

sage: tree3 = BinaryTree([None,[None,[[],[None,[]]]]])
sage: TamariIntervalPosets.from_binary_trees(tree1,tree3)
Traceback (most recent call last):
...
ValueError: the two binary trees are not comparable on the Tamari lattice
sage: TamariIntervalPosets.from_binary_trees(tree1,BinaryTree())
Traceback (most recent call last):
...
ValueError: the two binary trees are not comparable on the Tamari lattice
```

```
>>> from sage.all import *
>>> tree1 = BinaryTree([[],None])
>>> tree2 = BinaryTree([None,[]])
```

(continues on next page)

(continued from previous page)

```
>>> TamariIntervalPosets.from_binary_trees(tree1,tree2)
The Tamari interval of size 2 induced by relations []
>>> TamariIntervalPosets.from_binary_trees(tree1,tree1)
The Tamari interval of size 2 induced by relations [(1, 2)]
>>> TamariIntervalPosets.from_binary_trees(tree2,tree2)
The Tamari interval of size 2 induced by relations [(2, 1)]

>>> tree1 = BinaryTree([[],[[None,[]],[]]])
>>> tree2 = BinaryTree([None,[None,[None,[[],[]]]]])
>>> TamariIntervalPosets.from_binary_trees(tree1,tree2)
The Tamari interval of size 6 induced by relations
 [(4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]

>>> tree3 = BinaryTree([None,[None,[[],[None,[]]]]])
>>> TamariIntervalPosets.from_binary_trees(tree1,tree3)
Traceback (most recent call last):
...
ValueError: the two binary trees are not comparable on the Tamari lattice
>>> TamariIntervalPosets.from_binary_trees(tree1,BinaryTree())
Traceback (most recent call last):
...
ValueError: the two binary trees are not comparable on the Tamari lattice
```

static from_dyck_words(*dw1*, *dw2*)

Return the interval-poset corresponding to the interval [`dw1`, `dw2`] of the Tamari lattice.

Raise an exception if the two Dyck words `dw1` and `dw2` do not satisfy `dw1` $\leq$ `dw2` in the Tamari lattice.

INPUT:

- `dw1` – a Dyck word
- `dw2` – a Dyck word greater or equal than `dw1` for the Tamari lattice

EXAMPLES:

```
sage: # needs sage.combinat
sage: dw1 = DyckWord([1,0,1,0])
sage: dw2 = DyckWord([1,1,0,0])
sage: TamariIntervalPosets.from_dyck_words(dw1, dw2)
The Tamari interval of size 2 induced by relations []
sage: TamariIntervalPosets.from_dyck_words(dw1,dw1)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: TamariIntervalPosets.from_dyck_words(dw2,dw2)
The Tamari interval of size 2 induced by relations [(2, 1)]

sage: # needs sage.combinat
sage: dw1 = DyckWord([1,0,1,1,1,0,0,1,1,0,0,0])
sage: dw2 = DyckWord([1,1,1,1,0,1,1,0,0,0,0,0])
sage: TamariIntervalPosets.from_dyck_words(dw1,dw2)
The Tamari interval of size 6 induced by relations
 [(4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
sage: dw3 = DyckWord([1,1,1,0,1,1,1,0,0,0,0,0])
sage: TamariIntervalPosets.from_dyck_words(dw1,dw3)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: the two Dyck words are not comparable on the Tamari lattice
sage: TamariIntervalPosets.from_dyck_words(dw1,DyckWord([1,0]))
Traceback (most recent call last):
...
ValueError: the two Dyck words are not comparable on the Tamari lattice
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> dw1 = DyckWord([Integer(1),Integer(0),Integer(1),Integer(0)])
>>> dw2 = DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)])
>>> TamariIntervalPosets.from_dyck_words(dw1, dw2)
The Tamari interval of size 2 induced by relations []
>>> TamariIntervalPosets.from_dyck_words(dw1,dw1)
The Tamari interval of size 2 induced by relations [(1, 2)]
>>> TamariIntervalPosets.from_dyck_words(dw2,dw2)
The Tamari interval of size 2 induced by relations [(2, 1)]

>>> # needs sage.combinat
>>> dw1 = DyckWord([Integer(1),Integer(0),Integer(1),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0)])
>>> dw2 = DyckWord([Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0)])
>>> TamariIntervalPosets.from_dyck_words(dw1,dw2)
The Tamari interval of size 6 induced by relations
 [(4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]
>>> dw3 = DyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0)])
>>> TamariIntervalPosets.from_dyck_words(dw1,dw3)
Traceback (most recent call last):
...
ValueError: the two Dyck words are not comparable on the Tamari lattice
>>> TamariIntervalPosets.from_dyck_words(dw1,DyckWord([Integer(1),
↪Integer(0)]))
Traceback (most recent call last):
...
ValueError: the two Dyck words are not comparable on the Tamari lattice
```

static from_grafting_tree(*tree*)

Return an interval-poset from a grafting tree.

For the inverse method, see *TamariIntervalPoset.grafting_tree()*.

EXAMPLES:

```
sage: tip = TamariIntervalPoset(8, [(1,2), (2,4), (3,4), (6,7), (3,2), (5,4),␣
↪(6,4), (8,7)])
sage: t = tip.grafting_tree()
sage: TamariIntervalPosets.from_grafting_tree(t) == tip
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> tip = TamariIntervalPoset(Integer(8), [(Integer(1),Integer(2)),
↪(Integer(2),Integer(4)), (Integer(3),Integer(4)), (Integer(6),Integer(7)),
↪(Integer(3),Integer(2)), (Integer(5),Integer(4)), (Integer(6),Integer(4)),
↪(Integer(8),Integer(7))])
>>> t = tip.grafting_tree()
>>> TamariIntervalPosets.from_grafting_tree(t) == tip
True
```

REFERENCES:

- [Pons2018]

static from_minimal_schnyder_wood(*graph*)

Return a Tamari interval built from a minimal Schnyder wood.

This is an implementation of Bernardi and Bonichon's bijection [BeBo2009].

INPUT:

- `graph` – a minimal Schnyder wood, given as a graph with colored and oriented edges, without the three exterior unoriented edges

The three boundary vertices must be -1, -2 and -3.

One assumes moreover that the embedding around -1 is the list of neighbors of -1 and not just a cyclic permutation of that.

Beware that the embedding convention used here is the opposite of the one used by the plot method.

OUTPUT: a Tamari interval-poset

EXAMPLES:

A small example:

```
sage: TIP = TamariIntervalPosets
sage: G = DiGraph([(0,-1,0),(0,-2,1),(0,-3,2)], format='list_of_edges')
sage: G.set_embedding({-1:[0],-2:[0],-3:[0],0:[-1,-2,-3]})
sage: TIP.from_minimal_schnyder_wood(G)                                    #
↪needs sage.combinat
The Tamari interval of size 1 induced by relations []
```

```
>>> from sage.all import *
>>> TIP = TamariIntervalPosets
>>> G = DiGraph([(Integer(0),-Integer(1),Integer(0)),(Integer(0),-Integer(2),
↪Integer(1)),(Integer(0),-Integer(3),Integer(2))], format='list_of_edges')
>>> G.set_embedding({-Integer(1):[Integer(0)],-Integer(2):[Integer(0)],-
↪Integer(3):[Integer(0)],Integer(0):[-Integer(1),-Integer(2),-Integer(3)]})
>>> TIP.from_minimal_schnyder_wood(G)                                     #
↪needs sage.combinat
The Tamari interval of size 1 induced by relations []
```

An example from page 14 of [BeBo2009]:

```
sage: c0 = [(0,-1),(1,0),(2,0),(4,3),(3,-1),(5,3)]
sage: c1 = [(5,-2),(3,-2),(4,5),(1,3),(2,3),(0,3)]
sage: c2 = [(0,-3),(1,-3),(3,-3),(4,-3),(5,-3),(2,1)]
sage: ed = [(u,v,0) for u,v in c0]
sage: ed += [(u,v,1) for u,v in c1]
sage: ed += [(u,v,2) for u,v in c2]
sage: G = DiGraph(ed, format='list_of_edges')
sage: embed = {-1:[3,0],-2:[5,3],-3:[0,1,3,4,5]}
sage: data_emb = [[3,2,1,-3,-1],[2,3,-3,0],[3,1,0]]
sage: data_emb += [[-2,5,4,-3,1,2,0,-1],[5,-3,3],[-2,-3,4,3]]
sage: for k in range(6):
....:     embed[k] = data_emb[k]
sage: G.set_embedding(embed)
sage: TIP.from_minimal_schnyder_wood(G)                                         #␣
↪needs sage.combinat
The Tamari interval of size 6 induced by relations
 [(1, 4), (2, 4), (3, 4), (5, 6), (6, 4), (5, 4), (3, 1), (2, 1)]
```

```
>>> from sage.all import *
>>> c0 = [(Integer(0),-Integer(1)),(Integer(1),Integer(0)),(Integer(2),
↪Integer(0)),(Integer(4),Integer(3)),(Integer(3),-Integer(1)),(Integer(5),
↪Integer(3))]
>>> c1 = [(Integer(5),-Integer(2)),(Integer(3),-Integer(2)),(Integer(4),
↪Integer(5)),(Integer(1),Integer(3)),(Integer(2),Integer(3)),(Integer(0),
↪Integer(3))]
>>> c2 = [(Integer(0),-Integer(3)),(Integer(1),-Integer(3)),(Integer(3),-
↪Integer(3)),(Integer(4),-Integer(3)),(Integer(5),-Integer(3)),(Integer(2),
↪Integer(1))]
>>> ed = [(u,v,Integer(0)) for u,v in c0]
>>> ed += [(u,v,Integer(1)) for u,v in c1]
>>> ed += [(u,v,Integer(2)) for u,v in c2]
>>> G = DiGraph(ed, format='list_of_edges')
>>> embed = {-Integer(1):[Integer(3),Integer(0)],-Integer(2):[Integer(5),
↪Integer(3)],-Integer(3):[Integer(0),Integer(1),Integer(3),Integer(4),
↪Integer(5)]}
>>> data_emb = [[Integer(3),Integer(2),Integer(1),-Integer(3),-Integer(1)],
↪[Integer(2),Integer(3),-Integer(3),Integer(0)],[Integer(3),Integer(1),
↪Integer(0)]]
>>> data_emb += [[-Integer(2),Integer(5),Integer(4),-Integer(3),Integer(1),
↪Integer(2),Integer(0),-Integer(1)],[Integer(5),-Integer(3),Integer(3)],[-
↪Integer(2),-Integer(3),Integer(4),Integer(3)]]
>>> for k in range(Integer(6)):
...     embed[k] = data_emb[k]
>>> G.set_embedding(embed)
>>> TIP.from_minimal_schnyder_wood(G)                                         #␣
↪needs sage.combinat
The Tamari interval of size 6 induced by relations
 [(1, 4), (2, 4), (3, 4), (5, 6), (6, 4), (5, 4), (3, 1), (2, 1)]
```

An example from page 18 of [BeBo2009]:

```
sage: c0 = [(0,-1),(1,0),(2,-1),(3,2),(4,2),(5,-1)]
sage: c1 = [(5,-2),(2,-2),(4,-2),(3,4),(1,2),(0,2)]
```

(continues on next page)

(continued from previous page)

```
sage: c2 = [(0,-3),(1,-3),(3,-3),(4,-3),(2,-3),(5,2)]
sage: ed = [(u,v,0) for u,v in c0]
sage: ed += [(u,v,1) for u,v in c1]
sage: ed += [(u,v,2) for u,v in c2]
sage: G = DiGraph(ed, format='list_of_edges')
sage: embed = {-1:[5,2,0],-2:[4,2,5],-3:[0,1,2,3,4]}
sage: data_emb = [[2,1,-3,-1],[2,-3,0],[3,-3,1,0,-1,5,-2,4]]
sage: data_emb += [[4,-3,2],[-2,-3,3,2],[-2,2,-1]]
sage: for k in range(6):
....:     embed[k] = data_emb[k]
sage: G.set_embedding(embed)
sage: TIP.from_minimal_schnyder_wood(G)                                      #␣
↪needs sage.combinat
The Tamari interval of size 6 induced by relations
 [(1, 3), (2, 3), (4, 5), (5, 3), (4, 3), (2, 1)]
```

```
>>> from sage.all import *
>>> c0 = [(Integer(0),-Integer(1)),(Integer(1),Integer(0)),(Integer(2),-
↪Integer(1)),(Integer(3),Integer(2)),(Integer(4),Integer(2)),(Integer(5),-
↪Integer(1))]
>>> c1 = [(Integer(5),-Integer(2)),(Integer(2),-Integer(2)),(Integer(4),-
↪Integer(2)),(Integer(3),Integer(4)),(Integer(1),Integer(2)),(Integer(0),
↪Integer(2))]
>>> c2 = [(Integer(0),-Integer(3)),(Integer(1),-Integer(3)),(Integer(3),-
↪Integer(3)),(Integer(4),-Integer(3)),(Integer(2),-Integer(3)),(Integer(5),
↪Integer(2))]
>>> ed = [(u,v,Integer(0)) for u,v in c0]
>>> ed += [(u,v,Integer(1)) for u,v in c1]
>>> ed += [(u,v,Integer(2)) for u,v in c2]
>>> G = DiGraph(ed, format='list_of_edges')
>>> embed = {-Integer(1):[Integer(5),Integer(2),Integer(0)],-
↪Integer(2):[Integer(4),Integer(2),Integer(5)],-Integer(3):[Integer(0),
↪Integer(1),Integer(2),Integer(3),Integer(4)]}
>>> data_emb = [[Integer(2),Integer(1),-Integer(3),-Integer(1)],[Integer(2),-
↪Integer(3),Integer(0)],[Integer(3),-Integer(3),Integer(1),Integer(0),-
↪Integer(1),Integer(5),-Integer(2),Integer(4)]]
>>> data_emb += [[Integer(4),-Integer(3),Integer(2)],[-Integer(2),-Integer(3),
↪Integer(3),Integer(2)],[-Integer(2),Integer(2),-Integer(1)]]
>>> for k in range(Integer(6)):
...     embed[k] = data_emb[k]
>>> G.set_embedding(embed)
>>> TIP.from_minimal_schnyder_wood(G)                                      #␣
↪needs sage.combinat
The Tamari interval of size 6 induced by relations
 [(1, 3), (2, 3), (4, 5), (5, 3), (4, 3), (2, 1)]
```

Another small example:

```
sage: c0 = [(0,-1),(2,-1),(1,0)]
sage: c1 = [(2,-2),(1,-2),(0,2)]
sage: c2 = [(0,-3),(1,-3),(2,1)]
sage: ed = [(u,v,0) for u,v in c0]
```

(continues on next page)

(continued from previous page)

```
sage: ed += [(u,v,1) for u,v in c1]
sage: ed += [(u,v,2) for u,v in c2]
sage: G = DiGraph(ed, format='list_of_edges')
sage: embed = {-1:[2,0],-2:[1,2],-3:[0,1]}
sage: data_emb = [[2,1,-3,-1],[-3,0,2,-2],[-2,1,0,-1]]
sage: for k in range(3):
....:     embed[k] = data_emb[k]
sage: G.set_embedding(embed)
sage: TIP.from_minimal_schnyder_wood(G)                                  #␣
↪needs sage.combinat
The Tamari interval of size 3 induced by relations [(2, 3), (2, 1)]
```

```
>>> from sage.all import *
>>> c0 = [(Integer(0),-Integer(1)),(Integer(2),-Integer(1)),(Integer(1),
↪Integer(0))]
>>> c1 = [(Integer(2),-Integer(2)),(Integer(1),-Integer(2)),(Integer(0),
↪Integer(2))]
>>> c2 = [(Integer(0),-Integer(3)),(Integer(1),-Integer(3)),(Integer(2),
↪Integer(1))]
>>> ed = [(u,v,Integer(0)) for u,v in c0]
>>> ed += [(u,v,Integer(1)) for u,v in c1]
>>> ed += [(u,v,Integer(2)) for u,v in c2]
>>> G = DiGraph(ed, format='list_of_edges')
>>> embed = {-Integer(1):[Integer(2),Integer(0)],-Integer(2):[Integer(1),
↪Integer(2)],-Integer(3):[Integer(0),Integer(1)]}
>>> data_emb = [[Integer(2),Integer(1),-Integer(3),-Integer(1)],[-Integer(3),
↪Integer(0),Integer(2),-Integer(2)],[-Integer(2),Integer(1),Integer(0),-
↪Integer(1)]]
>>> for k in range(Integer(3)):
...     embed[k] = data_emb[k]
>>> G.set_embedding(embed)
>>> TIP.from_minimal_schnyder_wood(G)                                  #␣
↪needs sage.combinat
The Tamari interval of size 3 induced by relations [(2, 3), (2, 1)]
```

static initial_forest(*element*)

Return the initial forest of a binary tree, an interval-poset or a Dyck word.

An initial forest is an interval-poset corresponding to an initial interval of the Tamari lattice, i.e., containing only increasing relations.

It can be constructed from a binary tree by its binary search tree labeling with the rule: a precedes b in the initial forest iff a is in the left subtree of b in the binary search tree.

INPUT:

- `element` – a binary tree, a Dyck word or an interval-poset

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: TamariIntervalPosets.initial_forest(ip)
The Tamari interval of size 4 induced by relations [(1, 2), (2, 3)]
```

```
>>> from sage.all import *
>>> ip = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
→Integer(3)),(Integer(4),Integer(3))])
>>> TamariIntervalPosets.initial_forest(ip)
The Tamari interval of size 4 induced by relations [(1, 2), (2, 3)]
```

with binary trees:

```
sage: bt = BinaryTree(); bt
.
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 0 induced by relations []
sage: bt = BinaryTree([]); bt
[., .]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 1 induced by relations []
sage: bt = BinaryTree([[],None]); bt
[[., .], .]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: bt = BinaryTree([None,[]]); bt
[., [., .]]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 2 induced by relations []
sage: bt = BinaryTree([[],[]]); bt
[[., .], [., .]]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 3 induced by relations [(1, 2)]
sage: bt = BinaryTree([[None,[[],None]],[]]); bt
[[., [[., .], .]], [., .]]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 5 induced by relations [(1, 4), (2, 3), (3, 4)]
```

```
>>> from sage.all import *
>>> bt = BinaryTree(); bt
.
>>> TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 0 induced by relations []
>>> bt = BinaryTree([]); bt
[., .]
>>> TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 1 induced by relations []
>>> bt = BinaryTree([[],None]); bt
[[., .], .]
>>> TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 2 induced by relations [(1, 2)]
>>> bt = BinaryTree([None,[]]); bt
[., [., .]]
>>> TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 2 induced by relations []
>>> bt = BinaryTree([[],[]]); bt
[[., .], [., .]]
>>> TamariIntervalPosets.initial_forest(bt)
```

(continues on next page)

(continued from previous page)

```
The Tamari interval of size 3 induced by relations [(1, 2)]
>>> bt = BinaryTree([[None,[[],None]],[]]); bt
[[., [[., .], .]], [., .]]
>>> TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 5 induced by relations [(1, 4), (2, 3), (3, 4)]
```

from Dyck words:

```
sage: # needs sage.combinat
sage: dw = DyckWord([1,0])
sage: TamariIntervalPosets.initial_forest(dw)
The Tamari interval of size 1 induced by relations []
sage: dw = DyckWord([1,1,0,1,0,0,1,1,0,0])
sage: TamariIntervalPosets.initial_forest(dw)
The Tamari interval of size 5 induced by relations [(1, 4), (2, 3), (3, 4)]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> dw = DyckWord([Integer(1),Integer(0)])
>>> TamariIntervalPosets.initial_forest(dw)
The Tamari interval of size 1 induced by relations []
>>> dw = DyckWord([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(0),Integer(0)])
>>> TamariIntervalPosets.initial_forest(dw)
The Tamari interval of size 5 induced by relations [(1, 4), (2, 3), (3, 4)]
```

le(*el1*, *el2*)

Poset structure on the set of interval-posets.

The comparison is first by size, then using the cubical coordinates.

See also

cubical_coordinates()

INPUT:

- el1 – an interval-poset
- el2 – an interval-poset

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip2 = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: TamariIntervalPosets().le(ip1,ip2)
False
sage: TamariIntervalPosets().le(ip2,ip1)
True
```

```
>>> from sage.all import *
>>> ip1 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(4),Integer(3))])
```

(continues on next page)

(continued from previous page)

```
>>> ip2 = TamariIntervalPoset(Integer(4),[(Integer(1),Integer(2)),(Integer(2),
↪Integer(3))])
>>> TamariIntervalPosets().le(ip1,ip2)
False
>>> TamariIntervalPosets().le(ip2,ip1)
True
```

options = Current options for TamariIntervalPosets - latex_color_decreasing: red - latex_color_increasing: blue - latex_hspace: 1 - latex_line_width_scalar: 0.5 - latex_tikz_scale: 1 - latex_vspace: 1

static recomposition_from_triple(*left*, *right*, *r*)

Recompose an interval-poset from a triple (`left`, `right`, `r`).

For the inverse method, see *TamariIntervalPoset.decomposition_to_triple()*.

INPUT:

- `left` – an interval-poset
- `right` – an interval-poset
- `r` – the parameter of the decomposition, an integer

OUTPUT: an interval-poset

EXAMPLES:

```
sage: T1 = TamariIntervalPoset(3, [(1, 2), (3, 2)])
sage: T2 = TamariIntervalPoset(4, [(2, 3), (4, 3)])
sage: TamariIntervalPosets.recomposition_from_triple(T1, T2, 2)
The Tamari interval of size 8 induced by relations [(1, 2), (2, 4),
(3, 4), (6, 7), (8, 7), (6, 4), (5, 4), (3, 2)]
```

```
>>> from sage.all import *
>>> T1 = TamariIntervalPoset(Integer(3), [(Integer(1), Integer(2)),
↪(Integer(3), Integer(2))])
>>> T2 = TamariIntervalPoset(Integer(4), [(Integer(2), Integer(3)),
↪(Integer(4), Integer(3))])
>>> TamariIntervalPosets.recomposition_from_triple(T1, T2, Integer(2))
The Tamari interval of size 8 induced by relations [(1, 2), (2, 4),
(3, 4), (6, 7), (8, 7), (6, 4), (5, 4), (3, 2)]
```

REFERENCES:

- [Pons2018]

class sage.combinat.interval_posets.**TamariIntervalPosets_all**

Bases: `DisjointUnionEnumeratedSets`, *TamariIntervalPosets*

The enumerated set of all Tamari interval-posets.

Element

alias of *TamariIntervalPoset*

one()

Return the unit of the monoid.

This is the empty interval poset, of size 0.

EXAMPLES:

```
sage: TamariIntervalPosets().one()
The Tamari interval of size 0 induced by relations []
```

```
>>> from sage.all import *
>>> TamariIntervalPosets().one()
The Tamari interval of size 0 induced by relations []
```

class sage.combinat.interval_posets.**TamariIntervalPosets_size**(*size*)

Bases: *TamariIntervalPosets*

The enumerated set of interval-posets of a given size.

cardinality()

The cardinality of `self`. That is, the number of interval-posets of size n.

The formula was given in [Cha2008]:

$$\frac{2(4n+1)!}{(n+1)!(3n+2)!} = \frac{2}{n(n+1)}\binom{4n+1}{n-1}.$$

EXAMPLES:

```
sage: [TamariIntervalPosets(i).cardinality() for i in range(6)]
[1, 1, 3, 13, 68, 399]
```

```
>>> from sage.all import *
>>> [TamariIntervalPosets(i).cardinality() for i in range(Integer(6))]
[1, 1, 3, 13, 68, 399]
```

element_class()

random_element()

Return a random Tamari interval of fixed size.

This is obtained by first creating a random rooted planar triangulation, then computing its unique minimal Schnyder wood, then applying a bijection of Bernardi and Bonichon [BeBo2009].

Because the random rooted planar triangulation is chosen uniformly at random, the Tamari interval is also chosen according to the uniform distribution.

EXAMPLES:

```
sage: # needs sage.combinat
sage: T = TamariIntervalPosets(4).random_element()
sage: T.parent()
Interval-posets
sage: u = T.lower_dyck_word(); u   # random
[1, 1, 0, 1, 0, 0, 1, 0]
sage: v = T.lower_dyck_word(); v   # random
[1, 1, 0, 1, 0, 0, 1, 0]
sage: len(u)
8
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> T = TamariIntervalPosets(Integer(4)).random_element()
>>> T.parent()
Interval-posets
>>> u = T.lower_dyck_word(); u    # random
[1, 1, 0, 1, 0, 0, 1, 0]
>>> v = T.lower_dyck_word(); v    # random
[1, 1, 0, 1, 0, 0, 1, 0]
>>> len(u)
8
```

5.1.130 Strong and weak tableaux

There are two types of k-tableaux: strong k-tableaux and weak k-tableaux. Standard weak k-tableaux correspond to saturated chains in the weak order, whereas standard strong k-tableaux correspond to saturated chains in the strong Bruhat order. For semistandard tableaux, the notion of weak and strong horizontal strip is necessary. More information can be found in [LLMS2006] .

See also

sage.combinat.k_tableau.StrongTableau(), *sage.combinat.k_tableau.WeakTableau()*

Authors:

- Anne Schilling and Mike Zabrocki (2013): initial version
- Avi Dalal and Nate Gallup (2013): implementation of k-charge

class sage.combinat.k_tableau.**StrongTableau**(*parent*, *T*)

Bases: ClonableList

A (standard) strong k-tableau is a (saturated) chain in Bruhat order.

Combinatorially, it is a sequence of embedded $k+1$-cores (subject to some conditions) together with a set of markings.

A strong cover in terms of cores corresponds to certain translated ribbons. A marking corresponds to the choice of one of the translated ribbons, which is indicated by marking the head (southeast most cell in French notation) of the chosen ribbon. For more information, see [LLMS2006] and [LLMSSZ2013].

In Sage, a strong k-tableau is created by specifying k, a standard strong tableau together with its markings, and a weight μ. Here the standard tableau is represented by a sequence of $k+1$-cores

$$\lambda^{(0)} \subseteq \lambda^{(1)} \subseteq \cdots \subseteq \lambda^{(m)}$$

where each of the $\lambda^{(i)}$ is a $k+1$-core. The standard tableau is a filling of the diagram for the core $\lambda^{(m)}/\lambda^{(0)}$ where a strong cover is represented by letters $\pm i$ in the skew shape $\lambda^{(i)}/\lambda^{(i-1)}$. Each skew $(k+1)$-core $\lambda^{(i)}/\lambda^{(i-1)}$ is a ribbon or multiple copies of the same ribbon which are separated by $k+1$ diagonals. Precisely one of the copies of the ribbons will be marked in the largest diagonal of the connected component (the 'head' of the ribbon). The marked cells are indicated by negative signs.

The strong tableau is stored as a standard strong marked tableau (referred to as the standard part of the strong tableau) and a vector representing the weight.

EXAMPLES:

```
sage: StrongTableau( [[-1, -2, -3], [3]], 2, [3] )
[[-1, -1, -1], [1]]
sage: StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1])
[[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4]]
```

```
>>> from sage.all import *
>>> StrongTableau( [[-Integer(1), -Integer(2), -Integer(3)], [Integer(3)]],␣
↪Integer(2), [Integer(3)] )
[[-1, -1, -1], [1]]
>>> StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-Integer(3),
↪Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-Integer(5),-
↪Integer(8)]], Integer(3), [Integer(2),Integer(2),Integer(3),Integer(1)])
[[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4]]
```

Alternatively, the strong k-tableau can also be entered directly in semistandard format and then the standard tableau and the weight are computed and stored:

```
sage: T = StrongTableau([[-1,-1,-1],[1]], 2); T
[[-1, -1, -1], [1]]
sage: T.to_standard_list()
[[-1, -2, -3], [3]]
sage: T.weight()
(3,)
sage: T = StrongTableau([[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4]], 3);␣
↪T
[[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4]]
sage: T.to_standard_list()
[[-1, -2, -4, -7], [-3, 6, -6, 8], [4, 7], [-5, -8]]
sage: T.weight()
(2, 2, 3, 1)
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1),-Integer(1),-Integer(1)],[Integer(1)]],␣
↪Integer(2)); T
[[-1, -1, -1], [1]]
>>> T.to_standard_list()
[[-1, -2, -3], [3]]
>>> T.weight()
(3,)
>>> T = StrongTableau([[-Integer(1), -Integer(1), -Integer(2), -Integer(3)], [-
↪Integer(2), Integer(3), -Integer(3), Integer(4)], [Integer(2), Integer(3)], [-
↪Integer(3), -Integer(4)]], Integer(3)); T
[[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4]]
>>> T.to_standard_list()
[[-1, -2, -4, -7], [-3, 6, -6, 8], [4, 7], [-5, -8]]
>>> T.weight()
(2, 2, 3, 1)
```

cell_of_highest_head(*v*)

Return the cell of the highest head of label `v` in the standard part of `self`.

Return the cell where the head of the ribbon in the highest row is located in the underlying standard tableau. If there is no cell with entry `v` then the cell returned is $(0, r)$ where r is the length of the first row.

This cell is calculated by iterating through the diagonals of the tableau.

INPUT:

- `v` – integer indicating the label in the standard tableau

OUTPUT: a pair of integers indicating the coordinates of the head of the highest ribbon with label `v`

EXAMPLES:

```
sage: T = StrongTableau([[-1,2,-3],[-2,3],[3]], 1)
sage: [T.cell_of_highest_head(v) for v in range(1,5)]
[(0, 0), (1, 0), (2, 0), (0, 3)]
sage: T = StrongTableau([[None,None,-3,4],[3,-4]],2)
sage: [T.cell_of_highest_head(v) for v in range(1,5)]
[(1, 0), (1, 1), (0, 4), (0, 4)]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1),Integer(2),-Integer(3)],[-Integer(2),
↪Integer(3)],[Integer(3)]], Integer(1))
>>> [T.cell_of_highest_head(v) for v in range(Integer(1),Integer(5))]
[(0, 0), (1, 0), (2, 0), (0, 3)]
>>> T = StrongTableau([[None,None,-Integer(3),Integer(4)],[Integer(3),-
↪Integer(4)]],Integer(2))
>>> [T.cell_of_highest_head(v) for v in range(Integer(1),Integer(5))]
[(1, 0), (1, 1), (0, 4), (0, 4)]
```

cell_of_marked_head(*v*)

Return location of marked head labeled by `v` in the standard part of `self`.

Return the coordinates of the `v`-th marked cell in the strong standard tableau `self`. If there is no mark, then the value returned is $(0, r)$ where r is the length of the first row.

INPUT:

- `v` – integer representing the label in the standard tableau

OUTPUT: a pair of the coordinates of the marked cell with entry `v`

EXAMPLES:

```
sage: T = StrongTableau([[-1, -3, 4, -5], [-2], [-4]], 3)
sage: [ T.cell_of_marked_head(i) for i in range(1,7)]
[(0, 0), (1, 0), (0, 1), (2, 0), (0, 3), (0, 4)]
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: [ T.cell_of_marked_head(i) for i in range(1,7)]
[(2, 0), (0, 2), (2, 1), (0, 3), (4, 0), (0, 4)]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1), -Integer(3), Integer(4), -Integer(5)], [-
↪Integer(2)], [-Integer(4)]], Integer(3))
>>> [ T.cell_of_marked_head(i) for i in range(Integer(1),Integer(7))]
[(0, 0), (1, 0), (0, 1), (2, 0), (0, 3), (0, 4)]
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],␣
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> [ T.cell_of_marked_head(i) for i in range(Integer(1),Integer(7))]
[(2, 0), (0, 2), (2, 1), (0, 3), (4, 0), (0, 4)]
```

cells_head_dictionary()

Return a dictionary with the locations of the heads of all markings.

Return a dictionary of values and lists of cells where the heads with the values are located.

OUTPUT:

- a dictionary with keys the entries in the tableau and values are the coordinates of the heads with those entries

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,7],[-3,6,-6,8],[4,-7],[-5,-8]], 3)
sage: T.cells_head_dictionary()
{1: [(0, 0)],
 2: [(0, 1)],
 3: [(1, 0)],
 4: [(2, 0), (0, 2)],
 5: [(3, 0)],
 6: [(1, 2)],
 7: [(2, 1), (0, 3)],
 8: [(3, 1), (1, 3)]}
sage: T = StrongTableau([[None, 4, -4, -6, -7, 8, 8, -8], [None, -5, 8, 8, 8],
↪ [-3, 6]],3)
sage: T.cells_head_dictionary()
{1: [(2, 0)],
 2: [(0, 2)],
 3: [(1, 1)],
 4: [(2, 1), (0, 3)],
 5: [(0, 4)],
 6: [(1, 4), (0, 7)]}
 sage: StrongTableau([[None, None], [None, -1]], 4).cells_head_dictionary()
 {1: [(1, 1)]}
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1),-Integer(2),-Integer(4),Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),-Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3))
>>> T.cells_head_dictionary()
{1: [(0, 0)],
 2: [(0, 1)],
 3: [(1, 0)],
 4: [(2, 0), (0, 2)],
 5: [(3, 0)],
 6: [(1, 2)],
 7: [(2, 1), (0, 3)],
 8: [(3, 1), (1, 3)]}
>>> T = StrongTableau([[None, Integer(4), -Integer(4), -Integer(6), -
↪Integer(7), Integer(8), Integer(8), -Integer(8)], [None, -Integer(5),␣
↪Integer(8), Integer(8), Integer(8)], [-Integer(3), Integer(6)]],Integer(3))
>>> T.cells_head_dictionary()
```

(continues on next page)

(continued from previous page)

```
{1: [(2, 0)],
 2: [(0, 2)],
 3: [(1, 1)],
 4: [(2, 1), (0, 3)],
 5: [(0, 4)],
 6: [(1, 4), (0, 7)]}
>>> StrongTableau([[None, None], [None, -Integer(1)]], Integer(4)).cells_head_
↪dictionary()
 {1: [(1, 1)]}
```

cells_of_heads(*v*)

Return a list of cells of the heads with label v in the standard part of self.

A list of cells which are heads of the ribbons with label v in the standard part of the tableau self. If there is no cell labelled by v then return the empty list.

INPUT:

- v – integer label

OUTPUT: a list of pairs of integers of the coordinates of the heads of the ribbons with label v

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: T.cells_of_heads(1)
[(2, 0)]
sage: T.cells_of_heads(2)
[(3, 0), (0, 2)]
sage: T.cells_of_heads(3)
[(2, 1)]
sage: T.cells_of_heads(4)
[(3, 1), (0, 3)]
sage: T.cells_of_heads(5)
[(4, 0)]
sage: T.cells_of_heads(6)
[]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
>>> T.cells_of_heads(Integer(1))
[(2, 0)]
>>> T.cells_of_heads(Integer(2))
[(3, 0), (0, 2)]
>>> T.cells_of_heads(Integer(3))
[(2, 1)]
>>> T.cells_of_heads(Integer(4))
[(3, 1), (0, 3)]
>>> T.cells_of_heads(Integer(5))
[(4, 0)]
>>> T.cells_of_heads(Integer(6))
```

(continues on next page)

(continued from previous page)

```
[]
```

cells_of_marked_ribbon(*v*)

Return a list of all cells the marked ribbon labeled by `v` in the standard part of `self`.

Return the list of coordinates of the cells which are in the marked ribbon with label `v` in the standard part of the tableau. Note that the result is independent of the weight of the tableau.

The cells are listed from largest content (where the mark is located) to the smallest. Hence, the first entry in this list will be the marked cell.

INPUT:

- `v` – the entry of the standard tableau

OUTPUT:

- a list of pairs representing the coordinates of the cells of the marked ribbon

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -2, -2, 3], [2, -3], [-3]],3)
sage: T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
sage: T.cells_of_marked_ribbon(1)
[(0, 0)]
sage: T.cells_of_marked_ribbon(4)
[(0, 3)]
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3)
sage: T.cells_of_marked_ribbon(6)
[(1, 2), (1, 1)]
sage: T.cells_of_marked_ribbon(9)
[]
sage: T = StrongTableau([[None, None, -1, -1, 3], [1, -3], [-3]],3)
sage: T.to_standard_list()
[[None, None, -1, -2, 4], [2, -4], [-3]]
sage: T.cells_of_marked_ribbon(1)
[(0, 2)]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1), -Integer(1), -Integer(2), -Integer(2),␣
↪Integer(3)], [Integer(2), -Integer(3)], [-Integer(3)]],Integer(3))
>>> T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
>>> T.cells_of_marked_ribbon(Integer(1))
[(0, 0)]
>>> T.cells_of_marked_ribbon(Integer(4))
[(0, 3)]
>>> T = StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3))
>>> T.cells_of_marked_ribbon(Integer(6))
[(1, 2), (1, 1)]
>>> T.cells_of_marked_ribbon(Integer(9))
[]
```

(continues on next page)

(continued from previous page)

```
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(1), Integer(3)],
↪[Integer(1), -Integer(3)], [-Integer(3)]],Integer(3))
>>> T.to_standard_list()
[[None, None, -1, -2, 4], [2, -4], [-3]]
>>> T.cells_of_marked_ribbon(Integer(1))
[(0, 2)]
```

check()

Check that `self` is a valid strong k-tableau.

This function verifies that the outer and inner shape of the parent class is equal to the outer and inner shape of the tableau, that the tableau portion of `self` is a valid standard tableau, that the marks are placed correctly and that the size and weight agree.

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -2], [2]], 2)
sage: T.check()
sage: T = StrongTableau([[None, None, 2, -4, -4], [-1, 4], [-2]], 3)
sage: T.check()
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1), -Integer(1), -Integer(2)], [Integer(2)]],
↪ Integer(2))
>>> T.check()
>>> T = StrongTableau([[None, None, Integer(2), -Integer(4), -Integer(4)], [-
↪Integer(1), Integer(4)], [-Integer(2)]], Integer(3))
>>> T.check()
```

content_of_highest_head(*v*)

Return the diagonal of the highest head of the cells labeled `v` in the standard part of `self`.

Return the content of the cell of the head in the highest row of all ribbons labeled by `v` of the underlying standard tableau. If there is no cell with entry `v` then the value returned is the length of the first row.

INPUT:

- `v` – integer representing the label in the standard tableau

OUTPUT: an integer representing the content of the head of the highest ribbon with label `v`

EXAMPLES:

```
sage: [StrongTableau([[-1,2,-3],[-2,3],[3]], 1).content_of_highest_head(v)
↪for v in range(1,5)]
[0, -1, -2, 3]
```

```
>>> from sage.all import *
>>> [StrongTableau([[-Integer(1),Integer(2),-Integer(3)],[-Integer(2),
↪Integer(3)],[Integer(3)]], Integer(1)).content_of_highest_head(v) for v in
↪range(Integer(1),Integer(5))]
[0, -1, -2, 3]
```

content_of_marked_head(*v*)

Return the diagonal of the marked label `v` in the standard part of `self`.

Return the content (the $j - i$ coordinate of the cell) of the v-th marked cell in the strong standard tableau self. If there is no mark, then the value returned is the size of first row.

INPUT:

- v – integer representing the label in the standard tableau

OUTPUT: integer representing the residue of the location of the mark

EXAMPLES:

```
sage: [ StrongTableau([[-1, -3, 4, -5], [-2], [-4]], 3).content_of_marked_
↪head(i) for i in range(1,7)]
[0, -1, 1, -2, 3, 4]
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: [ T.content_of_marked_head(i) for i in range(1,7)]
[-2, 2, -1, 3, -4, 4]
```

```
>>> from sage.all import *
>>> [ StrongTableau([[-Integer(1), -Integer(3), Integer(4), -Integer(5)], [-
↪Integer(2)], [-Integer(4)]], Integer(3)).content_of_marked_head(i) for i in␣
↪range(Integer(1),Integer(7))]
[0, -1, 1, -2, 3, 4]
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],␣
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
>>> [ T.content_of_marked_head(i) for i in range(Integer(1),Integer(7))]
[-2, 2, -1, 3, -4, 4]
```

contents_of_heads(*v*)

A list of contents of the cells which are heads of the ribbons with label v.

If there is no cell labelled by v then return the empty list.

INPUT:

- v – integer label

OUTPUT: list of integers of the content of the heads of the ribbons with label v

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: T.contents_of_heads(1)
[-2]
sage: T.contents_of_heads(2)
[-3, 2]
sage: T.contents_of_heads(3)
[-1]
sage: T.contents_of_heads(4)
[-2, 3]
sage: T.contents_of_heads(5)
[-4]
sage: T.contents_of_heads(6)
[]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],␣
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
>>> T.contents_of_heads(Integer(1))
[-2]
>>> T.contents_of_heads(Integer(2))
[-3, 2]
>>> T.contents_of_heads(Integer(3))
[-1]
>>> T.contents_of_heads(Integer(4))
[-2, 3]
>>> T.contents_of_heads(Integer(5))
[-4]
>>> T.contents_of_heads(Integer(6))
[]
```

entries_by_content(*diag*)

Return the entries on the diagonal of `self`.

Return the entries in the tableau that are in the cells (i, j) with $j - i$ equal to `diag` (that is, with content equal to `diag`).

INPUT:

- `diag` – integer indicating the diagonal

OUTPUT: list (perhaps empty) of labels on the diagonal `diag`

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: T.entries_by_content(0)
[]
sage: T.entries_by_content(1)
[]
sage: T.entries_by_content(2)
[-1]
sage: T.entries_by_content(-2)
[-1, 2]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],␣
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
>>> T.entries_by_content(Integer(0))
[]
>>> T.entries_by_content(Integer(1))
[]
>>> T.entries_by_content(Integer(2))
[-1]
>>> T.entries_by_content(-Integer(2))
[-1, 2]
```

entries_by_content_standard(*diag*)

Return the entries on the diagonal of the standard part of `self`.

Return the entries in the tableau that are in the cells (i, j) with $j - i$ equal to `diag` (that is, with content equal to `diag`) in the standard tableau.

INPUT:

- `diag` – integer indicating the diagonal

OUTPUT: list (perhaps empty) of labels on the diagonal `diag`

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: T.entries_by_content_standard(0)
[]
sage: T.entries_by_content_standard(1)
[]
sage: T.entries_by_content_standard(2)
[-2]
sage: T.entries_by_content_standard(-2)
[-1, 4]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
>>> T.entries_by_content_standard(Integer(0))
[]
>>> T.entries_by_content_standard(Integer(1))
[]
>>> T.entries_by_content_standard(Integer(2))
[-2]
>>> T.entries_by_content_standard(-Integer(2))
[-1, 4]
```

follows_tableau()

Return a list of strong marked tableaux with length one longer than `self`.

Return list of all strong tableaux obtained from `self` by extending to a core which follows the shape of `self` in the strong order.

OUTPUT: list of strong tableaux which follow `self` in strong order

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,
↪1])
sage: T.follows_tableau()
[[[-1, -1, -2, -3, 5, 5, -5], [-2, 3, -3, 4], [2, 3], [-3, -4]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [-5]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, -5], [-3, -4], [5]],
 [[-1, -1, -2, -3, -5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [5]],
 [[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4], [-5], [5], [5]]]
sage: StrongTableau([[-1,-2],[-3,-4]],3).follows_tableau()
```

(continues on next page)

(continued from previous page)

```
[[[-1, -2, 5, 5, -5], [-3, -4]], [[-1, -2, 5], [-3, -4], [-5]],
 [[-1, -2, -5], [-3, -4], [5]], [[-1, -2], [-3, -4], [-5], [5], [5]]]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3), [Integer(2),Integer(2),Integer(3),
↪Integer(1)])
>>> T.follows_tableau()
[[[-1, -1, -2, -3, 5, 5, -5], [-2, 3, -3, 4], [2, 3], [-3, -4]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [-5]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, -5], [-3, -4], [5]],
 [[-1, -1, -2, -3, -5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [5]],
 [[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4], [-5], [5], [5]]]
>>> StrongTableau([[-Integer(1),-Integer(2)],[-Integer(3),-Integer(4)]],
↪Integer(3)).follows_tableau()
[[[-1, -2, 5, 5, -5], [-3, -4]], [[-1, -2, 5], [-3, -4], [-5]],
 [[-1, -2, -5], [-3, -4], [5]], [[-1, -2], [-3, -4], [-5], [5], [5]]]
```

height_of_ribbon(*v*)

The number of rows occupied by one of the ribbons with label `v`.

The number of rows occupied by the marked ribbon with label `v` (and by consequence the number of rows occupied by any ribbon with the same label) in the standard part of `self`.

INPUT:

- `v` – the label of the standard marked tableau

OUTPUT: nonnegative integer representing the number of rows occupied by the ribbon which is marked

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -2, -2, 3], [2, -3], [-3]],3)
sage: T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
sage: T.height_of_ribbon(1)
1
sage: T.height_of_ribbon(4)
1
sage: T = StrongTableau([[None,None,1,-2],[None,-3,4,-5],[-1,3],[-4,5]], 3)
sage: T.height_of_ribbon(3)
2
sage: T.height_of_ribbon(6)
0
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1), -Integer(1), -Integer(2), -Integer(2),␣
↪Integer(3)], [Integer(2), -Integer(3)], [-Integer(3)]],Integer(3))
>>> T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
>>> T.height_of_ribbon(Integer(1))
1
>>> T.height_of_ribbon(Integer(4))
```

(continues on next page)

(continued from previous page)

```
1
>>> T = StrongTableau([[None,None,Integer(1),-Integer(2)],[None,-Integer(3),
↪Integer(4),-Integer(5)],[-Integer(1),Integer(3)],[-Integer(4),Integer(5)]],␣
↪Integer(3))
>>> T.height_of_ribbon(Integer(3))
2
>>> T.height_of_ribbon(Integer(6))
0
```

inner_shape()

Return the inner shape of `self`.

If `self` is a strong skew tableau, then this method returns the inner shape (the shape of the cells labelled with `None`). If `self` is not skew, then the inner shape is empty.

OUTPUT:

- a $(k+1)$-core

EXAMPLES:

```
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-
↪3], [3], [3], [3]], 4).inner_shape()
[2, 2]
sage: StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1]).
↪inner_shape()
[]
```

```
>>> from sage.all import *
>>> StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None], [-
↪Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4)).inner_shape()
[2, 2]
>>> StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3), [Integer(2),Integer(2),Integer(3),
↪Integer(1)]).inner_shape()
[]
```

intermediate_shapes()

Return the intermediate shapes of `self`.

A (skew) tableau with letters $1, 2, \ldots, \ell$ can be viewed as a sequence of shapes, where the i-th shape is given by the shape of the subtableau on letters $1, 2, \ldots, i$.

The output is the list of these shapes. The marked cells are ignored so to recover the strong tableau one would need the intermediate shapes and the *content_of_marked_head()* for each pair of adjacent shapes in the list.

OUTPUT: list of lists of integers representing $k+1$-cores

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,
↪1])
sage: T.intermediate_shapes()
```

(continues on next page)

(continued from previous page)

```
[[], [2], [3, 1, 1], [4, 3, 2, 1], [4, 4, 2, 2]]
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: T.intermediate_shapes()
[[2, 2], [3, 2, 1, 1], [4, 2, 2, 2], [4, 2, 2, 2, 1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3), [Integer(2),Integer(2),Integer(3),
↪Integer(1)])
>>> T.intermediate_shapes()
[[], [2], [3, 1, 1], [4, 3, 2, 1], [4, 4, 2, 2]]
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],␣
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
>>> T.intermediate_shapes()
[[2, 2], [3, 2, 1, 1], [4, 2, 2, 2], [4, 2, 2, 2, 1, 1, 1, 1]]
```

is_column_strict_with_weight(*mu*)

Test if `self` is a column strict tableau with respect to the weight `mu`.

INPUT:

- `mu` – a vector of weights

OUTPUT: boolean; `True` means the underlying column strict strong marked tableau is valid

EXAMPLES:

```
sage: StrongTableau([[-1, -2, -3], [3]], 2).is_column_strict_with_weight([3])
True
sage: StrongTableau([[-1, -2, 3], [-3]], 2).is_column_strict_with_weight([3])
False
```

```
>>> from sage.all import *
>>> StrongTableau([[-Integer(1), -Integer(2), -Integer(3)], [Integer(3)]],␣
↪Integer(2)).is_column_strict_with_weight([Integer(3)])
True
>>> StrongTableau([[-Integer(1), -Integer(2), Integer(3)], [-Integer(3)]],␣
↪Integer(2)).is_column_strict_with_weight([Integer(3)])
False
```

left_action(*tij*)

Action of transposition `tij` on `self` by adding marked ribbons.

Computes the left action of the transposition `tij` on the tableau. If `tij` acting on the element of the affine Grassmannian raises the length by 1, then this function will add a cell to the standard tableau.

INPUT:

- `tij` – a transposition represented as a pair (i, j)

OUTPUT: `self` after it has been modified by the action of the transposition `tij`

EXAMPLES:

```
sage: StrongTableau( [[None, -1, -2, -3], [3], [-4]], 3, weight=[1,1,1,1] ).
↪left_action([0,1])
[[None, -1, -2, -3, 5], [3, -5], [-4]]
sage: StrongTableau( [[None, -1, -2, -3], [3], [-4]], 3, weight=[1,1,1,1] ).
↪left_action([4,5])
[[None, -1, -2, -3, -5], [3, 5], [-4]]
sage: T = StrongTableau( [[None, -1, -2, -3], [3], [-4]], 3, weight=[1,1,1,1]␣
↪)
sage: T.left_action([-3,-2])
[[None, -1, -2, -3], [3], [-4], [-5]]
sage: T = StrongTableau( [[None, -1, -2, -3], [3], [-4]], 3, weight=[3,1] )
sage: T.left_action([-3,-2])
[[None, -1, -1, -1], [1], [-2], [-3]]
sage: T
[[None, -1, -1, -1], [1], [-2]]
sage: T.check()
sage: T.weight()
(3, 1)
```

```
>>> from sage.all import *
>>> StrongTableau( [[None, -Integer(1), -Integer(2), -Integer(3)],␣
↪[Integer(3)], [-Integer(4)]], Integer(3), weight=[Integer(1),Integer(1),
↪Integer(1),Integer(1)] ).left_action([Integer(0),Integer(1)])
[[None, -1, -2, -3, 5], [3, -5], [-4]]
>>> StrongTableau( [[None, -Integer(1), -Integer(2), -Integer(3)],␣
↪[Integer(3)], [-Integer(4)]], Integer(3), weight=[Integer(1),Integer(1),
↪Integer(1),Integer(1)] ).left_action([Integer(4),Integer(5)])
[[None, -1, -2, -3, -5], [3, 5], [-4]]
>>> T = StrongTableau( [[None, -Integer(1), -Integer(2), -Integer(3)],␣
↪[Integer(3)], [-Integer(4)]], Integer(3), weight=[Integer(1),Integer(1),
↪Integer(1),Integer(1)] )
>>> T.left_action([-Integer(3),-Integer(2)])
[[None, -1, -2, -3], [3], [-4], [-5]]
>>> T = StrongTableau( [[None, -Integer(1), -Integer(2), -Integer(3)],␣
↪[Integer(3)], [-Integer(4)]], Integer(3), weight=[Integer(3),Integer(1)] )
>>> T.left_action([-Integer(3),-Integer(2)])
[[None, -1, -1, -1], [1], [-2], [-3]]
>>> T
[[None, -1, -1, -1], [1], [-2]]
>>> T.check()
>>> T.weight()
(3, 1)
```

number_of_connected_components(*v*)

Number of connected components of ribbons with label `v` in the standard part.

The number of connected components is calculated by finding the number of cells with label `v` in the standard part of the tableau and dividing by the number of cells in the ribbon.

INPUT:

- `v` – the label of the standard marked tableau

OUTPUT: nonnegative integer representing the number of connected components

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -2, -2, 3], [2, -3], [-3]],3)
sage: T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
sage: T.number_of_connected_components(1)
1
sage: T.number_of_connected_components(4)
2
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3)
sage: T.number_of_connected_components(6)
1
sage: T.number_of_connected_components(9)
0
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1), -Integer(1), -Integer(2), -Integer(2),␣
→Integer(3)], [Integer(2), -Integer(3)], [-Integer(3)]],Integer(3))
>>> T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
>>> T.number_of_connected_components(Integer(1))
1
>>> T.number_of_connected_components(Integer(4))
2
>>> T = StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
→Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
→Integer(5),-Integer(8)]], Integer(3))
>>> T.number_of_connected_components(Integer(6))
1
>>> T.number_of_connected_components(Integer(9))
0
```

outer_shape()

Return the outer shape of `self`.

This method returns the outer shape of `self` as viewed as a `Core`. The outer shape of a strong tableau is always a $(k+1)$-core.

OUTPUT:

- a $(k+1)$-core

EXAMPLES:

```
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-
→3], [3], [3], [3]], 4).outer_shape()
[4, 2, 2, 2, 1, 1, 1, 1]
sage: StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1]).
→outer_shape()
[4, 4, 2, 2]
```

```
>>> from sage.all import *
>>> StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None], [-
→Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
→[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4)).outer_shape()
```

(continues on next page)

(continued from previous page)

```
[4, 2, 2, 2, 1, 1, 1, 1]
>>> StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3), [Integer(2),Integer(2),Integer(3),
↪Integer(1)]).outer_shape()
[4, 4, 2, 2]
```

pp()

Print the strong tableau self in pretty print format.

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,
↪1])
sage: T.pp()
-1 -1 -2 -3
-2  3 -3  4
 2  3
-3 -4
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: T.pp()
  .  . -1 -2
  .  .
 -1 -2
  1  2
 -3
  3
  3
  3
sage: Tableaux.options(convention='French')
sage: T.pp()
  3
  3
  3
 -3
  1  2
 -1 -2
  .  .
  .  . -1 -2
sage: Tableaux.options(convention='English')
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3), [Integer(2),Integer(2),Integer(3),
↪Integer(1)])
>>> T.pp()
-1 -1 -2 -3
-2  3 -3  4
 2  3
-3 -4
```

(continues on next page)

(continued from previous page)

```
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],␣
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
>>> T.pp()
  .  . -1 -2
  .  .
 -1 -2
  1  2
 -3
  3
  3
  3
>>> Tableaux.options(convention='French')
>>> T.pp()
  3
  3
  3
 -3
  1  2
 -1 -2
  .  .
  .  . -1 -2
>>> Tableaux.options(convention='English')
```

restrict(*r*)

Restrict the standard part of the tableau to the labels $1, 2, \ldots, r$.

Return the tableau consisting of the labels of the standard part of `self` restricted to the labels of 1 through `r`. The result is another `StrongTableau` object.

INPUT:

- `r` – integer

OUTPUT: a strong tableau

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -4, 5, -5], [None, None], [-1, -3], [-
↪2], [2], [2], [3]], 4, weight=[1,1,1,1,1])
sage: T.restrict(3)
[[None, None], [None, None], [-1, -3], [-2], [2], [2], [3]]
sage: TT = T.restrict(0)
sage: TT
[[None, None], [None, None]]
sage: TT == StrongTableau( [[None, None], [None, None]], 4 )
True
sage: T.restrict(5) == T
True
```

```
>>> from sage.all import *
>>> T = StrongTableau([[None, None, -Integer(4), Integer(5), -Integer(5)],␣
↪[None, None], [-Integer(1), -Integer(3)], [-Integer(2)], [Integer(2)],␣
↪[Integer(2)], [Integer(3)]], Integer(4), weight=[Integer(1),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(1),Integer(1)])
>>> T.restrict(Integer(3))
[[None, None], [None, None], [-1, -3], [-2], [2], [2], [3]]
>>> TT = T.restrict(Integer(0))
>>> TT
[[None, None], [None, None]]
>>> TT == StrongTableau( [[None, None], [None, None]], Integer(4) )
True
>>> T.restrict(Integer(5)) == T
True
```

ribbons_above_marked(*v*)

Number of ribbons of label `v` higher than the marked ribbon in the standard part.

Return the number of copies of the ribbon with label `v` in the standard part of `self` which are in a higher row than the marked ribbon. Note that the result is independent of the weight of the tableau.

INPUT:

- `v` – the entry of the standard tableau

OUTPUT:

- an integer representing the number of copies of the ribbon above the marked ribbon

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3)
sage: T.ribbons_above_marked(4)
1
sage: T.ribbons_above_marked(6)
0
sage: T.ribbons_above_marked(9)
0
sage: StrongTableau([[-1,-2,-3,-4],[2,3,4],[3,4],[4]], 1).ribbons_above_
↪marked(4)
3
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3))
>>> T.ribbons_above_marked(Integer(4))
1
>>> T.ribbons_above_marked(Integer(6))
0
>>> T.ribbons_above_marked(Integer(9))
0
>>> StrongTableau([[-Integer(1),-Integer(2),-Integer(3),-Integer(4)],
↪[Integer(2),Integer(3),Integer(4)],[Integer(3),Integer(4)],[Integer(4)]],␣
↪Integer(1)).ribbons_above_marked(Integer(4))
3
```

set_weight(*mu*)

Set a new weight `mu` for `self`.

This method first tests if the underlying standard tableau is column-strict with respect to the weight `mu`. If it is, then it changes the weight and returns the tableau; otherwise it raises an error.

INPUT:

- `mu` – list of nonnegative integers representing the new weight

EXAMPLES:

```
sage: StrongTableau( [[-1, -2, -3], [3]], 2 ).set_weight( [3] )
[[-1, -1, -1], [1]]
sage: StrongTableau( [[-1, -2, -3], [3]], 2 ).set_weight( [0,3] )
[[-2, -2, -2], [2]]
sage: StrongTableau( [[-1, -2, 3], [-3]], 2 ).set_weight( [2, 0, 1] )
[[-1, -1, 3], [-3]]
sage: StrongTableau( [[-1, -2, 3], [-3]], 2 ).set_weight( [3] )
Traceback (most recent call last):
...
ValueError: [[-1, -2, 3], [-3]] is not a semistandard strong tableau with␣
↪respect to the partition [3]
```

```
>>> from sage.all import *
>>> StrongTableau( [[-Integer(1), -Integer(2), -Integer(3)], [Integer(3)]],␣
↪Integer(2) ).set_weight( [Integer(3)] )
[[-1, -1, -1], [1]]
>>> StrongTableau( [[-Integer(1), -Integer(2), -Integer(3)], [Integer(3)]],␣
↪Integer(2) ).set_weight( [Integer(0),Integer(3)] )
[[-2, -2, -2], [2]]
>>> StrongTableau( [[-Integer(1), -Integer(2), Integer(3)], [-Integer(3)]],␣
↪Integer(2) ).set_weight( [Integer(2), Integer(0), Integer(1)] )
[[-1, -1, 3], [-3]]
>>> StrongTableau( [[-Integer(1), -Integer(2), Integer(3)], [-Integer(3)]],␣
↪Integer(2) ).set_weight( [Integer(3)] )
Traceback (most recent call last):
...
ValueError: [[-1, -2, 3], [-3]] is not a semistandard strong tableau with␣
↪respect to the partition [3]
```

shape()

Return the shape of `self`.

If `self` is a skew tableau then return a pair of $k+1$-cores consisting of the outer and the inner shape. If `self` is strong tableau with no inner shape then return a $k+1$-core.

INPUT:

- `form` – argument to indicate `'inner'`, `'outer'` or `'skew'` (default: `'outer'`)

OUTPUT: a $k+1$-core or a pair of $k+1$-cores if form is not `'inner'` or `'outer'`

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2],
↪ [-3], [3], [3], [3]], 4)
sage: T.shape()
([4, 2, 2, 2, 1, 1, 1, 1], [2, 2])
sage: StrongTableau([[-1, -2, 3], [-3]], 2).shape()
```

(continues on next page)

(continued from previous page)

```
[3, 1]
sage: type(StrongTableau([[-1, -2, 3], [-3]], 2).shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
```

```
>>> from sage.all import *
>>> T = StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None],␣
↪[-Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4))
>>> T.shape()
([4, 2, 2, 2, 1, 1, 1, 1], [2, 2])
>>> StrongTableau([[-Integer(1), -Integer(2), Integer(3)], [-Integer(3)]],␣
↪Integer(2)).shape()
[3, 1]
>>> type(StrongTableau([[-Integer(1), -Integer(2), Integer(3)], [-
↪Integer(3)]], Integer(2)).shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
```

size()

Return the size of the strong tableau.

The size of the strong tableau is the sum of the entries in the *weight()*. It will also be equal to the length of the outer shape (as a $k+1$-core) minus the length of the inner shape.

See also

sage.combinat.core.Core.length()

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3).size()
5
sage: StrongTableau([[None, None, -1, 2], [-2], [-3]], 3).size()
3
```

```
>>> from sage.all import *
>>> StrongTableau([[-Integer(1), -Integer(2), -Integer(3), Integer(4)], [-
↪Integer(4)], [-Integer(5)]], Integer(3)).size()
5
>>> StrongTableau([[None, None, -Integer(1), Integer(2)], [-Integer(2)], [-
↪Integer(3)]], Integer(3)).size()
3
```

spin()

Return the spin statistic of the tableau `self`.

The spin is an integer statistic on a strong marked tableau. It is the sum of $(h-1)r$ plus the number of connected components above the marked one where h is the height of the marked ribbon and r is the number of connected components.

See also

height_of_ribbon(), *number_of_connected_components()*, *ribbons_above_marked()*

The k-Schur functions with a parameter t can be defined as

$$s_\lambda^{(k)}[X;t] = \sum_T t^{spin(T)} m_{weight(T)}[X]$$

where the sum is over all column strict marked strong k-tableaux of shape λ and partition content.

OUTPUT: integer value representing the spin

EXAMPLES:

```
sage: StrongTableau([[-1,-2,5,6],[-3,-4,-7,8],[-5,-6],[7,-8]], 3, [2,2,3,1]).
↪spin()
1
sage: StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1]).
↪spin()
2
sage: StrongTableau([[None,None,-1,-3],[-2,3,-3,4],[2,3],[-3,-4]], 3).spin()
2
sage: ks3 = SymmetricFunctions(QQ['t'].fraction_field()).kschur(3)
sage: t = ks3.realization_of().t
sage: m = ks3.ambient().realization_of().m()
sage: myks221 = sum(sum(t**T.spin() for T in StrongTableaux(3,[3,2,1],
↪weight=mu))*m(mu) for mu in Partitions(5, max_part=3))
sage: myks221 == m(ks3[2,2,1])
True
sage: h = ks3.ambient().realization_of().h()
sage: Core([4,4,2,2],4).to_bounded_partition()
[2, 2, 2, 2]
sage: ks3[2,2,2,2].lift().scalar(h[3,3,2]) == sum( t**T.spin() for T in␣
↪StrongTableaux(3, [4,4,2,2], weight=[3,3,2]) )
True
```

```
>>> from sage.all import *
>>> StrongTableau([[-Integer(1),-Integer(2),Integer(5),Integer(6)],[-
↪Integer(3),-Integer(4),-Integer(7),Integer(8)],[-Integer(5),-Integer(6)],
↪[Integer(7),-Integer(8)]], Integer(3), [Integer(2),Integer(2),Integer(3),
↪Integer(1)]).spin()
1
>>> StrongTableau([[-Integer(1),-Integer(2),-Integer(4),-Integer(7)],[-
↪Integer(3),Integer(6),-Integer(6),Integer(8)],[Integer(4),Integer(7)],[-
↪Integer(5),-Integer(8)]], Integer(3), [Integer(2),Integer(2),Integer(3),
↪Integer(1)]).spin()
2
>>> StrongTableau([[None,None,-Integer(1),-Integer(3)],[-Integer(2),
↪Integer(3),-Integer(3),Integer(4)],[Integer(2),Integer(3)],[-Integer(3),-
↪Integer(4)]], Integer(3)).spin()
2
>>> ks3 = SymmetricFunctions(QQ['t'].fraction_field()).kschur(Integer(3))
>>> t = ks3.realization_of().t
```

(continues on next page)

(continued from previous page)

```
>>> m = ks3.ambient().realization_of().m()
>>> myks221 = sum(sum(t**T.spin() for T in StrongTableaux(Integer(3),
↪[Integer(3),Integer(2),Integer(1)],weight=mu))*m(mu) for mu in␣
↪Partitions(Integer(5), max_part=Integer(3)))
>>> myks221 == m(ks3[Integer(2),Integer(2),Integer(1)])
True
>>> h = ks3.ambient().realization_of().h()
>>> Core([Integer(4),Integer(4),Integer(2),Integer(2)],Integer(4)).to_bounded_
↪partition()
[2, 2, 2, 2]
>>> ks3[Integer(2),Integer(2),Integer(2),Integer(2)].lift().
↪scalar(h[Integer(3),Integer(3),Integer(2)]) == sum( t**T.spin() for T in␣
↪StrongTableaux(Integer(3), [Integer(4),Integer(4),Integer(2),Integer(2)],␣
↪weight=[Integer(3),Integer(3),Integer(2)]) )
True
```

spin_of_ribbon(*v*)

Return the spin of the ribbon with label `v` in the standard part of `self`.

The spin of a ribbon is an integer statistic. It is the sum of $(h-1)r$ plus the number of connected components above the marked one where h is the height of the marked ribbon and r is the number of connected components.

See also

height_of_ribbon(), *number_of_connected_components()*, *ribbons_above_marked()*

INPUT:

- `v` – a label of the standard part of the tableau

OUTPUT: integer value representing the spin of the ribbon with label `v`

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,5,6],[-3,-4,-7,8],[-5,-6],[7,-8]], 3)
sage: [T.spin_of_ribbon(v) for v in range(1,9)]
[0, 0, 0, 0, 0, 0, 1, 0]
sage: T = StrongTableau([[None,None,-1,-3],[-2,3,-3,4],[2,3],[-3,-4]], 3)
sage: [T.spin_of_ribbon(v) for v in range(1,7)]
[0, 1, 0, 0, 1, 0]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1),-Integer(2),Integer(5),Integer(6)],[-
↪Integer(3),-Integer(4),-Integer(7),Integer(8)],[-Integer(5),-Integer(6)],
↪[Integer(7),-Integer(8)]], Integer(3))
>>> [T.spin_of_ribbon(v) for v in range(Integer(1),Integer(9))]
[0, 0, 0, 0, 0, 0, 1, 0]
>>> T = StrongTableau([[None,None,-Integer(1),-Integer(3)],[-Integer(2),
↪Integer(3),-Integer(3),Integer(4)],[Integer(2),Integer(3)],[-Integer(3),-
↪Integer(4)]], Integer(3))
>>> [T.spin_of_ribbon(v) for v in range(Integer(1),Integer(7))]
[0, 1, 0, 0, 1, 0]
```

to_list()

Return the marked column strict (possibly skew) tableau as a list of lists.

OUTPUT: list of lists of integers or None

EXAMPLES:

```
sage: StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3).set_weight([2,1,1,1]).
↪to_list()
[[-1, -1, -2, 3], [-3], [-4]]
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-
↪3], [3], [3], [3]], 4).to_list()
[[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3], [3]]
sage: StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1]).to_list()
[[-1, -1, -1, 2], [-2], [-3]]
```

```
>>> from sage.all import *
>>> StrongTableau([[-Integer(1), -Integer(2), -Integer(3), Integer(4)], [-
↪Integer(4)], [-Integer(5)]], Integer(3)).set_weight([Integer(2),Integer(1),
↪Integer(1),Integer(1)]).to_list()
[[-1, -1, -2, 3], [-3], [-4]]
>>> StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None], [-
↪Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4)).to_list()
[[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3], [3]]
>>> StrongTableau([[-Integer(1), -Integer(2), -Integer(3), Integer(4)], [-
↪Integer(4)], [-Integer(5)]], Integer(3), [Integer(3),Integer(1),
↪Integer(1)]).to_list()
[[-1, -1, -1, 2], [-2], [-3]]
```

to_standard_list()

Return the underlying standard strong tableau as a list of lists.

Internally, for a strong tableau the standard strong tableau and its weight is stored separately. This method returns the underlying standard part.

OUTPUT: list of lists of integers or None

EXAMPLES:

```
sage: StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1]).to_standard_
↪list()
[[-1, -2, -3, 4], [-4], [-5]]
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-
↪3], [3], [3], [3]], 4).to_standard_list()
[[None, None, -2, -4], [None, None], [-1, -3], [2, 4], [-5], [5], [5], [5]]
```

```
>>> from sage.all import *
>>> StrongTableau([[-Integer(1), -Integer(2), -Integer(3), Integer(4)], [-
↪Integer(4)], [-Integer(5)]], Integer(3), [Integer(3),Integer(1),
↪Integer(1)]).to_standard_list()
[[-1, -2, -3, 4], [-4], [-5]]
>>> StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None], [-
↪Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4)).to_standard_list()
[[None, None, -2, -4], [None, None], [-1, -3], [2, 4], [-5], [5], [5], [5]]
```

to_standard_tableau()

Return the underlying standard strong tableau as a `StrongTableau` object.

Internally, for a strong tableau the standard strong tableau and its weight is stored separately. This method returns the underlying standard part as a `StrongTableau`.

OUTPUT: a strong tableau with standard weight

EXAMPLES:

```
sage: T = StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1])
sage: T.to_standard_tableau()
[[-1, -2, -3, 4], [-4], [-5]]
sage: T.to_standard_tableau() == T.to_standard_list()
False
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-
↪3], [3], [3], [3]], 4).to_standard_tableau()
[[None, None, -2, -4], [None, None], [-1, -3], [2, 4], [-5], [5], [5], [5]]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1), -Integer(2), -Integer(3), Integer(4)], [-
↪Integer(4)], [-Integer(5)]], Integer(3), [Integer(3),Integer(1),Integer(1)])
>>> T.to_standard_tableau()
[[-1, -2, -3, 4], [-4], [-5]]
>>> T.to_standard_tableau() == T.to_standard_list()
False
>>> StrongTableau([[None, None, -Integer(1), -Integer(2)], [None, None], [-
↪Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4)).to_standard_tableau()
[[None, None, -2, -4], [None, None], [-1, -3], [2, 4], [-5], [5], [5], [5]]
```

to_transposition_sequence()

Return a list of transpositions corresponding to `self`.

Given a strong column strict tableau `self` returns the list of transpositions which when applied to the left of an empty tableau gives the corresponding strong standard tableau.

OUTPUT: list of pairs of values `[i,j]` representing the transpositions t_{ij}

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -1], [1]],2)
sage: T.to_transposition_sequence()
[[2, 3], [1, 2], [0, 1]]
sage: T = StrongTableau([[-1, -1, 2], [-2]],2)
sage: T.to_transposition_sequence()
[[-1, 0], [1, 2], [0, 1]]
sage: T = StrongTableau([[None, -1, 2, -3], [-2, 3]],2)
sage: T.to_transposition_sequence()
[[3, 4], [-1, 0], [1, 2]]
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1), -Integer(1), -Integer(1)], [Integer(1)]],
↪Integer(2))
>>> T.to_transposition_sequence()
[[2, 3], [1, 2], [0, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> T = StrongTableau([[-Integer(1), -Integer(1), Integer(2)], [-Integer(2)]],
↪Integer(2))
>>> T.to_transposition_sequence()
[[-1, 0], [1, 2], [0, 1]]
>>> T = StrongTableau([[None, -Integer(1), Integer(2), -Integer(3)], [-
↪Integer(2), Integer(3)]],Integer(2))
>>> T.to_transposition_sequence()
[[3, 4], [-1, 0], [1, 2]]
```

to_unmarked_list()

Return the tableau as a list of lists with markings removed.

Return the list of lists of the rows of the tableau where the markings have been removed.

OUTPUT: list of lists of integers or `None`

EXAMPLES:

```
sage: T = StrongTableau( [[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1])
sage: T.to_unmarked_list()
[[1, 1, 1, 2], [2], [3]]
sage: TT = T.set_weight([2,1,1,1])
sage: TT.to_unmarked_list()
[[1, 1, 2, 3], [3], [4]]
sage: StrongTableau( [[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-
↪3], [3], [3], [3]], 4).to_unmarked_list()
[[None, None, 1, 2], [None, None], [1, 2], [1, 2], [3], [3], [3], [3]]
```

```
>>> from sage.all import *
>>> T = StrongTableau( [[-Integer(1), -Integer(2), -Integer(3), Integer(4)],␣
↪[-Integer(4)], [-Integer(5)]], Integer(3), [Integer(3),Integer(1),
↪Integer(1)])
>>> T.to_unmarked_list()
[[1, 1, 1, 2], [2], [3]]
>>> TT = T.set_weight([Integer(2),Integer(1),Integer(1),Integer(1)])
>>> TT.to_unmarked_list()
[[1, 1, 2, 3], [3], [4]]
>>> StrongTableau( [[None, None, -Integer(1), -Integer(2)], [None, None], [-
↪Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4)).to_unmarked_list()
[[None, None, 1, 2], [None, None], [1, 2], [1, 2], [3], [3], [3], [3]]
```

to_unmarked_standard_list()

Return the standard part of the tableau as a list of lists with markings removed.

Return the list of lists of the rows of the tableau where the markings have been removed.

OUTPUT: list of lists of integers or `None`

EXAMPLES:

```
sage: StrongTableau( [[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1]).to_unmarked_
↪standard_list()
[[1, 2, 3, 4], [4], [5]]
sage: StrongTableau( [[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-
```

(continues on next page)

(continued from previous page)

```
↪3], [3], [3], [3]], 4).to_unmarked_standard_list()
[[None, None, 2, 4], [None, None], [1, 3], [2, 4], [5], [5], [5], [5]]
```

```
>>> from sage.all import *
>>> StrongTableau( [[-Integer(1), -Integer(2), -Integer(3), Integer(4)], [-
↪Integer(4)], [-Integer(5)]], Integer(3), [Integer(3),Integer(1),
↪Integer(1)]).to_unmarked_standard_list()
[[1, 2, 3, 4], [4], [5]]
>>> StrongTableau( [[None, None, -Integer(1), -Integer(2)], [None, None], [-
↪Integer(1), -Integer(2)], [Integer(1), Integer(2)], [-Integer(3)],␣
↪[Integer(3)], [Integer(3)], [Integer(3)]], Integer(4)).to_unmarked_standard_
↪list()
[[None, None, 2, 4], [None, None], [1, 3], [2, 4], [5], [5], [5], [5]]
```

weight()

Return the weight of the tableau.

The weight is a list of nonnegative integers indicating the number of 1s, number of 2s, number of 3s, etc.

OUTPUT: list of nonnegative integers

EXAMPLES:

```
sage: T = StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3); T.weight()
(1, 1, 1, 1, 1)
sage: T.set_weight([3,1,1]).weight()
(3, 1, 1)
sage: StrongTableau([[-1,-1,-2,-3],[-2,3,-3,4],[2,3],[-3,-4]], 3).weight()
(2, 2, 3, 1)
```

```
>>> from sage.all import *
>>> T = StrongTableau([[-Integer(1), -Integer(2), -Integer(3), Integer(4)], [-
↪Integer(4)], [-Integer(5)]], Integer(3)); T.weight()
(1, 1, 1, 1, 1)
>>> T.set_weight([Integer(3),Integer(1),Integer(1)]).weight()
(3, 1, 1)
>>> StrongTableau([[-Integer(1),-Integer(1),-Integer(2),-Integer(3)],[-
↪Integer(2),Integer(3),-Integer(3),Integer(4)],[Integer(2),Integer(3)],[-
↪Integer(3),-Integer(4)]], Integer(3)).weight()
(2, 2, 3, 1)
```

class sage.combinat.k_tableau.**StrongTableaux**(*k*, *shape*, *weight*)

Bases: UniqueRepresentation, Parent

Element

alias of *StrongTableau*

classmethod add_marking(*unmarkedT*, *marking*, *k*, *weight*)

Add markings to a partially marked strong tableau.

Given a partially marked standard tableau and a list of cells where the marks should be placed along with a weight, return the semi-standard marked strong tableau. The marking should complete the marking so that the result is a strong standard marked tableau.

INPUT:

- `unmarkedT` – list of lists which is a partially marked strong k-tableau
- `marking` – list of pairs of coordinates where cells are to be marked
- `k` – positive integer
- `weight` – tuple of the weight of the output tableau

OUTPUT: a `StrongTableau` object

EXAMPLES:

```
sage: StrongTableaux.add_marking([[None,1,2],[2]], [(0,1), (1,0)], 2, [1,1])
[[None, -1, 2], [-2]]
sage: StrongTableaux.add_marking([[None,1,2],[2]], [(0,1), (1,0)], 2, [2])
Traceback (most recent call last):
...
ValueError: The weight=(2,) and the markings on the standard tableau=[[None, -
↪1, 2], [-2]] do not agree.
sage: StrongTableaux.add_marking([[None,1,2],[2]], [(0,1), (0,2)], 2, [2])
[[None, -1, -1], [1]]
```

```
>>> from sage.all import *
>>> StrongTableaux.add_marking([[None,Integer(1),Integer(2)],[Integer(2)]],
↪[(Integer(0),Integer(1)), (Integer(1),Integer(0))], Integer(2), [Integer(1),
↪Integer(1)])
[[None, -1, 2], [-2]]
>>> StrongTableaux.add_marking([[None,Integer(1),Integer(2)],[Integer(2)]],
↪[(Integer(0),Integer(1)), (Integer(1),Integer(0))], Integer(2),
↪[Integer(2)])
Traceback (most recent call last):
...
ValueError: The weight=(2,) and the markings on the standard tableau=[[None, -
↪1, 2], [-2]] do not agree.
>>> StrongTableaux.add_marking([[None,Integer(1),Integer(2)],[Integer(2)]],
↪[(Integer(0),Integer(1)), (Integer(0),Integer(2))], Integer(2),
↪[Integer(2)])
[[None, -1, -1], [1]]
```

an_element()

Return the first generated element of the class of `StrongTableaux`.

EXAMPLES:

```
sage: ST = StrongTableaux(3, [3], weight=[3])
sage: ST.an_element()
[[-1, -1, -1]]
```

```
>>> from sage.all import *
>>> ST = StrongTableaux(Integer(3), [Integer(3)], weight=[Integer(3)])
>>> ST.an_element()
[[-1, -1, -1]]
```

classmethod cells_head_dictionary(T)

Return a dictionary with the locations of the heads of all markings.

Return a dictionary of values and lists of cells where the heads with the values are located in a strong standard unmarked tableau `T`.

INPUT:

- `T` – a strong standard unmarked tableau as a list of lists

OUTPUT:

- a dictionary with keys the entries in the tableau and values are the coordinates of the heads with those entries

EXAMPLES:

```
sage: StrongTableaux.cells_head_dictionary([[1,2,4,7],[3,6,6,8],[4,7],[5,8]])
{1: [(0, 0)],
 2: [(0, 1)],
 3: [(1, 0)],
 4: [(2, 0), (0, 2)],
 5: [(3, 0)],
 6: [(1, 2)],
 7: [(2, 1), (0, 3)],
 8: [(3, 1), (1, 3)]}
sage: StrongTableaux.cells_head_dictionary([[None, 2, 2, 4, 5, 6, 6, 6],
↪[None, 3, 6, 6, 6], [1, 4]])
{1: [(2, 0)],
 2: [(0, 2)],
 3: [(1, 1)],
 4: [(2, 1), (0, 3)],
 5: [(0, 4)],
 6: [(1, 4), (0, 7)]}
```

```
>>> from sage.all import *
>>> StrongTableaux.cells_head_dictionary([[Integer(1),Integer(2),Integer(4),
↪Integer(7)],[Integer(3),Integer(6),Integer(6),Integer(8)],[Integer(4),
↪Integer(7)],[Integer(5),Integer(8)]])
{1: [(0, 0)],
 2: [(0, 1)],
 3: [(1, 0)],
 4: [(2, 0), (0, 2)],
 5: [(3, 0)],
 6: [(1, 2)],
 7: [(2, 1), (0, 3)],
 8: [(3, 1), (1, 3)]}
>>> StrongTableaux.cells_head_dictionary([[None, Integer(2), Integer(2),
↪Integer(4), Integer(5), Integer(6), Integer(6), Integer(6)], [None,
↪Integer(3), Integer(6), Integer(6), Integer(6)], [Integer(1), Integer(4)]])
{1: [(2, 0)],
 2: [(0, 2)],
 3: [(1, 1)],
 4: [(2, 1), (0, 3)],
 5: [(0, 4)],
 6: [(1, 4), (0, 7)]}
```

classmethod follows_tableau_unsigned_standard(*Tlist*, *k*)

Return a list of strong tableaux one longer in length than `Tlist`.

Return list of all standard strong tableaux obtained from `Tlist` by extending to a core which follows the shape of `Tlist` in the strong order. It does not put the markings on the last entry that it adds but it does keep the markings on all entries smaller. The objects returned are not `StrongTableau` objects (and cannot be) because the last entry will not properly marked.

INPUT:

- `Tlist` – a filling of a $k+1$-core as a list of lists
- `k` – integer

OUTPUT: list of strong tableaux which follow `Tlist` in strong order

EXAMPLES:

```
sage: StrongTableaux.follows_tableau_unsigned_standard([[-1, -1, -2, -3], [-2,
↪ 3, -3, 4], [2, 3], [-3, -4]], 3)
[[[-1, -1, -2, -3, 5, 5, 5], [-2, 3, -3, 4], [2, 3], [-3, -4]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [5]],
 [[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4], [5], [5], [5]]]
sage: StrongTableaux.follows_tableau_unsigned_standard([[None,-1],[-2,-3]],3)
[[[None, -1, 4, 4, 4], [-2, -3]], [[None, -1, 4], [-2, -3], [4]],
 [[None, -1], [-2, -3], [4], [4], [4]]]
```

```
>>> from sage.all import *
>>> StrongTableaux.follows_tableau_unsigned_standard([[-Integer(1), -
↪Integer(1), -Integer(2), -Integer(3)], [-Integer(2), Integer(3), -
↪Integer(3), Integer(4)], [Integer(2), Integer(3)], [-Integer(3), -
↪Integer(4)]], Integer(3))
[[[-1, -1, -2, -3, 5, 5, 5], [-2, 3, -3, 4], [2, 3], [-3, -4]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [5]],
 [[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4], [5], [5], [5]]]
>>> StrongTableaux.follows_tableau_unsigned_standard([[None,-Integer(1)],[-
↪Integer(2),-Integer(3)]],Integer(3))
[[[None, -1, 4, 4, 4], [-2, -3]], [[None, -1, 4], [-2, -3], [4]],
 [[None, -1], [-2, -3], [4], [4], [4]]]
```

inner_shape()

Return the inner shape of the class of strong tableaux.

OUTPUT: a $k+1$-core

EXAMPLES:

```
sage: StrongTableaux( 2, [3,1] ).inner_shape()
[]
sage: type(StrongTableaux( 2, [3,1] ).inner_shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: StrongTableaux( 4, [[2,1], [1]] ).inner_shape()
[1]
```

```
>>> from sage.all import *
>>> StrongTableaux( Integer(2), [Integer(3),Integer(1)] ).inner_shape()
[]
>>> type(StrongTableaux( Integer(2), [Integer(3),Integer(1)] ).inner_shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
```

(continues on next page)

(continued from previous page)

```
>>> StrongTableaux( Integer(4), [[Integer(2),Integer(1)], [Integer(1)]] ).
↪inner_shape()
[1]
```

classmethod marked_CST_to_transposition_sequence(T, k)

Return a list of transpositions corresponding to `T`.

Given a strong column strict tableau `T` returns the list of transpositions which when applied to the left of an empty tableau gives the corresponding strong standard tableau.

INPUT:

- `T` – a non-empty column strict tableau as a list of lists
- `k` – positive integer

OUTPUT: list of pairs of values `[i,j]` representing the transpositions t_{ij}

EXAMPLES:

```
sage: CST_to_trans = StrongTableaux.marked_CST_to_transposition_sequence
sage: CST_to_trans([[-1, -1, -1], [1]], 2)
[[2, 3], [1, 2], [0, 1]]
sage: CST_to_trans([], 2)
[]
sage: CST_to_trans([[-2, -2, -2], [2]], 2)
[[2, 3], [1, 2], [0, 1]]
sage: CST_to_trans([[-1, -2, -2, -2, -2], [-2, 2], [2]], 3)
[[4, 5], [3, 4], [2, 3], [1, 2], [-1, 0], [0, 1]]
sage: CST_to_trans([[-1, -2, -5, 5, -5, 5, -5], [-3, -4, 5, 5], [5]],3)
[[5, 7], [3, 5], [2, 3], [0, 1], [-1, 0], [1, 2], [0, 1]]
sage: CST_to_trans([[-1, -2, -3, 4, -7], [-4, -6], [-5, 6]],3)
[[4, 5], [-1, 1], [-2, -1], [-1, 0], [2, 3], [1, 2], [0, 1]]
```

```
>>> from sage.all import *
>>> CST_to_trans = StrongTableaux.marked_CST_to_transposition_sequence
>>> CST_to_trans([[-Integer(1), -Integer(1), -Integer(1)], [Integer(1)]],␣
↪Integer(2))
[[2, 3], [1, 2], [0, 1]]
>>> CST_to_trans([], Integer(2))
[]
>>> CST_to_trans([[-Integer(2), -Integer(2), -Integer(2)], [Integer(2)]],␣
↪Integer(2))
[[2, 3], [1, 2], [0, 1]]
>>> CST_to_trans([[-Integer(1), -Integer(2), -Integer(2), -Integer(2), -
↪Integer(2)], [-Integer(2), Integer(2)], [Integer(2)]], Integer(3))
[[4, 5], [3, 4], [2, 3], [1, 2], [-1, 0], [0, 1]]
>>> CST_to_trans([[-Integer(1), -Integer(2), -Integer(5), Integer(5), -
↪Integer(5), Integer(5), -Integer(5)], [-Integer(3), -Integer(4), Integer(5),
↪ Integer(5)], [Integer(5)]],Integer(3))
[[5, 7], [3, 5], [2, 3], [0, 1], [-1, 0], [1, 2], [0, 1]]
>>> CST_to_trans([[-Integer(1), -Integer(2), -Integer(3), Integer(4), -
↪Integer(7)], [-Integer(4), -Integer(6)], [-Integer(5), Integer(6)]],
↪Integer(3))
[[4, 5], [-1, 1], [-2, -1], [-1, 0], [2, 3], [1, 2], [0, 1]]
```

classmethod marked_given_unmarked_and_weight_iterator(*unmarkedT*, *k*, *weight*)

An iterator generating strong marked tableaux from an unmarked strong tableau.

Iterator which lists all marked tableaux of weight `weight` such that the standard unmarked part of the tableau is equal to `unmarkedT`.

INPUT:

- `unmarkedT` – list of lists representing a strong unmarked tableau
- `k` – positive integer
- `weight` – list of nonnegative integers indicating the weight

OUTPUT: an iterator that returns `StrongTableau` objects

EXAMPLES:

```
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[1,2,3],
↪[3]], 2, [3])
sage: list(ST)
[[[-1, -1, -1], [1]]]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[1,2,3],
↪[3]], 2, [0,3])
sage: list(ST)
[[[-2, -2, -2], [2]]]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[1,2,3],
↪[3]], 2, [1,2])
sage: list(ST)
[[[-1, -2, -2], [2]]]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[1,2,3],
↪[3]], 2, [2,1])
sage: list(ST)
[[[-1, -1, 2], [-2]], [[-1, -1, -2], [2]]]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[None,␣
↪None, 1, 2, 4], [2, 4], [3]], 3, [3,1])
sage: list(ST)
[]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[None,␣
↪None, 1, 2, 4], [2, 4], [3]], 3, [2,2])
sage: list(ST)
[[[None, None, -1, -1, 2], [1, -2], [-2]],
 [[None, None, -1, -1, -2], [1, 2], [-2]]]
```

```
>>> from sage.all import *
>>> ST = StrongTableaux.marked_given_unmarked_and_weight_
↪iterator([[Integer(1),Integer(2),Integer(3)],[Integer(3)]], Integer(2),␣
↪[Integer(3)])
>>> list(ST)
[[[-1, -1, -1], [1]]]
>>> ST = StrongTableaux.marked_given_unmarked_and_weight_
↪iterator([[Integer(1),Integer(2),Integer(3)],[Integer(3)]], Integer(2),␣
↪[Integer(0),Integer(3)])
>>> list(ST)
[[[-2, -2, -2], [2]]]
>>> ST = StrongTableaux.marked_given_unmarked_and_weight_
```

(continues on next page)

(continued from previous page)

```
↪iterator([[Integer(1),Integer(2),Integer(3)],[Integer(3)]], Integer(2),␣
↪[Integer(1),Integer(2)])
>>> list(ST)
[[[-1, -2, -2], [2]]]
>>> ST = StrongTableaux.marked_given_unmarked_and_weight_
↪iterator([[Integer(1),Integer(2),Integer(3)],[Integer(3)]], Integer(2),␣
↪[Integer(2),Integer(1)])
>>> list(ST)
[[[-1, -1, 2], [-2]], [[-1, -1, -2], [2]]]
>>> ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[None,␣
↪None, Integer(1), Integer(2), Integer(4)], [Integer(2), Integer(4)],␣
↪[Integer(3)]], Integer(3), [Integer(3),Integer(1)])
>>> list(ST)
[]
>>> ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[None,␣
↪None, Integer(1), Integer(2), Integer(4)], [Integer(2), Integer(4)],␣
↪[Integer(3)]], Integer(3), [Integer(2),Integer(2)])
>>> list(ST)
[[[None, None, -1, -1, 2], [1, -2], [-2]],
 [[None, None, -1, -1, -2], [1, 2], [-2]]]
```

options = Current options for Tableaux - ascii_art: repr - convention: English - display: list - latex: diagram

outer_shape()

Return the outer shape of the class of strong tableaux.

OUTPUT: a $k + 1$-core

EXAMPLES:

```
sage: StrongTableaux( 2, [3,1] ).outer_shape()
[3, 1]
sage: type(StrongTableaux( 2, [3,1] ).outer_shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: StrongTableaux( 4, [[2,1], [1]] ).outer_shape()
[2, 1]
```

```
>>> from sage.all import *
>>> StrongTableaux( Integer(2), [Integer(3),Integer(1)] ).outer_shape()
[3, 1]
>>> type(StrongTableaux( Integer(2), [Integer(3),Integer(1)] ).outer_shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
>>> StrongTableaux( Integer(4), [[Integer(2),Integer(1)], [Integer(1)]] ).
↪outer_shape()
[2, 1]
```

shape()

Return the shape of `self`.

If the `self` has an inner shape return a pair consisting of an inner and an outer shape. If the inner shape is empty then return only the outer shape.

OUTPUT: a $k + 1$-core or a pair of $k + 1$-cores

EXAMPLES:

```
sage: StrongTableaux( 2, [3,1] ).shape()
[3, 1]
sage: type(StrongTableaux( 2, [3,1] ).shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: StrongTableaux( 4, [[2,1], [1]] ).shape()
([2, 1], [1])
```

```
>>> from sage.all import *
>>> StrongTableaux( Integer(2), [Integer(3),Integer(1)] ).shape()
[3, 1]
>>> type(StrongTableaux( Integer(2), [Integer(3),Integer(1)] ).shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
>>> StrongTableaux( Integer(4), [[Integer(2),Integer(1)], [Integer(1)]] ).
→shape()
([2, 1], [1])
```

classmethod standard_marked_iterator(*k*, *size*, *outer_shape=None*, *inner_shape=[]*)

An iterator for generating standard strong marked tableaux.

An iterator which generates all standard marked k-tableaux of a given `size` which are contained in `outer_shape` and contain the `inner_shape`. If `outer_shape` is `None` then there is no restriction on the shape of the tableaux which are created.

INPUT:

- `k` – positive integer
- `size` – positive integer
- `outer_shape` – list which is a $k + 1$-core (default: `None`)
- `inner_shape` – list which is a $k + 1$-core (default: `[]`)

OUTPUT:

- an iterator which returns the standard marked tableaux with `size` cells and that are contained in `outer_shape` and contain `inner_shape`

EXAMPLES:

```
sage: list(StrongTableaux.standard_marked_iterator(2, 3))
[[[-1, -2, 3], [-3]], [[-1, -2, -3], [3]], [[-1, -2], [-3], [3]], [[-1, 3, -
→3], [-2]], [[-1, 3], [-2], [-3]], [[-1, -3], [-2], [3]]]
sage: list(StrongTableaux.standard_marked_iterator(2, 1, inner_shape=[1,1]))
[[[None, 1, -1], [None]], [[None, 1], [None], [-1]], [[None, -1], [None],
→[1]]]
sage: len(list(StrongTableaux.standard_marked_iterator(4,4)))
10
sage: len(list(StrongTableaux.standard_marked_iterator(4,6)))
140
sage: len(list(StrongTableaux.standard_marked_iterator(4,4, inner_shape=[2,
→2])))
200
sage: len(list(StrongTableaux.standard_marked_iterator(4,4, outer_shape=[5,2,
→2,1], inner_shape=[2,2])))
24
```

```
>>> from sage.all import *
>>> list(StrongTableaux.standard_marked_iterator(Integer(2), Integer(3)))
[[[-1, -2, 3], [-3]], [[-1, -2, -3], [3]], [[-1, -2], [-3], [3]], [[-1, 3, -
↪3], [-2]], [[-1, 3], [-2], [-3]], [[-1, -3], [-2], [3]]]
>>> list(StrongTableaux.standard_marked_iterator(Integer(2), Integer(1),␣
↪inner_shape=[Integer(1),Integer(1)]))
[[[None, 1, -1], [None]], [[None, 1], [None], [-1]], [[None, -1], [None],␣
↪[1]]]
>>> len(list(StrongTableaux.standard_marked_iterator(Integer(4),Integer(4))))
10
>>> len(list(StrongTableaux.standard_marked_iterator(Integer(4),Integer(6))))
140
>>> len(list(StrongTableaux.standard_marked_iterator(Integer(4),Integer(4),␣
↪inner_shape=[Integer(2),Integer(2)])))
200
>>> len(list(StrongTableaux.standard_marked_iterator(Integer(4),Integer(4),␣
↪outer_shape=[Integer(5),Integer(2),Integer(2),Integer(1)], inner_
↪shape=[Integer(2),Integer(2)])))
24
```

classmethod standard_unmarked_iterator(*k*, *size*, *outer_shape=None*, *inner_shape=[]*)

An iterator for standard unmarked strong tableaux.

An iterator which generates all unmarked tableaux of a given `size` which are contained in `outer_shape` and which contain the `inner_shape`.

These are built recursively by building all standard marked strong tableaux of size `size` -1 and adding all possible covers.

If `outer_shape` is `None` then there is no restriction on the shape of the tableaux which are created.

INPUT:

- `k`, `size` – positive integers
- `outer_shape` – list representing a $k+1$-core (default: `None`)
- `inner_shape` – list representing a $k+1$-core (default: `[]`)

OUTPUT:

- an iterator which lists all standard strong unmarked tableaux with `size` cells and which are contained in `outer_shape` and contain `inner_shape`

EXAMPLES:

```
sage: list(StrongTableaux.standard_unmarked_iterator(2, 3))
[[[1, 2, 3], [3]], [[1, 2], [3], [3]], [[1, 3, 3], [2]], [[1, 3], [2], [3]]]
sage: list(StrongTableaux.standard_unmarked_iterator(2, 1, inner_shape=[1,1]))
[[[None, 1, 1], [None]], [[None, 1], [None], [1]]]
sage: len(list(StrongTableaux.standard_unmarked_iterator(4,4)))
10
sage: len(list(StrongTableaux.standard_unmarked_iterator(4,6)))
98
sage: len(list(StrongTableaux.standard_unmarked_iterator(4,4, inner_shape=[2,
↪2])))
92
sage: len(list(StrongTableaux.standard_unmarked_iterator(4,4, outer_shape=[5,
```

(continues on next page)

(continued from previous page)

```
↪2,2,1], inner_shape=[2,2])))
10
```

```
>>> from sage.all import *
>>> list(StrongTableaux.standard_unmarked_iterator(Integer(2), Integer(3)))
[[[1, 2, 3], [3]], [[1, 2], [3], [3]], [[1, 3, 3], [2]], [[1, 3], [2], [3]]]
>>> list(StrongTableaux.standard_unmarked_iterator(Integer(2), Integer(1),␣
↪inner_shape=[Integer(1),Integer(1)]))
[[[None, 1, 1], [None]], [[None, 1], [None], [1]]]
>>> len(list(StrongTableaux.standard_unmarked_iterator(Integer(4),
↪Integer(4))))
10
>>> len(list(StrongTableaux.standard_unmarked_iterator(Integer(4),
↪Integer(6))))
98
>>> len(list(StrongTableaux.standard_unmarked_iterator(Integer(4),Integer(4),␣
↪inner_shape=[Integer(2),Integer(2)])))
92
>>> len(list(StrongTableaux.standard_unmarked_iterator(Integer(4),Integer(4),␣
↪outer_shape=[Integer(5),Integer(2),Integer(2),Integer(1)], inner_
↪shape=[Integer(2),Integer(2)])))
10
```

classmethod transpositions_to_standard_strong(*transeq*, *k*, *emptyTableau=[]*)

Return a strong tableau corresponding to a sequence of transpositions.

This method returns the action by left multiplication on the empty strong tableau by transpositions specified by `transeq`.

INPUT:

- `transeq` – a sequence of transpositions t_{ij} (a list of pairs)
- `emptyTableau` – (default: `[]`) an empty list or a skew strong tableau possibly consisting of `None` entries

OUTPUT: a `StrongTableau` object

EXAMPLES:

```
sage: StrongTableaux.transpositions_to_standard_strong([[0,1]], 2)
[[-1]]
sage: StrongTableaux.transpositions_to_standard_strong([[-2,-1], [2,3]], 2,␣
↪[[None, None]])
[[None, None, -1], [1], [-2]]
sage: StrongTableaux.transpositions_to_standard_strong([[2, 3], [1, 2], [0,␣
↪1]], 2)
[[-1, -2, -3], [3]]
sage: StrongTableaux.transpositions_to_standard_strong([[-1, 0], [1, 2], [0,␣
↪1]], 2)
[[-1, -2, 3], [-3]]
sage: StrongTableaux.transpositions_to_standard_strong([[3, 4], [-1, 0], [1,␣
↪2]], 2, [[None]])
[[None, -1, 2, -3], [-2, 3]]
```

```
>>> from sage.all import *
>>> StrongTableaux.transpositions_to_standard_strong([[Integer(0),
↪Integer(1)]], Integer(2))
[[-1]]
>>> StrongTableaux.transpositions_to_standard_strong([[-Integer(2),-
↪Integer(1)], [Integer(2),Integer(3)]], Integer(2), [[None, None]])
[[None, None, -1], [1], [-2]]
>>> StrongTableaux.transpositions_to_standard_strong([[Integer(2),
↪Integer(3)], [Integer(1), Integer(2)], [Integer(0), Integer(1)]],
↪Integer(2))
[[-1, -2, -3], [3]]
>>> StrongTableaux.transpositions_to_standard_strong([[-Integer(1),
↪Integer(0)], [Integer(1), Integer(2)], [Integer(0), Integer(1)]],
↪Integer(2))
[[-1, -2, 3], [-3]]
>>> StrongTableaux.transpositions_to_standard_strong([[Integer(3),
↪Integer(4)], [-Integer(1), Integer(0)], [Integer(1), Integer(2)]],
↪Integer(2), [[None]])
[[None, -1, 2, -3], [-2, 3]]
```

`sage.combinat.k_tableau.`**`WeakTableau`**`(`*t*, *k*, *inner_shape=[]*, *representation='core'*`)`

This is the dispatcher method for the element class of weak k-tableaux.

Standard weak k-tableaux correspond to saturated chains in the weak order. There are three formulations of weak tableaux, one in terms of cores, one in terms of k-bounded partitions, and one in terms of factorizations of affine Grassmannian elements. For semistandard weak k-tableaux, all letters of the same value have to satisfy the conditions of a horizontal strip. In the affine Grassmannian formulation this means that all factors are cyclically decreasing elements. For more information, see for example [LLMSSZ2013].

INPUT:

- `t` – a weak k-tableau in the specified representation:
  - for the 'core' representation `t` is a list of lists where each subtableaux should have a $k + 1$-core shape; `None` is allowed as an entry for skew weak k-tableaux
  - for the 'bounded' representation `t` is a list of lists where each subtableaux should have a k-bounded shape; `None` is allowed as an entry for skew weak k-tableaux
  - for the 'factorized_permutation' representation `t` is either a list of cyclically decreasing Weyl group elements or a list of reduced words of cyclically decreasing Weyl group elements; to indicate a skew tableau in this representation, `inner_shape` should be the inner shape as a $(k + 1)$-core
- `k` – positive integer
- `inner_shape` – this entry is only relevant for the 'factorized_permutation' representation and specifies the inner shape in case the tableau is skew (default: `[]`)
- `representation` – 'core', 'bounded', or 'factorized_permutation' (default: `'core'`)

EXAMPLES:

Here is an example of a weak 3-tableau in core representation:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.shape()
[5, 2, 1]
sage: t.weight()
```

(continues on next page)

(continued from previous page)

```
(2, 2, 2)
sage: type(t)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class'>
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2), Integer(3)],
↪ [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> t.shape()
[5, 2, 1]
>>> t.weight()
(2, 2, 2)
>>> type(t)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class'>
```

And now we give a skew weak 3-tableau in core representation:

```
sage: ts = WeakTableau([[None, 1, 1, 2, 2], [None, 2], [1]], 3)
sage: ts.shape()
([5, 2, 1], [1, 1])
sage: ts.weight()
(2, 2)
sage: type(ts)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class'>
```

```
>>> from sage.all import *
>>> ts = WeakTableau([[None, Integer(1), Integer(1), Integer(2), Integer(2)],
↪[None, Integer(2)], [Integer(1)]], Integer(3))
>>> ts.shape()
([5, 2, 1], [1, 1])
>>> ts.weight()
(2, 2)
>>> type(ts)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class'>
```

Next we create the analogue of the first example in bounded representation:

```
sage: tb = WeakTableau([[1,1,2],[2,3],[3]], 3, representation='bounded')
sage: tb.shape()
[3, 2, 1]
sage: tb.weight()
(2, 2, 2)
sage: type(tb)
<class 'sage.combinat.k_tableau.WeakTableaux_bounded_with_category.element_class'>
sage: tb.to_core_tableau()
[[1, 1, 2, 2, 3], [2, 3], [3]]
sage: t == tb.to_core_tableau()
True
```

```
>>> from sage.all import *
>>> tb = WeakTableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),Integer(3)],
↪[Integer(3)]], Integer(3), representation='bounded')
>>> tb.shape()
```

(continues on next page)

(continued from previous page)

```
[3, 2, 1]
>>> tb.weight()
(2, 2, 2)
>>> type(tb)
<class 'sage.combinat.k_tableau.WeakTableaux_bounded_with_category.element_class'>
>>> tb.to_core_tableau()
[[1, 1, 2, 2, 3], [2, 3], [3]]
>>> t == tb.to_core_tableau()
True
```

And the analogue of the skew example in bounded representation:

```
sage: tbs = WeakTableau([[None, 1, 2], [None, 2], [1]], 3, representation =
↪"bounded")
sage: tbs.shape()
([3, 2, 1], [1, 1])
sage: tbs.weight()
(2, 2)
sage: tbs.to_core_tableau()
[[None, 1, 1, 2, 2], [None, 2], [1]]
sage: ts.to_bounded_tableau() == tbs
True
```

```
>>> from sage.all import *
>>> tbs = WeakTableau([[None, Integer(1), Integer(2)], [None, Integer(2)],
↪[Integer(1)]], Integer(3), representation = "bounded")
>>> tbs.shape()
([3, 2, 1], [1, 1])
>>> tbs.weight()
(2, 2)
>>> tbs.to_core_tableau()
[[None, 1, 1, 2, 2], [None, 2], [1]]
>>> ts.to_bounded_tableau() == tbs
True
```

Finally we do the same examples for the factorized permutation representation:

```
sage: tf = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = "factorized_
↪permutation")
sage: tf.shape()
[5, 2, 1]
sage: tf.weight()
(2, 2, 2)
sage: type(tf)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_category.
↪element_class'>
sage: tf.to_core_tableau() == t
True

sage: tfs = WeakTableau([[0,3],[2,1]], 3, inner_shape = [1,1], representation =
↪'factorized_permutation')
sage: tfs.shape()
```

(continues on next page)

(continued from previous page)

```
([5, 2, 1], [1, 1])
sage: tfs.weight()
(2, 2)
sage: type(tfs)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_category.
↪element_class'>
sage: tfs.to_core_tableau()
[[None, 1, 1, 2, 2], [None, 2], [1]]
```

```
>>> from sage.all import *
>>> tf = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)],[Integer(1),
↪Integer(0)]], Integer(3), representation = "factorized_permutation")
>>> tf.shape()
[5, 2, 1]
>>> tf.weight()
(2, 2, 2)
>>> type(tf)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_category.
↪element_class'>
>>> tf.to_core_tableau() == t
True

>>> tfs = WeakTableau([[Integer(0),Integer(3)],[Integer(2),Integer(1)]],
↪Integer(3), inner_shape = [Integer(1),Integer(1)], representation = 'factorized_
↪permutation')
>>> tfs.shape()
([5, 2, 1], [1, 1])
>>> tfs.weight()
(2, 2)
>>> type(tfs)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_category.
↪element_class'>
>>> tfs.to_core_tableau()
[[None, 1, 1, 2, 2], [None, 2], [1]]
```

Another way to pass from one representation to another is as follows:

```
sage: ts
[[None, 1, 1, 2, 2], [None, 2], [1]]
sage: ts.parent()._representation
'core'
sage: ts.representation('bounded')
[[None, 1, 2], [None, 2], [1]]
```

```
>>> from sage.all import *
>>> ts
[[None, 1, 1, 2, 2], [None, 2], [1]]
>>> ts.parent()._representation
'core'
>>> ts.representation('bounded')
[[None, 1, 2], [None, 2], [1]]
```

To test whether a given semistandard tableau is a weak k-tableau in the bounded representation, one can ask:

```
sage: t = Tableau([[1,1,2],[2,3],[3]])
sage: t.is_k_tableau(3)
True
sage: t = SkewTableau([[None, 1, 2], [None, 2], [1]])
sage: t.is_k_tableau(3)
True
sage: t = SkewTableau([[None, 1, 1], [None, 2], [2]])
sage: t.is_k_tableau(3)
False
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),Integer(3)],
↪[Integer(3)]])
>>> t.is_k_tableau(Integer(3))
True
>>> t = SkewTableau([[None, Integer(1), Integer(2)], [None, Integer(2)],␣
↪[Integer(1)]])
>>> t.is_k_tableau(Integer(3))
True
>>> t = SkewTableau([[None, Integer(1), Integer(1)], [None, Integer(2)],␣
↪[Integer(2)]])
>>> t.is_k_tableau(Integer(3))
False
```

class sage.combinat.k_tableau.**WeakTableau_abstract**

Bases: `ClonableList`

Abstract class for the various element classes of WeakTableau.

intermediate_shapes()

Return the intermediate shapes of `self`.

A (skew) tableau with letters $1, 2, \ldots, \ell$ can be viewed as a sequence of shapes, where the i-th shape is given by the shape of the subtableau on letters $1, 2, \ldots, i$. The output is the list of these shapes.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.intermediate_shapes()
[[], [2], [4, 1], [5, 2, 1]]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.intermediate_shapes()
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]

sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.intermediate_shapes()
[[], [3], [3, 2], [3, 2, 1]]

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation =
↪'bounded')
sage: t.intermediate_shapes()
[[2], [3], [3, 1], [3, 1, 1], [3, 2, 1]]
```

(continues on next page)

(continued from previous page)

```
sage: t = WeakTableau([[0],[3],[2],[3]], 3, inner_shape = [2], representation␣
↪= 'factorized_permutation')
sage: t.intermediate_shapes()
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]],Integer(3))
>>> t.intermediate_shapes()
[[], [2], [4, 1], [5, 2, 1]]

>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> t.intermediate_shapes()
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]

>>> t = WeakTableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.intermediate_shapes()
[[], [3], [3, 2], [3, 2, 1]]

>>> t = WeakTableau([[None, None, Integer(1)], [Integer(2), Integer(4)],␣
↪[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.intermediate_shapes()
[[2], [3], [3, 1], [3, 1, 1], [3, 2, 1]]

>>> t = WeakTableau([[Integer(0)],[Integer(3)],[Integer(2)],[Integer(3)]],␣
↪Integer(3), inner_shape = [Integer(2)], representation = 'factorized_
↪permutation')
>>> t.intermediate_shapes()
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]
```

pp()

Return a pretty print string of the tableau.

EXAMPLES:

```
sage: t = WeakTableau([[None, 1, 1, 2, 2], [None, 2], [1]], 3)
sage: t.pp()
.  1  1  2  2
.  2
1
sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation =
↪'factorized_permutation')
sage: t.pp()
[s2*s0, s3*s2]
```

```
>>> from sage.all import *
>>> t = WeakTableau([[None, Integer(1), Integer(1), Integer(2), Integer(2)],␣
↪[None, Integer(2)], [Integer(1)]], Integer(3))
>>> t.pp()
```

(continues on next page)

(continued from previous page)

```
.  1  1  2  2
.  2
1
>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)]],␣
↪Integer(3), inner_shape = [Integer(2)], representation = 'factorized_
↪permutation')
>>> t.pp()
[s2*s0, s3*s2]
```

representation(*representation='core'*)

Return the analogue of `self` in the specified representation.

INPUT:

- `representation` – 'core', 'bounded', or 'factorized_permutation' (default: `'core'`)

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4,␣
↪7], [5, 6], [6], [7]], 4)
sage: t.parent()._representation
'core'
sage: t.representation('bounded')
[[1, 1, 2, 4], [2, 3, 5], [3, 4], [5, 6], [6], [7]]
sage: t.representation('factorized_permutation')
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]

sage: tb = WeakTableau([[1, 1, 2, 4], [2, 3, 5], [3, 4], [5, 6], [6], [7]], 4,
↪ representation = 'bounded')
sage: tb.parent()._representation
'bounded'
sage: tb.representation('core') == t
True
sage: tb.representation('factorized_permutation')
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]

sage: tp = WeakTableau([[0],[3,1],[2,1],[0,4],[3,0],[4,2],[1,0]], 4,␣
↪representation = 'factorized_permutation')
sage: tp.parent()._representation
'factorized_permutation'
sage: tp.representation('core') == t
True
sage: tp.representation('bounded') == tb
True
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(3),␣
↪Integer(4), Integer(4), Integer(5), Integer(5), Integer(6)], [Integer(2),␣
↪Integer(3), Integer(5), Integer(5), Integer(6)], [Integer(3), Integer(4),␣
↪Integer(7)], [Integer(5), Integer(6)], [Integer(6)], [Integer(7)]],␣
↪Integer(4))
>>> t.parent()._representation
'core'
```

(continues on next page)

(continued from previous page)

```
>>> t.representation('bounded')
[[1, 1, 2, 4], [2, 3, 5], [3, 4], [5, 6], [6], [7]]
>>> t.representation('factorized_permutation')
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]

>>> tb = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(4)],
↪[Integer(2), Integer(3), Integer(5)], [Integer(3), Integer(4)], [Integer(5),
↪ Integer(6)], [Integer(6)], [Integer(7)]], Integer(4), representation =
↪'bounded')
>>> tb.parent()._representation
'bounded'
>>> tb.representation('core') == t
True
>>> tb.representation('factorized_permutation')
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]

>>> tp = WeakTableau([[Integer(0)],[Integer(3),Integer(1)],[Integer(2),
↪Integer(1)],[Integer(0),Integer(4)],[Integer(3),Integer(0)],[Integer(4),
↪Integer(2)],[Integer(1),Integer(0)]], Integer(4), representation =
↪'factorized_permutation')
>>> tp.parent()._representation
'factorized_permutation'
>>> tp.representation('core') == t
True
>>> tp.representation('bounded') == tb
True
```

shape()

Return the shape of `self`.

When the tableau is straight, the outer shape is returned. When the tableau is skew, the tuple of the outer and inner shape is returned.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.shape()
[5, 2, 1]
sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.shape()
([5, 2, 1], [2])

sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.shape()
[3, 2, 1]
sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation =
↪'bounded')
sage: t.shape()
([3, 2, 1], [2])

sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_
↪permutation')
sage: t.shape()
```

(continues on next page)

(continued from previous page)

```
[5, 2, 1]
sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation =
↪'factorized_permutation')
sage: t.shape()
([5, 2, 1], [2])
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> t.shape()
[5, 2, 1]
>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> t.shape()
([5, 2, 1], [2])

>>> t = WeakTableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.shape()
[3, 2, 1]
>>> t = WeakTableau([[None, None, Integer(1)], [Integer(2), Integer(4)],␣
↪[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.shape()
([3, 2, 1], [2])

>>> t = WeakTableau([[Integer(2)],[Integer(0),Integer(3)],[Integer(2),
↪Integer(1),Integer(0)]], Integer(3), representation = 'factorized_
↪permutation')
>>> t.shape()
[5, 2, 1]
>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)]],␣
↪Integer(3), inner_shape = [Integer(2)], representation = 'factorized_
↪permutation')
>>> t.shape()
([5, 2, 1], [2])
```

size()

Return the size of the shape of `self`.

In the bounded representation, the size of the shape is the number of boxes in the outer shape minus the number of boxes in the inner shape. For the core and factorized permutation representation, the size is the length of the outer shape minus the length of the inner shape.

See also

sage.combinat.core.Core.length()

EXAMPLES:

```
sage: t = WeakTableau([[None, 1, 1, 2, 2], [None, 2], [1]], 3)
sage: t.shape()
```

(continues on next page)

(continued from previous page)

```
([5, 2, 1], [1, 1])
sage: t.size()
4
sage: t = WeakTableau([[1,1,2],[2,3],[3]], 3, representation='bounded')
sage: t.shape()
[3, 2, 1]
sage: t.size()
6
```

```
>>> from sage.all import *
>>> t = WeakTableau([[None, Integer(1), Integer(1), Integer(2), Integer(2)],
↪[None, Integer(2)], [Integer(1)]], Integer(3))
>>> t.shape()
([5, 2, 1], [1, 1])
>>> t.size()
4
>>> t = WeakTableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),
↪Integer(3)],[Integer(3)]], Integer(3), representation='bounded')
>>> t.shape()
[3, 2, 1]
>>> t.size()
6
```

weight()

Return the weight of `self`.

The weight is a tuple whose i-th entry is the number of labels i in the bounded representation of `self`.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.weight()
(2, 2, 2)
sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.weight()
(1, 1, 1, 1)
sage: t = WeakTableau([[None,2,3],[3]],2)
sage: t.weight()
(0, 1, 1)

sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.weight()
(3, 2, 1)
sage: t = WeakTableau([[1,1,2],[2,3],[3]], 3, representation = 'bounded')
sage: t.weight()
(2, 2, 2)
sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation =
↪'bounded')
sage: t.weight()
(1, 1, 1, 1)

sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_
↪permutation')
```

(continues on next page)

(continued from previous page)

```
sage: t.weight()
(3, 2, 1)
sage: t = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = 'factorized_
↪permutation')
sage: t.weight()
(2, 2, 2)
sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation =
↪'factorized_permutation')
sage: t.weight()
(2, 2)
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> t.weight()
(2, 2, 2)
>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> t.weight()
(1, 1, 1, 1)
>>> t = WeakTableau([[None,Integer(2),Integer(3)],[Integer(3)]],Integer(2))
>>> t.weight()
(0, 1, 1)

>>> t = WeakTableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.weight()
(3, 2, 1)
>>> t = WeakTableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),
↪Integer(3)],[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.weight()
(2, 2, 2)
>>> t = WeakTableau([[None, None, Integer(1)], [Integer(2), Integer(4)],␣
↪[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.weight()
(1, 1, 1, 1)

>>> t = WeakTableau([[Integer(2)],[Integer(0),Integer(3)],[Integer(2),
↪Integer(1),Integer(0)]], Integer(3), representation = 'factorized_
↪permutation')
>>> t.weight()
(3, 2, 1)
>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)],
↪[Integer(1),Integer(0)]], Integer(3), representation = 'factorized_
↪permutation')
>>> t.weight()
(2, 2, 2)
>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)]],␣
↪Integer(3), inner_shape = [Integer(2)], representation = 'factorized_
↪permutation')
>>> t.weight()
(2, 2)
```

class sage.combinat.k_tableau.**WeakTableau_bounded**(*parent*, *t*)

Bases: *WeakTableau_abstract*

A (skew) weak k-tableau represented in terms of k-bounded partitions.

check()

Check that self is a valid weak k-tableau.

EXAMPLES:

```
sage: t = WeakTableau([[1,1],[2]], 2, representation = 'bounded')
sage: t.check()

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation =
↪'bounded')
sage: t.check()
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1),Integer(1)],[Integer(2)]], Integer(2),␣
↪representation = 'bounded')
>>> t.check()

>>> t = WeakTableau([[None, None, Integer(1)], [Integer(2), Integer(4)],␣
↪[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.check()
```

classmethod from_core_tableau(*t*, *k*)

Construct weak k-bounded tableau from in k-core tableau.

EXAMPLES:

```
sage: from sage.combinat.k_tableau import WeakTableau_bounded
sage: WeakTableau_bounded.from_core_tableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
[[1, 1, 2], [2, 3], [3]]

sage: WeakTableau_bounded.from_core_tableau([[None, None, 2, 3, 4], [1, 4],␣
↪[2]], 3)
[[None, None, 3], [1, 4], [2]]

sage: WeakTableau_bounded.from_core_tableau([[None,2,3],[3]], 2)
[[None, 2], [3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.k_tableau import WeakTableau_bounded
>>> WeakTableau_bounded.from_core_tableau([[Integer(1), Integer(1),␣
↪Integer(2), Integer(2), Integer(3)], [Integer(2), Integer(3)],␣
↪[Integer(3)]], Integer(3))
[[1, 1, 2], [2, 3], [3]]

>>> WeakTableau_bounded.from_core_tableau([[None, None, Integer(2),␣
↪Integer(3), Integer(4)], [Integer(1), Integer(4)], [Integer(2)]],␣
↪Integer(3))
[[None, None, 3], [1, 4], [2]]
```

(continues on next page)

(continued from previous page)

```
>>> WeakTableau_bounded.from_core_tableau([[None,Integer(2),Integer(3)],
↪[Integer(3)]], Integer(2))
[[None, 2], [3]]
```

k_charge(*algorithm='I'*)

Return the k-charge of `self`.

INPUT:

- `algorithm` – (default: `'I'`) if "I", computes k-charge using the I algorithm, otherwise uses the J-algorithm

OUTPUT: nonnegative integer

For the definition of k-charge and the various algorithms to compute it see Section 3.3 of [LLMSSZ2013].

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2], [2, 3], [3]], 3, representation = 'bounded')
sage: t.k_charge()
2
sage: t = WeakTableau([[1, 3, 5], [2, 6], [4]], 3, representation = 'bounded')
sage: t.k_charge()
8
sage: t = WeakTableau([[1, 1, 2, 4], [2, 3, 5], [3, 4], [5, 6], [6], [7]], 4,␣
↪representation = 'bounded')
sage: t.k_charge()
12
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2)], [Integer(2),␣
↪Integer(3)], [Integer(3)]], Integer(3), representation = 'bounded')
>>> t.k_charge()
2
>>> t = WeakTableau([[Integer(1), Integer(3), Integer(5)], [Integer(2),␣
↪Integer(6)], [Integer(4)]], Integer(3), representation = 'bounded')
>>> t.k_charge()
8
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(4)],␣
↪[Integer(2), Integer(3), Integer(5)], [Integer(3), Integer(4)], [Integer(5),
↪ Integer(6)], [Integer(6)], [Integer(7)]], Integer(4), representation =
↪'bounded')
>>> t.k_charge()
12
```

shape_bounded()

Return the shape of `self` as k-bounded partition.

When the tableau is straight, the outer shape is returned as a k-bounded partition. When the tableau is skew, the tuple of the outer and inner shape is returned as k-bounded partitions.

EXAMPLES:

```
sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.shape_bounded()
```

(continues on next page)

(continued from previous page)

```
[3, 2, 1]

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation =
↪'bounded')
sage: t.shape_bounded()
([3, 2, 1], [2])
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.shape_bounded()
[3, 2, 1]

>>> t = WeakTableau([[None, None, Integer(1)], [Integer(2), Integer(4)],␣
↪[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.shape_bounded()
([3, 2, 1], [2])
```

shape_core()

Return the shape of `self` as $(k+1)$-core.

When the tableau is straight, the outer shape is returned as a $(k+1)$-core. When the tableau is skew, the tuple of the outer and inner shape is returned as $(k+1)$-cores.

EXAMPLES:

```
sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.shape_core()
[5, 2, 1]

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation =
↪'bounded')
sage: t.shape_core()
([5, 2, 1], [2])
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.shape_core()
[5, 2, 1]

>>> t = WeakTableau([[None, None, Integer(1)], [Integer(2), Integer(4)],␣
↪[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.shape_core()
([5, 2, 1], [2])
```

to_core_tableau()

Return the weak k-tableau `self` where the shape of each restricted tableau is a $(k+1)$-core.

EXAMPLES:

```
sage: t = WeakTableau([[1,1,2,4],[2,3,5],[3,4],[5,6],[6],[7]], 4,␣
↪representation = 'bounded')
```

(continues on next page)

(continued from previous page)

```
sage: c = t.to_core_tableau(); c
[[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6], [7]]
sage: type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class
→'>
sage: t = WeakTableau([], 4, representation = 'bounded')
sage: t.to_core_tableau()
[]

sage: from sage.combinat.k_tableau import WeakTableau_bounded
sage: t = WeakTableau([[1,1,2],[2,3],[3]], 3, representation = 'bounded')
sage: WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),3)
[[1, 1, 2], [2, 3], [3]]
sage: t == WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),3)
True

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation =
→'bounded')
sage: t.to_core_tableau()
[[None, None, 1, 2, 4], [2, 4], [3]]
sage: t == WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),3)
True
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1),Integer(1),Integer(2),Integer(4)],
→[Integer(2),Integer(3),Integer(5)],[Integer(3),Integer(4)],[Integer(5),
→Integer(6)],[Integer(6)],[Integer(7)]], Integer(4), representation =
→'bounded')
>>> c = t.to_core_tableau(); c
[[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6], [7]]
>>> type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class
→'>
>>> t = WeakTableau([], Integer(4), representation = 'bounded')
>>> t.to_core_tableau()
[]

>>> from sage.combinat.k_tableau import WeakTableau_bounded
>>> t = WeakTableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),
→Integer(3)],[Integer(3)]], Integer(3), representation = 'bounded')
>>> WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),Integer(3))
[[1, 1, 2], [2, 3], [3]]
>>> t == WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),Integer(3))
True

>>> t = WeakTableau([[None, None, Integer(1)], [Integer(2), Integer(4)],␣
→[Integer(3)]], Integer(3), representation = 'bounded')
>>> t.to_core_tableau()
[[None, None, 1, 2, 4], [2, 4], [3]]
>>> t == WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),Integer(3))
True
```

class sage.combinat.k_tableau.**WeakTableau_core**(*parent*, *t*)

Bases: `WeakTableau_abstract`

A (skew) weak k-tableau represented in terms of $(k + 1)$-cores.

check()

Check that `self` is a valid weak k-tableau.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2], [2]], 2)
sage: t.check()
sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.check()
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2)], [Integer(2)]],␣
↪Integer(2))
>>> t.check()
>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> t.check()
```

dictionary_of_coordinates_at_residues(*v*)

Return a dictionary assigning to all residues of `self` with label `v` a list of cells with the given residue.

INPUT:

- `v` – a label of a cell in `self`

OUTPUT: dictionary assigning coordinates in `self` to residues

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.dictionary_of_coordinates_at_residues(3)
{0: [(0, 4), (1, 1)], 2: [(2, 0)]}

sage: t = WeakTableau([[None, None, 1, 1, 4], [1, 4], [3]], 3)
sage: t.dictionary_of_coordinates_at_residues(1)
{2: [(0, 2)], 3: [(0, 3), (1, 0)]}

sage: t = WeakTableau([], 3)
sage: t.dictionary_of_coordinates_at_residues(1)
{}
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]],Integer(3))
>>> t.dictionary_of_coordinates_at_residues(Integer(3))
{0: [(0, 4), (1, 1)], 2: [(2, 0)]}

>>> t = WeakTableau([[None, None, Integer(1), Integer(1), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(3)]], Integer(3))
>>> t.dictionary_of_coordinates_at_residues(Integer(1))
{2: [(0, 2)], 3: [(0, 3), (1, 0)]}
```

(continues on next page)

(continued from previous page)

```
>>> t = WeakTableau([], Integer(3))
>>> t.dictionary_of_coordinates_at_residues(Integer(1))
{}
```

k_charge(*algorithm='I'*)

Return the k-charge of `self`.

INPUT:

- `algorithm` – (default: `'I'`) if "I", computes k-charge using the I algorithm, otherwise uses the J-algorithm

OUTPUT: nonnegative integer

For the definition of k-charge and the various algorithms to compute it see Section 3.3 of [LLMSSZ2013].

See also:

k_charge_I() and *k_charge_J()*

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.k_charge()
2
sage: t = WeakTableau([[1, 3, 4, 5, 6], [2, 6], [4]], 3)
sage: t.k_charge()
8
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4,
→7], [5, 6], [6], [7]], 4)
sage: t.k_charge()
12
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),
→Integer(3)], [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> t.k_charge()
2
>>> t = WeakTableau([[Integer(1), Integer(3), Integer(4), Integer(5),
→Integer(6)], [Integer(2), Integer(6)], [Integer(4)]], Integer(3))
>>> t.k_charge()
8
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(3),
→Integer(4), Integer(4), Integer(5), Integer(5), Integer(6)], [Integer(2),
→Integer(3), Integer(5), Integer(5), Integer(6)], [Integer(3), Integer(4),
→Integer(7)], [Integer(5), Integer(6)], [Integer(6)], [Integer(7)]],
→Integer(4))
>>> t.k_charge()
12
```

k_charge_I()

Return the k-charge of `self` using the I-algorithm.

For the definition of k-charge and the I-algorithm see Section 3.3 of [LLMSSZ2013].

OUTPUT: nonnegative integer

See also:

k_charge() and *k_charge_J()*

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.k_charge_I()
2
sage: t = WeakTableau([[1, 3, 4, 5, 6], [2, 6], [4]], 3)
sage: t.k_charge_I()
8
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4,
→7], [5, 6], [6], [7]], 4)
sage: t.k_charge_I()
12
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),
→Integer(3)], [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> t.k_charge_I()
2
>>> t = WeakTableau([[Integer(1), Integer(3), Integer(4), Integer(5),
→Integer(6)], [Integer(2), Integer(6)], [Integer(4)]], Integer(3))
>>> t.k_charge_I()
8
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(3),
→Integer(4), Integer(4), Integer(5), Integer(5), Integer(6)], [Integer(2),
→Integer(3), Integer(5), Integer(5), Integer(6)], [Integer(3), Integer(4),
→Integer(7)], [Integer(5), Integer(6)], [Integer(6)], [Integer(7)]],
→Integer(4))
>>> t.k_charge_I()
12
```

k_charge_J()

Return the k-charge of `self` using the J-algorithm.

For the definition of k-charge and the J-algorithm see Section 3.3 of [LLMSSZ2013].

OUTPUT: nonnegative integer

See also:

k_charge() and *k_charge_I()*

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.k_charge_J()
```

(continues on next page)

(continued from previous page)

```
2
sage: t = WeakTableau([[1, 3, 4, 5, 6], [2, 6], [4]], 3)
sage: t.k_charge_J()
8
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4,␣
→7], [5, 6], [6], [7]], 4)
sage: t.k_charge_J()
12
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
→Integer(3)], [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> t.k_charge_J()
2
>>> t = WeakTableau([[Integer(1), Integer(3), Integer(4), Integer(5),␣
→Integer(6)], [Integer(2), Integer(6)], [Integer(4)]], Integer(3))
>>> t.k_charge_J()
8
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(3),␣
→Integer(4), Integer(4), Integer(5), Integer(5), Integer(6)], [Integer(2),␣
→Integer(3), Integer(5), Integer(5), Integer(6)], [Integer(3), Integer(4),␣
→Integer(7)], [Integer(5), Integer(6)], [Integer(6)], [Integer(7)]],␣
→Integer(4))
>>> t.k_charge_J()
12
```

list_of_standard_cells()

Return a list of lists of the coordinates of the standard cells of `self`.

INPUT:

- `self` – a weak k-tableau in core representation with partition weight

OUTPUT: list of lists of coordinates

> **Warning**
>
> This method currently only works for straight weak tableaux with partition weight.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.list_of_standard_cells()
[[(0, 1), (1, 0), (2, 0)], [(0, 0), (0, 2), (1, 1)]]
sage: t = WeakTableau([[1, 1, 1, 2], [2, 2, 3]], 5)
sage: t.list_of_standard_cells()
[[(0, 2), (1, 1), (1, 2)], [(0, 1), (1, 0)], [(0, 0), (0, 3)]]
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4,␣
→7], [5, 6], [6], [7]], 4)
sage: t.list_of_standard_cells()
[[(0, 1), (1, 0), (2, 0), (0, 5), (3, 0), (4, 0), (5, 0)], [(0, 0), (0, 2),␣
→(1, 1), (2, 1), (1, 2), (3, 1)]]
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> t.list_of_standard_cells()
[[(0, 1), (1, 0), (2, 0)], [(0, 0), (0, 2), (1, 1)]]
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(1), Integer(2)],␣
↪[Integer(2), Integer(2), Integer(3)]], Integer(5))
>>> t.list_of_standard_cells()
[[(0, 2), (1, 1), (1, 2)], [(0, 1), (1, 0)], [(0, 0), (0, 3)]]
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(3),␣
↪Integer(4), Integer(4), Integer(5), Integer(5), Integer(6)], [Integer(2),␣
↪Integer(3), Integer(5), Integer(5), Integer(6)], [Integer(3), Integer(4),␣
↪Integer(7)], [Integer(5), Integer(6)], [Integer(6)], [Integer(7)]],␣
↪Integer(4))
>>> t.list_of_standard_cells()
[[(0, 1), (1, 0), (2, 0), (0, 5), (3, 0), (4, 0), (5, 0)], [(0, 0), (0, 2),␣
↪(1, 1), (2, 1), (1, 2), (3, 1)]]
```

residues_of_entries(*v*)

Return a list of residues of cells of weak k-tableau `self` labeled by `v`.

INPUT:

- `v` – a label of a cell in `self`

OUTPUT: list of residues

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.residues_of_entries(1)
[0, 1]

sage: t = WeakTableau([[None, None, 1, 1, 4], [1, 4], [3]], 3)
sage: t.residues_of_entries(1)
[2, 3]
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]],Integer(3))
>>> t.residues_of_entries(Integer(1))
[0, 1]

>>> t = WeakTableau([[None, None, Integer(1), Integer(1), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(3)]], Integer(3))
>>> t.residues_of_entries(Integer(1))
[2, 3]
```

shape_bounded()

Return the shape of `self` as a k-bounded partition.

When the tableau is straight, the outer shape is returned as a k-bounded partition. When the tableau is skew, the tuple of the outer and inner shape is returned as k-bounded partitions.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.shape_bounded()
[3, 2, 1]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.shape_bounded()
([3, 2, 1], [2])
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]],Integer(3))
>>> t.shape_bounded()
[3, 2, 1]

>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> t.shape_bounded()
([3, 2, 1], [2])
```

shape_core()

Return the shape of `self` as a $(k+1)$-core.

When the tableau is straight, the outer shape is returned as a core. When the tableau is skew, the tuple of the outer and inner shape is returned as cores.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.shape_core()
[5, 2, 1]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.shape_core()
([5, 2, 1], [2])
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]],Integer(3))
>>> t.shape_core()
[5, 2, 1]

>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> t.shape_core()
([5, 2, 1], [2])
```

to_bounded_tableau()

Return the bounded representation of the weak k-tableau `self`.

Each restricted subtableau of the output is a k-bounded partition.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: c = t.to_bounded_tableau(); c
[[1, 1, 2], [2, 3], [3]]
sage: type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_bounded_with_category.element_
↪class'>

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.to_bounded_tableau()
[[None, None, 3], [1, 4], [2]]
sage: t.to_bounded_tableau().to_core_tableau() == t
True
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> c = t.to_bounded_tableau(); c
[[1, 1, 2], [2, 3], [3]]
>>> type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_bounded_with_category.element_
↪class'>

>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> t.to_bounded_tableau()
[[None, None, 3], [1, 4], [2]]
>>> t.to_bounded_tableau().to_core_tableau() == t
True
```

to_factorized_permutation_tableau()

Return the factorized permutation representation of the weak k-tableau `self`.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: c = t.to_factorized_permutation_tableau(); c
[s2*s0, s3*s2, s1*s0]
sage: type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_
↪category.element_class'>
sage: c.to_core_tableau() == t
True

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: c = t.to_factorized_permutation_tableau(); c
[s0, s3, s2, s3]
sage: c._inner_shape
[2]
sage: c.to_core_tableau() == t
True
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)], [Integer(2), Integer(3)], [Integer(3)]], Integer(3))
>>> c = t.to_factorized_permutation_tableau(); c
[s2*s0, s3*s2, s1*s0]
>>> type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_
↪category.element_class'>
>>> c.to_core_tableau() == t
True

>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> c = t.to_factorized_permutation_tableau(); c
[s0, s3, s2, s3]
>>> c._inner_shape
[2]
>>> c.to_core_tableau() == t
True
```

class sage.combinat.k_tableau.**WeakTableau_factorized_permutation**(*parent*, *t*)

Bases: *WeakTableau_abstract*

A weak (skew) k-tableau represented in terms of factorizations of affine permutations into cyclically decreasing elements.

check()

Check that self is a valid weak k-tableau.

EXAMPLES:

```
sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_
↪permutation')
sage: t.check()
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(2)],[Integer(0),Integer(3)],[Integer(2),
↪Integer(1),Integer(0)]], Integer(3), representation = 'factorized_
↪permutation')
>>> t.check()
```

classmethod from_core_tableau(*t*, *k*)

Construct weak factorized affine permutation tableau from a k-core tableau.

EXAMPLES:

```
sage: from sage.combinat.k_tableau import WeakTableau_factorized_permutation
sage: WeakTableau_factorized_permutation.from_core_tableau([[1, 1, 2, 2, 3],␣
↪[2, 3], [3]],3)
[s2*s0, s3*s2, s1*s0]
sage: WeakTableau_factorized_permutation.from_core_tableau([[1, 1, 2, 3, 4, 4,
↪ 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6], [7]], 4)
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]
sage: WeakTableau_factorized_permutation.from_core_tableau([[None, 1, 1, 2,␣
↪2], [None, 2], [1]], 3)
[s0*s3, s2*s1]
```

```
>>> from sage.all import *
>>> from sage.combinat.k_tableau import WeakTableau_factorized_permutation
>>> WeakTableau_factorized_permutation.from_core_tableau([[Integer(1),␣
→Integer(1), Integer(2), Integer(2), Integer(3)], [Integer(2), Integer(3)],␣
→[Integer(3)]],Integer(3))
[s2*s0, s3*s2, s1*s0]
>>> WeakTableau_factorized_permutation.from_core_tableau([[Integer(1),␣
→Integer(1), Integer(2), Integer(3), Integer(4), Integer(4), Integer(5),␣
→Integer(5), Integer(6)], [Integer(2), Integer(3), Integer(5), Integer(5),␣
→Integer(6)], [Integer(3), Integer(4), Integer(7)], [Integer(5), Integer(6)],
→ [Integer(6)], [Integer(7)]], Integer(4))
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]
>>> WeakTableau_factorized_permutation.from_core_tableau([[None, Integer(1),␣
→Integer(1), Integer(2), Integer(2)], [None, Integer(2)], [Integer(1)]],␣
→Integer(3))
[s0*s3, s2*s1]
```

k_charge(*algorithm='I'*)

Return the k-charge of `self`.

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: t = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = 'factorized_
→permutation')
sage: t.k_charge()
2
sage: t = WeakTableau([[0],[3],[2],[1],[3],[0]], 3, representation =
→'factorized_permutation')
sage: t.k_charge()
8
sage: t = WeakTableau([[0],[3,1],[2,1],[0,4],[3,0],[4,2],[1,0]], 4,␣
→representation = 'factorized_permutation')
sage: t.k_charge()
12
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)],
→[Integer(1),Integer(0)]], Integer(3), representation = 'factorized_
→permutation')
>>> t.k_charge()
2
>>> t = WeakTableau([[Integer(0)],[Integer(3)],[Integer(2)],[Integer(1)],
→[Integer(3)],[Integer(0)]], Integer(3), representation = 'factorized_
→permutation')
>>> t.k_charge()
8
>>> t = WeakTableau([[Integer(0)],[Integer(3),Integer(1)],[Integer(2),
→Integer(1)],[Integer(0),Integer(4)],[Integer(3),Integer(0)],[Integer(4),
→Integer(2)],[Integer(1),Integer(0)]], Integer(4), representation =
→'factorized_permutation')
>>> t.k_charge()
12
```

shape_bounded()

Return the shape of `self` as a k-bounded partition.

When the tableau is straight, the outer shape is returned as a k-bounded partition. When the tableau is skew, the tuple of the outer and inner shape is returned as k-bounded partitions.

EXAMPLES:

```
sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_
↪permutation')
sage: t.shape_bounded()
[3, 2, 1]

sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation =
↪'factorized_permutation')
sage: t.shape_bounded()
([3, 2, 1], [2])
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(2)],[Integer(0),Integer(3)],[Integer(2),
↪Integer(1),Integer(0)]], Integer(3), representation = 'factorized_
↪permutation')
>>> t.shape_bounded()
[3, 2, 1]

>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)]],␣
↪Integer(3), inner_shape = [Integer(2)], representation = 'factorized_
↪permutation')
>>> t.shape_bounded()
([3, 2, 1], [2])
```

shape_core()

Return the shape of `self` as a $(k+1)$-core.

When the tableau is straight, the outer shape is returned as a core. When the tableau is skew, the tuple of the outer and inner shape is returned as cores.

EXAMPLES:

```
sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_
↪permutation')
sage: t.shape_core()
[5, 2, 1]

sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation =
↪'factorized_permutation')
sage: t.shape()
([5, 2, 1], [2])
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(2)],[Integer(0),Integer(3)],[Integer(2),
↪Integer(1),Integer(0)]], Integer(3), representation = 'factorized_
↪permutation')
>>> t.shape_core()
[5, 2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)]],␣
↪Integer(3), inner_shape = [Integer(2)], representation = 'factorized_
↪permutation')
>>> t.shape()
([5, 2, 1], [2])
```

static straighten_input(*t*, *k*)

Straightens input.

INPUT:

- `t` – list of reduced words or a list of elements in the Weyl group of type $A_k^{(1)}$
- `k` – positive integer

EXAMPLES:

```
sage: from sage.combinat.k_tableau import WeakTableau_factorized_permutation
sage: WeakTableau_factorized_permutation.straighten_input([[2,0],[3,2],[1,0]],
↪ 3)
(s2*s0, s3*s2, s1*s0)
sage: W = WeylGroup(['A',4,1])
sage: WeakTableau_factorized_permutation.straighten_input([W.an_element(),W.
↪an_element()], 4)
(s0*s1*s2*s3*s4, s0*s1*s2*s3*s4)
```

```
>>> from sage.all import *
>>> from sage.combinat.k_tableau import WeakTableau_factorized_permutation
>>> WeakTableau_factorized_permutation.straighten_input([[Integer(2),
↪Integer(0)],[Integer(3),Integer(2)],[Integer(1),Integer(0)]], Integer(3))
(s2*s0, s3*s2, s1*s0)
>>> W = WeylGroup(['A',Integer(4),Integer(1)])
>>> WeakTableau_factorized_permutation.straighten_input([W.an_element(),W.an_
↪element()], Integer(4))
(s0*s1*s2*s3*s4, s0*s1*s2*s3*s4)
```

to_core_tableau()

Return the weak k-tableau `self` where the shape of each restricted tableau is a $(k + 1)$-core.

EXAMPLES:

```
sage: t = WeakTableau([[0], [3,1], [2,1], [0,4], [3,0], [4,2], [1,0]], 4,␣
↪representation = 'factorized_permutation'); t
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]
sage: c = t.to_core_tableau(); c
[[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6], [7]]
sage: type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class
↪'>
sage: t = WeakTableau([[]], 4, representation = 'factorized_permutation'); t
[1]
sage: t.to_core_tableau()
[]
```

(continues on next page)

(continued from previous page)

```
sage: from sage.combinat.k_tableau import WeakTableau_factorized_permutation
sage: t = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = 'factorized_
↪permutation')
sage: WeakTableau_factorized_permutation.from_core_tableau(t.to_core_
↪tableau(), 3)
[s2*s0, s3*s2, s1*s0]
sage: t == WeakTableau_factorized_permutation.from_core_tableau(t.to_core_
↪tableau(), 3)
True

sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation =
↪'factorized_permutation')
sage: t.to_core_tableau()
[[None, None, 1, 1, 2], [1, 2], [2]]
sage: t == WeakTableau_factorized_permutation.from_core_tableau(t.to_core_
↪tableau(), 3)
True
```

```
>>> from sage.all import *
>>> t = WeakTableau([[Integer(0)], [Integer(3),Integer(1)], [Integer(2),
↪Integer(1)], [Integer(0),Integer(4)], [Integer(3),Integer(0)], [Integer(4),
↪Integer(2)], [Integer(1),Integer(0)]], Integer(4), representation =
↪'factorized_permutation'); t
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]
>>> c = t.to_core_tableau(); c
[[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6], [7]]
>>> type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class
↪'>
>>> t = WeakTableau([[]], Integer(4), representation = 'factorized_permutation
↪'); t
[1]
>>> t.to_core_tableau()
[]

>>> from sage.combinat.k_tableau import WeakTableau_factorized_permutation
>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)],
↪[Integer(1),Integer(0)]], Integer(3), representation = 'factorized_
↪permutation')
>>> WeakTableau_factorized_permutation.from_core_tableau(t.to_core_tableau(),␣
↪Integer(3))
[s2*s0, s3*s2, s1*s0]
>>> t == WeakTableau_factorized_permutation.from_core_tableau(t.to_core_
↪tableau(), Integer(3))
True

>>> t = WeakTableau([[Integer(2),Integer(0)],[Integer(3),Integer(2)]],␣
↪Integer(3), inner_shape = [Integer(2)], representation = 'factorized_
↪permutation')
>>> t.to_core_tableau()
[[None, None, 1, 1, 2], [1, 2], [2]]
```

(continues on next page)

(continued from previous page)

```
>>> t == WeakTableau_factorized_permutation.from_core_tableau(t.to_core_
↪tableau(), Integer(3))
True
```

sage.combinat.k_tableau.**WeakTableaux**(*k*, *shape*, *weight*, *representation='core'*)

This is the dispatcher method for the parent class of weak k-tableaux.

INPUT:

- `k` – positive integer
- `shape` – shape of the weak k-tableaux; for the 'core' and 'factorized_permutation' representation, the shape is inputted as a $(k+1)$-core; for the 'bounded' representation, the shape is inputted as a k-bounded partition; for skew tableaux, the shape is inputted as a tuple of the outer and inner shape
- `weight` – the weight of the weak k-tableaux as a list or tuple
- `representation` – `'core'`, `'bounded'`, or `'factorized_permutation'` (default: `'core'`)

EXAMPLES:

```
sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1])
sage: T.list()
[[[1, 3, 4, 5, 6], [2, 6], [4]],
[[1, 2, 4, 5, 6], [3, 6], [4]],
[[1, 2, 3, 4, 6], [4, 6], [5]],
[[1, 2, 3, 4, 5], [4, 5], [6]]]
sage: T.cardinality()
4

sage: T = WeakTableaux(3, [[5,2,1], [2]], [1,1,1,1])
sage: T.list()
[[[None, None, 2, 3, 4], [1, 4], [2]],
[[None, None, 1, 2, 4], [2, 4], [3]],
[[None, None, 1, 2, 3], [2, 3], [4]]]

sage: T = WeakTableaux(3, [3,2,1], [1,1,1,1,1,1], representation = 'bounded')
sage: T.list()
[[[1, 3, 5], [2, 6], [4]],
[[1, 2, 5], [3, 6], [4]],
[[1, 2, 3], [4, 6], [5]],
[[1, 2, 3], [4, 5], [6]]]

sage: T = WeakTableaux(3, [[3,2,1], [2]], [1,1,1,1], representation = 'bounded')
sage: T.list()
[[[None, None, 3], [1, 4], [2]],
[[None, None, 1], [2, 4], [3]],
[[None, None, 1], [2, 3], [4]]]

sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1], representation = 'factorized_
↪permutation')
sage: T.list()
[[s0, s3, s2, s1, s3, s0],
[s0, s3, s2, s3, s1, s0],
[s0, s2, s3, s2, s1, s0],
```

(continues on next page)

(continued from previous page)

```
[s2, s0, s3, s2, s1, s0]]

sage: T = WeakTableaux(3, [[5,2,1], [2]], [1,1,1,1], representation = 'factorized_
↪permutation')
sage: T.list()
[[s0, s3, s2, s3], [s0, s2, s3, s2], [s2, s0, s3, s2]]
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(3), [Integer(5),Integer(2),Integer(1)], [Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)])
>>> T.list()
[[[1, 3, 4, 5, 6], [2, 6], [4]],
[[1, 2, 4, 5, 6], [3, 6], [4]],
[[1, 2, 3, 4, 6], [4, 6], [5]],
[[1, 2, 3, 4, 5], [4, 5], [6]]]
>>> T.cardinality()
4

>>> T = WeakTableaux(Integer(3), [[Integer(5),Integer(2),Integer(1)],␣
↪[Integer(2)]], [Integer(1),Integer(1),Integer(1),Integer(1)])
>>> T.list()
[[[None, None, 2, 3, 4], [1, 4], [2]],
[[None, None, 1, 2, 4], [2, 4], [3]],
[[None, None, 1, 2, 3], [2, 3], [4]]]

>>> T = WeakTableaux(Integer(3), [Integer(3),Integer(2),Integer(1)], [Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)], representation =
↪'bounded')
>>> T.list()
[[[1, 3, 5], [2, 6], [4]],
[[1, 2, 5], [3, 6], [4]],
[[1, 2, 3], [4, 6], [5]],
[[1, 2, 3], [4, 5], [6]]]

>>> T = WeakTableaux(Integer(3), [[Integer(3),Integer(2),Integer(1)],␣
↪[Integer(2)]], [Integer(1),Integer(1),Integer(1),Integer(1)], representation =
↪'bounded')
>>> T.list()
[[[None, None, 3], [1, 4], [2]],
[[None, None, 1], [2, 4], [3]],
[[None, None, 1], [2, 3], [4]]]

>>> T = WeakTableaux(Integer(3), [Integer(5),Integer(2),Integer(1)], [Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)], representation =
↪'factorized_permutation')
>>> T.list()
[[s0, s3, s2, s1, s3, s0],
[s0, s3, s2, s3, s1, s0],
[s0, s2, s3, s2, s1, s0],
[s2, s0, s3, s2, s1, s0]]

>>> T = WeakTableaux(Integer(3), [[Integer(5),Integer(2),Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(2)]], [Integer(1),Integer(1),Integer(1),Integer(1)], representation =
↪'factorized_permutation')
>>> T.list()
[[s0, s3, s2, s3], [s0, s2, s3, s2], [s2, s0, s3, s2]]
```

class sage.combinat.k_tableau.**WeakTableaux_abstract**

Bases: UniqueRepresentation, Parent

Abstract class for the various parent classes of WeakTableaux.

representation(*representation='core'*)

Return the analogue of `self` in the specified representation.

INPUT:

- `representation` – 'core', 'bounded', or 'factorized_permutation' (default: `'core'`)

EXAMPLES:

```
sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1])
sage: T._representation
'core'
sage: T.representation('bounded')
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1,␣
↪1, 1, 1, 1)
sage: T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight␣
↪(1, 1, 1, 1, 1, 1)

sage: T = WeakTableaux(3, [3,2,1], [1,1,1,1,1,1], representation = 'bounded')
sage: T._representation
'bounded'
sage: T.representation('core')
Core weak 3-Tableaux of (skew) core shape [5, 2, 1] and weight (1, 1, 1, 1, 1,
↪ 1)
sage: T.representation('bounded')
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1,␣
↪1, 1, 1, 1)
sage: T.representation('bounded') == T
True
sage: T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight␣
↪(1, 1, 1, 1, 1, 1)
sage: T.representation('factorized_permutation') == T
False

sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1], representation =
↪'factorized_permutation')
sage: T._representation
'factorized_permutation'
sage: T.representation('core')
Core weak 3-Tableaux of (skew) core shape [5, 2, 1] and weight (1, 1, 1, 1, 1,
↪ 1)
sage: T.representation('bounded')
```

(continues on next page)

(continued from previous page)

```
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1,␣
↪1, 1, 1, 1)
sage: T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight␣
↪(1, 1, 1, 1, 1, 1)
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(3), [Integer(5),Integer(2),Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)])
>>> T._representation
'core'
>>> T.representation('bounded')
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1,␣
↪1, 1, 1, 1)
>>> T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight␣
↪(1, 1, 1, 1, 1, 1)

>>> T = WeakTableaux(Integer(3), [Integer(3),Integer(2),Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)],␣
↪representation = 'bounded')
>>> T._representation
'bounded'
>>> T.representation('core')
Core weak 3-Tableaux of (skew) core shape [5, 2, 1] and weight (1, 1, 1, 1, 1,
↪ 1)
>>> T.representation('bounded')
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1,␣
↪1, 1, 1, 1)
>>> T.representation('bounded') == T
True
>>> T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight␣
↪(1, 1, 1, 1, 1, 1)
>>> T.representation('factorized_permutation') == T
False

>>> T = WeakTableaux(Integer(3), [Integer(5),Integer(2),Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)],␣
↪representation = 'factorized_permutation')
>>> T._representation
'factorized_permutation'
>>> T.representation('core')
Core weak 3-Tableaux of (skew) core shape [5, 2, 1] and weight (1, 1, 1, 1, 1,
↪ 1)
>>> T.representation('bounded')
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1,␣
↪1, 1, 1, 1)
>>> T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight␣
↪(1, 1, 1, 1, 1, 1)
```

shape()

Return the shape of the tableaux of `self`.

When `self` is the class of straight tableaux, the outer shape is returned. When `self` is the class of skew tableaux, the tuple of the outer and inner shape is returned.

Note that in the 'core' and 'factorized_permutation' representation, the shapes are $(k+1)$-cores. In the 'bounded' representation, the shapes are k-bounded partitions.

If the user wants to access the skew shape (even if the inner shape is empty), please use `self._shape`.

EXAMPLES:

```
sage: T = WeakTableaux(3, [5,2,2], [2,2,2,1])
sage: T.shape()
[5, 2, 2]
sage: T._shape
([5, 2, 2], [])
sage: T = WeakTableaux(3, [[5,2,2], [1]], [2,1,2,1])
sage: T.shape()
([5, 2, 2], [1])

sage: T = WeakTableaux(3, [3,2,2], [2,2,2,1], representation = 'bounded')
sage: T.shape()
[3, 2, 2]
sage: T._shape
([3, 2, 2], [])
sage: T = WeakTableaux(3, [[3,2,2], [1]], [2,1,2,1], representation = 'bounded
↪')
sage: T.shape()
([3, 2, 2], [1])

sage: T = WeakTableaux(3, [4,1], [2,2], representation = 'factorized_
↪permutation')
sage: T.shape()
[4, 1]
sage: T._shape
([4, 1], [])
sage: T = WeakTableaux(4, [[6,2,1], [2]], [2,1,1,1], representation =
↪'factorized_permutation')
sage: T.shape()
([6, 2, 1], [2])
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(3), [Integer(5),Integer(2),Integer(2)],␣
↪[Integer(2),Integer(2),Integer(2),Integer(1)])
>>> T.shape()
[5, 2, 2]
>>> T._shape
([5, 2, 2], [])
>>> T = WeakTableaux(Integer(3), [[Integer(5),Integer(2),Integer(2)],␣
↪[Integer(1)]], [Integer(2),Integer(1),Integer(2),Integer(1)])
>>> T.shape()
([5, 2, 2], [1])
```

(continues on next page)

(continued from previous page)

```
>>> T = WeakTableaux(Integer(3), [Integer(3),Integer(2),Integer(2)],␣
↪[Integer(2),Integer(2),Integer(2),Integer(1)], representation = 'bounded')
>>> T.shape()
[3, 2, 2]
>>> T._shape
([3, 2, 2], [])
>>> T = WeakTableaux(Integer(3), [[Integer(3),Integer(2),Integer(2)],␣
↪[Integer(1)]], [Integer(2),Integer(1),Integer(2),Integer(1)],␣
↪representation = 'bounded')
>>> T.shape()
([3, 2, 2], [1])

>>> T = WeakTableaux(Integer(3), [Integer(4),Integer(1)], [Integer(2),
↪Integer(2)], representation = 'factorized_permutation')
>>> T.shape()
[4, 1]
>>> T._shape
([4, 1], [])
>>> T = WeakTableaux(Integer(4), [[Integer(6),Integer(2),Integer(1)],␣
↪[Integer(2)]], [Integer(2),Integer(1),Integer(1),Integer(1)],␣
↪representation = 'factorized_permutation')
>>> T.shape()
([6, 2, 1], [2])
```

size()

Return the size of the shape.

In the bounded representation, the size of the shape is the number of boxes in the outer shape minus the number of boxes in the inner shape. For the core and factorized permutation representation, the size is the length of the outer shape minus the length of the inner shape.

EXAMPLES:

```
sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1])
sage: T.size()
6
sage: T = WeakTableaux(3, [3,2,1], [1,1,1,1,1,1], representation = 'bounded')
sage: T.size()
6
sage: T = WeakTableaux(4, [[6,2,1], [2]], [2,1,1,1], 'factorized_permutation')
sage: T.size()
5
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(3), [Integer(5),Integer(2),Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)])
>>> T.size()
6
>>> T = WeakTableaux(Integer(3), [Integer(3),Integer(2),Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)],␣
↪representation = 'bounded')
>>> T.size()
6
```

(continues on next page)

(continued from previous page)

```
>>> T = WeakTableaux(Integer(4), [[Integer(6),Integer(2),Integer(1)],␣
↪[Integer(2)]], [Integer(2),Integer(1),Integer(1),Integer(1)], 'factorized_
↪permutation')
>>> T.size()
5
```

class sage.combinat.k_tableau.**WeakTableaux_bounded**(*k*, *shape*, *weight*)

Bases: *WeakTableaux_abstract*

The class of (skew) weak k-tableaux in the bounded representation of shape `shape` (as k-bounded partition or tuple of k-bounded partitions in the skew case) and weight `weight`.

INPUT:

- `k` – positive integer
- `shape` – the shape of the k-tableaux represented as a k-bounded partition; if the tableaux are skew, the shape is a tuple of the outer and inner shape each represented as a k-bounded partition
- `weight` – the weight of the k-tableaux

EXAMPLES:

```
sage: T = WeakTableaux(3, [3,1], [2,2], representation = 'bounded')
sage: T.list()
[[[1, 1, 2], [2]]]

sage: T = WeakTableaux(3, [[3,2,1], [2]], [1,1,1,1], representation = 'bounded')
sage: T.list()
[[[None, None, 3], [1, 4], [2]],
[[None, None, 1], [2, 4], [3]],
[[None, None, 1], [2, 3], [4]]]
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(3), [Integer(3),Integer(1)], [Integer(2),Integer(2)],
↪ representation = 'bounded')
>>> T.list()
[[[1, 1, 2], [2]]]

>>> T = WeakTableaux(Integer(3), [[Integer(3),Integer(2),Integer(1)],␣
↪[Integer(2)]], [Integer(1),Integer(1),Integer(1),Integer(1)], representation =
↪'bounded')
>>> T.list()
[[[None, None, 3], [1, 4], [2]],
[[None, None, 1], [2, 4], [3]],
[[None, None, 1], [2, 3], [4]]]
```

Element

alias of *WeakTableau_bounded*

class sage.combinat.k_tableau.**WeakTableaux_core**(*k*, *shape*, *weight*)

Bases: *WeakTableaux_abstract*

The class of (skew) weak k-tableaux in the core representation of shape `shape` (as $k+1$-core) and weight `weight`.

INPUT:

- k – positive integer
- shape – the shape of the k-tableaux represented as a $(k+1)$-core; if the tableaux are skew, the shape is a tuple of the outer and inner shape (both as $(k+1)$-cores)
- weight – the weight of the k-tableaux

EXAMPLES:

```
sage: T = WeakTableaux(3, [4,1], [2,2])
sage: T.list()
[[[1, 1, 2, 2], [2]]]

sage: T = WeakTableaux(3, [[5,2,1], [2]], [1,1,1,1])
sage: T.list()
[[[None, None, 2, 3, 4], [1, 4], [2]],
[[None, None, 1, 2, 4], [2, 4], [3]],
[[None, None, 1, 2, 3], [2, 3], [4]]]
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(3), [Integer(4),Integer(1)], [Integer(2),Integer(2)])
>>> T.list()
[[[1, 1, 2, 2], [2]]]

>>> T = WeakTableaux(Integer(3), [[Integer(5),Integer(2),Integer(1)],
↪[Integer(2)]], [Integer(1),Integer(1),Integer(1),Integer(1)])
>>> T.list()
[[[None, None, 2, 3, 4], [1, 4], [2]],
[[None, None, 1, 2, 4], [2, 4], [3]],
[[None, None, 1, 2, 3], [2, 3], [4]]]
```

Element

alias of *WeakTableau_core*

circular_distance(*cr*, *r*)

Return the shortest counterclockwise distance between cr and r modulo $k+1$.

INPUT:

- cr, r – nonnegative integers between 0 and k

OUTPUT: positive integer

EXAMPLES:

```
sage: T = WeakTableaux(10, [], [])
sage: T.circular_distance(8, 6)
2
sage: T.circular_distance(8, 8)
0
sage: T.circular_distance(8, 9)
10
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(10), [], [])
>>> T.circular_distance(Integer(8), Integer(6))
2
```

(continues on next page)

(continued from previous page)

```
>>> T.circular_distance(Integer(8), Integer(8))
0
>>> T.circular_distance(Integer(8), Integer(9))
10
```

diag(*c*, *ha*)

Return the number of diagonals strictly between cells `c` and `ha` of the same residue as `c`.

INPUT:

- `c` – a cell in the lattice
- `ha` – another cell in the lattice with bigger row and smaller column than c

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: T = WeakTableaux(4, [5,2,2], [2,2,2,1])
sage: T.diag((1,2),(4,0))
0
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(4), [Integer(5),Integer(2),Integer(2)],␣
↪[Integer(2),Integer(2),Integer(2),Integer(1)])
>>> T.diag((Integer(1),Integer(2)),(Integer(4),Integer(0)))
0
```

class sage.combinat.k_tableau.**WeakTableaux_factorized_permutation**(*k*, *shape*, *weight*)

Bases: *WeakTableaux_abstract*

The class of (skew) weak k-tableaux in the factorized permutation representation of shape `shape` (as $k+1$-core or tuple of $(k+1)$-cores in the skew case) and weight `weight`.

INPUT:

- `k` – positive integer
- `shape` – the shape of the k-tableaux represented as a $(k+1)$-core; in the skew case the shape is a tuple of the outer and inner shape both as $(k+1)$-cores
- `weight` – the weight of the k-tableaux

EXAMPLES:

```
sage: T = WeakTableaux(3, [4,1], [2,2], representation = 'factorized_permutation')
sage: T.list()
[[s3*s2, s1*s0]]

sage: T = WeakTableaux(4, [[6,2,1], [2]], [2,1,1,1], representation = 'factorized_
↪permutation')
sage: T.list()
[[s0, s4, s3, s4*s2], [s0, s3, s4, s3*s2], [s3, s0, s4, s3*s2]]
```

```
>>> from sage.all import *
>>> T = WeakTableaux(Integer(3), [Integer(4),Integer(1)], [Integer(2),Integer(2)],
↪ representation = 'factorized_permutation')
```

(continues on next page)

(continued from previous page)

```
>>> T.list()
[[s3*s2, s1*s0]]

>>> T = WeakTableaux(Integer(4), [[Integer(6),Integer(2),Integer(1)],
↪[Integer(2)]], [Integer(2),Integer(1),Integer(1),Integer(1)], representation =
↪'factorized_permutation')
>>> T.list()
[[s0, s4, s3, s4*s2], [s0, s3, s4, s3*s2], [s3, s0, s4, s3*s2]]
```

Element

alias of *WeakTableau_factorized_permutation*

sage.combinat.k_tableau.**intermediate_shapes**(*t*)

Return the intermediate shapes of tableau `t`.

A (skew) tableau with letters $1, 2, \ldots, \ell$ can be viewed as a sequence of shapes, where the i-th shape is given by the shape of the subtableau on letters $1, 2, \ldots, i$. The output is the list of these shapes.

OUTPUT: list of lists representing partitions

EXAMPLES:

```
sage: from sage.combinat.k_tableau import intermediate_shapes
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: intermediate_shapes(t)
[[], [2], [4, 1], [5, 2, 1]]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: intermediate_shapes(t)
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.k_tableau import intermediate_shapes
>>> t = WeakTableau([[Integer(1), Integer(1), Integer(2), Integer(2), Integer(3)],
↪ [Integer(2), Integer(3)], [Integer(3)]],Integer(3))
>>> intermediate_shapes(t)
[[], [2], [4, 1], [5, 2, 1]]

>>> t = WeakTableau([[None, None, Integer(2), Integer(3), Integer(4)],
↪[Integer(1), Integer(4)], [Integer(2)]], Integer(3))
>>> intermediate_shapes(t)
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]
```

sage.combinat.k_tableau.**nabs**(*v*)

Return the absolute value of `v` or `None`.

INPUT:

- `v` – either an integer or `None`

OUTPUT: either a nonnegative integer or `None`

EXAMPLES:

```
sage: from sage.combinat.k_tableau import nabs
sage: nabs(None)
sage: nabs(-3)
3
sage: nabs(None)
```

```
>>> from sage.all import *
>>> from sage.combinat.k_tableau import nabs
>>> nabs(None)
>>> nabs(-Integer(3))
3
>>> nabs(None)
```

5.1.131 Kazhdan-Lusztig Polynomials

AUTHORS:

- Daniel Bump (2008): initial version
- Alan J.X. Guo (2014-03-18): `R_tilde()` method.

class `sage.combinat.kazhdan_lusztig.`**`KazhdanLusztigPolynomial`**(*W*, *q*, *trace=False*)

Bases: `UniqueRepresentation`, `SageObject`

A Kazhdan-Lusztig polynomial.

INPUT:

- `W` – a Weyl Group
- `q` – an indeterminate

OPTIONAL:

- `trace` – if `True`, then this displays the trace: the intermediate results. This is instructive and fun.

The parent of `q` may be a `PolynomialRing` or a `LaurentPolynomialRing`.

EXAMPLES:

```
sage: W = WeylGroup("B3",prefix='s')
sage: [s1,s2,s3] = W.simple_reflections()
sage: R.<q> = LaurentPolynomialRing(QQ)
sage: KL = KazhdanLusztigPolynomial(W,q)
sage: KL.P(s2,s3*s2*s3*s1*s2)
1 + q
```

```
>>> from sage.all import *
>>> W = WeylGroup("B3",prefix='s')
>>> [s1,s2,s3] = W.simple_reflections()
>>> R = LaurentPolynomialRing(QQ, names=('q',)); (q,) = R._first_ngens(1)
>>> KL = KazhdanLusztigPolynomial(W,q)
>>> KL.P(s2,s3*s2*s3*s1*s2)
1 + q
```

A faster implementation (using the optional package Coxeter 3) is given by:

```
sage: W = CoxeterGroup(['B', 3], implementation='coxeter3') # optional - coxeter3
sage: W.kazhdan_lusztig_polynomial([2], [3,2,3,1,2])        # optional - coxeter3
q + 1
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['B', Integer(3)], implementation='coxeter3') # optional -␣
↪coxeter3
>>> W.kazhdan_lusztig_polynomial([Integer(2)], [Integer(3),Integer(2),Integer(3),
↪Integer(1),Integer(2)])        # optional - coxeter3
q + 1
```

P(*x*, *y*)

Return the Kazhdan-Lusztig P polynomial.

If the rank is large, this runs slowly at first but speeds up as you do repeated calculations due to the caching.

INPUT:

- x, y – elements of the underlying Coxeter group

See also

kazhdan_lusztig_polynomial for a faster implementation using Fokko Ducloux's Coxeter3 C++ library.

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: W = WeylGroup("A3", prefix='s')
sage: [s1,s2,s3] = W.simple_reflections()
sage: KL = KazhdanLusztigPolynomial(W, q)
sage: KL.P(s2,s2*s1*s3*s2)
q + 1
```

```
>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> W = WeylGroup("A3", prefix='s')
>>> [s1,s2,s3] = W.simple_reflections()
>>> KL = KazhdanLusztigPolynomial(W, q)
>>> KL.P(s2,s2*s1*s3*s2)
q + 1
```

R(*x*, *y*)

Return the Kazhdan-Lusztig R polynomial.

INPUT:

- x, y – elements of the underlying Coxeter group

EXAMPLES:

```
sage: R.<q>=QQ[]
sage: W = WeylGroup("A2", prefix='s')
sage: [s1,s2]=W.simple_reflections()
```

(continues on next page)

(continued from previous page)

```
sage: KL = KazhdanLusztigPolynomial(W, q)
sage: [KL.R(x,s2*s1) for x in [1,s1,s2,s1*s2]]
[q^2 - 2*q + 1, q - 1, q - 1, 0]
```

```
>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> W = WeylGroup("A2", prefix='s')
>>> [s1,s2]=W.simple_reflections()
>>> KL = KazhdanLusztigPolynomial(W, q)
>>> [KL.R(x,s2*s1) for x in [Integer(1),s1,s2,s1*s2]]
[q^2 - 2*q + 1, q - 1, q - 1, 0]
```

R_tilde (*x*, *y*)

Return the Kazhdan-Lusztig $\tilde{R}$ polynomial.

Information about the $\tilde{R}$ polynomials can be found in [Dy1993] and [BB2005].

INPUT:

- `x`, `y` – elements of the underlying Coxeter group

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: W = WeylGroup("A2", prefix='s')
sage: [s1,s2] = W.simple_reflections()
sage: KL = KazhdanLusztigPolynomial(W, q)
sage: [KL.R_tilde(x,s2*s1) for x in [1,s1,s2,s1*s2]]
[q^2, q, q, 0]
```

```
>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> W = WeylGroup("A2", prefix='s')
>>> [s1,s2] = W.simple_reflections()
>>> KL = KazhdanLusztigPolynomial(W, q)
>>> [KL.R_tilde(x,s2*s1) for x in [Integer(1),s1,s2,s1*s2]]
[q^2, q, q, 0]
```

5.1.132 Key polynomials

Key polynomials (also known as type A Demazure characters) are defined by applying the divided difference operator π_σ, where σ is a permutation, to a monomial corresponding to an integer partition $\mu \vdash n$.

See also

For Demazure characters in other types, see

- *sage.combinat.root_system.weyl_characters.WeylCharacterRing.demazure_character()*
- sage.categories.classical_crystals.ClassicalCrystals.ParentMethods.demazure_character()

AUTHORS:

- Trevor K. Karn (2022-08-17): initial version

class sage.combinat.key_polynomial.**KeyPolynomial**

Bases: IndexedFreeModuleElement

A key polynomial.

Key polynomials are polynomials that form a basis for a polynomial ring and are indexed by weak compositions.

Elements should be created by first creating the basis *KeyPolynomialBasis* and passing a list representing the indexing composition.

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: f = k([4,3,2,1]) + k([1,2,3,4]); f
k[1, 2, 3, 4] + k[4, 3, 2, 1]
sage: f in k
True
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> f = k([Integer(4),Integer(3),Integer(2),Integer(1)]) + k([Integer(1),
↪Integer(2),Integer(3),Integer(4)]); f
k[1, 2, 3, 4] + k[4, 3, 2, 1]
>>> f in k
True
```

divided_difference(*w*)

Apply the divided difference operator ∂_w to self.

The convention is to apply from left to right so if w = [w1, w2, ..., wm] then we apply $\partial_{w_2 \cdots w_m} \circ \partial_{w_1}$

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: k([3,2,1]).divided_difference(2)
k[3, 1, 1]
sage: k([3,2,1]).divided_difference([2,3])
k[3, 1]

sage: k = KeyPolynomials(QQ, 4)
sage: k([3,2,1,0]).divided_difference(2)
k[3, 1, 1, 0]
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> k([Integer(3),Integer(2),Integer(1)]).divided_difference(Integer(2))
k[3, 1, 1]
>>> k([Integer(3),Integer(2),Integer(1)]).divided_difference([Integer(2),
↪Integer(3)])
k[3, 1]

>>> k = KeyPolynomials(QQ, Integer(4))
>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).divided_
↪difference(Integer(2))
k[3, 1, 1, 0]
```

expand()

Return `self` written in the monomial basis (i.e., as an element in the corresponding polynomial ring).

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: f = k([4,3,2,1])
sage: f.expand()
z_3*z_2^2*z_1^3*z_0^4

sage: f = k([1,2,3])
sage: f.expand()
z_2^3*z_1^2*z_0 + z_2^3*z_1*z_0^2 + z_2^2*z_1^3*z_0
 + 2*z_2^2*z_1^2*z_0^2 + z_2^2*z_1*z_0^3 + z_2*z_1^3*z_0^2
 + z_2*z_1^2*z_0^3
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> f = k([Integer(4),Integer(3),Integer(2),Integer(1)])
>>> f.expand()
z_3*z_2^2*z_1^3*z_0^4

>>> f = k([Integer(1),Integer(2),Integer(3)])
>>> f.expand()
z_2^3*z_1^2*z_0 + z_2^3*z_1*z_0^2 + z_2^2*z_1^3*z_0
 + 2*z_2^2*z_1^2*z_0^2 + z_2^2*z_1*z_0^3 + z_2*z_1^3*z_0^2
 + z_2*z_1^2*z_0^3
```

isobaric_divided_difference(*w*)

Apply the operator π_w to `self`.

`w` may be either a `Permutation` or a list of indices of simple transpositions (1-based).

The convention is to apply from left to right so if `w = [w1, w2, ..., wm]` then we apply $\pi_{w_2 \cdots w_m} \circ \pi_{w_1}$

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: k([3,2,1]).pi(2)
k[3, 1, 2]
sage: k([3,2,1]).pi([2,1])
k[1, 3, 2]
sage: k([3,2,1]).pi(Permutation([3,2,1]))
k[1, 2, 3]
sage: f = k([3,2,1]) + k([3,2,1,1])
sage: f.pi(2)
k[3, 1, 2] + k[3, 1, 2, 1]
sage: k.one().pi(1)
k[]

sage: k([3,2,1,0]).pi(2).pi(2)
k[3, 1, 2]
sage: (-k([3,2,1,0]) + 4*k([3,1,2,0])).pi(2)
3*k[3, 1, 2]
```

(continues on next page)

(continued from previous page)

```
sage: k = KeyPolynomials(QQ, 4)
sage: k([3,2,1,0]).pi(2)
k[3, 1, 2, 0]
sage: k([3,2,1,0]).pi([2,1])
k[1, 3, 2, 0]
sage: k([3,2,1,0]).pi(Permutation([3,2,1,4]))
k[1, 2, 3, 0]
sage: f = k([3,2,1,0]) + k([3,2,1,1])
sage: f.pi(2)
k[3, 1, 2, 0] + k[3, 1, 2, 1]
sage: k.one().pi(1)
k[0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> k([Integer(3),Integer(2),Integer(1)]).pi(Integer(2))
k[3, 1, 2]
>>> k([Integer(3),Integer(2),Integer(1)]).pi([Integer(2),Integer(1)])
k[1, 3, 2]
>>> k([Integer(3),Integer(2),Integer(1)]).pi(Permutation([Integer(3),
↪Integer(2),Integer(1)]))
k[1, 2, 3]
>>> f = k([Integer(3),Integer(2),Integer(1)]) + k([Integer(3),Integer(2),
↪Integer(1),Integer(1)])
>>> f.pi(Integer(2))
k[3, 1, 2] + k[3, 1, 2, 1]
>>> k.one().pi(Integer(1))
k[]

>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).pi(Integer(2)).
↪pi(Integer(2))
k[3, 1, 2]
>>> (-k([Integer(3),Integer(2),Integer(1),Integer(0)]) +␣
↪Integer(4)*k([Integer(3),Integer(1),Integer(2),Integer(0)])).pi(Integer(2))
3*k[3, 1, 2]

>>> k = KeyPolynomials(QQ, Integer(4))
>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).pi(Integer(2))
k[3, 1, 2, 0]
>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).pi([Integer(2),
↪Integer(1)])
k[1, 3, 2, 0]
>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).
↪pi(Permutation([Integer(3),Integer(2),Integer(1),Integer(4)]))
k[1, 2, 3, 0]
>>> f = k([Integer(3),Integer(2),Integer(1),Integer(0)]) + k([Integer(3),
↪Integer(2),Integer(1),Integer(1)])
>>> f.pi(Integer(2))
k[3, 1, 2, 0] + k[3, 1, 2, 1]
>>> k.one().pi(Integer(1))
k[0, 0, 0, 0]
```

pi(*w*)

Apply the operator π_w to `self`.

`w` may be either a `Permutation` or a list of indices of simple transpositions (1-based).

The convention is to apply from left to right so if `w = [w1, w2, ..., wm]` then we apply $\pi_{w_2 \cdots w_m} \circ \pi_{w_1}$

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: k([3,2,1]).pi(2)
k[3, 1, 2]
sage: k([3,2,1]).pi([2,1])
k[1, 3, 2]
sage: k([3,2,1]).pi(Permutation([3,2,1]))
k[1, 2, 3]
sage: f = k([3,2,1]) + k([3,2,1,1])
sage: f.pi(2)
k[3, 1, 2] + k[3, 1, 2, 1]
sage: k.one().pi(1)
k[]

sage: k([3,2,1,0]).pi(2).pi(2)
k[3, 1, 2]
sage: (-k([3,2,1,0]) + 4*k([3,1,2,0])).pi(2)
3*k[3, 1, 2]

sage: k = KeyPolynomials(QQ, 4)
sage: k([3,2,1,0]).pi(2)
k[3, 1, 2, 0]
sage: k([3,2,1,0]).pi([2,1])
k[1, 3, 2, 0]
sage: k([3,2,1,0]).pi(Permutation([3,2,1,4]))
k[1, 2, 3, 0]
sage: f = k([3,2,1,0]) + k([3,2,1,1])
sage: f.pi(2)
k[3, 1, 2, 0] + k[3, 1, 2, 1]
sage: k.one().pi(1)
k[0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> k([Integer(3),Integer(2),Integer(1)]).pi(Integer(2))
k[3, 1, 2]
>>> k([Integer(3),Integer(2),Integer(1)]).pi([Integer(2),Integer(1)])
k[1, 3, 2]
>>> k([Integer(3),Integer(2),Integer(1)]).pi(Permutation([Integer(3),
↪Integer(2),Integer(1)]))
k[1, 2, 3]
>>> f = k([Integer(3),Integer(2),Integer(1)]) + k([Integer(3),Integer(2),
↪Integer(1),Integer(1)])
>>> f.pi(Integer(2))
k[3, 1, 2] + k[3, 1, 2, 1]
>>> k.one().pi(Integer(1))
k[]
```

(continues on next page)

(continued from previous page)

```
>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).pi(Integer(2)).
→pi(Integer(2))
k[3, 1, 2]
>>> (-k([Integer(3),Integer(2),Integer(1),Integer(0)]) +␣
→Integer(4)*k([Integer(3),Integer(1),Integer(2),Integer(0)])).pi(Integer(2))
3*k[3, 1, 2]

>>> k = KeyPolynomials(QQ, Integer(4))
>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).pi(Integer(2))
k[3, 1, 2, 0]
>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).pi([Integer(2),
→Integer(1)])
k[1, 3, 2, 0]
>>> k([Integer(3),Integer(2),Integer(1),Integer(0)]).
→pi(Permutation([Integer(3),Integer(2),Integer(1),Integer(4)]))
k[1, 2, 3, 0]
>>> f = k([Integer(3),Integer(2),Integer(1),Integer(0)]) + k([Integer(3),
→Integer(2),Integer(1),Integer(1)])
>>> f.pi(Integer(2))
k[3, 1, 2, 0] + k[3, 1, 2, 1]
>>> k.one().pi(Integer(1))
k[0, 0, 0, 0]
```

to_polynomial()

Return `self` written in the monomial basis (i.e., as an element in the corresponding polynomial ring).

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: f = k([4,3,2,1])
sage: f.expand()
z_3*z_2^2*z_1^3*z_0^4

sage: f = k([1,2,3])
sage: f.expand()
z_2^3*z_1^2*z_0 + z_2^3*z_1*z_0^2 + z_2^2*z_1^3*z_0
 + 2*z_2^2*z_1^2*z_0^2 + z_2^2*z_1*z_0^3 + z_2*z_1^3*z_0^2
 + z_2*z_1^2*z_0^3
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> f = k([Integer(4),Integer(3),Integer(2),Integer(1)])
>>> f.expand()
z_3*z_2^2*z_1^3*z_0^4

>>> f = k([Integer(1),Integer(2),Integer(3)])
>>> f.expand()
z_2^3*z_1^2*z_0 + z_2^3*z_1*z_0^2 + z_2^2*z_1^3*z_0
 + 2*z_2^2*z_1^2*z_0^2 + z_2^2*z_1*z_0^3 + z_2*z_1^3*z_0^2
 + z_2*z_1^2*z_0^3
```

class sage.combinat.key_polynomial.**KeyPolynomialBasis**(*R=None*, *k=None*, *poly_ring=None*)

Bases: *CombinatorialFreeModule*

The key polynomial basis for a polynomial ring.

For a full definition, see SymmetricFunctions.com. Key polynomials are indexed by weak compositions with no trailing zeros, and σ is the permutation of shortest length which sorts the indexing composition into a partition.

EXAMPLES:

Key polynomials are a basis, indexed by (weak) compositions, for polynomial rings:

```
sage: k = KeyPolynomials(QQ)
sage: k([3,0,1,2])
k[3, 0, 1, 2]
sage: k([3,0,1,2])/2
1/2*k[3, 0, 1, 2]
sage: R = k.polynomial_ring(); R
Infinite polynomial ring in z over Rational Field

sage: K = KeyPolynomials(GF(5)); K
Key polynomial basis over Finite Field of size 5
sage: 2*K([3,0,1,2])
2*k[3, 0, 1, 2]
sage: 5*(K([3,0,1,2]) + K([3,1,1]))
0
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> k([Integer(3),Integer(0),Integer(1),Integer(2)])
k[3, 0, 1, 2]
>>> k([Integer(3),Integer(0),Integer(1),Integer(2)])/Integer(2)
1/2*k[3, 0, 1, 2]
>>> R = k.polynomial_ring(); R
Infinite polynomial ring in z over Rational Field

>>> K = KeyPolynomials(GF(Integer(5))); K
Key polynomial basis over Finite Field of size 5
>>> Integer(2)*K([Integer(3),Integer(0),Integer(1),Integer(2)])
2*k[3, 0, 1, 2]
>>> Integer(5)*(K([Integer(3),Integer(0),Integer(1),Integer(2)]) + K([Integer(3),
↪Integer(1),Integer(1)]))
0
```

We can expand them in the standard monomial basis:

```
sage: k([3,0,1,2]).expand()
z_3^2*z_2*z_0^3 + z_3^2*z_1*z_0^3 + z_3*z_2^2*z_0^3
 + 2*z_3*z_2*z_1*z_0^3 + z_3*z_1^2*z_0^3 + z_2^2*z_1*z_0^3
 + z_2*z_1^2*z_0^3

sage: k([0,0,2]).expand()
z_2^2 + z_2*z_1 + z_2*z_0 + z_1^2 + z_1*z_0 + z_0^2
```

```
>>> from sage.all import *
>>> k([Integer(3),Integer(0),Integer(1),Integer(2)]).expand()
z_3^2*z_2*z_0^3 + z_3^2*z_1*z_0^3 + z_3*z_2^2*z_0^3
 + 2*z_3*z_2*z_1*z_0^3 + z_3*z_1^2*z_0^3 + z_2^2*z_1*z_0^3
 + z_2*z_1^2*z_0^3

>>> k([Integer(0),Integer(0),Integer(2)]).expand()
z_2^2 + z_2*z_1 + z_2*z_0 + z_1^2 + z_1*z_0 + z_0^2
```

If we have a polynomial, we can express it in the key basis:

```
sage: z = R.gen()
sage: k.from_polynomial(z[2]^2*z[1]*z[0])
k[1, 1, 2] - k[1, 2, 1]

sage: f = z[3]^2*z[2]*z[0]^3 + z[3]^2*z[1]*z[0]^3 + z[3]*z[2]^2*z[0]^3 + \
....: 2*z[3]*z[2]*z[1]*z[0]^3 + z[3]*z[1]^2*z[0]^3 + z[2]^2*z[1]*z[0]^3 + \
....: z[2]*z[1]^2*z[0]^3
sage: k.from_polynomial(f)
k[3, 0, 1, 2]
```

```
>>> from sage.all import *
>>> z = R.gen()
>>> k.from_polynomial(z[Integer(2)]**Integer(2)*z[Integer(1)]*z[Integer(0)])
k[1, 1, 2] - k[1, 2, 1]

>>> f = z[Integer(3)]**Integer(2)*z[Integer(2)]*z[Integer(0)]**Integer(3) +␣
↪z[Integer(3)]**Integer(2)*z[Integer(1)]*z[Integer(0)]**Integer(3) +␣
↪z[Integer(3)]*z[Integer(2)]**Integer(2)*z[Integer(0)]**Integer(3) +␣
↪Integer(2)*z[Integer(3)]*z[Integer(2)]*z[Integer(1)]*z[Integer(0)]**Integer(3)␣
↪+ z[Integer(3)]*z[Integer(1)]**Integer(2)*z[Integer(0)]**Integer(3) +␣
↪z[Integer(2)]**Integer(2)*z[Integer(1)]*z[Integer(0)]**Integer(3) +␣
↪z[Integer(2)]*z[Integer(1)]**Integer(2)*z[Integer(0)]**Integer(3)
>>> k.from_polynomial(f)
k[3, 0, 1, 2]
```

Since the ring of key polynomials may be regarded as a different choice of basis for a polynomial ring, it forms an algebra, so we have multiplication:

```
sage: k([10,5,2])*k([1,1,1])
k[11, 6, 3]
```

```
>>> from sage.all import *
>>> k([Integer(10),Integer(5),Integer(2)])*k([Integer(1),Integer(1),Integer(1)])
k[11, 6, 3]
```

We can also multiply by polynomials in the monomial basis:

```
sage: k([10,9,1])*z[0]
k[11, 9, 1]
sage: z[0] * k([10,9,1])
k[11, 9, 1]
```

(continues on next page)

(continued from previous page)

```
sage: k([10,9,1])*(z[0] + z[3])
k[10, 9, 1, 1] + k[11, 9, 1]
```

```
>>> from sage.all import *
>>> k([Integer(10),Integer(9),Integer(1)])*z[Integer(0)]
k[11, 9, 1]
>>> z[Integer(0)] * k([Integer(10),Integer(9),Integer(1)])
k[11, 9, 1]
>>> k([Integer(10),Integer(9),Integer(1)])*(z[Integer(0)] + z[Integer(3)])
k[10, 9, 1, 1] + k[11, 9, 1]
```

When the sorting permutation is the longest element, the key polynomial agrees with the Schur polynomial:

```
sage: s = SymmetricFunctions(QQ).schur()
sage: k([1,2,3]).expand()
z_2^3*z_1^2*z_0 + z_2^3*z_1*z_0^2 + z_2^2*z_1^3*z_0
 + 2*z_2^2*z_1^2*z_0^2 + z_2^2*z_1*z_0^3 + z_2*z_1^3*z_0^2
 + z_2*z_1^2*z_0^3
sage: s[3,2,1].expand(3)
x0^3*x1^2*x2 + x0^2*x1^3*x2 + x0^3*x1*x2^2 + 2*x0^2*x1^2*x2^2
 + x0*x1^3*x2^2 + x0^2*x1*x2^3 + x0*x1^2*x2^3
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).schur()
>>> k([Integer(1),Integer(2),Integer(3)]).expand()
z_2^3*z_1^2*z_0 + z_2^3*z_1*z_0^2 + z_2^2*z_1^3*z_0
 + 2*z_2^2*z_1^2*z_0^2 + z_2^2*z_1*z_0^3 + z_2*z_1^3*z_0^2
 + z_2*z_1^2*z_0^3
>>> s[Integer(3),Integer(2),Integer(1)].expand(Integer(3))
x0^3*x1^2*x2 + x0^2*x1^3*x2 + x0^3*x1*x2^2 + 2*x0^2*x1^2*x2^2
 + x0*x1^3*x2^2 + x0^2*x1*x2^3 + x0*x1^2*x2^3
```

The polynomial expansions can be computed using crystals and expressed in terms of the key basis:

```
sage: T = crystals.Tableaux(['A',3],shape=[2,1])
sage: f = T.demazure_character([3,2,1])
sage: k.from_polynomial(f)
k[1, 0, 0, 2]
```

```
>>> from sage.all import *
>>> T = crystals.Tableaux(['A',Integer(3)],shape=[Integer(2),Integer(1)])
>>> f = T.demazure_character([Integer(3),Integer(2),Integer(1)])
>>> k.from_polynomial(f)
k[1, 0, 0, 2]
```

The default behavior is to work in a polynomial ring with infinitely many variables. One can work in a specicfied number of variables:

```
sage: k = KeyPolynomials(QQ, 4)
sage: k([3,0,1,2]).expand()
z_0^3*z_1^2*z_2 + z_0^3*z_1*z_2^2 + z_0^3*z_1^2*z_3
 + 2*z_0^3*z_1*z_2*z_3 + z_0^3*z_2^2*z_3 + z_0^3*z_1*z_3^2 + z_0^3*z_2*z_3^2
```

(continues on next page)

(continued from previous page)

```
sage: k([0,0,2,0]).expand()
z_0^2 + z_0*z_1 + z_1^2 + z_0*z_2  + z_1*z_2 + z_2^2

sage: k([0,0,2,0]).expand().parent()
Multivariate Polynomial Ring in z_0, z_1, z_2, z_3 over Rational Field
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ, Integer(4))
>>> k([Integer(3),Integer(0),Integer(1),Integer(2)]).expand()
z_0^3*z_1^2*z_2 + z_0^3*z_1*z_2^2 + z_0^3*z_1^2*z_3
 + 2*z_0^3*z_1*z_2*z_3 + z_0^3*z_2^2*z_3 + z_0^3*z_1*z_3^2 + z_0^3*z_2*z_3^2

>>> k([Integer(0),Integer(0),Integer(2),Integer(0)]).expand()
z_0^2 + z_0*z_1 + z_1^2 + z_0*z_2  + z_1*z_2 + z_2^2

>>> k([Integer(0),Integer(0),Integer(2),Integer(0)]).expand().parent()
Multivariate Polynomial Ring in z_0, z_1, z_2, z_3 over Rational Field
```

If working in a specified number of variables, the length of the indexing composition must be the same as the number of variables:

```
sage: k([0,0,2])
Traceback (most recent call last):
 ...
TypeError: do not know how to make x (= [0, 0, 2]) an element of self
 (=Key polynomial basis over Rational Field)
```

```
>>> from sage.all import *
>>> k([Integer(0),Integer(0),Integer(2)])
Traceback (most recent call last):
 ...
TypeError: do not know how to make x (= [0, 0, 2]) an element of self
 (=Key polynomial basis over Rational Field)
```

One can also work in a specified polynomial ring:

```
sage: k = KeyPolynomials(QQ['x0', 'x1', 'x2', 'x3'])
sage: k([0,2,0,0])
k[0, 2, 0, 0]
sage: k([4,0,0,0]).expand()
x0^4
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ['x0', 'x1', 'x2', 'x3'])
>>> k([Integer(0),Integer(2),Integer(0),Integer(0)])
k[0, 2, 0, 0]
>>> k([Integer(4),Integer(0),Integer(0),Integer(0)]).expand()
x0^4
```

If one wishes to use a polynomial ring as coefficients for the key polynomials, pass the keyword argument `poly_coeffs=True`:

```
sage: k = KeyPolynomials(QQ['q'], poly_coeffs=True)
sage: R = k.base_ring(); R
Univariate Polynomial Ring in q over Rational Field
sage: R.inject_variables()
Defining q
sage: (q^2 + q + 1)*k([0,2,2,0,3,2])
(q^2+q+1)*k[0, 2, 2, 0, 3, 2]
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ['q'], poly_coeffs=True)
>>> R = k.base_ring(); R
Univariate Polynomial Ring in q over Rational Field
>>> R.inject_variables()
Defining q
>>> (q**Integer(2) + q + Integer(1))*k([Integer(0),Integer(2),Integer(2),
↪Integer(0),Integer(3),Integer(2)])
(q^2+q+1)*k[0, 2, 2, 0, 3, 2]
```

Element

alias of *KeyPolynomial*

degree_on_basis(*alpha*)

Return the degree of the basis element indexed by `alpha`.

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: k.degree_on_basis([2,1,0,2])
5

sage: k = KeyPolynomials(QQ, 5)
sage: k.degree_on_basis([2,1,0,2,0])
5
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> k.degree_on_basis([Integer(2),Integer(1),Integer(0),Integer(2)])
5

>>> k = KeyPolynomials(QQ, Integer(5))
>>> k.degree_on_basis([Integer(2),Integer(1),Integer(0),Integer(2),
↪Integer(0)])
5
```

from_polynomial(*f*)

Expand a polynomial in terms of the key basis.

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: z = k.poly_gens(); z
z_*
sage: p = z[0]^4*z[1]^2*z[2]*z[3] + z[0]^4*z[1]*z[2]^2*z[3]
```

(continues on next page)

(continued from previous page)

```
sage: k.from_polynomial(p)
k[4, 1, 2, 1]

sage: all(k(c) == k.from_polynomial(k(c).expand()) for c in␣
↪IntegerVectors(n=5, k=4))
True

sage: T = crystals.Tableaux(['A', 4], shape=[4,2,1,1])
sage: k.from_polynomial(T.demazure_character([2]))
k[4, 1, 2, 1]
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> z = k.poly_gens(); z
z_*
>>> p = z[Integer(0)]**Integer(4)*z[Integer(1)]**Integer(2)*z[Integer(2)]*z[In
↪teger(3)] + z[Integer(0)]**Integer(4)*z[Integer(1)]*z[Integer(2)]**Integer(2
↪)*z[Integer(3)]
>>> k.from_polynomial(p)
k[4, 1, 2, 1]

>>> all(k(c) == k.from_polynomial(k(c).expand()) for c in␣
↪IntegerVectors(n=Integer(5), k=Integer(4)))
True

>>> T = crystals.Tableaux(['A', Integer(4)], shape=[Integer(4),Integer(2),
↪Integer(1),Integer(1)])
>>> k.from_polynomial(T.demazure_character([Integer(2)]))
k[4, 1, 2, 1]
```

from_schubert_polynomial(*x*)

Expand a Schubert polynomial in the key basis.

EXAMPLES:

```
sage: k = KeyPolynomials(ZZ)
sage: X = SchubertPolynomialRing(ZZ)
sage: f = X([2,1,5,4,3])
sage: k.from_schubert_polynomial(f)
k[1, 0, 2, 1] + k[2, 0, 2] + k[3, 0, 0, 1]
sage: k.from_schubert_polynomial(2)
2*k[]
sage: k(f)
k[1, 0, 2, 1] + k[2, 0, 2] + k[3, 0, 0, 1]

sage: k = KeyPolynomials(GF(7), 4)
sage: k.from_schubert_polynomial(f)
k[1, 0, 2, 1] + k[2, 0, 2, 0] + k[3, 0, 0, 1]
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(ZZ)
>>> X = SchubertPolynomialRing(ZZ)
```

(continues on next page)

(continued from previous page)

```
>>> f = X([Integer(2),Integer(1),Integer(5),Integer(4),Integer(3)])
>>> k.from_schubert_polynomial(f)
k[1, 0, 2, 1] + k[2, 0, 2] + k[3, 0, 0, 1]
>>> k.from_schubert_polynomial(Integer(2))
2*k[]
>>> k(f)
k[1, 0, 2, 1] + k[2, 0, 2] + k[3, 0, 0, 1]

>>> k = KeyPolynomials(GF(Integer(7)), Integer(4))
>>> k.from_schubert_polynomial(f)
k[1, 0, 2, 1] + k[2, 0, 2, 0] + k[3, 0, 0, 1]
```

one_basis()

Return the basis element indexing the identity.

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: k.one_basis()
[]

sage: k = KeyPolynomials(QQ, 4)
sage: k.one_basis()
[0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> k.one_basis()
[]

>>> k = KeyPolynomials(QQ, Integer(4))
>>> k.one_basis()
[0, 0, 0, 0]
```

poly_gens()

Return the polynomial generators for the polynomial ring associated to `self`.

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: k.poly_gens()
z_*

sage: k = KeyPolynomials(QQ, 4)
sage: k.poly_gens()
(z_0, z_1, z_2, z_3)
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> k.poly_gens()
z_*

```

(continues on next page)

(continued from previous page)

```
>>> k = KeyPolynomials(QQ, Integer(4))
>>> k.poly_gens()
(z_0, z_1, z_2, z_3)
```

polynomial_ring()

Return the polynomial ring associated to `self`.

EXAMPLES:

```
sage: k = KeyPolynomials(QQ)
sage: k.polynomial_ring()
Infinite polynomial ring in z over Rational Field

sage: k = KeyPolynomials(QQ, 4)
sage: k.polynomial_ring()
Multivariate Polynomial Ring in z_0, z_1, z_2, z_3 over Rational Field
```

```
>>> from sage.all import *
>>> k = KeyPolynomials(QQ)
>>> k.polynomial_ring()
Infinite polynomial ring in z over Rational Field

>>> k = KeyPolynomials(QQ, Integer(4))
>>> k.polynomial_ring()
Multivariate Polynomial Ring in z_0, z_1, z_2, z_3 over Rational Field
```

sage.combinat.key_polynomial.**divided_difference**(f, i)

Apply the `i`-th divided difference operator to the polynomial `f`.

EXAMPLES:

```
sage: from sage.combinat.key_polynomial import divided_difference
sage: k = KeyPolynomials(QQ)
sage: z = k.poly_gens()
sage: f = z[1]*z[2]^3 + z[1]*z[2]*z[3]
sage: divided_difference(f, 3)
z_3^2*z_1 + z_3*z_2*z_1 + z_2^2*z_1

sage: k = KeyPolynomials(QQ, 4)
sage: z = k.poly_gens()
sage: f = z[1]*z[2]^3 + z[1]*z[2]*z[3]
sage: divided_difference(f, 3)
z_1*z_2^2 + z_1*z_2*z_3 + z_1*z_3^2

sage: k = KeyPolynomials(QQ)
sage: R = k.polynomial_ring(); R
Infinite polynomial ring in z over Rational Field
sage: z = R.gen()
sage: divided_difference(z[1]*z[2]^3, 2)
-z_2^2*z_1 - z_2*z_1^2
sage: divided_difference(z[1]*z[2]*z[3], 3)
0
sage: divided_difference(z[1]*z[2]*z[3], 4)
```

(continues on next page)

(continued from previous page)

```
z_2*z_1
sage: divided_difference(z[1]*z[2]*z[4], 4)
-z_2*z_1

sage: k = KeyPolynomials(QQ, 5)
sage: z = k.polynomial_ring().gens()
sage: divided_difference(z[1]*z[2]^3, 2)
-z_1^2*z_2 - z_1*z_2^2
sage: divided_difference(z[1]*z[2]*z[3], 3)
0
sage: divided_difference(z[1]*z[2]*z[3], 4)
z_1*z_2
sage: divided_difference(z[1]*z[2]*z[4], 4)
-z_1*z_2
```

```
>>> from sage.all import *
>>> from sage.combinat.key_polynomial import divided_difference
>>> k = KeyPolynomials(QQ)
>>> z = k.poly_gens()
>>> f = z[Integer(1)]*z[Integer(2)]**Integer(3) +␣
↪z[Integer(1)]*z[Integer(2)]*z[Integer(3)]
>>> divided_difference(f, Integer(3))
z_3^2*z_1 + z_3*z_2*z_1 + z_2^2*z_1

>>> k = KeyPolynomials(QQ, Integer(4))
>>> z = k.poly_gens()
>>> f = z[Integer(1)]*z[Integer(2)]**Integer(3) +␣
↪z[Integer(1)]*z[Integer(2)]*z[Integer(3)]
>>> divided_difference(f, Integer(3))
z_1*z_2^2 + z_1*z_2*z_3 + z_1*z_3^2

>>> k = KeyPolynomials(QQ)
>>> R = k.polynomial_ring(); R
Infinite polynomial ring in z over Rational Field
>>> z = R.gen()
>>> divided_difference(z[Integer(1)]*z[Integer(2)]**Integer(3), Integer(2))
-z_2^2*z_1 - z_2*z_1^2
>>> divided_difference(z[Integer(1)]*z[Integer(2)]*z[Integer(3)], Integer(3))
0
>>> divided_difference(z[Integer(1)]*z[Integer(2)]*z[Integer(3)], Integer(4))
z_2*z_1
>>> divided_difference(z[Integer(1)]*z[Integer(2)]*z[Integer(4)], Integer(4))
-z_2*z_1

>>> k = KeyPolynomials(QQ, Integer(5))
>>> z = k.polynomial_ring().gens()
>>> divided_difference(z[Integer(1)]*z[Integer(2)]**Integer(3), Integer(2))
-z_1^2*z_2 - z_1*z_2^2
>>> divided_difference(z[Integer(1)]*z[Integer(2)]*z[Integer(3)], Integer(3))
0
>>> divided_difference(z[Integer(1)]*z[Integer(2)]*z[Integer(3)], Integer(4))
z_1*z_2
```

(continues on next page)

(continued from previous page)

```
>>> divided_difference(z[Integer(1)]*z[Integer(2)]*z[Integer(4)], Integer(4))
-z_1*z_2
```

sage.combinat.key_polynomial.**isobaric_divided_difference**(*f*, *w*)

Apply the isobaric divided difference operator π_w to the polynomial f.

w may be either a single index or a list of indices of simple transpositions.

> **Warning**
>
> The simple transpositions should be applied from left to right.

EXAMPLES:

```
sage: from sage.combinat.key_polynomial import isobaric_divided_difference as idd
sage: R.<z> = InfinitePolynomialRing(GF(3))
sage: idd(z[1]^4*z[2]^2*z[4], 4)
0

sage: idd(z[1]^4*z[2]^2*z[3]*z[4], 3)
z_4*z_3^2*z_2*z_1^4 + z_4*z_3*z_2^2*z_1^4

sage: idd(z[1]^4*z[2]^2*z[3]*z[4], [3, 4])
z_4^2*z_3*z_2*z_1^4 + z_4*z_3^2*z_2*z_1^4 + z_4*z_3*z_2^2*z_1^4

sage: idd(z[1]^4*z[2]^2*z[3]*z[4], [4, 3])
z_4*z_3^2*z_2*z_1^4 + z_4*z_3*z_2^2*z_1^4

sage: idd(z[1]^2*z[2], [3, 2])
z_3*z_2^2 + z_3*z_2*z_1 + z_3*z_1^2 + z_2^2*z_1 + z_2*z_1^2
```

```
>>> from sage.all import *
>>> from sage.combinat.key_polynomial import isobaric_divided_difference as idd
>>> R = InfinitePolynomialRing(GF(Integer(3)), names=('z',)); (z,) = R._first_
↪ngens(1)
>>> idd(z[Integer(1)]**Integer(4)*z[Integer(2)]**Integer(2)*z[Integer(4)],␣
↪Integer(4))
0

>>> idd(z[Integer(1)]**Integer(4)*z[Integer(2)]**Integer(2)*z[Integer(3)]*z[Inte-
ger(4)], Integer(3))
z_4*z_3^2*z_2*z_1^4 + z_4*z_3*z_2^2*z_1^4

>>> idd(z[Integer(1)]**Integer(4)*z[Integer(2)]**Integer(2)*z[Integer(3)]*z[Inte-
ger(4)], [Integer(3), Integer(4)])
z_4^2*z_3*z_2*z_1^4 + z_4*z_3^2*z_2*z_1^4 + z_4*z_3*z_2^2*z_1^4

>>> idd(z[Integer(1)]**Integer(4)*z[Integer(2)]**Integer(2)*z[Integer(3)]*z[Inte-
ger(4)], [Integer(4), Integer(3)])
z_4*z_3^2*z_2*z_1^4 + z_4*z_3*z_2^2*z_1^4
```

(continues on next page)

(continued from previous page)

```
>>> idd(z[Integer(1)]**Integer(2)*z[Integer(2)], [Integer(3), Integer(2)])
z_3*z_2^2 + z_3*z_2*z_1 + z_3*z_1^2 + z_2^2*z_1 + z_2*z_1^2
```

sage.combinat.key_polynomial.**sorting_word**(*alpha*)

Get a reduced word for the permutation which sorts `alpha` into a partition.

The result is a list `l = [i0, i1, i2, ...]` where each `ij` is a positive integer such that it applies the simple transposition $(i_j, i_j + 1)$. The transpositions are applied starting with `i0`, then `i1` is applied, followed by `i2`, and so on. See *sage.combinat.permutation.Permutation.reduced_words()* for the convention used.

EXAMPLES:

```
sage: IV = IntegerVectors()
sage: from sage.combinat.key_polynomial import sorting_word
sage: list(sorting_word(IV([2,3,2]))[0])
[1]
sage: sorting_word(IV([2,3,2]))[1]
[3, 2, 2]
sage: list(sorting_word(IV([5,6,7]))[0])
[1, 2, 1]
sage: list(sorting_word(IV([0,3,2]))[0])
[2, 1]
sage: list(sorting_word(IV([0,3,0,2]))[0])
[2, 3, 1]
sage: list(sorting_word(IV([3,2,1]))[0])
[]
sage: list(sorting_word(IV([2,3,3]))[0])
[2, 1]
```

```
>>> from sage.all import *
>>> IV = IntegerVectors()
>>> from sage.combinat.key_polynomial import sorting_word
>>> list(sorting_word(IV([Integer(2),Integer(3),Integer(2)]))[Integer(0)])
[1]
>>> sorting_word(IV([Integer(2),Integer(3),Integer(2)]))[Integer(1)]
[3, 2, 2]
>>> list(sorting_word(IV([Integer(5),Integer(6),Integer(7)]))[Integer(0)])
[1, 2, 1]
>>> list(sorting_word(IV([Integer(0),Integer(3),Integer(2)]))[Integer(0)])
[2, 1]
>>> list(sorting_word(IV([Integer(0),Integer(3),Integer(0),
↪Integer(2)]))[Integer(0)])
[2, 3, 1]
>>> list(sorting_word(IV([Integer(3),Integer(2),Integer(1)]))[Integer(0)])
[]
>>> list(sorting_word(IV([Integer(2),Integer(3),Integer(3)]))[Integer(0)])
[2, 1]
```

5.1.133 Knutson-Tao Puzzles

This module implements a generic algorithm to solve Knutson-Tao puzzles. An instance of this class will be callable: the arguments are the labels of north-east and north-west sides of the puzzle boundary; the output is the list of the fillings of the puzzle with the specified pieces.

Acknowledgements

This code was written during Sage Days 45 at ICERM with Franco Saliola, Anne Schilling, and Avinash Dalal in discussions with Allen Knutson. The code was tested afterwards by Liz Beazley and Ed Richmond.

Todo

Functionality to add:

- plotter will not plot edge labels higher than 2; e.g. in BK puzzles, the labels are 1,…, n and so in 3-step examples, none of the edge labels with 3 appear
- we should also have a 3-step puzzle pieces constructor, taken from p22 of arXiv math/0610538
- implement the bijection from puzzles to tableaux; see for example R. Vakil, A geometric Littlewood-Richardson rule, arXiv math/0302294 or K. Purbhoo, Puzzles, Tableaux and Mosaics, arXiv 0705.1184.

sage.combinat.knutson_tao_puzzles.**BK_pieces**(*max_letter*)

The puzzle pieces used in computing the Belkale-Kumar coefficients for any partial flag variety in type A.

There are two types of puzzle pieces:

- a triangle, with each edge labeled with the same letter;
- a rhombus, with edges labeled i, j, i, j in clockwise order with $i > j$.

Each of these is rotated by 60 degrees, but not reflected.

We model the rhombus pieces as two triangles: a delta piece north-west label i, north-east label j and south label $i(j)$; and a nabla piece with south-east label i, south-west label j and north label $i(j)$.

INPUT:

- max_letter – positive integer specifying the number of steps in the partial flag variety, equivalently, the number of elements in the alphabet for the edge labels. The smallest label is 1.

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import BK_pieces
sage: BK_pieces(3)
Nablas : [1\1/1, 1\2(1)/2, 1\3(1)/3, 2(1)\2/1, 2\1/2(1), 2\2/2, 2\3(2)/3, 3(1)\3/
↪1, 3(2)\3/2, 3\1/3(1), 3\2/3(2), 3\3/3]
Deltas : [1/1\1, 1/2\2(1), 1/3\3(1), 2(1)/1\2, 2/2(1)\1, 2/2\2, 2/3\3(2), 3(1)/1\
↪3, 3(2)/2\3, 3/3(1)\1, 3/3(2)\2, 3/3\3]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import BK_pieces
>>> BK_pieces(Integer(3))
Nablas : [1\1/1, 1\2(1)/2, 1\3(1)/3, 2(1)\2/1, 2\1/2(1), 2\2/2, 2\3(2)/3, 3(1)\3/
↪1, 3(2)\3/2, 3\1/3(1), 3\2/3(2), 3\3/3]
```

(continues on next page)

(continued from previous page)

```
Deltas : [1/1\1, 1/2\2(1), 1/3\3(1), 2(1)/1\2, 2/2(1)\1, 2/2\2, 2/3\3(2), 3(1)/1\
↪3, 3(2)/2\3, 3/3(1)\1, 3/3(2)\2, 3/3\3]
```

class sage.combinat.knutson_tao_puzzles.**DeltaPiece**(*south*, *north_west*, *north_east*)

Bases: *PuzzlePiece*

Delta Piece takes as input three labels, inputted as strings. They label the South, Northwest and Northeast edges, respectively.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: DeltaPiece('a','b','c')
b/a\c
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece
>>> DeltaPiece('a','b','c')
b/a\c
```

clockwise_rotation()

Rotate the Delta piece by 120 degree clockwise.

OUTPUT: Delta piece

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('1','2','3')
sage: delta.clockwise_rotation()
1/3\2
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece
>>> delta = DeltaPiece('1','2','3')
>>> delta.clockwise_rotation()
1/3\2
```

edges()

Return the tuple of edge names.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('1','2','3')
sage: delta.edges()
('south', 'north_west', 'north_east')
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece
>>> delta = DeltaPiece('1','2','3')
>>> delta.edges()
('south', 'north_west', 'north_east')
```

half_turn_rotation()

Rotate the Delta piece by 180 degree.

OUTPUT: Nabla piece

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('1','2','3')
sage: delta.half_turn_rotation()
3\1/2
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece
>>> delta = DeltaPiece('1','2','3')
>>> delta.half_turn_rotation()
3\1/2
```

sage.combinat.knutson_tao_puzzles.**HT_grassmannian_pieces**()

Define the puzzle pieces used in computing the torus-equivariant cohomology of the Grassmannian.

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import HT_grassmannian_pieces
sage: HT_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1, 1\T0|1/0]
Deltas : [0/0\0, 0/1\10, 0/T0|1\1, 1/10\0, 1/1\1, 10/0\1]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import HT_grassmannian_pieces
>>> HT_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1, 1\T0|1/0]
Deltas : [0/0\0, 0/1\10, 0/T0|1\1, 1/10\0, 1/1\1, 10/0\1]
```

sage.combinat.knutson_tao_puzzles.**HT_two_step_pieces**()

Define the puzzle pieces used in computing the equivariant two step puzzle pieces.

For the puzzle pieces, see Figure 26 on page 22 of [CoskunVakil06].

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import HT_two_step_pieces
sage: HT_two_step_pieces()
Nablas : [(21)0\21/0, 0\(21)0/21, 0\0/0, 0\10/1, 0\20/2, 10\1/0, 10\2(10)/2,
1\0/10, 1\1/1, 1\21/2, 1\T0|1/0, 2(10)\2/10, 20\2/0, 21\0/(21)0, 21\2/1, 21\T0|21/
↪0,
21\T10|21/10, 2\0/20, 2\1/21, 2\10/2(10), 2\2/2, 2\T0|2/0, 2\T10|2/10, 2\T1|2/1]
Deltas : [(21)0/0\21, 0/0\0, 0/1\10, 0/21\(21)0, 0/2\20, 0/T0|1\1, 0/T0|21\21, 0/
↪T0|2\2,
1/10\0, 1/1\1, 1/2\21, 1/T1|2\2, 10/0\1, 10/2\2(10), 10/T10|21\21, 10/T10|2\2,␣
↪2(10)/10\2,
2/2(10)\10, 2/20\0, 2/21\1, 2/2\2, 20/0\2, 21/(21)0\0, 21/1\2]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import HT_two_step_pieces
>>> HT_two_step_pieces()
Nablas : [(21)0\21/0, 0\(21)0/21, 0\0/0, 0\10/1, 0\20/2, 10\1/0, 10\2(10)/2,
1\0/10, 1\1/1, 1\21/2, 1\T0|1/0, 2(10)\2/10, 20\2/0, 21\0/(21)0, 21\2/1, 21\T0|21/
↪0,
21\T10|21/10, 2\0/20, 2\1/21, 2\10/2(10), 2\2/2, 2\T0|2/0, 2\T10|2/10, 2\T1|2/1]
Deltas : [(21)0/0\21, 0/0\0, 0/1\10, 0/21\(21)0, 0/2\20, 0/T0|1\1, 0/T0|21\21, 0/
↪T0|2\2,
1/10\0, 1/1\1, 1/2\21, 1/T1|2\2, 10/0\1, 10/2\2(10), 10/T10|21\21, 10/T10|2\2,␣
↪2(10)/10\2,
2/2(10)\10, 2/20\0, 2/21\1, 2/2\2, 20/0\2, 21/(21)0\0, 21/1\2]
```

sage.combinat.knutson_tao_puzzles.**H_grassmannian_pieces**()

Define the puzzle pieces used in computing the cohomology of the Grassmannian.

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import H_grassmannian_pieces
sage: H_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import H_grassmannian_pieces
>>> H_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

sage.combinat.knutson_tao_puzzles.**H_two_step_pieces**()

Define the puzzle pieces used in two step flags.

This rule is currently only conjecturally true. See [BuchKreschTamvakis03].

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import H_two_step_pieces
sage: H_two_step_pieces()
Nablas : [(21)0\21/0, 0\(21)0/21, 0\0/0, 0\10/1, 0\20/2, 10\1/0, 10\2(10)/2, 1\0/
↪10, 1\1/1, 1\21/2,
2(10)\2/10, 20\2/0, 21\0/(21)0, 21\2/1, 2\0/20, 2\1/21, 2\10/2(10), 2\2/2]
Deltas : [(21)0/0\21, 0/0\0, 0/1\10, 0/21\(21)0, 0/2\20, 1/10\0, 1/1\1, 1/2\21,␣
↪10/0\1, 10/2\2(10),
2(10)/10\2, 2/2(10)\10, 2/20\0, 2/21\1, 2/2\2, 20/0\2, 21/(21)0\0, 21/1\2]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import H_two_step_pieces
>>> H_two_step_pieces()
Nablas : [(21)0\21/0, 0\(21)0/21, 0\0/0, 0\10/1, 0\20/2, 10\1/0, 10\2(10)/2, 1\0/
↪10, 1\1/1, 1\21/2,
```

(continues on next page)

(continued from previous page)

```
2(10)\2/10, 20\2/0, 21\0/(21)0, 21\2/1, 2\0/20, 2\1/21, 2\10/2(10), 2\2/2]
Deltas : [(21)0/0\21, 0/0\0, 0/1\10, 0/21\(21)0, 0/2\20, 1/10\0, 1/1\1, 1/2\21,␣
↪10/0\1, 10/2\2(10),
2(10)/10\2, 2/2(10)\10, 2/20\0, 2/21\1, 2/2\2, 20/0\2, 21/(21)0\0, 21/1\2]
```

sage.combinat.knutson_tao_puzzles.**K_grassmannian_pieces**()

Define the puzzle pieces used in computing the K-theory of the Grassmannian.

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import K_grassmannian_pieces
sage: K_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 0\K/1, 10\1/0, 1\0/10, 1\0/K, 1\1/1, K\1/0]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1, K/K\K]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import K_grassmannian_pieces
>>> K_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 0\K/1, 10\1/0, 1\0/10, 1\0/K, 1\1/1, K\1/0]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1, K/K\K]
```

class sage.combinat.knutson_tao_puzzles.**KnutsonTaoPuzzleSolver**(*puzzle_pieces*)

Bases: `UniqueRepresentation`

Return puzzle solver function used to create all puzzles with given boundary conditions.

This class implements a generic algorithm to solve Knutson-Tao puzzles. An instance of this class will be callable: the arguments are the labels of north-east and north-west sides of the puzzle boundary; the output is the list of the fillings of the puzzle with the specified pieces.

INPUT:

- `puzzle_pieces` – takes either a collection of puzzle pieces or a string indicating a pre-programmed collection of puzzle pieces:
  - `H` – cohomology of the Grassmannian
  - `HT` – equivariant cohomology of the Grassmannian
  - `K` – K-theory
  - `H2step` – cohomology of the *2-step* Grassmannian
  - `HT2step` – equivariant cohomology of the *2-step* Grassmannian
  - `BK` – Belkale-Kumar puzzle pieces
- `max_letter` – `None` or a positive integer(default: `None`); this is only required for Belkale-Kumar puzzles

EXAMPLES:

Each puzzle piece is an edge-labelled triangle oriented in such a way that it has a south edge (called a *delta* piece) or a north edge (called a *nabla* piece). For example, the puzzle pieces corresponding to the cohomology of the Grassmannian are the following:

```
sage: from sage.combinat.knutson_tao_puzzles import H_grassmannian_pieces
sage: H_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import H_grassmannian_pieces
>>> H_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

In the string representation, the nabla pieces are depicted as `c\a/b`, where a is the label of the north edge, b is the label of the south-east edge, c is the label of the south-west edge. A similar string representation exists for the delta pieces.

To create a puzzle solver, one specifies a collection of puzzle pieces:

```
sage: KnutsonTaoPuzzleSolver(H_grassmannian_pieces())
Knutson-Tao puzzle solver with pieces:
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

```
>>> from sage.all import *
>>> KnutsonTaoPuzzleSolver(H_grassmannian_pieces())
Knutson-Tao puzzle solver with pieces:
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

The following shorthand to create the above puzzle solver is also supported:

```
sage: KnutsonTaoPuzzleSolver('H')
Knutson-Tao puzzle solver with pieces:
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

```
>>> from sage.all import *
>>> KnutsonTaoPuzzleSolver('H')
Knutson-Tao puzzle solver with pieces:
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

The solver will compute all fillings of the puzzle with the given puzzle pieces. The user specifies the labels of north-east and north-west sides of the puzzle boundary and the output is a list of the fillings of the puzzle with the specified pieces. For example, there is one solution to the puzzle whose north-west and north-east edges are both labeled '0':

```
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: ps('0', '0')
[{(1, 1): 0/0\0}]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('H')
```

(continues on next page)

(continued from previous page)

```
>>> ps('0', '0')
[{(1, 1): 0/0\0}]
```

There are two solutions to the puzzle whose north-west and north-east edges are both labeled '0101':

```
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: solns = ps('0101', '0101')
sage: len(solns)
2
sage: solns.sort(key=str)
sage: solns
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/10,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 10/\1  0\/0,
  (2, 4): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('H')
>>> solns = ps('0101', '0101')
>>> len(solns)
2
>>> solns.sort(key=str)
>>> solns
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/10,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
```

(continues on next page)

(continued from previous page)

```
 (2, 3): 10/\1  0\/0,
 (2, 4): 1/\1  1\/1,
 (3, 3): 0/0\0,
 (3, 4): 1/\0  0\/1,
 (4, 4): 1/1\1}]
```

The pieces in a puzzle filling are indexed by pairs of nonnegative integers (i, j) with $1 \leq i \leq j \leq n$, where n is the length of the word labelling the triangle edge. The pieces indexed by (i, i) are the triangles along the south edge of the puzzle.

```
sage: f = solns[0]
sage: [f[i, i] for i in range(1,5)]
[0/0\0, 1/1\1, 1/1\1, 10/0\1]
```

```
>>> from sage.all import *
>>> f = solns[Integer(0)]
>>> [f[i, i] for i in range(Integer(1),Integer(5))]
[0/0\0, 1/1\1, 1/1\1, 10/0\1]
```

The pieces indexed by (i, j) for $j > i$ are a pair consisting of a delta piece and nabla piece glued together along the south edge and north edge, respectively (these pairs are called *rhombi*).

```
sage: f = solns[0]
sage: f[1, 2]
1/\0  0\/1
```

```
>>> from sage.all import *
>>> f = solns[Integer(0)]
>>> f[Integer(1), Integer(2)]
1/\0  0\/1
```

There are various methods and options to display puzzle solutions. A single puzzle can be displayed using the plot method of the puzzle:

```
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: puzzle = ps('0101','1001')[0]
sage: puzzle.plot()  #not tested
sage: puzzle.plot(style='fill')  #not tested
sage: puzzle.plot(style='edges')  #not tested
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver("H")
>>> puzzle = ps('0101','1001')[Integer(0)]
>>> puzzle.plot()  #not tested
>>> puzzle.plot(style='fill')  #not tested
>>> puzzle.plot(style='edges')  #not tested
```

To plot several puzzle solutions, use the plot method of the puzzle solver:

```
sage: ps = KnutsonTaoPuzzleSolver('K')
sage: solns = ps('0101', '0101')
sage: ps.plot(solns)        # not tested
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('K')
>>> solns = ps('0101', '0101')
>>> ps.plot(solns)         # not tested
```

The code can also generate a PDF of a puzzle (using LaTeX and *tikz*):

```
sage: latex.extra_preamble(r'''\usepackage{tikz}''')
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: solns = ps('0101', '0101')
sage: view(solns[0], viewer='pdf')   # not tested
```

```
>>> from sage.all import *
>>> latex.extra_preamble(r'''\usepackage{tikz}''')
>>> ps = KnutsonTaoPuzzleSolver('H')
>>> solns = ps('0101', '0101')
>>> view(solns[Integer(0)], viewer='pdf')   # not tested
```

Below are examples of using each of the currently supported puzzles.

Cohomology of the Grassmannian:

```
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: solns = ps('0101', '0101')
sage: sorted(solns, key=str)
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/10,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 10/\1  0\/0,
  (2, 4): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver("H")
>>> solns = ps('0101', '0101')
>>> sorted(solns, key=str)
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
```

(continues on next page)

(continued from previous page)

```
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/10,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 10/\1  0\/0,
  (2, 4): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}]
```

Equivariant puzzles:

```
sage: ps = KnutsonTaoPuzzleSolver("HT")
sage: solns = ps('0101', '0101')
sage: sorted(solns, key=str)
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\1  1\/0,
  (2, 4): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}, {(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/10,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 10/\1  0\/0,
  (2, 4): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver("HT")
```

(continues on next page)

(continued from previous page)

```
>>> solns = ps('0101', '0101')
>>> sorted(solns, key=str)
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\1  1\/0,
  (2, 4): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}, {(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/10,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 10/\1  0\/0,
  (2, 4): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}]
```

K-Theory puzzles:

```
sage: ps = KnutsonTaoPuzzleSolver("K")
sage: solns = ps('0101', '0101')
sage: sorted(solns, key=str)
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/10,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 10/\1  0\/0,
  (2, 4): 1/\1  1\/1,
```

(continues on next page)

(continued from previous page)

```
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/K,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): K/\K  0\/1,
  (2, 4): 1/\1  K\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver("K")
>>> solns = ps('0101', '0101')
>>> sorted(solns, key=str)
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 0/\0  0\/0,
  (1, 4): 1/\0  0\/1,
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/10,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 10/\1  0\/0,
  (2, 4): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (4, 4): 1/1\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 0/\0  1\/K,
  (1, 4): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): K/\K  0\/1,
  (2, 4): 1/\1  K\/0,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (4, 4): 10/0\1}]
```

Two-step puzzles:

```
sage: ps = KnutsonTaoPuzzleSolver("H2step")
sage: solns = ps('01201', '01021')
sage: sorted(solns, key=str)
[{(1, 1): 0/0\0,
```

(continues on next page)

(continued from previous page)

```
  (1, 2): 1/\0  0\/1,
  (1, 3): 2/\0  0\/2,
  (1, 4): 0/\0  0\/0,
  (1, 5): 1/\0  0\/1,
  (2, 2): 1/2\21,
  (2, 3): 2/\2  21\/1,
  (2, 4): 0/\10  2\/21,
  (2, 5): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 21/\2  1\/1,
  (3, 5): 0/\0  2\/20,
  (4, 4): 1/1\1,
  (4, 5): 20/\2  1\/10,
  (5, 5): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 2/\1  1\/2,
  (1, 4): 0/\0  1\/10,
  (1, 5): 1/\0  0\/1,
  (2, 2): 0/2\20,
  (2, 3): 2/\2  20\/0,
  (2, 4): 10/\1  2\/20,
  (2, 5): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 20/\2  0\/0,
  (3, 5): 1/\0  2\/2(10),
  (4, 4): 0/0\0,
  (4, 5): 2(10)/\2  0\/1,
  (5, 5): 1/1\1}, {(1, 1): 0/2\20,
  (1, 2): 1/\21  20\/0,
  (1, 3): 2/\2  21\/1,
  (1, 4): 0/\0  2\/20,
  (1, 5): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 1/\0  0\/1,
  (2, 4): 20/\2  0\/0,
  (2, 5): 1/\1  2\/21,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (3, 5): 21/\0  0\/21,
  (4, 4): 10/0\1,
  (4, 5): 21/\2  1\/1,
  (5, 5): 1/1\1}]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver("H2step")
>>> solns = ps('01201', '01021')
>>> sorted(solns, key=str)
[{(1, 1): 0/0\0,
  (1, 2): 1/\0  0\/1,
  (1, 3): 2/\0  0\/2,
  (1, 4): 0/\0  0\/0,
  (1, 5): 1/\0  0\/1,
```

(continues on next page)

(continued from previous page)

```
  (2, 2): 1/2\21,
  (2, 3): 2/\2  21\/1,
  (2, 4): 0/\10  2\/21,
  (2, 5): 1/\1  10\/0,
  (3, 3): 1/1\1,
  (3, 4): 21/\2  1\/1,
  (3, 5): 0/\0  2\/20,
  (4, 4): 1/1\1,
  (4, 5): 20/\2  1\/10,
  (5, 5): 10/0\1}, {(1, 1): 0/1\10,
  (1, 2): 1/\1  10\/0,
  (1, 3): 2/\1  1\/2,
  (1, 4): 0/\0  1\/10,
  (1, 5): 1/\0  0\/1,
  (2, 2): 0/2\20,
  (2, 3): 2/\2  20\/0,
  (2, 4): 10/\1  2\/20,
  (2, 5): 1/\1  1\/1,
  (3, 3): 0/0\0,
  (3, 4): 20/\2  0\/0,
  (3, 5): 1/\0  2\/2(10),
  (4, 4): 0/0\0,
  (4, 5): 2(10)/\2  0\/1,
  (5, 5): 1/1\1}, {(1, 1): 0/2\20,
  (1, 2): 1/\21  20\/0,
  (1, 3): 2/\2  21\/1,
  (1, 4): 0/\0  2\/20,
  (1, 5): 1/\0  0\/1,
  (2, 2): 0/0\0,
  (2, 3): 1/\0  0\/1,
  (2, 4): 20/\2  0\/0,
  (2, 5): 1/\1  2\/21,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (3, 5): 21/\0  0\/21,
  (4, 4): 10/0\1,
  (4, 5): 21/\2  1\/1,
  (5, 5): 1/1\1}]
```

Two-step equivariant puzzles:

```
sage: ps = KnutsonTaoPuzzleSolver("HT2step")
sage: solns = ps('10212', '12012')
sage: sorted(solns, key=str)
[{(1, 1): 1/1\1,
  (1, 2): 0/\(21)0  1\/2,
  (1, 3): 2/\1  (21)0\/0,
  (1, 4): 1/\1  1\/1,
  (1, 5): 2/\1  1\/2,
  (2, 2): 2/2\2,
  (2, 3): 0/\2  2\/0,
  (2, 4): 1/\2  2\/1,
```

(continues on next page)

(continued from previous page)

```
(2, 5): 2/\2  2\/2,
(3, 3): 0/0\0,
(3, 4): 1/\0  0\/1,
(3, 5): 2/\0  0\/2,
(4, 4): 1/1\1,
(4, 5): 2/\1  1\/2,
(5, 5): 2/2\2}, {(1, 1): 1/1\1,
(1, 2): 0/\(21)0  1\/2,
(1, 3): 2/\1  (21)0\/0,
(1, 4): 1/\1  1\/1,
(1, 5): 2/\1  1\/2,
(2, 2): 2/2\2,
(2, 3): 0/\2  2\/0,
(2, 4): 1/\21  2\/2,
(2, 5): 2/\2  21\/1,
(3, 3): 0/0\0,
(3, 4): 2/\0  0\/2,
(3, 5): 1/\0  0\/1,
(4, 4): 2/2\2,
(4, 5): 1/\1  2\/21,
(5, 5): 21/1\2}, {(1, 1): 1/1\1,
(1, 2): 0/\(21)0  1\/2,
(1, 3): 2/\1  (21)0\/0,
(1, 4): 1/\1  1\/1,
(1, 5): 2/\1  1\/2,
(2, 2): 2/2\2,
(2, 3): 0/\20  2\/2,
(2, 4): 1/\21  20\/0,
(2, 5): 2/\2  21\/1,
(3, 3): 2/2\2,
(3, 4): 0/\0  2\/20,
(3, 5): 1/\0  0\/1,
(4, 4): 20/0\2,
(4, 5): 1/\1  2\/21,
(5, 5): 21/1\2}, {(1, 1): 1/1\1,
(1, 2): 0/\1  1\/0,
(1, 3): 2/\1  1\/2,
(1, 4): 1/\1  1\/1,
(1, 5): 2/\1  1\/2,
(2, 2): 0/2\20,
(2, 3): 2/\2  20\/0,
(2, 4): 1/\2  2\/1,
(2, 5): 2/\2  2\/2,
(3, 3): 0/0\0,
(3, 4): 1/\0  0\/1,
(3, 5): 2/\0  0\/2,
(4, 4): 1/1\1,
(4, 5): 2/\1  1\/2,
(5, 5): 2/2\2}, {(1, 1): 1/1\1,
(1, 2): 0/\1  1\/0,
(1, 3): 2/\1  1\/2,
(1, 4): 1/\1  1\/1,
```

(continues on next page)

(continued from previous page)

```
(1, 5): 2/\1  1\/2,
(2, 2): 0/2\20,
(2, 3): 2/\2  20\/0,
(2, 4): 1/\21  2\/2,
(2, 5): 2/\2  21\/1,
(3, 3): 0/0\0,
(3, 4): 2/\0  0\/2,
(3, 5): 1/\0  0\/1,
(4, 4): 2/2\2,
(4, 5): 1/\1  2\/21,
(5, 5): 21/1\2}, {(1, 1): 1/1\1,
(1, 2): 0/\10  1\/1,
(1, 3): 2/\10  10\/2,
(1, 4): 1/\1  10\/0,
(1, 5): 2/\1  1\/2,
(2, 2): 1/2\21,
(2, 3): 2/\2  21\/1,
(2, 4): 0/\2  2\/0,
(2, 5): 2/\2  2\/2,
(3, 3): 1/1\1,
(3, 4): 0/\0  1\/10,
(3, 5): 2/\0  0\/2,
(4, 4): 10/0\1,
(4, 5): 2/\1  1\/2,
(5, 5): 2/2\2}, {(1, 1): 1/1\1,
(1, 2): 0/\10  1\/1,
(1, 3): 2/\10  10\/2,
(1, 4): 1/\1  10\/0,
(1, 5): 2/\1  1\/2,
(2, 2): 1/2\21,
(2, 3): 2/\2  21\/1,
(2, 4): 0/\20  2\/2,
(2, 5): 2/\2  20\/0,
(3, 3): 1/1\1,
(3, 4): 2/\1  1\/2,
(3, 5): 0/\0  1\/10,
(4, 4): 2/2\2,
(4, 5): 10/\1  2\/20,
(5, 5): 20/0\2}, {(1, 1): 1/2\21,
(1, 2): 0/\20  21\/1,
(1, 3): 2/\2  20\/0,
(1, 4): 1/\1  2\/21,
(1, 5): 2/\1  1\/2,
(2, 2): 1/1\1,
(2, 3): 0/\1  1\/0,
(2, 4): 21/\2  1\/1,
(2, 5): 2/\2  2\/2,
(3, 3): 0/0\0,
(3, 4): 1/\0  0\/1,
(3, 5): 2/\0  0\/2,
(4, 4): 1/1\1,
(4, 5): 2/\1  1\/2,
```

(continues on next page)

(continued from previous page)

```
  (5, 5): 2/2\2}, {(1, 1): 1/2\21,
  (1, 2): 0/\20  21\/1,
  (1, 3): 2/\2  20\/0,
  (1, 4): 1/\1  2\/21,
  (1, 5): 2/\1  1\/2,
  (2, 2): 1/1\1,
  (2, 3): 0/\10  1\/1,
  (2, 4): 21/\2  10\/0,
  (2, 5): 2/\2  2\/2,
  (3, 3): 1/1\1,
  (3, 4): 0/\0  1\/10,
  (3, 5): 2/\0  0\/2,
  (4, 4): 10/0\1,
  (4, 5): 2/\1  1\/2,
  (5, 5): 2/2\2}, {(1, 1): 1/2\21,
  (1, 2): 0/\21  21\/0,
  (1, 3): 2/\2  21\/1,
  (1, 4): 1/\1  2\/21,
  (1, 5): 2/\1  1\/2,
  (2, 2): 0/1\10,
  (2, 3): 1/\1  10\/0,
  (2, 4): 21/\2  1\/1,
  (2, 5): 2/\2  2\/2,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (3, 5): 2/\0  0\/2,
  (4, 4): 1/1\1,
  (4, 5): 2/\1  1\/2,
  (5, 5): 2/2\2}]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver("HT2step")
>>> solns = ps('10212', '12012')
>>> sorted(solns, key=str)
[{(1, 1): 1/1\1,
  (1, 2): 0/\(21)0  1\/2,
  (1, 3): 2/\1  (21)0\/0,
  (1, 4): 1/\1  1\/1,
  (1, 5): 2/\1  1\/2,
  (2, 2): 2/2\2,
  (2, 3): 0/\2  2\/0,
  (2, 4): 1/\2  2\/1,
  (2, 5): 2/\2  2\/2,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (3, 5): 2/\0  0\/2,
  (4, 4): 1/1\1,
  (4, 5): 2/\1  1\/2,
  (5, 5): 2/2\2}, {(1, 1): 1/1\1,
  (1, 2): 0/\(21)0  1\/2,
  (1, 3): 2/\1  (21)0\/0,
  (1, 4): 1/\1  1\/1,
```

(continues on next page)

(continued from previous page)

```
(1, 5): 2/\1  1\/2,
(2, 2): 2/2\2,
(2, 3): 0/\2  2\/0,
(2, 4): 1/\21  2\/2,
(2, 5): 2/\2  21\/1,
(3, 3): 0/0\0,
(3, 4): 2/\0  0\/2,
(3, 5): 1/\0  0\/1,
(4, 4): 2/2\2,
(4, 5): 1/\1  2\/21,
(5, 5): 21/1\2}, {(1, 1): 1/1\1,
(1, 2): 0/\(21)0  1\/2,
(1, 3): 2/\1  (21)0\/0,
(1, 4): 1/\1  1\/1,
(1, 5): 2/\1  1\/2,
(2, 2): 2/2\2,
(2, 3): 0/\20  2\/2,
(2, 4): 1/\21  20\/0,
(2, 5): 2/\2  21\/1,
(3, 3): 2/2\2,
(3, 4): 0/\0  2\/20,
(3, 5): 1/\0  0\/1,
(4, 4): 20/0\2,
(4, 5): 1/\1  2\/21,
(5, 5): 21/1\2}, {(1, 1): 1/1\1,
(1, 2): 0/\1  1\/0,
(1, 3): 2/\1  1\/2,
(1, 4): 1/\1  1\/1,
(1, 5): 2/\1  1\/2,
(2, 2): 0/2\20,
(2, 3): 2/\2  20\/0,
(2, 4): 1/\2  2\/1,
(2, 5): 2/\2  2\/2,
(3, 3): 0/0\0,
(3, 4): 1/\0  0\/1,
(3, 5): 2/\0  0\/2,
(4, 4): 1/1\1,
(4, 5): 2/\1  1\/2,
(5, 5): 2/2\2}, {(1, 1): 1/1\1,
(1, 2): 0/\1  1\/0,
(1, 3): 2/\1  1\/2,
(1, 4): 1/\1  1\/1,
(1, 5): 2/\1  1\/2,
(2, 2): 0/2\20,
(2, 3): 2/\2  20\/0,
(2, 4): 1/\21  2\/2,
(2, 5): 2/\2  21\/1,
(3, 3): 0/0\0,
(3, 4): 2/\0  0\/2,
(3, 5): 1/\0  0\/1,
(4, 4): 2/2\2,
(4, 5): 1/\1  2\/21,
```

(continues on next page)

(continued from previous page)

```
(5, 5): 21/1\2}, {(1, 1): 1/1\1,
(1, 2): 0/\10  1\/1,
(1, 3): 2/\10  10\/2,
(1, 4): 1/\1  10\/0,
(1, 5): 2/\1  1\/2,
(2, 2): 1/2\21,
(2, 3): 2/\2  21\/1,
(2, 4): 0/\2  2\/0,
(2, 5): 2/\2  2\/2,
(3, 3): 1/1\1,
(3, 4): 0/\0  1\/10,
(3, 5): 2/\0  0\/2,
(4, 4): 10/0\1,
(4, 5): 2/\1  1\/2,
(5, 5): 2/2\2}, {(1, 1): 1/1\1,
(1, 2): 0/\10  1\/1,
(1, 3): 2/\10  10\/2,
(1, 4): 1/\1  10\/0,
(1, 5): 2/\1  1\/2,
(2, 2): 1/2\21,
(2, 3): 2/\2  21\/1,
(2, 4): 0/\20  2\/2,
(2, 5): 2/\2  20\/0,
(3, 3): 1/1\1,
(3, 4): 2/\1  1\/2,
(3, 5): 0/\0  1\/10,
(4, 4): 2/2\2,
(4, 5): 10/\1  2\/20,
(5, 5): 20/0\2}, {(1, 1): 1/2\21,
(1, 2): 0/\20  21\/1,
(1, 3): 2/\2  20\/0,
(1, 4): 1/\1  2\/21,
(1, 5): 2/\1  1\/2,
(2, 2): 1/1\1,
(2, 3): 0/\1  1\/0,
(2, 4): 21/\2  1\/1,
(2, 5): 2/\2  2\/2,
(3, 3): 0/0\0,
(3, 4): 1/\0  0\/1,
(3, 5): 2/\0  0\/2,
(4, 4): 1/1\1,
(4, 5): 2/\1  1\/2,
(5, 5): 2/2\2}, {(1, 1): 1/2\21,
(1, 2): 0/\20  21\/1,
(1, 3): 2/\2  20\/0,
(1, 4): 1/\1  2\/21,
(1, 5): 2/\1  1\/2,
(2, 2): 1/1\1,
(2, 3): 0/\10  1\/1,
(2, 4): 21/\2  10\/0,
(2, 5): 2/\2  2\/2,
(3, 3): 1/1\1,
```

(continues on next page)

(continued from previous page)

```
  (3, 4): 0/\0  1\/10,
  (3, 5): 2/\0  0\/2,
  (4, 4): 10/0\1,
  (4, 5): 2/\1  1\/2,
  (5, 5): 2/2\2}, {(1, 1): 1/2\21,
  (1, 2): 0/\21  21\/0,
  (1, 3): 2/\2  21\/1,
  (1, 4): 1/\1  2\/21,
  (1, 5): 2/\1  1\/2,
  (2, 2): 0/1\10,
  (2, 3): 1/\1  10\/0,
  (2, 4): 21/\2  1\/1,
  (2, 5): 2/\2  2\/2,
  (3, 3): 0/0\0,
  (3, 4): 1/\0  0\/1,
  (3, 5): 2/\0  0\/2,
  (4, 4): 1/1\1,
  (4, 5): 2/\1  1\/2,
  (5, 5): 2/2\2}]
```

Belkale-Kumar puzzles (the following example is Figure 2 of [KnutsonPurbhoo10]):

```
sage: ps = KnutsonTaoPuzzleSolver('BK', 3)
sage: solns = ps('12132', '23112')
sage: len(solns)
1
sage: solns[0].south_labels()
('3', '2', '1', '2', '1')
sage: solns
[{(1, 1): 1/3\3(1),
  (1, 2): 2/\3(2)  3(1)\/1,
  (1, 3): 1/\3(1)  3(2)\/2,
  (1, 4): 3/\3  3(1)\/1,
  (1, 5): 2/\2  3\/3(2),
  (2, 2): 1/2\2(1),
  (2, 3): 2/\2  2(1)\/1,
  (2, 4): 1/\2(1)  2\/2,
  (2, 5): 3(2)/\3  2(1)\/1,
  (3, 3): 1/1\1,
  (3, 4): 2/\1  1\/2,
  (3, 5): 1/\1  1\/1,
  (4, 4): 2/2\2,
  (4, 5): 1/\1  2\/2(1),
  (5, 5): 2(1)/1\2}]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('BK', Integer(3))
>>> solns = ps('12132', '23112')
>>> len(solns)
1
>>> solns[Integer(0)].south_labels()
('3', '2', '1', '2', '1')
```

(continues on next page)

(continued from previous page)

```
>>> solns
[{(1, 1): 1/3\3(1),
  (1, 2): 2/\3(2)  3(1)\/1,
  (1, 3): 1/\3(1)  3(2)\/2,
  (1, 4): 3/\3  3(1)\/1,
  (1, 5): 2/\2  3\/3(2),
  (2, 2): 1/2\2(1),
  (2, 3): 2/\2  2(1)\/1,
  (2, 4): 1/\2(1)  2\/2,
  (2, 5): 3(2)/\3  2(1)\/1,
  (3, 3): 1/1\1,
  (3, 4): 2/\1  1\/2,
  (3, 5): 1/\1  1\/1,
  (4, 4): 2/2\2,
  (4, 5): 1/\1  2\/2(1),
  (5, 5): 2(1)/1\2}]
```

plot(*puzzles*)

Return plot of puzzles.

INPUT:

- `puzzles` – list of puzzles

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver('K')
sage: solns = ps('0101', '0101')
sage: ps.plot(solns)        # not tested
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
>>> ps = KnutsonTaoPuzzleSolver('K')
>>> solns = ps('0101', '0101')
>>> ps.plot(solns)        # not tested
```

puzzle_pieces()

The puzzle pieces used for filling in the puzzles.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: ps.puzzle_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
>>> ps = KnutsonTaoPuzzleSolver('H')
>>> ps.puzzle_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

solutions(*lamda*, *mu*, *algorithm='strips'*)

structure_constants(*lamda*, *mu*, *nu=None*)

Compute cohomology structure coefficients from puzzles.

INPUT:

- `pieces` – puzzle pieces to be used
- `lambda`, `mu` – edge labels of puzzle for northwest and north east side
- `nu` – (default: `None`) if `nu` is not specified a dictionary is returned with the structure coefficients corresponding to all south labels; if `nu` is given, only the coefficients with the specified label is returned

OUTPUT: dictionary

EXAMPLES:

Note: In order to standardize the output of the following examples, we output a sorted list of items from the dictionary instead of the dictionary itself.

Grassmannian cohomology:

```
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: cp = ps.structure_constants('0101', '0101')
sage: sorted(cp.items(), key=str)
[(('0', '1', '1', '0'), 1), (('1', '0', '0', '1'), 1)]
sage: ps.structure_constants('001001', '001010', '010100')
1
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('H')
>>> cp = ps.structure_constants('0101', '0101')
>>> sorted(cp.items(), key=str)
[(('0', '1', '1', '0'), 1), (('1', '0', '0', '1'), 1)]
>>> ps.structure_constants('001001', '001010', '010100')
1
```

Equivariant cohomology:

```
sage: ps = KnutsonTaoPuzzleSolver('HT')
sage: cp = ps.structure_constants('0101', '0101')
sage: sorted(cp.items(), key=str)
[(('0', '1', '0', '1'), y2 - y3),
(('0', '1', '1', '0'), 1),
(('1', '0', '0', '1'), 1)]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('HT')
>>> cp = ps.structure_constants('0101', '0101')
>>> sorted(cp.items(), key=str)
[(('0', '1', '0', '1'), y2 - y3),
(('0', '1', '1', '0'), 1),
(('1', '0', '0', '1'), 1)]
```

K-theory:

```
sage: ps = KnutsonTaoPuzzleSolver('K')
sage: cp = ps.structure_constants('0101', '0101')
sage: sorted(cp.items(), key=str)
[(('0', '1', '1', '0'), 1), (('1', '0', '0', '1'), 1), (('1', '0', '1', '0'),␣
↪-1)]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('K')
>>> cp = ps.structure_constants('0101', '0101')
>>> sorted(cp.items(), key=str)
[(('0', '1', '1', '0'), 1), (('1', '0', '0', '1'), 1), (('1', '0', '1', '0'),␣
↪-1)]
```

Two-step:

```
sage: ps = KnutsonTaoPuzzleSolver('H2step')
sage: cp = ps.structure_constants('01122', '01122')
sage: sorted(cp.items(), key=str)
[(('0', '1', '1', '2', '2'), 1)]
sage: cp = ps.structure_constants('01201', '01021')
sage: sorted(cp.items(), key=str)
[(('0', '2', '1', '1', '0'), 1),
 (('1', '2', '0', '0', '1'), 1),
 (('2', '0', '1', '0', '1'), 1)]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('H2step')
>>> cp = ps.structure_constants('01122', '01122')
>>> sorted(cp.items(), key=str)
[(('0', '1', '1', '2', '2'), 1)]
>>> cp = ps.structure_constants('01201', '01021')
>>> sorted(cp.items(), key=str)
[(('0', '2', '1', '1', '0'), 1),
 (('1', '2', '0', '0', '1'), 1),
 (('2', '0', '1', '0', '1'), 1)]
```

Two-step equivariant:

```
sage: ps = KnutsonTaoPuzzleSolver('HT2step')
sage: cp = ps.structure_constants('10212', '12012')
sage: sorted(cp.items(), key=str)
[(('1', '2', '0', '1', '2'), y1*y2 - y2*y3 - y1*y4 + y3*y4),
 (('1', '2', '0', '2', '1'), y1 - y3),
 (('1', '2', '1', '0', '2'), y2 - y4),
 (('1', '2', '1', '2', '0'), 1),
 (('1', '2', '2', '0', '1'), 1),
 (('2', '1', '0', '1', '2'), y1 - y3),
 (('2', '1', '1', '0', '2'), 1)]
```

```
>>> from sage.all import *
>>> ps = KnutsonTaoPuzzleSolver('HT2step')
>>> cp = ps.structure_constants('10212', '12012')
```

(continues on next page)

(continued from previous page)

```
>>> sorted(cp.items(), key=str)
[(('1', '2', '0', '1', '2'), y1*y2 - y2*y3 - y1*y4 + y3*y4),
 (('1', '2', '0', '2', '1'), y1 - y3),
 (('1', '2', '1', '0', '2'), y2 - y4),
 (('1', '2', '1', '2', '0'), 1),
 (('1', '2', '2', '0', '1'), 1),
 (('2', '1', '0', '1', '2'), y1 - y3),
 (('2', '1', '1', '0', '2'), 1)]
```

class sage.combinat.knutson_tao_puzzles.**NablaPiece**(*north*, *south_east*, *south_west*)

Bases: *PuzzlePiece*

Nabla Piece takes as input three labels, inputted as strings. They label the North, Southeast and Southwest edges, respectively.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import NablaPiece
sage: NablaPiece('a','b','c')
c\a/b
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import NablaPiece
>>> NablaPiece('a','b','c')
c\a/b
```

clockwise_rotation()

Rotate the Nabla piece by 120 degree clockwise.

OUTPUT: Nabla piece

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import NablaPiece
sage: nabla = NablaPiece('1','2','3')
sage: nabla.clockwise_rotation()
2\3/1
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import NablaPiece
>>> nabla = NablaPiece('1','2','3')
>>> nabla.clockwise_rotation()
2\3/1
```

edges()

Return the tuple of edge names.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import NablaPiece
sage: nabla = NablaPiece('1','2','3')
sage: nabla.edges()
('north', 'south_east', 'south_west')
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import NablaPiece
>>> nabla = NablaPiece('1','2','3')
>>> nabla.edges()
('north', 'south_east', 'south_west')
```

half_turn_rotation()

Rotate the Nabla piece by 180 degree.

OUTPUT: Delta piece

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import NablaPiece
sage: nabla = NablaPiece('1','2','3')
sage: nabla.half_turn_rotation()
2/1\3
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import NablaPiece
>>> nabla = NablaPiece('1','2','3')
>>> nabla.half_turn_rotation()
2/1\3
```

class sage.combinat.knutson_tao_puzzles.**PuzzleFilling**(*north_west_labels*, *north_east_labels*)

Bases: `object`

Create partial puzzles and provides methods to build puzzles from them.

add_piece(*piece*)

Add `piece` to partial puzzle.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
sage: piece = DeltaPiece('0','1','0')
sage: P = PuzzleFilling('0101','0101'); P
{}
sage: P.add_piece(piece); P
{(1, 4): 1/0\0}
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
>>> piece = DeltaPiece('0','1','0')
>>> P = PuzzleFilling('0101','0101'); P
{}
>>> P.add_piece(piece); P
{(1, 4): 1/0\0}
```

add_pieces(*pieces*)

Add `piece` to partial puzzle.

INPUT:

- `pieces` – tuple of pieces

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
sage: P = PuzzleFilling('0101','0101'); P
{}
sage: piece = DeltaPiece('0','1','0')
sage: pieces = [piece,piece]
sage: P.add_pieces(pieces)
sage: P
{(1, 4): 1/0\0, (2, 4): 1/0\0}
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
>>> P = PuzzleFilling('0101','0101'); P
{}
>>> piece = DeltaPiece('0','1','0')
>>> pieces = [piece,piece]
>>> P.add_pieces(pieces)
>>> P
{(1, 4): 1/0\0, (2, 4): 1/0\0}
```

contribution()

Return equivariant contributions from `self` in polynomial ring.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("HT")
sage: puzzles = ps('0101','1001')
sage: sorted([p.contribution() for p in puzzles], key=str)
[1, y1 - y3]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
>>> ps = KnutsonTaoPuzzleSolver("HT")
>>> puzzles = ps('0101','1001')
>>> sorted([p.contribution() for p in puzzles], key=str)
[1, y1 - y3]
```

copy()

Return copy of `self`.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
sage: piece = DeltaPiece('0','1','0')
sage: P = PuzzleFilling('0101','0101'); P
{}
sage: PP = P.copy()
sage: P.add_piece(piece); P
{(1, 4): 1/0\0}
sage: PP
{}
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
>>> piece = DeltaPiece('0','1','0')
>>> P = PuzzleFilling('0101','0101'); P
{}
>>> PP = P.copy()
>>> P.add_piece(piece); P
{(1, 4): 1/0\0}
>>> PP
{}
```

is_completed()

Whether partial puzzle is complete (completely filled) or not.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P.is_completed()
False

sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: puzzle = ps('0101','1001')[0]
sage: puzzle.is_completed()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzleFilling
>>> P = PuzzleFilling('0101','0101')
>>> P.is_completed()
False

>>> from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
>>> ps = KnutsonTaoPuzzleSolver("H")
>>> puzzle = ps('0101','1001')[Integer(0)]
>>> puzzle.is_completed()
True
```

is_in_south_edge()

Check whether kink coordinates of partial puzzle is in south corner.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P.is_in_south_edge()
False
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzleFilling
>>> P = PuzzleFilling('0101','0101')
```

(continues on next page)

(continued from previous page)

```
>>> P.is_in_south_edge()
False
```

kink_coordinates()

Provide the coordinates of the kinks.

The kink coordinates are the coordinates up to which the puzzle has already been built. The kink starts in the north corner and then moves down the diagonals as the puzzles is built.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P
{}
sage: P.kink_coordinates()
(1, 4)
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzleFilling
>>> P = PuzzleFilling('0101','0101')
>>> P
{}
>>> P.kink_coordinates()
(1, 4)
```

north_east_label_of_kink()

Return north east label of kink.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P.north_east_label_of_kink()
'0'
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzleFilling
>>> P = PuzzleFilling('0101','0101')
>>> P.north_east_label_of_kink()
'0'
```

north_west_label_of_kink()

Return north-west label of kink.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P.north_west_label_of_kink()
'1'
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzleFilling
>>> P = PuzzleFilling('0101','0101')
>>> P.north_west_label_of_kink()
'1'
```

plot (*labels=True*, *style='fill'*)

Plot completed puzzle.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: puzzle = ps('0101','1001')[0]
sage: puzzle.plot()  #not tested
sage: puzzle.plot(style='fill')  #not tested
sage: puzzle.plot(style='edges')  #not tested
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
>>> ps = KnutsonTaoPuzzleSolver("H")
>>> puzzle = ps('0101','1001')[Integer(0)]
>>> puzzle.plot()  #not tested
>>> puzzle.plot(style='fill')  #not tested
>>> puzzle.plot(style='edges')  #not tested
```

south_labels ()

Return south labels for completed puzzle.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: ps('0101','1001')[0].south_labels()
('1', '0', '1', '0')
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
>>> ps = KnutsonTaoPuzzleSolver("H")
>>> ps('0101','1001')[Integer(0)].south_labels()
('1', '0', '1', '0')
```

class sage.combinat.knutson_tao_puzzles.**PuzzlePiece**

Bases: `object`

Abstract class for puzzle pieces.

This abstract class contains information on how to test equality of puzzle pieces, and sets color and plotting options.

border ()

Return the border of `self`.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('a','b','c')
sage: sorted(delta.border())
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece
>>> delta = DeltaPiece('a','b','c')
>>> sorted(delta.border())
['a', 'b', 'c']
```

color()

Return the color of `self`.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('a','b','c')
sage: delta.color()
'white'
sage: delta = DeltaPiece('0','0','0')
sage: delta.color()
'red'
sage: delta = DeltaPiece('1','1','1')
sage: delta.color()
'blue'
sage: delta = DeltaPiece('2','2','2')
sage: delta.color()
'green'
sage: delta = DeltaPiece('2','K','2')
sage: delta.color()
'orange'
sage: delta = DeltaPiece('2','T1/2','2')
sage: delta.color()
'yellow'
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece
>>> delta = DeltaPiece('a','b','c')
>>> delta.color()
'white'
>>> delta = DeltaPiece('0','0','0')
>>> delta.color()
'red'
>>> delta = DeltaPiece('1','1','1')
>>> delta.color()
'blue'
>>> delta = DeltaPiece('2','2','2')
>>> delta.color()
'green'
>>> delta = DeltaPiece('2','K','2')
>>> delta.color()
'orange'
```

(continues on next page)

(continued from previous page)

```
>>> delta = DeltaPiece('2','T1/2','2')
>>> delta.color()
'yellow'
```

edge_color(*edge*)

Color of the specified edge of `self` (to be used when plotting the piece).

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('1','0','10')
sage: delta.edge_color('south')
'blue'
sage: delta.edge_color('north_west')
'red'
sage: delta.edge_color('north_east')
'white'
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece
>>> delta = DeltaPiece('1','0','10')
>>> delta.edge_color('south')
'blue'
>>> delta.edge_color('north_west')
'red'
>>> delta.edge_color('north_east')
'white'
```

edge_label(*edge*)

Return the edge label of `edge`.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('2','K','2')
sage: delta.edge_label('south')
'2'
sage: delta.edge_label('north_east')
'2'
sage: delta.edge_label('north_west')
'K'
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece
>>> delta = DeltaPiece('2','K','2')
>>> delta.edge_label('south')
'2'
>>> delta.edge_label('north_east')
'2'
>>> delta.edge_label('north_west')
'K'
```

class sage.combinat.knutson_tao_puzzles.**PuzzlePieces**(*forbidden_border_labels=None*)

Bases: `object`

Construct a valid set of puzzle pieces.

This class constructs the set of valid puzzle pieces. It can take a list of forbidden border labels as input. These labels are forbidden from appearing on the south edge of a puzzle filling. The user can add valid nabla or delta pieces and specify which rotations of these pieces are legal. For example, `rotations=0` does not add any additional pieces (only the piece itself), `rotations=60` adds six pieces (the pieces and its rotations by 60, 120, 180, 240, 300), etc..

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, NablaPiece
sage: forbidden_border_labels = ['10']
sage: pieces = PuzzlePieces(forbidden_border_labels)
sage: pieces.add_piece(NablaPiece('0','0','0'), rotations=60)
sage: pieces.add_piece(NablaPiece('1','1','1'), rotations=60)
sage: pieces.add_piece(NablaPiece('1','0','10'), rotations=60)
sage: pieces
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzlePieces, NablaPiece
>>> forbidden_border_labels = ['10']
>>> pieces = PuzzlePieces(forbidden_border_labels)
>>> pieces.add_piece(NablaPiece('0','0','0'), rotations=Integer(60))
>>> pieces.add_piece(NablaPiece('1','1','1'), rotations=Integer(60))
>>> pieces.add_piece(NablaPiece('1','0','10'), rotations=Integer(60))
>>> pieces
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

The user can obtain the list of valid rhombi pieces as follows:

```
sage: sorted([p for p in pieces.rhombus_pieces()], key=str)
[0/\0  0\/0, 0/\0  1\/10, 0/\10  10\/0, 0/\10  1\/1, 1/\0  0\/1,
1/\1  10\/0, 1/\1  1\/1, 10/\1  0\/0, 10/\1  1\/10]
```

```
>>> from sage.all import *
>>> sorted([p for p in pieces.rhombus_pieces()], key=str)
[0/\0  0\/0, 0/\0  1\/10, 0/\10  10\/0, 0/\10  1\/1, 1/\0  0\/1,
1/\1  10\/0, 1/\1  1\/1, 10/\1  0\/0, 10/\1  1\/10]
```

add_T_piece(*label1*, *label2*)

Add a nabla and delta piece with `label1` and `label2`.

This method adds a nabla piece with edges `label2`T``label1``|``label2`` / `label1`. and a delta piece with edges `label1`/ T``label1``|``label2`` `label2`. It also adds T``label1``|``label2`` to the forbidden list.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces
sage: pieces = PuzzlePieces()
```

(continues on next page)

(continued from previous page)

```
sage: pieces.add_T_piece('1','3')
sage: pieces
Nablas : [3\T1|3/1]
Deltas : [1/T1|3\3]
sage: pieces._forbidden_border_labels
['T1|3']
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzlePieces
>>> pieces = PuzzlePieces()
>>> pieces.add_T_piece('1','3')
>>> pieces
Nablas : [3\T1|3/1]
Deltas : [1/T1|3\3]
>>> pieces._forbidden_border_labels
['T1|3']
```

add_forbidden_label(*label*)

Add forbidden border labels.

INPUT:

- `label` – string specifying a new forbidden label

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces
sage: pieces = PuzzlePieces()
sage: pieces.add_forbidden_label('1')
sage: pieces._forbidden_border_labels
['1']
sage: pieces.add_forbidden_label('2')
sage: pieces._forbidden_border_labels
['1', '2']
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzlePieces
>>> pieces = PuzzlePieces()
>>> pieces.add_forbidden_label('1')
>>> pieces._forbidden_border_labels
['1']
>>> pieces.add_forbidden_label('2')
>>> pieces._forbidden_border_labels
['1', '2']
```

add_piece(*piece*, *rotations=120*)

Add `piece` to the list of pieces.

INPUT:

- `piece` – a nabla piece or a delta piece
- `rotations` – (default: 120) 0, 60, 120, 180

The user can add valid nabla or delta pieces and specify which rotations of these pieces are legal. For example, `rotations=0` does not add any additional pieces (only the piece itself), `rotations=60` adds six pieces (namely three delta and three nabla pieces), while `rotations=120` adds only delta or nabla (depending on which piece `self` is). `rotations=180` adds the piece and its 180 degree rotation, i.e. one delta and one nabla piece.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: delta = DeltaPiece('a','b','c')
sage: pieces = PuzzlePieces()
sage: pieces
Nablas : []
Deltas : []
sage: pieces.add_piece(delta)
sage: pieces
Nablas : []
Deltas : [a/c\b, b/a\c, c/b\a]

sage: pieces = PuzzlePieces()
sage: pieces.add_piece(delta,rotations=0)
sage: pieces
Nablas : []
Deltas : [b/a\c]

sage: pieces = PuzzlePieces()
sage: pieces.add_piece(delta,rotations=60)
sage: pieces
Nablas : [a\b/c, b\c/a, c\a/b]
Deltas : [a/c\b, b/a\c, c/b\a]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
>>> delta = DeltaPiece('a','b','c')
>>> pieces = PuzzlePieces()
>>> pieces
Nablas : []
Deltas : []
>>> pieces.add_piece(delta)
>>> pieces
Nablas : []
Deltas : [a/c\b, b/a\c, c/b\a]

>>> pieces = PuzzlePieces()
>>> pieces.add_piece(delta,rotations=Integer(0))
>>> pieces
Nablas : []
Deltas : [b/a\c]

>>> pieces = PuzzlePieces()
>>> pieces.add_piece(delta,rotations=Integer(60))
>>> pieces
Nablas : [a\b/c, b\c/a, c\a/b]
Deltas : [a/c\b, b/a\c, c/b\a]
```

boundary_deltas()

Return deltas with south edges not in the forbidden list.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: pieces = PuzzlePieces(['a'])
sage: delta = DeltaPiece('a','b','c')
sage: pieces.add_piece(delta,rotations=60)
sage: sorted([p for p in pieces.boundary_deltas()], key=str)
[a/c\b, c/b\a]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
>>> pieces = PuzzlePieces(['a'])
>>> delta = DeltaPiece('a','b','c')
>>> pieces.add_piece(delta,rotations=Integer(60))
>>> sorted([p for p in pieces.boundary_deltas()], key=str)
[a/c\b, c/b\a]
```

delta_pieces()

Return the delta pieces as a set.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: pieces = PuzzlePieces()
sage: delta = DeltaPiece('a','b','c')
sage: pieces.add_piece(delta,rotations=60)
sage: sorted([p for p in pieces.delta_pieces()], key=str)
[a/c\b, b/a\c, c/b\a]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
>>> pieces = PuzzlePieces()
>>> delta = DeltaPiece('a','b','c')
>>> pieces.add_piece(delta,rotations=Integer(60))
>>> sorted([p for p in pieces.delta_pieces()], key=str)
[a/c\b, b/a\c, c/b\a]
```

nabla_pieces()

Return the nabla pieces as a set.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: pieces = PuzzlePieces()
sage: delta = DeltaPiece('a','b','c')
sage: pieces.add_piece(delta,rotations=60)
sage: sorted([p for p in pieces.nabla_pieces()], key=str)
[a\b/c, b\c/a, c\a/b]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
>>> pieces = PuzzlePieces()
```

(continues on next page)

(continued from previous page)

```
>>> delta = DeltaPiece('a','b','c')
>>> pieces.add_piece(delta,rotations=Integer(60))
>>> sorted([p for p in pieces.nabla_pieces()], key=str)
[a\b/c, b\c/a, c\a/b]
```

rhombus_pieces()

Return a set of all allowable rhombus pieces.

Allowable rhombus pieces are those where the south edge of the delta piece equals the north edge of the nabla piece.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: pieces = PuzzlePieces()
sage: delta = DeltaPiece('a','b','c')
sage: pieces.add_piece(delta,rotations=60)
sage: sorted([p for p in pieces.rhombus_pieces()], key=str)
[a/\b  b\/a, b/\c  c\/b, c/\a  a\/c]
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
>>> pieces = PuzzlePieces()
>>> delta = DeltaPiece('a','b','c')
>>> pieces.add_piece(delta,rotations=Integer(60))
>>> sorted([p for p in pieces.rhombus_pieces()], key=str)
[a/\b  b\/a, b/\c  c\/b, c/\a  a\/c]
```

class sage.combinat.knutson_tao_puzzles.**RhombusPiece**(*north_piece*, *south_piece*)

Bases: *PuzzlePiece*

Class of rhombi pieces.

To construct a rhombus piece we input a delta and a nabla piece. The delta and nabla pieces are joined along the south and north edge, respectively.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece,␣
↪RhombusPiece
sage: delta = DeltaPiece('1','2','3')
sage: nabla = NablaPiece('4','5','6')
sage: RhombusPiece(delta,nabla)
2/\3  6\/5
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece,␣
↪RhombusPiece
>>> delta = DeltaPiece('1','2','3')
>>> nabla = NablaPiece('4','5','6')
>>> RhombusPiece(delta,nabla)
2/\3  6\/5
```

edges()

Return the tuple of edge names.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece,
↪RhombusPiece
sage: delta = DeltaPiece('1','2','3')
sage: nabla = NablaPiece('4','5','6')
sage: RhombusPiece(delta,nabla).edges()
('north_west', 'north_east', 'south_east', 'south_west')
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece,
↪RhombusPiece
>>> delta = DeltaPiece('1','2','3')
>>> nabla = NablaPiece('4','5','6')
>>> RhombusPiece(delta,nabla).edges()
('north_west', 'north_east', 'south_east', 'south_west')
```

north_piece()

Return the north piece.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece,
↪RhombusPiece
sage: delta = DeltaPiece('1','2','3')
sage: nabla = NablaPiece('4','5','6')
sage: r = RhombusPiece(delta,nabla)
sage: r.north_piece()
2/1\3
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece,
↪RhombusPiece
>>> delta = DeltaPiece('1','2','3')
>>> nabla = NablaPiece('4','5','6')
>>> r = RhombusPiece(delta,nabla)
>>> r.north_piece()
2/1\3
```

south_piece()

Return the south piece.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece,
↪RhombusPiece
sage: delta = DeltaPiece('1','2','3')
sage: nabla = NablaPiece('4','5','6')
sage: r = RhombusPiece(delta,nabla)
sage: r.south_piece()
6\4/5
```

```
>>> from sage.all import *
>>> from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece,␣
↪RhombusPiece
>>> delta = DeltaPiece('1','2','3')
>>> nabla = NablaPiece('4','5','6')
>>> r = RhombusPiece(delta,nabla)
>>> r.south_piece()
6\4/5
```

5.1.134 Combinatorics on matrices

- *Dancing Links internal pyx code*
- *Dancing links C++ wrapper*
- *Hadamard matrices*
- *Latin Squares*

5.1.135 Dancing Links internal pyx code

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: x = dlx_solver(rows)
sage: x
Dancing links solver for 6 columns and 6 rows
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),Integer(5)],␣
↪[Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),Integer(5)],␣
↪[Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]]
>>> x = dlx_solver(rows)
>>> x
Dancing links solver for 6 columns and 6 rows
```

The number of solutions:

```
sage: x.number_of_solutions()
3
```

```
>>> from sage.all import *
>>> x.number_of_solutions()
3
```

Iterate over the solutions:

```
sage: sorted(map(sorted, x.solutions_iterator()))
[[0, 1], [2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> sorted(map(sorted, x.solutions_iterator()))
[[0, 1], [2, 3], [4, 5]]
```

All solutions (computed in parallel):

```
sage: sorted(map(sorted, x.all_solutions()))
[[0, 1], [2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> sorted(map(sorted, x.all_solutions()))
[[0, 1], [2, 3], [4, 5]]
```

Return the first solution found when the computation is done in parallel:

```
sage: sorted(x.one_solution(ncpus=2)) # random
[0, 1]
```

```
>>> from sage.all import *
>>> sorted(x.one_solution(ncpus=Integer(2))) # random
[0, 1]
```

Find all solutions using some specific rows:

```
sage: x_using_row_2 = x.restrict([2])
sage: x_using_row_2
Dancing links solver for 7 columns and 6 rows
sage: sorted(map(sorted, x_using_row_2.solutions_iterator()))
[[2, 3]]
```

```
>>> from sage.all import *
>>> x_using_row_2 = x.restrict([Integer(2)])
>>> x_using_row_2
Dancing links solver for 7 columns and 6 rows
>>> sorted(map(sorted, x_using_row_2.solutions_iterator()))
[[2, 3]]
```

The two basic methods that are wrapped in this class are `search` which returns `1` if a solution is found or `0` otherwise and `get_solution` which return the current solution:

```
sage: x = dlx_solver(rows)
sage: x.search()
1
sage: x.get_solution()
[0, 1]
sage: x.search()
1
sage: x.get_solution()
[2, 3]
sage: x.search()
1
sage: x.get_solution()
[4, 5]
```

(continues on next page)

(continued from previous page)

```
sage: x.search()
0
```

```
>>> from sage.all import *
>>> x = dlx_solver(rows)
>>> x.search()
1
>>> x.get_solution()
[0, 1]
>>> x.search()
1
>>> x.get_solution()
[2, 3]
>>> x.search()
1
>>> x.get_solution()
[4, 5]
>>> x.search()
0
```

There is also a method `reinitialize` to reinitialize the algorithm:

```
sage: x.reinitialize()
sage: x.search()
1
sage: x.get_solution()
[0, 1]
```

```
>>> from sage.all import *
>>> x.reinitialize()
>>> x.search()
1
>>> x.get_solution()
[0, 1]
```

class `sage.combinat.matrices.dancing_links.`**`dancing_linksWrapper`**

Bases: `object`

A simple class that implements dancing links.

The main methods to list the solutions are *`search()`* and *`get_solution()`*. You can also use *`number_of_solutions()`* to count them.

This class simply wraps a C++ implementation of Carlo Hamalainen.

`all_solutions`(*ncpus=None*, *column=None*)

Return all solutions found after splitting the problem to allow parallel computation.

INPUT:

- `ncpus` – integer (default: `None`); maximal number of subprocesses to use at the same time. If `None`, it detects the number of effective CPUs in the system using `sage.parallel.ncpus.ncpus()`.
- `column` – integer (default: `None`); the column used to split the problem, if `None` a random column is chosen

OUTPUT: list of solutions

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: S = d.all_solutions()
sage: sorted(sorted(s) for s in S)
[[0, 1], [2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
↪Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
↪Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]
>>> d = dlx_solver(rows)
>>> S = d.all_solutions()
>>> sorted(sorted(s) for s in S)
[[0, 1], [2, 3], [4, 5]]
```

The number of CPUs can be specified as input:

```
sage: S = Subsets(range(4))
sage: rows = map(list, S)
sage: dlx = dlx_solver(rows)
sage: dlx
Dancing links solver for 4 columns and 16 rows
sage: dlx.number_of_solutions()
15
sage: sorted(sorted(s) for s in dlx.all_solutions(ncpus=2))
[[1, 2, 3, 4],
 [1, 2, 10],
 [1, 3, 9],
 [1, 4, 8],
 [1, 14],
 [2, 3, 7],
 [2, 4, 6],
 [2, 13],
 [3, 4, 5],
 [3, 12],
 [4, 11],
 [5, 10],
 [6, 9],
 [7, 8],
 [15]]
```

```
>>> from sage.all import *
>>> S = Subsets(range(Integer(4)))
>>> rows = map(list, S)
>>> dlx = dlx_solver(rows)
>>> dlx
Dancing links solver for 4 columns and 16 rows
```

(continues on next page)

(continued from previous page)

```
>>> dlx.number_of_solutions()
15
>>> sorted(sorted(s) for s in dlx.all_solutions(ncpus=Integer(2)))
[[1, 2, 3, 4],
 [1, 2, 10],
 [1, 3, 9],
 [1, 4, 8],
 [1, 14],
 [2, 3, 7],
 [2, 4, 6],
 [2, 13],
 [3, 4, 5],
 [3, 12],
 [4, 11],
 [5, 10],
 [6, 9],
 [7, 8],
 [15]]
```

If ncpus=1, the computation is not done in parallel:

```
sage: sorted(sorted(s) for s in dlx.all_solutions(ncpus=1))
[[1, 2, 3, 4],
 [1, 2, 10],
 [1, 3, 9],
 [1, 4, 8],
 [1, 14],
 [2, 3, 7],
 [2, 4, 6],
 [2, 13],
 [3, 4, 5],
 [3, 12],
 [4, 11],
 [5, 10],
 [6, 9],
 [7, 8],
 [15]]
```

```
>>> from sage.all import *
>>> sorted(sorted(s) for s in dlx.all_solutions(ncpus=Integer(1)))
[[1, 2, 3, 4],
 [1, 2, 10],
 [1, 3, 9],
 [1, 4, 8],
 [1, 14],
 [2, 3, 7],
 [2, 4, 6],
 [2, 13],
 [3, 4, 5],
 [3, 12],
 [4, 11],
 [5, 10],
```

(continues on next page)

(continued from previous page)

```
[6, 9],
[7, 8],
[15]]
```

get_solution()

Return the current solution.

After a new solution is found using the method *search()* this method return the rows that make up the current solution.

ncols()

Return the number of columns.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [1,2], [0], [3,4,5]]
sage: dlx = dlx_solver(rows)
sage: dlx.ncols()
6
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(1),Integer(2)],␣
↪[Integer(0)], [Integer(3),Integer(4),Integer(5)]]
>>> dlx = dlx_solver(rows)
>>> dlx.ncols()
6
```

nrows()

Return the number of rows.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [1,2], [0], [3,4,5]]
sage: dlx = dlx_solver(rows)
sage: dlx.nrows()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(1),Integer(2)],␣
↪[Integer(0)], [Integer(3),Integer(4),Integer(5)]]
>>> dlx = dlx_solver(rows)
>>> dlx.nrows()
4
```

number_of_solutions(*ncpus=None*, *column=None*)

Return the number of distinct solutions.

INPUT:

- ncpus – integer (default: None); maximal number of subprocesses to use at the same time. If ncpus>1 the dancing links problem is split into independent subproblems to allow parallel computation. If None, it detects the number of effective CPUs in the system using sage.parallel.ncpus.ncpus().
- column – integer (default: None); the column used to split the problem, if None a random column is chosen (this argument is ignored if ncpus is 1)

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2]]
sage: rows += [[0,2]]
sage: rows += [[1]]
sage: rows += [[3]]
sage: x = dlx_solver(rows)
sage: x.number_of_solutions()
2
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)]]
>>> rows += [[Integer(0),Integer(2)]]
>>> rows += [[Integer(1)]]
>>> rows += [[Integer(3)]]
>>> x = dlx_solver(rows)
>>> x.number_of_solutions()
2
```

The number of CPUs can be specified as input:

```
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: x = dlx_solver(rows)
sage: x.number_of_solutions(ncpus=2, column=3)
3
```

```
>>> from sage.all import *
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
↪Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
↪Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]
>>> x = dlx_solver(rows)
>>> x.number_of_solutions(ncpus=Integer(2), column=Integer(3))
3
```

```
sage: S = Subsets(range(5))
sage: rows = map(list, S)
sage: d = dlx_solver(rows)
sage: d.number_of_solutions()
52
```

```
>>> from sage.all import *
>>> S = Subsets(range(Integer(5)))
```

(continues on next page)

(continued from previous page)

```
>>> rows = map(list, S)
>>> d = dlx_solver(rows)
>>> d.number_of_solutions()
52
```

one_solution(*ncpus=None*, *column=None*)

Return the first solution found.

This method allows parallel computations which might be useful for some kind of problems when it is very hard just to find one solution.

INPUT:

- `ncpus` – integer (default: `None`); maximal number of subprocesses to use at the same time. If `None`, it detects the number of effective CPUs in the system using `sage.parallel.ncpus.ncpus()`. If `ncpus=1`, the first solution is searched serially.
- `column` – integer (default: `None`); the column used to split the problem (see `restrict()`). If `None`, a random column is chosen. This argument is ignored if `ncpus=1`.

OUTPUT: list of rows or `None` if no solution is found

Note

For some case, increasing the number of cpus makes it faster. For other instances, `ncpus=1` is faster. It all depends on problem which is considered.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: solutions = [[0,1], [2,3], [4,5]]
sage: sorted(d.one_solution()) in solutions
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
→Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
→Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
→Integer(5)]]
>>> d = dlx_solver(rows)
>>> solutions = [[Integer(0),Integer(1)], [Integer(2),Integer(3)],
→[Integer(4),Integer(5)]]
>>> sorted(d.one_solution()) in solutions
True
```

The number of CPUs can be specified as input:

```
sage: sorted(d.one_solution(ncpus=2)) in solutions
True
```

```
>>> from sage.all import *
>>> sorted(d.one_solution(ncpus=Integer(2))) in solutions
True
```

The column used to split the problem for parallel computations can be given:

```
sage: sorted(d.one_solution(ncpus=2, column=4)) in solutions
True
```

```
>>> from sage.all import *
>>> sorted(d.one_solution(ncpus=Integer(2), column=Integer(4))) in solutions
True
```

When no solution is found:

```
sage: rows = [[0,1,2], [2,3,4,5], [0,1,2,3]]
sage: d = dlx_solver(rows)
sage: d.one_solution() is None
True
```

```
>>> from sage.all import *
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(2),Integer(3),
↪Integer(4),Integer(5)], [Integer(0),Integer(1),Integer(2),Integer(3)]]
>>> d = dlx_solver(rows)
>>> d.one_solution() is None
True
```

one_solution_using_milp_solver(*solver=None*, *integrality_tolerance=0.001*)

Return a solution found using a MILP solver.

INPUT:

- `solver` – string or `None` (default: `None`); possible values include `'GLPK'`, `'GLPK/exact'`, `'Coin'`, `'CPLEX'`, `'Gurobi'`, `'CVXOPT'`, `'PPL'`, `'InteractiveLP'`

OUTPUT: list of rows or `None` if no solution is found

Note

When comparing the time taken by method `one_solution`, have in mind that `one_solution_using_milp_solver` first creates (and caches) the MILP solver instance from the dancing links solver. This copy of data may take many seconds depending on the size of the problem.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: solutions = [[0,1], [2,3], [4,5]]
sage: d.one_solution_using_milp_solver() in solutions                    #␣
↪needs sage.numerical.mip
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
↪Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
↪Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]
>>> d = dlx_solver(rows)
>>> solutions = [[Integer(0),Integer(1)], [Integer(2),Integer(3)],␣
↪[Integer(4),Integer(5)]]
>>> d.one_solution_using_milp_solver() in solutions                    #␣
↪needs sage.numerical.mip
True
```

Using optional solvers:

```
sage: # optional - gurobi sage_numerical_backends_gurobi, needs sage.
↪numerical.mip
sage: s = d.one_solution_using_milp_solver('gurobi')
sage: s in solutions
True
```

```
>>> from sage.all import *
>>> # optional - gurobi sage_numerical_backends_gurobi, needs sage.numerical.
↪mip
>>> s = d.one_solution_using_milp_solver('gurobi')
>>> s in solutions
True
```

When no solution is found:

```
sage: rows = [[0,1,2], [2,3,4,5], [0,1,2,3]]
sage: d = dlx_solver(rows)
sage: d.one_solution_using_milp_solver() is None                       #␣
↪needs sage.numerical.mip
True
```

```
>>> from sage.all import *
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(2),Integer(3),
↪Integer(4),Integer(5)], [Integer(0),Integer(1),Integer(2),Integer(3)]]
>>> d = dlx_solver(rows)
>>> d.one_solution_using_milp_solver() is None                         #␣
↪needs sage.numerical.mip
True
```

one_solution_using_sat_solver(*solver=None*)

Return a solution found using a SAT solver.

INPUT:

- `solver` – string or `None` (default: `None`), possible values include `'picosat'`, `'cryptominisat'`, `'LP'`, `'glucose'`, `'glucose-syrup'`.

OUTPUT: list of rows or `None` if no solution is found

Note

When comparing the time taken by method `one_solution`, have in mind that `one_solution_using_sat_solver` first creates the SAT solver instance from the dancing links solver. This copy of data may take many seconds depending on the size of the problem.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: solutions = [[0,1], [2,3], [4,5]]
sage: d.one_solution_using_sat_solver() in solutions                           #␣
↪needs sage.sat
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
↪Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
↪Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]
>>> d = dlx_solver(rows)
>>> solutions = [[Integer(0),Integer(1)], [Integer(2),Integer(3)],␣
↪[Integer(4),Integer(5)]]
>>> d.one_solution_using_sat_solver() in solutions                            #␣
↪needs sage.sat
True
```

Using optional solvers:

```
sage: s = d.one_solution_using_sat_solver('glucose')                    #␣
↪optional - glucose, needs sage.sat
sage: s in solutions                                                    #␣
↪optional - glucose, needs sage.sat
True
```

```
>>> from sage.all import *
>>> s = d.one_solution_using_sat_solver('glucose')                  # optional -
↪ glucose, needs sage.sat
>>> s in solutions                                                  # optional -
↪ glucose, needs sage.sat
True
```

When no solution is found:

```
sage: rows = [[0,1,2], [2,3,4,5], [0,1,2,3]]
sage: d = dlx_solver(rows)
sage: d.one_solution_using_sat_solver() is None                                #␣
↪needs sage.sat
True
```

```
>>> from sage.all import *
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(2),Integer(3),
↪Integer(4),Integer(5)], [Integer(0),Integer(1),Integer(2),Integer(3)]]
>>> d = dlx_solver(rows)
>>> d.one_solution_using_sat_solver() is None                               #␣
↪needs sage.sat
True
```

reinitialize()

Reinitialization of the search algorithm.

This recreates an empty `dancing_links` object and adds the rows to the instance of `dancing_links`.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: x = dlx_solver(rows)
sage: x.get_solution() if x.search() else None
[0, 1]
sage: x.get_solution() if x.search() else None
[2, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
↪Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
↪Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]
>>> x = dlx_solver(rows)
>>> x.get_solution() if x.search() else None
[0, 1]
>>> x.get_solution() if x.search() else None
[2, 3]
```

Reinitialization of the algorithm:

```
sage: x.reinitialize()
sage: x.get_solution() if x.search() else None
[0, 1]
sage: x.get_solution() if x.search() else None
[2, 3]
sage: x.get_solution() if x.search() else None
[4, 5]
sage: x.get_solution() if x.search() else None
```

```
>>> from sage.all import *
>>> x.reinitialize()
>>> x.get_solution() if x.search() else None
[0, 1]
>>> x.get_solution() if x.search() else None
[2, 3]
>>> x.get_solution() if x.search() else None
```

(continues on next page)

(continued from previous page)

```
[4, 5]
>>> x.get_solution() if x.search() else None
```

Reinitialization works after solutions are exhausted:

```
sage: x.reinitialize()
sage: x.get_solution() if x.search() else None
[0, 1]
sage: x.get_solution() if x.search() else None
[2, 3]
sage: x.get_solution() if x.search() else None
[4, 5]
sage: x.get_solution() if x.search() else None
```

```
>>> from sage.all import *
>>> x.reinitialize()
>>> x.get_solution() if x.search() else None
[0, 1]
>>> x.get_solution() if x.search() else None
[2, 3]
>>> x.get_solution() if x.search() else None
[4, 5]
>>> x.get_solution() if x.search() else None
```

restrict(*indices*)

Return a dancing links solver solving the subcase which uses some given rows.

For every row that is wanted in the solution, we add a new column to the row to make sure it is in the solution.

INPUT:

- `indices` – list; row indices to be found in the solution

OUTPUT: dancing links solver

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: d
Dancing links solver for 6 columns and 6 rows
sage: sorted(map(sorted, d.solutions_iterator()))
[[0, 1], [2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
↪Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
↪Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]
>>> d = dlx_solver(rows)
>>> d
Dancing links solver for 6 columns and 6 rows
```

(continues on next page)

(continued from previous page)

```
>>> sorted(map(sorted, d.solutions_iterator()))
[[0, 1], [2, 3], [4, 5]]
```

To impose that the 0-th row is part of the solution, the rows of the new problem are:

```
sage: d_using_0 = d.restrict([0])
sage: d_using_0
Dancing links solver for 7 columns and 6 rows
sage: d_using_0.rows()
[[0, 1, 2, 6], [3, 4, 5], [0, 1], [2, 3, 4, 5], [0], [1, 2, 3, 4, 5]]
```

```
>>> from sage.all import *
>>> d_using_0 = d.restrict([Integer(0)])
>>> d_using_0
Dancing links solver for 7 columns and 6 rows
>>> d_using_0.rows()
[[0, 1, 2, 6], [3, 4, 5], [0, 1], [2, 3, 4, 5], [0], [1, 2, 3, 4, 5]]
```

After restriction the subproblem has one more columns and the same number of rows as the original one:

```
sage: d.restrict([1]).rows()
[[0, 1, 2], [3, 4, 5, 6], [0, 1], [2, 3, 4, 5], [0], [1, 2, 3, 4, 5]]
sage: d.restrict([2]).rows()
[[0, 1, 2], [3, 4, 5], [0, 1, 6], [2, 3, 4, 5], [0], [1, 2, 3, 4, 5]]
```

```
>>> from sage.all import *
>>> d.restrict([Integer(1)]).rows()
[[0, 1, 2], [3, 4, 5, 6], [0, 1], [2, 3, 4, 5], [0], [1, 2, 3, 4, 5]]
>>> d.restrict([Integer(2)]).rows()
[[0, 1, 2], [3, 4, 5], [0, 1, 6], [2, 3, 4, 5], [0], [1, 2, 3, 4, 5]]
```

This method allows to find solutions where the 0-th row is part of a solution:

```
sage: sorted(map(sorted, d.restrict([0]).solutions_iterator()))
[[0, 1]]
```

```
>>> from sage.all import *
>>> sorted(map(sorted, d.restrict([Integer(0)]).solutions_iterator()))
[[0, 1]]
```

Some other examples:

```
sage: sorted(map(sorted, d.restrict([1]).solutions_iterator()))
[[0, 1]]
sage: sorted(map(sorted, d.restrict([2]).solutions_iterator()))
[[2, 3]]
sage: sorted(map(sorted, d.restrict([3]).solutions_iterator()))
[[2, 3]]
```

```
>>> from sage.all import *
>>> sorted(map(sorted, d.restrict([Integer(1)]).solutions_iterator()))
[[0, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> sorted(map(sorted, d.restrict([Integer(2)]).solutions_iterator()))
[[2, 3]]
>>> sorted(map(sorted, d.restrict([Integer(3)]).solutions_iterator()))
[[2, 3]]
```

Here there are no solutions using both 0th and 3rd row:

```
sage: list(d.restrict([0,3]).solutions_iterator())
[]
```

```
>>> from sage.all import *
>>> list(d.restrict([Integer(0),Integer(3)]).solutions_iterator())
[]
```

rows()

Return the list of rows.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [1,2], [0]]
sage: x = dlx_solver(rows)
sage: x.rows()
[[0, 1, 2], [1, 2], [0]]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(1),Integer(2)],
→[Integer(0)]]
>>> x = dlx_solver(rows)
>>> x.rows()
[[0, 1, 2], [1, 2], [0]]
```

search()

Search for a new solution.

Return 1 if a new solution is found and 0 otherwise. To recover the solution, use the method *get_solution()*.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2]]
sage: rows+= [[0,2]]
sage: rows+= [[1]]
sage: rows+= [[3]]
sage: x = dlx_solver(rows)
sage: print(x.search())
1
sage: print(x.get_solution())
[3, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)]]
>>> rows+= [[Integer(0),Integer(2)]]
>>> rows+= [[Integer(1)]]
>>> rows+= [[Integer(3)]]
>>> x = dlx_solver(rows)
>>> print(x.search())
1
>>> print(x.get_solution())
[3, 0]
```

solutions_iterator()

Return an iterator of the solutions.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: sorted(map(sorted, d.solutions_iterator()))
[[0, 1], [2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
↪Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
↪Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]
>>> d = dlx_solver(rows)
>>> sorted(map(sorted, d.solutions_iterator()))
[[0, 1], [2, 3], [4, 5]]
```

split(*column*)

Return a dict of independent solvers.

For each `i`-th row containing a `1` in the `column`, the dict associates the solver giving all solution using the `i`-th row.

This is used for parallel computations.

INPUT:

- `column` – integer; the column used to split the problem into independent subproblems

OUTPUT: dict where keys are row numbers and values are dlx solvers

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: d
Dancing links solver for 6 columns and 6 rows
sage: sorted(map(sorted, d.solutions_iterator()))
[[0, 1], [2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(3),Integer(4),
↪Integer(5)], [Integer(0),Integer(1)], [Integer(2),Integer(3),Integer(4),
↪Integer(5)], [Integer(0)], [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]
>>> d = dlx_solver(rows)
>>> d
Dancing links solver for 6 columns and 6 rows
>>> sorted(map(sorted, d.solutions_iterator()))
[[0, 1], [2, 3], [4, 5]]
```

After the split each subproblem has one more column and the same number of rows as the original problem:

```
sage: D = d.split(0)
sage: D
{0: Dancing links solver for 7 columns and 6 rows,
 2: Dancing links solver for 7 columns and 6 rows,
 4: Dancing links solver for 7 columns and 6 rows}
```

```
>>> from sage.all import *
>>> D = d.split(Integer(0))
>>> D
{0: Dancing links solver for 7 columns and 6 rows,
 2: Dancing links solver for 7 columns and 6 rows,
 4: Dancing links solver for 7 columns and 6 rows}
```

The (disjoint) union of the solutions of the subproblems is equal to the set of solutions shown above:

```
sage: for x in D.values(): sorted(map(sorted, x.solutions_iterator()))
[[0, 1]]
[[2, 3]]
[[4, 5]]
```

```
>>> from sage.all import *
>>> for x in D.values(): sorted(map(sorted, x.solutions_iterator()))
[[0, 1]]
[[2, 3]]
[[4, 5]]
```

to_milp(*solver=None*)

Return the mixed integer linear program (MILP) representing an equivalent problem.

See also `sage.numerical.mip.MixedIntegerLinearProgram`.

INPUT:

- `solver` – string or `None` (default: `None`); possible values include `'GLPK'`, `'GLPK/exact'`, `'Coin'`, `'CPLEX'`, `'Gurobi'`, `'CVXOPT'`, `'PPL'`, `'InteractiveLP'`

OUTPUT:

- MixedIntegerLinearProgram instance
- MIPVariable with binary components

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [0,2], [1], [3]]
sage: d = dlx_solver(rows)
sage: p,x = d.to_milp()                                                     #␣
↪needs sage.numerical.mip
sage: p                                                                     #␣
↪needs sage.numerical.mip
Boolean Program (no objective, 4 variables, ... constraints)
sage: x                                                                     #␣
↪needs sage.numerical.mip
MIPVariable with 4 binary components
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(0),Integer(2)],␣
↪[Integer(1)], [Integer(3)]]
>>> d = dlx_solver(rows)
>>> p,x = d.to_milp()                                                       #␣
↪needs sage.numerical.mip
>>> p                                                                       #␣
↪needs sage.numerical.mip
Boolean Program (no objective, 4 variables, ... constraints)
>>> x                                                                       #␣
↪needs sage.numerical.mip
MIPVariable with 4 binary components
```

In the reduction, the boolean variable x_i is `True` if and only if the i-th row is in the solution:

```
sage: p.show()                                                              #␣
↪needs sage.numerical.mip
Maximization:


Constraints:...
  one 1 in 0-th column: 1.0 <= x_0 + x_1 <= 1.0
  one 1 in 1-th column: 1.0 <= x_0 + x_2 <= 1.0
  one 1 in 2-th column: 1.0 <= x_0 + x_1 <= 1.0
  one 1 in 3-th column: 1.0 <= x_3 <= 1.0
Variables:
  x_0 is a boolean variable (min=0.0, max=1.0)
  x_1 is a boolean variable (min=0.0, max=1.0)
  x_2 is a boolean variable (min=0.0, max=1.0)
  x_3 is a boolean variable (min=0.0, max=1.0)
```

```
>>> from sage.all import *
>>> p.show()                                                                #␣
↪needs sage.numerical.mip
Maximization:
<BLANKLINE>
<BLANKLINE>
Constraints:...
  one 1 in 0-th column: 1.0 <= x_0 + x_1 <= 1.0
```

(continues on next page)

(continued from previous page)

```
  one 1 in 1-th column: 1.0 <= x_0 + x_2 <= 1.0
  one 1 in 2-th column: 1.0 <= x_0 + x_1 <= 1.0
  one 1 in 3-th column: 1.0 <= x_3 <= 1.0
Variables:
  x_0 is a boolean variable (min=0.0, max=1.0)
  x_1 is a boolean variable (min=0.0, max=1.0)
  x_2 is a boolean variable (min=0.0, max=1.0)
  x_3 is a boolean variable (min=0.0, max=1.0)
```

Using some optional MILP solvers:

```
sage: d.to_milp('gurobi')            # optional - gurobi sage_numerical_
↪backends_gurobi, needs sage.numerical.mip
(Boolean Program (no objective, 4 variables, 4 constraints),
 MIPVariable with 4 binary components)
```

```
>>> from sage.all import *
>>> d.to_milp('gurobi')            # optional - gurobi sage_numerical_backends_
↪gurobi, needs sage.numerical.mip
(Boolean Program (no objective, 4 variables, 4 constraints),
 MIPVariable with 4 binary components)
```

to_sat_solver(*solver=None*)

Return the SAT solver solving an equivalent problem.

Note that row index i in the dancing links solver corresponds to the boolean variable index $+1$ for the SAT solver to avoid the variable index 0.

See also `sage.sat.solvers.satsolver`.

INPUT:

- `solver` – string or `None` (default: `None`), possible values include `'picosat'`, `'cryptominisat'`, `'LP'`, `'glucose'`, `'glucose-syrup'`.

OUTPUT: SAT solver instance

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [0,2], [1], [3]]
sage: x = dlx_solver(rows)
sage: s = x.to_sat_solver()                                              #␣
↪needs sage.sat
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)], [Integer(0),Integer(2)],␣
↪[Integer(1)], [Integer(3)]]
>>> x = dlx_solver(rows)
>>> s = x.to_sat_solver()                                              #␣
↪needs sage.sat
```

Using some optional SAT solvers:

```
sage: x.to_sat_solver('cryptominisat')      # optional - pycryptosat        #␣
↪needs sage.sat
CryptoMiniSat solver: 4 variables, 7 clauses.
```

```
>>> from sage.all import *
>>> x.to_sat_solver('cryptominisat')      # optional - pycryptosat        #␣
↪needs sage.sat
CryptoMiniSat solver: 4 variables, 7 clauses.
```

sage.combinat.matrices.dancing_links.**dlx_solver**(*rows*)

Internal-use wrapper for the dancing links C++ code.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2]]
sage: rows+= [[0,2]]
sage: rows+= [[1]]
sage: rows+= [[3]]
sage: x = dlx_solver(rows)
sage: print(x.search())
1
sage: print(x.get_solution())
[3, 0]
sage: print(x.search())
1
sage: print(x.get_solution())
[3, 1, 2]
sage: print(x.search())
0
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.dancing_links import dlx_solver
>>> rows = [[Integer(0),Integer(1),Integer(2)]]
>>> rows+= [[Integer(0),Integer(2)]]
>>> rows+= [[Integer(1)]]
>>> rows+= [[Integer(3)]]
>>> x = dlx_solver(rows)
>>> print(x.search())
1
>>> print(x.get_solution())
[3, 0]
>>> print(x.search())
1
>>> print(x.get_solution())
[3, 1, 2]
>>> print(x.search())
0
```

sage.combinat.matrices.dancing_links.**make_dlxwrapper**(*s*)

Create a dlx wrapper from a Python *string* s.

This was historically used in unpickling and is kept for backwards compatibility. We expect s to be `dumps(rows)` where rows is the list of rows used to instantiate the object.

5.1.136 Dancing links C++ wrapper

sage.combinat.matrices.dlxcpp.**AllExactCovers**(*M*)

Solve the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES: No exact covers:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])                                 #␣
↪needs sage.modules
sage: [cover for cover in AllExactCovers(M)]                                #␣
↪needs sage.modules
[]
```

```
>>> from sage.all import *
>>> M = Matrix([[Integer(1),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(1)],[Integer(0),Integer(1),Integer(1)]])                           ␣
↪      # needs sage.modules
>>> [cover for cover in AllExactCovers(M)]                                #␣
↪needs sage.modules
[]
```

Two exact covers:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]])                         #␣
↪needs sage.modules
sage: [cover for cover in AllExactCovers(M)]                                #␣
↪needs sage.modules
[[(1, 1, 0), (0, 0, 1)], [(1, 0, 1), (0, 1, 0)]]
```

```
>>> from sage.all import *
>>> M = Matrix([[Integer(1),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(1)],[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(1),
↪Integer(0)]])                              # needs sage.modules
>>> [cover for cover in AllExactCovers(M)]                                #␣
↪needs sage.modules
[[(1, 1, 0), (0, 0, 1)], [(1, 0, 1), (0, 1, 0)]]
```

sage.combinat.matrices.dlxcpp.**DLXCPP**(*rows*)

Solve the Exact Cover problem by using the Dancing Links algorithm described by Knuth.

Consider a matrix M with entries of 0 and 1, and compute a subset of the rows of this matrix which sum to the vector of all 1s.

The dancing links algorithm works particularly well for sparse matrices, so the input is a list of lists of the form:

```
[
 [i_11,i_12,...,i_1r]
 ...
 [i_m1,i_m2,...,i_ms]
]
```

where M[j][i_jk] = 1.

The first example below corresponds to the matrix:

```
1110
1010
0100
0001
```

which is exactly covered by:

```
1110
0001
```

and

```
1010
0100
0001
```

If soln is a solution given by DLXCPP(rows) then

[rows[soln[0]], rows[soln[1]], … rows[soln[len(soln)-1]]]

is an exact cover.

Solutions are given as a list.

EXAMPLES:

```
sage: rows = [[0,1,2]]
sage: rows+= [[0,2]]
sage: rows+= [[1]]
sage: rows+= [[3]]
sage: [x for x in DLXCPP(rows)]
[[3, 0], [3, 1, 2]]
```

```
>>> from sage.all import *
>>> rows = [[Integer(0),Integer(1),Integer(2)]]
>>> rows+= [[Integer(0),Integer(2)]]
>>> rows+= [[Integer(1)]]
>>> rows+= [[Integer(3)]]
>>> [x for x in DLXCPP(rows)]
[[3, 0], [3, 1, 2]]
```

sage.combinat.matrices.dlxcpp.**OneExactCover**(*M*)

Solve the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: # needs sage.modules
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])   # no exact covers
sage: print(OneExactCover(M))
None
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]]) # two exact covers
sage: OneExactCover(M)
[(1, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> M = Matrix([[Integer(1),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(1)],[Integer(0),Integer(1),Integer(1)]])  # no exact covers
>>> print(OneExactCover(M))
None
>>> M = Matrix([[Integer(1),Integer(1),Integer(0)],[Integer(1),Integer(0),
↪Integer(1)],[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(1),
↪Integer(0)]]) # two exact covers
>>> OneExactCover(M)
[(1, 1, 0), (0, 0, 1)]
```

5.1.137 Hadamard matrices

A Hadamard matrix is an $n \times n$ matrix H whose entries are either $+1$ or -1 and whose rows are mutually orthogonal. For example, the matrix H_2 defined by

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

is a Hadamard matrix. An $n \times n$ matrix H whose entries are either $+1$ or -1 is a Hadamard matrix if and only if:

(a) $|det(H)| = n^{n/2}$ or

(b) $H * H^t = n \cdot I_n$, where I_n is the identity matrix.

In general, the tensor product of an $m \times m$ Hadamard matrix and an $n \times n$ Hadamard matrix is an $(mn) \times (mn)$ matrix. In particular, if there is an $n \times n$ Hadamard matrix then there is a $(2n) \times (2n)$ Hadamard matrix (since one may tensor with H_2). This particular case is sometimes called the Sylvester construction.

The Hadamard conjecture (possibly due to Paley) states that a Hadamard matrix of order n exists if and only if $n = 1, 2$ or n is a multiple of 4.

The module below implements constructions of Hadamard and skew Hadamard matrices for all known orders ≤ 1200, plus some more greater than 1200. It also allows you to pull a Hadamard matrix from the database at [SloaHada].

The following code will test that a construction for all known orders $\leq 4k$ is implemented. The assertion above can be verified by setting `k=300` (note that it will take a long time to run):

```
sage: from sage.combinat.matrices.hadamard_matrix import (hadamard_matrix,
....:                                     skew_hadamard_matrix, is_hadamard_matrix,
....:                                     is_skew_hadamard_matrix)
sage: k = 20
sage: unknown_hadamard = [668, 716, 892, 1132]
sage: unknown_skew_hadamard = [356, 404, 428, 476, 596, 612, 668, 708, 712, 716,
....:                          764, 772, 804, 808, 820, 836, 856, 892, 900, 916,
....:                          932, 940, 952, 980, 996, 1004, 1012, 1028, 1036,
....:                          1044, 1060, 1076, 1100, 1108, 1132, 1140, 1148,
....:                          1156, 1180, 1192, 1196]
sage: for n in range(1, k+1):
....:   if 4*n not in unknown_hadamard:
....:       H = hadamard_matrix(4*n, check=False)
....:       assert is_hadamard_matrix(H)
....:   if 4*n not in unknown_skew_hadamard:
....:       H = skew_hadamard_matrix(4*n, check=False)
....:       assert is_skew_hadamard_matrix(H)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import (hadamard_matrix,
...                                  skew_hadamard_matrix, is_hadamard_matrix,
...                                  is_skew_hadamard_matrix)
>>> k = Integer(20)
>>> unknown_hadamard = [Integer(668), Integer(716), Integer(892), Integer(1132)]
>>> unknown_skew_hadamard = [Integer(356), Integer(404), Integer(428), Integer(476),
↪Integer(596), Integer(612), Integer(668), Integer(708), Integer(712), Integer(716),
...                          Integer(764), Integer(772), Integer(804), Integer(808),
↪Integer(820), Integer(836), Integer(856), Integer(892), Integer(900), Integer(916),
...                          Integer(932), Integer(940), Integer(952), Integer(980),
↪Integer(996), Integer(1004), Integer(1012), Integer(1028), Integer(1036),
...                          Integer(1044), Integer(1060), Integer(1076),
↪Integer(1100), Integer(1108), Integer(1132), Integer(1140), Integer(1148),
...                          Integer(1156), Integer(1180), Integer(1192),
↪Integer(1196)]
>>> for n in range(Integer(1), k+Integer(1)):
...   if Integer(4)*n not in unknown_hadamard:
...       H = hadamard_matrix(Integer(4)*n, check=False)
...       assert is_hadamard_matrix(H)
...   if Integer(4)*n not in unknown_skew_hadamard:
...       H = skew_hadamard_matrix(Integer(4)*n, check=False)
...       assert is_skew_hadamard_matrix(H)
```

AUTHORS:

- David Joyner (2009-05-17): initial version
- Matteo Cati (2023-03-18): implemented more constructions for Hadamard and skew Hadamard matrices, to cover all known orders up to 1200.

REFERENCES:

- [SloaHada]
- [HadaWiki]
- [Hora]
- [CP2023]

`sage.combinat.matrices.hadamard_matrix.`**`GS_skew_hadamard_smallcases`**(*n*, *existence=False*, *check=True*)

Data for Williamson-Goethals-Seidel construction of skew Hadamard matrices.

Here we keep the data for this construction. Namely, it needs 4 circulant matrices with extra properties, as described in `sage.combinat.matrices.hadamard_matrix.williamson_goethals_seidel_skew_hadamard_matrix()` Matrices are taken from:

- $n = 36, 52$: [GS70s]
- $n = 92$: [Wall71]
- $n = 188$: [Djo2008a]
- $n = 236$: [FKS2004]
- $n = 276$: [Djo2023a]

Additional data is obtained from skew supplementary difference sets contained in *sage.combinat.designs.difference_family.skew_supplementary_difference_set()*, using the construction described in [Djo1992a].

INPUT:

- `n` – integer; the order of the matrix
- `existence` – boolean (default: `True`); if `True`, only check that we can do the construction
- `check` – boolean (default: `False`); if `True`, check the result

`sage.combinat.matrices.hadamard_matrix.`**`RSHCD_324`**`(e)`

Return a size 324x324 Regular Symmetric Hadamard Matrix with Constant Diagonal.

We build the matrix M for the case $n = 324$, $\epsilon = 1$ directly from `JankoKharaghaniTonchevGraph` and for the case $\epsilon = -1$ from the "twist" M' of M, using Lemma 11 in [HX2010]. Namely, it turns out that the matrix

$$M' = \begin{pmatrix} M_{12} & M_{11} \\ M_{11}^\top & M_{21} \end{pmatrix}, \quad \text{where} \quad M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix},$$

and the M_{ij} are 162x162-blocks, also RSHCD, its diagonal blocks having zero row sums, as needed by [loc.cit.]. Interestingly, the corresponding $(324, 152, 70, 72)$-strongly regular graph has a vertex-transitive automorphism group of order 2592, twice the order of the (intransitive) automorphism group of the graph corresponding to M. Cf. [CP2016].

INPUT:

- `e` – -1 or $+1$; the value of ϵ

REFERENCE:

- [CP2016]

`sage.combinat.matrices.hadamard_matrix.`**`amicable_hadamard_matrices`**`(n, existence=False, check=True)`

Construct amicable Hadamard matrices of order `n` using the available methods.

INPUT:

- `n` – positive integer; the order of the amicable Hadamard matrices
- `existence` – boolean (default: `False`); if `True`, only return whether amicable Hadamard matrices of order n can be constructed
- `check` – boolean (default: `True`); if `True`, check that the matrices are amicable Hadamard matrices before returning them

OUTPUT:

If `existence` is true, the function returns a boolean representing whether amicable Hadamard matrices of order n can be constructed. If `existence` is false, returns two amicable Hadamard matrices, or raises an error if the matrices cannot be constructed.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import amicable_hadamard_
↪matrices
sage: amicable_hadamard_matrices(2)
(
[ 1  1]  [ 1  1]
[-1  1], [ 1 -1]
)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import amicable_hadamard_matrices
>>> amicable_hadamard_matrices(Integer(2))
(
[ 1  1]  [ 1  1]
[-1  1], [ 1 -1]
)
```

If `existence` is true, the function returns a boolean:

```
sage: amicable_hadamard_matrices(16, existence=True)
False
```

```
>>> from sage.all import *
>>> amicable_hadamard_matrices(Integer(16), existence=True)
False
```

`sage.combinat.matrices.hadamard_matrix.`**`amicable_hadamard_matrices_wallis`**`(`*n*, *check=True*`)`

Construct amicable Hadamard matrices of order $n = q + 1$ where q is a prime power.

If q is a prime power $\equiv 3 \mod 4$, then amicable Hadamard matrices of order $q+1$ can be constructed as described in [Wal1970b].

INPUT:

- `n` – integer; the order of the matrices to be constructed
- `check` – boolean (default: `True`); if `True`, check that the resulting matrices are amicable Hadamard before returning them

OUTPUT:

The function returns two amicable Hadamard matrices, or raises an error if such matrices cannot be created using this construction.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import amicable_hadamard_
↪matrices_wallis
sage: M, N = amicable_hadamard_matrices_wallis(28)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import amicable_hadamard_matrices_
↪wallis
>>> M, N = amicable_hadamard_matrices_wallis(Integer(28))
```

`sage.combinat.matrices.hadamard_matrix.`**`are_amicable_hadamard_matrices`**`(`*M*, *N*, *verbose=False*`)`

Check if `M` and `N` are amicable Hadamard matrices.

Two matrices M and N of order n are called amicable if they satisfy the following conditions (see [Seb2017]):

- M is a skew Hadamard matrix
- N is a symmetric Hadamard matrix
- $MN^T = NM^T$

INPUT:

- `M` – a square matrix
- `N` – a square matrix
- `verbose` – boolean (default: `False`); whether to be verbose when the matrices are not amicable Hadamard matrices

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import are_amicable_hadamard_
↪matrices
sage: M = matrix([[1, 1], [-1, 1]])
sage: N = matrix([[1, 1], [1, -1]])
sage: are_amicable_hadamard_matrices(M, N)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import are_amicable_hadamard_
↪matrices
>>> M = matrix([[Integer(1), Integer(1)], [-Integer(1), Integer(1)]])
>>> N = matrix([[Integer(1), Integer(1)], [Integer(1), -Integer(1)]])
>>> are_amicable_hadamard_matrices(M, N)
True
```

If `verbose` is true, the function will be verbose when returning False:

```
sage: N = matrix([[1, 1], [1, 1]])
sage: are_amicable_hadamard_matrices(M, N, verbose=True)
The second matrix is not Hadamard
False
```

```
>>> from sage.all import *
>>> N = matrix([[Integer(1), Integer(1)], [Integer(1), Integer(1)]])
>>> are_amicable_hadamard_matrices(M, N, verbose=True)
The second matrix is not Hadamard
False
```

`sage.combinat.matrices.hadamard_matrix.`**`construction_four_symbol_delta_code_I`**`(`X, Y, Z, W`)`

Construct 4-symbol δ code of length $2n + 1$.

The 4-symbol δ code is constructed from sequences X, Y, Z, W of length $n + 1$, $n + 1$, n, n satisfying for all $s > 0$:

$$N_X(s) + N_Y(s) + N_Z(s) + N_W(s) = 0$$

where $N_A(s)$ is the nonperiodic correlation function:

$$N_A(s) = \sum_{i=1}^{n-s} a_i a_{i+s}$$

The construction (detailed in [Tur1974]) is as follows:

$$\begin{aligned} T_1 &= X; Z \\ T_2 &= X; -Z \\ T_3 &= Y; W \\ T_4 &= Y; -W \end{aligned}$$

INPUT:

- X – list; the first sequence (length $n+1$)
- Y – list; the second sequence (length $n+1$)
- Z – list; the third sequence (length n)
- W – list; the fourth sequence (length n)

OUTPUT: tuple containing the 4-symbol δ code of length $2n+1$

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import construction_four_symbol_
↪delta_code_I
sage: construction_four_symbol_delta_code_I([1, 1], [1, -1], [1], [1])
([1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1])
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import construction_four_symbol_
↪delta_code_I
>>> construction_four_symbol_delta_code_I([Integer(1), Integer(1)], [Integer(1), -
↪Integer(1)], [Integer(1)], [Integer(1)])
([1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1])
```

sage.combinat.matrices.hadamard_matrix.**construction_four_symbol_delta_code_II**(X, Y, Z, W)

Construct 4-symbol δ code of length $4n+3$.

The 4-symbol δ code is constructed from sequences X, Y, Z, W of length $n+1$, $n+1$, n, n satisfying for all $s>0$:

$$N_X(s)+N_Y(s)+N_Z(s)+N_W(s)=0$$

where $N_A(s)$ is the nonperiodic correlation function:

$$N_A(s)=\sum_{i=1}^{n-s}a_ia_{i+s}$$

The construction (detailed in [Tur1974]) is as follows (writing A/B to mean A alternated with B):

$$\begin{aligned}T_1&=X/Z;Y/W;1\\T_2&=X/Z;Y/-W;-1\\T_3&=X/Z;-Y/-W;1\\T_4&=X/Z;-Y/W;-1\end{aligned}$$

INPUT:

- X – list; the first sequence (length $n+1$)
- Y – list; the second sequence (length $n+1$)
- Z – list; the third sequence (length n)
- W – list; the fourth sequence (length n)

OUTPUT: tuple containing the four 4-symbol δ code of length $4n+3$

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import construction_four_symbol_
↪delta_code_II
sage: construction_four_symbol_delta_code_II([1, 1], [1, -1], [1], [1])
([1, 1, 1, 1, 1, -1, 1],
 [1, 1, 1, 1, -1, -1, -1],
 [1, 1, 1, -1, -1, 1, 1],
 [1, 1, 1, -1, 1, 1, -1])
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import construction_four_symbol_
↪delta_code_II
>>> construction_four_symbol_delta_code_II([Integer(1), Integer(1)], [Integer(1),␣
↪-Integer(1)], [Integer(1)], [Integer(1)])
([1, 1, 1, 1, 1, -1, 1],
 [1, 1, 1, 1, -1, -1, -1],
 [1, 1, 1, -1, -1, 1, 1],
 [1, 1, 1, -1, 1, 1, -1])
```

sage.combinat.matrices.hadamard_matrix.**four_symbol_delta_code_smallcases**(*n*, *existence=False*)

Return the 4-symobl δ code of length `n` if available.

The 4-symbol δ codes are constructed using *construction_four_symbol_delta_code_I()* or *construction_four_symbol_delta_code_II()*. The base sequences used are taken from [Tur1974].

INPUT:

- `n` – integer; the length of the desired 4-symbol δ code
- `existence` – boolean (default: `False`); if `True`, only check if the sequences are available

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import four_symbol_delta_code_
↪smallcases
sage: four_symbol_delta_code_smallcases(3)
([1, -1, 1], [1, -1, -1], [1, 1, 1], [1, 1, -1])
sage: four_symbol_delta_code_smallcases(3, existence=True)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import four_symbol_delta_code_
↪smallcases
>>> four_symbol_delta_code_smallcases(Integer(3))
([1, -1, 1], [1, -1, -1], [1, 1, 1], [1, 1, -1])
>>> four_symbol_delta_code_smallcases(Integer(3), existence=True)
True
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix**(*n*, *existence=False*, *check=True*, *construction_name=False*)

This function is available as hadamard_matrix(…) and matrix.hadamard(…).

Try to construct a Hadamard matrix using the available methods.

Currently all orders ≤ 1200 for which a construction is known are implemented. For $n > 1200$, only some orders are available.

INPUT:

- `n` – integer; dimension of the matrix
- `existence` – boolean (default: `False`); whether to build the matrix or merely query if a construction is available in Sage. When set to `True`, the function returns:
  - `True` – meaning that Sage knows how to build the matrix
  - `Unknown` – meaning that Sage does not know how to build the matrix, although the matrix may exist (see `sage.misc.unknown`).
  - `False` – meaning that the matrix does not exist
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- `construction_name` – boolean (default: `False`); if it is `True`, `existence` is `True`, and a matrix exists, output the construction name. It has no effect if `existence` is set to `False`.

EXAMPLES:

```
sage: hadamard_matrix(12).det()
2985984
sage: 12^6
2985984
sage: hadamard_matrix(1)
[1]
sage: hadamard_matrix(2)
[ 1  1]
[ 1 -1]
sage: hadamard_matrix(8) # random
[ 1  1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1 -1]
[ 1  1 -1 -1  1  1 -1 -1]
[ 1 -1 -1  1  1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1  1 -1  1]
[ 1  1 -1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1  1  1 -1]
sage: hadamard_matrix(8).det() == 8^4
True
```

```
>>> from sage.all import *
>>> hadamard_matrix(Integer(12)).det()
2985984
>>> Integer(12)**Integer(6)
2985984
>>> hadamard_matrix(Integer(1))
[1]
>>> hadamard_matrix(Integer(2))
[ 1  1]
[ 1 -1]
>>> hadamard_matrix(Integer(8)) # random
[ 1  1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1 -1]
[ 1  1 -1 -1  1  1 -1 -1]
```

(continues on next page)

(continued from previous page)

```
[ 1 -1 -1  1  1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1  1 -1  1]
[ 1  1 -1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1  1  1 -1]
>>> hadamard_matrix(Integer(8)).det() == Integer(8)**Integer(4)
True
```

We note that `hadamard_matrix()` returns a normalised Hadamard matrix (the entries in the first row and column are all +1)

```
sage: hadamard_matrix(12) # random
[ 1  1| 1  1| 1  1| 1  1| 1  1| 1  1]
[ 1 -1|-1  1|-1  1|-1  1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1 -1| 1  1|-1 -1|-1 -1| 1  1]
[ 1  1|-1 -1| 1 -1|-1  1|-1  1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1| 1 -1| 1  1|-1 -1|-1 -1]
[ 1  1| 1 -1|-1 -1| 1 -1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1| 1  1| 1 -1| 1  1|-1 -1]
[ 1  1|-1  1| 1 -1|-1 -1| 1 -1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1|-1 -1| 1  1| 1 -1| 1  1]
[ 1  1|-1  1|-1  1| 1 -1|-1 -1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1|-1 -1|-1 -1| 1  1| 1 -1]
[ 1  1| 1 -1|-1  1|-1  1| 1 -1|-1 -1]
```

```
>>> from sage.all import *
>>> hadamard_matrix(Integer(12)) # random
[ 1  1| 1  1| 1  1| 1  1| 1  1| 1  1]
[ 1 -1|-1  1|-1  1|-1  1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1 -1| 1  1|-1 -1|-1 -1| 1  1]
[ 1  1|-1 -1| 1 -1|-1  1|-1  1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1| 1 -1| 1  1|-1 -1|-1 -1]
[ 1  1| 1 -1|-1 -1| 1 -1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1| 1  1| 1 -1| 1  1|-1 -1]
[ 1  1|-1  1| 1 -1|-1 -1| 1 -1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1|-1 -1| 1  1| 1 -1| 1  1]
[ 1  1|-1  1|-1  1| 1 -1|-1 -1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1|-1 -1|-1 -1| 1  1| 1 -1]
[ 1  1| 1 -1|-1  1|-1  1| 1 -1|-1 -1]
```

To find how the matrix is obtained, use `construction_name`

```
sage: matrix.hadamard(476, existence=True, construction_name=True)
'cooper-wallis small cases: 476'
```

```
>>> from sage.all import *
>>> matrix.hadamard(Integer(476), existence=True, construction_name=True)
'cooper-wallis small cases: 476'
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_156**()

Construct a Hadamard matrix of order 156.

The matrix is created using the construction detailed in [BH1965]. This uses four circulant matrices of size 13×13, which are composed into a 156×156 block matrix.

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_cooper_wallis_construction**(*x1*, *x2*, *x3*, *x4*, *A*, *B*, *C*, *D*, *check=True*)

Create a Hadamard matrix using the construction detailed in [CW1972].

Given four circulant matrices X_1, X_2, X_3, X_4` of order n with entries (0, 1, -1) such that the entrywise product of two distinct matrices is always equal to 0 and that $\sum_{i=1}^{4} X_iX_i^\top = nI_n$ holds, and four matrices A, B, C, D of order m with elements (1, -1) such that $MN^\top = NM^\top$ for all distinct M, N and $AA^\top + BB^\top + CC^\top + DD^\top = 4mI_n$ holds, we construct a Hadamard matrix of order $4nm$.

INPUT:

- `x1` – list or vector; the first row of the circulant matrix X_1
- `x2` – list or vector; the first row of the circulant matrix X_2
- `x3` – list or vector; the first row of the circulant matrix X_3
- `x4` – list or vector; the first row of the circulant matrix X_4
- `A` – the matrix described above
- `B` – the matrix described above
- `C` – the matrix described above
- `D` – the matrix described above
- `check` – boolean (default: `True`); if `True`, check that the resulting matrix is Hadamard before returning it

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_cooper_
↪wallis_construction
sage: from sage.combinat.t_sequences import T_sequences_smallcases
sage: seqs = T_sequences_smallcases(19)
sage: hadamard_matrix_cooper_wallis_construction(seqs[0], seqs[1], seqs[2],␣
↪seqs[3], matrix([1]), matrix([1]), matrix([1]), matrix([1]))
76 x 76 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_cooper_
↪wallis_construction
>>> from sage.combinat.t_sequences import T_sequences_smallcases
```

(continues on next page)

(continued from previous page)

```
>>> seqs = T_sequences_smallcases(Integer(19))
>>> hadamard_matrix_cooper_wallis_construction(seqs[Integer(0)], seqs[Integer(1)],
↪ seqs[Integer(2)], seqs[Integer(3)], matrix([Integer(1)]), matrix([Integer(1)]),
↪ matrix([Integer(1)]), matrix([Integer(1)]))
76 x 76 dense matrix over Integer Ring...
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_cooper_wallis_smallcases**(*n*, *check=True*, *existence=False*)

Construct Hadamard matrices using the Cooper-Wallis construction for some small values of n.

This function calls the function *hadamard_matrix_cooper_wallis_construction()* with the appropriate arguments. It constructs the matrices X_1, X_2, X_3, X_4 using either T-matrices or the T-sequences from *sage.combinat.t_sequences.T_sequences_smallcases()*. The matrices A, B, C, D are taken from *williamson_type_quadruples_smallcases()*.

Data for T-matrices of order 67 is taken from [Saw1985].

INPUT:

- `n` – integer; the order of the matrix to be constructed
- `check` – boolean (default: `True`); if `True`, check that the matrix is a Hadamard matrix before returning
- `existence` – boolean (default: `False`); if `True`, only check if the matrix exists

OUTPUT:

If `existence=False`, returns the Hadamard matrix of order n. It raises an error if no data is available to construct the matrix of the given order. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

See also

hadamard_matrix_cooper_wallis_construction()

EXAMPLES:

By default The function returns the Hadamard matrix

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_cooper_
↪wallis_smallcases
sage: hadamard_matrix_cooper_wallis_smallcases(28)
28 x 28 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_cooper_
↪wallis_smallcases
>>> hadamard_matrix_cooper_wallis_smallcases(Integer(28))
28 x 28 dense matrix over Integer Ring...
```

If `existence` is set to True, the function returns a boolean

```
sage: hadamard_matrix_cooper_wallis_smallcases(20, existence=True)
True
```

```
>>> from sage.all import *
>>> hadamard_matrix_cooper_wallis_smallcases(Integer(20), existence=True)
True
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_from_sds**(*n*, *existence=False*, *check=True*)

Construction of Hadamard matrices from supplementary difference sets.

Given four SDS with parameters $4-\{n; n_1, n_2, n_3, n_4; \lambda\}$ with $n_1 + n_2 + n_3 + n_4 = n + \lambda$ we can construct four (-1,+1) sequences $a_i = (a_{i,0}, ..., a_{i,n-1})$ where $a_{i,j} = -1$ iff $j \in S_i$. These will be the fist rows of four circulant matrices A_1, A_2, A_3, A_4 which, when plugged into the Goethals-Seidel array, create an Hadamard matrix of order $4n$ (see [Djo1994b]).

The supplementary difference sets are taken from *sage.combinat.designs.difference_family.supplementary_difference_set()*.

INPUT:

- `n` – integer; the order of the matrix to be constructed
- `check` – boolean (default: `True`); if `True`, check that the matrix is a Hadamard before returning
- `existence` – boolean (default: `False`); if `True`, only check if the matrix exists

OUTPUT:

If `existence=False`, returns the Hadamard matrix of order n. It raises an error if no data is available to construct the matrix of the given order, or if n is not a multiple of 4. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

EXAMPLES:

By default The function returns the Hadamard matrix

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_from_sds
sage: hadamard_matrix_from_sds(148)
148 x 148 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_from_sds
>>> hadamard_matrix_from_sds(Integer(148))
148 x 148 dense matrix over Integer Ring...
```

If `existence` is set to True, the function returns a boolean

```
sage: hadamard_matrix_from_sds(764, existence=True)
True
```

```
>>> from sage.all import *
>>> hadamard_matrix_from_sds(Integer(764), existence=True)
True
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_from_symmetric_conference_matrix**(*n*, *existence=False*, *check=True*)

Construct a Hadamard matrix of order n from a symmetric conference matrix of order $n/2$.

The construction is described in Theorem 4.3.24 of [IS2006]. The symmetric conference matrices are obtained from *sage.combinat.matrices.hadamard_matrix.symmetric_conference_matrix()*.

INPUT:

- `n` – integer; the order of the matrix to be constructed
- `existence` – boolean (default: `False`); if `True`, only check if the matrix exists
- `check` – boolean (default: `True`); if `True`, check that the matrix is a Hadamard before returning

OUTPUT:

If `existence=False`, returns the Hadamard matrix of order n. It raises an error if no data is available to construct the matrix of the given order, or if n does not satisfies the constraints. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

EXAMPLES:

By default the function returns the Hadamard matrix

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_from_
↪symmetric_conference_matrix
sage: hadamard_matrix_from_symmetric_conference_matrix(20)
20 x 20 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_from_
↪symmetric_conference_matrix
>>> hadamard_matrix_from_symmetric_conference_matrix(Integer(20))
20 x 20 dense matrix over Integer Ring...
```

If `existence` is set to True, the function returns True if the matrix exists, False if the conference matrix does not exist, and Unknown if the conference matrix cannot be constructed yet

```
sage: hadamard_matrix_from_symmetric_conference_matrix(12, existence=True)
True
sage: hadamard_matrix_from_symmetric_conference_matrix(4*787, existence=True)
True
```

```
>>> from sage.all import *
>>> hadamard_matrix_from_symmetric_conference_matrix(Integer(12), existence=True)
True
>>> hadamard_matrix_from_symmetric_conference_matrix(Integer(4)*Integer(787),␣
↪existence=True)
True
```

`sage.combinat.matrices.hadamard_matrix.`**`hadamard_matrix_miyamoto_construction`**`(`*n*, *existence=False*, *check=True*`)`

Construct Hadamard matrix using the Miyamoto construction.

If $q = n/4$ is a prime power, and there exists a Hadamard matrix of order $q - 1$, then a Hadamard matrix of order n can be constructed (see [Miy1991]).

INPUT:

- `n` – integer; the order of the matrix to be constructed
- `check` – boolean (default: `True`); if `True`, check that the matrix is a Hadamard before returning
- `existence` – boolean (default: `False`); if `True`, only check if the matrix exists

OUTPUT:

If `existence=False`, returns the Hadamard matrix of order n. It raises an error if no data is available to construct the matrix of the given order, or if n does not satisfies the constraints. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

EXAMPLES:

By default the function returns the Hadamard matrix

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_miyamoto_
↪construction
sage: hadamard_matrix_miyamoto_construction(20)
20 x 20 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_miyamoto_
↪construction
>>> hadamard_matrix_miyamoto_construction(Integer(20))
20 x 20 dense matrix over Integer Ring...
```

If `existence` is set to True, the function returns a boolean

```
sage: hadamard_matrix_miyamoto_construction(36, existence=True)
True
```

```
>>> from sage.all import *
>>> hadamard_matrix_miyamoto_construction(Integer(36), existence=True)
True
```

`sage.combinat.matrices.hadamard_matrix.`**`hadamard_matrix_paleyI`**(*n*, *normalize=True*)

Implement the Paley type I construction.

The Paley type I case corresponds to the case $p = n - 1 \cong 3 \mod 4$ for a prime power p (see [Hora]).

INPUT:

- `n` – the matrix size
- `normalize` – boolean (default: `True`); whether to normalize the result

EXAMPLES:

We note that this method by default returns a normalised Hadamard matrix

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_paleyI
sage: hadamard_matrix_paleyI(4)
[ 1  1  1  1]
[ 1 -1  1 -1]
[ 1 -1 -1  1]
[ 1  1 -1 -1]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_paleyI
>>> hadamard_matrix_paleyI(Integer(4))
[ 1  1  1  1]
[ 1 -1  1 -1]
[ 1 -1 -1  1]
[ 1  1 -1 -1]
```

Otherwise, it returns a skew Hadamard matrix H, i.e. $H = S + I$, with $S = -S^\top$

```
sage: M = hadamard_matrix_paleyI(4, normalize=False); M
[ 1  1  1  1]
[-1  1  1 -1]
[-1 -1  1  1]
[-1  1 -1  1]
sage: S = M - identity_matrix(4); -S == S.T
True
```

```
>>> from sage.all import *
>>> M = hadamard_matrix_paleyI(Integer(4), normalize=False); M
[ 1  1  1  1]
[-1  1  1 -1]
[-1 -1  1  1]
[-1  1 -1  1]
>>> S = M - identity_matrix(Integer(4)); -S == S.T
True
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_paleyII**(*n*)

Implement the Paley type II construction.

The Paley type II case corresponds to the case $p = n/2 - 1 \cong 1 \mod 4$ for a prime power p (see [Hora]).

EXAMPLES:

```
sage: sage.combinat.matrices.hadamard_matrix.hadamard_matrix_paleyII(12).det()
2985984
sage: 12^6
2985984
```

```
>>> from sage.all import *
>>> sage.combinat.matrices.hadamard_matrix.hadamard_matrix_paleyII(Integer(12)).
↪det()
2985984
>>> Integer(12)**Integer(6)
2985984
```

We note that the method returns a normalised Hadamard matrix

```
sage: sage.combinat.matrices.hadamard_matrix.hadamard_matrix_paleyII(12)
[ 1  1| 1  1| 1  1| 1  1| 1  1| 1  1]
[ 1 -1|-1  1|-1  1|-1  1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1 -1| 1  1|-1 -1|-1 -1| 1  1]
```

(continues on next page)

(continued from previous page)

```
[ 1  1|-1 -1| 1 -1|-1  1|-1  1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1| 1 -1| 1  1|-1 -1|-1 -1]
[ 1  1| 1 -1|-1 -1| 1 -1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1| 1  1| 1 -1| 1  1|-1 -1]
[ 1  1|-1  1| 1 -1|-1 -1| 1 -1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1|-1 -1| 1  1| 1 -1| 1  1]
[ 1  1|-1  1|-1  1| 1 -1|-1 -1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1|-1 -1|-1 -1| 1  1| 1 -1]
[ 1  1| 1 -1|-1  1|-1  1| 1 -1|-1 -1]
```

```
>>> from sage.all import *
>>> sage.combinat.matrices.hadamard_matrix.hadamard_matrix_paleyII(Integer(12))
[ 1  1| 1  1| 1  1| 1  1| 1  1| 1  1]
[ 1 -1|-1  1|-1  1|-1  1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1 -1| 1  1|-1 -1|-1 -1| 1  1]
[ 1  1|-1 -1| 1 -1|-1  1|-1  1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1| 1 -1| 1  1|-1 -1|-1 -1]
[ 1  1| 1 -1|-1 -1| 1 -1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1| 1  1| 1 -1| 1  1|-1 -1]
[ 1  1|-1  1| 1 -1|-1 -1| 1 -1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1|-1 -1| 1  1| 1 -1| 1  1]
[ 1  1|-1  1|-1  1| 1 -1|-1 -1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1|-1 -1|-1 -1| 1  1| 1 -1]
[ 1  1| 1 -1|-1  1|-1  1| 1 -1|-1 -1]
```

`sage.combinat.matrices.hadamard_matrix.`**`hadamard_matrix_spence_construction`**`(`*n*, *existence=False*, *check=True*`)`

Create a Hadamard matrix of order n using the Spence construction.

This construction (detailed in [Spe1975]), uses supplementary difference sets implemented in *`sage.combinat.designs.difference_family.supplementary_difference_set_from_rel_diff_set()`* to create the desired matrix.

INPUT:

- `n` – integer; the order of the matrix to be constructed
- `existence` – boolean (default: `False`); if `True`, only check if the matrix exists
- `check` – boolean (default: `True`); if `True`, check that the matrix is a Hadamard matrix before returning

OUTPUT:

If `existence=True`, returns a boolean representing whether the Hadamard matrix can be constructed. Otherwise, returns the Hadamard matrix, or raises an error if it cannot be constructed.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_spence_
↪construction
sage: hadamard_matrix_spence_construction(36)
36 x 36 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_spence_
↪construction
>>> hadamard_matrix_spence_construction(Integer(36))
36 x 36 dense matrix over Integer Ring...
```

If `existence` is `True`, the function returns a boolean

```
sage: hadamard_matrix_spence_construction(52, existence=True)
True
```

```
>>> from sage.all import *
>>> hadamard_matrix_spence_construction(Integer(52), existence=True)
True
```

`sage.combinat.matrices.hadamard_matrix.`**`hadamard_matrix_turyn_type`**(*a*, *b*, *c*, *d*, *e1*, *e2*, *e3*, *e4*, *check=True*)

Construction of Turyn type Hadamard matrix.

Given $n \times n$ circulant matrices A, B, C, D with 1,-1 entries, satisfying $AA^\top + BB^\top + CC^\top + DD^\top = 4nI$, and a set of Baumert-Hall units of order $4t$, one can construct a Hadamard matrix of order $4tn$ as detailed by Turyn in [Tur1974].

INPUT:

- `a` – 1,-1 list; the 1st row of A
- `b` – 1,-1 list; the 1st row of B
- `d` – 1,-1 list; the 1st row of C
- `c` – 1,-1 list; the 1st row of D
- `e1` – matrix; the first Baumert-Hall unit
- `e2` – matrix; the second Baumert-Hall unit
- `e3` – matrix; the third Baumert-Hall unit
- `e4` – matrix; the fourth Baumert-Hall unit
- `check` – boolean (default: `True`); whether to check that the output is a Hadamard matrix before returning it

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_turyn_
↪type, _get_baumert_hall_units
sage: A, B, C, D = _get_baumert_hall_units(28)
sage: hadamard_matrix_turyn_type([1], [1], [1], [1], A, B, C, D)
28 x 28 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_turyn_type,
↪ _get_baumert_hall_units
>>> A, B, C, D = _get_baumert_hall_units(Integer(28))
>>> hadamard_matrix_turyn_type([Integer(1)], [Integer(1)], [Integer(1)],
↪[Integer(1)], A, B, C, D)
28 x 28 dense matrix over Integer Ring...
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_williamson_type**(*a*, *b*, *c*, *d*, *check=True*)

Construction of Williamson type Hadamard matrix.

Given $n \times n$ circulant matrices A, B, C, D with 1,-1 entries, and satisfying $AA^\top + BB^\top + CC^\top + DD^\top = 4nI$, one can construct a Hadamard matrix of order $4n$, cf. [Ha83].

INPUT:

- a – (1,-1) list; the 1st row of A
- b – (1,-1) list; the 1st row of B
- d – (1,-1) list; the 1st row of C
- c – (1,-1) list; the 1st row of D
- check – boolean (default: True); whether to check that the output is a Hadamard matrix before returning it

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_
↪williamson_type
sage: a = [ 1,  1, 1]
sage: b = [ 1, -1, -1]
sage: c = [ 1, -1, -1]
sage: d = [ 1, -1, -1]
sage: M = hadamard_matrix_williamson_type(a,b,c,d,check=True)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_williamson_
↪type
>>> a = [ Integer(1),  Integer(1), Integer(1)]
>>> b = [ Integer(1), -Integer(1), -Integer(1)]
>>> c = [ Integer(1), -Integer(1), -Integer(1)]
>>> d = [ Integer(1), -Integer(1), -Integer(1)]
>>> M = hadamard_matrix_williamson_type(a,b,c,d,check=True)
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_www**(*url_file*, *comments=False*)

Pull file from Sloane's database and return the corresponding Hadamard matrix as a Sage matrix.

You must input a filename of the form "had.n.xxx.txt" as described on the webpage http://neilsloane.com/hadamard/, where "xxx" could be empty or a number of some characters.

If comments=True then the "Automorphism…" line of the had.n.xxx.txt file is printed if it exists. Otherwise nothing is done.

EXAMPLES:

```
sage: hadamard_matrix_www("had.4.txt")      # optional - internet
[ 1  1  1  1]
[ 1 -1  1 -1]
[ 1  1 -1 -1]
[ 1 -1 -1  1]
sage: hadamard_matrix_www("had.16.2.txt",comments=True)   # optional - internet
Automorphism group has order = 49152 = 2^14 * 3
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1]
[ 1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1]
[ 1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1]
[ 1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1  1 -1]
[ 1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1]
[ 1  1 -1 -1  1 -1  1 -1 -1 -1  1  1 -1  1 -1  1]
[ 1  1 -1 -1 -1  1 -1  1 -1 -1  1  1  1 -1  1 -1]
[ 1 -1  1 -1  1 -1 -1  1 -1  1 -1  1 -1  1  1 -1]
[ 1 -1  1 -1 -1  1  1 -1 -1  1 -1  1  1 -1 -1  1]
[ 1 -1 -1  1  1  1 -1 -1 -1  1  1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1 -1  1  1 -1  1  1 -1  1  1 -1 -1]
```

```
>>> from sage.all import *
>>> hadamard_matrix_www("had.4.txt")      # optional - internet
[ 1  1  1  1]
[ 1 -1  1 -1]
[ 1  1 -1 -1]
[ 1 -1 -1  1]
>>> hadamard_matrix_www("had.16.2.txt",comments=True)   # optional - internet
Automorphism group has order = 49152 = 2^14 * 3
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1]
[ 1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1]
[ 1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1]
[ 1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1  1 -1]
[ 1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1]
[ 1  1 -1 -1  1 -1  1 -1 -1 -1  1  1 -1  1 -1  1]
[ 1  1 -1 -1 -1  1 -1  1 -1 -1  1  1  1 -1  1 -1]
[ 1 -1  1 -1  1 -1 -1  1 -1  1 -1  1 -1  1  1 -1]
[ 1 -1  1 -1 -1  1  1 -1 -1  1 -1  1  1 -1 -1  1]
[ 1 -1 -1  1  1  1 -1 -1 -1  1  1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1 -1  1  1 -1  1  1 -1  1  1 -1 -1]
```

sage.combinat.matrices.hadamard_matrix.**is_hadamard_matrix**(*M*, *normalized=False*, *skew=False*, *verbose=False*)

Test if `M` is a Hadamard matrix.

INPUT:

- `M` – a matrix
- `normalized` – boolean (default: `False`); whether to test if `M` is a normalized Hadamard matrix, i.e. has its first row/column filled with +1
- `skew` – boolean (default: `False`); whether to test if `M` is a skew Hadamard matrix, i.e. $M = S + I$ for $-S = S^\top$, and I the identity matrix
- `verbose` – boolean (default: `False`); whether to be verbose when the matrix is not Hadamard

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import is_hadamard_matrix
sage: h = matrix.hadamard(12)
sage: is_hadamard_matrix(h)
True
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
sage: h = skew_hadamard_matrix(12)
sage: is_hadamard_matrix(h, skew=True)
True
sage: h = matrix.hadamard(12)
sage: h[0,0] = 2
sage: is_hadamard_matrix(h,verbose=True)
The matrix does not only contain +1 and -1 entries, e.g. 2
False
sage: h = matrix.hadamard(12)
sage: for i in range(12):
....:     h[i,2] = -h[i,2]
sage: is_hadamard_matrix(h,verbose=True,normalized=True)
The matrix is not normalized
False
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import is_hadamard_matrix
>>> h = matrix.hadamard(Integer(12))
>>> is_hadamard_matrix(h)
True
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
>>> h = skew_hadamard_matrix(Integer(12))
>>> is_hadamard_matrix(h, skew=True)
True
>>> h = matrix.hadamard(Integer(12))
>>> h[Integer(0),Integer(0)] = Integer(2)
>>> is_hadamard_matrix(h,verbose=True)
The matrix does not only contain +1 and -1 entries, e.g. 2
False
>>> h = matrix.hadamard(Integer(12))
>>> for i in range(Integer(12)):
...     h[i,Integer(2)] = -h[i,Integer(2)]
>>> is_hadamard_matrix(h,verbose=True,normalized=True)
The matrix is not normalized
False
```

`sage.combinat.matrices.hadamard_matrix.`**`is_skew_hadamard_matrix`**(*M*, *normalized=False*, *verbose=False*)

Test if M is a skew Hadamard matrix.

this is a wrapper around the function *is_hadamard_matrix()*

INPUT:

- M – a matrix
- normalized – boolean (default: False); whether to test if M is a skew-normalized Hadamard matrix, i.e. has its first row filled with +1
- verbose – boolean (default: False); whether to be verbose when the matrix is not skew Hadamard

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import is_skew_hadamard_matrix,␣
↪skew_hadamard_matrix
sage: h = matrix.hadamard(12)
sage: is_skew_hadamard_matrix(h, verbose=True)
The matrix is not skew
False
sage: h = skew_hadamard_matrix(12)
sage: is_skew_hadamard_matrix(h)
True
sage: from sage.combinat.matrices.hadamard_matrix import normalise_hadamard
sage: h = normalise_hadamard(skew_hadamard_matrix(12), skew=True)
sage: is_skew_hadamard_matrix(h, verbose=True, normalized=True)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import is_skew_hadamard_matrix,␣
↪skew_hadamard_matrix
>>> h = matrix.hadamard(Integer(12))
>>> is_skew_hadamard_matrix(h, verbose=True)
The matrix is not skew
False
>>> h = skew_hadamard_matrix(Integer(12))
>>> is_skew_hadamard_matrix(h)
True
>>> from sage.combinat.matrices.hadamard_matrix import normalise_hadamard
>>> h = normalise_hadamard(skew_hadamard_matrix(Integer(12)), skew=True)
>>> is_skew_hadamard_matrix(h, verbose=True, normalized=True)
True
```

sage.combinat.matrices.hadamard_matrix.**normalise_hadamard**(*H*, *skew=False*)

Return the normalised Hadamard matrix corresponding to H.

The normalised Hadamard matrix corresponding to a Hadamard matrix H is a matrix whose every entry in the first row and column is +1.

If skew is True, the matrix returned will be skew-normal: a skew Hadamard matrix with first row of all $+1$.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import normalise_hadamard, is_
↪hadamard_matrix, skew_hadamard_matrix
sage: H = normalise_hadamard(hadamard_matrix(4))
```

(continues on next page)

(continued from previous page)

```
sage: H == hadamard_matrix(4)
True
sage: H = normalise_hadamard(skew_hadamard_matrix(20, skew_normalize=False),␣
↪skew=True)
sage: is_hadamard_matrix(H, skew=True, normalized=True)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import normalise_hadamard, is_
↪hadamard_matrix, skew_hadamard_matrix
>>> H = normalise_hadamard(hadamard_matrix(Integer(4)))
>>> H == hadamard_matrix(Integer(4))
True
>>> H = normalise_hadamard(skew_hadamard_matrix(Integer(20), skew_
↪normalize=False), skew=True)
>>> is_hadamard_matrix(H, skew=True, normalized=True)
True
```

If `skew` is `True` but the Hadamard matrix is not skew, the matrix returned will not be normalized:

```
sage: H = normalise_hadamard(hadamard_matrix(92), skew=True)
sage: is_hadamard_matrix(H, normalized=True)
False
```

```
>>> from sage.all import *
>>> H = normalise_hadamard(hadamard_matrix(Integer(92)), skew=True)
>>> is_hadamard_matrix(H, normalized=True)
False
```

sage.combinat.matrices.hadamard_matrix.**regular_symmetric_hadamard_matrix_with_constant_diagonal**(*n*, *e*, *existence=*

Return a Regular Symmetric Hadamard Matrix with Constant Diagonal.

A Hadamard matrix is said to be *regular* if its rows all sum to the same value.

For $\epsilon \in \{-1,+1\}$, we say that M is a $(n,\epsilon)-RSHCD$ if M is a regular symmetric Hadamard matrix with constant diagonal $\delta \in \{-1,+1\}$ and row sums all equal to $\delta\epsilon\sqrt(n)$. For more information, see [HX2010] or 10.5.1 in [BH2012]. For the case $n = 324$, see `RSHCD_324()` and [CP2016].

INPUT:

- `n` – integer; side of the matrix
- `e` – -1 or $+1$; the value of ϵ

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import regular_symmetric_
↪hadamard_matrix_with_constant_diagonal
sage: regular_symmetric_hadamard_matrix_with_constant_diagonal(4,1)
[ 1  1  1 -1]
```

(continues on next page)

(continued from previous page)

```
[ 1  1 -1  1]
[ 1 -1  1  1]
[-1  1  1  1]
sage: regular_symmetric_hadamard_matrix_with_constant_diagonal(4,-1)
[ 1 -1 -1 -1]
[-1  1 -1 -1]
[-1 -1  1 -1]
[-1 -1 -1  1]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import regular_symmetric_hadamard_
↪matrix_with_constant_diagonal
>>> regular_symmetric_hadamard_matrix_with_constant_diagonal(Integer(4),
↪Integer(1))
[ 1  1  1 -1]
[ 1  1 -1  1]
[ 1 -1  1  1]
[-1  1  1  1]
>>> regular_symmetric_hadamard_matrix_with_constant_diagonal(Integer(4),-
↪Integer(1))
[ 1 -1 -1 -1]
[-1  1 -1 -1]
[-1 -1  1 -1]
[-1 -1 -1  1]
```

Other hardcoded values:

```
sage: for n,e in [(36,1),(36,-1),(100,1),(100,-1),(196, 1)]:  # long time
....:     print(repr(regular_symmetric_hadamard_matrix_with_constant_diagonal(n,
↪e)))
36 x 36 dense matrix over Integer Ring
36 x 36 dense matrix over Integer Ring
100 x 100 dense matrix over Integer Ring
100 x 100 dense matrix over Integer Ring
196 x 196 dense matrix over Integer Ring

sage: for n,e in [(324,1),(324,-1)]: # not tested - long time, tested in RSHCD_324
....:     print(repr(regular_symmetric_hadamard_matrix_with_constant_diagonal(n,
↪e)))
324 x 324 dense matrix over Integer Ring
324 x 324 dense matrix over Integer Ring
```

```
>>> from sage.all import *
>>> for n,e in [(Integer(36),Integer(1)),(Integer(36),-Integer(1)),(Integer(100),
↪Integer(1)),(Integer(100),-Integer(1)),(Integer(196), Integer(1))]:  # long time
...     print(repr(regular_symmetric_hadamard_matrix_with_constant_diagonal(n,e)))
36 x 36 dense matrix over Integer Ring
36 x 36 dense matrix over Integer Ring
100 x 100 dense matrix over Integer Ring
100 x 100 dense matrix over Integer Ring
196 x 196 dense matrix over Integer Ring
```

(continues on next page)

(continued from previous page)

```
>>> for n,e in [(Integer(324),Integer(1)),(Integer(324),-Integer(1))]: # not␣
↪tested - long time, tested in RSHCD_324
...     print(repr(regular_symmetric_hadamard_matrix_with_constant_diagonal(n,e)))
324 x 324 dense matrix over Integer Ring
324 x 324 dense matrix over Integer Ring
```

From two close prime powers:

```
sage: regular_symmetric_hadamard_matrix_with_constant_diagonal(64,-1)
64 x 64 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
```

```
>>> from sage.all import *
>>> regular_symmetric_hadamard_matrix_with_constant_diagonal(Integer(64),-
↪Integer(1))
64 x 64 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
```

From a prime power and a conference matrix:

```
sage: regular_symmetric_hadamard_matrix_with_constant_diagonal(676,1)  # long time
676 x 676 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
```

```
>>> from sage.all import *
>>> regular_symmetric_hadamard_matrix_with_constant_diagonal(Integer(676),
↪Integer(1))  # long time
676 x 676 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
```

Recursive construction:

```
sage: regular_symmetric_hadamard_matrix_with_constant_diagonal(144,-1)
144 x 144 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
```

```
>>> from sage.all import *
>>> regular_symmetric_hadamard_matrix_with_constant_diagonal(Integer(144),-
↪Integer(1))
144 x 144 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
```

REFERENCE:

- [BH2012]
- [HX2010]

sage.combinat.matrices.hadamard_matrix.**rshcd_from_close_prime_powers**(*n*)

Return a $(n^2, 1)$-RSHCD when $n-1$ and $n+1$ are odd prime powers and $n = 0 \pmod 4$.

The construction implemented here appears in Theorem 4.3 from [GS1970].

Note that the authors of [SWW1972] claim in Corollary 5.12 (page 342) to have proved the same result without the $n = 0 \pmod 4$ restriction with a *very* similar construction. So far, however, I (Nathann Cohen) have not been able to make it work.

INPUT:

- `n` – integer congruent to $0 \pmod 4$

See also:

`regular_symmetric_hadamard_matrix_with_constant_diagonal()`

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import rshcd_from_close_prime_
↪powers
sage: rshcd_from_close_prime_powers(4)
[-1 -1  1 -1  1 -1 -1  1 -1  1 -1 -1  1 -1  1 -1]
[-1 -1 -1  1  1 -1 -1  1 -1 -1  1 -1 -1  1 -1  1]
[ 1 -1 -1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1  1 -1]
[-1  1  1 -1  1  1 -1 -1 -1 -1 -1  1 -1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1  1 -1  1 -1 -1 -1]
[-1 -1  1  1 -1 -1  1 -1 -1  1 -1  1 -1  1 -1 -1]
[-1 -1  1 -1 -1  1 -1 -1  1 -1  1 -1 -1  1  1 -1]
[ 1  1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1  1  1  1]
[-1 -1 -1 -1 -1 -1  1 -1 -1 -1  1  1  1 -1  1  1]
[ 1 -1 -1 -1 -1  1 -1  1 -1 -1 -1  1  1  1 -1 -1]
[-1  1 -1 -1  1 -1  1 -1  1 -1 -1 -1  1  1 -1 -1]
[-1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1 -1 -1  1]
[ 1 -1 -1 -1  1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[-1  1 -1 -1 -1  1  1  1 -1  1  1 -1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1 -1  1  1  1 -1 -1 -1 -1 -1 -1  1]
[-1  1 -1  1 -1 -1 -1  1  1 -1 -1  1 -1 -1  1 -1]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import rshcd_from_close_prime_
↪powers
>>> rshcd_from_close_prime_powers(Integer(4))
[-1 -1  1 -1  1 -1 -1  1 -1  1 -1 -1  1 -1  1 -1]
[-1 -1 -1  1  1 -1 -1  1 -1 -1  1 -1 -1  1 -1  1]
[ 1 -1 -1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1  1 -1]
[-1  1  1 -1  1  1 -1 -1 -1 -1 -1  1 -1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1  1 -1  1 -1 -1 -1]
[-1 -1  1  1 -1 -1  1 -1 -1  1 -1  1 -1  1 -1 -1]
[-1 -1  1 -1 -1  1 -1 -1  1 -1  1 -1 -1  1  1 -1]
[ 1  1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1  1  1  1]
[-1 -1 -1 -1 -1 -1  1 -1 -1 -1  1  1  1 -1  1  1]
[ 1 -1 -1 -1 -1  1 -1  1 -1 -1 -1  1  1  1 -1 -1]
[-1  1 -1 -1  1 -1  1 -1  1 -1 -1 -1  1  1 -1 -1]
[-1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1 -1 -1  1]
[ 1 -1 -1 -1  1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[-1  1 -1 -1 -1  1  1  1 -1  1  1 -1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1 -1  1  1  1 -1 -1 -1 -1 -1 -1  1]
[-1  1 -1  1 -1 -1 -1  1  1 -1 -1  1 -1 -1  1 -1]
```

REFERENCE:

- [SWW1972]

sage.combinat.matrices.hadamard_matrix.**rshcd_from_prime_power_and_conference_matrix**(*n*)

Return a $((n-1)^2, 1)$-RSHCD if n is prime power, and symmetric $(n-1)$-conference matrix exists.

The construction implemented here is Theorem 16 (and Corollary 17) from [WW1972].

In [SWW1972] this construction (Theorem 5.15 and Corollary 5.16) is reproduced with a typo. Note that [WW1972] refers to [Sz1969] for the construction, provided by *szekeres_difference_set_pair()*, of complementary difference sets, and the latter has a typo.

From a *symmetric_conference_matrix()*, we only need the Seidel adjacency matrix of the underlying strongly regular conference (i.e. Paley type) graph, which we construct directly.

INPUT:

- n – integer

See also

regular_symmetric_hadamard_matrix_with_constant_diagonal()

EXAMPLES:

A 36x36 example

```
sage: from sage.combinat.matrices.hadamard_matrix import rshcd_from_prime_power_
↪and_conference_matrix
sage: from sage.combinat.matrices.hadamard_matrix import is_hadamard_matrix
sage: H = rshcd_from_prime_power_and_conference_matrix(7); H
36 x 36 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
sage: H == H.T and is_hadamard_matrix(H) and H.diagonal() == [1]*36 and␣
↪list(sum(H)) == [6]*36
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import rshcd_from_prime_power_and_
↪conference_matrix
>>> from sage.combinat.matrices.hadamard_matrix import is_hadamard_matrix
>>> H = rshcd_from_prime_power_and_conference_matrix(Integer(7)); H
36 x 36 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
>>> H == H.T and is_hadamard_matrix(H) and H.diagonal() ==␣
↪[Integer(1)]*Integer(36) and list(sum(H)) == [Integer(6)]*Integer(36)
True
```

Bigger examples, only provided by this construction

```
sage: H = rshcd_from_prime_power_and_conference_matrix(27)  # long time
sage: H == H.T and is_hadamard_matrix(H)                    # long time
True
sage: H.diagonal() == [1]*676 and list(sum(H)) == [26]*676  # long time
True
```

```
>>> from sage.all import *
>>> H = rshcd_from_prime_power_and_conference_matrix(Integer(27))   # long time
>>> H == H.T and is_hadamard_matrix(H)                     # long time
True
>>> H.diagonal() == [Integer(1)]*Integer(676) and list(sum(H)) ==␣
↪[Integer(26)]*Integer(676)   # long time
True
```

In this example the conference matrix is not Paley, as 45 is not a prime power

```
sage: H = rshcd_from_prime_power_and_conference_matrix(47)  # not tested (long␣
↪time)
```

```
>>> from sage.all import *
>>> H = rshcd_from_prime_power_and_conference_matrix(Integer(47))   # not tested␣
↪(long time)
```

REFERENCE:

- [WW1972]

sage.combinat.matrices.hadamard_matrix.**skew_hadamard_matrix**(*n*, *existence=False*, *skew_normalize=True*, *check=True*, *construction_name=False*)

Try to construct a skew Hadamard matrix.

A Hadamard matrix H is called skew if $H = S - I$, for I the identity matrix and $-S = S^\top$. Currently all orders ≤ 1200 for which a construction is known are implemented. For $n > 1200$, only some orders are available.

INPUT:

- `n` – integer; dimension of the matrix
- `existence` – boolean (default: `False`); whether to build the matrix or merely query if a construction is available in Sage. When set to `True`, the function returns:
  - `True` – meaning that Sage knows how to build the matrix
  - `Unknown` – meaning that Sage does not know how to build the matrix, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the matrix does not exist
- `skew_normalize` – boolean (default: `True`); whether to make the 1st row all-one, and adjust the 1st column accordingly
- `check` – boolean (default: `True`); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- `construction_name` – boolean (default: `False`); if it is `True`, `existence` is `True`, and a matrix exists, output the construction name. It has no effect if `existence` is set to `False`.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
sage: skew_hadamard_matrix(12).det()
2985984
sage: 12^6
2985984
```

(continues on next page)

(continued from previous page)

```
sage: skew_hadamard_matrix(1)
[1]
sage: skew_hadamard_matrix(2)
[ 1  1]
[-1  1]
sage: skew_hadamard_matrix(196, existence=True, construction_name=True)
'Williamson-Goethals-Seidel: 196'
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
>>> skew_hadamard_matrix(Integer(12)).det()
2985984
>>> Integer(12)**Integer(6)
2985984
>>> skew_hadamard_matrix(Integer(1))
[1]
>>> skew_hadamard_matrix(Integer(2))
[ 1  1]
[-1  1]
>>> skew_hadamard_matrix(Integer(196), existence=True, construction_name=True)
'Williamson-Goethals-Seidel: 196'
```

REFERENCES:

- [Ha83]

sage.combinat.matrices.hadamard_matrix.**skew_hadamard_matrix_from_complementary_difference_sets**(*n*, *existence=False*, *check=True*)

Construct a skew Hadamard matrix of order $n = 4(m + 1)$ from complementary difference sets.

If A, B are complementary difference sets over a group of order $2m + 1$, then they can be used to construct a skew Hadamard matrix, as described in [BS1969].

The complementary difference sets are constructed using the function *sage.combinat.designs.difference_family.complementary_difference_sets()*.

INPUT:

- `n` – positive integer; the order of the matrix to be constructed
- `existence` – boolean (default: `False`); if `True`, only return whether the skew Hadamard matrix can be constructed
- `check` – boolean (default: `True`); if `True`, check that the result is a skew Hadamard matrix before returning it

OUTPUT:

If `existence=False`, returns the skew Hadamard matrix of order n. It raises an error if n does not satisfy the required conditions. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪from_complementary_difference_sets
sage: skew_hadamard_matrix_from_complementary_difference_sets(20)
20 x 20 dense matrix over Integer Ring...
sage: skew_hadamard_matrix_from_complementary_difference_sets(52, existence=True)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_from_
↪complementary_difference_sets
>>> skew_hadamard_matrix_from_complementary_difference_sets(Integer(20))
20 x 20 dense matrix over Integer Ring...
>>> skew_hadamard_matrix_from_complementary_difference_sets(Integer(52),␣
↪existence=True)
True
```

sage.combinat.matrices.hadamard_matrix.**skew_hadamard_matrix_from_good_matrices**(*a*, *b*, *c*, *d*, *check=True*)

Construct skew Hadamard matrix from good matrices.

Given good matrices A, B, C, D (A circulant, B, C, D back-circulant) they can be used to construct a skew Hadamard matrix using the following block matrix (as described in [Sze1988]):

$$\begin{pmatrix} A & B & C & D \\ -B & A & D & -C \\ -C & -D & A & B \\ -D & C & -B & A \end{pmatrix}$$

INPUT:

- a – (1,-1) list; the 1st row of A
- b – (1,-1) list; the 1st row of B
- d – (1,-1) list; the 1st row of C
- c – (1,-1) list; the 1st row of D
- check – boolean (default: True); if True, check that the matrix is a skew Hadamard matrix before returning it

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪from_good_matrices
sage: a, b, c, d = ([1, 1, 1, -1, -1], [1, -1, 1, 1, -1], [1, -1, -1, -1, -1], [1,
↪ -1, -1, -1, -1])
sage: skew_hadamard_matrix_from_good_matrices(a, b, c, d)
20 x 20 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_from_
↪good_matrices
>>> a, b, c, d = ([Integer(1), Integer(1), Integer(1), -Integer(1), -Integer(1)],␣
↪[Integer(1), -Integer(1), Integer(1), Integer(1), -Integer(1)], [Integer(1), -
↪Integer(1), -Integer(1), -Integer(1), -Integer(1)], [Integer(1), -Integer(1), -
```

(continues on next page)

(continued from previous page)

```
↪Integer(1), -Integer(1), -Integer(1)])
>>> skew_hadamard_matrix_from_good_matrices(a, b, c, d)
20 x 20 dense matrix over Integer Ring...
```

sage.combinat.matrices.hadamard_matrix.**skew_hadamard_matrix_from_good_matrices_smallcases**(*n*, *existence=False*, *check=True*)

Construct skew Hadamard matrices from good matrices for some small values of $n = 4m$, with m odd.

The function stores good matrices of odd orders ≤ 31, taken from [Sze1988]. These are used to create skew Hadamard matrices of order $4m$, $1 \leq m \leq 31$ (m odd), using the function *skew_hadamard_matrix_from_good_matrices()*.

ALGORITHM:

Given four sequences (stored in `E_sequences`) of length m, they can be used to construct four $E-sequences$ of length $n = 2m + 1$, as follows:

$$\begin{aligned}
a &= 1, a_0, a_1, ..., a_m, -a_m, -a_{m-1}, ..., -a_0 \\
b &= 1, b_0, b_1, ..., b_m, b_m, b_{m-1}, ..., b_0 \\
c &= 1, c_0, c_1, ..., c_m, c_m, c_{m-1}, ..., c_0 \\
d &= 1, d_0, d_1, ..., d_m, d_m, d_{m-1}, ..., d_0
\end{aligned}$$

These E-sequences will be the first rows of the four good matrices needed to construct a skew Hadamard matrix of order $4n$.

INPUT:

- `n` – integer; the order of the skew Hadamard matrix to be constructed
- `existence` – boolean (default: `False`); if `True`, only return whether the Hadamard matrix can be constructed
- `check` – boolean (default: `True`); if `True`, check that the matrix is a Hadamard matrix before returning it

OUTPUT:

If `existence=False`, returns the skew Hadamard matrix of order n. It raises an error if no data is available to construct the matrix of the given order. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪from_good_matrices_smallcases
sage: skew_hadamard_matrix_from_good_matrices_smallcases(20)
20 x 20 dense matrix over Integer Ring...
sage: skew_hadamard_matrix_from_good_matrices_smallcases(20, existence=True)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_from_
↪good_matrices_smallcases
>>> skew_hadamard_matrix_from_good_matrices_smallcases(Integer(20))
```

(continues on next page)

(continued from previous page)

```
20 x 20 dense matrix over Integer Ring...
>>> skew_hadamard_matrix_from_good_matrices_smallcases(Integer(20),␣
↪existence=True)
True
```

sage.combinat.matrices.hadamard_matrix.**skew_hadamard_matrix_from_orthogonal_design**(*n*, *existence=False*, *check=True*)

Construct skew Hadamard matrices of order $mn(n-1)$ if suitable orthogonal designs exist.

In [Seb1978] is proved that if amicable Hadamard matrices of order n and an orthogonal design of type $(1, m, mn-m-1)$ in order mn exist, then a skew Hadamard matrix of order $mn(n-1)$ can be constructed. The paper uses amicable orthogonal designs instead of amicable Hadamard matrices, but the two are equivalent (see [Seb2017]).

Amicable Hadamard matrices are constructed using `amicable_hadamard_matrices()`, and the orthogonal designs are constructed using the Goethals-Seidel array, with data taken from [Seb2017].

INPUT:

- `n` – positive integer; the order of the matrix to be constructed
- `existence` – boolean (default: `False`); if `True`, only return whether the skew Hadamard matrix can be constructed
- `check` – boolean (default: `True`); if `True`, check that the result is a skew Hadamard matrix before returning it

OUTPUT:

If `existence=False`, returns the skew Hadamard matrix of order n. It raises an error if a construction for order n is not yet implemented, or if n does not satisfy the constraint. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪from_orthogonal_design
sage: skew_hadamard_matrix_from_orthogonal_design(756)
756 x 756 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_from_
↪orthogonal_design
>>> skew_hadamard_matrix_from_orthogonal_design(Integer(756))
756 x 756 dense matrix over Integer Ring...
```

If `existence` is True, the function returns a boolean:

```
sage: skew_hadamard_matrix_from_orthogonal_design(200, existence=True)
False
```

```
>>> from sage.all import *
>>> skew_hadamard_matrix_from_orthogonal_design(Integer(200), existence=True)
False
```

sage.combinat.matrices.hadamard_matrix.**skew_hadamard_matrix_spence_1975**(*n*, *existence=False*, *check=True*)

Construct a skew Hadamard matrix of order $n = 4(1 + q + q^2)$ using the Spence construction.

If $n = 4(1 + q + q^2)$ where q is a prime power such that either $1 + q + q^2$ is a prime congruent to $3, 5, 7$ mod 8 or $3 + 2q + 2q^2$ is a prime power, then a skew Hadamard matrix of order n can be constructed using the Goethals Seidel array. The four matrices A, B, C, D plugged into the GS-array are created using complementary difference sets of order $1 + q + q^2$ (which exist if q satisfies the conditions above), and a cyclic planar difference set with parameters $(1 + q^2 + q^4, 1 + q^2, 1)$. These are constructed by the functions `sage.combinat.designs.difference_family.complementary_difference_sets()` and `sage.combinat.designs.difference_family.difference_family()`.

For more details, see [Spe1975b].

INPUT:

- `n` – positive integer; the order of the matrix to be constructed
- `existence` – boolean (default: `False`); if `True`, only return whether the Hadamard matrix can be constructed
- `check` – boolean (default: `True`); check that the result is a skew Hadamard matrix before returning it

OUTPUT:

If `existence=False`, returns the skew Hadamard matrix of order n. It raises an error if n does not satisfy the required conditions. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪spence_1975
sage: skew_hadamard_matrix_spence_1975(52)
52 x 52 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪spence_1975
>>> skew_hadamard_matrix_spence_1975(Integer(52))
52 x 52 dense matrix over Integer Ring...
```

If `existence` is True, the function returns a boolean:

```
sage: skew_hadamard_matrix_spence_1975(52, existence=True)
True
```

```
>>> from sage.all import *
>>> skew_hadamard_matrix_spence_1975(Integer(52), existence=True)
True
```

sage.combinat.matrices.hadamard_matrix.**skew_hadamard_matrix_spence_construction**(*n*, *check=True*)

Construct skew Hadamard matrix of order n using Spence construction.

This function will construct skew Hadamard matrix of order $n = 2(q + 1)$ where q is a prime power with $q = 5$ (mod 8). The construction is taken from [Spe1977], and the relative difference sets are constructed using `sage.combinat.designs.difference_family.relative_difference_set_from_homomorphism()`.

INPUT:

- `n` – positive integer
- `check` – boolean (default: `True`); if `True`, check that the resulting matrix is Hadamard before returning it

OUTPUT:

If `n` satisfies the requirements described above, the function returns a $n \times n$ Hadamard matrix. Otherwise, an exception is raised.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪spence_construction
sage: skew_hadamard_matrix_spence_construction(28)
28 x 28 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪spence_construction
>>> skew_hadamard_matrix_spence_construction(Integer(28))
28 x 28 dense matrix over Integer Ring...
```

`sage.combinat.matrices.hadamard_matrix.`**`skew_hadamard_matrix_whiteman_construction`**(*n*, *existence=False*, *check=True*)

Construct a skew Hadamard matrix of order $n = 2(q + 1)$ where $q = p^t$ is a prime power with $p \cong 5 \mod 8$ and $t \cong 2 \mod 4$.

Assuming n satisfies the conditions above, it is possible to construct two supplementary difference sets A, B (see [Whi1971]), and these can be used to construct a skew Hadamard matrix, as described in [BS1969].

INPUT:

- `n` – positive integer; the order of the matrix to be constructed
- `existence` – boolean (default: `False`); if `True`, only return whether the Hadamard matrix can be constructed
- `check` – boolean (default: `True`); if `True`, check that the result is a skew Hadamard matrix before returning it

OUTPUT:

If `existence=False`, returns the skew Hadamard matrix of order n. It raises an error if n does not satisfy the required conditions. If `existence=True`, returns a boolean representing whether the matrix can be constructed or not.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪whiteman_construction
sage: skew_hadamard_matrix_whiteman_construction(52)
52 x 52 dense matrix over Integer Ring...
sage: skew_hadamard_matrix_whiteman_construction(52, existence=True)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix_
↪whiteman_construction
>>> skew_hadamard_matrix_whiteman_construction(Integer(52))
52 x 52 dense matrix over Integer Ring...
>>> skew_hadamard_matrix_whiteman_construction(Integer(52), existence=True)
True
```

Note

A more general version of this construction is *skew_hadamard_matrix_from_complementary_difference_sets()*.

sage.combinat.matrices.hadamard_matrix.**symmetric_conference_matrix**(*n*, *check=True*, *existence=False*)

Try to construct a symmetric conference matrix.

A conference matrix is an $n \times n$ matrix C with 0s on the main diagonal and 1s and -1s elsewhere, satisfying $CC^\top = (n-1)I$. If $C = C^\top$ then $n \cong 2 \mod 4$ and C is Seidel adjacency matrix of a graph, whose descendent graphs are strongly regular graphs with parameters $(n-1, (n-2)/2, (n-6)/4, (n-2)/4)$, see Sec.10.4 of [BH2012]. Thus we build C from the Seidel adjacency matrix of the latter by adding row and column of 1s.

INPUT:

- n – integer; dimension of the matrix
- check – boolean (default: True); whether to check that output is correct before returning it. As this is expected to be useless, you may want to disable it whenever you want speed.
- existence – boolean (default: False); if true, only check that such a matrix exists.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import symmetric_conference_
↪matrix
sage: C = symmetric_conference_matrix(10); C
[ 0  1  1  1  1  1  1  1  1  1]
[ 1  0 -1 -1  1 -1  1  1  1 -1]
[ 1 -1  0 -1  1  1 -1 -1  1  1]
[ 1 -1 -1  0 -1  1  1  1 -1  1]
[ 1  1  1 -1  0 -1 -1  1 -1  1]
[ 1 -1  1  1 -1  0 -1  1  1 -1]
[ 1  1 -1  1 -1 -1  0 -1  1  1]
[ 1  1 -1  1  1  1 -1  0 -1 -1]
[ 1  1  1 -1 -1  1  1 -1  0 -1]
[ 1 -1  1  1  1 -1  1 -1 -1  0]
sage: C^2 == 9*identity_matrix(10) and C == C.T
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import symmetric_conference_matrix
>>> C = symmetric_conference_matrix(Integer(10)); C
[ 0  1  1  1  1  1  1  1  1  1]
[ 1  0 -1 -1  1 -1  1  1  1 -1]
```

(continues on next page)

(continued from previous page)

```
[ 1 -1  0 -1  1  1 -1 -1  1  1]
[ 1 -1 -1  0 -1  1  1  1 -1  1]
[ 1  1  1 -1  0 -1 -1  1 -1  1]
[ 1 -1  1  1 -1  0 -1  1  1 -1]
[ 1  1 -1  1 -1 -1  0 -1  1  1]
[ 1  1 -1  1  1  1 -1  0 -1 -1]
[ 1  1  1 -1 -1  1  1 -1  0 -1]
[ 1 -1  1  1  1 -1  1 -1 -1  0]
>>> C**Integer(2) == Integer(9)*identity_matrix(Integer(10)) and C == C.T
True
```

sage.combinat.matrices.hadamard_matrix.**symmetric_conference_matrix_paley**(*n*)

Construct a symmetric conference matrix of order n.

A conference matrix is an $n \times n$ matrix C with 0s on the main diagonal and 1s and -1s elsewhere, satisfying $CC^\top = (n-1)I$. This construction assumes that $q = n - 1$ is a prime power, with $q \cong 1 \mod 4$. See [Hora] or [Lon2013].

These matrices are used in *hadamard_matrix_paleyII()*.

INPUT:

- n – integer; the order of the symmetric conference matrix to construct

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import symmetric_conference_
↪matrix_paley
sage: symmetric_conference_matrix_paley(6)
[ 0  1  1  1  1  1]
[ 1  0  1 -1 -1  1]
[ 1  1  0  1 -1 -1]
[ 1 -1  1  0  1 -1]
[ 1 -1 -1  1  0  1]
[ 1  1 -1 -1  1  0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import symmetric_conference_
↪matrix_paley
>>> symmetric_conference_matrix_paley(Integer(6))
[ 0  1  1  1  1  1]
[ 1  0  1 -1 -1  1]
[ 1  1  0  1 -1 -1]
[ 1 -1  1  0  1 -1]
[ 1 -1 -1  1  0  1]
[ 1  1 -1 -1  1  0]
```

sage.combinat.matrices.hadamard_matrix.**szekeres_difference_set_pair**(*m*, *check=True*)

Construct Szekeres $(2m + 1, m, 1)$-cyclic difference family.

Let $4m+3$ be a prime power. Theorem 3 in [Sz1969] contains a construction of a pair of *complementary difference sets* A, B in the subgroup G of the quadratic residues in F^*_{4m+3}. Namely $|A| = |B| = m$, $a \in A$ whenever $a - 1 \in G$, $b \in B$ whenever $b + 1 \in G$. See also Theorem 2.6 in [SWW1972] (there the formula for B is correct, as opposed to (4.2) in [Sz1969], where the sign before 1 is wrong.

In modern terminology, for $m > 1$ the sets A and B form a `difference family` with parameters $(2m+1, m, 1)$. I.e. each non-identity $g \in G$ can be expressed uniquely as xy^{-1} for $x, y \in A$ or $x, y \in B$. Other, specific to this construction, properties of A and B are: for a in A one has a^{-1} not in A, whereas for b in B one has b^{-1} in B.

INPUT:

- `m` – integer; dimension of the matrix
- `check` – boolean (default: `True`); whether to check A and B for correctness

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import szekeres_difference_set_
↪pair
sage: G,A,B=szekeres_difference_set_pair(6)
sage: G,A,B=szekeres_difference_set_pair(7)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import szekeres_difference_set_
↪pair
>>> G,A,B=szekeres_difference_set_pair(Integer(6))
>>> G,A,B=szekeres_difference_set_pair(Integer(7))
```

REFERENCE:

- [Sz1969]

`sage.combinat.matrices.hadamard_matrix.`**`turyn_type_hadamard_matrix_smallcases`**`(`*n*, *existence=False*, *check=True*`)`

Construct a Hadamard matrix of order n from available 4-symbol δ codes and Williamson quadruples.

The function looks for Baumert-Hall units and Williamson type matrices from `four_symbol_delta_code_smallcases()` and `williamson_type_quadruples_smallcases()` and use them to construct a Hadamard matrix with the Turyn construction defined in `hadamard_matrix_turyn_type()`.

INPUT:

- `n` – integer; the order of the matrix to be constructed
- `existence` – boolean (default: `False`); if `True`, only check if the matrix exists
- `check` – boolean (default: `True`); if `True`, check that the matrix is a Hadamard matrix before returning

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import turyn_type_hadamard_
↪matrix_smallcases
sage: turyn_type_hadamard_matrix_smallcases(28, existence=True)
True
sage: turyn_type_hadamard_matrix_smallcases(28)
28 x 28 dense matrix over Integer Ring...
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import turyn_type_hadamard_matrix_
↪smallcases
>>> turyn_type_hadamard_matrix_smallcases(Integer(28), existence=True)
True
```

(continues on next page)

(continued from previous page)

```
>>> turyn_type_hadamard_matrix_smallcases(Integer(28))
28 x 28 dense matrix over Integer Ring...
```

sage.combinat.matrices.hadamard_matrix.**typeI_matrix_difference_set**(G, A)

(1,-1)-incidence type I matrix of a difference set A in G.

Let A be a difference set in a group G of order n. Return $n \times n$ matrix M with $M_{ij} = 1$ if $A_i A_j^{-1} \in A$, and $M_{ij} = -1$ otherwise.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import szekeres_difference_set_
↪pair
sage: from sage.combinat.matrices.hadamard_matrix import typeI_matrix_difference_
↪set
sage: G,A,B=szekeres_difference_set_pair(2)
sage: typeI_matrix_difference_set(G,A)
[-1  1 -1 -1  1]
[-1 -1 -1  1  1]
[ 1  1 -1 -1 -1]
[ 1 -1  1 -1 -1]
[-1 -1  1  1 -1]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import szekeres_difference_set_
↪pair
>>> from sage.combinat.matrices.hadamard_matrix import typeI_matrix_difference_set
>>> G,A,B=szekeres_difference_set_pair(Integer(2))
>>> typeI_matrix_difference_set(G,A)
[-1  1 -1 -1  1]
[-1 -1 -1  1  1]
[ 1  1 -1 -1 -1]
[ 1 -1  1 -1 -1]
[-1 -1  1  1 -1]
```

sage.combinat.matrices.hadamard_matrix.**williamson_goethals_seidel_skew_hadamard_matrix**(*a*, *b*, *c*, *d*, *check=True*)

Williamson-Goethals-Seidel construction of a skew Hadamard matrix.

Given $n \times n$ (anti)circulant matrices A, B, C, D with 1,-1 entries, and satisfying $A + A^\top = 2I$, $AA^\top + BB^\top + CC^\top + DD^\top = 4nI$, one can construct a skew Hadamard matrix of order $4n$, cf. [GS70s].

INPUT:

- `a` – 1,-1 list; the 1st row of A
- `b` – 1,-1 list; the 1st row of B
- `d` – 1,-1 list; the 1st row of C
- `c` – 1,-1 list; the 1st row of D
- `check` – boolean (default: `True`); if `True`, check that the resulting matrix is skew Hadamard before returning it

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import williamson_goethals_
↪seidel_skew_hadamard_matrix as WGS
sage: a = [ 1,  1, 1, -1,  1, -1,  1, -1, -1]
sage: b = [ 1, -1, 1,  1, -1, -1,  1,  1, -1]
sage: c = [-1, -1]+[1]*6+[-1]
sage: d = [ 1,  1, 1, -1,  1,  1, -1,  1,  1]
sage: M = WGS(a,b,c,d,check=True)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import williamson_goethals_seidel_
↪skew_hadamard_matrix as WGS
>>> a = [ Integer(1),  Integer(1), Integer(1), -Integer(1),  Integer(1), -
↪Integer(1),  Integer(1), -Integer(1), -Integer(1)]
>>> b = [ Integer(1), -Integer(1), Integer(1),  Integer(1), -Integer(1), -
↪Integer(1),  Integer(1),  Integer(1), -Integer(1)]
>>> c = [-Integer(1), -Integer(1)]+[Integer(1)]*Integer(6)+[-Integer(1)]
>>> d = [ Integer(1),  Integer(1), Integer(1), -Integer(1),  Integer(1), 
↪Integer(1), -Integer(1),  Integer(1),  Integer(1)]
>>> M = WGS(a,b,c,d,check=True)
```

REFERENCES:

- [GS70s]
- [Wall71]
- [KoSt08]

sage.combinat.matrices.hadamard_matrix.**williamson_hadamard_matrix_smallcases**(*n*, *existence=False*, *check=True*)

Construct Williamson type Hadamard matrices for some small values of n.

This function uses the data contained in `sage.combinat.matrices.hadamard_matrix.williamson_type_quadruples_smallcases()` to create Hadamard matrices of the Williamson type, using the construction from `sage.combinat.matrices.hadamard_matrix.hadamard_matrix_williamson_type()`.

INPUT:

- `n` – integer; the order of the matrix
- `existence` – boolean (dafault: `False`); if `True`, only check that we can do the construction
- `check` – boolean (default: `True`); if `True` check the result

sage.combinat.matrices.hadamard_matrix.**williamson_type_quadruples_smallcases**(*n*, *existence=False*)

Quadruples of matrices that can be used to construct Williamson type Hadamard matrices.

This function contains for some values of n, four $n \times n$ matrices used in the Williamson construction of Hadamard matrices. Namely, the function returns the first row of 4 $n \times n$ circulant matrices with the properties described in `sage.combinat.matrices.hadamard_matrix.hadamard_matrix_williamson_type()`. The matrices for $n = 3, 5, ..., 29, 37, 43$ are given in [Ha83]. The matrices for $n = 31, 33, 39, 41, 45, 49, 51, 55, 57, 61, 63$ are given in [Lon2013].

INPUT:

- n – integer; the order of the matrices to be returned
- existence – boolean (dafault: False); if True, only check that we have the quadruple

OUTPUT:

If existence is false, returns a tuple containing four vectors, each being the first line of one of the four matrices. It raises an error if no such matrices are available. If existence is true, returns a boolean representing whether the matrices are available or not.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import williamson_type_
↪quadruples_smallcases
sage: williamson_type_quadruples_smallcases(29)
((1, 1, 1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, -1, -1, -1, -1, 1, -1, 1, -1, -1,␣
↪1, 1, -1, -1, -1, 1, 1),
 (1, -1, 1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, 1, -1, -1, 1,␣
↪1, -1, -1, -1, 1, -1),
 (1, 1, 1, 1, -1, 1, 1, -1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1, 1,
↪ 1, -1, 1, 1, 1),
 (1, 1, -1, -1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1, -1,
↪ -1, 1, -1, -1, 1))
sage: williamson_type_quadruples_smallcases(43, existence=True)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.hadamard_matrix import williamson_type_quadruples_
↪smallcases
>>> williamson_type_quadruples_smallcases(Integer(29))
((1, 1, 1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, -1, -1, -1, -1, 1, -1, 1, -1, -1,␣
↪1, 1, -1, -1, -1, 1, 1),
 (1, -1, 1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, 1, -1, -1, 1,␣
↪1, -1, -1, -1, 1, -1),
 (1, 1, 1, 1, -1, 1, 1, -1, 1, -1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1, 1,
↪ 1, -1, 1, 1, 1),
 (1, 1, -1, -1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1, -1,
↪ -1, 1, -1, -1, 1))
>>> williamson_type_quadruples_smallcases(Integer(43), existence=True)
True
```

5.1.138 Latin Squares

A *latin square* of order n is an $n \times n$ array such that each symbol $s \in \{0, 1, \dots, n-1\}$ appears precisely once in each row, and precisely once in each column. A *partial latin square* of order n is an $n \times n$ array such that each symbol $s \in \{0, 1, \dots, n-1\}$ appears at most once in each row, and at most once in each column. Empty cells are denoted by -1. A latin square L is a *completion* of a partial latin square P if $P \subseteq L$. If P completes to just L then P *has unique completion*.

A *latin bitrade* (T_1, T_2) is a pair of partial latin squares such that:

1. $\{(i, j) \mid (i, j, k) \in T_1 \text{ for some symbol } k\} = \{(i, j) \mid (i, j, k') \in T_2 \text{ for some symbol } k'\}$;
2. for each $(i, j, k) \in T_1$ and $(i, j, k') \in T_2$, $k \neq k'$;
3. the symbols appearing in row i of T_1 are the same as those of row i of T_2; the symbols appearing in column j of T_1 are the same as those of column j of T_2.

Intuitively speaking, a bitrade gives the difference between two latin squares, so if (T_1, T_2) is a bitrade for the pair of latin squares (L_1, L_2), then $L1 = (L2 \setminus T_1) \cup T_2$ and $L2 = (L1 \setminus T_2) \cup T_1$.

This file contains

1. LatinSquare class definition;
2. some named latin squares (back circulant, forward circulant, abelian 2-group);
3. methods `is_partial_latin_square()` and `is_latin_square()` to test if a `LatinSquare` object satisfies the definition of a latin square or partial latin square, respectively;
4. tests for completion and unique completion (these use the C++ implementation of Knuth's dancing links algorithm to solve the problem as a instance of $0-1$ matrix exact cover);
5. functions for calculating the τ_i representation of a bitrade and the genus of the associated hypermap embedding;
6. Markov chain of Jacobson and Matthews (1996) for generating latin squares uniformly at random (provides a generator interface);
7. a few examples of τ_i representations of bitrades constructed from the action of a group on itself by right multiplication, functions for converting to a pair of `LatinSquare` objects.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: B.is_latin_square()
True
sage: B[0, 1] = 0
sage: B.is_latin_square()
False

sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
sage: T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
sage: T1.nr_filled_cells()
12
sage: genus(T1, T2)
1
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
```

(continues on next page)

(continued from previous page)

```
>>> B = back_circulant(Integer(5))
>>> B
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
>>> B.is_latin_square()
True
>>> B[Integer(0), Integer(1)] = Integer(0)
>>> B.is_latin_square()
False

>>> (a, b, c, G) = alternating_group_bitrade_generators(Integer(1))
>>> (T1, T2) = bitrade_from_group(a, b, c, G)
>>> T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
>>> T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
>>> T1.nr_filled_cells()
12
>>> genus(T1, T2)
1
```

Todo

1. Latin squares with symbols from a ring instead of the integers $\{0, 1, \ldots, n-1\}$.
2. Isotopism testing of latin squares and bitrades via graph isomorphism (nauty?).
3. Combinatorial constructions for bitrades.

AUTHORS:

- Carlo Hamalainen (2008-03-23): initial version

class sage.combinat.matrices.latin.**LatinSquare**(**args*)

Bases: object

Latin squares.

This class implements a latin square of order n with rows and columns indexed by the set 0, 1, …, n-1 and symbols from the same set. The underlying latin square is a matrix(ZZ, n, n). If L is a latin square, then the cell at row r, column c is empty if and only if L[r, c] < 0. In this way we allow partial latin squares and can speak of completions to latin squares, etc.

There are two ways to declare a latin square:

Empty latin square of order n:

```
sage: n = 3
sage: L = LatinSquare(n)
sage: L
[-1 -1 -1]
[-1 -1 -1]
[-1 -1 -1]
```

```
>>> from sage.all import *
>>> n = Integer(3)
>>> L = LatinSquare(n)
>>> L
[-1 -1 -1]
[-1 -1 -1]
[-1 -1 -1]
```

Latin square from a matrix:

```
sage: M = matrix(ZZ, [[0, 1], [2, 3]])
sage: LatinSquare(M)
[0 1]
[2 3]
```

```
>>> from sage.all import *
>>> M = matrix(ZZ, [[Integer(0), Integer(1)], [Integer(2), Integer(3)]])
>>> LatinSquare(M)
[0 1]
[2 3]
```

actual_row_col_sym_sizes()

Bitrades sometimes end up in partial latin squares with unused rows, columns, or symbols. This function works out the actual number of used rows, columns, and symbols.

> **Warning**
>
> We assume that the unused rows/columns occur in the lower right of self, and that the used symbols are in the range {0, 1, …, m} (no holes in that list).

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
sage: B[0,2] = B[1,2] = B[2,2] = -1
sage: B[0,0] = B[2,1] = -1
sage: B
[-1  1 -1]
[ 1  2 -1]
[ 2 -1 -1]
sage: B.actual_row_col_sym_sizes()
(3, 2, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> B = back_circulant(Integer(3))
>>> B[Integer(0),Integer(2)] = B[Integer(1),Integer(2)] = B[Integer(2),
↪Integer(2)] = -Integer(1)
>>> B[Integer(0),Integer(0)] = B[Integer(2),Integer(1)] = -Integer(1)
>>> B
[-1  1 -1]
[ 1  2 -1]
[ 2 -1 -1]
>>> B.actual_row_col_sym_sizes()
(3, 2, 2)
```

apply_isotopism(*row_perm*, *col_perm*, *sym_perm*)

An isotopism is a permutation of the rows, columns, and symbols of a partial latin square `self`. Use isotopism() to convert a tuple (indexed from 0) to a Permutation object.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: alpha = isotopism((0,1,2,3,4))
sage: beta  = isotopism((1,0,2,3,4))
sage: gamma = isotopism((2,1,0,3,4))
sage: B.apply_isotopism(alpha, beta, gamma)
[3 4 2 0 1]
[0 2 3 1 4]
[1 3 0 4 2]
[4 0 1 2 3]
[2 1 4 3 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> B = back_circulant(Integer(5))
>>> B
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
>>> alpha = isotopism((Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(4)))
>>> beta  = isotopism((Integer(1),Integer(0),Integer(2),Integer(3),
↪Integer(4)))
>>> gamma = isotopism((Integer(2),Integer(1),Integer(0),Integer(3),
↪Integer(4)))
>>> B.apply_isotopism(alpha, beta, gamma)
```

(continues on next page)

(continued from previous page)

```
[3 4 2 0 1]
[0 2 3 1 4]
[1 3 0 4 2]
[4 0 1 2 3]
[2 1 4 3 0]
```

clear_cells()

Mark every cell in `self` as being empty.

EXAMPLES:

```
sage: A = LatinSquare(matrix(ZZ, [[0, 1], [2, 3]]))
sage: A.clear_cells()
sage: A
[-1 -1]
[-1 -1]
```

```
>>> from sage.all import *
>>> A = LatinSquare(matrix(ZZ, [[Integer(0), Integer(1)], [Integer(2),␣
↪Integer(3)]]))
>>> A.clear_cells()
>>> A
[-1 -1]
[-1 -1]
```

column(x)

Return column x of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).column(0)
(0, 1, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> back_circulant(Integer(3)).column(Integer(0))
(0, 1, 2)
```

contained_in(Q)

Return `True` if `self` is a subset of Q.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: P = elementary_abelian_2group(2)
sage: P[0, 0] = -1
sage: P.contained_in(elementary_abelian_2group(2))
True
sage: back_circulant(4).contained_in(elementary_abelian_2group(2))
False
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> P = elementary_abelian_2group(Integer(2))
>>> P[Integer(0), Integer(0)] = -Integer(1)
>>> P.contained_in(elementary_abelian_2group(Integer(2)))
True
>>> back_circulant(Integer(4)).contained_in(elementary_abelian_
↪2group(Integer(2)))
False
```

disjoint_mate_dlxcpp_rows_and_map(*allow_subtrade*)

Internal function for find_disjoint_mates.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(4)
sage: B.disjoint_mate_dlxcpp_rows_and_map(allow_subtrade = True)
([[0, 16, 32],
  [1, 17, 32],
  [2, 18, 32],
  [3, 19, 32],
  [4, 16, 33],
  [5, 17, 33],
  [6, 18, 33],
  [7, 19, 33],
  [8, 16, 34],
  [9, 17, 34],
  [10, 18, 34],
  [11, 19, 34],
  [12, 16, 35],
  [13, 17, 35],
  [14, 18, 35],
  [15, 19, 35],
  [0, 20, 36],
  [1, 21, 36],
  [2, 22, 36],
  [3, 23, 36],
  [4, 20, 37],
  [5, 21, 37],
  [6, 22, 37],
  [7, 23, 37],
  [8, 20, 38],
  [9, 21, 38],
  [10, 22, 38],
  [11, 23, 38],
  [12, 20, 39],
  [13, 21, 39],
  [14, 22, 39],
  [15, 23, 39],
  [0, 24, 40],
  [1, 25, 40],
  [2, 26, 40],
```

(continues on next page)

(continued from previous page)

```
 [3, 27, 40],
 [4, 24, 41],
 [5, 25, 41],
 [6, 26, 41],
 [7, 27, 41],
 [8, 24, 42],
 [9, 25, 42],
 [10, 26, 42],
 [11, 27, 42],
 [12, 24, 43],
 [13, 25, 43],
 [14, 26, 43],
 [15, 27, 43],
 [0, 28, 44],
 [1, 29, 44],
 [2, 30, 44],
 [3, 31, 44],
 [4, 28, 45],
 [5, 29, 45],
 [6, 30, 45],
 [7, 31, 45],
 [8, 28, 46],
 [9, 29, 46],
 [10, 30, 46],
 [11, 31, 46],
 [12, 28, 47],
 [13, 29, 47],
 [14, 30, 47],
 [15, 31, 47]],
{(0, 16, 32): (0, 0, 0),
 (0, 20, 36): (1, 0, 0),
 (0, 24, 40): (2, 0, 0),
 (0, 28, 44): (3, 0, 0),
 (1, 17, 32): (0, 0, 1),
 (1, 21, 36): (1, 0, 1),
 (1, 25, 40): (2, 0, 1),
 (1, 29, 44): (3, 0, 1),
 (2, 18, 32): (0, 0, 2),
 (2, 22, 36): (1, 0, 2),
 (2, 26, 40): (2, 0, 2),
 (2, 30, 44): (3, 0, 2),
 (3, 19, 32): (0, 0, 3),
 (3, 23, 36): (1, 0, 3),
 (3, 27, 40): (2, 0, 3),
 (3, 31, 44): (3, 0, 3),
 (4, 16, 33): (0, 1, 0),
 (4, 20, 37): (1, 1, 0),
 (4, 24, 41): (2, 1, 0),
 (4, 28, 45): (3, 1, 0),
 (5, 17, 33): (0, 1, 1),
 (5, 21, 37): (1, 1, 1),
 (5, 25, 41): (2, 1, 1),
```

(continues on next page)

(continued from previous page)

```
  (5, 29, 45): (3, 1, 1),
  (6, 18, 33): (0, 1, 2),
  (6, 22, 37): (1, 1, 2),
  (6, 26, 41): (2, 1, 2),
  (6, 30, 45): (3, 1, 2),
  (7, 19, 33): (0, 1, 3),
  (7, 23, 37): (1, 1, 3),
  (7, 27, 41): (2, 1, 3),
  (7, 31, 45): (3, 1, 3),
  (8, 16, 34): (0, 2, 0),
  (8, 20, 38): (1, 2, 0),
  (8, 24, 42): (2, 2, 0),
  (8, 28, 46): (3, 2, 0),
  (9, 17, 34): (0, 2, 1),
  (9, 21, 38): (1, 2, 1),
  (9, 25, 42): (2, 2, 1),
  (9, 29, 46): (3, 2, 1),
  (10, 18, 34): (0, 2, 2),
  (10, 22, 38): (1, 2, 2),
  (10, 26, 42): (2, 2, 2),
  (10, 30, 46): (3, 2, 2),
  (11, 19, 34): (0, 2, 3),
  (11, 23, 38): (1, 2, 3),
  (11, 27, 42): (2, 2, 3),
  (11, 31, 46): (3, 2, 3),
  (12, 16, 35): (0, 3, 0),
  (12, 20, 39): (1, 3, 0),
  (12, 24, 43): (2, 3, 0),
  (12, 28, 47): (3, 3, 0),
  (13, 17, 35): (0, 3, 1),
  (13, 21, 39): (1, 3, 1),
  (13, 25, 43): (2, 3, 1),
  (13, 29, 47): (3, 3, 1),
  (14, 18, 35): (0, 3, 2),
  (14, 22, 39): (1, 3, 2),
  (14, 26, 43): (2, 3, 2),
  (14, 30, 47): (3, 3, 2),
  (15, 19, 35): (0, 3, 3),
  (15, 23, 39): (1, 3, 3),
  (15, 27, 43): (2, 3, 3),
  (15, 31, 47): (3, 3, 3)})
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> B = back_circulant(Integer(4))
>>> B.disjoint_mate_dlxcpp_rows_and_map(allow_subtrade = True)
([[0, 16, 32],
  [1, 17, 32],
  [2, 18, 32],
  [3, 19, 32],
  [4, 16, 33],
  [5, 17, 33],
```

(continues on next page)

(continued from previous page)

```
 [6, 18, 33],
 [7, 19, 33],
 [8, 16, 34],
 [9, 17, 34],
 [10, 18, 34],
 [11, 19, 34],
 [12, 16, 35],
 [13, 17, 35],
 [14, 18, 35],
 [15, 19, 35],
 [0, 20, 36],
 [1, 21, 36],
 [2, 22, 36],
 [3, 23, 36],
 [4, 20, 37],
 [5, 21, 37],
 [6, 22, 37],
 [7, 23, 37],
 [8, 20, 38],
 [9, 21, 38],
 [10, 22, 38],
 [11, 23, 38],
 [12, 20, 39],
 [13, 21, 39],
 [14, 22, 39],
 [15, 23, 39],
 [0, 24, 40],
 [1, 25, 40],
 [2, 26, 40],
 [3, 27, 40],
 [4, 24, 41],
 [5, 25, 41],
 [6, 26, 41],
 [7, 27, 41],
 [8, 24, 42],
 [9, 25, 42],
 [10, 26, 42],
 [11, 27, 42],
 [12, 24, 43],
 [13, 25, 43],
 [14, 26, 43],
 [15, 27, 43],
 [0, 28, 44],
 [1, 29, 44],
 [2, 30, 44],
 [3, 31, 44],
 [4, 28, 45],
 [5, 29, 45],
 [6, 30, 45],
 [7, 31, 45],
 [8, 28, 46],
 [9, 29, 46],
```

(continues on next page)

(continued from previous page)

```
 [10, 30, 46],
 [11, 31, 46],
 [12, 28, 47],
 [13, 29, 47],
 [14, 30, 47],
 [15, 31, 47]],
{(0, 16, 32): (0, 0, 0),
 (0, 20, 36): (1, 0, 0),
 (0, 24, 40): (2, 0, 0),
 (0, 28, 44): (3, 0, 0),
 (1, 17, 32): (0, 0, 1),
 (1, 21, 36): (1, 0, 1),
 (1, 25, 40): (2, 0, 1),
 (1, 29, 44): (3, 0, 1),
 (2, 18, 32): (0, 0, 2),
 (2, 22, 36): (1, 0, 2),
 (2, 26, 40): (2, 0, 2),
 (2, 30, 44): (3, 0, 2),
 (3, 19, 32): (0, 0, 3),
 (3, 23, 36): (1, 0, 3),
 (3, 27, 40): (2, 0, 3),
 (3, 31, 44): (3, 0, 3),
 (4, 16, 33): (0, 1, 0),
 (4, 20, 37): (1, 1, 0),
 (4, 24, 41): (2, 1, 0),
 (4, 28, 45): (3, 1, 0),
 (5, 17, 33): (0, 1, 1),
 (5, 21, 37): (1, 1, 1),
 (5, 25, 41): (2, 1, 1),
 (5, 29, 45): (3, 1, 1),
 (6, 18, 33): (0, 1, 2),
 (6, 22, 37): (1, 1, 2),
 (6, 26, 41): (2, 1, 2),
 (6, 30, 45): (3, 1, 2),
 (7, 19, 33): (0, 1, 3),
 (7, 23, 37): (1, 1, 3),
 (7, 27, 41): (2, 1, 3),
 (7, 31, 45): (3, 1, 3),
 (8, 16, 34): (0, 2, 0),
 (8, 20, 38): (1, 2, 0),
 (8, 24, 42): (2, 2, 0),
 (8, 28, 46): (3, 2, 0),
 (9, 17, 34): (0, 2, 1),
 (9, 21, 38): (1, 2, 1),
 (9, 25, 42): (2, 2, 1),
 (9, 29, 46): (3, 2, 1),
 (10, 18, 34): (0, 2, 2),
 (10, 22, 38): (1, 2, 2),
 (10, 26, 42): (2, 2, 2),
 (10, 30, 46): (3, 2, 2),
 (11, 19, 34): (0, 2, 3),
 (11, 23, 38): (1, 2, 3),
```

(continues on next page)

(continued from previous page)

```
  (11, 27, 42): (2, 2, 3),
  (11, 31, 46): (3, 2, 3),
  (12, 16, 35): (0, 3, 0),
  (12, 20, 39): (1, 3, 0),
  (12, 24, 43): (2, 3, 0),
  (12, 28, 47): (3, 3, 0),
  (13, 17, 35): (0, 3, 1),
  (13, 21, 39): (1, 3, 1),
  (13, 25, 43): (2, 3, 1),
  (13, 29, 47): (3, 3, 1),
  (14, 18, 35): (0, 3, 2),
  (14, 22, 39): (1, 3, 2),
  (14, 26, 43): (2, 3, 2),
  (14, 30, 47): (3, 3, 2),
  (15, 19, 35): (0, 3, 3),
  (15, 23, 39): (1, 3, 3),
  (15, 27, 43): (2, 3, 3),
  (15, 31, 47): (3, 3, 3)})
```

dlxcpp_has_unique_completion()

Check if the partial latin square `self` of order n can be embedded in precisely one latin square of order n.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(2).dlxcpp_has_unique_completion()
True
sage: P = LatinSquare(2)
sage: P.dlxcpp_has_unique_completion()
False
sage: P[0, 0] = 0
sage: P.dlxcpp_has_unique_completion()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> back_circulant(Integer(2)).dlxcpp_has_unique_completion()
True
>>> P = LatinSquare(Integer(2))
>>> P.dlxcpp_has_unique_completion()
False
>>> P[Integer(0), Integer(0)] = Integer(0)
>>> P.dlxcpp_has_unique_completion()
True
```

dumps()

Since the latin square class does not hold any other private variables we just call dumps on self.square:

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(2) == loads(dumps(back_circulant(2)))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> back_circulant(Integer(2)) == loads(dumps(back_circulant(Integer(2))))
True
```

filled_cells_map()

Number the filled cells of `self` with integers from $\{1, 2, 3, \ldots\}$.

INPUT:

- `self` – partial latin square `self` (empty cells have negative values)

OUTPUT:

A dictionary `cells_map` where `cells_map[(i,j)] = m` means that `(i,j)` is the `m`-th filled cell in `P`, while `cells_map[m] = (i,j)`.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: D = T1.filled_cells_map()
sage: {i: v for i,v in D.items() if i in ZZ}
{1: (0, 0),
 2: (0, 2),
 3: (0, 3),
 4: (1, 1),
 5: (1, 2),
 6: (1, 3),
 7: (2, 0),
 8: (2, 1),
 9: (2, 2),
 10: (3, 0),
 11: (3, 1),
 12: (3, 3)}
sage: {i: v for i,v in D.items() if i not in ZZ}
{(0, 0): 1,
 (0, 2): 2,
 (0, 3): 3,
 (1, 1): 4,
 (1, 2): 5,
 (1, 3): 6,
 (2, 0): 7,
 (2, 1): 8,
 (2, 2): 9,
 (3, 0): 10,
 (3, 1): 11,
 (3, 3): 12}
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> (a, b, c, G) = alternating_group_bitrade_generators(Integer(1))
>>> (T1, T2) = bitrade_from_group(a, b, c, G)
>>> D = T1.filled_cells_map()
```

(continues on next page)

(continued from previous page)

```
>>> {i: v for i,v in D.items() if i in ZZ}
{1: (0, 0),
 2: (0, 2),
 3: (0, 3),
 4: (1, 1),
 5: (1, 2),
 6: (1, 3),
 7: (2, 0),
 8: (2, 1),
 9: (2, 2),
 10: (3, 0),
 11: (3, 1),
 12: (3, 3)}
>>> {i: v for i,v in D.items() if i not in ZZ}
{(0, 0): 1,
 (0, 2): 2,
 (0, 3): 3,
 (1, 1): 4,
 (1, 2): 5,
 (1, 3): 6,
 (2, 0): 7,
 (2, 1): 8,
 (2, 2): 9,
 (3, 0): 10,
 (3, 1): 11,
 (3, 3): 12}
```

find_disjoint_mates(*nr_to_find=None*, *allow_subtrade=False*)

> **Warning**
>
> If allow_subtrade is `True` then we may return a partial latin square that is *not* disjoint to `self`. In that case, use bitrade(P, Q) to get an actual bitrade.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(4)
sage: g = B.find_disjoint_mates(allow_subtrade = True)
sage: B1 = next(g)
sage: B0, B1 = bitrade(B, B1)
sage: assert is_bitrade(B0, B1)
sage: print(B0)
[-1  1  2 -1]
[-1  2 -1  0]
[-1 -1 -1 -1]
[-1  0  1  2]
sage: print(B1)
[-1  2  1 -1]
[-1  0 -1  2]
[-1 -1 -1 -1]
```

(continues on next page)

(continued from previous page)

```
[-1  1  2  0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> B = back_circulant(Integer(4))
>>> g = B.find_disjoint_mates(allow_subtrade = True)
>>> B1 = next(g)
>>> B0, B1 = bitrade(B, B1)
>>> assert is_bitrade(B0, B1)
>>> print(B0)
[-1  1  2 -1]
[-1  2 -1  0]
[-1 -1 -1 -1]
[-1  0  1  2]
>>> print(B1)
[-1  2  1 -1]
[-1  0 -1  2]
[-1 -1 -1 -1]
[-1  1  2  0]
```

gcs()

A greedy critical set of a latin square `self` is found by successively removing elements in a row-wise (bottom-up) manner, checking for unique completion at each step.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: A = elementary_abelian_2group(3)
sage: G = A.gcs()
sage: A
[0 1 2 3 4 5 6 7]
[1 0 3 2 5 4 7 6]
[2 3 0 1 6 7 4 5]
[3 2 1 0 7 6 5 4]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
sage: G
[ 0  1  2  3  4  5  6 -1]
[ 1  0  3  2  5  4 -1 -1]
[ 2  3  0  1  6 -1  4 -1]
[ 3  2  1  0 -1 -1 -1 -1]
[ 4  5  6 -1  0  1  2 -1]
[ 5  4 -1 -1  1  0 -1 -1]
[ 6 -1  4 -1  2 -1  0 -1]
[-1 -1 -1 -1 -1 -1 -1 -1]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> A = elementary_abelian_2group(Integer(3))
>>> G = A.gcs()
```

(continues on next page)

(continued from previous page)

```
>>> A
[0 1 2 3 4 5 6 7]
[1 0 3 2 5 4 7 6]
[2 3 0 1 6 7 4 5]
[3 2 1 0 7 6 5 4]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
>>> G
[ 0  1  2  3  4  5  6 -1]
[ 1  0  3  2  5  4 -1 -1]
[ 2  3  0  1  6 -1  4 -1]
[ 3  2  1  0 -1 -1 -1 -1]
[ 4  5  6 -1  0  1  2 -1]
[ 5  4 -1 -1  1  0 -1 -1]
[ 6 -1  4 -1  2 -1  0 -1]
[-1 -1 -1 -1 -1 -1 -1 -1]
```

is_completable()

Return `True` if the partial latin square can be completed to a latin square.

EXAMPLES:

The following partial latin square has no completion because there is nowhere that we can place the symbol 0 in the third row:

```
sage: B = LatinSquare(3)
```

```
>>> from sage.all import *
>>> B = LatinSquare(Integer(3))
```

```
sage: B[0, 0] = 0
sage: B[1, 1] = 0
sage: B[2, 2] = 1
```

```
>>> from sage.all import *
>>> B[Integer(0), Integer(0)] = Integer(0)
>>> B[Integer(1), Integer(1)] = Integer(0)
>>> B[Integer(2), Integer(2)] = Integer(1)
```

```
sage: B
[ 0 -1 -1]
[-1  0 -1]
[-1 -1  1]
```

```
>>> from sage.all import *
>>> B
[ 0 -1 -1]
[-1  0 -1]
[-1 -1  1]
```

```
sage: B.is_completable()
False
```

```
>>> from sage.all import *
>>> B.is_completable()
False
```

```
sage: B[2, 2] = 0
sage: B.is_completable()
True
```

```
>>> from sage.all import *
>>> B[Integer(2), Integer(2)] = Integer(0)
>>> B.is_completable()
True
```

is_empty_column(*c*)

Check if column c of the partial latin square `self` is empty.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(4)
sage: L.is_empty_column(0)
False
sage: L[0,0] = L[1,0] = L[2,0] = L[3,0] = -1
sage: L.is_empty_column(0)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> L = back_circulant(Integer(4))
>>> L.is_empty_column(Integer(0))
False
>>> L[Integer(0),Integer(0)] = L[Integer(1),Integer(0)] = L[Integer(2),
↪Integer(0)] = L[Integer(3),Integer(0)] = -Integer(1)
>>> L.is_empty_column(Integer(0))
True
```

is_empty_row(*r*)

Check if row r of the partial latin square `self` is empty.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(4)
sage: L.is_empty_row(0)
False
sage: L[0,0] = L[0,1] = L[0,2] = L[0,3] = -1
sage: L.is_empty_row(0)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> L = back_circulant(Integer(4))
>>> L.is_empty_row(Integer(0))
False
>>> L[Integer(0),Integer(0)] = L[Integer(0),Integer(1)] = L[Integer(0),
↪Integer(2)] = L[Integer(0),Integer(3)] = -Integer(1)
>>> L.is_empty_row(Integer(0))
True
```

is_latin_square()

self is a latin square if it is an n by n matrix, and each symbol in [0, 1, …, n-1] appears exactly once in each row, and exactly once in each column.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: elementary_abelian_2group(4).is_latin_square()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> elementary_abelian_2group(Integer(4)).is_latin_square()
True
```

```
sage: forward_circulant(7).is_latin_square()
True
```

```
>>> from sage.all import *
>>> forward_circulant(Integer(7)).is_latin_square()
True
```

is_partial_latin_square()

self is a partial latin square if it is an n by n matrix, and each symbol in [0, 1, …, n-1] appears at most once in each row, and at most once in each column.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: LatinSquare(4).is_partial_latin_square()
True
sage: back_circulant(3).gcs().is_partial_latin_square()
True
sage: back_circulant(6).is_partial_latin_square()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> LatinSquare(Integer(4)).is_partial_latin_square()
True
>>> back_circulant(Integer(3)).gcs().is_partial_latin_square()
True
```

(continues on next page)

(continued from previous page)

```
>>> back_circulant(Integer(6)).is_partial_latin_square()
True
```

is_uniquely_completable()

Return `True` if the partial latin square `self` has exactly one completion to a latin square. This is just a wrapper for the current best-known algorithm, Dancing Links by Knuth. See dancing_links.spyx

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(4).gcs().is_uniquely_completable()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> back_circulant(Integer(4)).gcs().is_uniquely_completable()
True
```

```
sage: G = elementary_abelian_2group(3).gcs()
sage: G.is_uniquely_completable()
True
```

```
>>> from sage.all import *
>>> G = elementary_abelian_2group(Integer(3)).gcs()
>>> G.is_uniquely_completable()
True
```

```
sage: G[0, 0] = -1
sage: G.is_uniquely_completable()
False
```

```
>>> from sage.all import *
>>> G[Integer(0), Integer(0)] = -Integer(1)
>>> G.is_uniquely_completable()
False
```

latex()

Return LaTeX code for the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: print(back_circulant(3).latex())
\begin{array}{|c|c|c|}\hline 0 & 1 & 2\\\hline 1 & 2 & 0\\\hline 2 & 0 & 1\\
↪hline\end{array}
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> print(back_circulant(Integer(3)).latex())
\begin{array}{|c|c|c|}\hline 0 & 1 & 2\\\hline 1 & 2 & 0\\\hline 2 & 0 & 1\\
↪hline\end{array}
```

list()

Convert the latin square into a list, in a row-wise manner.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).list()
[0, 1, 2, 1, 2, 0, 2, 0, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> back_circulant(Integer(3)).list()
[0, 1, 2, 1, 2, 0, 2, 0, 1]
```

ncols()

Number of columns in the latin square.

EXAMPLES:

```
sage: LatinSquare(3).ncols()
3
```

```
>>> from sage.all import *
>>> LatinSquare(Integer(3)).ncols()
3
```

nr_distinct_symbols()

Return the number of distinct symbols in the partial latin square `self`.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(5).nr_distinct_symbols()
5
sage: L = LatinSquare(10)
sage: L.nr_distinct_symbols()
0
sage: L[0, 0] = 0
sage: L[0, 1] = 1
sage: L.nr_distinct_symbols()
2
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> back_circulant(Integer(5)).nr_distinct_symbols()
5
>>> L = LatinSquare(Integer(10))
>>> L.nr_distinct_symbols()
0
>>> L[Integer(0), Integer(0)] = Integer(0)
>>> L[Integer(0), Integer(1)] = Integer(1)
>>> L.nr_distinct_symbols()
2
```

nr_filled_cells()

Return the number of filled cells (i.e. cells with a positive value) in the partial latin square `self`.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: LatinSquare(matrix([[0, -1], [-1, 0]])).nr_filled_cells()
2
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> LatinSquare(matrix([[Integer(0), -Integer(1)], [-Integer(1),
→Integer(0)]])).nr_filled_cells()
2
```

nrows()

Number of rows in the latin square.

EXAMPLES:

```
sage: LatinSquare(3).nrows()
3
```

```
>>> from sage.all import *
>>> LatinSquare(Integer(3)).nrows()
3
```

permissable_values(*r*, *c*)

Find all values that do not appear in row r and column c of the latin square `self`. If `self[r, c]` is filled then we return the empty list.

INPUT:

- `self` – LatinSquare
- `r` – integer; row of the latin square
- `c` – integer; column of the latin square

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(5)
sage: L[0, 0] = -1
sage: L.permissable_values(0, 0)
[0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> L = back_circulant(Integer(5))
>>> L[Integer(0), Integer(0)] = -Integer(1)
>>> L.permissable_values(Integer(0), Integer(0))
[0]
```

random_empty_cell()

Find an empty cell of self, uniformly at random.

INPUT:

- `self` – LatinSquare

OUTPUT:

- `[r, c]` – cell such that `self[r, c]` is empty, or returns `None` if `self` is a (full) latin square

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: P = back_circulant(2)
sage: P[1,1] = -1
sage: P.random_empty_cell()
[1, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> P = back_circulant(Integer(2))
>>> P[Integer(1),Integer(1)] = -Integer(1)
>>> P.random_empty_cell()
[1, 1]
```

row(*x*)

Return row x of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).row(0)
(0, 1, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> back_circulant(Integer(3)).row(Integer(0))
(0, 1, 2)
```

set_immutable()

A latin square is immutable if the underlying matrix is immutable.

EXAMPLES:

```
sage: L = LatinSquare(matrix(ZZ, [[0, 1], [2, 3]]))
sage: L.set_immutable()
sage: {L : 0}    # this would fail without set_immutable()
{[0 1]
[2 3]: 0}
```

```
>>> from sage.all import *
>>> L = LatinSquare(matrix(ZZ, [[Integer(0), Integer(1)], [Integer(2),
→Integer(3)]]))
>>> L.set_immutable()
>>> {L : Integer(0)}    # this would fail without set_immutable()
{[0 1]
[2 3]: 0}
```

top_left_empty_cell()

Return the least `[r, c]` such that `self[r, c]` is an empty cell. If all cells are filled then we return `None`.

INPUT:

- `self` – LatinSquare

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B[3, 4] = -1
sage: B.top_left_empty_cell()
[3, 4]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> B = back_circulant(Integer(5))
>>> B[Integer(3), Integer(4)] = -Integer(1)
>>> B.top_left_empty_cell()
[3, 4]
```

vals_in_col(*c*)

Return a dictionary with key e if and only if column c of `self` has the symbol e.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
sage: B[0, 0] = -1
sage: back_circulant(3).vals_in_col(0)
{0: True, 1: True, 2: True}
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> B = back_circulant(Integer(3))
>>> B[Integer(0), Integer(0)] = -Integer(1)
>>> back_circulant(Integer(3)).vals_in_col(Integer(0))
{0: True, 1: True, 2: True}
```

vals_in_row(*r*)

Return a dictionary with key e if and only if row r of `self` has the symbol e.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
sage: B[0, 0] = -1
sage: back_circulant(3).vals_in_row(0)
{0: True, 1: True, 2: True}
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> B = back_circulant(Integer(3))
>>> B[Integer(0), Integer(0)] = -Integer(1)
```

(continues on next page)

(continued from previous page)

```
>>> back_circulant(Integer(3)).vals_in_row(Integer(0))
{0: True, 1: True, 2: True}
```

sage.combinat.matrices.latin.**LatinSquare_generator**(*L_start*, *check_assertions=False*)

Generator for a sequence of uniformly distributed latin squares, given L_start as the initial latin square.

This code implements the Markov chain algorithm of Jacobson and Matthews (1996), see below for the BibTex entry. This generator will never throw the `StopIteration` exception, so it provides an infinite sequence of latin squares.

EXAMPLES:

Use the back circulant latin square of order 4 as the initial square and print the next two latin squares given by the Markov chain:

```
sage: from sage.combinat.matrices.latin import *
sage: g = LatinSquare_generator(back_circulant(4))
sage: next(g).is_latin_square()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> g = LatinSquare_generator(back_circulant(Integer(4)))
>>> next(g).is_latin_square()
True
```

REFERENCES:

sage.combinat.matrices.latin.**alternating_group_bitrade_generators**(*m*)

Construct generators a, b, c for the alternating group on 3m+1 points, such that a*b*c = 1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = alternating_group_bitrade_generators(1)
sage: (a, b, c, G)
((1,2,3), (1,4,2), (2,4,3), Permutation Group with generators [(1,2,3), (1,4,2)])
sage: a*b*c
()
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> a, b, c, G = alternating_group_bitrade_generators(Integer(1))
>>> (a, b, c, G)
((1,2,3), (1,4,2), (2,4,3), Permutation Group with generators [(1,2,3), (1,4,2)])
>>> a*b*c
()
```

```
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
```

(continues on next page)

(continued from previous page)

```
sage: T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
```

```
>>> from sage.all import *
>>> (T1, T2) = bitrade_from_group(a, b, c, G)
>>> T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
>>> T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
```

sage.combinat.matrices.latin.**back_circulant**(*n*)

The back-circulant latin square of order n is the Cayley table for (Z_n, +), the integers under addition modulo n.

INPUT:

- n – integer; order of the latin square

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(5)
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> back_circulant(Integer(5))
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
```

sage.combinat.matrices.latin.**beta1**(*rce*, *T1*, *T2*)

Find the unique (x, c, e) in T2 such that (r, c, e) is in T1.

INPUT:

- rce – tuple (or list) (r, c, e) in T1
- T1, T2 – latin bitrade

OUTPUT: (x, c, e) in T2

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta1([0, 0, 0], T1, T2)
(1, 0, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
>>> T2 = T1.apply_isotopism(x, y, z)
>>> is_bitrade(T1, T2)
True
>>> beta1([Integer(0), Integer(0), Integer(0)], T1, T2)
(1, 0, 0)
```

sage.combinat.matrices.latin.**beta2**(*rce*, *T1*, *T2*)

Find the unique (r, x, e) in T2 such that (r, c, e) is in T1.

INPUT:

- `rce` – tuple (or list) (r, c, e) in T1
- `T1`, `T2` – latin bitrade

OUTPUT: (r, x, e) in T2

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta2([0, 0, 0], T1, T2)
(0, 1, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
```

(continues on next page)

(continued from previous page)

```
>>> T2 = T1.apply_isotopism(x, y, z)
>>> is_bitrade(T1, T2)
True
>>> beta2([Integer(0), Integer(0), Integer(0)], T1, T2)
(0, 1, 0)
```

sage.combinat.matrices.latin.**beta3**(*rce*, *T1*, *T2*)

Find the unique (r, c, x) in T2 such that (r, c, e) is in T1.

INPUT:

- `rce` – tuple (or list) (r, c, e) in T1
- `T1, T2` – latin bitrade

OUTPUT: (r, c, x) in T2.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta3([0, 0, 0], T1, T2)
(0, 0, 4)
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
>>> T2 = T1.apply_isotopism(x, y, z)
>>> is_bitrade(T1, T2)
True
>>> beta3([Integer(0), Integer(0), Integer(0)], T1, T2)
(0, 0, 4)
```

sage.combinat.matrices.latin.**bitrade**(*T1*, *T2*)

Form the bitrade (Q1, Q2) from (T1, T2) by setting empty the cells (r, c) such that T1[r, c] == T2[r, c].

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B1 = back_circulant(5)
sage: alpha = isotopism((0,1,2,3,4))
sage: beta  = isotopism((1,0,2,3,4))
sage: gamma = isotopism((2,1,0,3,4))
sage: B2 = B1.apply_isotopism(alpha, beta, gamma)
sage: T1, T2 = bitrade(B1, B2)
sage: T1
```

(continues on next page)

(continued from previous page)

```
[ 0  1 -1  3  4]
[ 1 -1 -1  4  0]
[ 2 -1  4  0  1]
[ 3  4  0  1  2]
[ 4  0  1  2  3]
sage: T2
[ 3  4 -1  0  1]
[ 0 -1 -1  1  4]
[ 1 -1  0  4  2]
[ 4  0  1  2  3]
[ 2  1  4  3  0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> B1 = back_circulant(Integer(5))
>>> alpha = isotopism((Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)))
>>> beta  = isotopism((Integer(1),Integer(0),Integer(2),Integer(3),Integer(4)))
>>> gamma = isotopism((Integer(2),Integer(1),Integer(0),Integer(3),Integer(4)))
>>> B2 = B1.apply_isotopism(alpha, beta, gamma)
>>> T1, T2 = bitrade(B1, B2)
>>> T1
[ 0  1 -1  3  4]
[ 1 -1 -1  4  0]
[ 2 -1  4  0  1]
[ 3  4  0  1  2]
[ 4  0  1  2  3]
>>> T2
[ 3  4 -1  0  1]
[ 0 -1 -1  1  4]
[ 1 -1  0  4  2]
[ 4  0  1  2  3]
[ 2  1  4  3  0]
```

sage.combinat.matrices.latin.**bitrade_from_group**(*a*, *b*, *c*, *G*)

Given group elements a, b, c in G such that abc = 1 and the subgroups a, b, c intersect (pairwise) only in the identity, construct a bitrade (T1, T2) where rows, columns, and symbols correspond to cosets of a, b, and c, respectively.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
sage: T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> a, b, c, G = alternating_group_bitrade_generators(Integer(1))
>>> (T1, T2) = bitrade_from_group(a, b, c, G)
>>> T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
>>> T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
```

sage.combinat.matrices.latin.**cells_map_as_square**(*cells_map*, *n*)

Return a LatinSquare with cells numbered from 1, 2, … to given the dictionary cells_map.

Note

The value n should be the maximum of the number of rows and columns of the original partial latin square

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> (a, b, c, G) = alternating_group_bitrade_generators(Integer(1))
>>> (T1, T2) = bitrade_from_group(a, b, c, G)
>>> T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
```

There are 12 filled cells in T:

```
sage: cells_map_as_square(T1.filled_cells_map(), max(T1.nrows(), T1.ncols()))
[ 1 -1  2  3]
[-1  4  5  6]
[ 7  8  9 -1]
[10 11 -1 12]
```

```
>>> from sage.all import *
>>> cells_map_as_square(T1.filled_cells_map(), max(T1.nrows(), T1.ncols()))
[ 1 -1  2  3]
[-1  4  5  6]
[ 7  8  9 -1]
[10 11 -1 12]
```

sage.combinat.matrices.latin.**check_bitrade_generators**(*a*, *b*, *c*)

Three group elements a, b, c will generate a bitrade if a*b*c = 1 and the subgroups a, b, c intersect (pairwise) in just the identity.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = p3_group_bitrade_generators(3)
sage: check_bitrade_generators(a, b, c)
True
sage: check_bitrade_generators(a, b, libgap(gap('()')))
False
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> a, b, c, G = p3_group_bitrade_generators(Integer(3))
>>> check_bitrade_generators(a, b, c)
True
>>> check_bitrade_generators(a, b, libgap(gap('()')))
False
```

sage.combinat.matrices.latin.**coin**()

Simulate a fair coin (returns `True` or `False`) using ZZ.random_element(2).

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import coin
sage: x = coin()
sage: x == 0 or x == 1
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import coin
>>> x = coin()
>>> x == Integer(0) or x == Integer(1)
True
```

sage.combinat.matrices.latin.**column_containing_sym**(*L*, *r*, *x*)

Given an improper latin square L with L[r, c1] = L[r, c2] = x, return c1 or c2 with equal probability. This is an internal function and should only be used in LatinSquare_generator().

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = matrix([(1, 0, 2, 3), (0, 2, 3, 0), (2, 3, 0, 1), (3, 0, 1, 2)])
sage: L
```

(continues on next page)

(continued from previous page)

```
[1 0 2 3]
[0 2 3 0]
[2 3 0 1]
[3 0 1 2]
sage: c = column_containing_sym(L, 1, 0)
sage: c == 0 or c == 3
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> L = matrix([(Integer(1), Integer(0), Integer(2), Integer(3)), (Integer(0),␣
↪Integer(2), Integer(3), Integer(0)), (Integer(2), Integer(3), Integer(0),␣
↪Integer(1)), (Integer(3), Integer(0), Integer(1), Integer(2))])
>>> L
[1 0 2 3]
[0 2 3 0]
[2 3 0 1]
[3 0 1 2]
>>> c = column_containing_sym(L, Integer(1), Integer(0))
>>> c == Integer(0) or c == Integer(3)
True
```

sage.combinat.matrices.latin.**direct_product**(*L1*, *L2*, *L3*, *L4*)

The 'direct product' of four latin squares L1, L2, L3, L4 of order n is the latin square of order 2n consisting of

```
-----------
| L1 | L2 |
-----------
| L3 | L4 |
-----------
```

where the subsquares L2 and L3 have entries offset by n.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: direct_product(back_circulant(4), back_circulant(4), elementary_abelian_
↪2group(2), elementary_abelian_2group(2))
[0 1 2 3 4 5 6 7]
[1 2 3 0 5 6 7 4]
[2 3 0 1 6 7 4 5]
[3 0 1 2 7 4 5 6]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> direct_product(back_circulant(Integer(4)), back_circulant(Integer(4)),␣
↪elementary_abelian_2group(Integer(2)), elementary_abelian_2group(Integer(2)))
[0 1 2 3 4 5 6 7]
```

(continues on next page)

(continued from previous page)

```
[1 2 3 0 5 6 7 4]
[2 3 0 1 6 7 4 5]
[3 0 1 2 7 4 5 6]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
```

sage.combinat.matrices.latin.**dlxcpp_find_completions**(*P*, *nr_to_find=None*)

Return a list of all latin squares L of the same order as P such that P is contained in L. The optional parameter nr_to_find limits the number of latin squares that are found.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: dlxcpp_find_completions(LatinSquare(2))
[[0 1]
[1 0], [1 0]
[0 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> dlxcpp_find_completions(LatinSquare(Integer(2)))
[[0 1]
[1 0], [1 0]
[0 1]]
```

```
sage: dlxcpp_find_completions(LatinSquare(2), 1)
[[0 1]
[1 0]]
```

```
>>> from sage.all import *
>>> dlxcpp_find_completions(LatinSquare(Integer(2)), Integer(1))
[[0 1]
[1 0]]
```

sage.combinat.matrices.latin.**dlxcpp_rows_and_map**(*P*)

Internal function for `dlxcpp_find_completions`. Given a partial latin square P we construct a list of rows of a 0-1 matrix M such that an exact cover of M corresponds to a completion of P to a latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: dlxcpp_rows_and_map(LatinSquare(2))
([[0, 4, 8],
  [1, 5, 8],
  [2, 4, 9],
  [3, 5, 9],
  [0, 6, 10],
  [1, 7, 10],
  [2, 6, 11],
  [3, 7, 11]],
```

(continues on next page)

(continued from previous page)

```
 {(0, 4, 8): (0, 0, 0),
  (0, 6, 10): (1, 0, 0),
  (1, 5, 8): (0, 0, 1),
  (1, 7, 10): (1, 0, 1),
  (2, 4, 9): (0, 1, 0),
  (2, 6, 11): (1, 1, 0),
  (3, 5, 9): (0, 1, 1),
  (3, 7, 11): (1, 1, 1)})
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> dlxcpp_rows_and_map(LatinSquare(Integer(2)))
([[0, 4, 8],
  [1, 5, 8],
  [2, 4, 9],
  [3, 5, 9],
  [0, 6, 10],
  [1, 7, 10],
  [2, 6, 11],
  [3, 7, 11]],
 {(0, 4, 8): (0, 0, 0),
  (0, 6, 10): (1, 0, 0),
  (1, 5, 8): (0, 0, 1),
  (1, 7, 10): (1, 0, 1),
  (2, 4, 9): (0, 1, 0),
  (2, 6, 11): (1, 1, 0),
  (3, 5, 9): (0, 1, 1),
  (3, 7, 11): (1, 1, 1)})
```

sage.combinat.matrices.latin.**elementary_abelian_2group**(*s*)

Return the latin square based on the Cayley table for the elementary abelian 2-group of order 2s.

INPUT:

- `s` – integer; order of the latin square will be 2s

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: elementary_abelian_2group(3)
[0 1 2 3 4 5 6 7]
[1 0 3 2 5 4 7 6]
[2 3 0 1 6 7 4 5]
[3 2 1 0 7 6 5 4]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> elementary_abelian_2group(Integer(3))
[0 1 2 3 4 5 6 7]
```

(continues on next page)

(continued from previous page)

```
[1 0 3 2 5 4 7 6]
[2 3 0 1 6 7 4 5]
[3 2 1 0 7 6 5 4]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
```

sage.combinat.matrices.latin.**forward_circulant**(*n*)

The forward-circulant latin square of order n is the Cayley table for the operation r + c = (n-c+r) mod n.

INPUT:

- n – integer; order of the latin square

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: forward_circulant(5)
[0 4 3 2 1]
[1 0 4 3 2]
[2 1 0 4 3]
[3 2 1 0 4]
[4 3 2 1 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> forward_circulant(Integer(5))
[0 4 3 2 1]
[1 0 4 3 2]
[2 1 0 4 3]
[3 2 1 0 4]
[4 3 2 1 0]
```

sage.combinat.matrices.latin.**genus**(*T1*, *T2*)

Return the genus of hypermap embedding associated with the bitrade (T1, T2).

Informally, we compute the [tau_1, tau_2, tau_3] permutation representation of the bitrade. Each cycle of tau_1, tau_2, and tau_3 gives a rotation scheme for a black, white, and star vertex (respectively). The genus then comes from Euler's formula.

For more details see Carlo Hamalainen: *Partitioning 3-homogeneous latin bitrades*. To appear in Geometriae Dedicata, available at arXiv 0710.0938

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: genus(T1, T2)
1
sage: (a, b, c, G) = pq_group_bitrade_generators(3, 7)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: genus(T1, T2)
3
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> (a, b, c, G) = alternating_group_bitrade_generators(Integer(1))
>>> (T1, T2) = bitrade_from_group(a, b, c, G)
>>> genus(T1, T2)
1
>>> (a, b, c, G) = pq_group_bitrade_generators(Integer(3), Integer(7))
>>> (T1, T2) = bitrade_from_group(a, b, c, G)
>>> genus(T1, T2)
3
```

sage.combinat.matrices.latin.**group_to_LatinSquare**(*G*)

Construct a latin square on the symbols [0, 1, …, n-1] for a group with an n by n Cayley table.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import group_to_LatinSquare

sage: group_to_LatinSquare(DihedralGroup(2))
[0 1 2 3]
[1 0 3 2]
[2 3 0 1]
[3 2 1 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import group_to_LatinSquare

>>> group_to_LatinSquare(DihedralGroup(Integer(2)))
[0 1 2 3]
[1 0 3 2]
[2 3 0 1]
[3 2 1 0]
```

```
sage: G = libgap.Group(PermutationGroupElement((1,2,3)))
sage: group_to_LatinSquare(G)
[0 1 2]
[1 2 0]
[2 0 1]
```

```
>>> from sage.all import *
>>> G = libgap.Group(PermutationGroupElement((Integer(1),Integer(2),Integer(3))))
>>> group_to_LatinSquare(G)
[0 1 2]
[1 2 0]
[2 0 1]
```

sage.combinat.matrices.latin.**is_bitrade**(*T1*, *T2*)

Combinatorially, a pair (T1, T2) of partial latin squares is a bitrade if they are disjoint, have the same shape, and have row and column balance. For definitions of each of these terms see the relevant function in this file.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
>>> T2 = T1.apply_isotopism(x, y, z)
>>> is_bitrade(T1, T2)
True
```

sage.combinat.matrices.latin.**is_disjoint**(*T1*, *T2*)

The partial latin squares T1 and T2 are disjoint if T1[r, c] != T2[r, c] or T1[r, c] == T2[r, c] == -1 for each cell [r, c].

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import is_disjoint, back_circulant,␣
↪isotopism
sage: is_disjoint(back_circulant(2), back_circulant(2))
False
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import is_disjoint, back_circulant,␣
↪isotopism
>>> is_disjoint(back_circulant(Integer(2)), back_circulant(Integer(2)))
False
```

```
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_disjoint(T1, T2)
True
```

```
>>> from sage.all import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
>>> T2 = T1.apply_isotopism(x, y, z)
>>> is_disjoint(T1, T2)
True
```

sage.combinat.matrices.latin.**is_primary_bitrade**(*a*, *b*, *c*, *G*)

A bitrade generated from elements a, b, c is primary if a, b, c = G.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = p3_group_bitrade_generators(5)
sage: is_primary_bitrade(a, b, c, G)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> (a, b, c, G) = p3_group_bitrade_generators(Integer(5))
>>> is_primary_bitrade(a, b, c, G)
True
```

sage.combinat.matrices.latin.**is_row_and_col_balanced**(*T1*, *T2*)

Partial latin squares T1 and T2 are balanced if the symbols appearing in row r of T1 are the same as the symbols appearing in row r of T2, for each r, and if the same condition holds on columns.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = matrix([[0,1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: T2 = matrix([[0,1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: is_row_and_col_balanced(T1, T2)
True
sage: T2 = matrix([[0,3,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: is_row_and_col_balanced(T1, T2)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = matrix([[Integer(0),Integer(1),-Integer(1),-Integer(1)], [-Integer(1),-
↪Integer(1),-Integer(1),-Integer(1)], [-Integer(1),-Integer(1),-Integer(1),-
↪Integer(1)], [-Integer(1),-Integer(1),-Integer(1),-Integer(1)]])
>>> T2 = matrix([[Integer(0),Integer(1),-Integer(1),-Integer(1)], [-Integer(1),-
↪Integer(1),-Integer(1),-Integer(1)], [-Integer(1),-Integer(1),-Integer(1),-
↪Integer(1)], [-Integer(1),-Integer(1),-Integer(1),-Integer(1)]])
>>> is_row_and_col_balanced(T1, T2)
True
>>> T2 = matrix([[Integer(0),Integer(3),-Integer(1),-Integer(1)], [-Integer(1),-
↪Integer(1),-Integer(1),-Integer(1)], [-Integer(1),-Integer(1),-Integer(1),-
↪Integer(1)], [-Integer(1),-Integer(1),-Integer(1),-Integer(1)]])
>>> is_row_and_col_balanced(T1, T2)
False
```

sage.combinat.matrices.latin.**is_same_shape**(*T1*, *T2*)

Two partial latin squares T1, T2 have the same shape if T1[r, c] = 0 if and only if T2[r, c] = 0.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: is_same_shape(elementary_abelian_2group(2), back_circulant(4))
```

(continues on next page)

(continued from previous page)

```
True
sage: is_same_shape(LatinSquare(5), LatinSquare(5))
True
sage: is_same_shape(forward_circulant(5), LatinSquare(5))
False
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> is_same_shape(elementary_abelian_2group(Integer(2)), back_
↪circulant(Integer(4)))
True
>>> is_same_shape(LatinSquare(Integer(5)), LatinSquare(Integer(5)))
True
>>> is_same_shape(forward_circulant(Integer(5)), LatinSquare(Integer(5)))
False
```

sage.combinat.matrices.latin.**isotopism**(*p*)

Return a Permutation object that represents an isotopism (for rows, columns or symbols of a partial latin square).

Technically, all this function does is take as input a representation of a permutation of $0, ..., n-1$ and return a `Permutation` object defined on $1, ..., n$.

For a definition of isotopism, see the wikipedia section on isotopism.

INPUT:

According to the type of input (see examples below):

- an integer n – the function returns the identity on $1, ..., n$
- a string representing a permutation in disjoint cycles notation, e.g. $(0, 1, 2)(3, 4, 5)$ – the corresponding permutation is returned, shifted by 1 to act on $1, ..., n$
- list/tuple of tuples – assumes disjoint cycle notation, see previous entry
- a list of integers – the function adds 1 to each member of the list, and returns the corresponding permutation
- a `PermutationGroupElement` `p` – returns a permutation describing `p` **without** any shift

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: isotopism(5) # identity on 5 points
[1, 2, 3, 4, 5]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> isotopism(Integer(5)) # identity on 5 points
[1, 2, 3, 4, 5]
```

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: isotopism(g)
[2, 3, 1, 5, 4]
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> g = G.gen(Integer(0))
>>> isotopism(g)
[2, 3, 1, 5, 4]
```

```
sage: isotopism([0,3,2,1]) # 0 goes to 0, 1 goes to 3, etc.
[1, 4, 3, 2]
```

```
>>> from sage.all import *
>>> isotopism([Integer(0),Integer(3),Integer(2),Integer(1)]) # 0 goes to 0, 1␣
↪goes to 3, etc.
[1, 4, 3, 2]
```

```
sage: isotopism( (0,1,2) ) # single cycle, presented as a tuple
[2, 3, 1]
```

```
>>> from sage.all import *
>>> isotopism( (Integer(0),Integer(1),Integer(2)) ) # single cycle, presented as␣
↪a tuple
[2, 3, 1]
```

```
sage: x = isotopism( ((0,1,2), (3,4)) ) # tuple of cycles
sage: x
[2, 3, 1, 5, 4]
sage: x.to_cycles()
[(1, 2, 3), (4, 5)]
```

```
>>> from sage.all import *
>>> x = isotopism( ((Integer(0),Integer(1),Integer(2)), (Integer(3),Integer(4)))␣
↪) # tuple of cycles
>>> x
[2, 3, 1, 5, 4]
>>> x.to_cycles()
[(1, 2, 3), (4, 5)]
```

sage.combinat.matrices.latin.**next_conjugate**(*L*)

Permute L[r, c] = e to the conjugate L[c, e] = r.

We assume that L is an n by n matrix and has values in the range 0, 1, …, n-1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(6)
sage: L
[0 1 2 3 4 5]
[1 2 3 4 5 0]
[2 3 4 5 0 1]
[3 4 5 0 1 2]
[4 5 0 1 2 3]
[5 0 1 2 3 4]
```

(continues on next page)

(continued from previous page)

```
sage: next_conjugate(L)
[0 1 2 3 4 5]
[5 0 1 2 3 4]
[4 5 0 1 2 3]
[3 4 5 0 1 2]
[2 3 4 5 0 1]
[1 2 3 4 5 0]
sage: L == next_conjugate(next_conjugate(next_conjugate(L)))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> L = back_circulant(Integer(6))
>>> L
[0 1 2 3 4 5]
[1 2 3 4 5 0]
[2 3 4 5 0 1]
[3 4 5 0 1 2]
[4 5 0 1 2 3]
[5 0 1 2 3 4]
>>> next_conjugate(L)
[0 1 2 3 4 5]
[5 0 1 2 3 4]
[4 5 0 1 2 3]
[3 4 5 0 1 2]
[2 3 4 5 0 1]
[1 2 3 4 5 0]
>>> L == next_conjugate(next_conjugate(next_conjugate(L)))
True
```

sage.combinat.matrices.latin.**p3_group_bitrade_generators**(*p*)

Generators for a group of order p3 where p is a prime.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: p3_group_bitrade_generators(3)   # random output
((2,6,7)(3,8,9),
 (1,2,3)(4,7,8)(5,6,9),
 (1,9,2)(3,7,4)(5,8,6),
 Permutation Group with generators [(2,6,7)(3,8,9), (1,2,3)(4,7,8)(5,6,9)])
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> p3_group_bitrade_generators(Integer(3))   # random output
((2,6,7)(3,8,9),
 (1,2,3)(4,7,8)(5,6,9),
 (1,9,2)(3,7,4)(5,8,6),
 Permutation Group with generators [(2,6,7)(3,8,9), (1,2,3)(4,7,8)(5,6,9)])
```

sage.combinat.matrices.latin.**pq_group_bitrade_generators**(*p*, *q*)

Generators for a group of order pq where p and q are primes such that (q % p) == 1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: pq_group_bitrade_generators(3,7)
((2,3,5)(4,7,6), (1,2,3,4,5,6,7), (1,4,2)(3,5,6), Permutation Group with␣
→generators [(2,3,5)(4,7,6), (1,2,3,4,5,6,7)])
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> pq_group_bitrade_generators(Integer(3),Integer(7))
((2,3,5)(4,7,6), (1,2,3,4,5,6,7), (1,4,2)(3,5,6), Permutation Group with␣
→generators [(2,3,5)(4,7,6), (1,2,3,4,5,6,7)])
```

`sage.combinat.matrices.latin.`**`row_containing_sym`**`(`*L, c, x*`)`

Given an improper latin square L with L[r1, c] = L[r2, c] = x, return r1 or r2 with equal probability. This is an internal function and should only be used in LatinSquare_generator().

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = matrix([(0, 1, 0, 3), (3, 0, 2, 1), (1, 0, 3, 2), (2, 3, 1, 0)])
sage: L
[0 1 0 3]
[3 0 2 1]
[1 0 3 2]
[2 3 1 0]
sage: c = row_containing_sym(L, 1, 0)
sage: c == 1 or c == 2
True
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> L = matrix([(Integer(0), Integer(1), Integer(0), Integer(3)), (Integer(3),␣
→Integer(0), Integer(2), Integer(1)), (Integer(1), Integer(0), Integer(3),␣
→Integer(2)), (Integer(2), Integer(3), Integer(1), Integer(0))])
>>> L
[0 1 0 3]
[3 0 2 1]
[1 0 3 2]
[2 3 1 0]
>>> c = row_containing_sym(L, Integer(1), Integer(0))
>>> c == Integer(1) or c == Integer(2)
True
```

`sage.combinat.matrices.latin.`**`tau1`**`(`*T1, T2, cells_map*`)`

The definition of τ_1 is

$$\tau_1 : T1 \to T1$$
$$\tau_1 = \beta_2^{-1}\beta_3$$

where the composition is left to right and $\beta_i : T2 \to T1$ changes just the i-th coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t1 = tau1(T1, T2, cells_map)
sage: t1
[2, 3, 4, 5, 1, 7, 8, 9, 10, 6, 12, 13, 14, 15, 11, 17, 18, 19, 20, 16, 22, 23,␣
↪24, 25, 21]
sage: t1.to_cycles()
[(1, 2, 3, 4, 5), (6, 7, 8, 9, 10), (11, 12, 13, 14, 15), (16, 17, 18, 19, 20),␣
↪(21, 22, 23, 24, 25)]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
>>> T2 = T1.apply_isotopism(x, y, z)
>>> is_bitrade(T1, T2)
True
>>> (cells_map, t1, t2, t3) = tau123(T1, T2)
>>> t1 = tau1(T1, T2, cells_map)
>>> t1
[2, 3, 4, 5, 1, 7, 8, 9, 10, 6, 12, 13, 14, 15, 11, 17, 18, 19, 20, 16, 22, 23,␣
↪24, 25, 21]
>>> t1.to_cycles()
[(1, 2, 3, 4, 5), (6, 7, 8, 9, 10), (11, 12, 13, 14, 15), (16, 17, 18, 19, 20),␣
↪(21, 22, 23, 24, 25)]
```

sage.combinat.matrices.latin.**tau123**(*T1*, *T2*)

Compute the tau_i representation for a bitrade (T1, T2).

See the functions tau1, tau2, and tau3 for the mathematical definitions.

OUTPUT:

- (cells_map, t1, t2, t3)

where cells_map is a map to/from the filled cells of T1, and t1, t2, t3 are the tau1, tau2, tau3 permutations.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = pq_group_bitrade_generators(3, 7)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0  1  3 -1 -1 -1 -1]
[ 1  2  4 -1 -1 -1 -1]
[ 2  3  5 -1 -1 -1 -1]
```

(continues on next page)

(continued from previous page)

```
[ 3  4  6 -1 -1 -1 -1]
[ 4  5  0 -1 -1 -1 -1]
[ 5  6  1 -1 -1 -1 -1]
[ 6  0  2 -1 -1 -1 -1]
sage: T2
[ 1  3  0 -1 -1 -1 -1]
[ 2  4  1 -1 -1 -1 -1]
[ 3  5  2 -1 -1 -1 -1]
[ 4  6  3 -1 -1 -1 -1]
[ 5  0  4 -1 -1 -1 -1]
[ 6  1  5 -1 -1 -1 -1]
[ 0  2  6 -1 -1 -1 -1]
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: D = cells_map
sage: {i: v for i,v in D.items() if i in ZZ}
{1: (0, 0),
 2: (0, 1),
 3: (0, 2),
 4: (1, 0),
 5: (1, 1),
 6: (1, 2),
 7: (2, 0),
 8: (2, 1),
 9: (2, 2),
 10: (3, 0),
 11: (3, 1),
 12: (3, 2),
 13: (4, 0),
 14: (4, 1),
 15: (4, 2),
 16: (5, 0),
 17: (5, 1),
 18: (5, 2),
 19: (6, 0),
 20: (6, 1),
 21: (6, 2)}
sage: {i: v for i,v in D.items() if i not in ZZ}
{(0, 0): 1,
 (0, 1): 2,
 (0, 2): 3,
 (1, 0): 4,
 (1, 1): 5,
 (1, 2): 6,
 (2, 0): 7,
 (2, 1): 8,
 (2, 2): 9,
 (3, 0): 10,
 (3, 1): 11,
 (3, 2): 12,
 (4, 0): 13,
 (4, 1): 14,
 (4, 2): 15,
```

(continues on next page)

(continued from previous page)

```
 (5, 0): 16,
 (5, 1): 17,
 (5, 2): 18,
 (6, 0): 19,
 (6, 1): 20,
 (6, 2): 21}
sage: cells_map_as_square(cells_map, max(T1.nrows(), T1.ncols()))
[ 1  2  3 -1 -1 -1 -1]
[ 4  5  6 -1 -1 -1 -1]
[ 7  8  9 -1 -1 -1 -1]
[10 11 12 -1 -1 -1 -1]
[13 14 15 -1 -1 -1 -1]
[16 17 18 -1 -1 -1 -1]
[19 20 21 -1 -1 -1 -1]
sage: t1
[3, 1, 2, 6, 4, 5, 9, 7, 8, 12, 10, 11, 15, 13, 14, 18, 16, 17, 21, 19, 20]
sage: t2
[4, 8, 15, 7, 11, 18, 10, 14, 21, 13, 17, 3, 16, 20, 6, 19, 2, 9, 1, 5, 12]
sage: t3
[20, 18, 10, 2, 21, 13, 5, 3, 16, 8, 6, 19, 11, 9, 1, 14, 12, 4, 17, 15, 7]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> (a, b, c, G) = pq_group_bitrade_generators(Integer(3), Integer(7))
>>> (T1, T2) = bitrade_from_group(a, b, c, G)
>>> T1
[ 0  1  3 -1 -1 -1 -1]
[ 1  2  4 -1 -1 -1 -1]
[ 2  3  5 -1 -1 -1 -1]
[ 3  4  6 -1 -1 -1 -1]
[ 4  5  0 -1 -1 -1 -1]
[ 5  6  1 -1 -1 -1 -1]
[ 6  0  2 -1 -1 -1 -1]
>>> T2
[ 1  3  0 -1 -1 -1 -1]
[ 2  4  1 -1 -1 -1 -1]
[ 3  5  2 -1 -1 -1 -1]
[ 4  6  3 -1 -1 -1 -1]
[ 5  0  4 -1 -1 -1 -1]
[ 6  1  5 -1 -1 -1 -1]
[ 0  2  6 -1 -1 -1 -1]
>>> (cells_map, t1, t2, t3) = tau123(T1, T2)
>>> D = cells_map
>>> {i: v for i,v in D.items() if i in ZZ}
{1: (0, 0),
 2: (0, 1),
 3: (0, 2),
 4: (1, 0),
 5: (1, 1),
 6: (1, 2),
 7: (2, 0),
 8: (2, 1),
```

(continues on next page)

(continued from previous page)

```
 9: (2, 2),
 10: (3, 0),
 11: (3, 1),
 12: (3, 2),
 13: (4, 0),
 14: (4, 1),
 15: (4, 2),
 16: (5, 0),
 17: (5, 1),
 18: (5, 2),
 19: (6, 0),
 20: (6, 1),
 21: (6, 2)}
>>> {i: v for i,v in D.items() if i not in ZZ}
{(0, 0): 1,
 (0, 1): 2,
 (0, 2): 3,
 (1, 0): 4,
 (1, 1): 5,
 (1, 2): 6,
 (2, 0): 7,
 (2, 1): 8,
 (2, 2): 9,
 (3, 0): 10,
 (3, 1): 11,
 (3, 2): 12,
 (4, 0): 13,
 (4, 1): 14,
 (4, 2): 15,
 (5, 0): 16,
 (5, 1): 17,
 (5, 2): 18,
 (6, 0): 19,
 (6, 1): 20,
 (6, 2): 21}
>>> cells_map_as_square(cells_map, max(T1.nrows(), T1.ncols()))
[ 1  2  3 -1 -1 -1 -1]
[ 4  5  6 -1 -1 -1 -1]
[ 7  8  9 -1 -1 -1 -1]
[10 11 12 -1 -1 -1 -1]
[13 14 15 -1 -1 -1 -1]
[16 17 18 -1 -1 -1 -1]
[19 20 21 -1 -1 -1 -1]
>>> t1
[3, 1, 2, 6, 4, 5, 9, 7, 8, 12, 10, 11, 15, 13, 14, 18, 16, 17, 21, 19, 20]
>>> t2
[4, 8, 15, 7, 11, 18, 10, 14, 21, 13, 17, 3, 16, 20, 6, 19, 2, 9, 1, 5, 12]
>>> t3
[20, 18, 10, 2, 21, 13, 5, 3, 16, 8, 6, 19, 11, 9, 1, 14, 12, 4, 17, 15, 7]
```

```
sage: t1.to_cycles()
[(1, 3, 2), (4, 6, 5), (7, 9, 8), (10, 12, 11), (13, 15, 14), (16, 18, 17), (19,
```

(continues on next page)

(continued from previous page)

```
↪21, 20)]
sage: t2.to_cycles()
[(1, 4, 7, 10, 13, 16, 19), (2, 8, 14, 20, 5, 11, 17), (3, 15, 6, 18, 9, 21, 12)]
sage: t3.to_cycles()
[(1, 20, 15), (2, 18, 4), (3, 10, 8), (5, 21, 7), (6, 13, 11), (9, 16, 14), (12,␣
↪19, 17)]
```

```
>>> from sage.all import *
>>> t1.to_cycles()
[(1, 3, 2), (4, 6, 5), (7, 9, 8), (10, 12, 11), (13, 15, 14), (16, 18, 17), (19,␣
↪21, 20)]
>>> t2.to_cycles()
[(1, 4, 7, 10, 13, 16, 19), (2, 8, 14, 20, 5, 11, 17), (3, 15, 6, 18, 9, 21, 12)]
>>> t3.to_cycles()
[(1, 20, 15), (2, 18, 4), (3, 10, 8), (5, 21, 7), (6, 13, 11), (9, 16, 14), (12,␣
↪19, 17)]
```

The product t1*t2*t3 is the identity, i.e. it fixes every point:

```
sage: len((t1*t2*t3).fixed_points()) == T1.nr_filled_cells()
True
```

```
>>> from sage.all import *
>>> len((t1*t2*t3).fixed_points()) == T1.nr_filled_cells()
True
```

sage.combinat.matrices.latin.**tau2**(*T1*, *T2*, *cells_map*)

The definition of τ_2 is

$$\tau_2 : T1 \rightarrow T1$$
$$\tau_2 = \beta_3^{-1}\beta_1$$

where the composition is left to right and $\beta_i : T2 \rightarrow T1$ changes just the i-th coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t2 = tau2(T1, T2, cells_map)
sage: t2
[21, 22, 23, 24, 25, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,␣
↪18, 19, 20]
sage: t2.to_cycles()
[(1, 21, 16, 11, 6), (2, 22, 17, 12, 7), (3, 23, 18, 13, 8), (4, 24, 19, 14, 9),␣
↪(5, 25, 20, 15, 10)]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
>>> T2 = T1.apply_isotopism(x, y, z)
>>> is_bitrade(T1, T2)
True
>>> (cells_map, t1, t2, t3) = tau123(T1, T2)
>>> t2 = tau2(T1, T2, cells_map)
>>> t2
[21, 22, 23, 24, 25, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
↪18, 19, 20]
>>> t2.to_cycles()
[(1, 21, 16, 11, 6), (2, 22, 17, 12, 7), (3, 23, 18, 13, 8), (4, 24, 19, 14, 9),
↪(5, 25, 20, 15, 10)]
```

sage.combinat.matrices.latin.**tau3**(*T1*, *T2*, *cells_map*)

The definition of τ_3 is

$$\tau_3 : T1 \rightarrow T1$$
$$\tau_3 = \beta_1^{-1}\beta_2$$

where the composition is left to right and $\beta_i : T2 \rightarrow T1$ changes just the i-th coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t3 = tau3(T1, T2, cells_map)
sage: t3
[10, 6, 7, 8, 9, 15, 11, 12, 13, 14, 20, 16, 17, 18, 19, 25, 21, 22, 23, 24, 5, 1,
↪ 2, 3, 4]
sage: t3.to_cycles()
[(1, 10, 14, 18, 22), (2, 6, 15, 19, 23), (3, 7, 11, 20, 24), (4, 8, 12, 16, 25),
↪(5, 9, 13, 17, 21)]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
>>> T2 = T1.apply_isotopism(x, y, z)
>>> is_bitrade(T1, T2)
```

(continues on next page)

(continued from previous page)

```
True
>>> (cells_map, t1, t2, t3) = tau123(T1, T2)
>>> t3 = tau3(T1, T2, cells_map)
>>> t3
[10, 6, 7, 8, 9, 15, 11, 12, 13, 14, 20, 16, 17, 18, 19, 25, 21, 22, 23, 24, 5, 1,
↪ 2, 3, 4]
>>> t3.to_cycles()
[(1, 10, 14, 18, 22), (2, 6, 15, 19, 23), (3, 7, 11, 20, 24), (4, 8, 12, 16, 25),␣
↪(5, 9, 13, 17, 21)]
```

sage.combinat.matrices.latin.**tau_to_bitrade**(*t1*, *t2*, *t3*)

Given permutations t1, t2, t3 that represent a latin bitrade, convert them to an explicit latin bitrade (T1, T2). The result is unique up to isotopism.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: _, t1, t2, t3 = tau123(T1, T2)
sage: U1, U2 = tau_to_bitrade(t1, t2, t3)
sage: assert is_bitrade(U1, U2)
sage: U1
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: U2
[4 0 1 2 3]
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.matrices.latin import *
>>> T1 = back_circulant(Integer(5))
>>> x = isotopism( (Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)) )
>>> y = isotopism(Integer(5)) # identity
>>> z = isotopism(Integer(5)) # identity
>>> T2 = T1.apply_isotopism(x, y, z)
>>> _, t1, t2, t3 = tau123(T1, T2)
>>> U1, U2 = tau_to_bitrade(t1, t2, t3)
>>> assert is_bitrade(U1, U2)
>>> U1
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
```

(continues on next page)

(continued from previous page)

```
[3 4 0 1 2]
[4 0 1 2 3]
>>> U2
[4 0 1 2 3]
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
```

5.1.139 Miscellaneous

class sage.combinat.misc.**DoublyLinkedList**(*l*)

Bases: object

A doubly linked list class that provides constant time hiding and unhiding of entries.

Note that this list's indexing is 1-based.

EXAMPLES:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3]); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
sage: dll.hide(1); dll
Doubly linked list of [1, 2, 3]: [2, 3]
sage: dll.unhide(1); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
sage: dll.hide(2); dll
Doubly linked list of [1, 2, 3]: [1, 3]
sage: dll.unhide(2); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
```

```
>>> from sage.all import *
>>> dll = sage.combinat.misc.DoublyLinkedList([Integer(1),Integer(2),Integer(3)]);
↪ dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
>>> dll.hide(Integer(1)); dll
Doubly linked list of [1, 2, 3]: [2, 3]
>>> dll.unhide(Integer(1)); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
>>> dll.hide(Integer(2)); dll
Doubly linked list of [1, 2, 3]: [1, 3]
>>> dll.unhide(Integer(2)); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
```

head()

hide(*i*)

next(*j*)

prev(*j*)

unhide(*i*)

class sage.combinat.misc.**IterableFunctionCall**(*f*, **args*, ***kwargs*)

Bases: object

This class wraps functions with a yield statement (generators) by an object that can be iterated over. For example,

EXAMPLES:

```
sage: def f(): yield 'a'; yield 'b'
```

```
>>> from sage.all import *
>>> def f(): yield 'a'; yield 'b'
```

This does not work:

```
sage: for z in f: print(z)
Traceback (most recent call last):
...
TypeError: 'function' object is not iterable
```

```
>>> from sage.all import *
>>> for z in f: print(z)
Traceback (most recent call last):
...
TypeError: 'function' object is not iterable
```

Use IterableFunctionCall if you want something like the above to work:

```
sage: from sage.combinat.misc import IterableFunctionCall
sage: g = IterableFunctionCall(f)
sage: for z in g: print(z)
a
b
```

```
>>> from sage.all import *
>>> from sage.combinat.misc import IterableFunctionCall
>>> g = IterableFunctionCall(f)
>>> for z in g: print(z)
a
b
```

If your function takes arguments, just put them after the function name. You needn't enclose them in a tuple or anything, just put them there:

```
sage: def f(n, m): yield 'a' * n; yield 'b' * m; yield 'foo'
sage: g = IterableFunctionCall(f, 2, 3)
sage: for z in g: print(z)
aa
bbb
foo
```

```
>>> from sage.all import *
>>> def f(n, m): yield 'a' * n; yield 'b' * m; yield 'foo'
>>> g = IterableFunctionCall(f, Integer(2), Integer(3))
>>> for z in g: print(z)
```

(continues on next page)

(continued from previous page)

```
aa
bbb
foo
```

sage.combinat.misc.**check_integer_list_constraints**(*l*, ***kwargs*)

EXAMPLES:

```
sage: from sage.combinat.misc import check_integer_list_constraints
sage: cilc = check_integer_list_constraints
sage: l = [[2,1,3],[1,2],[3,3],[4,1,1]]
sage: cilc(l, min_part=2)
[[3, 3]]
sage: cilc(l, max_part=2)
[[1, 2]]
sage: cilc(l, length=2)
[[1, 2], [3, 3]]
sage: cilc(l, max_length=2)
[[1, 2], [3, 3]]
sage: cilc(l, min_length=3)
[[2, 1, 3], [4, 1, 1]]
sage: cilc(l, max_slope=0)
[[3, 3], [4, 1, 1]]
sage: cilc(l, min_slope=1)
[[1, 2]]
sage: cilc(l, outer=[2,2])
[[1, 2]]
sage: cilc(l, inner=[2,2])
[[3, 3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.misc import check_integer_list_constraints
>>> cilc = check_integer_list_constraints
>>> l = [[Integer(2),Integer(1),Integer(3)],[Integer(1),Integer(2)],[Integer(3),
↪Integer(3)],[Integer(4),Integer(1),Integer(1)]]
>>> cilc(l, min_part=Integer(2))
[[3, 3]]
>>> cilc(l, max_part=Integer(2))
[[1, 2]]
>>> cilc(l, length=Integer(2))
[[1, 2], [3, 3]]
>>> cilc(l, max_length=Integer(2))
[[1, 2], [3, 3]]
>>> cilc(l, min_length=Integer(3))
[[2, 1, 3], [4, 1, 1]]
>>> cilc(l, max_slope=Integer(0))
[[3, 3], [4, 1, 1]]
>>> cilc(l, min_slope=Integer(1))
[[1, 2]]
>>> cilc(l, outer=[Integer(2),Integer(2)])
[[1, 2]]
>>> cilc(l, inner=[Integer(2),Integer(2)])
[[3, 3]]
```

```
sage: cilc([1,2,3], length=3, singleton=True)
[1, 2, 3]
sage: cilc([1,2,3], length=2, singleton=True) is None
True
```

```
>>> from sage.all import *
>>> cilc([Integer(1),Integer(2),Integer(3)], length=Integer(3), singleton=True)
[1, 2, 3]
>>> cilc([Integer(1),Integer(2),Integer(3)], length=Integer(2), singleton=True)␣
↪is None
True
```

sage.combinat.misc.**umbral_operation**(*poly*)

Return the umbral operation $\downarrow$ applied to poly.

The umbral operation replaces each instance of $x_i^{a_i}$ with $x_i * (x_i - 1) * \cdots * (x_i - a_i + 1)$.

EXAMPLES:

```
sage: P = PolynomialRing(QQ, 2, 'x')
sage: x = P.gens()
sage: from sage.combinat.misc import umbral_operation
sage: umbral_operation(x[0]^3) == x[0]*(x[0]-1)*(x[0]-2)
True
sage: umbral_operation(x[0]*x[1])
x0*x1
sage: umbral_operation(x[0]+x[1])
x0 + x1
sage: umbral_operation(x[0]^2*x[1]^2) == x[0]*(x[0]-1)*x[1]*(x[1]-1)
True
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ, Integer(2), 'x')
>>> x = P.gens()
>>> from sage.combinat.misc import umbral_operation
>>> umbral_operation(x[Integer(0)]**Integer(3)) == x[Integer(0)]*(x[Integer(0)]-
↪Integer(1))*(x[Integer(0)]-Integer(2))
True
>>> umbral_operation(x[Integer(0)]*x[Integer(1)])
x0*x1
>>> umbral_operation(x[Integer(0)]+x[Integer(1)])
x0 + x1
>>> umbral_operation(x[Integer(0)]**Integer(2)*x[Integer(1)]**Integer(2)) ==␣
↪x[Integer(0)]*(x[Integer(0)]-Integer(1))*x[Integer(1)]*(x[Integer(1)]-
↪Integer(1))
True
```

5.1.140 Ordered Multiset Partitions into Sets and the Minimaj Crystal

This module provides element and parent classes for ordered multiset partitions. It also implements the minimaj crystal of Benkart et al. [BCHOPSY2017]. (See *MinimajCrystal*.)

AUTHORS:

- Aaron Lauve (2018): initial implementation. First draft of minimaj crystal code provided by Anne Schilling.

REFERENCES:

- [BCHOPSY2017]
- [HRW2015]
- [HRS2016]
- [LM2018]

EXAMPLES:

An ordered multiset partition into sets of the multiset $\{\{1, 3, 3, 5\}\}$:

```
sage: OrderedMultisetPartitionIntoSets([[5, 3], [1, 3]])
[{3,5}, {1,3}]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionIntoSets([[Integer(5), Integer(3)], [Integer(1),␣
↪Integer(3)]])
[{3,5}, {1,3}]
```

Ordered multiset partitions into sets of the multiset $\{\{1, 3, 3\}\}$:

```
sage: OrderedMultisetPartitionsIntoSets([1,1,3]).list()
[[{1}, {1}, {3}], [{1}, {1,3}], [{1}, {3}, {1}], [{1,3}, {1}], [{3}, {1}, {1}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets([Integer(1),Integer(1),Integer(3)]).list()
[[{1}, {1}, {3}], [{1}, {1,3}], [{1}, {3}, {1}], [{1,3}, {1}], [{3}, {1}, {1}]]
```

Ordered multiset partitions into sets of the integer 4:

```
sage: OrderedMultisetPartitionsIntoSets(4).list()
[[{4}], [{1,3}], [{3}, {1}], [{1,2}, {1}], [{2}, {2}], [{2}, {1}, {1}],
 [{1}, {3}], [{1}, {1,2}], [{1}, {2}, {1}], [{1}, {1}, {2}], [{1}, {1}, {1}, {1}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets(Integer(4)).list()
[[{4}], [{1,3}], [{3}, {1}], [{1,2}, {1}], [{2}, {2}], [{2}, {1}, {1}],
 [{1}, {3}], [{1}, {1,2}], [{1}, {2}, {1}], [{1}, {1}, {2}], [{1}, {1}, {1}, {1}]]
```

Ordered multiset partitions into sets on the alphabet $\{1, 4\}$ of order 3:

```
sage: OrderedMultisetPartitionsIntoSets([1,4], 3).list()
[[{1,4}, {1}], [{1,4}, {4}], [{1}, {1,4}], [{4}, {1,4}], [{1}, {1}, {1}],
 [{1}, {1}, {4}], [{1}, {4}, {1}], [{1}, {4}, {4}], [{4}, {1}, {1}],
 [{4}, {1}, {4}], [{4}, {4}, {1}], [{4}, {4}, {4}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets([Integer(1),Integer(4)], Integer(3)).list()
[[{1,4}, {1}], [{1,4}, {4}], [{1}, {1,4}], [{4}, {1,4}], [{1}, {1}, {1}],
 [{1}, {1}, {4}], [{1}, {4}, {1}], [{1}, {4}, {4}], [{4}, {1}, {1}],
 [{4}, {1}, {4}], [{4}, {4}, {1}], [{4}, {4}, {4}]]
```

Crystal of ordered multiset partitions into sets on the alphabet $\{1, 2, 3\}$ with 4 letters divided into 2 blocks:

```
sage: crystals.Minimaj(3, 4, 2).list()                                        #␣
↪needs sage.modules
[((2, 3, 1), (1,)), ((2, 3), (1, 2)), ((2, 3), (1, 3)), ((2, 1), (1, 2)),
 ((3, 1), (1, 2)), ((3, 1, 2), (2,)), ((3, 1), (1, 3)), ((3, 1), (2, 3)),
 ((3, 2), (2, 3)), ((2, 1), (1, 3)), ((2,), (1, 2, 3)), ((3,), (1, 2, 3)),
 ((1,), (1, 2, 3)), ((1, 2), (2, 3)), ((1, 2, 3), (3,))]
```

```
>>> from sage.all import *
>>> crystals.Minimaj(Integer(3), Integer(4), Integer(2)).list()               ␣
↪                      # needs sage.modules
[((2, 3, 1), (1,)), ((2, 3), (1, 2)), ((2, 3), (1, 3)), ((2, 1), (1, 2)),
 ((3, 1), (1, 2)), ((3, 1, 2), (2,)), ((3, 1), (1, 3)), ((3, 1), (2, 3)),
 ((3, 2), (2, 3)), ((2, 1), (1, 3)), ((2,), (1, 2, 3)), ((3,), (1, 2, 3)),
 ((1,), (1, 2, 3)), ((1, 2), (2, 3)), ((1, 2, 3), (3,))]
```

class sage.combinat.multiset_partition_into_sets_ordered.**MinimajCrystal**(*n*, *ell*, *k*)

Bases: UniqueRepresentation, Parent

Crystal of ordered multiset partitions into sets with ell letters from alphabet $\{1, 2, \ldots, n\}$ divided into k blocks.

Elements are represented in the minimaj ordering of blocks as in Benkart et al. [BCHOPSY2017].

Note

Elements are not stored internally as ordered multiset partitions into sets, but as certain (pairs of) words stemming from the minimaj bijection ϕ of [BCHOPSY2017]. See *sage.combinat.multiset_partition_into_sets_ordered.MinimajCrystal.Element* for further details.

AUTHORS:

- Anne Schilling (2018): initial draft
- Aaron Lauve (2018): changed to use `Letters` crystal for elements

EXAMPLES:

```
sage: list(crystals.Minimaj(2,3,2))                                           #␣
↪needs sage.modules
[((2, 1), (1,)), ((2,), (1, 2)), ((1,), (1, 2)), ((1, 2), (2,))]

sage: b = crystals.Minimaj(3, 5, 2).an_element(); b                           #␣
↪needs sage.modules
((2, 3, 1), (1, 2))
sage: b.f(2)                                                                  #␣
↪needs sage.modules
((2, 3, 1), (1, 3))
sage: b.e(2)                                                                  #␣
↪needs sage.modules
```

```
>>> from sage.all import *
>>> list(crystals.Minimaj(Integer(2),Integer(3),Integer(2)))                  ␣
↪                      # needs sage.modules
```

(continues on next page)

(continued from previous page)

```
[((2, 1), (1,)), ((2,), (1, 2)), ((1,), (1, 2)), ((1, 2), (2,))]

>>> b = crystals.Minimaj(Integer(3), Integer(5), Integer(2)).an_element(); b
↪                        # needs sage.modules
((2, 3, 1), (1, 2))
>>> b.f(Integer(2))
↪     # needs sage.modules
((2, 3, 1), (1, 3))
>>> b.e(Integer(2))
↪     # needs sage.modules
```

class Element

Bases: `ElementWrapper`

An element of a Minimaj crystal.

> **Note**
>
> Minimaj elements b are stored internally as pairs `(w, breaks)`, where:
>
> - `w` – a word of length `self.parent().ell` over the letters 1 up to `self.parent().n`;
> - `breaks` is a list of de-concatenation points to turn `w` into a list of row words of (skew-)tableaux that represent b under the minimaj bijection ϕ of [BCHOPSY2017].
>
> The pair `(w, breaks)` may be recovered via `b.value`.

e(*i*)

Return e_i on `self`.

EXAMPLES:

```
sage: B = crystals.Minimaj(4,3,2)                                         #
↪needs sage.modules
sage: b = B([[2,3], [3]]); b                                              #
↪needs sage.modules
((2, 3), (3,))
sage: [b.e(i) for i in range(1,4)]                                        #
↪needs sage.modules
[((1, 3), (3,)), ((2,), (2, 3)), None]
```

```
>>> from sage.all import *
>>> B = crystals.Minimaj(Integer(4),Integer(3),Integer(2))
↪                        # needs sage.modules
>>> b = B([[Integer(2),Integer(3)], [Integer(3)]]); b
↪                        # needs sage.modules
((2, 3), (3,))
>>> [b.e(i) for i in range(Integer(1),Integer(4))]
↪                   # needs sage.modules
[((1, 3), (3,)), ((2,), (2, 3)), None]
```

f(*i*)

Return f_i on `self`.

EXAMPLES:

```
sage: B = crystals.Minimaj(4,3,2)                                          #␣
↪needs sage.modules
sage: b = B([[2,3], [3]]); b                                               #␣
↪needs sage.modules
((2, 3), (3,))
sage: [b.f(i) for i in range(1,4)]                                         #␣
↪needs sage.modules
[None, None, ((2, 3), (4,))]
```

```
>>> from sage.all import *
>>> B = crystals.Minimaj(Integer(4),Integer(3),Integer(2))                  ␣
↪                        # needs sage.modules
>>> b = B([[Integer(2),Integer(3)], [Integer(3)]]); b                       ␣
↪                        # needs sage.modules
((2, 3), (3,))
>>> [b.f(i) for i in range(Integer(1),Integer(4))]                          ␣
↪              # needs sage.modules
[None, None, ((2, 3), (4,))]
```

to_tableaux_words()

Return the image of the ordered multiset partition into sets `self` under the minimaj bijection ϕ of [BCHOPSY2017].

EXAMPLES:

```
sage: # needs sage.modules
sage: B = crystals.Minimaj(4,5,3)
sage: b = B.an_element(); b
((2, 3, 1), (1,), (1,))
sage: b.to_tableaux_words()
[[1], [3], [2, 1, 1]]
sage: b = B([[1,3,4], [3], [3]]); b
((4, 1, 3), (3,), (3,))
sage: b.to_tableaux_words()
[[3, 1], [], [4, 3, 3]]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> B = crystals.Minimaj(Integer(4),Integer(5),Integer(3))
>>> b = B.an_element(); b
((2, 3, 1), (1,), (1,))
>>> b.to_tableaux_words()
[[1], [3], [2, 1, 1]]
>>> b = B([[Integer(1),Integer(3),Integer(4)], [Integer(3)],␣
↪[Integer(3)]]); b
((4, 1, 3), (3,), (3,))
>>> b.to_tableaux_words()
[[3, 1], [], [4, 3, 3]]
```

from_tableau(*t*)

Return the bijection ϕ^{-1} of [BCHOPSY2017] applied to `t`.

INPUT:

- `t` – a sequence of column tableaux and a ribbon tableau

EXAMPLES:

```
sage: # needs sage.modules
sage: B = crystals.Minimaj(3,6,3)
sage: b = B.an_element(); b
((3, 1, 2), (2, 1), (1,))
sage: t = b.to_tableaux_words(); t
[[1], [2, 1], [], [3, 2, 1]]
sage: B.from_tableau(t)
((3, 1, 2), (2, 1), (1,))
sage: B.from_tableau(t) == b
True
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> B = crystals.Minimaj(Integer(3),Integer(6),Integer(3))
>>> b = B.an_element(); b
((3, 1, 2), (2, 1), (1,))
>>> t = b.to_tableaux_words(); t
[[1], [2, 1], [], [3, 2, 1]]
>>> B.from_tableau(t)
((3, 1, 2), (2, 1), (1,))
>>> B.from_tableau(t) == b
True
```

val(*q='q'*)

Return the Val polynomial corresponding to `self`.

EXAMPLES:

Verifying Example 4.5 from [BCHOPSY2017]:

```
sage: B = crystals.Minimaj(3, 4, 2)  # for `Val_{4,1}^{(3)}`                   #␣
↪needs sage.modules
sage: B.val()                                                                   #␣
↪needs sage.modules
(q^2+q+1)*s[2, 1, 1] + q*s[2, 2]
```

```
>>> from sage.all import *
>>> B = crystals.Minimaj(Integer(3), Integer(4), Integer(2))  # for `Val_{4,1}
↪^{(3)}`                    # needs sage.modules
>>> B.val()                                                                   #␣
↪needs sage.modules
(q^2+q+1)*s[2, 1, 1] + q*s[2, 2]
```

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionIntoSets**(*parent*, *data*)

Bases: `ClonableArray`

Ordered Multiset Partition into sets.

An *ordered multiset partition into sets* c of a multiset X is a list $[c_1, \ldots, c_r]$ of nonempty subsets of X (note: not sub-multisets), called the *blocks* of c, whose multi-union is X.

EXAMPLES:

The simplest way to create an ordered multiset partition into sets is by specifying its blocks as a list or tuple:

```
sage: OrderedMultisetPartitionIntoSets([[3],[2,1]])
[{3}, {1,2}]
sage: OrderedMultisetPartitionIntoSets(((3,), (1,2)))
[{3}, {1,2}]
sage: OrderedMultisetPartitionIntoSets([set([i]) for i in range(2,5)])
[{2}, {3}, {4}]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionIntoSets([[Integer(3)],[Integer(2),Integer(1)]])
[{3}, {1,2}]
>>> OrderedMultisetPartitionIntoSets(((Integer(3),), (Integer(1),Integer(2))))
[{3}, {1,2}]
>>> OrderedMultisetPartitionIntoSets([set([i]) for i in range(Integer(2),
↪Integer(5))])
[{2}, {3}, {4}]
```

REFERENCES:

- [HRW2015]
- [HRS2016]
- [LM2018]

check()

Check that we are a valid ordered multiset partition into sets.

EXAMPLES:

```
sage: c = OrderedMultisetPartitionsIntoSets(4)([[1], [1,2]])
sage: c.check()

sage: OMPs = OrderedMultisetPartitionsIntoSets()
sage: c = OMPs([[1], [1], ['a']])
sage: c.check()
```

```
>>> from sage.all import *
>>> c = OrderedMultisetPartitionsIntoSets(Integer(4))([[Integer(1)],
↪[Integer(1),Integer(2)]])
>>> c.check()

>>> OMPs = OrderedMultisetPartitionsIntoSets()
>>> c = OMPs([[Integer(1)], [Integer(1)], ['a']])
>>> c.check()
```

deconcatenate(*k=2*)

Return the list of k-deconcatenations of `self`.

A k-tuple $(C_1, \ldots, C_k)$ of ordered multiset partitions into sets represents a k-deconcatenation of an ordered multiset partition into sets C if $C_1 + \cdots + C_k = C$.

Note

This is not to be confused with `self.split_blocks()`, which splits each block of `self` before making k-tuples of ordered multiset partitions into sets.

EXAMPLES:

```
sage: OrderedMultisetPartitionIntoSets([[7,1],[3,4,5]]).deconcatenate()
[([{1,7}, {3,4,5}], []), ([{1,7}], [{3,4,5}]), ([], [{1,7}, {3,4,5}])]
sage: OrderedMultisetPartitionIntoSets([['b','c'],['a']]).deconcatenate()
[([{'b','c'}, {'a'}], []), ([{'b','c'}], [{'a'}]), ([], [{'b','c'}, {'a'}])]
sage: OrderedMultisetPartitionIntoSets([['a','b','c']]).deconcatenate(3)
[([{'a','b','c'}], [], []),
 ([], [{'a','b','c'}], []),
 ([], [], [{'a','b','c'}])]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionIntoSets([[Integer(7),Integer(1)],[Integer(3),
↪Integer(4),Integer(5)]]).deconcatenate()
[([{1,7}, {3,4,5}], []), ([{1,7}], [{3,4,5}]), ([], [{1,7}, {3,4,5}])]
>>> OrderedMultisetPartitionIntoSets([['b','c'],['a']]).deconcatenate()
[([{'b','c'}, {'a'}], []), ([{'b','c'}], [{'a'}]), ([], [{'b','c'}, {'a'}])]
>>> OrderedMultisetPartitionIntoSets([['a','b','c']]).
↪deconcatenate(Integer(3))
[([{'a','b','c'}], [], []),
 ([], [{'a','b','c'}], []),
 ([], [], [{'a','b','c'}])]
```

fatten(*grouping*)

Return the ordered multiset partition into sets fatter than `self`, obtained by grouping together consecutive parts according to `grouping` (whenever this does not violate the strictness condition).

INPUT:

- `grouping` – a composition (or list) whose sum is the length of `self`

EXAMPLES:

Let us start with the composition:

```
sage: C = OrderedMultisetPartitionIntoSets([[4,1,5], [2], [7,1]]); C
[{1,4,5}, {2}, {1,7}]
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(4),Integer(1),Integer(5)],␣
↪[Integer(2)], [Integer(7),Integer(1)]]); C
[{1,4,5}, {2}, {1,7}]
```

With `grouping` equal to $(1,1,1)$, C is left unchanged:

```
sage: C.fatten([1,1,1])
[{1,4,5}, {2}, {1,7}]
```

```
>>> from sage.all import *
>>> C.fatten([Integer(1),Integer(1),Integer(1)])
[{1,4,5}, {2}, {1,7}]
```

With grouping equal to $(2, 1)$ or $(1, 2)$, a union of consecutive parts is achieved:

```
sage: C.fatten([2,1])
[{1,2,4,5}, {1,7}]
sage: C.fatten([1,2])
[{1,4,5}, {1,2,7}]
```

```
>>> from sage.all import *
>>> C.fatten([Integer(2),Integer(1)])
[{1,2,4,5}, {1,7}]
>>> C.fatten([Integer(1),Integer(2)])
[{1,4,5}, {1,2,7}]
```

However, the grouping (3) will throw an error, as 1 cannot appear twice in any block of C:

```
sage: C.fatten(Composition([3]))
Traceback (most recent call last):
...
ValueError: [{1,4,5,2,1,7}] is not a valid ordered multiset partition into
↪sets
```

```
>>> from sage.all import *
>>> C.fatten(Composition([Integer(3)]))
Traceback (most recent call last):
...
ValueError: [{1,4,5,2,1,7}] is not a valid ordered multiset partition into
↪sets
```

fatter()

Return the set of ordered multiset partitions into sets which are fatter than self.

An ordered multiset partition into sets A is fatter than another B if, reading left-to-right, every block of A is the union of some consecutive blocks of B.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([{1,4,5}, {2}, {1,7}]).fatter()
sage: len(C)
3
sage: sorted(C)
[[{1,4,5}, {2}, {1,7}], [{1,4,5}, {1,2,7}], [{1,2,4,5}, {1,7}]]
sage: sorted(OrderedMultisetPartitionIntoSets([['a','b'],['c'],['a']]).
↪fatter())
[[{'a','b'}, {'c'}, {'a'}], [{'a','b'}, {'a','c'}], [{'a','b','c'}, {'a'}]]
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([{Integer(1),Integer(4),Integer(5)},
↪{Integer(2)}, {Integer(1),Integer(7)}]).fatter()
>>> len(C)
```

(continues on next page)

(continued from previous page)

```
3
>>> sorted(C)
[[{1,4,5}, {2}, {1,7}], [{1,4,5}, {1,2,7}], [{1,2,4,5}, {1,7}]]
>>> sorted(OrderedMultisetPartitionIntoSets([['a','b'],['c'],['a']]).fatter())
[[{'a','b'}, {'c'}, {'a'}], [{'a','b'}, {'a','c'}], [{'a','b','c'}, {'a'}]]
```

Some extreme cases:

```
sage: list(OrderedMultisetPartitionIntoSets([['a','b','c']]).fatter())
[[{'a','b','c'}]]
sage: list(OrderedMultisetPartitionIntoSets([]).fatter())
[[]]
sage: A = OrderedMultisetPartitionIntoSets([[1], [2], [3], [4]])
sage: B = OrderedMultisetPartitionIntoSets([[1,2,3,4]])
sage: A.fatter().issubset(B.finer())
True
```

```
>>> from sage.all import *
>>> list(OrderedMultisetPartitionIntoSets([['a','b','c']]).fatter())
[[{'a','b','c'}]]
>>> list(OrderedMultisetPartitionIntoSets([]).fatter())
[[]]
>>> A = OrderedMultisetPartitionIntoSets([[Integer(1)], [Integer(2)],␣
↪[Integer(3)], [Integer(4)]])
>>> B = OrderedMultisetPartitionIntoSets([[Integer(1),Integer(2),Integer(3),
↪Integer(4)]])
>>> A.fatter().issubset(B.finer())
True
```

finer(*strong=False*)

Return the set of ordered multiset partitions into sets that are finer than `self`.

An ordered multiset partition into sets A is finer than another B if, reading left-to-right, every block of B is the union of some consecutive blocks of A.

If optional argument `strong` is set to `True`, then return only those A whose blocks are deconcatenations of blocks of B. (Here, we view blocks of B as sorted lists instead of sets.)

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[3,2]]).finer()
sage: len(C)
3
sage: sorted(C, key=str)
[[{2,3}], [{2}, {3}], [{3}, {2}]]
sage: OrderedMultisetPartitionIntoSets([]).finer()
{[]}
sage: O = OrderedMultisetPartitionsIntoSets([1, 1, 'a', 'b'])
sage: o = O([{1}, {'a', 'b'}, {1}])
sage: sorted(o.finer(), key=str)
[[{1}, {'a','b'}, {1}], [{1}, {'a'}, {'b'}, {1}], [{1}, {'b'}, {'a'}, {1}]]
sage: o.finer() & o.fatter() == set([o])
True
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(3),Integer(2)]]).finer()
>>> len(C)
3
>>> sorted(C, key=str)
[[{2,3}], [{2}, {3}], [{3}, {2}]]
>>> OrderedMultisetPartitionIntoSets([]).finer()
{[]}
>>> O = OrderedMultisetPartitionsIntoSets([Integer(1), Integer(1), 'a', 'b'])
>>> o = O([{Integer(1)}, {'a', 'b'}, {Integer(1)}])
>>> sorted(o.finer(), key=str)
[[{1}, {'a','b'}, {1}], [{1}, {'a'}, {'b'}, {1}], [{1}, {'b'}, {'a'}, {1}]]
>>> o.finer() & o.fatter() == set([o])
True
```

is_finer(*co*)

Return `True` if the ordered multiset partition into sets `self` is finer than the composition `co`; otherwise, return `False`.

EXAMPLES:

```
sage: OrderedMultisetPartitionIntoSets([[4],[1],[2]]).is_finer([[1,4],[2]])
True
sage: OrderedMultisetPartitionIntoSets([[1],[4],[2]]).is_finer([[1,4],[2]])
True
sage: OrderedMultisetPartitionIntoSets([[1,4],[1],[1]]).is_finer([[1,4],[2]])
False
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionIntoSets([[Integer(4)],[Integer(1)],
↪[Integer(2)]]).is_finer([[Integer(1),Integer(4)],[Integer(2)]])
True
>>> OrderedMultisetPartitionIntoSets([[Integer(1)],[Integer(4)],
↪[Integer(2)]]).is_finer([[Integer(1),Integer(4)],[Integer(2)]])
True
>>> OrderedMultisetPartitionIntoSets([[Integer(1),Integer(4)],[Integer(1)],
↪[Integer(1)]]).is_finer([[Integer(1),Integer(4)],[Integer(2)]])
False
```

length()

Return the number of blocks of `self`.

EXAMPLES:

```
sage: OrderedMultisetPartitionIntoSets([[7,1],[3]]).length()
2
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionIntoSets([[Integer(7),Integer(1)],[Integer(3)]]).
↪length()
2
```

letters()

Return the set of distinct elements occurring within the blocks of `self`.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[3, 4, 1], [2], [1, 2, 3, 7]]); C
[{1,3,4}, {2}, {1,2,3,7}]
sage: C.letters()
frozenset({1, 2, 3, 4, 7})
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(3), Integer(4),␣
→Integer(1)], [Integer(2)], [Integer(1), Integer(2), Integer(3),␣
→Integer(7)]]); C
[{1,3,4}, {2}, {1,2,3,7}]
>>> C.letters()
frozenset({1, 2, 3, 4, 7})
```

major_index()

Return the major index of `self`.

The major index is a statistic on ordered multiset partitions into sets, which we define here via an example.

1. Sort each block in the list `self` in descending order to create a word w, keeping track of the original separation into blocks:

```
in:  [{3,4,5}, {2,3,4}, {1}, {4,5}]
out: [ 5,4,3 /  4,3,2 /  1 /  5,4 ]
```

2. Create a sequence $v = (v_0, v_1, v_2, \ldots)$ of length `self.order()+1`, built recursively by:

   1. $v_0 = 0$
   2. $v_j = v_{j-1} + \delta(j)$, where $\delta(j) = 1$ if j is the index of an end of a block, and zero otherwise.

```
in:    [ 5,4,3 /  4,3,2 /  1 /  5,4]
out: (0, 0,0,1,   1,1,2,   3,   3,4)
```

3. Compute $\sum_j v_j$, restricted to descent positions in w, i.e., sum over those j with $w_j > w_{j+1}$:

```
in:  w:   [5, 4, 3, 4, 3, 2, 1, 5, 4]
     v: (0 0, 0, 1, 1, 1, 2, 3, 3, 4)
maj :=     0 +0    +1 +1 +2    +3     = 7
```

REFERENCES:

- [HRW2015]

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([{1,5,7}, {2,4}, {5,6}, {4,6,8},
→{1,3}, {1,2,3}])
sage: C.major_index()
27
sage: C = OrderedMultisetPartitionIntoSets([{3,4,5}, {2,3,4}, {1}, {4,5}])
sage: C.major_index()
7
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([{Integer(1),Integer(5),Integer(7)},
↪{Integer(2),Integer(4)}, {Integer(5),Integer(6)}, {Integer(4),Integer(6),
↪Integer(8)}, {Integer(1),Integer(3)}, {Integer(1),Integer(2),Integer(3)}])
>>> C.major_index()
27
>>> C = OrderedMultisetPartitionIntoSets([{Integer(3),Integer(4),Integer(5)},
↪{Integer(2),Integer(3),Integer(4)}, {Integer(1)}, {Integer(4),Integer(5)}])
>>> C.major_index()
7
```

max_letter()

Return the maximum letter appearing in `self.letters()` of `self`.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[3, 4, 1], [2], [1, 2, 3, 7]])
sage: C.max_letter()
7
sage: D = OrderedMultisetPartitionIntoSets([['a','b','c'],['a','b'],['a'],['b
↪','c','f'],['c','d']])
sage: D.max_letter()
'f'
sage: C = OrderedMultisetPartitionIntoSets([])
sage: C.max_letter()
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(3), Integer(4),␣
↪Integer(1)], [Integer(2)], [Integer(1), Integer(2), Integer(3),␣
↪Integer(7)]])
>>> C.max_letter()
7
>>> D = OrderedMultisetPartitionIntoSets([['a','b','c'],['a','b'],['a'],['b',
↪'c','f'],['c','d']])
>>> D.max_letter()
'f'
>>> C = OrderedMultisetPartitionIntoSets([])
>>> C.max_letter()
```

minimaj()

Return the minimaj statistic on ordered multiset partitions into sets.

We define *minimaj* via an example:

1. Sort the block in `self` as prescribed by `self.minimaj_word()`, keeping track of the original separation into blocks:

```
in:   [{1,5,7}, {2,4}, {5,6}, {4,6,8}, {1,3}, {1,2,3}]
out:  ( 5,7,1 /  2,4 /  5,6 /  4,6,8 /  3,1 /  1,2,3 )
```

2. Record the indices where descents in this word occur:

```
word:       (5, 7, 1 / 2, 4 / 5, 6 / 4, 6, 8 / 3, 1 / 1, 2, 3)
indices:     1  2  3   4  5   6  7   8  9 10   11 12  13 14 15
```

(continues on next page)

(continued from previous page)

```
descents:  {    2,                 7,        10, 11              }
```

3. Compute the sum of the descents:

```
minimaj = 2 + 7 + 10 + 11 = 30
```

REFERENCES:

- [HRW2015]

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([{1,5,7}, {2,4}, {5,6}, {4,6,8},
↪{1,3}, {1,2,3}])
sage: C, C.minimaj_word()
([{1,5,7}, {2,4}, {5,6}, {4,6,8}, {1,3}, {1,2,3}],
 (5, 7, 1, 2, 4, 5, 6, 4, 6, 8, 3, 1, 1, 2, 3))
sage: C.minimaj()
30
sage: C = OrderedMultisetPartitionIntoSets([{2,4}, {1,2,3}, {1,6,8}, {2,3}])
sage: C, C.minimaj_word()
([{2,4}, {1,2,3}, {1,6,8}, {2,3}], (2, 4, 1, 2, 3, 6, 8, 1, 2, 3))
sage: C.minimaj()
9
sage: OrderedMultisetPartitionIntoSets([]).minimaj()
0
sage: C = OrderedMultisetPartitionIntoSets([['b','d'],['a','b','c'],['b']])
sage: C, C.minimaj_word()
([{'b','d'}, {'a','b','c'}, {'b'}], ('d', 'b', 'c', 'a', 'b', 'b'))
sage: C.minimaj()
4
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([{Integer(1),Integer(5),Integer(7)},
↪{Integer(2),Integer(4)}, {Integer(5),Integer(6)}, {Integer(4),Integer(6),
↪Integer(8)}, {Integer(1),Integer(3)}, {Integer(1),Integer(2),Integer(3)}])
>>> C, C.minimaj_word()
([{1,5,7}, {2,4}, {5,6}, {4,6,8}, {1,3}, {1,2,3}],
 (5, 7, 1, 2, 4, 5, 6, 4, 6, 8, 3, 1, 1, 2, 3))
>>> C.minimaj()
30
>>> C = OrderedMultisetPartitionIntoSets([{Integer(2),Integer(4)},
↪{Integer(1),Integer(2),Integer(3)}, {Integer(1),Integer(6),Integer(8)},
↪{Integer(2),Integer(3)}])
>>> C, C.minimaj_word()
([{2,4}, {1,2,3}, {1,6,8}, {2,3}], (2, 4, 1, 2, 3, 6, 8, 1, 2, 3))
>>> C.minimaj()
9
>>> OrderedMultisetPartitionIntoSets([]).minimaj()
0
>>> C = OrderedMultisetPartitionIntoSets([['b','d'],['a','b','c'],['b']])
>>> C, C.minimaj_word()
([{'b','d'}, {'a','b','c'}, {'b'}], ('d', 'b', 'c', 'a', 'b', 'b'))
```

(continues on next page)

(continued from previous page)

```
>>> C.minimaj()
4
```

minimaj_blocks()

Return the minimaj ordering on blocks of `self`.

We define the ordering via the example below.

Sort the blocks $[B_1, ..., B_k]$ of `self` from right to left via:

1. Sort the last block B_k in increasing order, call it the word W_k
2. If blocks $B_{i+1}, \ldots, B_k$ have been converted to words $W_{i+1}, \ldots, W_k$, use the letters in B_i to make the unique word W_i that has a factorization $W_i = (u, v)$ satisfying:
   - letters of u and v appear in increasing order, with v possibly empty;
   - letters in vu appear in increasing order;
   - `v[-1]` is the largest letter $a \in B_i$ satisfying `a <= W_{i+1}[0]`.

EXAMPLES:

```
sage: OrderedMultisetPartitionIntoSets([[1,5,7], [2,4], [5,6], [4,6,8], [1,3],
↪ [1,2,3]])
[{1,5,7}, {2,4}, {5,6}, {4,6,8}, {1,3}, {1,2,3}]
sage: _.minimaj_blocks()
((5, 7, 1), (2, 4), (5, 6), (4, 6, 8), (3, 1), (1, 2, 3))
sage: OrderedMultisetPartitionIntoSets([]).minimaj_blocks()
()
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionIntoSets([[Integer(1),Integer(5),Integer(7)],
↪[Integer(2),Integer(4)], [Integer(5),Integer(6)], [Integer(4),Integer(6),
↪Integer(8)], [Integer(1),Integer(3)], [Integer(1),Integer(2),Integer(3)]])
[{1,5,7}, {2,4}, {5,6}, {4,6,8}, {1,3}, {1,2,3}]
>>> _.minimaj_blocks()
((5, 7, 1), (2, 4), (5, 6), (4, 6, 8), (3, 1), (1, 2, 3))
>>> OrderedMultisetPartitionIntoSets([]).minimaj_blocks()
()
```

minimaj_word()

Return an ordering of `self._multiset` derived from the minimaj ordering on blocks of `self`.

See also

OrderedMultisetPartitionIntoSets.minimaj_blocks().

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[2,1], [1,2,3], [1,2], [3], [1]]);
↪ C
[{1,2}, {1,2,3}, {1,2}, {3}, {1}]
sage: C.minimaj_blocks()
((1, 2), (2, 3, 1), (1, 2), (3,), (1,))
```

(continues on next page)

(continued from previous page)

```
sage: C.minimaj_word()
(1, 2, 2, 3, 1, 1, 2, 3, 1)
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(2),Integer(1)],
↪[Integer(1),Integer(2),Integer(3)], [Integer(1),Integer(2)], [Integer(3)],
↪[Integer(1)]]); C
[{1,2}, {1,2,3}, {1,2}, {3}, {1}]
>>> C.minimaj_blocks()
((1, 2), (2, 3, 1), (1, 2), (3,), (1,))
>>> C.minimaj_word()
(1, 2, 2, 3, 1, 1, 2, 3, 1)
```

multiset(*as_dict=False*)

Return the multiset corresponding to `self`.

INPUT:

- `as_dict` – boolean (default: `False`); whether to return the multiset as a tuple of a dict of multiplicities

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[3, 4, 1], [2], [1, 2, 3, 7]]); C
[{1,3,4}, {2}, {1,2,3,7}]
sage: C.multiset()
(1, 1, 2, 2, 3, 3, 4, 7)
sage: C.multiset(as_dict=True)
{1: 2, 2: 2, 3: 2, 4: 1, 7: 1}
sage: OrderedMultisetPartitionIntoSets([]).multiset() == ()
True
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(3), Integer(4),
↪Integer(1)], [Integer(2)], [Integer(1), Integer(2), Integer(3),
↪Integer(7)]]); C
[{1,3,4}, {2}, {1,2,3,7}]
>>> C.multiset()
(1, 1, 2, 2, 3, 3, 4, 7)
>>> C.multiset(as_dict=True)
{1: 2, 2: 2, 3: 2, 4: 1, 7: 1}
>>> OrderedMultisetPartitionIntoSets([]).multiset() == ()
True
```

order()

Return the total number of elements in all blocks of `self`.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[3, 4, 1], [2], [1, 2, 3, 7]]); C
[{1,3,4}, {2}, {1,2,3,7}]
sage: C.order()
8
sage: C.order() == sum(C.weight().values())
```

(continues on next page)

(continued from previous page)

```
True
sage: C.order() == sum(k for k in C.shape_from_cardinality())
True
sage: OrderedMultisetPartitionIntoSets([[7,1],[3]]).order()
3
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(3), Integer(4),
↪Integer(1)], [Integer(2)], [Integer(1), Integer(2), Integer(3),
↪Integer(7)]]); C
[{1,3,4}, {2}, {1,2,3,7}]
>>> C.order()
8
>>> C.order() == sum(C.weight().values())
True
>>> C.order() == sum(k for k in C.shape_from_cardinality())
True
>>> OrderedMultisetPartitionIntoSets([[Integer(7),Integer(1)],[Integer(3)]]).
↪order()
3
```

reversal()

Return the reverse ordered multiset partition into sets of `self`.

Given an ordered multiset partition into sets $(B_1, B_2, \ldots, B_k)$, its reversal is defined to be the ordered multiset partition into sets $(B_k, \ldots, B_2, B_1)$.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[1], [1, 3], [2, 3, 4]]); C
[{1}, {1,3}, {2,3,4}]
sage: C.reversal()
[{2,3,4}, {1,3}, {1}]
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(1)], [Integer(1),
↪Integer(3)], [Integer(2), Integer(3), Integer(4)]]); C
[{1}, {1,3}, {2,3,4}]
>>> C.reversal()
[{2,3,4}, {1,3}, {1}]
```

shape_from_cardinality()

Return a composition that records the cardinality of each block of `self`.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[3, 4, 1], [2], [1, 2, 3, 7]]); C
[{1,3,4}, {2}, {1,2,3,7}]
sage: C.shape_from_cardinality()
[3, 1, 4]
sage: OrderedMultisetPartitionIntoSets([]).shape_from_cardinality() ==
↪Composition([])
True
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(3), Integer(4),␣
↪Integer(1)], [Integer(2)], [Integer(1), Integer(2), Integer(3),␣
↪Integer(7)]]); C
[{1,3,4}, {2}, {1,2,3,7}]
>>> C.shape_from_cardinality()
[3, 1, 4]
>>> OrderedMultisetPartitionIntoSets([]).shape_from_cardinality() ==␣
↪Composition([])
True
```

shape_from_size()

Return a composition that records the sum of entries of each block of `self`.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[3, 4, 1], [2], [1, 2, 3, 7]]); C
[{1,3,4}, {2}, {1,2,3,7}]
sage: C.shape_from_size()
[8, 2, 13]
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(3), Integer(4),␣
↪Integer(1)], [Integer(2)], [Integer(1), Integer(2), Integer(3),␣
↪Integer(7)]]); C
[{1,3,4}, {2}, {1,2,3,7}]
>>> C.shape_from_size()
[8, 2, 13]
```

shuffle_product(*other*, *overlap=False*)

Return the shuffles (with multiplicity) of blocks of `self` with blocks of `other`.

In case optional argument `overlap` is `True`, instead return the allowable overlapping shuffles. An overlapping shuffle C is allowable if, whenever one of its blocks c comes from the union $c = a \cup b$ of a block of `self` and a block of `other`, then this union is disjoint.

See also

Composition.shuffle_product()

EXAMPLES:

```
sage: A = OrderedMultisetPartitionIntoSets([[2,1,3], [1,2]]); A
[{1,2,3}, {1,2}]
sage: B = OrderedMultisetPartitionIntoSets([[3,4]]); B
[{3,4}]
sage: C = OrderedMultisetPartitionIntoSets([[4,5]]); C
[{4,5}]
sage: list(A.shuffle_product(B))
[[{1,2,3}, {1,2}, {3,4}], [{3,4}, {1,2,3}, {1,2}], [{1,2,3}, {3,4}, {1,2}]]
sage: list(A.shuffle_product(B, overlap=True))
[[{1,2,3}, {1,2}, {3,4}], [{1,2,3}, {3,4}, {1,2}],
```

(continues on next page)

(continued from previous page)

```
 [{3,4}, {1,2,3}, {1,2}], [{1,2,3}, {1,2,3,4}]]
sage: list(A.shuffle_product(C, overlap=True))
[[{1,2,3}, {1,2}, {4,5}], [{1,2,3}, {4,5}, {1,2}], [{4,5}, {1,2,3}, {1,2}],
 [{1,2,3,4,5}, {1,2}], [{1,2,3}, {1,2,4,5}]]
```

```
>>> from sage.all import *
>>> A = OrderedMultisetPartitionIntoSets([[Integer(2),Integer(1),Integer(3)],␣
→[Integer(1),Integer(2)]]); A
[{1,2,3}, {1,2}]
>>> B = OrderedMultisetPartitionIntoSets([[Integer(3),Integer(4)]]); B
[{3,4}]
>>> C = OrderedMultisetPartitionIntoSets([[Integer(4),Integer(5)]]); C
[{4,5}]
>>> list(A.shuffle_product(B))
[[{1,2,3}, {1,2}, {3,4}], [{3,4}, {1,2,3}, {1,2}], [{1,2,3}, {3,4}, {1,2}]]
>>> list(A.shuffle_product(B, overlap=True))
[[{1,2,3}, {1,2}, {3,4}], [{1,2,3}, {3,4}, {1,2}],
 [{3,4}, {1,2,3}, {1,2}], [{1,2,3}, {1,2,3,4}]]
>>> list(A.shuffle_product(C, overlap=True))
[[{1,2,3}, {1,2}, {4,5}], [{1,2,3}, {4,5}, {1,2}], [{4,5}, {1,2,3}, {1,2}],
 [{1,2,3,4,5}, {1,2}], [{1,2,3}, {1,2,4,5}]]
```

size()

Return the size of `self` (that is, the sum of all integers in all blocks) if `self` is a list of subsets of positive integers.

Else, return `None`.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionIntoSets([[3, 4, 1], [2], [1, 2, 3, 7]]); C
[{1,3,4}, {2}, {1,2,3,7}]
sage: C.size()
23
sage: C.size() == sum(k for k in C.shape_from_size())
True
sage: OrderedMultisetPartitionIntoSets([[7,1],[3]]).size()
11
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionIntoSets([[Integer(3), Integer(4),␣
→Integer(1)], [Integer(2)], [Integer(1), Integer(2), Integer(3),␣
→Integer(7)]]); C
[{1,3,4}, {2}, {1,2,3,7}]
>>> C.size()
23
>>> C.size() == sum(k for k in C.shape_from_size())
True
>>> OrderedMultisetPartitionIntoSets([[Integer(7),Integer(1)],[Integer(3)]]).
→size()
11
```

split_blocks(*k=2*)

Return a dictionary representing the k-splittings of `self`.

A k-tuple $(A^1, \ldots, A^k)$ of ordered multiset partitions into sets represents a k-splitting of an ordered multiset partition into sets $A = [b_1, \ldots, b_r]$ if one can express each block b_i as an (ordered) disjoint union of sets $b_i = b_i^1 \sqcup \cdots \sqcup b_i^k$ (some possibly empty) so that each A^j is the ordered multiset partition into sets corresponding to the list $[b_1^j, b_2^j, \ldots, b_r^j]$, excising empty sets appearing therein.

This operation represents the coproduct in Hopf algebra of ordered multiset partitions into sets in its natural basis [LM2018].

EXAMPLES:

```
sage: sorted(OrderedMultisetPartitionIntoSets([[1,2],[3,4]]).split_blocks(),␣
↪key=str)
[([], [{1,2}, {3,4}]),
 ([{1,2}, {3,4}], []),
 ([{1,2}, {3}], [{4}]),
 ([{1,2}, {4}], [{3}]),
 ([{1,2}], [{3,4}]),
 ([{1}, {3,4}], [{2}]),
 ([{1}, {3}], [{2}, {4}]),
 ([{1}, {4}], [{2}, {3}]),
 ([{1}], [{2}, {3,4}]),
 ([{2}, {3,4}], [{1}]),
 ([{2}, {3}], [{1}, {4}]),
 ([{2}, {4}], [{1}, {3}]),
 ([{2}], [{1}, {3,4}]),
 ([{3,4}], [{1,2}]),
 ([{3}], [{1,2}, {4}]),
 ([{4}], [{1,2}, {3}])]
sage: sorted(OrderedMultisetPartitionIntoSets([[1,2]]).split_blocks(3),␣
↪key=str)
[([], [], [{1,2}]), ([], [{1,2}], []), ([], [{1}], [{2}]),
 ([], [{2}], [{1}]), ([{1,2}], [], []), ([{1}], [], [{2}]),
 ([{1}], [{2}], []), ([{2}], [], [{1}]), ([{2}], [{1}], [])]
sage: OrderedMultisetPartitionIntoSets([[4],[4]]).split_blocks()
{([], [{4}, {4}]): 1, ([{4}], [{4}]): 2, ([{4}, {4}], []): 1}
```

```
>>> from sage.all import *
>>> sorted(OrderedMultisetPartitionIntoSets([[Integer(1),Integer(2)],
↪[Integer(3),Integer(4)]]).split_blocks(), key=str)
[([], [{1,2}, {3,4}]),
 ([{1,2}, {3,4}], []),
 ([{1,2}, {3}], [{4}]),
 ([{1,2}, {4}], [{3}]),
 ([{1,2}], [{3,4}]),
 ([{1}, {3,4}], [{2}]),
 ([{1}, {3}], [{2}, {4}]),
 ([{1}, {4}], [{2}, {3}]),
 ([{1}], [{2}, {3,4}]),
 ([{2}, {3,4}], [{1}]),
 ([{2}, {3}], [{1}, {4}]),
 ([{2}, {4}], [{1}, {3}]),
 ([{2}], [{1}, {3,4}]),
```

(continues on next page)

(continued from previous page)

```
 ([{3,4}], [{1,2}]),
 ([{3}], [{1,2}, {4}]),
 ([{4}], [{1,2}, {3}])]
>>> sorted(OrderedMultisetPartitionIntoSets([[Integer(1),Integer(2)]]).split_
↪blocks(Integer(3)), key=str)
[([], [], [{1,2}]), ([], [{1,2}], []), ([], [{1}], [{2}]),
 ([], [{2}], [{1}]), ([{1,2}], [], []), ([{1}], [], [{2}]),
 ([{1}], [{2}], []), ([{2}], [], [{1}]), ([{2}], [{1}], [])]
>>> OrderedMultisetPartitionIntoSets([[Integer(4)],[Integer(4)]]).split_
↪blocks()
{([], [{4}, {4}]): 1, ([{4}], [{4}]): 2, ([{4}, {4}], []): 1}
```

to_tableaux_words()

Return a sequence of lists corresponding to row words of (skew-)tableaux.

OUTPUT:

The minimaj bijection ϕ of [BCHOPSY2017] applied to `self`.

Todo

Implement option for mapping to sequence of (skew-)tableaux?

EXAMPLES:

```
sage: co = ((1,2,4),(4,5),(3,),(4,6,1),(2,3,1),(1,),(2,5))
sage: OrderedMultisetPartitionIntoSets(co).to_tableaux_words()
[[5, 1], [3, 1], [6], [5, 4, 2], [1, 4, 3, 4, 2, 1, 2]]
```

```
>>> from sage.all import *
>>> co = ((Integer(1),Integer(2),Integer(4)),(Integer(4),Integer(5)),
↪(Integer(3),),(Integer(4),Integer(6),Integer(1)),(Integer(2),Integer(3),
↪Integer(1)),(Integer(1),),(Integer(2),Integer(5)))
>>> OrderedMultisetPartitionIntoSets(co).to_tableaux_words()
[[5, 1], [3, 1], [6], [5, 4, 2], [1, 4, 3, 4, 2, 1, 2]]
```

weight(*as_weak_comp=False*)

Return a dictionary, with keys being the letters in `self.letters()` and values being their (positive) frequency.

Alternatively, if `as_weak_comp` is `True`, count the number of instances n_i for each distinct positive integer i across all blocks of `self`. Return as a list $[n_1, n_2, n_3, ..., n_k]$, where k is the max letter appearing in `self.letters()`.

EXAMPLES:

```
sage: c = OrderedMultisetPartitionIntoSets([[6,1],[1,3],[1,3,6]])
sage: c.weight()
{1: 3, 3: 2, 6: 2}
sage: c.weight(as_weak_comp=True)
[3, 0, 2, 0, 0, 2]
```

```
>>> from sage.all import *
>>> c = OrderedMultisetPartitionIntoSets([[Integer(6),Integer(1)],[Integer(1),
↪Integer(3)],[Integer(1),Integer(3),Integer(6)]])
>>> c.weight()
{1: 3, 3: 2, 6: 2}
>>> c.weight(as_weak_comp=True)
[3, 0, 2, 0, 0, 2]
```

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionsIntoSets**(*is_finite=None*, ***constraints*)

Bases: UniqueRepresentation, Parent

Ordered Multiset Partitions into Sets.

An *ordered multiset partition into sets* c of a multiset X is a list of nonempty subsets (not multisets), called the *blocks* of c, whose multi-union is X.

The number of blocks of c is called its *length*. The *order* of c is the cardinality of the multiset X. If, additionally, X is a multiset of positive integers, then the *size* of c is the sum of all elements of X.

The user may wish to focus on ordered multiset partitions into sets of a given size, or over a given alphabet. Hence, this class allows a variety of arguments as input.

INPUT:

Expects one or two arguments, with different behaviors resulting:

- One Argument:
  - X – a dictionary or list or tuple (representing a multiset for c), or an integer (representing the size of c)
- Two Arguments:
  - A – list (representing allowable letters within blocks of c), or a positive integer (representing the maximal allowable letter)
  - n – a nonnegative integer (the total number of letters within c)

Optional keyword arguments are as follows: (See corresponding methods in see *OrderedMultisetPartitionIntoSets* for more details.)

- `weight=X` (list or dictionary X) specifies the multiset for c
- `size=n` (integer n) specifies the size of c
- `alphabet=A` (iterable A) specifies allowable elements for the blocks of c
- `length=k` (integer k) specifies the number of blocks in the partition
- `min_length=k` (integer k) specifies minimum number of blocks in the partition
- `max_length=k` (integer k) specifies maximum number of blocks in the partition
- `order=n` (integer n) specifies the cardinality of the multiset that c partitions
- `min_order=n` (integer n) specifies minimum number of elements in the partition
- `max_order=n` (integer n) specifies maximum number of elements in the partition

EXAMPLES:

Passing one argument to *OrderedMultisetPartitionsIntoSets*:

There are 5 ordered multiset partitions into sets of the multiset $\{\{1, 1, 4\}\}$:

```
sage: OrderedMultisetPartitionsIntoSets([1,1,4]).cardinality()
5
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets([Integer(1),Integer(1),Integer(4)]).
↪cardinality()
5
```

Here is the list of them:

```
sage: OrderedMultisetPartitionsIntoSets([1,1,4]).list()
[[{1}, {1}, {4}], [{1}, {1,4}], [{1}, {4}, {1}], [{1,4}, {1}], [{4}, {1}, {1}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets([Integer(1),Integer(1),Integer(4)]).list()
[[{1}, {1}, {4}], [{1}, {1,4}], [{1}, {4}, {1}], [{1,4}, {1}], [{4}, {1}, {1}]]
```

By chance, there are also 5 ordered multiset partitions into sets of the integer 3:

```
sage: OrderedMultisetPartitionsIntoSets(3).cardinality()
5
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets(Integer(3)).cardinality()
5
```

Here is the list of them:

```
sage: OrderedMultisetPartitionsIntoSets(3).list()
[[{3}], [{1,2}], [{2}, {1}], [{1}, {2}], [{1}, {1}, {1}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets(Integer(3)).list()
[[{3}], [{1,2}], [{2}, {1}], [{1}, {2}], [{1}, {1}, {1}]]
```

Passing two arguments to *OrderedMultisetPartitionsIntoSets*:

There are also 5 ordered multiset partitions into sets of order 2 over the alphabet $\{1, 4\}$:

```
sage: OrderedMultisetPartitionsIntoSets([1, 4], 2)
Ordered Multiset Partitions into Sets of order 2 over alphabet {1, 4}
sage: OrderedMultisetPartitionsIntoSets([1, 4], 2).cardinality()
5
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets([Integer(1), Integer(4)], Integer(2))
Ordered Multiset Partitions into Sets of order 2 over alphabet {1, 4}
>>> OrderedMultisetPartitionsIntoSets([Integer(1), Integer(4)], Integer(2)).
↪cardinality()
5
```

Here is the list of them:

```
sage: OrderedMultisetPartitionsIntoSets([1, 4], 2).list()
[[{1,4}], [{1}, {1}], [{1}, {4}], [{4}, {1}], [{4}, {4}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets([Integer(1), Integer(4)], Integer(2)).list()
[[{1,4}], [{1}, {1}], [{1}, {4}], [{4}, {1}], [{4}, {4}]]
```

If no arguments are passed to `OrderedMultisetPartitionsIntoSets`, then the code returns all ordered multiset partitions into sets:

```
sage: OrderedMultisetPartitionsIntoSets()
Ordered Multiset Partitions into Sets
sage: [] in OrderedMultisetPartitionsIntoSets()
True
sage: [[2,3], [1]] in OrderedMultisetPartitionsIntoSets()
True
sage: [['a','b'], ['a']] in OrderedMultisetPartitionsIntoSets()
True
sage: [[-2,3], [3]] in OrderedMultisetPartitionsIntoSets()
True
sage: [[2], [3,3]] in OrderedMultisetPartitionsIntoSets()
False
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets()
Ordered Multiset Partitions into Sets
>>> [] in OrderedMultisetPartitionsIntoSets()
True
>>> [[Integer(2),Integer(3)], [Integer(1)]] in OrderedMultisetPartitionsIntoSets()
True
>>> [['a','b'], ['a']] in OrderedMultisetPartitionsIntoSets()
True
>>> [[-Integer(2),Integer(3)], [Integer(3)]] in␣
↪OrderedMultisetPartitionsIntoSets()
True
>>> [[Integer(2)], [Integer(3),Integer(3)]] in OrderedMultisetPartitionsIntoSets()
False
```

The following examples show how to test whether or not an object is an ordered multiset partition into sets:

```
sage: [[3,2],[2]] in OrderedMultisetPartitionsIntoSets()
True
sage: [[3,2],[2]] in OrderedMultisetPartitionsIntoSets(7)
True
sage: [[3,2],[2]] in OrderedMultisetPartitionsIntoSets([2,2,3])
True
sage: [[3,2],[2]] in OrderedMultisetPartitionsIntoSets(5)
False
```

```
>>> from sage.all import *
>>> [[Integer(3),Integer(2)],[Integer(2)]] in OrderedMultisetPartitionsIntoSets()
True
```

(continues on next page)

(continued from previous page)

```
>>> [[Integer(3),Integer(2)],[Integer(2)]] in␣
↪OrderedMultisetPartitionsIntoSets(Integer(7))
True
>>> [[Integer(3),Integer(2)],[Integer(2)]] in␣
↪OrderedMultisetPartitionsIntoSets([Integer(2),Integer(2),Integer(3)])
True
>>> [[Integer(3),Integer(2)],[Integer(2)]] in␣
↪OrderedMultisetPartitionsIntoSets(Integer(5))
False
```

Optional keyword arguments

Passing keyword arguments that are incompatible with required requirements results in an error; otherwise, the collection of ordered multiset partitions into sets is restricted accordingly:

The `weight` *keyword:*

This is used to specify which multiset X is to be considered, if this multiset was not passed as one of the required arguments for *OrderedMultisetPartitionsIntoSets*. In principle, it is a dictionary, but weak compositions are also allowed. For example, the ordered multiset partitions into sets of integer 4 are listed by weight below:

```
sage: OrderedMultisetPartitionsIntoSets(4, weight=[0,0,0,1])
Ordered Multiset Partitions into Sets of integer 4 with constraint: weight={4: 1}
sage: OrderedMultisetPartitionsIntoSets(4, weight=[0,0,0,1]).list()
[[{4}]]
sage: OrderedMultisetPartitionsIntoSets(4, weight=[1,0,1]).list()
[[{1}, {3}], [{1,3}], [{3}, {1}]]
sage: OrderedMultisetPartitionsIntoSets(4, weight=[0,2]).list()
[[{2}, {2}]]
sage: OrderedMultisetPartitionsIntoSets(4, weight=[0,1,1]).list()
[]
sage: OrderedMultisetPartitionsIntoSets(4, weight=[2,1]).list()
[[{1}, {1}, {2}], [{1}, {1,2}], [{1}, {2}, {1}], [{1,2}, {1}], [{2}, {1}, {1}]]
sage: O1 = OrderedMultisetPartitionsIntoSets(weight=[2,0,1])
sage: O2 = OrderedMultisetPartitionsIntoSets(weight={1:2, 3:1})
sage: O1 == O2
True
sage: OrderedMultisetPartitionsIntoSets(4, weight=[4]).list()
[[{1}, {1}, {1}, {1}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets(Integer(4), weight=[Integer(0),Integer(0),
↪Integer(0),Integer(1)])
Ordered Multiset Partitions into Sets of integer 4 with constraint: weight={4: 1}
>>> OrderedMultisetPartitionsIntoSets(Integer(4), weight=[Integer(0),Integer(0),
↪Integer(0),Integer(1)]).list()
[[{4}]]
>>> OrderedMultisetPartitionsIntoSets(Integer(4), weight=[Integer(1),Integer(0),
↪Integer(1)]).list()
[[{1}, {3}], [{1,3}], [{3}, {1}]]
>>> OrderedMultisetPartitionsIntoSets(Integer(4), weight=[Integer(0),Integer(2)]).
↪list()
[[{2}, {2}]]
```

(continues on next page)

(continued from previous page)

```
>>> OrderedMultisetPartitionsIntoSets(Integer(4), weight=[Integer(0),Integer(1),
↪Integer(1)]).list()
[]
>>> OrderedMultisetPartitionsIntoSets(Integer(4), weight=[Integer(2),Integer(1)]).
↪list()
[[{1}, {1}, {2}], [{1}, {1,2}], [{1}, {2}, {1}], [{1,2}, {1}], [{2}, {1}, {1}]]
>>> O1 = OrderedMultisetPartitionsIntoSets(weight=[Integer(2),Integer(0),
↪Integer(1)])
>>> O2 = OrderedMultisetPartitionsIntoSets(weight={Integer(1):Integer(2),␣
↪Integer(3):Integer(1)})
>>> O1 == O2
True
>>> OrderedMultisetPartitionsIntoSets(Integer(4), weight=[Integer(4)]).list()
[[{1}, {1}, {1}, {1}]]
```

The `size` *keyword:*

This is used to constrain the sum of entries across all blocks of the ordered multiset partition into sets. (This size is not pre-determined when alphabet A and order d are passed as required arguments.) For example, the ordered multiset partitions into sets of order 3 over the alphabet $[1, 2, 4]$ that have size equal to 5 are as follows:

```
sage: OMPs = OrderedMultisetPartitionsIntoSets
sage: OMPs([1,2,4], 3, size=5).list()
[[{1,2}, {2}], [{2}, {1,2}], [{2}, {2}, {1}],
 [{2}, {1}, {2}], [{1}, {2}, {2}]]
```

```
>>> from sage.all import *
>>> OMPs = OrderedMultisetPartitionsIntoSets
>>> OMPs([Integer(1),Integer(2),Integer(4)], Integer(3), size=Integer(5)).list()
[[{1,2}, {2}], [{2}, {1,2}], [{2}, {2}, {1}],
 [{2}, {1}, {2}], [{1}, {2}, {2}]]
```

The `alphabet` *option:*

This is used to constrain which integers appear across all blocks of the ordered multiset partition into sets. For example, the ordered multiset partitions into sets of integer 4 are listed for different choices of alphabet below. Note that `alphabet` is allowed to be an integer or an iterable:

```
sage: OMPs = OrderedMultisetPartitionsIntoSets
sage: OMPs(4, alphabet=3).list()
[[{1,3}], [{3}, {1}],
 [{1,2}, {1}], [{2}, {2}],
 [{2}, {1}, {1}], [{1}, {3}],
 [{1}, {1,2}], [{1}, {2}, {1}],
 [{1}, {1}, {2}], [{1}, {1}, {1}, {1}]]
sage: OMPs(4, alphabet=3) == OMPs(4, alphabet=[1,2,3])
True
sage: OMPs(4, alphabet=[3]).list()
[]
sage: OMPs(4, alphabet=[1,3]).list()
[[{1,3}], [{3}, {1}], [{1}, {3}], [{1}, {1}, {1}, {1}]]
sage: OMPs(4, alphabet=[2]).list()
[[{2}, {2}]]
```

(continues on next page)

(continued from previous page)

```
sage: OMPs(4, alphabet=[1,2]).list()
[[{1,2}, {1}], [{2}, {2}], [{2}, {1}, {1}], [{1}, {1,2}],
 [{1}, {2}, {1}], [{1}, {1}, {2}], [{1}, {1}, {1}, {1}]]
sage: OMPs(4, alphabet=4).list() == OMPs(4).list()
True
```

```
>>> from sage.all import *
>>> OMPs = OrderedMultisetPartitionsIntoSets
>>> OMPs(Integer(4), alphabet=Integer(3)).list()
[[{1,3}], [{3}, {1}],
 [{1,2}, {1}], [{2}, {2}],
 [{2}, {1}, {1}], [{1}, {3}],
 [{1}, {1,2}], [{1}, {2}, {1}],
 [{1}, {1}, {2}], [{1}, {1}, {1}, {1}]]
>>> OMPs(Integer(4), alphabet=Integer(3)) == OMPs(Integer(4),␣
↪alphabet=[Integer(1),Integer(2),Integer(3)])
True
>>> OMPs(Integer(4), alphabet=[Integer(3)]).list()
[]
>>> OMPs(Integer(4), alphabet=[Integer(1),Integer(3)]).list()
[[{1,3}], [{3}, {1}], [{1}, {3}], [{1}, {1}, {1}, {1}]]
>>> OMPs(Integer(4), alphabet=[Integer(2)]).list()
[[{2}, {2}]]
>>> OMPs(Integer(4), alphabet=[Integer(1),Integer(2)]).list()
[[{1,2}, {1}], [{2}, {2}], [{2}, {1}, {1}], [{1}, {1,2}],
 [{1}, {2}, {1}], [{1}, {1}, {2}], [{1}, {1}, {1}, {1}]]
>>> OMPs(Integer(4), alphabet=Integer(4)).list() == OMPs(Integer(4)).list()
True
```

The `length`, `min_length`, *and* `max_length` *options:*

These are used to constrain the number of blocks within the ordered multiset partitions into sets. For example, the ordered multiset partitions into sets of integer 4 of length exactly 2, at least 2, and at most 2 are given by:

```
sage: OrderedMultisetPartitionsIntoSets(4, length=2).list()
[[{3}, {1}], [{1,2}, {1}], [{2}, {2}], [{1}, {3}], [{1}, {1,2}]]
sage: OrderedMultisetPartitionsIntoSets(4, min_length=3).list()
[[{2}, {1}, {1}], [{1}, {2}, {1}], [{1}, {1}, {2}], [{1}, {1}, {1}, {1}]]
sage: OrderedMultisetPartitionsIntoSets(4, max_length=2).list()
[[{4}], [{1,3}], [{3}, {1}], [{1,2}, {1}], [{2}, {2}], [{1}, {3}],
 [{1}, {1,2}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets(Integer(4), length=Integer(2)).list()
[[{3}, {1}], [{1,2}, {1}], [{2}, {2}], [{1}, {3}], [{1}, {1,2}]]
>>> OrderedMultisetPartitionsIntoSets(Integer(4), min_length=Integer(3)).list()
[[{2}, {1}, {1}], [{1}, {2}, {1}], [{1}, {1}, {2}], [{1}, {1}, {1}, {1}]]
>>> OrderedMultisetPartitionsIntoSets(Integer(4), max_length=Integer(2)).list()
[[{4}], [{1,3}], [{3}, {1}], [{1,2}, {1}], [{2}, {2}], [{1}, {3}],
 [{1}, {1,2}]]
```

The `order`, `min_order`, *and* `max_order` *options:*

These are used to constrain the number of elements across all blocks of the ordered multiset partitions into sets. For example, the ordered multiset partitions into sets of integer 4 are listed by order below:

```
sage: OrderedMultisetPartitionsIntoSets(4, order=1).list()
[[{4}]]
sage: OrderedMultisetPartitionsIntoSets(4, order=2).list()
[[{1,3}], [{3}, {1}], [{2}, {2}], [{1}, {3}]]
sage: OrderedMultisetPartitionsIntoSets(4, order=3).list()
[[{1,2}, {1}], [{2}, {1}, {1}], [{1}, {1,2}], [{1}, {2}, {1}], [{1}, {1}, {2}]]
sage: OrderedMultisetPartitionsIntoSets(4, order=4).list()
[[{1}, {1}, {1}, {1}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets(Integer(4), order=Integer(1)).list()
[[{4}]]
>>> OrderedMultisetPartitionsIntoSets(Integer(4), order=Integer(2)).list()
[[{1,3}], [{3}, {1}], [{2}, {2}], [{1}, {3}]]
>>> OrderedMultisetPartitionsIntoSets(Integer(4), order=Integer(3)).list()
[[{1,2}, {1}], [{2}, {1}, {1}], [{1}, {1,2}], [{1}, {2}, {1}], [{1}, {1}, {2}]]
>>> OrderedMultisetPartitionsIntoSets(Integer(4), order=Integer(4)).list()
[[{1}, {1}, {1}, {1}]]
```

Also, here is a use of `max_order`, giving the ordered multiset partitions into sets of integer 4 with order 1 or 2:

```
sage: OrderedMultisetPartitionsIntoSets(4, max_order=2).list()
[[{4}], [{1,3}], [{3}, {1}], [{2}, {2}], [{1}, {3}]]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets(Integer(4), max_order=Integer(2)).list()
[[{4}], [{1,3}], [{3}, {1}], [{2}, {2}], [{1}, {3}]]
```

Element

alias of *OrderedMultisetPartitionIntoSets*

subset (*size*)

Return a subset of all ordered multiset partitions into sets.

INPUT:

- `size` – integer representing a slice of all ordered multiset partitions into sets

The slice alluded to above is taken with respect to length, or to order, or to size, depending on the constraints of `self`.

EXAMPLES:

```
sage: C = OrderedMultisetPartitionsIntoSets(weight={2:2, 3:1, 5:1})
sage: C.subset(3)
Ordered Multiset Partitions into Sets of multiset {{2, 2, 3, 5}} with␣
↪constraint: length=3
sage: C = OrderedMultisetPartitionsIntoSets(weight={2:2, 3:1, 5:1}, min_
↪length=2)
sage: C.subset(3)
Ordered Multiset Partitions into Sets of multiset {{2, 2, 3, 5}} with␣
↪constraint: length=3
```

(continues on next page)

(continued from previous page)

```
sage: C = OrderedMultisetPartitionsIntoSets(alphabet=[2,3,5])
sage: C.subset(3)
Ordered Multiset Partitions into Sets of order 3 over alphabet {2, 3, 5}
sage: C = OrderedMultisetPartitionsIntoSets(order=5)
sage: C.subset(3)
Ordered Multiset Partitions into Sets of integer 3 with constraint: order=5
sage: C = OrderedMultisetPartitionsIntoSets(alphabet=[2,3,5], order=5,␣
↪length=3)
sage: C.subset(3)
Ordered Multiset Partitions into Sets of order 3 over alphabet {2, 3, 5} with␣
↪constraint: length=3
sage: C = OrderedMultisetPartitionsIntoSets()
sage: C.subset(3)
Ordered Multiset Partitions into Sets of integer 3
sage: C.subset(3) == OrderedMultisetPartitionsIntoSets(3)
True
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionsIntoSets(weight={Integer(2):Integer(2),␣
↪Integer(3):Integer(1), Integer(5):Integer(1)})
>>> C.subset(Integer(3))
Ordered Multiset Partitions into Sets of multiset {{2, 2, 3, 5}} with␣
↪constraint: length=3
>>> C = OrderedMultisetPartitionsIntoSets(weight={Integer(2):Integer(2),␣
↪Integer(3):Integer(1), Integer(5):Integer(1)}, min_length=Integer(2))
>>> C.subset(Integer(3))
Ordered Multiset Partitions into Sets of multiset {{2, 2, 3, 5}} with␣
↪constraint: length=3
>>> C = OrderedMultisetPartitionsIntoSets(alphabet=[Integer(2),Integer(3),
↪Integer(5)])
>>> C.subset(Integer(3))
Ordered Multiset Partitions into Sets of order 3 over alphabet {2, 3, 5}
>>> C = OrderedMultisetPartitionsIntoSets(order=Integer(5))
>>> C.subset(Integer(3))
Ordered Multiset Partitions into Sets of integer 3 with constraint: order=5
>>> C = OrderedMultisetPartitionsIntoSets(alphabet=[Integer(2),Integer(3),
↪Integer(5)], order=Integer(5), length=Integer(3))
>>> C.subset(Integer(3))
Ordered Multiset Partitions into Sets of order 3 over alphabet {2, 3, 5} with␣
↪constraint: length=3
>>> C = OrderedMultisetPartitionsIntoSets()
>>> C.subset(Integer(3))
Ordered Multiset Partitions into Sets of integer 3
>>> C.subset(Integer(3)) == OrderedMultisetPartitionsIntoSets(Integer(3))
True
```

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionsIntoSets_X**(X)

Bases: *OrderedMultisetPartitionsIntoSets*

Class of ordered multiset partitions into sets of a fixed multiset X.

cardinality()

Return the number of ordered partitions of multiset X.

random_element()

Return a random element of `self`.

This method does not return elements of `self` with uniform probability, but it does cover all elements. The scheme is as follows:

- produce a random permutation `p` of the multiset;
- create blocks of an OMP `fat` by breaking `p` after non-ascents;
- take a random element of `fat.finer()`.

EXAMPLES:

```
sage: OrderedMultisetPartitionsIntoSets([1,1,3]).random_element()   # random
[{1}, {1,3}]
sage: OrderedMultisetPartitionsIntoSets([1,1,3]).random_element()   # random
[{3}, {1}, {1}]

sage: OMP = OrderedMultisetPartitionsIntoSets([1,1,3,3])
sage: d = {}
sage: for _ in range(1000):
....:     x = OMP.random_element()
....:     d[x] = d.get(x, 0) + 1
sage: d.values()   # random
[102, 25, 76, 24, 66, 88, 327, 27, 83, 83, 239, 72, 88]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets([Integer(1),Integer(1),Integer(3)]).
↪random_element()   # random
[{1}, {1,3}]
>>> OrderedMultisetPartitionsIntoSets([Integer(1),Integer(1),Integer(3)]).
↪random_element()   # random
[{3}, {1}, {1}]

>>> OMP = OrderedMultisetPartitionsIntoSets([Integer(1),Integer(1),Integer(3),
↪Integer(3)])
>>> d = {}
>>> for _ in range(Integer(1000)):
...     x = OMP.random_element()
...     d[x] = d.get(x, Integer(0)) + Integer(1)
>>> d.values()   # random
[102, 25, 76, 24, 66, 88, 327, 27, 83, 83, 239, 72, 88]
```

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionsIntoSets_X_constrai**

Bases: *OrderedMultisetPartitionsIntoSets*

Class of ordered multiset partitions into sets of a fixed multiset X satisfying constraints.

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionsIntoSets_all_constr**

Bases: *OrderedMultisetPartitionsIntoSets*

All ordered multiset partitions into sets (with or without constraints).

EXAMPLES:

```
sage: C = OrderedMultisetPartitionsIntoSets(); C
Ordered Multiset Partitions into Sets
sage: [[1],[1,'a']] in C
True

sage: OrderedMultisetPartitionsIntoSets(weight=[2,0,1], length=2)
Ordered Multiset Partitions into Sets of multiset {{1, 1, 3}} with constraint:␣
↪length=2
```

```
>>> from sage.all import *
>>> C = OrderedMultisetPartitionsIntoSets(); C
Ordered Multiset Partitions into Sets
>>> [[Integer(1)],[Integer(1),'a']] in C
True

>>> OrderedMultisetPartitionsIntoSets(weight=[Integer(2),Integer(0),Integer(1)],␣
↪length=Integer(2))
Ordered Multiset Partitions into Sets of multiset {{1, 1, 3}} with constraint:␣
↪length=2
```

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionsIntoSets_alph_d**(A, d)

Bases: *OrderedMultisetPartitionsIntoSets*

Class of ordered multiset partitions into sets of specified order d over a fixed alphabet A.

cardinality()

Return the number of ordered partitions of order `self._order` on alphabet `self._alphabet`.

random_element()

Return a random element of `self`.

This method does not return elements of `self` with uniform probability, but it does cover all elements. The scheme is as follows:

- produce a random composition C;
- choose random subsets of `self._alphabet` of size c for each c in C.

EXAMPLES:

```
sage: OrderedMultisetPartitionsIntoSets([1,4], 3).random_element()   # random
[{4}, {1,4}]
sage: OrderedMultisetPartitionsIntoSets([1,3], 4).random_element()   # random
[{1,3}, {1}, {3}]

sage: OMP = OrderedMultisetPartitionsIntoSets([2,3,4], 2)
sage: d = {}
sage: for _ in range(1200):
....:     x = OMP.random_element()
....:     d[x] = d.get(x, 0) + 1
sage: d.values()  # random
[192, 68, 73, 61, 69, 60, 77, 204, 210, 66, 53, 67]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets([Integer(1),Integer(4)], Integer(3)).
↪random_element()  # random
[{4}, {1,4}]
>>> OrderedMultisetPartitionsIntoSets([Integer(1),Integer(3)], Integer(4)).
↪random_element()  # random
[{1,3}, {1}, {3}]

>>> OMP = OrderedMultisetPartitionsIntoSets([Integer(2),Integer(3),
↪Integer(4)], Integer(2))
>>> d = {}
>>> for _ in range(Integer(1200)):
...     x = OMP.random_element()
...     d[x] = d.get(x, Integer(0)) + Integer(1)
>>> d.values()  # random
[192, 68, 73, 61, 69, 60, 77, 204, 210, 66, 53, 67]
```

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionsIntoSets_alph_d_con**

Bases: *OrderedMultisetPartitionsIntoSets*

Class of ordered multiset partitions into sets of specified order d over a fixed alphabet A satisfying constraints.

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionsIntoSets_n**(*n*)

Bases: *OrderedMultisetPartitionsIntoSets*

Ordered multiset partitions into sets of a fixed integer n.

cardinality()

Return the number of elements in `self`.

random_element()

Return a random element of `self`.

This method does not return elements of `self` with uniform probability, but it does cover all elements. The scheme is as follows:

- produce a random composition C;
- choose a random partition of c into distinct parts for each c in C.

EXAMPLES:

```
sage: OrderedMultisetPartitionsIntoSets(5).random_element()  # random
[{1,2}, {1}, {1}]
sage: OrderedMultisetPartitionsIntoSets(5).random_element()  # random
[{2}, {1,2}]

sage: OMP = OrderedMultisetPartitionsIntoSets(5)
sage: d = {}
sage: for _ in range(1100):
....:     x = OMP.random_element()
....:     d[x] = d.get(x, 0) + 1
sage: d.values()  # random
[72, 73, 162, 78, 135, 75, 109, 65, 135, 134, 62]
```

```
>>> from sage.all import *
>>> OrderedMultisetPartitionsIntoSets(Integer(5)).random_element()   # random
[{1,2}, {1}, {1}]
>>> OrderedMultisetPartitionsIntoSets(Integer(5)).random_element()   # random
[{2}, {1,2}]

>>> OMP = OrderedMultisetPartitionsIntoSets(Integer(5))
>>> d = {}
>>> for _ in range(Integer(1100)):
...     x = OMP.random_element()
...     d[x] = d.get(x, Integer(0)) + Integer(1)
>>> d.values()  # random
[72, 73, 162, 78, 135, 75, 109, 65, 135, 134, 62]
```

class sage.combinat.multiset_partition_into_sets_ordered.**OrderedMultisetPartitionsIntoSets_n_constra**

Bases: *OrderedMultisetPartitionsIntoSets*

Class of ordered multiset partitions into sets of a fixed integer n satisfying constraints.

5.1.141 Non-commutative symmetric functions and quasi-symmetric functions

- *Introduction to Quasisymmetric Functions*
- *Non-Commutative Symmetric Functions (NCSF)*
- *Quasi-Symmetric Functions (QSym)*
- *Generic code for bases*

5.1.142 Common combinatorial tools

REFERENCES:

sage.combinat.ncsf_qsym.combinatorics.**coeff_dab**(I, J)

Return the number of standard composition tableaux of shape I with descent composition J.

INPUT:

- I, J – compositions

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_dab
sage: coeff_dab(Composition([2,1]),Composition([2,1]))
1
sage: coeff_dab(Composition([1,1,2]),Composition([1,2,1]))
0
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import coeff_dab
>>> coeff_dab(Composition([Integer(2),Integer(1)]),Composition([Integer(2),
↪Integer(1)]))
1
```

(continues on next page)

(continued from previous page)

```
>>> coeff_dab(Composition([Integer(1),Integer(1),Integer(2)]),
↪Composition([Integer(1),Integer(2),Integer(1)]))
0
```

sage.combinat.ncsf_qsym.combinatorics.**coeff_ell**(J, I)

Return the coefficient $\ell_{J,I}$ as defined in [NCSF].

INPUT:

- `J` – a composition
- `I` – a composition refining `J`

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_ell
sage: coeff_ell(Composition([1,1,1]), Composition([2,1]))
2
sage: coeff_ell(Composition([2,1]), Composition([3]))
2
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import coeff_ell
>>> coeff_ell(Composition([Integer(1),Integer(1),Integer(1)]),␣
↪Composition([Integer(2),Integer(1)]))
2
>>> coeff_ell(Composition([Integer(2),Integer(1)]), Composition([Integer(3)]))
2
```

sage.combinat.ncsf_qsym.combinatorics.**coeff_lp**(J, I)

Return the coefficient $lp_{J,I}$ as defined in [NCSF].

INPUT:

- `J` – a composition
- `I` – a composition refining `J`

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_lp
sage: coeff_lp(Composition([1,1,1]), Composition([2,1]))
1
sage: coeff_lp(Composition([2,1]), Composition([3]))
1
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import coeff_lp
>>> coeff_lp(Composition([Integer(1),Integer(1),Integer(1)]),␣
↪Composition([Integer(2),Integer(1)]))
1
>>> coeff_lp(Composition([Integer(2),Integer(1)]), Composition([Integer(3)]))
1
```

sage.combinat.ncsf_qsym.combinatorics.**coeff_pi**(*J*, *I*)

Return the coefficient $\pi_{J,I}$ as defined in [NCSF].

INPUT:

- `J` – a composition
- `I` – a composition refining `J`

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_pi
sage: coeff_pi(Composition([1,1,1]), Composition([2,1]))
2
sage: coeff_pi(Composition([2,1]), Composition([3]))
6
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import coeff_pi
>>> coeff_pi(Composition([Integer(1),Integer(1),Integer(1)]),␣
↪Composition([Integer(2),Integer(1)]))
2
>>> coeff_pi(Composition([Integer(2),Integer(1)]), Composition([Integer(3)]))
6
```

sage.combinat.ncsf_qsym.combinatorics.**coeff_sp**(*J*, *I*)

Return the coefficient $sp_{J,I}$ as defined in [NCSF].

INPUT:

- `J` – a composition
- `I` – a composition refining `J`

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_sp
sage: coeff_sp(Composition([1,1,1]), Composition([2,1]))
2
sage: coeff_sp(Composition([2,1]), Composition([3]))
4
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import coeff_sp
>>> coeff_sp(Composition([Integer(1),Integer(1),Integer(1)]),␣
↪Composition([Integer(2),Integer(1)]))
2
>>> coeff_sp(Composition([Integer(2),Integer(1)]), Composition([Integer(3)]))
4
```

sage.combinat.ncsf_qsym.combinatorics.**compositions_order**(*n*)

Return the compositions of n ordered as defined in [QSCHUR].

Let $S(\gamma)$ return the composition γ after sorting. For compositions α and β, we order $\alpha \rhd \beta$ if

1) $S(\alpha) > S(\beta)$ lexicographically, or
2) $S(\alpha) = S(\beta)$ and $\alpha > \beta$ lexicographically.

INPUT:

- n – positive integer

OUTPUT: list of the compositions of n sorted into decreasing order by $\rhd$

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import compositions_order
sage: compositions_order(3)
[[3], [2, 1], [1, 2], [1, 1, 1]]
sage: compositions_order(4)
[[4], [3, 1], [1, 3], [2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import compositions_order
>>> compositions_order(Integer(3))
[[3], [2, 1], [1, 2], [1, 1, 1]]
>>> compositions_order(Integer(4))
[[4], [3, 1], [1, 3], [2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

sage.combinat.ncsf_qsym.combinatorics.**m_to_s_stat**(*R*, *I*, *K*)

Return the coefficient of the complete non-commutative symmetric function S^K in the expansion of the monomial non-commutative symmetric function M^I with respect to the complete basis over the ring R. This is the coefficient in formula (36) of Tevlin's paper [Tev2007].

INPUT:

- R – a ring; supposed to be a **Q**-algebra
- I, K – compositions

OUTPUT:

- The coefficient of S^K in the expansion of M^I in the complete basis of the non-commutative symmetric functions over R.

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import m_to_s_stat
sage: m_to_s_stat(QQ, Composition([2,1]), Composition([1,1,1]))
-1
sage: m_to_s_stat(QQ, Composition([3]), Composition([1,2]))
-2
sage: m_to_s_stat(QQ, Composition([2,1,2]), Composition([2,1,2]))
8/3
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import m_to_s_stat
>>> m_to_s_stat(QQ, Composition([Integer(2),Integer(1)]), Composition([Integer(1),
↪Integer(1),Integer(1)]))
-1
>>> m_to_s_stat(QQ, Composition([Integer(3)]), Composition([Integer(1),
↪Integer(2)]))
```

(continues on next page)

(continued from previous page)

```
-2
>>> m_to_s_stat(QQ, Composition([Integer(2),Integer(1),Integer(2)]),␣
↪Composition([Integer(2),Integer(1),Integer(2)]))
8/3
```

sage.combinat.ncsf_qsym.combinatorics.**number_of_SSRCT**(*shape_comp*)

The number of semi-standard reverse composition tableaux.

The dual quasisymmetric-Schur functions satisfy a left Pieri rule where $S_n dQS_\gamma$ is a sum over dual quasisymmetric-Schur functions indexed by compositions which contain the composition γ. The definition of an SSRCT comes from this rule. The number of SSRCT of content β and shape α is equal to the number of SSRCT of content $(\beta_2, \ldots, \beta_\ell)$ and shape γ where dQS_α appears in the expansion of $S_{\beta_1} dQS_\gamma$.

In sage the recording tableau for these objects are called `CompositionTableaux`.

INPUT:

- `content_comp`, `shape_comp` – compositions

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import number_of_SSRCT
sage: number_of_SSRCT(Composition([3,1]), Composition([1,3]))
0
sage: number_of_SSRCT(Composition([1,2,1]), Composition([1,3]))
1
sage: number_of_SSRCT(Composition([1,1,2,2]), Composition([3,3]))
2
sage: all(CompositionTableaux(be).cardinality()
....:     == sum(number_of_SSRCT(al,be)*binomial(4,len(al))
....:            for al in Compositions(4))
....:     for be in Compositions(4))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import number_of_SSRCT
>>> number_of_SSRCT(Composition([Integer(3),Integer(1)]), Composition([Integer(1),
↪Integer(3)]))
0
>>> number_of_SSRCT(Composition([Integer(1),Integer(2),Integer(1)]),␣
↪Composition([Integer(1),Integer(3)]))
1
>>> number_of_SSRCT(Composition([Integer(1),Integer(1),Integer(2),Integer(2)]),␣
↪Composition([Integer(3),Integer(3)]))
2
>>> all(CompositionTableaux(be).cardinality()
...     == sum(number_of_SSRCT(al,be)*binomial(Integer(4),len(al))
...            for al in Compositions(Integer(4)))
...     for be in Compositions(Integer(4)))
True
```

sage.combinat.ncsf_qsym.combinatorics.**number_of_fCT**(*shape_comp*)

Return the number of Immaculate tableaux of shape `shape_comp` and content `content_comp`.

See [BBSSZ2012], Definition 3.9, for the notion of an immaculate tableau.

INPUT:

- `content_comp`, `shape_comp` – compositions

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import number_of_fCT
sage: number_of_fCT(Composition([3,1]), Composition([1,3]))
0
sage: number_of_fCT(Composition([1,2,1]), Composition([1,3]))
1
sage: number_of_fCT(Composition([1,1,3,1]), Composition([2,1,3]))
2
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.combinatorics import number_of_fCT
>>> number_of_fCT(Composition([Integer(3),Integer(1)]), Composition([Integer(1),
↪Integer(3)]))
0
>>> number_of_fCT(Composition([Integer(1),Integer(2),Integer(1)]),␣
↪Composition([Integer(1),Integer(3)]))
1
>>> number_of_fCT(Composition([Integer(1),Integer(1),Integer(3),Integer(1)]),␣
↪Composition([Integer(2),Integer(1),Integer(3)]))
2
```

5.1.143 Generic code for bases

This is a collection of code that is shared by bases of noncommutative symmetric functions and quasisymmetric functions.

AUTHORS:

- Jason Bandlow
- Franco Saliola
- Chris Berg

class `sage.combinat.ncsf_qsym.generic_basis_code.`**`AlgebraMorphism`**(*domain*, *on_generators*, *position=0*, *codomain=None*, *category=None*, *anti=False*)

Bases: `ModuleMorphismByLinearity`

A class for algebra morphism defined on a free algebra from the image of the generators

class `sage.combinat.ncsf_qsym.generic_basis_code.`**`BasesOfQSymOrNCSF`**(*parent_with_realization*)

Bases: `Category_realization_of_parent`

class ElementMethods

Bases: `object`

`degree()`

The maximum of the degrees of the homogeneous summands.

See also

`homogeneous_degree()`

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: x.degree()
2
sage: (x^3 + 4*y^2).degree()
6
sage: ((1 + x)^3).degree()
6
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> (x, y) = (S[Integer(2)], S[Integer(3)])
>>> x.degree()
2
>>> (x**Integer(3) + Integer(4)*y**Integer(2)).degree()
6
>>> ((Integer(1) + x)**Integer(3)).degree()
6
```

```
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: (x, y) = (F[2], F[3])
sage: x.degree()
2
sage: (x^3 + 4*y^2).degree()
6
sage: ((1 + x)^3).degree()
6
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(QQ).F()
>>> (x, y) = (F[Integer(2)], F[Integer(3)])
>>> x.degree()
2
>>> (x**Integer(3) + Integer(4)*y**Integer(2)).degree()
6
>>> ((Integer(1) + x)**Integer(3)).degree()
6
```

degree_negation()

Return the image of `self` under the degree negation automorphism of the parent of `self`.

The degree negation is the automorphism which scales every homogeneous element of degree k by $(-1)^k$ (for all k).

Calling `degree_negation(self)` is equivalent to calling `self.parent().degree_negation(self)`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: f = 2*S[2,1] + 4*S[1,1] - 5*S[1,2] - 3*S[[]]
sage: f.degree_negation()
-3*S[] + 4*S[1, 1] + 5*S[1, 2] - 2*S[2, 1]

sage: QSym = QuasiSymmetricFunctions(QQ)
sage: dI = QSym.dualImmaculate()
sage: f = -3*dI[2,1] + 4*dI[2] + 2*dI[1]
sage: f.degree_negation()
-2*dI[1] + 4*dI[2] + 3*dI[2, 1]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> f = Integer(2)*S[Integer(2),Integer(1)] + Integer(4)*S[Integer(1),
↪Integer(1)] - Integer(5)*S[Integer(1),Integer(2)] - Integer(3)*S[[]]
>>> f.degree_negation()
-3*S[] + 4*S[1, 1] + 5*S[1, 2] - 2*S[2, 1]

>>> QSym = QuasiSymmetricFunctions(QQ)
>>> dI = QSym.dualImmaculate()
>>> f = -Integer(3)*dI[Integer(2),Integer(1)] + Integer(4)*dI[Integer(2)]␣
↪+ Integer(2)*dI[Integer(1)]
>>> f.degree_negation()
-2*dI[1] + 4*dI[2] + 3*dI[2, 1]
```

Todo

Generalize this to all graded vector spaces?

duality_pairing(*y*)

The duality pairing between elements of $NSym$ and elements of $QSym$.

The complete basis is dual to the monomial basis with respect to this pairing.

INPUT:

- `y` – an element of the dual Hopf algebra of `self`

OUTPUT: the result of pairing `self` with `y`

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).Ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R[1,1,2].duality_pairing(F[1,1,2])
1
sage: R[1,2,1].duality_pairing(F[1,1,2])
0
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).Ribbon()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
```

(continues on next page)

(continued from previous page)

```
>>> R[Integer(1),Integer(1),Integer(2)].duality_pairing(F[Integer(1),
↪Integer(1),Integer(2)])
1
>>> R[Integer(1),Integer(2),Integer(1)].duality_pairing(F[Integer(1),
↪Integer(1),Integer(2)])
0
```

```
sage: L = NonCommutativeSymmetricFunctions(QQ).Elementary()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: L[1,2].duality_pairing(F[1,2])
0
sage: L[1,1,1].duality_pairing(F[1,2])
1
```

```
>>> from sage.all import *
>>> L = NonCommutativeSymmetricFunctions(QQ).Elementary()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> L[Integer(1),Integer(2)].duality_pairing(F[Integer(1),Integer(2)])
0
>>> L[Integer(1),Integer(1),Integer(1)].duality_pairing(F[Integer(1),
↪Integer(2)])
1
```

skew_by(*y*, *side='left'*)

The operation which is dual to multiplication by `y`, where `y` is an element of the dual space of `self`.

This is calculated through the coproduct of `self` and the expansion of `y` in the dual basis.

INPUT:

- `y` – an element of the dual Hopf algebra of `self`
- `side` – (default: `'left'`) either `'left'` or `'right'`

OUTPUT: the result of skewing `self` by `y`, on the side `side`

EXAMPLES:

Skewing an element of NCSF by an element of QSym:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R([2,2,2]).skew_by(F[1,1])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 2*R[2, 2] + R[3, 1] +
↪R[4]
sage: R([2,2,2]).skew_by(F[2])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 3*R[2, 2] + R[3, 1] +
↪R[4]
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).ribbon()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> R([Integer(2),Integer(2),Integer(2)]).skew_by(F[Integer(1),
↪Integer(1)])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 2*R[2, 2] + R[3, 1] +
↪R[4]
```

(continues on next page)

(continued from previous page)

```
>>> R([Integer(2),Integer(2),Integer(2)]).skew_by(F[Integer(2)])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 3*R[2, 2] + R[3, 1] +␣
↪R[4]
```

Skewing an element of QSym by an element of NCSF:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F[3,2].skew_by(R[1,1])
0
sage: F[3,2].skew_by(R[1,1], side='right')
0
sage: F[3,2].skew_by(S[1,1,1], side='right')
F[2]
sage: F[3,2].skew_by(S[1,2], side='right')
F[2]
sage: F[3,2].skew_by(S[2,1], side='right')
0
sage: F[3,2].skew_by(S[1,1,1])
F[2]
sage: F[3,2].skew_by(S[1,1])
F[1, 2]
sage: F[3,2].skew_by(S[1])
F[2, 2]
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> R = NonCommutativeSymmetricFunctions(QQ).R()
>>> F = QuasiSymmetricFunctions(QQ).F()
>>> F[Integer(3),Integer(2)].skew_by(R[Integer(1),Integer(1)])
0
>>> F[Integer(3),Integer(2)].skew_by(R[Integer(1),Integer(1)], side='right
↪')
0
>>> F[Integer(3),Integer(2)].skew_by(S[Integer(1),Integer(1),Integer(1)],␣
↪side='right')
F[2]
>>> F[Integer(3),Integer(2)].skew_by(S[Integer(1),Integer(2)], side='right
↪')
F[2]
>>> F[Integer(3),Integer(2)].skew_by(S[Integer(2),Integer(1)], side='right
↪')
0
>>> F[Integer(3),Integer(2)].skew_by(S[Integer(1),Integer(1),Integer(1)])
F[2]
>>> F[Integer(3),Integer(2)].skew_by(S[Integer(1),Integer(1)])
F[1, 2]
>>> F[Integer(3),Integer(2)].skew_by(S[Integer(1)])
F[2, 2]
```

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M[3,2].skew_by(S[2])
0
sage: M[3,2].skew_by(S[2], side='right')
M[3]
sage: M[3,2].skew_by(S[3])
M[2]
sage: M[3,2].skew_by(S[3], side='right')
0
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> R = NonCommutativeSymmetricFunctions(QQ).R()
>>> M = QuasiSymmetricFunctions(QQ).M()
>>> M[Integer(3),Integer(2)].skew_by(S[Integer(2)])
0
>>> M[Integer(3),Integer(2)].skew_by(S[Integer(2)], side='right')
M[3]
>>> M[Integer(3),Integer(2)].skew_by(S[Integer(3)])
M[2]
>>> M[Integer(3),Integer(2)].skew_by(S[Integer(3)], side='right')
0
```

class ParentMethods

Bases: `object`

alternating_sum_of_compositions(*n*)

Alternating sum over compositions of `n`.

Note that this differs from the method *alternating_sum_of_finer_compositions()* because the coefficient of the composition 1^n is positive. This method is used in the expansion of the elementary generators into the complete generators and vice versa.

INPUT:

- `n` – positive integer

OUTPUT:

- The expansion of the complete generator indexed by `n` into the elementary basis.

EXAMPLES:

```
sage: L = NonCommutativeSymmetricFunctions(QQ).L()
sage: L.alternating_sum_of_compositions(0)
L[]
sage: L.alternating_sum_of_compositions(1)
L[1]
sage: L.alternating_sum_of_compositions(2)
L[1, 1] - L[2]
sage: L.alternating_sum_of_compositions(3)
L[1, 1, 1] - L[1, 2] - L[2, 1] + L[3]
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.alternating_sum_of_compositions(3)
S[1, 1, 1] - S[1, 2] - S[2, 1] + S[3]
```

```
>>> from sage.all import *
>>> L = NonCommutativeSymmetricFunctions(QQ).L()
>>> L.alternating_sum_of_compositions(Integer(0))
L[]
>>> L.alternating_sum_of_compositions(Integer(1))
L[1]
>>> L.alternating_sum_of_compositions(Integer(2))
L[1, 1] - L[2]
>>> L.alternating_sum_of_compositions(Integer(3))
L[1, 1, 1] - L[1, 2] - L[2, 1] + L[3]
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S.alternating_sum_of_compositions(Integer(3))
S[1, 1, 1] - S[1, 2] - S[2, 1] + S[3]
```

alternating_sum_of_fatter_compositions(*composition*)

Return the alternating sum of fatter compositions in a basis of the non-commutative symmetric functions.

INPUT:

- `composition` – a composition

OUTPUT:

- The alternating sum of the compositions fatter than `composition`, in the basis `self`. The alternation is upon the length of the compositions, and is normalized so that `composition` has coefficient 1.

EXAMPLES:

```
sage: NCSF=NonCommutativeSymmetricFunctions(QQ)
sage: elementary = NCSF.elementary()
sage: elementary.alternating_sum_of_fatter_compositions(Composition([2,2,
→1]))
L[2, 2, 1] - L[2, 3] - L[4, 1] + L[5]
sage: elementary.alternating_sum_of_fatter_compositions(Composition([1,
→2]))
L[1, 2] - L[3]
```

```
>>> from sage.all import *
>>> NCSF=NonCommutativeSymmetricFunctions(QQ)
>>> elementary = NCSF.elementary()
>>> elementary.alternating_sum_of_fatter_
→compositions(Composition([Integer(2),Integer(2),Integer(1)]))
L[2, 2, 1] - L[2, 3] - L[4, 1] + L[5]
>>> elementary.alternating_sum_of_fatter_
→compositions(Composition([Integer(1),Integer(2)]))
L[1, 2] - L[3]
```

alternating_sum_of_finer_compositions(*composition*, *conjugate=False*)

Return the alternating sum of finer compositions in a basis of the non-commutative symmetric functions.

INPUT:

- `composition` – a composition
- `conjugate` – boolean (default: `False`)

OUTPUT:

- The alternating sum of the compositions finer than `composition`, in the basis `self`. The alternation is upon the length of the compositions, and is normalized so that `composition` has coefficient 1. If the variable `conjugate` is set to `True`, then the conjugate of `composition` is used instead of composition.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: elementary = NCSF.elementary()
sage: elementary.alternating_sum_of_finer_compositions(Composition([2,2,
↪1]))
L[1, 1, 1, 1, 1] - L[1, 1, 2, 1] - L[2, 1, 1, 1] + L[2, 2, 1]
sage: elementary.alternating_sum_of_finer_compositions(Composition([1,2]))
-L[1, 1, 1] + L[1, 2]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> elementary = NCSF.elementary()
>>> elementary.alternating_sum_of_finer_
↪compositions(Composition([Integer(2),Integer(2),Integer(1)]))
L[1, 1, 1, 1, 1] - L[1, 1, 2, 1] - L[2, 1, 1, 1] + L[2, 2, 1]
>>> elementary.alternating_sum_of_finer_
↪compositions(Composition([Integer(1),Integer(2)]))
-L[1, 1, 1] + L[1, 2]
```

counit_on_basis(*I*)

The counit is defined by sending all elements of positive degree to zero.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.counit_on_basis([1,3])
0
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M.counit_on_basis([1,3])
0
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S.counit_on_basis([Integer(1),Integer(3)])
0
>>> M = QuasiSymmetricFunctions(QQ).M()
>>> M.counit_on_basis([Integer(1),Integer(3)])
0
```

degree_negation(*element*)

Return the image of `element` under the degree negation automorphism of `self`.

The degree negation is the automorphism which scales every homogeneous element of degree k by $(-1)^k$ (for all k).

INPUT:

- `element` – element of `self`

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: f = 2*S[2,1] + 4*S[1,1] - 5*S[1,2] - 3*S[[]]
```

(continues on next page)

(continued from previous page)

```
sage: S.degree_negation(f)
-3*S[] + 4*S[1, 1] + 5*S[1, 2] - 2*S[2, 1]

sage: QSym = QuasiSymmetricFunctions(QQ)
sage: dI = QSym.dualImmaculate()
sage: f = -3*dI[2,1] + 4*dI[2] + 2*dI[1]
sage: dI.degree_negation(f)
-2*dI[1] + 4*dI[2] + 3*dI[2, 1]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> f = Integer(2)*S[Integer(2),Integer(1)] + Integer(4)*S[Integer(1),
↪Integer(1)] - Integer(5)*S[Integer(1),Integer(2)] - Integer(3)*S[[]]
>>> S.degree_negation(f)
-3*S[] + 4*S[1, 1] + 5*S[1, 2] - 2*S[2, 1]

>>> QSym = QuasiSymmetricFunctions(QQ)
>>> dI = QSym.dualImmaculate()
>>> f = -Integer(3)*dI[Integer(2),Integer(1)] + Integer(4)*dI[Integer(2)]␣
↪+ Integer(2)*dI[Integer(1)]
>>> dI.degree_negation(f)
-2*dI[1] + 4*dI[2] + 3*dI[2, 1]
```

Todo

Generalize this to all graded vector spaces?

degree_on_basis(*I*)

Return the degree of the basis element indexed by I.

INPUT:

- `I` – a composition

OUTPUT:

- The degree of the non-commutative symmetric function basis element of `self` indexed by `I`. By definition, this is the size of the composition `I`.

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: R.degree_on_basis(Composition([2,3]))
5
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: M.degree_on_basis(Composition([3,2]))
5
sage: M.degree_on_basis(Composition([]))
0
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).ribbon()
>>> R.degree_on_basis(Composition([Integer(2),Integer(3)]))
```

(continues on next page)

(continued from previous page)

```
5
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> M.degree_on_basis(Composition([Integer(3),Integer(2)]))
5
>>> M.degree_on_basis(Composition([]))
0
```

duality_pairing(*x*, *y*)

The duality pairing between elements of $NSym$ and elements of $QSym$.

This is a default implementation that uses `self.realizations_of().a_realization()` and its dual basis.

INPUT:

- `x` – an element of `self`
- `y` – an element in the dual basis of `self`

OUTPUT:

- The result of pairing the function `x` from `self` with the function `y` from the dual basis of `self`

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).Ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R.duality_pairing(R[1,1,2], F[1,1,2])
1
sage: R.duality_pairing(R[1,2,1], F[1,1,2])
0
sage: F.duality_pairing(F[1,2,1], R[1,1,2])
0
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).Ribbon()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> R.duality_pairing(R[Integer(1),Integer(1),Integer(2)], F[Integer(1),
↪Integer(1),Integer(2)])
1
>>> R.duality_pairing(R[Integer(1),Integer(2),Integer(1)], F[Integer(1),
↪Integer(1),Integer(2)])
0
>>> F.duality_pairing(F[Integer(1),Integer(2),Integer(1)], R[Integer(1),
↪Integer(1),Integer(2)])
0
```

```
sage: S = NonCommutativeSymmetricFunctions(QQ).Complete()
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: S.duality_pairing(S[1,1,2], M[1,1,2])
1
sage: S.duality_pairing(S[1,2,1], M[1,1,2])
0
sage: M.duality_pairing(M[1,1,2], S[1,1,2])
1
sage: M.duality_pairing(M[1,2,1], S[1,1,2])
0
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).Complete()
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> S.duality_pairing(S[Integer(1),Integer(1),Integer(2)], M[Integer(1),
↪Integer(1),Integer(2)])
1
>>> S.duality_pairing(S[Integer(1),Integer(2),Integer(1)], M[Integer(1),
↪Integer(1),Integer(2)])
0
>>> M.duality_pairing(M[Integer(1),Integer(1),Integer(2)], S[Integer(1),
↪Integer(1),Integer(2)])
1
>>> M.duality_pairing(M[Integer(1),Integer(2),Integer(1)], S[Integer(1),
↪Integer(1),Integer(2)])
0
```

```
sage: S = NonCommutativeSymmetricFunctions(QQ).Complete()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: S.duality_pairing(S[1,2], F[1,1,1])
0
sage: S.duality_pairing(S[1,1,1,1], F[4])
1
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).Complete()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> S.duality_pairing(S[Integer(1),Integer(2)], F[Integer(1),Integer(1),
↪Integer(1)])
0
>>> S.duality_pairing(S[Integer(1),Integer(1),Integer(1),Integer(1)],␣
↪F[Integer(4)])
1
```

duality_pairing_by_coercion(x, y)

The duality pairing between elements of NSym and elements of QSym.

This is a default implementation that uses `self.realizations_of().a_realization()` and its dual basis.

INPUT:

- `x` – an element of `self`
- `y` – an element in the dual basis of `self`

OUTPUT:

- The result of pairing the function `x` from `self` with the function `y` from the dual basis of `self`

EXAMPLES:

```
sage: L = NonCommutativeSymmetricFunctions(QQ).Elementary()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: L.duality_pairing_by_coercion(L[1,2], F[1,2])
0
sage: F.duality_pairing_by_coercion(F[1,2], L[1,2])
0
sage: L.duality_pairing_by_coercion(L[1,1,1], F[1,2])
```

(continues on next page)

(continued from previous page)

```
1
sage: F.duality_pairing_by_coercion(F[1,2], L[1,1,1])
1
```

```
>>> from sage.all import *
>>> L = NonCommutativeSymmetricFunctions(QQ).Elementary()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> L.duality_pairing_by_coercion(L[Integer(1),Integer(2)], F[Integer(1),
↪Integer(2)])
0
>>> F.duality_pairing_by_coercion(F[Integer(1),Integer(2)], L[Integer(1),
↪Integer(2)])
0
>>> L.duality_pairing_by_coercion(L[Integer(1),Integer(1),Integer(1)],
↪F[Integer(1),Integer(2)])
1
>>> F.duality_pairing_by_coercion(F[Integer(1),Integer(2)], L[Integer(1),
↪Integer(1),Integer(1)])
1
```

duality_pairing_matrix(*basis*, *degree*)

The matrix of scalar products between elements of NSym and elements of QSym.

INPUT:

- `basis` – a basis of the dual Hopf algebra
- `degree` – nonnegative integer

OUTPUT:

- The matrix of scalar products between the basis `self` and the basis `basis` in the dual Hopf algebra in degree `degree`.

EXAMPLES:

The ribbon basis of NCSF is dual to the fundamental basis of QSym:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R.duality_pairing_matrix(F, 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: F.duality_pairing_matrix(R, 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).ribbon()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> R.duality_pairing_matrix(F, Integer(3))
[1 0 0 0]
[0 1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0]
[0 0 0 1]
>>> F.duality_pairing_matrix(R, Integer(3))
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

The complete basis of NCSF is dual to the monomial basis of QSym:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: S.duality_pairing_matrix(M, 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: M.duality_pairing_matrix(S, 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).complete()
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> S.duality_pairing_matrix(M, Integer(3))
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> M.duality_pairing_matrix(S, Integer(3))
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

The matrix between the ribbon basis of NCSF and the monomial basis of QSym:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: R.duality_pairing_matrix(M, 3)
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
sage: M.duality_pairing_matrix(R, 3)
[ 1  0  0  0]
[-1  1  0  0]
[-1  0  1  0]
[ 1 -1 -1  1]
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).ribbon()
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> R.duality_pairing_matrix(M, Integer(3))
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
>>> M.duality_pairing_matrix(R, Integer(3))
[ 1  0  0  0]
[-1  1  0  0]
[-1  0  1  0]
[ 1 -1 -1  1]
```

The matrix between the complete basis of NCSF and the fundamental basis of QSym:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: S.duality_pairing_matrix(F, 3)
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).complete()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> S.duality_pairing_matrix(F, Integer(3))
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

A base case test:

```
sage: R.duality_pairing_matrix(M,0)
[1]
```

```
>>> from sage.all import *
>>> R.duality_pairing_matrix(M,Integer(0))
[1]
```

one_basis()

Return the empty composition.

OUTPUT: the empty composition

EXAMPLES:

```
sage: L = NonCommutativeSymmetricFunctions(QQ).L()
sage: parent(L)
<class 'sage.combinat.ncsf_qsym.ncsf.NonCommutativeSymmetricFunctions.
↪Elementary_with_category'>
```

(continues on next page)

(continued from previous page)

```
sage: parent(L).one_basis()
[]
```

```
>>> from sage.all import *
>>> L = NonCommutativeSymmetricFunctions(QQ).L()
>>> parent(L)
<class 'sage.combinat.ncsf_qsym.ncsf.NonCommutativeSymmetricFunctions.
↪Elementary_with_category'>
>>> parent(L).one_basis()
[]
```

skew(*x*, *y*, *side='left'*)

Return a function x in self skewed by a function y in the Hopf dual of self.

INPUT:

- x – a non-commutative or quasi-symmetric function; it is an element of self
- y – a quasi-symmetric or non-commutative symmetric function; it is an element of the dual algebra of self
- side – (default: 'left') either 'left' or 'right'

OUTPUT:

- The result of skewing the element x by the Hopf algebra element y (either from the left or from the right, as determined by side), written in the basis self.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: S.skew(S[2,2,2], F[1,1])
S[1, 1, 2] + S[1, 2, 1] + S[2, 1, 1]
sage: S.skew(S[2,2,2], F[2])
S[1, 1, 2] + S[1, 2, 1] + S[2, 1, 1] + 3*S[2, 2]
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).complete()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> S.skew(S[Integer(2),Integer(2),Integer(2)], F[Integer(1),Integer(1)])
S[1, 1, 2] + S[1, 2, 1] + S[2, 1, 1]
>>> S.skew(S[Integer(2),Integer(2),Integer(2)], F[Integer(2)])
S[1, 1, 2] + S[1, 2, 1] + S[2, 1, 1] + 3*S[2, 2]
```

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R.skew(R[2,2,2], F[1,1])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 2*R[2, 2] + R[3, 1] +␣
↪R[4]
sage: R.skew(R[2,2,2], F[2])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 3*R[2, 2] + R[3, 1] +␣
↪R[4]
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).ribbon()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
```

(continues on next page)

(continued from previous page)

```
>>> R.skew(R[Integer(2),Integer(2),Integer(2)], F[Integer(1),Integer(1)])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 2*R[2, 2] + R[3, 1] +␣
↪R[4]
>>> R.skew(R[Integer(2),Integer(2),Integer(2)], F[Integer(2)])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 3*R[2, 2] + R[3, 1] +␣
↪R[4]
```

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M.skew(M[3,2], S[2])
0
sage: M.skew(M[3,2], S[2], side='right')
M[3]
sage: M.skew(M[3,2], S[3])
M[2]
sage: M.skew(M[3,2], S[3], side='right')
0
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> R = NonCommutativeSymmetricFunctions(QQ).R()
>>> M = QuasiSymmetricFunctions(QQ).M()
>>> M.skew(M[Integer(3),Integer(2)], S[Integer(2)])
0
>>> M.skew(M[Integer(3),Integer(2)], S[Integer(2)], side='right')
M[3]
>>> M.skew(M[Integer(3),Integer(2)], S[Integer(3)])
M[2]
>>> M.skew(M[Integer(3),Integer(2)], S[Integer(3)], side='right')
0
```

sum_of_fatter_compositions(*composition*)

Return the sum of all fatter compositions.

INPUT:

- `composition` – a composition

OUTPUT:

- the sum of all basis elements which are indexed by compositions fatter (coarser?) than `composition`.

EXAMPLES:

```
sage: L = NonCommutativeSymmetricFunctions(QQ).L()
sage: L.sum_of_fatter_compositions(Composition([2,1]))
L[2, 1] + L[3]
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: R.sum_of_fatter_compositions(Composition([1,3]))
R[1, 3] + R[4]
```

```
>>> from sage.all import *
>>> L = NonCommutativeSymmetricFunctions(QQ).L()
>>> L.sum_of_fatter_compositions(Composition([Integer(2),Integer(1)]))
```

(continues on next page)

(continued from previous page)

```
L[2, 1] + L[3]
>>> R = NonCommutativeSymmetricFunctions(QQ).R()
>>> R.sum_of_fatter_compositions(Composition([Integer(1),Integer(3)]))
R[1, 3] + R[4]
```

sum_of_finer_compositions(*composition*)

Return the sum of all finer compositions.

INPUT:

- `composition` – a composition

OUTPUT:

- The sum of all basis `self` elements which are indexed by compositions finer than `composition`.

EXAMPLES:

```
sage: L = NonCommutativeSymmetricFunctions(QQ).L()
sage: L.sum_of_finer_compositions(Composition([2,1]))
L[1, 1, 1] + L[2, 1]
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: R.sum_of_finer_compositions(Composition([1,3]))
R[1, 1, 1, 1] + R[1, 1, 2] + R[1, 2, 1] + R[1, 3]
```

```
>>> from sage.all import *
>>> L = NonCommutativeSymmetricFunctions(QQ).L()
>>> L.sum_of_finer_compositions(Composition([Integer(2),Integer(1)]))
L[1, 1, 1] + L[2, 1]
>>> R = NonCommutativeSymmetricFunctions(QQ).R()
>>> R.sum_of_finer_compositions(Composition([Integer(1),Integer(3)]))
R[1, 1, 1, 1] + R[1, 1, 2] + R[1, 2, 1] + R[1, 3]
```

sum_of_partition_rearrangements(*par*)

Return the sum of all basis elements indexed by compositions which can be sorted to obtain a given partition.

INPUT:

- `par` – a partition

OUTPUT:

- The sum of all `self` basis elements indexed by compositions which are permutations of `par` (without multiplicity).

EXAMPLES:

```
sage: NCSF=NonCommutativeSymmetricFunctions(QQ)
sage: elementary = NCSF.elementary()
sage: elementary.sum_of_partition_rearrangements(Partition([2,2,1]))
L[1, 2, 2] + L[2, 1, 2] + L[2, 2, 1]
sage: elementary.sum_of_partition_rearrangements(Partition([3,2,1]))
L[1, 2, 3] + L[1, 3, 2] + L[2, 1, 3] + L[2, 3, 1] + L[3, 1, 2] + L[3, 2,
↪1]
sage: elementary.sum_of_partition_rearrangements(Partition([]))
L[]
```

```
>>> from sage.all import *
>>> NCSF=NonCommutativeSymmetricFunctions(QQ)
```

(continues on next page)

(continued from previous page)

```
>>> elementary = NCSF.elementary()
>>> elementary.sum_of_partition_rearrangements(Partition([Integer(2),
↪Integer(2),Integer(1)]))
L[1, 2, 2] + L[2, 1, 2] + L[2, 2, 1]
>>> elementary.sum_of_partition_rearrangements(Partition([Integer(3),
↪Integer(2),Integer(1)]))
L[1, 2, 3] + L[1, 3, 2] + L[2, 1, 3] + L[2, 3, 1] + L[3, 1, 2] + L[3, 2,␣
↪1]
>>> elementary.sum_of_partition_rearrangements(Partition([]))
L[]
```

super_categories()

class sage.combinat.ncsf_qsym.generic_basis_code.**GradedModulesWithInternalProduct**(*base*, *name=None*)

Bases: `Category_over_base_ring`

Construct the class of modules with internal product. This is used to give an internal product structure to the non-commutative symmetric functions.

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.generic_basis_code import␣
↪GradedModulesWithInternalProduct
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R in GradedModulesWithInternalProduct(QQ)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.generic_basis_code import␣
↪GradedModulesWithInternalProduct
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> R in GradedModulesWithInternalProduct(QQ)
True
```

class ElementMethods

Bases: `object`

internal_product(*other*)

Return the internal product of two non-commutative symmetric functions.

The internal product on the algebra of non-commutative symmetric functions is adjoint to the internal coproduct on the algebra of quasisymmetric functions with respect to the duality pairing between these two algebras. This means, explicitly, that any two non-commutative symmetric functions f and g and any quasi-symmetric function h satisfy

$$\langle f * g, h \rangle = \sum_i \langle f, h'_i \rangle \langle g, h''_i \rangle ,$$

where we write $\Delta^\times(h)$ as $\sum_i h'_i \otimes h''_i$. Here, $f * g$ denotes the internal product of the non-commutative symmetric functions f and g.

If f and g are two homogeneous elements of $NSym$ having distinct degrees, then the internal product $f * g$ is zero.

Explicit formulas can be given for internal products of elements of the complete and the Psi bases. First, the formula for the Complete basis ([NCSF1] Proposition 5.1): If I and J are two compositions of lengths p and q, respectively, then the corresponding Complete homogeneous non-commutative symmetric functions S^I and S^J have internal product

$$S^I * S^J = \sum S^{\mathrm{comp}\, M},$$

where the sum ranges over all $p \times q$-matrices $M \in \mathbf{N}^{p \times q}$ (with nonnegative integers as entries) whose row sum vector is I (that is, the sum of the entries of the r-th row is the r-th part of I for all r) and whose column sum vector is J (that is, the sum of all entries of the s-th row is the s-th part of J for all s). Here, for any $M \in \mathbf{N}^{p \times q}$, we denote by $\mathrm{comp}\, M$ the composition obtained by reading the entries of the matrix M in the usual order (row by row, proceeding left to right in each row, traversing the rows from top to bottom).

The formula on the Psi basis ([NCSF2] Lemma 3.10) is more complicated. Let I and J be two compositions of lengths p and q, respectively, having the same size $|I| = |J|$. We denote by Ψ^K the element of the Psi basis corresponding to any composition K.

- If $p > q$, then $\Psi^I * \Psi^J$ is plainly 0.
- Assume that $p = q$. Let $\widetilde{\delta}_{I,J}$ denote the integer 1 if the compositions I and J are permutations of each other, and the integer 0 otherwise. For every positive integer i, let m_i denote the number of parts of I equal to i. Then, $\Psi^I * \Psi^J$ equals $\widetilde{\delta}_{I,J} \prod_{i>0} i^{m_i} m_i! \Psi^I$.
- Now assume that $p < q$. Write the composition I as $I = (i_1, i_2, \ldots, i_p)$. For every nonempty composition $K = (k_1, k_2, \ldots, k_s)$, denote by Γ_K the non-commutative symmetric function $k_1[\ldots[[\Psi_{k_1}, \Psi_{k_2}], \Psi_{k_3}], \ldots \Psi_{k_s}]$. For any subset A of $\{1, 2, \ldots, q\}$, let J_A be the composition obtained from J by removing the r-th parts for all $r \notin A$ (while keeping the r-th parts for all $r \in A$ in order). Then, $\Psi^I * \Psi^J$ equals the sum of $\Gamma_{J_{K_1}} \Gamma_{J_{K_2}} \cdots \Gamma_{J_{K_p}}$ over all ordered set partitions $(K_1, K_2, \ldots, K_p)$ of $\{1, 2, \ldots, q\}$ into p parts such that each $1 \leq k \leq p$ satisfies $|J_{K_k}| = i_k$. (See `OrderedSetPartition()` for the meaning of "ordered set partition".)

Aliases for `internal_product()` are `itensor()` and `kronecker_product()`.

INPUT:

- `other` – another non-commutative symmetric function

OUTPUT:

- The result of taking the internal product of `self` with `other`.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
sage: x.internal_product(S[2])
3*S[1, 1]
sage: x.internal_product(S[1])
2*S[1]
sage: S[1,2].internal_product(S[1,2])
S[1, 1, 1] + S[1, 2]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
>>> x.internal_product(S[Integer(2)])
3*S[1, 1]
```

(continues on next page)

(continued from previous page)

```
>>> x.internal_product(S[Integer(1)])
2*S[1]
>>> S[Integer(1),Integer(2)].internal_product(S[Integer(1),Integer(2)])
S[1, 1, 1] + S[1, 2]
```

Let us check the duality between the inner product and the inner coproduct in degree 4:

```
sage: M = QuasiSymmetricFunctions(FiniteField(29)).M()
sage: S = NonCommutativeSymmetricFunctions(FiniteField(29)).S()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \
↪right> \left< g, h''_i \right>
....:     result = h.base_ring().zero()
....:     h_parent = h.parent()
....:     for partition_pair, coeff in h.internal_coproduct().monomial_
↪coefficients().items():
....:         result += coeff * f.duality_pairing(h_parent[partition_
↪pair[0]]) * g.duality_pairing(h_parent[partition_pair[1]])
....:     return result
sage: def testall(n):
....:     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].
↪itensor(S[v])).duality_pairing(M[w])
....:                               for w in Compositions(n) )
....:                          for v in Compositions(n) )
....:                     for u in Compositions(n) )
sage: testall(2)
True
sage: testall(3)  # long time
True
sage: testall(4)  # not tested, too long
True
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(FiniteField(Integer(29))).M()
>>> S = NonCommutativeSymmetricFunctions(FiniteField(Integer(29))).S()
>>> def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right>␣
↪\left< g, h''_i \right>
...     result = h.base_ring().zero()
...     h_parent = h.parent()
...     for partition_pair, coeff in h.internal_coproduct().monomial_
↪coefficients().items():
...         result += coeff * f.duality_pairing(h_parent[partition_
↪pair[Integer(0)]]) * g.duality_pairing(h_parent[partition_
↪pair[Integer(1)]])
...     return result
>>> def testall(n):
...     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].
↪itensor(S[v])).duality_pairing(M[w])
...                               for w in Compositions(n) )
...                          for v in Compositions(n) )
...                     for u in Compositions(n) )
>>> testall(Integer(2))
True
```

(continues on next page)

(continued from previous page)

```
>>> testall(Integer(3))  # long time
True
>>> testall(Integer(4))  # not tested, too long
True
```

The internal product on the algebra of non-commutative symmetric functions commutes with the canonical commutative projection on the symmetric functions:

```
sage: S = NonCommutativeSymmetricFunctions(ZZ).S()
sage: e = SymmetricFunctions(ZZ).e()
sage: def int_pr_of_S_in_e(I, J):
....:     return (S[I].internal_product(S[J])).to_symmetric_function()
sage: all( all( int_pr_of_S_in_e(I, J)
....:           == S[I].to_symmetric_function().internal_product(S[J].to_
↪symmetric_function())
....:           for I in Compositions(3) )
....:      for J in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(ZZ).S()
>>> e = SymmetricFunctions(ZZ).e()
>>> def int_pr_of_S_in_e(I, J):
...     return (S[I].internal_product(S[J])).to_symmetric_function()
>>> all( all( int_pr_of_S_in_e(I, J)
...           == S[I].to_symmetric_function().internal_product(S[J].to_
↪symmetric_function())
...           for I in Compositions(Integer(3)) )
...      for J in Compositions(Integer(3)) )
True
```

itensor(*other*)

Return the internal product of two non-commutative symmetric functions.

The internal product on the algebra of non-commutative symmetric functions is adjoint to the internal coproduct on the algebra of quasisymmetric functions with respect to the duality pairing between these two algebras. This means, explicitly, that any two non-commutative symmetric functions f and g and any quasi-symmetric function h satisfy

$$\langle f * g, h \rangle = \sum_i \langle f, h'_i \rangle \langle g, h''_i \rangle ,$$

where we write $\Delta^{\times}(h)$ as $\sum_i h'_i \otimes h''_i$. Here, $f * g$ denotes the internal product of the non-commutative symmetric functions f and g.

If f and g are two homogeneous elements of $NSym$ having distinct degrees, then the internal product $f * g$ is zero.

Explicit formulas can be given for internal products of elements of the complete and the Psi bases. First, the formula for the Complete basis ([NCSF1] Proposition 5.1): If I and J are two compositions of lengths p and q, respectively, then the corresponding Complete homogeneous non-commutative symmetric functions S^I and S^J have internal product

$$S^I * S^J = \sum S^{\mathrm{comp}\, M},$$

where the sum ranges over all $p \times q$-matrices $M \in \mathbf{N}^{p\times q}$ (with nonnegative integers as entries) whose row sum vector is I (that is, the sum of the entries of the r-th row is the r-th part of I for all r) and whose column sum vector is J (that is, the sum of all entries of the s-th row is the s-th part of J for all s). Here, for any $M \in \mathbf{N}^{p\times q}$, we denote by comp M the composition obtained by reading the entries of the matrix M in the usual order (row by row, proceeding left to right in each row, traversing the rows from top to bottom).

The formula on the Psi basis ([NCSF2] Lemma 3.10) is more complicated. Let I and J be two compositions of lengths p and q, respectively, having the same size $|I| = |J|$. We denote by Ψ^K the element of the Psi basis corresponding to any composition K.

- If $p > q$, then $\Psi^I * \Psi^J$ is plainly 0.
- Assume that $p = q$. Let $\widetilde{\delta}_{I,J}$ denote the integer 1 if the compositions I and J are permutations of each other, and the integer 0 otherwise. For every positive integer i, let m_i denote the number of parts of I equal to i. Then, $\Psi^I * \Psi^J$ equals $\widetilde{\delta}_{I,J} \prod_{i>0} i^{m_i} m_i! \Psi^I$.
- Now assume that $p < q$. Write the composition I as $I = (i_1, i_2, \ldots, i_p)$. For every nonempty composition $K = (k_1, k_2, \ldots, k_s)$, denote by Γ_K the non-commutative symmetric function $k_1[\ldots[[\Psi_{k_1}, \Psi_{k_2}], \Psi_{k_3}], \ldots \Psi_{k_s}]$. For any subset A of $\{1, 2, \ldots, q\}$, let J_A be the composition obtained from J by removing the r-th parts for all $r \notin A$ (while keeping the r-th parts for all $r \in A$ in order). Then, $\Psi^I * \Psi^J$ equals the sum of $\Gamma_{J_{K_1}} \Gamma_{J_{K_2}} \cdots \Gamma_{J_{K_p}}$ over all ordered set partitions $(K_1, K_2, \ldots, K_p)$ of $\{1, 2, \ldots, q\}$ into p parts such that each $1 \leq k \leq p$ satisfies $|J_{K_k}| = i_k$. (See `OrderedSetPartition()` for the meaning of "ordered set partition".)

Aliases for `internal_product()` are `itensor()` and `kronecker_product()`.

INPUT:

- `other` – another non-commutative symmetric function

OUTPUT:

- The result of taking the internal product of `self` with `other`.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
sage: x.internal_product(S[2])
3*S[1, 1]
sage: x.internal_product(S[1])
2*S[1]
sage: S[1,2].internal_product(S[1,2])
S[1, 1, 1] + S[1, 2]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
>>> x.internal_product(S[Integer(2)])
3*S[1, 1]
>>> x.internal_product(S[Integer(1)])
2*S[1]
>>> S[Integer(1),Integer(2)].internal_product(S[Integer(1),Integer(2)])
S[1, 1, 1] + S[1, 2]
```

Let us check the duality between the inner product and the inner coproduct in degree 4:

```
sage: M = QuasiSymmetricFunctions(FiniteField(29)).M()
sage: S = NonCommutativeSymmetricFunctions(FiniteField(29)).S()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \
↪right> \left< g, h''_i \right>
....:     result = h.base_ring().zero()
....:     h_parent = h.parent()
....:     for partition_pair, coeff in h.internal_coproduct().monomial_
↪coefficients().items():
....:         result += coeff * f.duality_pairing(h_parent[partition_
↪pair[0]]) * g.duality_pairing(h_parent[partition_pair[1]])
....:     return result
sage: def testall(n):
....:     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].
↪itensor(S[v])).duality_pairing(M[w])
....:                               for w in Compositions(n) )
....:                         for v in Compositions(n) )
....:                   for u in Compositions(n) )
sage: testall(2)
True
sage: testall(3)  # long time
True
sage: testall(4)  # not tested, too long
True
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(FiniteField(Integer(29))).M()
>>> S = NonCommutativeSymmetricFunctions(FiniteField(Integer(29))).S()
>>> def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right>␣
↪\left< g, h''_i \right>
...     result = h.base_ring().zero()
...     h_parent = h.parent()
...     for partition_pair, coeff in h.internal_coproduct().monomial_
↪coefficients().items():
...         result += coeff * f.duality_pairing(h_parent[partition_
↪pair[Integer(0)]]) * g.duality_pairing(h_parent[partition_
↪pair[Integer(1)]])
...     return result
>>> def testall(n):
...     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].
↪itensor(S[v])).duality_pairing(M[w])
...                               for w in Compositions(n) )
...                         for v in Compositions(n) )
...                   for u in Compositions(n) )
>>> testall(Integer(2))
True
>>> testall(Integer(3))  # long time
True
>>> testall(Integer(4))  # not tested, too long
True
```

The internal product on the algebra of non-commutative symmetric functions commutes with the canonical commutative projection on the symmetric functions:

```
sage: S = NonCommutativeSymmetricFunctions(ZZ).S()
sage: e = SymmetricFunctions(ZZ).e()
sage: def int_pr_of_S_in_e(I, J):
....:     return (S[I].internal_product(S[J])).to_symmetric_function()
sage: all( all( int_pr_of_S_in_e(I, J)
....:             == S[I].to_symmetric_function().internal_product(S[J].to_
↪symmetric_function())
....:             for I in Compositions(3) )
....:       for J in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(ZZ).S()
>>> e = SymmetricFunctions(ZZ).e()
>>> def int_pr_of_S_in_e(I, J):
...     return (S[I].internal_product(S[J])).to_symmetric_function()
>>> all( all( int_pr_of_S_in_e(I, J)
...             == S[I].to_symmetric_function().internal_product(S[J].to_
↪symmetric_function())
...             for I in Compositions(Integer(3)) )
...       for J in Compositions(Integer(3)) )
True
```

kronecker_product(*other*)

Return the internal product of two non-commutative symmetric functions.

The internal product on the algebra of non-commutative symmetric functions is adjoint to the internal coproduct on the algebra of quasisymmetric functions with respect to the duality pairing between these two algebras. This means, explicitly, that any two non-commutative symmetric functions f and g and any quasi-symmetric function h satisfy

$$\langle f * g, h \rangle = \sum_i \langle f, h_i' \rangle \langle g, h_i'' \rangle,$$

where we write $\Delta^{\times}(h)$ as $\sum_i h_i' \otimes h_i''$. Here, $f * g$ denotes the internal product of the non-commutative symmetric functions f and g.

If f and g are two homogeneous elements of $NSym$ having distinct degrees, then the internal product $f * g$ is zero.

Explicit formulas can be given for internal products of elements of the complete and the Psi bases. First, the formula for the Complete basis ([NCSF1] Proposition 5.1): If I and J are two compositions of lengths p and q, respectively, then the corresponding Complete homogeneous non-commutative symmetric functions S^I and S^J have internal product

$$S^I * S^J = \sum S^{\operatorname{comp} M},$$

where the sum ranges over all $p \times q$-matrices $M \in \mathbf{N}^{p \times q}$ (with nonnegative integers as entries) whose row sum vector is I (that is, the sum of the entries of the r-th row is the r-th part of I for all r) and whose column sum vector is J (that is, the sum of all entries of the s-th row is the s-th part of J for all s). Here, for any $M \in \mathbf{N}^{p \times q}$, we denote by $\operatorname{comp} M$ the composition obtained by reading the entries of the matrix M in the usual order (row by row, proceeding left to right in each row, traversing the rows from top to bottom).

The formula on the Psi basis ([NCSF2] Lemma 3.10) is more complicated. Let I and J be two compositions of lengths p and q, respectively, having the same size $|I| = |J|$. We denote by Ψ^K the element of the Psi basis corresponding to any composition K.

- If $p > q$, then $\Psi^I * \Psi^J$ is plainly 0.
- Assume that $p = q$. Let $\widetilde{\delta}_{I,J}$ denote the integer 1 if the compositions I and J are permutations of each other, and the integer 0 otherwise. For every positive integer i, let m_i denote the number of parts of I equal to i. Then, $\Psi^I * \Psi^J$ equals $\widetilde{\delta}_{I,J} \prod_{i>0} i^{m_i} m_i! \Psi^I$.
- Now assume that $p < q$. Write the composition I as $I = (i_1, i_2, \ldots, i_p)$. For every nonempty composition $K = (k_1, k_2, \ldots, k_s)$, denote by Γ_K the non-commutative symmetric function $k_1[\ldots[[\Psi_{k_1}, \Psi_{k_2}], \Psi_{k_3}], \ldots \Psi_{k_s}]$. For any subset A of $\{1, 2, \ldots, q\}$, let J_A be the composition obtained from J by removing the r-th parts for all $r \notin A$ (while keeping the r-th parts for all $r \in A$ in order). Then, $\Psi^I * \Psi^J$ equals the sum of $\Gamma_{J_{K_1}} \Gamma_{J_{K_2}} \cdots \Gamma_{J_{K_p}}$ over all ordered set partitions $(K_1, K_2, \ldots, K_p)$ of $\{1, 2, \ldots, q\}$ into p parts such that each $1 \leq k \leq p$ satisfies $|J_{K_k}| = i_k$. (See `OrderedSetPartition()` for the meaning of "ordered set partition".)

Aliases for `internal_product()` are `itensor()` and `kronecker_product()`.

INPUT:

- `other` – another non-commutative symmetric function

OUTPUT:

- The result of taking the internal product of `self` with `other`.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
sage: x.internal_product(S[2])
3*S[1, 1]
sage: x.internal_product(S[1])
2*S[1]
sage: S[1,2].internal_product(S[1,2])
S[1, 1, 1] + S[1, 2]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
>>> x.internal_product(S[Integer(2)])
3*S[1, 1]
>>> x.internal_product(S[Integer(1)])
2*S[1]
>>> S[Integer(1),Integer(2)].internal_product(S[Integer(1),Integer(2)])
S[1, 1, 1] + S[1, 2]
```

Let us check the duality between the inner product and the inner coproduct in degree 4:

```
sage: M = QuasiSymmetricFunctions(FiniteField(29)).M()
sage: S = NonCommutativeSymmetricFunctions(FiniteField(29)).S()
sage: def tensor_incopr(f, g, h):   # computes \sum_i \left< f, h'_i \
↪right> \left< g, h''_i \right>
....:     result = h.base_ring().zero()
....:     h_parent = h.parent()
....:     for partition_pair, coeff in h.internal_coproduct().monomial_
↪coefficients().items():
....:         result += coeff * f.duality_pairing(h_parent[partition_
↪pair[0]]) * g.duality_pairing(h_parent[partition_pair[1]])
```

(continues on next page)

(continued from previous page)

```
....:     return result
sage: def testall(n):
....:     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].
↪itensor(S[v])).duality_pairing(M[w])
....:                             for w in Compositions(n) )
....:                       for v in Compositions(n) )
....:                 for u in Compositions(n) )
sage: testall(2)
True
sage: testall(3)  # long time
True
sage: testall(4)  # not tested, too long
True
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(FiniteField(Integer(29))).M()
>>> S = NonCommutativeSymmetricFunctions(FiniteField(Integer(29))).S()
>>> def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right>␣
↪\left< g, h''_i \right>
...     result = h.base_ring().zero()
...     h_parent = h.parent()
...     for partition_pair, coeff in h.internal_coproduct().monomial_
↪coefficients().items():
...         result += coeff * f.duality_pairing(h_parent[partition_
↪pair[Integer(0)]]) * g.duality_pairing(h_parent[partition_
↪pair[Integer(1)]])
...     return result
>>> def testall(n):
...     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].
↪itensor(S[v])).duality_pairing(M[w])
...                             for w in Compositions(n) )
...                       for v in Compositions(n) )
...                 for u in Compositions(n) )
>>> testall(Integer(2))
True
>>> testall(Integer(3))  # long time
True
>>> testall(Integer(4))  # not tested, too long
True
```

The internal product on the algebra of non-commutative symmetric functions commutes with the canonical commutative projection on the symmetric functions:

```
sage: S = NonCommutativeSymmetricFunctions(ZZ).S()
sage: e = SymmetricFunctions(ZZ).e()
sage: def int_pr_of_S_in_e(I, J):
....:     return (S[I].internal_product(S[J])).to_symmetric_function()
sage: all( all( int_pr_of_S_in_e(I, J)
....:           == S[I].to_symmetric_function().internal_product(S[J].to_
↪symmetric_function())
....:           for I in Compositions(3) )
....:      for J in Compositions(3) )
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(ZZ).S()
>>> e = SymmetricFunctions(ZZ).e()
>>> def int_pr_of_S_in_e(I, J):
...     return (S[I].internal_product(S[J])).to_symmetric_function()
>>> all( all( int_pr_of_S_in_e(I, J)
...           == S[I].to_symmetric_function().internal_product(S[J].to_
↪symmetric_function())
...           for I in Compositions(Integer(3)) )
...      for J in Compositions(Integer(3)) )
True
```

class ParentMethods

Bases: `object`

internal_product()

The bilinear product inherited from the isomorphism with the descent algebra.

This is constructed by extending the method *internal_product_on_basis()* bilinearly, if available, or using the method *internal_product_by_coercion()*.

OUTPUT:

- The internal product map of the algebra the non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Complete basis
sage: S.internal_product(S[2,2], S[1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product(S[2,2], S[1,2])
0
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Complete basis
>>> S.internal_product(S[Integer(2),Integer(2)], S[Integer(1),Integer(2),
↪Integer(1)])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
>>> S.internal_product(S[Integer(2),Integer(2)], S[Integer(1),Integer(2)])
0
```

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R.internal_product
```

(continues on next page)

(continued from previous page)

```
<bound method ....internal_product_by_coercion ...>
sage: R.internal_product_by_coercion(R[1, 1], R[1,1])
R[2]
sage: R.internal_product(R[2,2], R[1,2])
0
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> R.internal_product
<bound method ....internal_product_by_coercion ...>
>>> R.internal_product_by_coercion(R[Integer(1), Integer(1)],␣
↪R[Integer(1),Integer(1)])
R[2]
>>> R.internal_product(R[Integer(2),Integer(2)], R[Integer(1),Integer(2)])
0
```

internal_product_on_basis(*I*, *J*)

The internal product of the two basis elements indexed by `I` and `J` (optional)

INPUT:

- `I`, `J` – compositions indexing two elements of the basis of self

Returns the internal product of the corresponding basis elements. If this method is implemented, the internal product is defined from it by linearity.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product_on_basis([2,2], [1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product_on_basis([2,2], [2,1])
0
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> S.internal_product_on_basis([Integer(2),Integer(2)], [Integer(1),
↪Integer(2),Integer(1)])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
>>> S.internal_product_on_basis([Integer(2),Integer(2)], [Integer(2),
↪Integer(1)])
0
```

itensor()

The bilinear product inherited from the isomorphism with the descent algebra.

This is constructed by extending the method *internal_product_on_basis()* bilinearly, if available, or using the method *internal_product_by_coercion()*.

OUTPUT:

- The internal product map of the algebra the non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Complete basis
sage: S.internal_product(S[2,2], S[1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product(S[2,2], S[1,2])
0
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Complete basis
>>> S.internal_product(S[Integer(2),Integer(2)], S[Integer(1),Integer(2),
↪Integer(1)])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
>>> S.internal_product(S[Integer(2),Integer(2)], S[Integer(1),Integer(2)])
0
```

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R.internal_product
<bound method ....internal_product_by_coercion ...>
sage: R.internal_product_by_coercion(R[1, 1], R[1,1])
R[2]
sage: R.internal_product(R[2,2], R[1,2])
0
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> R.internal_product
<bound method ....internal_product_by_coercion ...>
>>> R.internal_product_by_coercion(R[Integer(1), Integer(1)],␣
↪R[Integer(1),Integer(1)])
R[2]
>>> R.internal_product(R[Integer(2),Integer(2)], R[Integer(1),Integer(2)])
0
```

kronecker_product()

The bilinear product inherited from the isomorphism with the descent algebra.

This is constructed by extending the method *internal_product_on_basis()* bilinearly, if available, or using the method *internal_product_by_coercion()*.

OUTPUT:

- The internal product map of the algebra the non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Complete basis
sage: S.internal_product(S[2,2], S[1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product(S[2,2], S[1,2])
0
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Complete basis
>>> S.internal_product(S[Integer(2),Integer(2)], S[Integer(1),Integer(2),
↪Integer(1)])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
>>> S.internal_product(S[Integer(2),Integer(2)], S[Integer(1),Integer(2)])
0
```

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R.internal_product
<bound method ....internal_product_by_coercion ...>
sage: R.internal_product_by_coercion(R[1, 1], R[1,1])
R[2]
sage: R.internal_product(R[2,2], R[1,2])
0
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> R.internal_product
<bound method ....internal_product_by_coercion ...>
>>> R.internal_product_by_coercion(R[Integer(1), Integer(1)],␣
↪R[Integer(1),Integer(1)])
R[2]
>>> R.internal_product(R[Integer(2),Integer(2)], R[Integer(1),Integer(2)])
0
```

class Realizations(*category*, **args*)

Bases: `RealizationsCategory`

class ParentMethods

Bases: `object`

internal_product_by_coercion(*left*, *right*)

Internal product of `left` and `right`.

This is a default implementation that computes the internal product in the realization specified by `self.realization_of().a_realization()`.

INPUT:

- `left` – an element of the non-commutative symmetric functions
- `right` – an element of the non-commutative symmetric functions

OUTPUT: the internal product of `left` and `right`

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.internal_product_by_coercion(S[2,1], S[3])
S[2, 1]
sage: S.internal_product_by_coercion(S[2,1], S[4])
0
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S.internal_product_by_coercion(S[Integer(2),Integer(1)],␣
↪S[Integer(3)])
S[2, 1]
>>> S.internal_product_by_coercion(S[Integer(2),Integer(1)],␣
↪S[Integer(4)])
0
```

super_categories()

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.generic_basis_code import␣
↪GradedModulesWithInternalProduct
sage: GradedModulesWithInternalProduct(ZZ).super_categories()
[Category of graded modules over Integer Ring]
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsf_qsym.generic_basis_code import␣
↪GradedModulesWithInternalProduct
>>> GradedModulesWithInternalProduct(ZZ).super_categories()
[Category of graded modules over Integer Ring]
```

5.1.144 Non-Commutative Symmetric Functions

class sage.combinat.ncsf_qsym.ncsf.**NonCommutativeSymmetricFunctions**(R)

Bases: `UniqueRepresentation`, `Parent`

The abstract algebra of non-commutative symmetric functions.

We construct the abstract algebra of non-commutative symmetric functions over the rational numbers:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: NCSF
Non-Commutative Symmetric Functions over the Rational Field
sage: S = NCSF.complete()
sage: R = NCSF.ribbon()
sage: S[2,1]*R[1,2]
S[2, 1, 1, 2] - S[2, 1, 3]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> NCSF
Non-Commutative Symmetric Functions over the Rational Field
>>> S = NCSF.complete()
>>> R = NCSF.ribbon()
>>> S[Integer(2),Integer(1)]*R[Integer(1),Integer(2)]
S[2, 1, 1, 2] - S[2, 1, 3]
```

NCSF is the unique free (non-commutative!) graded connected algebra with one generator in each degree:

```
sage: NCSF.category()
Join of Category of Hopf algebras over Rational Field
    and Category of graded algebras over Rational Field
    and Category of monoids with realizations
    and Category of graded coalgebras over Rational Field
    and Category of coalgebras over Rational Field with realizations
    and Category of cocommutative coalgebras over Rational Field

sage: [S[i].degree() for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> NCSF.category()
Join of Category of Hopf algebras over Rational Field
    and Category of graded algebras over Rational Field
    and Category of monoids with realizations
    and Category of graded coalgebras over Rational Field
    and Category of coalgebras over Rational Field with realizations
    and Category of cocommutative coalgebras over Rational Field

>>> [S[i].degree() for i in range(Integer(10))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We use the Sage standard renaming idiom to get shorter outputs:

```
sage: NCSF.rename('NCSF')
sage: NCSF
NCSF
```

```
>>> from sage.all import *
>>> NCSF.rename('NCSF')
>>> NCSF
NCSF
```

NCSF has many representations as a concrete algebra. Each of them has a distinguished basis, and its elements are expanded in this basis. Here is the Ψ (`Psi`) representation:

```
sage: Psi = NCSF.Psi()
sage: Psi
NCSF in the Psi basis
```

```
>>> from sage.all import *
>>> Psi = NCSF.Psi()
>>> Psi
NCSF in the Psi basis
```

Elements of `Psi` are linear combinations of basis elements indexed by compositions:

```
sage: Psi.an_element()
2*Psi[] + 2*Psi[1] + 3*Psi[1, 1]
```

```
>>> from sage.all import *
>>> Psi.an_element()
2*Psi[] + 2*Psi[1] + 3*Psi[1, 1]
```

The basis itself is accessible through:

```
sage: Psi.basis()
Lazy family (Term map from Compositions of nonnegative integers...
sage: Psi.basis().keys()
Compositions of nonnegative integers
```

```
>>> from sage.all import *
>>> Psi.basis()
Lazy family (Term map from Compositions of nonnegative integers...
>>> Psi.basis().keys()
Compositions of nonnegative integers
```

To construct an element one can therefore do:

```
sage: Psi.basis()[Composition([2,1,3])]
Psi[2, 1, 3]
```

```
>>> from sage.all import *
>>> Psi.basis()[Composition([Integer(2),Integer(1),Integer(3)])]
Psi[2, 1, 3]
```

As this is rather cumbersome, the following abuses of notation are allowed:

```
sage: Psi[Composition([2, 1, 3])]
Psi[2, 1, 3]
sage: Psi[[2, 1, 3]]
Psi[2, 1, 3]
sage: Psi[2, 1, 3]
Psi[2, 1, 3]
```

```
>>> from sage.all import *
>>> Psi[Composition([Integer(2), Integer(1), Integer(3)])]
Psi[2, 1, 3]
>>> Psi[[Integer(2), Integer(1), Integer(3)]]
Psi[2, 1, 3]
>>> Psi[Integer(2), Integer(1), Integer(3)]
Psi[2, 1, 3]
```

or even:

```
sage: Psi[(i for i in [2, 1, 3])]
Psi[2, 1, 3]
```

```
>>> from sage.all import *
>>> Psi[(i for i in [Integer(2), Integer(1), Integer(3)])]
Psi[2, 1, 3]
```

Unfortunately, due to a limitation in Python syntax, one cannot use:

```
sage: Psi[]       # not implemented
```

```
>>> from sage.all import *
>>> Psi[]       # not implemented
```

Instead, you can use:

```
sage: Psi[[]]
Psi[]
```

```
>>> from sage.all import *
>>> Psi[[]]
Psi[]
```

Now, we can construct linear combinations of basis elements:

```
sage: Psi[2,1,3] + 2 * (Psi[4] + Psi[2,1])
2*Psi[2, 1] + Psi[2, 1, 3] + 2*Psi[4]
```

```
>>> from sage.all import *
>>> Psi[Integer(2),Integer(1),Integer(3)] + Integer(2) * (Psi[Integer(4)] +␣
↪Psi[Integer(2),Integer(1)])
2*Psi[2, 1] + Psi[2, 1, 3] + 2*Psi[4]
```

Algebra structure

To start with, `Psi` is a graded algebra, the grading being induced by the size of compositions. The one is the basis element indexed by the empty composition:

```
sage: Psi.one()
Psi[]
sage: S.one()
S[]
sage: R.one()
R[]
```

```
>>> from sage.all import *
>>> Psi.one()
Psi[]
>>> S.one()
S[]
```

(continues on next page)

(continued from previous page)

```
>>> R.one()
R[]
```

As we have seen above, the `Psi` basis is multiplicative; that is multiplication is induced by linearity from the concatenation of compositions:

```
sage: Psi[1,3] * Psi[2,1]
Psi[1, 3, 2, 1]
sage: (Psi.one() + 2 * Psi[1,3]) * Psi[2, 4]
2*Psi[1, 3, 2, 4] + Psi[2, 4]
```

```
>>> from sage.all import *
>>> Psi[Integer(1),Integer(3)] * Psi[Integer(2),Integer(1)]
Psi[1, 3, 2, 1]
>>> (Psi.one() + Integer(2) * Psi[Integer(1),Integer(3)]) * Psi[Integer(2),
↪Integer(4)]
2*Psi[1, 3, 2, 4] + Psi[2, 4]
```

Hopf algebra structure

`Psi` is further endowed with a coalgebra structure. The coproduct is an algebra morphism, and therefore determined by its values on the generators; those are primitive:

```
sage: Psi[1].coproduct()
Psi[] # Psi[1] + Psi[1] # Psi[]
sage: Psi[2].coproduct()
Psi[] # Psi[2] + Psi[2] # Psi[]
```

```
>>> from sage.all import *
>>> Psi[Integer(1)].coproduct()
Psi[] # Psi[1] + Psi[1] # Psi[]
>>> Psi[Integer(2)].coproduct()
Psi[] # Psi[2] + Psi[2] # Psi[]
```

The coproduct, being cocommutative on the generators, is cocommutative everywhere:

```
sage: Psi[1,2].coproduct()
Psi[] # Psi[1, 2] + Psi[1] # Psi[2] + Psi[1, 2] # Psi[] + Psi[2] # Psi[1]
```

```
>>> from sage.all import *
>>> Psi[Integer(1),Integer(2)].coproduct()
Psi[] # Psi[1, 2] + Psi[1] # Psi[2] + Psi[1, 2] # Psi[] + Psi[2] # Psi[1]
```

The algebra and coalgebra structures on `Psi` combine to form a bialgebra structure, which cooperates with the grading to form a connected graded bialgebra. Thus, as any connected graded bialgebra, `Psi` is a Hopf algebra. Over `QQ` (or any other **Q**-algebra), this Hopf algebra `Psi` is isomorphic to the universal enveloping algebra of its space of primitive elements. Here, the primitives form a countably generated free Lie algebra, so `Psi` is isomorphic to the tensor algebra on a countably infinite dimensional vector space.

The antipode is an anti-algebra morphism; in the `Psi` basis, it sends the generators to their opposites and changes their sign if they are of odd degree:

```
sage: Psi[3].antipode()
-Psi[3]
sage: Psi[1,3,2].antipode()
-Psi[2, 3, 1]
sage: Psi[1,3,2].coproduct().apply_multilinear_morphism(lambda be,ga:␣
↪Psi(be)*Psi(ga).antipode())
0
```

```
>>> from sage.all import *
>>> Psi[Integer(3)].antipode()
-Psi[3]
>>> Psi[Integer(1),Integer(3),Integer(2)].antipode()
-Psi[2, 3, 1]
>>> Psi[Integer(1),Integer(3),Integer(2)].coproduct().apply_multilinear_
↪morphism(lambda be,ga: Psi(be)*Psi(ga).antipode())
0
```

The counit is defined by sending all elements of positive degree to zero:

```
sage: S[3].degree(), S[3,1,2].degree(), S.one().degree()
(3, 6, 0)
sage: S[3].counit()
0
sage: S[3,1,2].counit()
0
sage: S.one().counit()
1
sage: (S[3] - 2*S[3,1,2] + 7).counit()
7
sage: (R[3] - 2*R[3,1,2] + 7).counit()
7
```

```
>>> from sage.all import *
>>> S[Integer(3)].degree(), S[Integer(3),Integer(1),Integer(2)].degree(), S.one().
↪degree()
(3, 6, 0)
>>> S[Integer(3)].counit()
0
>>> S[Integer(3),Integer(1),Integer(2)].counit()
0
>>> S.one().counit()
1
>>> (S[Integer(3)] - Integer(2)*S[Integer(3),Integer(1),Integer(2)] + Integer(7)).
↪counit()
7
>>> (R[Integer(3)] - Integer(2)*R[Integer(3),Integer(1),Integer(2)] + Integer(7)).
↪counit()
7
```

It is possible to change the prefix used to display the basis elements using the method `print_options()`. Say that for instance one wanted to display the *`Complete`* basis as having a prefix `H` instead of the default `S`:

```
sage: H = NCSF.complete()
sage: H.an_element()
2*S[] + 2*S[1] + 3*S[1, 1]
sage: H.print_options(prefix='H')
sage: H.an_element()
2*H[] + 2*H[1] + 3*H[1, 1]
sage: H.print_options(prefix='S') #restore to 'S'
```

```
>>> from sage.all import *
>>> H = NCSF.complete()
>>> H.an_element()
2*S[] + 2*S[1] + 3*S[1, 1]
>>> H.print_options(prefix='H')
>>> H.an_element()
2*H[] + 2*H[1] + 3*H[1, 1]
>>> H.print_options(prefix='S') #restore to 'S'
```

Concrete representations

NCSF admits the concrete realizations defined in [NCSF1]:

```
sage: Phi        = NCSF.Phi()
sage: Psi        = NCSF.Psi()
sage: ribbon     = NCSF.ribbon()
sage: complete   = NCSF.complete()
sage: elementary = NCSF.elementary()
```

```
>>> from sage.all import *
>>> Phi        = NCSF.Phi()
>>> Psi        = NCSF.Psi()
>>> ribbon     = NCSF.ribbon()
>>> complete   = NCSF.complete()
>>> elementary = NCSF.elementary()
```

To change from one basis to another, one simply does:

```
sage: Phi(Psi[1])
Phi[1]
sage: Phi(Psi[3])
-1/4*Phi[1, 2] + 1/4*Phi[2, 1] + Phi[3]
```

```
>>> from sage.all import *
>>> Phi(Psi[Integer(1)])
Phi[1]
>>> Phi(Psi[Integer(3)])
-1/4*Phi[1, 2] + 1/4*Phi[2, 1] + Phi[3]
```

In general, one can mix up different bases in computations:

```
sage: Phi[1] * Psi[1]
Phi[1, 1]
```

```
>>> from sage.all import *
>>> Phi[Integer(1)] * Psi[Integer(1)]
Phi[1, 1]
```

Some of the changes of basis are easy to guess:

```
sage: ribbon(complete[1,3,2])
R[1, 3, 2] + R[1, 5] + R[4, 2] + R[6]
```

```
>>> from sage.all import *
>>> ribbon(complete[Integer(1),Integer(3),Integer(2)])
R[1, 3, 2] + R[1, 5] + R[4, 2] + R[6]
```

This is the sum of all fatter compositions. Using the usual Möbius function for the boolean lattice, the inverse change of basis is given by the alternating sum of all fatter compositions:

```
sage: complete(ribbon[1,3,2])
S[1, 3, 2] - S[1, 5] - S[4, 2] + S[6]
```

```
>>> from sage.all import *
>>> complete(ribbon[Integer(1),Integer(3),Integer(2)])
S[1, 3, 2] - S[1, 5] - S[4, 2] + S[6]
```

The analogue of the elementary basis is the sum over all finer compositions than the 'complement' of the composition in the ribbon basis:

```
sage: Composition([1,3,2]).complement()
[2, 1, 2, 1]
sage: ribbon(elementary([1,3,2]))
R[1, 1, 1, 1, 1, 1] + R[1, 1, 1, 2, 1] + R[2, 1, 1, 1, 1] + R[2, 1, 2, 1]
```

```
>>> from sage.all import *
>>> Composition([Integer(1),Integer(3),Integer(2)]).complement()
[2, 1, 2, 1]
>>> ribbon(elementary([Integer(1),Integer(3),Integer(2)]))
R[1, 1, 1, 1, 1, 1] + R[1, 1, 1, 2, 1] + R[2, 1, 1, 1, 1] + R[2, 1, 2, 1]
```

By Möbius inversion on the composition poset, the ribbon basis element corresponding to a composition I is then the alternating sum over all compositions fatter than the complement composition of I in the elementary basis:

```
sage: elementary(ribbon[2,1,2,1])
L[1, 3, 2] - L[1, 5] - L[4, 2] + L[6]
```

```
>>> from sage.all import *
>>> elementary(ribbon[Integer(2),Integer(1),Integer(2),Integer(1)])
L[1, 3, 2] - L[1, 5] - L[4, 2] + L[6]
```

The Φ (`Phi`) and Ψ bases are computed by changing to and from the `Complete` basis. The expansion of Ψ basis is given in Proposition 4.5 of [NCSF1] by the formulae

$$S^I = \sum_{J \geq I} \frac{1}{\pi_u(J,I)} \Psi^J$$

and

$$\Psi^I = \sum_{J \geq I} (-1)^{\ell(J)-\ell(I)} lp(J, I) S^J$$

where the coefficients $\pi_u(J, I)$ and $lp(J, I)$ are coefficients in the methods `coeff_pi()` and `coeff_lp()` respectively. For example:

```
sage: Psi(complete[3])
1/6*Psi[1, 1, 1] + 1/3*Psi[1, 2] + 1/6*Psi[2, 1] + 1/3*Psi[3]
sage: complete(Psi[3])
S[1, 1, 1] - 2*S[1, 2] - S[2, 1] + 3*S[3]
```

```
>>> from sage.all import *
>>> Psi(complete[Integer(3)])
1/6*Psi[1, 1, 1] + 1/3*Psi[1, 2] + 1/6*Psi[2, 1] + 1/3*Psi[3]
>>> complete(Psi[Integer(3)])
S[1, 1, 1] - 2*S[1, 2] - S[2, 1] + 3*S[3]
```

The `Phi` basis is another analogue of the power sum basis from the algebra of symmetric functions and the expansion in the Complete basis is given in Proposition 4.9 of [NCSF1] by the formulae

$$S^I = \sum_{J \geq I} \frac{1}{sp(J, I)} \Phi^J$$

and

$$\Phi^I = \sum_{J \geq I} (-1)^{\ell(J)-\ell(I)} \frac{\prod_i I_i}{\ell(J, I)} S^J$$

where the coefficients $sp(J, I)$ and $\ell(J, I)$ are coefficients in the methods `coeff_sp()` and `coeff_ell()` respectively. For example:

```
sage: Phi(complete[3])
1/6*Phi[1, 1, 1] + 1/4*Phi[1, 2] + 1/4*Phi[2, 1] + 1/3*Phi[3]
sage: complete(Phi[3])
S[1, 1, 1] - 3/2*S[1, 2] - 3/2*S[2, 1] + 3*S[3]
```

```
>>> from sage.all import *
>>> Phi(complete[Integer(3)])
1/6*Phi[1, 1, 1] + 1/4*Phi[1, 2] + 1/4*Phi[2, 1] + 1/3*Phi[3]
>>> complete(Phi[Integer(3)])
S[1, 1, 1] - 3/2*S[1, 2] - 3/2*S[2, 1] + 3*S[3]
```

Here is how to fetch the conversion morphisms:

```
sage: f = complete.coerce_map_from(elementary); f
Generic morphism:
  From: NCSF in the Elementary basis
  To:   NCSF in the Complete basis
sage: g = elementary.coerce_map_from(complete); g
Generic morphism:
  From: NCSF in the Complete basis
  To:   NCSF in the Elementary basis
```

(continues on next page)

(continued from previous page)

```
sage: f.category()
Category of homsets of unital magmas and right modules over Rational Field and
  left modules over Rational Field
sage: f(elementary[1,2,2])
S[1, 1, 1, 1, 1] - S[1, 1, 1, 2] - S[1, 2, 1, 1] + S[1, 2, 2]
sage: g(complete[1,2,2])
L[1, 1, 1, 1, 1] - L[1, 1, 1, 2] - L[1, 2, 1, 1] + L[1, 2, 2]
sage: h = f*g; h
Composite map:
  From: NCSF in the Complete basis
  To:   NCSF in the Complete basis
  Defn:   Generic morphism:
          From: NCSF in the Complete basis
          To:   NCSF in the Elementary basis
        then
          Generic morphism:
          From: NCSF in the Elementary basis
          To:   NCSF in the Complete basis
sage: h(complete[1,3,2])
S[1, 3, 2]
```

```
>>> from sage.all import *
>>> f = complete.coerce_map_from(elementary); f
Generic morphism:
  From: NCSF in the Elementary basis
  To:   NCSF in the Complete basis
>>> g = elementary.coerce_map_from(complete); g
Generic morphism:
  From: NCSF in the Complete basis
  To:   NCSF in the Elementary basis
>>> f.category()
Category of homsets of unital magmas and right modules over Rational Field and
  left modules over Rational Field
>>> f(elementary[Integer(1),Integer(2),Integer(2)])
S[1, 1, 1, 1, 1] - S[1, 1, 1, 2] - S[1, 2, 1, 1] + S[1, 2, 2]
>>> g(complete[Integer(1),Integer(2),Integer(2)])
L[1, 1, 1, 1, 1] - L[1, 1, 1, 2] - L[1, 2, 1, 1] + L[1, 2, 2]
>>> h = f*g; h
Composite map:
  From: NCSF in the Complete basis
  To:   NCSF in the Complete basis
  Defn:   Generic morphism:
          From: NCSF in the Complete basis
          To:   NCSF in the Elementary basis
        then
          Generic morphism:
          From: NCSF in the Elementary basis
          To:   NCSF in the Complete basis
>>> h(complete[Integer(1),Integer(3),Integer(2)])
S[1, 3, 2]
```

Additional concrete representations

NCSF has some additional bases which appear in the literature:

```
sage: Monomial                  = NCSF.Monomial()
sage: Immaculate                = NCSF.Immaculate()
sage: dualQuasisymmetric_Schur = NCSF.dualQuasisymmetric_Schur()
```

```
>>> from sage.all import *
>>> Monomial                  = NCSF.Monomial()
>>> Immaculate                = NCSF.Immaculate()
>>> dualQuasisymmetric_Schur = NCSF.dualQuasisymmetric_Schur()
```

The *Monomial* basis was introduced in [Tev2007] and the *Immaculate* basis was introduced in [BBSSZ2012]. The *Quasisymmetric_Schur* were defined in [QSCHUR] and the dual basis is implemented here as *dualQuasisymmetric_Schur*. Refer to the documentation for the use and definition of these bases.

Todo

- implement fundamental, forgotten, and simple (coming from the simple modules of HS_n) bases.

We revert back to the original name from our custom short name NCSF:

```
sage: NCSF
NCSF
sage: NCSF.rename()
sage: NCSF
Non-Commutative Symmetric Functions over the Rational Field
```

```
>>> from sage.all import *
>>> NCSF
NCSF
>>> NCSF.rename()
>>> NCSF
Non-Commutative Symmetric Functions over the Rational Field
```

class Bases(*parent_with_realization*)

Bases: `Category_realization_of_parent`

Category of bases of non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.Bases()
Category of bases of Non-Commutative Symmetric Functions over the Rational␣
↪Field
sage: R = N.Ribbon()
sage: R in N.Bases()
True
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
```

(continues on next page)

(continued from previous page)

```
>>> N.Bases()
Category of bases of Non-Commutative Symmetric Functions over the Rational␣
↪Field
>>> R = N.Ribbon()
>>> R in N.Bases()
True
```

class ElementMethods

Bases: `object`

bernstein_creation_operator(*n*)

Return the image of `self` under the n-th Bernstein creation operator.

Let n be an integer. The n-th Bernstein creation operator $\mathbb{B}_n$ is defined as the endomorphism of the space $NSym$ of noncommutative symmetric functions which sends every f to

$$\sum_{i \geq 0} (-1)^i H_{n+i} F_{1^i}^{\perp},$$

where usual notations are in place (the letter H stands for the complete basis of $NSym$, the letter F stands for the fundamental basis of the algebra $QSym$ of quasisymmetric functions, and $F_{1^i}^{\perp}$ means skewing (`skew_by()`) by F_{1^i}). Notice that F_{1^i} is nothing other than the elementary symmetric function e_i.

This has been introduced in [BBSSZ2012], section 3.1, in analogy to the Bernstein creation operators on the symmetric functions (`bernstein_creation_operator()`), and studied further in [BBSSZ2012], mainly in the context of immaculate functions (`Immaculate`). In fact, if $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ is an m-tuple of integers, then

$$\mathbb{B}_n I_{(\alpha_1, \alpha_2, \ldots, \alpha_m)} = I_{(n, \alpha_1, \alpha_2, \ldots, \alpha_m)},$$

where $I_{(\alpha_1, \alpha_2, \ldots, \alpha_m)}$ is the immaculate function associated to the m-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ (see `immaculate_function()`).

EXAMPLES:

We get the immaculate functions by repeated application of Bernstein creation operators:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: I = NSym.I()
sage: S = NSym.S()
sage: def immaculate_by_bernstein(xs):
....:     # immaculate function corresponding to integer
....:     # tuple ``xs``, computed by iterated application
....:     # of Bernstein creation operators.
....:     res = S.one()
....:     for i in reversed(xs):
....:         res = res.bernstein_creation_operator(i)
....:     return res
sage: import itertools
sage: all( immaculate_by_bernstein(p) == I.immaculate_function(p)
....:      for p in itertools.product(range(-1, 3), repeat=3))
True
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> I = NSym.I()
>>> S = NSym.S()
>>> def immaculate_by_bernstein(xs):
...     # immaculate function corresponding to integer
...     # tuple ``xs``, computed by iterated application
...     # of Bernstein creation operators.
...     res = S.one()
...     for i in reversed(xs):
...         res = res.bernstein_creation_operator(i)
...     return res
>>> import itertools
>>> all( immaculate_by_bernstein(p) == I.immaculate_function(p)
...      for p in itertools.product(range(-Integer(1), Integer(3)),
↪repeat=Integer(3)))
True
```

Some examples:

```
sage: S[3,2].bernstein_creation_operator(-2)
S[2, 1]
sage: S[3,2].bernstein_creation_operator(-1)
S[1, 2, 1] - S[2, 2] - S[3, 1]
sage: S[3,2].bernstein_creation_operator(0)
-S[1, 2, 2] - S[1, 3, 1] + S[2, 2, 1] + S[3, 2]
sage: S[3,2].bernstein_creation_operator(1)
S[1, 3, 2] - S[2, 2, 2] - S[2, 3, 1] + S[3, 2, 1]
sage: S[3,2].bernstein_creation_operator(2)
S[2, 3, 2] - S[3, 2, 2] - S[3, 3, 1] + S[4, 2, 1]
```

```
>>> from sage.all import *
>>> S[Integer(3),Integer(2)].bernstein_creation_operator(-Integer(2))
S[2, 1]
>>> S[Integer(3),Integer(2)].bernstein_creation_operator(-Integer(1))
S[1, 2, 1] - S[2, 2] - S[3, 1]
>>> S[Integer(3),Integer(2)].bernstein_creation_operator(Integer(0))
-S[1, 2, 2] - S[1, 3, 1] + S[2, 2, 1] + S[3, 2]
>>> S[Integer(3),Integer(2)].bernstein_creation_operator(Integer(1))
S[1, 3, 2] - S[2, 2, 2] - S[2, 3, 1] + S[3, 2, 1]
>>> S[Integer(3),Integer(2)].bernstein_creation_operator(Integer(2))
S[2, 3, 2] - S[3, 2, 2] - S[3, 3, 1] + S[4, 2, 1]
```

chi()

Return the commutative image of a non-commutative symmetric function.

OUTPUT: the commutative image of `self`; this will be a symmetric function

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
```

(continues on next page)

(continued from previous page)

```
sage: x.to_symmetric_function()
2*s[] + 2*s[1] + 3*s[1, 1]
sage: y = N.Phi()[1,3]
sage: y.to_symmetric_function()
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
>>> x.to_symmetric_function()
2*s[] + 2*s[1] + 3*s[1, 1]
>>> y = N.Phi()[Integer(1),Integer(3)]
>>> y.to_symmetric_function()
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

expand(*n*, *alphabet='x'*)

Expand the noncommutative symmetric function into an element of a free algebra in n indeterminates of an alphabet, which by default is `'x'`.

INPUT:

- `n` – nonnegative integer; the number of variables in the expansion
- `alphabet` – (default: `'x'`) the alphabet in which `self` is to be expanded

OUTPUT: an expansion of `self` into the n variables specified by `alphabet`

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: S[3].expand(3)
x0^3 + x0^2*x1 + x0^2*x2 + x0*x1^2 + x0*x1*x2
 + x0*x2^2 + x1^3 + x1^2*x2 + x1*x2^2 + x2^3
sage: L = NSym.L()
sage: L[3].expand(3)
x2*x1*x0
sage: L[2].expand(3)
x1*x0 + x2*x0 + x2*x1
sage: L[3].expand(4)
x2*x1*x0 + x3*x1*x0 + x3*x2*x0 + x3*x2*x1
sage: Psi = NSym.Psi()
sage: Psi[2, 1].expand(3)
x0^3 + x0^2*x1 + x0^2*x2 + x0*x1*x0 + x0*x1^2 + x0*x1*x2
 + x0*x2*x0 + x0*x2*x1 + x0*x2^2 - x1*x0^2 - x1*x0*x1
 - x1*x0*x2 + x1^2*x0 + x1^3 + x1^2*x2 + x1*x2*x0
 + x1*x2*x1 + x1*x2^2 - x2*x0^2 - x2*x0*x1 - x2*x0*x2
 - x2*x1*x0 - x2*x1^2 - x2*x1*x2 + x2^2*x0 + x2^2*x1 + x2^3
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> S = NSym.S()
>>> S[Integer(3)].expand(Integer(3))
```

(continues on next page)

(continued from previous page)

```
x0^3 + x0^2*x1 + x0^2*x2 + x0*x1^2 + x0*x1*x2
 + x0*x2^2 + x1^3 + x1^2*x2 + x1*x2^2 + x2^3
>>> L = NSym.L()
>>> L[Integer(3)].expand(Integer(3))
x2*x1*x0
>>> L[Integer(2)].expand(Integer(3))
x1*x0 + x2*x0 + x2*x1
>>> L[Integer(3)].expand(Integer(4))
x2*x1*x0 + x3*x1*x0 + x3*x2*x0 + x3*x2*x1
>>> Psi = NSym.Psi()
>>> Psi[Integer(2), Integer(1)].expand(Integer(3))
x0^3 + x0^2*x1 + x0^2*x2 + x0*x1*x0 + x0*x1^2 + x0*x1*x2
 + x0*x2*x0 + x0*x2*x1 + x0*x2^2 - x1*x0^2 - x1*x0*x1
 - x1*x0*x2 + x1^2*x0 + x1^3 + x1^2*x2 + x1*x2*x0
 + x1*x2*x1 + x1*x2^2 - x2*x0^2 - x2*x0*x1 - x2*x0*x2
 - x2*x1*x0 - x2*x1^2 - x2*x1*x2 + x2^2*x0 + x2^2*x1 + x2^3
```

One can use a different set of variables by adding an optional argument `alphabet=...`:

```
sage: L[3].expand(4, alphabet='y')
y2*y1*y0 + y3*y1*y0 + y3*y2*y0 + y3*y2*y1
```

```
>>> from sage.all import *
>>> L[Integer(3)].expand(Integer(4), alphabet='y')
y2*y1*y0 + y3*y1*y0 + y3*y2*y0 + y3*y2*y1
```

Todo

So far this is only implemented on the elementary basis, and everything else goes through coercion. Maybe it is worth shortcutting some of the other bases?

left_padded_kronecker_product(*x*)

Return the left-padded Kronecker product of `self` and `x` in the basis of `self`.

The left-padded Kronecker product is a bilinear map mapping two non-commutative symmetric functions to another, not necessarily preserving degree. It can be defined as follows: Let $*$ denote the internal product (`internal_product()`) on the space of non-commutative symmetric functions. For any composition I, let S^I denote the complete homogeneous symmetric function indexed by I. For any compositions α, β, γ, let $g^{\gamma}_{\alpha,\beta}$ denote the coefficient of S^{γ} in the internal product $S^{\alpha} * S^{\beta}$. For every composition $I = (i_1, i_2, \ldots, i_k)$ and every integer $n > |I|$, define the *`n`-completion of `I`* to be the composition $(n - |I|, i_1, i_2, \ldots, i_k)$; this n-completion is denoted by $I[n]$. Then, for any compositions α and β and every integer $n > |\alpha| + |\beta|$, we can write the internal product $S^{\alpha[n]} * S^{\beta[n]}$ in the form

$$S^{\alpha[n]} * S^{\beta[n]} = \sum_{\gamma} g^{\gamma[n]}_{\alpha[n],\beta[n]} S^{\gamma[n]}$$

with γ ranging over all compositions. The coefficients $g^{\gamma[n]}_{\alpha[n],\beta[n]}$ are independent on n. These coefficients $g^{\gamma[n]}_{\alpha[n],\beta[n]}$ are denoted by $\widetilde{g}^{\gamma}_{\alpha,\beta}$, and the non-commutative symmetric function

$$\sum_{\gamma} \widetilde{g}^{\gamma}_{\alpha,\beta} S^{\gamma}$$

is said to be the *left-padded Kronecker product* of S^{α} and S^{β}. By bilinearity, this extends to a definition of a left-padded Kronecker product of any two non-commutative symmetric functions.

The left-padded Kronecker product on the non-commutative symmetric functions lifts the left-padded Kronecker product on the symmetric functions. More precisely: Let π denote the canonical projection (`to_symmetric_function()`) from the non-commutative symmetric functions to the symmetric functions. Then, any two non-commutative symmetric functions f and g satisfy

$$\pi(f\overline{*}g) = \pi(f)\overline{*}\pi(g),$$

where the $\overline{*}$ on the left-hand side denotes the left-padded Kronecker product on the non-commutative symmetric functions, and the $\overline{*}$ on the right-hand side denotes the left-padded Kronecker product on the symmetric functions.

INPUT:

- `x` – element of the ring of non-commutative symmetric functions over the same base ring as `self`

OUTPUT:

- the left-padded Kronecker product of `self` with `x` (an element of the ring of non-commutative symmetric functions in the same basis as `self`)

AUTHORS:

- Darij Grinberg (15 Mar 2014)

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: S[2,1].left_padded_kronecker_product(S[3])
S[1, 1, 1, 1] + S[1, 2, 1] + S[2, 1] + S[2, 1, 1, 1] + S[2, 2, 1] +␣
↪S[3, 2, 1]
sage: S[2,1].left_padded_kronecker_product(S[1])
S[1, 1, 1] + S[1, 2, 1] + S[2, 1]
sage: S[1].left_padded_kronecker_product(S[2,1])
S[1, 1, 1] + S[2, 1] + S[2, 1, 1]
sage: S[1,1].left_padded_kronecker_product(S[2])
S[1, 1] + 2*S[1, 1, 1] + S[2, 1, 1]
sage: S[1].left_padded_kronecker_product(S[1,2,1])
S[1, 1, 1, 1] + S[1, 2, 1] + S[1, 2, 1, 1] + S[2, 1, 1]
sage: S[2].left_padded_kronecker_product(S[3])
S[1, 2] + S[2, 1, 1] + S[3, 2]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> S = NSym.S()
>>> S[Integer(2),Integer(1)].left_padded_kronecker_
↪product(S[Integer(3)])
S[1, 1, 1, 1] + S[1, 2, 1] + S[2, 1] + S[2, 1, 1, 1] + S[2, 2, 1] +␣
↪S[3, 2, 1]
>>> S[Integer(2),Integer(1)].left_padded_kronecker_
↪product(S[Integer(1)])
S[1, 1, 1] + S[1, 2, 1] + S[2, 1]
>>> S[Integer(1)].left_padded_kronecker_product(S[Integer(2),
↪Integer(1)])
S[1, 1, 1] + S[2, 1] + S[2, 1, 1]
>>> S[Integer(1),Integer(1)].left_padded_kronecker_
↪product(S[Integer(2)])
S[1, 1] + 2*S[1, 1, 1] + S[2, 1, 1]
```

(continues on next page)

(continued from previous page)

```
>>> S[Integer(1)].left_padded_kronecker_product(S[Integer(1),
↪Integer(2),Integer(1)])
S[1, 1, 1, 1] + S[1, 2, 1] + S[1, 2, 1, 1] + S[2, 1, 1]
>>> S[Integer(2)].left_padded_kronecker_product(S[Integer(3)])
S[1, 2] + S[2, 1, 1] + S[3, 2]
```

Taking the left-padded Kronecker product with $1 = S$ is the identity map on the ring of non-commutative symmetric functions:

```
sage: all( S[Composition([])].left_padded_kronecker_product(S[lam])
....:      == S[lam].left_padded_kronecker_product(S[Composition([])])
....:      == S[lam] for i in range(4)
....:      for lam in Compositions(i) )
True
```

```
>>> from sage.all import *
>>> all( S[Composition([])].left_padded_kronecker_product(S[lam])
...      == S[lam].left_padded_kronecker_product(S[Composition([])])
...      == S[lam] for i in range(Integer(4))
...      for lam in Compositions(i) )
True
```

Here is a rule for the left-padded Kronecker product of S_1 (this is the same as $S^{(1)}$) with any complete homogeneous function: Let I be a composition. Then, the left-padded Kronecker product of S_1 and S^I is $\sum_K a_K S^K$, where the sum runs over all compositions K, and the coefficient a_K is defined as the number of ways to obtain K from I by one of the following two operations:

- Insert a 1 at the end of I.
- Subtract 1 from one of the entries of I (and remove the entry if it thus becomes 0), and insert a 1 at the end of I.

We check this for compositions of size ≤ 4:

```
sage: def mults1(I):
....:     # Left left-padded Kronecker multiplication by S[1].
....:     res = S[I[:] + [1]]
....:     for k in range(len(I)):
....:         I2 = I[:]
....:         if I2[k] == 1:
....:             I2 = I2[:k] + I2[k+1:]
....:         else:
....:             I2[k] -= 1
....:         res += S[I2 + [1]]
....:     return res
sage: all( mults1(I) == S[1].left_padded_kronecker_product(S[I])
....:      for i in range(5) for I in Compositions(i) )
True
```

```
>>> from sage.all import *
>>> def mults1(I):
...     # Left left-padded Kronecker multiplication by S[1].
...     res = S[I[:] + [Integer(1)]]
...     for k in range(len(I)):
...         I2 = I[:]
```

(continues on next page)

(continued from previous page)

```
...             if I2[k] == Integer(1):
...                 I2 = I2[:k] + I2[k+Integer(1):]
...             else:
...                 I2[k] -= Integer(1)
...             res += S[I2 + [Integer(1)]]
...         return res
>>> all( mults1(I) == S[Integer(1)].left_padded_kronecker_product(S[I])
...          for i in range(Integer(5)) for I in Compositions(i) )
True
```

A similar rule can be made for the left-padded Kronecker product of any complete homogeneous function with S_1: Let I be a composition. Then, the left-padded Kronecker product of S^I and S_1 is $\sum_K b_K S^K$, where the sum runs over all compositions K, and the coefficient b_K is defined as the number of ways to obtain K from I by one of the following two operations:

- Insert a 1 at the front of I.
- Subtract 1 from one of the entries of I (and remove the entry if it thus becomes 0), and insert a 1 right after this entry.

We check this for compositions of size ≤ 4:

```
sage: def mults2(I):
....:     # Left left-padded Kronecker multiplication by S[1].
....:     res = S[[1] + I[:]]
....:     for k in range(len(I)):
....:         I2 = I[:]
....:         i2k = I2[k]
....:         if i2k != 1:
....:             I2 = I2[:k] + [i2k-1, 1] + I2[k+1:]
....:         res += S[I2]
....:     return res
sage: all( mults2(I) == S[I].left_padded_kronecker_product(S[1])
....:      for i in range(5) for I in Compositions(i) )
True
```

```
>>> from sage.all import *
>>> def mults2(I):
...     # Left left-padded Kronecker multiplication by S[1].
...     res = S[[Integer(1)] + I[:]]
...     for k in range(len(I)):
...         I2 = I[:]
...         i2k = I2[k]
...         if i2k != Integer(1):
...             I2 = I2[:k] + [i2k-Integer(1), Integer(1)] +␣
↪I2[k+Integer(1):]
...         res += S[I2]
...     return res
>>> all( mults2(I) == S[I].left_padded_kronecker_product(S[Integer(1)])
...          for i in range(Integer(5)) for I in Compositions(i) )
True
```

Checking the $\pi(f \overline{*} g) = \pi(f) \overline{*} \pi(g)$ equality:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
```

(continues on next page)

(continued from previous page)

```
sage: R = NSym.R()
sage: def testpi(n):
....:     for I in Compositions(n):
....:         for J in Compositions(n):
....:             a = R[I].to_symmetric_function()
....:             b = R[J].to_symmetric_function()
....:             x = a.left_padded_kronecker_product(b)
....:             y = R[I].left_padded_kronecker_product(R[J])
....:             y = y.to_symmetric_function()
....:             if x != y:
....:                 return False
....:     return True
sage: testpi(3)
True
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> R = NSym.R()
>>> def testpi(n):
...     for I in Compositions(n):
...         for J in Compositions(n):
...             a = R[I].to_symmetric_function()
...             b = R[J].to_symmetric_function()
...             x = a.left_padded_kronecker_product(b)
...             y = R[I].left_padded_kronecker_product(R[J])
...             y = y.to_symmetric_function()
...             if x != y:
...                 return False
...     return True
>>> testpi(Integer(3))
True
```

omega_involution()

Return the image of the noncommutative symmetric function `self` under the omega involution.

The omega involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to the n-th elementary non-commutative symmetric function Λ_n. This omega involution is denoted by ω. It can be shown that every composition I satisfies

$$\omega(S^I) = \Lambda^{I^r}, \quad \omega(\Lambda^I) = S^{I^r}, \quad \omega(R_I) = R_{I^t}, \quad \omega(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^{I^r}, \omega(\Psi^I) = (-1)^{|I|-\ell(I)}\Psi^{I^r},$$

where I^r denotes the reversed composition of I, and I^t denotes the conjugate composition of I, and $\ell(I)$ denotes the length of the composition I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, Φ for that of the power-sums of the second kind, and Ψ for that of the power-sums of the first kind). More generally, if f is a homogeneous element of $NCSF$ of degree n, then

$$\omega(f) = (-1)^n S(f),$$

where S denotes the antipode of $NCSF$.

The omega involution ω is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(\omega(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The omega involution on $NCSF$ is adjoint to the omega involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The omega involution has been denoted by ω in [LMvW13], section 3.6. See [NCSF1], section 3.1 for the properties of this map.

See also:

`omega involution of QSym`, `psi involution of NCSF`, `star involution of NCSF`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: L = NSym.L()
sage: L(S[3,2].omega_involution())
L[2, 3]
sage: L(S[6,3].omega_involution())
L[3, 6]
sage: L(S[1,3].omega_involution())
L[3, 1]
sage: L((S[9,1] - S[8,2] + 2*S[6,4] - 3*S[3] + 4*S[[]]).omega_
↪involution()) # long time
4*L[] + L[1, 9] - L[2, 8] - 3*L[3] + 2*L[4, 6]
sage: L((S[3,3] - 2*S[2]).omega_involution())
-2*L[2] + L[3, 3]
sage: L(S([4,2]).omega_involution())
L[2, 4]
sage: R = NSym.R()
sage: R([4,2]).omega_involution()
R[1, 2, 1, 1, 1]
sage: R.zero().omega_involution()
0
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi([2,1]).omega_involution()
-Phi[1, 2]
sage: Psi = NSym.Psi()
sage: Psi([2,1]).omega_involution()
-Psi[1, 2]
sage: Psi([3,1]).omega_involution()
Psi[1, 3]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> L = NSym.L()
>>> L(S[Integer(3),Integer(2)].omega_involution())
L[2, 3]
```

(continues on next page)

(continued from previous page)

```
>>> L(S[Integer(6),Integer(3)].omega_involution())
L[3, 6]
>>> L(S[Integer(1),Integer(3)].omega_involution())
L[3, 1]
>>> L((S[Integer(9),Integer(1)] - S[Integer(8),Integer(2)] +␣
↪Integer(2)*S[Integer(6),Integer(4)] - Integer(3)*S[Integer(3)] +␣
↪Integer(4)*S[[]]).omega_involution()) # long time
4*L[] + L[1, 9] - L[2, 8] - 3*L[3] + 2*L[4, 6]
>>> L((S[Integer(3),Integer(3)] - Integer(2)*S[Integer(2)]).omega_
↪involution())
-2*L[2] + L[3, 3]
>>> L(S([Integer(4),Integer(2)]).omega_involution())
L[2, 4]
>>> R = NSym.R()
>>> R([Integer(4),Integer(2)]).omega_involution()
R[1, 2, 1, 1, 1]
>>> R.zero().omega_involution()
0
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> Phi = NSym.Phi()
>>> Phi([Integer(2),Integer(1)]).omega_involution()
-Phi[1, 2]
>>> Psi = NSym.Psi()
>>> Psi([Integer(2),Integer(1)]).omega_involution()
-Psi[1, 2]
>>> Psi([Integer(3),Integer(1)]).omega_involution()
Psi[1, 3]
```

Testing the $\pi(\omega(f)) = \omega(\pi(f))$ relation noticed above:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: R = NSym.R()
sage: all( R(I).omega_involution().to_symmetric_function()
....:      == R(I).to_symmetric_function().omega_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> R = NSym.R()
>>> all( R(I).omega_involution().to_symmetric_function()
...      == R(I).to_symmetric_function().omega_involution()
...      for I in Compositions(Integer(4)) )
True
```

The omega involution on $QSym$ is adjoint to the omega involution on $NSym$ with respect to the duality pairing:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
```

(continues on next page)

(continued from previous page)

```
sage: all( all( M(I).omega_involution().duality_pairing(S(J))
....:           == M(I).duality_pairing(S(J).omega_involution())
....:           for I in Compositions(2) )
....:       for J in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.M()
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> S = NSym.S()
>>> all( all( M(I).omega_involution().duality_pairing(S(J))
...           == M(I).duality_pairing(S(J).omega_involution())
...           for I in Compositions(Integer(2)) )
...       for J in Compositions(Integer(3)) )
True
```

psi_involution()

Return the image of the noncommutative symmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $NCSF \to NCSF$ which, for every composition I, sends the complete noncommutative symmetric function S^I to the elementary noncommutative symmetric function Λ^I. It can be shown that every composition I satisfies

$$\psi(R_I) = R_{I^c}, \quad \psi(S^I) = \Lambda^I, \quad \psi(\Lambda^I) = S^I, \quad \psi(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^I$$

where I^c denotes the complement of the composition I, and $\ell(I)$ denotes the length of I, and where standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, Φ for the basis of the power sums of the second kind, and R for the ribbon basis). The map ψ is an involution and a graded Hopf algebra automorphism of $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (*to_symmetric_function()*), then $\pi(\psi(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The involution ψ of $NCSF$ is adjoint to the involution ψ of $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also

psi involution of QSym, *star involution of NCSF*.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: R = NSym.R()
sage: R[3,2].psi_involution()
R[1, 1, 2, 1]
sage: R[6,3].psi_involution()
R[1, 1, 1, 1, 1, 2, 1, 1]
sage: (R[9,1] - R[8,2] + 2*R[2,4] - 3*R[3] + 4*R[[]]).psi_involution()
4*R[] - 3*R[1, 1, 1] + R[1, 1, 1, 1, 1, 1, 1, 1, 2] - R[1, 1, 1, 1, 1,
```

(continues on next page)

(continued from previous page)

```
↪1, 1, 2, 1] + 2*R[1, 2, 1, 1, 1]
sage: (R[3,3] - 2*R[2]).psi_involution()
-2*R[1, 1] + R[1, 1, 2, 1, 1]
sage: R([2,1,1]).psi_involution()
R[1, 3]
sage: S = NSym.S()
sage: S([2,1]).psi_involution()
S[1, 1, 1] - S[2, 1]
sage: S.zero().psi_involution()
0
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi([2,1]).psi_involution()
-Phi[2, 1]
sage: Phi([3,1]).psi_involution()
Phi[3, 1]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> R = NSym.R()
>>> R[Integer(3),Integer(2)].psi_involution()
R[1, 1, 2, 1]
>>> R[Integer(6),Integer(3)].psi_involution()
R[1, 1, 1, 1, 1, 2, 1, 1]
>>> (R[Integer(9),Integer(1)] - R[Integer(8),Integer(2)] +␣
↪Integer(2)*R[Integer(2),Integer(4)] - Integer(3)*R[Integer(3)] +␣
↪Integer(4)*R[[]]).psi_involution()
4*R[] - 3*R[1, 1, 1] + R[1, 1, 1, 1, 1, 1, 1, 1, 2] - R[1, 1, 1, 1, 1,␣
↪1, 1, 2, 1] + 2*R[1, 2, 1, 1, 1]
>>> (R[Integer(3),Integer(3)] - Integer(2)*R[Integer(2)]).psi_
↪involution()
-2*R[1, 1] + R[1, 1, 2, 1, 1]
>>> R([Integer(2),Integer(1),Integer(1)]).psi_involution()
R[1, 3]
>>> S = NSym.S()
>>> S([Integer(2),Integer(1)]).psi_involution()
S[1, 1, 1] - S[2, 1]
>>> S.zero().psi_involution()
0
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> Phi = NSym.Phi()
>>> Phi([Integer(2),Integer(1)]).psi_involution()
-Phi[2, 1]
>>> Phi([Integer(3),Integer(1)]).psi_involution()
Phi[3, 1]
```

The Psi basis doesn't behave as nicely:

```
sage: Psi = NSym.Psi()
sage: Psi([2,1]).psi_involution()
-Psi[2, 1]
sage: Psi([3,1]).psi_involution()
```

(continues on next page)

(continued from previous page)

```
1/2*Psi[1, 2, 1] - 1/2*Psi[2, 1, 1] + Psi[3, 1]
```

```
>>> from sage.all import *
>>> Psi = NSym.Psi()
>>> Psi([Integer(2),Integer(1)]).psi_involution()
-Psi[2, 1]
>>> Psi([Integer(3),Integer(1)]).psi_involution()
1/2*Psi[1, 2, 1] - 1/2*Psi[2, 1, 1] + Psi[3, 1]
```

The involution ψ commutes with the antipode:

```
sage: all( R(I).psi_involution().antipode()
....:      == R(I).antipode().psi_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( R(I).psi_involution().antipode()
...      == R(I).antipode().psi_involution()
...      for I in Compositions(Integer(4)) )
True
```

Testing the $\pi(\psi(f)) = \omega(\pi(f))$ relation noticed above:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: R = NSym.R()
sage: all( R(I).psi_involution().to_symmetric_function()
....:      == R(I).to_symmetric_function().omega()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> R = NSym.R()
>>> all( R(I).psi_involution().to_symmetric_function()
...      == R(I).to_symmetric_function().omega()
...      for I in Compositions(Integer(4)) )
True
```

The involution ψ of $QSym$ is adjoint to the involution ψ of $NSym$ with respect to the duality pairing:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: all( all( M(I).psi_involution().duality_pairing(S(J))
....:           == M(I).duality_pairing(S(J).psi_involution())
....:           for I in Compositions(2) )
....:      for J in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.M()
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> S = NSym.S()
>>> all( all( M(I).psi_involution().duality_pairing(S(J))
...           == M(I).duality_pairing(S(J).psi_involution())
...           for I in Compositions(Integer(2)) )
...       for J in Compositions(Integer(3)) )
True
```

star_involution()

Return the image of the noncommutative symmetric function `self` under the star involution.

The star involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to S_n. Denoting by f^* the image of an element $f \in NCSF$ under this star involution, it can be shown that every composition I satisfies

$$(S^I)^* = S^{I^r}, \quad (\Lambda^I)^* = \Lambda^{I^r}, \quad R_I^* = R_{I^r}, \quad (\Phi^I)^* = \Phi^{I^r},$$

where I^r denotes the reversed composition of I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, and Φ for that of the power-sums of the second kind). The star involution is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. Under the canonical isomorphism between the n-th graded component of $NCSF$ and the descent algebra of the symmetric group S_n (see `to_descent_algebra()`), the star involution (restricted to the n-th graded component) corresponds to the automorphism of the descent algebra given by $x \mapsto \omega_n x \omega_n$, where ω_n is the permutation $(n, n-1, \ldots, 1) \in S_n$ (written here in one-line notation). If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(f^*) = \pi(f)$ for every $f \in NCSF$.

The star involution on $NCSF$ is adjoint to the star involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6. See [NCSF2], section 2.3 for the properties of this map.

See also

`star involution of QSym`, `psi involution of NCSF`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: S[3,2].star_involution()
S[2, 3]
sage: S[6,3].star_involution()
S[3, 6]
sage: (S[9,1] - S[8,2] + 2*S[6,4] - 3*S[3] + 4*S[[]]).star_involution()
4*S[] + S[1, 9] - S[2, 8] - 3*S[3] + 2*S[4, 6]
sage: (S[3,3] - 2*S[2]).star_involution()
-2*S[2] + S[3, 3]
```

(continues on next page)

(continued from previous page)

```
sage: S([4,2]).star_involution()
S[2, 4]
sage: R = NSym.R()
sage: R([4,2]).star_involution()
R[2, 4]
sage: R.zero().star_involution()
0
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi([2,1]).star_involution()
Phi[1, 2]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> S[Integer(3),Integer(2)].star_involution()
S[2, 3]
>>> S[Integer(6),Integer(3)].star_involution()
S[3, 6]
>>> (S[Integer(9),Integer(1)] - S[Integer(8),Integer(2)] +␣
↪Integer(2)*S[Integer(6),Integer(4)] - Integer(3)*S[Integer(3)] +␣
↪Integer(4)*S[[]]).star_involution()
4*S[] + S[1, 9] - S[2, 8] - 3*S[3] + 2*S[4, 6]
>>> (S[Integer(3),Integer(3)] - Integer(2)*S[Integer(2)]).star_
↪involution()
-2*S[2] + S[3, 3]
>>> S([Integer(4),Integer(2)]).star_involution()
S[2, 4]
>>> R = NSym.R()
>>> R([Integer(4),Integer(2)]).star_involution()
R[2, 4]
>>> R.zero().star_involution()
0
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> Phi = NSym.Phi()
>>> Phi([Integer(2),Integer(1)]).star_involution()
Phi[1, 2]
```

The Psi basis doesn't behave as nicely:

```
sage: Psi = NSym.Psi()
sage: Psi([2,1]).star_involution()
Psi[1, 2]
sage: Psi([3,1]).star_involution()
1/2*Psi[1, 1, 2] - 1/2*Psi[1, 2, 1] + Psi[1, 3]
```

```
>>> from sage.all import *
>>> Psi = NSym.Psi()
>>> Psi([Integer(2),Integer(1)]).star_involution()
Psi[1, 2]
>>> Psi([Integer(3),Integer(1)]).star_involution()
1/2*Psi[1, 1, 2] - 1/2*Psi[1, 2, 1] + Psi[1, 3]
```

The star involution commutes with the antipode:

```
sage: all( R(I).star_involution().antipode()
....:      == R(I).antipode().star_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( R(I).star_involution().antipode()
...      == R(I).antipode().star_involution()
...      for I in Compositions(Integer(4)) )
True
```

Checking the relation with the descent algebra described above:

```
sage: def descent_test(n):
....:     DA = DescentAlgebra(QQ, n)
....:     NSym = NonCommutativeSymmetricFunctions(QQ)
....:     S = NSym.S()
....:     DAD = DA.D()
....:     w_n = DAD(set(range(1, n)))
....:     for I in Compositions(n):
....:         if not (S[I].star_involution()
....:                 == w_n * S[I].to_descent_algebra(n) * w_n):
....:             return False
....:         return True
sage: all( descent_test(i) for i in range(4) )
True
sage: all( descent_test(i) for i in range(6) ) # long time
True
```

```
>>> from sage.all import *
>>> def descent_test(n):
...     DA = DescentAlgebra(QQ, n)
...     NSym = NonCommutativeSymmetricFunctions(QQ)
...     S = NSym.S()
...     DAD = DA.D()
...     w_n = DAD(set(range(Integer(1), n)))
...     for I in Compositions(n):
...         if not (S[I].star_involution()
...                 == w_n * S[I].to_descent_algebra(n) * w_n):
...             return False
...         return True
>>> all( descent_test(i) for i in range(Integer(4)) )
True
>>> all( descent_test(i) for i in range(Integer(6)) ) # long time
True
```

Testing the $\pi(f^*) = \pi(f)$ relation noticed above:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: R = NSym.R()
sage: all( R(I).star_involution().to_symmetric_function()
```

(continues on next page)

(continued from previous page)

```
....:       == R(I).to_symmetric_function()
....:       for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> R = NSym.R()
>>> all( R(I).star_involution().to_symmetric_function()
...       == R(I).to_symmetric_function()
...       for I in Compositions(Integer(4)) )
True
```

The star involution on $QSym$ is adjoint to the star involution on $NSym$ with respect to the duality pairing:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: all( all( M(I).star_involution().duality_pairing(S(J))
....:             == M(I).duality_pairing(S(J).star_involution())
....:             for I in Compositions(2) )
....:       for J in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.M()
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> S = NSym.S()
>>> all( all( M(I).star_involution().duality_pairing(S(J))
...             == M(I).duality_pairing(S(J).star_involution())
...             for I in Compositions(Integer(2)) )
...       for J in Compositions(Integer(3)) )
True
```

to_descent_algebra(*n=None*)

Return the image of the `n`-th degree homogeneous component of `self` in the descent algebra of S_n over the same base ring as `self`.

This is based upon the canonical isomorphism from the n-th degree homogeneous component of the algebra of noncommutative symmetric functions to the descent algebra of S_n. This isomorphism maps the inner product of noncommutative symmetric functions either to the product in the descent algebra of S_n or to its opposite (depending on how the latter is defined).

If `n` is not specified, it will be taken to be the highest homogeneous component of `self`.

OUTPUT:

- The image of the `n`-th homogeneous component of `self` under the isomorphism into the descent algebra of S_n over the same base ring as `self`.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(ZZ).S()
sage: S[2,1].to_descent_algebra(3)
B[2, 1]
sage: (S[1,2,1] - 3 * S[1,1,2]).to_descent_algebra(4)
-3*B[1, 1, 2] + B[1, 2, 1]
sage: S[2,1].to_descent_algebra(2)
0
sage: S[2,1].to_descent_algebra()
B[2, 1]
sage: S.zero().to_descent_algebra().parent()
Descent algebra of 0 over Integer Ring in the subset basis
sage: (S[1,2,1] - 3 * S[1,1,2]).to_descent_algebra(1)
0
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(ZZ).S()
>>> S[Integer(2),Integer(1)].to_descent_algebra(Integer(3))
B[2, 1]
>>> (S[Integer(1),Integer(2),Integer(1)] - Integer(3) * S[Integer(1),
↪Integer(1),Integer(2)]).to_descent_algebra(Integer(4))
-3*B[1, 1, 2] + B[1, 2, 1]
>>> S[Integer(2),Integer(1)].to_descent_algebra(Integer(2))
0
>>> S[Integer(2),Integer(1)].to_descent_algebra()
B[2, 1]
>>> S.zero().to_descent_algebra().parent()
Descent algebra of 0 over Integer Ring in the subset basis
>>> (S[Integer(1),Integer(2),Integer(1)] - Integer(3) * S[Integer(1),
↪Integer(1),Integer(2)]).to_descent_algebra(Integer(1))
0
```

to_fqsym()

Return the image of the non-commutative symmetric function `self` under the morphism $\iota : NSym \to FQSym$.

This morphism is the injective algebra homomorphism $NSym \to FQSym$ that sends each Complete generator S_n to $F_{[1,2,\ldots,n]}$. It is the inclusion map, if we regard both $NSym$ and $FQSym$ as rings of noncommutative power series.

See also

FreeQuasisymmetricFunctions for a definition of $FQSym$.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = 2*R[[]] + 2*R[1] + 3*R[2]
sage: x.to_fqsym()
2*F[] + 2*F[1] + 3*F[1, 2]
sage: R[2,1].to_fqsym()
F[1, 3, 2] + F[3, 1, 2]
```

(continues on next page)

(continued from previous page)

```
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: x.to_fqsym()
2*F[] + 2*F[1] + 3*F[2, 1]

sage: y = N.Phi()[1,2]
sage: y.to_fqsym()
F[1, 2, 3] - F[1, 3, 2] + F[2, 1, 3] + F[2, 3, 1]
 - F[3, 1, 2] - F[3, 2, 1]

sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S[2].to_fqsym()
F[1, 2]
sage: S[1,2].to_fqsym()
F[1, 2, 3] + F[2, 1, 3] + F[2, 3, 1]
sage: S[2,1].to_fqsym()
F[1, 2, 3] + F[1, 3, 2] + F[3, 1, 2]
sage: S[1,2,1].to_fqsym()
F[1, 2, 3, 4] + F[1, 2, 4, 3] + F[1, 4, 2, 3]
 + F[2, 1, 3, 4] + F[2, 1, 4, 3] + F[2, 3, 1, 4]
 + F[2, 3, 4, 1] + F[2, 4, 1, 3] + F[2, 4, 3, 1]
 + F[4, 1, 2, 3] + F[4, 2, 1, 3] + F[4, 2, 3, 1]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> x = Integer(2)*R[[]] + Integer(2)*R[Integer(1)] +␣
↪Integer(3)*R[Integer(2)]
>>> x.to_fqsym()
2*F[] + 2*F[1] + 3*F[1, 2]
>>> R[Integer(2),Integer(1)].to_fqsym()
F[1, 3, 2] + F[3, 1, 2]
>>> x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
>>> x.to_fqsym()
2*F[] + 2*F[1] + 3*F[2, 1]

>>> y = N.Phi()[Integer(1),Integer(2)]
>>> y.to_fqsym()
F[1, 2, 3] - F[1, 3, 2] + F[2, 1, 3] + F[2, 3, 1]
 - F[3, 1, 2] - F[3, 2, 1]

>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S[Integer(2)].to_fqsym()
F[1, 2]
>>> S[Integer(1),Integer(2)].to_fqsym()
F[1, 2, 3] + F[2, 1, 3] + F[2, 3, 1]
>>> S[Integer(2),Integer(1)].to_fqsym()
F[1, 2, 3] + F[1, 3, 2] + F[3, 1, 2]
>>> S[Integer(1),Integer(2),Integer(1)].to_fqsym()
F[1, 2, 3, 4] + F[1, 2, 4, 3] + F[1, 4, 2, 3]
 + F[2, 1, 3, 4] + F[2, 1, 4, 3] + F[2, 3, 1, 4]
```

(continues on next page)

(continued from previous page)

```
 + F[2, 3, 4, 1] + F[2, 4, 1, 3] + F[2, 4, 3, 1]
 + F[4, 1, 2, 3] + F[4, 2, 1, 3] + F[4, 2, 3, 1]
```

to_fsym()

Return the image of `self` under the natural map to $FSym$.

There is an injective Hopf algebra morphism from $NSym$ to $FSym$ (see *FreeSymmetricFunctions*), which maps the ribbon R_α indexed by a composition α to the sum of all tableaux whose descent composition is α. If we regard $NSym$ as a Hopf subalgebra of $FQSym$ via the morphism $\iota : NSym \to FQSym$ (implemented as *to_fqsym()*), then this injective morphism is just the inclusion map.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = 2*R[[]] + 2*R[1] + 3*R[2]
sage: x.to_fsym()
2*G[] + 2*G[1] + 3*G[12]
sage: R[2,1].to_fsym()
G[12|3]
sage: R[1,2].to_fsym()
G[13|2]
sage: R[2,1,2].to_fsym()
G[12|35|4] + G[125|3|4]
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: x.to_fsym()
2*G[] + 2*G[1] + 3*G[1|2]

sage: y = N.Phi()[1,2]
sage: y.to_fsym()
-G[1|2|3] - G[12|3] + G[123] + G[13|2]

sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S[2].to_fsym()
G[12]
sage: S[2,1].to_fsym()
G[12|3] + G[123]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> x = Integer(2)*R[[]] + Integer(2)*R[Integer(1)] +␣
↪Integer(3)*R[Integer(2)]
>>> x.to_fsym()
2*G[] + 2*G[1] + 3*G[12]
>>> R[Integer(2),Integer(1)].to_fsym()
G[12|3]
>>> R[Integer(1),Integer(2)].to_fsym()
G[13|2]
>>> R[Integer(2),Integer(1),Integer(2)].to_fsym()
G[12|35|4] + G[125|3|4]
```

(continues on next page)

(continued from previous page)

```
>>> x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
>>> x.to_fsym()
2*G[] + 2*G[1] + 3*G[1|2]

>>> y = N.Phi()[Integer(1),Integer(2)]
>>> y.to_fsym()
-G[1|2|3] - G[12|3] + G[123] + G[13|2]

>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S[Integer(2)].to_fsym()
G[12]
>>> S[Integer(2),Integer(1)].to_fsym()
G[12|3] + G[123]
```

to_ncsym()

Return the image of `self` under the injective algebra homomorphism $\kappa : NSym \to NCSym$ that fixes the symmetric functions.

As usual, $NCSym$ denotes the ring of symmetric functions in non-commuting variables. Let S_n denote a generator of the complete basis. The algebra homomorphism $\kappa : NSym \to NCSym$ is defined by

$$S_n \mapsto \sum_{A \vdash [n]} \frac{\lambda(A)!\lambda(A)^!}{n!} \mathbf{m}_A.$$

It has the property that the canonical maps $\chi : NCSym \to Sym$ and $\rho : NSym \to Sym$ satisfy $\chi \circ \kappa = \rho$.

Note

A remark in [BRRZ08] makes it clear that the embedding of $NSym$ into $NCSym$ that preserves the projection into the symmetric functions is not unique. While this seems to be a natural embedding, any free set of algebraic generators of $NSym$ can be sent to a set of free elements in $NCSym$ to form another embedding.

See also

NonCommutativeSymmetricFunctions for a definition of $NCSym$.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S[2].to_ncsym()
1/2*m{{1}, {2}} + m{{1, 2}}
sage: S[1,2,1].to_ncsym()
1/2*m{{1}, {2}, {3}, {4}} + 1/2*m{{1}, {2}, {3, 4}} + m{{1}, {2, 3},
↪{4}}
 + m{{1}, {2, 3, 4}} + 1/2*m{{1}, {2, 4}, {3}} + 1/2*m{{1, 2}, {3}, {4}
```

(continues on next page)

(continued from previous page)

```
↪}
 + 1/2*m{{1, 2}, {3, 4}} + m{{1, 2, 3}, {4}} + m{{1, 2, 3, 4}}
 + 1/2*m{{1, 2, 4}, {3}} + 1/2*m{{1, 3}, {2}, {4}} + 1/2*m{{1, 3}, {2,␣
↪4}}
 + 1/2*m{{1, 3, 4}, {2}} + 1/2*m{{1, 4}, {2}, {3}} + m{{1, 4}, {2, 3}}
sage: S[1,2].to_ncsym()
1/2*m{{1}, {2}, {3}} + m{{1}, {2, 3}} + 1/2*m{{1, 2}, {3}}
 + m{{1, 2, 3}} + 1/2*m{{1, 3}, {2}}
sage: S[[]].to_ncsym()
m{}

sage: R = N.ribbon()
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: x.to_ncsym()
2*m{} + 2*m{{1}} + 3/2*m{{1}, {2}}
sage: R[2,1].to_ncsym()
1/3*m{{1}, {2}, {3}} + 1/6*m{{1}, {2, 3}}
 + 2/3*m{{1, 2}, {3}} + 1/6*m{{1, 3}, {2}}

sage: Phi = N.Phi()
sage: Phi[1,2].to_ncsym()
m{{1}, {2, 3}} + m{{1, 2, 3}}
sage: Phi[1,3].to_ncsym()
-1/4*m{{1}, {2}, {3, 4}} - 1/4*m{{1}, {2, 3}, {4}} + m{{1}, {2, 3, 4}}
 + 1/2*m{{1}, {2, 4}, {3}} - 1/4*m{{1, 2}, {3, 4}} - 1/4*m{{1, 2, 3},
↪{4}}
 + m{{1, 2, 3, 4}} + 1/2*m{{1, 2, 4}, {3}} + 1/2*m{{1, 3}, {2, 4}}
 - 1/4*m{{1, 3, 4}, {2}} - 1/4*m{{1, 4}, {2, 3}}
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> S[Integer(2)].to_ncsym()
1/2*m{{1}, {2}} + m{{1, 2}}
>>> S[Integer(1),Integer(2),Integer(1)].to_ncsym()
1/2*m{{1}, {2}, {3}, {4}} + 1/2*m{{1}, {2}, {3, 4}} + m{{1}, {2, 3},
↪{4}}
 + m{{1}, {2, 3, 4}} + 1/2*m{{1}, {2, 4}, {3}} + 1/2*m{{1, 2}, {3}, {4}
↪}
 + 1/2*m{{1, 2}, {3, 4}} + m{{1, 2, 3}, {4}} + m{{1, 2, 3, 4}}
 + 1/2*m{{1, 2, 4}, {3}} + 1/2*m{{1, 3}, {2}, {4}} + 1/2*m{{1, 3}, {2,␣
↪4}}
 + 1/2*m{{1, 3, 4}, {2}} + 1/2*m{{1, 4}, {2}, {3}} + m{{1, 4}, {2, 3}}
>>> S[Integer(1),Integer(2)].to_ncsym()
1/2*m{{1}, {2}, {3}} + m{{1}, {2, 3}} + 1/2*m{{1, 2}, {3}}
 + m{{1, 2, 3}} + 1/2*m{{1, 3}, {2}}
>>> S[[]].to_ncsym()
m{}

>>> R = N.ribbon()
>>> x = R.an_element(); x
```

(continues on next page)

(continued from previous page)

```
2*R[] + 2*R[1] + 3*R[1, 1]
>>> x.to_ncsym()
2*m{} + 2*m{{1}} + 3/2*m{{1}, {2}}
>>> R[Integer(2),Integer(1)].to_ncsym()
1/3*m{{1}, {2}, {3}} + 1/6*m{{1}, {2, 3}}
 + 2/3*m{{1, 2}, {3}} + 1/6*m{{1, 3}, {2}}

>>> Phi = N.Phi()
>>> Phi[Integer(1),Integer(2)].to_ncsym()
m{{1}, {2, 3}} + m{{1, 2, 3}}
>>> Phi[Integer(1),Integer(3)].to_ncsym()
-1/4*m{{1}, {2}, {3, 4}} - 1/4*m{{1}, {2, 3}, {4}} + m{{1}, {2, 3, 4}}
 + 1/2*m{{1}, {2, 4}, {3}} - 1/4*m{{1, 2}, {3, 4}} - 1/4*m{{1, 2, 3},
→{4}}
 + m{{1, 2, 3, 4}} + 1/2*m{{1, 2, 4}, {3}} + 1/2*m{{1, 3}, {2, 4}}
 - 1/4*m{{1, 3, 4}, {2}} - 1/4*m{{1, 4}, {2, 3}}
```

to_symmetric_function()

Return the commutative image of a non-commutative symmetric function.

OUTPUT: the commutative image of `self`; this will be a symmetric function

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: x.to_symmetric_function()
2*s[] + 2*s[1] + 3*s[1, 1]
sage: y = N.Phi()[1,3]
sage: y.to_symmetric_function()
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
>>> x.to_symmetric_function()
2*s[] + 2*s[1] + 3*s[1, 1]
>>> y = N.Phi()[Integer(1),Integer(3)]
>>> y.to_symmetric_function()
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

to_symmetric_group_algebra()

Return the image of a non-commutative symmetric function into the symmetric group algebra where the ribbon basis element indexed by a composition is associated with the sum of all permutations which have descent set equal to said composition. In compliance with the anti- isomorphism between the descent algebra and the non-commutative symmetric functions, we index descent positions by the reversed composition.

OUTPUT:

- The image of `self` under the embedding of the n-th degree homogeneous component of the non-commutative symmetric functions in the symmetric group algebra of S_n. This can behave unexpectedly when `self` is not homogeneous.

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: R[2,1].to_symmetric_group_algebra()
[1, 3, 2] + [2, 3, 1]
sage: R([]).to_symmetric_group_algebra()
[]
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).R()
>>> R[Integer(2),Integer(1)].to_symmetric_group_algebra()
[1, 3, 2] + [2, 3, 1]
>>> R([]).to_symmetric_group_algebra()
[]
```

verschiebung(n)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the $\mathbf{k}$-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (*verschiebung()*) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $(\underbrace{n, n, \ldots, n}_{|I|/n \text{ times}})$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also

adams_operator method of QSym, *verschiebung method of Sym*

INPUT:

- n – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to self.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: S[3,2].verschiebung(2)
0
sage: S[6,4].verschiebung(2)
S[3, 2]
sage: (S[9,1] - S[8,2] + 2*S[6,4] - 3*S[3] + 4*S[[]]).verschiebung(2)
4*S[] + 2*S[3, 2] - S[4, 1]
sage: (S[3,3] - 2*S[2]).verschiebung(3)
S[1, 1]
sage: S([4,2]).verschiebung(1)
S[4, 2]
sage: R = NSym.R()
sage: R([4,2]).verschiebung(2)
R[2, 1]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> S[Integer(3),Integer(2)].verschiebung(Integer(2))
0
>>> S[Integer(6),Integer(4)].verschiebung(Integer(2))
S[3, 2]
>>> (S[Integer(9),Integer(1)] - S[Integer(8),Integer(2)] +␣
↪Integer(2)*S[Integer(6),Integer(4)] - Integer(3)*S[Integer(3)] +␣
↪Integer(4)*S[[]]).verschiebung(Integer(2))
4*S[] + 2*S[3, 2] - S[4, 1]
>>> (S[Integer(3),Integer(3)] - Integer(2)*S[Integer(2)]).
↪verschiebung(Integer(3))
S[1, 1]
>>> S([Integer(4),Integer(2)]).verschiebung(Integer(1))
S[4, 2]
>>> R = NSym.R()
>>> R([Integer(4),Integer(2)]).verschiebung(Integer(2))
R[2, 1]
```

Being Hopf algebra endomorphisms, the Verschiebung operators commute with the antipode:

```
sage: all( R(I).verschiebung(2).antipode()
....:      == R(I).antipode().verschiebung(2)
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( R(I).verschiebung(Integer(2)).antipode()
```

(continues on next page)

(continued from previous page)

```
...        == R(I).antipode().verschiebung(Integer(2))
...        for I in Compositions(Integer(4)) )
True
```

They lift the Verschiebung operators of the ring of symmetric functions:

```
sage: all( S(I).verschiebung(2).to_symmetric_function()
....:       == S(I).to_symmetric_function().verschiebung(2)
....:       for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( S(I).verschiebung(Integer(2)).to_symmetric_function()
...        == S(I).to_symmetric_function().verschiebung(Integer(2))
...        for I in Compositions(Integer(4)) )
True
```

The Frobenius operators on $QSym$ are adjoint to the Verschiebung operators on $NSym$ with respect to the duality pairing:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: M = QSym.M()
sage: all( all( M(I).adams_operator(3).duality_pairing(S(J))
....:             == M(I).duality_pairing(S(J).verschiebung(3))
....:             for I in Compositions(2) )
....:       for J in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> M = QSym.M()
>>> all( all( M(I).adams_operator(Integer(3)).duality_pairing(S(J))
...             == M(I).duality_pairing(S(J).verschiebung(Integer(3)))
...             for I in Compositions(Integer(2)) )
...        for J in Compositions(Integer(3)) )
True
```

class ParentMethods

Bases: `object`

immaculate_function(*xs*)

Return the immaculate function corresponding to the integer vector `xs`, written in the basis `self`.

If α is any integer vector – i.e., an element of $\mathbf{Z}^m$ for some $m \in \mathbf{N}$ –, the *immaculate function corresponding to* α is a non-commutative symmetric function denoted by $\mathfrak{S}_\alpha$. One way to define this function is by setting

$$\mathfrak{S}_\alpha = \sum_{\sigma \in S_m} (-1)^\sigma S_{\alpha_1+\sigma(1)-1} S_{\alpha_2+\sigma(2)-2} \cdots S_{\alpha_m+\sigma(m)-m},$$

where α is written in the form $(\alpha_1, \alpha_2, \ldots, \alpha_m)$, and where S stands for the complete basis (`Complete`).

The immaculate function $\mathfrak{S}_\alpha$ first appeared in [BBSSZ2012] (where it was defined differently, but the definition we gave above appeared as Theorem 3.27).

The immaculate functions $\mathfrak{S}_\alpha$ for α running over all compositions (i.e., finite sequences of positive integers) form a basis of $NCSF$. This is the *immaculate basis* (`Immaculate`).

INPUT:

- `xs` – list (or tuple or any iterable – possibly a composition) of integers

OUTPUT:

The immaculate function $\mathfrak{S}_{xs}$ written in the basis `self`.

EXAMPLES:

Let us first check that, for `xs` a composition, we get the same as the result of `self.realization_of().I()[xs]`:

```
sage: def test_comp(xs):
....:     NSym = NonCommutativeSymmetricFunctions(QQ)
....:     I = NSym.I()
....:     return I[xs] == I.immaculate_function(xs)
sage: def test_allcomp(n):
....:     return all( test_comp(c) for c in Compositions(n) )
sage: test_allcomp(1)
True
sage: test_allcomp(2)
True
sage: test_allcomp(3)
True
```

```
>>> from sage.all import *
>>> def test_comp(xs):
...     NSym = NonCommutativeSymmetricFunctions(QQ)
...     I = NSym.I()
...     return I[xs] == I.immaculate_function(xs)
>>> def test_allcomp(n):
...     return all( test_comp(c) for c in Compositions(n) )
>>> test_allcomp(Integer(1))
True
>>> test_allcomp(Integer(2))
True
>>> test_allcomp(Integer(3))
True
```

Now some examples of non-composition immaculate functions:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: I = NSym.I()
sage: I.immaculate_function([0, 1])
0
sage: I.immaculate_function([0, 2])
-I[1, 1]
sage: I.immaculate_function([-1, 1])
-I[]
sage: I.immaculate_function([2, -1])
0
sage: I.immaculate_function([2, 0])
I[2]
```

(continues on next page)

(continued from previous page)

```
sage: I.immaculate_function([2, 0, 1])
0
sage: I.immaculate_function([1, 0, 2])
-I[1, 1, 1]
sage: I.immaculate_function([2, 0, 2])
-I[2, 1, 1]
sage: I.immaculate_function([0, 2, 0, 2])
I[1, 1, 1, 1] + I[1, 2, 1]
sage: I.immaculate_function([2, 0, 2, 0, 2])
I[2, 1, 1, 1, 1] + I[2, 1, 2, 1]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> I = NSym.I()
>>> I.immaculate_function([Integer(0), Integer(1)])
0
>>> I.immaculate_function([Integer(0), Integer(2)])
-I[1, 1]
>>> I.immaculate_function([-Integer(1), Integer(1)])
-I[]
>>> I.immaculate_function([Integer(2), -Integer(1)])
0
>>> I.immaculate_function([Integer(2), Integer(0)])
I[2]
>>> I.immaculate_function([Integer(2), Integer(0), Integer(1)])
0
>>> I.immaculate_function([Integer(1), Integer(0), Integer(2)])
-I[1, 1, 1]
>>> I.immaculate_function([Integer(2), Integer(0), Integer(2)])
-I[2, 1, 1]
>>> I.immaculate_function([Integer(0), Integer(2), Integer(0),␣
↪Integer(2)])
I[1, 1, 1, 1] + I[1, 2, 1]
>>> I.immaculate_function([Integer(2), Integer(0), Integer(2),␣
↪Integer(0), Integer(2)])
I[2, 1, 1, 1, 1] + I[2, 1, 2, 1]
```

to_symmetric_function()

Morphism to the algebra of symmetric functions.

This is constructed by extending the computation on the basis or by coercion to the complete basis.

OUTPUT:

- The module morphism from the basis `self` to the symmetric functions which corresponds to taking a commutative image.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: R.to_symmetric_function(x)
2*s[] + 2*s[1] + 3*s[1, 1]
```

(continues on next page)

(continued from previous page)

```
sage: nM = N.Monomial()
sage: nM.to_symmetric_function(nM[3,1])
h[1, 1, 1, 1] - 7/2*h[2, 1, 1] + h[2, 2] + 7/2*h[3, 1] - 2*h[4]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
>>> R.to_symmetric_function(x)
2*s[] + 2*s[1] + 3*s[1, 1]
>>> nM = N.Monomial()
>>> nM.to_symmetric_function(nM[Integer(3),Integer(1)])
h[1, 1, 1, 1] - 7/2*h[2, 1, 1] + h[2, 2] + 7/2*h[3, 1] - 2*h[4]
```

to_symmetric_function_on_basis(*I*)

The image of the basis element indexed by `I` under the map to the symmetric functions.

This default implementation does a change of basis and computes the image in the complete basis.

INPUT:

- `I` – a composition

OUTPUT:

- The image of the non-commutative basis element of `self` indexed by the composition `I` under the map from non-commutative symmetric functions to the symmetric functions. This will be a symmetric function.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: I = N.Immaculate()
sage: I.to_symmetric_function(I[1,3])
-h[2, 2] + h[3, 1]
sage: I.to_symmetric_function(I[1,2])
0
sage: Phi = N.Phi()
sage: Phi.to_symmetric_function_on_basis([3,1,2])==Phi.to_symmetric_
↪function(Phi[3,1,2])
True
sage: Phi.to_symmetric_function_on_basis([])
h[]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> I = N.Immaculate()
>>> I.to_symmetric_function(I[Integer(1),Integer(3)])
-h[2, 2] + h[3, 1]
>>> I.to_symmetric_function(I[Integer(1),Integer(2)])
0
>>> Phi = N.Phi()
>>> Phi.to_symmetric_function_on_basis([Integer(3),Integer(1),
↪Integer(2)])==Phi.to_symmetric_function(Phi[Integer(3),Integer(1),
↪Integer(2)])
```

(continues on next page)

(continued from previous page)

```
True
>>> Phi.to_symmetric_function_on_basis([])
h[]
```

super_categories()

Return the super categories of the category of bases of the non-commutative symmetric functions.

OUTPUT: list

class Complete(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The Hopf algebra of non-commutative symmetric functions in the Complete basis.

The Complete basis is defined in Definition 3.4 of [NCSF1], where it is denoted by $(S^I)_I$. It is a multiplicative basis, and is connected to the elementary generators Λ_i of the ring of non-commutative symmetric functions by the following relation: Define a non-commutative symmetric function S_n for every nonnegative integer n by the power series identity

$$\sum_{k\geq 0} t^k S_k = \left(\sum_{k\geq 0} (-t)^k \Lambda_k \right)^{-1},$$

with Λ_0 denoting 1. For every composition $(i_1, i_2, \ldots, i_k)$, we have $S^{(i_1,i_2,\ldots,i_k)} = S_{i_1} S_{i_2} \cdots S_{i_k}$.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: S = NCSF.Complete(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete␣
↪basis
sage: S.an_element()
2*S[] + 2*S[1] + 3*S[1, 1]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> S = NCSF.Complete(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete␣
↪basis
>>> S.an_element()
2*S[] + 2*S[1] + 3*S[1, 1]
```

The following are aliases for this basis:

```
sage: NCSF.complete()
Non-Commutative Symmetric Functions over the Rational Field in the Complete␣
↪basis
sage: NCSF.S()
Non-Commutative Symmetric Functions over the Rational Field in the Complete␣
↪basis
```

```
>>> from sage.all import *
>>> NCSF.complete()
Non-Commutative Symmetric Functions over the Rational Field in the Complete␣
```

(continues on next page)

(continued from previous page)

```
↪basis
>>> NCSF.S()
Non-Commutative Symmetric Functions over the Rational Field in the Complete␣
↪basis
```

class Element

Bases: `IndexedFreeModuleElement`

An element in the Complete basis.

psi_involution()

Return the image of the noncommutative symmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $NCSF \to NCSF$ which, for every composition I, sends the complete noncommutative symmetric function S^I to the elementary noncommutative symmetric function Λ^I. It can be shown that every composition I satisfies

$$\psi(R_I) = R_{I^c}, \quad \psi(S^I) = \Lambda^I, \quad \psi(\Lambda^I) = S^I, \quad \psi(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^I$$

where I^c denotes the complement of the composition I, and $\ell(I)$ denotes the length of I, and where standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, Φ for the basis of the power sums of the second kind, and R for the ribbon basis). The map ψ is an involution and a graded Hopf algebra automorphism of $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(\psi(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The involution ψ of $NCSF$ is adjoint to the involution ψ of $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also

psi involution of NCSF, *psi involution of QSym*, *star involution of NCSF*.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: L = NSym.L()
sage: S[3,1].psi_involution()
S[1, 1, 1, 1] - S[1, 2, 1] - S[2, 1, 1] + S[3, 1]
sage: L(S[3,1].psi_involution())
L[3, 1]
sage: S[[]].psi_involution()
S[]
sage: S[1,1].psi_involution()
S[1, 1]
sage: (S[2,1] - 2*S[2]).psi_involution()
-2*S[1, 1] + S[1, 1, 1] + 2*S[2] - S[2, 1]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> S = NSym.S()
>>> L = NSym.L()
>>> S[Integer(3),Integer(1)].psi_involution()
S[1, 1, 1, 1] - S[1, 2, 1] - S[2, 1, 1] + S[3, 1]
>>> L(S[Integer(3),Integer(1)].psi_involution())
L[3, 1]
>>> S[[]].psi_involution()
S[]
>>> S[Integer(1),Integer(1)].psi_involution()
S[1, 1]
>>> (S[Integer(2),Integer(1)] - Integer(2)*S[Integer(2)]).psi_
↪involution()
-2*S[1, 1] + S[1, 1, 1] + 2*S[2] - S[2, 1]
```

The implementation at hand is tailored to the complete basis. It is equivalent to the generic implementation via the ribbon basis:

```
sage: R = NSym.R()
sage: all( R(S[I].psi_involution()) == R(S[I]).psi_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> R = NSym.R()
>>> all( R(S[I].psi_involution()) == R(S[I]).psi_involution()
...      for I in Compositions(Integer(4)) )
True
```

dual()

Return the dual basis to the complete basis of non-commutative symmetric functions. This is the Monomial basis of quasi-symmetric functions.

OUTPUT: the Monomial basis of quasi-symmetric functions

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.dual()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S.dual()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

internal_product_on_basis(I, J)

The internal product of two non-commutative symmetric complete functions.

See *internal_product()* for a thorough documentation of this operation.

INPUT:

- I, J – compositions

OUTPUT:

- The internal product of the complete non-commutative symmetric function basis elements indexed by `I` and `J`, expressed in the complete basis.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product_on_basis([2,2],[1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product_on_basis([2,2],[1,2])
0
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> S.internal_product_on_basis([Integer(2),Integer(2)],[Integer(1),
→Integer(2),Integer(1)])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
>>> S.internal_product_on_basis([Integer(2),Integer(2)],[Integer(1),
→Integer(2)])
0
```

to_symmetric_function()

Morphism to the algebra of symmetric functions.

This is constructed by extending the computation on the complete basis.

OUTPUT:

- The module morphism from the basis `self` to the symmetric functions which corresponds to taking a commutative image.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.to_symmetric_function(S[3,1,2])
h[3, 2, 1]
sage: S.to_symmetric_function(S[[]])
h[]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> S.to_symmetric_function(S[Integer(3),Integer(1),Integer(2)])
h[3, 2, 1]
>>> S.to_symmetric_function(S[[]])
h[]
```

to_symmetric_function_on_basis(*I*)

The commutative image of a complete element.

The commutative image of a basis element is obtained by sorting the indexing composition of the basis element and the output is in the complete basis of the symmetric functions.

INPUT:

- `I` – a composition

OUTPUT:

- The commutative image of the complete basis element indexed by `I`. The result is the complete symmetric function indexed by the partition obtained by sorting `I`.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: S.to_symmetric_function_on_basis([2,1,3])
h[3, 2, 1]
sage: S.to_symmetric_function_on_basis([])
h[]
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).complete()
>>> S.to_symmetric_function_on_basis([Integer(2),Integer(1),Integer(3)])
h[3, 2, 1]
>>> S.to_symmetric_function_on_basis([])
h[]
```

class Elementary(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The Hopf algebra of non-commutative symmetric functions in the Elementary basis.

The Elementary basis is defined in Definition 3.4 of [NCSF1], where it is denoted by $(\Lambda^I)_I$. It is a multiplicative basis, and is obtained from the elementary generators Λ_i of the ring of non-commutative symmetric functions through the formula $\Lambda^{(i_1,i_2,\ldots,i_k)} = \Lambda_{i_1}\Lambda_{i_2}\cdots\Lambda_{i_k}$ for every composition $(i_1, i_2, \ldots, i_k)$.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: L = NCSF.Elementary(); L
Non-Commutative Symmetric Functions over the Rational Field in the Elementary␣
↪basis
sage: L.an_element()
2*L[] + 2*L[1] + 3*L[1, 1]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> L = NCSF.Elementary(); L
Non-Commutative Symmetric Functions over the Rational Field in the Elementary␣
↪basis
>>> L.an_element()
2*L[] + 2*L[1] + 3*L[1, 1]
```

The following are aliases for this basis:

```
sage: NCSF.elementary()
Non-Commutative Symmetric Functions over the Rational Field in the Elementary␣
↪basis
sage: NCSF.L()
Non-Commutative Symmetric Functions over the Rational Field in the Elementary␣
↪basis
```

```
>>> from sage.all import *
>>> NCSF.elementary()
```

(continues on next page)

(continued from previous page)

```
Non-Commutative Symmetric Functions over the Rational Field in the Elementary
↪basis
>>> NCSF.L()
Non-Commutative Symmetric Functions over the Rational Field in the Elementary
↪basis
```

class Element

Bases: `IndexedFreeModuleElement`

psi_involution()

Return the image of the noncommutative symmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $NCSF \to NCSF$ which, for every composition I, sends the complete noncommutative symmetric function S^I to the elementary noncommutative symmetric function Λ^I. It can be shown that every composition I satisfies

$$\psi(R_I) = R_{I^c}, \quad \psi(S^I) = \Lambda^I, \quad \psi(\Lambda^I) = S^I, \quad \psi(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^I$$

where I^c denotes the complement of the composition I, and $\ell(I)$ denotes the length of I, and where standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, Φ for the basis of the power sums of the second kind, and R for the ribbon basis). The map ψ is an involution and a graded Hopf algebra automorphism of $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(\psi(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The involution ψ of $NCSF$ is adjoint to the involution ψ of $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also

psi involution of NCSF, *psi involution of QSym*, *star involution of NCSF*.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: L = NSym.L()
sage: L[3,1].psi_involution()
L[1, 1, 1, 1] - L[1, 2, 1] - L[2, 1, 1] + L[3, 1]
sage: S(L[3,1].psi_involution())
S[3, 1]
sage: L[[]].psi_involution()
L[]
sage: L[1,1].psi_involution()
L[1, 1]
sage: (L[2,1] - 2*L[2]).psi_involution()
-2*L[1, 1] + L[1, 1, 1] + 2*L[2] - L[2, 1]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
```

(continues on next page)

(continued from previous page)

```
>>> S = NSym.S()
>>> L = NSym.L()
>>> L[Integer(3),Integer(1)].psi_involution()
L[1, 1, 1, 1] - L[1, 2, 1] - L[2, 1, 1] + L[3, 1]
>>> S(L[Integer(3),Integer(1)].psi_involution())
S[3, 1]
>>> L[[]].psi_involution()
L[]
>>> L[Integer(1),Integer(1)].psi_involution()
L[1, 1]
>>> (L[Integer(2),Integer(1)] - Integer(2)*L[Integer(2)]).psi_
↪involution()
-2*L[1, 1] + L[1, 1, 1] + 2*L[2] - L[2, 1]
```

The implementation at hand is tailored to the elementary basis. It is equivalent to the generic implementation via the ribbon basis:

```
sage: R = NSym.R()
sage: all( R(L[I].psi_involution()) == R(L[I]).psi_involution()
....:      for I in Compositions(3) )
True
sage: all( R(L[I].psi_involution()) == R(L[I]).psi_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> R = NSym.R()
>>> all( R(L[I].psi_involution()) == R(L[I]).psi_involution()
...      for I in Compositions(Integer(3)) )
True
>>> all( R(L[I].psi_involution()) == R(L[I]).psi_involution()
...      for I in Compositions(Integer(4)) )
True
```

star_involution()

Return the image of the noncommutative symmetric function `self` under the star involution.

The star involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to S_n. Denoting by f^* the image of an element $f \in NCSF$ under this star involution, it can be shown that every composition I satisfies

$$(S^I)^* = S^{I^r}, \quad (\Lambda^I)^* = \Lambda^{I^r}, \quad R_I^* = R_{I^r}, \quad (\Phi^I)^* = \Phi^{I^r},$$

where I^r denotes the reversed composition of I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, and Φ for that of the power-sums of the second kind). The star involution is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. Under the canonical isomorphism between the n-th graded component of $NCSF$ and the descent algebra of the symmetric group S_n (see *to_descent_algebra()*), the star involution (restricted to the n-th graded component) corresponds to the automorphism of the descent algebra given by $x \mapsto \omega_n x \omega_n$, where ω_n is the permutation $(n, n-1, \ldots, 1) \in S_n$ (written here in one-line notation). If π denotes the projection from $NCSF$ to the ring of symmetric functions (*to_symmetric_function()*), then $\pi(f^*) = \pi(f)$ for every $f \in NCSF$.

The star involution on $NCSF$ is adjoint to the star involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6. See [NCSF2], section 2.3 for the properties of this map.

See also

star involution of NCSF, *psi involution of NCSF*, *star involution of QSym*.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: L = NSym.L()
sage: L[3,3,2,3].star_involution()
L[3, 2, 3, 3]
sage: L[6,3,3].star_involution()
L[3, 3, 6]
sage: (L[1,9,1] - L[8,2] + 2*L[6,4] - 3*L[3] + 4*L[[]]).star_
↪involution()
4*L[] + L[1, 9, 1] - L[2, 8] - 3*L[3] + 2*L[4, 6]
sage: (L[3,3] - 2*L[2]).star_involution()
-2*L[2] + L[3, 3]
sage: L([4,1]).star_involution()
L[1, 4]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> L = NSym.L()
>>> L[Integer(3),Integer(3),Integer(2),Integer(3)].star_involution()
L[3, 2, 3, 3]
>>> L[Integer(6),Integer(3),Integer(3)].star_involution()
L[3, 3, 6]
>>> (L[Integer(1),Integer(9),Integer(1)] - L[Integer(8),Integer(2)] +␣
↪Integer(2)*L[Integer(6),Integer(4)] - Integer(3)*L[Integer(3)] +␣
↪Integer(4)*L[[]]).star_involution()
4*L[] + L[1, 9, 1] - L[2, 8] - 3*L[3] + 2*L[4, 6]
>>> (L[Integer(3),Integer(3)] - Integer(2)*L[Integer(2)]).star_
↪involution()
-2*L[2] + L[3, 3]
>>> L([Integer(4),Integer(1)]).star_involution()
L[1, 4]
```

The implementation at hand is tailored to the elementary basis. It is equivalent to the generic implementation via the complete basis:

```
sage: S = NSym.S()
sage: all( S(L[I].star_involution()) == S(L[I]).star_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> S = NSym.S()
```

(continues on next page)

(continued from previous page)

```
>>> all( S(L[I].star_involution()) == S(L[I]).star_involution()
...      for I in Compositions(Integer(4)) )
True
```

verschiebung(n)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the $\mathbf{k}$-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (*verschiebung()*) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $\underbrace{(n, n, \ldots, n)}_{|I|/n \text{ times}}$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also:

verschiebung method of NCSF, *adams_operator method of QSym*, *verschiebung method of Sym*

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to `self`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: L = NSym.L()
sage: L([4,2]).verschiebung(2)
-L[2, 1]
sage: L([2,4]).verschiebung(2)
-L[1, 2]
sage: L([6]).verschiebung(2)
-L[3]
sage: L([2,1]).verschiebung(3)
0
sage: L([3]).verschiebung(2)
0
sage: L([]).verschiebung(2)
L[]
sage: L([5, 1]).verschiebung(3)
0
sage: L([5, 1]).verschiebung(6)
0
sage: L([5, 1]).verschiebung(2)
0
sage: L([1, 2, 3, 1]).verschiebung(7)
0
sage: L([7]).verschiebung(7)
L[1]
sage: L([1, 2, 3, 1]).verschiebung(5)
0
sage: (L[1] - L[2] + 2*L[3]).verschiebung(1)
L[1] - L[2] + 2*L[3]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> L = NSym.L()
>>> L([Integer(4),Integer(2)]).verschiebung(Integer(2))
-L[2, 1]
>>> L([Integer(2),Integer(4)]).verschiebung(Integer(2))
-L[1, 2]
>>> L([Integer(6)]).verschiebung(Integer(2))
-L[3]
>>> L([Integer(2),Integer(1)]).verschiebung(Integer(3))
0
>>> L([Integer(3)]).verschiebung(Integer(2))
0
>>> L([]).verschiebung(Integer(2))
L[]
>>> L([Integer(5), Integer(1)]).verschiebung(Integer(3))
0
>>> L([Integer(5), Integer(1)]).verschiebung(Integer(6))
0
>>> L([Integer(5), Integer(1)]).verschiebung(Integer(2))
0
>>> L([Integer(1), Integer(2), Integer(3), Integer(1)]).
↪verschiebung(Integer(7))
```

(continues on next page)

(continued from previous page)

```
0
>>> L([Integer(7)]).verschiebung(Integer(7))
L[1]
>>> L([Integer(1), Integer(2), Integer(3), Integer(1)]).
↪verschiebung(Integer(5))
0
>>> (L[Integer(1)] - L[Integer(2)] + Integer(2)*L[Integer(3)]).
↪verschiebung(Integer(1))
L[1] - L[2] + 2*L[3]
```

I

alias of *Immaculate*

class Immaculate(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The immaculate basis of the non-commutative symmetric functions.

The immaculate basis first appears in Berg, Bergeron, Saliola, Serrano and Zabrocki's [BBSSZ2012]. It can be described as the family $(\mathfrak{S}_\alpha)$, where α runs over all compositions, and $\mathfrak{S}_\alpha$ denotes the immaculate function corresponding to α (see *immaculate_function()*).

If α is a composition $(\alpha_1, \alpha_2, \ldots, \alpha_m)$, then

$$\mathfrak{S}_\alpha = \sum_{\sigma \in S_m} (-1)^\sigma S_{\alpha_1+\sigma(1)-1} S_{\alpha_2+\sigma(2)-2} \cdots S_{\alpha_m+\sigma(m)-m}.$$

Warning

This *basis* contains only the immaculate functions indexed by compositions (i.e., finite sequences of positive integers). To obtain the remaining immaculate functions (sensu lato), use the *immaculate_function()* method. Calling the immaculate *basis* with a list which is not a composition will currently return garbage!

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: I = NCSF.I()
sage: I([1,3,2])*I([1])
I[1, 3, 2, 1] + I[1, 3, 3] + I[1, 4, 2] + I[2, 3, 2]
sage: I([1])*I([1,3,2])
I[1, 1, 3, 2] - I[2, 2, 1, 2] - I[2, 2, 2, 1] - I[2, 2, 3] - I[3, 2, 2]
sage: I([1,3])*I([1,1])
I[1, 3, 1, 1] + I[1, 4, 1] + I[2, 3, 1] + I[2, 4]
sage: I([3,1])*I([2,1])
I[3, 1, 2, 1] + I[3, 2, 1, 1] + I[3, 2, 2] + I[3, 3, 1] + I[4, 1, 1, 1] + I[4,
↪ 1, 2] + 2*I[4, 2, 1] + I[4, 3] + I[5, 1, 1] + I[5, 2]
sage: R = NCSF.ribbon()
sage: I(R[1,3,1])
I[1, 3, 1] + I[2, 2, 1] + I[2, 3] + I[3, 1, 1] + I[3, 2]
sage: R(I(R([2,1,3])))
R[2, 1, 3]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> I = NCSF.I()
>>> I([Integer(1),Integer(3),Integer(2)])*I([Integer(1)])
I[1, 3, 2, 1] + I[1, 3, 3] + I[1, 4, 2] + I[2, 3, 2]
>>> I([Integer(1)])*I([Integer(1),Integer(3),Integer(2)])
I[1, 1, 3, 2] - I[2, 2, 1, 2] - I[2, 2, 2, 1] - I[2, 2, 3] - I[3, 2, 2]
>>> I([Integer(1),Integer(3)])*I([Integer(1),Integer(1)])
I[1, 3, 1, 1] + I[1, 4, 1] + I[2, 3, 1] + I[2, 4]
>>> I([Integer(3),Integer(1)])*I([Integer(2),Integer(1)])
I[3, 1, 2, 1] + I[3, 2, 1, 1] + I[3, 2, 2] + I[3, 3, 1] + I[4, 1, 1, 1] + I[4,
↪ 1, 2] + 2*I[4, 2, 1] + I[4, 3] + I[5, 1, 1] + I[5, 2]
>>> R = NCSF.ribbon()
>>> I(R[Integer(1),Integer(3),Integer(1)])
I[1, 3, 1] + I[2, 2, 1] + I[2, 3] + I[3, 1, 1] + I[3, 2]
>>> R(I(R([Integer(2),Integer(1),Integer(3)])))
R[2, 1, 3]
```

class Element

Bases: `IndexedFreeModuleElement`

An element in the Immaculate basis.

bernstein_creation_operator(*n*)

Return the image of `self` under the n-th Bernstein creation operator.

Let n be an integer. The n-th Bernstein creation operator $\mathbb{B}_n$ is defined as the endomorphism of the space $NSym$ of noncommutative symmetric functions given by

$$\mathbb{B}_n I_{(\alpha_1,\alpha_2,\ldots,\alpha_m)} = I_{(n,\alpha_1,\alpha_2,\ldots,\alpha_m)},$$

where $I_{(\alpha_1,\alpha_2,\ldots,\alpha_m)}$ is the immaculate function associated to the m-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_m) \in \mathbf{Z}^m$.

This has been introduced in [BBSSZ2012], section 3.1, in analogy to the Bernstein creation operators on the symmetric functions.

For more information on the n-th Bernstein creation operator, see *bernstein_creation_operator()*.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: I = NSym.I()
sage: b = I[1,3,2,1]
sage: b.bernstein_creation_operator(3)
I[3, 1, 3, 2, 1]
sage: b.bernstein_creation_operator(5)
I[5, 1, 3, 2, 1]
sage: elt = b + 3*I[4,1,2]
sage: elt.bernstein_creation_operator(1)
I[1, 1, 3, 2, 1] + 3*I[1, 4, 1, 2]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> I = NSym.I()
```

(continues on next page)

(continued from previous page)

```
>>> b = I[Integer(1),Integer(3),Integer(2),Integer(1)]
>>> b.bernstein_creation_operator(Integer(3))
I[3, 1, 3, 2, 1]
>>> b.bernstein_creation_operator(Integer(5))
I[5, 1, 3, 2, 1]
>>> elt = b + Integer(3)*I[Integer(4),Integer(1),Integer(2)]
>>> elt.bernstein_creation_operator(Integer(1))
I[1, 1, 3, 2, 1] + 3*I[1, 4, 1, 2]
```

We check that this agrees with the definition on the Complete basis:

```
sage: S = NSym.S()
sage: S(elt).bernstein_creation_operator(1) == S(elt.bernstein_
↪creation_operator(1))
True
```

```
>>> from sage.all import *
>>> S = NSym.S()
>>> S(elt).bernstein_creation_operator(Integer(1)) == S(elt.bernstein_
↪creation_operator(Integer(1)))
True
```

Check on nonpositive values of n:

```
sage: I[2,2,2].bernstein_creation_operator(-1)
I[1, 1, 1, 2] + I[1, 1, 2, 1] + I[1, 2, 1, 1] - I[1, 2, 2]
sage: I[2,3,2].bernstein_creation_operator(0)
-I[1, 1, 3, 2] - I[1, 2, 2, 2] - I[1, 2, 3, 1] + I[2, 3, 2]
```

```
>>> from sage.all import *
>>> I[Integer(2),Integer(2),Integer(2)].bernstein_creation_operator(-
↪Integer(1))
I[1, 1, 1, 2] + I[1, 1, 2, 1] + I[1, 2, 1, 1] - I[1, 2, 2]
>>> I[Integer(2),Integer(3),Integer(2)].bernstein_creation_
↪operator(Integer(0))
-I[1, 1, 3, 2] - I[1, 2, 2, 2] - I[1, 2, 3, 1] + I[2, 3, 2]
```

dual()

Return the dual basis to the Immaculate basis of NCSF.

The basis returned is the dualImmaculate basis of QSym.

OUTPUT: the dualImmaculate basis of the quasi-symmetric functions

EXAMPLES:

```
sage: I = NonCommutativeSymmetricFunctions(QQ).Immaculate()
sage: I.dual()
Quasisymmetric functions over the Rational Field in the dualImmaculate
basis
```

```
>>> from sage.all import *
>>> I = NonCommutativeSymmetricFunctions(QQ).Immaculate()
```

(continues on next page)

(continued from previous page)

```
>>> I.dual()
Quasisymmetric functions over the Rational Field in the dualImmaculate
basis
```

L

alias of *Elementary*

class Monomial(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The monomial basis defined in Tevlin's paper [Tev2007].

The monomial basis is well-defined only when the base ring is a **Q**-algebra. It is the basis denoted by $(M^I)_I$ in [Tev2007].

class MultiplicativeBases(*parent_with_realization*)

Bases: Category_realization_of_parent

Category of multiplicative bases of non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBases()
Category of multiplicative bases of Non-Commutative Symmetric Functions over␣
↪the Rational Field
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> N.MultiplicativeBases()
Category of multiplicative bases of Non-Commutative Symmetric Functions over␣
↪the Rational Field
```

The complete basis is a multiplicative basis, but the ribbon basis is not:

```
sage: N.Complete() in N.MultiplicativeBases()
True
sage: N.Ribbon() in N.MultiplicativeBases()
False
```

```
>>> from sage.all import *
>>> N.Complete() in N.MultiplicativeBases()
True
>>> N.Ribbon() in N.MultiplicativeBases()
False
```

class ParentMethods

Bases: object

algebra_generators()

Return the algebra generators of a given multiplicative basis of non-commutative symmetric functions.

OUTPUT: the family of generators of the multiplicative basis `self`

EXAMPLES:

```
sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
sage: f = Psi.algebra_generators()
sage: f
Lazy family (<lambda>(i))_{i in Positive integers}
sage: f[1], f[2], f[3]
(Psi[1], Psi[2], Psi[3])
```

```
>>> from sage.all import *
>>> Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
>>> f = Psi.algebra_generators()
>>> f
Lazy family (<lambda>(i))_{i in Positive integers}
>>> f[Integer(1)], f[Integer(2)], f[Integer(3)]
(Psi[1], Psi[2], Psi[3])
```

algebra_morphism(*on_generators*, ***keywords*)

Given a map defined on the generators of the multiplicative basis `self`, return the algebra morphism that extends this map to the whole algebra of non-commutative symmetric functions.

INPUT:

- `on_generators` – a function defined on the index set of the generators (that is, on the positive integers)
- `anti` – boolean (default: `False`)
- `category` – a category (default: `None`)

OUTPUT:

- The algebra morphism of `self` which is defined by `on_generators` in the basis `self`. When `anti` is set to `True`, an algebra anti-morphism is computed instead of an algebra morphism.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: Psi = NCSF.Psi()
sage: double = lambda i: Psi[i,i]
sage: f = Psi.algebra_morphism(double, codomain = Psi)
sage: f
Generic endomorphism of Non-Commutative Symmetric Functions over the
↪Rational Field in the Psi basis
sage: f(2*Psi[[]] + 3 * Psi[1,3,2] + Psi[2,4] )
2*Psi[] + 3*Psi[1, 1, 3, 3, 2, 2] + Psi[2, 2, 4, 4]
sage: f.category()
Category of endsets of unital magmas and right modules over Rational
↪Field and left modules over Rational Field
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> Psi = NCSF.Psi()
>>> double = lambda i: Psi[i,i]
>>> f = Psi.algebra_morphism(double, codomain = Psi)
>>> f
Generic endomorphism of Non-Commutative Symmetric Functions over the
↪Rational Field in the Psi basis
>>> f(Integer(2)*Psi[[]] + Integer(3) * Psi[Integer(1),Integer(3),
↪Integer(2)] + Psi[Integer(2),Integer(4)] )
2*Psi[] + 3*Psi[1, 1, 3, 3, 2, 2] + Psi[2, 2, 4, 4]
```

(continues on next page)

(continued from previous page)

```
>>> f.category()
Category of endsets of unital magmas and right modules over Rational␣
↪Field and left modules over Rational Field
```

When extra properties about the morphism are known, one can specify the category of which it is a morphism:

```
sage: negate = lambda i: -Psi[i]
sage: f = Psi.algebra_morphism(negate, codomain = Psi, category =␣
↪GradedHopfAlgebrasWithBasis(QQ))
sage: f
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Psi basis
sage: f(2*Psi[[]] + 3 * Psi[1,3,2] + Psi[2,4] )
2*Psi[] - 3*Psi[1, 3, 2] + Psi[2, 4]
sage: f.category()
Category of endsets of Hopf algebras over Rational Field and graded␣
↪modules over Rational Field
```

```
>>> from sage.all import *
>>> negate = lambda i: -Psi[i]
>>> f = Psi.algebra_morphism(negate, codomain = Psi, category =␣
↪GradedHopfAlgebrasWithBasis(QQ))
>>> f
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Psi basis
>>> f(Integer(2)*Psi[[]] + Integer(3) * Psi[Integer(1),Integer(3),
↪Integer(2)] + Psi[Integer(2),Integer(4)] )
2*Psi[] - 3*Psi[1, 3, 2] + Psi[2, 4]
>>> f.category()
Category of endsets of Hopf algebras over Rational Field and graded␣
↪modules over Rational Field
```

If `anti` is true, this returns an anti-algebra morphism:

```
sage: f = Psi.algebra_morphism(double, codomain = Psi, anti=True)
sage: f
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Psi basis
sage: f(2*Psi[[]] + 3 * Psi[1,3,2] + Psi[2,4] )
2*Psi[] + 3*Psi[2, 2, 3, 3, 1, 1] + Psi[4, 4, 2, 2]
sage: f.category()
Category of endsets of modules with basis over Rational Field
```

```
>>> from sage.all import *
>>> f = Psi.algebra_morphism(double, codomain = Psi, anti=True)
>>> f
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
↪Rational Field in the Psi basis
>>> f(Integer(2)*Psi[[]] + Integer(3) * Psi[Integer(1),Integer(3),
↪Integer(2)] + Psi[Integer(2),Integer(4)] )
2*Psi[] + 3*Psi[2, 2, 3, 3, 1, 1] + Psi[4, 4, 2, 2]
```

(continues on next page)

(continued from previous page)

```
>>> f.category()
Category of endsets of modules with basis over Rational Field
```

antipode()

Return the antipode morphism on the basis `self`.

The antipode of $NSym$ is closely related to the omega involution; see `omega_involution()` for the latter.

OUTPUT:

- The antipode module map from non-commutative symmetric functions on basis `self`.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.antipode
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
→Rational Field in the Complete basis
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S.antipode
Generic endomorphism of Non-Commutative Symmetric Functions over the␣
→Rational Field in the Complete basis
```

coproduct()

Return the coproduct morphism in the basis `self`.

OUTPUT:

- The coproduct module map from non-commutative symmetric functions on basis `self`.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.coproduct
Generic morphism:
  From: Non-Commutative Symmetric Functions over the Rational Field in␣
→the Complete basis
  To:   Non-Commutative Symmetric Functions over the Rational Field in␣
→the Complete basis # Non-Commutative Symmetric Functions over the␣
→Rational Field in the Complete basis
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S.coproduct
Generic morphism:
  From: Non-Commutative Symmetric Functions over the Rational Field in␣
→the Complete basis
  To:   Non-Commutative Symmetric Functions over the Rational Field in␣
→the Complete basis # Non-Commutative Symmetric Functions over the␣
→Rational Field in the Complete basis
```

product_on_basis(*composition1*, *composition2*)

Return the product of two basis elements from the multiplicative basis. Multiplication is just concatenation on compositions.

INPUT:

- `composition1`, `composition2` – integer compositions

OUTPUT:

- The product of the two non-commutative symmetric functions indexed by `composition1` and `composition2` in the multiplicative basis `self`. This will be again a non-commutative symmetric function.

EXAMPLES:

```
sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
sage: Psi[3,1,2] * Psi[4,2] == Psi[3,1,2,4,2]
True
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.product_on_basis(Composition([2,1]), Composition([1,2]))
S[2, 1, 1, 2]
```

```
>>> from sage.all import *
>>> Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
>>> Psi[Integer(3),Integer(1),Integer(2)] * Psi[Integer(4),Integer(2)]␣
↪== Psi[Integer(3),Integer(1),Integer(2),Integer(4),Integer(2)]
True
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> S.product_on_basis(Composition([Integer(2),Integer(1)]),␣
↪Composition([Integer(1),Integer(2)]))
S[2, 1, 1, 2]
```

to_symmetric_function()

Morphism to the algebra of symmetric functions.

This is constructed by extending the algebra morphism by the image of the generators.

OUTPUT:

- The module morphism from the basis `self` to the symmetric functions which corresponds to taking a commutative image.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.to_symmetric_function(S[1,3])
h[3, 1]
sage: Phi = N.Phi()
sage: Phi.to_symmetric_function(Phi[1,3])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
sage: Psi = N.Psi()
sage: Psi.to_symmetric_function(Psi[1,3])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> S.to_symmetric_function(S[Integer(1),Integer(3)])
h[3, 1]
>>> Phi = N.Phi()
>>> Phi.to_symmetric_function(Phi[Integer(1),Integer(3)])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

(continues on next page)

(continued from previous page)

```
>>> Psi = N.Psi()
>>> Psi.to_symmetric_function(Psi[Integer(1),Integer(3)])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

to_symmetric_function_on_generators(*i*)

Morphism of the generators to symmetric functions.

This is constructed by coercion to the complete basis and applying the morphism.

OUTPUT:

- The module morphism from the basis `self` to the symmetric functions which corresponds to taking a commutative image.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = N.Phi()
sage: Phi.to_symmetric_function_on_generators(3)
h[1, 1, 1] - 3*h[2, 1] + 3*h[3]
sage: Phi.to_symmetric_function_on_generators(0)
h[]
sage: Psi = N.Psi()
sage: Psi.to_symmetric_function_on_generators(3)
h[1, 1, 1] - 3*h[2, 1] + 3*h[3]
sage: L = N.elementary()
sage: L.to_symmetric_function_on_generators(3)
h[1, 1, 1] - 2*h[2, 1] + h[3]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> Phi = N.Phi()
>>> Phi.to_symmetric_function_on_generators(Integer(3))
h[1, 1, 1] - 3*h[2, 1] + 3*h[3]
>>> Phi.to_symmetric_function_on_generators(Integer(0))
h[]
>>> Psi = N.Psi()
>>> Psi.to_symmetric_function_on_generators(Integer(3))
h[1, 1, 1] - 3*h[2, 1] + 3*h[3]
>>> L = N.elementary()
>>> L.to_symmetric_function_on_generators(Integer(3))
h[1, 1, 1] - 2*h[2, 1] + h[3]
```

super_categories()

Return the super categories of the category of multiplicative bases of the non-commutative symmetric functions.

OUTPUT: list

class MultiplicativeBasesOnGroupLikeElements(*parent_with_realization*)

Bases: `Category_realization_of_parent`

Category of multiplicative bases on grouplike elements of non-commutative symmetric functions.

Here, a "multiplicative basis on grouplike elements" means a multiplicative basis whose generators

$(f_1, f_2, f_3, \ldots)$ satisfy

$$\Delta(f_i) = \sum_{j=0}^{i} f_j \otimes f_{i-j}$$

with $f_0 = 1$. (In other words, the generators are to form a divided power sequence in the sense of a coalgebra.) This does not mean that the generators are grouplike, but means that the element $1 + f_1 + f_2 + f_3 + \cdots$ in the completion of the ring of non-commutative symmetric functions with respect to the grading is grouplike.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBasesOnGroupLikeElements()
Category of multiplicative bases on group like elements of Non-Commutative␣
↪Symmetric Functions over the Rational Field
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> N.MultiplicativeBasesOnGroupLikeElements()
Category of multiplicative bases on group like elements of Non-Commutative␣
↪Symmetric Functions over the Rational Field
```

The complete basis is a multiplicative basis, but the ribbon basis is not:

```
sage: N.Complete() in N.MultiplicativeBasesOnGroupLikeElements()
True
sage: N.Ribbon() in N.MultiplicativeBasesOnGroupLikeElements()
False
```

```
>>> from sage.all import *
>>> N.Complete() in N.MultiplicativeBasesOnGroupLikeElements()
True
>>> N.Ribbon() in N.MultiplicativeBasesOnGroupLikeElements()
False
```

class ParentMethods

Bases: `object`

antipode_on_basis(*composition*)

Return the application of the antipode to a basis element.

INPUT:

- `composition` – a composition

OUTPUT:

- The image of the basis element indexed by `composition` under the antipode map.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: S.antipode_on_basis(Composition([2,1]))
-S[1, 1, 1] + S[1, 2]
sage: S[1].antipode() # indirect doctest
-S[1]
sage: S[2].antipode() # indirect doctest
S[1, 1] - S[2]
```

(continues on next page)

(continued from previous page)

```
sage: S[3].antipode() # indirect doctest
-S[1, 1, 1] + S[1, 2] + S[2, 1] - S[3]
sage: S[2,3].coproduct().apply_multilinear_morphism(lambda be,ga:␣
↪S(be)*S(ga).antipode())
0
sage: S[2,3].coproduct().apply_multilinear_morphism(lambda be,ga:␣
↪S(be).antipode()*S(ga))
0
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).complete()
>>> S.antipode_on_basis(Composition([Integer(2),Integer(1)]))
-S[1, 1, 1] + S[1, 2]
>>> S[Integer(1)].antipode() # indirect doctest
-S[1]
>>> S[Integer(2)].antipode() # indirect doctest
S[1, 1] - S[2]
>>> S[Integer(3)].antipode() # indirect doctest
-S[1, 1, 1] + S[1, 2] + S[2, 1] - S[3]
>>> S[Integer(2),Integer(3)].coproduct().apply_multilinear_
↪morphism(lambda be,ga: S(be)*S(ga).antipode())
0
>>> S[Integer(2),Integer(3)].coproduct().apply_multilinear_
↪morphism(lambda be,ga: S(be).antipode()*S(ga))
0
```

coproduct_on_generators(*i*)

Return the image of the i-th generator of the algebra under the coproduct.

INPUT:

- `i` – positive integer

OUTPUT: the result of applying the coproduct to the i-th generator of `self`

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: S.coproduct_on_generators(3)
S[] # S[3] + S[1] # S[2] + S[2] # S[1] + S[3] # S[]
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).complete()
>>> S.coproduct_on_generators(Integer(3))
S[] # S[3] + S[1] # S[2] + S[2] # S[1] + S[3] # S[]
```

super_categories()

Return the super categories of the category of multiplicative bases of group-like elements of the non-commutative symmetric functions.

OUTPUT: list

class MultiplicativeBasesOnPrimitiveElements(*parent_with_realization*)

Bases: `Category_realization_of_parent`

Category of multiplicative bases of the non-commutative symmetric functions whose generators are primitive elements.

An element x of a bialgebra is *primitive* if $\Delta(x) = x \otimes 1 + 1 \otimes x$, where Δ is the coproduct of the bialgebra.

Given a multiplicative basis and knowing the coproducts and antipodes of its generators, one can compute the coproduct and the antipode of any element, since they are respectively algebra morphisms and anti-morphisms. See `antipode_on_generators()` and `coproduct_on_generators()`.

Todo

this could be generalized to any free algebra.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBasesOnPrimitiveElements()
Category of multiplicative bases on primitive elements of Non-Commutative␣
↪Symmetric Functions over the Rational Field
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> N.MultiplicativeBasesOnPrimitiveElements()
Category of multiplicative bases on primitive elements of Non-Commutative␣
↪Symmetric Functions over the Rational Field
```

The Phi and Psi bases are multiplicative bases whose generators are primitive elements, but the complete and ribbon bases are not:

```
sage: N.Phi() in N.MultiplicativeBasesOnPrimitiveElements()
True
sage: N.Psi() in N.MultiplicativeBasesOnPrimitiveElements()
True
sage: N.Complete() in N.MultiplicativeBasesOnPrimitiveElements()
False
sage: N.Ribbon() in N.MultiplicativeBasesOnPrimitiveElements()
False
```

```
>>> from sage.all import *
>>> N.Phi() in N.MultiplicativeBasesOnPrimitiveElements()
True
>>> N.Psi() in N.MultiplicativeBasesOnPrimitiveElements()
True
>>> N.Complete() in N.MultiplicativeBasesOnPrimitiveElements()
False
>>> N.Ribbon() in N.MultiplicativeBasesOnPrimitiveElements()
False
```

class ParentMethods

Bases: `object`

antipode_on_generators(*i*)

Return the image of a generator of a primitive basis of the non-commutative symmetric functions under the antipode map.

INPUT:

- `i` – positive integer

OUTPUT:

The image of the i-th generator of the multiplicative basis `self` under the antipode of the algebra of non-commutative symmetric functions.

EXAMPLES:

```
sage: Psi=NonCommutativeSymmetricFunctions(QQ).Psi()
sage: Psi.antipode_on_generators(2)
-Psi[2]
```

```
>>> from sage.all import *
>>> Psi=NonCommutativeSymmetricFunctions(QQ).Psi()
>>> Psi.antipode_on_generators(Integer(2))
-Psi[2]
```

coproduct_on_generators(*i*)

Return the image of the i-th generator of the multiplicative basis `self` under the coproduct.

INPUT:

- `i` – positive integer

OUTPUT: the result of applying the coproduct to the i-th generator of `self`

EXAMPLES:

```
sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
sage: Psi.coproduct_on_generators(3)
Psi[] # Psi[3] + Psi[3] # Psi[]
```

```
>>> from sage.all import *
>>> Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
>>> Psi.coproduct_on_generators(Integer(3))
Psi[] # Psi[3] + Psi[3] # Psi[]
```

super_categories()

Return the super categories of the category of multiplicative bases of primitive elements of the non-commutative symmetric functions.

OUTPUT: list

class Phi(*NCSF*)

Bases: *CombinatorialFreeModule*, `BindableClass`

The Hopf algebra of non-commutative symmetric functions in the Phi basis.

The Phi basis is defined in Definition 3.4 of [NCSF1], where it is denoted by $(\Phi^I)_I$. It is a multiplicative basis, and is connected to the elementary generators Λ_i of the ring of non-commutative symmetric functions by the following relation: Define a non-commutative symmetric function Φ_n for every positive integer n by the power series identity

$$\sum_{k\geq 1} t^k \frac{1}{k} \Phi_k = -\log\left(\sum_{k\geq 0} (-t)^k \Lambda_k\right),$$

with Λ_0 denoting 1. For every composition $(i_1, i_2, \ldots, i_k)$, we have $\Phi^{(i_1,i_2,\ldots,i_k)} = \Phi_{i_1}\Phi_{i_2}\cdots\Phi_{i_k}$.

The Φ-basis is well-defined only when the base ring is a $\mathbf{Q}$-algebra. The elements of the Φ-basis are known as the "power-sum non-commutative symmetric functions of the second kind".

The generators Φ_n are related to the (first) Eulerian idempotents in the descent algebras of the symmetric groups (see [NCSF1], 5.4 for details).

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NCSF.Phi(); Phi
Non-Commutative Symmetric Functions over the Rational Field in the Phi basis
sage: Phi.an_element()
2*Phi[] + 2*Phi[1] + 3*Phi[1, 1]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> Phi = NCSF.Phi(); Phi
Non-Commutative Symmetric Functions over the Rational Field in the Phi basis
>>> Phi.an_element()
2*Phi[] + 2*Phi[1] + 3*Phi[1, 1]
```

class Element

Bases: `IndexedFreeModuleElement`

psi_involution()

Return the image of the noncommutative symmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $NCSF \to NCSF$ which, for every composition I, sends the complete noncommutative symmetric function S^I to the elementary noncommutative symmetric function Λ^I. It can be shown that every composition I satisfies

$$\psi(R_I) = R_{I^c}, \quad \psi(S^I) = \Lambda^I, \quad \psi(\Lambda^I) = S^I, \quad \psi(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^I$$

where I^c denotes the complement of the composition I, and $\ell(I)$ denotes the length of I, and where standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, Φ for the basis of the power sums of the second kind, and R for the ribbon basis). The map ψ is an involution and a graded Hopf algebra automorphism of $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(\psi(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The involution ψ of $NCSF$ is adjoint to the involution ψ of $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also

psi involution of NCSF, *psi involution of QSym*, *star involution of NCSF*.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi[3,2].psi_involution()
```

(continues on next page)

(continued from previous page)

```
-Phi[3, 2]
sage: Phi[2,2].psi_involution()
Phi[2, 2]
sage: Phi[[]].psi_involution()
Phi[]
sage: (Phi[2,1] - 2*Phi[2]).psi_involution()
2*Phi[2] - Phi[2, 1]
sage: Phi(0).psi_involution()
0
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> Phi = NSym.Phi()
>>> Phi[Integer(3),Integer(2)].psi_involution()
-Phi[3, 2]
>>> Phi[Integer(2),Integer(2)].psi_involution()
Phi[2, 2]
>>> Phi[[]].psi_involution()
Phi[]
>>> (Phi[Integer(2),Integer(1)] - Integer(2)*Phi[Integer(2)]).psi_
↪involution()
2*Phi[2] - Phi[2, 1]
>>> Phi(Integer(0)).psi_involution()
0
```

The implementation at hand is tailored to the Phi basis. It is equivalent to the generic implementation via the ribbon basis:

```
sage: R = NSym.R()
sage: all( R(Phi[I].psi_involution()) == R(Phi[I]).psi_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> R = NSym.R()
>>> all( R(Phi[I].psi_involution()) == R(Phi[I]).psi_involution()
...      for I in Compositions(Integer(4)) )
True
```

star_involution()

Return the image of the noncommutative symmetric function `self` under the star involution.

The star involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to S_n. Denoting by f^* the image of an element $f \in NCSF$ under this star involution, it can be shown that every composition I satisfies

$$(S^I)^* = S^{I^r}, \quad (\Lambda^I)^* = \Lambda^{I^r}, \quad R_I^* = R_{I^r}, \quad (\Phi^I)^* = \Phi^{I^r},$$

where I^r denotes the reversed composition of I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, and Φ for that of the power-sums of the second kind). The star involution is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. Under the canonical isomorphism between the n-th graded component of $NCSF$ and the descent algebra of the symmetric group S_n (see `to_descent_algebra()`), the star involution (restricted to the n-th graded component) corresponds to the automorphism of the descent algebra given by $x \mapsto \omega_n x \omega_n$, where ω_n is the permutation $(n, n-1, \ldots, 1) \in S_n$ (written here in one-line notation). If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(f^*) = \pi(f)$ for every $f \in NCSF$.

The star involution on $NCSF$ is adjoint to the star involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6. See [NCSF2], section 2.3 for the properties of this map.

See also:

`star involution of NCSF`, `psi involution of NCSF`, `star involution of QSym`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi[3,1,1,4].star_involution()
Phi[4, 1, 1, 3]
sage: Phi[4,2,1].star_involution()
Phi[1, 2, 4]
sage: (Phi[1,4] - Phi[2,3] + 2*Phi[5,4] - 3*Phi[3] + 4*Phi[[]]).star_
↪involution()
4*Phi[] - 3*Phi[3] - Phi[3, 2] + Phi[4, 1] + 2*Phi[4, 5]
sage: (Phi[3,3] + 3*Phi[1]).star_involution()
3*Phi[1] + Phi[3, 3]
sage: Phi([2,1]).star_involution()
Phi[1, 2]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(QQ)
>>> Phi = NSym.Phi()
>>> Phi[Integer(3),Integer(1),Integer(1),Integer(4)].star_involution()
Phi[4, 1, 1, 3]
>>> Phi[Integer(4),Integer(2),Integer(1)].star_involution()
Phi[1, 2, 4]
>>> (Phi[Integer(1),Integer(4)] - Phi[Integer(2),Integer(3)] +␣
↪Integer(2)*Phi[Integer(5),Integer(4)] - Integer(3)*Phi[Integer(3)] +␣
↪Integer(4)*Phi[[]]).star_involution()
4*Phi[] - 3*Phi[3] - Phi[3, 2] + Phi[4, 1] + 2*Phi[4, 5]
>>> (Phi[Integer(3),Integer(3)] + Integer(3)*Phi[Integer(1)]).star_
↪involution()
3*Phi[1] + Phi[3, 3]
>>> Phi([Integer(2),Integer(1)]).star_involution()
Phi[1, 2]
```

The implementation at hand is tailored to the Phi basis. It is equivalent to the generic implementation via the complete basis:

```
sage: S = NSym.S()
sage: all( S(Phi[I].star_involution()) == S(Phi[I]).star_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> S = NSym.S()
>>> all( S(Phi[I].star_involution()) == S(Phi[I]).star_involution()
...      for I in Compositions(Integer(4)) )
True
```

verschiebung(*n*)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the **k**-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (*verschiebung()*) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $(\underbrace{n, n, \ldots, n}_{|I|/n \text{ times}})$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also

verschiebung method of NCSF, *adams_operator method of QSym*, *verschiebung method of Sym*

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to `self`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: Phi = NSym.Phi()
sage: Phi([4,2]).verschiebung(2)
4*Phi[2, 1]
sage: Phi([2,4]).verschiebung(2)
4*Phi[1, 2]
sage: Phi([6]).verschiebung(2)
2*Phi[3]
sage: Phi([2,1]).verschiebung(3)
0
sage: Phi([3]).verschiebung(2)
0
sage: Phi([]).verschiebung(2)
Phi[]
sage: Phi([5, 1]).verschiebung(3)
0
sage: Phi([5, 1]).verschiebung(6)
0
sage: Phi([5, 1]).verschiebung(2)
0
sage: Phi([1, 2, 3, 1]).verschiebung(7)
0
sage: Phi([7]).verschiebung(7)
7*Phi[1]
sage: Phi([1, 2, 3, 1]).verschiebung(5)
0
sage: (Phi[1] - Phi[2] + 2*Phi[3]).verschiebung(1)
Phi[1] - Phi[2] + 2*Phi[3]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> Phi = NSym.Phi()
>>> Phi([Integer(4),Integer(2)]).verschiebung(Integer(2))
4*Phi[2, 1]
>>> Phi([Integer(2),Integer(4)]).verschiebung(Integer(2))
4*Phi[1, 2]
>>> Phi([Integer(6)]).verschiebung(Integer(2))
2*Phi[3]
>>> Phi([Integer(2),Integer(1)]).verschiebung(Integer(3))
0
>>> Phi([Integer(3)]).verschiebung(Integer(2))
0
>>> Phi([]).verschiebung(Integer(2))
Phi[]
>>> Phi([Integer(5), Integer(1)]).verschiebung(Integer(3))
0
>>> Phi([Integer(5), Integer(1)]).verschiebung(Integer(6))
```

(continues on next page)

(continued from previous page)

```
0
>>> Phi([Integer(5), Integer(1)]).verschiebung(Integer(2))
0
>>> Phi([Integer(1), Integer(2), Integer(3), Integer(1)]).
↪verschiebung(Integer(7))
0
>>> Phi([Integer(7)]).verschiebung(Integer(7))
7*Phi[1]
>>> Phi([Integer(1), Integer(2), Integer(3), Integer(1)]).
↪verschiebung(Integer(5))
0
>>> (Phi[Integer(1)] - Phi[Integer(2)] + Integer(2)*Phi[Integer(3)]).
↪verschiebung(Integer(1))
Phi[1] - Phi[2] + 2*Phi[3]
```

class Psi(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The Hopf algebra of non-commutative symmetric functions in the Psi basis.

The Psi basis is defined in Definition 3.4 of [NCSF1], where it is denoted by $(\Psi^I)_I$. It is a multiplicative basis, and is connected to the elementary generators Λ_i of the ring of non-commutative symmetric functions by the following relation: Define a non-commutative symmetric function Ψ_n for every positive integer n by the power series identity

$$\frac{d}{dt}\sigma(t) = \sigma(t) \cdot \left(\sum_{k \geq 1} t^{k-1} \Psi_k \right),$$

where

$$\sigma(t) = \left(\sum_{k \geq 0} (-t)^k \Lambda_k \right)^{-1}$$

and where Λ_0 denotes 1. For every composition $(i_1, i_2, \ldots, i_k)$, we have $\Psi^{(i_1,i_2,\ldots,i_k)} = \Psi_{i_1}\Psi_{i_2}\cdots\Psi_{i_k}$.

The Ψ-basis is a basis only when the base ring is a **Q**-algebra (although the Ψ^I can be defined over any base ring). The elements of the Ψ-basis are known as the "power-sum non-commutative symmetric functions of the first kind". The generators Ψ_n correspond to the Dynkin (quasi-)idempotents in the descent algebras of the symmetric groups (see [NCSF1], 5.2 for details).

Another (equivalent) definition of Ψ_n is

$$\Psi_n = \sum_{i=0}^{n-1} (-1)^i R_{1^i,n-i},$$

where R denotes the ribbon basis of $NCSF$, and where 1^i stands for i repetitions of the integer 1.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: Psi = NCSF.Psi(); Psi
Non-Commutative Symmetric Functions over the Rational Field in the Psi basis
sage: Psi.an_element()
2*Psi[] + 2*Psi[1] + 3*Psi[1, 1]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> Psi = NCSF.Psi(); Psi
Non-Commutative Symmetric Functions over the Rational Field in the Psi basis
>>> Psi.an_element()
2*Psi[] + 2*Psi[1] + 3*Psi[1, 1]
```

Checking the equivalent definition of Ψ_n:

```
sage: def test_psi(n):
....:     NCSF = NonCommutativeSymmetricFunctions(ZZ)
....:     R = NCSF.R()
....:     Psi = NCSF.Psi()
....:     a = R.sum([(-1) ** i * R[[1]*i + [n-i]]
....:                for i in range(n)])
....:     return a == R(Psi[n])
sage: test_psi(2)
True
sage: test_psi(3)
True
sage: test_psi(4)
True
```

```
>>> from sage.all import *
>>> def test_psi(n):
...     NCSF = NonCommutativeSymmetricFunctions(ZZ)
...     R = NCSF.R()
...     Psi = NCSF.Psi()
...     a = R.sum([(-Integer(1)) ** i * R[[Integer(1)]*i + [n-i]]
...                for i in range(n)])
...     return a == R(Psi[n])
>>> test_psi(Integer(2))
True
>>> test_psi(Integer(3))
True
>>> test_psi(Integer(4))
True
```

class Element

Bases: `IndexedFreeModuleElement`

verschiebung(*n*)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the $\mathbf{k}$-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (*verschiebung()*) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $(\underbrace{n, n, \ldots, n}_{|I|/n \text{ times}})$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also:

verschiebung method of NCSF, *adams_operator method of QSym*, *verschiebung method of Sym*

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to `self`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: Psi = NSym.Psi()
sage: Psi([4,2]).verschiebung(2)
4*Psi[2, 1]
sage: Psi([2,4]).verschiebung(2)
4*Psi[1, 2]
sage: Psi([6]).verschiebung(2)
2*Psi[3]
sage: Psi([2,1]).verschiebung(3)
0
sage: Psi([3]).verschiebung(2)
0
sage: Psi([]).verschiebung(2)
Psi[]
sage: Psi([5, 1]).verschiebung(3)
0
sage: Psi([5, 1]).verschiebung(6)
0
sage: Psi([5, 1]).verschiebung(2)
0
sage: Psi([1, 2, 3, 1]).verschiebung(7)
```

(continues on next page)

(continued from previous page)

```
0
sage: Psi([7]).verschiebung(7)
7*Psi[1]
sage: Psi([1, 2, 3, 1]).verschiebung(5)
0
sage: (Psi[1] - Psi[2] + 2*Psi[3]).verschiebung(1)
Psi[1] - Psi[2] + 2*Psi[3]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> Psi = NSym.Psi()
>>> Psi([Integer(4),Integer(2)]).verschiebung(Integer(2))
4*Psi[2, 1]
>>> Psi([Integer(2),Integer(4)]).verschiebung(Integer(2))
4*Psi[1, 2]
>>> Psi([Integer(6)]).verschiebung(Integer(2))
2*Psi[3]
>>> Psi([Integer(2),Integer(1)]).verschiebung(Integer(3))
0
>>> Psi([Integer(3)]).verschiebung(Integer(2))
0
>>> Psi([]).verschiebung(Integer(2))
Psi[]
>>> Psi([Integer(5), Integer(1)]).verschiebung(Integer(3))
0
>>> Psi([Integer(5), Integer(1)]).verschiebung(Integer(6))
0
>>> Psi([Integer(5), Integer(1)]).verschiebung(Integer(2))
0
>>> Psi([Integer(1), Integer(2), Integer(3), Integer(1)]).
→verschiebung(Integer(7))
0
>>> Psi([Integer(7)]).verschiebung(Integer(7))
7*Psi[1]
>>> Psi([Integer(1), Integer(2), Integer(3), Integer(1)]).
→verschiebung(Integer(5))
0
>>> (Psi[Integer(1)] - Psi[Integer(2)] + Integer(2)*Psi[Integer(3)]).
→verschiebung(Integer(1))
Psi[1] - Psi[2] + 2*Psi[3]
```

internal_product_on_basis_by_bracketing(I, J)

The internal product of two elements of the Psi basis.

See *internal_product()* for a thorough documentation of this operation.

This is an implementation using [NCSF2] Lemma 3.10. It is fast when the length of I is small, but can get very slow otherwise. Therefore it is not being used by default for internally multiplying Psi functions.

INPUT:

- I, J – compositions

OUTPUT:

- The internal product of the elements of the Psi basis of $NSym$ indexed by I and J, expressed in the Psi basis.

AUTHORS:

- Travis Scrimshaw, 29 Mar 2014

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: Psi = N.Psi()
sage: Psi.internal_product_on_basis_by_bracketing([2,2],[1,2,1])
0
sage: Psi.internal_product_on_basis_by_bracketing([1,2,1],[2,1,1])
4*Psi[1, 2, 1]
sage: Psi.internal_product_on_basis_by_bracketing([2,1,1],[1,2,1])
4*Psi[2, 1, 1]
sage: Psi.internal_product_on_basis_by_bracketing([1,2,1], [1,1,1,1])
0
sage: Psi.internal_product_on_basis_by_bracketing([3,1], [1,2,1])
-Psi[1, 2, 1] + Psi[2, 1, 1]
sage: Psi.internal_product_on_basis_by_bracketing([1,2,1], [3,1])
0
sage: Psi.internal_product_on_basis_by_bracketing([2,2],[1,2])
0
sage: Psi.internal_product_on_basis_by_bracketing([4], [1,2,1])
-Psi[1, 1, 2] + 2*Psi[1, 2, 1] - Psi[2, 1, 1]
```

```
>>> from sage.all import *
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> Psi = N.Psi()
>>> Psi.internal_product_on_basis_by_bracketing([Integer(2),Integer(2)],
↪[Integer(1),Integer(2),Integer(1)])
0
>>> Psi.internal_product_on_basis_by_bracketing([Integer(1),Integer(2),
↪Integer(1)],[Integer(2),Integer(1),Integer(1)])
4*Psi[1, 2, 1]
>>> Psi.internal_product_on_basis_by_bracketing([Integer(2),Integer(1),
↪Integer(1)],[Integer(1),Integer(2),Integer(1)])
4*Psi[2, 1, 1]
>>> Psi.internal_product_on_basis_by_bracketing([Integer(1),Integer(2),
↪Integer(1)], [Integer(1),Integer(1),Integer(1),Integer(1)])
0
>>> Psi.internal_product_on_basis_by_bracketing([Integer(3),Integer(1)],␣
↪[Integer(1),Integer(2),Integer(1)])
-Psi[1, 2, 1] + Psi[2, 1, 1]
>>> Psi.internal_product_on_basis_by_bracketing([Integer(1),Integer(2),
↪Integer(1)], [Integer(3),Integer(1)])
0
>>> Psi.internal_product_on_basis_by_bracketing([Integer(2),Integer(2)],
↪[Integer(1),Integer(2)])
0
>>> Psi.internal_product_on_basis_by_bracketing([Integer(4)], [Integer(1),
↪Integer(2),Integer(1)])
-Psi[1, 1, 2] + 2*Psi[1, 2, 1] - Psi[2, 1, 1]
```

R

alias of *Ribbon*

class Ribbon(*NCSF*)

Bases: `CombinatorialFreeModule`, `BindableClass`

The Hopf algebra of non-commutative symmetric functions in the Ribbon basis.

The Ribbon basis is defined in Definition 3.12 of [NCSF1], where it is denoted by $(R_I)_I$. It is connected to the complete basis of the ring of non-commutative symmetric functions by the following relation: For every composition I, we have

$$R_I = \sum_J (-1)^{\ell(I)-\ell(J)} S^J,$$

where the sum is over all compositions J which are coarser than I and $\ell(I)$ denotes the length of I. (See the proof of Proposition 4.13 in [NCSF1].)

The elements of the Ribbon basis are commonly referred to as the ribbon Schur functions.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: R = NCSF.Ribbon(); R
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon␣
↪basis
sage: R.an_element()
2*R[] + 2*R[1] + 3*R[1, 1]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> R = NCSF.Ribbon(); R
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon␣
↪basis
>>> R.an_element()
2*R[] + 2*R[1] + 3*R[1, 1]
```

The following are aliases for this basis:

```
sage: NCSF.ribbon()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon␣
↪basis
sage: NCSF.R()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon␣
↪basis
```

```
>>> from sage.all import *
>>> NCSF.ribbon()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon␣
↪basis
>>> NCSF.R()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon␣
↪basis
```

class Element

Bases: `IndexedFreeModuleElement`

star_involution()

Return the image of the noncommutative symmetric function `self` under the star involution.

The star involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to S_n. Denoting by f^* the image of an element $f \in NCSF$ under this star involution, it can be shown that every composition I satisfies

$$(S^I)^* = S^{I^r}, \quad (\Lambda^I)^* = \Lambda^{I^r}, \quad R_I^* = R_{I^r}, \quad (\Phi^I)^* = \Phi^{I^r},$$

where I^r denotes the reversed composition of I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, and Φ for that of the power-sums of the second kind). The star involution is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. Under the canonical isomorphism between the n-th graded component of $NCSF$ and the descent algebra of the symmetric group S_n (see `to_descent_algebra()`), the star involution (restricted to the n-th graded component) corresponds to the automorphism of the descent algebra given by $x \mapsto \omega_n x \omega_n$, where ω_n is the permutation $(n, n-1, \ldots, 1) \in S_n$ (written here in one-line notation). If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(f^*) = \pi(f)$ for every $f \in NCSF$.

The star involution on $NCSF$ is adjoint to the star involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6. See [NCSF2], section 2.3 for the properties of this map.

See also

`star involution of NCSF`, `star involution of QSym`, `psi involution of NCSF`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: R = NSym.R()
sage: R[3,1,4,2].star_involution()
R[2, 4, 1, 3]
sage: R[4,1,2].star_involution()
R[2, 1, 4]
sage: (R[1] - R[2] + 2*R[5,4] - 3*R[3] + 4*R[[]]).star_involution()
4*R[] + R[1] - R[2] - 3*R[3] + 2*R[4, 5]
sage: (R[3,3] - 21*R[1]).star_involution()
-21*R[1] + R[3, 3]
sage: R([14,1]).star_involution()
R[1, 14]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> R = NSym.R()
>>> R[Integer(3),Integer(1),Integer(4),Integer(2)].star_involution()
R[2, 4, 1, 3]
>>> R[Integer(4),Integer(1),Integer(2)].star_involution()
R[2, 1, 4]
>>> (R[Integer(1)] - R[Integer(2)] + Integer(2)*R[Integer(5),
↪Integer(4)] - Integer(3)*R[Integer(3)] + Integer(4)*R[[]]).star_
↪involution()
4*R[] + R[1] - R[2] - 3*R[3] + 2*R[4, 5]
```

(continues on next page)

(continued from previous page)

```
>>> (R[Integer(3),Integer(3)] - Integer(21)*R[Integer(1)]).star_
↪involution()
-21*R[1] + R[3, 3]
>>> R([Integer(14),Integer(1)]).star_involution()
R[1, 14]
```

The implementation at hand is tailored to the ribbon basis. It is equivalent to the generic implementation via the complete basis:

```
sage: S = NSym.S()
sage: all( S(R[I].star_involution()) == S(R[I]).star_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> S = NSym.S()
>>> all( S(R[I].star_involution()) == S(R[I]).star_involution()
...      for I in Compositions(Integer(4)) )
True
```

verschiebung(n)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the **k**-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (`verschiebung()`) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $(\underbrace{n, n, \ldots, n}_{|I|/n \text{ times}})$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also

verschiebung method of NCSF, *adams_operator method of QSym*, *verschiebung method of Sym*

INPUT:

- n – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to self.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: R = NSym.R()
sage: R([4,2]).verschiebung(2)
R[2, 1]
sage: R([2,1]).verschiebung(3)
-R[1]
sage: R([3]).verschiebung(2)
0
sage: R([]).verschiebung(2)
R[]
sage: R([5, 1]).verschiebung(3)
-R[2]
sage: R([5, 1]).verschiebung(6)
-R[1]
sage: R([5, 1]).verschiebung(2)
-R[3]
sage: R([1, 2, 3, 1]).verschiebung(7)
-R[1]
sage: R([1, 2, 3, 1]).verschiebung(5)
0
sage: (R[1] - R[2] + 2*R[3]).verschiebung(1)
R[1] - R[2] + 2*R[3]
```

```
>>> from sage.all import *
>>> NSym = NonCommutativeSymmetricFunctions(ZZ)
>>> R = NSym.R()
>>> R([Integer(4),Integer(2)]).verschiebung(Integer(2))
R[2, 1]
>>> R([Integer(2),Integer(1)]).verschiebung(Integer(3))
-R[1]
>>> R([Integer(3)]).verschiebung(Integer(2))
0
>>> R([]).verschiebung(Integer(2))
R[]
>>> R([Integer(5), Integer(1)]).verschiebung(Integer(3))
-R[2]
>>> R([Integer(5), Integer(1)]).verschiebung(Integer(6))
-R[1]
>>> R([Integer(5), Integer(1)]).verschiebung(Integer(2))
```

(continues on next page)

(continued from previous page)

```
-R[3]
>>> R([Integer(1), Integer(2), Integer(3), Integer(1)]).
↪verschiebung(Integer(7))
-R[1]
>>> R([Integer(1), Integer(2), Integer(3), Integer(1)]).
↪verschiebung(Integer(5))
0
>>> (R[Integer(1)] - R[Integer(2)] + Integer(2)*R[Integer(3)]).
↪verschiebung(Integer(1))
R[1] - R[2] + 2*R[3]
```

antipode_on_basis(*composition*)

Return the application of the antipode to a basis element of the ribbon basis `self`.

INPUT:

- `composition` – a composition

OUTPUT:

- The image of the basis element indexed by `composition` under the antipode map.

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: R.antipode_on_basis(Composition([2,1]))
-R[2, 1]
sage: R[3,1].antipode() # indirect doctest
R[2, 1, 1]
sage: R[[]].antipode() # indirect doctest
R[]
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).ribbon()
>>> R.antipode_on_basis(Composition([Integer(2),Integer(1)]))
-R[2, 1]
>>> R[Integer(3),Integer(1)].antipode() # indirect doctest
R[2, 1, 1]
>>> R[[]].antipode() # indirect doctest
R[]
```

We check that the implementation of the antipode at hand does not contradict the generic one:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: all( S(R[I].antipode()) == S(R[I]).antipode()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> S = NonCommutativeSymmetricFunctions(QQ).S()
>>> all( S(R[I].antipode()) == S(R[I]).antipode()
...      for I in Compositions(Integer(4)) )
True
```

dual()

Return the dual basis to the ribbon basis of the non-commutative symmetric functions. This is the Fundamental basis of the quasi-symmetric functions.

OUTPUT: the fundamental basis of the quasi-symmetric functions

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: R.dual()
Quasisymmetric functions over the Rational Field in the Fundamental basis
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).ribbon()
>>> R.dual()
Quasisymmetric functions over the Rational Field in the Fundamental basis
```

product_on_basis(*I*, *J*)

Return the product of two ribbon basis elements of the non-commutative symmetric functions.

INPUT:

- I, J – compositions

OUTPUT: the product of the ribbon functions indexed by I and J

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: R[1,2,1] * R[3,1]
R[1, 2, 1, 3, 1] + R[1, 2, 4, 1]
sage: ( R[1,2] + R[3] ) * ( R[3,1] + R[1,2,1] )
R[1, 2, 1, 2, 1] + R[1, 2, 3, 1] + R[1, 3, 2, 1] + R[1, 5, 1] + R[3, 1, 2,
↪ 1] + R[3, 3, 1] + R[4, 2, 1] + R[6, 1]
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).ribbon()
>>> R[Integer(1),Integer(2),Integer(1)] * R[Integer(3),Integer(1)]
R[1, 2, 1, 3, 1] + R[1, 2, 4, 1]
>>> ( R[Integer(1),Integer(2)] + R[Integer(3)] ) * ( R[Integer(3),
↪Integer(1)] + R[Integer(1),Integer(2),Integer(1)] )
R[1, 2, 1, 2, 1] + R[1, 2, 3, 1] + R[1, 3, 2, 1] + R[1, 5, 1] + R[3, 1, 2,
↪ 1] + R[3, 3, 1] + R[4, 2, 1] + R[6, 1]
```

to_symmetric_function_on_basis(*I*)

Return the commutative image of a ribbon basis element of the non-commutative symmetric functions.

INPUT:

- I – a composition

OUTPUT:

- The commutative image of the ribbon basis element indexed by I. This will be expressed as a symmetric function in the Schur basis.

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: R.to_symmetric_function_on_basis(Composition([3,1,1]))
s[3, 1, 1]
sage: R.to_symmetric_function_on_basis(Composition([4,2,1]))
s[4, 2, 1] + s[5, 1, 1] + s[5, 2]
sage: R.to_symmetric_function_on_basis(Composition([]))
s[]
```

```
>>> from sage.all import *
>>> R = NonCommutativeSymmetricFunctions(QQ).R()
>>> R.to_symmetric_function_on_basis(Composition([Integer(3),Integer(1),
↪Integer(1)]))
s[3, 1, 1]
>>> R.to_symmetric_function_on_basis(Composition([Integer(4),Integer(2),
↪Integer(1)]))
s[4, 2, 1] + s[5, 1, 1] + s[5, 2]
>>> R.to_symmetric_function_on_basis(Composition([]))
s[]
```

S

alias of *Complete*

ZL

alias of *Zassenhaus_left*

ZR

alias of *Zassenhaus_right*

class Zassenhaus_left(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The Hopf algebra of non-commutative symmetric functions in the left Zassenhaus basis.

This basis is the left-version of the basis defined in Section 2.5.1 of [HLNT09]. It is multiplicative, with Z_n defined as the element of $NCSF_n$ satisfying the equation

$$\sigma_1 = \cdots exp(Z_n) \cdots exp(Z_2) exp(Z_1),$$

where

$$\sigma_1 = \sum_{n \geq 0} S_n.$$

It can be recursively computed by the formula

$$S_n = \sum_{\lambda \vdash n} \frac{1}{m_1(\lambda)! m_2(\lambda)! m_3(\lambda)! \cdots} Z_{\lambda_1} Z_{\lambda_2} Z_{\lambda_3} \cdots$$

for all $n \geq 0$.

class Zassenhaus_right(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The Hopf algebra of non-commutative symmetric functions in the right Zassenhaus basis.

This basis is defined in Section 2.5.1 of [HLNT09]. It is multiplicative, with Z_n defined as the element of $NCSF_n$ satisfying the equation

$$\sigma_1 = exp(Z_1) exp(Z_2) exp(Z_3) \cdots exp(Z_n) \cdots$$

where

$$\sigma_1 = \sum_{n \geq 0} S_n.$$

It can be recursively computed by the formula

$$S_n = \sum_{\lambda \vdash n} \frac{1}{m_1(\lambda)!m_2(\lambda)!m_3(\lambda)!\cdots} \cdots Z_{\lambda_3} Z_{\lambda_2} Z_{\lambda_1}$$

for all $n \geq 0$.

Note that there is a variant (called the "noncommutative power sum symmetric functions of the third kind") in Definition 5.26 of [NCSF2] that satisfies:

$$\sigma_1 = exp(Z_1)exp(Z_2/2)exp(Z_3/3)\cdots exp(Z_n/n)\cdots.$$

a_realization()

Give a realization of the algebra of non-commutative symmetric functions. This particular realization is the complete basis of non-commutative symmetric functions.

OUTPUT:

- The realization of the non-commutative symmetric functions in the complete basis.

EXAMPLES:

```
sage: NonCommutativeSymmetricFunctions(ZZ).a_realization()
Non-Commutative Symmetric Functions over the Integer Ring in the Complete␣
↪basis
```

```
>>> from sage.all import *
>>> NonCommutativeSymmetricFunctions(ZZ).a_realization()
Non-Commutative Symmetric Functions over the Integer Ring in the Complete␣
↪basis
```

complete

alias of *Complete*

dQS

alias of *dualQuasisymmetric_Schur*

dYQS

alias of *dualYoungQuasisymmetric_Schur*

dual()

Return the dual to the non-commutative symmetric functions.

OUTPUT:

- The dual of the non-commutative symmetric functions over a ring. This is the algebra of quasi-symmetric functions over the ring.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: NCSF.dual()
Quasisymmetric functions over the Rational Field
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> NCSF.dual()
Quasisymmetric functions over the Rational Field
```

class dualQuasisymmetric_Schur(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The basis of NCSF dual to the Quasisymmetric-Schur basis of QSym.

The *Quasisymmetric_Schur* functions are defined in [QSCHUR] (see also Definition 5.1.1 of [LMvW13]). The dual basis in the algebra of non-commutative symmetric functions is defined by the following formula:

$$R_\alpha = \sum_T dQS_{shape(T)},$$

where the sum is over all standard composition tableaux with descent composition equal to α. The *Quasisymmetric_Schur* basis QS_α has the property that

$$s_\lambda = \sum_{sort(\alpha)=\lambda} QS_\alpha.$$

As a consequence the commutative image of a dual Quasisymmetric-Schur element in the algebra of symmetric functions (the map defined in the method *to_symmetric_function()*) is equal to the Schur function indexed by the decreasing sort of the indexing composition.

See also

CompositionTableaux, *CompositionTableau*.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: dQS = NCSF.dQS()
sage: dQS([1,3,2])*dQS([1])
dQS[1, 2, 4] + dQS[1, 3, 2, 1] + dQS[1, 3, 3] + dQS[3, 2, 2]
sage: dQS([1])*dQS([1,3,2])
dQS[1, 1, 3, 2] + dQS[1, 3, 3] + dQS[1, 4, 2] + dQS[2, 3, 2]
sage: dQS([1,3])*dQS([1,1])
dQS[1, 3, 1, 1] + dQS[1, 4, 1] + dQS[3, 2, 1] + dQS[4, 2]
sage: dQS([3,1])*dQS([2,1])
dQS[1, 1, 4, 1] + dQS[1, 4, 2] + dQS[1, 5, 1] + dQS[2, 4, 1] + dQS[3, 1,
2, 1] + dQS[3, 2, 2] + dQS[3, 3, 1] + dQS[4, 3] + dQS[5, 2]
sage: dQS([1,1]).coproduct()
dQS[] # dQS[1, 1] + dQS[1] # dQS[1] + dQS[1, 1] # dQS[]
sage: dQS([3,3]).coproduct().monomial_coefficients()[(Composition([1,2]),
↪Composition([1,2]))]
-1
sage: S = NCSF.complete()
sage: dQS(S[1,3,1])
dQS[1, 3, 1] + dQS[1, 4] + dQS[3, 2] + dQS[4, 1] + dQS[5]
sage: S(dQS[1,3,1])
S[1, 3, 1] - S[3, 2] - S[4, 1] + S[5]
sage: s = SymmetricFunctions(QQ).s()
sage: s(S(dQS([2,1,3])).to_symmetric_function())
s[3, 2, 1]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> dQS = NCSF.dQS()
>>> dQS([Integer(1),Integer(3),Integer(2)])*dQS([Integer(1)])
dQS[1, 2, 4] + dQS[1, 3, 2, 1] + dQS[1, 3, 3] + dQS[3, 2, 2]
>>> dQS([Integer(1)])*dQS([Integer(1),Integer(3),Integer(2)])
dQS[1, 1, 3, 2] + dQS[1, 3, 3] + dQS[1, 4, 2] + dQS[2, 3, 2]
>>> dQS([Integer(1),Integer(3)])*dQS([Integer(1),Integer(1)])
dQS[1, 3, 1, 1] + dQS[1, 4, 1] + dQS[3, 2, 1] + dQS[4, 2]
>>> dQS([Integer(3),Integer(1)])*dQS([Integer(2),Integer(1)])
dQS[1, 1, 4, 1] + dQS[1, 4, 2] + dQS[1, 5, 1] + dQS[2, 4, 1] + dQS[3, 1,
2, 1] + dQS[3, 2, 2] + dQS[3, 3, 1] + dQS[4, 3] + dQS[5, 2]
>>> dQS([Integer(1),Integer(1)]).coproduct()
dQS[] # dQS[1, 1] + dQS[1] # dQS[1] + dQS[1, 1] # dQS[]
>>> dQS([Integer(3),Integer(3)]).coproduct().monomial_
↪coefficients()[(Composition([Integer(1),Integer(2)]),
↪Composition([Integer(1),Integer(2)]))]
-1
>>> S = NCSF.complete()
>>> dQS(S[Integer(1),Integer(3),Integer(1)])
dQS[1, 3, 1] + dQS[1, 4] + dQS[3, 2] + dQS[4, 1] + dQS[5]
>>> S(dQS[Integer(1),Integer(3),Integer(1)])
S[1, 3, 1] - S[3, 2] - S[4, 1] + S[5]
>>> s = SymmetricFunctions(QQ).s()
>>> s(S(dQS([Integer(2),Integer(1),Integer(3)])).to_symmetric_function())
s[3, 2, 1]
```

dual()

The dual basis to the dual Quasisymmetric-Schur basis of NCSF.

The basis returned is the *Quasisymmetric_Schur* basis of QSym.

OUTPUT: the Quasisymmetric-Schur basis of the quasi-symmetric functions

EXAMPLES:

```
sage: dQS=NonCommutativeSymmetricFunctions(QQ).dualQuasisymmetric_Schur()
sage: dQS.dual()
Quasisymmetric functions over the Rational Field in the Quasisymmetric
Schur basis
sage: dQS.duality_pairing_matrix(dQS.dual(),3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> dQS=NonCommutativeSymmetricFunctions(QQ).dualQuasisymmetric_Schur()
>>> dQS.dual()
Quasisymmetric functions over the Rational Field in the Quasisymmetric
Schur basis
>>> dQS.duality_pairing_matrix(dQS.dual(),Integer(3))
[1 0 0 0]
[0 1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0]
[0 0 0 1]
```

to_symmetric_function_on_basis(*I*)

The commutative image of a dual quasi-symmetric Schur element.

The commutative image of a basis element is obtained by sorting the indexing composition of the basis element.

INPUT:

- I – a composition

OUTPUT:

- The commutative image of the dual quasi-Schur basis element indexed by I. The result is the Schur symmetric function indexed by the partition obtained by sorting I.

EXAMPLES:

```
sage: dQS=NonCommutativeSymmetricFunctions(QQ).dQS()
sage: dQS.to_symmetric_function_on_basis([2,1,3])
s[3, 2, 1]
sage: dQS.to_symmetric_function_on_basis([])
s[]
```

```
>>> from sage.all import *
>>> dQS=NonCommutativeSymmetricFunctions(QQ).dQS()
>>> dQS.to_symmetric_function_on_basis([Integer(2),Integer(1),Integer(3)])
s[3, 2, 1]
>>> dQS.to_symmetric_function_on_basis([])
s[]
```

class dualYoungQuasisymmetric_Schur(*NCSF*)

Bases: *CombinatorialFreeModule*, BindableClass

The basis of NCSF dual to the Young Quasisymmetric-Schur basis of QSym.

The YoungQuasisymmetric_Schur functions are given in Definition 5.2.1 of [LMvW13]. The dual basis in the algebra of non-commutative symmetric functions are related by an involution reversing the indexing composition of the complete expansion of a quasi-Schur basis element. This basis has many of the same properties as the Quasisymmetric Schur basis and is related to that basis by an algebraic transformation.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: dYQS = NCSF.dYQS()
sage: dYQS([1,3,2])*dYQS([1])
dYQS[1, 3, 2, 1] + dYQS[1, 3, 3] + dYQS[1, 4, 2] + dYQS[2, 3, 2]
sage: dYQS([1])*dYQS([1,3,2])
dYQS[1, 1, 3, 2] + dYQS[2, 3, 2] + dYQS[3, 3, 1] + dYQS[4, 1, 2]
sage: dYQS([1,3])*dYQS([1,1])
dYQS[1, 3, 1, 1] + dYQS[1, 4, 1] + dYQS[2, 3, 1] + dYQS[2, 4]
sage: dYQS([3,1])*dYQS([2,1])
dYQS[3, 1, 2, 1] + dYQS[3, 2, 2] + dYQS[3, 3, 1] + dYQS[4, 1, 1, 1]
 + dYQS[4, 1, 2] + dYQS[4, 2, 1] + dYQS[4, 3] + dYQS[5, 1, 1]
 + dYQS[5, 2]
sage: dYQS([1,1]).coproduct()
```

(continues on next page)

(continued from previous page)

```
dYQS[] # dYQS[1, 1] + dYQS[1] # dYQS[1] + dYQS[1, 1] # dYQS[]
sage: dYQS([3,3]).coproduct().monomial_coefficients()[(Composition([1,2]),
↪Composition([2,1]))]
1
sage: S = NCSF.complete()
sage: dYQS(S[1,3,1])
dYQS[1, 3, 1] + dYQS[1, 4] + dYQS[2, 3] + dYQS[4, 1] + dYQS[5]
sage: S(dYQS[1,3,1])
S[1, 3, 1] - S[1, 4] - S[2, 3] + S[5]
sage: s = SymmetricFunctions(QQ).s()
sage: s(S(dYQS([2,1,3])).to_symmetric_function())
s[3, 2, 1]
```

```
>>> from sage.all import *
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> dYQS = NCSF.dYQS()
>>> dYQS([Integer(1),Integer(3),Integer(2)])*dYQS([Integer(1)])
dYQS[1, 3, 2, 1] + dYQS[1, 3, 3] + dYQS[1, 4, 2] + dYQS[2, 3, 2]
>>> dYQS([Integer(1)])*dYQS([Integer(1),Integer(3),Integer(2)])
dYQS[1, 1, 3, 2] + dYQS[2, 3, 2] + dYQS[3, 3, 1] + dYQS[4, 1, 2]
>>> dYQS([Integer(1),Integer(3)])*dYQS([Integer(1),Integer(1)])
dYQS[1, 3, 1, 1] + dYQS[1, 4, 1] + dYQS[2, 3, 1] + dYQS[2, 4]
>>> dYQS([Integer(3),Integer(1)])*dYQS([Integer(2),Integer(1)])
dYQS[3, 1, 2, 1] + dYQS[3, 2, 2] + dYQS[3, 3, 1] + dYQS[4, 1, 1, 1]
 + dYQS[4, 1, 2] + dYQS[4, 2, 1] + dYQS[4, 3] + dYQS[5, 1, 1]
 + dYQS[5, 2]
>>> dYQS([Integer(1),Integer(1)]).coproduct()
dYQS[] # dYQS[1, 1] + dYQS[1] # dYQS[1] + dYQS[1, 1] # dYQS[]
>>> dYQS([Integer(3),Integer(3)]).coproduct().monomial_
↪coefficients()[(Composition([Integer(1),Integer(2)]),
↪Composition([Integer(2),Integer(1)]))]
1
>>> S = NCSF.complete()
>>> dYQS(S[Integer(1),Integer(3),Integer(1)])
dYQS[1, 3, 1] + dYQS[1, 4] + dYQS[2, 3] + dYQS[4, 1] + dYQS[5]
>>> S(dYQS[Integer(1),Integer(3),Integer(1)])
S[1, 3, 1] - S[1, 4] - S[2, 3] + S[5]
>>> s = SymmetricFunctions(QQ).s()
>>> s(S(dYQS([Integer(2),Integer(1),Integer(3)])).to_symmetric_function())
s[3, 2, 1]
```

dual()

The dual basis to the dual Quasisymmetric-Schur basis of NCSF.

The basis returned is the *Quasisymmetric_Schur* basis of QSym.

OUTPUT: the Young Quasisymmetric-Schur basis of quasi-symmetric functions

EXAMPLES:

```
sage: dYQS=NonCommutativeSymmetricFunctions(QQ).dualYoungQuasisymmetric_
↪Schur()
sage: dYQS.dual()
```

(continues on next page)

(continued from previous page)

```
Quasisymmetric functions over the Rational Field in the Young
Quasisymmetric Schur basis
sage: dYQS.duality_pairing_matrix(dYQS.dual(),3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> dYQS=NonCommutativeSymmetricFunctions(QQ).dualYoungQuasisymmetric_
↪Schur()
>>> dYQS.dual()
Quasisymmetric functions over the Rational Field in the Young
Quasisymmetric Schur basis
>>> dYQS.duality_pairing_matrix(dYQS.dual(),Integer(3))
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

to_symmetric_function_on_basis(*I*)

The commutative image of a dual Young quasi-symmetric Schur element.

The commutative image of a basis element is obtained by sorting the indexing composition of the basis element.

INPUT:

- `I` – a composition

OUTPUT:

- The commutative image of the dual Young quasi-Schur basis element indexed by `I`. The result is the Schur symmetric function indexed by the partition obtained by sorting `I`.

EXAMPLES:

```
sage: dYQS=NonCommutativeSymmetricFunctions(QQ).dYQS()
sage: dYQS.to_symmetric_function_on_basis([2,1,3])
s[3, 2, 1]
sage: dYQS.to_symmetric_function_on_basis([])
s[]
```

```
>>> from sage.all import *
>>> dYQS=NonCommutativeSymmetricFunctions(QQ).dYQS()
>>> dYQS.to_symmetric_function_on_basis([Integer(2),Integer(1),
↪Integer(3)])
s[3, 2, 1]
>>> dYQS.to_symmetric_function_on_basis([])
s[]
```

elementary

alias of `Elementary`

monomial

alias of `Monomial`

nM

alias of *Monomial*

ribbon

alias of *Ribbon*

5.1.145 Quasisymmetric functions

REFERENCES:

AUTHOR:

- Jason Bandlow
- Franco Saliola
- Chris Berg
- Darij Grinberg

class sage.combinat.ncsf_qsym.qsym.**QuasiSymmetricFunctions**(R)

Bases: UniqueRepresentation, Parent

The Hopf algebra of quasisymmetric functions.

Let R be a commutative ring with unity. The R-algebra of quasi-symmetric functions may be realized as an R-subalgebra of the ring of power series in countably many variables $R[[x_1, x_2, x_3, \ldots]]$. It consists of those formal power series p which are degree-bounded (i. e., the degrees of all monomials occurring with nonzero coefficient in p are bounded from above, although the bound can depend on p) and satisfy the following condition: For every tuple $(a_1, a_2, \ldots, a_m)$ of positive integers, the coefficient of the monomial $x_{i_1}^{a_1} x_{i_2}^{a_2} \cdots x_{i_m}^{a_m}$ in p is the same for all strictly increasing sequences $(i_1 < i_2 < \cdots < i_m)$ of positive integers. (In other words, the coefficient of a monomial in p depends only on the sequence of nonzero exponents in the monomial. If "sequence" were to be replaced by "multiset" here, we would obtain the definition of a symmetric function.)

The R-algebra of quasi-symmetric functions is commonly called QSym_R or occasionally just QSym (when R is clear from the context or **Z** or **Q**). It is graded by the total degree of the power series. Its homogeneous elements of degree k form a free R-submodule of rank equal to the number of compositions of k (that is, 2^{k-1} if $k \geq 1$, else 1).

The two classical bases of QSym, the monomial basis $(M_I)_I$ and the fundamental basis $(F_I)_I$, are indexed by compositions $I = (I_1, I_2, \cdots, I_\ell)$ and defined by the formulas:

$$M_I = \sum_{1 \leq i_1 < i_2 < \cdots < i_\ell} x_{i_1}^{I_1} x_{i_2}^{I_2} \cdots x_{i_\ell}^{I_\ell}$$

and

$$F_I = \sum_{(j_1, j_2, \ldots, j_n)} x_{j_1} x_{j_2} \cdots x_{j_n}$$

where in the second equation the sum runs over all weakly increasing n-tuples $(j_1, j_2, \ldots, j_n)$ of positive integers (where n is the size of I) which increase strictly from j_r to j_{r+1} if r is a descent of the composition I.

These bases are related by the formula

$F_I = \sum_{J \leq I} M_J$

where the inequality $J \leq I$ indicates that J is finer than I.

The R-algebra of quasi-symmetric functions is a Hopf algebra, with the coproduct satisfying

$$\Delta M_I = \sum_{k=0}^{\ell} M_{(I_1,I_2,\cdots,I_k)} \otimes M_{(I_{k+1},I_{k+2},\cdots,I_\ell)}$$

for every composition $I = (I_1, I_2, \cdots, I_\ell)$.

It is possible to define an R-algebra of quasi-symmetric functions in a finite number of variables as well (but it is not a bialgebra). These quasi-symmetric functions are actual polynomials then, not just power series.

Chapter 5 of [GriRei18] and Section 11 of [HazWitt1] are devoted to quasi-symmetric functions, as are Malvenuto's thesis [Mal1993] and part of Chapter 7 of [Sta-EC2].

The implementation of the quasi-symmetric function Hopf algebra

We realize the R-algebra of quasi-symmetric functions in Sage as a graded Hopf algebra with basis elements indexed by compositions:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.category()
Join of Category of Hopf algebras over Rational Field
    and Category of graded algebras over Rational Field
    and Category of commutative algebras over Rational Field
    and Category of monoids with realizations
    and Category of graded coalgebras over Rational Field
    and Category of coalgebras over Rational Field with realizations
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> QSym.category()
Join of Category of Hopf algebras over Rational Field
    and Category of graded algebras over Rational Field
    and Category of commutative algebras over Rational Field
    and Category of monoids with realizations
    and Category of graded coalgebras over Rational Field
    and Category of coalgebras over Rational Field with realizations
```

The most standard two bases for this R-algebra are the monomial and fundamental bases, and are accessible by the `M()` and `F()` methods:

```
sage: M = QSym.M()
sage: F = QSym.F()
sage: M(F[2,1,2])
M[1, 1, 1, 1, 1] + M[1, 1, 1, 2] + M[2, 1, 1, 1] + M[2, 1, 2]
sage: F(M[2,1,2])
F[1, 1, 1, 1, 1] - F[1, 1, 1, 2] - F[2, 1, 1, 1] + F[2, 1, 2]
```

```
>>> from sage.all import *
>>> M = QSym.M()
>>> F = QSym.F()
>>> M(F[Integer(2),Integer(1),Integer(2)])
M[1, 1, 1, 1, 1] + M[1, 1, 1, 2] + M[2, 1, 1, 1] + M[2, 1, 2]
>>> F(M[Integer(2),Integer(1),Integer(2)])
F[1, 1, 1, 1, 1] - F[1, 1, 1, 2] - F[2, 1, 1, 1] + F[2, 1, 2]
```

The product on this space is commutative and is inherited from the product on the realization within the ring of power series:

```
sage: M[3]*M[1,1] == M[1,1]*M[3]
True
sage: M[3]*M[1,1]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[3, 1, 1] + M[4, 1]
sage: F[3]*F[1,1]
F[1, 1, 3] + F[1, 2, 2] + F[1, 3, 1] + F[1, 4] + F[2, 1, 2]
 + F[2, 2, 1] + F[2, 3] + F[3, 1, 1] + F[3, 2] + F[4, 1]
sage: M[3]*F[2]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[2, 3] + M[3, 1, 1] + M[3, 2]
 + M[4, 1] + M[5]
sage: F[2]*M[3]
F[1, 1, 1, 2] - F[1, 2, 2] + F[2, 1, 1, 1] - F[2, 1, 2] - F[2, 2, 1]
 + F[5]
```

```
>>> from sage.all import *
>>> M[Integer(3)]*M[Integer(1),Integer(1)] == M[Integer(1),
↪Integer(1)]*M[Integer(3)]
True
>>> M[Integer(3)]*M[Integer(1),Integer(1)]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[3, 1, 1] + M[4, 1]
>>> F[Integer(3)]*F[Integer(1),Integer(1)]
F[1, 1, 3] + F[1, 2, 2] + F[1, 3, 1] + F[1, 4] + F[2, 1, 2]
 + F[2, 2, 1] + F[2, 3] + F[3, 1, 1] + F[3, 2] + F[4, 1]
>>> M[Integer(3)]*F[Integer(2)]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[2, 3] + M[3, 1, 1] + M[3, 2]
 + M[4, 1] + M[5]
>>> F[Integer(2)]*M[Integer(3)]
F[1, 1, 1, 2] - F[1, 2, 2] + F[2, 1, 1, 1] - F[2, 1, 2] - F[2, 2, 1]
 + F[5]
```

There is a coproduct on QSym as well, which in the Monomial basis acts by cutting the composition into a left half and a right half. The coproduct is not co-commutative:

```
sage: M[1,3,1].coproduct()
M[] # M[1, 3, 1] + M[1] # M[3, 1] + M[1, 3] # M[1] + M[1, 3, 1] # M[]
sage: F[1,3,1].coproduct()
F[] # F[1, 3, 1] + F[1] # F[3, 1] + F[1, 1] # F[2, 1]
 + F[1, 2] # F[1, 1] + F[1, 3] # F[1] + F[1, 3, 1] # F[]
```

```
>>> from sage.all import *
>>> M[Integer(1),Integer(3),Integer(1)].coproduct()
M[] # M[1, 3, 1] + M[1] # M[3, 1] + M[1, 3] # M[1] + M[1, 3, 1] # M[]
>>> F[Integer(1),Integer(3),Integer(1)].coproduct()
F[] # F[1, 3, 1] + F[1] # F[3, 1] + F[1, 1] # F[2, 1]
 + F[1, 2] # F[1, 1] + F[1, 3] # F[1] + F[1, 3, 1] # F[]
```

The duality pairing with non-commutative symmetric functions

These two operations endow the quasi-symmetric functions QSym with the structure of a Hopf algebra. It is the graded dual Hopf algebra of the non-commutative symmetric functions $NCSF$. Under this duality, the Monomial basis of QSym is dual to the Complete basis of $NCSF$, and the Fundamental basis of QSym is dual to the Ribbon basis of $NCSF$ (see [MR]).

```
sage: S = M.dual(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
sage: M[1,3,1].duality_pairing( S[1,3,1] )
1
sage: M.duality_pairing_matrix( S, degree=4 )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
sage: F.duality_pairing_matrix( S, degree=4 )
[1 0 0 0 0 0 0 0]
[1 1 0 0 0 0 0 0]
[1 0 1 0 0 0 0 0]
[1 1 1 1 0 0 0 0]
[1 0 0 0 1 0 0 0]
[1 1 0 0 1 1 0 0]
[1 0 1 0 1 0 1 0]
[1 1 1 1 1 1 1 1]
sage: NCSF = M.realization_of().dual()
sage: R = NCSF.Ribbon()
sage: F.duality_pairing_matrix( R, degree=4 )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
sage: M.duality_pairing_matrix( R, degree=4 )
[ 1  0  0  0  0  0  0  0]
[-1  1  0  0  0  0  0  0]
[-1  0  1  0  0  0  0  0]
[ 1 -1 -1  1  0  0  0  0]
[-1  0  0  0  1  0  0  0]
[ 1 -1  0  0 -1  1  0  0]
[ 1  0 -1  0 -1  0  1  0]
[-1  1  1 -1  1 -1 -1  1]
```

```
>>> from sage.all import *
>>> S = M.dual(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
>>> M[Integer(1),Integer(3),Integer(1)].duality_pairing( S[Integer(1),Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)] )
1
>>> M.duality_pairing_matrix( S, degree=Integer(4) )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
>>> F.duality_pairing_matrix( S, degree=Integer(4) )
[1 0 0 0 0 0 0 0]
[1 1 0 0 0 0 0 0]
[1 0 1 0 0 0 0 0]
[1 1 1 1 0 0 0 0]
[1 0 0 0 1 0 0 0]
[1 1 0 0 1 1 0 0]
[1 0 1 0 1 0 1 0]
[1 1 1 1 1 1 1 1]
>>> NCSF = M.realization_of().dual()
>>> R = NCSF.Ribbon()
>>> F.duality_pairing_matrix( R, degree=Integer(4) )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
>>> M.duality_pairing_matrix( R, degree=Integer(4) )
[ 1  0  0  0  0  0  0  0]
[-1  1  0  0  0  0  0  0]
[-1  0  1  0  0  0  0  0]
[ 1 -1 -1  1  0  0  0  0]
[-1  0  0  0  1  0  0  0]
[ 1 -1  0  0 -1  1  0  0]
[ 1  0 -1  0 -1  0  1  0]
[-1  1  1 -1  1 -1 -1  1]
```

Let H and G be elements of QSym, and h an element of $NCSF$. Then, if we represent the duality pairing with the mathematical notation $[\cdot,\cdot]$,

$$[HG, h] = [H \otimes G, \Delta(h)] .$$

For example, the coefficient of `M[2,1,4,1]` in `M[1,3]*M[2,1,1]` may be computed with the duality pairing:

```
sage: I, J = Composition([1,3]), Composition([2,1,1])
sage: (M[I]*M[J]).duality_pairing(S[2,1,4,1])
1
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> I, J = Composition([Integer(1),Integer(3)]), Composition([Integer(2),
↪Integer(1),Integer(1)])
>>> (M[I]*M[J]).duality_pairing(S[Integer(2),Integer(1),Integer(4),Integer(1)])
1
```

And the coefficient of `S[1,3] # S[2,1,1]` in `S[2,1,4,1].coproduct()` is equal to this result:

```
sage: S[2,1,4,1].coproduct()
S[] # S[2, 1, 4, 1] + ... + S[1, 3] # S[2, 1, 1] + ... + S[4, 1] # S[2, 1]
```

```
>>> from sage.all import *
>>> S[Integer(2),Integer(1),Integer(4),Integer(1)].coproduct()
S[] # S[2, 1, 4, 1] + ... + S[1, 3] # S[2, 1, 1] + ... + S[4, 1] # S[2, 1]
```

The duality pairing on the tensor space is another way of getting this coefficient, but currently the method `duality_pairing` is not defined on the tensor squared space. However, we can extend this functionality by applying a linear morphism to the terms in the coproduct, as follows:

```
sage: X = S[2,1,4,1].coproduct()
sage: def linear_morphism(x, y):
....:   return x.duality_pairing(M[1,3]) * y.duality_pairing(M[2,1,1])
sage: X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

```
>>> from sage.all import *
>>> X = S[Integer(2),Integer(1),Integer(4),Integer(1)].coproduct()
>>> def linear_morphism(x, y):
...   return x.duality_pairing(M[Integer(1),Integer(3)]) * y.duality_
↪pairing(M[Integer(2),Integer(1),Integer(1)])
>>> X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

Similarly, if H is an element of QSym and g and h are elements of $NCSF$, then

$$[H, gh] = [\Delta(H), g \otimes h].$$

For example, the coefficient of `R[2,3,1]` in `R[2,1]*R[2,1]` is computed with the duality pairing by the following command:

```
sage: (R[2,1]*R[2,1]).duality_pairing(F[2,3,1])
1
sage: R[2,1]*R[2,1]
R[2, 1, 2, 1] + R[2, 3, 1]
```

```
>>> from sage.all import *
>>> (R[Integer(2),Integer(1)]*R[Integer(2),Integer(1)]).duality_
↪pairing(F[Integer(2),Integer(3),Integer(1)])
1
>>> R[Integer(2),Integer(1)]*R[Integer(2),Integer(1)]
R[2, 1, 2, 1] + R[2, 3, 1]
```

This coefficient should then be equal to the coefficient of `F[2,1] # F[2,1]` in `F[2,3,1].coproduct()`:

```
sage: F[2,3,1].coproduct()
F[] # F[2, 3, 1] + ... + F[2, 1] # F[2, 1]  + ... + F[2, 3, 1] # F[]
```

```
>>> from sage.all import *
>>> F[Integer(2),Integer(3),Integer(1)].coproduct()
F[] # F[2, 3, 1] + ... + F[2, 1] # F[2, 1]  + ... + F[2, 3, 1] # F[]
```

This can also be computed by the duality pairing on the tensor space, as above:

```
sage: X = F[2,3,1].coproduct()
sage: def linear_morphism(x, y):
....:   return x.duality_pairing(R[2,1]) * y.duality_pairing(R[2,1])
sage: X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

```
>>> from sage.all import *
>>> X = F[Integer(2),Integer(3),Integer(1)].coproduct()
>>> def linear_morphism(x, y):
...   return x.duality_pairing(R[Integer(2),Integer(1)]) * y.duality_
↪pairing(R[Integer(2),Integer(1)])
>>> X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

The operation dual to multiplication by a non-commutative symmetric function

Let $g \in NCSF$ and consider the linear endomorphism of $NCSF$ defined by left (respectively, right) multiplication by g. Since there is a duality between QSym and $NCSF$, this linear transformation induces an operator $g^{\perp}$ on QSym satisfying

$$[g^{\perp}(H), h] = [H, gh].$$

for any non-commutative symmetric function h.

This is implemented by the method *skew_by()*. Explicitly, if `H` is a quasi-symmetric function and `g` a non-commutative symmetric function, then `H.skew_by(g)` and `H.skew_by(g, side='right')` are expressions that satisfy, for any non-commutative symmetric function `h`, the following equalities:

```
H.skew_by(g).duality_pairing(h) == H.duality_pairing(g*h)
H.skew_by(g, side='right').duality_pairing(h) == H.duality_pairing(h*g)
```

For example, `M[J].skew_by(S[I])` is 0 unless the composition `J` begins with `I` and `M(J).skew_by(S(I), side='right')` is 0 unless the composition `J` ends with `I`. For example:

```
sage: M[3,2,2].skew_by(S[3])
M[2, 2]
sage: M[3,2,2].skew_by(S[2])
0
sage: M[3,2,2].coproduct().apply_multilinear_morphism( lambda x,y: x.duality_
↪pairing(S[3])*y )
M[2, 2]
sage: M[3,2,2].skew_by(S[3], side='right')
0
sage: M[3,2,2].skew_by(S[2], side='right')
M[3, 2]
```

```
>>> from sage.all import *
>>> M[Integer(3),Integer(2),Integer(2)].skew_by(S[Integer(3)])
M[2, 2]
>>> M[Integer(3),Integer(2),Integer(2)].skew_by(S[Integer(2)])
0
>>> M[Integer(3),Integer(2),Integer(2)].coproduct().apply_multilinear_morphism(
↪lambda x,y: x.duality_pairing(S[Integer(3)])*y )
M[2, 2]
>>> M[Integer(3),Integer(2),Integer(2)].skew_by(S[Integer(3)], side='right')
0
>>> M[Integer(3),Integer(2),Integer(2)].skew_by(S[Integer(2)], side='right')
M[3, 2]
```

The counit

The counit is defined by sending all elements of positive degree to zero:

```
sage: M[3].degree(), M[3,1,2].degree(), M.one().degree()
(3, 6, 0)
sage: M[3].counit()
0
sage: M[3,1,2].counit()
0
sage: M.one().counit()
1
sage: (M[3] - 2*M[3,1,2] + 7).counit()
7
sage: (F[3] - 2*F[3,1,2] + 7).counit()
7
```

```
>>> from sage.all import *
>>> M[Integer(3)].degree(), M[Integer(3),Integer(1),Integer(2)].degree(), M.one().
↪degree()
(3, 6, 0)
>>> M[Integer(3)].counit()
0
>>> M[Integer(3),Integer(1),Integer(2)].counit()
0
>>> M.one().counit()
1
>>> (M[Integer(3)] - Integer(2)*M[Integer(3),Integer(1),Integer(2)] + Integer(7)).
↪counit()
7
>>> (F[Integer(3)] - Integer(2)*F[Integer(3),Integer(1),Integer(2)] + Integer(7)).
↪counit()
7
```

The antipode

The antipode sends the Fundamental basis element indexed by the composition I to $(-1)^{|I|}$ times the Fundamental basis element indexed by the conjugate composition to I (where $|I|$ stands for the size of I, that is, the sum of all entries of I).

```
sage: F[3,2,2].antipode()
-F[1, 2, 2, 1, 1]
sage: Composition([3,2,2]).conjugate()
[1, 2, 2, 1, 1]
```

```
>>> from sage.all import *
>>> F[Integer(3),Integer(2),Integer(2)].antipode()
-F[1, 2, 2, 1, 1]
>>> Composition([Integer(3),Integer(2),Integer(2)]).conjugate()
[1, 2, 2, 1, 1]
```

The antipodes of the Monomial quasisymmetric functions can also be computed easily: Every composition I satisfies

$$\omega(M_I) = (-1)^{\ell(I)} \sum M_J,$$

where the sum ranges over all compositions J of $|I|$ which are coarser than the reversed composition I^r of I. Here, $\ell(I)$ denotes the length of the composition I (that is, the number of its parts).

```
sage: M[3,2,1].antipode()
-M[1, 2, 3] - M[1, 5] - M[3, 3] - M[6]
sage: M[3,2,2].antipode()
-M[2, 2, 3] - M[2, 5] - M[4, 3] - M[7]
```

```
>>> from sage.all import *
>>> M[Integer(3),Integer(2),Integer(1)].antipode()
-M[1, 2, 3] - M[1, 5] - M[3, 3] - M[6]
>>> M[Integer(3),Integer(2),Integer(2)].antipode()
-M[2, 2, 3] - M[2, 5] - M[4, 3] - M[7]
```

We demonstrate here the defining relation of the antipode:

```
sage: X = F[3,2,2].coproduct()
sage: X.apply_multilinear_morphism(lambda x,y: x*y.antipode())
0
sage: X.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

```
>>> from sage.all import *
>>> X = F[Integer(3),Integer(2),Integer(2)].coproduct()
>>> X.apply_multilinear_morphism(lambda x,y: x*y.antipode())
0
>>> X.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

The relation with symmetric functions

The quasi-symmetric functions are a ring which contain the symmetric functions as a subring. The Monomial quasi-symmetric functions are related to the monomial symmetric functions by

$$m_\lambda = \sum_{\mathrm{sort}(I)=\lambda} M_I$$

(where $\mathrm{sort}(I)$ denotes the result of sorting the entries of I in decreasing order).

There are methods to test if an expression in the quasi-symmetric functions is a symmetric function and, if it is, send it to an expression in the symmetric functions:

```
sage: f = M[1,1,2] + M[1,2,1]
sage: f.is_symmetric()
False
sage: g = M[3,1] + M[1,3]
sage: g.is_symmetric()
True
sage: g.to_symmetric_function()
m[3, 1]
```

```
>>> from sage.all import *
>>> f = M[Integer(1),Integer(1),Integer(2)] + M[Integer(1),Integer(2),Integer(1)]
>>> f.is_symmetric()
False
>>> g = M[Integer(3),Integer(1)] + M[Integer(1),Integer(3)]
>>> g.is_symmetric()
True
>>> g.to_symmetric_function()
m[3, 1]
```

The expansion of the Schur function in terms of the Fundamental quasi-symmetric functions is due to [Ges]. There is one term in the expansion for each standard tableau of shape equal to the partition indexing the Schur function.

```
sage: f = F[3,2] + F[2,2,1] + F[2,3] + F[1,3,1] + F[1,2,2]
sage: f.is_symmetric()
True
sage: f.to_symmetric_function()
5*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + 2*m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
sage: s = SymmetricFunctions(QQ).s()
sage: s(f.to_symmetric_function())
s[3, 2]
```

```
>>> from sage.all import *
>>> f = F[Integer(3),Integer(2)] + F[Integer(2),Integer(2),Integer(1)] +␣
↪F[Integer(2),Integer(3)] + F[Integer(1),Integer(3),Integer(1)] + F[Integer(1),
↪Integer(2),Integer(2)]
>>> f.is_symmetric()
True
>>> f.to_symmetric_function()
5*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + 2*m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
>>> s = SymmetricFunctions(QQ).s()
>>> s(f.to_symmetric_function())
s[3, 2]
```

It is also possible to convert a symmetric function to a quasi-symmetric function:

```
sage: m = SymmetricFunctions(QQ).m()
sage: M( m[3,1,1] )
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
sage: F( s[2,2,1] )
F[1, 1, 2, 1] + F[1, 2, 1, 1] + F[1, 2, 2] + F[2, 1, 2] + F[2, 2, 1]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> M( m[Integer(3),Integer(1),Integer(1)] )
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
>>> F( s[Integer(2),Integer(2),Integer(1)] )
F[1, 1, 2, 1] + F[1, 2, 1, 1] + F[1, 2, 2] + F[2, 1, 2] + F[2, 2, 1]
```

It is possible to experiment with the quasi-symmetric function expansion of other bases, but it is important that the base ring be the same for both algebras.

```
sage: R = QQ['t']
sage: Qp = SymmetricFunctions(R).hall_littlewood().Qp()
sage: QSymt = QuasiSymmetricFunctions(R)
sage: Ft = QSymt.F()
sage: Ft( Qp[2,2] )
F[1, 2, 1] + t*F[1, 3] + (t+1)*F[2, 2] + t*F[3, 1] + t^2*F[4]
```

```
>>> from sage.all import *
>>> R = QQ['t']
>>> Qp = SymmetricFunctions(R).hall_littlewood().Qp()
>>> QSymt = QuasiSymmetricFunctions(R)
>>> Ft = QSymt.F()
>>> Ft( Qp[Integer(2),Integer(2)] )
F[1, 2, 1] + t*F[1, 3] + (t+1)*F[2, 2] + t*F[3, 1] + t^2*F[4]
```

```
sage: K = QQ['q','t'].fraction_field()
sage: Ht = SymmetricFunctions(K).macdonald().Ht()
sage: Fqt = QuasiSymmetricFunctions(Ht.base_ring()).F()
sage: Fqt(Ht[2,1])
q*t*F[1, 1, 1] + (q+t)*F[1, 2] + (q+t)*F[2, 1] + F[3]
```

```
>>> from sage.all import *
>>> K = QQ['q','t'].fraction_field()
>>> Ht = SymmetricFunctions(K).macdonald().Ht()
>>> Fqt = QuasiSymmetricFunctions(Ht.base_ring()).F()
>>> Fqt(Ht[Integer(2),Integer(1)])
q*t*F[1, 1, 1] + (q+t)*F[1, 2] + (q+t)*F[2, 1] + F[3]
```

The following will raise an error because the base ring of `F` is not equal to the base ring of `Ht`:

```
sage: F(Ht[2,1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= McdHt[2, 1]) an element of self␣
↪(=Quasisymmetric functions over the Rational Field in the Fundamental basis)
```

```
>>> from sage.all import *
>>> F(Ht[Integer(2),Integer(1)])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= McdHt[2, 1]) an element of self␣
↪(=Quasisymmetric functions over the Rational Field in the Fundamental basis)
```

The map to the ring of polynomials

The quasi-symmetric functions can be seen as an inverse limit of a subring of a polynomial ring as the number of variables increases. Indeed, there exists a projection from the quasi-symmetric functions onto the polynomial ring $R[x_1, x_2, \ldots, x_n]$. This projection is defined by sending the variables $x_{n+1}, x_{n+2}, \cdots$ to 0, while the remaining n variables remain fixed. Note that this projection sends M_I to 0 if the length of the composition I is higher than n.

```
sage: M[1,3,1].expand(4)
x0*x1^3*x2 + x0*x1^3*x3 + x0*x2^3*x3 + x1*x2^3*x3
sage: F[1,3,1].expand(4)
x0*x1^3*x2 + x0*x1^3*x3 + x0*x1^2*x2*x3 + x0*x1*x2^2*x3 + x0*x2^3*x3 + x1*x2^3*x3
sage: M[1,3,1].expand(2)
0
```

```
>>> from sage.all import *
>>> M[Integer(1),Integer(3),Integer(1)].expand(Integer(4))
x0*x1^3*x2 + x0*x1^3*x3 + x0*x2^3*x3 + x1*x2^3*x3
>>> F[Integer(1),Integer(3),Integer(1)].expand(Integer(4))
x0*x1^3*x2 + x0*x1^3*x3 + x0*x1^2*x2*x3 + x0*x1*x2^2*x3 + x0*x2^3*x3 + x1*x2^3*x3
>>> M[Integer(1),Integer(3),Integer(1)].expand(Integer(2))
0
```

class Bases(*parent_with_realization*)

Bases: `Category_realization_of_parent`

Category of bases of quasi-symmetric functions.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.Bases()
Category of bases of Quasisymmetric functions over the Rational Field
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> QSym.Bases()
Category of bases of Quasisymmetric functions over the Rational Field
```

class ElementMethods

Bases: `object`

Methods common to all elements of `QuasiSymmetricFunctions`.

adams_operator(*n*)

Return the image of the quasi-symmetric function `self` under the n-th Adams operator.

The n-th Adams operator $\mathbf{f}_n$ is defined to be the map from the R-algebra of quasi-symmetric functions to itself that sends every symmetric function $P(x_1, x_2, x_3, \ldots)$ to $P(x_1^n, x_2^n, x_3^n, \ldots)$. This operator $\mathbf{f}_n$ is a Hopf algebra endomorphism, and satisfies

$$f_n M_{(i_1, i_2, i_3, \ldots)} = M_{(ni_1, ni_2, ni_3, \ldots)}$$

for every composition $(i_1, i_2, i_3, \ldots)$ (where M means the monomial basis).

The n-th Adams operator is also called the n-th Frobenius endomorphism. It is not related to the Frobenius map which connects the ring of symmetric functions with the representation theory of the symmetric group.

The n-th Adams operator is the n-th Adams operator of the Λ-ring of quasi-symmetric functions over the integers.

The restriction of the n-th Adams operator to the subring formed by all symmetric functions is, not unexpectedly, the n-th Adams operator of the ring of symmetric functions.

See also

Symmetric functions plethysm

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Adams operator (on the ring of quasi-symmetric functions) to `self`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: M = QSym.M()
sage: F = QSym.F()
sage: M[3,2].adams_operator(2)
M[6, 4]
sage: (M[2,1] - 2*M[3]).adams_operator(4)
M[8, 4] - 2*M[12]
sage: M([]).adams_operator(3)
M[]
sage: F[1,1].adams_operator(2)
F[1, 1, 1, 1] - F[1, 1, 2] - F[2, 1, 1] + F[2, 2]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> M = QSym.M()
>>> F = QSym.F()
>>> M[Integer(3),Integer(2)].adams_operator(Integer(2))
M[6, 4]
>>> (M[Integer(2),Integer(1)] - Integer(2)*M[Integer(3)]).adams_
↪operator(Integer(4))
M[8, 4] - 2*M[12]
>>> M([]).adams_operator(Integer(3))
M[]
>>> F[Integer(1),Integer(1)].adams_operator(Integer(2))
F[1, 1, 1, 1] - F[1, 1, 2] - F[2, 1, 1] + F[2, 2]
```

The Adams endomorphisms are multiplicative:

```
sage: all( all( M(I).adams_operator(3) * M(J).adams_operator(3)
....:           == (M(I) * M(J)).adams_operator(3)
....:           for I in Compositions(3) )
....:      for J in Compositions(2) )
True
```

```
>>> from sage.all import *
>>> all( all( M(I).adams_operator(Integer(3)) * M(J).adams_
```

(continues on next page)

(continued from previous page)

```
↪operator(Integer(3))
...              == (M(I) * M(J)).adams_operator(Integer(3))
...              for I in Compositions(Integer(3)) )
...         for J in Compositions(Integer(2)) )
True
```

Being Hopf algebra endomorphisms, the Adams operators commute with the antipode:

```
sage: all( M(I).adams_operator(4).antipode()
....:      == M(I).antipode().adams_operator(4)
....:      for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> all( M(I).adams_operator(Integer(4)).antipode()
...      == M(I).antipode().adams_operator(Integer(4))
...      for I in Compositions(Integer(3)) )
True
```

The restriction of the Adams operators to the subring of symmetric functions are the Adams operators of the latter:

```
sage: e = SymmetricFunctions(ZZ).e()
sage: all( M(e(lam)).adams_operator(3)
....:      == M(e(lam).adams_operator(3))
....:      for lam in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(ZZ).e()
>>> all( M(e(lam)).adams_operator(Integer(3))
...      == M(e(lam).adams_operator(Integer(3)))
...      for lam in Partitions(Integer(3)) )
True
```

dendriform_leq(*other*)

Return the result of applying the dendriform smaller-or-equal operation to the two quasi-symmetric functions `self` and `other`.

The dendriform smaller-or-equal operation is a binary operation, denoted by $\preceq$ and written infix, on the ring of quasi-symmetric functions. It can be defined as a restriction of a binary operation (denoted by $\preceq$ and written infix as well) on the ring of formal power series $R[[x_1, x_2, x_3, \ldots]]$, which is defined as follows: If m and n are two monomials in $x_1, x_2, x_3, \ldots$, then we let $m \preceq n$ be the product mn if the smallest positive integer i for which x_i occurs in m is smaller or equal to the smallest positive integer j for which x_j occurs in n (this is understood to be false when $m = 1$ and $n \neq 1$, and true when $n = 1$), and we let $m \preceq n$ be 0 otherwise. Having thus defined $\preceq$ on monomials, we extend $\preceq$ to a binary operation on $R[[x_1, x_2, x_3, \ldots]]$ by requiring it to be continuous (in both inputs) and R-bilinear. It is easily seen that $QSym \preceq QSym \subseteq QSym$, so that $\preceq$ restricts to a binary operation on $QSym$.

This operation $\preceq$ is related to the dendriform smaller relation $\prec$ (`dendriform_less()`). Namely, if we define a binary operation $\succ$ on $QSym$ by $a \succ b = b \prec a$, then $(QSym, \preceq, \succ)$ is a dendriform R-algebra. Thus, any $a, b \in QSym$ satisfy $a \preceq b = ab - b \prec a$.

See also

dendriform_less()

INPUT:

- `other` – a quasi-symmetric function over the same base ring as `self`

OUTPUT:

The quasi-symmetric function `self` $\preceq$ `other`, written in the basis of `self`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: M[2, 1].dendriform_leq(M[1, 2])
2*M[2, 1, 1, 2] + M[2, 1, 2, 1] + M[2, 1, 3] + M[2, 2, 2]
 + M[3, 1, 2] + M[3, 2, 1] + M[3, 3]
sage: F = QSym.F()
sage: F[2, 1].dendriform_leq(F[1, 2])
F[2, 1, 1, 2] + F[2, 1, 2, 1] + F[2, 1, 3] + F[2, 2, 1, 1]
 + 2*F[2, 2, 2] + F[2, 3, 1] + F[3, 1, 2] + F[3, 2, 1] + F[3, 3]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.M()
>>> M[Integer(2), Integer(1)].dendriform_leq(M[Integer(1), Integer(2)])
2*M[2, 1, 1, 2] + M[2, 1, 2, 1] + M[2, 1, 3] + M[2, 2, 2]
 + M[3, 1, 2] + M[3, 2, 1] + M[3, 3]
>>> F = QSym.F()
>>> F[Integer(2), Integer(1)].dendriform_leq(F[Integer(1), Integer(2)])
F[2, 1, 1, 2] + F[2, 1, 2, 1] + F[2, 1, 3] + F[2, 2, 1, 1]
 + 2*F[2, 2, 2] + F[2, 3, 1] + F[3, 1, 2] + F[3, 2, 1] + F[3, 3]
```

dendriform_less(*other*)

Return the result of applying the dendriform smaller operation to the two quasi-symmetric functions `self` and `other`.

The dendriform smaller operation is a binary operation, denoted by $\prec$ and written infix, on the ring of quasi-symmetric functions. It can be defined as a restriction of a binary operation (denoted by $\prec$ and written infix as well) on the ring of formal power series $R[[x_1, x_2, x_3, \ldots]]$, which is defined as follows: If m and n are two monomials in $x_1, x_2, x_3, \ldots$, then we let $m \prec n$ be the product mn if the smallest positive integer i for which x_i occurs in m is smaller than the smallest positive integer j for which x_j occurs in n (this is understood to be false when $m = 1$, and true when $m \neq 1$ and $n = 1$), and we let $m \prec n$ be 0 otherwise. Having thus defined $\prec$ on monomials, we extend $\prec$ to a binary operation on $R[[x_1, x_2, x_3, \ldots]]$ by requiring it to be continuous (in both inputs) and R-bilinear. It is easily seen that $QSym \prec QSym \subseteq QSym$, so that $\prec$ restricts to a binary operation on $QSym$.

See also

dendriform_leq()

INPUT:

- `other` – a quasi-symmetric function over the same base ring as `self`

OUTPUT:

The quasi-symmetric function `self` $\prec$ `other`, written in the basis of `self`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: M[2, 1].dendriform_less(M[1, 2])
2*M[2, 1, 1, 2] + M[2, 1, 2, 1] + M[2, 1, 3] + M[2, 2, 2]
sage: F = QSym.F()
sage: F[2, 1].dendriform_less(F[1, 2])
F[1, 1, 2, 1, 1] + F[1, 1, 2, 2] + F[1, 1, 3, 1]
 + F[1, 2, 1, 2] + F[1, 2, 2, 1] + F[1, 2, 3]
 + F[2, 1, 1, 2] + F[2, 1, 2, 1] + F[2, 1, 3] + F[2, 2, 2]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.M()
>>> M[Integer(2), Integer(1)].dendriform_less(M[Integer(1),␣
↪Integer(2)])
2*M[2, 1, 1, 2] + M[2, 1, 2, 1] + M[2, 1, 3] + M[2, 2, 2]
>>> F = QSym.F()
>>> F[Integer(2), Integer(1)].dendriform_less(F[Integer(1),␣
↪Integer(2)])
F[1, 1, 2, 1, 1] + F[1, 1, 2, 2] + F[1, 1, 3, 1]
 + F[1, 2, 1, 2] + F[1, 2, 2, 1] + F[1, 2, 3]
 + F[2, 1, 1, 2] + F[2, 1, 2, 1] + F[2, 1, 3] + F[2, 2, 2]
```

The operation $\prec$ can be used to recursively construct the dual immaculate basis: For every positive integer m and every composition I, the dual immaculate function $\mathrm{dI}_{[m,I]}$ of the composition $[m, I]$ (this composition is I with m prepended to it) is $F_{[m]} \prec \mathrm{dI}_I$.

```
sage: dI = QSym.dI()
sage: dI(F[2]).dendriform_less(dI[1, 2])
dI[2, 1, 2]
```

```
>>> from sage.all import *
>>> dI = QSym.dI()
>>> dI(F[Integer(2)]).dendriform_less(dI[Integer(1), Integer(2)])
dI[2, 1, 2]
```

We check with the identity element:

```
sage: M.one().dendriform_less(M[1,2])
0
sage: M[1,2].dendriform_less(M.one())
M[1, 2]
```

```
>>> from sage.all import *
>>> M.one().dendriform_less(M[Integer(1),Integer(2)])
0
>>> M[Integer(1),Integer(2)].dendriform_less(M.one())
M[1, 2]
```

The operation $\prec$ is not symmetric, nor if $a \prec b \neq 0$, then $b \prec a = 0$ (as it would be for a single monomial):

```
sage: M[1,2,1].dendriform_less(M[1,2])
M[1, 1, 2, 1, 2] + 2*M[1, 1, 2, 2, 1] + M[1, 1, 2, 3]
 + M[1, 1, 4, 1] + 2*M[1, 2, 1, 1, 2] + M[1, 2, 1, 2, 1]
 + M[1, 2, 1, 3] + M[1, 2, 2, 2] + M[1, 3, 1, 2]
 + M[1, 3, 2, 1] + M[1, 3, 3]
sage: M[1,2].dendriform_less(M[1,2,1])
M[1, 1, 2, 1, 2] + 2*M[1, 1, 2, 2, 1] + M[1, 1, 2, 3]
 + M[1, 1, 4, 1] + M[1, 2, 1, 2, 1] + M[1, 3, 2, 1]
```

```
>>> from sage.all import *
>>> M[Integer(1),Integer(2),Integer(1)].dendriform_less(M[Integer(1),
→Integer(2)])
M[1, 1, 2, 1, 2] + 2*M[1, 1, 2, 2, 1] + M[1, 1, 2, 3]
 + M[1, 1, 4, 1] + 2*M[1, 2, 1, 1, 2] + M[1, 2, 1, 2, 1]
 + M[1, 2, 1, 3] + M[1, 2, 2, 2] + M[1, 3, 1, 2]
 + M[1, 3, 2, 1] + M[1, 3, 3]
>>> M[Integer(1),Integer(2)].dendriform_less(M[Integer(1),Integer(2),
→Integer(1)])
M[1, 1, 2, 1, 2] + 2*M[1, 1, 2, 2, 1] + M[1, 1, 2, 3]
 + M[1, 1, 4, 1] + M[1, 2, 1, 2, 1] + M[1, 3, 2, 1]
```

expand(*n*, *alphabet='x'*)

Expand the quasi-symmetric function into `n` variables in an alphabet, which by default is `'x'`.

INPUT:

- `n` – nonnegative integer; the number of variables in the expansion
- `alphabet` – (default: `'x'`) the alphabet in which `self` is to be expanded

OUTPUT: an expansion of `self` into the `n` variables specified by `alphabet`

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: F[3].expand(3)
x0^3 + x0^2*x1 + x0*x1^2 + x1^3 + x0^2*x2 + x0*x1*x2 + x1^2*x2 + x0*x2^
→2 + x1*x2^2 + x2^3
sage: F[2,1].expand(3)
x0^2*x1 + x0^2*x2 + x0*x1*x2 + x1^2*x2
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> F[Integer(3)].expand(Integer(3))
x0^3 + x0^2*x1 + x0*x1^2 + x1^3 + x0^2*x2 + x0*x1*x2 + x1^2*x2 + x0*x2^
→2 + x1*x2^2 + x2^3
>>> F[Integer(2),Integer(1)].expand(Integer(3))
x0^2*x1 + x0^2*x2 + x0*x1*x2 + x1^2*x2
```

One can use a different set of variable by adding an optional argument `alphabet=...`

```
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: F[3].expand(2,alphabet='y')
y0^3 + y0^2*y1 + y0*y1^2 + y1^3
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> F[Integer(3)].expand(Integer(2),alphabet='y')
y0^3 + y0^2*y1 + y0*y1^2 + y1^3
```

frobenius(**args*, ***kwds*)

Deprecated: Use `adams_operator()` instead. See Issue #36396 for details.

internal_coproduct()

Return the inner coproduct of `self` in the basis of `self`.

The inner coproduct (also known as the Kronecker coproduct, or as the second comultiplication on the R-algebra of quasi-symmetric functions) is an R-algebra homomorphism $\Delta^\times$ from the R-algebra of quasi-symmetric functions to the tensor square (over R) of quasi-symmetric functions. It can be defined in the following two ways:

1. If I is a composition, then a $(0, I)$-matrix will mean a matrix whose entries are nonnegative integers such that no row and no column of this matrix is zero, and such that if all the nonzero entries of the matrix are read (row by row, starting at the topmost row, reading every row from left to right), then the reading word obtained is I. If A is a $(0, I)$-matrix, then row(A) will denote the vector of row sums of A (regarded as a composition), and column(A) will denote the vector of column sums of A (regarded as a composition).

   For every composition I, the internal coproduct $\Delta^\times(M_I)$ of the I-th monomial quasisymmetric function M_I is the sum

   $$\sum_{A \text{ is a } (0,I)\text{-matrix}} M_{\text{row}(A)} \otimes M_{\text{column}(A)}.$$

   See Section 11.39 of [HazWitt1].
2. For every permutation w, let $C(w)$ denote the descent composition of w. Then, for any composition I of size n, the internal coproduct $\Delta^\times(F_I)$ of the I-th fundamental quasisymmetric function F_I is the sum

   $$\sum_{\substack{\sigma\in S_n,\\ \tau\in S_n,\\ \tau\sigma=\pi}} F_{C(\sigma)} \otimes F_{C(\tau)},$$

   where π is any permutation in S_n having descent composition I and where permutations act from the left and multiply accordingly, so $\tau\sigma$ means first applying σ and then τ. See Theorem 4.23 in [Mal1993], but beware of the notations which are apparently different from those in [HazWitt1].

The restriction of the internal coproduct to the R-algebra of symmetric functions is the well-known internal coproduct on the symmetric functions.

The method `kronecker_coproduct()` is a synonym of this one.

EXAMPLES:

Let us compute the internal coproduct of M_{21} (which is short for $M_{[2,1]}$). The $(0, [2, 1])$-matrices are

$$\begin{bmatrix}2 & 1\end{bmatrix}, \begin{bmatrix}2\\1\end{bmatrix}, \begin{bmatrix}2 & 0\\0 & 1\end{bmatrix}, \text{ and } \begin{bmatrix}0 & 2\\1 & 0\end{bmatrix}$$

so

$$\Delta^\times(M_{21}) = M_3 \otimes M_{21} + M_{21} \otimes M_3 + M_{21} \otimes M_{21} + M_{21} \otimes M_{12}.$$

This is confirmed by the following Sage computation (incidentally demonstrating the non-cocommutativity of the internal coproduct):

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: a = M([2,1])
sage: a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(ZZ).M()
>>> a = M([Integer(2),Integer(1)])
>>> a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

Further examples:

```
sage: all( M([i]).internal_coproduct() == tensor([M([i]), M([i])])
....:      for i in range(1, 4) )
True

sage: M([1, 2]).internal_coproduct()
M[1, 2] # M[1, 2] + M[1, 2] # M[2, 1] + M[1, 2] # M[3] + M[3] # M[1, 2]
```

```
>>> from sage.all import *
>>> all( M([i]).internal_coproduct() == tensor([M([i]), M([i])])
...      for i in range(Integer(1), Integer(4)) )
True

>>> M([Integer(1), Integer(2)]).internal_coproduct()
M[1, 2] # M[1, 2] + M[1, 2] # M[2, 1] + M[1, 2] # M[3] + M[3] # M[1, 2]
```

The definition of $\Delta^{\times}(M_I)$ in terms of $(0, I)$-matrices is not suitable for computation in cases where the length of I is large, but we can use it as a doctest. Here is a naive implementation:

```
sage: def naive_internal_coproduct_on_M(I):
....:     # INPUT: composition I
....:     #        (not quasi-symmetric function)
....:     # OUTPUT: interior coproduct of M_I
....:     M = QuasiSymmetricFunctions(ZZ).M()
....:     M2 = M.tensor(M)
....:     res = M2.zero()
....:     l = len(I)
....:     n = I.size()
....:     for S in Subsets(range(l**2), l):
....:         M_list = sorted(S)
....:         row_M = [sum([I[M_list.index(l * i + j)]
....:                       for j in range(l) if
....:                       l * i + j in S])
....:                  for i in range(l)]
....:         col_M = [sum([I[M_list.index(l * i + j)]
....:                       for i in range(l) if
....:                       l * i + j in S])
....:                  for j in range(l)]
....:         if 0 in row_M:
....:             first_zero = row_M.index(0)
....:             row_M = row_M[:first_zero]
```

(continues on next page)

(continued from previous page)

```
....:             if sum(row_M) != n:
....:                 continue
....:         if 0 in col_M:
....:             first_zero = col_M.index(0)
....:             col_M = col_M[:first_zero]
....:             if sum(col_M) != n:
....:                 continue
....:         res += tensor([M(Compositions(n)(row_M)),
....:                        M(Compositions(n)(col_M))])
....:     return res
sage: all( naive_internal_coproduct_on_M(I)
....:      == M(I).internal_coproduct()
....:      for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> def naive_internal_coproduct_on_M(I):
...     # INPUT: composition I
...     #        (not quasi-symmetric function)
...     # OUTPUT: interior coproduct of M_I
...     M = QuasiSymmetricFunctions(ZZ).M()
...     M2 = M.tensor(M)
...     res = M2.zero()
...     l = len(I)
...     n = I.size()
...     for S in Subsets(range(l**Integer(2)), l):
...         M_list = sorted(S)
...         row_M = [sum([I[M_list.index(l * i + j)]
...                       for j in range(l) if
...                       l * i + j in S])
...                  for i in range(l)]
...         col_M = [sum([I[M_list.index(l * i + j)]
...                       for i in range(l) if
...                       l * i + j in S])
...                  for j in range(l)]
...         if Integer(0) in row_M:
...             first_zero = row_M.index(Integer(0))
...             row_M = row_M[:first_zero]
...             if sum(row_M) != n:
...                 continue
...         if Integer(0) in col_M:
...             first_zero = col_M.index(Integer(0))
...             col_M = col_M[:first_zero]
...             if sum(col_M) != n:
...                 continue
...         res += tensor([M(Compositions(n)(row_M)),
...                        M(Compositions(n)(col_M))])
...     return res
>>> all( naive_internal_coproduct_on_M(I)
...      == M(I).internal_coproduct()
...      for I in Compositions(Integer(3)) )
True
```

Todo

Implement this directly on the monomial basis maybe? The $(0, I)$-matrices are a pain to generate from their definition, but maybe there is a good algorithm. If so, the above "further examples" should be moved to the M-method.

is_symmetric()

Return `True` if `self` is an element of the symmetric functions.

This is being tested by looking at the expansion in the Monomial basis and checking if the coefficients are the same if the indexing compositions are permutations of each other.

OUTPUT:

- `True` if `self` is symmetric. `False` if `self` is not symmetric.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: F = QSym.Fundamental()
sage: (F[3,2] + F[2,3]).is_symmetric()
False
sage: (F[1, 1, 1, 2] + F[1, 1, 3] + F[1, 3, 1] + F[2, 1, 1, 1] + F[3,
↪1, 1]).is_symmetric()
True
sage: F([]).is_symmetric()
True
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> F = QSym.Fundamental()
>>> (F[Integer(3),Integer(2)] + F[Integer(2),Integer(3)]).is_
↪symmetric()
False
>>> (F[Integer(1), Integer(1), Integer(1), Integer(2)] + F[Integer(1),
↪Integer(1), Integer(3)] + F[Integer(1), Integer(3), Integer(1)] +
↪F[Integer(2), Integer(1), Integer(1), Integer(1)] + F[Integer(3),
↪Integer(1), Integer(1)]).is_symmetric()
True
>>> F([]).is_symmetric()
True
```

kronecker_coproduct()

Return the inner coproduct of `self` in the basis of `self`.

The inner coproduct (also known as the Kronecker coproduct, or as the second comultiplication on the R-algebra of quasi-symmetric functions) is an R-algebra homomorphism $\Delta^{\times}$ from the R-algebra of quasi-symmetric functions to the tensor square (over R) of quasi-symmetric functions. It can be defined in the following two ways:

1. If I is a composition, then a $(0, I)$-matrix will mean a matrix whose entries are nonnegative integers such that no row and no column of this matrix is zero, and such that if all the nonzero entries of the matrix are read (row by row, starting at the topmost row, reading every row from left to right), then the reading word obtained is I. If A is a $(0, I)$-matrix, then row(A) will denote the vector of row sums of A (regarded as a composition), and column(A) will denote the vector of column sums of A (regarded as a composition).

For every composition I, the internal coproduct $\Delta^\times(M_I)$ of the I-th monomial quasisymmetric function M_I is the sum

$$\sum_{A \text{ is a } (0,I)\text{-matrix}} M_{\text{row}(A)} \otimes M_{\text{column}(A)}.$$

See Section 11.39 of [HazWitt1].

2. For every permutation w, let $C(w)$ denote the descent composition of w. Then, for any composition I of size n, the internal coproduct $\Delta^\times(F_I)$ of the I-th fundamental quasisymmetric function F_I is the sum

$$\sum_{\substack{\sigma\in S_n,\\ \tau\in S_n,\\ \tau\sigma=\pi}} F_{C(\sigma)} \otimes F_{C(\tau)},$$

where π is any permutation in S_n having descent composition I and where permutations act from the left and multiply accordingly, so $\tau\sigma$ means first applying σ and then τ. See Theorem 4.23 in [Mal1993], but beware of the notations which are apparently different from those in [HazWitt1].

The restriction of the internal coproduct to the R-algebra of symmetric functions is the well-known internal coproduct on the symmetric functions.

The method `kronecker_coproduct()` is a synonym of this one.

EXAMPLES:

Let us compute the internal coproduct of M_{21} (which is short for $M_{[2,1]}$). The $(0,[2,1])$-matrices are

$$\begin{bmatrix}2 & 1\end{bmatrix}, \begin{bmatrix}2\\1\end{bmatrix}, \begin{bmatrix}2 & 0\\0 & 1\end{bmatrix}, \text{ and } \begin{bmatrix}0 & 2\\1 & 0\end{bmatrix}$$

so

$$\Delta^\times(M_{21}) = M_3 \otimes M_{21} + M_{21} \otimes M_3 + M_{21} \otimes M_{21} + M_{21} \otimes M_{12}.$$

This is confirmed by the following Sage computation (incidentally demonstrating the non-cocommutativity of the internal coproduct):

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: a = M([2,1])
sage: a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(ZZ).M()
>>> a = M([Integer(2),Integer(1)])
>>> a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

Further examples:

```
sage: all( M([i]).internal_coproduct() == tensor([M([i]), M([i])])
....:      for i in range(1, 4) )
True

sage: M([1, 2]).internal_coproduct()
M[1, 2] # M[1, 2] + M[1, 2] # M[2, 1] + M[1, 2] # M[3] + M[3] # M[1, 2]
```

```
>>> from sage.all import *
>>> all( M([i]).internal_coproduct() == tensor([M([i]), M([i])])
...      for i in range(Integer(1), Integer(4)) )
True

>>> M([Integer(1), Integer(2)]).internal_coproduct()
M[1, 2] # M[1, 2] + M[1, 2] # M[2, 1] + M[1, 2] # M[3] + M[3] # M[1, 2]
```

The definition of $\Delta^{\times}(M_I)$ in terms of $(0, I)$-matrices is not suitable for computation in cases where the length of I is large, but we can use it as a doctest. Here is a naive implementation:

```
sage: def naive_internal_coproduct_on_M(I):
....:     # INPUT: composition I
....:     #        (not quasi-symmetric function)
....:     # OUTPUT: interior coproduct of M_I
....:     M = QuasiSymmetricFunctions(ZZ).M()
....:     M2 = M.tensor(M)
....:     res = M2.zero()
....:     l = len(I)
....:     n = I.size()
....:     for S in Subsets(range(l**2), l):
....:         M_list = sorted(S)
....:         row_M = [sum([I[M_list.index(l * i + j)]
....:                       for j in range(l) if
....:                       l * i + j in S])
....:                  for i in range(l)]
....:         col_M = [sum([I[M_list.index(l * i + j)]
....:                       for i in range(l) if
....:                       l * i + j in S])
....:                  for j in range(l)]
....:         if 0 in row_M:
....:             first_zero = row_M.index(0)
....:             row_M = row_M[:first_zero]
....:             if sum(row_M) != n:
....:                 continue
....:         if 0 in col_M:
....:             first_zero = col_M.index(0)
....:             col_M = col_M[:first_zero]
....:             if sum(col_M) != n:
....:                 continue
....:         res += tensor([M(Compositions(n)(row_M)),
....:                        M(Compositions(n)(col_M))])
....:     return res
sage: all( naive_internal_coproduct_on_M(I)
....:      == M(I).internal_coproduct()
....:      for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> def naive_internal_coproduct_on_M(I):
...     # INPUT: composition I
...     #        (not quasi-symmetric function)
...     # OUTPUT: interior coproduct of M_I
```

(continues on next page)

(continued from previous page)

```
...     M = QuasiSymmetricFunctions(ZZ).M()
...     M2 = M.tensor(M)
...     res = M2.zero()
...     l = len(I)
...     n = I.size()
...     for S in Subsets(range(l**Integer(2)), l):
...         M_list = sorted(S)
...         row_M = [sum([I[M_list.index(l * i + j)]
...                       for j in range(l) if
...                       l * i + j in S])
...                  for i in range(l)]
...         col_M = [sum([I[M_list.index(l * i + j)]
...                       for i in range(l) if
...                       l * i + j in S])
...                  for j in range(l)]
...         if Integer(0) in row_M:
...             first_zero = row_M.index(Integer(0))
...             row_M = row_M[:first_zero]
...             if sum(row_M) != n:
...                 continue
...         if Integer(0) in col_M:
...             first_zero = col_M.index(Integer(0))
...             col_M = col_M[:first_zero]
...             if sum(col_M) != n:
...                 continue
...         res += tensor([M(Compositions(n)(row_M)),
...                        M(Compositions(n)(col_M))])
...     return res
>>> all( naive_internal_coproduct_on_M(I)
...      == M(I).internal_coproduct()
...      for I in Compositions(Integer(3)) )
True
```

Todo

Implement this directly on the monomial basis maybe? The $(0, I)$-matrices are a pain to generate from their definition, but maybe there is a good algorithm. If so, the above "further examples" should be moved to the M-method.

omega_involution()

Return the image of the quasisymmetric function `self` under the omega involution.

The omega involution is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the fundamental quasisymmetric function F_I to F_{I^t}, where I^t denotes the conjugate (`conjugate()`) of the composition I. This map is commonly denoted by ω. It is an algebra homomorphism and a coalgebra antihomomorphism; it also is an involution and an automorphism of the graded vector space $QSym$. Also, every composition I satisfies

$$\omega(M_I) = (-1)^{|I|-\ell(I)} \sum M_J,$$

where the sum ranges over all compositions J of $|I|$ which are coarser than the reversed composition I^r of I. Here, $\ell(I)$ denotes the length of the composition I (that is, the number of parts of I).

If f is a homogeneous element of $NCSF$ of degree n, then

$$\omega(f) = (-1)^n S(f),$$

where S denotes the antipode of $QSym$.

The restriction of ω to the ring of symmetric functions (which is a subring of $QSym$) is precisely the omega involution (`omega()`) of said ring.

The omega involution on $QSym$ is adjoint to the omega involution on $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The omega involution has been denoted by ω in [LMvW13], section 3.6.

See also:

`omega involution on NCSF`, `psi involution on QSym`, `star involution on QSym`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F()
sage: F[3,2].omega_involution()
F[1, 2, 1, 1]
sage: F[6,3].omega_involution()
F[1, 1, 2, 1, 1, 1, 1, 1]
sage: (F[9,1] - F[8,2] + 2*F[2,4] - 3*F[3] + 4*F[[]]).omega_
↪involution()
4*F[] - 3*F[1, 1, 1] + 2*F[1, 1, 1, 2, 1] - F[1, 2, 1, 1, 1, 1, 1, 1,␣
↪1] + F[2, 1, 1, 1, 1, 1, 1, 1, 1]
sage: (F[3,3] - 2*F[2]).omega_involution()
-2*F[1, 1] + F[1, 1, 2, 1, 1]
sage: F([2,1,1]).omega_involution()
F[3, 1]
sage: M = QSym.M()
sage: M([2,1]).omega_involution()
-M[1, 2] - M[3]
sage: M.zero().omega_involution()
0
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> F = QSym.F()
>>> F[Integer(3),Integer(2)].omega_involution()
F[1, 2, 1, 1]
>>> F[Integer(6),Integer(3)].omega_involution()
F[1, 1, 2, 1, 1, 1, 1, 1]
>>> (F[Integer(9),Integer(1)] - F[Integer(8),Integer(2)] +␣
↪Integer(2)*F[Integer(2),Integer(4)] - Integer(3)*F[Integer(3)] +␣
↪Integer(4)*F[[]]).omega_involution()
4*F[] - 3*F[1, 1, 1] + 2*F[1, 1, 1, 2, 1] - F[1, 2, 1, 1, 1, 1, 1, 1,␣
↪1] + F[2, 1, 1, 1, 1, 1, 1, 1, 1]
>>> (F[Integer(3),Integer(3)] - Integer(2)*F[Integer(2)]).omega_
```

(continues on next page)

(continued from previous page)

```
↪involution()
-2*F[1, 1] + F[1, 1, 2, 1, 1]
>>> F([Integer(2),Integer(1),Integer(1)]).omega_involution()
F[3, 1]
>>> M = QSym.M()
>>> M([Integer(2),Integer(1)]).omega_involution()
-M[1, 2] - M[3]
>>> M.zero().omega_involution()
0
```

Testing the fact that the restriction of ω to Sym is the omega automorphism of Sym:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: all( F(e[lam]).omega_involution() == F(e[lam].omega())
....:      for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> e = Sym.e()
>>> all( F(e[lam]).omega_involution() == F(e[lam].omega())
...      for lam in Partitions(Integer(4)) )
True
```

psi_involution()

Return the image of the quasisymmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the fundamental quasisymmetric function F_I to F_{I^c}, where I^c denotes the complement of the composition I. The map ψ is an involution and a graded Hopf algebra automorphism of $QSym$. Its restriction to the ring of symmetric functions coincides with the omega automorphism of the latter ring.

The involution ψ of $QSym$ is adjoint to the involution ψ of $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also

psi involution on NCSF, *star involution on QSym*.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F()
sage: F[3,2].psi_involution()
F[1, 1, 2, 1]
sage: F[6,3].psi_involution()
F[1, 1, 1, 1, 1, 2, 1, 1]
sage: (F[9,1] - F[8,2] + 2*F[2,4] - 3*F[3] + 4*F[[]]).psi_involution()
```

(continues on next page)

(continued from previous page)

```
4*F[] - 3*F[1, 1, 1] + F[1, 1, 1, 1, 1, 1, 1, 1, 2] - F[1, 1, 1, 1, 1,␣
↪1, 1, 2, 1] + 2*F[1, 2, 1, 1, 1]
sage: (F[3,3] - 2*F[2]).psi_involution()
-2*F[1, 1] + F[1, 1, 2, 1, 1]
sage: F([2,1,1]).psi_involution()
F[1, 3]
sage: M = QSym.M()
sage: M([2,1]).psi_involution()
-M[2, 1] - M[3]
sage: M.zero().psi_involution()
0
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> F = QSym.F()
>>> F[Integer(3),Integer(2)].psi_involution()
F[1, 1, 2, 1]
>>> F[Integer(6),Integer(3)].psi_involution()
F[1, 1, 1, 1, 1, 2, 1, 1]
>>> (F[Integer(9),Integer(1)] - F[Integer(8),Integer(2)] +␣
↪Integer(2)*F[Integer(2),Integer(4)] - Integer(3)*F[Integer(3)] +␣
↪Integer(4)*F[[]]).psi_involution()
4*F[] - 3*F[1, 1, 1] + F[1, 1, 1, 1, 1, 1, 1, 1, 2] - F[1, 1, 1, 1, 1,␣
↪1, 1, 2, 1] + 2*F[1, 2, 1, 1, 1]
>>> (F[Integer(3),Integer(3)] - Integer(2)*F[Integer(2)]).psi_
↪involution()
-2*F[1, 1] + F[1, 1, 2, 1, 1]
>>> F([Integer(2),Integer(1),Integer(1)]).psi_involution()
F[1, 3]
>>> M = QSym.M()
>>> M([Integer(2),Integer(1)]).psi_involution()
-M[2, 1] - M[3]
>>> M.zero().psi_involution()
0
```

The involution ψ commutes with the antipode:

```
sage: all( F(I).psi_involution().antipode()
....:      == F(I).antipode().psi_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( F(I).psi_involution().antipode()
...      == F(I).antipode().psi_involution()
...      for I in Compositions(Integer(4)) )
True
```

Testing the fact that the restriction of ψ to Sym is the omega automorphism of Sym:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
```

(continues on next page)

(continued from previous page)

```
sage: all( F(e[lam]).psi_involution() == F(e[lam].omega())
....:      for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> e = Sym.e()
>>> all( F(e[lam]).psi_involution() == F(e[lam].omega())
...      for lam in Partitions(Integer(4)) )
True
```

star_involution()

Return the image of the quasisymmetric function `self` under the star involution.

The star involution is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the monomial quasisymmetric function M_I to M_{I^r}. Here, if I is a composition, we denote by I^r the reversed composition of I. Denoting by f^* the image of an element $f \in QSym$ under the star involution, it can be shown that every composition I satisfies

$$(M_I)^* = M_{I^r}, \quad (F_I)^* = F_{I^r},$$

where F_I denotes the fundamental quasisymmetric function corresponding to the composition I. The star involution is an involution, an algebra automorphism and a coalgebra anti-automorphism of $QSym$. It also is an automorphism of the graded vector space $QSym$, and is the identity on the subspace Sym of $QSym$. It is adjoint to the star involution on $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6.

See also

star involution on NCSF.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: M = QSym.M()
sage: M[3,2].star_involution()
M[2, 3]
sage: M[6,3].star_involution()
M[3, 6]
sage: (M[9,1] - M[6,2] + 2*M[6,4] - 3*M[3] + 4*M[[]]).star_involution()
4*M[] + M[1, 9] - M[2, 6] - 3*M[3] + 2*M[4, 6]
sage: (M[3,3] - 2*M[2]).star_involution()
-2*M[2] + M[3, 3]
sage: M([4,2]).star_involution()
M[2, 4]
sage: dI = QSym.dI()
sage: dI([1,2]).star_involution()
-dI[1, 2] + dI[2, 1]
sage: dI.zero().star_involution()
0
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> M = QSym.M()
>>> M[Integer(3),Integer(2)].star_involution()
M[2, 3]
>>> M[Integer(6),Integer(3)].star_involution()
M[3, 6]
>>> (M[Integer(9),Integer(1)] - M[Integer(6),Integer(2)] +␣
↪Integer(2)*M[Integer(6),Integer(4)] - Integer(3)*M[Integer(3)] +␣
↪Integer(4)*M[[]]).star_involution()
4*M[] + M[1, 9] - M[2, 6] - 3*M[3] + 2*M[4, 6]
>>> (M[Integer(3),Integer(3)] - Integer(2)*M[Integer(2)]).star_
↪involution()
-2*M[2] + M[3, 3]
>>> M([Integer(4),Integer(2)]).star_involution()
M[2, 4]
>>> dI = QSym.dI()
>>> dI([Integer(1),Integer(2)]).star_involution()
-dI[1, 2] + dI[2, 1]
>>> dI.zero().star_involution()
0
```

The star involution commutes with the antipode:

```
sage: all( M(I).star_involution().antipode()
....:      == M(I).antipode().star_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> all( M(I).star_involution().antipode()
...      == M(I).antipode().star_involution()
...      for I in Compositions(Integer(4)) )
True
```

The star involution is the identity on Sym:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: all( M(e(lam)).star_involution() == M(e(lam))
....:      for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> e = Sym.e()
>>> all( M(e(lam)).star_involution() == M(e(lam))
...      for lam in Partitions(Integer(4)) )
True
```

to_symmetric_function()

Convert a quasi-symmetric function to a symmetric function.

OUTPUT:

- If `self` is a symmetric function, then return the expansion in the monomial basis. Otherwise raise an error.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: F = QSym.Fundamental()
sage: (F[3,2] + F[2,3]).to_symmetric_function()
Traceback (most recent call last):
...
ValueError: F[2, 3] + F[3, 2] is not a symmetric function
sage: m = SymmetricFunctions(QQ).m()
sage: s = SymmetricFunctions(QQ).s()
sage: F(s[3,1,1]).to_symmetric_function()
6*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1]
sage: m(s[3,1,1])
6*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> F = QSym.Fundamental()
>>> (F[Integer(3),Integer(2)] + F[Integer(2),Integer(3)]).to_symmetric_
→function()
Traceback (most recent call last):
...
ValueError: F[2, 3] + F[3, 2] is not a symmetric function
>>> m = SymmetricFunctions(QQ).m()
>>> s = SymmetricFunctions(QQ).s()
>>> F(s[Integer(3),Integer(1),Integer(1)]).to_symmetric_function()
6*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1]
>>> m(s[Integer(3),Integer(1),Integer(1)])
6*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1]
```

class ParentMethods

Bases: `object`

Methods common to all bases of `QuasiSymmetricFunctions`.

Eulerian(*n*, *j*, *k=None*)

Return the Eulerian (quasi)symmetric function $Q_{n,j}$ in terms of `self`.

INPUT:

- `n` – the value n or a partition
- `j` – the number of excedances
- `k` – (optional) if specified, determines the number of fixed points of the permutation

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: M.Eulerian(3, 1)
4*M[1, 1, 1] + 3*M[1, 2] + 3*M[2, 1] + 2*M[3]
sage: M.Eulerian(4, 1, 2)
6*M[1, 1, 1, 1] + 4*M[1, 1, 2] + 4*M[1, 2, 1]
 + 2*M[1, 3] + 4*M[2, 1, 1] + 3*M[2, 2] + 2*M[3, 1] + M[4]
sage: QS = QSym.QS()
```

(continues on next page)

(continued from previous page)

```
sage: QS.Eulerian(4, 2)
2*QS[1, 3] + QS[2, 2] + 2*QS[3, 1] + 3*QS[4]
sage: QS.Eulerian([2, 2, 1], 2)
QS[1, 2, 2] + QS[1, 4] + QS[2, 1, 2] + QS[2, 2, 1]
 + QS[2, 3] + QS[3, 2] + QS[4, 1] + QS[5]
sage: dI = QSym.dI()
sage: dI.Eulerian(5, 2)
-dI[1, 3, 1] - 5*dI[1, 4] + dI[2, 2, 1] + dI[3, 1, 1]
 + 5*dI[3, 2] + 6*dI[4, 1] + 6*dI[5]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.M()
>>> M.Eulerian(Integer(3), Integer(1))
4*M[1, 1, 1] + 3*M[1, 2] + 3*M[2, 1] + 2*M[3]
>>> M.Eulerian(Integer(4), Integer(1), Integer(2))
6*M[1, 1, 1, 1] + 4*M[1, 1, 2] + 4*M[1, 2, 1]
 + 2*M[1, 3] + 4*M[2, 1, 1] + 3*M[2, 2] + 2*M[3, 1] + M[4]
>>> QS = QSym.QS()
>>> QS.Eulerian(Integer(4), Integer(2))
2*QS[1, 3] + QS[2, 2] + 2*QS[3, 1] + 3*QS[4]
>>> QS.Eulerian([Integer(2), Integer(2), Integer(1)], Integer(2))
QS[1, 2, 2] + QS[1, 4] + QS[2, 1, 2] + QS[2, 2, 1]
 + QS[2, 3] + QS[3, 2] + QS[4, 1] + QS[5]
>>> dI = QSym.dI()
>>> dI.Eulerian(Integer(5), Integer(2))
-dI[1, 3, 1] - 5*dI[1, 4] + dI[2, 2, 1] + dI[3, 1, 1]
 + 5*dI[3, 2] + 6*dI[4, 1] + 6*dI[5]
```

from_polynomial(*f*, *check=True*)

The quasi-symmetric function expanded in this basis corresponding to the quasi-symmetric polynomial `f`.

This is a default implementation that computes the expansion in the Monomial basis and converts to this basis.

INPUT:

- `f` – a polynomial in finitely many variables over the same base ring as `self`. It is assumed that this polynomial is quasi-symmetric.
- `check` – boolean (default: `True`); checks whether the polynomial is indeed quasi-symmetric

OUTPUT: quasi-symmetric function

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: P = PolynomialRing(QQ, 'x', 3)
sage: x = P.gens()
sage: f = x[0] + x[1] + x[2]
sage: M.from_polynomial(f)
M[1]
sage: F.from_polynomial(f)
F[1]
```

(continues on next page)

(continued from previous page)

```
sage: f = x[0]**2+x[1]**2+x[2]**2
sage: M.from_polynomial(f)
M[2]
sage: F.from_polynomial(f)
-F[1, 1] + F[2]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> P = PolynomialRing(QQ, 'x', Integer(3))
>>> x = P.gens()
>>> f = x[Integer(0)] + x[Integer(1)] + x[Integer(2)]
>>> M.from_polynomial(f)
M[1]
>>> F.from_polynomial(f)
F[1]
>>> f = x[Integer(0)]**Integer(2)+x[Integer(1)]**Integer(2)+x[Integer(2
↪)]**Integer(2)
>>> M.from_polynomial(f)
M[2]
>>> F.from_polynomial(f)
-F[1, 1] + F[2]
```

If the polynomial is not quasi-symmetric, an error is raised:

```
sage: f = x[0]^2+x[1]
sage: M.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a quasi-symmetric polynomial
sage: F.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a quasi-symmetric polynomial
```

```
>>> from sage.all import *
>>> f = x[Integer(0)]**Integer(2)+x[Integer(1)]
>>> M.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a quasi-symmetric polynomial
>>> F.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a quasi-symmetric polynomial
```

super_categories()

Return the super categories of bases of the Quasi-symmetric functions.

OUTPUT: list of categories

E

alias of *Essential*

class Essential(*QSym*)

Bases: `CombinatorialFreeModule`, `BindableClass`

The Hopf algebra of quasi-symmetric functions in the Essential basis.

The Essential quasi-symmetric functions are defined by

$$E_I = \sum_{J \geq I} M_J = \sum_{i_1 \leq \cdots \leq i_k} x_{i_1}^{I_1} \cdots x_{i_k}^{I_k},$$

where $I = (I_1, \ldots, I_k)$.

Note: Our convention of $\leq$ and $\geq$ of compositions is opposite that of [Hoff2015].

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: E = QSym.E()
sage: M = QSym.M()
sage: E(M[2,2])
E[2, 2] - E[4]
sage: s = SymmetricFunctions(QQ).s()
sage: E(s[3,2])
5*E[1, 1, 1, 1, 1] - 2*E[1, 1, 1, 2] - 2*E[1, 1, 2, 1]
 - 2*E[1, 2, 1, 1] + E[1, 2, 2] - 2*E[2, 1, 1, 1]
 + E[2, 1, 2] + E[2, 2, 1]
sage: (1 + E[1])^3
E[] + 3*E[1] + 6*E[1, 1] + 6*E[1, 1, 1] - 3*E[1, 2]
 - 3*E[2] - 3*E[2, 1] + E[3]
sage: E[1,2,1].coproduct()
E[] # E[1, 2, 1] + E[1] # E[2, 1] + E[1, 2] # E[1] + E[1, 2, 1] # E[]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> E = QSym.E()
>>> M = QSym.M()
>>> E(M[Integer(2),Integer(2)])
E[2, 2] - E[4]
>>> s = SymmetricFunctions(QQ).s()
>>> E(s[Integer(3),Integer(2)])
5*E[1, 1, 1, 1, 1] - 2*E[1, 1, 1, 2] - 2*E[1, 1, 2, 1]
 - 2*E[1, 2, 1, 1] + E[1, 2, 2] - 2*E[2, 1, 1, 1]
 + E[2, 1, 2] + E[2, 2, 1]
>>> (Integer(1) + E[Integer(1)])**Integer(3)
E[] + 3*E[1] + 6*E[1, 1] + 6*E[1, 1, 1] - 3*E[1, 2]
 - 3*E[2] - 3*E[2, 1] + E[3]
>>> E[Integer(1),Integer(2),Integer(1)].coproduct()
E[] # E[1, 2, 1] + E[1] # E[2, 1] + E[1, 2] # E[1] + E[1, 2, 1] # E[]
```

The following is an alias for this basis:

```
sage: QSym.Essential()
Quasisymmetric functions over the Rational Field in the Essential basis
```

```
>>> from sage.all import *
>>> QSym.Essential()
Quasisymmetric functions over the Rational Field in the Essential basis
```

antipode_on_basis(*compo*)

Return the result of the antipode applied to a quasi-symmetric Essential basis element.

INPUT:

- `compo` – composition

OUTPUT:

- The result of the antipode applied to the composition `compo`, expressed in the Essential basis.

EXAMPLES:

```
sage: E = QuasiSymmetricFunctions(QQ).E()
sage: E.antipode_on_basis(Composition([2,1]))
E[1, 2] - E[3]
sage: E.antipode_on_basis(Composition([]))
E[]
```

```
>>> from sage.all import *
>>> E = QuasiSymmetricFunctions(QQ).E()
>>> E.antipode_on_basis(Composition([Integer(2),Integer(1)]))
E[1, 2] - E[3]
>>> E.antipode_on_basis(Composition([]))
E[]
```

coproduct_on_basis(*compo*)

Return the coproduct of a Essential basis element.

Combinatorial rule: deconcatenation.

INPUT:

- `compo` – composition

OUTPUT:

- The coproduct applied to the Essential quasi-symmetric function indexed by `compo`, expressed in the Essential basis.

EXAMPLES:

```
sage: E = QuasiSymmetricFunctions(QQ).Essential()
sage: E[4,2,3].coproduct()
E[] # E[4, 2, 3] + E[4] # E[2, 3] + E[4, 2] # E[3] + E[4, 2, 3] # E[]
sage: E.coproduct_on_basis(Composition([]))
E[] # E[]
```

```
>>> from sage.all import *
>>> E = QuasiSymmetricFunctions(QQ).Essential()
>>> E[Integer(4),Integer(2),Integer(3)].coproduct()
E[] # E[4, 2, 3] + E[4] # E[2, 3] + E[4, 2] # E[3] + E[4, 2, 3] # E[]
>>> E.coproduct_on_basis(Composition([]))
E[] # E[]
```

product_on_basis(*I*, *J*)

The product on Essential basis elements.

The product of the basis elements indexed by two compositions I and J is the sum of the basis elements indexed by compositions K in the stuffle product (also called the overlapping shuffle product) of I and J with a coefficient of $(-1)^{\ell(I)+\ell(J)-\ell(K)}$, where $\ell(C)$ is the length of the composition C.

INPUT:

- `I`, `J` – compositions

OUTPUT:

- The product of the Essential quasi-symmetric functions indexed by `I` and `J`, expressed in the Essential basis.

EXAMPLES:

```
sage: E = QuasiSymmetricFunctions(QQ).E()
sage: c1 = Composition([2])
sage: c2 = Composition([1,3])
sage: E.product_on_basis(c1, c2)
E[1, 2, 3] + E[1, 3, 2] - E[1, 5] + E[2, 1, 3] - E[3, 3]
sage: E.product_on_basis(c1, Composition([]))
E[2]
sage: E.product_on_basis(c1, Composition([3]))
E[2, 3] + E[3, 2] - E[5]
```

```
>>> from sage.all import *
>>> E = QuasiSymmetricFunctions(QQ).E()
>>> c1 = Composition([Integer(2)])
>>> c2 = Composition([Integer(1),Integer(3)])
>>> E.product_on_basis(c1, c2)
E[1, 2, 3] + E[1, 3, 2] - E[1, 5] + E[2, 1, 3] - E[3, 3]
>>> E.product_on_basis(c1, Composition([]))
E[2]
>>> E.product_on_basis(c1, Composition([Integer(3)]))
E[2, 3] + E[3, 2] - E[5]
```

F

alias of `Fundamental`

class Fundamental(*QSym*)

Bases: `CombinatorialFreeModule`, `BindableClass`

The Hopf algebra of quasi-symmetric functions in the Fundamental basis.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: F = QSym.F()
sage: M = QSym.M()
sage: F(M[2,2])
F[1, 1, 1, 1] - F[1, 1, 2] - F[2, 1, 1] + F[2, 2]
sage: s = SymmetricFunctions(QQ).s()
sage: F(s[3,2])
F[1, 2, 2] + F[1, 3, 1] + F[2, 2, 1] + F[2, 3] + F[3, 2]
sage: (1+F[1])^3
F[] + 3*F[1] + 3*F[1, 1] + F[1, 1, 1] + 2*F[1, 2] + 3*F[2] + 2*F[2, 1] + F[3]
```

(continues on next page)

(continued from previous page)

```
sage: F[1,2,1].coproduct()
F[] # F[1, 2, 1] + F[1] # F[2, 1] + F[1, 1] # F[1, 1] + F[1, 2] # F[1] + F[1,
→2, 1] # F[]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> F = QSym.F()
>>> M = QSym.M()
>>> F(M[Integer(2),Integer(2)])
F[1, 1, 1, 1] - F[1, 1, 2] - F[2, 1, 1] + F[2, 2]
>>> s = SymmetricFunctions(QQ).s()
>>> F(s[Integer(3),Integer(2)])
F[1, 2, 2] + F[1, 3, 1] + F[2, 2, 1] + F[2, 3] + F[3, 2]
>>> (Integer(1)+F[Integer(1)])**Integer(3)
F[] + 3*F[1] + 3*F[1, 1] + F[1, 1, 1] + 2*F[1, 2] + 3*F[2] + 2*F[2, 1] + F[3]
>>> F[Integer(1),Integer(2),Integer(1)].coproduct()
F[] # F[1, 2, 1] + F[1] # F[2, 1] + F[1, 1] # F[1, 1] + F[1, 2] # F[1] + F[1,
→2, 1] # F[]
```

The following is an alias for this basis:

```
sage: QSym.Fundamental()
Quasisymmetric functions over the Rational Field in the Fundamental basis
```

```
>>> from sage.all import *
>>> QSym.Fundamental()
Quasisymmetric functions over the Rational Field in the Fundamental basis
```

class Element

Bases: `IndexedFreeModuleElement`

internal_coproduct()

Return the inner coproduct of `self` in the Fundamental basis.

The inner coproduct (also known as the Kronecker coproduct, or as the second comultiplication on the R-algebra of quasi-symmetric functions) is an R-algebra homomorphism $\Delta^\times$ from the R-algebra of quasi-symmetric functions to the tensor square (over R) of quasi-symmetric functions. It can be defined in the following two ways:

1. If I is a composition, then a $(0, I)$-matrix will mean a matrix whose entries are nonnegative integers such that no row and no column of this matrix is zero, and such that if all the nonzero entries of the matrix are read (row by row, starting at the topmost row, reading every row from left to right), then the reading word obtained is I. If A is a $(0, I)$-matrix, then $\mathrm{row}(A)$ will denote the vector of row sums of A (regarded as a composition), and $\mathrm{column}(A)$ will denote the vector of column sums of A (regarded as a composition).

   For every composition I, the internal coproduct $\Delta^\times(M_I)$ of the I-th monomial quasisymmetric function M_I is the sum

$$\sum_{A \text{ is a } (0,I)\text{-matrix}} M_{\mathrm{row}(A)} \otimes M_{\mathrm{column}(A)}.$$

   See Section 11.39 of [HazWitt1].

2. For every permutation w, let $C(w)$ denote the descent composition of w. Then, for any composition I of size n, the internal coproduct $\Delta^\times(F_I)$ of the I-th fundamental quasisymmetric function

F_I is the sum

$$\sum_{\substack{\sigma \in S_n, \\ \tau \in S_n, \\ \tau\sigma = \pi}} F_{C(\sigma)} \otimes F_{C(\tau)},$$

where π is any permutation in S_n having descent composition I and where permutations act from the left and multiply accordingly, so $\tau\sigma$ means first applying σ and then τ. See Theorem 4.23 in [Mal1993], but beware of the notations which are apparently different from those in [HazWitt1].

The restriction of the internal coproduct to the R-algebra of symmetric functions is the well-known internal coproduct on the symmetric functions.

The method `kronecker_coproduct()` is a synonym of this one.

EXAMPLES:

Let us compute the internal coproduct of M_{21} (which is short for $M_{[2,1]}$). The $(0, [2, 1])$-matrices are

$$\begin{bmatrix} 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}$$

so

$$\Delta^{\times}(M_{21}) = M_3 \otimes M_{21} + M_{21} \otimes M_3 + M_{21} \otimes M_{21} + M_{21} \otimes M_{12}.$$

This is confirmed by the following Sage computation (incidentally demonstrating the non-cocommutativity of the internal coproduct):

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: a = M([2,1])
sage: a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(ZZ).M()
>>> a = M([Integer(2),Integer(1)])
>>> a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

Some examples on the Fundamental basis:

```
sage: F = QuasiSymmetricFunctions(ZZ).F()
sage: F([1,1]).internal_coproduct()
F[1, 1] # F[2] + F[2] # F[1, 1]
sage: F([2]).internal_coproduct()
F[1, 1] # F[1, 1] + F[2] # F[2]
sage: F([3]).internal_coproduct()
F[1, 1, 1] # F[1, 1, 1] + F[1, 2] # F[1, 2] + F[1, 2] # F[2, 1]
 + F[2, 1] # F[1, 2] + F[2, 1] # F[2, 1] + F[3] # F[3]
sage: F([1,2]).internal_coproduct()
F[1, 1, 1] # F[1, 2] + F[1, 2] # F[2, 1] + F[1, 2] # F[3]
 + F[2, 1] # F[1, 1, 1] + F[2, 1] # F[2, 1] + F[3] # F[1, 2]
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(ZZ).F()
```

(continues on next page)

(continued from previous page)

```
>>> F([Integer(1),Integer(1)]).internal_coproduct()
F[1, 1] # F[2] + F[2] # F[1, 1]
>>> F([Integer(2)]).internal_coproduct()
F[1, 1] # F[1, 1] + F[2] # F[2]
>>> F([Integer(3)]).internal_coproduct()
F[1, 1, 1] # F[1, 1, 1] + F[1, 2] # F[1, 2] + F[1, 2] # F[2, 1]
 + F[2, 1] # F[1, 2] + F[2, 1] # F[2, 1] + F[3] # F[3]
>>> F([Integer(1),Integer(2)]).internal_coproduct()
F[1, 1, 1] # F[1, 2] + F[1, 2] # F[2, 1] + F[1, 2] # F[3]
 + F[2, 1] # F[1, 1, 1] + F[2, 1] # F[2, 1] + F[3] # F[1, 2]
```

kronecker_coproduct()

Return the inner coproduct of `self` in the Fundamental basis.

The inner coproduct (also known as the Kronecker coproduct, or as the second comultiplication on the R-algebra of quasi-symmetric functions) is an R-algebra homomorphism $\Delta^{\times}$ from the R-algebra of quasi-symmetric functions to the tensor square (over R) of quasi-symmetric functions. It can be defined in the following two ways:

1. If I is a composition, then a $(0, I)$-matrix will mean a matrix whose entries are nonnegative integers such that no row and no column of this matrix is zero, and such that if all the nonzero entries of the matrix are read (row by row, starting at the topmost row, reading every row from left to right), then the reading word obtained is I. If A is a $(0, I)$-matrix, then row(A) will denote the vector of row sums of A (regarded as a composition), and column(A) will denote the vector of column sums of A (regarded as a composition).

   For every composition I, the internal coproduct $\Delta^{\times}(M_I)$ of the I-th monomial quasisymmetric function M_I is the sum

   $$\sum_{A \text{ is a } (0,I)\text{-matrix}} M_{\text{row}(A)} \otimes M_{\text{column}(A)}.$$

   See Section 11.39 of [HazWitt1].

2. For every permutation w, let $C(w)$ denote the descent composition of w. Then, for any composition I of size n, the internal coproduct $\Delta^{\times}(F_I)$ of the I-th fundamental quasisymmetric function F_I is the sum

   $$\sum_{\substack{\sigma \in S_n, \\ \tau \in S_n, \\ \tau\sigma = \pi}} F_{C(\sigma)} \otimes F_{C(\tau)},$$

   where π is any permutation in S_n having descent composition I and where permutations act from the left and multiply accordingly, so $\tau\sigma$ means first applying σ and then τ. See Theorem 4.23 in [Mal1993], but beware of the notations which are apparently different from those in [HazWitt1].

The restriction of the internal coproduct to the R-algebra of symmetric functions is the well-known internal coproduct on the symmetric functions.

The method `kronecker_coproduct()` is a synonym of this one.

EXAMPLES:

Let us compute the internal coproduct of M_{21} (which is short for $M_{[2,1]}$). The $(0, [2, 1])$-matrices are

$$\begin{bmatrix} 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}$$

so

$$\Delta^{\times}(M_{21}) = M_3 \otimes M_{21} + M_{21} \otimes M_3 + M_{21} \otimes M_{21} + M_{21} \otimes M_{12}.$$

This is confirmed by the following Sage computation (incidentally demonstrating the non-cocommutativity of the internal coproduct):

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: a = M([2,1])
sage: a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(ZZ).M()
>>> a = M([Integer(2),Integer(1)])
>>> a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

Some examples on the Fundamental basis:

```
sage: F = QuasiSymmetricFunctions(ZZ).F()
sage: F([1,1]).internal_coproduct()
F[1, 1] # F[2] + F[2] # F[1, 1]
sage: F([2]).internal_coproduct()
F[1, 1] # F[1, 1] + F[2] # F[2]
sage: F([3]).internal_coproduct()
F[1, 1, 1] # F[1, 1, 1] + F[1, 2] # F[1, 2] + F[1, 2] # F[2, 1]
 + F[2, 1] # F[1, 2] + F[2, 1] # F[2, 1] + F[3] # F[3]
sage: F([1,2]).internal_coproduct()
F[1, 1, 1] # F[1, 2] + F[1, 2] # F[2, 1] + F[1, 2] # F[3]
 + F[2, 1] # F[1, 1, 1] + F[2, 1] # F[2, 1] + F[3] # F[1, 2]
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(ZZ).F()
>>> F([Integer(1),Integer(1)]).internal_coproduct()
F[1, 1] # F[2] + F[2] # F[1, 1]
>>> F([Integer(2)]).internal_coproduct()
F[1, 1] # F[1, 1] + F[2] # F[2]
>>> F([Integer(3)]).internal_coproduct()
F[1, 1, 1] # F[1, 1, 1] + F[1, 2] # F[1, 2] + F[1, 2] # F[2, 1]
 + F[2, 1] # F[1, 2] + F[2, 1] # F[2, 1] + F[3] # F[3]
>>> F([Integer(1),Integer(2)]).internal_coproduct()
F[1, 1, 1] # F[1, 2] + F[1, 2] # F[2, 1] + F[1, 2] # F[3]
 + F[2, 1] # F[1, 1, 1] + F[2, 1] # F[2, 1] + F[3] # F[1, 2]
```

star_involution()

Return the image of the quasisymmetric function `self` under the star involution.

The star involution is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the monomial quasisymmetric function M_I to M_{I^r}. Here, if I is a composition, we denote by I^r the reversed composition of I. Denoting by f^* the image of an element $f \in QSym$ under the star involution, it can be shown that every composition I satisfies

$$(M_I)^* = M_{I^r}, \quad (F_I)^* = F_{I^r},$$

where F_I denotes the fundamental quasisymmetric function corresponding to the composition I. The star involution is an involution, an algebra automorphism and a coalgebra anti-automorphism of $QSym$. It also is an automorphism of the graded vector space $QSym$, and is the identity on the subspace Sym of $QSym$. It is adjoint to the star involution on $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6.

See also:

star involution on QSym, *star involution on NCSF*.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F()
sage: F[3,1].star_involution()
F[1, 3]
sage: F[5,3].star_involution()
F[3, 5]
sage: (F[9,1] - F[6,2] + 2*F[6,4] - 3*F[3] + 4*F[[]]).star_involution()
4*F[] + F[1, 9] - F[2, 6] - 3*F[3] + 2*F[4, 6]
sage: (F[3,3] - 2*F[2]).star_involution()
-2*F[2] + F[3, 3]
sage: F([4,2]).star_involution()
F[2, 4]
sage: dI = QSym.dI()
sage: dI([1,2]).star_involution()
-dI[1, 2] + dI[2, 1]
sage: dI.zero().star_involution()
0
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> F = QSym.F()
>>> F[Integer(3),Integer(1)].star_involution()
F[1, 3]
>>> F[Integer(5),Integer(3)].star_involution()
F[3, 5]
>>> (F[Integer(9),Integer(1)] - F[Integer(6),Integer(2)] +␣
↪Integer(2)*F[Integer(6),Integer(4)] - Integer(3)*F[Integer(3)] +␣
↪Integer(4)*F[[]]).star_involution()
4*F[] + F[1, 9] - F[2, 6] - 3*F[3] + 2*F[4, 6]
>>> (F[Integer(3),Integer(3)] - Integer(2)*F[Integer(2)]).star_
↪involution()
-2*F[2] + F[3, 3]
>>> F([Integer(4),Integer(2)]).star_involution()
F[2, 4]
>>> dI = QSym.dI()
>>> dI([Integer(1),Integer(2)]).star_involution()
-dI[1, 2] + dI[2, 1]
>>> dI.zero().star_involution()
0
```

Eulerian(*n*, *j*, *k=None*)

Return the Eulerian (quasi)symmetric function $Q_{n,j}$ (with n either an integer or a partition) defined in

[SW2010] in terms of the fundamental quasisymmetric functions. Or, if the optional argument `k` is specified, return the function $Q_{n,j,k}$ defined ibidem.

If n and j are nonnegative integers, then the Eulerian quasisymmetric function $Q_{n,j}$ is defined as

$$Q_{n,j} := \sum_{\sigma} F_{\mathrm{Dex}(\sigma)},$$

where we sum over all permutations $\sigma \in S_n$ such that the number of excedances of σ is j, and where $\mathrm{Dex}(\sigma)$ is a composition of n defined as follows: Let S be the set of all $i \in \{1, 2, \ldots, n-1\}$ such that either $\sigma_i > \sigma_{i+1} > i+1$ or $i \geq \sigma_i > \sigma_{i+1}$ or $\sigma_{i+1} > i+1 > \sigma_i$. Then, $\mathrm{Dex}(\sigma)$ is set to be the composition of n whose descent set is S.

Here, an excedance of a permutation $\sigma \in S_n$ means an element $i \in \{1, 2, \ldots, n-1\}$ satisfying $\sigma_i > i$.

Similarly we can define a quasisymmetric function $Q_{\lambda,j}$ for every partition λ and every nonnegative integer j. This differs from $Q_{n,j}$ only in that the sum is restricted to all permutations $\sigma \in S_n$ whose cycle type is λ (where $n = |\lambda|$, and where we still require the number of excedances to be j). The method at hand allows computing these functions by passing λ as the `n` parameter.

Analogously we can define a quasisymmetric function $Q_{n,j,k}$ for any nonnegative integers n, j and k by restricting the sum to all permutations $\sigma \in S_n$ that have exactly k fixed points (and j excedances). This can be obtained by specifying the optional `k` argument in this method.

All three versions of Eulerian quasisymmetric functions ($Q_{n,j}$, $Q_{\lambda,j}$ and $Q_{n,j,k}$) are actually symmetric functions. See `Eulerian()`.

INPUT:

- `n` – the nonnegative integer n or a partition
- `j` – the number of excedances
- `k` – (optional) if specified, determines the number of fixed points of the permutations which are being summed over

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F.Eulerian(3, 1)
F[1, 2] + F[2, 1] + 2*F[3]
sage: F.Eulerian(4, 2)
F[1, 2, 1] + 2*F[1, 3] + 3*F[2, 2] + 2*F[3, 1] + 3*F[4]
sage: F.Eulerian(5, 2)
F[1, 1, 2, 1] + F[1, 1, 3] + F[1, 2, 1, 1] + 7*F[1, 2, 2] + 6*F[1, 3, 1]
↪+ 6*F[1, 4] + 2*F[2, 1, 2] + 7*F[2, 2, 1] + 11*F[2, 3] + F[3, 1, 1] +
↪11*F[3, 2] + 6*F[4, 1] + 6*F[5]
sage: F.Eulerian(4, 0)
F[4]
sage: F.Eulerian(4, 3)
F[4]
sage: F.Eulerian(4, 1, 2)
F[1, 2, 1] + F[1, 3] + 2*F[2, 2] + F[3, 1] + F[4]
sage: F.Eulerian(Partition([2, 2, 1]), 2)
F[1, 1, 2, 1] + F[1, 2, 1, 1] + 2*F[1, 2, 2] + F[1, 3, 1]
 + F[1, 4] + F[2, 1, 2] + 2*F[2, 2, 1] + 2*F[2, 3]
 + 2*F[3, 2] + F[4, 1] + F[5]
sage: F.Eulerian(0, 0)
F[]
sage: F.Eulerian(0, 1)
0
```

(continues on next page)

(continued from previous page)

```
sage: F.Eulerian(1, 0)
F[1]
sage: F.Eulerian(1, 1)
0
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(QQ).F()
>>> F.Eulerian(Integer(3), Integer(1))
F[1, 2] + F[2, 1] + 2*F[3]
>>> F.Eulerian(Integer(4), Integer(2))
F[1, 2, 1] + 2*F[1, 3] + 3*F[2, 2] + 2*F[3, 1] + 3*F[4]
>>> F.Eulerian(Integer(5), Integer(2))
F[1, 1, 2, 1] + F[1, 1, 3] + F[1, 2, 1, 1] + 7*F[1, 2, 2] + 6*F[1, 3, 1]␣
↪+ 6*F[1, 4] + 2*F[2, 1, 2] + 7*F[2, 2, 1] + 11*F[2, 3] + F[3, 1, 1] +␣
↪11*F[3, 2] + 6*F[4, 1] + 6*F[5]
>>> F.Eulerian(Integer(4), Integer(0))
F[4]
>>> F.Eulerian(Integer(4), Integer(3))
F[4]
>>> F.Eulerian(Integer(4), Integer(1), Integer(2))
F[1, 2, 1] + F[1, 3] + 2*F[2, 2] + F[3, 1] + F[4]
>>> F.Eulerian(Partition([Integer(2), Integer(2), Integer(1)]),␣
↪Integer(2))
F[1, 1, 2, 1] + F[1, 2, 1, 1] + 2*F[1, 2, 2] + F[1, 3, 1]
 + F[1, 4] + F[2, 1, 2] + 2*F[2, 2, 1] + 2*F[2, 3]
 + 2*F[3, 2] + F[4, 1] + F[5]
>>> F.Eulerian(Integer(0), Integer(0))
F[]
>>> F.Eulerian(Integer(0), Integer(1))
0
>>> F.Eulerian(Integer(1), Integer(0))
F[1]
>>> F.Eulerian(Integer(1), Integer(1))
0
```

antipode_on_basis(*compo*)

Return the antipode to a Fundamental quasi-symmetric basis element.

INPUT:

- `compo` – composition

OUTPUT:

- The result of the antipode applied to the quasi-symmetric Fundamental basis element indexed by `compo`.

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F.antipode_on_basis(Composition([2,1]))
-F[2, 1]
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(QQ).F()
>>> F.antipode_on_basis(Composition([Integer(2),Integer(1)]))
```

(continues on next page)

(continued from previous page)

```
-F[2, 1]
```

coproduct_on_basis(*compo*)

Return the coproduct to a Fundamental quasi-symmetric basis element.

Combinatorial rule: quasi-deconcatenation.

INPUT:

- compo – composition

OUTPUT:

- The application of the coproduct to the Fundamental quasi-symmetric function indexed by the composition compo.

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: F[4].coproduct()
F[] # F[4] + F[1] # F[3] + F[2] # F[2] + F[3] # F[1] + F[4] # F[]
sage: F[2,1,3].coproduct()
F[] # F[2, 1, 3] + F[1] # F[1, 1, 3] + F[2] # F[1, 3] + F[2, 1] # F[3] +
↪F[2, 1, 1] # F[2] + F[2, 1, 2] # F[1] + F[2, 1, 3] # F[]
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(QQ).Fundamental()
>>> F[Integer(4)].coproduct()
F[] # F[4] + F[1] # F[3] + F[2] # F[2] + F[3] # F[1] + F[4] # F[]
>>> F[Integer(2),Integer(1),Integer(3)].coproduct()
F[] # F[2, 1, 3] + F[1] # F[1, 1, 3] + F[2] # F[1, 3] + F[2, 1] # F[3] +
↪F[2, 1, 1] # F[2] + F[2, 1, 2] # F[1] + F[2, 1, 3] # F[]
```

dual()

Return the dual basis to the Fundamental basis. This is the ribbon basis of the non-commutative symmetric functions.

OUTPUT: the ribbon basis of the non-commutative symmetric functions

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F.dual()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon
↪basis
```

```
>>> from sage.all import *
>>> F = QuasiSymmetricFunctions(QQ).F()
>>> F.dual()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon
↪basis
```

class HazewinkelLambda(*QSym*)

Bases: *CombinatorialFreeModule*, BindableClass

The Hazewinkel lambda basis of the quasi-symmetric functions.

This basis goes back to [Haz2004], albeit it is indexed in a different way here. It is a multiplicative basis in a weak sense of this word (the product of any two basis elements is a basis element, but of course not the one obtained by concatenating the indexing compositions).

In [Haz2004], Hazewinkel showed that the **k**-algebra QSym is a polynomial algebra. (The proof is correct but rests upon an unproven claim that the lexicographically largest term of the n-th shuffle power of a Lyndon word is the n-fold concatenation of this Lyndon word with itself, occurring $n!$ times in that shuffle power. But this can be deduced from Section 2 of [Rad1979]. See also Chapter 6 of [GriRei18], specifically Theorem 6.5.13, for a complete proof.) More precisely, he showed that QSym is generated, as a free commutative **k**-algebra, by the elements $\lambda^n(M_I)$, where n ranges over the positive integers, and I ranges over all compositions which are Lyndon words and whose entries have gcd 1. Here, λ^n denotes the n-th lambda operation as explained in `lambda_of_monomial()`.

Thus, products of these generators form a **k**-module basis of QSym. We index this basis by compositions here. More precisely, we define the Hazewinkel lambda basis $(\mathrm{HWL}_I)_I$ (with I ranging over all compositions) as follows:

Let I be a composition. Let $I = I_1 I_2 \ldots I_k$ be the Chen-Fox-Lyndon factorization of I (see `lyndon_factorization()`). For every $j \in \{1, 2, \ldots, k\}$, let g_j be the gcd of the entries of the Lyndon word I_j, and let J_j be the result of dividing the entries of I_j by this gcd. Then, HWL_I is defined to be $\prod_{j=1}^{k} \lambda^{g_j}(M_{J_j})$.

Todo

The conversion from the M basis to the HWL basis is currently implemented in the naive way (inverting the base-change matrix in the other direction). This matrix is not triangular (not even after any permutations of the bases), and there could very well be a faster method (the one given by Hazewinkel?).

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: HWL = QSym.HazewinkelLambda()
sage: M = QSym.M()
sage: M(HWL([2]))
M[1, 1]
sage: M(HWL([1,1]))
2*M[1, 1] + M[2]
sage: M(HWL([1,2]))
M[1, 2]
sage: M(HWL([2,1]))
3*M[1, 1, 1] + M[1, 2] + M[2, 1]
sage: M(HWL(Composition([])))
M[]
sage: HWL(M([1,1]))
HWL[2]
sage: HWL(M(Composition([2])))
HWL[1, 1] - 2*HWL[2]
sage: HWL(M([1]))
HWL[1]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> HWL = QSym.HazewinkelLambda()
>>> M = QSym.M()
>>> M(HWL([Integer(2)]))
M[1, 1]
```

(continues on next page)

(continued from previous page)

```
>>> M(HWL([Integer(1),Integer(1)]))
2*M[1, 1] + M[2]
>>> M(HWL([Integer(1),Integer(2)]))
M[1, 2]
>>> M(HWL([Integer(2),Integer(1)]))
3*M[1, 1, 1] + M[1, 2] + M[2, 1]
>>> M(HWL(Composition([])))
M[]
>>> HWL(M([Integer(1),Integer(1)]))
HWL[2]
>>> HWL(M(Composition([Integer(2)])))
HWL[1, 1] - 2*HWL[2]
>>> HWL(M([Integer(1)]))
HWL[1]
```

product_on_basis(I, J)

The product on Hazewinkel Lambda basis elements.

The product of the basis elements indexed by two compositions I and J is the basis element obtained by concatenating the Lyndon factorizations of the words I and J, then reordering the Lyndon factors in nonincreasing order, and finally concatenating them in this order (giving a new composition).

INPUT:

- `I`, `J` – compositions

OUTPUT:

- The product of the Hazewinkel Lambda quasi-symmetric functions indexed by `I` and `J`, expressed in the Hazewinkel Lambda basis.

EXAMPLES:

```
sage: HWL = QuasiSymmetricFunctions(QQ).HazewinkelLambda()
sage: c1 = Composition([1, 2, 1])
sage: c2 = Composition([2, 1, 3, 2])
sage: HWL.product_on_basis(c1, c2)
HWL[2, 1, 3, 2, 1, 2, 1]
sage: HWL.product_on_basis(c1, Composition([]))
HWL[1, 2, 1]
sage: HWL.product_on_basis(Composition([]), Composition([]))
HWL[]
```

```
>>> from sage.all import *
>>> HWL = QuasiSymmetricFunctions(QQ).HazewinkelLambda()
>>> c1 = Composition([Integer(1), Integer(2), Integer(1)])
>>> c2 = Composition([Integer(2), Integer(1), Integer(3), Integer(2)])
>>> HWL.product_on_basis(c1, c2)
HWL[2, 1, 3, 2, 1, 2, 1]
>>> HWL.product_on_basis(c1, Composition([]))
HWL[1, 2, 1]
>>> HWL.product_on_basis(Composition([]), Composition([]))
HWL[]
```

M

alias of `Monomial`

class Monomial(*QSym*)

Bases: *CombinatorialFreeModule*, BindableClass

The Hopf algebra of quasi-symmetric function in the Monomial basis.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: F = QSym.F()
sage: M(F[2,2])
M[1, 1, 1, 1] + M[1, 1, 2] + M[2, 1, 1] + M[2, 2]
sage: m = SymmetricFunctions(QQ).m()
sage: M(m[3,1,1])
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
sage: (1+M[1])^3
M[] + 3*M[1] + 6*M[1, 1] + 6*M[1, 1, 1] + 3*M[1, 2] + 3*M[2] + 3*M[2, 1] +
→M[3]
sage: M[1,2,1].coproduct()
M[] # M[1, 2, 1] + M[1] # M[2, 1] + M[1, 2] # M[1] + M[1, 2, 1] # M[]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.M()
>>> F = QSym.F()
>>> M(F[Integer(2),Integer(2)])
M[1, 1, 1, 1] + M[1, 1, 2] + M[2, 1, 1] + M[2, 2]
>>> m = SymmetricFunctions(QQ).m()
>>> M(m[Integer(3),Integer(1),Integer(1)])
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
>>> (Integer(1)+M[Integer(1)])**Integer(3)
M[] + 3*M[1] + 6*M[1, 1] + 6*M[1, 1, 1] + 3*M[1, 2] + 3*M[2] + 3*M[2, 1] +
→M[3]
>>> M[Integer(1),Integer(2),Integer(1)].coproduct()
M[] # M[1, 2, 1] + M[1] # M[2, 1] + M[1, 2] # M[1] + M[1, 2, 1] # M[]
```

The following is an alias for this basis:

```
sage: QSym.Monomial()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

```
>>> from sage.all import *
>>> QSym.Monomial()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

class Element

Bases: IndexedFreeModuleElement

Element methods of the `Monomial` basis of `QuasiSymmetricFunctions`.

expand(*n*, *alphabet='x'*)

Expand the quasi-symmetric function written in the monomial basis in n variables.

INPUT:

- `n` – integer
- `alphabet` – string (default: `'x'`)

OUTPUT: the quasi-symmetric function `self` expressed in the n variables described by `alphabet`

Todo

accept an *alphabet* as input

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: M[4,2].expand(3)
x0^4*x1^2 + x0^4*x2^2 + x1^4*x2^2
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> M[Integer(4),Integer(2)].expand(Integer(3))
x0^4*x1^2 + x0^4*x2^2 + x1^4*x2^2
```

One can use a different set of variables by using the optional argument `alphabet`:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: M[2,1,1].expand(4,alphabet='y')
y0^2*y1*y2 + y0^2*y1*y3 + y0^2*y2*y3 + y1^2*y2*y3
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> M[Integer(2),Integer(1),Integer(1)].expand(Integer(4),alphabet='y')
y0^2*y1*y2 + y0^2*y1*y3 + y0^2*y2*y3 + y1^2*y2*y3
```

is_symmetric()

Determine if a quasi-symmetric function, written in the Monomial basis, is symmetric.

This is being tested by looking at the expansion in the Monomial basis and checking if the coefficients are the same if the indexing compositions are permutations of each other.

OUTPUT:

- `True` if `self` is an element of the symmetric functions and `False` otherwise.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.Monomial()
sage: (M[3,2] + M[2,3] + M[4,1]).is_symmetric()
False
sage: (M[3,2] + M[2,3]).is_symmetric()
True
sage: (M[1,2,1] + M[1,1,2]).is_symmetric()
False
sage: (M[1,2,1] + M[1,1,2] + M[2,1,1]).is_symmetric()
True
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.Monomial()
>>> (M[Integer(3),Integer(2)] + M[Integer(2),Integer(3)] +
```

(continues on next page)

(continued from previous page)

```
↪M[Integer(4),Integer(1)]).is_symmetric()
False
>>> (M[Integer(3),Integer(2)] + M[Integer(2),Integer(3)]).is_
↪symmetric()
True
>>> (M[Integer(1),Integer(2),Integer(1)] + M[Integer(1),Integer(1),
↪Integer(2)]).is_symmetric()
False
>>> (M[Integer(1),Integer(2),Integer(1)] + M[Integer(1),Integer(1),
↪Integer(2)] + M[Integer(2),Integer(1),Integer(1)]).is_symmetric()
True
```

psi_involution()

Return the image of the quasisymmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the fundamental quasisymmetric function F_I to F_{I^c}, where I^c denotes the complement of the composition I. The map ψ is an involution and a graded Hopf algebra automorphism of $QSym$. Its restriction to the ring of symmetric functions coincides with the omega automorphism of the latter ring.

The involution ψ of $QSym$ is adjoint to the involution ψ of $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also

psi involution on QSym, *psi involution on NCSF*, *star involution on QSym*.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: M = QSym.M()
sage: M[3,2].psi_involution()
-M[3, 2] - M[5]
sage: M[3,1].psi_involution()
M[3, 1] + M[4]
sage: M[3,1,1].psi_involution()
M[3, 1, 1] + M[3, 2] + M[4, 1] + M[5]
sage: M[1,1,1].psi_involution()
M[1, 1, 1] + M[1, 2] + M[2, 1] + M[3]
sage: M[[]].psi_involution()
M[]
sage: M(0).psi_involution()
0
sage: (2*M[[]] - M[3,1] + 4*M[2]).psi_involution()
2*M[] - 4*M[2] - M[3, 1] - M[4]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> M = QSym.M()
```

(continues on next page)

(continued from previous page)

```
>>> M[Integer(3),Integer(2)].psi_involution()
-M[3, 2] - M[5]
>>> M[Integer(3),Integer(1)].psi_involution()
M[3, 1] + M[4]
>>> M[Integer(3),Integer(1),Integer(1)].psi_involution()
M[3, 1, 1] + M[3, 2] + M[4, 1] + M[5]
>>> M[Integer(1),Integer(1),Integer(1)].psi_involution()
M[1, 1, 1] + M[1, 2] + M[2, 1] + M[3]
>>> M[[]].psi_involution()
M[]
>>> M(Integer(0)).psi_involution()
0
>>> (Integer(2)*M[[]] - M[Integer(3),Integer(1)] +␣
↪Integer(4)*M[Integer(2)]).psi_involution()
2*M[] - 4*M[2] - M[3, 1] - M[4]
```

This particular implementation is tailored to the monomial basis. It is semantically equivalent to the generic implementation it overshadows:

```
sage: F = QSym.F()
sage: all( F(M[I].psi_involution()) == F(M[I]).psi_involution()
....:      for I in Compositions(3) )
True

sage: F = QSym.F()
sage: all( F(M[I].psi_involution()) == F(M[I]).psi_involution()
....:      for I in Compositions(4) )
True
```

```
>>> from sage.all import *
>>> F = QSym.F()
>>> all( F(M[I].psi_involution()) == F(M[I]).psi_involution()
...      for I in Compositions(Integer(3)) )
True

>>> F = QSym.F()
>>> all( F(M[I].psi_involution()) == F(M[I]).psi_involution()
...      for I in Compositions(Integer(4)) )
True
```

to_symmetric_function()

Take a quasi-symmetric function, expressed in the monomial basis, and return its symmetric realization, when possible, expressed in the monomial basis of symmetric functions.

OUTPUT:

- If `self` is a symmetric function, then the expansion in the monomial basis of the symmetric functions is returned. Otherwise an error is raised.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.Monomial()
sage: (M[3,2] + M[2,3] + M[4,1]).to_symmetric_function()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: M[2, 3] + M[3, 2] + M[4, 1] is not a symmetric function
sage: (M[3,2] + M[2,3] + 2*M[4,1] + 2*M[1,4]).to_symmetric_function()
m[3, 2] + 2*m[4, 1]
sage: m = SymmetricFunctions(QQ).m()
sage: M(m[3,1,1]).to_symmetric_function()
m[3, 1, 1]
sage: (M(m[2,1])*M(m[2,1])).to_symmetric_function()-m[2,1]*m[2,1]
0
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> M = QSym.Monomial()
>>> (M[Integer(3),Integer(2)] + M[Integer(2),Integer(3)] +␣
↪M[Integer(4),Integer(1)]).to_symmetric_function()
Traceback (most recent call last):
...
ValueError: M[2, 3] + M[3, 2] + M[4, 1] is not a symmetric function
>>> (M[Integer(3),Integer(2)] + M[Integer(2),Integer(3)] +␣
↪Integer(2)*M[Integer(4),Integer(1)] + Integer(2)*M[Integer(1),
↪Integer(4)]).to_symmetric_function()
m[3, 2] + 2*m[4, 1]
>>> m = SymmetricFunctions(QQ).m()
>>> M(m[Integer(3),Integer(1),Integer(1)]).to_symmetric_function()
m[3, 1, 1]
>>> (M(m[Integer(2),Integer(1)])*M(m[Integer(2),Integer(1)])).to_
↪symmetric_function()-m[Integer(2),Integer(1)]*m[Integer(2),
↪Integer(1)]
0
```

antipode_on_basis(*compo*)

Return the result of the antipode applied to a quasi-symmetric Monomial basis element.

INPUT:

- `compo` – composition

OUTPUT:

- The result of the antipode applied to the composition `compo`, expressed in the Monomial basis.

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M.antipode_on_basis(Composition([2,1]))
M[1, 2] + M[3]
sage: M.antipode_on_basis(Composition([]))
M[]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(QQ).M()
>>> M.antipode_on_basis(Composition([Integer(2),Integer(1)]))
M[1, 2] + M[3]
>>> M.antipode_on_basis(Composition([]))
M[]
```

coproduct_on_basis(*compo*)

Return the coproduct of a Monomial basis element.

Combinatorial rule: deconcatenation.

INPUT:

- compo – composition

OUTPUT:

- The coproduct applied to the Monomial quasi-symmetric function indexed by compo, expressed in the Monomial basis.

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: M[4,2,3].coproduct()
M[] # M[4, 2, 3] + M[4] # M[2, 3] + M[4, 2] # M[3] + M[4, 2, 3] # M[]
sage: M.coproduct_on_basis(Composition([]))
M[] # M[]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> M[Integer(4),Integer(2),Integer(3)].coproduct()
M[] # M[4, 2, 3] + M[4] # M[2, 3] + M[4, 2] # M[3] + M[4, 2, 3] # M[]
>>> M.coproduct_on_basis(Composition([]))
M[] # M[]
```

dual()

Return the dual basis to the Monomial basis. This is the complete basis of the non-commutative symmetric functions.

OUTPUT: the complete basis of the non-commutative symmetric functions

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M.dual()
Non-Commutative Symmetric Functions over the Rational Field in the␣
↪Complete basis
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(QQ).M()
>>> M.dual()
Non-Commutative Symmetric Functions over the Rational Field in the␣
↪Complete basis
```

lambda_of_monomial(I, n)

Return the image of the monomial quasi-symmetric function M_I under the lambda-map λ^n, expanded in the monomial basis.

The ring of quasi-symmetric functions over the integers, $\mathrm{QSym}_{\mathbf{Z}}$ (and more generally, the ring of quasi-symmetric functions over any binomial ring) becomes a λ-ring (with the λ-structure inherited from the ring of formal power series, so that $\lambda^i(x_j)$ is x_j if $i = 1$ and 0 if $i > 1$).

The Adams operations of this λ-ring are the Adams endomorphisms $\mathbf{f}_n$ (see *adams_operator()* for their definition). Using these endomorphisms, the λ-operations can be explicitly computed via the formula

$$\exp\left(-\sum_{n=1}^{\infty}\frac{1}{n}\mathbf{f}_n(x)t^n\right)=\sum_{j=0}^{\infty}(-1)^j\lambda^j(x)t^j$$

in the ring of formal power series in a variable t over the ring of quasi-symmetric functions. In particular, every composition $I=(I_1,I_2,\cdots,I_\ell)$ satisfies

$$\exp\left(-\sum_{n=1}^{\infty}\frac{1}{n}M_{(nI_1,nI_2,\cdots,nI_\ell)}t^n\right)=\sum_{j=0}^{\infty}(-1)^j\lambda^j(M_I)t^j$$

(corrected version of Remark 2.4 in [Haz2004]).

The quasi-symmetric functions $\lambda^i(M_I)$ with n ranging over the positive integers and I ranging over the reduced Lyndon compositions (i. e., compositions which are Lyndon words and have the gcd of their entries equal to 1) form a set of free polynomial generators for QSym. See [GriRei18], Chapter 6, for the proof, and [Haz2004] for a major part of it.

INPUT:

- `I` – composition
- `n` – nonnegative integer

OUTPUT:

The quasi-symmetric function $\lambda^n(M_I)$, expanded in the monomial basis over the ground ring of `self`.

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(CyclotomicField()).Monomial()
sage: M.lambda_of_monomial([1, 2], 2)
2*M[1, 1, 2, 2] + M[1, 1, 4] + M[1, 2, 1, 2] + M[1, 3, 2] + M[2, 2, 2]
sage: M.lambda_of_monomial([1, 1], 2)
3*M[1, 1, 1, 1] + M[1, 1, 2] + M[1, 2, 1] + M[2, 1, 1]
sage: M = QuasiSymmetricFunctions(Integers(19)).Monomial()
sage: M.lambda_of_monomial([1, 2], 3)
6*M[1, 1, 1, 2, 2, 2] + 3*M[1, 1, 1, 2, 4] + 3*M[1, 1, 1, 4, 2]
+ M[1, 1, 1, 6] + 4*M[1, 1, 2, 1, 2, 2] + 2*M[1, 1, 2, 1, 4]
+ 2*M[1, 1, 2, 2, 1, 2] + 2*M[1, 1, 2, 3, 2] + 4*M[1, 1, 3, 2, 2]
+ 2*M[1, 1, 3, 4] + M[1, 1, 4, 1, 2] + M[1, 1, 5, 2]
+ 2*M[1, 2, 1, 1, 2, 2] + M[1, 2, 1, 1, 4] + M[1, 2, 1, 2, 1, 2]
+ M[1, 2, 1, 3, 2] + 4*M[1, 2, 2, 2, 2] + M[1, 2, 2, 4] + M[1, 2, 4, 2]
+ 2*M[1, 3, 1, 2, 2] + M[1, 3, 1, 4] + M[1, 3, 2, 1, 2] + M[1, 3, 3, 2]
+ M[1, 4, 2, 2] + 3*M[2, 1, 2, 2, 2] + M[2, 1, 2, 4] + M[2, 1, 4, 2]
+ 2*M[2, 2, 1, 2, 2] + M[2, 2, 1, 4] + M[2, 2, 2, 1, 2] + M[2, 2, 3, 2]
+ 2*M[2, 3, 2, 2] + M[2, 3, 4] + M[3, 2, 2, 2]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(CyclotomicField()).Monomial()
>>> M.lambda_of_monomial([Integer(1), Integer(2)], Integer(2))
2*M[1, 1, 2, 2] + M[1, 1, 4] + M[1, 2, 1, 2] + M[1, 3, 2] + M[2, 2, 2]
>>> M.lambda_of_monomial([Integer(1), Integer(1)], Integer(2))
3*M[1, 1, 1, 1] + M[1, 1, 2] + M[1, 2, 1] + M[2, 1, 1]
>>> M = QuasiSymmetricFunctions(Integers(Integer(19))).Monomial()
>>> M.lambda_of_monomial([Integer(1), Integer(2)], Integer(3))
6*M[1, 1, 1, 2, 2, 2] + 3*M[1, 1, 1, 2, 4] + 3*M[1, 1, 1, 4, 2]
+ M[1, 1, 1, 6] + 4*M[1, 1, 2, 1, 2, 2] + 2*M[1, 1, 2, 1, 4]
```

(continues on next page)

(continued from previous page)

```
+ 2*M[1, 1, 2, 2, 1, 2] + 2*M[1, 1, 2, 3, 2] + 4*M[1, 1, 3, 2, 2]
+ 2*M[1, 1, 3, 4] + M[1, 1, 4, 1, 2] + M[1, 1, 5, 2]
+ 2*M[1, 2, 1, 1, 2, 2] + M[1, 2, 1, 1, 4] + M[1, 2, 1, 2, 1, 2]
+ M[1, 2, 1, 3, 2] + 4*M[1, 2, 2, 2, 2] + M[1, 2, 2, 4] + M[1, 2, 4, 2]
+ 2*M[1, 3, 1, 2, 2] + M[1, 3, 1, 4] + M[1, 3, 2, 1, 2] + M[1, 3, 3, 2]
+ M[1, 4, 2, 2] + 3*M[2, 1, 2, 2, 2] + M[2, 1, 2, 4] + M[2, 1, 4, 2]
+ 2*M[2, 2, 1, 2, 2] + M[2, 2, 1, 4] + M[2, 2, 2, 1, 2] + M[2, 2, 3, 2]
+ 2*M[2, 3, 2, 2] + M[2, 3, 4] + M[3, 2, 2, 2]
```

The map λ^0 sends everything to 1:

```
sage: M = QuasiSymmetricFunctions(ZZ).Monomial()
sage: all( M.lambda_of_monomial(I, 0) == M.one()
....:      for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(ZZ).Monomial()
>>> all( M.lambda_of_monomial(I, Integer(0)) == M.one()
...      for I in Compositions(Integer(3)) )
True
```

The map λ^1 is the identity map:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: all( M.lambda_of_monomial(I, 1) == M(I)
....:      for I in Compositions(3) )
True
sage: M = QuasiSymmetricFunctions(Integers(5)).Monomial()
sage: all( M.lambda_of_monomial(I, 1) == M(I)
....:      for I in Compositions(3) )
True
sage: M = QuasiSymmetricFunctions(ZZ).Monomial()
sage: all( M.lambda_of_monomial(I, 1) == M(I)
....:      for I in Compositions(3) )
True
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> all( M.lambda_of_monomial(I, Integer(1)) == M(I)
...      for I in Compositions(Integer(3)) )
True
>>> M = QuasiSymmetricFunctions(Integers(Integer(5))).Monomial()
>>> all( M.lambda_of_monomial(I, Integer(1)) == M(I)
...      for I in Compositions(Integer(3)) )
True
>>> M = QuasiSymmetricFunctions(ZZ).Monomial()
>>> all( M.lambda_of_monomial(I, Integer(1)) == M(I)
...      for I in Compositions(Integer(3)) )
True
```

product_on_basis (I, J)

The product on Monomial basis elements.

The product of the basis elements indexed by two compositions I and J is the sum of the basis elements indexed by compositions in the stuffle product (also called the overlapping shuffle product) of I and J.

INPUT:

- `I`, `J` – compositions

OUTPUT:

- The product of the Monomial quasi-symmetric functions indexed by `I` and `J`, expressed in the Monomial basis.

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: c1 = Composition([2])
sage: c2 = Composition([1,3])
sage: M.product_on_basis(c1, c2)
M[1, 2, 3] + M[1, 3, 2] + M[1, 5] + M[2, 1, 3] + M[3, 3]
sage: M.product_on_basis(c1, Composition([]))
M[2]
```

```
>>> from sage.all import *
>>> M = QuasiSymmetricFunctions(QQ).Monomial()
>>> c1 = Composition([Integer(2)])
>>> c2 = Composition([Integer(1),Integer(3)])
>>> M.product_on_basis(c1, c2)
M[1, 2, 3] + M[1, 3, 2] + M[1, 5] + M[2, 1, 3] + M[3, 3]
>>> M.product_on_basis(c1, Composition([]))
M[2]
```

QS

alias of *Quasisymmetric_Schur*

class Quasisymmetric_Schur(*QSym*)

Bases: *CombinatorialFreeModule*, `BindableClass`

The Hopf algebra of quasi-symmetric function in the Quasisymmetric Schur basis.

The basis of Quasisymmetric Schur functions is defined in [QSCHUR] and in Definition 5.1.1 of [LMvW13]. Don't mistake them for the completely unrelated quasi-Schur functions of [NCSF1]!

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QS = QSym.QS()
sage: F = QSym.F()
sage: M = QSym.M()
sage: F(QS[1,2])
F[1, 2]
sage: M(QS[1,2])
M[1, 1, 1] + M[1, 2]
sage: s = SymmetricFunctions(QQ).s()
sage: QS(s[2,1,1])
QS[1, 1, 2] + QS[1, 2, 1] + QS[2, 1, 1]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> QS = QSym.QS()
```

(continues on next page)

(continued from previous page)

```
>>> F = QSym.F()
>>> M = QSym.M()
>>> F(QS[Integer(1),Integer(2)])
F[1, 2]
>>> M(QS[Integer(1),Integer(2)])
M[1, 1, 1] + M[1, 2]
>>> s = SymmetricFunctions(QQ).s()
>>> QS(s[Integer(2),Integer(1),Integer(1)])
QS[1, 1, 2] + QS[1, 2, 1] + QS[2, 1, 1]
```

dual()

The dual basis to the Quasisymmetric Schur basis.

The dual basis to the Quasisymmetric Schur basis is implemented as dual.

OUTPUT:

- the dual Quasisymmetric Schur basis of the non-commutative symmetric functions

EXAMPLES:

```
sage: QS = QuasiSymmetricFunctions(QQ).Quasisymmetric_Schur()
sage: QS.dual()
Non-Commutative Symmetric Functions over the Rational Field
 in the dual Quasisymmetric-Schur basis
```

```
>>> from sage.all import *
>>> QS = QuasiSymmetricFunctions(QQ).Quasisymmetric_Schur()
>>> QS.dual()
Non-Commutative Symmetric Functions over the Rational Field
 in the dual Quasisymmetric-Schur basis
```

YQS

alias of *Young_Quasisymmetric_Schur*

class Young_Quasisymmetric_Schur(*QSym*)

Bases: *CombinatorialFreeModule*, BindableClass

The Hopf algebra of quasi-symmetric functions in the Young Quasisymmetric Schur basis.

The basis of Young Quasisymmetric Schur functions is from Definition 5.2.1 of [LMvW13].

This basis is related to the Quasisymmetric Schur basis `QS` by `QS(alpha.reversed()) == YQS(alpha).star_involution()`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: YQS = QSym.YQS()
sage: F = QSym.F()
sage: QS = QSym.QS()
sage: F(YQS[1,2])
F[1, 2]
sage: all(QS(al.reversed())==YQS(al).star_involution() for al in
↪Compositions(5))
True
sage: s = SymmetricFunctions(QQ).s()
```

(continues on next page)

(continued from previous page)

```
sage: YQS(s[2,1,1])
YQS[1, 1, 2] + YQS[1, 2, 1] + YQS[2, 1, 1]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> YQS = QSym.YQS()
>>> F = QSym.F()
>>> QS = QSym.QS()
>>> F(YQS[Integer(1),Integer(2)])
F[1, 2]
>>> all(QS(al.reversed())==YQS(al).star_involution() for al in␣
↪Compositions(Integer(5)))
True
>>> s = SymmetricFunctions(QQ).s()
>>> YQS(s[Integer(2),Integer(1),Integer(1)])
YQS[1, 1, 2] + YQS[1, 2, 1] + YQS[2, 1, 1]
```

a_realization()

Return the realization of the Monomial basis of the ring of quasi-symmetric functions.

OUTPUT: the Monomial basis of quasi-symmetric functions

EXAMPLES:

```
sage: QuasiSymmetricFunctions(QQ).a_realization()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

```
>>> from sage.all import *
>>> QuasiSymmetricFunctions(QQ).a_realization()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

dI

alias of *dualImmaculate*

dual()

Return the dual Hopf algebra of the quasi-symmetric functions, which is the non-commutative symmetric functions.

OUTPUT: the non-commutative symmetric functions

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.dual()
Non-Commutative Symmetric Functions over the Rational Field
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> QSym.dual()
Non-Commutative Symmetric Functions over the Rational Field
```

class dualImmaculate(*QSym*)

Bases: *CombinatorialFreeModule*, BindableClass

The dual immaculate basis of the quasi-symmetric functions.

This basis first appears in [BBSSZ2012].

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: dI = QSym.dI()
sage: dI([1,3,2])*dI([1])  # long time (6s on sage.math, 2013)
dI[1, 1, 3, 2] + dI[2, 3, 2]
sage: dI([1,3])*dI([1,1])
dI[1, 1, 1, 3] + dI[1, 1, 4] + dI[1, 2, 3] - dI[1, 3, 2] - dI[1, 4, 1] - dI[1,
→ 5] + dI[2, 3, 1] + dI[2, 4]
sage: dI([3,1])*dI([2,1])  # long time (7s on sage.math, 2013)
dI[1, 1, 5] - dI[1, 4, 1, 1] - dI[1, 4, 2] - 2*dI[1, 5, 1] - dI[1, 6] - dI[2,␣
→4, 1] - dI[2, 5] - dI[3, 1, 3] + dI[3, 2, 1, 1] + dI[3, 2, 2] + dI[3, 3, 1]␣
→+ dI[4, 1, 1, 1] + 2*dI[4, 2, 1] + dI[4, 3] + dI[5, 1, 1] + dI[5, 2]
sage: F = QSym.F()
sage: dI(F[1,3,1])
-dI[1, 1, 1, 2] + dI[1, 1, 2, 1] - dI[1, 2, 2] + dI[1, 3, 1]
sage: F(dI(F([2,1,3])))
F[2, 1, 3]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> dI = QSym.dI()
>>> dI([Integer(1),Integer(3),Integer(2)])*dI([Integer(1)])  # long time (6s␣
→on sage.math, 2013)
dI[1, 1, 3, 2] + dI[2, 3, 2]
>>> dI([Integer(1),Integer(3)])*dI([Integer(1),Integer(1)])
dI[1, 1, 1, 3] + dI[1, 1, 4] + dI[1, 2, 3] - dI[1, 3, 2] - dI[1, 4, 1] - dI[1,
→ 5] + dI[2, 3, 1] + dI[2, 4]
>>> dI([Integer(3),Integer(1)])*dI([Integer(2),Integer(1)])  # long time (7s␣
→on sage.math, 2013)
dI[1, 1, 5] - dI[1, 4, 1, 1] - dI[1, 4, 2] - 2*dI[1, 5, 1] - dI[1, 6] - dI[2,␣
→4, 1] - dI[2, 5] - dI[3, 1, 3] + dI[3, 2, 1, 1] + dI[3, 2, 2] + dI[3, 3, 1]␣
→+ dI[4, 1, 1, 1] + 2*dI[4, 2, 1] + dI[4, 3] + dI[5, 1, 1] + dI[5, 2]
>>> F = QSym.F()
>>> dI(F[Integer(1),Integer(3),Integer(1)])
-dI[1, 1, 1, 2] + dI[1, 1, 2, 1] - dI[1, 2, 2] + dI[1, 3, 1]
>>> F(dI(F([Integer(2),Integer(1),Integer(3)])))
F[2, 1, 3]
```

from_polynomial(*f*, *check=True*)

Return the quasi-symmetric function in the Monomial basis corresponding to the quasi-symmetric polynomial `f`.

INPUT:

- `f` – a polynomial in finitely many variables over the same base ring as `self`. It is assumed that this polynomial is quasi-symmetric.
- `check` – boolean (default: `True`); checks whether the polynomial is indeed quasi-symmetric

OUTPUT: quasi-symmetric function in the Monomial basis

EXAMPLES:

```
sage: P = PolynomialRing(QQ, 'x', 3)
sage: x = P.gens()
sage: f = x[0] + x[1] + x[2]
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.from_polynomial(f)
M[1]
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ, 'x', Integer(3))
>>> x = P.gens()
>>> f = x[Integer(0)] + x[Integer(1)] + x[Integer(2)]
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> QSym.from_polynomial(f)
M[1]
```

Beware of setting `check=False`:

```
sage: f = x[0] + 2*x[1] + x[2]
sage: QSym.from_polynomial(f, check=True)
Traceback (most recent call last):
...
ValueError: x0 + 2*x1 + x2 is not a quasi-symmetric polynomial
sage: QSym.from_polynomial(f, check=False)
M[1]
```

```
>>> from sage.all import *
>>> f = x[Integer(0)] + Integer(2)*x[Integer(1)] + x[Integer(2)]
>>> QSym.from_polynomial(f, check=True)
Traceback (most recent call last):
...
ValueError: x0 + 2*x1 + x2 is not a quasi-symmetric polynomial
>>> QSym.from_polynomial(f, check=False)
M[1]
```

To expand the quasi-symmetric function in a basis other than the Monomial basis, the following shorthands are provided:

```
sage: M = QSym.Monomial()
sage: f = x[0]**2+x[1]**2+x[2]**2
sage: g = M.from_polynomial(f); g
M[2]
sage: F = QSym.Fundamental()
sage: F(g)
-F[1, 1] + F[2]
sage: F.from_polynomial(f)
-F[1, 1] + F[2]
```

```
>>> from sage.all import *
>>> M = QSym.Monomial()
>>> f =␣
↪x[Integer(0)]**Integer(2)+x[Integer(1)]**Integer(2)+x[Integer(2)]**Integer(2)
>>> g = M.from_polynomial(f); g
```

(continues on next page)

(continued from previous page)

```
M[2]
>>> F = QSym.Fundamental()
>>> F(g)
-F[1, 1] + F[2]
>>> F.from_polynomial(f)
-F[1, 1] + F[2]
```

class phi(*QSym*)

Bases: `CombinatorialFreeModule`, `BindableClass`

The Hopf algebra of quasi-symmetric functions in the ϕ basis.

The ϕ basis is defined as a rescaled Hopf dual of the Φ basis of the non-commutative symmetric functions (see Section 3.1 of [BDHMN2017]), where the pairing is

$$(\phi_I, \Phi_J) = z_I \delta_{I,J},$$

where $z_I = 1^{m_1} m_1! 2^{m_2} m_2! \cdots$ with m_i being the multiplicity of i in the composition I. Therefore, we call these the *quasi-symmetric power sums of the second kind*.

Using the duality, we can directly define the ϕ basis by

$$\phi_I = \sum_{J \succ I} z_I / sp_{I,J} M_J,$$

where $sp_{I,J}$ is as defined in [NCSF].

The ϕ-basis is well-defined only when the base ring is a **Q**-algebra.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: phi = QSym.phi(); phi
Quasisymmetric functions over the Rational Field in the phi basis
sage: phi.an_element()
2*phi[] + 2*phi[1] + 3*phi[1, 1]
sage: p = SymmetricFunctions(QQ).p()
sage: phi(p[2,2,1])
phi[1, 2, 2] + phi[2, 1, 2] + phi[2, 2, 1]
sage: all(sum(phi(list(al)) for al in Permutations(la))==phi(p(la)) for la in␣
↪Partitions(6))
True
sage: p = SymmetricFunctions(QQ).p()
sage: phi(p[3,2,2])
phi[2, 2, 3] + phi[2, 3, 2] + phi[3, 2, 2]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> phi = QSym.phi(); phi
Quasisymmetric functions over the Rational Field in the phi basis
>>> phi.an_element()
2*phi[] + 2*phi[1] + 3*phi[1, 1]
>>> p = SymmetricFunctions(QQ).p()
>>> phi(p[Integer(2),Integer(2),Integer(1)])
phi[1, 2, 2] + phi[2, 1, 2] + phi[2, 2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> all(sum(phi(list(al)) for al in Permutations(la))==phi(p(la)) for la in␣
↪Partitions(Integer(6)))
True
>>> p = SymmetricFunctions(QQ).p()
>>> phi(p[Integer(3),Integer(2),Integer(2)])
phi[2, 2, 3] + phi[2, 3, 2] + phi[3, 2, 2]
```

Checking the equivalent definition of ϕ_n:

```
sage: def test_phi(n):
....:     phi = QuasiSymmetricFunctions(QQ).phi()
....:     Phi = NonCommutativeSymmetricFunctions(QQ).Phi()
....:     M = matrix([[phi[I].duality_pairing(Phi[J])
....:                  for I in Compositions(n)]
....:                 for J in Compositions(n)])
....:     def z(J): return J.to_partition().centralizer_size()
....:     return M == matrix.diagonal([z(I) for I in Compositions(n)])
sage: all(test_phi(k) for k in range(1,5))
True
```

```
>>> from sage.all import *
>>> def test_phi(n):
...     phi = QuasiSymmetricFunctions(QQ).phi()
...     Phi = NonCommutativeSymmetricFunctions(QQ).Phi()
...     M = matrix([[phi[I].duality_pairing(Phi[J])
...                  for I in Compositions(n)]
...                 for J in Compositions(n)])
...     def z(J): return J.to_partition().centralizer_size()
...     return M == matrix.diagonal([z(I) for I in Compositions(n)])
>>> all(test_phi(k) for k in range(Integer(1),Integer(5)))
True
```

class psi(*QSym*)

Bases: `CombinatorialFreeModule`, `BindableClass`

The Hopf algebra of quasi-symmetric functions in the ψ basis.

The ψ basis is defined as a rescaled Hopf dual of the Ψ basis of the non-commutative symmetric functions (see Section 3.1 of [BDHMN2017]), where the pairing is

$$(\psi_I, \Psi_J) = z_I \delta_{I,J},$$

where $z_I = 1^{m_1} m_1! 2^{m_2} m_2! \cdots$ with m_i being the multiplicity of i in the composition I. Therefore, we call these the *quasi-symmetric power sums of the first kind*.

Using the duality, we can directly define the ψ basis by

$$\psi_I = \sum_{J \succ I} z_I / \pi_{I,J} M_J,$$

where $\pi_{I,J}$ is as defined in [NCSF].

The ψ-basis is well-defined only when the base ring is a $\mathbf{Q}$-algebra.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: psi = QSym.psi(); psi
Quasisymmetric functions over the Rational Field in the psi basis
sage: psi.an_element()
2*psi[] + 2*psi[1] + 3*psi[1, 1]
sage: p = SymmetricFunctions(QQ).p()
sage: psi(p[2,2,1])
psi[1, 2, 2] + psi[2, 1, 2] + psi[2, 2, 1]
sage: all(sum(psi(list(al)) for al in Permutations(la))==psi(p(la)) for la in␣
↪Partitions(6))
True
sage: p = SymmetricFunctions(QQ).p()
sage: psi(p[3,2,2])
psi[2, 2, 3] + psi[2, 3, 2] + psi[3, 2, 2]
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> psi = QSym.psi(); psi
Quasisymmetric functions over the Rational Field in the psi basis
>>> psi.an_element()
2*psi[] + 2*psi[1] + 3*psi[1, 1]
>>> p = SymmetricFunctions(QQ).p()
>>> psi(p[Integer(2),Integer(2),Integer(1)])
psi[1, 2, 2] + psi[2, 1, 2] + psi[2, 2, 1]
>>> all(sum(psi(list(al)) for al in Permutations(la))==psi(p(la)) for la in␣
↪Partitions(Integer(6)))
True
>>> p = SymmetricFunctions(QQ).p()
>>> psi(p[Integer(3),Integer(2),Integer(2)])
psi[2, 2, 3] + psi[2, 3, 2] + psi[3, 2, 2]
```

Checking the equivalent definition of ψ_n:

```
sage: def test_psi(n):
....:     psi = QuasiSymmetricFunctions(QQ).psi()
....:     Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
....:     M = matrix([[psi[I].duality_pairing(Psi[J])
....:                  for I in Compositions(n)]
....:                 for J in Compositions(n)])
....:     def z(J): return J.to_partition().centralizer_size()
....:     return M == matrix.diagonal([z(I) for I in Compositions(n)])
sage: all(test_psi(k) for k in range(1,5))
True
```

```
>>> from sage.all import *
>>> def test_psi(n):
...     psi = QuasiSymmetricFunctions(QQ).psi()
...     Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
...     M = matrix([[psi[I].duality_pairing(Psi[J])
...                  for I in Compositions(n)]
...                 for J in Compositions(n)])
...     def z(J): return J.to_partition().centralizer_size()
```

(continues on next page)

(continued from previous page)

```
...     return M == matrix.diagonal([z(I) for I in Compositions(n)])
>>> all(test_psi(k) for k in range(Integer(1),Integer(5)))
True
```

5.1.146 Introduction to Quasisymmetric Functions

In this document we briefly explain the quasisymmetric function bases and related functionality in Sage. We assume the reader is familiar with the package `SymmetricFunctions`.

Quasisymmetric functions, denoted $QSym$, form a subring of the power series ring in countably many variables. $QSym$ contains the symmetric functions. These functions first arose in the theory of P-partitions. The initial ideas in this field are attributed to MacMahon, Knuth, Kreweras, Glânffrwd Thomas, Stanley. In 1984, Gessel formalized the study of quasisymmetric functions and introduced the basis of fundamental quasisymmetric functions [Ges]. In 1995, Gelfand, Krob, Lascoux, Leclerc, Retakh, and Thibon showed that the ring of quasisymmetric functions is Hopf dual to the non-commutative symmetric functions [NCSF]. Many results have built on these.

One advantage of working in $QSym$ is that many interesting families of symmetric functions have explicit expansions in fundamental quasisymmetric functions such as Schur functions [Ges], Macdonald polynomials [HHL05], and plethysm of Schur functions [LW12].

For more background see Wikipedia article Quasisymmetric_function.

To begin, initialize the ring. Below we chose to use the rational numbers $\mathbf{Q}$. Other options include the integers $\mathbf{Z}$ and $\mathbf{C}$:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym
Quasisymmetric functions over the Rational Field

sage: QSym = QuasiSymmetricFunctions(CC); QSym
Quasisymmetric functions over the Complex Field with 53 bits of precision

sage: QSym = QuasiSymmetricFunctions(ZZ); QSym
Quasisymmetric functions over the Integer Ring
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> QSym
Quasisymmetric functions over the Rational Field

>>> QSym = QuasiSymmetricFunctions(CC); QSym
Quasisymmetric functions over the Complex Field with 53 bits of precision

>>> QSym = QuasiSymmetricFunctions(ZZ); QSym
Quasisymmetric functions over the Integer Ring
```

All bases of $QSym$ are indexed by compositions e.g. $[3, 1, 1, 4]$. The convention is to use capital letters for bases of $QSym$ and lowercase letters for bases of the symmetric functions Sym. Next set up names for the known bases by running `inject_shorthands()`. As with symmetric functions, you do not need to run this command and you could assign these bases other names.

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.inject_shorthands()
Defining M as shorthand for Quasisymmetric functions over the Rational Field in the
↪Monomial basis
```

(continues on next page)

(continued from previous page)

```
Defining F as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Fundamental basis
Defining E as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Essential basis
Defining dI as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪dualImmaculate basis
Defining QS as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Quasisymmetric Schur basis
Defining YQS as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Young Quasisymmetric Schur basis
Defining phi as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪phi basis
Defining psi as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪psi basis
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(QQ)
>>> QSym.inject_shorthands()
Defining M as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Monomial basis
Defining F as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Fundamental basis
Defining E as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Essential basis
Defining dI as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪dualImmaculate basis
Defining QS as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Quasisymmetric Schur basis
Defining YQS as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪Young Quasisymmetric Schur basis
Defining phi as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪phi basis
Defining psi as shorthand for Quasisymmetric functions over the Rational Field in the␣
↪psi basis
```

Now one can start constructing quasisymmetric functions.

Note

It is best to use variables other than `M` and `F`.

```
sage: x = M[2,1] + M[1,2]
sage: x
M[1, 2] + M[2, 1]

sage: y = 3*M[1,2] + M[3]^2; y
3*M[1, 2] + 2*M[3, 3] + M[6]

sage: F[3,1,3] + 7*F[2,1]
7*F[2, 1] + F[3, 1, 3]
```

(continues on next page)

(continued from previous page)

```
sage: 3*F[2,1,2] + F[3]^2
F[1, 2, 2, 1] + F[1, 2, 3] + 2*F[1, 3, 2] + F[1, 4, 1] + F[1, 5] + 3*F[2, 1, 2]
 + 2*F[2, 2, 2] + 2*F[2, 3, 1] + 2*F[2, 4] + F[3, 2, 1] + 3*F[3, 3] + 2*F[4, 2] + F[5,
↪ 1] + F[6]
```

```
>>> from sage.all import *
>>> x = M[Integer(2),Integer(1)] + M[Integer(1),Integer(2)]
>>> x
M[1, 2] + M[2, 1]

>>> y = Integer(3)*M[Integer(1),Integer(2)] + M[Integer(3)]**Integer(2); y
3*M[1, 2] + 2*M[3, 3] + M[6]

>>> F[Integer(3),Integer(1),Integer(3)] + Integer(7)*F[Integer(2),Integer(1)]
7*F[2, 1] + F[3, 1, 3]

>>> Integer(3)*F[Integer(2),Integer(1),Integer(2)] + F[Integer(3)]**Integer(2)
F[1, 2, 2, 1] + F[1, 2, 3] + 2*F[1, 3, 2] + F[1, 4, 1] + F[1, 5] + 3*F[2, 1, 2]
 + 2*F[2, 2, 2] + 2*F[2, 3, 1] + 2*F[2, 4] + F[3, 2, 1] + 3*F[3, 3] + 2*F[4, 2] + F[5,
↪ 1] + F[6]
```

To convert from one basis to another is easy:

```
sage: z = M[1,2,1]
sage: z
M[1, 2, 1]

sage: F(z)
-F[1, 1, 1, 1] + F[1, 2, 1]

sage: M(F(z))
M[1, 2, 1]
```

```
>>> from sage.all import *
>>> z = M[Integer(1),Integer(2),Integer(1)]
>>> z
M[1, 2, 1]

>>> F(z)
-F[1, 1, 1, 1] + F[1, 2, 1]

>>> M(F(z))
M[1, 2, 1]
```

To expand in variables, one can specify a finite size alphabet $x_1, x_2, \ldots, x_m$:

```
sage: y = M[1,2,1]
sage: y.expand(4)
x0*x1^2*x2 + x0*x1^2*x3 + x0*x2^2*x3 + x1*x2^2*x3
```

```
>>> from sage.all import *
>>> y = M[Integer(1),Integer(2),Integer(1)]
```

(continues on next page)

(continued from previous page)

```
>>> y.expand(Integer(4))
x0*x1^2*x2 + x0*x1^2*x3 + x0*x2^2*x3 + x1*x2^2*x3
```

The usual methods on free modules are available such as coefficients, degrees, and the support:

```
sage: z = 3*M[1,2]+M[3]^2; z
3*M[1, 2] + 2*M[3, 3] + M[6]

sage: z.coefficient([1,2])
3

sage: z.degree()
6

sage: sorted(z.coefficients())
[1, 2, 3]

sage: sorted(z.monomials(), key=lambda x: tuple(x.support()))
[M[1, 2], M[3, 3], M[6]]

sage: z.monomial_coefficients()
{[1, 2]: 3, [3, 3]: 2, [6]: 1}
```

```
>>> from sage.all import *
>>> z = Integer(3)*M[Integer(1),Integer(2)]+M[Integer(3)]**Integer(2); z
3*M[1, 2] + 2*M[3, 3] + M[6]

>>> z.coefficient([Integer(1),Integer(2)])
3

>>> z.degree()
6

>>> sorted(z.coefficients())
[1, 2, 3]

>>> sorted(z.monomials(), key=lambda x: tuple(x.support()))
[M[1, 2], M[3, 3], M[6]]

>>> z.monomial_coefficients()
{[1, 2]: 3, [3, 3]: 2, [6]: 1}
```

As with the symmetric functions package, the quasisymmetric function `1` has several instantiations. However, the most obvious way to write `1` leads to an error (this is due to the semantics of python):

```
sage: M[[]]
M[]
sage: M.one()
M[]
sage: M(1)
M[]
sage: M[[]] == 1
```

(continues on next page)

(continued from previous page)

```
True
sage: M[]
Traceback (most recent call last):
...
SyntaxError: invalid ...
```

```
>>> from sage.all import *
>>> M[[]]
M[]
>>> M.one()
M[]
>>> M(Integer(1))
M[]
>>> M[[]] == Integer(1)
True
>>> M[]
Traceback (most recent call last):
...
SyntaxError: invalid ...
```

Working with symmetric functions

The quasisymmetric functions are a ring which contains the symmetric functions as a subring. The Monomial quasisymmetric functions are related to the monomial symmetric functions by $m_\lambda = \sum_{\mathrm{sort}(c)=\lambda} M_c$, where $\mathrm{sort}(c)$ means the partition obtained by sorting the composition c:

```
sage: SymmetricFunctions(QQ).inject_shorthands()
Defining e as shorthand for Symmetric Functions over Rational Field in the elementary␣
↪basis
Defining f as shorthand for Symmetric Functions over Rational Field in the forgotten␣
↪basis
Defining h as shorthand for Symmetric Functions over Rational Field in the␣
↪homogeneous basis
Defining m as shorthand for Symmetric Functions over Rational Field in the monomial␣
↪basis
Defining p as shorthand for Symmetric Functions over Rational Field in the powersum␣
↪basis
Defining s as shorthand for Symmetric Functions over Rational Field in the Schur basis

sage: m[2,1]
m[2, 1]
sage: M(m[2,1])
M[1, 2] + M[2, 1]
sage: M(s[2,1])
2*M[1, 1, 1] + M[1, 2] + M[2, 1]
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).inject_shorthands()
Defining e as shorthand for Symmetric Functions over Rational Field in the elementary␣
↪basis
Defining f as shorthand for Symmetric Functions over Rational Field in the forgotten␣
```

(continues on next page)

(continued from previous page)

```
↪basis
Defining h as shorthand for Symmetric Functions over Rational Field in the␣
↪homogeneous basis
Defining m as shorthand for Symmetric Functions over Rational Field in the monomial␣
↪basis
Defining p as shorthand for Symmetric Functions over Rational Field in the powersum␣
↪basis
Defining s as shorthand for Symmetric Functions over Rational Field in the Schur basis

>>> m[Integer(2),Integer(1)]
m[2, 1]
>>> M(m[Integer(2),Integer(1)])
M[1, 2] + M[2, 1]
>>> M(s[Integer(2),Integer(1)])
2*M[1, 1, 1] + M[1, 2] + M[2, 1]
```

There are methods to test if an expression f in the quasisymmetric functions is a symmetric function:

```
sage: f = M[1,1,2] + M[1,2,1]
sage: f.is_symmetric()
False
sage: f = M[3,1] + M[1,3]
sage: f.is_symmetric()
True
```

```
>>> from sage.all import *
>>> f = M[Integer(1),Integer(1),Integer(2)] + M[Integer(1),Integer(2),Integer(1)]
>>> f.is_symmetric()
False
>>> f = M[Integer(3),Integer(1)] + M[Integer(1),Integer(3)]
>>> f.is_symmetric()
True
```

If f is symmetric, there are methods to convert f to an expression in the symmetric functions:

```
sage: f.to_symmetric_function()
m[3, 1]
```

```
>>> from sage.all import *
>>> f.to_symmetric_function()
m[3, 1]
```

The expansion of the Schur function in terms of the Fundamental quasisymmetric functions is due to [Ges]. There is one term in the expansion for each standard tableau of shape equal to the partition indexing the Schur function.

```
sage: f = F[3,2] + F[2,2,1] + F[2,3] + F[1,3,1] + F[1,2,2]
sage: f.is_symmetric()
True
sage: f.to_symmetric_function()
5*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + 2*m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
sage: s(f.to_symmetric_function())
s[3, 2]
```

```
>>> from sage.all import *
>>> f = F[Integer(3),Integer(2)] + F[Integer(2),Integer(2),Integer(1)] + F[Integer(2),
↪Integer(3)] + F[Integer(1),Integer(3),Integer(1)] + F[Integer(1),Integer(2),
↪Integer(2)]
>>> f.is_symmetric()
True
>>> f.to_symmetric_function()
5*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + 2*m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
>>> s(f.to_symmetric_function())
s[3, 2]
```

It is also possible to convert any symmetric function to the quasisymmetric function expansion in any known basis. The converse is not true:

```
sage: M( m[3,1,1] )
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
sage: F( s[2,2,1] )
F[1, 1, 2, 1] + F[1, 2, 1, 1] + F[1, 2, 2] + F[2, 1, 2] + F[2, 2, 1]

sage: s(M[2,1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= M[2, 1]) an element of self
```

```
>>> from sage.all import *
>>> M( m[Integer(3),Integer(1),Integer(1)] )
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
>>> F( s[Integer(2),Integer(2),Integer(1)] )
F[1, 1, 2, 1] + F[1, 2, 1, 1] + F[1, 2, 2] + F[2, 1, 2] + F[2, 2, 1]

>>> s(M[Integer(2),Integer(1)])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= M[2, 1]) an element of self
```

It is possible to experiment with the quasisymmetric function expansion of other bases, but it is important that the base ring be the same for both algebras.

```
sage: R = QQ['t']
sage: Qp = SymmetricFunctions(R).hall_littlewood().Qp()
sage: QSymt = QuasiSymmetricFunctions(R)
sage: Ft = QSymt.F()
sage: Ft( Qp[2,2] )
F[1, 2, 1] + t*F[1, 3] + (t+1)*F[2, 2] + t*F[3, 1] + t^2*F[4]
```

```
>>> from sage.all import *
>>> R = QQ['t']
>>> Qp = SymmetricFunctions(R).hall_littlewood().Qp()
>>> QSymt = QuasiSymmetricFunctions(R)
>>> Ft = QSymt.F()
>>> Ft( Qp[Integer(2),Integer(2)] )
F[1, 2, 1] + t*F[1, 3] + (t+1)*F[2, 2] + t*F[3, 1] + t^2*F[4]
```

```
sage: K = QQ['q','t'].fraction_field()
sage: Ht = SymmetricFunctions(K).macdonald().Ht()
sage: Fqt = QuasiSymmetricFunctions(Ht.base_ring()).F()
sage: Fqt(Ht[2,1])
q*t*F[1, 1, 1] + (q+t)*F[1, 2] + (q+t)*F[2, 1] + F[3]
```

```
>>> from sage.all import *
>>> K = QQ['q','t'].fraction_field()
>>> Ht = SymmetricFunctions(K).macdonald().Ht()
>>> Fqt = QuasiSymmetricFunctions(Ht.base_ring()).F()
>>> Fqt(Ht[Integer(2),Integer(1)])
q*t*F[1, 1, 1] + (q+t)*F[1, 2] + (q+t)*F[2, 1] + F[3]
```

The following will raise an error because the base ring of `F` is not equal to the base ring of `Ht`:

```
sage: F(Ht[2,1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= McdHt[2, 1]) an element of self␣
↪(=Quasisymmetric functions over the Rational Field in the Fundamental basis)
```

```
>>> from sage.all import *
>>> F(Ht[Integer(2),Integer(1)])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= McdHt[2, 1]) an element of self␣
↪(=Quasisymmetric functions over the Rational Field in the Fundamental basis)
```

QSym is a Hopf algebra

The product on $QSym$ is commutative and is inherited from the product by the realization within the polynomial ring:

```
sage: M[3]*M[1,1] == M[1,1]*M[3]
True
sage: M[3]*M[1,1]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[3, 1, 1] + M[4, 1]
sage: F[3]*F[1,1]
F[1, 1, 3] + F[1, 2, 2] + F[1, 3, 1] + F[1, 4] + F[2, 1, 2] + F[2, 2, 1] + F[2, 3] +␣
↪F[3, 1, 1] + F[3, 2] + F[4, 1]
sage: M[3]*F[2]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[2, 3] + M[3, 1, 1] + M[3, 2] + M[4, 1] + M[5]
sage: F[2]*M[3]
F[1, 1, 1, 2] - F[1, 2, 2] + F[2, 1, 1, 1] - F[2, 1, 2] - F[2, 2, 1] + F[5]
```

```
>>> from sage.all import *
>>> M[Integer(3)]*M[Integer(1),Integer(1)] == M[Integer(1),Integer(1)]*M[Integer(3)]
True
>>> M[Integer(3)]*M[Integer(1),Integer(1)]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[3, 1, 1] + M[4, 1]
>>> F[Integer(3)]*F[Integer(1),Integer(1)]
F[1, 1, 3] + F[1, 2, 2] + F[1, 3, 1] + F[1, 4] + F[2, 1, 2] + F[2, 2, 1] + F[2, 3] +␣
↪F[3, 1, 1] + F[3, 2] + F[4, 1]
```

(continues on next page)

(continued from previous page)

```
>>> M[Integer(3)]*F[Integer(2)]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[2, 3] + M[3, 1, 1] + M[3, 2] + M[4, 1] + M[5]
>>> F[Integer(2)]*M[Integer(3)]
F[1, 1, 1, 2] - F[1, 2, 2] + F[2, 1, 1, 1] - F[2, 1, 2] - F[2, 2, 1] + F[5]
```

There is a coproduct on this ring as well, which in the Monomial basis acts by cutting the composition into a left half and a right half. The co-product is non-co-commutative:

```
sage: M[1,3,1].coproduct()
M[] # M[1, 3, 1] + M[1] # M[3, 1] + M[1, 3] # M[1] + M[1, 3, 1] # M[]
sage: F[1,3,1].coproduct()
F[] # F[1, 3, 1] + F[1] # F[3, 1] + F[1, 1] # F[2, 1] + F[1, 2] # F[1, 1] + F[1, 3] #
↪F[1] + F[1, 3, 1] # F[]
```

```
>>> from sage.all import *
>>> M[Integer(1),Integer(3),Integer(1)].coproduct()
M[] # M[1, 3, 1] + M[1] # M[3, 1] + M[1, 3] # M[1] + M[1, 3, 1] # M[]
>>> F[Integer(1),Integer(3),Integer(1)].coproduct()
F[] # F[1, 3, 1] + F[1] # F[3, 1] + F[1, 1] # F[2, 1] + F[1, 2] # F[1, 1] + F[1, 3] #
↪F[1] + F[1, 3, 1] # F[]
```

The Duality Pairing with Non-Commutative Symmetric Functions

These two operations endow $QSym$ with the structure of a Hopf algebra. It is the dual Hopf algebra of the non-commutative symmetric functions $NCSF$. Under this duality, the Monomial basis of $QSym$ is dual to the Complete basis of $NCSF$, and the Fundamental basis of $QSym$ is dual to the Ribbon basis of $NCSF$ (see [MR]):

```
sage: S = M.dual(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
sage: M[1,3,1].duality_pairing( S[1,3,1] )
1
sage: M.duality_pairing_matrix( S, degree=4 )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
sage: F.duality_pairing_matrix( S, degree=4 )
[1 0 0 0 0 0 0 0]
[1 1 0 0 0 0 0 0]
[1 0 1 0 0 0 0 0]
[1 1 1 1 0 0 0 0]
[1 0 0 0 1 0 0 0]
[1 1 0 0 1 1 0 0]
[1 0 1 0 1 0 1 0]
[1 1 1 1 1 1 1 1]
sage: NCSF = M.realization_of().dual()
sage: R = NCSF.Ribbon()
sage: F.duality_pairing_matrix( R, degree=4 )
```

(continues on next page)

(continued from previous page)

```
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
sage: M.duality_pairing_matrix( R, degree=4 )
[ 1  0  0  0  0  0  0  0]
[-1  1  0  0  0  0  0  0]
[-1  0  1  0  0  0  0  0]
[ 1 -1 -1  1  0  0  0  0]
[-1  0  0  0  1  0  0  0]
[ 1 -1  0  0 -1  1  0  0]
[ 1  0 -1  0 -1  0  1  0]
[-1  1  1 -1  1 -1 -1  1]
```

```
>>> from sage.all import *
>>> S = M.dual(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
>>> M[Integer(1),Integer(3),Integer(1)].duality_pairing( S[Integer(1),Integer(3),
↪Integer(1)] )
1
>>> M.duality_pairing_matrix( S, degree=Integer(4) )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
>>> F.duality_pairing_matrix( S, degree=Integer(4) )
[1 0 0 0 0 0 0 0]
[1 1 0 0 0 0 0 0]
[1 0 1 0 0 0 0 0]
[1 1 1 1 0 0 0 0]
[1 0 0 0 1 0 0 0]
[1 1 0 0 1 1 0 0]
[1 0 1 0 1 0 1 0]
[1 1 1 1 1 1 1 1]
>>> NCSF = M.realization_of().dual()
>>> R = NCSF.Ribbon()
>>> F.duality_pairing_matrix( R, degree=Integer(4) )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
```

(continues on next page)

(continued from previous page)

```
[0  0  0  0  0  0  0  1]
>>> M.duality_pairing_matrix( R, degree=Integer(4) )
[ 1   0   0   0   0   0   0   0]
[-1   1   0   0   0   0   0   0]
[-1   0   1   0   0   0   0   0]
[ 1  -1  -1   1   0   0   0   0]
[-1   0   0   0   1   0   0   0]
[ 1  -1   0   0  -1   1   0   0]
[ 1   0  -1   0  -1   0   1   0]
[-1   1   1  -1   1  -1  -1   1]
```

Let H and G be elements of $QSym$ and h an element of $NCSF$. Then if we represent the duality pairing with the mathematical notation $[\cdot,\cdot]$, we have:

$$[H\cdot G, h] = [H\otimes G, \Delta(h)].$$

For example, the coefficient of `M[2,1,4,1]` in `M[1,3]*M[2,1,1]` may be computed with the duality pairing:

```
sage: I, J = Composition([1,3]), Composition([2,1,1])
sage: (M[I]*M[J]).duality_pairing(S[2,1,4,1])
1
```

```
>>> from sage.all import *
>>> I, J = Composition([Integer(1),Integer(3)]), Composition([Integer(2),Integer(1),
↪Integer(1)])
>>> (M[I]*M[J]).duality_pairing(S[Integer(2),Integer(1),Integer(4),Integer(1)])
1
```

And the coefficient of `S[1,3] # S[2,1,1]` in `S[2,1,4,1].coproduct()` is equal to this result:

```
sage: S[2,1,4,1].coproduct()
S[] # S[2, 1, 4, 1] + ... + S[1, 3] # S[2, 1, 1] + ... + S[4, 1] # S[2, 1]
```

```
>>> from sage.all import *
>>> S[Integer(2),Integer(1),Integer(4),Integer(1)].coproduct()
S[] # S[2, 1, 4, 1] + ... + S[1, 3] # S[2, 1, 1] + ... + S[4, 1] # S[2, 1]
```

The duality pairing on the tensor space is another way of getting this coefficient, but currently the method *duality_pairing()* is not defined on the tensor squared space. However, we can extend this functionality by applying a linear morphism to the terms in the coproduct, as follows:

```
sage: X = S[2,1,4,1].coproduct()
sage: def linear_morphism(x, y):
....:     return x.duality_pairing(M[1,3]) * y.duality_pairing(M[2,1,1])
sage: X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

```
>>> from sage.all import *
>>> X = S[Integer(2),Integer(1),Integer(4),Integer(1)].coproduct()
>>> def linear_morphism(x, y):
...     return x.duality_pairing(M[Integer(1),Integer(3)]) * y.duality_
↪pairing(M[Integer(2),Integer(1),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

Similarly, if H is an element of $QSym$ and g and h are elements of $NCSF$, then

$$[H, g \cdot h] = [\Delta(H), g \otimes h].$$

For example, the coefficient of `R[2,3,1]` in `R[2,1]*R[2,1]` is computed with the duality pairing by the following command:

```
sage: (R[2,1]*R[2,1]).duality_pairing(F[2,3,1])
1
sage: R[2,1]*R[2,1]
R[2, 1, 2, 1] + R[2, 3, 1]
```

```
>>> from sage.all import *
>>> (R[Integer(2),Integer(1)]*R[Integer(2),Integer(1)]).duality_pairing(F[Integer(2),
↪Integer(3),Integer(1)])
1
>>> R[Integer(2),Integer(1)]*R[Integer(2),Integer(1)]
R[2, 1, 2, 1] + R[2, 3, 1]
```

This coefficient should then be equal to the coefficient of `F[2,1] # F[2,1]` in `F[2,3,1].coproduct()`:

```
sage: F[2,3,1].coproduct()
F[] # F[2, 3, 1] + ... + F[2, 1] # F[2, 1]  + ... + F[2, 3, 1] # F[]
```

```
>>> from sage.all import *
>>> F[Integer(2),Integer(3),Integer(1)].coproduct()
F[] # F[2, 3, 1] + ... + F[2, 1] # F[2, 1]  + ... + F[2, 3, 1] # F[]
```

This can also be computed by the duality pairing on the tensor space, as above:

```
sage: X = F[2,3,1].coproduct()
sage: def linear_morphism(x, y):
....:     return x.duality_pairing(R[2,1]) * y.duality_pairing(R[2,1])
sage: X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

```
>>> from sage.all import *
>>> X = F[Integer(2),Integer(3),Integer(1)].coproduct()
>>> def linear_morphism(x, y):
...     return x.duality_pairing(R[Integer(2),Integer(1)]) * y.duality_
↪pairing(R[Integer(2),Integer(1)])
>>> X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

The Operation Adjoint to Multiplication by a Non-Commutative Symmetric Function

Let $g \in NCSF$ and consider the linear endomorphism of $NCSF$ defined by left (respectively, right) multiplication by g. Since there is a duality between $QSym$ and $NCSF$, this linear transformation induces an operator $g^{\perp}$ on $QSym$ satisfying

$$[g^{\perp}(H), h] = [H, g \cdot h].$$

for any non-commutative symmetric function h.

This is implemented by the method `skew_by()`. Explicitly, if `H` is a quasisymmetric function and `g` a non-commutative symmetric function, then `H.skew_by(g)` and `H.skew_by(g, side='right')` are expressions that satisfy, for any non-commutative symmetric function `h`, the following identities:

```
H.skew_by(g).duality_pairing(h) == H.duality_pairing(g*h)
H.skew_by(g, side='right').duality_pairing(h) == H.duality_pairing(h*g)
```

For example, `M[J].skew_by(S[I])` is 0 unless the composition J begins with I and `M(J).skew_by(S(I), side='right')` is 0 unless the composition J ends with I:

```
sage: M[3,2,2].skew_by(S[3])
M[2, 2]
sage: M[3,2,2].skew_by(S[2])
0
sage: M[3,2,2].coproduct().apply_multilinear_morphism( lambda x,y: x.duality_
↪pairing(S[3])*y )
M[2, 2]
sage: M[3,2,2].skew_by(S[3], side='right')
0
sage: M[3,2,2].skew_by(S[2], side='right')
M[3, 2]
```

```
>>> from sage.all import *
>>> M[Integer(3),Integer(2),Integer(2)].skew_by(S[Integer(3)])
M[2, 2]
>>> M[Integer(3),Integer(2),Integer(2)].skew_by(S[Integer(2)])
0
>>> M[Integer(3),Integer(2),Integer(2)].coproduct().apply_multilinear_morphism(␣
↪lambda x,y: x.duality_pairing(S[Integer(3)])*y )
M[2, 2]
>>> M[Integer(3),Integer(2),Integer(2)].skew_by(S[Integer(3)], side='right')
0
>>> M[Integer(3),Integer(2),Integer(2)].skew_by(S[Integer(2)], side='right')
M[3, 2]
```

The antipode

The antipode sends the Fundamental basis element indexed by the composition I to -1 to the size of I times the Fundamental basis element indexed by the conjugate composition to I:

```
sage: F[3,2,2].antipode()
-F[1, 2, 2, 1, 1]
sage: Composition([3,2,2]).conjugate()
[1, 2, 2, 1, 1]
sage: M[3,2,2].antipode()
-M[2, 2, 3] - M[2, 5] - M[4, 3] - M[7]
```

```
>>> from sage.all import *
>>> F[Integer(3),Integer(2),Integer(2)].antipode()
-F[1, 2, 2, 1, 1]
>>> Composition([Integer(3),Integer(2),Integer(2)]).conjugate()
```

(continues on next page)

(continued from previous page)

```
[1, 2, 2, 1, 1]
>>> M[Integer(3),Integer(2),Integer(2)].antipode()
-M[2, 2, 3] - M[2, 5] - M[4, 3] - M[7]
```

We demonstrate here the defining relation of the antipode:

```
sage: X = F[3,2,2].coproduct()
sage: X.apply_multilinear_morphism(lambda x,y: x*y.antipode())
0
sage: X.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

```
>>> from sage.all import *
>>> X = F[Integer(3),Integer(2),Integer(2)].coproduct()
>>> X.apply_multilinear_morphism(lambda x,y: x*y.antipode())
0
>>> X.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

REFERENCES:

5.1.147 Symmetric functions in non-commuting variables

- *Introduction to Symmetric Functions in Non-Commuting Variables*
- *Bases for NCSym*
- *Dual Symmetric Functions in Non-Commuting Variables*
- *Symmetric Functions in Non-Commuting Variables*

5.1.148 Bases for $NCSym$

AUTHORS:

- Travis Scrimshaw (08-04-2013): Initial version

class sage.combinat.ncsym.bases.**MultiplicativeNCSymBases**(*parent_with_realization*)

Bases: Category_realization_of_parent

Category of multiplicative bases of symmetric functions in non-commuting variables.

A multiplicative basis is one for which $\mathbf{b}_A \mathbf{b}_B = \mathbf{b}_{A|B}$ where $A|B$ is the pipe() operation on set partitions.

EXAMPLES:

```
sage: from sage.combinat.ncsym.bases import MultiplicativeNCSymBases
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: MultiplicativeNCSymBases(NCSym)
Category of multiplicative bases of symmetric functions in non-commuting␣
↪variables over the Rational Field
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsym.bases import MultiplicativeNCSymBases
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> MultiplicativeNCSymBases(NCSym)
```

(continues on next page)

(continued from previous page)

```
Category of multiplicative bases of symmetric functions in non-commuting␣
↪variables over the Rational Field
```

class ElementMethods

Bases: `object`

class ParentMethods

Bases: `object`

product_on_basis(A, B)

The product on basis elements.

The product on a multiplicative basis is given by $\mathbf{b}_A \cdot \mathbf{b}_B = \mathbf{b}_{A|B}$.

The bases $\{\mathbf{e}, \mathbf{h}, \mathbf{x}, \mathbf{cp}, \mathbf{p}, \mathbf{chi}, \mathbf{rho}\}$ are all multiplicative.

INPUT:

- `A`, `B` – set partitions

OUTPUT: an element in the basis `self`

EXAMPLES:

```
sage: e = SymmetricFunctionsNonCommutingVariables(QQ).e()
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: x = SymmetricFunctionsNonCommutingVariables(QQ).x()
sage: cp = SymmetricFunctionsNonCommutingVariables(QQ).cp()
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).p()
sage: chi = SymmetricFunctionsNonCommutingVariables(QQ).chi()
sage: rho = SymmetricFunctionsNonCommutingVariables(QQ).rho()
sage: A = SetPartition([[1], [2, 3]])
sage: B = SetPartition([[1], [3], [2,4]])
sage: e.product_on_basis(A, B)
e{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: h.product_on_basis(A, B)
h{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: x.product_on_basis(A, B)
x{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: cp.product_on_basis(A, B)
cp{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: p.product_on_basis(A, B)
p{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: chi.product_on_basis(A, B)
chi{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: rho.product_on_basis(A, B)
rho{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: e.product_on_basis(A,B)==e(h(e(A))*h(e(B)))
True
sage: h.product_on_basis(A,B)==h(x(h(A))*x(h(B)))
True
sage: x.product_on_basis(A,B)==x(h(x(A))*h(x(B)))
True
sage: cp.product_on_basis(A,B)==cp(p(cp(A))*p(cp(B)))
True
sage: p.product_on_basis(A,B)==p(e(p(A))*e(p(B)))
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctionsNonCommutingVariables(QQ).e()
>>> h = SymmetricFunctionsNonCommutingVariables(QQ).h()
>>> x = SymmetricFunctionsNonCommutingVariables(QQ).x()
>>> cp = SymmetricFunctionsNonCommutingVariables(QQ).cp()
>>> p = SymmetricFunctionsNonCommutingVariables(QQ).p()
>>> chi = SymmetricFunctionsNonCommutingVariables(QQ).chi()
>>> rho = SymmetricFunctionsNonCommutingVariables(QQ).rho()
>>> A = SetPartition([[Integer(1)], [Integer(2), Integer(3)]])
>>> B = SetPartition([[Integer(1)], [Integer(3)], [Integer(2),
↪Integer(4)]])
>>> e.product_on_basis(A, B)
e{{1}, {2, 3}, {4}, {5, 7}, {6}}
>>> h.product_on_basis(A, B)
h{{1}, {2, 3}, {4}, {5, 7}, {6}}
>>> x.product_on_basis(A, B)
x{{1}, {2, 3}, {4}, {5, 7}, {6}}
>>> cp.product_on_basis(A, B)
cp{{1}, {2, 3}, {4}, {5, 7}, {6}}
>>> p.product_on_basis(A, B)
p{{1}, {2, 3}, {4}, {5, 7}, {6}}
>>> chi.product_on_basis(A, B)
chi{{1}, {2, 3}, {4}, {5, 7}, {6}}
>>> rho.product_on_basis(A, B)
rho{{1}, {2, 3}, {4}, {5, 7}, {6}}
>>> e.product_on_basis(A,B)==e(h(e(A))*h(e(B)))
True
>>> h.product_on_basis(A,B)==h(x(h(A))*x(h(B)))
True
>>> x.product_on_basis(A,B)==x(h(x(A))*h(x(B)))
True
>>> cp.product_on_basis(A,B)==cp(p(cp(A))*p(cp(B)))
True
>>> p.product_on_basis(A,B)==p(e(p(A))*e(p(B)))
True
```

super_categories()

Return the super categories of bases of the Hopf dual of the symmetric functions in non-commuting variables.

OUTPUT: list of categories

class sage.combinat.ncsym.bases.**NCSymBases**(*parent_with_realization*)

Bases: Category_realization_of_parent

Category of bases of symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: from sage.combinat.ncsym.bases import NCSymBases
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: NCSymBases(NCSym)
Category of bases of symmetric functions in non-commuting variables over the␣
↪Rational Field
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsym.bases import NCSymBases
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> NCSymBases(NCSym)
Category of bases of symmetric functions in non-commuting variables over the␣
↪Rational Field
```

class ElementMethods

Bases: `object`

expand(*n*, *alphabet='x'*)

Expand the symmetric function into `n` non-commuting variables in an alphabet, which by default is `'x'`.

This computation is completed by coercing the element `self` into the monomial basis and computing the expansion in the `alphabet` there.

INPUT:

- `n` – the number of variables in the expansion
- `alphabet` – (default: `'x'`) the alphabet in which `self` is to be expanded

OUTPUT:

- an expansion of `self` into the `n` non-commuting variables specified by `alphabet`

EXAMPLES:

```
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: h[[1,3],[2]].expand(3)
2*x0^3 + x0^2*x1 + x0^2*x2 + 2*x0*x1*x0 + x0*x1^2 + x0*x1*x2 + 2*x0*x2*x0
 + x0*x2*x1 + x0*x2^2 + x1*x0^2 + 2*x1*x0*x1 + x1*x0*x2 + x1^2*x0 + 2*x1^3
 + x1^2*x2 + x1*x2*x0 + 2*x1*x2*x1 + x1*x2^2 + x2*x0^2 + x2*x0*x1 +␣
↪2*x2*x0*x2
 + x2*x1*x0 + x2*x1^2 + 2*x2*x1*x2 + x2^2*x0 + x2^2*x1 + 2*x2^3
sage: x = SymmetricFunctionsNonCommutingVariables(QQ).x()
sage: x[[1,3],[2]].expand(3)
-x0^2*x1 - x0^2*x2 - x0*x1^2 - x0*x1*x2 - x0*x2*x1 - x0*x2^2 - x1*x0^2
 - x1*x0*x2 - x1^2*x0 - x1^2*x2 - x1*x2*x0 - x1*x2^2 - x2*x0^2 - x2*x0*x1
 - x2*x1*x0 - x2*x1^2 - x2^2*x0 - x2^2*x1
```

```
>>> from sage.all import *
>>> h = SymmetricFunctionsNonCommutingVariables(QQ).h()
>>> h[[Integer(1),Integer(3)],[Integer(2)]].expand(Integer(3))
2*x0^3 + x0^2*x1 + x0^2*x2 + 2*x0*x1*x0 + x0*x1^2 + x0*x1*x2 + 2*x0*x2*x0
 + x0*x2*x1 + x0*x2^2 + x1*x0^2 + 2*x1*x0*x1 + x1*x0*x2 + x1^2*x0 + 2*x1^3
 + x1^2*x2 + x1*x2*x0 + 2*x1*x2*x1 + x1*x2^2 + x2*x0^2 + x2*x0*x1 +␣
↪2*x2*x0*x2
 + x2*x1*x0 + x2*x1^2 + 2*x2*x1*x2 + x2^2*x0 + x2^2*x1 + 2*x2^3
>>> x = SymmetricFunctionsNonCommutingVariables(QQ).x()
>>> x[[Integer(1),Integer(3)],[Integer(2)]].expand(Integer(3))
-x0^2*x1 - x0^2*x2 - x0*x1^2 - x0*x1*x2 - x0*x2*x1 - x0*x2^2 - x1*x0^2
 - x1*x0*x2 - x1^2*x0 - x1^2*x2 - x1*x2*x0 - x1*x2^2 - x2*x0^2 - x2*x0*x1
 - x2*x1*x0 - x2*x1^2 - x2^2*x0 - x2^2*x1
```

internal_coproduct()

Return the internal coproduct of `self`.

The internal coproduct is defined on the power sum basis as

$$\mathbf{p}_A \mapsto \mathbf{p}_A \otimes \mathbf{p}_A$$

and the map is extended linearly.

OUTPUT: an element of the tensor square of the basis of `self`

EXAMPLES:

```
sage: x = SymmetricFunctionsNonCommutingVariables(QQ).x()
sage: x[[1,3],[2]].internal_coproduct()
x{{1}, {2}, {3}} # x{{1, 3}, {2}} + x{{1, 3}, {2}} # x{{1}, {2}, {3}}
 + x{{1, 3}, {2}} # x{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> x = SymmetricFunctionsNonCommutingVariables(QQ).x()
>>> x[[Integer(1),Integer(3)],[Integer(2)]].internal_coproduct()
x{{1}, {2}, {3}} # x{{1, 3}, {2}} + x{{1, 3}, {2}} # x{{1}, {2}, {3}}
 + x{{1, 3}, {2}} # x{{1, 3}, {2}}
```

omega()

Return the involution ω applied to `self`.

The involution ω is defined by

$$\mathbf{e}_A \mapsto \mathbf{h}_A$$

and the result is extended linearly.

OUTPUT: an element in the same basis as `self`

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: m[[1,3],[2]].omega()
-2*m{{1, 2, 3}} - m{{1, 3}, {2}}
sage: p = NCSym.p()
sage: p[[1,3],[2]].omega()
-p{{1, 3}, {2}}
sage: cp = NCSym.cp()
sage: cp[[1,3],[2]].omega()
-2*cp{{1, 2, 3}} - cp{{1, 3}, {2}}
sage: x = NCSym.x()
sage: x[[1,3],[2]].omega()
-2*x{{1}, {2}, {3}} - x{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> m = NCSym.m()
>>> m[[Integer(1),Integer(3)],[Integer(2)]].omega()
-2*m{{1, 2, 3}} - m{{1, 3}, {2}}
>>> p = NCSym.p()
>>> p[[Integer(1),Integer(3)],[Integer(2)]].omega()
-p{{1, 3}, {2}}
```

(continues on next page)

(continued from previous page)

```
>>> cp = NCSym.cp()
>>> cp[[Integer(1),Integer(3)],[Integer(2)]].omega()
-2*cp{{1, 2, 3}} - cp{{1, 3}, {2}}
>>> x = NCSym.x()
>>> x[[Integer(1),Integer(3)],[Integer(2)]].omega()
-2*x{{1}, {2}, {3}} - x{{1, 3}, {2}}
```

to_symmetric_function()

Compute the projection of an element of symmetric function in non-commuting variables to the symmetric functions.

The projection of a monomial symmetric function in non-commuting variables indexed by the set partition A is defined as

$$\mathbf{m}_A \mapsto m_{\lambda(A)} \prod_i n_i(\lambda(A))!$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts and $n_i(\mu)$ is the multiplicity of i in μ. For other bases this map is extended linearly.

OUTPUT: an element of the symmetric functions in the monomial basis

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: e = NCSym.e()
sage: h = NCSym.h()
sage: p = NCSym.p()
sage: cp = NCSym.cp()
sage: x = NCSym.x()
sage: cp[[1,3],[2]].to_symmetric_function()
m[2, 1]
sage: x[[1,3],[2]].to_symmetric_function()
-6*m[1, 1, 1] - 2*m[2, 1]
sage: e[[1,3],[2]].to_symmetric_function()
2*e[2, 1]
sage: h[[1,3],[2]].to_symmetric_function()
2*h[2, 1]
sage: p[[1,3],[2]].to_symmetric_function()
p[2, 1]
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> e = NCSym.e()
>>> h = NCSym.h()
>>> p = NCSym.p()
>>> cp = NCSym.cp()
>>> x = NCSym.x()
>>> cp[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
m[2, 1]
>>> x[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
-6*m[1, 1, 1] - 2*m[2, 1]
>>> e[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
2*e[2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> h[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
2*h[2, 1]
>>> p[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
p[2, 1]
```

to_wqsym()

Return the image of `self` under the canonical inclusion map $NCSym \to WQSym$.

The canonical inclusion map $NCSym \to WQSym$ is an injective homomorphism of algebras. It sends a basis element $\mathbf{m}_A$ of $NCSym$ to the sum of basis elements $\mathbf{M}_P$ of $WQSym$, where P ranges over all ordered set partitions that become A when the ordering is forgotten. This map is denoted by θ in [BZ05] (17).

See also:

WordQuasiSymmetricFunctions for a definition of $WQSym$.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: e = NCSym.e()
sage: h = NCSym.h()
sage: p = NCSym.p()
sage: cp = NCSym.cp()
sage: x = NCSym.x()
sage: m = NCSym.m()
sage: m[[1,3],[2]].to_wqsym()
M[{1, 3}, {2}] + M[{2}, {1, 3}]
sage: x[[1,3],[2]].to_wqsym()
-M[{1}, {2}, {3}] - M[{1}, {2, 3}] - M[{1}, {3}, {2}]
 - M[{1, 2}, {3}] - M[{2}, {1}, {3}] - M[{2}, {3}, {1}]
 - M[{2, 3}, {1}] - M[{3}, {1}, {2}] - M[{3}, {1, 2}]
 - M[{3}, {2}, {1}]
sage: (4*p[[1,3],[2]]-p[[1]]).to_wqsym()
-M[{1}] + 4*M[{1, 2, 3}] + 4*M[{1, 3}, {2}] + 4*M[{2}, {1, 3}]
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> e = NCSym.e()
>>> h = NCSym.h()
>>> p = NCSym.p()
>>> cp = NCSym.cp()
>>> x = NCSym.x()
>>> m = NCSym.m()
>>> m[[Integer(1),Integer(3)],[Integer(2)]].to_wqsym()
M[{1, 3}, {2}] + M[{2}, {1, 3}]
>>> x[[Integer(1),Integer(3)],[Integer(2)]].to_wqsym()
-M[{1}, {2}, {3}] - M[{1}, {2, 3}] - M[{1}, {3}, {2}]
 - M[{1, 2}, {3}] - M[{2}, {1}, {3}] - M[{2}, {3}, {1}]
 - M[{2, 3}, {1}] - M[{3}, {1}, {2}] - M[{3}, {1, 2}]
 - M[{3}, {2}, {1}]
```

(continues on next page)

(continued from previous page)

```
>>> (Integer(4)*p[[Integer(1),Integer(3)],[Integer(2)]]-p[[Integer(1)]]).
↪to_wqsym()
-M[{1}] + 4*M[{1, 2, 3}] + 4*M[{1, 3}, {2}] + 4*M[{2}, {1, 3}]
```

class ParentMethods

Bases: `object`

from_symmetric_function(*f*)

Return the image of the symmetric function `f` in `self`.

This is performed by converting to the monomial basis and extending the method `sum_of_partitions()` linearly. This is a linear map from the symmetric functions to the symmetric functions in non-commuting variables that does not preserve the product or coproduct structure of the Hopf algebra.

See also:

`to_symmetric_function()`

INPUT:

- `f` – a symmetric function

OUTPUT: an element of `self`

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: Sym = SymmetricFunctions(QQ)
sage: e = NCSym.e()
sage: elem = Sym.e()
sage: elt = e.from_symmetric_function(elem[2,1,1]); elt
1/12*e{{1}, {2}, {3, 4}} + 1/12*e{{1}, {2, 3}, {4}} + 1/12*e{{1}, {2, 4},
↪{3}}
 + 1/12*e{{1, 2}, {3}, {4}} + 1/12*e{{1, 3}, {2}, {4}} + 1/12*e{{1, 4},
↪{2}, {3}}
sage: elem(elt.to_symmetric_function())
e[2, 1, 1]
sage: e.from_symmetric_function(elem[4])
1/24*e{{1, 2, 3, 4}}
sage: p = NCSym.p()
sage: pow = Sym.p()
sage: elt = p.from_symmetric_function(pow[2,1,1]); elt
1/6*p{{1}, {2}, {3, 4}} + 1/6*p{{1}, {2, 3}, {4}} + 1/6*p{{1}, {2, 4}, {3}
↪}
 + 1/6*p{{1, 2}, {3}, {4}} + 1/6*p{{1, 3}, {2}, {4}} + 1/6*p{{1, 4}, {2},
↪{3}}
sage: pow(elt.to_symmetric_function())
p[2, 1, 1]
sage: p.from_symmetric_function(pow[4])
p{{1, 2, 3, 4}}
sage: h = NCSym.h()
sage: comp = Sym.complete()
sage: elt = h.from_symmetric_function(comp[2,1,1]); elt
1/12*h{{1}, {2}, {3, 4}} + 1/12*h{{1}, {2, 3}, {4}} + 1/12*h{{1}, {2, 4},
```

(continues on next page)

(continued from previous page)

```
↪{3}}
 + 1/12*h{{1, 2}, {3}, {4}} + 1/12*h{{1, 3}, {2}, {4}} + 1/12*h{{1, 4},
↪{2}, {3}}
sage: comp(elt.to_symmetric_function())
h[2, 1, 1]
sage: h.from_symmetric_function(comp[4])
1/24*h{{1, 2, 3, 4}}
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> Sym = SymmetricFunctions(QQ)
>>> e = NCSym.e()
>>> elem = Sym.e()
>>> elt = e.from_symmetric_function(elem[Integer(2),Integer(1),
↪Integer(1)]); elt
1/12*e{{1}, {2}, {3, 4}} + 1/12*e{{1}, {2, 3}, {4}} + 1/12*e{{1}, {2, 4},
↪{3}}
 + 1/12*e{{1, 2}, {3}, {4}} + 1/12*e{{1, 3}, {2}, {4}} + 1/12*e{{1, 4},
↪{2}, {3}}
>>> elem(elt.to_symmetric_function())
e[2, 1, 1]
>>> e.from_symmetric_function(elem[Integer(4)])
1/24*e{{1, 2, 3, 4}}
>>> p = NCSym.p()
>>> pow = Sym.p()
>>> elt = p.from_symmetric_function(pow[Integer(2),Integer(1),
↪Integer(1)]); elt
1/6*p{{1}, {2}, {3, 4}} + 1/6*p{{1}, {2, 3}, {4}} + 1/6*p{{1}, {2, 4}, {3}
↪}
 + 1/6*p{{1, 2}, {3}, {4}} + 1/6*p{{1, 3}, {2}, {4}} + 1/6*p{{1, 4}, {2},
↪{3}}
>>> pow(elt.to_symmetric_function())
p[2, 1, 1]
>>> p.from_symmetric_function(pow[Integer(4)])
p{{1, 2, 3, 4}}
>>> h = NCSym.h()
>>> comp = Sym.complete()
>>> elt = h.from_symmetric_function(comp[Integer(2),Integer(1),
↪Integer(1)]); elt
1/12*h{{1}, {2}, {3, 4}} + 1/12*h{{1}, {2, 3}, {4}} + 1/12*h{{1}, {2, 4},
↪{3}}
 + 1/12*h{{1, 2}, {3}, {4}} + 1/12*h{{1, 3}, {2}, {4}} + 1/12*h{{1, 4},
↪{2}, {3}}
>>> comp(elt.to_symmetric_function())
h[2, 1, 1]
>>> h.from_symmetric_function(comp[Integer(4)])
1/24*h{{1, 2, 3, 4}}
```

internal_coproduct()

Compute the internal coproduct of `self`.

If *internal_coproduct_on_basis()* is available, construct the internal coproduct morphism from `self` to `self` $\otimes$ `self` by extending it by linearity. Otherwise, this uses *internal_coprod-*

uct_by_coercion(), if available.

OUTPUT: an element of the tensor squared of `self`

EXAMPLES:

```
sage: cp = SymmetricFunctionsNonCommutingVariables(QQ).cp()
sage: cp.internal_coproduct(cp[[1,3],[2]] - 2*cp[[1]])
-2*cp{{1}} # cp{{1}} + cp{{1, 2, 3}} # cp{{1, 3}, {2}} + cp{{1, 3}, {2}}
↪# cp{{1, 2, 3}}
 + cp{{1, 3}, {2}} # cp{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> cp = SymmetricFunctionsNonCommutingVariables(QQ).cp()
>>> cp.internal_coproduct(cp[[Integer(1),Integer(3)],[Integer(2)]] -␣
↪Integer(2)*cp[[Integer(1)]])
-2*cp{{1}} # cp{{1}} + cp{{1, 2, 3}} # cp{{1, 3}, {2}} + cp{{1, 3}, {2}}
↪# cp{{1, 2, 3}}
 + cp{{1, 3}, {2}} # cp{{1, 3}, {2}}
```

internal_coproduct_by_coercion(*x*)

Return the internal coproduct by coercing the element to the powersum basis.

INPUT:

- `x` – an element of `self`

OUTPUT: an element of the tensor squared of `self`

EXAMPLES:

```
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: h[[1,3],[2]].internal_coproduct() # indirect doctest
2*h{{1}, {2}, {3}} # h{{1}, {2}, {3}} - h{{1}, {2}, {3}} # h{{1, 3}, {2}}
 - h{{1, 3}, {2}} # h{{1}, {2}, {3}} + h{{1, 3}, {2}} # h{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> h = SymmetricFunctionsNonCommutingVariables(QQ).h()
>>> h[[Integer(1),Integer(3)],[Integer(2)]].internal_coproduct() #␣
↪indirect doctest
2*h{{1}, {2}, {3}} # h{{1}, {2}, {3}} - h{{1}, {2}, {3}} # h{{1, 3}, {2}}
 - h{{1, 3}, {2}} # h{{1}, {2}, {3}} + h{{1, 3}, {2}} # h{{1, 3}, {2}}
```

internal_coproduct_on_basis(*i*)

The internal coproduct of the algebra on the basis (optional).

INPUT:

- `i` – the indices of an element of the basis of `self`

OUTPUT: an element of the tensor squared of `self`

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.internal_coproduct_on_basis(SetPartition([[1,2]]))
m{{1, 2}} # m{{1, 2}}
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).m()
>>> m.internal_coproduct_on_basis(SetPartition([[Integer(1),Integer(2)]]))
m{{1, 2}} # m{{1, 2}}
```

primitive(*A*, *i=1*)

Return the primitive associated to `A` in `self`.

See also

primitive()

INPUT:

- `A` – set partition
- `i` – positive integer

OUTPUT: an element of `self`

EXAMPLES:

```
sage: e = SymmetricFunctionsNonCommutingVariables(QQ).e()
sage: elt = e.primitive(SetPartition([[1,3],[2]])); elt
e{{1, 2}, {3}} - e{{1, 3}, {2}}
sage: elt.coproduct()
e{} # e{{1, 2}, {3}} - e{} # e{{1, 3}, {2}} + e{{1, 2}, {3}} # e{} - e{{1,
↪ 3}, {2}} # e{}
```

```
>>> from sage.all import *
>>> e = SymmetricFunctionsNonCommutingVariables(QQ).e()
>>> elt = e.primitive(SetPartition([[Integer(1),Integer(3)],
↪[Integer(2)]])); elt
e{{1, 2}, {3}} - e{{1, 3}, {2}}
>>> elt.coproduct()
e{} # e{{1, 2}, {3}} - e{} # e{{1, 3}, {2}} + e{{1, 2}, {3}} # e{} - e{{1,
↪ 3}, {2}} # e{}
```

super_categories()

Return the super categories of bases of the Hopf dual of the symmetric functions in non-commuting variables.

OUTPUT: list of categories

class sage.combinat.ncsym.bases.**NCSymBasis_abstract**(*R*, *basis_keys=None*, *element_class=None*, *category=None*, *prefix=None*, *names=None*, ***kwds*)

Bases: *CombinatorialFreeModule*, `BindableClass`

Abstract base class for a basis of $NCSym$ or its dual.

class sage.combinat.ncsym.bases.**NCSymDualBases**(*parent_with_realization*)

Bases: `Category_realization_of_parent`

Category of bases of dual symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: from sage.combinat.ncsym.bases import NCSymDualBases
sage: DNCSym = SymmetricFunctionsNonCommutingVariables(QQ).dual()
sage: NCSymDualBases(DNCSym)
Category of bases of dual symmetric functions in non-commuting variables over the␣
↪Rational Field
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsym.bases import NCSymDualBases
>>> DNCSym = SymmetricFunctionsNonCommutingVariables(QQ).dual()
>>> NCSymDualBases(DNCSym)
Category of bases of dual symmetric functions in non-commuting variables over the␣
↪Rational Field
```

super_categories()

Return the super categories of bases of the Hopf dual of the symmetric functions in non-commuting variables.

OUTPUT: list of categories

class sage.combinat.ncsym.bases.**NCSymOrNCSymDualBases**(*parent_with_realization*)

Bases: Category_realization_of_parent

Base category for the category of bases of symmetric functions in non-commuting variables or its Hopf dual for the common code.

class ElementMethods

Bases: object

duality_pairing(*other*)

Compute the pairing between self and an element other of the dual.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: w = m.dual_basis()
sage: elt = m[[1,3],[2]] - 3*m[[1,2],[3]]
sage: elt.duality_pairing(w[[1,3],[2]])
1
sage: elt.duality_pairing(w[[1,2],[3]])
-3
sage: elt.duality_pairing(w[[1,2]])
0
sage: e = NCSym.e()
sage: w[[1,3],[2]].duality_pairing(e[[1,3],[2]])
0
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> m = NCSym.m()
>>> w = m.dual_basis()
>>> elt = m[[Integer(1),Integer(3)],[Integer(2)]] -␣
↪Integer(3)*m[[Integer(1),Integer(2)],[Integer(3)]]
>>> elt.duality_pairing(w[[Integer(1),Integer(3)],[Integer(2)]])
1
```

(continues on next page)

(continued from previous page)

```
>>> elt.duality_pairing(w[[Integer(1),Integer(2)],[Integer(3)]])
-3
>>> elt.duality_pairing(w[[Integer(1),Integer(2)]])
0
>>> e = NCSym.e()
>>> w[[Integer(1),Integer(3)],[Integer(2)]].duality_pairing(e[[Integer(1),
↪Integer(3)],[Integer(2)]])
0
```

class ParentMethods

Bases: `object`

counit_on_basis(*A*)

The counit is defined by sending all elements of positive degree to zero.

INPUT:

- `A` – set partition

OUTPUT: either the 0 or the 1 of the base ring of `self`

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.counit_on_basis(SetPartition([[1,3], [2]]))
0
sage: m.counit_on_basis(SetPartition([]))
1
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.counit_on_basis(SetPartition([[1,3], [2]]))
0
sage: w.counit_on_basis(SetPartition([]))
1
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).m()
>>> m.counit_on_basis(SetPartition([[Integer(1),Integer(3)],␣
↪[Integer(2)]]))
0
>>> m.counit_on_basis(SetPartition([]))
1
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> w.counit_on_basis(SetPartition([[Integer(1),Integer(3)],␣
↪[Integer(2)]]))
0
>>> w.counit_on_basis(SetPartition([]))
1
```

duality_pairing(*x*, *y*)

Compute the pairing between an element of `self` and an element of the dual.

Carry out this computation by converting `x` to the **m** basis and `y` to the **w** basis.

INPUT:

- `x` – an element of symmetric functions in non-commuting variables
- `y` – an element of the dual of symmetric functions in non-commuting variables

OUTPUT: an element of the base ring of `self`

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: h = NCSym.h()
sage: w = NCSym.m().dual_basis()
sage: matrix([[h(A).duality_pairing(w(B)) for A in SetPartitions(3)] for␣
↪B in SetPartitions(3)])
[6 2 2 2 1]
[2 2 1 1 1]
[2 1 2 1 1]
[2 1 1 2 1]
[1 1 1 1 1]
sage: (h[[1,2],[3]] + 3*h[[1,3],[2]]).duality_pairing(2*w[[1,3],[2]] +␣
↪w[[1,2,3]] + 2*w[[1,2],[3]])
32
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> h = NCSym.h()
>>> w = NCSym.m().dual_basis()
>>> matrix([[h(A).duality_pairing(w(B)) for A in␣
↪SetPartitions(Integer(3))] for B in SetPartitions(Integer(3))])
[6 2 2 2 1]
[2 2 1 1 1]
[2 1 2 1 1]
[2 1 1 2 1]
[1 1 1 1 1]
>>> (h[[Integer(1),Integer(2)],[Integer(3)]] + Integer(3)*h[[Integer(1),
↪Integer(3)],[Integer(2)]]).duality_pairing(Integer(2)*w[[Integer(1),
↪Integer(3)],[Integer(2)]] + w[[Integer(1),Integer(2),Integer(3)]] +␣
↪Integer(2)*w[[Integer(1),Integer(2)],[Integer(3)]])
32
```

duality_pairing_matrix(*basis*, *degree*)

The matrix of scalar products between elements of $NCSym$ and elements of $NCSym^*$.

INPUT:

- `basis` – a basis of the dual Hopf algebra
- `degree` – nonnegative integer

OUTPUT:

The matrix of scalar products between the basis `self` and the basis `basis` in the dual Hopf algebra of degree `degree`.

EXAMPLES:

The matrix between the **m** basis and the **w** basis:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: w = NCSym.dual().w()
sage: m.duality_pairing_matrix(w, 3)
[1 0 0 0 0]
[0 1 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> m = NCSym.m()
>>> w = NCSym.dual().w()
>>> m.duality_pairing_matrix(w, Integer(3))
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

Similarly for some of the other basis of $NCSym$ and the **w** basis:

```
sage: e = NCSym.e()
sage: e.duality_pairing_matrix(w, 3)
[0 0 0 0 1]
[0 0 1 1 1]
[0 1 0 1 1]
[0 1 1 0 1]
[1 1 1 1 1]
sage: p = NCSym.p()
sage: p.duality_pairing_matrix(w, 3)
[1 0 0 0 0]
[1 1 0 0 0]
[1 0 1 0 0]
[1 0 0 1 0]
[1 1 1 1 1]
sage: cp = NCSym.cp()
sage: cp.duality_pairing_matrix(w, 3)
[1 0 0 0 0]
[1 1 0 0 0]
[0 0 1 0 0]
[1 0 0 1 0]
[1 1 1 1 1]
sage: x = NCSym.x()
sage: w.duality_pairing_matrix(x, 3)
[ 0  0  0  0  1]
[ 1  0 -1 -1  1]
[ 1 -1  0 -1  1]
[ 1 -1 -1  0  1]
[ 2 -1 -1 -1  1]
```

```
>>> from sage.all import *
>>> e = NCSym.e()
>>> e.duality_pairing_matrix(w, Integer(3))
[0 0 0 0 1]
[0 0 1 1 1]
[0 1 0 1 1]
```

(continues on next page)

(continued from previous page)

```
[0 1 1 0 1]
[1 1 1 1 1]
>>> p = NCSym.p()
>>> p.duality_pairing_matrix(w, Integer(3))
[1 0 0 0 0]
[1 1 0 0 0]
[1 0 1 0 0]
[1 0 0 1 0]
[1 1 1 1 1]
>>> cp = NCSym.cp()
>>> cp.duality_pairing_matrix(w, Integer(3))
[1 0 0 0 0]
[1 1 0 0 0]
[0 0 1 0 0]
[1 0 0 1 0]
[1 1 1 1 1]
>>> x = NCSym.x()
>>> w.duality_pairing_matrix(x, Integer(3))
[ 0  0  0  0  1]
[ 1  0 -1 -1  1]
[ 1 -1  0 -1  1]
[ 1 -1 -1  0  1]
[ 2 -1 -1 -1  1]
```

A base case test:

```
sage: m.duality_pairing_matrix(w, 0)
[1]
```

```
>>> from sage.all import *
>>> m.duality_pairing_matrix(w, Integer(0))
[1]
```

one_basis()

Return the index of the basis element containing 1.

OUTPUT: the empty set partition

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.one_basis()
{}
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.one_basis()
{}
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).m()
>>> m.one_basis()
{}
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
```

(continues on next page)

(continued from previous page)

```
>>> w.one_basis()
{}
```

super_categories()

Return the super categories of bases of (the Hopf dual of) the symmetric functions in non-commuting variables.

OUTPUT: list of categories

5.1.149 Dual Symmetric Functions in Non-Commuting Variables

AUTHORS:

- Travis Scrimshaw (08-04-2013): Initial version

class sage.combinat.ncsym.dual.**SymmetricFunctionsNonCommutingVariablesDual**(*R*)

Bases: UniqueRepresentation, Parent

The Hopf dual to the symmetric functions in non-commuting variables.

See Section 2.3 of [BZ05] for a study.

a_realization()

Return the realization of the **w** basis of self.

EXAMPLES:

```
sage: SymmetricFunctionsNonCommutingVariables(QQ).dual().a_realization()
Dual symmetric functions in non-commuting variables over the Rational Field␣
↪in the w basis
```

```
>>> from sage.all import *
>>> SymmetricFunctionsNonCommutingVariables(QQ).dual().a_realization()
Dual symmetric functions in non-commuting variables over the Rational Field␣
↪in the w basis
```

dual()

Return the dual Hopf algebra of the dual symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: NCSymD = SymmetricFunctionsNonCommutingVariables(QQ).dual()
sage: NCSymD.dual()
Symmetric functions in non-commuting variables over the Rational Field
```

```
>>> from sage.all import *
>>> NCSymD = SymmetricFunctionsNonCommutingVariables(QQ).dual()
>>> NCSymD.dual()
Symmetric functions in non-commuting variables over the Rational Field
```

class w(*NCSymD*)

Bases: *NCSymBasis_abstract*

The dual Hopf algebra of symmetric functions in non-commuting variables in the **w** basis.

EXAMPLES:

```
sage: NCSymD = SymmetricFunctionsNonCommutingVariables(QQ).dual()
sage: w = NCSymD.w()
```

```
>>> from sage.all import *
>>> NCSymD = SymmetricFunctionsNonCommutingVariables(QQ).dual()
>>> w = NCSymD.w()
```

We have the embedding χ^* of Sym into $NCSym^*$ available as a coercion:

```
sage: h = SymmetricFunctions(QQ).h()
sage: w(h[2,1])
w{{1}, {2, 3}} + w{{1, 2}, {3}} + w{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> w(h[Integer(2),Integer(1)])
w{{1}, {2, 3}} + w{{1, 2}, {3}} + w{{1, 3}, {2}}
```

Similarly we can pull back when we are in the image of χ^*:

```
sage: elt = 3*(w[[1],[2,3]] + w[[1,2],[3]] + w[[1,3],[2]])
sage: h(elt)
3*h[2, 1]
```

```
>>> from sage.all import *
>>> elt = Integer(3)*(w[[Integer(1)],[Integer(2),Integer(3)]] + w[[Integer(1),
↪Integer(2)],[Integer(3)]] + w[[Integer(1),Integer(3)],[Integer(2)]])
>>> h(elt)
3*h[2, 1]
```

class Element

Bases: `IndexedFreeModuleElement`

An element in the **w** basis.

expand(*n*, *letter='x'*)

Expand `self` written in the **w** basis in n^2 commuting variables which satisfy the relation $x_{ij}x_{ik} = 0$ for all i, j, and k.

The expansion of an element of the **w** basis is given by equations (26) and (55) in [HNT06].

INPUT:
- `n` – integer
- `letter` – string (default: `'x'`)

OUTPUT:
- The symmetric function of `self` expressed in the `n*n` non-commuting variables described by `letter`.

REFERENCES:

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w[[1,3],[2]].expand(4)
x02*x11*x20 + x03*x11*x30 + x03*x22*x30 + x13*x22*x31
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> w[[Integer(1),Integer(3)],[Integer(2)]].expand(Integer(4))
x02*x11*x20 + x03*x11*x30 + x03*x22*x30 + x13*x22*x31
```

One can use a different set of variable by using the optional argument `letter`:

```
sage: w[[1,3],[2]].expand(3, letter='y')
y02*y11*y20
```

```
>>> from sage.all import *
>>> w[[Integer(1),Integer(3)],[Integer(2)]].expand(Integer(3), letter=
↪'y')
y02*y11*y20
```

is_symmetric()

Determine if a $NCSym^*$ function, expressed in the **w** basis, is symmetric.

A function f in the **w** basis is a symmetric function if it is in the image of χ^*. That is to say we have

$$f = \sum_{\lambda} c_{\lambda} \prod_i m_i(\lambda)! \sum_{\lambda(A)=\lambda} \mathbf{w}_A$$

where the second sum is over all set partitions A whose shape $\lambda(A)$ is equal to λ and $m_i(\mu)$ is the multiplicity of i in the partition μ.

OUTPUT:

- `True` if $\lambda(A) = \lambda(B)$ implies the coefficients of $\mathbf{w}_A$ and $\mathbf{w}_B$ are equal, `False` otherwise

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: elt = w.sum_of_partitions([2,1,1])
sage: elt.is_symmetric()
True
sage: elt -= 3*w.sum_of_partitions([1,1])
sage: elt.is_symmetric()
True
sage: w = SymmetricFunctionsNonCommutingVariables(ZZ).dual().w()
sage: elt = w.sum_of_partitions([2,1,1]) / 2
sage: elt.is_symmetric()
False
sage: elt = w[[1,3],[2]]
sage: elt.is_symmetric()
False
sage: elt = w[[1],[2,3]] + w[[1,2],[3]] + 2*w[[1,3],[2]]
sage: elt.is_symmetric()
False
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> elt = w.sum_of_partitions([Integer(2),Integer(1),Integer(1)])
>>> elt.is_symmetric()
True
>>> elt -= Integer(3)*w.sum_of_partitions([Integer(1),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> elt.is_symmetric()
True
>>> w = SymmetricFunctionsNonCommutingVariables(ZZ).dual().w()
>>> elt = w.sum_of_partitions([Integer(2),Integer(1),Integer(1)]) /
↪Integer(2)
>>> elt.is_symmetric()
False
>>> elt = w[[Integer(1),Integer(3)],[Integer(2)]]
>>> elt.is_symmetric()
False
>>> elt = w[[Integer(1)],[Integer(2),Integer(3)]] + w[[Integer(1),
↪Integer(2)],[Integer(3)]] + Integer(2)*w[[Integer(1),Integer(3)],
↪[Integer(2)]]
>>> elt.is_symmetric()
False
```

to_symmetric_function()

Take a function in the **w** basis, and return its symmetric realization, when possible, expressed in the homogeneous basis of symmetric functions.

OUTPUT:

- If `self` is a symmetric function, then the expansion in the homogeneous basis of the symmetric functions is returned. Otherwise an error is raised.

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: elt = w[[1],[2,3]] + w[[1,2],[3]] + w[[1,3],[2]]
sage: elt.to_symmetric_function()
h[2, 1]
sage: elt = w.sum_of_partitions([2,1,1]) / 2
sage: elt.to_symmetric_function()
1/2*h[2, 1, 1]
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> elt = w[[Integer(1)],[Integer(2),Integer(3)]] + w[[Integer(1),
↪Integer(2)],[Integer(3)]] + w[[Integer(1),Integer(3)],[Integer(2)]]
>>> elt.to_symmetric_function()
h[2, 1]
>>> elt = w.sum_of_partitions([Integer(2),Integer(1),Integer(1)]) /
↪Integer(2)
>>> elt.to_symmetric_function()
1/2*h[2, 1, 1]
```

antipode_on_basis(*A*)

Return the antipode applied to the basis element indexed by `A`.

INPUT:

- `A` – set partition

OUTPUT: an element in the basis `self`

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.antipode_on_basis(SetPartition([[1],[2,3]]))
-3*w{{1}, {2}, {3}} + w{{1, 2}, {3}} + w{{1, 3}, {2}}
sage: F = w[[1,3],[5],[2,4]].coproduct()
sage: F.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> w.antipode_on_basis(SetPartition([[Integer(1)],[Integer(2),
↪Integer(3)]]))
-3*w{{1}, {2}, {3}} + w{{1, 2}, {3}} + w{{1, 3}, {2}}
>>> F = w[[Integer(1),Integer(3)],[Integer(5)],[Integer(2),Integer(4)]].
↪coproduct()
>>> F.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

coproduct_on_basis(A)

Return the coproduct of a **w** basis element.

The coproduct on the basis element $\mathbf{w}_A$ is the sum over tensor product terms $\mathbf{w}_B \otimes \mathbf{w}_C$ where B is the restriction of A to $\{1, 2, \ldots, k\}$ and C is the restriction of A to $\{k + 1, k + 2, \ldots, n\}$.

INPUT:

- `A` – set partition

OUTPUT:

- The coproduct applied to the **w** dual symmetric function in non-commuting variables indexed by `A` expressed in the **w** basis.

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w[[1], [2,3]].coproduct()
w{} # w{{1}, {2, 3}} + w{{1}} # w{{1, 2}}
 + w{{1}, {2}} # w{{1}} + w{{1}, {2, 3}} # w{}
sage: w.coproduct_on_basis(SetPartition([]))
w{} # w{}
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> w[[Integer(1)], [Integer(2),Integer(3)]].coproduct()
w{} # w{{1}, {2, 3}} + w{{1}} # w{{1, 2}}
 + w{{1}, {2}} # w{{1}} + w{{1}, {2, 3}} # w{}
>>> w.coproduct_on_basis(SetPartition([]))
w{} # w{}
```

dual_basis()

Return the dual basis to the **w** basis.

The dual basis to the **w** basis is the monomial basis of the symmetric functions in non-commuting variables.

OUTPUT: the monomial basis of the symmetric functions in non-commuting variables

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.dual_basis()
Symmetric functions in non-commuting variables over the Rational Field in␣
↪the monomial basis
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> w.dual_basis()
Symmetric functions in non-commuting variables over the Rational Field in␣
↪the monomial basis
```

duality_pairing(*x*, *y*)

Compute the pairing between an element of `self` and an element of the dual.

INPUT:

- `x` – an element of the dual of symmetric functions in non-commuting variables
- `y` – an element of the symmetric functions in non-commuting variables

OUTPUT: an element of the base ring of `self`

EXAMPLES:

```
sage: DNCSym = SymmetricFunctionsNonCommutingVariablesDual(QQ)
sage: w = DNCSym.w()
sage: m = w.dual_basis()
sage: matrix([[w(A).duality_pairing(m(B)) for A in SetPartitions(3)] for␣
↪B in SetPartitions(3)])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: (w[[1,2],[3]] + 3*w[[1,3],[2]]).duality_pairing(2*m[[1,3],[2]] +␣
↪m[[1,2,3]] + 2*m[[1,2],[3]])
8
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: matrix([[w(A).duality_pairing(h(B)) for A in SetPartitions(3)] for␣
↪B in SetPartitions(3)])
[6 2 2 2 1]
[2 2 1 1 1]
[2 1 2 1 1]
[2 1 1 2 1]
[1 1 1 1 1]
sage: (2*w[[1,3],[2]] + w[[1,2,3]] + 2*w[[1,2],[3]]).duality_pairing(h[[1,
↪2],[3]] + 3*h[[1,3],[2]])
32
```

```
>>> from sage.all import *
>>> DNCSym = SymmetricFunctionsNonCommutingVariablesDual(QQ)
>>> w = DNCSym.w()
>>> m = w.dual_basis()
>>> matrix([[w(A).duality_pairing(m(B)) for A in␣
↪SetPartitions(Integer(3))] for B in SetPartitions(Integer(3))])
[1 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
>>> (w[[Integer(1),Integer(2)],[Integer(3)]] + Integer(3)*w[[Integer(1),
↪Integer(3)],[Integer(2)]]).duality_pairing(Integer(2)*m[[Integer(1),
↪Integer(3)],[Integer(2)]] + m[[Integer(1),Integer(2),Integer(3)]] +␣
↪Integer(2)*m[[Integer(1),Integer(2)],[Integer(3)]])
8
>>> h = SymmetricFunctionsNonCommutingVariables(QQ).h()
>>> matrix([[w(A).duality_pairing(h(B)) for A in␣
↪SetPartitions(Integer(3))] for B in SetPartitions(Integer(3))])
[6 2 2 2 1]
[2 2 1 1 1]
[2 1 2 1 1]
[2 1 1 2 1]
[1 1 1 1 1]
>>> (Integer(2)*w[[Integer(1),Integer(3)],[Integer(2)]] + w[[Integer(1),
↪Integer(2),Integer(3)]] + Integer(2)*w[[Integer(1),Integer(2)],
↪[Integer(3)]]).duality_pairing(h[[Integer(1),Integer(2)],[Integer(3)]]␣
↪+ Integer(3)*h[[Integer(1),Integer(3)],[Integer(2)]])
32
```

product_on_basis(A, B)

The product on **w** basis elements.

The product on the **w** is the dual to the coproduct on the **m** basis. On the basis **w** it is defined as

$$\mathbf{w}_A \mathbf{w}_B = \sum_{S \subseteq [n]} \mathbf{w}_{A\uparrow_S \cup B\uparrow_{S^c}}$$

where the sum is over all possible subsets S of $[n]$ such that $|S| = |A|$ with a term indexed the union of $A \uparrow_S$ and $B \uparrow_{S^c}$. The notation $A \uparrow_S$ represents the unique set partition of the set S such that the standardization is A. This product is commutative.

INPUT:

- A, B – set partitions

OUTPUT: an element of the **w** basis

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: A = SetPartition([[1], [2,3]])
sage: B = SetPartition([[1, 2, 3]])
sage: w.product_on_basis(A, B)
w{{1}, {2, 3}, {4, 5, 6}} + w{{1}, {2, 3, 4}, {5, 6}}
 + w{{1}, {2, 3, 5}, {4, 6}} + w{{1}, {2, 3, 6}, {4, 5}}
 + w{{1}, {2, 4}, {3, 5, 6}} + w{{1}, {2, 4, 5}, {3, 6}}
 + w{{1}, {2, 4, 6}, {3, 5}} + w{{1}, {2, 5}, {3, 4, 6}}
 + w{{1}, {2, 5, 6}, {3, 4}} + w{{1}, {2, 6}, {3, 4, 5}}
 + w{{1, 2, 3}, {4}, {5, 6}} + w{{1, 2, 4}, {3}, {5, 6}}
 + w{{1, 2, 5}, {3}, {4, 6}} + w{{1, 2, 6}, {3}, {4, 5}}
 + w{{1, 3, 4}, {2}, {5, 6}} + w{{1, 3, 5}, {2}, {4, 6}}
 + w{{1, 3, 6}, {2}, {4, 5}} + w{{1, 4, 5}, {2}, {3, 6}}
```

(continues on next page)

(continued from previous page)

```
 + w{{1, 4, 6}, {2}, {3, 5}} + w{{1, 5, 6}, {2}, {3, 4}}
sage: B = SetPartition([[1], [2]])
sage: w.product_on_basis(A, B)
3*w{{1}, {2}, {3}, {4, 5}} + 2*w{{1}, {2}, {3, 4}, {5}}
 + 2*w{{1}, {2}, {3, 5}, {4}} + w{{1}, {2, 3}, {4}, {5}}
 + w{{1}, {2, 4}, {3}, {5}} + w{{1}, {2, 5}, {3}, {4}}
sage: w.product_on_basis(A, SetPartition([]))
w{{1}, {2, 3}}
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> A = SetPartition([[Integer(1)], [Integer(2),Integer(3)]])
>>> B = SetPartition([[Integer(1), Integer(2), Integer(3)]])
>>> w.product_on_basis(A, B)
w{{1}, {2, 3}, {4, 5, 6}} + w{{1}, {2, 3, 4}, {5, 6}}
 + w{{1}, {2, 3, 5}, {4, 6}} + w{{1}, {2, 3, 6}, {4, 5}}
 + w{{1}, {2, 4}, {3, 5, 6}} + w{{1}, {2, 4, 5}, {3, 6}}
 + w{{1}, {2, 4, 6}, {3, 5}} + w{{1}, {2, 5}, {3, 4, 6}}
 + w{{1}, {2, 5, 6}, {3, 4}} + w{{1}, {2, 6}, {3, 4, 5}}
 + w{{1, 2, 3}, {4}, {5, 6}} + w{{1, 2, 4}, {3}, {5, 6}}
 + w{{1, 2, 5}, {3}, {4, 6}} + w{{1, 2, 6}, {3}, {4, 5}}
 + w{{1, 3, 4}, {2}, {5, 6}} + w{{1, 3, 5}, {2}, {4, 6}}
 + w{{1, 3, 6}, {2}, {4, 5}} + w{{1, 4, 5}, {2}, {3, 6}}
 + w{{1, 4, 6}, {2}, {3, 5}} + w{{1, 5, 6}, {2}, {3, 4}}
>>> B = SetPartition([[Integer(1)], [Integer(2)]])
>>> w.product_on_basis(A, B)
3*w{{1}, {2}, {3}, {4, 5}} + 2*w{{1}, {2}, {3, 4}, {5}}
 + 2*w{{1}, {2}, {3, 5}, {4}} + w{{1}, {2, 3}, {4}, {5}}
 + w{{1}, {2, 4}, {3}, {5}} + w{{1}, {2, 5}, {3}, {4}}
>>> w.product_on_basis(A, SetPartition([]))
w{{1}, {2, 3}}
```

sum_of_partitions(*la*)

Return the sum over all sets partitions whose shape is `la`, scaled by $\prod_i m_i!$ where m_i is the multiplicity of i in `la`.

INPUT:

- `la` – integer partition

OUTPUT: an element of `self`

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.sum_of_partitions([2,1,1])
2*w{{1}, {2}, {3, 4}} + 2*w{{1}, {2, 3}, {4}} + 2*w{{1}, {2, 4}, {3}}
 + 2*w{{1, 2}, {3}, {4}} + 2*w{{1, 3}, {2}, {4}} + 2*w{{1, 4}, {2}, {3}}
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> w.sum_of_partitions([Integer(2),Integer(1),Integer(1)])
2*w{{1}, {2}, {3, 4}} + 2*w{{1}, {2, 3}, {4}} + 2*w{{1}, {2, 4}, {3}}
 + 2*w{{1, 2}, {3}, {4}} + 2*w{{1, 3}, {2}, {4}} + 2*w{{1, 4}, {2}, {3}}
```

to_symmetric_function()

The preimage of χ^* in the $\mathbf{w}$ basis.

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.to_symmetric_function
Generic morphism:
  From: Dual symmetric functions in non-commuting variables over the␣
↪Rational Field in the w basis
  To:   Symmetric Functions over Rational Field in the homogeneous basis
```

```
>>> from sage.all import *
>>> w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
>>> w.to_symmetric_function
Generic morphism:
  From: Dual symmetric functions in non-commuting variables over the␣
↪Rational Field in the w basis
  To:   Symmetric Functions over Rational Field in the homogeneous basis
```

5.1.150 Symmetric Functions in Non-Commuting Variables

AUTHORS:

- Travis Scrimshaw (08-04-2013): Initial version

class sage.combinat.ncsym.ncsym.**SymmetricFunctionsNonCommutingVariables**(R)

Bases: UniqueRepresentation, Parent

Symmetric functions in non-commutative variables.

The ring of symmetric functions in non-commutative variables, which is not to be confused with the *non-commutative symmetric functions*, is the ring of all bounded-degree noncommutative power series in countably many indeterminates (i.e., elements in $R\langle\langle x_1, x_2, x_3, \ldots\rangle\rangle$ of bounded degree) which are invariant with respect to the action of the symmetric group S_∞ on the indices of the indeterminates. It can be regarded as a direct limit over all $n \to \infty$ of rings of S_n-invariant polynomials in n non-commuting variables (that is, S_n-invariant elements of $R\langle x_1, x_2, \ldots, x_n\rangle$).

This ring is implemented as a Hopf algebra whose basis elements are indexed by set partitions.

Let $A = \{A_1, A_2, \ldots, A_r\}$ be a set partition of the integers $[k] := \{1, 2, \ldots, k\}$. This partition A determines an equivalence relation $\sim_A$ on $[k]$, which has $c \sim_A d$ if and only if c and d are in the same part A_j of A. The monomial basis element $\mathbf{m}_A$ indexed by A is the sum of monomials $x_{i_1} x_{i_2} \cdots x_{i_k}$ such that $i_c = i_d$ if and only if $c \sim_A d$.

The k-th graded component of the ring of symmetric functions in non-commutative variables has its dimension equal to the number of set partitions of $[k]$. (If we work, instead, with finitely many – say, n – variables, then its dimension is equal to the number of set partitions of $[k]$ where the number of parts is at most n.)

Note

All set partitions are considered standard (i.e., set partitions of $[n]$ for some n) unless otherwise stated.

REFERENCES:

EXAMPLES:

We begin by first creating the ring of $NCSym$ and the bases that are analogues of the usual symmetric functions:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: e = NCSym.e()
sage: h = NCSym.h()
sage: p = NCSym.p()
sage: m
Symmetric functions in non-commuting variables over the Rational Field in the␣
↪monomial basis
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> m = NCSym.m()
>>> e = NCSym.e()
>>> h = NCSym.h()
>>> p = NCSym.p()
>>> m
Symmetric functions in non-commuting variables over the Rational Field in the␣
↪monomial basis
```

The basis is indexed by set partitions, so we create a few elements and convert them between these bases:

```
sage: elt = m(SetPartition([[1,3],[2]])) - 2*m(SetPartition([[1],[2]])); elt
-2*m{{1}, {2}} + m{{1, 3}, {2}}
sage: e(elt)
1/2*e{{1}, {2, 3}} - 2*e{{1, 2}} + 1/2*e{{1, 2}, {3}} - 1/2*e{{1, 2, 3}} - 1/2*e{
↪{1, 3}, {2}}
sage: h(elt)
-4*h{{1}, {2}} - 2*h{{1}, {2}, {3}} + 1/2*h{{1}, {2, 3}} + 2*h{{1, 2}}
 + 1/2*h{{1, 2}, {3}} - 1/2*h{{1, 2, 3}} + 3/2*h{{1, 3}, {2}}
sage: p(elt)
-2*p{{1}, {2}} + 2*p{{1, 2}} - p{{1, 2, 3}} + p{{1, 3}, {2}}
sage: m(p(elt))
-2*m{{1}, {2}} + m{{1, 3}, {2}}

sage: elt = p(SetPartition([[1,3],[2]])) - 4*p(SetPartition([[1],[2]])) + 2; elt
2*p{} - 4*p{{1}, {2}} + p{{1, 3}, {2}}
sage: e(elt)
2*e{} - 4*e{{1}, {2}} + e{{1}, {2}, {3}} - e{{1, 3}, {2}}
sage: m(elt)
2*m{} - 4*m{{1}, {2}} - 4*m{{1, 2}} + m{{1, 2, 3}} + m{{1, 3}, {2}}
sage: h(elt)
2*h{} - 4*h{{1}, {2}} - h{{1}, {2}, {3}} + h{{1, 3}, {2}}
sage: p(m(elt))
2*p{} - 4*p{{1}, {2}} + p{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> elt = m(SetPartition([[Integer(1),Integer(3)],[Integer(2)]])) -␣
↪Integer(2)*m(SetPartition([[Integer(1)],[Integer(2)]])); elt
-2*m{{1}, {2}} + m{{1, 3}, {2}}
>>> e(elt)
1/2*e{{1}, {2, 3}} - 2*e{{1, 2}} + 1/2*e{{1, 2}, {3}} - 1/2*e{{1, 2, 3}} - 1/2*e{
```

(continues on next page)

(continued from previous page)

```
↪{1, 3}, {2}}
>>> h(elt)
-4*h{{1}, {2}} - 2*h{{1}, {2}, {3}} + 1/2*h{{1}, {2, 3}} + 2*h{{1, 2}}
 + 1/2*h{{1, 2}, {3}} - 1/2*h{{1, 2, 3}} + 3/2*h{{1, 3}, {2}}
>>> p(elt)
-2*p{{1}, {2}} + 2*p{{1, 2}} - p{{1, 2, 3}} + p{{1, 3}, {2}}
>>> m(p(elt))
-2*m{{1}, {2}} + m{{1, 3}, {2}}

>>> elt = p(SetPartition([[Integer(1),Integer(3)],[Integer(2)]])) -␣
↪Integer(4)*p(SetPartition([[Integer(1)],[Integer(2)]])) + Integer(2); elt
2*p{} - 4*p{{1}, {2}} + p{{1, 3}, {2}}
>>> e(elt)
2*e{} - 4*e{{1}, {2}} + e{{1}, {2}, {3}} - e{{1, 3}, {2}}
>>> m(elt)
2*m{} - 4*m{{1}, {2}} - 4*m{{1, 2}} + m{{1, 2, 3}} + m{{1, 3}, {2}}
>>> h(elt)
2*h{} - 4*h{{1}, {2}} - h{{1}, {2}, {3}} + h{{1, 3}, {2}}
>>> p(m(elt))
2*p{} - 4*p{{1}, {2}} + p{{1, 3}, {2}}
```

There is also a shorthand for creating elements. We note that we must use `p[[]]` to create the empty set partition due to python's syntax.

```
sage: eltm = m[[1,3],[2]] - 3*m[[1],[2]]; eltm
-3*m{{1}, {2}} + m{{1, 3}, {2}}
sage: elte = e[[1,3],[2]]; elte
e{{1, 3}, {2}}
sage: elth = h[[1,3],[2,4]]; elth
h{{1, 3}, {2, 4}}
sage: eltp = p[[1,3],[2,4]] + 2*p[[1]] - 4*p[[]]; eltp
-4*p{} + 2*p{{1}} + p{{1, 3}, {2, 4}}
```

```
>>> from sage.all import *
>>> eltm = m[[Integer(1),Integer(3)],[Integer(2)]] - Integer(3)*m[[Integer(1)],
↪[Integer(2)]]; eltm
-3*m{{1}, {2}} + m{{1, 3}, {2}}
>>> elte = e[[Integer(1),Integer(3)],[Integer(2)]]; elte
e{{1, 3}, {2}}
>>> elth = h[[Integer(1),Integer(3)],[Integer(2),Integer(4)]]; elth
h{{1, 3}, {2, 4}}
>>> eltp = p[[Integer(1),Integer(3)],[Integer(2),Integer(4)]] +␣
↪Integer(2)*p[[Integer(1)]] - Integer(4)*p[[]]; eltp
-4*p{} + 2*p{{1}} + p{{1, 3}, {2, 4}}
```

There is also a natural projection to the usual symmetric functions by letting the variables commute. This projection map preserves the product and coproduct structure. We check that Theorem 2.1 of [RS06] holds:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sm = Sym.m()
sage: Se = Sym.e()
sage: Sh = Sym.h()
```

(continues on next page)

(continued from previous page)

```
sage: Sp = Sym.p()
sage: eltm.to_symmetric_function()
-6*m[1, 1] + m[2, 1]
sage: Sm(p(eltm).to_symmetric_function())
-6*m[1, 1] + m[2, 1]
sage: elte.to_symmetric_function()
2*e[2, 1]
sage: Se(h(elte).to_symmetric_function())
2*e[2, 1]
sage: elth.to_symmetric_function()
4*h[2, 2]
sage: Sh(m(elth).to_symmetric_function())
4*h[2, 2]
sage: eltp.to_symmetric_function()
-4*p[] + 2*p[1] + p[2, 2]
sage: Sp(e(eltp).to_symmetric_function())
-4*p[] + 2*p[1] + p[2, 2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Sm = Sym.m()
>>> Se = Sym.e()
>>> Sh = Sym.h()
>>> Sp = Sym.p()
>>> eltm.to_symmetric_function()
-6*m[1, 1] + m[2, 1]
>>> Sm(p(eltm).to_symmetric_function())
-6*m[1, 1] + m[2, 1]
>>> elte.to_symmetric_function()
2*e[2, 1]
>>> Se(h(elte).to_symmetric_function())
2*e[2, 1]
>>> elth.to_symmetric_function()
4*h[2, 2]
>>> Sh(m(elth).to_symmetric_function())
4*h[2, 2]
>>> eltp.to_symmetric_function()
-4*p[] + 2*p[1] + p[2, 2]
>>> Sp(e(eltp).to_symmetric_function())
-4*p[] + 2*p[1] + p[2, 2]
```

a_realization()

Return the realization of the powersum basis of `self`.

OUTPUT: the powersum basis of symmetric functions in non-commuting variables

EXAMPLES:

```
sage: SymmetricFunctionsNonCommutingVariables(QQ).a_realization()
Symmetric functions in non-commuting variables over the Rational Field in the
↪powersum basis
```

```
>>> from sage.all import *
>>> SymmetricFunctionsNonCommutingVariables(QQ).a_realization()
Symmetric functions in non-commuting variables over the Rational Field in the␣
↪powersum basis
```

chi

alias of *supercharacter*

class coarse_powersum(*NCSym*)

Bases: *NCSymBasis_abstract*

The Hopf algebra of symmetric functions in non-commuting variables in the **cp** basis.

This basis was defined in [BZ05] as

$$\mathbf{cp}_A = \sum_{A \leq_* B} \mathbf{m}_B,$$

where we sum over all strict coarsenings of the set partition A. An alternative description of this basis was given in [BT13] as

$$\mathbf{cp}_A = \sum_{A \subseteq B} \mathbf{m}_B,$$

where we sum over all set partitions whose arcs are a subset of the arcs of the set partition A.

Note

In [BZ05], this basis was denoted by **q**. In [BT13], this basis was called the powersum basis and denoted by p. However it is a coarser basis than the usual powersum basis in the sense that it does not yield the usual powersum basis of the symmetric function under the natural map of letting the variables commute.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: cp = NCSym.cp()
sage: cp[[1,3],[2,4]]*cp[[1,2,3]]
cp{{1, 3}, {2, 4}, {5, 6, 7}}
sage: cp[[1,2],[3]].internal_coproduct()
cp{{1, 2}, {3}} # cp{{1, 2}, {3}}
sage: ps = SymmetricFunctions(NCSym.base_ring()).p()
sage: ps(cp[[1,3],[2]].to_symmetric_function())
p[2, 1] - p[3]
sage: ps(cp[[1,2],[3]].to_symmetric_function())
p[2, 1]
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> cp = NCSym.cp()
>>> cp[[Integer(1),Integer(3)],[Integer(2),Integer(4)]]*cp[[Integer(1),
↪Integer(2),Integer(3)]]
cp{{1, 3}, {2, 4}, {5, 6, 7}}
>>> cp[[Integer(1),Integer(2)],[Integer(3)]].internal_coproduct()
cp{{1, 2}, {3}} # cp{{1, 2}, {3}}
```

(continues on next page)

(continued from previous page)

```
>>> ps = SymmetricFunctions(NCSym.base_ring()).p()
>>> ps(cp[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function())
p[2, 1] - p[3]
>>> ps(cp[[Integer(1),Integer(2)],[Integer(3)]].to_symmetric_function())
p[2, 1]
```

cp

alias of *coarse_powersum*

class deformed_coarse_powersum(*NCSym*, *q=2*)

Bases: *NCSymBasis_abstract*

The Hopf algebra of symmetric functions in non-commuting variables in the ρ basis.

This basis was defined in [BT13] as a q-deformation of the **cp** basis:

$$\rho_A = \sum_{A \subseteq B} \frac{1}{q^{\mathrm{nst}_{B-A}^{A}}} \mathbf{m}_B,$$

where we sum over all set partitions whose arcs are a subset of the arcs of the set partition A.

INPUT:

- q – (default: 2) the parameter q

EXAMPLES:

```
sage: R = QQ['q'].fraction_field()
sage: q = R.gen()
sage: NCSym = SymmetricFunctionsNonCommutingVariables(R)
sage: rho = NCSym.rho(q)
```

```
>>> from sage.all import *
>>> R = QQ['q'].fraction_field()
>>> q = R.gen()
>>> NCSym = SymmetricFunctionsNonCommutingVariables(R)
>>> rho = NCSym.rho(q)
```

We construct Example 3.1 in [BT13]:

```
sage: rnode = lambda A: sorted([a[1] for a in A.arcs()], reverse=True)
sage: dimv = lambda A: sorted([a[1]-a[0] for a in A.arcs()], reverse=True)
sage: lst = list(SetPartitions(4))
sage: S = sorted(lst, key=lambda A: (dimv(A), rnode(A)))
sage: m = NCSym.m()
sage: matrix([[m(rho[A])[B] for B in S] for A in S])
[  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1]
[  0   1   0   0   1   1   0   1   0   0   1   0   0   0   0]
[  0   0   1   0   1   0   1   1   0   0   0   0   0   0   1]
[  0   0   0   1   0   1   1   1   0   0   0   1   0   0   0]
[  0   0   0   0   1   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   1   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   1   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   1   0   0   1   1   0   0]
```

(continues on next page)

(continued from previous page)

```
[  0   0   0   0   0   0   0   0   0   1   1   0   1   0   0]
[  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   1 1/q]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1]
```

```
>>> from sage.all import *
>>> rnode = lambda A: sorted([a[Integer(1)] for a in A.arcs()], reverse=True)
>>> dimv = lambda A: sorted([a[Integer(1)]-a[Integer(0)] for a in A.arcs()],␣
↪reverse=True)
>>> lst = list(SetPartitions(Integer(4)))
>>> S = sorted(lst, key=lambda A: (dimv(A), rnode(A)))
>>> m = NCSym.m()
>>> matrix([[m(rho[A])[B] for B in S] for A in S])
[  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1]
[  0   1   0   0   1   1   0   1   0   0   1   0   0   0   0]
[  0   0   1   0   1   0   1   1   0   0   0   0   0   0   1]
[  0   0   0   1   0   1   1   1   0   0   0   1   0   0   0]
[  0   0   0   0   1   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   1   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   1   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   1   0   0   1   1   0   0]
[  0   0   0   0   0   0   0   0   0   1   1   0   1   0   0]
[  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   1 1/q]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1]
```

q()

Return the deformation parameter q of `self`.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: rho = NCSym.rho(5)
sage: rho.q()
5

sage: R = QQ['q'].fraction_field()
sage: q = R.gen()
sage: NCSym = SymmetricFunctionsNonCommutingVariables(R)
sage: rho = NCSym.rho(q)
sage: rho.q() == q
True
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> rho = NCSym.rho(Integer(5))
>>> rho.q()
```

(continues on next page)

(continued from previous page)

```
5

>>> R = QQ['q'].fraction_field()
>>> q = R.gen()
>>> NCSym = SymmetricFunctionsNonCommutingVariables(R)
>>> rho = NCSym.rho(q)
>>> rho.q() == q
True
```

dual()

Return the dual Hopf algebra of the symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: SymmetricFunctionsNonCommutingVariables(QQ).dual()
Dual symmetric functions in non-commuting variables over the Rational Field
```

```
>>> from sage.all import *
>>> SymmetricFunctionsNonCommutingVariables(QQ).dual()
Dual symmetric functions in non-commuting variables over the Rational Field
```

e

alias of *elementary*

class elementary(*NCSym*)

Bases: *NCSymBasis_abstract*

The Hopf algebra of symmetric functions in non-commuting variables in the elementary basis.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: e = NCSym.e()
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> e = NCSym.e()
```

class Element

Bases: `IndexedFreeModuleElement`

An element in the elementary basis of $NCSym$.

omega()

Return the involution ω applied to `self`.

The involution ω on $NCSym$ is defined by $\omega(\mathbf{e}_A) = \mathbf{h}_A$.

OUTPUT: an element in the basis `self`

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: e = NCSym.e()
sage: h = NCSym.h()
```

(continues on next page)

(continued from previous page)

```
sage: elt = e[[1,3],[2]].omega(); elt
2*e{{1}, {2}, {3}} - e{{1, 3}, {2}}
sage: elt.omega()
e{{1, 3}, {2}}
sage: h(elt)
h{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> e = NCSym.e()
>>> h = NCSym.h()
>>> elt = e[[Integer(1),Integer(3)],[Integer(2)]].omega(); elt
2*e{{1}, {2}, {3}} - e{{1, 3}, {2}}
>>> elt.omega()
e{{1, 3}, {2}}
>>> h(elt)
h{{1, 3}, {2}}
```

to_symmetric_function()

The projection of `self` to the symmetric functions.

Take a symmetric function in non-commuting variables expressed in the **e** basis, and return the projection of expressed in the elementary basis of symmetric functions.

The map $\chi: NCSym \to Sym$ is given by

$$\mathbf{e}_A \mapsto e_{\lambda(A)} \prod_i \lambda(A)_i!$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts.

OUTPUT: an element of the symmetric functions in the elementary basis

EXAMPLES:

```
sage: e = SymmetricFunctionsNonCommutingVariables(QQ).e()
sage: e[[1,3],[2]].to_symmetric_function()
2*e[2, 1]
sage: e[[1],[3],[2]].to_symmetric_function()
e[1, 1, 1]
```

```
>>> from sage.all import *
>>> e = SymmetricFunctionsNonCommutingVariables(QQ).e()
>>> e[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
2*e[2, 1]
>>> e[[Integer(1)],[Integer(3)],[Integer(2)]].to_symmetric_function()
e[1, 1, 1]
```

h

alias of *homogeneous*

class homogeneous(*NCSym*)

Bases: *NCSymBasis_abstract*

The Hopf algebra of symmetric functions in non-commuting variables in the homogeneous basis.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: h = NCSym.h()
sage: h[[1,3],[2,4]]*h[[1,2,3]]
h{{1, 3}, {2, 4}, {5, 6, 7}}
sage: h[[1,2]].coproduct()
h{} # h{{1, 2}} + 2*h{{1}} # h{{1}} + h{{1, 2}} # h{}
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> h = NCSym.h()
>>> h[[Integer(1),Integer(3)],[Integer(2),Integer(4)]]*h[[Integer(1),
↪Integer(2),Integer(3)]]
h{{1, 3}, {2, 4}, {5, 6, 7}}
>>> h[[Integer(1),Integer(2)]].coproduct()
h{} # h{{1, 2}} + 2*h{{1}} # h{{1}} + h{{1, 2}} # h{}
```

class Element

Bases: `IndexedFreeModuleElement`

An element in the homogeneous basis of $NCSym$.

omega()

Return the involution ω applied to `self`.

The involution ω on $NCSym$ is defined by $\omega(\mathbf{h}_A) = \mathbf{e}_A$.

OUTPUT: an element in the basis `self`

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: h = NCSym.h()
sage: e = NCSym.e()
sage: elt = h[[1,3],[2]].omega(); elt
2*h{{1}, {2}, {3}} - h{{1, 3}, {2}}
sage: elt.omega()
h{{1, 3}, {2}}
sage: e(elt)
e{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> h = NCSym.h()
>>> e = NCSym.e()
>>> elt = h[[Integer(1),Integer(3)],[Integer(2)]].omega(); elt
2*h{{1}, {2}, {3}} - h{{1, 3}, {2}}
>>> elt.omega()
h{{1, 3}, {2}}
>>> e(elt)
e{{1, 3}, {2}}
```

to_symmetric_function()

The projection of `self` to the symmetric functions.

Take a symmetric function in non-commuting variables expressed in the **h** basis, and return the projection of expressed in the complete basis of symmetric functions.

The map $\chi: NCSym \to Sym$ is given by

$$\mathbf{h}_A \mapsto h_{\lambda(A)} \prod_i \lambda(A)_i!$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts.

OUTPUT: an element of the symmetric functions in the complete basis

EXAMPLES:

```
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: h[[1,3],[2]].to_symmetric_function()
2*h[2, 1]
sage: h[[1],[3],[2]].to_symmetric_function()
h[1, 1, 1]
```

```
>>> from sage.all import *
>>> h = SymmetricFunctionsNonCommutingVariables(QQ).h()
>>> h[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
2*h[2, 1]
>>> h[[Integer(1)],[Integer(3)],[Integer(2)]].to_symmetric_function()
h[1, 1, 1]
```

m

alias of *monomial*

class monomial(*NCSym*)

Bases: *NCSymBasis_abstract*

The Hopf algebra of symmetric functions in non-commuting variables in the monomial basis.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: m[[1,3],[2]]*m[[1,2]]
m{{1, 3}, {2}, {4, 5}} + m{{1, 3}, {2, 4, 5}} + m{{1, 3, 4, 5}, {2}}
sage: m[[1,3],[2]].coproduct()
m{} # m{{1, 3}, {2}} + m{{1}} # m{{1, 2}} + m{{1, 2}} # m{{1}} + m{{1,
 3}, {2}} # m{}
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> m = NCSym.m()
>>> m[[Integer(1),Integer(3)],[Integer(2)]]*m[[Integer(1),Integer(2)]]
m{{1, 3}, {2}, {4, 5}} + m{{1, 3}, {2, 4, 5}} + m{{1, 3, 4, 5}, {2}}
>>> m[[Integer(1),Integer(3)],[Integer(2)]].coproduct()
m{} # m{{1, 3}, {2}} + m{{1}} # m{{1, 2}} + m{{1, 2}} # m{{1}} + m{{1,
 3}, {2}} # m{}
```

class Element

Bases: `IndexedFreeModuleElement`

An element in the monomial basis of $NCSym$.

expand(*n*, *alphabet='x'*)

Expand `self` written in the monomial basis in n non-commuting variables.

INPUT:

- `n` – integer
- `alphabet` – string (default: `'x'`)

OUTPUT:

- The symmetric function of `self` expressed in the `n` non-commuting variables described by `alphabet`.

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: m[[1,3],[2]].expand(4)
x0*x1*x0 + x0*x2*x0 + x0*x3*x0 + x1*x0*x1 + x1*x2*x1 + x1*x3*x1
 + x2*x0*x2 + x2*x1*x2 + x2*x3*x2 + x3*x0*x3 + x3*x1*x3 + x3*x2*x3
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
>>> m[[Integer(1),Integer(3)],[Integer(2)]].expand(Integer(4))
x0*x1*x0 + x0*x2*x0 + x0*x3*x0 + x1*x0*x1 + x1*x2*x1 + x1*x3*x1
 + x2*x0*x2 + x2*x1*x2 + x2*x3*x2 + x3*x0*x3 + x3*x1*x3 + x3*x2*x3
```

One can use a different set of variables by using the optional argument `alphabet`:

```
sage: m[[1],[2,3]].expand(3,alphabet='y')
y0*y1^2 + y0*y2^2 + y1*y0^2 + y1*y2^2 + y2*y0^2 + y2*y1^2
```

```
>>> from sage.all import *
>>> m[[Integer(1)],[Integer(2),Integer(3)]].expand(Integer(3),alphabet=
↪'y')
y0*y1^2 + y0*y2^2 + y1*y0^2 + y1*y2^2 + y2*y0^2 + y2*y1^2
```

to_symmetric_function()

The projection of `self` to the symmetric functions.

Take a symmetric function in non-commuting variables expressed in the **m** basis, and return the projection of expressed in the monomial basis of symmetric functions.

The map $\chi \colon NCSym \to Sym$ is defined by

$$\mathbf{m}_A \mapsto m_{\lambda(A)} \prod_i n_i(\lambda(A))!$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts and $n_i(\mu)$ is the multiplicity of i in μ.

OUTPUT: an element of the symmetric functions in the monomial basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: m[[1,3],[2]].to_symmetric_function()
m[2, 1]
sage: m[[1],[3],[2]].to_symmetric_function()
6*m[1, 1, 1]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
>>> m[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
m[2, 1]
>>> m[[Integer(1)],[Integer(3)],[Integer(2)]].to_symmetric_function()
6*m[1, 1, 1]
```

coproduct_on_basis(*A*)

Return the coproduct of a monomial basis element.

INPUT:

- `A` – set partition

OUTPUT:

- The coproduct applied to the monomial symmetric function in non-commuting variables indexed by `A` expressed in the monomial basis.

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: m[[1, 3], [2]].coproduct()
m{} # m{{1, 3}, {2}} + m{{1}} # m{{1, 2}} + m{{1, 2}} # m{{1}} + m{{1, 3},
→ {2}} # m{}
sage: m.coproduct_on_basis(SetPartition([]))
m{} # m{}
sage: m.coproduct_on_basis(SetPartition([[1,2,3]]))
m{} # m{{1, 2, 3}} + m{{1, 2, 3}} # m{}
sage: m[[1,5],[2,4],[3,7],[6]].coproduct()
m{} # m{{1, 5}, {2, 4}, {3, 7}, {6}} + m{{1}} # m{{1, 5}, {2, 4}, {3, 6}}
 + 2*m{{1, 2}} # m{{1, 3}, {2, 5}, {4}} + m{{1, 2}} # m{{1, 4}, {2, 3},
→{5}}
 + 2*m{{1, 2}, {3}} # m{{1, 3}, {2, 4}} + m{{1, 3}, {2}} # m{{1, 4}, {2,␣
→3}}
 + 2*m{{1, 3}, {2, 4}} # m{{1, 2}, {3}} + 2*m{{1, 3}, {2, 5}, {4}} # m{{1,
→ 2}}
 + m{{1, 4}, {2, 3}} # m{{1, 3}, {2}} + m{{1, 4}, {2, 3}, {5}} # m{{1, 2}}
 + m{{1, 5}, {2, 4}, {3, 6}} # m{{1}} + m{{1, 5}, {2, 4}, {3, 7}, {6}} # m
→{}
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
>>> m[[Integer(1), Integer(3)], [Integer(2)]].coproduct()
m{} # m{{1, 3}, {2}} + m{{1}} # m{{1, 2}} + m{{1, 2}} # m{{1}} + m{{1, 3},
→ {2}} # m{}
>>> m.coproduct_on_basis(SetPartition([]))
m{} # m{}
>>> m.coproduct_on_basis(SetPartition([[Integer(1),Integer(2),
→Integer(3)]]))
m{} # m{{1, 2, 3}} + m{{1, 2, 3}} # m{}
>>> m[[Integer(1),Integer(5)],[Integer(2),Integer(4)],[Integer(3),
→Integer(7)],[Integer(6)]].coproduct()
m{} # m{{1, 5}, {2, 4}, {3, 7}, {6}} + m{{1}} # m{{1, 5}, {2, 4}, {3, 6}}
 + 2*m{{1, 2}} # m{{1, 3}, {2, 5}, {4}} + m{{1, 2}} # m{{1, 4}, {2, 3},
→{5}}
 + 2*m{{1, 2}, {3}} # m{{1, 3}, {2, 4}} + m{{1, 3}, {2}} # m{{1, 4}, {2,␣
```

(continues on next page)

(continued from previous page)

```
↪3}}
 + 2*m{{1, 3}, {2, 4}} # m{{1, 2}, {3}} + 2*m{{1, 3}, {2, 5}, {4}} # m{{1,
↪ 2}}
 + m{{1, 4}, {2, 3}} # m{{1, 3}, {2}} + m{{1, 4}, {2, 3}, {5}} # m{{1, 2}}
 + m{{1, 5}, {2, 4}, {3, 6}} # m{{1}} + m{{1, 5}, {2, 4}, {3, 7}, {6}} # m
↪{}
```

dual_basis()

Return the dual basis to the monomial basis.

OUTPUT: the **w** basis of the dual Hopf algebra

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.dual_basis()
Dual symmetric functions in non-commuting variables over the Rational␣
↪Field in the w basis
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).m()
>>> m.dual_basis()
Dual symmetric functions in non-commuting variables over the Rational␣
↪Field in the w basis
```

duality_pairing(*x*, *y*)

Compute the pairing between an element of `self` and an element of the dual.

INPUT:

- `x` – an element of symmetric functions in non-commuting variables
- `y` – an element of the dual of symmetric functions in non-commuting variables

OUTPUT: an element of the base ring of `self`

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: w = m.dual_basis()
sage: matrix([[m(A).duality_pairing(w(B)) for A in SetPartitions(3)] for␣
↪B in SetPartitions(3)])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: (m[[1,2],[3]] + 3*m[[1,3],[2]]).duality_pairing(2*w[[1,3],[2]] +␣
↪w[[1,2,3]] + 2*w[[1,2],[3]])
8
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> m = NCSym.m()
>>> w = m.dual_basis()
```

(continues on next page)

(continued from previous page)

```
>>> matrix([[m(A).duality_pairing(w(B)) for A in␣
↪SetPartitions(Integer(3))] for B in SetPartitions(Integer(3))])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
>>> (m[[Integer(1),Integer(2)],[Integer(3)]] + Integer(3)*m[[Integer(1),
↪Integer(3)],[Integer(2)]]).duality_pairing(Integer(2)*w[[Integer(1),
↪Integer(3)],[Integer(2)]] + w[[Integer(1),Integer(2),Integer(3)]] +␣
↪Integer(2)*w[[Integer(1),Integer(2)],[Integer(3)]])
8
```

from_symmetric_function(*f*)

Return the image of the symmetric function `f` in `self`.

This is performed by converting to the monomial basis and extending the method *sum_of_partitions()* linearly. This is a linear map from the symmetric functions to the symmetric functions in non-commuting variables that does not preserve the product or coproduct structure of the Hopf algebra.

See also

to_symmetric_function()

INPUT:

- `f` – an element of the symmetric functions

OUTPUT: an element of the **m** basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: mon = SymmetricFunctions(QQ).m()
sage: elt = m.from_symmetric_function(mon[2,1,1]); elt
1/12*m{{1}, {2}, {3, 4}} + 1/12*m{{1}, {2, 3}, {4}} + 1/12*m{{1}, {2, 4},
↪{3}}
 + 1/12*m{{1, 2}, {3}, {4}} + 1/12*m{{1, 3}, {2}, {4}} + 1/12*m{{1, 4},
↪{2}, {3}}
sage: elt.to_symmetric_function()
m[2, 1, 1]
sage: e = SymmetricFunctionsNonCommutingVariables(QQ).e()
sage: elm = SymmetricFunctions(QQ).e()
sage: e(m.from_symmetric_function(elm[4]))
1/24*e{{1, 2, 3, 4}}
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: hom = SymmetricFunctions(QQ).h()
sage: h(m.from_symmetric_function(hom[4]))
1/24*h{{1, 2, 3, 4}}
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).p()
sage: pow = SymmetricFunctions(QQ).p()
sage: p(m.from_symmetric_function(pow[4]))
p{{1, 2, 3, 4}}
sage: p(m.from_symmetric_function(pow[2,1]))
```

(continues on next page)

(continued from previous page)

```
1/3*p{{1}, {2, 3}} + 1/3*p{{1, 2}, {3}} + 1/3*p{{1, 3}, {2}}
sage: p([[1,2]])*p([[1]])
p{{1, 2}, {3}}
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).m()
>>> mon = SymmetricFunctions(QQ).m()
>>> elt = m.from_symmetric_function(mon[Integer(2),Integer(1),
↪Integer(1)]); elt
1/12*m{{1}, {2}, {3, 4}} + 1/12*m{{1}, {2, 3}, {4}} + 1/12*m{{1}, {2, 4},
↪{3}}
 + 1/12*m{{1, 2}, {3}, {4}} + 1/12*m{{1, 3}, {2}, {4}} + 1/12*m{{1, 4},
↪{2}, {3}}
>>> elt.to_symmetric_function()
m[2, 1, 1]
>>> e = SymmetricFunctionsNonCommutingVariables(QQ).e()
>>> elm = SymmetricFunctions(QQ).e()
>>> e(m.from_symmetric_function(elm[Integer(4)]))
1/24*e{{1, 2, 3, 4}}
>>> h = SymmetricFunctionsNonCommutingVariables(QQ).h()
>>> hom = SymmetricFunctions(QQ).h()
>>> h(m.from_symmetric_function(hom[Integer(4)]))
1/24*h{{1, 2, 3, 4}}
>>> p = SymmetricFunctionsNonCommutingVariables(QQ).p()
>>> pow = SymmetricFunctions(QQ).p()
>>> p(m.from_symmetric_function(pow[Integer(4)]))
p{{1, 2, 3, 4}}
>>> p(m.from_symmetric_function(pow[Integer(2),Integer(1)]))
1/3*p{{1}, {2, 3}} + 1/3*p{{1, 2}, {3}} + 1/3*p{{1, 3}, {2}}
>>> p([[Integer(1),Integer(2)]])*p([[Integer(1)]])
p{{1, 2}, {3}}
```

Check that $\chi \circ \tilde{\chi}$ is the identity on Sym:

```
sage: all(m.from_symmetric_function(pow(la)).to_symmetric_function() ==␣
↪pow(la)
....:     for la in Partitions(4))
True
```

```
>>> from sage.all import *
>>> all(m.from_symmetric_function(pow(la)).to_symmetric_function() ==␣
↪pow(la)
...     for la in Partitions(Integer(4)))
True
```

internal_coproduct_on_basis(*A*)

Return the internal coproduct of a monomial basis element.

The internal coproduct is defined by

$$\Delta^{\odot}(\mathbf{m}_A) = \sum_{B \wedge C = A} \mathbf{m}_B \otimes \mathbf{m}_C$$

where we sum over all pairs of set partitions B and C whose infimum is A.

INPUT:

- A – set partition

OUTPUT: an element of the tensor square of the **m** basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: m.internal_coproduct_on_basis(SetPartition([[1,3],[2]]))
m{{1, 2, 3}} # m{{1, 3}, {2}} + m{{1, 3}, {2}} # m{{1, 2, 3}} + m{{1, 3},
↪{2}} # m{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
>>> m.internal_coproduct_on_basis(SetPartition([[Integer(1),Integer(3)],
↪[Integer(2)]]))
m{{1, 2, 3}} # m{{1, 3}, {2}} + m{{1, 3}, {2}} # m{{1, 2, 3}} + m{{1, 3},
↪{2}} # m{{1, 3}, {2}}
```

product_on_basis(*A*, *B*)

The product on monomial basis elements.

The product of the basis elements indexed by two set partitions A and B is the sum of the basis elements indexed by set partitions C such that $C \wedge ([n]|[k]) = A|B$ where $n = |A|$ and $k = |B|$. Here $A \wedge B$ is the infimum of A and B and $A|B$ is the *SetPartition.pipe()* operation. Equivalently we can describe all C as matchings between the parts of A and B where if $a \in A$ is matched with $b \in B$, we take $a \cup b$ instead of a and b in C.

INPUT:

- A, B – set partitions

OUTPUT: an element of the **m** basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: A = SetPartition([[1], [2,3]])
sage: B = SetPartition([[1], [3], [2,4]])
sage: m.product_on_basis(A, B)
m{{1}, {2, 3}, {4}, {5, 7}, {6}} + m{{1}, {2, 3, 4}, {5, 7}, {6}}
 + m{{1}, {2, 3, 5, 7}, {4}, {6}} + m{{1}, {2, 3, 6}, {4}, {5, 7}}
 + m{{1, 4}, {2, 3}, {5, 7}, {6}} + m{{1, 4}, {2, 3, 5, 7}, {6}}
 + m{{1, 4}, {2, 3, 6}, {5, 7}} + m{{1, 5, 7}, {2, 3}, {4}, {6}}
 + m{{1, 5, 7}, {2, 3, 4}, {6}} + m{{1, 5, 7}, {2, 3, 6}, {4}}
 + m{{1, 6}, {2, 3}, {4}, {5, 7}} + m{{1, 6}, {2, 3, 4}, {5, 7}}
 + m{{1, 6}, {2, 3, 5, 7}, {4}}
sage: B = SetPartition([[1], [2]])
sage: m.product_on_basis(A, B)
m{{1}, {2, 3}, {4}, {5}} + m{{1}, {2, 3, 4}, {5}}
 + m{{1}, {2, 3, 5}, {4}} + m{{1, 4}, {2, 3}, {5}} + m{{1, 4}, {2, 3, 5}}
 + m{{1, 5}, {2, 3}, {4}} + m{{1, 5}, {2, 3, 4}}
sage: m.product_on_basis(A, SetPartition([]))
m{{1}, {2, 3}}
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
>>> A = SetPartition([[Integer(1)], [Integer(2),Integer(3)]])
```

(continues on next page)

(continued from previous page)

```
>>> B = SetPartition([[Integer(1)], [Integer(3)], [Integer(2),
↪Integer(4)]])
>>> m.product_on_basis(A, B)
m{{1}, {2, 3}, {4}, {5, 7}, {6}} + m{{1}, {2, 3, 4}, {5, 7}, {6}}
 + m{{1}, {2, 3, 5, 7}, {4}, {6}} + m{{1}, {2, 3, 6}, {4}, {5, 7}}
 + m{{1, 4}, {2, 3}, {5, 7}, {6}} + m{{1, 4}, {2, 3, 5, 7}, {6}}
 + m{{1, 4}, {2, 3, 6}, {5, 7}} + m{{1, 5, 7}, {2, 3}, {4}, {6}}
 + m{{1, 5, 7}, {2, 3, 4}, {6}} + m{{1, 5, 7}, {2, 3, 6}, {4}}
 + m{{1, 6}, {2, 3}, {4}, {5, 7}} + m{{1, 6}, {2, 3, 4}, {5, 7}}
 + m{{1, 6}, {2, 3, 5, 7}, {4}}
>>> B = SetPartition([[Integer(1)], [Integer(2)]])
>>> m.product_on_basis(A, B)
m{{1}, {2, 3}, {4}, {5}} + m{{1}, {2, 3, 4}, {5}}
 + m{{1}, {2, 3, 5}, {4}} + m{{1, 4}, {2, 3}, {5}} + m{{1, 4}, {2, 3, 5}}
 + m{{1, 5}, {2, 3}, {4}} + m{{1, 5}, {2, 3, 4}}
>>> m.product_on_basis(A, SetPartition([]))
m{{1}, {2, 3}}
```

sum_of_partitions(*la*)

Return the sum over all set partitions whose shape is `la` with a fixed coefficient C defined below.

Fix a partition λ, we define $\lambda! := \prod_i \lambda_i!$ and $\lambda^! := \prod_i m_i!$. Recall that $|\lambda| = \sum_i \lambda_i$ and m_i is the number of parts of length i of λ. Thus we defined the coefficient as

$$C := \frac{\lambda! \lambda^!}{|\lambda|!}.$$

Hence we can define a lift $\widetilde{\chi}$ from Sym to $NCSym$ by

$$m_\lambda \mapsto C \sum_A \mathbf{m}_A$$

where the sum is over all set partitions whose shape is λ.

INPUT:

- `la` – integer partition

OUTPUT: an element of the **m** basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.sum_of_partitions(Partition([2,1,1]))
1/12*m{{1}, {2}, {3, 4}} + 1/12*m{{1}, {2, 3}, {4}} + 1/12*m{{1}, {2, 4},
↪{3}}
 + 1/12*m{{1, 2}, {3}, {4}} + 1/12*m{{1, 3}, {2}, {4}} + 1/12*m{{1, 4},
↪{2}, {3}}
```

```
>>> from sage.all import *
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).m()
>>> m.sum_of_partitions(Partition([Integer(2),Integer(1),Integer(1)]))
1/12*m{{1}, {2}, {3, 4}} + 1/12*m{{1}, {2, 3}, {4}} + 1/12*m{{1}, {2, 4},
↪{3}}
 + 1/12*m{{1, 2}, {3}, {4}} + 1/12*m{{1, 3}, {2}, {4}} + 1/12*m{{1, 4},
↪{2}, {3}}
```

p

alias of `powersum`

class powersum(*NCSym*)

Bases: `NCSymBasis_abstract`

The Hopf algebra of symmetric functions in non-commuting variables in the powersum basis.

The powersum basis is given by

$$\mathbf{p}_A = \sum_{A \leq B} \mathbf{m}_B,$$

where we sum over all coarsenings of the set partition A. If we allow our variables to commute, then $\mathbf{p}_A$ goes to the usual powersum symmetric function p_λ whose (integer) partition λ is the shape of A.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: p = NCSym.p()

sage: x = p.an_element()**2; x
4*p{} + 8*p{{1}} + 4*p{{1}, {2}} + 6*p{{1}, {2, 3}}
 + 12*p{{1, 2}} + 6*p{{1, 2}, {3}} + 9*p{{1, 2}, {3, 4}}
sage: x.to_symmetric_function()
4*p[] + 8*p[1] + 4*p[1, 1] + 12*p[2] + 12*p[2, 1] + 9*p[2, 2]
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> p = NCSym.p()

>>> x = p.an_element()**Integer(2); x
4*p{} + 8*p{{1}} + 4*p{{1}, {2}} + 6*p{{1}, {2, 3}}
 + 12*p{{1, 2}} + 6*p{{1, 2}, {3}} + 9*p{{1, 2}, {3, 4}}
>>> x.to_symmetric_function()
4*p[] + 8*p[1] + 4*p[1, 1] + 12*p[2] + 12*p[2, 1] + 9*p[2, 2]
```

class Element

Bases: `IndexedFreeModuleElement`

An element in the powersum basis of $NCSym$.

to_symmetric_function()

The projection of `self` to the symmetric functions.

Take a symmetric function in non-commuting variables expressed in the **p** basis, and return the projection of expressed in the powersum basis of symmetric functions.

The map $\chi \colon NCSym \to Sym$ is given by

$$\mathbf{p}_A \mapsto p_{\lambda(A)}$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts.

OUTPUT: an element of symmetric functions in the power sum basis

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).p()
sage: p[[1,3],[2]].to_symmetric_function()
p[2, 1]
sage: p[[1],[3],[2]].to_symmetric_function()
p[1, 1, 1]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctionsNonCommutingVariables(QQ).p()
>>> p[[Integer(1),Integer(3)],[Integer(2)]].to_symmetric_function()
p[2, 1]
>>> p[[Integer(1)],[Integer(3)],[Integer(2)]].to_symmetric_function()
p[1, 1, 1]
```

antipode_on_basis(A)

Return the result of the antipode applied to a powersum basis element.

Let A be a set partition. The antipode given in [LM2011] is

$$S(\mathbf{p}_A) = \sum_{\gamma} (-1)^{\ell(\gamma)} \mathbf{p}_{\gamma[A]}$$

where we sum over all ordered set partitions (i.e. set compositions) of $[\ell(A)]$ and

$$\gamma[A] = A^{\downarrow}_{\gamma_1} | \cdots | A^{\downarrow}_{\gamma_{\ell(A)}}$$

is the action of γ on A defined in *SetPartition.ordered_set_partition_action()*.

INPUT:

- `A` – set partition

OUTPUT: an element in the basis `self`

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
sage: p.antipode_on_basis(SetPartition([[1], [2,3]]))
p{{1, 2}, {3}}
sage: p.antipode_on_basis(SetPartition([]))
p{}
sage: F = p[[1,3],[5],[2,4]].coproduct()
sage: F.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

```
>>> from sage.all import *
>>> p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
>>> p.antipode_on_basis(SetPartition([[Integer(1)], [Integer(2),
↪Integer(3)]]))
p{{1, 2}, {3}}
>>> p.antipode_on_basis(SetPartition([]))
p{}
>>> F = p[[Integer(1),Integer(3)],[Integer(5)],[Integer(2),Integer(4)]].
↪coproduct()
>>> F.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

coproduct_on_basis(*A*)

Return the coproduct of a monomial basis element.

INPUT:

- A – set partition

OUTPUT:

- The coproduct applied to the monomial symmetric function in non-commuting variables indexed by A expressed in the monomial basis.

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
sage: p[[1, 3], [2]].coproduct()
p{} # p{{1, 3}, {2}} + p{{1}} # p{{1, 2}} + p{{1, 2}} # p{{1}} + p{{1, 3},
↪ {2}} # p{}
sage: p.coproduct_on_basis(SetPartition([[1]]))
p{} # p{{1}} + p{{1}} # p{}
sage: p.coproduct_on_basis(SetPartition([]))
p{} # p{}
```

```
>>> from sage.all import *
>>> p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
>>> p[[Integer(1), Integer(3)], [Integer(2)]].coproduct()
p{} # p{{1, 3}, {2}} + p{{1}} # p{{1, 2}} + p{{1, 2}} # p{{1}} + p{{1, 3},
↪ {2}} # p{}
>>> p.coproduct_on_basis(SetPartition([[Integer(1)]]))
p{} # p{{1}} + p{{1}} # p{}
>>> p.coproduct_on_basis(SetPartition([]))
p{} # p{}
```

internal_coproduct_on_basis(*A*)

Return the internal coproduct of a powersum basis element.

The internal coproduct is defined by

$$\Delta^{\odot}(\mathbf{p}_A) = \mathbf{p}_A \otimes \mathbf{p}_A$$

INPUT:

- A – set partition

OUTPUT: an element of the tensor square of self

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
sage: p.internal_coproduct_on_basis(SetPartition([[1,3],[2]]))
p{{1, 3}, {2}} # p{{1, 3}, {2}}
```

```
>>> from sage.all import *
>>> p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
>>> p.internal_coproduct_on_basis(SetPartition([[Integer(1),Integer(3)],
↪[Integer(2)]]))
p{{1, 3}, {2}} # p{{1, 3}, {2}}
```

primitive(*A*, *i=1*)

Return the primitive associated to A in self.

Fix some $i \in S$. Let A be an atomic set partition of S, then the primitive $p(A)$ given in [LM2011] is

$$p(A) = \sum_{\gamma} (-1)^{\ell(\gamma)-1} \mathbf{p}_{\gamma[A]}$$

where we sum over all ordered set partitions of $[\ell(A)]$ such that $i \in \gamma_1$ and $\gamma[A]$ is the action of γ on A defined in `SetPartition.ordered_set_partition_action()`. If A is not atomic, then $p(A) = 0$.

See also:

`SetPartition.is_atomic()`

INPUT:

- `A` – set partition
- `i` – (default: 1) index in the base set for `A` specifying which set of primitives this belongs to

OUTPUT: an element in the basis `self`

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
sage: elt = p.primitive(SetPartition([[1,3], [2]])); elt
-p{{1, 2}, {3}} + p{{1, 3}, {2}}
sage: elt.coproduct()
-p{} # p{{1, 2}, {3}} + p{} # p{{1, 3}, {2}} - p{{1, 2}, {3}} # p{} + p{
↪{1, 3}, {2}} # p{}
sage: p.primitive(SetPartition([[1], [2,3]]))
0
sage: p.primitive(SetPartition([]))
p{}
```

```
>>> from sage.all import *
>>> p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
>>> elt = p.primitive(SetPartition([[Integer(1),Integer(3)],
↪[Integer(2)]])); elt
-p{{1, 2}, {3}} + p{{1, 3}, {2}}
>>> elt.coproduct()
-p{} # p{{1, 2}, {3}} + p{} # p{{1, 3}, {2}} - p{{1, 2}, {3}} # p{} + p{
↪{1, 3}, {2}} # p{}
>>> p.primitive(SetPartition([[Integer(1)], [Integer(2),Integer(3)]]))
0
>>> p.primitive(SetPartition([]))
p{}
```

rho

alias of `deformed_coarse_powersum`

class supercharacter(*NCSym*, *q=2*)

Bases: `NCSymBasis_abstract`

The Hopf algebra of symmetric functions in non-commuting variables in the supercharacter χ basis.

This basis was defined in [BT13] as a q-deformation of the supercharacter basis.

$$\chi_A = \sum_B \chi_A(B) \mathbf{m}_B,$$

where we sum over all set partitions A and $\chi_A(B)$ is the evaluation of the supercharacter χ_A on the superclass μ_B.

Note: The supercharacters considered in [BT13] are coarser than those considered by Aguiar et. al.

INPUT:

- q – (default: 2) the parameter q

EXAMPLES:

```
sage: R = QQ['q'].fraction_field()
sage: q = R.gen()
sage: NCSym = SymmetricFunctionsNonCommutingVariables(R)
sage: chi = NCSym.chi(q)
sage: chi[[1,3],[2]]*chi[[1,2]]
chi{{1, 3}, {2}, {4, 5}}
sage: chi[[1,3],[2]].coproduct()
chi{} # chi{{1, 3}, {2}} + (2*q-2)*chi{{1}} # chi{{1}, {2}} +
 (3*q-2)*chi{{1}} # chi{{1, 2}} + (2*q-2)*chi{{1}, {2}} # chi{{1}} +
 (3*q-2)*chi{{1, 2}} # chi{{1}} + chi{{1, 3}, {2}} # chi{}
sage: chi2 = NCSym.chi()
sage: chi(chi2[[1,2],[3]])
((-q+2)/q)*chi{{1}, {2}, {3}} + 2/q*chi{{1, 2}, {3}}
sage: chi2
Symmetric functions in non-commuting variables over the Fraction Field
 of Univariate Polynomial Ring in q over Rational Field in the
 supercharacter basis with parameter q=2
```

```
>>> from sage.all import *
>>> R = QQ['q'].fraction_field()
>>> q = R.gen()
>>> NCSym = SymmetricFunctionsNonCommutingVariables(R)
>>> chi = NCSym.chi(q)
>>> chi[[Integer(1),Integer(3)],[Integer(2)]]*chi[[Integer(1),Integer(2)]]
chi{{1, 3}, {2}, {4, 5}}
>>> chi[[Integer(1),Integer(3)],[Integer(2)]].coproduct()
chi{} # chi{{1, 3}, {2}} + (2*q-2)*chi{{1}} # chi{{1}, {2}} +
 (3*q-2)*chi{{1}} # chi{{1, 2}} + (2*q-2)*chi{{1}, {2}} # chi{{1}} +
 (3*q-2)*chi{{1, 2}} # chi{{1}} + chi{{1, 3}, {2}} # chi{}
>>> chi2 = NCSym.chi()
>>> chi(chi2[[Integer(1),Integer(2)],[Integer(3)]])
((-q+2)/q)*chi{{1}, {2}, {3}} + 2/q*chi{{1, 2}, {3}}
>>> chi2
Symmetric functions in non-commuting variables over the Fraction Field
 of Univariate Polynomial Ring in q over Rational Field in the
 supercharacter basis with parameter q=2
```

q()

Return the deformation parameter q of self.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: chi = NCSym.chi(5)
sage: chi.q()
5

sage: R = QQ['q'].fraction_field()
sage: q = R.gen()
sage: NCSym = SymmetricFunctionsNonCommutingVariables(R)
sage: chi = NCSym.chi(q)
sage: chi.q() == q
True
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> chi = NCSym.chi(Integer(5))
>>> chi.q()
5

>>> R = QQ['q'].fraction_field()
>>> q = R.gen()
>>> NCSym = SymmetricFunctionsNonCommutingVariables(R)
>>> chi = NCSym.chi(q)
>>> chi.q() == q
True
```

x

alias of *x_basis*

class x_basis(*NCSym*)

Bases: *NCSymBasis_abstract*

The Hopf algebra of symmetric functions in non-commuting variables in the **x** basis.

This basis is defined in [BHRZ06] by the formula:

$$\mathbf{x}_A = \sum_{B \leq A} \mu(B, A)\mathbf{p}_B$$

and has the following properties:

$$\mathbf{x}_A\mathbf{x}_B = \mathbf{x}_{A|B}, \qquad \Delta^{\odot}(\mathbf{x}_C) = \sum_{A \vee B = C} \mathbf{x}_A \otimes \mathbf{x}_B.$$

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: x = NCSym.x()
sage: x[[1,3],[2,4]]*x[[1,2,3]]
x{{1, 3}, {2, 4}, {5, 6, 7}}
sage: x[[1,2],[3]].internal_coproduct()
x{{1}, {2}, {3}} # x{{1, 2}, {3}} + x{{1, 2}, {3}} # x{{1}, {2}, {3}} +
 x{{1, 2}, {3}} # x{{1, 2}, {3}}
```

```
>>> from sage.all import *
>>> NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
>>> x = NCSym.x()
>>> x[[Integer(1),Integer(3)],[Integer(2),Integer(4)]]*x[[Integer(1),
↪Integer(2),Integer(3)]]
x{{1, 3}, {2, 4}, {5, 6, 7}}
>>> x[[Integer(1),Integer(2)],[Integer(3)]].internal_coproduct()
x{{1}, {2}, {3}} # x{{1, 2}, {3}} + x{{1, 2}, {3}} # x{{1}, {2}, {3}} +
 x{{1, 2}, {3}} # x{{1, 2}, {3}}
```

sage.combinat.ncsym.ncsym.**matchings**(A, B)

Iterate through all matchings of the sets A and B.

EXAMPLES:

```
sage: from sage.combinat.ncsym.ncsym import matchings
sage: list(matchings([1, 2, 3], [-1, -2]))
[[[1], [2], [3], [-1], [-2]],
 [[1], [2], [3, -1], [-2]],
 [[1], [2], [3, -2], [-1]],
 [[1], [2, -1], [3], [-2]],
 [[1], [2, -1], [3, -2]],
 [[1], [2, -2], [3], [-1]],
 [[1], [2, -2], [3, -1]],
 [[1, -1], [2], [3], [-2]],
 [[1, -1], [2], [3, -2]],
 [[1, -1], [2, -2], [3]],
 [[1, -2], [2], [3], [-1]],
 [[1, -2], [2], [3, -1]],
 [[1, -2], [2, -1], [3]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsym.ncsym import matchings
>>> list(matchings([Integer(1), Integer(2), Integer(3)], [-Integer(1), -
↪Integer(2)]))
[[[1], [2], [3], [-1], [-2]],
 [[1], [2], [3, -1], [-2]],
 [[1], [2], [3, -2], [-1]],
 [[1], [2, -1], [3], [-2]],
 [[1], [2, -1], [3, -2]],
 [[1], [2, -2], [3], [-1]],
 [[1], [2, -2], [3, -1]],
 [[1, -1], [2], [3], [-2]],
 [[1, -1], [2], [3, -2]],
 [[1, -1], [2, -2], [3]],
 [[1, -2], [2], [3], [-1]],
 [[1, -2], [2], [3, -1]],
 [[1, -2], [2, -1], [3]]]
```

sage.combinat.ncsym.ncsym.**nesting**(*la*, *nu*)

Return the nesting number of `la` inside of `nu`.

If we consider a set partition A as a set of arcs $i - j$ where i and j are in the same part of A. Define

$$\mathrm{nst}_{\lambda}^{\nu} = \#\{i < j < k < l \mid i - l \in \nu, j - k \in \lambda\},$$

and this corresponds to the number of arcs of λ strictly contained inside of ν.

EXAMPLES:

```
sage: from sage.combinat.ncsym.ncsym import nesting
sage: nu = SetPartition([[1,4], [2], [3]])
sage: mu = SetPartition([[1,4], [2,3]])
sage: nesting(set(mu).difference(nu), nu)
1

sage: A = SetPartition([[1], [2,5], [3,4]])
sage: B = SetPartition([[1,3,4], [2,5]])
sage: nesting(A, B)
1
sage: nesting(B, A)
1
```

```
>>> from sage.all import *
>>> from sage.combinat.ncsym.ncsym import nesting
>>> nu = SetPartition([[Integer(1),Integer(4)], [Integer(2)], [Integer(3)]])
>>> mu = SetPartition([[Integer(1),Integer(4)], [Integer(2),Integer(3)]])
>>> nesting(set(mu).difference(nu), nu)
1

>>> A = SetPartition([[Integer(1)], [Integer(2),Integer(5)], [Integer(3),
↪Integer(4)]])
>>> B = SetPartition([[Integer(1),Integer(3),Integer(4)], [Integer(2),
↪Integer(5)]])
>>> nesting(A, B)
1
>>> nesting(B, A)
1
```

```
sage: lst = list(SetPartitions(4))
sage: d = {}
sage: for i, nu in enumerate(lst):
....:     for mu in nu.coarsenings():
....:         if set(nu.arcs()).issubset(mu.arcs()):
....:             d[i, lst.index(mu)] = nesting(set(mu).difference(nu), nu)
sage: matrix(d)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
>>> from sage.all import *
>>> lst = list(SetPartitions(Integer(4)))
>>> d = {}
>>> for i, nu in enumerate(lst):
...     for mu in nu.coarsenings():
...         if set(nu.arcs()).issubset(mu.arcs()):
...             d[i, lst.index(mu)] = nesting(set(mu).difference(nu), nu)
>>> matrix(d)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

5.1.151 Necklaces

The algorithm used in this file comes from

- Sawada, Joe. *A fast algorithm to generate necklaces with fixed content*, Theoretical Computer Science archive Volume 301, Issue 1-3 (May 2003) doi:10.1016/S0304-3975(03)00049-5

`sage.combinat.necklace.`**`Necklaces`**`(`*content*`)`

Return the set of necklaces with evaluation `content`.

A necklace is a list of integers that such that the list is the smallest lexicographic representative of all the cyclic shifts of the list.

See also

LyndonWords

INPUT:

- `content` – list or tuple of nonnegative integers

EXAMPLES:

```
sage: Necklaces([2,1,1])
Necklaces with evaluation [2, 1, 1]
sage: Necklaces([2,1,1]).cardinality()
3
sage: Necklaces([2,1,1]).first()
[1, 1, 2, 3]
sage: Necklaces([2,1,1]).last()
[1, 2, 1, 3]
sage: Necklaces([2,1,1]).list()
[[1, 1, 2, 3], [1, 1, 3, 2], [1, 2, 1, 3]]
sage: Necklaces([0,2,1,1]).list()
[[2, 2, 3, 4], [2, 2, 4, 3], [2, 3, 2, 4]]
sage: Necklaces([2,0,1,1]).list()
[[1, 1, 3, 4], [1, 1, 4, 3], [1, 3, 1, 4]]
```

```
>>> from sage.all import *
>>> Necklaces([Integer(2),Integer(1),Integer(1)])
Necklaces with evaluation [2, 1, 1]
>>> Necklaces([Integer(2),Integer(1),Integer(1)]).cardinality()
3
>>> Necklaces([Integer(2),Integer(1),Integer(1)]).first()
[1, 1, 2, 3]
>>> Necklaces([Integer(2),Integer(1),Integer(1)]).last()
[1, 2, 1, 3]
>>> Necklaces([Integer(2),Integer(1),Integer(1)]).list()
[[1, 1, 2, 3], [1, 1, 3, 2], [1, 2, 1, 3]]
>>> Necklaces([Integer(0),Integer(2),Integer(1),Integer(1)]).list()
[[2, 2, 3, 4], [2, 2, 4, 3], [2, 3, 2, 4]]
>>> Necklaces([Integer(2),Integer(0),Integer(1),Integer(1)]).list()
[[1, 1, 3, 4], [1, 1, 4, 3], [1, 3, 1, 4]]
```

class sage.combinat.necklace.**Necklaces_evaluation**(*content*)

Bases: UniqueRepresentation, Parent

Necklaces with a fixed evaluation (content).

INPUT:

- `content` – list or tuple of nonnegative integers

cardinality()

Return the number of integer necklaces with the evaluation `content`.

The formula for the number of necklaces of content α a composition of n is:

$$\sum_{d|gcd(\alpha)} \phi(d) \binom{n/d}{\alpha_1/d, \ldots, \alpha_\ell/d},$$

where $\phi(d)$ is the Euler ϕ function.

EXAMPLES:

```
sage: Necklaces([]).cardinality()
0
sage: Necklaces([2,2]).cardinality()
```

(continues on next page)

(continued from previous page)

```
2
sage: Necklaces([2,3,2]).cardinality()
30
sage: Necklaces([0,3,2]).cardinality()
2
```

```
>>> from sage.all import *
>>> Necklaces([]).cardinality()
0
>>> Necklaces([Integer(2),Integer(2)]).cardinality()
2
>>> Necklaces([Integer(2),Integer(3),Integer(2)]).cardinality()
30
>>> Necklaces([Integer(0),Integer(3),Integer(2)]).cardinality()
2
```

Check to make sure that the count matches up with the number of necklace words generated.

```
sage: comps = [[],[2,2],[3,2,7],[4,2],[0,4,2],[2,0,4]] + Compositions(4).
↪list()
sage: ns = [Necklaces(comp) for comp in comps]
sage: all(n.cardinality() == len(n.list()) for n in ns)                    #␣
↪needs sage.libs.pari
True
```

```
>>> from sage.all import *
>>> comps = [[],[Integer(2),Integer(2)],[Integer(3),Integer(2),Integer(7)],
↪[Integer(4),Integer(2)],[Integer(0),Integer(4),Integer(2)],[Integer(2),
↪Integer(0),Integer(4)]] + Compositions(Integer(4)).list()
>>> ns = [Necklaces(comp) for comp in comps]
>>> all(n.cardinality() == len(n.list()) for n in ns)                    #␣
↪needs sage.libs.pari
True
```

content()

Return the content (or evaluation) of the necklaces.

EXAMPLES:

```
sage: N = Necklaces([2,2,2])
sage: N.content()
[2, 2, 2]
```

```
>>> from sage.all import *
>>> N = Necklaces([Integer(2),Integer(2),Integer(2)])
>>> N.content()
[2, 2, 2]
```

5.1.152 Non-Decreasing Parking Functions

A *non-decreasing parking function* of size n is a non-decreasing function f from $\{1, \ldots, n\}$ to itself such that for all i, one has $f(i) \leq i$.

The number of non-decreasing parking functions of size n is the n-th `Catalan number`.

The set of non-decreasing parking functions of size n is in bijection with the set of `Dyck words` of size n.

AUTHORS:

- Florent Hivert (2009-04)
- Christian Stump (2012-11) added pretty printing

class sage.combinat.non_decreasing_parking_function.**NonDecreasingParkingFunction**(*lst*)

Bases: `Element`

A *non decreasing parking function* of size n is a non-decreasing function f from $\{1, \ldots, n\}$ to itself such that for all i, one has $f(i) \leq i$.

EXAMPLES:

```
sage: NonDecreasingParkingFunction([])
[]
sage: NonDecreasingParkingFunction([1])
[1]
sage: NonDecreasingParkingFunction([2])
Traceback (most recent call last):
...
ValueError: [2] is not a non-decreasing parking function
sage: NonDecreasingParkingFunction([1,2])
[1, 2]
sage: NonDecreasingParkingFunction([1,1,2])
[1, 1, 2]
sage: NonDecreasingParkingFunction([1,1,4])
Traceback (most recent call last):
...
ValueError: [1, 1, 4] is not a non-decreasing parking function
```

```
>>> from sage.all import *
>>> NonDecreasingParkingFunction([])
[]
>>> NonDecreasingParkingFunction([Integer(1)])
[1]
>>> NonDecreasingParkingFunction([Integer(2)])
Traceback (most recent call last):
...
ValueError: [2] is not a non-decreasing parking function
>>> NonDecreasingParkingFunction([Integer(1),Integer(2)])
[1, 2]
>>> NonDecreasingParkingFunction([Integer(1),Integer(1),Integer(2)])
[1, 1, 2]
>>> NonDecreasingParkingFunction([Integer(1),Integer(1),Integer(4)])
Traceback (most recent call last):
...
ValueError: [1, 1, 4] is not a non-decreasing parking function
```

classmethod from_dyck_word(*dw*)

Bijection from `Dyck words`. It is the inverse of the bijection `to_dyck_word()`. You can find there the mathematical definition.

EXAMPLES:

```
sage: NonDecreasingParkingFunction.from_dyck_word([])
[]
sage: NonDecreasingParkingFunction.from_dyck_word([1,0])
[1]
sage: NonDecreasingParkingFunction.from_dyck_word([1,1,0,0])
[1, 1]
sage: NonDecreasingParkingFunction.from_dyck_word([1,0,1,0])
[1, 2]
sage: NonDecreasingParkingFunction.from_dyck_word([1,0,1,1,0,1,0,0,1,0])
[1, 2, 2, 3, 5]
```

```
>>> from sage.all import *
>>> NonDecreasingParkingFunction.from_dyck_word([])
[]
>>> NonDecreasingParkingFunction.from_dyck_word([Integer(1),Integer(0)])
[1]
>>> NonDecreasingParkingFunction.from_dyck_word([Integer(1),Integer(1),
↪Integer(0),Integer(0)])
[1, 1]
>>> NonDecreasingParkingFunction.from_dyck_word([Integer(1),Integer(0),
↪Integer(1),Integer(0)])
[1, 2]
>>> NonDecreasingParkingFunction.from_dyck_word([Integer(1),Integer(0),
↪Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(0)])
[1, 2, 2, 3, 5]
```

grade()

Return the length of `self`.

EXAMPLES:

```
sage: ndpf = NonDecreasingParkingFunctions(5)
sage: len(ndpf.random_element())
5
```

```
>>> from sage.all import *
>>> ndpf = NonDecreasingParkingFunctions(Integer(5))
>>> len(ndpf.random_element())
5
```

to_dyck_word()

Implement the bijection to `Dyck words`, which is defined as follows. Take a non decreasing parking function, say [1,1,2,4,5,5], and draw its graph:

```
         ___
        |  . 5
       _|  . 5
```

(continues on next page)

(continued from previous page)

```
  ___|  . . 4
 _|  . . . . 2
|  . . . . . 1
|  . . . . . 1
```

The corresponding Dyck word [1,1,0,1,0,0,1,0,1,1,0,0] is then read off from the sequence of horizontal and vertical steps. The converse bijection is `from_dyck_word()`.

EXAMPLES:

```
sage: NonDecreasingParkingFunction([1,1,2,4,5,5]).to_dyck_word()
[1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0]
sage: NonDecreasingParkingFunction([]).to_dyck_word()
[]
sage: NonDecreasingParkingFunction([1,1,1]).to_dyck_word()
[1, 1, 1, 0, 0, 0]
sage: NonDecreasingParkingFunction([1,2,3]).to_dyck_word()
[1, 0, 1, 0, 1, 0]
sage: NonDecreasingParkingFunction([1,1,3,3,4,6,6]).to_dyck_word()
[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> NonDecreasingParkingFunction([Integer(1),Integer(1),Integer(2),Integer(4),
↪Integer(5),Integer(5)]).to_dyck_word()
[1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0]
>>> NonDecreasingParkingFunction([]).to_dyck_word()
[]
>>> NonDecreasingParkingFunction([Integer(1),Integer(1),Integer(1)]).to_dyck_
↪word()
[1, 1, 1, 0, 0, 0]
>>> NonDecreasingParkingFunction([Integer(1),Integer(2),Integer(3)]).to_dyck_
↪word()
[1, 0, 1, 0, 1, 0]
>>> NonDecreasingParkingFunction([Integer(1),Integer(1),Integer(3),Integer(3),
↪Integer(4),Integer(6),Integer(6)]).to_dyck_word()
[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0]
```

`sage.combinat.non_decreasing_parking_function.`**`NonDecreasingParkingFunctions`**`(`*n=None*`)`

Return the set of Non-Decreasing Parking Functions.

A *non-decreasing parking function* of size n is a non-decreasing function f from $\{1, \dots, n\}$ to itself such that for all i, one has $f(i) \leq i$.

EXAMPLES:

Here are all the-non decreasing parking functions of size 5:

```
sage: NonDecreasingParkingFunctions(3).list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]
```

```
>>> from sage.all import *
>>> NonDecreasingParkingFunctions(Integer(3)).list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]
```

If no size is specified, then NonDecreasingParkingFunctions returns the set of all non-decreasing parking functions.

```
sage: PF = NonDecreasingParkingFunctions(); PF
Non-decreasing parking functions
sage: [] in PF
True
sage: [1] in PF
True
sage: [2] in PF
False
sage: [1,1,3] in PF
True
sage: [1,1,4] in PF
False
```

```
>>> from sage.all import *
>>> PF = NonDecreasingParkingFunctions(); PF
Non-decreasing parking functions
>>> [] in PF
True
>>> [Integer(1)] in PF
True
>>> [Integer(2)] in PF
False
>>> [Integer(1),Integer(1),Integer(3)] in PF
True
>>> [Integer(1),Integer(1),Integer(4)] in PF
False
```

If the size n is specified, then NonDecreasingParkingFunctions returns the set of all non-decreasing parking functions of size n.

```
sage: PF = NonDecreasingParkingFunctions(0)
sage: PF.list()
[[]]
sage: PF = NonDecreasingParkingFunctions(1)
sage: PF.list()
[[1]]
sage: PF = NonDecreasingParkingFunctions(3)
sage: PF.list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]

sage: PF3 = NonDecreasingParkingFunctions(3); PF3
Non-decreasing parking functions of size 3
sage: [] in PF3
False
sage: [1] in PF3
False
sage: [1,1,3] in PF3
True
sage: [1,1,4] in PF3
False
```

```
>>> from sage.all import *
>>> PF = NonDecreasingParkingFunctions(Integer(0))
>>> PF.list()
[[]]
>>> PF = NonDecreasingParkingFunctions(Integer(1))
>>> PF.list()
[[1]]
>>> PF = NonDecreasingParkingFunctions(Integer(3))
>>> PF.list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]

>>> PF3 = NonDecreasingParkingFunctions(Integer(3)); PF3
Non-decreasing parking functions of size 3
>>> [] in PF3
False
>>> [Integer(1)] in PF3
False
>>> [Integer(1),Integer(1),Integer(3)] in PF3
True
>>> [Integer(1),Integer(1),Integer(4)] in PF3
False
```

class sage.combinat.non_decreasing_parking_function.**NonDecreasingParkingFunctions_all**

Bases: UniqueRepresentation, Parent

graded_component(*n*)

Return the graded component.

EXAMPLES:

```
sage: P = NonDecreasingParkingFunctions()
sage: P.graded_component(4) == NonDecreasingParkingFunctions(4)
True
```

```
>>> from sage.all import *
>>> P = NonDecreasingParkingFunctions()
>>> P.graded_component(Integer(4)) ==␣
↪NonDecreasingParkingFunctions(Integer(4))
True
```

class sage.combinat.non_decreasing_parking_function.**NonDecreasingParkingFunctions_n**(*n*)

Bases: UniqueRepresentation, Parent

The combinatorial class of non-decreasing parking functions of size n.

A *non-decreasing parking function* of size n is a non-decreasing function f from $\{1, \dots, n\}$ to itself such that for all i, one has $f(i) \leq i$.

The number of non-decreasing parking functions of size n is the n-th Catalan number.

EXAMPLES:

```
sage: PF = NonDecreasingParkingFunctions(3)
sage: PF.list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]
```

(continues on next page)

(continued from previous page)

```
sage: PF = NonDecreasingParkingFunctions(4)
sage: PF.list()
[[1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 1, 3], [1, 1, 1, 4], [1, 1, 2, 2], [1, 1, 2,␣
↪3], [1, 1, 2, 4], [1, 1, 3, 3], [1, 1, 3, 4], [1, 2, 2, 2], [1, 2, 2, 3], [1, 2,
↪ 2, 4], [1, 2, 3, 3], [1, 2, 3, 4]]
sage: [ NonDecreasingParkingFunctions(i).cardinality() for i in range(10)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> from sage.all import *
>>> PF = NonDecreasingParkingFunctions(Integer(3))
>>> PF.list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]
>>> PF = NonDecreasingParkingFunctions(Integer(4))
>>> PF.list()
[[1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 1, 3], [1, 1, 1, 4], [1, 1, 2, 2], [1, 1, 2,␣
↪3], [1, 1, 2, 4], [1, 1, 3, 3], [1, 1, 3, 4], [1, 2, 2, 2], [1, 2, 2, 3], [1, 2,
↪ 2, 4], [1, 2, 3, 3], [1, 2, 3, 4]]
>>> [ NonDecreasingParkingFunctions(i).cardinality() for i in range(Integer(10))]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

Warning

The precise order in which the parking function are generated or listed is not fixed, and may change in the future.

AUTHORS:

- Florent Hivert

Element

alias of *NonDecreasingParkingFunction*

cardinality()

Return the number of non-decreasing parking functions of size n.

This number is the n-th *Catalan number*.

EXAMPLES:

```
sage: PF = NonDecreasingParkingFunctions(0)
sage: PF.cardinality()
1
sage: PF = NonDecreasingParkingFunctions(1)
sage: PF.cardinality()
1
sage: PF = NonDecreasingParkingFunctions(3)
sage: PF.cardinality()
5
sage: PF = NonDecreasingParkingFunctions(5)
sage: PF.cardinality()
42
```

```
>>> from sage.all import *
>>> PF = NonDecreasingParkingFunctions(Integer(0))
>>> PF.cardinality()
1
>>> PF = NonDecreasingParkingFunctions(Integer(1))
>>> PF.cardinality()
1
>>> PF = NonDecreasingParkingFunctions(Integer(3))
>>> PF.cardinality()
5
>>> PF = NonDecreasingParkingFunctions(Integer(5))
>>> PF.cardinality()
42
```

one()

Return the unit of this monoid.

This is the non-decreasing parking function [1, 2, …, n].

EXAMPLES:

```
sage: ndpf = NonDecreasingParkingFunctions(5)
sage: x = ndpf.random_element(); x  # random
sage: e = ndpf.one()
sage: x == e*x == x*e
True
```

```
>>> from sage.all import *
>>> ndpf = NonDecreasingParkingFunctions(Integer(5))
>>> x = ndpf.random_element(); x  # random
>>> e = ndpf.one()
>>> x == e*x == x*e
True
```

random_element()

Return a random parking function of the given size.

EXAMPLES:

```
sage: ndpf = NonDecreasingParkingFunctions(5)
sage: x = ndpf.random_element(); x  # random
[1, 2, 2, 4, 5]
sage: x in ndpf
True
```

```
>>> from sage.all import *
>>> ndpf = NonDecreasingParkingFunctions(Integer(5))
>>> x = ndpf.random_element(); x  # random
[1, 2, 2, 4, 5]
>>> x in ndpf
True
```

sage.combinat.non_decreasing_parking_function.**is_a**(*x*, *n=None*)

Check whether a list is a non-decreasing parking function.

If a size n is specified, checks if a list is a non-decreasing parking function of size n.

5.1.153 ν-Dyck Words

A class of the ν-Dyck word, see [PRV2017] for details.

AUTHORS:

- Aram Dermenjian (2020-09-26)

This file is based off the class `DyckWords` written by Mike Hansen, Dan Drake, Florent Hivert, Christian Stump, Mike Zabrocki, Jean–Baptiste Priez and Travis Scrimshaw

class `sage.combinat.nu_dyck_word.`**`NuDyckWord`**(*parent*, *dw*, *latex_options=None*)

Bases: `CombinatorialElement`

A ν-Dyck word.

Given a lattice path ν in the $\mathbf{Z}^2$ grid starting at the origin $(0,0)$ consisting of North $N = (0,1)$ and East $E = (1,0)$ steps, a ν-Dyck path is a lattice path in the $\mathbf{Z}^2$ grid starting at the origin $(0,0)$ and ending at the same coordinate as ν such that it is weakly above ν. A ν-Dyck word is the representation of a ν-Dyck path where a North step is represented by a 1 and an East step is represented by a 0.

INPUT:

- `k1` – a path for the ν-Dyck word
- `k2` – a path for ν

EXAMPLES:

```
sage: dw = NuDyckWord([1,0,1,0],[1,0,0,1]); dw
[1, 0, 1, 0]
sage: print(dw)
NENE
sage: dw.height()
2

sage: dw = NuDyckWord('1010',[1,0,0,1]); dw
[1, 0, 1, 0]

sage: dw = NuDyckWord('NENE',[1,0,0,1]); dw
[1, 0, 1, 0]

sage: NuDyckWord([1,0,1,0],[1,0,0,1]).pretty_print()
   __
 _|x
| . .

sage: from sage.combinat.nu_dyck_word import update_ndw_symbols
sage: update_ndw_symbols(0,1)
sage: dw = NuDyckWord('0101001','0110010'); dw
[0, 1, 0, 1, 0, 0, 1]
sage: dw.pp()
     __
    |x
   _| .
 _|x  .
```

(continues on next page)

(continued from previous page)

```
| . . .
sage: update_ndw_symbols(1,0)
```

```
>>> from sage.all import *
>>> dw = NuDyckWord([Integer(1),Integer(0),Integer(1),Integer(0)],[Integer(1),
↪Integer(0),Integer(0),Integer(1)]); dw
[1, 0, 1, 0]
>>> print(dw)
NENE
>>> dw.height()
2

>>> dw = NuDyckWord('1010',[Integer(1),Integer(0),Integer(0),Integer(1)]); dw
[1, 0, 1, 0]

>>> dw = NuDyckWord('NENE',[Integer(1),Integer(0),Integer(0),Integer(1)]); dw
[1, 0, 1, 0]

>>> NuDyckWord([Integer(1),Integer(0),Integer(1),Integer(0)],[Integer(1),
↪Integer(0),Integer(0),Integer(1)]).pretty_print()
   __
 _|x
| . .

>>> from sage.combinat.nu_dyck_word import update_ndw_symbols
>>> update_ndw_symbols(Integer(0),Integer(1))
>>> dw = NuDyckWord('0101001','0110010'); dw
[0, 1, 0, 1, 0, 0, 1]
>>> dw.pp()
     __
    |x
   _| .
 _|x  .
| . . .
>>> update_ndw_symbols(Integer(1),Integer(0))
```

can_mutate(i)

Return True/False based off if mutable at height i.

Can only mutate if an east step is followed by a north step at height i.

OUTPUT: whether we can mutate at height of i

EXAMPLES:

```
sage: NDW = NuDyckWord('10010100','00000111')
sage: NDW.can_mutate(1)
False
sage: NDW.can_mutate(3)
5
```

```
>>> from sage.all import *
>>> NDW = NuDyckWord('10010100','00000111')
```

(continues on next page)

(continued from previous page)

```
>>> NDW.can_mutate(Integer(1))
False
>>> NDW.can_mutate(Integer(3))
5
```

height()

Return the height of `self`.

The height is the number of `north` steps.

EXAMPLES:

```
sage: NuDyckWord('1101110011010010001101111000110000',
....: '1010101010101010101010101010101010').height()
17
```

```
>>> from sage.all import *
>>> NuDyckWord('1101110011010010001101111000110000',
... '1010101010101010101010101010101010').height()
17
```

heights()

Return the heights of each point on `self`.

We view the Dyck word as a Dyck path from $(0, 0)$ to (x, y) in the first quadrant by letting `1`'s represent steps in the direction $(0, 1)$ and `0`'s represent steps in the direction $(1, 0)$.

The heights is the sequence of the y-coordinates of all $x + y$ lattice points along the path.

EXAMPLES:

```
sage: NuDyckWord('010','010').heights()
[0, 0, 1, 1]
sage: NuDyckWord('110100','101010').heights()
[0, 1, 2, 2, 3, 3, 3]
```

```
>>> from sage.all import *
>>> NuDyckWord('010','010').heights()
[0, 0, 1, 1]
>>> NuDyckWord('110100','101010').heights()
[0, 1, 2, 2, 3, 3, 3]
```

horizontal_distance()

Return a list of how far each point is from ν.

EXAMPLES:

```
sage: NDW = NuDyckWord('10010100','00000111')
sage: NDW.horizontal_distance()
[5, 5, 4, 3, 3, 2, 2, 1, 0]
sage: NDW = NuDyckWord('10010100','00001011')
sage: NDW.horizontal_distance()
[4, 5, 4, 3, 3, 2, 2, 1, 0]
sage: NDW = NuDyckWord('10011001000','00100101001')
```

(continues on next page)

(continued from previous page)

```
sage: NDW.horizontal_distance()
[2, 4, 3, 2, 3, 5, 4, 3, 3, 2, 1, 0]
```

```
>>> from sage.all import *
>>> NDW = NuDyckWord('10010100','00000111')
>>> NDW.horizontal_distance()
[5, 5, 4, 3, 3, 2, 2, 1, 0]
>>> NDW = NuDyckWord('10010100','00001011')
>>> NDW.horizontal_distance()
[4, 5, 4, 3, 3, 2, 2, 1, 0]
>>> NDW = NuDyckWord('10011001000','00100101001')
>>> NDW.horizontal_distance()
[2, 4, 3, 2, 3, 5, 4, 3, 3, 2, 1, 0]
```

latex_options()

Return the latex options for use in the `_latex_` function as a dictionary.

The default values are set using the options.

- `color` – (default: black) the line color
- `line width` – (default: 2*``tikz_scale``) value representing the line width
- `nu_options` – (default: `'rounded corners=1, color=red, line width=1'`) string to indicate what the tikz options should be for path of ν
- `points_color` – (default: `'black'`) str to indicate color points should be drawn with
- `show_grid` – boolean (default: `True`); value to indicate if grid should be shown
- `show_nu` – boolean (default: `True`); value to indicate if ν should be shown
- `show_points` – boolean (default: `False`); value to indicate if points should be shown on path
- `tikz_scale` – (default: 1) scale for use with the tikz package

EXAMPLES:

```
sage: NDW = NuDyckWord('010','010')
sage: NDW.latex_options()
{'color': black,
 'line width': 2,
 'nu_options': rounded corners=1, color=red, line width=1,
 'points_color': black,
 'show_grid': True,
 'show_nu': True,
 'show_points': False,
 'tikz_scale': 1}
```

```
>>> from sage.all import *
>>> NDW = NuDyckWord('010','010')
>>> NDW.latex_options()
{'color': black,
 'line width': 2,
 'nu_options': rounded corners=1, color=red, line width=1,
 'points_color': black,
```

(continues on next page)

(continued from previous page)

```
'show_grid': True,
'show_nu': True,
'show_points': False,
'tikz_scale': 1}
```

Todo

This should probably be merged into NuDyckWord.options.

length()

Return the length of `self`.

The length is the total number of steps.

EXAMPLES:

```
sage: NDW = NuDyckWord('10011001000','00100101001')
sage: NDW.length()
11
```

```
>>> from sage.all import *
>>> NDW = NuDyckWord('10011001000','00100101001')
>>> NDW.length()
11
```

mutate(i)

Return a new ν-Dyck Word if possible.

If at height i we have an east step E meeting a north step N then we calculate all horizontal distances from this point until we find the first point that has the same horizontal distance to ν. We let

- d is everything up until EN (not including EN)
- f be everything between N and the point with the same horizontal distance (including N)
- g is everything after f

See also

can_mutate()

EXAMPLES:

```
sage: NDW = NuDyckWord('10010100','00000111')
sage: NDW.mutate(1)
sage: NDW.mutate(3)
[1, 0, 0, 1, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> NDW = NuDyckWord('10010100','00000111')
>>> NDW.mutate(Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> NDW.mutate(Integer(3))
[1, 0, 0, 1, 1, 0, 0, 0]
```

path()

Return the underlying path object.

EXAMPLES:

```
sage: NDW = NuDyckWord('10011001000','00100101001')
sage: NDW.path()
Path: 10011001000
```

```
>>> from sage.all import *
>>> NDW = NuDyckWord('10011001000','00100101001')
>>> NDW.path()
Path: 10011001000
```

plot(***kwds*)

Plot a ν-Dyck word as a continuous path.

EXAMPLES:

```
sage: NDW = NuDyckWord('010','010')
sage: NDW.plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> NDW = NuDyckWord('010','010')
>>> NDW.plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

points()

Return an iterator for the points on the ν-Dyck path.

EXAMPLES:

```
sage: list(NuDyckWord('110111001101001000110111100011000',
....: '101010101010101010101010101010101')._path.points())
[(0, 0),
 (0, 1),
 (0, 2),
 (1, 2),
 (1, 3),
 (1, 4),
 (1, 5),
 (2, 5),
 (3, 5),
 (3, 6),
 (3, 7),
 (4, 7),
 (4, 8),
```

(continues on next page)

(continued from previous page)

```
 (5, 8),
 (6, 8),
 (6, 9),
 (7, 9),
 (8, 9),
 (9, 9),
 (9, 10),
 (9, 11),
 (10, 11),
 (10, 12),
 (10, 13),
 (10, 14),
 (10, 15),
 (11, 15),
 (12, 15),
 (13, 15),
 (13, 16),
 (13, 17),
 (14, 17),
 (15, 17),
 (16, 17)]
```

```
>>> from sage.all import *
>>> list(NuDyckWord('110111001101001000110111100011000',
... '101010101010101010101010101010101')._path.points())
[(0, 0),
 (0, 1),
 (0, 2),
 (1, 2),
 (1, 3),
 (1, 4),
 (1, 5),
 (2, 5),
 (3, 5),
 (3, 6),
 (3, 7),
 (4, 7),
 (4, 8),
 (5, 8),
 (6, 8),
 (6, 9),
 (7, 9),
 (8, 9),
 (9, 9),
 (9, 10),
 (9, 11),
 (10, 11),
 (10, 12),
 (10, 13),
 (10, 14),
 (10, 15),
 (11, 15),
```

(continues on next page)

(continued from previous page)

```
 (12, 15),
 (13, 15),
 (13, 16),
 (13, 17),
 (14, 17),
 (15, 17),
 (16, 17)]
```

pp (*style=None*, *labelling=None*)

Display a NuDyckWord as a lattice path in the $\mathbf{Z}^2$ grid.

If the `style` is "N-E", then a cell below the diagonal is indicated by a period, whereas a cell below the path but above the diagonal is indicated by an x. If a list of labels is included, they are displayed along the vertical edges of the Dyck path.

INPUT:

- `style` – (default: `None`) can either be:
  - `None` to use the option default
  - "N-E" to show `self` as a path of north and east steps, or
- `labelling` – (if style is "N-E") a list of labels assigned to the up steps in `self`
- `underpath` – (if style is "N-E", default: `True`) if `True`, an `x` to show the boxes between ν and the ν-Dyck Path

EXAMPLES:

```
sage: for ND in NuDyckWords('101010'): ND.pretty_print()
     __
   _| .
 _| . .
| . . .
     __
 ___| .
|x  . .
| . . .
   ____
  |x  .
 _| . .
| . . .
   ____
 _|x  .
|x  . .
| . . .
 ______
|x x  .
|x  . .
| . . .
```

```
>>> from sage.all import *
>>> for ND in NuDyckWords('101010'): ND.pretty_print()
     __
   _| .
```

(continues on next page)

(continued from previous page)

```
 _| . .
| . . .
    __
 ___| .
|x  . .
| . . .
  _____
  |x  .
 _| . .
| . . .
  _____
 _|x  .
|x  . .
| . . .
 _______
|x x  .
|x  . .
| . . .
```

```
sage: nu = [1,0,1,0,1,0,1,0,1,0,1,0]
sage: ND = NuDyckWord([1,1,1,0,1,0,0,1,1,0,0,0],nu)
sage: ND.pretty_print()
       _______
      |x x  .
   ___|x  . .
 _|x x  . . .
|x x  . . . .
|x  . . . . .
| . . . . . .
```

```
>>> from sage.all import *
>>> nu = [Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0)]
>>> ND = NuDyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0)],nu)
>>> ND.pretty_print()
       _______
      |x x  .
   ___|x  . .
 _|x x  . . .
|x x  . . . .
|x  . . . . .
| . . . . . .
```

```
sage: NuDyckWord([1,1,0,0,1,0],[1,0,1,0,1,0]).pretty_print(
....: labelling=[1,3,2])
    __
 ___| . 2
|x  . . 3
| . . . 1
```

```
>>> from sage.all import *
>>> NuDyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)]).pretty_print(
... labelling=[Integer(1),Integer(3),Integer(2)])
     __
 ___| . 2
|x  . . 3
| . . . 1
```

```
sage: NuDyckWord('1101110011010010001101111000110000',
....: '1010101010101010101010101010101010').pretty_print(
....: labelling=list(range(1,18)))
                           ________
                          |x x x  . 17
                     _____|x x  . . 16
                    |x x x x  . . . 15
                    |x x x  . . . . 14
                    |x x  . . . . . 13
                   _|x  . . . . . . 12
                  |x  . . . . . . . 11
             _____| . . . . . . . . 10
         ___|x x  . . . . . . . . .  9
       _|x x x  . . . . . . . . . .  8
      |x x x  . . . . . . . . . . .  7
   ___|x x  . . . . . . . . . . . .  6
  |x x x  . . . . . . . . . . . . .  5
  |x x  . . . . . . . . . . . . . .  4
 _|x  . . . . . . . . . . . . . . .  3
|x  . . . . . . . . . . . . . . . .  2
| . . . . . . . . . . . . . . . . .  1
```

```
>>> from sage.all import *
>>> NuDyckWord('1101110011010010001101111000110000',
... '1010101010101010101010101010101010').pretty_print(
... labelling=list(range(Integer(1),Integer(18))))
                           ________
                          |x x x  . 17
                     _____|x x  . . 16
                    |x x x x  . . . 15
                    |x x x  . . . . 14
                    |x x  . . . . . 13
                   _|x  . . . . . . 12
                  |x  . . . . . . . 11
             _____| . . . . . . . . 10
         ___|x x  . . . . . . . . .  9
       _|x x x  . . . . . . . . . .  8
      |x x x  . . . . . . . . . . .  7
   ___|x x  . . . . . . . . . . . .  6
  |x x x  . . . . . . . . . . . . .  5
  |x x  . . . . . . . . . . . . . .  4
 _|x  . . . . . . . . . . . . . . .  3
```

(continues on next page)

(continued from previous page)

```
|x  . . . . . . . . . . . . . . . .  2
| . . . . . . . . . . . . . . . . .  1
```

```
sage: NuDyckWord().pretty_print()
.
```

```
>>> from sage.all import *
>>> NuDyckWord().pretty_print()
.
```

pretty_print(*style=None*, *labelling=None*)

Display a NuDyckWord as a lattice path in the $\mathbf{Z}^2$ grid.

If the `style` is "N-E", then a cell below the diagonal is indicated by a period, whereas a cell below the path but above the diagonal is indicated by an x. If a list of labels is included, they are displayed along the vertical edges of the Dyck path.

INPUT:

- `style` – (default: `None`) can either be:
  - `None` to use the option default
  - "N-E" to show `self` as a path of north and east steps, or
- `labelling` – (if style is "N-E") a list of labels assigned to the up steps in `self`
- `underpath` – (if style is "N-E", default: `True`) if `True`, an `x` to show the boxes between ν and the ν-Dyck Path

EXAMPLES:

```
sage: for ND in NuDyckWords('101010'): ND.pretty_print()
     __
   _| .
 _| . .
| . . .
     __
 ___| .
|x  . .
| . . .
   ____
  |x  .
 _| . .
| . . .
   ____
 _|x  .
|x  . .
| . . .
 ______
|x x  .
|x  . .
| . . .
```

```
>>> from sage.all import *
>>> for ND in NuDyckWords('101010'): ND.pretty_print()
     __
   _| .
 _| . .
| . . .
     __
 ___| .
|x  . .
| . . .
   ____
  |x  .
 _| . .
| . . .
   ____
 _|x  .
|x  . .
| . . .
 ______
|x x  .
|x  . .
| . . .
```

```
sage: nu = [1,0,1,0,1,0,1,0,1,0,1,0]
sage: ND = NuDyckWord([1,1,1,0,1,0,0,1,1,0,0,0],nu)
sage: ND.pretty_print()
       ______
      |x x  .
   ___|x  . .
 _|x x  . . .
|x x  . . . .
|x  . . . . .
| . . . . . .
```

```
>>> from sage.all import *
>>> nu = [Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0)]
>>> ND = NuDyckWord([Integer(1),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0)],nu)
>>> ND.pretty_print()
       ______
      |x x  .
   ___|x  . .
 _|x x  . . .
|x x  . . . .
|x  . . . . .
| . . . . . .
```

```
sage: NuDyckWord([1,1,0,0,1,0],[1,0,1,0,1,0]).pretty_print(
....: labelling=[1,3,2])
     __
```

(continues on next page)

(continued from previous page)

```
 ___| . 2
|x  . . 3
| . . . 1
```

```
>>> from sage.all import *
>>> NuDyckWord([Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)]).pretty_print(
... labelling=[Integer(1),Integer(3),Integer(2)])
     __
 ___| . 2
|x  . . 3
| . . . 1
```

```
sage: NuDyckWord('1101110011010010001101111000110000',
....: '1010101010101010101010101010101010').pretty_print(
....: labelling=list(range(1,18)))
                           ________
                          |x x x  . 17
                     _____|x x . . 16
                    |x x x x  . . . 15
                    |x x x  . . . . 14
                    |x x  . . . . . 13
                   _|x  . . . . . . 12
                  |x  . . . . . . . 11
             _____| . . . . . . . . 10
         ___|x x  . . . . . . . . .  9
       _|x x x  . . . . . . . . . .  8
      |x x x  . . . . . . . . . . .  7
   ___|x x  . . . . . . . . . . . .  6
  |x x x  . . . . . . . . . . . . .  5
  |x x  . . . . . . . . . . . . . .  4
 _|x  . . . . . . . . . . . . . . .  3
|x  . . . . . . . . . . . . . . . .  2
| . . . . . . . . . . . . . . . . .  1
```

```
>>> from sage.all import *
>>> NuDyckWord('1101110011010010001101111000110000',
... '1010101010101010101010101010101010').pretty_print(
... labelling=list(range(Integer(1),Integer(18))))
                           ________
                          |x x x  . 17
                     _____|x x . . 16
                    |x x x x  . . . 15
                    |x x x  . . . . 14
                    |x x  . . . . . 13
                   _|x  . . . . . . 12
                  |x  . . . . . . . 11
             _____| . . . . . . . . 10
         ___|x x  . . . . . . . . .  9
       _|x x x  . . . . . . . . . .  8
```

(continues on next page)

(continued from previous page)

```
       |x x x  . . . . . . . . . . .  7
    ___|x x  . . . . . . . . . . . .  6
   |x x x  . . . . . . . . . . . . .  5
   |x x  . . . . . . . . . . . . . .  4
 _|x  . . . . . . . . . . . . . . .  3
|x  . . . . . . . . . . . . . . . .  2
| . . . . . . . . . . . . . . . . .  1
```

```
sage: NuDyckWord().pretty_print()
.
```

```
>>> from sage.all import *
>>> NuDyckWord().pretty_print()
.
```

set_latex_options(*D*)

Set the latex options for use in the `_latex_` function.

The default values are set in the `__init__` function.

- `color` – (default: black) the line color
- `line width` – (default: $2\times$ `tikz_scale`) value representing the line width
- `nu_options` – (default: `'rounded corners=1, color=red, line width=1'`) string to indicate what the tikz options should be for path of ν
- `points_color` – (default: `'black'`) str to indicate color points should be drawn with
- `show_grid` – boolean (default: `True`); value to indicate if grid should be shown
- `show_nu` – boolean (default: `True`); value to indicate if ν should be shown
- `show_points` – boolean (default: `False`); value to indicate if points should be shown on path
- `tikz_scale` – (default: 1) scale for use with the tikz package

INPUT:

- `D` – dictionary with a list of latex parameters to change

EXAMPLES:

```
sage: NDW = NuDyckWord('010','010')
sage: NDW.set_latex_options({"tikz_scale":2})
sage: NDW.set_latex_options({"color":"blue", "show_points":True})
```

```
>>> from sage.all import *
>>> NDW = NuDyckWord('010','010')
>>> NDW.set_latex_options({"tikz_scale":Integer(2)})
>>> NDW.set_latex_options({"color":"blue", "show_points":True})
```

Todo

This should probably be merged into NuDyckWord.options.

width()

Return the width of `self`.

The width is the number of `east` steps.

EXAMPLES:

```
sage: NuDyckWord('110111001101001000110111100011000',
....: '101010101010101010101010101010101').width()
16
```

```
>>> from sage.all import *
>>> NuDyckWord('110111001101001000110111100011000',
... '101010101010101010101010101010101').width()
16
```

widths()

Return the widths of each point on `self`.

We view the Dyck word as a Dyck path from $(0,0)$ to (x,y) in the first quadrant by letting `1`'s represent steps in the direction $(0,1)$ and `0`'s represent steps in the direction $(1,0)$.

The widths is the sequence of the x-coordinates of all $x+y$ lattice points along the path.

EXAMPLES:

```
sage: NuDyckWord('010','010').widths()
[0, 1, 1, 2]
sage: NuDyckWord('110100','101010').widths()
[0, 0, 0, 1, 1, 2, 3]
```

```
>>> from sage.all import *
>>> NuDyckWord('010','010').widths()
[0, 1, 1, 2]
>>> NuDyckWord('110100','101010').widths()
[0, 0, 0, 1, 1, 2, 3]
```

class sage.combinat.nu_dyck_word.**NuDyckWords**(*nu=()*)

Bases: `Parent`

ν-Dyck words.

Given a lattice path ν in the $\mathbf{Z}^2$ grid starting at the origin $(0,0)$ consisting of North $N=(0,1)$ and East $E=(1,0)$ steps, a ν-Dyck path is a lattice path in the`ZZ^2` grid starting at the origin $(0,0)$ and ending at the same coordinate as ν such that it is weakly above ν. A ν-Dyck word is the representation of a ν-Dyck path where a North step is represented by a 1 and an East step is represented by a 0.

INPUT:

- `nu` – the base lattice path

EXAMPLES:

```
sage: NDW = NuDyckWords('1010'); NDW
[1, 0, 1, 0] Dyck words
sage: [1,0,1,0] in NDW
True
```

(continues on next page)

(continued from previous page)

```
sage: [1,1,0,0] in NDW
True
sage: [1,0,0,1] in NDW
False
sage: [0,1,0,1] in NDW
False
sage: NDW.cardinality()
2
```

```
>>> from sage.all import *
>>> NDW = NuDyckWords('1010'); NDW
[1, 0, 1, 0] Dyck words
>>> [Integer(1),Integer(0),Integer(1),Integer(0)] in NDW
True
>>> [Integer(1),Integer(1),Integer(0),Integer(0)] in NDW
True
>>> [Integer(1),Integer(0),Integer(0),Integer(1)] in NDW
False
>>> [Integer(0),Integer(1),Integer(0),Integer(1)] in NDW
False
>>> NDW.cardinality()
2
```

Element

alias of *NuDyckWord*

cardinality()

Return the number of ν-Dyck words.

EXAMPLES:

```
sage: NDW = NuDyckWords('101010'); NDW.cardinality()
5
sage: NDW = NuDyckWords('1010010'); NDW.cardinality()
7
sage: NDW = NuDyckWords('100100100'); NDW.cardinality()
12
```

```
>>> from sage.all import *
>>> NDW = NuDyckWords('101010'); NDW.cardinality()
5
>>> NDW = NuDyckWords('1010010'); NDW.cardinality()
7
>>> NDW = NuDyckWords('100100100'); NDW.cardinality()
12
```

options = Current options for NuDyckWords - ascii_art: pretty_output - diagram_style: grid - display: list - latex_color: black - latex_line_width_scalar: 2 - latex_nu_options: rounded corners=1, color=red, line width=1 - latex_points_color: black - latex_show_grid: True - latex_show_nu: True - latex_show_points: False - latex_tikz_scale: 1

sage.combinat.nu_dyck_word.**path_weakly_above_other**(*path*, *other*)

Test if `path` is weakly above `other`.

A path P is wealy above another path Q if P and Q are the same length and if any prefix of length n of Q contains more North steps than the prefix of length n of P.

INPUT:

- `path` – the path to verify is weakly above the other path
- `other` – the other path to verify is weakly below the path

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.combinat.nu_dyck_word import path_weakly_above_other
sage: path_weakly_above_other('1001','0110')
False
sage: path_weakly_above_other('1001','0101')
True
sage: path_weakly_above_other('1111','0101')
False
sage: path_weakly_above_other('111100','0101')
False
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_dyck_word import path_weakly_above_other
>>> path_weakly_above_other('1001','0110')
False
>>> path_weakly_above_other('1001','0101')
True
>>> path_weakly_above_other('1111','0101')
False
>>> path_weakly_above_other('111100','0101')
False
```

sage.combinat.nu_dyck_word.**replace_dyck_char**(*x*)

A map sending an opening character (`'1'`, `'N'`, and `'('`) to `ndw_open_symbol`, a closing character (`'0'`, `'E'`, and `')'`) to `ndw_close_symbol`, and raising an error on any input other than one of the opening or closing characters.

This is the inverse map of `replace_dyck_symbol()`.

INPUT:

- `x` – string; a `'1'`, `'0'`, `'N'`, `'E'`, `'('` or `')'`

OUTPUT:

- If `x` is an opening character, replace `x` with the constant `ndw_open_symbol`.
- If `x` is a closing character, replace `x` with the constant `ndw_close_symbol`.
- Raise a `ValueError` if `x` is neither an opening nor a closing character.

See also

`replace_dyck_symbol()`

EXAMPLES:

```
sage: from sage.combinat.nu_dyck_word import replace_dyck_char
sage: replace_dyck_char('(')
1
sage: replace_dyck_char(')')
0
sage: replace_dyck_char(1)
Traceback (most recent call last):
...
ValueError
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_dyck_word import replace_dyck_char
>>> replace_dyck_char('(')
1
>>> replace_dyck_char(')')
0
>>> replace_dyck_char(Integer(1))
Traceback (most recent call last):
...
ValueError
```

sage.combinat.nu_dyck_word.**replace_dyck_symbol**(*x*, *open_char='N'*, *close_char='E'*)

A map sending `ndw_open_symbol` to `open_char` and `ndw_close_symbol` to `close_char`, and raising an error on any input other than `ndw_open_symbol` and `ndw_close_symbol`. The values of the constants `ndw_open_symbol` and `ndw_close_symbol` are subject to change.

This is the inverse map of *replace_dyck_char()*.

INPUT:

- `x` – either `ndw_open_symbol` or `ndw_close_symbol`
- `open_char` – string (optional) default `'N'`
- `close_char` – string (optional) default `'E'`

OUTPUT: if `x` is `ndw_open_symbol`, replace `x` with `open_char`

- If `x` is `ndw_close_symbol`, replace `x` with `close_char`.
- If `x` is neither `ndw_open_symbol` nor `ndw_close_symbol`, a `ValueError` is raised.

See also

replace_dyck_char()

EXAMPLES:

```
sage: from sage.combinat.nu_dyck_word import replace_dyck_symbol
sage: replace_dyck_symbol(1)
'N'
sage: replace_dyck_symbol(0)
'E'
sage: replace_dyck_symbol(3)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_dyck_word import replace_dyck_symbol
>>> replace_dyck_symbol(Integer(1))
'N'
>>> replace_dyck_symbol(Integer(0))
'E'
>>> replace_dyck_symbol(Integer(3))
Traceback (most recent call last):
...
ValueError
```

sage.combinat.nu_dyck_word.**to_word_path**(*word*)

Convert input into a word path over an appropriate alphabet.

Helper function.

INPUT:

- `word` – word to convert to wordpath

OUTPUT: a `FiniteWordPath_north_east` object

EXAMPLES:

```
sage: from sage.combinat.nu_dyck_word import to_word_path
sage: wp = to_word_path('NEENENEN'); wp
Path: 10010101
sage: from sage.combinat.words.paths import FiniteWordPath_north_east
sage: isinstance(wp,FiniteWordPath_north_east)
True
sage: to_word_path('1001')
Path: 1001
sage: to_word_path([0,1,0,0,1,0])
Path: 010010
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_dyck_word import to_word_path
>>> wp = to_word_path('NEENENEN'); wp
Path: 10010101
>>> from sage.combinat.words.paths import FiniteWordPath_north_east
>>> isinstance(wp,FiniteWordPath_north_east)
True
>>> to_word_path('1001')
Path: 1001
>>> to_word_path([Integer(0),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)])
Path: 010010
```

sage.combinat.nu_dyck_word.**update_ndw_symbols**(*os*, *cs*)

A way to alter the open and close symbols from sage.

INPUT:

- `os` – the open symbol
- `cs` – the close symbol

EXAMPLES:

```
sage: from sage.combinat.nu_dyck_word import update_ndw_symbols
sage: update_ndw_symbols(0,1)
sage: dw = NuDyckWord('0101001','0110010'); dw
[0, 1, 0, 1, 0, 0, 1]

sage: dw = NuDyckWord('1010110','1001101'); dw
Traceback (most recent call last):
...
ValueError: invalid nu-Dyck word
sage: update_ndw_symbols(1,0)
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_dyck_word import update_ndw_symbols
>>> update_ndw_symbols(Integer(0),Integer(1))
>>> dw = NuDyckWord('0101001','0110010'); dw
[0, 1, 0, 1, 0, 0, 1]

>>> dw = NuDyckWord('1010110','1001101'); dw
Traceback (most recent call last):
...
ValueError: invalid nu-Dyck word
>>> update_ndw_symbols(Integer(1),Integer(0))
```

5.1.154 ν-Tamari lattice

A class of the ν-Tamari lattice, and (δ, ν)-Tamari lattice (or alt ν-Tamari lattices), see [PRV2017] and [CC2023] for details.

These lattices depend on a parameter ν where ν is a path of North and East steps. The alt ν-Tamari lattice depends on an additional parameter δ, which is an increment vector with respect to ν.

The elements are *nu-Dyck paths* which are weakly above ν.

To use the provided functionality, you should import ν-Tamari lattices by typing:

```
sage: from sage.combinat.nu_tamari_lattice import NuTamariLattice, AltNuTamariLattice
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_tamari_lattice import NuTamariLattice, AltNuTamariLattice
```

Then,

```
sage: NuTamariLattice([1,1,1,0,0,1,1,0])
Finite lattice containing 6 elements
sage: NuTamariLattice([0,0,0,1,1,0,0,1])
Finite lattice containing 40 elements
sage: AltNuTamariLattice([0,0,0,1,1,0,0,1])
Finite lattice containing 40 elements
```

(continues on next page)

(continued from previous page)

```
sage: AltNuTamariLattice([0,0,0,1,1,0,0,1], [0,1,0])
Finite lattice containing 40 elements
```

```
>>> from sage.all import *
>>> NuTamariLattice([Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(1),Integer(0)])
Finite lattice containing 6 elements
>>> NuTamariLattice([Integer(0),Integer(0),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(1)])
Finite lattice containing 40 elements
>>> AltNuTamariLattice([Integer(0),Integer(0),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(1)])
Finite lattice containing 40 elements
>>> AltNuTamariLattice([Integer(0),Integer(0),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(1)], [Integer(0),Integer(1),Integer(0)])
Finite lattice containing 40 elements
```

The classical **Tamari lattices** and the **Generalized Tamari lattices** are special cases of this construction and are also available with this poset:

```
sage: NuTamariLattice([1,0,1,0,1,0])
Finite lattice containing 5 elements

sage: NuTamariLattice([1,0,0,1,0,0,1,0,0])
Finite lattice containing 12 elements
```

```
>>> from sage.all import *
>>> NuTamariLattice([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)])
Finite lattice containing 5 elements

>>> NuTamariLattice([Integer(1),Integer(0),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(0),Integer(0)])
Finite lattice containing 12 elements
```

See also

For more detailed information see `NuTamariLattice()` and `AltNuTamariLattice()`. For more information on the standard Tamari lattice see `sage.combinat.tamari_lattices.TamariLattice()`, `sage.combinat.tamari_lattices.GeneralizedTamariLattice()`

AUTHORS:

- Aram Dermenjian (2020-09-26): initial version
- Clément Chenevière (2024-02-01): added the alt ν-Tamari lattices

`sage.combinat.nu_tamari_lattice.`**`AltNuTamariLattice`**`(`*nu*, *delta=None*`)`

Return the (δ, ν)-Tamari lattice (or alt ν-Tamari lattice).

For more information, see [CC2023].

The path ν is a path of North steps (represented as 1 s) and East steps (represented as 0 s).

The vector $\delta = (\delta_1, \ldots, \delta_n)$ is an increment vector with respect to the path ν, that is to say $\delta_i \leq \nu_i$, where ν_i is the number of 0 s following the i-th 1 of ν. If not provided, δ is set by default to produce the classical ν-Tamari lattice.

INPUT:

- ν – list of 0s and 1s or a string of 0s and 1s
- δ – list of nonnegative integers

OUTPUT: a finite lattice

EXAMPLES:

```
sage: from sage.combinat.nu_tamari_lattice import AltNuTamariLattice,␣
↪NuTamariLattice
sage: AltNuTamariLattice('01001', [0, 0])
Finite lattice containing 7 elements
sage: AltNuTamariLattice('01001', [1, 0])
Finite lattice containing 7 elements
sage: AltNuTamariLattice('01001') == AltNuTamariLattice('01001', [2, 0])
True
sage: nu = '00100100101'; P = AltNuTamariLattice(nu); Q = NuTamariLattice(nu); P␣
↪== Q
True
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_tamari_lattice import AltNuTamariLattice,␣
↪NuTamariLattice
>>> AltNuTamariLattice('01001', [Integer(0), Integer(0)])
Finite lattice containing 7 elements
>>> AltNuTamariLattice('01001', [Integer(1), Integer(0)])
Finite lattice containing 7 elements
>>> AltNuTamariLattice('01001') == AltNuTamariLattice('01001', [Integer(2),␣
↪Integer(0)])
True
>>> nu = '00100100101'; P = AltNuTamariLattice(nu); Q = NuTamariLattice(nu); P ==␣
↪Q
True
```

REFERENCES:

- [PRV2017]
- [CC2023]

`sage.combinat.nu_tamari_lattice.`**`NuTamariLattice`**`(`*nu*`)`

Return the ν-Tamari lattice.

INPUT:

- ν – list of 0s and 1s or a string of 0s and 1s

OUTPUT: a finite lattice

The elements of the lattice are `nu-Dyck paths` weakly above ν.

The usual Tamari lattice is the special case where $\nu = (NE)^h$ where h is the height.

Other special cases give the m-Tamari lattices studied in [BMFPR].

EXAMPLES:

```
sage: from sage.combinat.nu_tamari_lattice import NuTamariLattice
sage: NuTamariLattice([1,0,1,0,0,1,0])
Finite lattice containing 7 elements
sage: NuTamariLattice([1,0,1,0,1,0])
Finite lattice containing 5 elements
sage: NuTamariLattice([1,0,1,0,1,0,1,0])
Finite lattice containing 14 elements
sage: NuTamariLattice([1,0,1,0,1,0,0,0,1])
Finite lattice containing 24 elements
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_tamari_lattice import NuTamariLattice
>>> NuTamariLattice([Integer(1),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(1),Integer(0)])
Finite lattice containing 7 elements
>>> NuTamariLattice([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0)])
Finite lattice containing 5 elements
>>> NuTamariLattice([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(0)])
Finite lattice containing 14 elements
>>> NuTamariLattice([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0),Integer(1)])
Finite lattice containing 24 elements
```

sage.combinat.nu_tamari_lattice.**delta_swap**(*p*, *k*, *delta*)

Perform a covering move in the (δ, ν)-Tamari lattice (or alt ν-Tamari lattice, see [CC2023]).

The letter at position k is a North step of the ν-Dyck word p, and must be preceded by an East step.

The vector $\delta = (\delta_1, \ldots, \delta_n)$ is an increment vector with respect to the path ν, that is to say $\delta_i \leq \nu_i$, where ν_i is the number of East steps following the i-th North step of ν.

INPUT:

- `p` – a ν-Dyck word
- `k` – integer between 0 and `p.length()-1`
- `delta` – list of nonnegative integers of length `p.height()`

OUTPUT: a ν-Dyck word

EXAMPLES:

```
sage: from sage.combinat.nu_tamari_lattice import delta_swap
sage: delta_swap(NuDyckWord('0101', '0101'), 3, delta = [1, 0])
[0, 1, 1, 0]
sage: delta_swap(NuDyckWord('1001110100', '0100010111'), 3, [3, 1, 0, 0, 0])
[1, 0, 1, 1, 1, 0, 0, 1, 0, 0]
sage: delta_swap(NuDyckWord('10100101000', '01001000110'), 2, [2, 3, 0, 1])
[1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0]
sage: delta_swap(NuDyckWord('10100101000', '01001000110'), 2, [1, 1, 0, 0])
[1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.nu_tamari_lattice import delta_swap
>>> delta_swap(NuDyckWord('0101', '0101'), Integer(3), delta = [Integer(1),
↪Integer(0)])
[0, 1, 1, 0]
>>> delta_swap(NuDyckWord('1001110100', '0100010111'), Integer(3), [Integer(3),
↪Integer(1), Integer(0), Integer(0), Integer(0)])
[1, 0, 1, 1, 1, 0, 0, 1, 0, 0]
>>> delta_swap(NuDyckWord('10100101000', '01001000110'), Integer(2), [Integer(2),
↪Integer(3), Integer(0), Integer(1)])
[1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0]
>>> delta_swap(NuDyckWord('10100101000', '01001000110'), Integer(2), [Integer(1),
↪Integer(1), Integer(0), Integer(0)])
[1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0]
```

5.1.155 Ordered Rooted Trees

AUTHORS:

- Florent Hivert (2010-2011): initial revision
- Frédéric Chapoton (2010): contributed some methods

class sage.combinat.ordered_tree.**LabelledOrderedTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: *AbstractLabelledClonableTree*, *OrderedTree*

Labelled ordered trees.

A labelled ordered tree is an ordered tree with a label attached at each node.

INPUT:

- `children` – list or tuple or more generally any iterable of trees or object convertible to trees
- `label` – any Sage object (default: `None`)

EXAMPLES:

```
sage: x = LabelledOrderedTree([], label = 3); x
3[]
sage: LabelledOrderedTree([x, x, x], label = 2)
2[3[], 3[], 3[]]
sage: LabelledOrderedTree((x, x, x), label = 2)
2[3[], 3[], 3[]]
sage: LabelledOrderedTree([[],[[], []]], label = 3)
3[None[], None[None[], None[]]]
```

```
>>> from sage.all import *
>>> x = LabelledOrderedTree([], label = Integer(3)); x
3[]
>>> LabelledOrderedTree([x, x, x], label = Integer(2))
2[3[], 3[], 3[]]
>>> LabelledOrderedTree((x, x, x), label = Integer(2))
2[3[], 3[], 3[]]
>>> LabelledOrderedTree([[],[[], []]], label = Integer(3))
3[None[], None[None[], None[]]]
```

left_right_symmetry()

Return the symmetric tree of `self`.

The symmetric tree $s(T)$ of a labelled ordered tree T is defined as follows: If T is a labelled ordered tree with children $C_1, C_2, \ldots, C_k$ (listed from left to right), then the symmetric tree $s(T)$ of T is a labelled ordered tree with children $s(C_k), s(C_{k-1}), \ldots, s(C_1)$ (from left to right), and with the same root label as T.

Note

If you have a subclass of *LabelledOrderedTree()* which also inherits from another subclass of *OrderedTree()* which does not come with a labelling, then (depending on the method resolution order) it might happen that this method gets overridden by an implementation from that other subclass, and thus forgets about the labels. In this case you need to manually override this method on your subclass.

EXAMPLES:

```
sage: L2 = LabelledOrderedTree([], label=2)
sage: L3 = LabelledOrderedTree([], label=3)
sage: T23 = LabelledOrderedTree([L2, L3], label=4)
sage: T23.left_right_symmetry()
4[3[], 2[]]
sage: T223 = LabelledOrderedTree([L2, T23], label=17)
sage: T223.left_right_symmetry()
17[4[3[], 2[]], 2[]]
sage: T223.left_right_symmetry().left_right_symmetry() == T223
True
```

```
>>> from sage.all import *
>>> L2 = LabelledOrderedTree([], label=Integer(2))
>>> L3 = LabelledOrderedTree([], label=Integer(3))
>>> T23 = LabelledOrderedTree([L2, L3], label=Integer(4))
>>> T23.left_right_symmetry()
4[3[], 2[]]
>>> T223 = LabelledOrderedTree([L2, T23], label=Integer(17))
>>> T223.left_right_symmetry()
17[4[3[], 2[]], 2[]]
>>> T223.left_right_symmetry().left_right_symmetry() == T223
True
```

sort_key()

Return a tuple of nonnegative integers encoding the labelled tree `self`.

The first entry of the tuple is a pair consisting of the number of children of the root and the label of the root. Then the rest of the tuple is the concatenation of the tuples associated to these children (we view the children of a tree as trees themselves) from left to right.

This tuple characterizes the labelled tree uniquely, and can be used to sort the labelled ordered trees provided that the labels belong to a type which is totally ordered.

Warning

This method overrides *OrderedTree.sort_key()* and returns a result different from what the latter would return, as it wants to encode the whole labelled tree including its labelling rather than just the unlabelled tree. Therefore, be careful with using this method on subclasses of `LabelledOrderedTree`; under some circumstances they could inherit it from another superclass instead of from `OrderedTree`, which would cause the method to forget the labelling. See the docstring of `OrderedTree.sort_key()`.

EXAMPLES:

```
sage: L2 = LabelledOrderedTree([], label=2)
sage: L3 = LabelledOrderedTree([], label=3)
sage: T23 = LabelledOrderedTree([L2, L3], label=4)
sage: T23.sort_key()
((2, 4), (0, 2), (0, 3))
sage: T32 = LabelledOrderedTree([L3, L2], label=5)
sage: T32.sort_key()
((2, 5), (0, 3), (0, 2))
sage: T23322 = LabelledOrderedTree([T23, T32, L2], label=14)
sage: T23322.sort_key()
((3, 14), (2, 4), (0, 2), (0, 3), (2, 5), (0, 3), (0, 2), (0, 2))
```

```
>>> from sage.all import *
>>> L2 = LabelledOrderedTree([], label=Integer(2))
>>> L3 = LabelledOrderedTree([], label=Integer(3))
>>> T23 = LabelledOrderedTree([L2, L3], label=Integer(4))
>>> T23.sort_key()
((2, 4), (0, 2), (0, 3))
>>> T32 = LabelledOrderedTree([L3, L2], label=Integer(5))
>>> T32.sort_key()
((2, 5), (0, 3), (0, 2))
>>> T23322 = LabelledOrderedTree([T23, T32, L2], label=Integer(14))
>>> T23322.sort_key()
((3, 14), (2, 4), (0, 2), (0, 3), (2, 5), (0, 3), (0, 2), (0, 2))
```

class `sage.combinat.ordered_tree.`**`LabelledOrderedTrees`**(*category=None*)

Bases: `UniqueRepresentation`, `Parent`

This is a parent stub to serve as a factory class for trees with various label constraints.

EXAMPLES:

```
sage: LOT = LabelledOrderedTrees(); LOT
Labelled ordered trees
sage: x = LOT([], label = 3); x
3[]
sage: x.parent() is LOT
True
sage: y = LOT([x, x, x], label = 2); y
2[3[], 3[], 3[]]
sage: y.parent() is LOT
True
```

```
>>> from sage.all import *
>>> LOT = LabelledOrderedTrees(); LOT
Labelled ordered trees
>>> x = LOT([], label = Integer(3)); x
```

(continues on next page)

(continued from previous page)

```
3[]
>>> x.parent() is LOT
True
>>> y = LOT([x, x, x], label = Integer(2)); y
2[3[], 3[], 3[]]
>>> y.parent() is LOT
True
```

Element

alias of *LabelledOrderedTree*

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: LabelledOrderedTrees().cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> LabelledOrderedTrees().cardinality()
+Infinity
```

labelled_trees()

Return the set of labelled trees associated to `self`.

This is precisely `self`, because `self` already is the set of labelled ordered trees.

EXAMPLES:

```
sage: LabelledOrderedTrees().labelled_trees()
Labelled ordered trees
sage: LOT = LabelledOrderedTrees()
sage: x = LOT([], label = 3)
sage: y = LOT([x, x, x], label = 2)
sage: y.canonical_labelling()
1[2[], 3[], 4[]]
```

```
>>> from sage.all import *
>>> LabelledOrderedTrees().labelled_trees()
Labelled ordered trees
>>> LOT = LabelledOrderedTrees()
>>> x = LOT([], label = Integer(3))
>>> y = LOT([x, x, x], label = Integer(2))
>>> y.canonical_labelling()
1[2[], 3[], 4[]]
```

unlabelled_trees()

Return the set of unlabelled trees associated to `self`.

This is the set of ordered trees, since `self` is the set of labelled ordered trees.

EXAMPLES:

```
sage: LabelledOrderedTrees().unlabelled_trees()
Ordered trees
```

```
>>> from sage.all import *
>>> LabelledOrderedTrees().unlabelled_trees()
Ordered trees
```

class sage.combinat.ordered_tree.**OrderedTree**(*parent=None*, *children=None*, *check=True*)

Bases: *AbstractClonableTree*, ClonableList

The class of (ordered rooted) trees.

An ordered tree is constructed from a node, called the root, on which one has grafted a possibly empty list of trees. There is a total order on the children of a node which is given by the order of the elements in the list. Note that there is no empty ordered tree (so the smallest ordered tree consists of just one node).

INPUT:

One can create a tree from any list (or more generally iterable) of trees or objects convertible to a tree. Alternatively a string is also accepted. The syntax is the same as for printing: children are grouped by square brackets.

EXAMPLES:

```
sage: x = OrderedTree([])
sage: x1 = OrderedTree([x,x])
sage: x2 = OrderedTree([[],[]])
sage: x1 == x2
True
sage: tt1 = OrderedTree([x,x1,x2])
sage: tt2 = OrderedTree([[], [[], []], x2])
sage: tt1 == tt2
True

sage: OrderedTree([]) == OrderedTree()
True
```

```
>>> from sage.all import *
>>> x = OrderedTree([])
>>> x1 = OrderedTree([x,x])
>>> x2 = OrderedTree([[],[]])
>>> x1 == x2
True
>>> tt1 = OrderedTree([x,x1,x2])
>>> tt2 = OrderedTree([[], [[], []], x2])
>>> tt1 == tt2
True

>>> OrderedTree([]) == OrderedTree()
True
```

is_empty()

Return if `self` is the empty tree.

For ordered trees, this always returns `False`.

Note: this is different from `bool(t)` which returns whether `t` has some child or not.

EXAMPLES:

```
sage: t = OrderedTrees(4)([[],[[]]])
sage: t.is_empty()
False
sage: bool(t)
True
```

```
>>> from sage.all import *
>>> t = OrderedTrees(Integer(4))([[],[[]]])
>>> t.is_empty()
False
>>> bool(t)
True
```

left_right_symmetry()

Return the symmetric tree of `self`.

The symmetric tree $s(T)$ of an ordered tree T is defined as follows: If T is an ordered tree with children $C_1, C_2, \ldots, C_k$ (listed from left to right), then the symmetric tree $s(T)$ of T is the ordered tree with children $s(C_k), s(C_{k-1}), \ldots, s(C_1)$ (from left to right).

EXAMPLES:

```
sage: T = OrderedTree([[],[[]]])
sage: T.left_right_symmetry()
[[[]], []]
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
sage: T.left_right_symmetry()
[[[[]], []], [[], []], []]
```

```
>>> from sage.all import *
>>> T = OrderedTree([[],[[]]])
>>> T.left_right_symmetry()
[[[]], []]
>>> T = OrderedTree([[], [[], []], [[], [[]]]])
>>> T.left_right_symmetry()
[[[[]], []], [[], []], []]
```

normalize(*inplace=False*)

Return the normalized tree of `self`.

INPUT:

- `inplace` – boolean (default: `False`); if `True`, then `self` is modified and nothing returned. Otherwise the normalized tree is returned.

The normalization of an ordered tree t is an ordered tree s which has the property that t and s are isomorphic as *unordered* rooted trees, and that if two ordered trees t and t' are isomorphic as *unordered* rooted trees, then the normalizations of t and t' are identical. In other words, normalization is a map from the set of ordered trees to itself which picks a representative from every equivalence class with respect to the relation of "being isomorphic as unordered trees", and maps every ordered tree to the representative chosen from its class.

This map proceeds recursively by first normalizing every subtree, and then sorting the subtrees according to the value of the `sort_key()` method.

Consider the quotient map π that sends a planar rooted tree to the associated unordered rooted tree. Normalization is the composite $s \circ \pi$, where s is a section of π.

EXAMPLES:

```
sage: OT = OrderedTree
sage: ta = OT([[],[[]]])
sage: tb = OT([[[]],[]])
sage: ta.normalize() == tb.normalize()
True
sage: ta == tb
False
```

```
>>> from sage.all import *
>>> OT = OrderedTree
>>> ta = OT([[],[[]]])
>>> tb = OT([[[]],[]])
>>> ta.normalize() == tb.normalize()
True
>>> ta == tb
False
```

An example with inplace normalization:

```
sage: OT = OrderedTree
sage: ta = OT([[],[[]]])
sage: tb = OT([[[]],[]])
sage: ta.normalize(inplace=True); ta
[[], [[]]]
sage: tb.normalize(inplace=True); tb
[[], [[]]]
```

```
>>> from sage.all import *
>>> OT = OrderedTree
>>> ta = OT([[],[[]]])
>>> tb = OT([[[]],[]])
>>> ta.normalize(inplace=True); ta
[[], [[]]]
>>> tb.normalize(inplace=True); tb
[[], [[]]]
```

plot()

Plot the tree `self`.

> **Warning**
>
> For a labelled tree, this will fail unless all labels are distinct. For unlabelled trees, some arbitrary labels are chosen. Use `_latex_()`, `view`, `_ascii_art_()` or `pretty_print` for more faithful representations

of the data of the tree.

EXAMPLES:

```
sage: p = OrderedTree([[[]],[],[]])
sage: ascii_art(p)
  _o__
 / / /
o o o
|
o
sage: p.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 10 graphics primitives
```

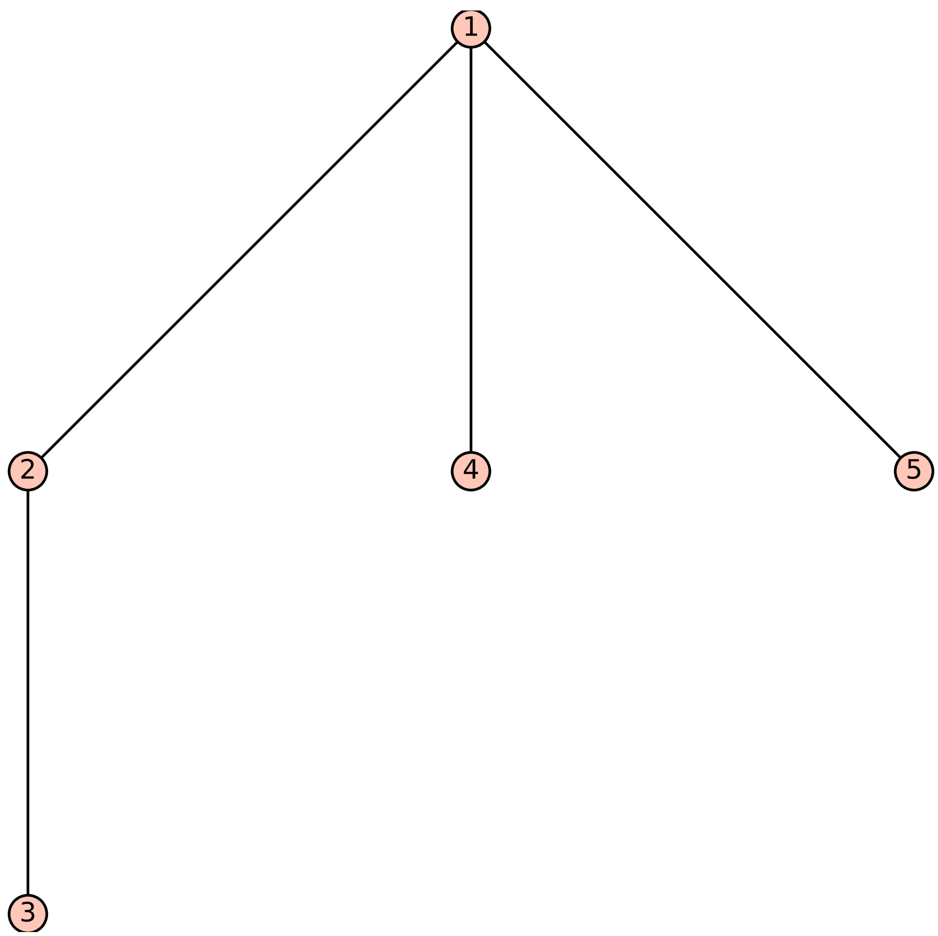
```
>>> from sage.all import *
>>> p = OrderedTree([[[]],[],[]])
>>> ascii_art(p)
  _o__
 / / /
o o o
|
o
>>> p.plot()                                                        #␣
↪needs sage.plot
Graphics object consisting of 10 graphics primitives
```

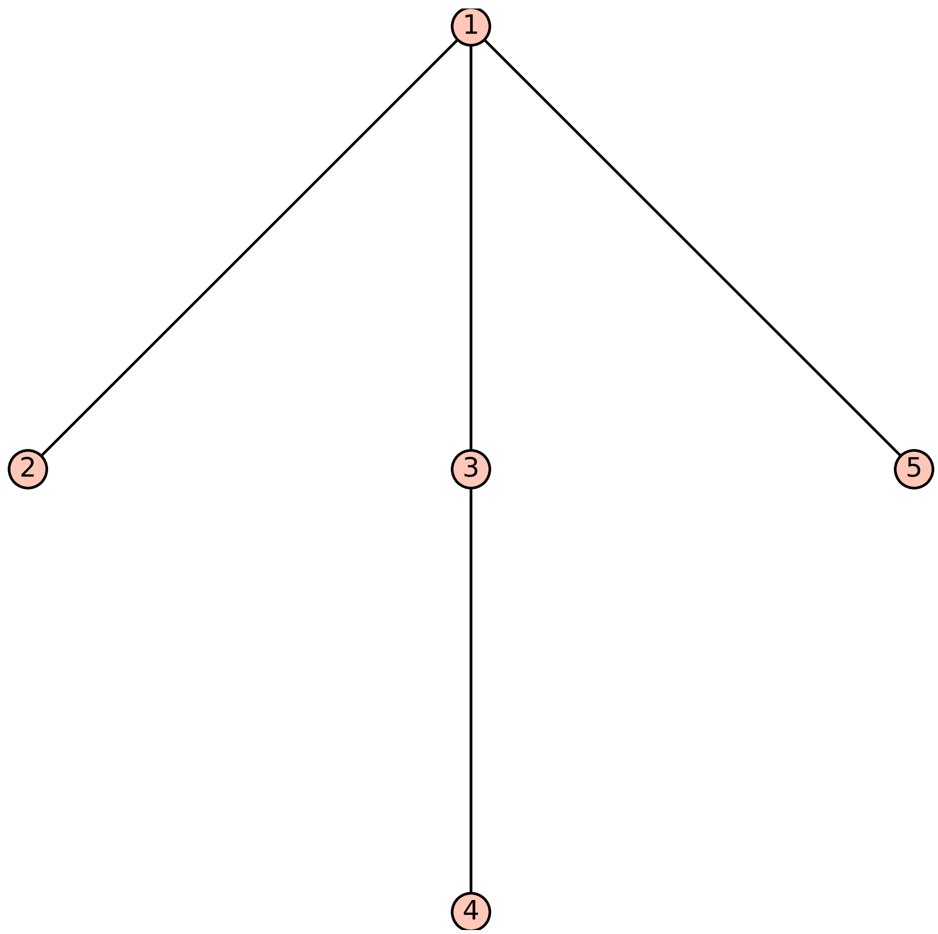
Now a labelled example:

```
sage: g = OrderedTree([[],[[]],[]]).canonical_labelling()
sage: ascii_art(g)
  _1__
 / / /
2 3 5
  |
  4
sage: g.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 10 graphics primitives
```

```
>>> from sage.all import *
>>> g = OrderedTree([[],[[]],[]]).canonical_labelling()
>>> ascii_art(g)
  _1__
 / / /
2 3 5
  |
  4
>>> g.plot()                                                        #␣
↪needs sage.plot
Graphics object consisting of 10 graphics primitives
```

sort_key()


1
2
4
5
3


1
2
3
5
4

Return a tuple of nonnegative integers encoding the ordered tree `self`.

The first entry of the tuple is the number of children of the root. Then the rest of the tuple is the concatenation of the tuples associated to these children (we view the children of a tree as trees themselves) from left to right.

This tuple characterizes the tree uniquely, and can be used to sort the ordered trees.

Note: By default, this method does not encode any extra structure that `self` might have – e.g., if you were to define a class `EdgeColoredOrderedTree` which implements edge-colored trees and which inherits from `OrderedTree`, then the `sort_key()` method it would inherit would forget about the colors of the edges (and thus would not characterize edge-colored trees uniquely). If you want to preserve extra data, you need to override this method or use a new method. For instance, on the `LabelledOrderedTree` subclass, this method is overridden by a slightly different method, which encodes not only the numbers of children of the nodes of `self`, but also their labels. Be careful with using overridden methods, however: If you have (say) a class `BalancedTree` which inherits from `OrderedTree` and which encodes balanced trees, and if you have another class `BalancedLabelledOrderedTree` which inherits both from `BalancedOrderedTree` and from `LabelledOrderedTree`, then (depending on the MRO) the default `sort_key()` method on `BalancedLabelledOrderedTree` (unless manually overridden) will be taken either from `BalancedTree` or from `LabelledOrderedTree`, and in the former case will ignore the labelling!

EXAMPLES:

```
sage: RT = OrderedTree
sage: RT([[],[[]]]).sort_key()
(2, 0, 1, 0)
sage: RT([[[]],[]]).sort_key()
(2, 1, 0, 0)
```

```
>>> from sage.all import *
>>> RT = OrderedTree
>>> RT([[],[[]]]).sort_key()
(2, 0, 1, 0)
>>> RT([[[]],[]]).sort_key()
(2, 1, 0, 0)
```

to_binary_tree_left_branch()

Return a binary tree of size $n - 1$ (where n is the size of t, and where t is `self`) obtained from t by the following recursive rule:

- if x is the left brother of y in t, then x becomes the left child of y;
- if x is the last child of y in t, then x becomes the right child of y,

and removing the root of t.

EXAMPLES:

```
sage: T = OrderedTree([[],[]])
sage: T.to_binary_tree_left_branch()
[[., .], .]
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
```

(continues on next page)

(continued from previous page)

```
sage: T.to_binary_tree_left_branch()
[[[., .], [[., .], .]], [[., .], [., .]]]
```

```
>>> from sage.all import *
>>> T = OrderedTree([[],[]])
>>> T.to_binary_tree_left_branch()
[[., .], .]
>>> T = OrderedTree([[], [[], []], [[], [[]]]])
>>> T.to_binary_tree_left_branch()
[[[., .], [[., .], .]], [[., .], [., .]]]
```

to_binary_tree_right_branch()

Return a binary tree of size $n - 1$ (where n is the size of t, and where t is `self`) obtained from t by the following recursive rule:

- if x is the right brother of y in t, then`x` becomes the right child of y;
- if x is the first child of y in t, then x becomes the left child of y,

and removing the root of t.

EXAMPLES:

```
sage: T = OrderedTree([[],[]])
sage: T.to_binary_tree_right_branch()
[., [., .]]
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
sage: T.to_binary_tree_right_branch()
[., [[., [., .]], [[., [[., .], .]], .]]]
```

```
>>> from sage.all import *
>>> T = OrderedTree([[],[]])
>>> T.to_binary_tree_right_branch()
[., [., .]]
>>> T = OrderedTree([[], [[], []], [[], [[]]]])
>>> T.to_binary_tree_right_branch()
[., [[., [., .]], [[., [[., .], .]], .]]]
```

to_dyck_word()

Return the Dyck path corresponding to `self` where the maximal height of the Dyck path is the depth of `self` .

EXAMPLES:

```
sage: T = OrderedTree([[],[]])
sage: T.to_dyck_word()                                                  #␣
↪needs sage.combinat
[1, 0, 1, 0]
sage: T = OrderedTree([[],[[]]])
sage: T.to_dyck_word()                                                  #␣
↪needs sage.combinat
[1, 0, 1, 1, 0, 0]
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
sage: T.to_dyck_word()                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
[1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> T = OrderedTree([[],[]])
>>> T.to_dyck_word()                                                    #␣
↪needs sage.combinat
[1, 0, 1, 0]
>>> T = OrderedTree([[],[[]]])
>>> T.to_dyck_word()                                                    #␣
↪needs sage.combinat
[1, 0, 1, 1, 0, 0]
>>> T = OrderedTree([[], [[], []], [[], [[]]]])
>>> T.to_dyck_word()                                                    #␣
↪needs sage.combinat
[1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
```

to_parallelogram_polyomino(*bijection=None*)

Return a polyomino parallelogram.

INPUT:

- `bijection` – (default: `'Boussicault-Socci'`) is the name of the bijection to use. Possible values are `'Boussicault-Socci'`, `'via dyck and Delest-Viennot'`.

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: T = OrderedTree([[[], [[], [[]]]], [], [[[],[]]], [], []])
sage: T.to_parallelogram_polyomino(bijection='Boussicault-Socci')
[[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
 [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0]]
sage: T = OrderedTree( [] )
sage: T.to_parallelogram_polyomino()
[[1], [1]]
sage: T = OrderedTree( [[]] )
sage: T.to_parallelogram_polyomino()
[[0, 1], [1, 0]]
sage: T = OrderedTree( [[],[]] )
sage: T.to_parallelogram_polyomino()
[[0, 1, 1], [1, 1, 0]]
sage: T = OrderedTree( [[[]]] )
sage: T.to_parallelogram_polyomino()
[[0, 0, 1], [1, 0, 0]]
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> T = OrderedTree([[[], [[], [[]]]], [], [[[],[]]], [], []])
>>> T.to_parallelogram_polyomino(bijection='Boussicault-Socci')
[[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
 [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0]]
>>> T = OrderedTree( [] )
>>> T.to_parallelogram_polyomino()
```

(continues on next page)

(continued from previous page)

```
[[1], [1]]
>>> T = OrderedTree( [[]] )
>>> T.to_parallelogram_polyomino()
[[0, 1], [1, 0]]
>>> T = OrderedTree( [[],[]] )
>>> T.to_parallelogram_polyomino()
[[0, 1, 1], [1, 1, 0]]
>>> T = OrderedTree( [[[]]] )
>>> T.to_parallelogram_polyomino()
[[0, 0, 1], [1, 0, 0]]
```

to_poset (*root_to_leaf=False*)

Return the poset obtained by interpreting the tree as a Hasse diagram. The default orientation is from leaves to root but you can pass `root_to_leaf=True` to obtain the inverse orientation.

INPUT:

- `root_to_leaf` – boolean (default: `False`); `True` if the poset orientation should be from root to leaves

EXAMPLES:

```
sage: t = OrderedTree([])
sage: t.to_poset()
Finite poset containing 1 elements
sage: p = OrderedTree([[[]],[],[]]).to_poset()
sage: p.height(), p.width()                                            #␣
↪needs networkx
(3, 3)
```

```
>>> from sage.all import *
>>> t = OrderedTree([])
>>> t.to_poset()
Finite poset containing 1 elements
>>> p = OrderedTree([[[]],[],[]]).to_poset()
>>> p.height(), p.width()                                            #␣
↪needs networkx
(3, 3)
```

If the tree is labelled, we use its labelling to label the poset. Otherwise, we use the poset canonical labelling:

```
sage: t = OrderedTree([[[]],[],[]]).canonical_labelling().to_poset()
sage: t.height(), t.width()                                            #␣
↪needs networkx
(3, 3)
```

```
>>> from sage.all import *
>>> t = OrderedTree([[[]],[],[]]).canonical_labelling().to_poset()
>>> t.height(), t.width()                                            #␣
↪needs networkx
(3, 3)
```

to_undirected_graph ()

Return the undirected graph obtained from the tree nodes and edges.

The graph is endowed with an embedding, so that it will be displayed correctly.

EXAMPLES:

```
sage: t = OrderedTree([])
sage: t.to_undirected_graph()
Graph on 1 vertex
sage: t = OrderedTree([[[]],[],[]])
sage: t.to_undirected_graph()
Graph on 5 vertices
```

```
>>> from sage.all import *
>>> t = OrderedTree([])
>>> t.to_undirected_graph()
Graph on 1 vertex
>>> t = OrderedTree([[[]],[],[]])
>>> t.to_undirected_graph()
Graph on 5 vertices
```

If the tree is labelled, we use its labelling to label the graph. This will fail if the labels are not all distinct. Otherwise, we use the graph canonical labelling which means that two different trees can have the same graph.

EXAMPLES:

```
sage: t = OrderedTree([[[]],[],[]])
sage: t.canonical_labelling().to_undirected_graph()
Graph on 5 vertices
```

```
>>> from sage.all import *
>>> t = OrderedTree([[[]],[],[]])
>>> t.canonical_labelling().to_undirected_graph()
Graph on 5 vertices
```

class sage.combinat.ordered_tree.**OrderedTrees**

Bases: `UniqueRepresentation`, `Parent`

Factory for ordered trees.

INPUT:

- `size` – integer (optional)

OUTPUT:

- the set of all ordered trees (of the given `size` if specified)

EXAMPLES:

```
sage: OrderedTrees()
Ordered trees

sage: OrderedTrees(2)
Ordered trees of size 2
```

```
>>> from sage.all import *
>>> OrderedTrees()
Ordered trees
```

(continues on next page)

(continued from previous page)

```
>>> OrderedTrees(Integer(2))
Ordered trees of size 2
```

Note: this is a factory class whose constructor returns instances of subclasses.

Note: the fact that OrderedTrees is a class instead of a simple callable is an implementation detail. It could be changed in the future and one should not rely on it.

leaf()

Return a leaf tree with `self` as parent.

EXAMPLES:

```
sage: OrderedTrees().leaf()
[]
```

```
>>> from sage.all import *
>>> OrderedTrees().leaf()
[]
```

class sage.combinat.ordered_tree.**OrderedTrees_all**

Bases: `DisjointUnionEnumeratedSets`, *OrderedTrees*

The set of all ordered trees.

EXAMPLES:

```
sage: OT = OrderedTrees(); OT
Ordered trees
sage: OT.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> OT = OrderedTrees(); OT
Ordered trees
>>> OT.cardinality()
+Infinity
```

Element

alias of *OrderedTree*

labelled_trees()

Return the set of labelled trees associated to `self`.

EXAMPLES:

```
sage: OrderedTrees().labelled_trees()
Labelled ordered trees
```

```
>>> from sage.all import *
>>> OrderedTrees().labelled_trees()
Labelled ordered trees
```

unlabelled_trees()

Return the set of unlabelled trees associated to `self`.

EXAMPLES:

```
sage: OrderedTrees().unlabelled_trees()
Ordered trees
```

```
>>> from sage.all import *
>>> OrderedTrees().unlabelled_trees()
Ordered trees
```

class sage.combinat.ordered_tree.**OrderedTrees_size**(*size*)

Bases: *OrderedTrees*

The enumerated sets of binary trees of a given size.

EXAMPLES:

```
sage: S = OrderedTrees(3); S
Ordered trees of size 3
sage: S.cardinality()
2
sage: S.list()
[[[], []], [[[]]]]
```

```
>>> from sage.all import *
>>> S = OrderedTrees(Integer(3)); S
Ordered trees of size 3
>>> S.cardinality()
2
>>> S.list()
[[[], []], [[[]]]]
```

cardinality()

The cardinality of `self`.

This is a Catalan number.

element_class()

The class of the element of `self`.

EXAMPLES:

```
sage: from sage.combinat.ordered_tree import OrderedTrees_size, OrderedTrees_
↪all
sage: S = OrderedTrees_size(3)
```

(continues on next page)

(continued from previous page)

```
sage: S.element_class is OrderedTrees().element_class
True
sage: S.first().__class__ == OrderedTrees_all().first().__class__
True
```

```
>>> from sage.all import *
>>> from sage.combinat.ordered_tree import OrderedTrees_size, OrderedTrees_all
>>> S = OrderedTrees_size(Integer(3))
>>> S.element_class is OrderedTrees().element_class
True
>>> S.first().__class__ == OrderedTrees_all().first().__class__
True
```

random_element()

Return a random `OrderedTree` with uniform probability.

This method generates a random `DyckWord` and then uses a bijection between Dyck words and ordered trees.

EXAMPLES:

```
sage: OrderedTrees(5).random_element()  # random                          #␣
↪needs sage.combinat
[[[], []], []]
sage: OrderedTrees(0).random_element()
Traceback (most recent call last):
...
EmptySetError: there are no ordered trees of size 0
sage: OrderedTrees(1).random_element()                                      #␣
↪needs sage.combinat
[]
```

```
>>> from sage.all import *
>>> OrderedTrees(Integer(5)).random_element()  # random                      ␣
↪     # needs sage.combinat
[[[], []], []]
>>> OrderedTrees(Integer(0)).random_element()
Traceback (most recent call last):
...
EmptySetError: there are no ordered trees of size 0
>>> OrderedTrees(Integer(1)).random_element()                                ␣
↪     # needs sage.combinat
[]
```

5.1.156 Output functions

These are the output functions for latexing and ascii/unicode art versions of partitions and tableaux.

AUTHORS:

- Mike Hansen (?): initial version
- Andrew Mathas (2013-02-14): Added support for displaying conventions and lines, and tableaux of skew partition, composition, and skew/composition/partition/tableaux tuple shape.
- Travis Scrimshaw (2020-08): Added support for ascii/unicode art

sage.combinat.output.**ascii_art_table**(*data*, *use_unicode=False*, *convention='English'*)

Return an ascii art table of `data`.

EXAMPLES:

```
sage: from sage.combinat.output import ascii_art_table

sage: data = [[None, None, 1], [2, 2], [3,4,5], [None, None, 10], [], [6]]
sage: print(ascii_art_table(data))
        +----+
        | 1  |
+---+---+----+
| 2 | 2 |
+---+---+----+
| 3 | 4 | 5  |
+---+---+----+
        | 10 |
        +----+


+---+
| 6 |
+---+
sage: print(ascii_art_table(data, use_unicode=True))
        ┌────┐
        │ 1  │
┌───┬───┼────┘
│ 2 │ 2 │
├───┼───┼────┐
│ 3 │ 4 │ 5  │
└───┴───┼────┤
        │ 10 │
        └────┘


┌───┐
│ 6 │
└───┘

sage: data = [[1, None, 2], [None, 2]]
sage: print(ascii_art_table(data))
+---+   +---+
| 1 |   | 2 |
+---+---+---+
    | 2 |
    +---+
sage: print(ascii_art_table(data, use_unicode=True))
┌───┐   ┌───┐
│ 1 │   │ 2 │
└───┼───┼───┘
    │ 2 │
    └───┘
```

```
>>> from sage.all import *
>>> from sage.combinat.output import ascii_art_table
```

(continues on next page)

(continued from previous page)

```
>>> data = [[None, None, Integer(1)], [Integer(2), Integer(2)], [Integer(3),
↪Integer(4),Integer(5)], [None, None, Integer(10)], [], [Integer(6)]]
>>> print(ascii_art_table(data))
        +----+
        | 1  |
+---+---+----+
| 2 | 2 |
+---+---+----+
| 3 | 4 | 5  |
+---+---+----+
        | 10 |
        +----+
<BLANKLINE>
+---+
| 6 |
+---+
>>> print(ascii_art_table(data, use_unicode=True))
        ┌────┐
        │ 1  │
┌───┬───┼────┘
│ 2 │ 2 │
├───┼───┼────┐
│ 3 │ 4 │ 5  │
└───┴───┼────┤
        │ 10 │
        └────┘
<BLANKLINE>
┌───┐
│ 6 │
└───┘

>>> data = [[Integer(1), None, Integer(2)], [None, Integer(2)]]
>>> print(ascii_art_table(data))
+---+   +---+
| 1 |   | 2 |
+---+---+---+
    | 2 |
    +---+
>>> print(ascii_art_table(data, use_unicode=True))
┌───┐   ┌───┐
│ 1 │   │ 2 │
└───┼───┼───┘
    │ 2 │
    └───┘
```

`sage.combinat.output.`**`ascii_art_table_russian`**(*data*, *use_unicode=False*, *compact=False*)

Return an ascii art table of `data` for the russian convention.

EXAMPLES:

```
sage: from sage.combinat.output import ascii_art_table_russian
```

(continues on next page)

(continued from previous page)

```
sage: data = [[None, None, 1], [2, 2], [3,4,5], [None, None, 10], [], [6]]
sage: print(ascii_art_table_russian(data))
   / \         / \
  /   \       /   \
 \  6  /     \ 10  \
  \   /       \   / \
   \ /         \ /   \
                X  5  /
               / \   /
              /   \ /
             /  4  X
            / \   / \   / \
           /   \ /   \ /   \
          \  3  X  2  X  1  /
           \   / \   / \   /
            \ /   \ /   \ /
             \  2  /
              \   /
               \ /
sage: print(ascii_art_table_russian(data, use_unicode=True))
   ╱ ╲         ╱ ╲
  ╱   ╲       ╱   ╲
 ╲  6  ╱     ╲ 10  ╲
  ╲   ╱       ╲   ╱ ╲
   ╲ ╱         ╲ ╱   ╲
                ╳  5  ╱
               ╱ ╲   ╱
              ╱   ╲ ╱
             ╱  4  ╳
            ╱ ╲   ╱ ╲   ╱ ╲
           ╱   ╲ ╱   ╲ ╱   ╲
          ╲  3  ╳  2  ╳  1  ╱
           ╲   ╱ ╲   ╱ ╲   ╱
            ╲ ╱   ╲ ╱   ╲ ╱
             ╲  2  ╱
              ╲   ╱
               ╲ ╱
sage: data = [[1, None, 2], [None, 2]]
sage: print(ascii_art_table_russian(data))
  / \ / \
 \ 2 X 2 /
  \ / \ /
   X
  / \
 \ 1 /
  \ /
sage: print(ascii_art_table_russian(data, use_unicode=True))
  ╱ ╲ ╱ ╲
 ╲ 2 ╳ 2 ╱
  ╲ ╱ ╲ ╱
   ╳
  ╱ ╲
```

(continues on next page)

(continued from previous page)

```
 \ 1 /
  \ /
```

```
>>> from sage.all import *
>>> from sage.combinat.output import ascii_art_table_russian
>>> data = [[None, None, Integer(1)], [Integer(2), Integer(2)], [Integer(3),
↪Integer(4),Integer(5)], [None, None, Integer(10)], [], [Integer(6)]]
>>> print(ascii_art_table_russian(data))
   / \         / \
  /   \       /   \
 \  6  /     \ 10  \
  \   /       \   / \
   \ /         \ /   \
                X  5  /
               / \   /
              /   \ /
             /  4  X
            / \   / \   / \
           /   \ /   \ /   \
          \  3  X  2  X  1  /
           \   / \   / \   /
            \ /   \ /   \ /
             \  2  /
              \   /
               \ /
>>> print(ascii_art_table_russian(data, use_unicode=True))
   ╱ ╲         ╱ ╲
  ╱   ╲       ╱   ╲
 ╲  6  ╱     ╲ 10  ╲
  ╲   ╱       ╲   ╱ ╲
   ╲ ╱         ╲ ╱   ╲
                ╳  5  ╱
               ╱ ╲   ╱
              ╱   ╲ ╱
             ╱  4  ╳
            ╱ ╲   ╱ ╲   ╱ ╲
           ╱   ╲ ╱   ╲ ╱   ╲
          ╲  3  ╳  2  ╳  1  ╱
           ╲   ╱ ╲   ╱ ╲   ╱
            ╲ ╱   ╲ ╱   ╲ ╱
             ╲  2  ╱
              ╲   ╱
               ╲ ╱
>>> data = [[Integer(1), None, Integer(2)], [None, Integer(2)]]
>>> print(ascii_art_table_russian(data))
  / \ / \
 \ 2 X 2 /
  \ / \ /
   X
  / \
 \ 1 /
  \ /
```

(continues on next page)

(continued from previous page)

```
>>> print(ascii_art_table_russian(data, use_unicode=True))
 / \ / \
\ 2 ╳ 2 /
 \ / \ /
  ╳
 / \
\ 1 /
 \ /
```

sage.combinat.output.**box_exists**(*tab*, *i*, *j*)

Return `True` if `tab[i][j]` exists and is not `None`; in particular this allows for $tab[i][j]$ to be `''` or `0`.

INPUT:

- `tab` – list of lists
- `i` – first coordinate
- `j` – second coordinate

sage.combinat.output.**tex_from_array**(*array*, *with_lines=True*)

Return a latex string for a two dimensional array of partition, composition or skew composition shape.

INPUT:

- `array` – list of list
- `with_lines` – boolean (default: `True`); whether to draw a line to separate the entries in the array

Empty rows are allowed; however, such rows should be given as `[None]` rather than `[]`.

The array is drawn using either the English or French convention following `Tableaux.options()`.

See also

`tex_from_array_tuple()`

EXAMPLES:

```
sage: from sage.combinat.output import tex_from_array
sage: print(tex_from_array([[1,2,3],[4,5]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\lr{4}&\lr{5}\\\cline{1-2}
\end{array}$}
}
sage: print(tex_from_array([[1,2,3],[4,5]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}\\
\end{array}$}
}
```

(continues on next page)

(continued from previous page)

```
sage: print(tex_from_array([[1,2,3],[4,5,6,7],[8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-4}
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{8}\\\cline{1-1}
\end{array}$}
}
sage: print(tex_from_array([[1,2,3],[4,5,6,7],[8]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\
\lr{8}\\
\end{array}$}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{3-3}
&&\lr{3}\\\cline{2-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{8}\\\cline{1-1}
\end{array}$}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[None,8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{3-3}
&&\lr{3}\\\cline{2-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&\lr{8}\\\cline{2-2}
\end{array}$}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[8]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&&\lr{3}\\
&\lr{5}&\lr{6}&\lr{7}\\
\lr{8}\\
\end{array}$}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[None,8]], with_
↪lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&&\lr{3}\\
&\lr{5}&\lr{6}&\lr{7}\\
```

(continues on next page)

(continued from previous page)

```
&\lr{8}\\
\end{array}$}
}
sage: Tableaux.options.convention="french"
sage: print(tex_from_array([[1,2,3],[4,5]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{4}&\lr{5}\\\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$}
}
sage: print(tex_from_array([[1,2,3],[4,5]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{4}&\lr{5}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$}
}
sage: print(tex_from_array([[1,2,3],[4,5,6,7],[8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{8}\\\cline{1-4}
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$}
}
sage: print(tex_from_array([[1,2,3],[4,5,6,7],[8]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
\lr{8}\\
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{8}\\\cline{1-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&&\lr{3}\\\cline{3-3}
\end{array}$}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[None,8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{2-2}
&\lr{8}\\\cline{2-4}
```

(continues on next page)

(continued from previous page)

```
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&&\lr{3}\\\cline{3-3}
\end{array}$}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[8]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
\lr{8}\\
&\lr{5}&\lr{6}&\lr{7}\\
&&\lr{3}\\
\end{array}$}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[None,8]], with_
↪lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
&\lr{8}\\
&\lr{5}&\lr{6}&\lr{7}\\
&&\lr{3}\\
\end{array}$}
}
sage: Tableaux.options.convention="russian"
sage: print(tex_from_array([[1,2,3],[4,5]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}\\\cline{1-3}
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\\cline{1-
↪3}
\end{array}$}}
}
sage: print(tex_from_array([[1,2,3],[4,5]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\\
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}\\
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\
\end{array}$}}
}
sage: print(tex_from_array([[1,2,3],[4,5,6,7],[8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{\rotatebox{-45}{8}}\\\cline{1-4}
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\
↪rotatebox{-45}{7}}\\\cline{1-4}
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\\cline{1-
↪3}
\end{array}$}}
}
```

(continues on next page)

(continued from previous page)

```
sage: print(tex_from_array([[1,2,3],[4,5,6,7],[8]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\\
\lr{\rotatebox{-45}{8}}\\
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\
↪rotatebox{-45}{7}}\\
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\
\end{array}$}}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{\rotatebox{-45}{8}}\\\cline{1-4}
&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\\cline
↪{2-4}
&&\lr{\rotatebox{-45}{3}}\\\cline{3-3}
\end{array}$}}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[None,8]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\cline{2-2}
&\lr{\rotatebox{-45}{8}}\\\cline{2-4}
&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\\cline
↪{2-4}
&&\lr{\rotatebox{-45}{3}}\\\cline{3-3}
\end{array}$}}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[8]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\\
\lr{\rotatebox{-45}{8}}\\
&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\
&&\lr{\rotatebox{-45}{3}}\\
\end{array}$}}
}
sage: print(tex_from_array([[None,None,3],[None,5,6,7],[None,8]], with_
↪lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\\
&\lr{\rotatebox{-45}{8}}\\
&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\
&&\lr{\rotatebox{-45}{3}}\\
\end{array}$}}
}

sage: Tableaux.options._reset()
```

```
>>> from sage.all import *
>>> from sage.combinat.output import tex_from_array
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\lr{4}&\lr{5}\\\cline{1-2}
\end{array}$}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}\\
\end{array}$}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6),Integer(7)],[Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-4}
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{8}\\\cline{1-1}
\end{array}$}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6),Integer(7)],[Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\
\lr{8}\\
\end{array}$}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{3-3}
&&\lr{3}\\\cline{2-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{8}\\\cline{1-1}
\end{array}$}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[None,Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
```

(continues on next page)

(continued from previous page)

```
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{3-3}
&&\lr{3}\\\cline{2-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&\lr{8}\\\cline{2-2}
\end{array}$}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&&\lr{3}\\
&\lr{5}&\lr{6}&\lr{7}\\
\lr{8}\\
\end{array}$}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[None,Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&&\lr{3}\\
&\lr{5}&\lr{6}&\lr{7}\\
&\lr{8}\\
\end{array}$}
}
>>> Tableaux.options.convention="french"
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{4}&\lr{5}\\\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{4}&\lr{5}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6),Integer(7)],[Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{8}\\\cline{1-4}
```

(continues on next page)

(continued from previous page)

```
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6),Integer(7)],[Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
\lr{8}\\
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{8}\\\cline{1-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&&\lr{3}\\\cline{3-3}
\end{array}$}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[None,Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{2-2}
&\lr{8}\\\cline{2-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&&\lr{3}\\\cline{3-3}
\end{array}$}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
\lr{8}\\
&\lr{5}&\lr{6}&\lr{7}\\
&&\lr{3}\\
\end{array}$}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[None,Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
&\lr{8}\\
&\lr{5}&\lr{6}&\lr{7}\\
&&\lr{3}\\
```

(continues on next page)

(continued from previous page)

```
\end{array}$}
}
>>> Tableaux.options.convention="russian"
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}\\\cline{1-3}
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\\cline{1-
↪3}
\end{array}$}}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\\
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}\\
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\
\end{array}$}}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6),Integer(7)],[Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{\rotatebox{-45}{8}}\\\cline{1-4}
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\
↪rotatebox{-45}{7}}\\\cline{1-4}
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\\cline{1-
↪3}
\end{array}$}}
}
>>> print(tex_from_array([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6),Integer(7)],[Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\\
\lr{\rotatebox{-45}{8}}\\
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\
↪rotatebox{-45}{7}}\\
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\
\end{array}$}}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{\rotatebox{-45}{8}}\\\cline{1-4}
&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\\cline
```

(continues on next page)

(continued from previous page)

```
↪{2-4}
&&\lr{\rotatebox{-45}{3}}\\\cline{3-3}
\end{array}$}}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[None,Integer(8)]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\cline{2-2}
&\lr{\rotatebox{-45}{8}}\\\cline{2-4}
&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\\cline
↪{2-4}
&&\lr{\rotatebox{-45}{3}}\\\cline{3-3}
\end{array}$}}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\\
\lr{\rotatebox{-45}{8}}\\
&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\
&&\lr{\rotatebox{-45}{3}}\\
\end{array}$}}
}
>>> print(tex_from_array([[None,None,Integer(3)],[None,Integer(5),Integer(6),
↪Integer(7)],[None,Integer(8)]], with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{4}c}\\
&\lr{\rotatebox{-45}{8}}\\
&\lr{\rotatebox{-45}{5}}&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\
&&\lr{\rotatebox{-45}{3}}\\
\end{array}$}}
}

>>> Tableaux.options._reset()
```

`sage.combinat.output.`**`tex_from_array_tuple`**`(`*a_tuple*, *with_lines=True*`)`

Return a latex string for a tuple of two dimensional array of partition, composition or skew composition shape.

INPUT:

- `a_tuple` – tuple of lists of lists
- `with_lines` – boolean (default: `True`); whether to draw lines to separate the entries in the components of `a_tuple`

See also

`tex_from_array()` for the description of each array

EXAMPLES:

```
sage: from sage.combinat.output import tex_from_array_tuple
sage: print(tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\lr{4}&\lr{5}\\\cline{1-2}
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{2-3}
&\lr{6}&\lr{7}\\\cline{2-3}
&\lr{8}\\\cline{1-2}
\lr{9}\\\cline{1-1}
\end{array}$}
}
sage: print(tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]],␣
↪with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}\\
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
&\lr{6}&\lr{7}\\
&\lr{8}\\
\lr{9}\\
\end{array}$}
}
sage: Tableaux.options.convention="french"
sage: print(tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{4}&\lr{5}\\\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-1}
\lr{9}\\\cline{1-2}
&\lr{8}\\\cline{2-3}
&\lr{6}&\lr{7}\\\cline{2-3}
\end{array}$}
}
sage: print(tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]],␣
↪with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{4}&\lr{5}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{9}\\
&\lr{8}\\
&\lr{6}&\lr{7}\\
\end{array}$}
}
sage: Tableaux.options.convention="russian"
```

(continues on next page)

(continued from previous page)

```
sage: print(tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}\\\cline{1-3}
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\\cline{1-
↪3}
\end{array}$}},\emptyset,\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\
↪cline{1-1}
\lr{\rotatebox{-45}{9}}\\\cline{1-2}
&\lr{\rotatebox{-45}{8}}\\\cline{2-3}
&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\\cline{2-3}
\end{array}$}}
}
sage: print(tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]],
↪with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\\
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}\\
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\
\end{array}$}},\emptyset,\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\
↪\
\lr{\rotatebox{-45}{9}}\\
&\lr{\rotatebox{-45}{8}}\\
&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\
\end{array}$}}
}

sage: Tableaux.options._reset()
```

```
>>> from sage.all import *
>>> from sage.combinat.output import tex_from_array_tuple
>>> print(tex_from_array_tuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]],[],[[None,Integer(6),Integer(7)],[None,Integer(8)],[Integer(9)]]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\lr{4}&\lr{5}\\\cline{1-2}
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{2-3}
&\lr{6}&\lr{7}\\\cline{2-3}
&\lr{8}\\\cline{1-2}
\lr{9}\\\cline{1-1}
\end{array}$}
}
>>> print(tex_from_array_tuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]],[],[[None,Integer(6),Integer(7)],[None,Integer(8)],[Integer(9)]]],
↪with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
```

(continues on next page)

(continued from previous page)

```
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}\\
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
&\lr{6}&\lr{7}\\
&\lr{8}\\
\lr{9}\\
\end{array}$}
}
>>> Tableaux.options.convention="french"
>>> print(tex_from_array_tuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]],[],[[None,Integer(6),Integer(7)],[None,Integer(8)],[Integer(9)]]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{4}&\lr{5}\\\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-1}
\lr{9}\\\cline{1-2}
&\lr{8}\\\cline{2-3}
&\lr{6}&\lr{7}\\\cline{2-3}
\end{array}$}
}
>>> print(tex_from_array_tuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]],[],[[None,Integer(6),Integer(7)],[None,Integer(8)],[Integer(9)]]],
↪with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{4}&\lr{5}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{9}\\
&\lr{8}\\
&\lr{6}&\lr{7}\\
\end{array}$}
}
>>> Tableaux.options.convention="russian"
>>> print(tex_from_array_tuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]],[],[[None,Integer(6),Integer(7)],[None,Integer(8)],[Integer(9)]]]))
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$
↪#1$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}\\\cline{1-3}
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\\cline{1-
↪3}
\end{array}$}},\emptyset,\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\
↪cline{1-1}
\lr{\rotatebox{-45}{9}}\\\cline{1-2}
&\lr{\rotatebox{-45}{8}}\\\cline{2-3}
&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\\cline{2-3}
\end{array}$}}
}
```

(continues on next page)

(continued from previous page)

```
>>> print(tex_from_array_tuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]],[],[[None,Integer(6),Integer(7)],[None,Integer(8)],[Integer(9)]]],␣
↪with_lines=False))
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1
↪$}}}
\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\\
\lr{\rotatebox{-45}{4}}&\lr{\rotatebox{-45}{5}}\\
\lr{\rotatebox{-45}{1}}&\lr{\rotatebox{-45}{2}}&\lr{\rotatebox{-45}{3}}\\
\end{array}$}},\emptyset,\raisebox{-.6ex}{\rotatebox{45}{$\begin{array}[t]{*{3}c}\
↪\
\lr{\rotatebox{-45}{9}}\\
&\lr{\rotatebox{-45}{8}}\\
&\lr{\rotatebox{-45}{6}}&\lr{\rotatebox{-45}{7}}\\
\end{array}$}}
}

>>> Tableaux.options._reset()
```

sage.combinat.output.**tex_from_skew_array**(*array*, *with_lines=False*, *align='b'*)

This function creates latex code for a "skew composition" `array`. That is, for a two dimensional array in which each row can begin with an arbitrary number `None`'s and the remaining entries could, in principle, be anything but probably should be strings or integers of similar width. A row consisting completely of `None`'s is allowed.

INPUT:

- `array` – the array
- `with_lines` – (default: `False`) if `True` lines are drawn, if `False` they are not
- `align` – (default: `'b'`) determine the alignment on the latex array environments

EXAMPLES:

```
sage: array=[[None, 2,3,4],[None,None],[5,6,7,8]]
sage: print(sage.combinat.output.tex_from_skew_array(array))
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&\lr{2}&\lr{3}&\lr{4}\\
&\\
\lr{5}&\lr{6}&\lr{7}&\lr{8}\\
\end{array}$}
```

```
>>> from sage.all import *
>>> array=[[None, Integer(2),Integer(3),Integer(4)],[None,None],[Integer(5),
↪Integer(6),Integer(7),Integer(8)]]
>>> print(sage.combinat.output.tex_from_skew_array(array))
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&\lr{2}&\lr{3}&\lr{4}\\
&\\
\lr{5}&\lr{6}&\lr{7}&\lr{8}\\
\end{array}$}
```

5.1.157 Parallelogram Polyominoes

The goal of this module is to give some tools to manipulate the parallelogram polyominoes.

class sage.combinat.parallelogram_polyomino.**LocalOptions**(*name=''*, ***options*)

Bases: object

This class allow to add local options to an object. LocalOptions is like a dictionary, it has keys and values that represent options and the values associated to the option. This is useful to decorate an object with some optional informations.

LocalOptions should be used as follow.

INPUT:

- name – the name of the LocalOptions
- <options>=dict(...) – dictionary specifying an option

The options are specified by keyword arguments with their values being a dictionary which describes the option. The allowed/expected keys in the dictionary are:

- checker – a function for checking whether a particular value for the option is valid
- default – the default value of the option
- values – dictionary of the legal values for this option (this automatically defines the corresponding checker); this dictionary gives the possible options, as keys, together with a brief description of them

```
sage: from sage.combinat.parallelogram_polyomino import LocalOptions
sage: o = LocalOptions(
....:     'Name Example',
....:     delim=dict(
....:         default='b',
....:         values={'b':'the option b', 'p':'the option p'}
....:     )
....: )
sage: class Ex:
....:     options=o
....:     def _repr_b(self): return "b"
....:     def _repr_p(self): return "p"
....:     def __repr__(self): return self.options._dispatch(
....:         self, '_repr_','delim'
....:     )
sage: e = Ex(); e
b
sage: e.options(delim='p'); e
p
```

```
>>> from sage.all import *
>>> from sage.combinat.parallelogram_polyomino import LocalOptions
>>> o = LocalOptions(
...     'Name Example',
...     delim=dict(
...         default='b',
...         values={'b':'the option b', 'p':'the option p'}
```

(continues on next page)

(continued from previous page)

```
...     )
... )
>>> class Ex:
...     options=o
...     def _repr_b(self): return "b"
...     def _repr_p(self): return "p"
...     def __repr__(self): return self.options._dispatch(
...         self, '_repr_','delim'
...     )
>>> e = Ex(); e
b
>>> e.options(delim='p'); e
p
```

This class is temporary, in the future, this class should be integrated in sage.structure.global_options.py. We should split global_option in two classes LocalOptions and GlobalOptions.

keys()

Return the list of the options in `self`.

EXAMPLES:

```
sage: from sage.combinat.parallelogram_polyomino import (
....:     LocalOptions
....: )
sage: o = LocalOptions(
....:     "Name Example",
....:     tikz_options=dict(
....:         default='toto',
....:         values=dict(
....:             toto='name',
....:             x="3"
....:         )
....:     ),
....:     display=dict(
....:         default='list',
....:         values=dict(
....:             list="list representation",
....:             diagram="diagram representation"
....:         )
....:     )
....: )
sage: keys=o.keys()
sage: keys.sort()
sage: keys
['display', 'tikz_options']
```

```
>>> from sage.all import *
>>> from sage.combinat.parallelogram_polyomino import (
...     LocalOptions
... )
>>> o = LocalOptions(
...     "Name Example",
```

(continues on next page)

(continued from previous page)

```
...     tikz_options=dict(
...         default='toto',
...         values=dict(
...             toto='name',
...             x="3"
...         )
...     ),
...     display=dict(
...         default='list',
...         values=dict(
...             list="list representation",
...             diagram="diagram representation"
...         )
...     )
... )
>>> keys=o.keys()
>>> keys.sort()
>>> keys
['display', 'tikz_options']
```

class sage.combinat.parallelogram_polyomino.**ParallelogramPolyomino**(*parent*, *value*, *check=True*)

Bases: ClonableList

Parallelogram Polyominoes.

A parallelogram polyomino is a finite connected union of cells whose boundary can be decomposed in two paths, the upper and the lower paths, which are comprised of north and east unit steps and meet only at their starting and final points.

Parallelogram Polyominoes can be defined with those two paths.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp
[[0, 1], [1, 0]]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp
[[0, 1], [1, 0]]
```

area()

Return the area of the parallelogram polyomino. The area of a parallelogram polyomino is the number of cells of the parallelogram polyomino.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1],
....:         [1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0]
```

(continues on next page)

(continued from previous page)

```
....:     ]
....: )
sage: pp.area()
13

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.area()
1

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.area()
0
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(0), Integer(1), Integer(0), Integer(0)]
...     ]
... )
>>> pp.area()
13

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.area()
1

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.area()
0
```

bounce(*direction=1*)

Return the bounce of the parallelogram polyomino.

Let `p` be the bounce path of the parallelogram polyomino (`bounce_path()`). The bounce is defined by:

```
sum([(1+ floor(i/2))*p[i] for i in range(len(p))])
```

INPUT:

- `direction` – the initial direction of the bounce path (see `bounce_path()` for the definition)

EXAMPLES:

```
sage: PP = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 1, 1], [1, 1, 0, 0, 1, 0]]
....: )
sage: PP.bounce(direction=1)
6
sage: PP.bounce(direction=0)
7
```

(continues on next page)

(continued from previous page)

```
sage: PP = ParallelogramPolyomino(
....:     [
....:         [0, 0, 1, 1, 1, 0, 0, 1, 1],
....:         [1, 1, 1, 0, 1, 1, 0, 0, 0]
....:     ]
....: )
sage: PP.bounce(direction=1)
12
sage: PP.bounce(direction=0)
10

sage: PP = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: PP.bounce(direction=1)
1
sage: PP.bounce(direction=0)
1

sage: PP = ParallelogramPolyomino([[1], [1]])
sage: PP.bounce(direction=1)
0
sage: PP.bounce(direction=0)
0
```

```
>>> from sage.all import *
>>> PP = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]]
... )
>>> PP.bounce(direction=Integer(1))
6
>>> PP.bounce(direction=Integer(0))
7

>>> PP = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0)]
...     ]
... )
>>> PP.bounce(direction=Integer(1))
12
>>> PP.bounce(direction=Integer(0))
10

>>> PP = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> PP.bounce(direction=Integer(1))
1
```

(continues on next page)

(continued from previous page)

```
>>> PP.bounce(direction=Integer(0))
1

>>> PP = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> PP.bounce(direction=Integer(1))
0
>>> PP.bounce(direction=Integer(0))
0
```

bounce_path(*direction=1*)

Return the bounce path of the parallelogram polyomino.

The bounce path is a path with two steps (1, 0) and (0, 1).

If 'direction' is 1 (resp. 0), the bounce path is the path starting at position (h=1, w=0) (resp. (h=0, w=1)) with initial direction, the vector (0, 1) (resp. (1, 0)), and turning each time the path crosses the perimeter of the parallelogram polyomino.

The path is coded by a list of integers. Each integer represents the size of the path between two turnings.

You can visualize the two bounce paths by using the following commands.

INPUT:

- `direction` – the initial direction of the bounce path (see above for the definition)

EXAMPLES:

```
sage: PP = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 1, 1], [1, 1, 0, 0, 1, 0]]
....: )
sage: PP.bounce_path(direction=1)
[2, 2, 1]
sage: PP.bounce_path(direction=0)
[2, 1, 1, 1]

sage: PP = ParallelogramPolyomino(
....:     [
....:         [0, 0, 1, 1, 1, 0, 0, 1, 1],
....:         [1, 1, 1, 0, 1, 1, 0, 0, 0]
....:     ]
....: )
sage: PP.bounce_path(direction=1)
[3, 1, 2, 2]
sage: PP.bounce_path(direction=0)
[2, 4, 2]

sage: PP = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 1, 1], [1, 1, 0, 0, 1, 0]]
....: )
sage: PP.set_options(
....:     drawing_components=dict(
....:         diagram = True
....:         , bounce_0 = True
....:         , bounce_1 = True
```

(continues on next page)

(continued from previous page)

```
....:     )
....: )
sage: view(PP) # not tested

sage: PP = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: PP.bounce_path(direction=1)
[1]
sage: PP.bounce_path(direction=0)
[1]

sage: PP = ParallelogramPolyomino([[1], [1]])
sage: PP.bounce_path(direction=1)
[]
sage: PP.bounce_path(direction=0)
[]
```

```
>>> from sage.all import *
>>> PP = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]]
... )
>>> PP.bounce_path(direction=Integer(1))
[2, 2, 1]
>>> PP.bounce_path(direction=Integer(0))
[2, 1, 1, 1]

>>> PP = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0)]
...     ]
... )
>>> PP.bounce_path(direction=Integer(1))
[3, 1, 2, 2]
>>> PP.bounce_path(direction=Integer(0))
[2, 4, 2]

>>> PP = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(1), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)]]
... )
>>> PP.set_options(
...     drawing_components=dict(
...         diagram = True
...         , bounce_0 = True
...         , bounce_1 = True
...     )
... )
```

(continues on next page)

(continued from previous page)

```
>>> view(PP) # not tested

>>> PP = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> PP.bounce_path(direction=Integer(1))
[1]
>>> PP.bounce_path(direction=Integer(0))
[1]

>>> PP = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> PP.bounce_path(direction=Integer(1))
[]
>>> PP.bounce_path(direction=Integer(0))
[]
```

box_is_node(*pos*)

Return True if the box contains a node in the context of the Aval-Boussicault bijection between parallelogram polyomino and binary tree.

A box is a node if there is no cell on the top of the box in the same column or on the left of the box.in the same row.

INPUT:

- pos – the [x,y] coordinate of the box

OUTPUT: boolean

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 0, 0, 1, 1], [1, 1, 0, 1, 0, 0, 0, 0]]
....: )
sage: pp.set_options(display='drawing')
sage: pp
[1 1 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
sage: pp.box_is_node([2,1])
True
sage: pp.box_is_node([2,0])
False
sage: pp.box_is_node([1,1])
False
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]]
... )
>>> pp.set_options(display='drawing')
```

(continues on next page)

(continued from previous page)

```
>>> pp
[1 1 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
>>> pp.box_is_node([Integer(2),Integer(1)])
True
>>> pp.box_is_node([Integer(2),Integer(0)])
False
>>> pp.box_is_node([Integer(1),Integer(1)])
False
```

box_is_root(*box*)

Return `True` if the box contains the root of the tree : it is the top-left box of the parallelogram polyomino.

INPUT:

- `box` – the x,y coordinate of the cell

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 0, 0, 1, 1], [1, 1, 0, 1, 0, 0, 0, 0]]
....: )
sage: pp.box_is_root([0, 0])
True
sage: pp.box_is_root([0, 1])
False
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]]
... )
>>> pp.box_is_root([Integer(0), Integer(0)])
True
>>> pp.box_is_root([Integer(0), Integer(1)])
False
```

cell_is_inside(*w*, *h*)

Determine whether the cell at a given position is inside the parallelogram polyomino.

INPUT:

- `w` – the x coordinate of the box position
- `h` – the y coordinate of the box position

OUTPUT:

Return 0 if there is no cell at the given position, return 1 if there is a cell.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 1, 0, 0, 1, 1, 0, 1, 1, 1],
....:         [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
....:     ]
....: )
sage: pp.cell_is_inside(0, 0)
1
sage: pp.cell_is_inside(1, 0)
1
sage: pp.cell_is_inside(0, 1)
0
sage: pp.cell_is_inside(3, 0)
0
sage: pp.cell_is_inside(pp.width()-1,pp.height()-1)
1
sage: pp.cell_is_inside(pp.width(),pp.height()-1)
0
sage: pp.cell_is_inside(pp.width()-1,pp.height())
0
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(1), Integer(0), Integer(0), Integer(1),
↪Integer(1), Integer(0), Integer(1), Integer(1), Integer(1)],
...         [Integer(1), Integer(1), Integer(1), Integer(0), Integer(1),
↪Integer(0), Integer(0), Integer(1), Integer(1), Integer(0)]
...     ]
... )
>>> pp.cell_is_inside(Integer(0), Integer(0))
1
>>> pp.cell_is_inside(Integer(1), Integer(0))
1
>>> pp.cell_is_inside(Integer(0), Integer(1))
0
>>> pp.cell_is_inside(Integer(3), Integer(0))
0
>>> pp.cell_is_inside(pp.width()-Integer(1),pp.height()-Integer(1))
1
>>> pp.cell_is_inside(pp.width(),pp.height()-Integer(1))
0
>>> pp.cell_is_inside(pp.width()-Integer(1),pp.height())
0
```

check()

This method raises an error if the internal data of the class does not represent a parallelogram polyomino.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 0, 1, 0, 1, 0, 1, 1],
```

(continues on next page)

(continued from previous page)

```
....:         [1, 0, 1, 1, 0, 0, 1, 0, 0]
....:     ]
....: )
sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp = ParallelogramPolyomino([[1], [1]])

sage: pp = ParallelogramPolyomino(                # indirect doctest
....:     [[1, 0], [0, 1]]
....: )
Traceback (most recent call last):
...
ValueError: the lower and upper paths are crossing

sage: pp = ParallelogramPolyomino([[1], [0, 1]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: the lower and upper paths have different sizes (2 != 1)

sage: pp = ParallelogramPolyomino([[1], [0]])     # indirect doctest
Traceback (most recent call last):
...
ValueError: the two paths have distinct ends

sage: pp = ParallelogramPolyomino([[0], [1]])     # indirect doctest
Traceback (most recent call last):
...
ValueError: the two paths have distinct ends

sage: pp = ParallelogramPolyomino([[0], [0]])     # indirect doctest
Traceback (most recent call last):
...
ValueError: the lower or the upper path can...t be equal to [0]

sage: pp = ParallelogramPolyomino([[], [0]])      # indirect doctest
Traceback (most recent call last):
...
ValueError: the lower or the upper path can...t be equal to []

sage: pp = ParallelogramPolyomino([[0], []])      # indirect doctest
Traceback (most recent call last):
...
ValueError: the lower or the upper path can...t be equal to []

sage: pp = ParallelogramPolyomino([[], []])       # indirect doctest
Traceback (most recent call last):
...
ValueError: the lower or the upper path can...t be equal to []
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0), Integer(0)]
...     ]
... )
>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])

>>> pp = ParallelogramPolyomino(                 # indirect doctest
...     [[Integer(1), Integer(0)], [Integer(0), Integer(1)]]
... )
Traceback (most recent call last):
...
ValueError: the lower and upper paths are crossing

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(0), Integer(1)]])  #␣
↪indirect doctest
Traceback (most recent call last):
...
ValueError: the lower and upper paths have different sizes (2 != 1)

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(0)]])     # indirect␣
↪doctest
Traceback (most recent call last):
...
ValueError: the two paths have distinct ends

>>> pp = ParallelogramPolyomino([[Integer(0)], [Integer(1)]])     # indirect␣
↪doctest
Traceback (most recent call last):
...
ValueError: the two paths have distinct ends

>>> pp = ParallelogramPolyomino([[Integer(0)], [Integer(0)]])     # indirect␣
↪doctest
Traceback (most recent call last):
...
ValueError: the lower or the upper path can...t be equal to [0]

>>> pp = ParallelogramPolyomino([[], [Integer(0)]])      # indirect doctest
Traceback (most recent call last):
...
ValueError: the lower or the upper path can...t be equal to []

>>> pp = ParallelogramPolyomino([[Integer(0)], []])      # indirect doctest
Traceback (most recent call last):
...
ValueError: the lower or the upper path can...t be equal to []

>>> pp = ParallelogramPolyomino([[], []])       # indirect doctest
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: the lower or the upper path can...t be equal to []
```

degree_convexity()

Return the degree convexity of a parallelogram polyomino.

A convex polyomino is said to be k-convex if every pair of its cells can be connected by a monotone path (path with south and east steps) with at most k changes of direction. The degree of convexity of a convex polyomino P is the smallest integer k such that P is k-convex.

If the parallelogram polyomino is empty, the function return -1.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 0, 1, 0, 1, 0, 1, 1],
....:         [1, 0, 1, 1, 0, 0, 1, 0, 0]
....:     ]
....: )
sage: pp.degree_convexity()
3

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.degree_convexity()
0

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.degree_convexity()
-1
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0), Integer(0)]
...     ]
... )
>>> pp.degree_convexity()
3

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.degree_convexity()
0

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.degree_convexity()
-1
```

static **from_dyck_word**(*dyck*, *bijection=None*)

Convert a Dyck word to parallelogram polyomino.

INPUT:

- `dyck` – a Dyck word
- `bijection` – string or `None` (default: `None`); the bijection to use. See `to_dyck_word()` for more details.

OUTPUT: a parallelogram polyomino

EXAMPLES:

```
sage: dyck = DyckWord([1, 1, 0, 1, 1, 0, 1, 0, 0, 0])
sage: ParallelogramPolyomino.from_dyck_word(dyck)
[[0, 1, 0, 0, 1, 1], [1, 1, 1, 0, 0, 0]]
sage: ParallelogramPolyomino.from_dyck_word(dyck, bijection='Delest-Viennot')
[[0, 1, 0, 0, 1, 1], [1, 1, 1, 0, 0, 0]]
sage: ParallelogramPolyomino.from_dyck_word(dyck, bijection='Delest-Viennot-
↪beta')
[[0, 0, 1, 0, 1, 1], [1, 1, 1, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> dyck = DyckWord([Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1), Integer(0), Integer(0), Integer(0)])
>>> ParallelogramPolyomino.from_dyck_word(dyck)
[[0, 1, 0, 0, 1, 1], [1, 1, 1, 0, 0, 0]]
>>> ParallelogramPolyomino.from_dyck_word(dyck, bijection='Delest-Viennot')
[[0, 1, 0, 0, 1, 1], [1, 1, 1, 0, 0, 0]]
>>> ParallelogramPolyomino.from_dyck_word(dyck, bijection='Delest-Viennot-beta
↪')
[[0, 0, 1, 0, 1, 1], [1, 1, 1, 0, 0, 0]]
```

geometry()

Return a pair [h, w] containing the height and the width of the parallelogram polyomino.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0, 1, 1, 1, 1], [1, 1, 1, 1, 0]]
....: )
sage: pp.geometry()
[1, 4]

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.geometry()
[1, 1]

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.geometry()
[0, 1]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(1), Integer(1), Integer(1), Integer(1)],␣
↪[Integer(1), Integer(1), Integer(1), Integer(1), Integer(0)]]
... )
```

(continues on next page)

(continued from previous page)

```
>>> pp.geometry()
[1, 4]

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.geometry()
[1, 1]

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.geometry()
[0, 1]
```

get_BS_nodes()

Return the list of cells containing node of the left and right planar tree in the Boussicault-Socci bijection.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 0, 0, 1, 1], [1, 1, 0, 1, 0, 0, 0, 0]]
....: )
sage: pp.set_options(display='drawing')
sage: pp
[1 1 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
sage: sorted(pp.get_BS_nodes())
[[0, 1], [1, 0], [1, 2], [2, 1], [3, 1], [4, 1]]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]]
... )
>>> pp.set_options(display='drawing')
>>> pp
[1 1 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
>>> sorted(pp.get_BS_nodes())
[[0, 1], [1, 0], [1, 2], [2, 1], [3, 1], [4, 1]]
```

You can draw the point inside the parallelogram polyomino by typing (the left nodes are in blue, and the right node are in red)

```
sage: pp.set_options(drawing_components=dict(tree=True))
sage: view(pp) # not tested
```

```
>>> from sage.all import *
>>> pp.set_options(drawing_components=dict(tree=True))
>>> view(pp) # not tested
```

get_array()

Return an array of 0s and 1s such that the 1s represent the boxes of the parallelogram polyomino.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 0, 0, 1, 0, 1, 0, 1],
....:         [1, 0, 0, 0, 1, 1, 0, 0, 0]
....:     ]
....: )
sage: matrix(pp.get_array())
[1 0 0]
[1 0 0]
[1 0 0]
[1 1 1]
[0 1 1]
[0 0 1]

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.get_array()
[[1]]

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.get_array()
[]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(0), Integer(1)],
...         [Integer(1), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0)]
...     ]
... )
>>> matrix(pp.get_array())
[1 0 0]
[1 0 0]
[1 0 0]
[1 1 1]
[0 1 1]
[0 0 1]

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.get_array()
[[1]]
```

(continues on next page)

(continued from previous page)

```
>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.get_array()
[]
```

get_left_BS_nodes()

Return the list of cells containing node of the left planar tree in the Boussicault-Socci bijection between parallelogram polyominoes and pair of ordered trees.

OUTPUT: list of [row,column] position of cells

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 0, 0, 1, 1], [1, 1, 0, 1, 0, 0, 0, 0]]
....: )
sage: pp.set_options(display='drawing')
sage: pp
[1 1 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
sage: sorted(pp.get_left_BS_nodes())
[[0, 1], [2, 1], [3, 1], [4, 1]]

sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 0, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0]]
....: )
sage: pp.set_options(display='drawing')
sage: pp
[1 0 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
sage: sorted(pp.get_left_BS_nodes())
[]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]]
... )
>>> pp.set_options(display='drawing')
>>> pp
[1 1 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
>>> sorted(pp.get_left_BS_nodes())
[[0, 1], [2, 1], [3, 1], [4, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]]
... )
>>> pp.set_options(display='drawing')
>>> pp
[1 0 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
>>> sorted(pp.get_left_BS_nodes())
[]
```

You can draw the point inside the parallelogram polyomino by typing (the left nodes are in blue, and the right node are in red)

```
sage: pp.set_options(drawing_components=dict(tree=True))
sage: view(pp) # not tested
```

```
>>> from sage.all import *
>>> pp.set_options(drawing_components=dict(tree=True))
>>> view(pp) # not tested
```

get_node_position_from_box(*box_position*, *direction*, *nb_crossed_nodes=None*)

This function starts from a cell inside a parallelogram polyomino and a direction.

If `direction` is equal to 0, the function selects the column associated with the y-coordinate of `box_position` and then returns the topmost cell of the column that is on the top of `box_position` (the cell of `box_position` is included).

If `direction` is equal to 1, the function selects the row associated with the x-coordinate of `box_position` and then returns the leftmost cell of the row that is on the left of `box_position`. (the cell of `box_position` is included).

This function updates the entry of `nb_crossed_nodes`. The function increases the entry of `nb_crossed_nodes` by the number of boxes that is a node (see `box_is_node`) located on the top if `direction` is 0 (resp. on the left if `direction` is 1) of `box_position` (cell at `box_position` is excluded).

INPUT:

- `box_position` – the position of the starting cell
- `direction` – the direction (0 or 1)
- `nb_crossed_nodes` – `[0]` (default) a list containing just one integer

OUTPUT: a [row,column] position of the cell

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 0, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0]]
....: )
```

(continues on next page)

(continued from previous page)

```
sage: matrix(pp.get_array())
[1 0 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
sage: l = [0]
sage: pp.get_node_position_from_box([3, 2], 0, l)
[1, 2]
sage: l
[1]
sage: l = [0]
sage: pp.get_node_position_from_box([3, 2], 1, l)
[3, 1]
sage: l
[1]
sage: l = [0]
sage: pp.get_node_position_from_box([1, 2], 0, l)
[1, 2]
sage: l
[0]
sage: l = [0]
sage: pp.get_node_position_from_box([1, 2], 1, l)
[1, 0]
sage: l
[2]
sage: l = [0]
sage: pp.get_node_position_from_box([3, 1], 0, l)
[1, 1]
sage: l
[2]
sage: l = [0]
sage: pp.get_node_position_from_box([3, 1], 1, l)
[3, 1]
sage: l
[0]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]]
... )
>>> matrix(pp.get_array())
[1 0 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
>>> l = [Integer(0)]
>>> pp.get_node_position_from_box([Integer(3), Integer(2)], Integer(0), l)
[1, 2]
```

(continues on next page)

(continued from previous page)

```
>>> l
[1]
>>> l = [Integer(0)]
>>> pp.get_node_position_from_box([Integer(3), Integer(2)], Integer(1), l)
[3, 1]
>>> l
[1]
>>> l = [Integer(0)]
>>> pp.get_node_position_from_box([Integer(1), Integer(2)], Integer(0), l)
[1, 2]
>>> l
[0]
>>> l = [Integer(0)]
>>> pp.get_node_position_from_box([Integer(1), Integer(2)], Integer(1), l)
[1, 0]
>>> l
[2]
>>> l = [Integer(0)]
>>> pp.get_node_position_from_box([Integer(3), Integer(1)], Integer(0), l)
[1, 1]
>>> l
[2]
>>> l = [Integer(0)]
>>> pp.get_node_position_from_box([Integer(3), Integer(1)], Integer(1), l)
[3, 1]
>>> l
[0]
```

get_options()

Return all the options of the object.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.get_options()
Current options for ParallelogramPolyominoes_size
  - display:            'list'
  - drawing_components: {'bounce_0': False,
 'bounce_1': False,
 'bounce_values': False,
 'diagram': True,
 'tree': False}
  - latex:              'drawing'
  - tikz_options:       {'color_bounce_0': 'red',
 'color_bounce_1': 'blue',
 'color_line': 'black',
 'color_point': 'black',
 'line_size': 1,
 'mirror': None,
 'point_size': 3.5,
 'rotation': 0,
 'scale': 1,
 'translation': [0, 0]}
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.get_options()
Current options for ParallelogramPolyominoes_size
  - display:            'list'
  - drawing_components: {'bounce_0': False,
 'bounce_1': False,
 'bounce_values': False,
 'diagram': True,
 'tree': False}
  - latex:              'drawing'
  - tikz_options:       {'color_bounce_0': 'red',
 'color_bounce_1': 'blue',
 'color_line': 'black',
 'color_point': 'black',
 'line_size': 1,
 'mirror': None,
 'point_size': 3.5,
 'rotation': 0,
 'scale': 1,
 'translation': [0, 0]}
```

get_right_BS_nodes()

Return the list of cells containing node of the right planar tree in the Boussicault-Socci bijection between parallelogram polyominoes and pair of ordered trees.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 0, 0, 1, 1], [1, 1, 0, 1, 0, 0, 0, 0]]
....: )
sage: pp.set_options(display='drawing')
sage: pp
[1 1 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
sage: sorted(pp.get_right_BS_nodes())
[[1, 0], [1, 2]]

sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 1, 0, 0, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0]]
....: )
sage: pp.set_options(display='drawing')
sage: pp
[1 0 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
sage: sorted(pp.get_right_BS_nodes())
```

(continues on next page)

(continued from previous page)

```
[[1, 0], [1, 1], [1, 2], [2, 1], [3, 1], [4, 1]]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]]
... )
>>> pp.set_options(display='drawing')
>>> pp
[1 1 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
>>> sorted(pp.get_right_BS_nodes())
[[1, 0], [1, 2]]

>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0), Integer(0)]]
... )
>>> pp.set_options(display='drawing')
>>> pp
[1 0 0]
[1 1 1]
[0 1 1]
[0 1 1]
[0 1 1]
>>> sorted(pp.get_right_BS_nodes())
[[1, 0], [1, 1], [1, 2], [2, 1], [3, 1], [4, 1]]
```

You can draw the point inside the parallelogram polyomino by typing, (the left nodes are in blue, and the right node are in red)

```
sage: pp.set_options(drawing_components=dict(tree=True))
sage: view(pp) # not tested
```

```
>>> from sage.all import *
>>> pp.set_options(drawing_components=dict(tree=True))
>>> view(pp) # not tested
```

get_tikz_options()

Return all the tikz options permitting to draw the parallelogram polyomino.

See `LocalOption` to have more informations about the modification of those options.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.get_tikz_options()
{'color_bounce_0': 'red',
```

(continues on next page)

(continued from previous page)

```
 'color_bounce_1': 'blue',
 'color_line': 'black',
 'color_point': 'black',
 'line_size': 1,
 'mirror': None,
 'point_size': 3.5,
 'rotation': 0,
 'scale': 1,
 'translation': [0, 0]}
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.get_tikz_options()
{'color_bounce_0': 'red',
 'color_bounce_1': 'blue',
 'color_line': 'black',
 'color_point': 'black',
 'line_size': 1,
 'mirror': None,
 'point_size': 3.5,
 'rotation': 0,
 'scale': 1,
 'translation': [0, 0]}
```

height()

Return the height of the parallelogram polyomino.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 1, 0, 0, 1, 1, 0, 1, 1, 1],
....:         [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
....:     ]
....: )
sage: pp.height()
4

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.height()
1

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.height()
0
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1), Integer(1), Integer(1)],
```

(continues on next page)

(continued from previous page)

```
...         [Integer(1), Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1), Integer(1), Integer(0)]
...     ]
... )
>>> pp.height()
4

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.height()
1

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.height()
0
```

heights()

Return a list of heights of the parallelogram polyomino.

Namely, the parallelogram polyomino is split column by column and the method returns the list containing the sizes of the columns.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 0, 1, 0, 1, 0, 1, 1],
....:         [1, 0, 1, 1, 0, 0, 1, 0, 0]
....:     ]
....: )
sage: pp.heights()
[3, 3, 4, 2]

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.heights()
[1]

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.heights()
[0]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0), Integer(0)]
...     ]
... )
>>> pp.heights()
[3, 3, 4, 2]
```

(continues on next page)

(continued from previous page)

```
>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.heights()
[1]

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.heights()
[0]
```

is_flat()

Return whether the two bounce paths join together in the rightmost cell of the bottom row of P.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 0, 1, 0, 1, 0, 1, 1],
....:         [1, 0, 1, 1, 0, 0, 1, 0, 0]
....:     ]
....: )
sage: pp.is_flat()
False

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.is_flat()
True

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.is_flat()
True
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0), Integer(0)]
...     ]
... )
>>> pp.is_flat()
False

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.is_flat()
True

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.is_flat()
True
```

is_k_directed(k)

Return whether the Polyomino Parallelogram is k-directed.

A convex polyomino is said to be k-convex if every pair of its cells can be connected by a monotone path (path with south and east steps) with at most k changes of direction.

The degree of convexity of a convex polyomino P is the smallest integer k such that P is k-convex.

INPUT:

- `k` – nonnegative integer

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 0, 1, 0, 1, 0, 1, 1],
....:         [1, 0, 1, 1, 0, 0, 1, 0, 0]
....:     ]
....: )
sage: pp.is_k_directed(3)
True
sage: pp.is_k_directed(4)
True
sage: pp.is_k_directed(5)
True
sage: pp.is_k_directed(0)
False
sage: pp.is_k_directed(1)
False
sage: pp.is_k_directed(2)
False

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.is_k_directed(0)
True
sage: pp.is_k_directed(1)
True

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.is_k_directed(0)
True
sage: pp.is_k_directed(1)
True
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),
↪Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),
↪Integer(0), Integer(1), Integer(0), Integer(0)]
...     ]
... )
>>> pp.is_k_directed(Integer(3))
True
```

(continues on next page)

(continued from previous page)

```
>>> pp.is_k_directed(Integer(4))
True
>>> pp.is_k_directed(Integer(5))
True
>>> pp.is_k_directed(Integer(0))
False
>>> pp.is_k_directed(Integer(1))
False
>>> pp.is_k_directed(Integer(2))
False

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.is_k_directed(Integer(0))
True
>>> pp.is_k_directed(Integer(1))
True

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.is_k_directed(Integer(0))
True
>>> pp.is_k_directed(Integer(1))
True
```

lower_heights()

Return the list of heights associated to each vertical step of the parallelogram polyomino's lower path.

OUTPUT: list of integers

EXAMPLES:

```
sage: ParallelogramPolyomino([[0, 1], [1, 0]]).lower_heights()
[1]
sage: ParallelogramPolyomino(
....:     [[0, 0, 1, 1, 0, 1, 1, 1], [1, 0, 1, 1, 0, 1, 1, 0]]
....: ).lower_heights()
[2, 2, 3, 3, 3]
```

```
>>> from sage.all import *
>>> ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]]).lower_heights()
[1]
>>> ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)], [Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1), Integer(1), Integer(0)]]
... ).lower_heights()
[2, 2, 3, 3, 3]
```

lower_path()

Get the lower path of the parallelogram polyomino.

EXAMPLES:

```
sage: lower_path = [0, 0, 1, 0, 1, 1]
sage: upper_path = [1, 1, 0, 1, 0, 0]
sage: pp = ParallelogramPolyomino([lower_path, upper_path])
sage: pp.lower_path()
[0, 0, 1, 0, 1, 1]
```

```
>>> from sage.all import *
>>> lower_path = [Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(1)]
>>> upper_path = [Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(0)]
>>> pp = ParallelogramPolyomino([lower_path, upper_path])
>>> pp.lower_path()
[0, 0, 1, 0, 1, 1]
```

lower_widths()

Return the list of widths associated to each horizontal step of the parallelogram polyomino's lower path.

OUTPUT: list of integers

EXAMPLES:

```
sage: ParallelogramPolyomino([[0, 1], [1, 0]]).lower_widths()
[0]
sage: ParallelogramPolyomino(
....:     [[0, 0, 1, 1, 0, 1, 1, 1], [1, 0, 1, 1, 0, 1, 1, 0]]
....: ).lower_widths()
[0, 0, 2]
```

```
>>> from sage.all import *
>>> ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]]).lower_widths()
[0]
>>> ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)], [Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1), Integer(1), Integer(0)]]
... ).lower_widths()
[0, 0, 2]
```

plot()

Return a plot of `self`.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino([[0,1],[1,0]])
sage: pp.plot()                                                         #␣
↪needs sage.plot
Graphics object consisting of 4 graphics primitives
sage: pp.set_options(
....:     drawing_components=dict(
....:         diagram=True,
....:         bounce_0=True,
....:         bounce_1=True,
```

(continues on next page)

(continued from previous page)

```
....:         bounce_values=0,
....:     )
....: )
sage: pp.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 7 graphics primitives
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino([[Integer(0),Integer(1)],[Integer(1),
↪Integer(0)]])
>>> pp.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 4 graphics primitives
>>> pp.set_options(
...     drawing_components=dict(
...         diagram=True,
...         bounce_0=True,
...         bounce_1=True,
...         bounce_values=Integer(0),
...     )
... )
>>> pp.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 7 graphics primitives
```

reflect()

Return the parallelogram polyomino obtained by switching rows and columns.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino([[0,0,0,0,1,1,0,1,0,1], [1,0,1,0,0,1,1,0,0,
↪0]])
sage: pp.heights(), pp.upper_heights()
([4, 3, 2, 3], [0, 1, 3, 3])
sage: pp = pp.reflect()
sage: pp.widths(), pp.lower_widths()
([4, 3, 2, 3], [0, 1, 3, 3])

sage: pp = ParallelogramPolyomino([[0,0,0,1,1], [1,0,0,1,0]])
sage: ascii_art(pp)
*
*
**
sage: ascii_art(pp.reflect())
***
  *
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino([[Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),Integer(1)],␣
↪[Integer(1),Integer(0),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0),Integer(0)]])
```

(continues on next page)

(continued from previous page)

```
>>> pp.heights(), pp.upper_heights()
([4, 3, 2, 3], [0, 1, 3, 3])
>>> pp = pp.reflect()
>>> pp.widths(), pp.lower_widths()
([4, 3, 2, 3], [0, 1, 3, 3])

>>> pp = ParallelogramPolyomino([[Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(1)], [Integer(1),Integer(0),Integer(0),Integer(1),Integer(0)]])
>>> ascii_art(pp)
*
*
**
>>> ascii_art(pp.reflect())
***
  *
```

rotate()

Return the parallelogram polyomino obtained by rotation of 180 degrees.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino([[0,0,0,1,1], [1,0,0,1,0]])
sage: ascii_art(pp)
*
*
**
sage: ascii_art(pp.rotate())
**
 *
 *
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino([[Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(1)], [Integer(1),Integer(0),Integer(0),Integer(1),Integer(0)]])
>>> ascii_art(pp)
*
*
**
>>> ascii_art(pp.rotate())
**
 *
 *
```

set_options(**get_value*, ***set_value*)

Set new options to the object. See `LocalOptions` for more info.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 0, 0, 1, 0, 1, 0, 1],
....:         [1, 0, 0, 0, 1, 1, 0, 0, 0]
....:     ]
```

(continues on next page)

(continued from previous page)

```
....: )
sage: pp
[[0, 0, 0, 0, 1, 0, 1, 0, 1], [1, 0, 0, 0, 1, 1, 0, 0, 0]]
sage: pp.set_options(display='drawing')
sage: pp
[1 0 0]
[1 0 0]
[1 0 0]
[1 1 1]
[0 1 1]
[0 0 1]

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: view(PP) # not tested
sage: pp.set_options(
....:     drawing_components=dict(
....:         diagram = True,
....:         bounce_0 = True,
....:         bounce_1 = True,
....:     )
....: )
sage: view(PP) # not tested
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(0), Integer(1)],
...         [Integer(1), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(0), Integer(0)]
...     ]
... )
>>> pp
[[0, 0, 0, 0, 1, 0, 1, 0, 1], [1, 0, 0, 0, 1, 1, 0, 0, 0]]
>>> pp.set_options(display='drawing')
>>> pp
[1 0 0]
[1 0 0]
[1 0 0]
[1 1 1]
[0 1 1]
[0 0 1]

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> view(PP) # not tested
>>> pp.set_options(
...     drawing_components=dict(
...         diagram = True,
...         bounce_0 = True,
...         bounce_1 = True,
...     )
```

(continues on next page)

(continued from previous page)

```
... )
>>> view(PP) # not tested
```

size()

Return the size of the parallelogram polyomino.

The size of a parallelogram polyomino is its half-perimeter.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0, 0, 0, 0, 1, 0, 1, 1], [1, 0, 0, 0, 1, 1, 0, 0]]
....: )
sage: pp.size()
8

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.size()
2

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.size()
1
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(0), Integer(0)]]
... )
>>> pp.size()
8

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.size()
2

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.size()
1
```

to_binary_tree(*bijection=None*)

Convert to a binary tree.

INPUT:

- `bijection` – string or `None` (default: `None`); the name of bijection to use for the conversion. The possible values are `None` or `'Aval-Boussicault'`. The `None` value is equivalent to `'Aval-Boussicault'`.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 1, 0, 1, 0, 1, 0, 1, 1],
....:         [1, 1, 0, 1, 1, 0, 0, 0, 1, 0]
....:     ]
....: )
sage: pp.to_binary_tree()
[[., [[., .], [[., [., .]], .]]], [[., .], .]]

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.to_binary_tree()
[., .]

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.to_binary_tree()
.
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),
↪Integer(0), Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),
↪Integer(0), Integer(0), Integer(0), Integer(1), Integer(0)]
...     ]
... )
>>> pp.to_binary_tree()
[[., [[., .], [[., [., .]], .]]], [[., .], .]]

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),
↪Integer(0)]])
>>> pp.to_binary_tree()
[., .]

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.to_binary_tree()
.
```

to_dyck_word(*bijection=None*)

Convert to a Dyck word.

INPUT:

- `bijection` – string or `None` (default: `None`); the name of the bijection. If it is set to `None` then the `'Delest-Viennot'` bijection is used. Expected values are `None`, `'Delest-Viennot'`, or `'Delest-Viennot-beta'`.

OUTPUT: a Dyck word

EXAMPLES:

```
sage: pp = ParallelogramPolyomino([[0, 1, 0, 0, 1, 1], [1, 1, 1, 0, 0, 0]])
sage: pp.to_dyck_word()
[1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
```

(continues on next page)

(continued from previous page)

```
sage: pp.to_dyck_word(bijection='Delest-Viennot')
[1, 1, 0, 1, 1, 0, 1, 0, 0, 0]

sage: pp.to_dyck_word(bijection='Delest-Viennot-beta')
[1, 0, 1, 1, 1, 0, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1), Integer(0),␣
→Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1), Integer(1),␣
→Integer(0), Integer(0), Integer(0)]])
>>> pp.to_dyck_word()
[1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
>>> pp.to_dyck_word(bijection='Delest-Viennot')
[1, 1, 0, 1, 1, 0, 1, 0, 0, 0]

>>> pp.to_dyck_word(bijection='Delest-Viennot-beta')
[1, 0, 1, 1, 1, 0, 1, 0, 0, 0]
```

to_ordered_tree(*bijection=None*)

Return an ordered tree from the parallelogram polyomino.

Different bijections can be specified.

The bijection 'via dyck and Delest-Viennot' is the composition of `_to_dyck_delest_viennot()` and the classical bijection between dyck paths and ordered trees.

The bijection between Dyck Word and ordered trees is described in [DerZak1980] (See page 12 and 13 and Figure 3.1).

The bijection 'Boussicault-Socci' is described in [BRS2015].

INPUT:

- `bijection` – string or `None` (default: `None`); the name of bijection to use for the conversion. The possible value are `None`, `'Boussicault-Socci'` or `'via dyck and Delest-Viennot'`. The `None` value is equivalent to the `'Boussicault-Socci'` value.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 1, 0, 1, 0, 1, 0, 1, 1],
....:         [1, 1, 0, 1, 1, 0, 0, 0, 1, 0]
....:     ]
....: )
sage: pp.to_ordered_tree()
[[[[[]], [[[]]]]], [[]]]

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.to_ordered_tree()
[[]]

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.to_ordered_tree()
[]
```

(continues on next page)

(continued from previous page)

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 1, 0, 1, 0, 1, 0, 1, 1],
....:         [1, 1, 0, 1, 1, 0, 0, 0, 1, 0]
....:     ]
....: )
sage: pp.to_ordered_tree('via dyck and Delest-Viennot')
[[[[]], [[[]], []]], [[]]]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(0), Integer(0), Integer(0), Integer(1), Integer(0)]
...     ]
... )
>>> pp.to_ordered_tree()
[[[[[]], [[[]]]]], [[]]]

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.to_ordered_tree()
[[]]

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.to_ordered_tree()
[]

>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(0), Integer(0), Integer(0), Integer(1), Integer(0)]
...     ]
... )
>>> pp.to_ordered_tree('via dyck and Delest-Viennot')
[[[[]], [[[]], []]], [[]]]
```

to_tikz()

Return the tikz code of the parallelogram polyomino.

This code is the code present inside a tikz latex environment.

We can modify the output with the options.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [[0,0,0,1,1,0,1,0,0,1,1,1],[1,1,1,0,0,1,1,0,0,1,0,0]]
```

(continues on next page)

(continued from previous page)

```
....: )
sage: print(pp.to_tikz())

  \draw[color=black, line width=1] (0.000000, 6.000000) --
(0.000000, 3.000000);
  \draw[color=black, line width=1] (6.000000, 2.000000) --
(6.000000, 0.000000);
  \draw[color=black, line width=1] (0.000000, 6.000000) --
(3.000000, 6.000000);
  \draw[color=black, line width=1] (3.000000, 0.000000) --
(6.000000, 0.000000);
  \draw[color=black, line width=1] (1.000000, 6.000000) --
(1.000000, 3.000000);
  \draw[color=black, line width=1] (2.000000, 6.000000) --
(2.000000, 2.000000);
  \draw[color=black, line width=1] (3.000000, 6.000000) --
(3.000000, 0.000000);
  \draw[color=black, line width=1] (4.000000, 4.000000) --
(4.000000, 0.000000);
  \draw[color=black, line width=1] (5.000000, 4.000000) --
(5.000000, 0.000000);
  \draw[color=black, line width=1] (0.000000, 5.000000) --
(3.000000, 5.000000);
  \draw[color=black, line width=1] (0.000000, 4.000000) --
(5.000000, 4.000000);
  \draw[color=black, line width=1] (0.000000, 3.000000) --
(5.000000, 3.000000);
  \draw[color=black, line width=1] (2.000000, 2.000000) --
(6.000000, 2.000000);
  \draw[color=black, line width=1] (3.000000, 1.000000) --
(6.000000, 1.000000);
sage: pp.set_options(
....:     drawing_components=dict(
....:         diagram=True,
....:         tree=True,
....:         bounce_0=True,
....:         bounce_1=True
....:     )
....: )
sage: print(pp.to_tikz())

  \draw[color=black, line width=1] (0.000000, 6.000000) --
(0.000000, 3.000000);
  \draw[color=black, line width=1] (6.000000, 2.000000) --
(6.000000, 0.000000);
  \draw[color=black, line width=1] (0.000000, 6.000000) --
(3.000000, 6.000000);
  \draw[color=black, line width=1] (3.000000, 0.000000) --
(6.000000, 0.000000);
  \draw[color=black, line width=1] (1.000000, 6.000000) --
(1.000000, 3.000000);
  \draw[color=black, line width=1] (2.000000, 6.000000) --
```

(continues on next page)

(continued from previous page)

```
(2.000000, 2.000000);
  \draw[color=black, line width=1] (3.000000, 6.000000) --
(3.000000, 0.000000);
  \draw[color=black, line width=1] (4.000000, 4.000000) --
(4.000000, 0.000000);
  \draw[color=black, line width=1] (5.000000, 4.000000) --
(5.000000, 0.000000);
  \draw[color=black, line width=1] (0.000000, 5.000000) --
(3.000000, 5.000000);
  \draw[color=black, line width=1] (0.000000, 4.000000) --
(5.000000, 4.000000);
  \draw[color=black, line width=1] (0.000000, 3.000000) --
(5.000000, 3.000000);
  \draw[color=black, line width=1] (2.000000, 2.000000) --
(6.000000, 2.000000);
  \draw[color=black, line width=1] (3.000000, 1.000000) --
(6.000000, 1.000000);
  \draw[color=blue, line width=3] (0.000000, 5.000000) --
(3.000000, 5.000000);
  \draw[color=blue, line width=3] (3.000000, 5.000000) --
(3.000000, 2.000000);
  \draw[color=blue, line width=3] (3.000000, 2.000000) --
(5.000000, 2.000000);
  \draw[color=blue, line width=3] (5.000000, 2.000000) --
(5.000000, 0.000000);
  \draw[color=blue, line width=3] (5.000000, 0.000000) --
(6.000000, 0.000000);
  \draw[color=red, line width=2] (1.000000, 6.000000) --
(1.000000, 3.000000);
  \draw[color=red, line width=2] (1.000000, 3.000000) --
(5.000000, 3.000000);
  \draw[color=red, line width=2] (5.000000, 3.000000) --
(5.000000, 0.000000);
  \draw[color=red, line width=2] (5.000000, 0.000000) --
(6.000000, 0.000000);
  \filldraw[color=black] (0.500000, 4.500000) circle (3.5pt);
  \filldraw[color=black] (0.500000, 3.500000) circle (3.5pt);
  \filldraw[color=black] (2.500000, 2.500000) circle (3.5pt);
  \filldraw[color=black] (3.500000, 1.500000) circle (3.5pt);
  \filldraw[color=black] (3.500000, 0.500000) circle (3.5pt);
  \filldraw[color=black] (1.500000, 5.500000) circle (3.5pt);
  \filldraw[color=black] (2.500000, 5.500000) circle (3.5pt);
  \filldraw[color=black] (3.500000, 3.500000) circle (3.5pt);
  \filldraw[color=black] (4.500000, 3.500000) circle (3.5pt);
  \filldraw[color=black] (5.500000, 1.500000) circle (3.5pt);
  \filldraw[color=black] (0.500000, 5.500000) circle (3.5pt);
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [[Integer(0),Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(0),Integer(1),Integer(1),Integer(1)],
↪[Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(0),Integer(0),Integer(1),Integer(0),Integer(0)]]
... )
>>> print(pp.to_tikz())
<BLANKLINE>
  \draw[color=black, line width=1] (0.000000, 6.000000) --
(0.000000, 3.000000);
  \draw[color=black, line width=1] (6.000000, 2.000000) --
(6.000000, 0.000000);
  \draw[color=black, line width=1] (0.000000, 6.000000) --
(3.000000, 6.000000);
  \draw[color=black, line width=1] (3.000000, 0.000000) --
(6.000000, 0.000000);
  \draw[color=black, line width=1] (1.000000, 6.000000) --
(1.000000, 3.000000);
  \draw[color=black, line width=1] (2.000000, 6.000000) --
(2.000000, 2.000000);
  \draw[color=black, line width=1] (3.000000, 6.000000) --
(3.000000, 0.000000);
  \draw[color=black, line width=1] (4.000000, 4.000000) --
(4.000000, 0.000000);
  \draw[color=black, line width=1] (5.000000, 4.000000) --
(5.000000, 0.000000);
  \draw[color=black, line width=1] (0.000000, 5.000000) --
(3.000000, 5.000000);
  \draw[color=black, line width=1] (0.000000, 4.000000) --
(5.000000, 4.000000);
  \draw[color=black, line width=1] (0.000000, 3.000000) --
(5.000000, 3.000000);
  \draw[color=black, line width=1] (2.000000, 2.000000) --
(6.000000, 2.000000);
  \draw[color=black, line width=1] (3.000000, 1.000000) --
(6.000000, 1.000000);
>>> pp.set_options(
...     drawing_components=dict(
...         diagram=True,
...         tree=True,
...         bounce_0=True,
...         bounce_1=True
...     )
... )
>>> print(pp.to_tikz())
<BLANKLINE>
  \draw[color=black, line width=1] (0.000000, 6.000000) --
(0.000000, 3.000000);
  \draw[color=black, line width=1] (6.000000, 2.000000) --
(6.000000, 0.000000);
  \draw[color=black, line width=1] (0.000000, 6.000000) --
(3.000000, 6.000000);
  \draw[color=black, line width=1] (3.000000, 0.000000) --
(6.000000, 0.000000);
  \draw[color=black, line width=1] (1.000000, 6.000000) --
(1.000000, 3.000000);
```

(continues on next page)

(continued from previous page)

```
  \draw[color=black, line width=1] (2.000000, 6.000000) --
(2.000000, 2.000000);
  \draw[color=black, line width=1] (3.000000, 6.000000) --
(3.000000, 0.000000);
  \draw[color=black, line width=1] (4.000000, 4.000000) --
(4.000000, 0.000000);
  \draw[color=black, line width=1] (5.000000, 4.000000) --
(5.000000, 0.000000);
  \draw[color=black, line width=1] (0.000000, 5.000000) --
(3.000000, 5.000000);
  \draw[color=black, line width=1] (0.000000, 4.000000) --
(5.000000, 4.000000);
  \draw[color=black, line width=1] (0.000000, 3.000000) --
(5.000000, 3.000000);
  \draw[color=black, line width=1] (2.000000, 2.000000) --
(6.000000, 2.000000);
  \draw[color=black, line width=1] (3.000000, 1.000000) --
(6.000000, 1.000000);
  \draw[color=blue, line width=3] (0.000000, 5.000000) --
(3.000000, 5.000000);
  \draw[color=blue, line width=3] (3.000000, 5.000000) --
(3.000000, 2.000000);
  \draw[color=blue, line width=3] (3.000000, 2.000000) --
(5.000000, 2.000000);
  \draw[color=blue, line width=3] (5.000000, 2.000000) --
(5.000000, 0.000000);
  \draw[color=blue, line width=3] (5.000000, 0.000000) --
(6.000000, 0.000000);
  \draw[color=red, line width=2] (1.000000, 6.000000) --
(1.000000, 3.000000);
  \draw[color=red, line width=2] (1.000000, 3.000000) --
(5.000000, 3.000000);
  \draw[color=red, line width=2] (5.000000, 3.000000) --
(5.000000, 0.000000);
  \draw[color=red, line width=2] (5.000000, 0.000000) --
(6.000000, 0.000000);
  \filldraw[color=black] (0.500000, 4.500000) circle (3.5pt);
  \filldraw[color=black] (0.500000, 3.500000) circle (3.5pt);
  \filldraw[color=black] (2.500000, 2.500000) circle (3.5pt);
  \filldraw[color=black] (3.500000, 1.500000) circle (3.5pt);
  \filldraw[color=black] (3.500000, 0.500000) circle (3.5pt);
  \filldraw[color=black] (1.500000, 5.500000) circle (3.5pt);
  \filldraw[color=black] (2.500000, 5.500000) circle (3.5pt);
  \filldraw[color=black] (3.500000, 3.500000) circle (3.5pt);
  \filldraw[color=black] (4.500000, 3.500000) circle (3.5pt);
  \filldraw[color=black] (5.500000, 1.500000) circle (3.5pt);
  \filldraw[color=black] (0.500000, 5.500000) circle (3.5pt);
```

upper_heights()

Return the list of heights associated to each vertical step of the parallelogram polyomino's upper path.

OUTPUT: list of integers

EXAMPLES:

```
sage: ParallelogramPolyomino([[0, 1], [1, 0]]).upper_heights()
[0]
sage: ParallelogramPolyomino(
....:     [[0, 0, 1, 1, 0, 1, 1, 1], [1, 0, 1, 1, 0, 1, 1, 0]]
....: ).upper_heights()
[0, 1, 1, 2, 2]
```

```
>>> from sage.all import *
>>> ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]]).upper_heights()
[0]
>>> ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)], [Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1), Integer(1), Integer(0)]]
... ).upper_heights()
[0, 1, 1, 2, 2]
```

upper_path()

Get the upper path of the parallelogram polyomino.

EXAMPLES:

```
sage: lower_path = [0, 0, 1, 0, 1, 1]
sage: upper_path = [1, 1, 0, 1, 0, 0]
sage: pp = ParallelogramPolyomino([lower_path, upper_path])
sage: pp.upper_path()
[1, 1, 0, 1, 0, 0]
```

```
>>> from sage.all import *
>>> lower_path = [Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(1)]
>>> upper_path = [Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(0)]
>>> pp = ParallelogramPolyomino([lower_path, upper_path])
>>> pp.upper_path()
[1, 1, 0, 1, 0, 0]
```

upper_widths()

Return the list of widths associated to each horizontal step of the parallelogram polyomino's upper path.

OUTPUT: list of integers

EXAMPLES:

```
sage: ParallelogramPolyomino([[0, 1], [1, 0]]).upper_widths()
[1]
sage: ParallelogramPolyomino(
....:     [[0, 0, 1, 1, 0, 1, 1, 1], [1, 0, 1, 1, 0, 1, 1, 0]]
....: ).upper_widths()
[1, 3, 5]
```

```
>>> from sage.all import *
>>> ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]]).upper_widths()
[1]
>>> ParallelogramPolyomino(
...     [[Integer(0), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)], [Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1), Integer(1), Integer(0)]]
... ).upper_widths()
[1, 3, 5]
```

width()

Return the width of the parallelogram polyomino.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 1, 0, 0, 1, 1, 0, 1, 1, 1],
....:         [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
....:     ]
....: )
sage: pp.width()
6

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.width()
1

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.width()
1
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1), Integer(1), Integer(1)],
...         [Integer(1), Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1), Integer(1), Integer(0)]
...     ]
... )
>>> pp.width()
6

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.width()
1

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.width()
1
```

widths()

Return a list of the widths of the parallelogram polyomino.

Namely, the parallelogram polyomino is split row by row and the method returns the list containing the sizes of the rows.

EXAMPLES:

```
sage: pp = ParallelogramPolyomino(
....:     [
....:         [0, 0, 0, 1, 0, 1, 0, 1, 1],
....:         [1, 0, 1, 1, 0, 0, 1, 0, 0]
....:     ]
....: )
sage: pp.widths()
[1, 3, 3, 3, 2]

sage: pp = ParallelogramPolyomino([[0, 1], [1, 0]])
sage: pp.widths()
[1]

sage: pp = ParallelogramPolyomino([[1], [1]])
sage: pp.widths()
[]
```

```
>>> from sage.all import *
>>> pp = ParallelogramPolyomino(
...     [
...         [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1)],
...         [Integer(1), Integer(0), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0), Integer(0)]
...     ]
... )
>>> pp.widths()
[1, 3, 3, 3, 2]

>>> pp = ParallelogramPolyomino([[Integer(0), Integer(1)], [Integer(1),␣
↪Integer(0)]])
>>> pp.widths()
[1]

>>> pp = ParallelogramPolyomino([[Integer(1)], [Integer(1)]])
>>> pp.widths()
[]
```

sage.combinat.parallelogram_polyomino.**ParallelogramPolyominoes**(*size=None*, *policy=None*)

Return a family of parallelogram polyominoes enumerated with the parameter constraints.

INPUT:

- **size – integer (default: None); the size of the parallelogram** polyominoes contained in the family. If set to None, the family returned contains all the parallelogram polyominoes.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes(size=4)
sage: PPS
Parallelogram polyominoes of size 4
sage: sorted(PPS)
[[[0, 0, 0, 1], [1, 0, 0, 0]],
 [[0, 0, 1, 1], [1, 0, 1, 0]],
 [[0, 0, 1, 1], [1, 1, 0, 0]],
 [[0, 1, 0, 1], [1, 1, 0, 0]],
 [[0, 1, 1, 1], [1, 1, 1, 0]]]

sage: PPS = ParallelogramPolyominoes()
sage: PPS
Parallelogram polyominoes
sage: PPS.cardinality()
+Infinity

sage: PPS = ParallelogramPolyominoes(size=None)
sage: PPS
Parallelogram polyominoes
sage: PPS.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes(size=Integer(4))
>>> PPS
Parallelogram polyominoes of size 4
>>> sorted(PPS)
[[[0, 0, 0, 1], [1, 0, 0, 0]],
 [[0, 0, 1, 1], [1, 0, 1, 0]],
 [[0, 0, 1, 1], [1, 1, 0, 0]],
 [[0, 1, 0, 1], [1, 1, 0, 0]],
 [[0, 1, 1, 1], [1, 1, 1, 0]]]

>>> PPS = ParallelogramPolyominoes()
>>> PPS
Parallelogram polyominoes
>>> PPS.cardinality()
+Infinity

>>> PPS = ParallelogramPolyominoes(size=None)
>>> PPS
Parallelogram polyominoes
>>> PPS.cardinality()
+Infinity
```

class sage.combinat.parallelogram_polyomino.**ParallelogramPolyominoesFactory**

Bases: `SetFactory`

The parallelogram polyominoes factory.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes(size=4)
```

(continues on next page)

(continued from previous page)

```
sage: PPS
Parallelogram polyominoes of size 4

sage: sorted(PPS)
[[[0, 0, 0, 1], [1, 0, 0, 0]],
 [[0, 0, 1, 1], [1, 0, 1, 0]],
 [[0, 0, 1, 1], [1, 1, 0, 0]],
 [[0, 1, 0, 1], [1, 1, 0, 0]],
 [[0, 1, 1, 1], [1, 1, 1, 0]]]

sage: PPS = ParallelogramPolyominoes()
sage: PPS
Parallelogram polyominoes
sage: PPS.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes(size=Integer(4))
>>> PPS
Parallelogram polyominoes of size 4

>>> sorted(PPS)
[[[0, 0, 0, 1], [1, 0, 0, 0]],
 [[0, 0, 1, 1], [1, 0, 1, 0]],
 [[0, 0, 1, 1], [1, 1, 0, 0]],
 [[0, 1, 0, 1], [1, 1, 0, 0]],
 [[0, 1, 1, 1], [1, 1, 1, 0]]]

>>> PPS = ParallelogramPolyominoes()
>>> PPS
Parallelogram polyominoes
>>> PPS.cardinality()
+Infinity
```

sage.combinat.parallelogram_polyomino.**ParallelogramPolyominoesOptions = Current options for ParallelogramPolyominoes_size - display: 'list' - drawing_components: {'bounce_0': False, 'bounce_1': False, 'bounce_values': False, 'diagram': True, 'tree': False} - latex: 'drawing' - tikz_options: {'color_bounce_0': 'red', 'color_bounce_1': 'blue', 'color_line': 'black', 'color_point': 'black', 'line_size': 1, 'mirror': None, 'point_size': 3.5, 'rotation': 0, 'scale': 1, 'translation': [0, 0]}**

This global option contains all the data needed by the Parallelogram classes to draw, display in ASCII, compile in latex a parallelogram polyomino.

The available options are:

- tikz_options : this option configurate all the information useful to generate TIKZ code. For example, color, line size, etc …
- drawing_components : this option is used to explain to the system which component of the drawing you want to draw. For example, you can ask to draw some elements of the following list: - the diagram, - the tree inside the parallelogram polyomino, - the bounce paths inside the parallelogram polyomino, - the value of the bounce on each square of a bounce path.
- display : this option is used to configurate the ASCII display. The available options are: - list : (this is the

default value) is used to represent PP as a list containing the upper and lower path. - drawing : this value is used to explain we want to display an array with the PP drawn inside (with connected 1).

- latex : Same as display. The default is "drawing".

See `ParallelogramPolyomino.get_options()` for more details and for an user use of options.

EXAMPLES:

```
sage: from sage.combinat.parallelogram_polyomino import (
....:     ParallelogramPolyominoesOptions
....: )
sage: opt = ParallelogramPolyominoesOptions['tikz_options']
sage: opt
{'color_bounce_0': 'red',
 'color_bounce_1': 'blue',
 'color_line': 'black',
 'color_point': 'black',
 'line_size': 1,
 'mirror': None,
 'point_size': 3.5,
 'rotation': 0,
 'scale': 1,
 'translation': [0, 0]}
```

```
>>> from sage.all import *
>>> from sage.combinat.parallelogram_polyomino import (
...     ParallelogramPolyominoesOptions
... )
>>> opt = ParallelogramPolyominoesOptions['tikz_options']
>>> opt
{'color_bounce_0': 'red',
 'color_bounce_1': 'blue',
 'color_line': 'black',
 'color_point': 'black',
 'line_size': 1,
 'mirror': None,
 'point_size': 3.5,
 'rotation': 0,
 'scale': 1,
 'translation': [0, 0]}
```

class sage.combinat.parallelogram_polyomino.**ParallelogramPolyominoes_all**(*policy*)

Bases: `ParentWithSetFactory`, `DisjointUnionEnumeratedSets`

This class enumerates all the parallelogram polyominoes.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes()
sage: PPS
Parallelogram polyominoes
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes()
```

(continues on next page)

(continued from previous page)

```
>>> PPS
Parallelogram polyominoes
```

check_element (*el*, *check*)

Check is a given element *el* is in the set of parallelogram polyominoes.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes()
sage: ParallelogramPolyomino(                          # indirect doctest
....:     [[0, 1, 1], [1, 1, 0]]
....: ) in PPS
True
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes()
>>> ParallelogramPolyomino(                          # indirect doctest
...     [[Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1),␣
↪Integer(0)]]
... ) in PPS
True
```

get_options ()

Return all the options associated with the set of parallelogram polyominoes.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes()
sage: options = PPS.get_options()
sage: options
Current options for ParallelogramPolyominoes_size
  - display:            'list'
...
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes()
>>> options = PPS.get_options()
>>> options
Current options for ParallelogramPolyominoes_size
  - display:            'list'
...
```

options = Current options for ParallelogramPolyominoes_size - display: 'list' - drawing_components: {'bounce_0': False, 'bounce_1': False, 'bounce_values': False, 'diagram': True, 'tree': False} - latex: 'drawing' - tikz_options: {'color_bounce_0': 'red', 'color_bounce_1': 'blue', 'color_line': 'black', 'color_point': 'black', 'line_size': 1, 'mirror': None, 'point_size': 3.5, 'rotation': 0, 'scale': 1, 'translation': [0, 0]}

The options for ParallelogramPolyominoes.

set_options (**get_value*, ***set_value*)

Set new options to the object.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes()
sage: PPS.set_options(
....:     drawing_components=dict(
....:         diagram = True,
....:         bounce_0 = True,
....:         bounce_1 = True,
....:     )
....: )
sage: pp = next(iter(PPS))
sage: view(pp) # not tested
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes()
>>> PPS.set_options(
...     drawing_components=dict(
...         diagram = True,
...         bounce_0 = True,
...         bounce_1 = True,
...     )
... )
>>> pp = next(iter(PPS))
>>> view(pp) # not tested
```

class sage.combinat.parallelogram_polyomino.**ParallelogramPolyominoes_size**(*size*, *policy*)

Bases: ParentWithSetFactory, UniqueRepresentation

The parallelogram polyominoes of size n.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes(4)
sage: PPS
Parallelogram polyominoes of size 4
sage: sorted(PPS)
[[[0, 0, 0, 1], [1, 0, 0, 0]],
 [[0, 0, 1, 1], [1, 0, 1, 0]],
 [[0, 0, 1, 1], [1, 1, 0, 0]],
 [[0, 1, 0, 1], [1, 1, 0, 0]],
 [[0, 1, 1, 1], [1, 1, 1, 0]]]
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes(Integer(4))
>>> PPS
Parallelogram polyominoes of size 4
>>> sorted(PPS)
[[[0, 0, 0, 1], [1, 0, 0, 0]],
 [[0, 0, 1, 1], [1, 0, 1, 0]],
 [[0, 0, 1, 1], [1, 1, 0, 0]],
 [[0, 1, 0, 1], [1, 1, 0, 0]],
 [[0, 1, 1, 1], [1, 1, 1, 0]]]
```

an_element()

Return an element of a parallelogram polyomino of a given size.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes(4)
sage: PPS.an_element() in PPS
True
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes(Integer(4))
>>> PPS.an_element() in PPS
True
```

cardinality()

Return the number of parallelogram polyominoes.

The number of parallelogram polyominoes of size n is given by the Catalan number c_{n-1}.

EXAMPLES:

```
sage: ParallelogramPolyominoes(1).cardinality()
1
sage: ParallelogramPolyominoes(2).cardinality()
1
sage: ParallelogramPolyominoes(3).cardinality()
2
sage: ParallelogramPolyominoes(4).cardinality()
5

sage: all(
....:     ParallelogramPolyominoes(i).cardinality()
....:     == len(list(ParallelogramPolyominoes(i)))
....:     for i in range(1,7)
....: )
True
```

```
>>> from sage.all import *
>>> ParallelogramPolyominoes(Integer(1)).cardinality()
1
>>> ParallelogramPolyominoes(Integer(2)).cardinality()
1
>>> ParallelogramPolyominoes(Integer(3)).cardinality()
2
>>> ParallelogramPolyominoes(Integer(4)).cardinality()
5

>>> all(
...     ParallelogramPolyominoes(i).cardinality()
...     == len(list(ParallelogramPolyominoes(i)))
...     for i in range(Integer(1),Integer(7))
... )
True
```

check_element(*el*, *check*)

Check is a given element *el* is in the set of parallelogram polyominoes of a fixed size.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes(3)
sage: ParallelogramPolyomino(                        # indirect doctest
....:     [[0, 1, 1], [1, 1, 0]]
....: ) in PPS
True
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes(Integer(3))
>>> ParallelogramPolyomino(                        # indirect doctest
...     [[Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1),␣
↪Integer(0)]]
... ) in PPS
True
```

get_options()

Return all the options associated with all the elements of the set of parallelogram polyominoes with a fixed size.

EXAMPLES:

```
sage: pps = ParallelogramPolyominoes(5)
sage: pps.get_options()
Current options for ParallelogramPolyominoes_size
  - display:            'list'
...
```

```
>>> from sage.all import *
>>> pps = ParallelogramPolyominoes(Integer(5))
>>> pps.get_options()
Current options for ParallelogramPolyominoes_size
  - display:            'list'
...
```

options = Current options for ParallelogramPolyominoes_size - display: 'list' - drawing_components: {'bounce_0': False, 'bounce_1': False, 'bounce_values': False, 'diagram': True, 'tree': False} - latex: 'drawing' - tikz_options: {'color_bounce_0': 'red', 'color_bounce_1': 'blue', 'color_line': 'black', 'color_point': 'black', 'line_size': 1, 'mirror': None, 'point_size': 3.5, 'rotation': 0, 'scale': 1, 'translation': [0, 0]}

The options for ParallelogramPolyominoes.

set_options(**get_value*, ***set_value*)

Set new options to the object.

EXAMPLES:

```
sage: PPS = ParallelogramPolyominoes(3)
sage: PPS.set_options(
....:     drawing_components=dict(
....:         diagram = True,
....:         bounce_0 = True,
....:         bounce_1 = True,
....:     )
```

(continues on next page)

(continued from previous page)

```
....: )
sage: pp = PPS[0]
sage: view(pp) # not tested
```

```
>>> from sage.all import *
>>> PPS = ParallelogramPolyominoes(Integer(3))
>>> PPS.set_options(
...     drawing_components=dict(
...         diagram = True,
...         bounce_0 = True,
...         bounce_1 = True,
...     )
... )
>>> pp = PPS[Integer(0)]
>>> view(pp) # not tested
```

size()

Return the size of the parallelogram polyominoes generated by this parent.

EXAMPLES:

```
sage: ParallelogramPolyominoes(0).size()
0
sage: ParallelogramPolyominoes(1).size()
1
sage: ParallelogramPolyominoes(5).size()
5
```

```
>>> from sage.all import *
>>> ParallelogramPolyominoes(Integer(0)).size()
0
>>> ParallelogramPolyominoes(Integer(1)).size()
1
>>> ParallelogramPolyominoes(Integer(5)).size()
5
```

sage.combinat.parallelogram_polyomino.**default_tikz_options = {'color_bounce_0': 'red', 'color_bounce_1': 'blue', 'color_line': 'black', 'color_point': 'black', 'line_size': 1, 'mirror': None, 'point_size': 3.5, 'rotation': 0, 'scale': 1, 'translation': [0, 0]}**

This is the default TIKZ options.

This option is used to configurate element of a drawing to allow TIKZ code generation.

5.1.158 Parking Functions

INFORMALLY (reference [Beck]):

Imagine a one-way cul-de-sac with n parking spots. We will give the first parking spot the number 1, the next one number 2, etc., down to the last one, number n. Initially they are all free, but there are n cars approaching the street, and they would all like to park there. To make life interesting, every car has a parking preference, and we record the preferences in a sequence; For example, if $n = 3$, the sequence $(2, 1, 1)$ means that the first car would like to park at spot number 2, the second car prefers parking spot number 1, and the last car would also like to part at number 1. The street is very narrow, so there is no way to back up. Now each car enters the street and approaches its preferred parking spot; if it is free, it parks there, and if not, it moves down the street to the first available spot. We call a sequence a parking function (of length n) if all cars end up finding a parking spot. For example, the sequence $(2, 1, 1)$ is a parking sequence (of length 3), whereas the sequence $(2, 3, 2)$ is not.

FORMALLY:

A parking function of size n is a sequence $(a_1, \ldots, a_n)$ of positive integers such that if $b_1 \leq b_2 \leq \cdots \leq b_n$ is the increasing rearrangement of $a_1, \ldots, a_n$, then $b_i \leq i$.

A parking function of size n is a pair (L, D) of two sequences L and D where L is a permutation and D is an area sequence of a Dyck path of size n such that $D[i] \geq 0$, $D[i+1] \leq D[i]+1$ and if $D[i+1] = D[i]+1$ then $L[i+1] > L[i]$.

The number of parking functions of size n is equal to the number of rooted forests on n vertices and is equal to $(n+1)^{n-1}$.

REFERENCES:

AUTHORS:

- used non-decreasing_parking_functions code by Florent Hivert (2009 - 04)
- Dorota Mazur (2012 - 09)

`sage.combinat.parking_functions.`**`PF`**

alias of `ParkingFunction`

class `sage.combinat.parking_functions.`**`ParkingFunction`**(*parent*, *lst*)

Bases: `ClonableArray`

A Parking Function.

A *parking function* of size n is a sequence $(a_1, \ldots, a_n)$ of positive integers such that if $b_1 \leq b_2 \leq \cdots \leq b_n$ is the increasing rearrangement of $a_1, \ldots, a_n$, then $b_i \leq i$.

A *parking function* of size n is a pair (L, D) of two sequences L and D where L is a permutation and D is an area sequence of a Dyck Path of size n such that $D[i] \geq 0$, $D[i+1] \leq D[i]+1$ and if $D[i+1] = D[i]+1$ then $L[i+1] > L[i]$.

The number of parking functions of size n is equal to the number of rooted forests on n vertices and is equal to $(n+1)^{n-1}$.

INPUT:

- `pf` – (default: `None`) a list whose increasing rearrangement satisfies $b_i \leq i$
- `labelling` – (default: `None`) a labelling of the Dyck path
- `area_sequence` – (default: `None`) an area sequence of a Dyck path
- `labelled_dyck_word` – (default: `None`) a Dyck word with 1s replaced by labelling

OUTPUT: a parking function

EXAMPLES:

```
sage: ParkingFunction([])
[]
sage: ParkingFunction([1])
[1]
sage: ParkingFunction([2])
Traceback (most recent call last):
...
ValueError: [2] is not a parking function
```

(continues on next page)

(continued from previous page)

```
sage: ParkingFunction([1,2])
[1, 2]
sage: ParkingFunction([1,1,2])
[1, 1, 2]
sage: ParkingFunction([1,4,1])
Traceback (most recent call last):
...
ValueError: [1, 4, 1] is not a parking function
sage: ParkingFunction(labelling=[3,1,2], area_sequence=[0,0,1])
[2, 2, 1]
sage: ParkingFunction([2,2,1]).to_labelled_dyck_word()
[3, 0, 1, 2, 0, 0]
sage: ParkingFunction(labelled_dyck_word=[3,0,1,2,0,0])
[2, 2, 1]
sage: ParkingFunction(labelling=[3,1,2], area_sequence=[0,1,1])
Traceback (most recent call last):
...
ValueError: [3, 1, 2] is not a valid labeling of area sequence [0, 1, 1]
```

```
>>> from sage.all import *
>>> ParkingFunction([])
[]
>>> ParkingFunction([Integer(1)])
[1]
>>> ParkingFunction([Integer(2)])
Traceback (most recent call last):
...
ValueError: [2] is not a parking function
>>> ParkingFunction([Integer(1),Integer(2)])
[1, 2]
>>> ParkingFunction([Integer(1),Integer(1),Integer(2)])
[1, 1, 2]
>>> ParkingFunction([Integer(1),Integer(4),Integer(1)])
Traceback (most recent call last):
...
ValueError: [1, 4, 1] is not a parking function
>>> ParkingFunction(labelling=[Integer(3),Integer(1),Integer(2)], area_
↪sequence=[Integer(0),Integer(0),Integer(1)])
[2, 2, 1]
>>> ParkingFunction([Integer(2),Integer(2),Integer(1)]).to_labelled_dyck_word()
[3, 0, 1, 2, 0, 0]
>>> ParkingFunction(labelled_dyck_word=[Integer(3),Integer(0),Integer(1),
↪Integer(2),Integer(0),Integer(0)])
[2, 2, 1]
>>> ParkingFunction(labelling=[Integer(3),Integer(1),Integer(2)], area_
↪sequence=[Integer(0),Integer(1),Integer(1)])
Traceback (most recent call last):
...
ValueError: [3, 1, 2] is not a valid labeling of area sequence [0, 1, 1]
```

area()

Return the area of the labelled Dyck path corresponding to the parking function.

OUTPUT:

- the sum of squares under and over the main diagonal the Dyck Path, corresponding to the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.area()
6
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.area()
6
```

```
sage: ParkingFunction([3,1,1,4]).area()
1
sage: ParkingFunction([4,1,1,1]).area()
3
sage: ParkingFunction([2,1,4,1]).area()
2
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).area()
1
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).area()
3
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).area()
2
```

cars_permutation()

Return the sequence of cars that take parking spots 1 through n and corresponding to the parking function.

For example, `cars_permutation(PF) = [2, 4, 5, 6, 3, 1, 7]` means that car 2 takes spots 1, car 4 takes spot 2, …, car 1 takes spot 6 and car 7 takes spot 7.

OUTPUT:

- the permutation of cars corresponding to the parking function and which is the same size as parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.cars_permutation()
[2, 4, 5, 6, 3, 1, 7]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.cars_permutation()
[2, 4, 5, 6, 3, 1, 7]
```

```
sage: ParkingFunction([3,1,1,4]).cars_permutation()
[2, 3, 1, 4]
sage: ParkingFunction([4,1,1,1]).cars_permutation()
[2, 3, 4, 1]
sage: ParkingFunction([2,1,4,1]).cars_permutation()
[2, 1, 4, 3]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).cars_
↪permutation()
[2, 3, 1, 4]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).cars_
↪permutation()
[2, 3, 4, 1]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).cars_
↪permutation()
[2, 1, 4, 3]
```

characteristic_quasisymmetric_function(*q=None*, *R=Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field*)

Return the characteristic quasisymmetric function of `self`.

The characteristic function of the Parking Function is the sum over all permutation labellings of the Dyck path $q^{dinv(PF)} F_{ides(PF)}$ where $ides(PF)$ (`ides_composition()`) is the descent composition of diagonal reading word of the parking function.

INPUT:

- `q` – (default: `q = R('q')`) a parameter for the generating function power
- `R` – (default: `R = QQ['q','t'].fraction_field()`) the base ring to do the calculations over

OUTPUT: an element of the quasisymmetric functions over the ring `R`

EXAMPLES:

```
sage: # needs sage.modules
sage: R = QQ['q','t'].fraction_field()
sage: (q,t) = R.gens()
sage: cqf = sum(t**PF.area() * PF.characteristic_quasisymmetric_function()
....:           for PF in ParkingFunctions(3)); cqf
(q^3+q^2*t+q*t^2+t^3+q*t)*F[1, 1, 1] + (q^2+q*t+t^2+q+t)*F[1, 2]
 + (q^2+q*t+t^2+q+t)*F[2, 1] + F[3]
sage: s = SymmetricFunctions(R).s()
sage: s(cqf.to_symmetric_function())
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
sage: s(cqf.to_symmetric_function()).nabla(power=-1)
s[1, 1, 1]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> R = QQ['q','t'].fraction_field()
>>> (q,t) = R.gens()
>>> cqf = sum(t**PF.area() * PF.characteristic_quasisymmetric_function()
...           for PF in ParkingFunctions(Integer(3))); cqf
```

(continues on next page)

(continued from previous page)

```
(q^3+q^2*t+q*t^2+t^3+q*t)*F[1, 1, 1] + (q^2+q*t+t^2+q+t)*F[1, 2]
 + (q^2+q*t+t^2+q+t)*F[2, 1] + F[3]
>>> s = SymmetricFunctions(R).s()
>>> s(cqf.to_symmetric_function())
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
>>> s(cqf.to_symmetric_function()).nabla(power=-Integer(1))
s[1, 1, 1]
```

```
sage: p = ParkingFunction([3, 1, 2])
sage: p.characteristic_quasisymmetric_function()                          #␣
↪needs sage.modules
q*F[2, 1]
sage: pf = ParkingFunction([1,2,7,2,1,2,3,2,1])
sage: pf.characteristic_quasisymmetric_function()                         #␣
↪needs sage.modules
q^2*F[1, 1, 1, 2, 1, 3]
```

```
>>> from sage.all import *
>>> p = ParkingFunction([Integer(3), Integer(1), Integer(2)])
>>> p.characteristic_quasisymmetric_function()                          #␣
↪needs sage.modules
q*F[2, 1]
>>> pf = ParkingFunction([Integer(1),Integer(2),Integer(7),Integer(2),
↪Integer(1),Integer(2),Integer(3),Integer(2),Integer(1)])
>>> pf.characteristic_quasisymmetric_function()                         #␣
↪needs sage.modules
q^2*F[1, 1, 1, 2, 1, 3]
```

check()

Check that `self` is a valid parking function.

EXAMPLES:

```
sage: PF = ParkingFunction([1, 1, 2, 2, 5, 6])
sage: PF.check()
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(5), Integer(6)])
>>> PF.check()
```

diagonal_composition()

Return the composition of the labelled Dyck path corresponding to the parking function.

For example, `touch_composition(PF) = [4, 3]` means that the first touch is four diagonal units from the starting point, and the second is three units further (see [GXZ] p. 2).

OUTPUT:

- the length between the corresponding touch points which of the labelled Dyck path that corresponds to the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.touch_composition()
[4, 3]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.touch_composition()
[4, 3]
```

```
sage: ParkingFunction([3,1,1,4]).touch_composition()
[2, 1, 1]
sage: ParkingFunction([4,1,1,1]).touch_composition()
[3, 1]
sage: ParkingFunction([2,1,4,1]).touch_composition()
[3, 1]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).touch_
↪composition()
[2, 1, 1]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).touch_
↪composition()
[3, 1]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).touch_
↪composition()
[3, 1]
```

diagonal_reading_word()

Return a diagonal word of the labelled Dyck path corresponding to parking function (see [Hag08] p. 75).

OUTPUT:

- returns a word, read diagonally from NE to SW of the pretty print of the labelled Dyck path that corresponds to `self` and the same size as `self`

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.diagonal_reading_word()
[5, 1, 7, 4, 6, 3, 2]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.diagonal_reading_word()
[5, 1, 7, 4, 6, 3, 2]
```

```
sage: ParkingFunction([1, 1, 1]).diagonal_reading_word()
[3, 2, 1]
sage: ParkingFunction([1, 2, 3]).diagonal_reading_word()
[3, 2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: ParkingFunction([1, 1, 3, 4]).diagonal_reading_word()
[2, 4, 3, 1]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(1), Integer(1), Integer(1)]).diagonal_reading_
↪word()
[3, 2, 1]
>>> ParkingFunction([Integer(1), Integer(2), Integer(3)]).diagonal_reading_
↪word()
[3, 2, 1]
>>> ParkingFunction([Integer(1), Integer(1), Integer(3), Integer(4)]).
↪diagonal_reading_word()
[2, 4, 3, 1]
```

```
sage: ParkingFunction([1, 1, 1]).diagonal_word()
[3, 2, 1]
sage: ParkingFunction([1, 2, 3]).diagonal_word()
[3, 2, 1]
sage: ParkingFunction([1, 4, 3, 1]).diagonal_word()
[4, 2, 3, 1]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(1), Integer(1), Integer(1)]).diagonal_word()
[3, 2, 1]
>>> ParkingFunction([Integer(1), Integer(2), Integer(3)]).diagonal_word()
[3, 2, 1]
>>> ParkingFunction([Integer(1), Integer(4), Integer(3), Integer(1)]).
↪diagonal_word()
[4, 2, 3, 1]
```

diagonal_word()

Return a diagonal word of the labelled Dyck path corresponding to parking function (see [Hag08] p. 75).

OUTPUT:

- returns a word, read diagonally from NE to SW of the pretty print of the labelled Dyck path that corresponds to `self` and the same size as `self`

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.diagonal_reading_word()
[5, 1, 7, 4, 6, 3, 2]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.diagonal_reading_word()
[5, 1, 7, 4, 6, 3, 2]
```

```
sage: ParkingFunction([1, 1, 1]).diagonal_reading_word()
[3, 2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: ParkingFunction([1, 2, 3]).diagonal_reading_word()
[3, 2, 1]
sage: ParkingFunction([1, 1, 3, 4]).diagonal_reading_word()
[2, 4, 3, 1]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(1), Integer(1), Integer(1)]).diagonal_reading_
↪word()
[3, 2, 1]
>>> ParkingFunction([Integer(1), Integer(2), Integer(3)]).diagonal_reading_
↪word()
[3, 2, 1]
>>> ParkingFunction([Integer(1), Integer(1), Integer(3), Integer(4)]).
↪diagonal_reading_word()
[2, 4, 3, 1]
```

```
sage: ParkingFunction([1, 1, 1]).diagonal_word()
[3, 2, 1]
sage: ParkingFunction([1, 2, 3]).diagonal_word()
[3, 2, 1]
sage: ParkingFunction([1, 4, 3, 1]).diagonal_word()
[4, 2, 3, 1]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(1), Integer(1), Integer(1)]).diagonal_word()
[3, 2, 1]
>>> ParkingFunction([Integer(1), Integer(2), Integer(3)]).diagonal_word()
[3, 2, 1]
>>> ParkingFunction([Integer(1), Integer(4), Integer(3), Integer(1)]).
↪diagonal_word()
[4, 2, 3, 1]
```

dinv()

Return the number of inversions of a labelled Dyck path corresponding to the parking function (see [Hag08] p. 74).

Same as the cardinality of *dinversion_pairs()*.

OUTPUT: the number of dinversion pairs

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.dinv()
6
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.dinv()
6
```

```
sage: ParkingFunction([3,1,1,4]).dinv()
3
sage: ParkingFunction([4,1,1,1]).dinv()
1
sage: ParkingFunction([2,1,4,1]).dinv()
2
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).dinv()
3
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).dinv()
1
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).dinv()
2
```

dinversion_pairs()

Return the descent inversion pairs of a labelled Dyck path corresponding to the parking function.

OUTPUT: the primary and secondary diversion pairs

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.dinversion_pairs()
[(0, 4), (1, 5), (2, 5), (1, 4), (2, 4), (3, 6)]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.dinversion_pairs()
[(0, 4), (1, 5), (2, 5), (1, 4), (2, 4), (3, 6)]
```

```
sage: ParkingFunction([3,1,1,4]).dinversion_pairs()
[(0, 3), (2, 3), (1, 2)]
sage: ParkingFunction([4,1,1,1]).dinversion_pairs()
[(1, 3)]
sage: ParkingFunction([2,1,4,1]).dinversion_pairs()
[(0, 3), (1, 3)]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).dinversion_
↪pairs()
[(0, 3), (2, 3), (1, 2)]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).dinversion_
↪pairs()
[(1, 3)]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).dinversion_
↪pairs()
[(0, 3), (1, 3)]
```

grade()

Return the length of the parking function.

EXAMPLES:

```
sage: PF = ParkingFunction([1, 1, 2, 2, 5, 6])
sage: PF.grade()
6
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(5), Integer(6)])
>>> PF.grade()
6
```

ides()

Return the *descents()* sequence of the inverse of the *diagonal_reading_word()* of self.

> **Warning**
>
> Here we use the standard convention that descent labels start at 1. This behaviour has been changed in Issue #20555.

For example, ides(PF) = [2, 3, 4, 6] means that descents are at the 2nd, 3rd, 4th and 6th positions in the inverse of the *diagonal_reading_word()* of the parking function (see [GXZ] p. 2).

OUTPUT:

- the descents sequence of the inverse of the *diagonal_reading_word()* of the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.ides()
[2, 3, 4, 6]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.ides()
[2, 3, 4, 6]
```

```
sage: ParkingFunction([3,1,1,4]).ides()
[2]
sage: ParkingFunction([4,1,1,1]).ides()
[2, 3]
sage: ParkingFunction([4,3,1,1]).ides()
[3]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).ides()
[2]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).ides()
[2, 3]
>>> ParkingFunction([Integer(4),Integer(3),Integer(1),Integer(1)]).ides()
[3]
```

ides_composition()

Return the *descents_composition()* of the inverse of the *diagonal_reading_word()* of corresponding parking function.

For example, `ides_composition(PF) = [4, 2, 1]` means that the descents of the inverse of the permutation *diagonal_reading_word()* of the parking function with word `PF` are at the 4th and 6th positions.

OUTPUT:

- the descents composition of the inverse of the *diagonal_reading_word()* of the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.ides_composition()
[2, 1, 1, 2, 1]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.ides_composition()
[2, 1, 1, 2, 1]
```

```
sage: ParkingFunction([3,1,1,4]).ides_composition()
[2, 2]
sage: ParkingFunction([4,1,1,1]).ides_composition()
[2, 1, 1]
sage: ParkingFunction([4,3,1,1]).ides_composition()
[3, 1]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).ides_
↪composition()
[2, 2]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).ides_
↪composition()
[2, 1, 1]
>>> ParkingFunction([Integer(4),Integer(3),Integer(1),Integer(1)]).ides_
↪composition()
[3, 1]
```

jump()

Return the sum of the differences between the parked and preferred parking spots.

See [Shin] p. 18.

OUTPUT:

- the sum of the differences between the parked and preferred parking spots

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.jump()
6
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.jump()
6
```

```
sage: ParkingFunction([3,1,1,4]).jump()
1
sage: ParkingFunction([4,1,1,1]).jump()
3
sage: ParkingFunction([2,1,4,1]).jump()
2
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).jump()
1
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).jump()
3
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).jump()
2
```

jump_list()

Return the displacements of cars that corresponds to the parking function.

For example, `jump_list(PF) = [0, 0, 0, 0, 1, 3, 2]` means that car 1 through 4 parked in their preferred spots, car 5 had to park one spot farther (jumped or was displaced by one spot), car 6 had to jump 3 spots, and car 7 had to jump two spots.

OUTPUT:

- the displacements sequence of parked cars which corresponds to the parking function and which is the same size as parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.jump_list()
[0, 0, 0, 0, 1, 3, 2]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.jump_list()
[0, 0, 0, 0, 1, 3, 2]
```

```
sage: ParkingFunction([3,1,1,4]).jump_list()
[0, 0, 1, 0]
sage: ParkingFunction([4,1,1,1]).jump_list()
[0, 0, 1, 2]
sage: ParkingFunction([2,1,4,1]).jump_list()
[0, 0, 0, 2]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).jump_list()
[0, 0, 1, 0]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).jump_list()
[0, 0, 1, 2]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).jump_list()
[0, 0, 0, 2]
```

luck()

Return the number of cars that parked in their preferred parking spots (see [Shin] p. 33).

OUTPUT: the number of cars that parked in their preferred parking spots

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.luck()
4
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.luck()
4
```

```
sage: ParkingFunction([3,1,1,4]).luck()
3
sage: ParkingFunction([4,1,1,1]).luck()
2
sage: ParkingFunction([2,1,4,1]).luck()
3
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).luck()
3
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).luck()
2
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).luck()
3
```

lucky_cars()

Return the cars that can park in their preferred spots. For example, `lucky_cars(PF) = [1, 2, 7]` means that cars 1, 2 and 7 parked in their preferred spots and all the other cars did not.

OUTPUT: the cars that can park in their preferred spots

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.lucky_cars()
[1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.lucky_cars()
[1, 2, 3, 4]
```

```
sage: ParkingFunction([3,1,1,4]).lucky_cars()
[1, 2, 4]
sage: ParkingFunction([4,1,1,1]).lucky_cars()
[1, 2]
sage: ParkingFunction([2,1,4,1]).lucky_cars()
[1, 2, 3]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).lucky_
↪cars()
[1, 2, 4]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).lucky_
↪cars()
[1, 2]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).lucky_
↪cars()
[1, 2, 3]
```

parking_permutation()

Return the sequence of parking spots that are taken by cars 1 through n and corresponding to the parking function.

For example, `parking_permutation(PF) = [6, 1, 5, 2, 3, 4, 7]` means that spot 6 is taken by car 1, spot 1 by car 2, spot 5 by car 3, spot 2 is taken by car 4, spot 3 is taken by car 5, spot 4 is taken by car 6 and spot 7 is taken by car 7.

OUTPUT:

- the permutation of parking spots that corresponds to the parking function and which is the same size as parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.parking_permutation()
[6, 1, 5, 2, 3, 4, 7]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.parking_permutation()
[6, 1, 5, 2, 3, 4, 7]
```

```
sage: ParkingFunction([3,1,1,4]).parking_permutation()
[3, 1, 2, 4]
sage: ParkingFunction([4,1,1,1]).parking_permutation()
[4, 1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
sage: ParkingFunction([2,1,4,1]).parking_permutation()
[2, 1, 4, 3]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).parking_
↪permutation()
[3, 1, 2, 4]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).parking_
↪permutation()
[4, 1, 2, 3]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).parking_
↪permutation()
[2, 1, 4, 3]
```

pretty_print(*underpath=True*)

Displays a parking function as a lattice path consisting of a Dyck path and a labelling with the labels displayed along the edges of the Dyck path.

INPUT:

- `underpath` – if the length of the parking function is less than or equal to 9 then display the labels under the path if `underpath` is `True` otherwise display them to the right of the path (default: `True`)

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.pretty_print()
           ___
         _|1x
        |7x   .
  ______|3 . .
 |5x x   . . .
_|4x   . . . .
|6x   . . . . .
|2 . . . . . .

sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.pretty_print(underpath = false)
           ___
         _| x   1
        | x   . 7
  ______|   . . 3
 | x x   . . . 5
_| x   . . . . 4
| x   . . . . . 6
|   . . . . . . 2
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.pretty_print()
           ___
         _|1x
```

(continues on next page)

(continued from previous page)

```
        |7x  .
   _____|3 . .
  |5x x  . . .
 _|4x  . . . .
|6x  . . . . .
|2 . . . . . .

>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),
↪Integer(2), Integer(1), Integer(5)])
>>> PF.pretty_print(underpath = false)
           ___
         _| x  1
        | x  . 7
   _____|  . . 3
  | x x  . . . 5
 _| x  . . . . 4
| x  . . . . . 6
|  . . . . . . 2
```

```
sage: ParkingFunction([3, 1, 1, 4]).pretty_print()
       _
     _|4
 ___|1 .
|3x  . .
|2 . . .

sage: ParkingFunction([1,1,1]).pretty_print()
 _____
|3x x
|2x  .
|1 . .

sage: ParkingFunction([4,1,1,1]).pretty_print()
       _
 _____|1
|4x x  .
|3x  . .
|2 . . .

sage: ParkingFunction([2,1,4,1]).pretty_print()
       _
   ___|3
 _|1x  .
|4x  . .
|2 . . .

sage: ParkingFunction([2,1,4,1]).pretty_print(underpath = false)
       _
   ___|  3
 _| x  . 1
| x  . . 4
|  . . . 2
```

(continues on next page)

(continued from previous page)

```
sage: pf = ParkingFunction([1,2,3,7,3,2,1,2,3,2,1])
sage: pf.pretty_print()
             _________
     _______| x x x x   4
    | x x x x x x x  .  9
    | x x x x x x  . .  5
   _| x x x x x  . . .  3
  | x x x x x  . . . . 10
  | x x x x  . . . . .  8
  | x x x  . . . . . .  6
 _| x x  . . . . . . .  2
| x x  . . . . . . . . 11
| x  . . . . . . . . .  7
|  . . . . . . . . . .  1
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3), Integer(1), Integer(1), Integer(4)]).pretty_
↪print()
       _
     _|4
 ___|1 .
|3x  . .
|2 . . .

>>> ParkingFunction([Integer(1),Integer(1),Integer(1)]).pretty_print()
 _____
|3x x
|2x  .
|1 . .

>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).pretty_
↪print()
       _
 _____|1
|4x x  .
|3x  . .
|2 . . .

>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).pretty_
↪print()
       _
   ___|3
 _|1x  .
|4x  . .
|2 . . .

>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).pretty_
↪print(underpath = false)
       _
   ___|  3
 _| x  . 1
```

(continues on next page)

(continued from previous page)

```
| x  . . 4
|  . . . 2

>>> pf = ParkingFunction([Integer(1),Integer(2),Integer(3),Integer(7),
↪Integer(3),Integer(2),Integer(1),Integer(2),Integer(3),Integer(2),
↪Integer(1)])
>>> pf.pretty_print()
             _________
    ________| x x x x   4
    | x x x x x x x  .  9
    | x x x x x x  . .  5
   _| x x x x x  . . .  3
  | x x x x x  . . . . 10
  | x x x x  . . . . .  8
  | x x x  . . . . . .  6
 _| x x  . . . . . . .  2
| x x  . . . . . . . . 11
| x  . . . . . . . . .  7
|  . . . . . . . . . .  1
```

primary_dinversion_pairs()

Return the primary descent inversion pairs of a labelled Dyck path corresponding to the parking function.

OUTPUT:

The pairs (i, j) such that $i < j$, and i-th area = j-th area, and i-th label < j-th label

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.primary_dinversion_pairs()
[(0, 4), (1, 5), (2, 5)]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.primary_dinversion_pairs()
[(0, 4), (1, 5), (2, 5)]
```

```
sage: ParkingFunction([3,1,1,4]).primary_dinversion_pairs()
[(0, 3), (2, 3)]
sage: ParkingFunction([4,1,1,1]).primary_dinversion_pairs()
[]
sage: ParkingFunction([2,1,4,1]).primary_dinversion_pairs()
[(0, 3)]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).primary_
↪dinversion_pairs()
[(0, 3), (2, 3)]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).primary_
↪dinversion_pairs()
[]
```

(continues on next page)

(continued from previous page)

```
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).primary_
↪dinversion_pairs()
[(0, 3)]
```

secondary_dinversion_pairs()

Return the secondary descent inversion pairs of a labelled Dyck path corresponding to the parking function.

OUTPUT:

The pairs (i, j) such that $i < j$, and i-th area = j-th area +1, and i-th label > j-th label

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.secondary_dinversion_pairs()
[(1, 4), (2, 4), (3, 6)]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.secondary_dinversion_pairs()
[(1, 4), (2, 4), (3, 6)]
```

```
sage: ParkingFunction([3,1,1,4]).secondary_dinversion_pairs()
[(1, 2)]
sage: ParkingFunction([4,1,1,1]).secondary_dinversion_pairs()
[(1, 3)]
sage: ParkingFunction([2,1,4,1]).secondary_dinversion_pairs()
[(1, 3)]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).secondary_
↪dinversion_pairs()
[(1, 2)]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).secondary_
↪dinversion_pairs()
[(1, 3)]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).secondary_
↪dinversion_pairs()
[(1, 3)]
```

to_NonDecreasingParkingFunction()

Return the non-decreasing parking function which underlies the parking function.

OUTPUT: a sorted parking function

See also

NonDecreasingParkingFunction()

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_NonDecreasingParkingFunction()
[1, 1, 2, 2, 5, 5, 6]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.to_NonDecreasingParkingFunction()
[1, 1, 2, 2, 5, 5, 6]
```

```
sage: ParkingFunction([3,1,1,4]).to_NonDecreasingParkingFunction()
[1, 1, 3, 4]
sage: ParkingFunction([4,1,1,1]).to_NonDecreasingParkingFunction()
[1, 1, 1, 4]
sage: ParkingFunction([2,1,4,1]).to_NonDecreasingParkingFunction()
[1, 1, 2, 4]
sage: ParkingFunction([4,1,2,1]).to_NonDecreasingParkingFunction()
[1, 1, 2, 4]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).to_
↪NonDecreasingParkingFunction()
[1, 1, 3, 4]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).to_
↪NonDecreasingParkingFunction()
[1, 1, 1, 4]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).to_
↪NonDecreasingParkingFunction()
[1, 1, 2, 4]
>>> ParkingFunction([Integer(4),Integer(1),Integer(2),Integer(1)]).to_
↪NonDecreasingParkingFunction()
[1, 1, 2, 4]
```

to_area_sequence()

Return the area sequence of the support Dyck path of the parking function.

OUTPUT: the area sequence of the Dyck path

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_area_sequence()
[0, 1, 1, 2, 0, 1, 1]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.to_area_sequence()
[0, 1, 1, 2, 0, 1, 1]
```

```
sage: ParkingFunction([3,1,1,4]).to_area_sequence()
[0, 1, 0, 0]
```

(continues on next page)

(continued from previous page)

```
sage: ParkingFunction([4,1,1,1]).to_area_sequence()
[0, 1, 2, 0]
sage: ParkingFunction([2,1,4,1]).to_area_sequence()
[0, 1, 1, 0]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).to_area_
↪sequence()
[0, 1, 0, 0]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).to_area_
↪sequence()
[0, 1, 2, 0]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).to_area_
↪sequence()
[0, 1, 1, 0]
```

to_dyck_word()

Return the support Dyck word of the parking function.

OUTPUT: the Dyck word of the corresponding parking function

See also

DyckWord()

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_dyck_word()
[1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.to_dyck_word()
[1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0]
```

```
sage: ParkingFunction([3,1,1,4]).to_dyck_word()
[1, 1, 0, 0, 1, 0, 1, 0]
sage: ParkingFunction([4,1,1,1]).to_dyck_word()
[1, 1, 1, 0, 0, 0, 1, 0]
sage: ParkingFunction([2,1,4,1]).to_dyck_word()
[1, 1, 0, 1, 0, 0, 1, 0]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).to_dyck_
↪word()
[1, 1, 0, 0, 1, 0, 1, 0]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).to_dyck_
↪word()
```

(continues on next page)

(continued from previous page)

```
[1, 1, 1, 0, 0, 0, 1, 0]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).to_dyck_
↪word()
[1, 1, 0, 1, 0, 0, 1, 0]
```

to_labelled_dyck_word()

Return the labelled Dyck word corresponding to the parking function.

This is a representation of the parking function as a list where the entries of 1 in the Dyck word are replaced with the corresponding label.

OUTPUT:

- the labelled Dyck word of the corresponding parking function which is twice the size of parking function word

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_labelled_dyck_word()
[2, 6, 0, 4, 5, 0, 0, 0, 3, 7, 0, 1, 0, 0]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.to_labelled_dyck_word()
[2, 6, 0, 4, 5, 0, 0, 0, 3, 7, 0, 1, 0, 0]
```

```
sage: ParkingFunction([3,1,1,4]).to_labelled_dyck_word()
[2, 3, 0, 0, 1, 0, 4, 0]
sage: ParkingFunction([4,1,1,1]).to_labelled_dyck_word()
[2, 3, 4, 0, 0, 0, 1, 0]
sage: ParkingFunction([2,1,4,1]).to_labelled_dyck_word()
[2, 4, 0, 1, 0, 0, 3, 0]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).to_
↪labelled_dyck_word()
[2, 3, 0, 0, 1, 0, 4, 0]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).to_
↪labelled_dyck_word()
[2, 3, 4, 0, 0, 0, 1, 0]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).to_
↪labelled_dyck_word()
[2, 4, 0, 1, 0, 0, 3, 0]
```

to_labelling_area_sequence_pair()

Return a pair consisting of a labelling and an area sequence of a Dyck path which corresponds to the given parking function.

OUTPUT:

- returns a pair `(L, D)` where `L` is a labelling and `D` is the area sequence of the underlying Dyck path

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_labelling_area_sequence_pair()
([2, 6, 4, 5, 3, 7, 1], [0, 1, 1, 2, 0, 1, 1])
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.to_labelling_area_sequence_pair()
([2, 6, 4, 5, 3, 7, 1], [0, 1, 1, 2, 0, 1, 1])
```

```
sage: ParkingFunction([1, 1, 1]).to_labelling_area_sequence_pair()
([1, 2, 3], [0, 1, 2])
sage: ParkingFunction([1, 2, 3]).to_labelling_area_sequence_pair()
([1, 2, 3], [0, 0, 0])
sage: ParkingFunction([1, 1, 2]).to_labelling_area_sequence_pair()
([1, 2, 3], [0, 1, 1])
sage: ParkingFunction([1, 1, 3, 1]).to_labelling_area_sequence_pair()
([1, 2, 4, 3], [0, 1, 2, 1])
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(1), Integer(1), Integer(1)]).to_labelling_area_
↪sequence_pair()
([1, 2, 3], [0, 1, 2])
>>> ParkingFunction([Integer(1), Integer(2), Integer(3)]).to_labelling_area_
↪sequence_pair()
([1, 2, 3], [0, 0, 0])
>>> ParkingFunction([Integer(1), Integer(1), Integer(2)]).to_labelling_area_
↪sequence_pair()
([1, 2, 3], [0, 1, 1])
>>> ParkingFunction([Integer(1), Integer(1), Integer(3), Integer(1)]).to_
↪labelling_area_sequence_pair()
([1, 2, 4, 3], [0, 1, 2, 1])
```

to_labelling_dyck_word_pair()

Return the pair `(L, D)` where `L` is a labelling and `D` is the Dyck word of the parking function.

OUTPUT:

- the pair `(L, D)`, where `L` is the labelling and `D` is the Dyck word of the parking function

See also

DyckWord()

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_labelling_dyck_word_pair()
([2, 6, 4, 5, 3, 7, 1], [1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0])
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(2), Integer(1), Integer(5)])
>>> PF.to_labelling_dyck_word_pair()
([2, 6, 4, 5, 3, 7, 1], [1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0])
```

```
sage: ParkingFunction([3,1,1,4]).to_labelling_dyck_word_pair()
([2, 3, 1, 4], [1, 1, 0, 0, 1, 0, 1, 0])
sage: ParkingFunction([4,1,1,1]).to_labelling_dyck_word_pair()
([2, 3, 4, 1], [1, 1, 1, 0, 0, 0, 1, 0])
sage: ParkingFunction([2,1,4,1]).to_labelling_dyck_word_pair()
([2, 4, 1, 3], [1, 1, 0, 1, 0, 0, 1, 0])
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).to_
↪labelling_dyck_word_pair()
([2, 3, 1, 4], [1, 1, 0, 0, 1, 0, 1, 0])
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).to_
↪labelling_dyck_word_pair()
([2, 3, 4, 1], [1, 1, 1, 0, 0, 0, 1, 0])
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).to_
↪labelling_dyck_word_pair()
([2, 4, 1, 3], [1, 1, 0, 1, 0, 0, 1, 0])
```

to_labelling_permutation()

Return the labelling of the support Dyck path of the parking function.

OUTPUT: the labelling of the Dyck path

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_labelling_permutation()
[2, 6, 4, 5, 3, 7, 1]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.to_labelling_permutation()
[2, 6, 4, 5, 3, 7, 1]
```

```
sage: ParkingFunction([3,1,1,4]).to_labelling_permutation()
[2, 3, 1, 4]
sage: ParkingFunction([4,1,1,1]).to_labelling_permutation()
[2, 3, 4, 1]
sage: ParkingFunction([2,1,4,1]).to_labelling_permutation()
[2, 4, 1, 3]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).to_
↪labelling_permutation()
[2, 3, 1, 4]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).to_
↪labelling_permutation()
```

(continues on next page)

(continued from previous page)

```
[2, 3, 4, 1]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).to_
↪labelling_permutation()
[2, 4, 1, 3]
```

touch_composition()

Return the composition of the labelled Dyck path corresponding to the parking function.

For example, `touch_composition(PF) = [4, 3]` means that the first touch is four diagonal units from the starting point, and the second is three units further (see [GXZ] p. 2).

OUTPUT:

- the length between the corresponding touch points which of the labelled Dyck path that corresponds to the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.touch_composition()
[4, 3]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.touch_composition()
[4, 3]
```

```
sage: ParkingFunction([3,1,1,4]).touch_composition()
[2, 1, 1]
sage: ParkingFunction([4,1,1,1]).touch_composition()
[3, 1]
sage: ParkingFunction([2,1,4,1]).touch_composition()
[3, 1]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).touch_
↪composition()
[2, 1, 1]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).touch_
↪composition()
[3, 1]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).touch_
↪composition()
[3, 1]
```

touch_points()

Return the sequence of touch points which corresponds to the labelled Dyck path after initial step.

For example, `touch_points(PF) = [4, 7]` means that after the initial step, the path touches the main diagonal at points $(4, 4)$ and $(7, 7)$.

OUTPUT:

- the sequence of touch points after the initial step of the labelled Dyck path that corresponds to the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.touch_points()
[4, 7]
```

```
>>> from sage.all import *
>>> PF = ParkingFunction([Integer(6), Integer(1), Integer(5), Integer(2),␣
↪Integer(2), Integer(1), Integer(5)])
>>> PF.touch_points()
[4, 7]
```

```
sage: ParkingFunction([3,1,1,4]).touch_points()
[2, 3, 4]
sage: ParkingFunction([4,1,1,1]).touch_points()
[3, 4]
sage: ParkingFunction([2,1,4,1]).touch_points()
[3, 4]
```

```
>>> from sage.all import *
>>> ParkingFunction([Integer(3),Integer(1),Integer(1),Integer(4)]).touch_
↪points()
[2, 3, 4]
>>> ParkingFunction([Integer(4),Integer(1),Integer(1),Integer(1)]).touch_
↪points()
[3, 4]
>>> ParkingFunction([Integer(2),Integer(1),Integer(4),Integer(1)]).touch_
↪points()
[3, 4]
```

class sage.combinat.parking_functions.**ParkingFunctions**

Bases: UniqueRepresentation, Parent

Return the combinatorial class of Parking Functions.

A *parking function* of size n is a sequence $(a_1, \ldots, a_n)$ of positive integers such that if $b_1 \leq b_2 \leq \cdots \leq b_n$ is the increasing rearrangement of $a_1, \ldots, a_n$, then $b_i \leq i$.

A *parking function* of size n is a pair (L, D) of two sequences L and D where L is a permutation and D is an area sequence of a Dyck Path of size n such that $D[i] \geq 0$, $D[i+1] \leq D[i] + 1$ and if $D[i+1] = D[i] + 1$ then $L[i+1] > L[i]$.

The number of parking functions of size n is equal to the number of rooted forests on n vertices and is equal to $(n+1)^{n-1}$.

EXAMPLES:

Here are all parking functions of size 3:

```
sage: from sage.combinat.parking_functions import ParkingFunctions
sage: ParkingFunctions(3).list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3], [1, 3, 1],
```

(continues on next page)

(continued from previous page)

```
 [3, 1, 1], [1, 2, 2], [2, 1, 2], [2, 2, 1], [1, 2, 3], [1, 3, 2],
 [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.parking_functions import ParkingFunctions
>>> ParkingFunctions(Integer(3)).list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3], [1, 3, 1],
 [3, 1, 1], [1, 2, 2], [2, 1, 2], [2, 2, 1], [1, 2, 3], [1, 3, 2],
 [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

If no size is specified, then ParkingFunctions returns the combinatorial class of all parking functions.

```
sage: PF = ParkingFunctions(); PF
Parking functions
sage: [] in PF
True
sage: [1] in PF
True
sage: [2] in PF
False
sage: [1,3,1] in PF
True
sage: [1,4,1] in PF
False
```

```
>>> from sage.all import *
>>> PF = ParkingFunctions(); PF
Parking functions
>>> [] in PF
True
>>> [Integer(1)] in PF
True
>>> [Integer(2)] in PF
False
>>> [Integer(1),Integer(3),Integer(1)] in PF
True
>>> [Integer(1),Integer(4),Integer(1)] in PF
False
```

If the size n is specified, then ParkingFunctions returns the combinatorial class of all parking functions of size n.

```
sage: PF = ParkingFunctions(0)
sage: PF.list()
[[]]
sage: PF = ParkingFunctions(1)
sage: PF.list()
[[1]]
sage: PF = ParkingFunctions(3)
sage: PF.list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3],
 [1, 3, 1], [3, 1, 1], [1, 2, 2], [2, 1, 2], [2, 2, 1],
 [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

```
>>> from sage.all import *
>>> PF = ParkingFunctions(Integer(0))
>>> PF.list()
[[]]
>>> PF = ParkingFunctions(Integer(1))
>>> PF.list()
[[1]]
>>> PF = ParkingFunctions(Integer(3))
>>> PF.list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3],
 [1, 3, 1], [3, 1, 1], [1, 2, 2], [2, 1, 2], [2, 2, 1],
 [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

```
sage: PF3 = ParkingFunctions(3); PF3
Parking functions of size 3
sage: [] in PF3
False
sage: [1] in PF3
False
sage: [1,3,1] in PF3
True
sage: [1,4,1] in PF3
False
```

```
>>> from sage.all import *
>>> PF3 = ParkingFunctions(Integer(3)); PF3
Parking functions of size 3
>>> [] in PF3
False
>>> [Integer(1)] in PF3
False
>>> [Integer(1),Integer(3),Integer(1)] in PF3
True
>>> [Integer(1),Integer(4),Integer(1)] in PF3
False
```

class sage.combinat.parking_functions.**ParkingFunctions_all**

Bases: *ParkingFunctions*

Element

alias of *ParkingFunction*

graded_component(*n*)

Return the graded component.

EXAMPLES:

```
sage: PF = ParkingFunctions()
sage: PF.graded_component(4) == ParkingFunctions(4)
True
sage: it = iter(ParkingFunctions()) # indirect doctest
sage: [next(it) for i in range(8)]
[[], [1], [1, 1], [1, 2], [2, 1], [1, 1, 1], [1, 1, 2], [1, 2, 1]]
```

```
>>> from sage.all import *
>>> PF = ParkingFunctions()
>>> PF.graded_component(Integer(4)) == ParkingFunctions(Integer(4))
True
>>> it = iter(ParkingFunctions()) # indirect doctest
>>> [next(it) for i in range(Integer(8))]
[[], [1], [1, 1], [1, 2], [2, 1], [1, 1, 1], [1, 1, 2], [1, 2, 1]]
```

class sage.combinat.parking_functions.**ParkingFunctions_n**(*n*)

Bases: *ParkingFunctions*

The combinatorial class of parking functions of size n.

A *parking function* of size n is a sequence $(a_1, \ldots, a_n)$ of positive integers such that if $b_1 \leq b_2 \leq \cdots \leq b_n$ is the increasing rearrangement of $a_1, \ldots, a_n$, then $b_i \leq i$.

A *parking function* of size n is a pair (L, D) of two sequences L and D where L is a permutation and D is an area sequence of a Dyck Path of size n such that $D[i] \geq 0$, $D[i+1] \leq D[i]+1$ and if $D[i+1] = D[i]+1$ then $L[i+1] > L[i]$.

The number of parking functions of size n is equal to the number of rooted forests on n vertices and is equal to $(n+1)^{n-1}$.

EXAMPLES:

```
sage: PF = ParkingFunctions(3)
sage: PF.list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3],
[1, 3, 1], [3, 1, 1], [1, 2, 2], [2, 1, 2], [2, 2, 1],
[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

sage: [ParkingFunctions(i).cardinality() for i in range(6)]
[1, 1, 3, 16, 125, 1296]
```

```
>>> from sage.all import *
>>> PF = ParkingFunctions(Integer(3))
>>> PF.list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3],
[1, 3, 1], [3, 1, 1], [1, 2, 2], [2, 1, 2], [2, 2, 1],
[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

>>> [ParkingFunctions(i).cardinality() for i in range(Integer(6))]
[1, 1, 3, 16, 125, 1296]
```

Warning

The precise order in which the parking function are generated or listed is not fixed, and may change in the future.

Element

alias of *ParkingFunction*

cardinality()

Return the number of parking functions of size `n`.

The cardinality is equal to $(n+1)^{n-1}$.

EXAMPLES:

```
sage: [ParkingFunctions(i).cardinality() for i in range(6)]
[1, 1, 3, 16, 125, 1296]
```

```
>>> from sage.all import *
>>> [ParkingFunctions(i).cardinality() for i in range(Integer(6))]
[1, 1, 3, 16, 125, 1296]
```

random_element()

Return a random parking function of size n.

The algorithm uses a circular parking space with $n+1$ spots. Then all n cars can park and there remains one empty spot. Spots are then renumbered so that the empty spot is 0.

The probability distribution is uniform on the set of $(n+1)^{n-1}$ parking functions of size n.

EXAMPLES:

```
sage: pf = ParkingFunctions(8)
sage: a = pf.random_element(); a  # random
[5, 7, 2, 4, 2, 5, 1, 3]
sage: a in pf
True
```

```
>>> from sage.all import *
>>> pf = ParkingFunctions(Integer(8))
>>> a = pf.random_element(); a  # random
[5, 7, 2, 4, 2, 5, 1, 3]
>>> a in pf
True
```

sage.combinat.parking_functions.**from_labelled_dyck_word**(*LDW*)

Return the parking function corresponding to the labelled Dyck word.

INPUT:

- `LDW` – labelled Dyck word

OUTPUT:

- the parking function corresponding to the labelled Dyck word that is half the size of `LDW`

EXAMPLES:

```
sage: from sage.combinat.parking_functions import from_labelled_dyck_word
sage: LDW = [2, 6, 0, 4, 5, 0, 0, 0, 3, 7, 0, 1, 0, 0]
sage: from_labelled_dyck_word(LDW)
[6, 1, 5, 2, 2, 1, 5]
```

```
>>> from sage.all import *
>>> from sage.combinat.parking_functions import from_labelled_dyck_word
>>> LDW = [Integer(2), Integer(6), Integer(0), Integer(4), Integer(5), Integer(0),
→ Integer(0), Integer(0), Integer(3), Integer(7), Integer(0), Integer(1),␣
→Integer(0), Integer(0)]
```

(continues on next page)

(continued from previous page)

```
>>> from_labelled_dyck_word(LDW)
[6, 1, 5, 2, 2, 1, 5]
```

```
sage: from_labelled_dyck_word([2, 3, 0, 0, 1, 0, 4, 0])
[3, 1, 1, 4]
sage: from_labelled_dyck_word([2, 3, 4, 0, 0, 0, 1, 0])
[4, 1, 1, 1]
sage: from_labelled_dyck_word([2, 4, 0, 1, 0, 0, 3, 0])
[2, 1, 4, 1]
```

```
>>> from sage.all import *
>>> from_labelled_dyck_word([Integer(2), Integer(3), Integer(0), Integer(0),␣
↪Integer(1), Integer(0), Integer(4), Integer(0)])
[3, 1, 1, 4]
>>> from_labelled_dyck_word([Integer(2), Integer(3), Integer(4), Integer(0),␣
↪Integer(0), Integer(0), Integer(1), Integer(0)])
[4, 1, 1, 1]
>>> from_labelled_dyck_word([Integer(2), Integer(4), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(3), Integer(0)])
[2, 1, 4, 1]
```

sage.combinat.parking_functions.**from_labelling_and_area_sequence**(L, D)

Return the parking function corresponding to the labelling area sequence pair.

INPUT:

- `L` – a labelling permutation
- `D` – an area sequence for a Dyck word

OUTPUT:

- the parking function corresponding the labelling permutation `L` and `D` an area sequence of the corresponding Dyck path

EXAMPLES:

```
sage: from sage.combinat.parking_functions import from_labelling_and_area_sequence
sage: from_labelling_and_area_sequence([2, 6, 4, 5, 3, 7, 1], [0, 1, 1, 2, 0, 1,␣
↪1])
[6, 1, 5, 2, 2, 1, 5]
```

```
>>> from sage.all import *
>>> from sage.combinat.parking_functions import from_labelling_and_area_sequence
>>> from_labelling_and_area_sequence([Integer(2), Integer(6), Integer(4),␣
↪Integer(5), Integer(3), Integer(7), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(1), Integer(2), Integer(0), Integer(1), Integer(1)])
[6, 1, 5, 2, 2, 1, 5]
```

```
sage: from_labelling_and_area_sequence([1, 2, 3], [0, 1, 2])
[1, 1, 1]
sage: from_labelling_and_area_sequence([1, 2, 3], [0, 0, 0])
[1, 2, 3]
sage: from_labelling_and_area_sequence([1, 2, 3], [0, 1, 1])
```

(continues on next page)

(continued from previous page)

```
[1, 1, 2]
sage: from_labelling_and_area_sequence([1, 2, 4, 3], [0, 1, 2, 1])
[1, 1, 3, 1]
```

```
>>> from sage.all import *
>>> from_labelling_and_area_sequence([Integer(1), Integer(2), Integer(3)],
↪[Integer(0), Integer(1), Integer(2)])
[1, 1, 1]
>>> from_labelling_and_area_sequence([Integer(1), Integer(2), Integer(3)],
↪[Integer(0), Integer(0), Integer(0)])
[1, 2, 3]
>>> from_labelling_and_area_sequence([Integer(1), Integer(2), Integer(3)],
↪[Integer(0), Integer(1), Integer(1)])
[1, 1, 2]
>>> from_labelling_and_area_sequence([Integer(1), Integer(2), Integer(4),
↪Integer(3)], [Integer(0), Integer(1), Integer(2), Integer(1)])
[1, 1, 3, 1]
```

`sage.combinat.parking_functions.`**`is_a`**(*x*, *n=None*)

Check whether a list is a parking function.

If a size n is specified, checks if a list is a parking function of size n.

5.1.159 Catalog of Path Tableaux

The `path_tableaux` object may be used to access examples of various path tableau objects currently implemented in Sage. Using tab-completion on this object is an easy way to discover and quickly create the path tableaux that are available (as listed here).

Let `<tab>` indicate pressing the `Tab` key. So begin by typing `path_tableaux.<tab>` to the see the currently implemented path tableaux.

- *`CylindricalDiagram`*
- *`DyckPath`*
- *`DyckPaths`*
- *`FriezePattern`*
- *`FriezePatterns`*
- *`SemistandardPathTableau`*
- *`SemistandardPathTableaux`*

5.1.160 Dyck Paths

This is an implementation of the abstract base class *`sage.combinat.path_tableaux.path_tableau.PathTableau`*. This is the simplest implementation of a path tableau and is included to provide a convenient test case and for pedagogical purposes.

In this implementation we have sequences of nonnegative integers. These are required to be the heights Dyck words (except that we do not require the sequence to start or end at height zero). These are in bijection with skew standard tableaux with at most two rows. Sequences which start and end at height zero are in bijection with noncrossing perfect matchings.

AUTHORS:

- Bruce Westbury (2018): initial version

class sage.combinat.path_tableaux.dyck_path.**DyckPath**(*parent*, *ot*, *check=True*)

Bases: *PathTableau*

An instance is the sequence of nonnegative integers given by the heights of a Dyck word.

INPUT:

- a sequence of nonnegative integers
- a two row standard skew tableau
- a Dyck word
- a noncrossing perfect matching

EXAMPLES:

```
sage: path_tableaux.DyckPath([0,1,2,1,0])
[0, 1, 2, 1, 0]

sage: w = DyckWord([1,1,0,0])
sage: path_tableaux.DyckPath(w)
[0, 1, 2, 1, 0]

sage: p = PerfectMatching([(1,2), (3,4)])
sage: path_tableaux.DyckPath(p)
[0, 1, 0, 1, 0]

sage: t = Tableau([[1,2,4],[3,5,6]])
sage: path_tableaux.DyckPath(t)
[0, 1, 2, 1, 2, 1, 0]

sage: st = SkewTableau([[None, 1,4],[2,3]])
sage: path_tableaux.DyckPath(st)
[1, 2, 1, 0, 1]
```

```
>>> from sage.all import *
>>> path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(0)])
[0, 1, 2, 1, 0]

>>> w = DyckWord([Integer(1),Integer(1),Integer(0),Integer(0)])
>>> path_tableaux.DyckPath(w)
[0, 1, 2, 1, 0]

>>> p = PerfectMatching([(Integer(1),Integer(2)), (Integer(3),Integer(4))])
>>> path_tableaux.DyckPath(p)
[0, 1, 0, 1, 0]

>>> t = Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3),Integer(5),
↪Integer(6)]])
>>> path_tableaux.DyckPath(t)
[0, 1, 2, 1, 2, 1, 0]

>>> st = SkewTableau([[None, Integer(1),Integer(4)],[Integer(2),Integer(3)]])
```

(continues on next page)

(continued from previous page)

```
>>> path_tableaux.DyckPath(st)
[1, 2, 1, 0, 1]
```

Here we illustrate the slogan that promotion = rotation:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.to_perfect_matching()
[(0, 5), (1, 4), (2, 3)]

sage: t = t.promotion()
sage: t.to_perfect_matching()
[(0, 3), (1, 2), (4, 5)]

sage: t = t.promotion()
sage: t.to_perfect_matching()
[(0, 1), (2, 5), (3, 4)]

sage: t = t.promotion()
sage: t.to_perfect_matching()
[(0, 5), (1, 4), (2, 3)]
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.to_perfect_matching()
[(0, 5), (1, 4), (2, 3)]

>>> t = t.promotion()
>>> t.to_perfect_matching()
[(0, 3), (1, 2), (4, 5)]

>>> t = t.promotion()
>>> t.to_perfect_matching()
[(0, 1), (2, 5), (3, 4)]

>>> t = t.promotion()
>>> t.to_perfect_matching()
[(0, 5), (1, 4), (2, 3)]
```

check()

Check that `self` is a valid path.

EXAMPLES:

```
sage: path_tableaux.DyckPath([0,1,0,-1,0]) # indirect doctest
Traceback (most recent call last):
...
ValueError: [0, 1, 0, -1, 0] has a negative entry

sage: path_tableaux.DyckPath([0,1,3,1,0]) # indirect doctest
Traceback (most recent call last):
...
ValueError: [0, 1, 3, 1, 0] is not a Dyck path
```

```
>>> from sage.all import *
>>> path_tableaux.DyckPath([Integer(0),Integer(1),Integer(0),-Integer(1),
↪Integer(0)]) # indirect doctest
Traceback (most recent call last):
...
ValueError: [0, 1, 0, -1, 0] has a negative entry

>>> path_tableaux.DyckPath([Integer(0),Integer(1),Integer(3),Integer(1),
↪Integer(0)]) # indirect doctest
Traceback (most recent call last):
...
ValueError: [0, 1, 3, 1, 0] is not a Dyck path
```

descents()

Return the descent set of `self`.

EXAMPLES:

```
sage: path_tableaux.DyckPath([0,1,2,1,2,1,0,1,0]).descents()
{3, 6}
```

```
>>> from sage.all import *
>>> path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(1),Integer(0),Integer(1),Integer(0)]).descents()
{3, 6}
```

is_skew()

Return `True` if `self` is skew and `False` if not.

EXAMPLES:

```
sage: path_tableaux.DyckPath([0,1,2,1]).is_skew()
False

sage: path_tableaux.DyckPath([1,0,1,2,1]).is_skew()
True
```

```
>>> from sage.all import *
>>> path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(1)]).is_
↪skew()
False

>>> path_tableaux.DyckPath([Integer(1),Integer(0),Integer(1),Integer(2),
↪Integer(1)]).is_skew()
True
```

local_rule(i)

This has input a list of objects. This method first takes the list of objects of length three consisting of the $(i-1)$-st, i-th and $(i+1)$-term and applies the rule. It then replaces the i-th object by the object returned by the rule.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.local_rule(3)
[0, 1, 2, 1, 2, 1, 0]
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.local_rule(Integer(3))
[0, 1, 2, 1, 2, 1, 0]
```

to_DyckWord()

Convert `self` to a Dyck word.

EXAMPLES:

```
sage: c = path_tableaux.DyckPath([0,1,2,1,0])
sage: c.to_DyckWord()
[1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> c = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(0)])
>>> c.to_DyckWord()
[1, 1, 0, 0]
```

to_perfect_matching()

Return the perfect matching associated to `self`.

EXAMPLES:

```
sage: path_tableaux.DyckPath([0,1,2,1,2,1,0,1,0]).to_perfect_matching()
[(0, 5), (1, 2), (3, 4), (6, 7)]
```

```
>>> from sage.all import *
>>> path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(1),Integer(0),Integer(1),Integer(0)]).to_perfect_
↪matching()
[(0, 5), (1, 2), (3, 4), (6, 7)]
```

to_tableau()

Return the skew tableau associated to `self`.

EXAMPLES:

```
sage: T = path_tableaux.DyckPath([0,1,2,3,2,3])
sage: T.to_tableau()
[[1, 2, 3, 5], [4]]

sage: U = path_tableaux.DyckPath([2,3,2,3])
sage: U.to_tableau()
[[None, None, 1, 3], [2]]
```

```
>>> from sage.all import *
>>> T = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(3)])
>>> T.to_tableau()
[[1, 2, 3, 5], [4]]

>>> U = path_tableaux.DyckPath([Integer(2),Integer(3),Integer(2),Integer(3)])
>>> U.to_tableau()
[[None, None, 1, 3], [2]]
```

to_word()

Return the word in the alphabet $\{0, 1\}$ associated to `self`.

EXAMPLES:

```
sage: path_tableaux.DyckPath([1,0,1,2,1]).to_word()
[0, 1, 1, 0]
```

```
>>> from sage.all import *
>>> path_tableaux.DyckPath([Integer(1),Integer(0),Integer(1),Integer(2),
↪Integer(1)]).to_word()
[0, 1, 1, 0]
```

class sage.combinat.path_tableaux.dyck_path.**DyckPaths**

Bases: *PathTableaux*

The parent class for DyckPath.

Element

alias of *DyckPath*

5.1.161 Frieze Patterns

This implements the original frieze patterns due to Conway and Coxeter. Such a frieze pattern is considered as a sequence of nonnegative integers following [CoCo1] and [CoCo2] using *sage.combinat.path_tableaux.path_tableau*.

AUTHORS:

- Bruce Westbury (2019): initial version

class sage.combinat.path_tableaux.frieze.**FriezePattern**

Bases: *PathTableau*

A frieze pattern.

We encode a frieze pattern as a sequence in a fixed ground field.

INPUT:

- `fp` – a sequence of elements of `field`
- `field` – (default: `QQ`) the ground field

EXAMPLES:

```
sage: t = path_tableaux.FriezePattern([1,2,1,2,3,1])
sage: path_tableaux.CylindricalDiagram(t)
[0, 1, 2, 1, 2, 3, 1, 0]
```

(continues on next page)

(continued from previous page)

```
[ , 0, 1, 1, 3, 5, 2, 1, 0]
[ ,  , 0, 1, 4, 7, 3, 2, 1, 0]
[ ,  ,  , 0, 1, 2, 1, 1, 1, 1, 0]
[ ,  ,  ,  , 0, 1, 1, 2, 3, 4, 1, 0]
[ ,  ,  ,  ,  , 0, 1, 3, 5, 7, 2, 1, 0]
[ ,  ,  ,  ,  ,  , 0, 1, 2, 3, 1, 1, 1, 0]
[ ,  ,  ,  ,  ,  ,  , 0, 1, 2, 1, 2, 3, 1, 0]

sage: TestSuite(t).run()

sage: t = path_tableaux.FriezePattern([1,2,7,5,3,7,4,1])
sage: path_tableaux.CylindricalDiagram(t)
[0, 1, 2, 7, 5, 3, 7, 4, 1, 0]
[ , 0, 1, 4, 3, 2, 5, 3, 1, 1, 0]
[ ,  , 0, 1, 1, 1, 3, 2, 1, 2, 1, 0]
[ ,  ,  , 0, 1, 2, 7, 5, 3, 7, 4, 1, 0]
[ ,  ,  ,  , 0, 1, 4, 3, 2, 5, 3, 1, 1, 0]
[ ,  ,  ,  ,  , 0, 1, 1, 1, 3, 2, 1, 2, 1, 0]
[ ,  ,  ,  ,  ,  , 0, 1, 2, 7, 5, 3, 7, 4, 1, 0]
[ ,  ,  ,  ,  ,  ,  , 0, 1, 4, 3, 2, 5, 3, 1, 1, 0]
[ ,  ,  ,  ,  ,  ,  ,  , 0, 1, 1, 1, 3, 2, 1, 2, 1, 0]
[ ,  ,  ,  ,  ,  ,  ,  ,  , 0, 1, 2, 7, 5, 3, 7, 4, 1, 0]

 sage: TestSuite(t).run()

sage: t = path_tableaux.FriezePattern([1,3,4,5,1])
sage: path_tableaux.CylindricalDiagram(t)
[  0,   1,   3,   4,   5,   1,   0]
[   ,   0,   1, 5/3, 7/3, 2/3,   1,   0]
[   ,    ,   0,   1,   2,   1,   3,   1,   0]
[   ,    ,    ,   0,   1,   1,   4, 5/3,   1,   0]
[   ,    ,    ,    ,   0,   1,   5, 7/3,   2,   1,   0]
[   ,    ,    ,    ,    ,   0,   1, 2/3,   1,   1,   1,   0]
[   ,    ,    ,    ,    ,    ,   0,   1,   3,   4,   5,   1,   0]

sage: TestSuite(t).run()
```

```
>>> from sage.all import *
>>> t = path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(3),Integer(1)])
>>> path_tableaux.CylindricalDiagram(t)
[0, 1, 2, 1, 2, 3, 1, 0]
[ , 0, 1, 1, 3, 5, 2, 1, 0]
[ ,  , 0, 1, 4, 7, 3, 2, 1, 0]
[ ,  ,  , 0, 1, 2, 1, 1, 1, 1, 0]
[ ,  ,  ,  , 0, 1, 1, 2, 3, 4, 1, 0]
[ ,  ,  ,  ,  , 0, 1, 3, 5, 7, 2, 1, 0]
[ ,  ,  ,  ,  ,  , 0, 1, 2, 3, 1, 1, 1, 0]
[ ,  ,  ,  ,  ,  ,  , 0, 1, 2, 1, 2, 3, 1, 0]

>>> TestSuite(t).run()
```

(continues on next page)

(continued from previous page)

```
>>> t = path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(7),Integer(5),
↪Integer(3),Integer(7),Integer(4),Integer(1)])
>>> path_tableaux.CylindricalDiagram(t)
[0, 1, 2, 7, 5, 3, 7, 4, 1, 0]
[ , 0, 1, 4, 3, 2, 5, 3, 1, 1, 0]
[ ,  , 0, 1, 1, 1, 3, 2, 1, 2, 1, 0]
[ ,  ,  , 0, 1, 2, 7, 5, 3, 7, 4, 1, 0]
[ ,  ,  ,  , 0, 1, 4, 3, 2, 5, 3, 1, 1, 0]
[ ,  ,  ,  ,  , 0, 1, 1, 1, 3, 2, 1, 2, 1, 0]
[ ,  ,  ,  ,  ,  , 0, 1, 2, 7, 5, 3, 7, 4, 1, 0]
[ ,  ,  ,  ,  ,  ,  , 0, 1, 4, 3, 2, 5, 3, 1, 1, 0]
[ ,  ,  ,  ,  ,  ,  ,  , 0, 1, 1, 1, 3, 2, 1, 2, 1, 0]
[ ,  ,  ,  ,  ,  ,  ,  ,  , 0, 1, 2, 7, 5, 3, 7, 4, 1, 0]

>>> TestSuite(t).run()

>>> t = path_tableaux.FriezePattern([Integer(1),Integer(3),Integer(4),Integer(5),
↪Integer(1)])
>>> path_tableaux.CylindricalDiagram(t)
[  0,   1,   3,   4,   5,   1,   0]
[   ,   0,   1, 5/3, 7/3, 2/3,   1,   0]
[   ,    ,   0,   1,   2,   1,   3,   1,   0]
[   ,    ,    ,   0,   1,   1,   4, 5/3,   1,   0]
[   ,    ,    ,    ,   0,   1,   5, 7/3,   2,   1,   0]
[   ,    ,    ,    ,    ,   0,   1, 2/3,   1,   1,   1,   0]
[   ,    ,    ,    ,    ,    ,   0,   1,   3,   4,   5,   1,   0]

>>> TestSuite(t).run()
```

This constructs the examples from [HJ18]:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<sqrt3> = NumberField(x^2 - 3)
sage: t = path_tableaux.FriezePattern([1, sqrt3, 2, sqrt3, 1, 1], field=K)
sage: path_tableaux.CylindricalDiagram(t)
[        0,         1,     sqrt3,         2,     sqrt3,         1,         1,    ␣
↪    0]
[         ,         0,         1,     sqrt3,         2,     sqrt3, sqrt3 + 1,    ␣
↪    1,         0]
[         ,          ,         0,         1,     sqrt3,         2, sqrt3 + 2,    ␣
↪sqrt3,         1,         0]
[         ,          ,          ,         0,         1,     sqrt3, sqrt3 + 2,    ␣
↪    2,     sqrt3,         1,         0]
[         ,          ,          ,          ,         0,         1, sqrt3 + 1,    ␣
↪sqrt3,         2,     sqrt3,         1,         0]
[         ,          ,          ,          ,          ,         0,         1,    ␣
↪    1,     sqrt3,         2,     sqrt3,         1,         0]
[         ,          ,          ,          ,          ,          ,         0,    ␣
↪    1, sqrt3 + 1, sqrt3 + 2, sqrt3 + 2, sqrt3 + 1,         1,         0]
[         ,          ,          ,          ,          ,          ,          ,    ␣
↪    0,         1,     sqrt3,         2,     sqrt3,         1,         1,    ␣
```

(continues on next page)

(continued from previous page)

```
↪ 0]
sage: TestSuite(t).run()

sage: # needs sage.rings.number_field
sage: K.<sqrt2> = NumberField(x^2 - 2)
sage: t = path_tableaux.FriezePattern([1, sqrt2, 1, sqrt2, 3, 2*sqrt2, 5, 3*sqrt2,
↪ 1],
....:                                    field=K)
sage: path_tableaux.CylindricalDiagram(t)
[      0,       1,   sqrt2,       1,   sqrt2,       3, 2*sqrt2,       5, 3*sqrt2,␣
↪      1,       0]
[       ,       0,       1,   sqrt2,       3, 5*sqrt2,       7, 9*sqrt2,      11,␣
↪2*sqrt2,       1,       0]
[       ,        ,       0,       1, 2*sqrt2,       7, 5*sqrt2,      13, 8*sqrt2,␣
↪      3,   sqrt2,       1,       0]
[       ,        ,        ,       0,       1, 2*sqrt2,       3, 4*sqrt2,       5,␣
↪  sqrt2,       1,   sqrt2,       1,       0]
[       ,        ,        ,        ,       0,       1,   sqrt2,       3, 2*sqrt2,␣
↪      1,   sqrt2,       3, 2*sqrt2,       1,       0]
[       ,        ,        ,        ,        ,       0,       1, 2*sqrt2,       3,␣
↪  sqrt2,       3, 5*sqrt2,       7, 2*sqrt2,       1,       0]
[       ,        ,        ,        ,        ,        ,       0,       1,   sqrt2,␣
↪      1, 2*sqrt2,       7, 5*sqrt2,       3,   sqrt2,       1,       0]
[       ,        ,        ,        ,        ,        ,        ,       0,       1,␣
↪  sqrt2,       5, 9*sqrt2,      13, 4*sqrt2,       3, 2*sqrt2,       1,       0]
[       ,        ,        ,        ,        ,        ,        ,        ,       0,␣
↪      1, 3*sqrt2,      11, 8*sqrt2,       5, 2*sqrt2,       3,   sqrt2,       1,
↪       0]
[       ,        ,        ,        ,        ,        ,        ,        ,        ,␣
↪      0,       1, 2*sqrt2,       3,   sqrt2,       1,   sqrt2,       1,   sqrt2,
↪       1,       0]
[       ,        ,        ,        ,        ,        ,        ,        ,        ,␣
↪       ,       0,       1,   sqrt2,       1,   sqrt2,       3, 2*sqrt2,       5,
↪ 3*sqrt2,       1,       0]
sage: TestSuite(t).run()
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(ZZ, 'x')
>>> K = NumberField(x**Integer(2) - Integer(3), names=('sqrt3',)); (sqrt3,) = K._
↪first_ngens(1)
>>> t = path_tableaux.FriezePattern([Integer(1), sqrt3, Integer(2), sqrt3,␣
↪Integer(1), Integer(1)], field=K)
>>> path_tableaux.CylindricalDiagram(t)
[         0,          1,      sqrt3,          2,      sqrt3,          1,          1,     ␣
↪    0]
[          ,          0,          1,      sqrt3,          2,      sqrt3, sqrt3 + 1,     ␣
↪    1,          0]
[          ,           ,          0,          1,      sqrt3,          2, sqrt3 + 2,     ␣
↪sqrt3,          1,          0]
[          ,           ,           ,          0,          1,      sqrt3, sqrt3 + 2,     ␣
↪    2,      sqrt3,          1,          0]
```

(continues on next page)

(continued from previous page)

```
[           ,           ,           ,           ,          0,          1, sqrt3 + 1,    ␣
↪sqrt3,          2,      sqrt3,          1,          0]
[           ,           ,           ,           ,           ,          0,          1,    ␣
↪    1,      sqrt3,          2,      sqrt3,          1,          0]
[           ,           ,           ,           ,           ,           ,          0,    ␣
↪    1, sqrt3 + 1, sqrt3 + 2, sqrt3 + 2, sqrt3 + 1,          1,          0]
[           ,           ,           ,           ,           ,           ,           ,    ␣
↪    0,          1,      sqrt3,          2,      sqrt3,          1,          1,         ␣
↪ 0]
>>> TestSuite(t).run()

>>> # needs sage.rings.number_field
>>> K = NumberField(x**Integer(2) - Integer(2), names=('sqrt2',)); (sqrt2,) = K._
↪first_ngens(1)
>>> t = path_tableaux.FriezePattern([Integer(1), sqrt2, Integer(1), sqrt2,␣
↪Integer(3), Integer(2)*sqrt2, Integer(5), Integer(3)*sqrt2, Integer(1)],
...                                  field=K)
>>> path_tableaux.CylindricalDiagram(t)
[      0,        1,    sqrt2,        1,    sqrt2,        3, 2*sqrt2,        5, 3*sqrt2,␣
↪      1,        0]
[       ,        0,        1,    sqrt2,        3, 5*sqrt2,        7, 9*sqrt2,       11,␣
↪2*sqrt2,        1,        0]
[       ,         ,        0,        1, 2*sqrt2,        7, 5*sqrt2,       13, 8*sqrt2,␣
↪      3,    sqrt2,        1,        0]
[       ,         ,         ,        0,        1, 2*sqrt2,        3, 4*sqrt2,        5,␣
↪  sqrt2,        1,    sqrt2,        1,        0]
[       ,         ,         ,         ,        0,        1,    sqrt2,        3, 2*sqrt2,␣
↪      1,    sqrt2,        3, 2*sqrt2,        1,        0]
[       ,         ,         ,         ,         ,        0,        1, 2*sqrt2,        3,␣
↪  sqrt2,        3, 5*sqrt2,        7, 2*sqrt2,        1,        0]
[       ,         ,         ,         ,         ,         ,        0,        1,    sqrt2,␣
↪      1, 2*sqrt2,        7, 5*sqrt2,        3,    sqrt2,        1,        0]
[       ,         ,         ,         ,         ,         ,         ,        0,        1,␣
↪  sqrt2,        5, 9*sqrt2,       13, 4*sqrt2,        3, 2*sqrt2,        1,        0]
[       ,         ,         ,         ,         ,         ,         ,         ,        0,␣
↪      1, 3*sqrt2,       11, 8*sqrt2,        5, 2*sqrt2,        3,    sqrt2,        1,
↪       0]
[       ,         ,         ,         ,         ,         ,         ,         ,         ,␣
↪      0,        1, 2*sqrt2,        3,    sqrt2,        1,    sqrt2,        1,    sqrt2,
↪       1,        0]
[       ,         ,         ,         ,         ,         ,         ,         ,         ,␣
↪       ,        0,        1,    sqrt2,        1,    sqrt2,        3, 2*sqrt2,        5,
↪ 3*sqrt2,        1,        0]
>>> TestSuite(t).run()
```

change_ring(R)

Return `self` as a frieze pattern with coefficients in `R`.

This assumes that there is a canonical coerce map from the base ring of `self` to `R`.

EXAMPLES:

```
sage: fp = path_tableaux.FriezePattern([1,2,7,5,3,7,4,1])
```

(continues on next page)

(continued from previous page)

```
sage: fp.change_ring(RealField())                                           #␣
↪needs sage.rings.real_mpfr
[0.000000000000000, 1.00000000000000, ...
 4.00000000000000, 1.00000000000000, 0.000000000000000]
sage: fp.change_ring(GF(7))
Traceback (most recent call last):
...
TypeError: no base extension defined
```

```
>>> from sage.all import *
>>> fp = path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(7),
↪Integer(5),Integer(3),Integer(7),Integer(4),Integer(1)])
>>> fp.change_ring(RealField())                                             #␣
↪needs sage.rings.real_mpfr
[0.000000000000000, 1.00000000000000, ...
 4.00000000000000, 1.00000000000000, 0.000000000000000]
>>> fp.change_ring(GF(Integer(7)))
Traceback (most recent call last):
...
TypeError: no base extension defined
```

check()

Check that `self` is a valid frieze pattern.

is_integral()

Return `True` if all entries of the frieze pattern are positive integers.

EXAMPLES:

```
sage: path_tableaux.FriezePattern([1,2,7,5,3,7,4,1]).is_integral()
True

sage: path_tableaux.FriezePattern([1,3,4,5,1]).is_integral()
False
```

```
>>> from sage.all import *
>>> path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(7),Integer(5),
↪Integer(3),Integer(7),Integer(4),Integer(1)]).is_integral()
True

>>> path_tableaux.FriezePattern([Integer(1),Integer(3),Integer(4),Integer(5),
↪Integer(1)]).is_integral()
False
```

is_positive()

Return `True` if all elements of `self` are positive.

This implies that all entries of `CylindricalDiagram(self)` are positive.

> **Warning**
>
> There are orders on all fields. These may not be ordered fields as they may not be compatible with the

field operations. This method is intended to be used with ordered fields only.

EXAMPLES:

```
sage: path_tableaux.FriezePattern([1,2,7,5,3,7,4,1]).is_positive()
True

sage: path_tableaux.FriezePattern([1,-3,4,5,1]).is_positive()
False

sage: x = polygen(ZZ, 'x')
sage: K.<sqrt3> = NumberField(x^2 - 3)                                    #␣
↪needs sage.rings.number_field
sage: path_tableaux.FriezePattern([1,sqrt3,1], K).is_positive()           #␣
↪needs sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(7),Integer(5),
↪Integer(3),Integer(7),Integer(4),Integer(1)]).is_positive()
True

>>> path_tableaux.FriezePattern([Integer(1),-Integer(3),Integer(4),Integer(5),
↪Integer(1)]).is_positive()
False

>>> x = polygen(ZZ, 'x')
>>> K = NumberField(x**Integer(2) - Integer(3), names=('sqrt3',)); (sqrt3,) =␣
↪K._first_ngens(1)# needs sage.rings.number_field
>>> path_tableaux.FriezePattern([Integer(1),sqrt3,Integer(1)], K).is_
↪positive()             # needs sage.rings.number_field
True
```

is_skew()

Return `True` if `self` is skew and `False` if not.

EXAMPLES:

```
sage: path_tableaux.FriezePattern([1,2,1,2,3,1]).is_skew()
False

sage: path_tableaux.FriezePattern([2,2,1,2,3,1]).is_skew()
True
```

```
>>> from sage.all import *
>>> path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(3),Integer(1)]).is_skew()
False

>>> path_tableaux.FriezePattern([Integer(2),Integer(2),Integer(1),Integer(2),
↪Integer(3),Integer(1)]).is_skew()
True
```

local_rule(*i*)

Return the i-th local rule on `self`.

This interprets `self` as a list of objects. This method first takes the list of objects of length three consisting of the $(i-1)$-st, i-th and $(i+1)$-term and applies the rule. It then replaces the i-th object by the object returned by the rule.

EXAMPLES:

```
sage: t = path_tableaux.FriezePattern([1,2,1,2,3,1])
sage: t.local_rule(3)
[1, 2, 5, 2, 3, 1]

sage: t = path_tableaux.FriezePattern([1,2,1,2,3,1])
sage: t.local_rule(0)
Traceback (most recent call last):
...
ValueError: 0 is not a valid integer
```

```
>>> from sage.all import *
>>> t = path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(3),Integer(1)])
>>> t.local_rule(Integer(3))
[1, 2, 5, 2, 3, 1]

>>> t = path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(3),Integer(1)])
>>> t.local_rule(Integer(0))
Traceback (most recent call last):
...
ValueError: 0 is not a valid integer
```

plot(*model='UHP'*)

Plot the frieze as an ideal hyperbolic polygon.

This is only defined up to isometry of the hyperbolic plane.

We are identifying the boundary of the hyperbolic plane with the real projective line.

The option `model` must be one of

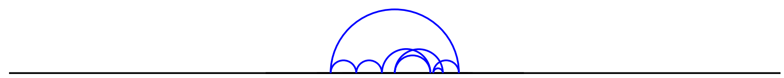
- `'UHP'` – (default) for the upper half plane model
- `'PD'` – for the Poincare disk model
- `'KM'` – for the Klein model

The hyperboloid model is not an option as this does not implement boundary points.

EXAMPLES:

```
sage: # needs sage.plot sage.symbolic
sage: t = path_tableaux.FriezePattern([1,2,7,5,3,7,4,1])
sage: t.plot()
Graphics object consisting of 18 graphics primitives
sage: t.plot(model='UHP')
Graphics object consisting of 18 graphics primitives
sage: t.plot(model='PD')
```

(continues on next page)



(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: '>' not supported between instances of 'NotANumber' and 'Pi'
sage: t.plot(model='KM')
Graphics object consisting of 18 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot sage.symbolic
>>> t = path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(7),
↪Integer(5),Integer(3),Integer(7),Integer(4),Integer(1)])
>>> t.plot()
Graphics object consisting of 18 graphics primitives
>>> t.plot(model='UHP')
Graphics object consisting of 18 graphics primitives
>>> t.plot(model='PD')
Traceback (most recent call last):
...
TypeError: '>' not supported between instances of 'NotANumber' and 'Pi'
>>> t.plot(model='KM')
Graphics object consisting of 18 graphics primitives
```

triangulation()

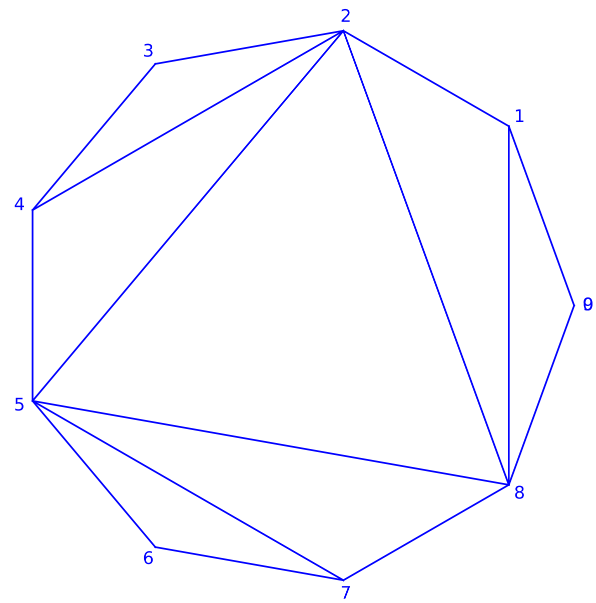
Plot a regular polygon with some diagonals.

If `self` is positive and integral then this will be a triangulation.

EXAMPLES:

```
sage: path_tableaux.FriezePattern([1,2,7,5,3,7,4,1]).triangulation()          #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 25 graphics primitives
```

(continues on next page)



(continued from previous page)

```
sage: path_tableaux.FriezePattern([1,2,1/7,5,3]).triangulation()          #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 12 graphics primitives


sage: x = polygen(ZZ, 'x')
sage: K.<sqrt2> = NumberField(x^2 - 2)                                     #␣
↪needs sage.rings.number_field
sage: path_tableaux.FriezePattern([1,sqrt2,1,sqrt2,3,2*sqrt2,5,3*sqrt2,1],   #␣
↪needs sage.plot sage.rings.number_field sage.symbolic
....:                             field=K).triangulation()
Graphics object consisting of 24 graphics primitives
```

```
>>> from sage.all import *
>>> path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(7),Integer(5),
↪Integer(3),Integer(7),Integer(4),Integer(1)]).triangulation()        #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 25 graphics primitives

>>> path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(1)/Integer(7),
↪Integer(5),Integer(3)]).triangulation()            # needs sage.plot sage.
↪symbolic
Graphics object consisting of 12 graphics primitives


>>> x = polygen(ZZ, 'x')
>>> K = NumberField(x**Integer(2) - Integer(2), names=('sqrt2',)); (sqrt2,) =␣
↪K._first_ngens(1)# needs sage.rings.number_field
>>> path_tableaux.FriezePattern([Integer(1),sqrt2,Integer(1),sqrt2,Integer(3),
↪Integer(2)*sqrt2,Integer(5),Integer(3)*sqrt2,Integer(1)],   # needs sage.
↪plot sage.rings.number_field sage.symbolic
```

(continues on next page)

(continued from previous page)

```
...                             field=K).triangulation()
Graphics object consisting of 24 graphics primitives
```

width()

Return the width of `self`.

If the first and last terms of `self` are 1 then this is the length of `self` plus two and otherwise is undefined.

EXAMPLES:

```
sage: path_tableaux.FriezePattern([1,2,1,2,3,1]).width()
8

sage: path_tableaux.FriezePattern([1,2,1,2,3,4]).width() is None
True
```

```
>>> from sage.all import *
>>> path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(3),Integer(1)]).width()
8

>>> path_tableaux.FriezePattern([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(3),Integer(4)]).width() is None
True
```

class sage.combinat.path_tableaux.frieze.**FriezePatterns**(*field*)

Bases: *PathTableaux*

The set of all frieze patterns.

EXAMPLES:

```
sage: P = path_tableaux.FriezePatterns(QQ)
sage: fp = P((1, 1, 1))
sage: fp
[1]
sage: path_tableaux.CylindricalDiagram(fp)
[1, 1, 1]
[ , 1, 2, 1]
[ ,  , 1, 1, 1]
```

```
>>> from sage.all import *
>>> P = path_tableaux.FriezePatterns(QQ)
>>> fp = P((Integer(1), Integer(1), Integer(1)))
>>> fp
[1]
>>> path_tableaux.CylindricalDiagram(fp)
[1, 1, 1]
[ , 1, 2, 1]
[ ,  , 1, 1, 1]
```

Element

alias of *FriezePattern*

5.1.162 Path Tableaux

This is an abstract base class for using local rules to construct rectification and the action of the cactus group [Wes2017].

This is a construction of the Henriques-Kamnitzer construction of the action of the cactus group on tensor powers of a crystal. This is also a generalisation of the Fomin growth rules, which are a version of the operations on standard tableaux which were previously constructed using jeu de taquin.

The basic operations are rectification, evacuation and promotion. Rectification of standard skew tableaux agrees with the rectification by jeu de taquin as does evacuation. Promotion agrees with promotion by jeu de taquin on rectangular tableaux but in general they are different.

REFERENCES:

- [Wes2017]

AUTHORS:

- Bruce Westbury (2018): initial version

class sage.combinat.path_tableaux.path_tableau.**CylindricalDiagram**(T)

Bases: SageObject

Cylindrical growth diagrams.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: path_tableaux.CylindricalDiagram(t)
[0, 1, 2, 3, 2, 1, 0]
[ , 0, 1, 2, 1, 0, 1, 0]
[ ,  , 0, 1, 0, 1, 2, 1, 0]
[ ,  ,  , 0, 1, 2, 3, 2, 1, 0]
[ ,  ,  ,  , 0, 1, 2, 1, 0, 1, 0]
[ ,  ,  ,  ,  , 0, 1, 0, 1, 2, 1, 0]
[ ,  ,  ,  ,  ,  , 0, 1, 2, 3, 2, 1, 0]
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> path_tableaux.CylindricalDiagram(t)
[0, 1, 2, 3, 2, 1, 0]
[ , 0, 1, 2, 1, 0, 1, 0]
[ ,  , 0, 1, 0, 1, 2, 1, 0]
[ ,  ,  , 0, 1, 2, 3, 2, 1, 0]
[ ,  ,  ,  , 0, 1, 2, 1, 0, 1, 0]
[ ,  ,  ,  ,  , 0, 1, 0, 1, 2, 1, 0]
[ ,  ,  ,  ,  ,  , 0, 1, 2, 3, 2, 1, 0]
```

pp()

A pretty print utility method.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: path_tableaux.CylindricalDiagram(t).pp()
0 1 2 3 2 1 0
  0 1 2 1 0 1 0
```

(continues on next page)

(continued from previous page)

```
    0 1 0 1 2 1 0
      0 1 2 3 2 1 0
        0 1 2 1 0 1 0
          0 1 0 1 2 1 0
            0 1 2 3 2 1 0

sage: t = path_tableaux.FriezePattern([1,3,4,5,1])
sage: path_tableaux.CylindricalDiagram(t).pp()
  0   1   3   4   5   1   0
      0   1 5/3 7/3 2/3   1   0
          0   1   2   1   3   1   0
              0   1   1   4 5/3   1   0
                  0   1   5 7/3   2   1   0
                      0   1 2/3   1   1   1   0
                          0   1   3   4   5   1   0
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> path_tableaux.CylindricalDiagram(t).pp()
0 1 2 3 2 1 0
  0 1 2 1 0 1 0
    0 1 0 1 2 1 0
      0 1 2 3 2 1 0
        0 1 2 1 0 1 0
          0 1 0 1 2 1 0
            0 1 2 3 2 1 0

>>> t = path_tableaux.FriezePattern([Integer(1),Integer(3),Integer(4),
↪Integer(5),Integer(1)])
>>> path_tableaux.CylindricalDiagram(t).pp()
  0   1   3   4   5   1   0
      0   1 5/3 7/3 2/3   1   0
          0   1   2   1   3   1   0
              0   1   1   4 5/3   1   0
                  0   1   5 7/3   2   1   0
                      0   1 2/3   1   1   1   0
                          0   1   3   4   5   1   0
```

class sage.combinat.path_tableaux.path_tableau.**PathTableau**

Bases: ClonableArray

This is the abstract base class for a path tableau.

cactus(*i*, *j*)

Return the action of the generator $s_{i,j}$ of the cactus group on self.

INPUT:

- i – positive integer
- j – positive integer weakly greater than i

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.cactus(1,5)
[0, 1, 0, 1, 2, 1, 0]

sage: t.cactus(1,6)
[0, 1, 2, 1, 0, 1, 0]

sage: t.cactus(1,7) == t.evacuation()
True
sage: t.cactus(1,7).cactus(1,6) == t.promotion()
True
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.cactus(Integer(1),Integer(5))
[0, 1, 0, 1, 2, 1, 0]

>>> t.cactus(Integer(1),Integer(6))
[0, 1, 2, 1, 0, 1, 0]

>>> t.cactus(Integer(1),Integer(7)) == t.evacuation()
True
>>> t.cactus(Integer(1),Integer(7)).cactus(Integer(1),Integer(6)) == t.
↪promotion()
True
```

commutor(*other*, *verbose=False*)

Return the commutor of `self` with `other`.

If `verbose=True` then the function will print the rectangle.

EXAMPLES:

```
sage: t1 = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t2 = path_tableaux.DyckPath([0,1,2,1,0])
sage: t1.commutor(t2)
([0, 1, 2, 1, 0], [0, 1, 2, 3, 2, 1, 0])
sage: t1.commutor(t2,verbose=True)
[0, 1, 2, 1, 0]
[1, 2, 3, 2, 1]
[2, 3, 4, 3, 2]
[3, 4, 5, 4, 3]
[2, 3, 4, 3, 2]
[1, 2, 3, 2, 1]
[0, 1, 2, 1, 0]
([0, 1, 2, 1, 0], [0, 1, 2, 3, 2, 1, 0])
```

```
>>> from sage.all import *
>>> t1 = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t2 = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> t1.commutor(t2)
([0, 1, 2, 1, 0], [0, 1, 2, 3, 2, 1, 0])
>>> t1.commutor(t2,verbose=True)
[0, 1, 2, 1, 0]
[1, 2, 3, 2, 1]
[2, 3, 4, 3, 2]
[3, 4, 5, 4, 3]
[2, 3, 4, 3, 2]
[1, 2, 3, 2, 1]
[0, 1, 2, 1, 0]
([0, 1, 2, 1, 0], [0, 1, 2, 3, 2, 1, 0])
```

dual_equivalence_graph()

Return the graph with vertices the orbit of `self` and edges given by the action of the cactus group generators.

In most implementations the generators $s_{i,i+1}$ will act as the identity operators. The usual dual equivalence graphs are given by replacing the label $i, i+2$ by i and removing edges with other labels.

EXAMPLES:

```
sage: s = path_tableaux.DyckPath([0,1,2,3,2,3,2,1,0])
sage: s.dual_equivalence_graph().adjacency_matrix()                          #␣
↪needs sage.graphs sage.modules
[0 1 1 1 0 1 0 1 1 0 0 0 0 0]
[1 0 1 1 1 1 1 0 1 0 0 1 1 0]
[1 1 0 1 1 1 0 1 0 1 1 1 0 0]
[1 1 1 0 1 0 1 1 1 1 0 1 1 0]
[0 1 1 1 0 0 1 0 0 1 1 0 1 1]
[1 1 1 0 0 0 1 1 1 1 1 0 1 0]
[0 1 0 1 1 1 0 1 0 1 1 1 0 1]
[1 0 1 1 0 1 1 0 1 1 1 1 1 0]
[1 1 0 1 0 1 0 1 0 1 0 1 1 0]
[0 0 1 1 1 1 1 1 1 0 0 1 1 1]
[0 0 1 0 1 1 1 1 0 0 0 1 1 1]
[0 1 1 1 0 0 1 1 1 1 1 0 1 1]
[0 1 0 1 1 1 0 1 1 1 1 1 0 1]
[0 0 0 0 1 0 1 0 0 1 1 1 1 0]
sage: s = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: s.dual_equivalence_graph().edges(sort=True)                            #␣
↪needs sage.graphs
[([0, 1, 0, 1, 0, 1, 0], [0, 1, 0, 1, 2, 1, 0], '4,7'),
 ([0, 1, 0, 1, 0, 1, 0], [0, 1, 2, 1, 0, 1, 0], '2,5'),
 ([0, 1, 0, 1, 0, 1, 0], [0, 1, 2, 1, 2, 1, 0], '2,7'),
 ([0, 1, 0, 1, 2, 1, 0], [0, 1, 2, 1, 0, 1, 0], '2,6'),
 ([0, 1, 0, 1, 2, 1, 0], [0, 1, 2, 1, 2, 1, 0], '1,4'),
 ([0, 1, 0, 1, 2, 1, 0], [0, 1, 2, 3, 2, 1, 0], '2,7'),
 ([0, 1, 2, 1, 0, 1, 0], [0, 1, 2, 1, 2, 1, 0], '4,7'),
 ([0, 1, 2, 1, 0, 1, 0], [0, 1, 2, 3, 2, 1, 0], '3,7'),
 ([0, 1, 2, 1, 2, 1, 0], [0, 1, 2, 3, 2, 1, 0], '3,6')]
```

```
>>> from sage.all import *
>>> s = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(3),Integer(2),Integer(1),Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> s.dual_equivalence_graph().adjacency_matrix()                             #␣
↪needs sage.graphs sage.modules
[0 1 1 1 0 1 0 1 1 0 0 0 0 0]
[1 0 1 1 1 1 1 0 1 0 0 1 1 0]
[1 1 0 1 1 1 0 1 0 1 1 1 0 0]
[1 1 1 0 1 0 1 1 1 1 0 1 1 0]
[0 1 1 1 0 0 1 0 0 1 1 0 1 1]
[1 1 1 0 0 0 1 1 1 1 1 0 1 0]
[0 1 0 1 1 1 0 1 0 1 1 1 0 1]
[1 0 1 1 0 1 1 0 1 1 1 1 1 0]
[1 1 0 1 0 1 0 1 0 1 0 1 1 0]
[0 0 1 1 1 1 1 1 1 0 0 1 1 1]
[0 0 1 0 1 1 1 1 0 0 0 1 1 1]
[0 1 1 1 0 0 1 1 1 1 1 0 1 1]
[0 1 0 1 1 1 0 1 1 1 1 1 0 1]
[0 0 0 0 1 0 1 0 0 1 1 1 1 0]
>>> s = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> s.dual_equivalence_graph().edges(sort=True)                               #␣
↪needs sage.graphs
[([0, 1, 0, 1, 0, 1, 0], [0, 1, 0, 1, 2, 1, 0], '4,7'),
 ([0, 1, 0, 1, 0, 1, 0], [0, 1, 2, 1, 0, 1, 0], '2,5'),
 ([0, 1, 0, 1, 0, 1, 0], [0, 1, 2, 1, 2, 1, 0], '2,7'),
 ([0, 1, 0, 1, 2, 1, 0], [0, 1, 2, 1, 0, 1, 0], '2,6'),
 ([0, 1, 0, 1, 2, 1, 0], [0, 1, 2, 1, 2, 1, 0], '1,4'),
 ([0, 1, 0, 1, 2, 1, 0], [0, 1, 2, 3, 2, 1, 0], '2,7'),
 ([0, 1, 2, 1, 0, 1, 0], [0, 1, 2, 1, 2, 1, 0], '4,7'),
 ([0, 1, 2, 1, 0, 1, 0], [0, 1, 2, 3, 2, 1, 0], '3,7'),
 ([0, 1, 2, 1, 2, 1, 0], [0, 1, 2, 3, 2, 1, 0], '3,6')]
```

evacuation()

Return the evacuation operator applied to `self`.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.evacuation()
[0, 1, 2, 3, 2, 1, 0]
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.evacuation()
[0, 1, 2, 3, 2, 1, 0]
```

final_shape()

Return the final shape of `self`.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.final_shape()
0
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.final_shape()
0
```

initial_shape()

Return the initial shape of `self`.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.initial_shape()
0
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.initial_shape()
0
```

local_rule(*i*)

This is the abstract local rule defined in any coboundary category.

This has input a list of objects. This method first takes the list of objects of length three consisting of the $(i-1)$-st, i-th and $(i+1)$-term and applies the rule. It then replaces the i-th object by the object returned by the rule.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.local_rule(3)
[0, 1, 2, 1, 2, 1, 0]
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.local_rule(Integer(3))
[0, 1, 2, 1, 2, 1, 0]
```

orbit()

Return the orbit of `self` under the action of the cactus group.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.orbit()
{[0, 1, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 2, 1, 0],
 [0, 1, 2, 1, 0, 1, 0],
 [0, 1, 2, 1, 2, 1, 0],
 [0, 1, 2, 3, 2, 1, 0]}
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.orbit()
{[0, 1, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 2, 1, 0],
 [0, 1, 2, 1, 0, 1, 0],
 [0, 1, 2, 1, 2, 1, 0],
 [0, 1, 2, 3, 2, 1, 0]}
```

promotion()

Return the promotion operator applied to `self`.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.promotion()
[0, 1, 2, 1, 0, 1, 0]
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.promotion()
[0, 1, 2, 1, 0, 1, 0]
```

size()

Return the size or length of `self`.

EXAMPLES:

```
sage: t = path_tableaux.DyckPath([0,1,2,3,2,1,0])
sage: t.size()
7
```

```
>>> from sage.all import *
>>> t = path_tableaux.DyckPath([Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(1),Integer(0)])
>>> t.size()
7
```

class sage.combinat.path_tableaux.path_tableau.**PathTableaux**

Bases: `UniqueRepresentation`, `Parent`

The abstract parent class for PathTableau.

5.1.163 Semistandard Tableaux

This is an implementation of the abstract base class *sage.combinat.path_tableaux.path_tableau.PathTableau*.

This implementation is for semistandard tableaux, represented as a chain of partitions (essentially, the Gelfand-Tsetlin pattern). This generalises the jeu de taquin operations of rectification, promotion, evacuation from standard tableaux to semistandard tableaux. The local rule is the Bender-Knuth involution.

EXAMPLES:

```
sage: pt = path_tableaux.SemistandardPathTableau([[], [3], [3,2], [3,3,1],
....:                                            [3,3,2,1], [4,3,3,1,0]])
sage: pt.promotion()
[(), (2,), (3, 1), (3, 2, 1), (4, 3, 1, 0), (4, 3, 3, 1, 0)]
sage: pt.evacuation()
[(), (2,), (4, 0), (4, 2, 0), (4, 3, 1, 0), (4, 3, 3, 1, 0)]

sage: pt = path_tableaux.SemistandardPathTableau([[], [3], [3,2], [3,3,1],
....:                                            [3,3,2,1], [9/2,3,3,1,0]])
sage: pt.promotion()
[(), (2,), (3, 1), (3, 2, 1), (9/2, 3, 1, 0), (9/2, 3, 3, 1, 0)]
sage: pt.evacuation()
[(), (5/2,), (9/2, 0), (9/2, 2, 0), (9/2, 3, 1, 0), (9/2, 3, 3, 1, 0)]

sage: pt = path_tableaux.SemistandardPathTableau([[], [3], [4,2], [5,4,1]])
sage: path_tableaux.CylindricalDiagram(pt)
[       (),      (3,),    (4, 2), (5, 4, 1)]
[         ,        (),      (3,),    (5, 2), (5, 4, 1)]
[         ,          ,        (),      (4,),    (4, 3), (5, 4, 1)]
[         ,          ,          ,        (),      (3,),    (5, 1), (5, 4, 1)]

sage: pt2 = path_tableaux.SemistandardPathTableau([[3,2], [3,3,1],
....:                                             [3,3,2,1], [4,3,3,1,0]])
sage: pt1 = path_tableaux.SemistandardPathTableau([[], [3], [3,2]])
sage: pt1.commutor(pt2)
([(), (2,), (2, 2), (4, 2, 0)], [(4, 2, 0), (4, 3, 2, 0), (4, 3, 3, 1, 0)])
sage: pt1.commutor(pt2,verbose=True)
[(3, 2), (3, 3, 1), (3, 3, 2, 1), (4, 3, 3, 1, 0)]
[(3,), (3, 2), (3, 2, 2), (4, 3, 2, 0)]
[(), (2,), (2, 2), (4, 2, 0)]
([(), (2,), (2, 2), (4, 2, 0)], [(4, 2, 0), (4, 3, 2, 0), (4, 3, 3, 1, 0)])

sage: st = SkewTableau([[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7],
....:                   [None, 4, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11],
....:                   [None], [4]])
sage: pt = path_tableaux.SemistandardPathTableau(st)
sage: bk = [SkewTableau(st.bender_knuth_involution(i+1)) for i in range(10)]
sage: lr = [pt.local_rule(i+1) for i in range(10)]
sage: [r.to_tableau() for r in lr] == bk
True
```

```
>>> from sage.all import *
>>> pt = path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),
↪Integer(2)], [Integer(3),Integer(3),Integer(1)],
...                                            [Integer(3),Integer(3),Integer(2),
↪Integer(1)], [Integer(4),Integer(3),Integer(3),Integer(1),Integer(0)]])
>>> pt.promotion()
[(), (2,), (3, 1), (3, 2, 1), (4, 3, 1, 0), (4, 3, 3, 1, 0)]
>>> pt.evacuation()
[(), (2,), (4, 0), (4, 2, 0), (4, 3, 1, 0), (4, 3, 3, 1, 0)]
```

(continues on next page)

(continued from previous page)

```
>>> pt = path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),
↪Integer(2)], [Integer(3),Integer(3),Integer(1)],
...                                             [Integer(3),Integer(3),Integer(2),
↪Integer(1)], [Integer(9)/Integer(2),Integer(3),Integer(3),Integer(1),Integer(0)]])
>>> pt.promotion()
[(), (2,), (3, 1), (3, 2, 1), (9/2, 3, 1, 0), (9/2, 3, 3, 1, 0)]
>>> pt.evacuation()
[(), (5/2,), (9/2, 0), (9/2, 2, 0), (9/2, 3, 1, 0), (9/2, 3, 3, 1, 0)]

>>> pt = path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(4),
↪Integer(2)], [Integer(5),Integer(4),Integer(1)]])
>>> path_tableaux.CylindricalDiagram(pt)
[       (),      (3,),     (4, 2), (5, 4, 1)]
[         ,         (),      (3,),     (5, 2), (5, 4, 1)]
[         ,           ,         (),      (4,),     (4, 3), (5, 4, 1)]
[         ,           ,           ,         (),      (3,),     (5, 1), (5, 4, 1)]

>>> pt2 = path_tableaux.SemistandardPathTableau([[Integer(3),Integer(2)], [Integer(3),
↪Integer(3),Integer(1)],
...                                              [Integer(3),Integer(3),Integer(2),
↪Integer(1)], [Integer(4),Integer(3),Integer(3),Integer(1),Integer(0)]])
>>> pt1 = path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),
↪Integer(2)]])
>>> pt1.commutor(pt2)
([(), (2,), (2, 2), (4, 2, 0)], [(4, 2, 0), (4, 3, 2, 0), (4, 3, 3, 1, 0)])
>>> pt1.commutor(pt2,verbose=True)
[(3, 2), (3, 3, 1), (3, 3, 2, 1), (4, 3, 3, 1, 0)]
[(3,), (3, 2), (3, 2, 2), (4, 3, 2, 0)]
[(), (2,), (2, 2), (4, 2, 0)]
([(), (2,), (2, 2), (4, 2, 0)], [(4, 2, 0), (4, 3, 2, 0), (4, 3, 3, 1, 0)])

>>> st = SkewTableau([[None, None, None, Integer(4), Integer(4), Integer(5),␣
↪Integer(6), Integer(7)], [None, Integer(2), Integer(4), Integer(6), Integer(7),␣
↪Integer(7), Integer(7)],
...                   [None, Integer(4), Integer(5), Integer(8), Integer(8),␣
↪Integer(9)], [None, Integer(6), Integer(7), Integer(10)], [None, Integer(8),␣
↪Integer(8), Integer(11)],
...                   [None], [Integer(4)]])
>>> pt = path_tableaux.SemistandardPathTableau(st)
>>> bk = [SkewTableau(st.bender_knuth_involution(i+Integer(1))) for i in␣
↪range(Integer(10))]
>>> lr = [pt.local_rule(i+Integer(1)) for i in range(Integer(10))]
>>> [r.to_tableau() for r in lr] == bk
True
```

AUTHORS:

- Bruce Westbury (2020): initial version

class sage.combinat.path_tableaux.semistandard.**SemistandardPathTableau**(*parent*, *st*, *check=True*)

Bases: *PathTableau*

An instance is a sequence of lists. Usually the entries will be nonnegative integers in which case this is the chain of partitions of a (skew) semistandard tableau. In general the entries are elements of an ordered abelian group; each list is weakly decreasing and successive lists are interleaved.

INPUT:

Can be any of the following

- a sequence of partitions
- a sequence of lists/tuples
- a semistandard tableau
- a semistandard skew tableau
- a Gelfand-Tsetlin pattern

EXAMPLES:

```
sage: path_tableaux.SemistandardPathTableau([[], [2], [2,1]])
[(), (2,), (2, 1)]

sage: gt = GelfandTsetlinPattern([[2,1], [2]])
sage: path_tableaux.SemistandardPathTableau(gt)
[(), (2,), (2, 1)]

sage: st = SemistandardTableau([[1,1], [2]])
sage: path_tableaux.SemistandardPathTableau(st)
[(), (2,), (2, 1)]

sage: st = SkewTableau([[1,1], [2]])
sage: path_tableaux.SemistandardPathTableau(st)
[(), (2,), (2, 1)]

sage: st = SkewTableau([[None,1,1], [2]])
sage: path_tableaux.SemistandardPathTableau(st)
[(1,), (3, 0), (3, 1, 0)]

sage: path_tableaux.SemistandardPathTableau([[], [5/2], [7/2,2]])
[(), (5/2,), (7/2, 2)]

sage: path_tableaux.SemistandardPathTableau([[], [2.5], [3.5,2]])
[(), (2.50000000000000,), (3.50000000000000, 2)]
```

```
>>> from sage.all import *
>>> path_tableaux.SemistandardPathTableau([[], [Integer(2)], [Integer(2),
↪Integer(1)]])
[(), (2,), (2, 1)]

>>> gt = GelfandTsetlinPattern([[Integer(2),Integer(1)], [Integer(2)]])
>>> path_tableaux.SemistandardPathTableau(gt)
[(), (2,), (2, 1)]

>>> st = SemistandardTableau([[Integer(1),Integer(1)], [Integer(2)]])
>>> path_tableaux.SemistandardPathTableau(st)
[(), (2,), (2, 1)]
```

(continues on next page)

(continued from previous page)

```
>>> st = SkewTableau([[Integer(1),Integer(1)], [Integer(2)]])
>>> path_tableaux.SemistandardPathTableau(st)
[(), (2,), (2, 1)]

>>> st = SkewTableau([[None,Integer(1),Integer(1)], [Integer(2)]])
>>> path_tableaux.SemistandardPathTableau(st)
[(1,), (3, 0), (3, 1, 0)]

>>> path_tableaux.SemistandardPathTableau([[], [Integer(5)/Integer(2)],␣
↪[Integer(7)/Integer(2),Integer(2)]])
[(), (5/2,), (7/2, 2)]

>>> path_tableaux.SemistandardPathTableau([[], [RealNumber('2.5')], [RealNumber(
↪'3.5'),Integer(2)]])
[(), (2.50000000000000,), (3.50000000000000, 2)]
```

check()

Check that `self` is a valid path.

EXAMPLES:

```
sage: path_tableaux.SemistandardPathTableau([[], [3], [2,2]])        #␣
↪indirect doctest
Traceback (most recent call last):
...
ValueError: [(), (3,), (2, 2)] does not satisfy
the required inequalities in row 1

sage: path_tableaux.SemistandardPathTableau([[], [3/2], [2,5/2]])    #␣
↪indirect doctest
Traceback (most recent call last):
...
ValueError: [(), (3/2,), (2, 5/2)] does not satisfy
the required inequalities in row 1
```

```
>>> from sage.all import *
>>> path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(2),
↪Integer(2)]])        # indirect doctest
Traceback (most recent call last):
...
ValueError: [(), (3,), (2, 2)] does not satisfy
the required inequalities in row 1

>>> path_tableaux.SemistandardPathTableau([[], [Integer(3)/Integer(2)],␣
↪[Integer(2),Integer(5)/Integer(2)]])    # indirect doctest
Traceback (most recent call last):
...
ValueError: [(), (3/2,), (2, 5/2)] does not satisfy
the required inequalities in row 1
```

is_integral()

Return `True` if all entries are nonnegative integers.

EXAMPLES:

```
sage: path_tableaux.SemistandardPathTableau([[], [3], [3,2]]).is_integral()
True
sage: path_tableaux.SemistandardPathTableau([[], [5/2], [7/2,2]]).is_
→integral()
False
sage: path_tableaux.SemistandardPathTableau([[], [3], [3,-2]]).is_integral()
False
```

```
>>> from sage.all import *
>>> path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),
→Integer(2)]]).is_integral()
True
>>> path_tableaux.SemistandardPathTableau([[], [Integer(5)/Integer(2)],␣
→[Integer(7)/Integer(2),Integer(2)]]).is_integral()
False
>>> path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),-
→Integer(2)]]).is_integral()
False
```

is_skew()

Return `True` if `self` is skew.

EXAMPLES:

```
sage: path_tableaux.SemistandardPathTableau([[], [2]]).is_skew()
False
sage: path_tableaux.SemistandardPathTableau([[2,1]]).is_skew()
True
```

```
>>> from sage.all import *
>>> path_tableaux.SemistandardPathTableau([[], [Integer(2)]]).is_skew()
False
>>> path_tableaux.SemistandardPathTableau([[Integer(2),Integer(1)]]).is_skew()
True
```

local_rule(*i*)

This is the Bender-Knuth involution.

This is implemented by toggling the entries of the i-th list. The allowed range for `i` is `0 < i < len(self)-1` so any row except the first and last can be changed.

EXAMPLES:

```
sage: pt = path_tableaux.SemistandardPathTableau([[], [3], [3,2],
....:                                              [3,3,1], [3,3,2,1]])
sage: pt.local_rule(1)
[(), (2,), (3, 2), (3, 3, 1), (3, 3, 2, 1)]
sage: pt.local_rule(2)
[(), (3,), (3, 2), (3, 3, 1), (3, 3, 2, 1)]
sage: pt.local_rule(3)
[(), (3,), (3, 2), (3, 2, 2), (3, 3, 2, 1)]
```

```
>>> from sage.all import *
>>> pt = path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),
↪Integer(2)],
...                                             [Integer(3),Integer(3),
↪Integer(1)], [Integer(3),Integer(3),Integer(2),Integer(1)]])
>>> pt.local_rule(Integer(1))
[(), (2,), (3, 2), (3, 3, 1), (3, 3, 2, 1)]
>>> pt.local_rule(Integer(2))
[(), (3,), (3, 2), (3, 3, 1), (3, 3, 2, 1)]
>>> pt.local_rule(Integer(3))
[(), (3,), (3, 2), (3, 2, 2), (3, 3, 2, 1)]
```

rectify(*inner=None*, *verbose=False*)

Rectify `self`.

This gives the usual rectification of a skew standard tableau and gives a generalisation to skew semistandard tableaux. The usual construction uses jeu de taquin but here we use the Bender-Knuth involutions.

EXAMPLES:

```
sage: st = SkewTableau([[None, None, None, 4], [None, None, 1, 6],
....:                    [None, None, 5], [2, 3]])
sage: path_tableaux.SemistandardPathTableau(st).rectify()
[(), (1,), (1, 1), (2, 1, 0), (3, 1, 0, 0), (3, 2, 0, 0, 0), (4, 2, 0, 0, 0,␣
↪0)]
sage: path_tableaux.SemistandardPathTableau(st).rectify(verbose=True)
[[(3, 2, 2), (3, 3, 2, 0), (3, 3, 2, 1, 0), (3, 3, 2, 2, 0, 0),
  (4, 3, 2, 2, 0, 0, 0), (4, 3, 3, 2, 0, 0, 0, 0), (4, 4, 3, 2, 0, 0, 0, 0,␣
↪0)],
 [(3, 2), (3, 3, 0), (3, 3, 1, 0), (3, 3, 2, 0, 0), (4, 3, 2, 0, 0, 0),
  (4, 3, 3, 0, 0, 0, 0), (4, 4, 3, 0, 0, 0, 0, 0)],
 [(3,), (3, 1), (3, 1, 1), (3, 2, 1, 0), (4, 2, 1, 0, 0), (4, 3, 1, 0, 0, 0),
  (4, 4, 1, 0, 0, 0, 0)],
 [(), (1,), (1, 1), (2, 1, 0), (3, 1, 0, 0), (3, 2, 0, 0, 0), (4, 2, 0, 0, 0,␣
↪0)]]
```

```
>>> from sage.all import *
>>> st = SkewTableau([[None, None, None, Integer(4)], [None, None, Integer(1),
↪ Integer(6)],
...                   [None, None, Integer(5)], [Integer(2), Integer(3)]])
>>> path_tableaux.SemistandardPathTableau(st).rectify()
[(), (1,), (1, 1), (2, 1, 0), (3, 1, 0, 0), (3, 2, 0, 0, 0), (4, 2, 0, 0, 0,␣
↪0)]
>>> path_tableaux.SemistandardPathTableau(st).rectify(verbose=True)
[[(3, 2, 2), (3, 3, 2, 0), (3, 3, 2, 1, 0), (3, 3, 2, 2, 0, 0),
  (4, 3, 2, 2, 0, 0, 0), (4, 3, 3, 2, 0, 0, 0, 0), (4, 4, 3, 2, 0, 0, 0, 0,␣
↪0)],
 [(3, 2), (3, 3, 0), (3, 3, 1, 0), (3, 3, 2, 0, 0), (4, 3, 2, 0, 0, 0),
  (4, 3, 3, 0, 0, 0, 0), (4, 4, 3, 0, 0, 0, 0, 0)],
 [(3,), (3, 1), (3, 1, 1), (3, 2, 1, 0), (4, 2, 1, 0, 0), (4, 3, 1, 0, 0, 0),
  (4, 4, 1, 0, 0, 0, 0)],
 [(), (1,), (1, 1), (2, 1, 0), (3, 1, 0, 0), (3, 2, 0, 0, 0), (4, 2, 0, 0, 0,␣
↪0)]]
```

size()

Return the size or length of `self`.

EXAMPLES:

```
sage: path_tableaux.SemistandardPathTableau([[], [3], [3,2], [3,3,1], [3,3,2,
↪1]]).size()
5
```

```
>>> from sage.all import *
>>> path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),
↪Integer(2)], [Integer(3),Integer(3),Integer(1)], [Integer(3),Integer(3),
↪Integer(2),Integer(1)]]).size()
5
```

to_pattern()

Convert `self` to a Gelfand-Tsetlin pattern.

EXAMPLES:

```
sage: pt = path_tableaux.SemistandardPathTableau([[], [3], [3,2], [3,3,1],
....:                                              [3,3,2,1], [4,3,3,1]])
sage: pt.to_pattern()
[[4, 3, 3, 1, 0], [3, 3, 2, 1], [3, 3, 1], [3, 2], [3]]
```

```
>>> from sage.all import *
>>> pt = path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),
↪Integer(2)], [Integer(3),Integer(3),Integer(1)],
...                                              [Integer(3),Integer(3),
↪Integer(2),Integer(1)], [Integer(4),Integer(3),Integer(3),Integer(1)]])
>>> pt.to_pattern()
[[4, 3, 3, 1, 0], [3, 3, 2, 1], [3, 3, 1], [3, 2], [3]]
```

to_tableau()

Convert `self` to a *SemistandardTableau*.

The `SemistandardSkewTableau` is not implemented so this returns a *SkewTableau*

EXAMPLES:

```
sage: pt = path_tableaux.SemistandardPathTableau([[], [3], [3,2], [3,3,1],
....:                                              [3,3,2,1], [4,3,3,1,0]])
sage: pt.to_tableau()
[[1, 1, 1, 5], [2, 2, 3], [3, 4, 5], [4]]
```

```
>>> from sage.all import *
>>> pt = path_tableaux.SemistandardPathTableau([[], [Integer(3)], [Integer(3),
↪Integer(2)], [Integer(3),Integer(3),Integer(1)],
...                                              [Integer(3),Integer(3),
↪Integer(2),Integer(1)], [Integer(4),Integer(3),Integer(3),Integer(1),
↪Integer(0)]])
>>> pt.to_tableau()
[[1, 1, 1, 5], [2, 2, 3], [3, 4, 5], [4]]
```

class sage.combinat.path_tableaux.semistandard.**SemistandardPathTableaux**

Bases: *PathTableaux*

The parent class for *SemistandardPathTableau*.

Element

alias of *SemistandardPathTableau*

5.1.164 Plane Partitions

AUTHORS:

- Jang Soo Kim (2016): Initial implementation
- Jessica Striker (2016): Added additional methods
- Kevin Dilks (2021): Added symmetry classes

class sage.combinat.plane_partition.**PlanePartition**(*parent*, *pp*, *check=True*)

Bases: ClonableArray

A plane partition.

A *plane partition* is a stack of cubes in the positive orthant.

INPUT:

- PP – list of lists which represents a tableau
- box_size – (optional) a list [A, B, C] of 3 positive integers, where A, B, C are the lengths of the box in the x-axis, y-axis, z-axis, respectively; if this is not given, it is determined by the smallest box bounding PP

OUTPUT: the plane partition whose tableau representation is PP

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP
Plane partition [[4, 3, 3, 1], [2, 1, 1], [1, 1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP
Plane partition [[4, 3, 3, 1], [2, 1, 1], [1, 1]]
```

bounding_box()

Return the smallest box (a, b, c) that self is contained in.

EXAMPLES:

```
sage: PP = PlanePartition([[5,2,1,1], [2,2], [2]])
sage: PP.bounding_box()
(3, 4, 5)
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(5),Integer(2),Integer(1),Integer(1)],␣
↪[Integer(2),Integer(2)], [Integer(2)]])
>>> PP.bounding_box()
(3, 4, 5)
```

cells()

Return the list of cells inside `self`.

Each cell is a tuple.

EXAMPLES:

```
sage: PP = PlanePartition([[3,1],[2]])
sage: PP.cells()
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (1, 0, 0), (1, 0, 1)]
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(3),Integer(1)],[Integer(2)]])
>>> PP.cells()
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (1, 0, 0), (1, 0, 1)]
```

check()

Check to see that `self` is a valid plane partition.

EXAMPLES:

```
sage: a = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: a.check()
sage: b = PlanePartition([[1,2],[1]])
Traceback (most recent call last):
...
ValueError: not weakly decreasing along rows
sage: c = PlanePartition([[1,1],[2]])
Traceback (most recent call last):
...
ValueError: not weakly decreasing along columns
sage: d = PlanePartition([[2,-1],[-2]])
Traceback (most recent call last):
...
ValueError: entries not all nonnegative
sage: e = PlanePartition([[3/2,1],[.5]])
Traceback (most recent call last):
...
ValueError: entries not all integers
```

```
>>> from sage.all import *
>>> a = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> a.check()
>>> b = PlanePartition([[Integer(1),Integer(2)],[Integer(1)]])
Traceback (most recent call last):
...
ValueError: not weakly decreasing along rows
>>> c = PlanePartition([[Integer(1),Integer(1)],[Integer(2)]])
Traceback (most recent call last):
...
ValueError: not weakly decreasing along columns
>>> d = PlanePartition([[Integer(2),-Integer(1)],[-Integer(2)]])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: entries not all nonnegative
>>> e = PlanePartition([[Integer(3)/Integer(2),Integer(1)],[RealNumber('.5
↪')]])
Traceback (most recent call last):
...
ValueError: entries not all integers
```

complement(*tableau_only=False*)

Return the complement of `self`.

If the parent of `self` consists only of partitions inside a given box, then the complement is taken in this box. Otherwise, the complement is taken in the smallest box containing the plane partition. The empty plane partition with no box specified is its own complement.

If `tableau_only` is set to `True`, then only the tableau consisting of the projection of boxes size onto the xy-plane is returned instead of a `PlanePartition`. This output will not have empty trailing rows or trailing zeros removed.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.complement()
Plane partition [[4, 4, 3, 3], [4, 3, 3, 2], [3, 1, 1]]
sage: PP.complement(True)
[[4, 4, 3, 3], [4, 3, 3, 2], [3, 1, 1, 0]]
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.complement()
Plane partition [[4, 4, 3, 3], [4, 3, 3, 2], [3, 1, 1]]
>>> PP.complement(True)
[[4, 4, 3, 3], [4, 3, 3, 2], [3, 1, 1, 0]]
```

contains(*PP*)

Return `True` if `PP` is a plane partition that fits inside `self`.

Specifically, `self` contains `PP` if, for all i, j, the height of `PP` at ij is less than or equal to the height of `self` at ij.

EXAMPLES:

```
sage: P1 = PlanePartition([[5,4,3], [3,2,2], [1]])
sage: P2 = PlanePartition([[3,2], [1,1], [0,0], [0,0]])
sage: P3 = PlanePartition([[5,5,5], [2,1,0]])
sage: P1.contains(P2)
True
sage: P2.contains(P1)
False
sage: P1.contains(P3)
False
sage: P3.contains(P2)
True
```

```
>>> from sage.all import *
>>> P1 = PlanePartition([[Integer(5),Integer(4),Integer(3)], [Integer(3),
↪Integer(2),Integer(2)], [Integer(1)]])
>>> P2 = PlanePartition([[Integer(3),Integer(2)], [Integer(1),Integer(1)],␣
↪[Integer(0),Integer(0)], [Integer(0),Integer(0)]])
>>> P3 = PlanePartition([[Integer(5),Integer(5),Integer(5)], [Integer(2),
↪Integer(1),Integer(0)]])
>>> P1.contains(P2)
True
>>> P2.contains(P1)
False
>>> P1.contains(P3)
False
>>> P3.contains(P2)
True
```

cyclically_rotate(*preserve_parent=False*)

Return the cyclic rotation of `self`.

By default, if the parent of `self` consists of plane partitions inside an $a \times b \times c$ box, the result will have a parent consisting of partitions inside a $c \times a \times b$ box, unless the optional parameter `preserve_parent` is set to `True`. Enabling this setting may give an element that is **not** an element of its parent.

EXAMPLES:

```
sage: PlanePartition([[3,2,1],[2,2],[2]]).cyclically_rotate()
Plane partition [[3, 3, 1], [2, 2], [1]]
sage: PP = PlanePartition([[4,1],[1],[1]])
sage: PP.cyclically_rotate()
Plane partition [[3, 1, 1, 1], [1]]
sage: PP == PP.cyclically_rotate().cyclically_rotate().cyclically_rotate()
True

sage: # needs sage.graphs sage.modules
sage: PP = PlanePartitions([4,3,2]).random_element()
sage: PP.cyclically_rotate().parent()
Plane partitions inside a 2 x 4 x 3 box
sage: PP = PlanePartitions([3,4,2])([[2,2,2,2],[2,2,2,2],[2,2,2,2]])
sage: PP_rotated = PP.cyclically_rotate(preserve_parent=True)
sage: PP_rotated in PP_rotated.parent()
False
```

```
>>> from sage.all import *
>>> PlanePartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(2)]]).cyclically_rotate()
Plane partition [[3, 3, 1], [2, 2], [1]]
>>> PP = PlanePartition([[Integer(4),Integer(1)],[Integer(1)],[Integer(1)]])
>>> PP.cyclically_rotate()
Plane partition [[3, 1, 1, 1], [1]]
>>> PP == PP.cyclically_rotate().cyclically_rotate().cyclically_rotate()
True

>>> # needs sage.graphs sage.modules
```

(continues on next page)

(continued from previous page)

```
>>> PP = PlanePartitions([Integer(4),Integer(3),Integer(2)]).random_element()
>>> PP.cyclically_rotate().parent()
Plane partitions inside a 2 x 4 x 3 box
>>> PP = PlanePartitions([Integer(3),Integer(4),Integer(2)])([[Integer(2),
↪Integer(2),Integer(2),Integer(2)],[Integer(2),Integer(2),Integer(2),
↪Integer(2)],[Integer(2),Integer(2),Integer(2),Integer(2)]])
>>> PP_rotated = PP.cyclically_rotate(preserve_parent=True)
>>> PP_rotated in PP_rotated.parent()
False
```

is_CSPP()

Return whether `self` is a cyclically symmetric plane partition.

A plane partition is cyclically symmetric if its x, y, and z tableaux are all equal.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_CSPP()
False
sage: PP = PlanePartition([[3,2,2],[3,1,0],[1,1,0]])
sage: PP.is_CSPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_CSPP()
False
>>> PP = PlanePartition([[Integer(3),Integer(2),Integer(2)],[Integer(3),
↪Integer(1),Integer(0)],[Integer(1),Integer(1),Integer(0)]])
>>> PP.is_CSPP()
True
```

is_CSSCPP()

Return whether `self` is a cyclically symmetric and self-complementary plane partition.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_CSSCPP()
False
sage: PP = PlanePartition([[4,4,4,1],[3,3,2,1],[3,2,1,1],[3,0,0,0]])
sage: PP.is_CSSCPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_CSSCPP()
False
>>> PP = PlanePartition([[Integer(4),Integer(4),Integer(4),Integer(1)],
↪[Integer(3),Integer(3),Integer(2),Integer(1)],[Integer(3),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(1)],[Integer(3),Integer(0),Integer(0),Integer(0)]])
>>> PP.is_CSSCPP()
True
```

is_CSTCPP()

Return whether self is a cyclically symmetric and transpose-complementary plane partition.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_CSTCPP()
False
sage: PP = PlanePartition([[4,4,3,2],[4,3,2,1],[3,2,1,0],[2,1,0,0]])
sage: PP.is_CSTCPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_CSTCPP()
False
>>> PP = PlanePartition([[Integer(4),Integer(4),Integer(3),Integer(2)],
↪[Integer(4),Integer(3),Integer(2),Integer(1)],[Integer(3),Integer(2),
↪Integer(1),Integer(0)],[Integer(2),Integer(1),Integer(0),Integer(0)]])
>>> PP.is_CSTCPP()
True
```

is_SCPP()

Return whether self is a self-complementary plane partition.

Note that the complement of a plane partition (and thus the property of being self-complementary) is dependent on the choice of a box that it is contained in. If no parent/bounding box is specified, the box is taken to be the smallest box that contains the plane partition.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_SCPP()
False
sage: PP = PlanePartition([[4,4,4,4],[4,4,2,0],[4,2,0,0],[0,0,0,0]])
sage: PP.is_SCPP()
False
sage: PP = PlanePartitions([4,4,4])([[4,4,4,4],[4,4,2,0],[4,2,0,0],[0,0,0,0]])
sage: PP.is_SCPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_SCPP()
False
>>> PP = PlanePartition([[Integer(4),Integer(4),Integer(4),Integer(4)],
↪[Integer(4),Integer(4),Integer(2),Integer(0)],[Integer(4),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(0)]])
>>> PP.is_SCPP()
False
>>> PP = PlanePartitions([Integer(4),Integer(4),Integer(4)])([[Integer(4),
↪Integer(4),Integer(4),Integer(4)],[Integer(4),Integer(4),Integer(2),
↪Integer(0)],[Integer(4),Integer(2),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(0),Integer(0)]])
>>> PP.is_SCPP()
True
```

is_SPP()

Return whether `self` is a symmetric plane partition.

A plane partition is symmetric if the corresponding tableau is symmetric about the diagonal.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_SPP()
False
sage: PP = PlanePartition([[3,3,2],[3,3,2],[2,2,2]])
sage: PP.is_SPP()
True
sage: PP = PlanePartition([[3,2,1],[2,0,0]])
sage: PP.is_SPP()
False
sage: PP = PlanePartition([[3,2,0],[2,0,0]])
sage: PP.is_SPP()
True
sage: PP = PlanePartition([[3,2],[2,0],[1,0]])
sage: PP.is_SPP()
False
sage: PP = PlanePartition([[3,2],[2,0],[0,0]])
sage: PP.is_SPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_SPP()
False
>>> PP = PlanePartition([[Integer(3),Integer(3),Integer(2)],[Integer(3),
↪Integer(3),Integer(2)],[Integer(2),Integer(2),Integer(2)]])
>>> PP.is_SPP()
True
>>> PP = PlanePartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(0),Integer(0)]])
>>> PP.is_SPP()
False
>>> PP = PlanePartition([[Integer(3),Integer(2),Integer(0)],[Integer(2),
↪Integer(0),Integer(0)]])
>>> PP.is_SPP()
True
```

(continues on next page)

(continued from previous page)

```
>>> PP = PlanePartition([[Integer(3),Integer(2)],[Integer(2),Integer(0)],
↪[Integer(1),Integer(0)]])
>>> PP.is_SPP()
False
>>> PP = PlanePartition([[Integer(3),Integer(2)],[Integer(2),Integer(0)],
↪[Integer(0),Integer(0)]])
>>> PP.is_SPP()
True
```

is_SSCPP()

Return whether `self` is a symmetric, self-complementary plane partition.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_SSCPP()
False
sage: PP = PlanePartition([[4,3,3,2],[3,2,2,1],[3,2,2,1],[2,1,1,0]])
sage: PP.is_SSCPP()
True
sage: PP = PlanePartition([[2,1],[1,0]])
sage: PP.is_SSCPP()
True
sage: PP = PlanePartition([[4,3,2],[3,2,1],[2,1,0]])
sage: PP.is_SSCPP()
True
sage: PP = PlanePartition([[4,2,2,2],[2,2,2,2],[2,2,2,2],[2,2,2,0]])
sage: PP.is_SSCPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_SSCPP()
False
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(2)],
↪[Integer(3),Integer(2),Integer(2),Integer(1)],[Integer(3),Integer(2),
↪Integer(2),Integer(1)],[Integer(2),Integer(1),Integer(1),Integer(0)]])
>>> PP.is_SSCPP()
True
>>> PP = PlanePartition([[Integer(2),Integer(1)],[Integer(1),Integer(0)]])
>>> PP.is_SSCPP()
True
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(2)],[Integer(3),
↪Integer(2),Integer(1)],[Integer(2),Integer(1),Integer(0)]])
>>> PP.is_SSCPP()
True
>>> PP = PlanePartition([[Integer(4),Integer(2),Integer(2),Integer(2)],
↪[Integer(2),Integer(2),Integer(2),Integer(2)],[Integer(2),Integer(2),
↪Integer(2),Integer(2)],[Integer(2),Integer(2),Integer(2),Integer(0)]])
>>> PP.is_SSCPP()
True
```

is_TCPP()

Return whether `self` is a transpose-complementary plane partition.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_TCPP()
False
sage: PP = PlanePartition([[4,4,3,2],[4,4,2,1],[4,2,0,0],[2,0,0,0]])
sage: PP.is_TCPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_TCPP()
False
>>> PP = PlanePartition([[Integer(4),Integer(4),Integer(3),Integer(2)],
↪[Integer(4),Integer(4),Integer(2),Integer(1)],[Integer(4),Integer(2),
↪Integer(0),Integer(0)],[Integer(2),Integer(0),Integer(0),Integer(0)]])
>>> PP.is_TCPP()
True
```

is_TSPP()

Return whether `self` is a totally symmetric plane partition.

A plane partition is totally symmetric if it is both symmetric and cyclically symmetric.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_TSPP()
False
sage: PP = PlanePartition([[3,3,3],[3,3,2],[3,2,1]])
sage: PP.is_TSPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_TSPP()
False
>>> PP = PlanePartition([[Integer(3),Integer(3),Integer(3)],[Integer(3),
↪Integer(3),Integer(2)],[Integer(3),Integer(2),Integer(1)]])
>>> PP.is_TSPP()
True
```

is_TSSCPP()

Return whether `self` is a totally symmetric self-complementary plane partition.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.is_TSSCPP()
False
```

(continues on next page)

(continued from previous page)

```
sage: PP = PlanePartition([[4,4,3,2],[4,3,2,1],[3,2,1,0],[2,1,0,0]])
sage: PP.is_TSSCPP()
True
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.is_TSSCPP()
False
>>> PP = PlanePartition([[Integer(4),Integer(4),Integer(3),Integer(2)],
↪[Integer(4),Integer(3),Integer(2),Integer(1)],[Integer(3),Integer(2),
↪Integer(1),Integer(0)],[Integer(2),Integer(1),Integer(0),Integer(0)]])
>>> PP.is_TSSCPP()
True
```

maximal_boxes()

Return the coordinates of the maximal boxes of `self`.

The maximal boxes of a plane partitions are the boxes that can be removed from a plane partition and still yield a valid plane partition.

EXAMPLES:

```
sage: sorted(PlanePartition([[3,2,1],[2,2],[2]]).maximal_boxes())
[[0, 0, 2], [0, 2, 0], [1, 1, 1], [2, 0, 1]]
sage: sorted(PlanePartition([[2,1],[1],[1]]).maximal_boxes())
[[0, 0, 1], [0, 1, 0], [2, 0, 0]]
```

```
>>> from sage.all import *
>>> sorted(PlanePartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(2)]]).maximal_boxes())
[[0, 0, 2], [0, 2, 0], [1, 1, 1], [2, 0, 1]]
>>> sorted(PlanePartition([[Integer(2),Integer(1)],[Integer(1)],
↪[Integer(1)]]).maximal_boxes())
[[0, 0, 1], [0, 1, 0], [2, 0, 0]]
```

number_of_boxes()

Return the number of boxes in the plane partition.

EXAMPLES:

```
sage: PP = PlanePartition([[3,1],[2]])
sage: PP.number_of_boxes()
6
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(3),Integer(1)],[Integer(2)]])
>>> PP.number_of_boxes()
6
```

plot(*show_box=False*, *colors=None*)

Return a plot of `self`.

INPUT:

- `show_box` – boolean (default: `False`); if `True`, also shows the visible tiles on the xy-, yz-, zx-planes
- `colors` – (default: `["white", "lightgray", "darkgray"]`) list `[A, B, C]` of 3 strings representing colors

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 27 graphics primitives
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 27 graphics primitives
```

plot3d(*colors=None*)

Return a 3D-plot of `self`.

INPUT:

- `colors` – (default: `["white", "lightgray", "darkgray"]`) list `[A, B, C]` of 3 strings representing colors

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.plot3d()                                                            #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.plot3d()                                                            #␣
↪needs sage.plot
Graphics3d Object
```

pp(*show_box=False*)

Return a pretty print of the plane partition.

INPUT:

- `show_box` – boolean (default: `False`); if `True`, also shows the visible tiles on the xy-, yz-, zx-planes

OUTPUT: a pretty print of the plane partition

EXAMPLES:

```
sage: PlanePartition([[4,3,3,1],[2,1,1],[1,1]]).pp()
        __
       /\_\
    __/\/_/
 __/\_\/\_\
```

(continues on next page)

(continued from previous page)

```
/\_\/_/\/\_\
\/\_\_\/\/_/
 \/_/\_\/_/
    \/_/\_\
       \/_/
sage: PlanePartition([[4,3,3,1],[2,1,1],[1,1]]).pp(True)
    ______
   /_/_/\_\
  /_/_/\/_/\
 /_/\_\/\_\/\
/\_\/_/\/\_\/\
\/\_\_\/\/_/\/
 \/_/\_\/_/\/
  \_\/_/\_\/
   \_\_\/_/
```

```
>>> from sage.all import *
>>> PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],[Integer(2),
↪Integer(1),Integer(1)],[Integer(1),Integer(1)]]).pp()
        __
       /\_\
    __/\/_/
 __/\_\/\_\
/\_\/_/\/\_\
\/\_\_\/\/_/
 \/_/\_\/_/
    \/_/\_\
       \/_/
>>> PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],[Integer(2),
↪Integer(1),Integer(1)],[Integer(1),Integer(1)]]).pp(True)
    ______
   /_/_/\_\
  /_/_/\/_/\
 /_/\_\/\_\/\
/\_\/_/\/\_\/\
\/\_\_\/\/_/\/
 \/_/\_\/_/\/
  \_\/_/\_\/
   \_\_\/_/
```

to_order_ideal()

Return the order ideal corresponding to `self`.

> **Todo**
>
> As many families of symmetric plane partitions are in bijection with order ideals in an associated poset, this function could feasibly have options to send symmetric plane partitions to the associated order ideal in that poset, instead.

EXAMPLES:

```
sage: PlanePartition([[3,2,1],[2,2],[2]]).to_order_ideal()                    #␣
↪needs sage.graphs sage.modules
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (0, 1, 1), (0, 2, 0),
 (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1), (2, 0, 0), (2, 0, 1)]
sage: PlanePartition([[2,1],[1],[1]]).to_order_ideal()                        #␣
↪needs sage.graphs sage.modules
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (2, 0, 0)]
```

```
>>> from sage.all import *
>>> PlanePartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(2)],[Integer(2)]]).to_order_ideal()                        # needs sage.
↪graphs sage.modules
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (0, 1, 1), (0, 2, 0),
 (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1), (2, 0, 0), (2, 0, 1)]
>>> PlanePartition([[Integer(2),Integer(1)],[Integer(1)],[Integer(1)]]).to_
↪order_ideal()                          # needs sage.graphs sage.modules
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (2, 0, 0)]
```

to_tableau()

Return the tableau class of `self`.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.to_tableau()
[[4, 3, 3, 1], [2, 1, 1], [1, 1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.to_tableau()
[[4, 3, 3, 1], [2, 1, 1], [1, 1]]
```

transpose(*tableau_only=False*)

Return the transpose of `self`.

If `tableau_only` is set to `True`, then only the tableau consisting of the projection of boxes size onto the xy-plane is returned instead of a `PlanePartition`. This will not necessarily have trailing rows or trailing zeros removed.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.transpose()
Plane partition [[4, 2, 1], [3, 1, 1], [3, 1], [1]]
sage: PP.transpose(True)
[[4, 2, 1], [3, 1, 1], [3, 1, 0], [1, 0, 0]]

sage: PPP = PlanePartitions([1, 2, 3])
sage: PP = PPP([[1, 1]])
sage: PT = PP.transpose(); PT
Plane partition [[1], [1]]
sage: PT.parent()
Plane partitions inside a 2 x 1 x 3 box
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.transpose()
Plane partition [[4, 2, 1], [3, 1, 1], [3, 1], [1]]
>>> PP.transpose(True)
[[4, 2, 1], [3, 1, 1], [3, 1, 0], [1, 0, 0]]

>>> PPP = PlanePartitions([Integer(1), Integer(2), Integer(3)])
>>> PP = PPP([[Integer(1), Integer(1)]])
>>> PT = PP.transpose(); PT
Plane partition [[1], [1]]
>>> PT.parent()
Plane partitions inside a 2 x 1 x 3 box
```

x_tableau(*tableau=True*)

Return the projection of `self` in the x direction.

If `tableau` is set to `False`, then only the list of lists consisting of the projection of boxes size onto the yz-plane is returned instead of a `Tableau` object. This output will not have empty trailing rows or trailing zeros removed.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.x_tableau()
[[3, 2, 1, 1], [3, 1, 1, 0], [2, 1, 1, 0], [1, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.x_tableau()
[[3, 2, 1, 1], [3, 1, 1, 0], [2, 1, 1, 0], [1, 0, 0, 0]]
```

y_tableau(*tableau=True*)

Return the projection of `self` in the y direction.

If `tableau` is set to `False`, then only the list of lists consisting of the projection of boxes size onto the xz-plane is returned instead of a `Tableau` object. This output will not have empty trailing rows or trailing zeros removed.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.y_tableau()
[[4, 3, 2], [3, 1, 0], [3, 0, 0], [1, 0, 0]]
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.y_tableau()
[[4, 3, 2], [3, 1, 0], [3, 0, 0], [1, 0, 0]]
```

z_tableau(*tableau=True*)

Return the projection of `self` in the z direction.

If `tableau` is set to `False`, then only the list of lists consisting of the projection of boxes size onto the xy-plane is returned instead of a *Tableau* object. This output will not have empty trailing rows or trailing zeros removed.

EXAMPLES:

```
sage: PP = PlanePartition([[4,3,3,1],[2,1,1],[1,1]])
sage: PP.z_tableau()
[[4, 3, 3, 1], [2, 1, 1, 0], [1, 1, 0, 0]]
```

```
>>> from sage.all import *
>>> PP = PlanePartition([[Integer(4),Integer(3),Integer(3),Integer(1)],
↪[Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> PP.z_tableau()
[[4, 3, 3, 1], [2, 1, 1, 0], [1, 1, 0, 0]]
```

class `sage.combinat.plane_partition.`**`PlanePartitions`**(*box_size=None*, *symmetry=None*, *category=None*)

Bases: `UniqueRepresentation`, `Parent`

Plane partitions.

`PlanePartitions()` returns the class of all plane partitions.

`PlanePartitions(n)` return the class of all plane partitions with precisely n boxes.

`PlanePartitions([a, b, c])` returns the class of plane partitions that fit inside an $a \times b \times c$ box.

`PlanePartitions([a, b, c])` has the optional keyword `symmetry`, which restricts the plane partitions inside a box of the specified size satisfying certain symmetry conditions.

- `symmetry='SPP'` gives the class of symmetric plane partitions. which is all plane partitions fixed under reflection across the diagonal. Requires that $a = b$.
- `symmetry='CSPP'` gives the class of cyclic plane partitions, which is all plane partitions fixed under cyclic rotation of coordinates. Requires that $a = b = c$.
- `symmetry='TSPP'` gives the class of totally symmetric plane partitions, which is all plane partitions fixed under any interchanging of coordinates. Requires that $a = b = c$.
- `symmetry='SCPP'` gives the class of self-complementary plane partitions. which is all plane partitions that are equal to their own complement in the specified box. Requires at least one of a, b, c be even.
- `symmetry='TCPP'` gives the class of transpose complement plane partitions, which is all plane partitions whose complement in the box of the specified size is equal to their transpose. Requires $a = b$ and at least one of a, b, c be even.
- `symmetry='SSCPP'` gives the class of symmetric self-complementary plane partitions, which is all plane partitions that are both symmetric and self-complementary. Requires $a = b$ and at least one of a, b, c be even.
- `symmetry='CSTCPP'` gives the class of cyclically symmetric transpose complement plane partitions, which is all plane partitions that are both symmetric and equal to the transpose of their complement. Requires $a = b = c$.
- `symmetry='CSSCPP'` gives the class of cyclically symmetric self-complementary plane partitions, which is all plane partitions that are both cyclically symmetric and self-complementary. Requires $a = b = c$ and all a, b, c be even.

- `symmetry='TSSCPP'` gives the class of totally symmetric self-complementary plane partitions, which is all plane partitions that are totally symmetric and also self-complementary. Requires $a = b = c$ and all a, b, c be even.

EXAMPLES:

If no arguments are passed, then the class of all plane partitions is returned:

```
sage: PlanePartitions()
Plane partitions
sage: [[2,1],[1]] in PlanePartitions()
True
```

```
>>> from sage.all import *
>>> PlanePartitions()
Plane partitions
>>> [[Integer(2),Integer(1)],[Integer(1)]] in PlanePartitions()
True
```

If an integer n is passed, then the class of plane partitions of n is returned:

```
sage: PlanePartitions(3)
Plane partitions of size 3
sage: PlanePartitions(3).list()
[Plane partition [[3]],
 Plane partition [[2, 1]],
 Plane partition [[1, 1, 1]],
 Plane partition [[2], [1]],
 Plane partition [[1, 1], [1]],
 Plane partition [[1], [1], [1]]]
```

```
>>> from sage.all import *
>>> PlanePartitions(Integer(3))
Plane partitions of size 3
>>> PlanePartitions(Integer(3)).list()
[Plane partition [[3]],
 Plane partition [[2, 1]],
 Plane partition [[1, 1, 1]],
 Plane partition [[2], [1]],
 Plane partition [[1, 1], [1]],
 Plane partition [[1], [1], [1]]]
```

If a three-element tuple or list $[a, b, c]$ is passed, then the class of all plane partitions that fit inside and $a \times b \times c$ box is returned:

```
sage: PlanePartitions([2,2,2])
Plane partitions inside a 2 x 2 x 2 box
sage: [[2,1],[1]] in PlanePartitions([2,2,2])
True
```

```
>>> from sage.all import *
>>> PlanePartitions([Integer(2),Integer(2),Integer(2)])
Plane partitions inside a 2 x 2 x 2 box
>>> [[Integer(2),Integer(1)],[Integer(1)]] in PlanePartitions([Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(2)])
True
```

If an additional keyword `symmetry` is pass along with a three-element tuple or list $[a, b, c]$, then the class of all plane partitions that fit inside an $a \times b \times c$ box with the specified symmetry is returned:

```
sage: PlanePartitions([2,2,2], symmetry='CSPP')
Cyclically symmetric plane partitions inside a 2 x 2 x 2 box
sage: [[2,1],[1]] in PlanePartitions([2,2,2], symmetry='CSPP')
True
```

```
>>> from sage.all import *
>>> PlanePartitions([Integer(2),Integer(2),Integer(2)], symmetry='CSPP')
Cyclically symmetric plane partitions inside a 2 x 2 x 2 box
>>> [[Integer(2),Integer(1)],[Integer(1)]] in PlanePartitions([Integer(2),
↪Integer(2),Integer(2)], symmetry='CSPP')
True
```

See also:

- *PlanePartition*
- *PlanePartitions_all*
- *PlanePartitions_n*
- *PlanePartitions_box*
- *PlanePartitions_SPP*
- *PlanePartitions_CSPP*
- *PlanePartitions_TSPP*
- *PlanePartitions_SCPP*
- *PlanePartitions_TCPP*
- *PlanePartitions_SSCPP*
- *PlanePartitions_CSTCPP*
- *PlanePartitions_CSSCPP*
- *PlanePartitions_TSSCPP*

Element

alias of *PlanePartition*

box()

Return the size of the box of the plane partition of `self` is contained in.

EXAMPLES:

```
sage: P = PlanePartitions([4,3,5])
sage: P.box()
(4, 3, 5)
```

(continues on next page)

(continued from previous page)

```
sage: PP = PlanePartitions()
sage: PP.box() is None
True
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(4),Integer(3),Integer(5)])
>>> P.box()
(4, 3, 5)

>>> PP = PlanePartitions()
>>> PP.box() is None
True
```

symmetry()

Return the symmetry class of `self`.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,2], symmetry='SPP')
sage: PP.symmetry()
'SPP'
sage: PP = PlanePartitions()
sage: PP.symmetry() is None
True
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(2)], symmetry='SPP')
>>> PP.symmetry()
'SPP'
>>> PP = PlanePartitions()
>>> PP.symmetry() is None
True
```

class sage.combinat.plane_partition.**PlanePartitions_CSPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are cyclically symmetric.

cardinality()

Return the cardinality of `self`.

The number of cyclically symmetric plane partitions inside an $a \times a \times a$ box is equal to

$$\left(\prod_{i=1}^{a} \frac{3i-1}{3i-2}\right)\left(\prod_{1\leq i<j\leq a} \frac{i+j+a-1}{2i+j-1}\right).$$

EXAMPLES:

```
sage: P = PlanePartitions([4,4,4], symmetry='CSPP')
sage: P.cardinality()
132
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(4),Integer(4),Integer(4)], symmetry='CSPP')
>>> P.cardinality()
132
```

from_antichain(*acl*)

Return the cyclically symmetric plane partition corresponding to an antichain in the poset given in *to_poset()*.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,3], symmetry='CSPP')
sage: A = [(0, 2, 2), (1, 1, 1)]
sage: PP.from_antichain(A)
Plane partition [[3, 3, 3], [3, 2, 1], [3, 1, 1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(3)], symmetry='CSPP')
>>> A = [(Integer(0), Integer(2), Integer(2)), (Integer(1), Integer(1),␣
↪Integer(1))]
>>> PP.from_antichain(A)
Plane partition [[3, 3, 3], [3, 2, 1], [3, 1, 1]]
```

from_order_ideal(*I*)

Return the cylically symmetric plane partition corresponding to an order ideal in the poset given in *to_poset()*.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,3], symmetry='CSPP')
sage: I = [(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 1), (0, 1, 2),
....:      (1, 0, 2), (0, 2, 2), (1, 1, 1), (1, 1, 2), (1, 2, 2)]
sage: PP.from_order_ideal(I)                                             #␣
↪needs sage.graphs
Plane partition [[3, 3, 3], [3, 3, 3], [3, 3, 2]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(3)], symmetry='CSPP')
>>> I = [(Integer(0), Integer(0), Integer(0)), (Integer(0), Integer(0),␣
↪Integer(1)), (Integer(0), Integer(0), Integer(2)), (Integer(0), Integer(1),␣
↪Integer(1)), (Integer(0), Integer(1), Integer(2)),
...      (Integer(1), Integer(0), Integer(2)), (Integer(0), Integer(2),␣
↪Integer(2)), (Integer(1), Integer(1), Integer(1)), (Integer(1), Integer(1),␣
↪Integer(2)), (Integer(1), Integer(2), Integer(2))]
>>> PP.from_order_ideal(I)                                             #␣
↪needs sage.graphs
Plane partition [[3, 3, 3], [3, 3, 3], [3, 3, 2]]
```

random_element()

Return a uniformly random element of `self`.

ALGORITHM:

This uses the *random_order_ideal()* method and the natural bijection between cyclically symmetric plane partitions and order ideals in an associated poset.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,3], symmetry='CSPP')
sage: PP.random_element()   # random                                        #␣
↪needs sage.graphs
Plane partition [[3, 2, 2], [3, 1], [1, 1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(3)], symmetry='CSPP')
>>> PP.random_element()   # random                                        #␣
↪needs sage.graphs
Plane partition [[3, 2, 2], [3, 1], [1, 1]]
```

to_poset()

Return a partially ordered set whose order ideals are in bijection with cyclically symmetric plane partitions.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,3], symmetry='CSPP')
sage: PP.to_poset()                                                         #␣
↪needs sage.graphs
Finite poset containing 11 elements
sage: PP.to_poset().order_ideals_lattice().cardinality() == PP.cardinality() ␣
↪          # needs sage.graphs
True
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(3)], symmetry='CSPP')
>>> PP.to_poset()                                                         #␣
↪needs sage.graphs
Finite poset containing 11 elements
>>> PP.to_poset().order_ideals_lattice().cardinality() == PP.cardinality()   ␣
↪          # needs sage.graphs
True
```

class sage.combinat.plane_partition.**PlanePartitions_CSSCPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are cyclically symmetric self-complementary.

cardinality()

Return the cardinality of `self`.

The number of cyclically symmetric self-complementary plane partitions inside a $2a \times 2a \times 2a$ box is equal to

$$\left(\prod_{i=0}^{a-1} \frac{(3i+1)!}{(a+i)!}\right)^2.$$

EXAMPLES:

```
sage: P = PlanePartitions([6,6,6], symmetry='CSSCPP')
sage: P.cardinality()
49
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(6),Integer(6),Integer(6)], symmetry='CSSCPP')
>>> P.cardinality()
49
```

class sage.combinat.plane_partition.**PlanePartitions_CSTCPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are cyclically symmetric and transpose-complement.

cardinality()

Return the cardinality of self.

The number of cyclically symmetric transpose complement plane partitions inside a $2a \times 2a \times 2a$ box is equal to

$$\prod_{i=0}^{a-1} \frac{(3i+1)(6i)!(2i)!}{(4i+1)!(4i)!}.$$

EXAMPLES:

```
sage: P = PlanePartitions([6,6,6], symmetry='CSTCPP')
sage: P.cardinality()
11
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(6),Integer(6),Integer(6)], symmetry='CSTCPP')
>>> P.cardinality()
11
```

class sage.combinat.plane_partition.**PlanePartitions_SCPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are self-complementary.

cardinality()

Return the cardinality of self.

The number of self complementary plane partitions inside a $2a \times 2b \times 2c$ box is equal to

$$\left(\prod_{i=1}^{r}\prod_{j=1}^{b} \frac{i+j+c-1}{i+j-1}\right)^2.$$

The number of self complementary plane partitions inside an $(2a+1) \times 2b \times 2c$ box is equal to

$$\left(\prod_{i=1}^{a}\prod_{j=1}^{b} \frac{i+j+c-1}{i+j-1}\right)\left(\prod_{i=1}^{a+1}\prod_{j=1}^{b} \frac{i+j+c-1}{i+j-1}\right).$$

The number of self complementary plane partitions inside an $(2a+1) \times (2b+1) \times 2c$ box is equal to

$$\left(\prod_{i=1}^{a+1}\prod_{j=1}^{b} \frac{i+j+c-1}{i+j-1}\right)\left(\prod_{i=1}^{a}\prod_{j=1}^{b+1} \frac{i+j+c-1}{i+j-1}\right).$$

EXAMPLES:

```
sage: P = PlanePartitions([4,4,4], symmetry='SCPP')
sage: P.cardinality()
400

sage: P = PlanePartitions([5,4,4], symmetry='SCPP')
sage: P.cardinality()
1000
sage: P = PlanePartitions([4,5,4], symmetry='SCPP')
sage: P.cardinality()
1000
sage: P = PlanePartitions([4,4,5], symmetry='SCPP')
sage: P.cardinality()
1000

sage: P = PlanePartitions([5,5,4], symmetry='SCPP')
sage: P.cardinality()
2500
sage: P = PlanePartitions([5,4,5], symmetry='SCPP')
sage: P.cardinality()
2500
sage: P = PlanePartitions([4,5,5], symmetry='SCPP')
sage: P.cardinality()
2500
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(4),Integer(4),Integer(4)], symmetry='SCPP')
>>> P.cardinality()
400

>>> P = PlanePartitions([Integer(5),Integer(4),Integer(4)], symmetry='SCPP')
>>> P.cardinality()
1000
>>> P = PlanePartitions([Integer(4),Integer(5),Integer(4)], symmetry='SCPP')
>>> P.cardinality()
1000
>>> P = PlanePartitions([Integer(4),Integer(4),Integer(5)], symmetry='SCPP')
>>> P.cardinality()
1000

>>> P = PlanePartitions([Integer(5),Integer(5),Integer(4)], symmetry='SCPP')
>>> P.cardinality()
2500
>>> P = PlanePartitions([Integer(5),Integer(4),Integer(5)], symmetry='SCPP')
>>> P.cardinality()
2500
>>> P = PlanePartitions([Integer(4),Integer(5),Integer(5)], symmetry='SCPP')
>>> P.cardinality()
2500
```

class sage.combinat.plane_partition.**PlanePartitions_SPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are symmetric.

cardinality()

Return the cardinality of `self`.

The number of symmetric plane partitions inside an $a \times a \times b$ box is equal to

$$\left(\prod_{i=1}^{a} \frac{2i+b-1}{2i-1}\right)\left(\prod_{1\leq i<j\leq a} \frac{i+j+b-1}{i+j-1}\right).$$

EXAMPLES:

```
sage: P = PlanePartitions([3,3,2], symmetry='SPP')
sage: P.cardinality()
35
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(3),Integer(3),Integer(2)], symmetry='SPP')
>>> P.cardinality()
35
```

from_antichain(A)

Return the symmetric plane partition corresponding to an antichain in the poset given in `to_poset()`.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,2], symmetry='SPP')
sage: A = [(2, 2, 0), (1, 0, 1), (1, 1, 0)]
sage: PP.from_antichain(A)
Plane partition [[2, 2, 1], [2, 1, 1], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(2)], symmetry='SPP')
>>> A = [(Integer(2), Integer(2), Integer(0)), (Integer(1), Integer(0),␣
↪Integer(1)), (Integer(1), Integer(1), Integer(0))]
>>> PP.from_antichain(A)
Plane partition [[2, 2, 1], [2, 1, 1], [1, 1, 1]]
```

from_order_ideal(I)

Return the symmetric plane partition corresponding to an order ideal in the poset given in `to_poset()`.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,2], symmetry='SPP')
sage: I = [(0, 0, 0), (1, 0, 0), (1, 1, 0), (2, 0, 0)]
sage: PP.from_order_ideal(I)                                            #␣
↪needs sage.graphs
Plane partition [[1, 1, 1], [1, 1], [1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(2)], symmetry='SPP')
>>> I = [(Integer(0), Integer(0), Integer(0)), (Integer(1), Integer(0),␣
↪Integer(0)), (Integer(1), Integer(1), Integer(0)), (Integer(2), Integer(0),␣
↪Integer(0))]
>>> PP.from_order_ideal(I)                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
Plane partition [[1, 1, 1], [1, 1], [1]]
```

random_element()

Return a uniformly random element of `self`.

ALGORITHM:

This uses the *random_order_ideal()* method and the natural bijection between symmetric plane partitions and order ideals in an associated poset.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,2], symmetry='SPP')
sage: PP.random_element()  # random                                        #␣
↪needs sage.graphs
Plane partition [[2, 2, 2], [2, 2], [2]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(2)], symmetry='SPP')
>>> PP.random_element()  # random                                        #␣
↪needs sage.graphs
Plane partition [[2, 2, 2], [2, 2], [2]]
```

to_poset()

Return a poset whose order ideals are in bijection with symmetric plane partitions.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,2], symmetry='SPP')
sage: PP.to_poset()                                                        #␣
↪needs sage.graphs
Finite poset containing 12 elements
sage: PP.to_poset().order_ideals_lattice().cardinality() == PP.cardinality() ␣
↪           # needs sage.graphs sage.modules sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(2)], symmetry='SPP')
>>> PP.to_poset()                                                        #␣
↪needs sage.graphs
Finite poset containing 12 elements
>>> PP.to_poset().order_ideals_lattice().cardinality() == PP.cardinality()   ␣
↪         # needs sage.graphs sage.modules sage.rings.finite_rings
True
```

class sage.combinat.plane_partition.**PlanePartitions_SSCPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are symmetric self-complementary.

cardinality()

Return the cardinality of `self`.

The number of symmetric self-complementary plane partitions inside a $2a \times 2a \times 2b$ box is equal to

$$\prod_{i=1}^{a}\prod_{j=1}^{a}\frac{i+j+b-1}{i+j-1}.$$

The number of symmetric self-complementary plane partitions inside a $(2a+1)\times(2a+1)\times 2b$ box is equal to

$$\prod_{i=1}^{a}\prod_{j=1}^{a+1}\frac{i+j+b-1}{i+j-1}.$$

EXAMPLES:

```
sage: P = PlanePartitions([4,4,2], symmetry='SSCPP')
sage: P.cardinality()
6
sage: Q = PlanePartitions([3,3,2], symmetry='SSCPP')
sage: Q.cardinality()
3
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(4),Integer(4),Integer(2)], symmetry='SSCPP')
>>> P.cardinality()
6
>>> Q = PlanePartitions([Integer(3),Integer(3),Integer(2)], symmetry='SSCPP')
>>> Q.cardinality()
3
```

class sage.combinat.plane_partition.**PlanePartitions_TCPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are transpose-complement.

cardinality()

Return the cardinality of `self`.

The number of transpose complement plane partitions inside an $a \times a \times 2b$ box is equal to

$$\binom{b+1-1}{a-1}\prod_{1\leq i,j\leq a-2}\frac{i+j+2b-1}{i+j-1}.$$

EXAMPLES:

```
sage: P = PlanePartitions([3,3,2], symmetry='TCPP')
sage: P.cardinality()
5
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(3),Integer(3),Integer(2)], symmetry='TCPP')
>>> P.cardinality()
5
```

class sage.combinat.plane_partition.**PlanePartitions_TSPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are totally symmetric.

cardinality()

Return the cardinality of `self`.

The number of totally symmetric plane partitions inside an $a \times a \times a$ box is equal to

$$\prod_{1 \leq i \leq j \leq a} \frac{i+j+a-1}{i+2j-2}.$$

EXAMPLES:

```
sage: P = PlanePartitions([4,4,4], symmetry='TSPP')
sage: P.cardinality()
66
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(4),Integer(4),Integer(4)], symmetry='TSPP')
>>> P.cardinality()
66
```

from_antichain(*acl*)

Return the totally symmetric plane partition corresponding to an antichain in the poset given in `to_poset()`.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,3], symmetry='TSPP')
sage: A = [(0, 0, 2), (0, 1, 1)]
sage: PP.from_antichain(A)
Plane partition [[3, 2, 1], [2, 1], [1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(3)], symmetry='TSPP')
>>> A = [(Integer(0), Integer(0), Integer(2)), (Integer(0), Integer(1),␣
↪Integer(1))]
>>> PP.from_antichain(A)
Plane partition [[3, 2, 1], [2, 1], [1]]
```

from_order_ideal(*I*)

Return the totally symmetric plane partition corresponding to an order ideal in the poset given in `to_poset()`.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,3], symmetry='TSPP')
sage: I = [(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 1)]
sage: PP.from_order_ideal(I)                                              #␣
↪needs sage.graphs
Plane partition [[3, 2, 1], [2, 1], [1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(3)], symmetry='TSPP')
>>> I = [(Integer(0), Integer(0), Integer(0)), (Integer(0), Integer(0),␣
↪Integer(1)), (Integer(0), Integer(0), Integer(2)), (Integer(0), Integer(1),␣
↪Integer(1))]
>>> PP.from_order_ideal(I)                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
Plane partition [[3, 2, 1], [2, 1], [1]]
```

to_poset()

Return a poset whose order ideals are in bijection with totally symmetric plane partitions.

EXAMPLES:

```
sage: PP = PlanePartitions([3,3,3], symmetry='TSPP')
sage: PP.to_poset()                                                    #␣
↪needs sage.graphs
Finite poset containing 10 elements
sage: (PP.to_poset().order_ideals_lattice().cardinality()              #␣
↪needs sage.graphs sage.modules sage.rings.finite_rings
....:     == PP.cardinality())
True
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(3),Integer(3),Integer(3)], symmetry='TSPP')
>>> PP.to_poset()                                                   #␣
↪needs sage.graphs
Finite poset containing 10 elements
>>> (PP.to_poset().order_ideals_lattice().cardinality()             #␣
↪needs sage.graphs sage.modules sage.rings.finite_rings
...     == PP.cardinality())
True
```

class sage.combinat.plane_partition.**PlanePartitions_TSSCPP**(*box_size*)

Bases: *PlanePartitions*

Plane partitions that fit inside a box of a specified size that are totally symmetric self-complementary.

cardinality()

Return the cardinality of `self`.

The number of totally symmetric self-complementary plane partitions inside a $2a \times 2a \times 2a$ box is equal to

$$\prod_{i=0}^{a-1} \frac{(3i+1)!}{(a+i)!}.$$

EXAMPLES:

```
sage: P = PlanePartitions([6,6,6], symmetry='TSSCPP')
sage: P.cardinality()
7
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(6),Integer(6),Integer(6)], symmetry='TSSCPP')
>>> P.cardinality()
7
```

from_antichain(*acl*)

Return the totally symmetric self-complementary plane partition corresponding to an antichain in the poset given in *to_poset()*.

EXAMPLES:

```
sage: PP = PlanePartitions([6,6,6], symmetry='TSSCPP')
sage: A = [(0, 0, 1), (1, 1, 0)]
sage: PP.from_antichain(A)
Plane partition [[6, 6, 6, 5, 5, 3], [6, 5, 5, 4, 3, 1], [6, 5, 4, 3, 2, 1],
                 [5, 4, 3, 2, 1], [5, 3, 2, 1, 1], [3, 1, 1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(6),Integer(6),Integer(6)], symmetry='TSSCPP
↪')
>>> A = [(Integer(0), Integer(0), Integer(1)), (Integer(1), Integer(1),␣
↪Integer(0))]
>>> PP.from_antichain(A)
Plane partition [[6, 6, 6, 5, 5, 3], [6, 5, 5, 4, 3, 1], [6, 5, 4, 3, 2, 1],
                 [5, 4, 3, 2, 1], [5, 3, 2, 1, 1], [3, 1, 1]]
```

from_order_ideal(*I*)

Return the totally symmetric self-complementary plane partition corresponding to an order ideal in the poset given in `to_poset()`.

EXAMPLES:

```
sage: PP = PlanePartitions([6,6,6], symmetry='TSSCPP')                     #␣
↪needs sage.graphs
sage: I = [(0, 0, 0), (0, 1, 0), (1, 1, 0)]
sage: PP.from_order_ideal(I)                                               #␣
↪needs sage.graphs
Plane partition [[6, 6, 6, 5, 5, 3], [6, 5, 5, 3, 3, 1], [6, 5, 5, 3, 3, 1],
                 [5, 3, 3, 1, 1], [5, 3, 3, 1, 1], [3, 1, 1]]
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(6),Integer(6),Integer(6)], symmetry='TSSCPP
↪')                          # needs sage.graphs
>>> I = [(Integer(0), Integer(0), Integer(0)), (Integer(0), Integer(1),␣
↪Integer(0)), (Integer(1), Integer(1), Integer(0))]
>>> PP.from_order_ideal(I)                                               #␣
↪needs sage.graphs
Plane partition [[6, 6, 6, 5, 5, 3], [6, 5, 5, 3, 3, 1], [6, 5, 5, 3, 3, 1],
                 [5, 3, 3, 1, 1], [5, 3, 3, 1, 1], [3, 1, 1]]
```

to_poset()

Return a poset whose order ideals are in bijection with totally symmetric self-complementary plane partitions.

EXAMPLES:

```
sage: PP = PlanePartitions([6,6,6], symmetry='TSSCPP')
sage: PP.to_poset()                                                        #␣
↪needs sage.graphs sage.modules
Finite poset containing 4 elements
sage: PP.to_poset().order_ideals_lattice().cardinality() == PP.cardinality() ␣
↪           # needs sage.graphs sage.modules
True
```

```
>>> from sage.all import *
>>> PP = PlanePartitions([Integer(6),Integer(6),Integer(6)], symmetry='TSSCPP
↪')
>>> PP.to_poset()                                                          #␣
↪needs sage.graphs sage.modules
Finite poset containing 4 elements
>>> PP.to_poset().order_ideals_lattice().cardinality() == PP.cardinality()   ␣
↪        # needs sage.graphs sage.modules
True
```

class sage.combinat.plane_partition.**PlanePartitions_all**

Bases: *PlanePartitions*, DisjointUnionEnumeratedSets

All plane partitions.

an_element()

Return a particular element of the class.

class sage.combinat.plane_partition.**PlanePartitions_box**(*box_size*)

Bases: *PlanePartitions*

All plane partitions that fit inside a box of a specified size.

By convention, a plane partition in an $a \times b \times c$ box will have at most a rows, of lengths at most b, with entries at most c.

cardinality()

Return the cardinality of self.

The number of plane partitions inside an $a \times b \times c$ box is equal to

$$\prod_{i=1}^{a} \prod_{j=1}^{b} \prod_{k=1}^{c} \frac{i+j+k-1}{i+j+k-2}.$$

EXAMPLES:

```
sage: P = PlanePartitions([4,3,5])
sage: P.cardinality()
116424
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(4),Integer(3),Integer(5)])
>>> P.cardinality()
116424
```

from_antichain(A)

Return the plane partition corresponding to an antichain in the poset given in *to_poset()*.

EXAMPLES:

```
sage: A = [(1,0,1), (0,1,1), (1,1,0)]
sage: PlanePartitions([2,2,2]).from_antichain(A)
Plane partition [[2, 2], [2, 1]]
```

```
>>> from sage.all import *
>>> A = [(Integer(1),Integer(0),Integer(1)), (Integer(0),Integer(1),
↪Integer(1)), (Integer(1),Integer(1),Integer(0))]
>>> PlanePartitions([Integer(2),Integer(2),Integer(2)]).from_antichain(A)
Plane partition [[2, 2], [2, 1]]
```

from_order_ideal(*I*)

Return the plane partition corresponding to an order ideal in the poset given in *to_poset()*.

EXAMPLES:

```
sage: I = [(1, 0, 0), (1, 0, 1), (1, 1, 0), (0, 1, 0),
....:      (0, 0, 0), (0, 0, 1), (0, 1, 1)]
sage: PlanePartitions([2,2,2]).from_order_ideal(I)                    #␣
↪needs sage.graphs sage.modules
Plane partition [[2, 2], [2, 1]]
```

```
>>> from sage.all import *
>>> I = [(Integer(1), Integer(0), Integer(0)), (Integer(1), Integer(0),␣
↪Integer(1)), (Integer(1), Integer(1), Integer(0)), (Integer(0), Integer(1),␣
↪Integer(0)),
...      (Integer(0), Integer(0), Integer(0)), (Integer(0), Integer(0),␣
↪Integer(1)), (Integer(0), Integer(1), Integer(1))]
>>> PlanePartitions([Integer(2),Integer(2),Integer(2)]).from_order_ideal(I)  ␣
↪                         # needs sage.graphs sage.modules
Plane partition [[2, 2], [2, 1]]
```

random_element()

Return a uniformly random plane partition inside a box.

ALGORITHM:

This uses the *random_order_ideal()* method and the natural bijection with plane partitions.

EXAMPLES:

```
sage: P = PlanePartitions([4,3,5])
sage: P.random_element()   # random                                   #␣
↪needs sage.graphs sage.modules
Plane partition [[4, 3, 3], [4], [2]]
```

```
>>> from sage.all import *
>>> P = PlanePartitions([Integer(4),Integer(3),Integer(5)])
>>> P.random_element()   # random                                   #␣
↪needs sage.graphs sage.modules
Plane partition [[4, 3, 3], [4], [2]]
```

to_poset()

Return the product of three chains poset, whose order ideals are naturally in bijection with plane partitions inside a box.

EXAMPLES:

```
sage: PlanePartitions([2,2,2]).to_poset()                                   #␣
↪needs sage.graphs sage.modules
Finite lattice containing 8 elements
```

```
>>> from sage.all import *
>>> PlanePartitions([Integer(2),Integer(2),Integer(2)]).to_poset()          ␣
↪                       # needs sage.graphs sage.modules
Finite lattice containing 8 elements
```

class sage.combinat.plane_partition.**PlanePartitions_n**(*n*)

Bases: *PlanePartitions*

Plane partitions with a fixed number of boxes.

cardinality()

Return the number of plane partitions with `n` boxes.

Calculated using the recurrence relation

$$PL(n) = \sum_{k=1}^{n} PL(n-k)\sigma_2(k),$$

where $\sigma_k(n)$ is the sum of the k-th powers of divisors of n.

EXAMPLES:

```
sage: P = PlanePartitions(17)
sage: P.cardinality()
18334
```

```
>>> from sage.all import *
>>> P = PlanePartitions(Integer(17))
>>> P.cardinality()
18334
```

5.1.165 Integer partitions

A partition p of a nonnegative integer n is a non-increasing list of positive integers (the *parts* of the partition) with total sum n.

A partition can be depicted by a diagram made of rows of cells, where the number of cells in the i-th row starting from the top is the i-th part of the partition.

The coordinate system related to a partition applies from the top to the bottom and from left to right. So, the corners of the partition $[5, 3, 1]$ are $[[0, 4], [1, 2], [2, 0]]$.

For display options, see *Partitions.options*.

Note

- Boxes is a synonym for cells. All methods will use 'cell' and 'cells' instead of 'box' and 'boxes'.
- Partitions are 0 based with coordinates in the form of (row-index, column-index).
- If given coordinates of the form `(r, c)`, then use Python's *-operator.

- Throughout this documentation, for a partition λ we will denote its conjugate partition by λ'. For more on conjugate partitions, see `Partition.conjugate()`.
- The comparisons on partitions use lexicographic order.

Note: We use the convention that lexicographic ordering is read from left-to-right. That is to say $[1, 3, 7]$ is smaller than $[2, 3, 4]$.

AUTHORS:

- Mike Hansen (2007): initial version
- Dan Drake (2009-03-28): deprecate RestrictedPartitions and implement Partitions_parts_in
- Travis Scrimshaw (2012-01-12): Implemented latex function to Partition_class
- Travis Scrimshaw (2012-05-09): Fixed Partitions(-1).list() infinite recursion loop by saying Partitions_n is the empty set.
- Travis Scrimshaw (2012-05-11): Fixed bug in inner where if the length was longer than the length of the inner partition, it would include 0's.
- Andrew Mathas (2012-06-01): Removed deprecated functions and added compatibility with the PartitionTuple classes. See Issue #13072
- Travis Scrimshaw (2012-10-12): Added options. Made `Partition_class` to the element `Partition`. `Partitions*` are now all in the category framework except `PartitionsRestricted` (which will eventually be removed). Cleaned up documentation.
- Matthew Lancellotti (2018-09-14): Added a bunch of "k" methods to Partition.

EXAMPLES:

There are 5 partitions of the integer 4:

```
sage: Partitions(4).cardinality()                                               #␣
↪needs sage.libs.flint
5
sage: Partitions(4).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4)).cardinality()                                        ␣
↪     # needs sage.libs.flint
5
>>> Partitions(Integer(4)).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

We can use the method `.first()` to get the 'first' partition of a number:

```
sage: Partitions(4).first()
[4]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4)).first()
[4]
```

Using the method `.next(p)`, we can calculate the 'next' partition after p. When we are at the last partition, `None` will be returned:

```
sage: Partitions(4).next([4])
[3, 1]
sage: Partitions(4).next([1,1,1,1]) is None
True
```

```
>>> from sage.all import *
>>> Partitions(Integer(4)).next([Integer(4)])
[3, 1]
>>> Partitions(Integer(4)).next([Integer(1),Integer(1),Integer(1),Integer(1)]) is None
True
```

We can use `iter` to get an object which iterates over the partitions one by one to save memory. Note that when we do something like `for part in Partitions(4)` this iterator is used in the background:

```
sage: g = iter(Partitions(4))
sage: next(g)
[4]
sage: next(g)
[3, 1]
sage: next(g)
[2, 2]
sage: for p in Partitions(4): print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> g = iter(Partitions(Integer(4)))
>>> next(g)
[4]
>>> next(g)
[3, 1]
>>> next(g)
[2, 2]
>>> for p in Partitions(Integer(4)): print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
```

We can add constraints to the type of partitions we want. For example, to get all of the partitions of 4 of length 2, we'd do the following:

```
sage: Partitions(4, length=2).list()
[[3, 1], [2, 2]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), length=Integer(2)).list()
[[3, 1], [2, 2]]
```

Here is the list of partitions of length at least 2 and the list of ones with length at most 2:

```
sage: Partitions(4, min_length=2).list()
[[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: Partitions(4, max_length=2).list()
[[4], [3, 1], [2, 2]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), min_length=Integer(2)).list()
[[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
>>> Partitions(Integer(4), max_length=Integer(2)).list()
[[4], [3, 1], [2, 2]]
```

The options `min_part` and `max_part` can be used to set constraints on the sizes of all parts. Using `max_part`, we can select partitions having only 'small' entries. The following is the list of the partitions of 4 with parts at most 2:

```
sage: Partitions(4, max_part=2).list()
[[2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), max_part=Integer(2)).list()
[[2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

The `min_part` options is complementary to `max_part` and selects partitions having only 'large' parts. Here is the list of all partitions of 4 with each part at least 2:

```
sage: Partitions(4, min_part=2).list()
[[4], [2, 2]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), min_part=Integer(2)).list()
[[4], [2, 2]]
```

The options `inner` and `outer` can be used to set part-by-part constraints. This is the list of partitions of 4 with `[3, 1, 1]` as an outer bound (that is, partitions of 4 contained in the partition `[3, 1, 1]`):

```
sage: Partitions(4, outer=[3,1,1]).list()
[[3, 1], [2, 1, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), outer=[Integer(3),Integer(1),Integer(1)]).list()
[[3, 1], [2, 1, 1]]
```

`outer` sets `max_length` to the length of its argument. Moreover, the parts of `outer` may be infinite to clear constraints on specific parts. Here is the list of the partitions of 4 of length at most 3 such that the second and third part are 1 when they exist:

```
sage: Partitions(4, outer=[oo,1,1]).list()
[[4], [3, 1], [2, 1, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), outer=[oo,Integer(1),Integer(1)]).list()
[[4], [3, 1], [2, 1, 1]]
```

Finally, here are the partitions of 4 with `[1,1,1]` as an inner bound (i. e., the partitions of 4 containing the partition `[1,1,1]`). Note that `inner` sets `min_length` to the length of its argument:

```
sage: Partitions(4, inner=[1,1,1]).list()
[[2, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), inner=[Integer(1),Integer(1),Integer(1)]).list()
[[2, 1, 1], [1, 1, 1, 1]]
```

The options `min_slope` and `max_slope` can be used to set constraints on the slope, that is on the difference `p[i+1]-p[i]` of two consecutive parts. Here is the list of the strictly decreasing partitions of 4:

```
sage: Partitions(4, max_slope=-1).list()
[[4], [3, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), max_slope=-Integer(1)).list()
[[4], [3, 1]]
```

The constraints can be combined together in all reasonable ways. Here are all the partitions of 11 of length between 2 and 4 such that the difference between two consecutive parts is between -3 and -1:

```
sage: Partitions(11, min_slope=-3, max_slope=-1, min_length=2, max_length=4).list()
[[7, 4], [6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2], [5, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(11), min_slope=-Integer(3), max_slope=-Integer(1), min_
↪length=Integer(2), max_length=Integer(4)).list()
[[7, 4], [6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2], [5, 3, 2, 1]]
```

Partition objects can also be created individually with *Partition*:

```
sage: Partition([2,1])
[2, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(1)])
[2, 1]
```

Once we have a partition object, then there are a variety of methods that we can use. For example, we can get the conjugate of a partition. Geometrically, the conjugate of a partition is the reflection of that partition through its main diagonal. Of course, this operation is an involution:

```
sage: Partition([4,1]).conjugate()
[2, 1, 1, 1]
sage: Partition([4,1]).conjugate().conjugate()
[4, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(1)]).conjugate()
[2, 1, 1, 1]
>>> Partition([Integer(4),Integer(1)]).conjugate().conjugate()
[4, 1]
```

If we create a partition with extra zeros at the end, they will be dropped:

```
sage: Partition([4,1,0,0])
[4, 1]
sage: Partition([0])
[]
sage: Partition([0,0])
[]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(1),Integer(0),Integer(0)])
[4, 1]
>>> Partition([Integer(0)])
[]
>>> Partition([Integer(0),Integer(0)])
[]
```

The idea of a partition being followed by infinitely many parts of size 0 is consistent with the `get_part` method:

```
sage: p = Partition([5, 2])
sage: p.get_part(0)
5
sage: p.get_part(10)
0
```

```
>>> from sage.all import *
>>> p = Partition([Integer(5), Integer(2)])
>>> p.get_part(Integer(0))
5
>>> p.get_part(Integer(10))
0
```

We can go back and forth between the standard and the exponential notations of a partition. The exponential notation can be padded with extra zeros:

```
sage: Partition([6,4,4,2,1]).to_exp()
[1, 1, 0, 2, 0, 1]
sage: Partition(exp=[1,1,0,2,0,1])
[6, 4, 4, 2, 1]
sage: Partition([6,4,4,2,1]).to_exp(5)
[1, 1, 0, 2, 0, 1]
```

(continues on next page)

(continued from previous page)

```
sage: Partition([6,4,4,2,1]).to_exp(7)
[1, 1, 0, 2, 0, 1, 0]
sage: Partition([6,4,4,2,1]).to_exp(10)
[1, 1, 0, 2, 0, 1, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> Partition([Integer(6),Integer(4),Integer(4),Integer(2),Integer(1)]).to_exp()
[1, 1, 0, 2, 0, 1]
>>> Partition(exp=[Integer(1),Integer(1),Integer(0),Integer(2),Integer(0),Integer(1)])
[6, 4, 4, 2, 1]
>>> Partition([Integer(6),Integer(4),Integer(4),Integer(2),Integer(1)]).to_
↪exp(Integer(5))
[1, 1, 0, 2, 0, 1]
>>> Partition([Integer(6),Integer(4),Integer(4),Integer(2),Integer(1)]).to_
↪exp(Integer(7))
[1, 1, 0, 2, 0, 1, 0]
>>> Partition([Integer(6),Integer(4),Integer(4),Integer(2),Integer(1)]).to_
↪exp(Integer(10))
[1, 1, 0, 2, 0, 1, 0, 0, 0, 0]
```

We can get the (zero-based!) coordinates of the corners of a partition:

```
sage: Partition([4,3,1]).corners()
[(0, 3), (1, 2), (2, 0)]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(1)]).corners()
[(0, 3), (1, 2), (2, 0)]
```

We can compute the core and quotient of a partition and build the partition back up from them:

```
sage: Partition([6,3,2,2]).core(3)
[2, 1, 1]
sage: Partition([7,7,5,3,3,3,1]).quotient(3)
([2], [1], [2, 2, 2])
sage: p = Partition([11,5,5,3,2,2,2])
sage: p.core(3)
[]
sage: p.quotient(3)
([2, 1], [4], [1, 1, 1])
sage: Partition(core=[],quotient=([2, 1], [4], [1, 1, 1]))
[11, 5, 5, 3, 2, 2, 2]
```

```
>>> from sage.all import *
>>> Partition([Integer(6),Integer(3),Integer(2),Integer(2)]).core(Integer(3))
[2, 1, 1]
>>> Partition([Integer(7),Integer(7),Integer(5),Integer(3),Integer(3),Integer(3),
↪Integer(1)]).quotient(Integer(3))
([2], [1], [2, 2, 2])
>>> p = Partition([Integer(11),Integer(5),Integer(5),Integer(3),Integer(2),Integer(2),
↪Integer(2)])
>>> p.core(Integer(3))
```

(continues on next page)

(continued from previous page)

```
[]
>>> p.quotient(Integer(3))
([2, 1], [4], [1, 1, 1])
>>> Partition(core=[],quotient=([Integer(2), Integer(1)], [Integer(4)], [Integer(1),␣
↪Integer(1), Integer(1)]))
[11, 5, 5, 3, 2, 2, 2]
```

We can compute the $0 - 1$ sequence and go back and forth:

```
sage: Partitions().from_zero_one([1, 1, 1, 1, 0, 1, 0])
[5, 4]
sage: all(Partitions().from_zero_one(mu.zero_one_sequence())
....:     == mu for n in range(5) for mu in Partitions(n))
True
```

```
>>> from sage.all import *
>>> Partitions().from_zero_one([Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(1), Integer(0)])
[5, 4]
>>> all(Partitions().from_zero_one(mu.zero_one_sequence())
...     == mu for n in range(Integer(5)) for mu in Partitions(n))
True
```

We can compute the Frobenius coordinates and go back and forth:

```
sage: Partition([7,3,1]).frobenius_coordinates()
([6, 1], [2, 0])
sage: Partition(frobenius_coordinates=([6,1],[2,0]))
[7, 3, 1]
sage: all(mu == Partition(frobenius_coordinates=mu.frobenius_coordinates())
....:     for n in range(12) for mu in Partitions(n))
True
```

```
>>> from sage.all import *
>>> Partition([Integer(7),Integer(3),Integer(1)]).frobenius_coordinates()
([6, 1], [2, 0])
>>> Partition(frobenius_coordinates=([Integer(6),Integer(1)],[Integer(2),Integer(0)]))
[7, 3, 1]
>>> all(mu == Partition(frobenius_coordinates=mu.frobenius_coordinates())
...     for n in range(Integer(12)) for mu in Partitions(n))
True
```

We use the lexicographic ordering:

```
sage: pl = Partition([4,1,1])
sage: ql = Partitions()([3,3])
sage: pl > ql
True
sage: PL = Partitions()
sage: pl = PL([4,1,1])
sage: ql = PL([3,3])
sage: pl > ql
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> pl = Partition([Integer(4),Integer(1),Integer(1)])
>>> ql = Partitions()([Integer(3),Integer(3)])
>>> pl > ql
True
>>> PL = Partitions()
>>> pl = PL([Integer(4),Integer(1),Integer(1)])
>>> ql = PL([Integer(3),Integer(3)])
>>> pl > ql
True
```

class sage.combinat.partition.**OrderedPartitions**(*n*, *k*)

Bases: *Partitions*

The class of ordered partitions of n. If k is specified, then this contains only the ordered partitions of length k.

An *ordered partition* of a nonnegative integer n means a list of positive integers whose sum is n. This is the same as a composition of n.

> **Note**
>
> It is recommended that you use *Compositions()* instead as *OrderedPartitions()* wraps GAP.

EXAMPLES:

```
sage: OrderedPartitions(3)
Ordered partitions of 3
sage: OrderedPartitions(3).list()                                              #␣
↪needs sage.libs.gap
[[3], [2, 1], [1, 2], [1, 1, 1]]
sage: OrderedPartitions(3,2)
Ordered partitions of 3 of length 2
sage: OrderedPartitions(3,2).list()                                            #␣
↪needs sage.libs.gap
[[2, 1], [1, 2]]

sage: OrderedPartitions(10, k=2).list()                                        #␣
↪needs sage.libs.gap
[[9, 1], [8, 2], [7, 3], [6, 4], [5, 5], [4, 6], [3, 7], [2, 8], [1, 9]]
sage: OrderedPartitions(4).list()                                              #␣
↪needs sage.libs.gap
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> OrderedPartitions(Integer(3))
Ordered partitions of 3
>>> OrderedPartitions(Integer(3)).list()                                       ␣
↪     # needs sage.libs.gap
[[3], [2, 1], [1, 2], [1, 1, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> OrderedPartitions(Integer(3),Integer(2))
Ordered partitions of 3 of length 2
>>> OrderedPartitions(Integer(3),Integer(2)).list()                          ␣
↪              # needs sage.libs.gap
[[2, 1], [1, 2]]

>>> OrderedPartitions(Integer(10), k=Integer(2)).list()                      ␣
↪              # needs sage.libs.gap
[[9, 1], [8, 2], [7, 3], [6, 4], [5, 5], [4, 6], [3, 7], [2, 8], [1, 9]]
>>> OrderedPartitions(Integer(4)).list()                                     ␣
↪     # needs sage.libs.gap
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: OrderedPartitions(3).cardinality()
4
sage: OrderedPartitions(3,2).cardinality()
2
sage: OrderedPartitions(10,2).cardinality()
9
sage: OrderedPartitions(15).cardinality()
16384
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> OrderedPartitions(Integer(3)).cardinality()
4
>>> OrderedPartitions(Integer(3),Integer(2)).cardinality()
2
>>> OrderedPartitions(Integer(10),Integer(2)).cardinality()
9
>>> OrderedPartitions(Integer(15)).cardinality()
16384
```

list()

Return a list of partitions in `self`.

EXAMPLES:

```
sage: OrderedPartitions(3).list()                                            #␣
↪needs sage.libs.gap
[[3], [2, 1], [1, 2], [1, 1, 1]]
sage: OrderedPartitions(3,2).list()                                          #␣
↪needs sage.libs.gap
[[2, 1], [1, 2]]
```

```
>>> from sage.all import *
>>> OrderedPartitions(Integer(3)).list()                                     ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.libs.gap
[[3], [2, 1], [1, 2], [1, 1, 1]]
>>> OrderedPartitions(Integer(3),Integer(2)).list()                                    ␣
↪              # needs sage.libs.gap
[[2, 1], [1, 2]]
```

class sage.combinat.partition.**Partition**(*parent*, *mu*)

Bases: `CombinatorialElement`

A partition p of a nonnegative integer n is a non-increasing list of positive integers (the *parts* of the partition) with total sum n.

A partition is often represented as a diagram consisting of **cells**, or **boxes**, placed in rows on top of each other such that the number of cells in the i-th row, reading from top to bottom, is the i-th part of the partition. The rows are left-justified (and become shorter and shorter the farther down one goes). This diagram is called the **Young diagram** of the partition, or more precisely its Young diagram in English notation. (French and Russian notations are variations on this representation.)

The coordinate system related to a partition applies from the top to the bottom and from left to right. So, the corners of the partition `[5, 3, 1]` are `[[0,4], [1,2], [2,0]]`.

For display options, see `Partitions.options()`.

Note

Partitions are 0 based with coordinates in the form of (row-index, column-index). For example consider the partition `mu=Partition([4,3,2,2])`, the first part is `mu[0]` (which is 4), the second is `mu[1]`, and so on, and the upper-left cell in English convention is `(0, 0)`.

A partition can be specified in one of the following ways:

- a list (the default)
- using exponential notation
- by Frobenius coordinates
- specifying its $0 - 1$ sequence
- specifying the core and the quotient

See the examples below.

EXAMPLES:

Creating partitions though parents:

```
sage: mu = Partitions(8)([3,2,1,1,1]); mu
[3, 2, 1, 1, 1]
sage: nu = Partition([3,2,1,1,1]); nu
[3, 2, 1, 1, 1]
sage: mu == nu
True
sage: mu is nu
False
sage: mu in Partitions()
True
```

(continues on next page)

(continued from previous page)

```
sage: mu.parent()
Partitions of the integer 8
sage: mu.size()
8
sage: mu.category()
Category of elements of Partitions of the integer 8
sage: nu.parent()
Partitions
sage: nu.category()
Category of elements of Partitions
sage: mu[0]
3
sage: mu[1]
2
sage: mu[2]
1
sage: mu.pp()
***
**
*
*
*
sage: mu.removable_cells()
[(0, 2), (1, 1), (4, 0)]
sage: mu.down_list()
[[2, 2, 1, 1, 1], [3, 1, 1, 1, 1], [3, 2, 1, 1]]
sage: mu.addable_cells()
[(0, 3), (1, 2), (2, 1), (5, 0)]
sage: mu.up_list()
[[4, 2, 1, 1, 1], [3, 3, 1, 1, 1], [3, 2, 2, 1, 1], [3, 2, 1, 1, 1, 1]]
sage: mu.conjugate()
[5, 2, 1]
sage: mu.dominates(nu)
True
sage: nu.dominates(mu)
True
```

```
>>> from sage.all import *
>>> mu = Partitions(Integer(8))([Integer(3),Integer(2),Integer(1),Integer(1),
↪Integer(1)]); mu
[3, 2, 1, 1, 1]
>>> nu = Partition([Integer(3),Integer(2),Integer(1),Integer(1),Integer(1)]); nu
[3, 2, 1, 1, 1]
>>> mu == nu
True
>>> mu is nu
False
>>> mu in Partitions()
True
>>> mu.parent()
Partitions of the integer 8
>>> mu.size()
```

(continues on next page)

(continued from previous page)

```
8
>>> mu.category()
Category of elements of Partitions of the integer 8
>>> nu.parent()
Partitions
>>> nu.category()
Category of elements of Partitions
>>> mu[Integer(0)]
3
>>> mu[Integer(1)]
2
>>> mu[Integer(2)]
1
>>> mu.pp()
***
**
*
*
*
>>> mu.removable_cells()
[(0, 2), (1, 1), (4, 0)]
>>> mu.down_list()
[[2, 2, 1, 1, 1], [3, 1, 1, 1, 1], [3, 2, 1, 1]]
>>> mu.addable_cells()
[(0, 3), (1, 2), (2, 1), (5, 0)]
>>> mu.up_list()
[[4, 2, 1, 1, 1], [3, 3, 1, 1, 1], [3, 2, 2, 1, 1], [3, 2, 1, 1, 1, 1]]
>>> mu.conjugate()
[5, 2, 1]
>>> mu.dominates(nu)
True
>>> nu.dominates(mu)
True
```

Creating partitions using `Partition`:

```
sage: Partition([3,2,1])
[3, 2, 1]
sage: Partition(exp=[2,1,1])
[3, 2, 1, 1]
sage: Partition(core=[2,1], quotient=[[2,1],[3],[1,1,1]])
[11, 5, 5, 3, 2, 2, 2]
sage: Partition(frobenius_coordinates=([3,2],[4,0]))
[4, 4, 1, 1, 1]
sage: Partitions().from_zero_one([1, 1, 1, 1, 0, 1, 0])
[5, 4]
sage: [2,1] in Partitions()
True
sage: [2,1,0] in Partitions()
True
sage: Partition([1,2,3])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: [1, 2, 3] is not an element of Partitions
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)])
[3, 2, 1]
>>> Partition(exp=[Integer(2),Integer(1),Integer(1)])
[3, 2, 1, 1]
>>> Partition(core=[Integer(2),Integer(1)], quotient=[[Integer(2),Integer(1)],
↪[Integer(3)],[Integer(1),Integer(1),Integer(1)]])
[11, 5, 5, 3, 2, 2, 2]
>>> Partition(frobenius_coordinates=([Integer(3),Integer(2)],[Integer(4),
↪Integer(0)]))
[4, 4, 1, 1, 1]
>>> Partitions().from_zero_one([Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(1), Integer(0)])
[5, 4]
>>> [Integer(2),Integer(1)] in Partitions()
True
>>> [Integer(2),Integer(1),Integer(0)] in Partitions()
True
>>> Partition([Integer(1),Integer(2),Integer(3)])
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not an element of Partitions
```

Sage ignores trailing zeros at the end of partitions:

```
sage: Partition([3,2,1,0])
[3, 2, 1]
sage: Partitions()([3,2,1,0])
[3, 2, 1]
sage: Partitions(6)([3,2,1,0])
[3, 2, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1),Integer(0)])
[3, 2, 1]
>>> Partitions()([Integer(3),Integer(2),Integer(1),Integer(0)])
[3, 2, 1]
>>> Partitions(Integer(6))([Integer(3),Integer(2),Integer(1),Integer(0)])
[3, 2, 1]
```

add_cell(*i*, *j=None*)

Return a partition corresponding to `self` with a cell added in row `i`. (This does not change `self`.)

EXAMPLES:

```
sage: Partition([3, 2, 1, 1]).add_cell(0)
[4, 2, 1, 1]
sage: cell = [4, 0]; Partition([3, 2, 1, 1]).add_cell(*cell)
[3, 2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(3), Integer(2), Integer(1), Integer(1)]).add_
↪cell(Integer(0))
[4, 2, 1, 1]
>>> cell = [Integer(4), Integer(0)]; Partition([Integer(3), Integer(2),␣
↪Integer(1), Integer(1)]).add_cell(*cell)
[3, 2, 1, 1, 1]
```

add_horizontal_border_strip(*k*)

Return a list of all the partitions that can be obtained by adding a horizontal border strip of length `k` to `self`.

EXAMPLES:

```
sage: Partition([]).add_horizontal_border_strip(0)
[[]]
sage: Partition([3,2,1]).add_horizontal_border_strip(0)
[[3, 2, 1]]
sage: Partition([]).add_horizontal_border_strip(2)
[[2]]
sage: Partition([2,2]).add_horizontal_border_strip(2)
[[4, 2], [3, 2, 1], [2, 2, 2]]
sage: Partition([3,2,2]).add_horizontal_border_strip(2)
[[5, 2, 2], [4, 3, 2], [4, 2, 2, 1], [3, 3, 2, 1], [3, 2, 2, 2]]
```

```
>>> from sage.all import *
>>> Partition([]).add_horizontal_border_strip(Integer(0))
[[]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).add_horizontal_border_
↪strip(Integer(0))
[[3, 2, 1]]
>>> Partition([]).add_horizontal_border_strip(Integer(2))
[[2]]
>>> Partition([Integer(2),Integer(2)]).add_horizontal_border_strip(Integer(2))
[[4, 2], [3, 2, 1], [2, 2, 2]]
>>> Partition([Integer(3),Integer(2),Integer(2)]).add_horizontal_border_
↪strip(Integer(2))
[[5, 2, 2], [4, 3, 2], [4, 2, 2, 1], [3, 3, 2, 1], [3, 2, 2, 2]]
```

add_vertical_border_strip(*k*)

Return a list of all the partitions that can be obtained by adding a vertical border strip of length `k` to `self`.

EXAMPLES:

```
sage: Partition([]).add_vertical_border_strip(0)
[[]]
sage: Partition([3,2,1]).add_vertical_border_strip(0)
[[3, 2, 1]]
sage: Partition([]).add_vertical_border_strip(2)
[[1, 1]]
sage: Partition([2,2]).add_vertical_border_strip(2)
[[3, 3], [3, 2, 1], [2, 2, 1, 1]]
sage: Partition([3,2,2]).add_vertical_border_strip(2)
[[4, 3, 2], [4, 2, 2, 1], [3, 3, 3], [3, 3, 2, 1], [3, 2, 2, 1, 1]]
```

```
>>> from sage.all import *
>>> Partition([]).add_vertical_border_strip(Integer(0))
[[]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).add_vertical_border_
↪strip(Integer(0))
[[3, 2, 1]]
>>> Partition([]).add_vertical_border_strip(Integer(2))
[[1, 1]]
>>> Partition([Integer(2),Integer(2)]).add_vertical_border_strip(Integer(2))
[[3, 3], [3, 2, 1], [2, 2, 1, 1]]
>>> Partition([Integer(3),Integer(2),Integer(2)]).add_vertical_border_
↪strip(Integer(2))
[[4, 3, 2], [4, 2, 2, 1], [3, 3, 3], [3, 3, 2, 1], [3, 2, 2, 1, 1]]
```

addable_cells()

Return a list of the outside corners of the partition `self`.

An outside corner (also called a cocorner) of a partition λ is a cell on $\mathbf{Z}^2$ which does not belong to the Young diagram of λ but can be added to this Young diagram to still form a straight-shape Young diagram.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

Note

These are called "outer corners" in [Sag2001].

EXAMPLES:

```
sage: Partition([2,2,1]).outside_corners()
[(0, 2), (2, 1), (3, 0)]
sage: Partition([2,2]).outside_corners()
[(0, 2), (2, 0)]
sage: Partition([6,3,3,1,1,1]).outside_corners()
[(0, 6), (1, 3), (3, 1), (6, 0)]
sage: Partition([]).outside_corners()
[(0, 0)]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).outside_corners()
[(0, 2), (2, 1), (3, 0)]
>>> Partition([Integer(2),Integer(2)]).outside_corners()
[(0, 2), (2, 0)]
>>> Partition([Integer(6),Integer(3),Integer(3),Integer(1),Integer(1),
↪Integer(1)]).outside_corners()
[(0, 6), (1, 3), (3, 1), (6, 0)]
>>> Partition([]).outside_corners()
[(0, 0)]
```

addable_cells_residue(i, l)

Return a list of the outside corners of the partition `self` having `l`-residue `i`.

An outside corner (also called a cocorner) of a partition λ is a cell on $\mathbf{Z}^2$ which does not belong to the Young diagram of λ but can be added to this Young diagram to still form a straight-shape Young diagram. See `residue()` for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).outside_corners_residue(0, 3)
[(0, 3), (3, 0)]
sage: Partition([3,2,1]).outside_corners_residue(1, 3)
[(1, 2)]
sage: Partition([3,2,1]).outside_corners_residue(2, 3)
[(2, 1)]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).outside_corners_
→residue(Integer(0), Integer(3))
[(0, 3), (3, 0)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).outside_corners_
→residue(Integer(1), Integer(3))
[(1, 2)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).outside_corners_
→residue(Integer(2), Integer(3))
[(2, 1)]
```

arm_cells(i, j)

Return the list of the cells of the arm of cell (i, j) in `self`.

The arm of cell $c = (i, j)$ is the boxes that appear to the right of c.

The cell coordinates are zero-based, i. e., the northwesternmost cell is $(0, 0)$.

INPUT:

- `i, j` – two integers

OUTPUT: list of pairs of integers

EXAMPLES:

```
sage: Partition([4,4,3,1]).arm_cells(1,1)
[(1, 2), (1, 3)]

sage: Partition([]).arm_cells(0,0)
Traceback (most recent call last):
...
ValueError: the cell is not in the diagram
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(4),Integer(3),Integer(1)]).arm_
→cells(Integer(1),Integer(1))
[(1, 2), (1, 3)]

>>> Partition([]).arm_cells(Integer(0),Integer(0))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: the cell is not in the diagram
```

arm_length(*i*, *j*)

Return the length of the arm of cell (i, j) in `self`.

The arm of cell (i, j) is the cells that appear to the right of cell (i, j).

The cell coordinates are zero-based, i. e., the northwesternmost cell is $(0, 0)$.

INPUT:

- `i, j` – two integers

OUTPUT: integer or a `ValueError`

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.arm_length(0, 0)
1
sage: p.arm_length(0, 1)
0
sage: p.arm_length(2, 0)
0
sage: Partition([3,3]).arm_length(0, 0)
2
sage: Partition([3,3]).arm_length(*[0,0])
2
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(2),Integer(1)])
>>> p.arm_length(Integer(0), Integer(0))
1
>>> p.arm_length(Integer(0), Integer(1))
0
>>> p.arm_length(Integer(2), Integer(0))
0
>>> Partition([Integer(3),Integer(3)]).arm_length(Integer(0), Integer(0))
2
>>> Partition([Integer(3),Integer(3)]).arm_length(*[Integer(0),Integer(0)])
2
```

arm_lengths(*flat=False*)

Return a tableau of shape `self` where each cell is filled with its arm length.

The optional boolean parameter `flat` provides the option of returning a flat list.

EXAMPLES:

```
sage: Partition([2,2,1]).arm_lengths()
[[1, 0], [1, 0], [0]]
sage: Partition([2,2,1]).arm_lengths(flat=True)
[1, 0, 1, 0, 0]
sage: Partition([3,3]).arm_lengths()
[[2, 1, 0], [2, 1, 0]]
```

(continues on next page)

(continued from previous page)

```
sage: Partition([3,3]).arm_lengths(flat=True)
[2, 1, 0, 2, 1, 0]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).arm_lengths()
[[1, 0], [1, 0], [0]]
>>> Partition([Integer(2),Integer(2),Integer(1)]).arm_lengths(flat=True)
[1, 0, 1, 0, 0]
>>> Partition([Integer(3),Integer(3)]).arm_lengths()
[[2, 1, 0], [2, 1, 0]]
>>> Partition([Integer(3),Integer(3)]).arm_lengths(flat=True)
[2, 1, 0, 2, 1, 0]
```

arms_legs_coeff (i, j)

This is a statistic on a cell $c = (i, j)$ in the diagram of partition p given by

$$\frac{1 - q^a \cdot t^{\ell+1}}{1 - q^{a+1} \cdot t^{\ell}}$$

where a is the arm length of c and ℓ is the leg length of c.

The coordinates `i` and `j` of the cell are understood to be 0-based, so that `(0, 0)` is the northwesternmost cell (in English notation).

EXAMPLES:

```
sage: Partition([3,2,1]).arms_legs_coeff(1,1)
(-t + 1)/(-q + 1)
sage: Partition([3,2,1]).arms_legs_coeff(0,0)
(-q^2*t^3 + 1)/(-q^3*t^2 + 1)
sage: Partition([3,2,1]).arms_legs_coeff(*[0,0])
(-q^2*t^3 + 1)/(-q^3*t^2 + 1)
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).arms_legs_coeff(Integer(1),
↪Integer(1))
(-t + 1)/(-q + 1)
>>> Partition([Integer(3),Integer(2),Integer(1)]).arms_legs_coeff(Integer(0),
↪Integer(0))
(-q^2*t^3 + 1)/(-q^3*t^2 + 1)
>>> Partition([Integer(3),Integer(2),Integer(1)]).arms_legs_
↪coeff(*[Integer(0),Integer(0)])
(-q^2*t^3 + 1)/(-q^3*t^2 + 1)
```

atom ()

Return a list of the standard tableaux of size `self.size()` whose atom is equal to `self`.

EXAMPLES:

```
sage: Partition([2,1]).atom()
[[[1, 2], [3]]]
sage: Partition([3,2,1]).atom()
[[[1, 2, 3, 6], [4, 5]], [[1, 2, 3], [4, 5], [6]]]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(1)]).atom()
[[[1, 2], [3]]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).atom()
[[[1, 2, 3, 6], [4, 5]], [[1, 2, 3], [4, 5], [6]]]
```

attacking_pairs()

Return a list of the attacking pairs of the Young diagram of `self`.

A pair of cells (c, d) of a Young diagram (in English notation) is said to be attacking if one of the following conditions holds:

1. c and d lie in the same row with c strictly to the west of d.
2. c is in the row immediately to the south of d, and c lies strictly east of d.

This particular method returns each pair (c, d) as a tuple, where each of c and d is given as a tuple (i, j) with i and j zero-based (so $i = 0$ means that the cell lies in the topmost row).

EXAMPLES:

```
sage: p = Partition([3, 2])
sage: p.attacking_pairs()
[((0, 0), (0, 1)),
 ((0, 0), (0, 2)),
 ((0, 1), (0, 2)),
 ((1, 0), (1, 1)),
 ((1, 1), (0, 0))]
sage: Partition([]).attacking_pairs()
[]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(3), Integer(2)])
>>> p.attacking_pairs()
[((0, 0), (0, 1)),
 ((0, 0), (0, 2)),
 ((0, 1), (0, 2)),
 ((1, 0), (1, 1)),
 ((1, 1), (0, 0))]
>>> Partition([]).attacking_pairs()
[]
```

aut(*t=0*, *q=0*)

Return the size of the centralizer of any permutation of cycle type `self`.

If m_i is the multiplicity of i as a part of p, this is given by

$$\prod_i m_i! i^{m_i}.$$

Including the optional parameters t and q gives the q, t analog, which is the former product times

$$\prod_{i=1}^{\mathrm{length}(p)} \frac{1 - q^{p_i}}{1 - t^{p_i}}.$$

See Section 1.3, p. 24, in [Ke1991].

EXAMPLES:

```
sage: Partition([2,2,1]).centralizer_size()
8
sage: Partition([2,2,2]).centralizer_size()
48
sage: Partition([2,2,1]).centralizer_size(q=2, t=3)
9/16
sage: Partition([]).centralizer_size()
1
sage: Partition([]).centralizer_size(q=2, t=4)
1
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).centralizer_size()
8
>>> Partition([Integer(2),Integer(2),Integer(2)]).centralizer_size()
48
>>> Partition([Integer(2),Integer(2),Integer(1)]).centralizer_
↪size(q=Integer(2), t=Integer(3))
9/16
>>> Partition([]).centralizer_size()
1
>>> Partition([]).centralizer_size(q=Integer(2), t=Integer(4))
1
```

beta_numbers(*length=None*)

Return the set of beta numbers corresponding to `self`.

The optional argument `length` specifies the length of the beta set (which must be at least the length of `self`).

For more on beta numbers, see *frobenius_coordinates()*.

EXAMPLES:

```
sage: Partition([4,3,2]).beta_numbers()
[6, 4, 2]
sage: Partition([4,3,2]).beta_numbers(5)
[8, 6, 4, 1, 0]
sage: Partition([]).beta_numbers()
[]
sage: Partition([]).beta_numbers(3)
[2, 1, 0]
sage: Partition([6,4,1,1]).beta_numbers()
[9, 6, 2, 1]
sage: Partition([6,4,1,1]).beta_numbers(6)
[11, 8, 4, 3, 1, 0]
sage: Partition([1,1,1]).beta_numbers()
[3, 2, 1]
sage: Partition([1,1,1]).beta_numbers(4)
[4, 3, 2, 0]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(2)]).beta_numbers()
[6, 4, 2]
>>> Partition([Integer(4),Integer(3),Integer(2)]).beta_numbers(Integer(5))
```

(continues on next page)

(continued from previous page)

```
[8, 6, 4, 1, 0]
>>> Partition([]).beta_numbers()
[]
>>> Partition([]).beta_numbers(Integer(3))
[2, 1, 0]
>>> Partition([Integer(6),Integer(4),Integer(1),Integer(1)]).beta_numbers()
[9, 6, 2, 1]
>>> Partition([Integer(6),Integer(4),Integer(1),Integer(1)]).beta_
→numbers(Integer(6))
[11, 8, 4, 3, 1, 0]
>>> Partition([Integer(1),Integer(1),Integer(1)]).beta_numbers()
[3, 2, 1]
>>> Partition([Integer(1),Integer(1),Integer(1)]).beta_numbers(Integer(4))
[4, 3, 2, 0]
```

block (*e*, *multicharge=(0,)*)

Return a dictionary β that determines the block associated to the partition `self` and the *quantum_characteristic()* `e`.

INPUT:

- `e` – the quantum characteristic
- `multicharge` – the multicharge (default: $(0,)$)

OUTPUT:

- A dictionary giving the multiplicities of the residues in the partition tuple `self`

In more detail, the value `beta[i]` is equal to the number of nodes of residue `i`. This corresponds to the positive root

$$\sum_{i\in I}\beta_i\alpha_i \in Q^+,$$

a element of the positive root lattice of the corresponding Kac-Moody algebra. See [DJM1998] and [BK2009] for more details.

This is a useful statistics because two Specht modules for a Hecke algebra of type A belong to the same block if and only if they correspond to same element β of the root lattice, given above.

We return a dictionary because when the quantum characteristic is 0, the Cartan type is A_∞, in which case the simple roots are indexed by the integers.

EXAMPLES:

```
sage: Partition([4,3,2]).block(0)
{-2: 1, -1: 2, 0: 2, 1: 2, 2: 1, 3: 1}
sage: Partition([4,3,2]).block(2)
{0: 4, 1: 5}
sage: Partition([4,3,2]).block(2, multicharge=(1,))
{0: 5, 1: 4}
sage: Partition([4,3,2]).block(3)
{0: 3, 1: 3, 2: 3}
sage: Partition([4,3,2]).block(4)
{0: 2, 1: 2, 2: 2, 3: 3}
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(2)]).block(Integer(0))
{-2: 1, -1: 2, 0: 2, 1: 2, 2: 1, 3: 1}
>>> Partition([Integer(4),Integer(3),Integer(2)]).block(Integer(2))
{0: 4, 1: 5}
>>> Partition([Integer(4),Integer(3),Integer(2)]).block(Integer(2),␣
↪multicharge=(Integer(1),))
{0: 5, 1: 4}
>>> Partition([Integer(4),Integer(3),Integer(2)]).block(Integer(3))
{0: 3, 1: 3, 2: 3}
>>> Partition([Integer(4),Integer(3),Integer(2)]).block(Integer(4))
{0: 2, 1: 2, 2: 2, 3: 3}
```

boundary()

Return the integer coordinates of points on the boundary of `self`.

For the following description, picture the Ferrer's diagram of `self` using the French convention. Recall that the French convention puts the longest row on the bottom and the shortest row on the top. In addition, interpret the Ferrer's diagram as 1 x 1 cells in the Euclidean plane. So if `self` was the partition [3, 1], the lower-left vertices of the 1 x 1 cells in the Ferrer's diagram would be (0, 0), (1, 0), (2, 0), and (0, 1).

The boundary of a partition is the set $\{\mathrm{NE}(d) \mid \forall d \text{ diagonal}\}$. That is, for every diagonal line $y = x + b$ where $b \in \mathbb{Z}$, we find the northeasternmost (NE) point on that diagonal which is also in the Ferrer's diagram.

The boundary will go from bottom-right to top-left.

EXAMPLES:

Consider the partition (1) depicted as a square on a cartesian plane with vertices (0, 0), (1, 0), (1, 1), and (0, 1). Three of those vertices in the appropriate order form the boundary:

```
sage: Partition([1]).boundary()
[(1, 0), (1, 1), (0, 1)]
```

```
>>> from sage.all import *
>>> Partition([Integer(1)]).boundary()
[(1, 0), (1, 1), (0, 1)]
```

The partition (3, 1) can be visualized as three squares on a cartesian plane. The coordinates of the appropriate vertices form the boundary:

```
sage: Partition([3, 1]).boundary()
[(3, 0), (3, 1), (2, 1), (1, 1), (1, 2), (0, 2)]
```

```
>>> from sage.all import *
>>> Partition([Integer(3), Integer(1)]).boundary()
[(3, 0), (3, 1), (2, 1), (1, 1), (1, 2), (0, 2)]
```

See also

k_rim(). You might have been looking for *k_boundary()* instead.

cell_poset(*orientation='SE'*)

Return the Young diagram of `self` as a poset. The optional keyword variable `orientation` determines the order relation of the poset.

The poset always uses the set of cells of the Young diagram of `self` as its ground set. The order relation of the poset depends on the `orientation` variable (which defaults to `'SE'`). Concretely, `orientation` has to be specified to one of the strings `'NW'`, `'NE'`, `'SW'`, and `'SE'`, standing for "northwest", "northeast", "southwest" and "southeast", respectively. If `orientation` is `'SE'`, then the order relation of the poset is such that a cell u is greater or equal to a cell v in the poset if and only if u lies weakly southeast of v (this means that u can be reached from v by a sequence of south and east steps; the sequence is allowed to consist of south steps only, or of east steps only, or even be empty). Similarly the order relation is defined for the other three orientations. The Young diagram is supposed to be drawn in English notation.

The elements of the poset are the cells of the Young diagram of `self`, written as tuples of zero-based coordinates (so that $(3, 7)$ stands for the 8-th cell of the 4-th row, etc.).

EXAMPLES:

```
sage: # needs sage.graphs
sage: p = Partition([3,3,1])
sage: Q = p.cell_poset(); Q
Finite poset containing 7 elements
sage: sorted(Q)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: sorted(Q.maximal_elements())
[(1, 2), (2, 0)]
sage: Q.minimal_elements()
[(0, 0)]
sage: sorted(Q.upper_covers((1, 0)))
[(1, 1), (2, 0)]
sage: Q.upper_covers((1, 1))
[(1, 2)]

sage: # needs sage.graphs
sage: P = p.cell_poset(orientation="NW"); P
Finite poset containing 7 elements
sage: sorted(P)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: sorted(P.minimal_elements())
[(1, 2), (2, 0)]
sage: P.maximal_elements()
[(0, 0)]
sage: P.upper_covers((2, 0))
[(1, 0)]
sage: sorted(P.upper_covers((1, 2)))
[(0, 2), (1, 1)]
sage: sorted(P.upper_covers((1, 1)))
[(0, 1), (1, 0)]
sage: sorted([len(P.upper_covers(v)) for v in P])
[0, 1, 1, 1, 1, 2, 2]

sage: # needs sage.graphs
sage: R = p.cell_poset(orientation="NE"); R
Finite poset containing 7 elements
sage: sorted(R)
```

(continues on next page)

(continued from previous page)

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: R.maximal_elements()
[(0, 2)]
sage: R.minimal_elements()
[(2, 0)]
sage: sorted([len(R.upper_covers(v)) for v in R])
[0, 1, 1, 1, 1, 2, 2]
sage: R.is_isomorphic(P)
False
sage: R.is_isomorphic(P.dual())
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> p = Partition([Integer(3),Integer(3),Integer(1)])
>>> Q = p.cell_poset(); Q
Finite poset containing 7 elements
>>> sorted(Q)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
>>> sorted(Q.maximal_elements())
[(1, 2), (2, 0)]
>>> Q.minimal_elements()
[(0, 0)]
>>> sorted(Q.upper_covers((Integer(1), Integer(0))))
[(1, 1), (2, 0)]
>>> Q.upper_covers((Integer(1), Integer(1)))
[(1, 2)]

>>> # needs sage.graphs
>>> P = p.cell_poset(orientation="NW"); P
Finite poset containing 7 elements
>>> sorted(P)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
>>> sorted(P.minimal_elements())
[(1, 2), (2, 0)]
>>> P.maximal_elements()
[(0, 0)]
>>> P.upper_covers((Integer(2), Integer(0)))
[(1, 0)]
>>> sorted(P.upper_covers((Integer(1), Integer(2))))
[(0, 2), (1, 1)]
>>> sorted(P.upper_covers((Integer(1), Integer(1))))
[(0, 1), (1, 0)]
>>> sorted([len(P.upper_covers(v)) for v in P])
[0, 1, 1, 1, 1, 2, 2]

>>> # needs sage.graphs
>>> R = p.cell_poset(orientation="NE"); R
Finite poset containing 7 elements
>>> sorted(R)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
>>> R.maximal_elements()
```

(continues on next page)

(continued from previous page)

```
[(0, 2)]
>>> R.minimal_elements()
[(2, 0)]
>>> sorted([len(R.upper_covers(v)) for v in R])
[0, 1, 1, 1, 1, 2, 2]
>>> R.is_isomorphic(P)
False
>>> R.is_isomorphic(P.dual())
False
```

Linear extensions of `p.cell_poset()` are in 1-to-1 correspondence with standard Young tableaux of shape p:

```
sage: all( len(p.cell_poset().linear_extensions())                         #␣
↪needs sage.graphs
....:      == len(p.standard_tableaux())
....:      for n in range(8) for p in Partitions(n) )
True
```

```
>>> from sage.all import *
>>> all( len(p.cell_poset().linear_extensions())                         #␣
↪needs sage.graphs
...      == len(p.standard_tableaux())
...      for n in range(Integer(8)) for p in Partitions(n) )
True
```

This is not the case for northeast orientation:

```
sage: q = Partition([3, 1])
sage: q.cell_poset(orientation="NE").is_chain()                            #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> q = Partition([Integer(3), Integer(1)])
>>> q.cell_poset(orientation="NE").is_chain()                            #␣
↪needs sage.graphs
True
```

cells()

Return the coordinates of the cells of `self`.

EXAMPLES:

```
sage: Partition([2,2]).cells()
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: Partition([3,2]).cells()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2)]).cells()
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

(continues on next page)

(continued from previous page)

```
>>> Partition([Integer(3),Integer(2)]).cells()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
```

centralizer_size(*t=0*, *q=0*)

Return the size of the centralizer of any permutation of cycle type `self`.

If m_i is the multiplicity of i as a part of p, this is given by

$$\prod_i m_i! i^{m_i}.$$

Including the optional parameters t and q gives the q,t analog, which is the former product times

$$\prod_{i=1}^{\mathrm{length}(p)} \frac{1-q^{p_i}}{1-t^{p_i}}.$$

See Section 1.3, p. 24, in [Ke1991].

EXAMPLES:

```
sage: Partition([2,2,1]).centralizer_size()
8
sage: Partition([2,2,2]).centralizer_size()
48
sage: Partition([2,2,1]).centralizer_size(q=2, t=3)
9/16
sage: Partition([]).centralizer_size()
1
sage: Partition([]).centralizer_size(q=2, t=4)
1
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).centralizer_size()
8
>>> Partition([Integer(2),Integer(2),Integer(2)]).centralizer_size()
48
>>> Partition([Integer(2),Integer(2),Integer(1)]).centralizer_
↪size(q=Integer(2), t=Integer(3))
9/16
>>> Partition([]).centralizer_size()
1
>>> Partition([]).centralizer_size(q=Integer(2), t=Integer(4))
1
```

character_polynomial()

Return the character polynomial associated to the partition `self`.

The character polynomial q_μ associated to a partition μ is defined by

$$q_\mu(x_1, x_2, \ldots, x_k) = \downarrow \sum_{\alpha \vdash k} \frac{\chi^\mu_\alpha}{1^{a_1}2^{a_2}\cdots k^{a_k}a_1!a_2!\cdots a_k!} \prod_{i=1}^{k}(ix_i - 1)^{a_i}$$

where k is the size of μ, and a_i is the multiplicity of i in α.

It is computed in the following manner:

1. Expand the Schur function s_μ in the power-sum basis,
2. Replace each p_i with $ix_i - 1$,
3. Apply the umbral operator $\downarrow$ to the resulting polynomial.

EXAMPLES:

```
sage: Partition([1]).character_polynomial()                                    #␣
↪needs sage.modules
x - 1
sage: Partition([1,1]).character_polynomial()                                  #␣
↪needs sage.modules
1/2*x0^2 - 3/2*x0 - x1 + 1
sage: Partition([2,1]).character_polynomial()                                  #␣
↪needs sage.modules
1/3*x0^3 - 2*x0^2 + 8/3*x0 - x2
```

```
>>> from sage.all import *
>>> Partition([Integer(1)]).character_polynomial()                              ␣
↪     # needs sage.modules
x - 1
>>> Partition([Integer(1),Integer(1)]).character_polynomial()                   ␣
↪                # needs sage.modules
1/2*x0^2 - 3/2*x0 - x1 + 1
>>> Partition([Integer(2),Integer(1)]).character_polynomial()                   ␣
↪                # needs sage.modules
1/3*x0^3 - 2*x0^2 + 8/3*x0 - x2
```

components()

Return a list containing the shape of `self`.

This method exists only for compatibility with *PartitionTuples*.

EXAMPLES:

```
sage: Partition([3,2]).components()
[[3, 2]]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2)]).components()
[[3, 2]]
```

conjugacy_class_size()

Return the size of the conjugacy class of the symmetric group indexed by `self`.

EXAMPLES:

```
sage: Partition([2,2,2]).conjugacy_class_size()
15
sage: Partition([2,2,1]).conjugacy_class_size()
15
sage: Partition([2,1,1]).conjugacy_class_size()
6
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(2)]).conjugacy_class_size()
15
>>> Partition([Integer(2),Integer(2),Integer(1)]).conjugacy_class_size()
15
>>> Partition([Integer(2),Integer(1),Integer(1)]).conjugacy_class_size()
6
```

conjugate()

Return the conjugate partition of the partition `self`. This is also called the associated partition or the transpose in the literature.

EXAMPLES:

```
sage: Partition([2,2]).conjugate()
[2, 2]
sage: Partition([6,3,1]).conjugate()
[3, 2, 2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2)]).conjugate()
[2, 2]
>>> Partition([Integer(6),Integer(3),Integer(1)]).conjugate()
[3, 2, 2, 1, 1, 1]
```

The conjugate partition is obtained by transposing the Ferrers diagram of the partition (see *ferrers_diagram()*):

```
sage: print(Partition([6,3,1]).ferrers_diagram())
******
***
*
sage: print(Partition([6,3,1]).conjugate().ferrers_diagram())
***
**
**
*
*
*
```

```
>>> from sage.all import *
>>> print(Partition([Integer(6),Integer(3),Integer(1)]).ferrers_diagram())
******
***
*
>>> print(Partition([Integer(6),Integer(3),Integer(1)]).conjugate().ferrers_
↪diagram())
***
**
**
*
*
*
```

contains(*x*)

Return True if x is a partition whose Ferrers diagram is contained in the Ferrers diagram of self.

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.contains([2,1])
True
sage: all(p.contains(mu) for mu in Partitions(3))
True
sage: all(p.contains(mu) for mu in Partitions(4))
False
```

```
>>> from sage.all import *
>>> p = Partition([Integer(3),Integer(2),Integer(1)])
>>> p.contains([Integer(2),Integer(1)])
True
>>> all(p.contains(mu) for mu in Partitions(Integer(3)))
True
>>> all(p.contains(mu) for mu in Partitions(Integer(4)))
False
```

content(*r*, *c*, *multicharge=(0,)*)

Return the content of the cell at row r and column c.

The content of a cell is $c - r$.

For consistency with partition tuples there is also an optional multicharge argument which is an offset to the usual content. By setting the multicharge equal to the 0-element of the ring $\mathbf{Z}/e\mathbf{Z}$, the corresponding e-residue will be returned. This is the content modulo e.

The content (and residue) do not strictly depend on the partition, however, this method is included because it is often useful in the context of partitions.

EXAMPLES:

```
sage: Partition([2,1]).content(1,0)
-1
sage: p = Partition([3,2])
sage: sum([p.content(*c) for c in p.cells()])
2
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(1)]).content(Integer(1),Integer(0))
-1
>>> p = Partition([Integer(3),Integer(2)])
>>> sum([p.content(*c) for c in p.cells()])
2
```

and now we return the 3-residue of a cell:

```
sage: Partition([2,1]).content(1,0, multicharge=[IntegerModRing(3)(0)])
2
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(1)]).content(Integer(1),Integer(0),␣
↪multicharge=[IntegerModRing(Integer(3))(Integer(0))])
2
```

contents_tableau(*multicharge=(0,)*)

Return the tableau which has `(k,r,c)`-th cell equal to the content `multicharge[k] - r + c` of the cell.

EXAMPLES:

```
sage: Partition([2,1]).contents_tableau()
[[0, 1], [-1]]
sage: Partition([3,2,1,1]).contents_tableau().pp()
    0  1  2
   -1  0
   -2
   -3
sage: Partition([3,2,1,1]).contents_tableau([ IntegerModRing(3)(0)] ).pp()
    0  1  2
    2  0
    1
    0
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(1)]).contents_tableau()
[[0, 1], [-1]]
>>> Partition([Integer(3),Integer(2),Integer(1),Integer(1)]).contents_
↪tableau().pp()
    0  1  2
   -1  0
   -2
   -3
>>> Partition([Integer(3),Integer(2),Integer(1),Integer(1)]).contents_
↪tableau([ IntegerModRing(Integer(3))(Integer(0))] ).pp()
    0  1  2
    2  0
    1
    0
```

core(*length*)

Return the `length`-core of the partition – in the literature the core is commonly referred to as the k-core, p-core, r-core, … .

The r-core of a partition λ can be obtained by repeatedly removing rim hooks of size r from (the Young diagram of) λ until this is no longer possible. The remaining partition is the core.

EXAMPLES:

```
sage: Partition([6,3,2,2]).core(3)
[2, 1, 1]
sage: Partition([]).core(3)
[]
sage: Partition([8,7,7,4,1,1,1,1,1]).core(3)
[2, 1, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(6),Integer(3),Integer(2),Integer(2)]).core(Integer(3))
[2, 1, 1]
>>> Partition([]).core(Integer(3))
[]
>>> Partition([Integer(8),Integer(7),Integer(7),Integer(4),Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1)]).core(Integer(3))
[2, 1, 1]
```

corners()

Return a list of the corners of the partition `self`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

Note

This is referred to as an "inner corner" in [Sag2001].

EXAMPLES:

```
sage: Partition([3,2,1]).corners()
[(0, 2), (1, 1), (2, 0)]
sage: Partition([3,3,1]).corners()
[(1, 2), (2, 0)]
sage: Partition([]).corners()
[]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners()
[(0, 2), (1, 1), (2, 0)]
>>> Partition([Integer(3),Integer(3),Integer(1)]).corners()
[(1, 2), (2, 0)]
>>> Partition([]).corners()
[]
```

corners_residue(i, l)

Return a list of the corners of the partition `self` having `l`-residue `i`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind. See `residue()` for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners_residue(0, 3)
[(1, 1)]
sage: Partition([3,2,1]).corners_residue(1, 3)
[(2, 0)]
```

(continues on next page)

(continued from previous page)

```
sage: Partition([3,2,1]).corners_residue(2, 3)
[(0, 2)]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(0),
↪Integer(3))
[(1, 1)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(1),
↪Integer(3))
[(2, 0)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(2),
↪Integer(3))
[(0, 2)]
```

crank()

Return the Dyson crank of `self`.

The Dyson crank of a partition λ is defined as follows: If λ contains at least one 1, then the crank is $\mu(\lambda) - \omega(\lambda)$, where $\omega(\lambda)$ is the number of $1`sin`\lambda$, and $\mu(\lambda)$ is the number of parts of λ larger than $\omega(\lambda)$. If λ contains no 1, then the crank is simply the largest part of λ.

REFERENCES:

- [AG1988]

EXAMPLES:

```
sage: Partition([]).crank()
0
sage: Partition([3,2,2]).crank()
3
sage: Partition([5,4,2,1,1]).crank()
0
sage: Partition([1,1,1]).crank()
-3
sage: Partition([6,4,4,3]).crank()
6
sage: Partition([6,3,3,1,1]).crank()
1
sage: Partition([6]).crank()
6
sage: Partition([5,1]).crank()
0
sage: Partition([4,2]).crank()
4
sage: Partition([4,1,1]).crank()
-1
sage: Partition([3,3]).crank()
3
sage: Partition([3,2,1]).crank()
1
sage: Partition([3,1,1,1]).crank()
-3
sage: Partition([2,2,2]).crank()
```

(continues on next page)

(continued from previous page)

```
2
sage: Partition([2,2,1,1]).crank()
-2
sage: Partition([2,1,1,1,1]).crank()
-4
sage: Partition([1,1,1,1,1,1]).crank()
-6
```

```
>>> from sage.all import *
>>> Partition([]).crank()
0
>>> Partition([Integer(3),Integer(2),Integer(2)]).crank()
3
>>> Partition([Integer(5),Integer(4),Integer(2),Integer(1),Integer(1)]).
↪crank()
0
>>> Partition([Integer(1),Integer(1),Integer(1)]).crank()
-3
>>> Partition([Integer(6),Integer(4),Integer(4),Integer(3)]).crank()
6
>>> Partition([Integer(6),Integer(3),Integer(3),Integer(1),Integer(1)]).
↪crank()
1
>>> Partition([Integer(6)]).crank()
6
>>> Partition([Integer(5),Integer(1)]).crank()
0
>>> Partition([Integer(4),Integer(2)]).crank()
4
>>> Partition([Integer(4),Integer(1),Integer(1)]).crank()
-1
>>> Partition([Integer(3),Integer(3)]).crank()
3
>>> Partition([Integer(3),Integer(2),Integer(1)]).crank()
1
>>> Partition([Integer(3),Integer(1),Integer(1),Integer(1)]).crank()
-3
>>> Partition([Integer(2),Integer(2),Integer(2)]).crank()
2
>>> Partition([Integer(2),Integer(2),Integer(1),Integer(1)]).crank()
-2
>>> Partition([Integer(2),Integer(1),Integer(1),Integer(1),Integer(1)]).
↪crank()
-4
>>> Partition([Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1)]).crank()
-6
```

defect (*e*, *multicharge=(0,)*)

Return the e-defect or the e-weight of `self`.

The e-defect is the number of (connected) e-rim hooks that can be removed from the partition.

The defect of a partition is given by

$$\operatorname{defect}(\beta) = (\Lambda, \beta) - \tfrac{1}{2}(\beta, \beta),$$

where $\Lambda = \sum_r \Lambda_{\kappa_r}$ for the multicharge $(\kappa_1, \ldots, \kappa_\ell)$ and $\beta = \sum_{(r,c)} \alpha_{(c-r) \pmod e}$, with the sum being over the cells in the partition.

INPUT:

- `e` – the quantum characteristic
- `multicharge` – the multicharge (default: $(0,)$)

OUTPUT: nonnegative integer, which is the defect of the block containing the partition `self`

EXAMPLES:

```
sage: Partition([4,3,2]).defect(2)
3
sage: Partition([0]).defect(2)
0
sage: Partition([3]).defect(2)
1
sage: Partition([6]).defect(2)
3
sage: Partition([9]).defect(2)
4
sage: Partition([12]).defect(2)
6
sage: Partition([4,3,2]).defect(3)
3
sage: Partition([0]).defect(3)
0
sage: Partition([3]).defect(3)
1
sage: Partition([6]).defect(3)
2
sage: Partition([9]).defect(3)
3
sage: Partition([12]).defect(3)
4
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(2)]).defect(Integer(2))
3
>>> Partition([Integer(0)]).defect(Integer(2))
0
>>> Partition([Integer(3)]).defect(Integer(2))
1
>>> Partition([Integer(6)]).defect(Integer(2))
3
>>> Partition([Integer(9)]).defect(Integer(2))
4
>>> Partition([Integer(12)]).defect(Integer(2))
6
>>> Partition([Integer(4),Integer(3),Integer(2)]).defect(Integer(3))
```

(continues on next page)

(continued from previous page)

```
3
>>> Partition([Integer(0)]).defect(Integer(3))
0
>>> Partition([Integer(3)]).defect(Integer(3))
1
>>> Partition([Integer(6)]).defect(Integer(3))
2
>>> Partition([Integer(9)]).defect(Integer(3))
3
>>> Partition([Integer(12)]).defect(Integer(3))
4
```

degree(*e*)

Return the e-th degree of `self`.

The e-th degree of a partition λ is the sum of the e-th degrees of the standard tableaux of shape λ. The e-th degree is the exponent of $\Phi_e(q)$ in the Gram determinant of the Specht module for a semisimple Iwahori-Hecke algebra of type A with parameter q.

INPUT:

- `e` – an integer $e > 1$

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: Partition([4,3]).degree(2)
28
sage: Partition([4,3]).degree(3)
15
sage: Partition([4,3]).degree(4)
8
sage: Partition([4,3]).degree(5)
13
sage: Partition([4,3]).degree(6)
0
sage: Partition([4,3]).degree(7)
0
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3)]).degree(Integer(2))
28
>>> Partition([Integer(4),Integer(3)]).degree(Integer(3))
15
>>> Partition([Integer(4),Integer(3)]).degree(Integer(4))
8
>>> Partition([Integer(4),Integer(3)]).degree(Integer(5))
13
>>> Partition([Integer(4),Integer(3)]).degree(Integer(6))
0
>>> Partition([Integer(4),Integer(3)]).degree(Integer(7))
0
```

Therefore, the Gram determinant of $S(5,3)$ when the Hecke parameter q is "generic" is

$$q^N \Phi_2(q)^{28} \Phi_3(q)^{15} \Phi_4(q)^8 \Phi_5(q)^{13}$$

for some integer N. Compare with `prime_degree()`.

dimension(*smaller=None*, *k=1*)

Return the number of paths from the `smaller` partition to the partition `self`, where each step consists of adding a k-ribbon while keeping a partition.

Note that a 1-ribbon is just a single cell, so this counts paths in the Young graph when $k = 1$.

Note also that the default case ($k = 1$ and `smaller = []`) gives the dimension of the irreducible representation of the symmetric group corresponding to `self`.

INPUT:

- `smaller` – a partition (default: an empty list `[]`)
- `k` – positive integer (default: 1)

OUTPUT: the number of such paths

EXAMPLES:

Looks at the number of ways of getting from `[5,4]` to the empty partition, removing one cell at a time:

```
sage: mu = Partition([5,4])
sage: mu.dimension()
42
```

```
>>> from sage.all import *
>>> mu = Partition([Integer(5),Integer(4)])
>>> mu.dimension()
42
```

Same, but removing one 3-ribbon at a time. Note that the 3-core of `mu` is empty:

```
sage: mu.dimension(k=3)
3
```

```
>>> from sage.all import *
>>> mu.dimension(k=Integer(3))
3
```

The 2-core of `mu` is not the empty partition:

```
sage: mu.dimension(k=2)
0
```

```
>>> from sage.all import *
>>> mu.dimension(k=Integer(2))
0
```

Indeed, the 2-core of `mu` is `[1]`:

```
sage: mu.dimension(Partition([1]),k=2)
2
```

```
>>> from sage.all import *
>>> mu.dimension(Partition([Integer(1)]),k=Integer(2))
2
```

ALGORITHM:

Depending on the parameters given, different simplifications occur. When $k = 1$ and `smaller` is empty, this function uses the hook formula. When $k = 1$ and `smaller` is not empty, it uses a formula from [ORV].

When $k \neq 1$, we first check that both `self` and `smaller` have the same k-core, then use the k-quotients and the same algorithm on each of the k-quotients.

AUTHORS:

- Paul-Olivier Dehaye (2011-06-07)

dominated_partitions(*rows=None*)

Return a list of the partitions dominated by n. If `rows` is specified, then it only returns the ones whose number of rows is at most `rows`.

EXAMPLES:

```
sage: Partition([3,2,1]).dominated_partitions()
[[3, 2, 1], [3, 1, 1, 1], [2, 2, 2], [2, 2, 1, 1], [2, 1, 1, 1, 1], [1, 1, 1,␣
↪1, 1, 1]]
sage: Partition([3,2,1]).dominated_partitions(rows=3)
[[3, 2, 1], [2, 2, 2]]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).dominated_partitions()
[[3, 2, 1], [3, 1, 1, 1], [2, 2, 2], [2, 2, 1, 1], [2, 1, 1, 1, 1], [1, 1, 1,␣
↪1, 1, 1]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).dominated_
↪partitions(rows=Integer(3))
[[3, 2, 1], [2, 2, 2]]
```

dominates(*p2*)

Return `True` if `self` dominates the partition `p2`. Otherwise it returns `False`.

EXAMPLES:

```
sage: p = Partition([3,2])
sage: p.dominates([3,1])
True
sage: p.dominates([2,2])
True
sage: p.dominates([2,1,1])
True
sage: p.dominates([3,3])
False
sage: p.dominates([4])
False
sage: Partition([4]).dominates(p)
False
sage: Partition([]).dominates([1])
False
```

(continues on next page)

(continued from previous page)

```
sage: Partition([]).dominates([])
True
sage: Partition([1]).dominates([])
True
```

```
>>> from sage.all import *
>>> p = Partition([Integer(3),Integer(2)])
>>> p.dominates([Integer(3),Integer(1)])
True
>>> p.dominates([Integer(2),Integer(2)])
True
>>> p.dominates([Integer(2),Integer(1),Integer(1)])
True
>>> p.dominates([Integer(3),Integer(3)])
False
>>> p.dominates([Integer(4)])
False
>>> Partition([Integer(4)]).dominates(p)
False
>>> Partition([]).dominates([Integer(1)])
False
>>> Partition([]).dominates([])
True
>>> Partition([Integer(1)]).dominates([])
True
```

down()

Return a generator for partitions that can be obtained from `self` by removing a cell.

EXAMPLES:

```
sage: [p for p in Partition([2,1,1]).down()]
[[1, 1, 1], [2, 1]]
sage: [p for p in Partition([3,2]).down()]
[[2, 2], [3, 1]]
sage: [p for p in Partition([3,2,1]).down()]
[[2, 2, 1], [3, 1, 1], [3, 2]]
```

```
>>> from sage.all import *
>>> [p for p in Partition([Integer(2),Integer(1),Integer(1)]).down()]
[[1, 1, 1], [2, 1]]
>>> [p for p in Partition([Integer(3),Integer(2)]).down()]
[[2, 2], [3, 1]]
>>> [p for p in Partition([Integer(3),Integer(2),Integer(1)]).down()]
[[2, 2, 1], [3, 1, 1], [3, 2]]
```

down_list()

Return a list of the partitions that can be obtained from `self` by removing a cell.

EXAMPLES:

```
sage: Partition([2,1,1]).down_list()
[[1, 1, 1], [2, 1]]
```

(continues on next page)

(continued from previous page)

```
sage: Partition([3,2]).down_list()
[[2, 2], [3, 1]]
sage: Partition([3,2,1]).down_list()
[[2, 2, 1], [3, 1, 1], [3, 2]]
sage: Partition([]).down_list()  #checks :issue:`11435`
[]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(1),Integer(1)]).down_list()
[[1, 1, 1], [2, 1]]
>>> Partition([Integer(3),Integer(2)]).down_list()
[[2, 2], [3, 1]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).down_list()
[[2, 2, 1], [3, 1, 1], [3, 2]]
>>> Partition([]).down_list()  #checks :issue:`11435`
[]
```

dual_equivalence_graph(*directed=False*, *coloring=None*)

Return the dual equivalence graph of `self`.

Two permutations p and q in the symmetric group S_n differ by an *i-elementary dual equivalence (or dual Knuth) relation* (where i is an integer with $1 < i < n$) when the following two conditions are satisfied:

- In the one-line notation of the permutation p, the letter i does not appear between $i-1$ and $i+1$.
- The permutation q is obtained from p by switching two of the three letters $i-1, i, i+1$ (in its one-line notation) – namely, the leftmost and the rightmost one in order of their appearance in p.

Notice that this is equivalent to the statement that the permutations p^{-1} and q^{-1} differ by an elementary Knuth equivalence at positions $i-1, i, i+1$.

Two standard Young tableaux of shape λ differ by an i-elementary dual equivalence relation (of color i), if their reading words differ by an i-elementary dual equivalence relation.

The *dual equivalence graph* of the partition λ is the edge-colored graph whose vertices are the standard Young tableaux of shape λ, and whose edges colored by i are given by the i-elementary dual equivalences.

INPUT:

- `directed` – boolean (default: `False`); whether to have the dual equivalence graph be directed (where we have a directed edge $S \to T$ if i appears to the left of $i+1$ in the reading word of T; otherwise we have the directed edge $T \to S$)
- `coloring` – (optional) a function which sends each integer $i > 1$ to a color (as a string, e.g., `'red'` or `'black'`) to be used when visually representing the resulting graph using dot2tex; the default choice is `2 -> 'red', 3 -> 'blue', 4 -> 'green', 5 -> 'purple', 6 -> 'brown', 7 -> 'orange', 8 -> 'yellow', anything greater than 8 -> 'black'`.

REFERENCES:

- [As2008b]

EXAMPLES:

```
sage: # needs sage.graphs
sage: P = Partition([3,1,1])
sage: G = P.dual_equivalence_graph()
sage: G.edges(sort=True)
```

(continues on next page)

(continued from previous page)

```
[(([1, 2, 3], [4], [5]], [[1, 2, 4], [3], [5]], 3),
 ([[1, 2, 4], [3], [5]], [[1, 2, 5], [3], [4]], 4),
 ([[1, 2, 4], [3], [5]], [[1, 3, 4], [2], [5]], 2),
 ([[1, 2, 5], [3], [4]], [[1, 3, 5], [2], [4]], 2),
 ([[1, 3, 4], [2], [5]], [[1, 3, 5], [2], [4]], 4),
 ([[1, 3, 5], [2], [4]], [[1, 4, 5], [2], [3]], 3)]
sage: G = P.dual_equivalence_graph(directed=True)
sage: G.edges(sort=True)
[(([1, 2, 4], [3], [5]], [[1, 2, 3], [4], [5]], 3),
 ([[1, 2, 5], [3], [4]], [[1, 2, 4], [3], [5]], 4),
 ([[1, 3, 4], [2], [5]], [[1, 2, 4], [3], [5]], 2),
 ([[1, 3, 5], [2], [4]], [[1, 2, 5], [3], [4]], 2),
 ([[1, 3, 5], [2], [4]], [[1, 3, 4], [2], [5]], 4),
 ([[1, 4, 5], [2], [3]], [[1, 3, 5], [2], [4]], 3)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> P = Partition([Integer(3),Integer(1),Integer(1)])
>>> G = P.dual_equivalence_graph()
>>> G.edges(sort=True)
[(([1, 2, 3], [4], [5]], [[1, 2, 4], [3], [5]], 3),
 ([[1, 2, 4], [3], [5]], [[1, 2, 5], [3], [4]], 4),
 ([[1, 2, 4], [3], [5]], [[1, 3, 4], [2], [5]], 2),
 ([[1, 2, 5], [3], [4]], [[1, 3, 5], [2], [4]], 2),
 ([[1, 3, 4], [2], [5]], [[1, 3, 5], [2], [4]], 4),
 ([[1, 3, 5], [2], [4]], [[1, 4, 5], [2], [3]], 3)]
>>> G = P.dual_equivalence_graph(directed=True)
>>> G.edges(sort=True)
[(([1, 2, 4], [3], [5]], [[1, 2, 3], [4], [5]], 3),
 ([[1, 2, 5], [3], [4]], [[1, 2, 4], [3], [5]], 4),
 ([[1, 3, 4], [2], [5]], [[1, 2, 4], [3], [5]], 2),
 ([[1, 3, 5], [2], [4]], [[1, 2, 5], [3], [4]], 2),
 ([[1, 3, 5], [2], [4]], [[1, 3, 4], [2], [5]], 4),
 ([[1, 4, 5], [2], [3]], [[1, 3, 5], [2], [4]], 3)]
```

evaluation()

Return the evaluation of `self`.

The **commutative evaluation**, often shortened to **evaluation**, of a word (we think of a partition as a word in $\{1, 2, 3, \ldots\}$) is its image in the free commutative monoid. In other words, this counts how many occurrences there are of each letter.

This is also is known as **Parikh vector** and **abelianization** and has the same output as `to_exp()`.

EXAMPLES:

```
sage: Partition([4,3,1,1]).evaluation()
[2, 0, 1, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(1),Integer(1)]).evaluation()
[2, 0, 1, 1]
```

ferrers_diagram()

Return the Ferrers diagram of `self`.

EXAMPLES:

```
sage: mu = Partition([5,5,2,1])
sage: Partitions.options(diagram_str='*', convention='english')
sage: print(mu.ferrers_diagram())
*****
*****
**
*
sage: Partitions.options(diagram_str='█')
sage: print(mu.ferrers_diagram())
█████
█████
██
█
sage: Partitions.options.convention="french"
sage: print(mu.ferrers_diagram())
█
██
█████
█████
sage: print(Partition([]).ferrers_diagram())
-
sage: Partitions.options(diagram_str='-')
sage: print(Partition([]).ferrers_diagram())
(/)
sage: Partitions.options._reset()
```

```
>>> from sage.all import *
>>> mu = Partition([Integer(5),Integer(5),Integer(2),Integer(1)])
>>> Partitions.options(diagram_str='*', convention='english')
>>> print(mu.ferrers_diagram())
*****
*****
**
*
>>> Partitions.options(diagram_str='█')
>>> print(mu.ferrers_diagram())
█████
█████
██
█
>>> Partitions.options.convention="french"
>>> print(mu.ferrers_diagram())
█
██
█████
█████
>>> print(Partition([]).ferrers_diagram())
-
```

(continues on next page)

(continued from previous page)

```
>>> Partitions.options(diagram_str='-')
>>> print(Partition([]).ferrers_diagram())
(/)
>>> Partitions.options._reset()
```

frobenius_coordinates()

Return a pair of sequences of Frobenius coordinates aka beta numbers of the partition.

These are two strictly decreasing sequences of nonnegative integers of the same length.

EXAMPLES:

```
sage: Partition([]).frobenius_coordinates()
([], [])
sage: Partition([1]).frobenius_coordinates()
([0], [0])
sage: Partition([3,3,3]).frobenius_coordinates()
([2, 1, 0], [2, 1, 0])
sage: Partition([9,1,1,1,1,1,1]).frobenius_coordinates()
([8], [6])
```

```
>>> from sage.all import *
>>> Partition([]).frobenius_coordinates()
([], [])
>>> Partition([Integer(1)]).frobenius_coordinates()
([0], [0])
>>> Partition([Integer(3),Integer(3),Integer(3)]).frobenius_coordinates()
([2, 1, 0], [2, 1, 0])
>>> Partition([Integer(9),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(1)]).frobenius_coordinates()
([8], [6])
```

frobenius_rank()

Return the Frobenius rank of the partition `self`.

The Frobenius rank of a partition $\lambda = (\lambda_1, \lambda_2, \lambda_3, \cdots)$ is defined to be the largest i such that $\lambda_i \geq i$. In other words, it is the number of cells on the main diagonal of λ. In yet other words, it is the size of the largest square fitting into the Young diagram of λ.

EXAMPLES:

```
sage: Partition([]).frobenius_rank()
0
sage: Partition([1]).frobenius_rank()
1
sage: Partition([3,3,3]).frobenius_rank()
3
sage: Partition([9,1,1,1,1,1]).frobenius_rank()
1
sage: Partition([2,1,1,1,1,1]).frobenius_rank()
1
sage: Partition([2,2,1,1,1,1]).frobenius_rank()
2
sage: Partition([3,2]).frobenius_rank()
```

(continues on next page)

(continued from previous page)

```
2
sage: Partition([3,2,2]).frobenius_rank()
2
sage: Partition([8,4,4,4,4]).frobenius_rank()
4
sage: Partition([8,4,1]).frobenius_rank()
2
sage: Partition([3,3,1]).frobenius_rank()
2
```

```
>>> from sage.all import *
>>> Partition([]).frobenius_rank()
0
>>> Partition([Integer(1)]).frobenius_rank()
1
>>> Partition([Integer(3),Integer(3),Integer(3)]).frobenius_rank()
3
>>> Partition([Integer(9),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1)]).frobenius_rank()
1
>>> Partition([Integer(2),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1)]).frobenius_rank()
1
>>> Partition([Integer(2),Integer(2),Integer(1),Integer(1),Integer(1),
↪Integer(1)]).frobenius_rank()
2
>>> Partition([Integer(3),Integer(2)]).frobenius_rank()
2
>>> Partition([Integer(3),Integer(2),Integer(2)]).frobenius_rank()
2
>>> Partition([Integer(8),Integer(4),Integer(4),Integer(4),Integer(4)]).
↪frobenius_rank()
4
>>> Partition([Integer(8),Integer(4),Integer(1)]).frobenius_rank()
2
>>> Partition([Integer(3),Integer(3),Integer(1)]).frobenius_rank()
2
```

from_kbounded_to_grassmannian(k)

Map a k-bounded partition to a Grassmannian element in the affine Weyl group of type $A_k^{(1)}$.

For details, see the documentation of the method *from_kbounded_to_reduced_word()* .

EXAMPLES:

```
sage: p = Partition([2,1,1])
sage: p.from_kbounded_to_grassmannian(2)                                    #␣
↪needs sage.modules
[-1  1  1]
[-2  2  1]
[-2  1  2]
sage: p = Partition([])
```

(continues on next page)

(continued from previous page)

```
sage: p.from_kbounded_to_grassmannian(2)                                    #␣
↪needs sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(1),Integer(1)])
>>> p.from_kbounded_to_grassmannian(Integer(2))                            ␣
↪     # needs sage.modules
[-1  1  1]
[-2  2  1]
[-2  1  2]
>>> p = Partition([])
>>> p.from_kbounded_to_grassmannian(Integer(2))                            ␣
↪     # needs sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
```

from_kbounded_to_reduced_word(k)

Map a k-bounded partition to a reduced word for an element in the affine permutation group.

This uses the fact that there is a bijection between k-bounded partitions and $(k+1)$-cores and an action of the affine nilCoxeter algebra of type $A_k^{(1)}$ on $(k+1)$-cores as described in [LM2006b].

EXAMPLES:

```
sage: p = Partition([2,1,1])
sage: p.from_kbounded_to_reduced_word(2)
[2, 1, 2, 0]
sage: p = Partition([3,1])
sage: p.from_kbounded_to_reduced_word(3)
[3, 2, 1, 0]
sage: p.from_kbounded_to_reduced_word(2)
Traceback (most recent call last):
...
ValueError: the partition must be 2-bounded
sage: p = Partition([])
sage: p.from_kbounded_to_reduced_word(2)
[]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(1),Integer(1)])
>>> p.from_kbounded_to_reduced_word(Integer(2))
[2, 1, 2, 0]
>>> p = Partition([Integer(3),Integer(1)])
>>> p.from_kbounded_to_reduced_word(Integer(3))
[3, 2, 1, 0]
>>> p.from_kbounded_to_reduced_word(Integer(2))
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: the partition must be 2-bounded
>>> p = Partition([])
>>> p.from_kbounded_to_reduced_word(Integer(2))
[]
```

garnir_tableau(**cell*)

Return the Garnir tableau of shape `self` corresponding to the cell `cell`. If `cell` $= (a, c)$ then $(a + 1, c)$ must belong to the diagram of `self`.

The Garnir tableaux play an important role in integral and non-semisimple representation theory because they determine the "straightening" rules for the Specht modules over an arbitrary ring.

The Garnir tableaux are the "first" non-standard tableaux which arise when you act by simple transpositions. If (a, c) is a cell in the Young diagram of a partition, which is not at the bottom of its column, then the corresponding Garnir tableau has the integers $1, 2, \ldots, n$ entered in order from left to right along the rows of the diagram up to the cell $(a, c - 1)$, then along the cells $(a + 1, 1)$ to $(a + 1, c)$, then (a, c) until the end of row a and then continuing from left to right in the remaining positions. The examples below probably make this clearer!

Note: The function also sets `g._garnir_cell`, where `g` is the resulting Garnir tableau, equal to `cell` which is used by some other functions.

EXAMPLES:

```
sage: g = Partition([5,3,3,2]).garnir_tableau((0,2)); g.pp()
  1  2  6  7  8
  3  4  5
  9 10 11
 12 13
sage: g.is_row_strict(); g.is_column_strict()
True
False

sage: Partition([5,3,3,2]).garnir_tableau(0,2).pp()
  1  2  6  7  8
  3  4  5
  9 10 11
 12 13
sage: Partition([5,3,3,2]).garnir_tableau(2,1).pp()
  1  2  3  4  5
  6  7  8
  9 12 13
 10 11
sage: Partition([5,3,3,2]).garnir_tableau(2,2).pp()
Traceback (most recent call last):
...
ValueError: (row+1, col) must be inside the diagram
```

```
>>> from sage.all import *
>>> g = Partition([Integer(5),Integer(3),Integer(3),Integer(2)]).garnir_
```

(continues on next page)

(continued from previous page)

```
↪tableau((Integer(0),Integer(2))); g.pp()
  1  2  6  7  8
  3  4  5
  9 10 11
 12 13
>>> g.is_row_strict(); g.is_column_strict()
True
False

>>> Partition([Integer(5),Integer(3),Integer(3),Integer(2)]).garnir_
↪tableau(Integer(0),Integer(2)).pp()
  1  2  6  7  8
  3  4  5
  9 10 11
 12 13
>>> Partition([Integer(5),Integer(3),Integer(3),Integer(2)]).garnir_
↪tableau(Integer(2),Integer(1)).pp()
  1  2  3  4  5
  6  7  8
  9 12 13
 10 11
>>> Partition([Integer(5),Integer(3),Integer(3),Integer(2)]).garnir_
↪tableau(Integer(2),Integer(2)).pp()
Traceback (most recent call last):
...
ValueError: (row+1, col) must be inside the diagram
```

See also

- `top_garnir_tableau()`

garsia_procesi_module(*base_ring=None*)

Return the `Garsia-Procesi module` corresponding to `self`.

INPUT:

- `base_ring` – (default: $\mathbf{Q}$) the base ring

EXAMPLES:

```
sage: GP = Partition([3,2,1]).garsia_procesi_module(QQ); GP
Garsia-Procesi module of shape [3, 2, 1] over Rational Field
sage: GP.graded_frobenius_image()
q^4*s[3, 2, 1] + q^3*s[3, 3] + q^3*s[4, 1, 1] + (q^3+q^2)*s[4, 2]
 + (q^2+q)*s[5, 1] + s[6]

sage: Partition([3,2,1]).garsia_procesi_module(GF(3))
Garsia-Procesi module of shape [3, 2, 1] over Finite Field of size 3
```

```
>>> from sage.all import *
>>> GP = Partition([Integer(3),Integer(2),Integer(1)]).garsia_procesi_
```

(continues on next page)

(continued from previous page)

```
↪module(QQ); GP
Garsia-Procesi module of shape [3, 2, 1] over Rational Field
>>> GP.graded_frobenius_image()
q^4*s[3, 2, 1] + q^3*s[3, 3] + q^3*s[4, 1, 1] + (q^3+q^2)*s[4, 2]
 + (q^2+q)*s[5, 1] + s[6]

>>> Partition([Integer(3),Integer(2),Integer(1)]).garsia_procesi_
↪module(GF(Integer(3)))
Garsia-Procesi module of shape [3, 2, 1] over Finite Field of size 3
```

generalized_pochhammer_symbol(*a*, *alpha*)

Return the generalized Pochhammer symbol $(a)^{(\alpha)}_{self}$. This is the product over all cells (i, j) in `self` of $a - (i - 1)/\alpha + j - 1$.

EXAMPLES:

```
sage: Partition([2,2]).generalized_pochhammer_symbol(2,1)
12
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2)]).generalized_pochhammer_
↪symbol(Integer(2),Integer(1))
12
```

get_part(*i*, *default=0*)

Return the i-th part of `self`, or `default` if it does not exist.

EXAMPLES:

```
sage: p = Partition([2,1])
sage: p.get_part(0), p.get_part(1), p.get_part(2)
(2, 1, 0)
sage: p.get_part(10,-1)
-1
sage: Partition([]).get_part(0)
0
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(1)])
>>> p.get_part(Integer(0)), p.get_part(Integer(1)), p.get_part(Integer(2))
(2, 1, 0)
>>> p.get_part(Integer(10),-Integer(1))
-1
>>> Partition([]).get_part(Integer(0))
0
```

glaisher_franklin(*s*)

Apply the Glaisher-Franklin bijection to `self`.

The Franklin-Glaisher bijection, with parameter s, returns a partition whose set of parts that are repeated at least s times equals the set of parts divisible by s in `self`, after dividing each part by s.

INPUT:

- s – positive integer

EXAMPLES:

```
sage: Partition([4, 3, 2, 2, 1]).glaisher_franklin(2)
[3, 2, 2, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(4), Integer(3), Integer(2), Integer(2), Integer(1)]).
→glaisher_franklin(Integer(2))
[3, 2, 2, 1, 1, 1, 1, 1]
```

glaisher_franklin_inverse(*s*)

Apply the inverse of the Glaisher-Franklin bijection to self.

The inverse of the Franklin-Glaisher bijection, with parameter s, returns a partition whose set of parts that are divisible by s, after dividing each by s, equals the equals the set of parts repeated at least s times in self.

INPUT:

- s – positive integer

EXAMPLES:

```
sage: Partition([4, 3, 2, 2, 1]).glaisher_franklin(2)
[3, 2, 2, 1, 1, 1, 1, 1]
sage: Partition([3, 2, 2, 1, 1, 1, 1, 1]).glaisher_franklin_inverse(2)
[4, 3, 2, 2, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(4), Integer(3), Integer(2), Integer(2), Integer(1)]).
→glaisher_franklin(Integer(2))
[3, 2, 2, 1, 1, 1, 1, 1]
>>> Partition([Integer(3), Integer(2), Integer(2), Integer(1), Integer(1),␣
→Integer(1), Integer(1), Integer(1)]).glaisher_franklin_inverse(Integer(2))
[4, 3, 2, 2, 1]
```

has_k_rectangle(*k*)

Return True if the Ferrer's diagram of self contains $k - i + 1$ rows (*or more*) of length i (*exactly*) for any i in $[1, k]$.

This is mainly a helper function for *is_k_reducible()* and *is_k_irreducible()*, the only difference between this function and *is_k_reducible()* being that this function allows any partition as input while *is_k_reducible()* requires the input to be k-bounded.

EXAMPLES:

The partition [1, 1, 1] has at least 2 rows of length 1:

```
sage: Partition([1, 1, 1]).has_k_rectangle(2)
True
```

```
>>> from sage.all import *
>>> Partition([Integer(1), Integer(1), Integer(1)]).has_k_
→rectangle(Integer(2))
True
```

The partition [1, 1, 1] does *not* have 4 rows of length 1, 3 rows of length 2, 2 rows of length 3, nor 1 row of length 4:

```
sage: Partition([1, 1, 1]).has_k_rectangle(4)
False
```

```
>>> from sage.all import *
>>> Partition([Integer(1), Integer(1), Integer(1)]).has_k_
↪rectangle(Integer(4))
False
```

See also:

is_k_irreducible(), *is_k_reducible()*, *has_rectangle()*

has_rectangle(*h*, *w*)

Return True if the Ferrer's diagram of self has h (*or more*) rows of length w (*exactly*).

INPUT:

- h – integer $h \geq 1$; the (*minimum*) height of the rectangle
- w – integer $w \geq 1$; the width of the rectangle

EXAMPLES:

```
sage: Partition([3, 3, 3, 3]).has_rectangle(2, 3)
True
sage: Partition([3, 3]).has_rectangle(2, 3)
True
sage: Partition([4, 3]).has_rectangle(2, 3)
False
sage: Partition([3]).has_rectangle(2, 3)
False
```

```
>>> from sage.all import *
>>> Partition([Integer(3), Integer(3), Integer(3), Integer(3)]).has_
↪rectangle(Integer(2), Integer(3))
True
>>> Partition([Integer(3), Integer(3)]).has_rectangle(Integer(2), Integer(3))
True
>>> Partition([Integer(4), Integer(3)]).has_rectangle(Integer(2), Integer(3))
False
>>> Partition([Integer(3)]).has_rectangle(Integer(2), Integer(3))
False
```

See also:

has_k_rectangle()

hook_length(*i*, *j*)

Return the length of the hook of cell (i, j) in self.

The (length of the) hook of cell (i, j) of a partition λ is

$$\lambda_i + \lambda'_j - i - j + 1$$

where λ' is the conjugate partition. In English convention, the hook length is the number of cells horizontally to the right and vertically below the cell (i, j) (including that cell).

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.hook_length(0, 0)
4
sage: p.hook_length(0, 1)
2
sage: p.hook_length(2, 0)
1
sage: Partition([3,3]).hook_length(0, 0)
4
sage: cell = [0,0]; Partition([3,3]).hook_length(*cell)
4
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(2),Integer(1)])
>>> p.hook_length(Integer(0), Integer(0))
4
>>> p.hook_length(Integer(0), Integer(1))
2
>>> p.hook_length(Integer(2), Integer(0))
1
>>> Partition([Integer(3),Integer(3)]).hook_length(Integer(0), Integer(0))
4
>>> cell = [Integer(0),Integer(0)]; Partition([Integer(3),Integer(3)]).hook_
↪length(*cell)
4
```

hook_lengths()

Return a tableau of shape `self` with the cells filled in with the hook lengths.

In each cell, put the sum of one plus the number of cells horizontally to the right and vertically below the cell (the hook length).

For example, consider the partition `[3,2,1]` of 6 with Ferrers diagram:

```
# # #
# #
#
```

When we fill in the cells with the hook lengths, we obtain:

```
5 3 1
3 1
1
```

EXAMPLES:

```
sage: Partition([2,2,1]).hook_lengths()
[[4, 2], [3, 1], [1]]
sage: Partition([3,3]).hook_lengths()
[[4, 3, 2], [3, 2, 1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
sage: Partition([2,2]).hook_lengths()
[[3, 2], [2, 1]]
sage: Partition([5]).hook_lengths()
[[5, 4, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).hook_lengths()
[[4, 2], [3, 1], [1]]
>>> Partition([Integer(3),Integer(3)]).hook_lengths()
[[4, 3, 2], [3, 2, 1]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
>>> Partition([Integer(2),Integer(2)]).hook_lengths()
[[3, 2], [2, 1]]
>>> Partition([Integer(5)]).hook_lengths()
[[5, 4, 3, 2, 1]]
```

REFERENCES:

- http://mathworld.wolfram.com/HookLengthFormula.html

hook_polynomial (q, t)

Return the two-variable hook polynomial.

EXAMPLES:

```
sage: R.<q,t> = PolynomialRing(QQ)
sage: a = Partition([2,2]).hook_polynomial(q,t)
sage: a == (1 - t)*(1 - q*t)*(1 - t^2)*(1 - q*t^2)
True
sage: a = Partition([3,2,1]).hook_polynomial(q,t)
sage: a == (1 - t)^3*(1 - q*t^2)^2*(1 - q^2*t^3)
True
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('q', 't',)); (q, t,) = R._first_ngens(2)
>>> a = Partition([Integer(2),Integer(2)]).hook_polynomial(q,t)
>>> a == (Integer(1) - t)*(Integer(1) - q*t)*(Integer(1) -
↪t**Integer(2))*(Integer(1) - q*t**Integer(2))
True
>>> a = Partition([Integer(3),Integer(2),Integer(1)]).hook_polynomial(q,t)
>>> a == (Integer(1) - t)**Integer(3)*(Integer(1) -
↪q*t**Integer(2))**Integer(2)*(Integer(1) - q**Integer(2)*t**Integer(3))
True
```

hook_product (a)

Return the Jack hook-product.

EXAMPLES:

```
sage: Partition([3,2,1]).hook_product(x)                                    #␣
↪needs sage.symbolic
(2*x + 3)*(x + 2)^2
sage: Partition([2,2]).hook_product(x)                                      #␣
↪needs sage.symbolic
2*(x + 2)*(x + 1)
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).hook_product(x)          ␣
↪                       # needs sage.symbolic
(2*x + 3)*(x + 2)^2
>>> Partition([Integer(2),Integer(2)]).hook_product(x)                     ␣
↪                  # needs sage.symbolic
2*(x + 2)*(x + 1)
```

hooks()

Return a sorted list of the hook lengths in `self`.

EXAMPLES:

```
sage: Partition([3,2,1]).hooks()
[5, 3, 3, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).hooks()
[5, 3, 3, 1, 1, 1]
```

horizontal_border_strip_cells(*k*)

Return a list of all the horizontal border strips of length `k` which can be added to `self`, where each horizontal border strip is a `generator` of cells.

EXAMPLES:

```
sage: list(Partition([]).horizontal_border_strip_cells(0))
[]
sage: list(Partition([3,2,1]).horizontal_border_strip_cells(0))
[]
sage: list(Partition([]).horizontal_border_strip_cells(2))
[[(0, 0), (0, 1)]]
sage: list(Partition([2,2]).horizontal_border_strip_cells(2))
[[(0, 2), (0, 3)], [(0, 2), (2, 0)], [(2, 0), (2, 1)]]
sage: list(Partition([3,2,2]).horizontal_border_strip_cells(2))
[[(0, 3), (0, 4)],
 [(0, 3), (1, 2)],
 [(0, 3), (3, 0)],
 [(1, 2), (3, 0)],
 [(3, 0), (3, 1)]]
```

```
>>> from sage.all import *
>>> list(Partition([]).horizontal_border_strip_cells(Integer(0)))
[]
>>> list(Partition([Integer(3),Integer(2),Integer(1)]).horizontal_border_
↪strip_cells(Integer(0)))
```

(continues on next page)

(continued from previous page)

```
[]
>>> list(Partition([]).horizontal_border_strip_cells(Integer(2)))
[[(0, 0), (0, 1)]]
>>> list(Partition([Integer(2),Integer(2)]).horizontal_border_strip_
↪cells(Integer(2)))
[[(0, 2), (0, 3)], [(0, 2), (2, 0)], [(2, 0), (2, 1)]]
>>> list(Partition([Integer(3),Integer(2),Integer(2)]).horizontal_border_
↪strip_cells(Integer(2)))
[[(0, 3), (0, 4)],
 [(0, 3), (1, 2)],
 [(0, 3), (3, 0)],
 [(1, 2), (3, 0)],
 [(3, 0), (3, 1)]]
```

initial_column_tableau()

Return the initial column tableau of shape `self`.

The initial column tableau of shape `self` is the standard tableau that has the numbers 1 to n, where n is the *size()* of `self`, entered in order from top to bottom and then left to right down the columns of `self`.

EXAMPLES:

```
sage: Partition([3,2]).initial_column_tableau()
[[1, 3, 5], [2, 4]]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2)]).initial_column_tableau()
[[1, 3, 5], [2, 4]]
```

initial_tableau()

Return the *standard tableau* which has the numbers $1, 2, \ldots, n$ where n is the *size()* of `self` entered in order from left to right along the rows of each component, where the components are ordered from left to right.

EXAMPLES:

```
sage: Partition([3,2,2]).initial_tableau()
[[1, 2, 3], [4, 5], [6, 7]]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(2)]).initial_tableau()
[[1, 2, 3], [4, 5], [6, 7]]
```

inside_corners()

Return a list of the corners of the partition `self`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

> **Note**
>
> This is referred to as an "inner corner" in [Sag2001].

EXAMPLES:

```
sage: Partition([3,2,1]).corners()
[(0, 2), (1, 1), (2, 0)]
sage: Partition([3,3,1]).corners()
[(1, 2), (2, 0)]
sage: Partition([]).corners()
[]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners()
[(0, 2), (1, 1), (2, 0)]
>>> Partition([Integer(3),Integer(3),Integer(1)]).corners()
[(1, 2), (2, 0)]
>>> Partition([]).corners()
[]
```

inside_corners_residue(*i*, *l*)

Return a list of the corners of the partition `self` having `l`-residue `i`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind. See *residue()* for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners_residue(0, 3)
[(1, 1)]
sage: Partition([3,2,1]).corners_residue(1, 3)
[(2, 0)]
sage: Partition([3,2,1]).corners_residue(2, 3)
[(0, 2)]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(0),
→Integer(3))
[(1, 1)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(1),
→Integer(3))
[(2, 0)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(2),
→Integer(3))
[(0, 2)]
```

is_core(*k*)

Return `True` if the Partition `self` is a `k`-core.

A partition is said to be a *`k`-core* if it has no hooks of length k. Equivalently, a partition is said to be a k-core if it is its own k-core (where the latter is defined as in `core()`).

Visually, this can be checked by trying to remove border strips of size k from `self`. If this is not possible, then `self` is a k-core.

EXAMPLES:

In the partition (2, 1), a hook length of 2 does not occur, but a hook length of 3 does:

```
sage: p = Partition([2, 1])
sage: p.is_core(2)
True
sage: p.is_core(3)
False

sage: q = Partition([12, 8, 5, 5, 2, 2, 1])
sage: q.is_core(4)
False
sage: q.is_core(5)
True
sage: q.is_core(0)
True
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2), Integer(1)])
>>> p.is_core(Integer(2))
True
>>> p.is_core(Integer(3))
False

>>> q = Partition([Integer(12), Integer(8), Integer(5), Integer(5),␣
→Integer(2), Integer(2), Integer(1)])
>>> q.is_core(Integer(4))
False
>>> q.is_core(Integer(5))
True
>>> q.is_core(Integer(0))
True
```

See also

`core()`, `Core`

is_empty()

Return `True` if `self` is the empty partition.

EXAMPLES:

```
sage: Partition([]).is_empty()
True
sage: Partition([2,1,1]).is_empty()
False
```

```
>>> from sage.all import *
>>> Partition([]).is_empty()
True
>>> Partition([Integer(2),Integer(1),Integer(1)]).is_empty()
False
```

is_k_bounded(*k*)

Return `True` if the partition `self` is bounded by `k`.

EXAMPLES:

```
sage: Partition([4, 3, 1]).is_k_bounded(4)
True
sage: Partition([4, 3, 1]).is_k_bounded(7)
True
sage: Partition([4, 3, 1]).is_k_bounded(3)
False
```

```
>>> from sage.all import *
>>> Partition([Integer(4), Integer(3), Integer(1)]).is_k_bounded(Integer(4))
True
>>> Partition([Integer(4), Integer(3), Integer(1)]).is_k_bounded(Integer(7))
True
>>> Partition([Integer(4), Integer(3), Integer(1)]).is_k_bounded(Integer(3))
False
```

is_k_irreducible(*k*)

Return `True` if the partition `self` is `k`-irreducible.

A k-bounded partition is k-*irreducible* if its Ferrer's diagram does *not* contain $k - i + 1$ rows (or more) of length i (exactly) for every $i \in [1, k]$.

(Also, a k-bounded partition is k-irreducible if and only if it is not k-reducible.)

EXAMPLES:

The partition [1, 1, 1] has at least 2 rows of length 1:

```
sage: Partition([1, 1, 1]).is_k_irreducible(2)
False
```

```
>>> from sage.all import *
>>> Partition([Integer(1), Integer(1), Integer(1)]).is_k_
↪irreducible(Integer(2))
False
```

The partition [1, 1, 1] does *not* have 4 rows of length 1, 3 rows of length 2, 2 rows of length 3, nor 1 row of length 4:

```
sage: Partition([1, 1, 1]).is_k_irreducible(4)
True
```

```
>>> from sage.all import *
>>> Partition([Integer(1), Integer(1), Integer(1)]).is_k_
```

(continues on next page)

(continued from previous page)

```
↪irreducible(Integer(4))
True
```

See also:

is_k_reducible(), *has_k_rectangle()*

is_k_reducible(*k*)

Return True if the partition self is k-reducible.

A k-bounded partition is *k-reducible* if its Ferrer's diagram contains $k - i + 1$ rows (or more) of length i (exactly) for some $i \in [1, k]$.

(Also, a k-bounded partition is k-reducible if and only if it is not k-irreducible.)

EXAMPLES:

The partition [1, 1, 1] has at least 2 rows of length 1:

```
sage: Partition([1, 1, 1]).is_k_reducible(2)
True
```

```
>>> from sage.all import *
>>> Partition([Integer(1), Integer(1), Integer(1)]).is_k_reducible(Integer(2))
True
```

The partition [1, 1, 1] does *not* have 4 rows of length 1, 3 rows of length 2, 2 rows of length 3, nor 1 row of length 4:

```
sage: Partition([1, 1, 1]).is_k_reducible(4)
False
```

```
>>> from sage.all import *
>>> Partition([Integer(1), Integer(1), Integer(1)]).is_k_reducible(Integer(4))
False
```

See also:

is_k_irreducible(), *has_k_rectangle()*

is_regular(*e*, *multicharge=(0,)*)

Return True is this is an e-regular partition.

A partition is e-regular if it does not have e equal nonzero parts.

EXAMPLES:

```
sage: Partition([4,3,3,3]).is_regular(2)
False
sage: Partition([4,3,3,3]).is_regular(3)
False
```

(continues on next page)

(continued from previous page)

```
sage: Partition([4,3,3,3]).is_regular(4)
True
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(3),Integer(3)]).is_
↪regular(Integer(2))
False
>>> Partition([Integer(4),Integer(3),Integer(3),Integer(3)]).is_
↪regular(Integer(3))
False
>>> Partition([Integer(4),Integer(3),Integer(3),Integer(3)]).is_
↪regular(Integer(4))
True
```

is_restricted(*e*, *multicharge=(0,)*)

Return `True` is this is an `e`-restricted partition.

An e-restricted partition is a partition such that the difference of consecutive parts is always strictly less than e, where partitions are considered to have an infinite number of 0 parts. I.e., the last part must be strictly less than e.

EXAMPLES:

```
sage: Partition([4,3,3,2]).is_restricted(2)
False
sage: Partition([4,3,3,2]).is_restricted(3)
True
sage: Partition([4,3,3,2]).is_restricted(4)
True
sage: Partition([4]).is_restricted(4)
False
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(3),Integer(2)]).is_
↪restricted(Integer(2))
False
>>> Partition([Integer(4),Integer(3),Integer(3),Integer(2)]).is_
↪restricted(Integer(3))
True
>>> Partition([Integer(4),Integer(3),Integer(3),Integer(2)]).is_
↪restricted(Integer(4))
True
>>> Partition([Integer(4)]).is_restricted(Integer(4))
False
```

is_symmetric()

Return `True` if the partition `self` equals its own transpose.

EXAMPLES:

```
sage: Partition([2, 1]).is_symmetric()
True
```

(continues on next page)

(continued from previous page)

```
sage: Partition([3, 1]).is_symmetric()
False
```

```
>>> from sage.all import *
>>> Partition([Integer(2), Integer(1)]).is_symmetric()
True
>>> Partition([Integer(3), Integer(1)]).is_symmetric()
False
```

jacobi_trudi()

Return the Jacobi-Trudi matrix of `self` thought of as a skew partition. See *SkewPartition.jacobi_trudi()*.

EXAMPLES:

```
sage: # needs sage.modules
sage: part = Partition([3,2,1])
sage: jt = part.jacobi_trudi(); jt
[h[3] h[1]    0]
[h[4] h[2]  h[]]
[h[5] h[3] h[1]]
sage: s = SymmetricFunctions(QQ).schur()
sage: h = SymmetricFunctions(QQ).homogeneous()
sage: h( s(part) )
h[3, 2, 1] - h[3, 3] - h[4, 1, 1] + h[5, 1]
sage: jt.det()
h[3, 2, 1] - h[3, 3] - h[4, 1, 1] + h[5, 1]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> part = Partition([Integer(3),Integer(2),Integer(1)])
>>> jt = part.jacobi_trudi(); jt
[h[3] h[1]    0]
[h[4] h[2]  h[]]
[h[5] h[3] h[1]]
>>> s = SymmetricFunctions(QQ).schur()
>>> h = SymmetricFunctions(QQ).homogeneous()
>>> h( s(part) )
h[3, 2, 1] - h[3, 3] - h[4, 1, 1] + h[5, 1]
>>> jt.det()
h[3, 2, 1] - h[3, 3] - h[4, 1, 1] + h[5, 1]
```

k_atom(*k*)

Return a list of the standard tableaux of size `self.size()` whose `k`-atom is equal to `self`.

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.k_atom(1)
[]
sage: p.k_atom(3)
[[[1, 1, 1, 2, 3], [2]],
```

(continues on next page)

(continued from previous page)

```
 [[1, 1, 1, 3], [2, 2]],
 [[1, 1, 1, 2], [2], [3]],
 [[1, 1, 1], [2, 2], [3]]]
sage: Partition([3,2,1]).k_atom(4)
[[[1, 1, 1, 3], [2, 2]], [[1, 1, 1], [2, 2], [3]]]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(3),Integer(2),Integer(1)])
>>> p.k_atom(Integer(1))
[]
>>> p.k_atom(Integer(3))
[[[1, 1, 1, 2, 3], [2]],
 [[1, 1, 1, 3], [2, 2]],
 [[1, 1, 1, 2], [2], [3]],
 [[1, 1, 1], [2, 2], [3]]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).k_atom(Integer(4))
[[[1, 1, 1, 3], [2, 2]], [[1, 1, 1], [2, 2], [3]]]
```

k_boundary(*k*)

Return the skew partition formed by removing the cells of the k-interior, see *k_interior()*.

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.k_boundary(2)
[3, 2, 1] / [2, 1]
sage: p.k_boundary(3)
[3, 2, 1] / [1]

sage: p = Partition([12,8,5,5,2,2,1])
sage: p.k_boundary(4)
[12, 8, 5, 5, 2, 2, 1] / [8, 5, 2, 2]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(3),Integer(2),Integer(1)])
>>> p.k_boundary(Integer(2))
[3, 2, 1] / [2, 1]
>>> p.k_boundary(Integer(3))
[3, 2, 1] / [1]

>>> p = Partition([Integer(12),Integer(8),Integer(5),Integer(5),Integer(2),
↪Integer(2),Integer(1)])
>>> p.k_boundary(Integer(4))
[12, 8, 5, 5, 2, 2, 1] / [8, 5, 2, 2]
```

k_column_lengths(*k*)

Return the k-column-shape of the partition self.

This is the 'column' analog of *k_row_lengths()*.

EXAMPLES:

```
sage: Partition([6, 1]).k_column_lengths(2)
[1, 0, 0, 0, 1, 1]

sage: Partition([4, 4, 4, 3, 2]).k_column_lengths(2)
[1, 1, 1, 2]
```

```
>>> from sage.all import *
>>> Partition([Integer(6), Integer(1)]).k_column_lengths(Integer(2))
[1, 0, 0, 0, 1, 1]

>>> Partition([Integer(4), Integer(4), Integer(4), Integer(3), Integer(2)]).k_
↪column_lengths(Integer(2))
[1, 1, 1, 2]
```

See also

k_row_lengths(), *k_boundary()*, *SkewPartition.row_lengths()*, *SkewPartition.column_lengths()*

k_conjugate(*k*)

Return the k-conjugate of self.

The k-conjugate is the partition that is given by the columns of the k-skew diagram of the partition.

We can also define the k-conjugate in the following way. Let P denote the bijection from $(k+1)$-cores to k-bounded partitions. The k-conjugate of a $(k+1)$-core λ is

$$\lambda^{(k)} = P^{-1}\left((P(\lambda))'\right).$$

EXAMPLES:

```
sage: p = Partition([4,3,2,2,1,1])
sage: p.k_conjugate(4)
[3, 2, 2, 1, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(4),Integer(3),Integer(2),Integer(2),Integer(1),
↪Integer(1)])
>>> p.k_conjugate(Integer(4))
[3, 2, 2, 1, 1, 1, 1, 1, 1]
```

k_interior(*k*)

Return the partition consisting of the cells of self whose hook lengths are greater than k.

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.hook_lengths()
[[5, 3, 1], [3, 1], [1]]
sage: p.k_interior(2)
[2, 1]
sage: p.k_interior(3)
```

(continues on next page)

(continued from previous page)

```
[1]

sage: p = Partition([])
sage: p.k_interior(3)
[]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(3),Integer(2),Integer(1)])
>>> p.hook_lengths()
[[5, 3, 1], [3, 1], [1]]
>>> p.k_interior(Integer(2))
[2, 1]
>>> p.k_interior(Integer(3))
[1]

>>> p = Partition([])
>>> p.k_interior(Integer(3))
[]
```

k_irreducible(k)

Return the partition with all $r \times (k + 1 - r)$ rectangles removed.

If `self` is a k-bounded partition, then this method will return the partition where all rectangles of dimension $r \times (k + 1 - r)$ for $1 \leq r \leq k$ have been deleted.

If `self` is not a k-bounded partition then the method will raise an error.

INPUT:

- `k` – nonnegative integer

OUTPUT: a partition

EXAMPLES:

```
sage: Partition([3,2,2,1,1,1]).k_irreducible(4)
[3, 2, 2, 1, 1, 1]
sage: Partition([3,2,2,1,1,1]).k_irreducible(3)
[]
sage: Partition([3,3,3,2,2,2,2,2,1,1,1,1]).k_irreducible(3)
[2, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(1)]).k_irreducible(Integer(4))
[3, 2, 2, 1, 1, 1]
>>> Partition([Integer(3),Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(1)]).k_irreducible(Integer(3))
[]
>>> Partition([Integer(3),Integer(3),Integer(3),Integer(2),Integer(2),
↪Integer(2),Integer(2),Integer(2),Integer(1),Integer(1),Integer(1),
↪Integer(1)]).k_irreducible(Integer(3))
[2, 1]
```

k_rim(*k*)

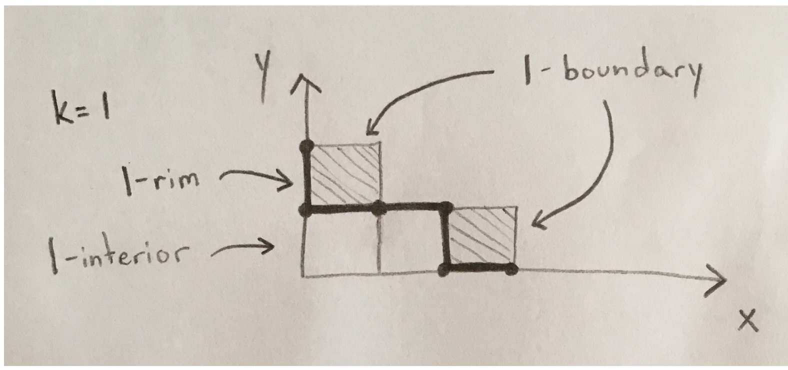
Return the k-rim of self as a list of integer coordinates.

The k-rim of a partition is the "line between" (or "intersection of") the k-boundary and the k-interior. (Section 2.3 of [HM2011])

It will be output as an ordered list of integer coordinates, where the origin is $(0, 0)$. It will start at the top-left of the k-rim (using French convention) and end at the bottom-right.

EXAMPLES:

Consider the partition (3, 1) split up into its 1-interior and 1-boundary:



The line shown in bold is the 1-rim, and that information is equivalent to the integer coordinates of the points that occur along that line:

```
sage: Partition([3, 1]).k_rim(1)
[(3, 0), (2, 0), (2, 1), (1, 1), (0, 1), (0, 2)]
```

```
>>> from sage.all import *
>>> Partition([Integer(3), Integer(1)]).k_rim(Integer(1))
[(3, 0), (2, 0), (2, 1), (1, 1), (0, 1), (0, 2)]
```

See also

k_interior(), *k_boundary()*, *boundary()*

k_row_lengths(*k*)

Return the k-row-shape of the partition self.

This is equivalent to taking the k-boundary of the partition and then returning the row-shape of that. We do *not* discard rows of length 0. (Section 2.2 of [LLMS2013])

EXAMPLES:

```
sage: Partition([6, 1]).k_row_lengths(2)
[2, 1]

sage: Partition([4, 4, 4, 3, 2]).k_row_lengths(2)
[0, 1, 1, 1, 2]
```

```
>>> from sage.all import *
>>> Partition([Integer(6), Integer(1)]).k_row_lengths(Integer(2))
```

(continues on next page)

(continued from previous page)

```
[2, 1]

>>> Partition([Integer(4), Integer(4), Integer(4), Integer(3), Integer(2)]).k_
→row_lengths(Integer(2))
[0, 1, 1, 1, 2]
```

See also

k_column_lengths(), *k_boundary()*, *SkewPartition.row_lengths()*, *SkewPartition.column_lengths()*

k_size(*k*)

Given a partition `self` and a `k`, return the size of the k-boundary.

This is the same as the length method *sage.combinat.core.Core.length()* of the *sage.combinat.core.Core* object, with the exception that here we don't require `self` to be a $k+1$-core.

EXAMPLES:

```
sage: Partition([2, 1, 1]).k_size(1)
2
sage: Partition([2, 1, 1]).k_size(2)
3
sage: Partition([2, 1, 1]).k_size(3)
3
sage: Partition([2, 1, 1]).k_size(4)
4
```

```
>>> from sage.all import *
>>> Partition([Integer(2), Integer(1), Integer(1)]).k_size(Integer(1))
2
>>> Partition([Integer(2), Integer(1), Integer(1)]).k_size(Integer(2))
3
>>> Partition([Integer(2), Integer(1), Integer(1)]).k_size(Integer(3))
3
>>> Partition([Integer(2), Integer(1), Integer(1)]).k_size(Integer(4))
4
```

See also

k_boundary(), *SkewPartition.size()*

k_skew(*k*)

Return the k-skew partition.

The k-skew diagram of a k-bounded partition is the skew diagram denoted $\lambda/^k$ satisfying the conditions:

1. row i of $\lambda/^k$ has length λ_i,
2. no cell in $\lambda/^k$ has hook-length exceeding k,
3. every square above the diagram of $\lambda/^k$ has hook length exceeding k.

REFERENCES:

- [LM2004]

EXAMPLES:

```
sage: p = Partition([4,3,2,2,1,1])
sage: p.k_skew(4)
[9, 5, 3, 2, 1, 1] / [5, 2, 1]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(4),Integer(3),Integer(2),Integer(2),Integer(1),
↪Integer(1)])
>>> p.k_skew(Integer(4))
[9, 5, 3, 2, 1, 1] / [5, 2, 1]
```

k_split(*k*)

Return the k-split of self.

EXAMPLES:

```
sage: Partition([4,3,2,1]).k_split(3)
[]
sage: Partition([4,3,2,1]).k_split(4)
[[4], [3, 2], [1]]
sage: Partition([4,3,2,1]).k_split(5)
[[4, 3], [2, 1]]
sage: Partition([4,3,2,1]).k_split(6)
[[4, 3, 2], [1]]
sage: Partition([4,3,2,1]).k_split(7)
[[4, 3, 2, 1]]
sage: Partition([4,3,2,1]).k_split(8)
[[4, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(2),Integer(1)]).k_
↪split(Integer(3))
[]
>>> Partition([Integer(4),Integer(3),Integer(2),Integer(1)]).k_
↪split(Integer(4))
[[4], [3, 2], [1]]
>>> Partition([Integer(4),Integer(3),Integer(2),Integer(1)]).k_
↪split(Integer(5))
[[4, 3], [2, 1]]
>>> Partition([Integer(4),Integer(3),Integer(2),Integer(1)]).k_
↪split(Integer(6))
[[4, 3, 2], [1]]
>>> Partition([Integer(4),Integer(3),Integer(2),Integer(1)]).k_
↪split(Integer(7))
[[4, 3, 2, 1]]
>>> Partition([Integer(4),Integer(3),Integer(2),Integer(1)]).k_
↪split(Integer(8))
[[4, 3, 2, 1]]
```

ladder_tableau(*e*, *ladder_lengths=False*)

Return the ladder tableau of shape `self`.

The e-*ladder tableau* is the standard Young tableau obtained by reading the *ladders*, the set of cells (i, j) that differ from $(i + e - 1, j - 1)$, of the partition λ from left-to-right.

INPUT:

- `e` – nonnegative integer; 0 is considered as ∞ (analogous to the characteristic of a ring)
- `ladder_sizes` – boolean (default: `False`); if `True`, also return the sizes of the ladders

See also

ladders()

EXAMPLES:

```
sage: la = Partition([6, 5, 3, 1])
sage: ascii_art(la.ladder_tableau(3))
  1  2  3  5  7 10
  4  6  8 11 13
  9 12 14
 15
sage: la.ladder_tableau(3, ladder_lengths=True)[1]
[1, 1, 2, 2, 3, 3, 3]

sage: ascii_art(la.ladder_tableau(0))
  1  2  3  4  5  6
  7  8  9 10 11
 12 13 14
 15
sage: all(ll == 1 for ll in la.ladder_tableau(0, ladder_lengths=True)[1])
True
```

```
>>> from sage.all import *
>>> la = Partition([Integer(6), Integer(5), Integer(3), Integer(1)])
>>> ascii_art(la.ladder_tableau(Integer(3)))
  1  2  3  5  7 10
  4  6  8 11 13
  9 12 14
 15
>>> la.ladder_tableau(Integer(3), ladder_lengths=True)[Integer(1)]
[1, 1, 2, 2, 3, 3, 3]

>>> ascii_art(la.ladder_tableau(Integer(0)))
  1  2  3  4  5  6
  7  8  9 10 11
 12 13 14
 15
>>> all(ll == Integer(1) for ll in la.ladder_tableau(Integer(0), ladder_
→lengths=True)[Integer(1)])
True
```

ladders(*e*)

Return a dictionary containing the ladders in the diagram of `self`.

For $e > 0$, a node (i, j) in a partition belongs to the l-th e-ladder if $l = (e - 1)r + c$.

INPUT:

- `e` – nonnegative integer; if `0`, then we set `e = self.size() + 1`

EXAMPLES:

```
sage: Partition([3, 2]).ladders(3)
{0: [(0, 0)], 1: [(0, 1)], 2: [(0, 2), (1, 0)], 3: [(1, 1)]}
```

```
>>> from sage.all import *
>>> Partition([Integer(3), Integer(2)]).ladders(Integer(3))
{0: [(0, 0)], 1: [(0, 1)], 2: [(0, 2), (1, 0)], 3: [(1, 1)]}
```

When `e` is `0`, the cells are in bijection with the ladders, but the index of the ladder depends on the size of the partition:

```
sage: Partition([3, 2]).ladders(0)
{0: [(0, 0)], 1: [(0, 1)], 2: [(0, 2)], 5: [(1, 0)], 6: [(1, 1)]}
sage: Partition([3, 2, 1]).ladders(0)
{0: [(0, 0)], 1: [(0, 1)], 2: [(0, 2)], 6: [(1, 0)], 7: [(1, 1)],
 12: [(2, 0)]}
sage: Partition([3, 1, 1]).ladders(0)
{0: [(0, 0)], 1: [(0, 1)], 2: [(0, 2)], 5: [(1, 0)], 10: [(2, 0)]}
sage: Partition([1, 1, 1]).ladders(0)
{0: [(0, 0)], 3: [(1, 0)], 6: [(2, 0)]}
```

```
>>> from sage.all import *
>>> Partition([Integer(3), Integer(2)]).ladders(Integer(0))
{0: [(0, 0)], 1: [(0, 1)], 2: [(0, 2)], 5: [(1, 0)], 6: [(1, 1)]}
>>> Partition([Integer(3), Integer(2), Integer(1)]).ladders(Integer(0))
{0: [(0, 0)], 1: [(0, 1)], 2: [(0, 2)], 6: [(1, 0)], 7: [(1, 1)],
 12: [(2, 0)]}
>>> Partition([Integer(3), Integer(1), Integer(1)]).ladders(Integer(0))
{0: [(0, 0)], 1: [(0, 1)], 2: [(0, 2)], 5: [(1, 0)], 10: [(2, 0)]}
>>> Partition([Integer(1), Integer(1), Integer(1)]).ladders(Integer(0))
{0: [(0, 0)], 3: [(1, 0)], 6: [(2, 0)]}
```

larger_lex(*rhs*)

Return `True` if `self` is larger than `rhs` in lexicographic order. Otherwise return `False`.

EXAMPLES:

```
sage: p = Partition([3,2])
sage: p.larger_lex([3,1])
True
sage: p.larger_lex([1,4])
True
sage: p.larger_lex([3,2,1])
False
sage: p.larger_lex([3])
```

(continues on next page)

(continued from previous page)

```
True
sage: p.larger_lex([5])
False
sage: p.larger_lex([3,1,1,1,1,1,1,1])
True
```

```
>>> from sage.all import *
>>> p = Partition([Integer(3),Integer(2)])
>>> p.larger_lex([Integer(3),Integer(1)])
True
>>> p.larger_lex([Integer(1),Integer(4)])
True
>>> p.larger_lex([Integer(3),Integer(2),Integer(1)])
False
>>> p.larger_lex([Integer(3)])
True
>>> p.larger_lex([Integer(5)])
False
>>> p.larger_lex([Integer(3),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(1),Integer(1)])
True
```

leg_cells(i, j)

Return the list of the cells of the leg of cell (i, j) in `self`.

The leg of cell $c = (i, j)$ is defined to be the cells below c (in English convention).

The cell coordinates are zero-based, i. e., the northwesternmost cell is $(0, 0)$.

INPUT:

- `i`, `j` – two integers

OUTPUT: list of pairs of integers

EXAMPLES:

```
sage: Partition([4,4,3,1]).leg_cells(1,1)
[(2, 1)]
sage: Partition([4,4,3,1]).leg_cells(0,1)
[(1, 1), (2, 1)]

sage: Partition([]).leg_cells(0,0)
Traceback (most recent call last):
...
ValueError: the cell is not in the diagram
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(4),Integer(3),Integer(1)]).leg_
↪cells(Integer(1),Integer(1))
[(2, 1)]
>>> Partition([Integer(4),Integer(4),Integer(3),Integer(1)]).leg_
↪cells(Integer(0),Integer(1))
[(1, 1), (2, 1)]
```

(continues on next page)

(continued from previous page)

```
>>> Partition([]).leg_cells(Integer(0),Integer(0))
Traceback (most recent call last):
...
ValueError: the cell is not in the diagram
```

leg_length(*i*, *j*)

Return the length of the leg of cell (i, j) in `self`.

The leg of cell $c = (i, j)$ is defined to be the cells below c (in English convention).

The cell coordinates are zero-based, i. e., the northwesternmost cell is $(0, 0)$.

INPUT:

- `i, j` – two integers

OUTPUT: integer or a `ValueError`

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.leg_length(0, 0)
2
sage: p.leg_length(0,1)
1
sage: p.leg_length(2,0)
0
sage: Partition([3,3]).leg_length(0, 0)
1
sage: cell = [0,0]; Partition([3,3]).leg_length(*cell)
1
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(2),Integer(1)])
>>> p.leg_length(Integer(0), Integer(0))
2
>>> p.leg_length(Integer(0),Integer(1))
1
>>> p.leg_length(Integer(2),Integer(0))
0
>>> Partition([Integer(3),Integer(3)]).leg_length(Integer(0), Integer(0))
1
>>> cell = [Integer(0),Integer(0)]; Partition([Integer(3),Integer(3)]).leg_
↪length(*cell)
1
```

leg_lengths(*flat=False*)

Return a tableau of shape `self` with each cell filled in with its leg length. The optional boolean parameter `flat` provides the option of returning a flat list.

EXAMPLES:

```
sage: Partition([2,2,1]).leg_lengths()
[[2, 1], [1, 0], [0]]
```

(continues on next page)

(continued from previous page)

```
sage: Partition([2,2,1]).leg_lengths(flat=True)
[2, 1, 1, 0, 0]
sage: Partition([3,3]).leg_lengths()
[[1, 1, 1], [0, 0, 0]]
sage: Partition([3,3]).leg_lengths(flat=True)
[1, 1, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).leg_lengths()
[[2, 1], [1, 0], [0]]
>>> Partition([Integer(2),Integer(2),Integer(1)]).leg_lengths(flat=True)
[2, 1, 1, 0, 0]
>>> Partition([Integer(3),Integer(3)]).leg_lengths()
[[1, 1, 1], [0, 0, 0]]
>>> Partition([Integer(3),Integer(3)]).leg_lengths(flat=True)
[1, 1, 1, 0, 0, 0]
```

length()

Return the number of parts in `self`.

EXAMPLES:

```
sage: Partition([3,2]).length()
2
sage: Partition([2,2,1]).length()
3
sage: Partition([]).length()
0
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2)]).length()
2
>>> Partition([Integer(2),Integer(2),Integer(1)]).length()
3
>>> Partition([]).length()
0
```

level()

Return the level of `self`, which is always 1.

This method exists only for compatibility with *PartitionTuples*.

EXAMPLES:

```
sage: Partition([4,3,2]).level()
1
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3),Integer(2)]).level()
1
```

lower_hook(i, j, *alpha*)

Return the lower hook length of the cell (i, j) in `self`. When `alpha = 1`, this is just the normal hook length.

The lower hook length of a cell (i, j) in a partition κ is defined by

$$h_*^\kappa(i,j) = \kappa_j' - i + 1 + \alpha(\kappa_i - j).$$

EXAMPLES:

```
sage: p = Partition([2,1])
sage: p.lower_hook(0,0,1)
3
sage: p.hook_length(0,0)
3
sage: [ p.lower_hook(i,j,x) for i,j in p.cells() ]                          #␣
↪needs sage.symbolic
[x + 2, 1, 1]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(1)])
>>> p.lower_hook(Integer(0),Integer(0),Integer(1))
3
>>> p.hook_length(Integer(0),Integer(0))
3
>>> [ p.lower_hook(i,j,x) for i,j in p.cells() ]                        #␣
↪needs sage.symbolic
[x + 2, 1, 1]
```

lower_hook_lengths(*alpha*)

Return a tableau of shape `self` with the cells filled in with the lower hook lengths. When `alpha = 1`, these are just the normal hook lengths.

The lower hook length of a cell (i, j) in a partition κ is defined by

$$h_*^\kappa(i,j) = \kappa_j' - i + 1 + \alpha(\kappa_i - j).$$

EXAMPLES:

```
sage: Partition([3,2,1]).lower_hook_lengths(x)                              #␣
↪needs sage.symbolic
[[2*x + 3, x + 2, 1], [x + 2, 1], [1]]
sage: Partition([3,2,1]).lower_hook_lengths(1)
[[5, 3, 1], [3, 1], [1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).lower_hook_lengths(x)      ␣
↪                        # needs sage.symbolic
[[2*x + 3, x + 2, 1], [x + 2, 1], [1]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).lower_hook_
↪lengths(Integer(1))
[[5, 3, 1], [3, 1], [1]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
```

next()

Return the partition that lexicographically follows `self`, of the same size. If `self` is the last partition, then return `False`.

EXAMPLES:

```
sage: next(Partition([4]))
[3, 1]
sage: next(Partition([1,1,1,1]))
False
```

```
>>> from sage.all import *
>>> next(Partition([Integer(4)]))
[3, 1]
>>> next(Partition([Integer(1),Integer(1),Integer(1),Integer(1)]))
False
```

next_within_bounds(*min=[]*, *max=None*, *partition_type=None*)

Get the next partition lexicographically that contains `min` and is contained in `max`.

INPUT:

- `min` – (default: `[]`, the empty partition) the 'minimum partition' that `next_within_bounds(self)` must contain
- `max` – (default: `None`) the 'maximum partition' that `next_within_bounds(self)` must be contained in; if set to `None`, then there is no restriction
- `partition_type` – (default: `None`) the type of partitions allowed; for example, 'strict' for strictly decreasing partitions, or `None` to allow any valid partition

EXAMPLES:

```
sage: m = [1, 1]
sage: M = [3, 2, 1]
sage: Partition([1, 1]).next_within_bounds(min=m, max=M)
[1, 1, 1]
sage: Partition([1, 1, 1]).next_within_bounds(min=m, max=M)
[2, 1]
sage: Partition([2, 1]).next_within_bounds(min=m, max=M)
[2, 1, 1]
sage: Partition([2, 1, 1]).next_within_bounds(min=m, max=M)
[2, 2]
sage: Partition([2, 2]).next_within_bounds(min=m, max=M)
[2, 2, 1]
sage: Partition([2, 2, 1]).next_within_bounds(min=m, max=M)
[3, 1]
sage: Partition([3, 1]).next_within_bounds(min=m, max=M)
[3, 1, 1]
sage: Partition([3, 1, 1]).next_within_bounds(min=m, max=M)
[3, 2]
sage: Partition([3, 2]).next_within_bounds(min=m, max=M)
[3, 2, 1]
sage: Partition([3, 2, 1]).next_within_bounds(min=m, max=M) == None
True
```

```
>>> from sage.all import *
>>> m = [Integer(1), Integer(1)]
>>> M = [Integer(3), Integer(2), Integer(1)]
>>> Partition([Integer(1), Integer(1)]).next_within_bounds(min=m, max=M)
[1, 1, 1]
>>> Partition([Integer(1), Integer(1), Integer(1)]).next_within_bounds(min=m,␣
↪max=M)
[2, 1]
>>> Partition([Integer(2), Integer(1)]).next_within_bounds(min=m, max=M)
[2, 1, 1]
>>> Partition([Integer(2), Integer(1), Integer(1)]).next_within_bounds(min=m,␣
↪max=M)
[2, 2]
>>> Partition([Integer(2), Integer(2)]).next_within_bounds(min=m, max=M)
[2, 2, 1]
>>> Partition([Integer(2), Integer(2), Integer(1)]).next_within_bounds(min=m,␣
↪max=M)
[3, 1]
>>> Partition([Integer(3), Integer(1)]).next_within_bounds(min=m, max=M)
[3, 1, 1]
>>> Partition([Integer(3), Integer(1), Integer(1)]).next_within_bounds(min=m,␣
↪max=M)
[3, 2]
>>> Partition([Integer(3), Integer(2)]).next_within_bounds(min=m, max=M)
[3, 2, 1]
>>> Partition([Integer(3), Integer(2), Integer(1)]).next_within_bounds(min=m,␣
↪max=M) == None
True
```

See also

next()

outer_rim()

Return the outer rim of `self`.

The outer rim of a partition λ is defined as the cells which do not belong to λ and which are adjacent to cells in λ.

EXAMPLES:

The outer rim of the partition $[4, 1]$ consists of the cells marked with `#` below:

```
****#
*####
##
```

```
sage: Partition([4,1]).outer_rim()
[(2, 0), (2, 1), (1, 1), (1, 2), (1, 3), (1, 4), (0, 4)]

sage: Partition([2,2,1]).outer_rim()
[(3, 0), (3, 1), (2, 1), (2, 2), (1, 2), (0, 2)]
sage: Partition([2,2]).outer_rim()
```

(continues on next page)

(continued from previous page)

```
[(2, 0), (2, 1), (2, 2), (1, 2), (0, 2)]
sage: Partition([6,3,3,1,1]).outer_rim()
[(5, 0), (5, 1), (4, 1), (3, 1), (3, 2), (3, 3), (2, 3), (1, 3), (1, 4), (1,␣
↪5), (1, 6), (0, 6)]
sage: Partition([]).outer_rim()
[(0, 0)]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(1)]).outer_rim()
[(2, 0), (2, 1), (1, 1), (1, 2), (1, 3), (1, 4), (0, 4)]

>>> Partition([Integer(2),Integer(2),Integer(1)]).outer_rim()
[(3, 0), (3, 1), (2, 1), (2, 2), (1, 2), (0, 2)]
>>> Partition([Integer(2),Integer(2)]).outer_rim()
[(2, 0), (2, 1), (2, 2), (1, 2), (0, 2)]
>>> Partition([Integer(6),Integer(3),Integer(3),Integer(1),Integer(1)]).outer_
↪rim()
[(5, 0), (5, 1), (4, 1), (3, 1), (3, 2), (3, 3), (2, 3), (1, 3), (1, 4), (1,␣
↪5), (1, 6), (0, 6)]
>>> Partition([]).outer_rim()
[(0, 0)]
```

outline(*variable=None*)

Return the outline of the partition `self`.

This is a piecewise linear function, normalized so that the area under the partition `[1]` is 2.

INPUT:

- `variable` – a variable (default: `'x'` in the symbolic ring)

EXAMPLES:

```
sage: # needs sage.symbolic
sage: [Partition([5,4]).outline()(x=i) for i in range(-10, 11)]
[10, 9, 8, 7, 6, 5, 6, 5, 6, 5, 4, 3, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: Partition([]).outline()
abs(x)
sage: Partition([1]).outline()
abs(x + 1) + abs(x - 1) - abs(x)
sage: y = SR.var("y")
sage: Partition([6,5,1]).outline(variable=y)
abs(y + 6) - abs(y + 5) + abs(y + 4) - abs(y + 3)
 + abs(y - 1) - abs(y - 2) + abs(y - 3)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> [Partition([Integer(5),Integer(4)]).outline()(x=i) for i in range(-
↪Integer(10), Integer(11))]
[10, 9, 8, 7, 6, 5, 6, 5, 6, 5, 4, 3, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> Partition([]).outline()
abs(x)
>>> Partition([Integer(1)]).outline()
```

(continues on next page)

(continued from previous page)

```
abs(x + 1) + abs(x - 1) - abs(x)
>>> y = SR.var("y")
>>> Partition([Integer(6),Integer(5),Integer(1)]).outline(variable=y)
abs(y + 6) - abs(y + 5) + abs(y + 4) - abs(y + 3)
 + abs(y - 1) - abs(y - 2) + abs(y - 3)
```

outside_corners()

Return a list of the outside corners of the partition `self`.

An outside corner (also called a cocorner) of a partition λ is a cell on $\mathbf{Z}^2$ which does not belong to the Young diagram of λ but can be added to this Young diagram to still form a straight-shape Young diagram.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

Note

These are called "outer corners" in [Sag2001].

EXAMPLES:

```
sage: Partition([2,2,1]).outside_corners()
[(0, 2), (2, 1), (3, 0)]
sage: Partition([2,2]).outside_corners()
[(0, 2), (2, 0)]
sage: Partition([6,3,3,1,1,1]).outside_corners()
[(0, 6), (1, 3), (3, 1), (6, 0)]
sage: Partition([]).outside_corners()
[(0, 0)]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).outside_corners()
[(0, 2), (2, 1), (3, 0)]
>>> Partition([Integer(2),Integer(2)]).outside_corners()
[(0, 2), (2, 0)]
>>> Partition([Integer(6),Integer(3),Integer(3),Integer(1),Integer(1),
↪Integer(1)]).outside_corners()
[(0, 6), (1, 3), (3, 1), (6, 0)]
>>> Partition([]).outside_corners()
[(0, 0)]
```

outside_corners_residue(i, l)

Return a list of the outside corners of the partition `self` having `l`-residue `i`.

An outside corner (also called a cocorner) of a partition λ is a cell on $\mathbf{Z}^2$ which does not belong to the Young diagram of λ but can be added to this Young diagram to still form a straight-shape Young diagram. See *residue()* for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).outside_corners_residue(0, 3)
[(0, 3), (3, 0)]
sage: Partition([3,2,1]).outside_corners_residue(1, 3)
[(1, 2)]
sage: Partition([3,2,1]).outside_corners_residue(2, 3)
[(2, 1)]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).outside_corners_
↪residue(Integer(0), Integer(3))
[(0, 3), (3, 0)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).outside_corners_
↪residue(Integer(1), Integer(3))
[(1, 2)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).outside_corners_
↪residue(Integer(2), Integer(3))
[(2, 1)]
```

plancherel_measure()

Return the probability of `self` under the Plancherel probability measure on partitions of the same size.

This probability distribution comes from the uniform distribution on permutations via the Robinson-Schensted correspondence.

See Wikipedia article Plancherel_measure and *Partitions_n.random_element_plancherel()*.

EXAMPLES:

```
sage: Partition([]).plancherel_measure()
1
sage: Partition([1]).plancherel_measure()
1
sage: Partition([2]).plancherel_measure()
1/2
sage: [mu.plancherel_measure() for mu in Partitions(3)]
[1/6, 2/3, 1/6]
sage: Partition([5,4]).plancherel_measure()
7/1440
```

```
>>> from sage.all import *
>>> Partition([]).plancherel_measure()
1
>>> Partition([Integer(1)]).plancherel_measure()
1
>>> Partition([Integer(2)]).plancherel_measure()
1/2
>>> [mu.plancherel_measure() for mu in Partitions(Integer(3))]
[1/6, 2/3, 1/6]
>>> Partition([Integer(5),Integer(4)]).plancherel_measure()
7/1440
```

power(k)

Return the cycle type of the k-th power of any permutation with cycle type `self` (thus describes the powermap of symmetric groups).

Equivalent to GAP's `PowerPartition`.

EXAMPLES:

```
sage: p = Partition([5,3])
sage: p.power(1)
[5, 3]
sage: p.power(2)
[5, 3]
sage: p.power(3)
[5, 1, 1, 1]
sage: p.power(4)
[5, 3]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(5),Integer(3)])
>>> p.power(Integer(1))
[5, 3]
>>> p.power(Integer(2))
[5, 3]
>>> p.power(Integer(3))
[5, 1, 1, 1]
>>> p.power(Integer(4))
[5, 3]
```

Now let us compare this to the power map on S_8:

```
sage: # needs sage.groups
sage: G = SymmetricGroup(8)
sage: g = G([(1,2,3,4,5),(6,7,8)]); g
(1,2,3,4,5)(6,7,8)
sage: g^2
(1,3,5,2,4)(6,8,7)
sage: g^3
(1,4,2,5,3)
sage: g^4
(1,5,4,3,2)(6,7,8)
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> G = SymmetricGroup(Integer(8))
>>> g = G([(Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)),
↪(Integer(6),Integer(7),Integer(8))]); g
(1,2,3,4,5)(6,7,8)
>>> g**Integer(2)
(1,3,5,2,4)(6,8,7)
>>> g**Integer(3)
(1,4,2,5,3)
>>> g**Integer(4)
(1,5,4,3,2)(6,7,8)
```

```
sage: Partition([3,2,1]).power(3)
[2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).power(Integer(3))
[2, 1, 1, 1, 1]
```

pp()

Print the Ferrers diagram.

See *ferrers_diagram()* for more on the Ferrers diagram.

EXAMPLES:

```
sage: Partition([5,5,2,1]).pp()
*****
*****
**
*
sage: Partitions.options.convention='French'
sage: Partition([5,5,2,1]).pp()
*
**
*****
*****
sage: Partitions.options._reset()
```

```
>>> from sage.all import *
>>> Partition([Integer(5),Integer(5),Integer(2),Integer(1)]).pp()
*****
*****
**
*
>>> Partitions.options.convention='French'
>>> Partition([Integer(5),Integer(5),Integer(2),Integer(1)]).pp()
*
**
*****
*****
>>> Partitions.options._reset()
```

prime_degree(*p*)

Return the prime degree for the prime integer``p`` for `self`.

INPUT:

- `p` – prime integer

OUTPUT: nonnegative integer

The degree of a partition λ is the sum of the e-*degree()* of the standard tableaux of shape λ, for e a power of the prime p. The prime degree gives the exponent of p in the Gram determinant of the integral Specht module of the symmetric group.

EXAMPLES:

```
sage: Partition([4,3]).prime_degree(2)
36
sage: Partition([4,3]).prime_degree(3)
```

(continues on next page)

(continued from previous page)

```
15
sage: Partition([4,3]).prime_degree(5)
13
sage: Partition([4,3]).prime_degree(7)
0
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3)]).prime_degree(Integer(2))
36
>>> Partition([Integer(4),Integer(3)]).prime_degree(Integer(3))
15
>>> Partition([Integer(4),Integer(3)]).prime_degree(Integer(5))
13
>>> Partition([Integer(4),Integer(3)]).prime_degree(Integer(7))
0
```

Therefore, the Gram determinant of $S(5,3)$ when $q = 1$ is $2^{36}3^{15}5^{13}$. Compare with `degree()`.

quotient(*length*)

Return the quotient of the partition – in the literature the quotient is commonly referred to as the k-quotient, p-quotient, r-quotient, … .

The r-quotient of a partition λ is a list of r partitions (labelled from 0 to $r-1$), constructed in the following way. Label each cell in the Young diagram of λ with its content modulo r. Let R_i be the set of rows ending in a cell labelled i, and C_i be the set of columns ending in a cell labelled i. Then the j-th component of the quotient of λ is the partition defined by intersecting R_j with C_{j+1}. (See Theorem 2.7.37 in [JK1981].)

EXAMPLES:

```
sage: Partition([7,7,5,3,3,3,1]).quotient(3)
([2], [1], [2, 2, 2])
```

```
>>> from sage.all import *
>>> Partition([Integer(7),Integer(7),Integer(5),Integer(3),Integer(3),
↪Integer(3),Integer(1)]).quotient(Integer(3))
([2], [1], [2, 2, 2])
```

reading_tableau()

Return the RSK recording tableau of the reading word of the (standard) tableau T labeled down (in English convention) each column to the shape of `self`.

For an example of the tableau T, consider the partition $\lambda = (3,2,1)$, then we have:

```
1 4 6
2 5
3
```

For more, see `RSK()`.

EXAMPLES:

```
sage: Partition([3,2,1]).reading_tableau()
[[1, 3, 6], [2, 5], [4]]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).reading_tableau()
[[1, 3, 6], [2, 5], [4]]
```

removable_cells()

Return a list of the corners of the partition `self`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

> **Note**
>
> This is referred to as an "inner corner" in [Sag2001].

EXAMPLES:

```
sage: Partition([3,2,1]).corners()
[(0, 2), (1, 1), (2, 0)]
sage: Partition([3,3,1]).corners()
[(1, 2), (2, 0)]
sage: Partition([]).corners()
[]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners()
[(0, 2), (1, 1), (2, 0)]
>>> Partition([Integer(3),Integer(3),Integer(1)]).corners()
[(1, 2), (2, 0)]
>>> Partition([]).corners()
[]
```

removable_cells_residue(i, l)

Return a list of the corners of the partition `self` having `l`-residue `i`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind. See `residue()` for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners_residue(0, 3)
[(1, 1)]
sage: Partition([3,2,1]).corners_residue(1, 3)
[(2, 0)]
sage: Partition([3,2,1]).corners_residue(2, 3)
[(0, 2)]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(0),␣
↪Integer(3))
[(1, 1)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(1),␣
↪Integer(3))
[(2, 0)]
>>> Partition([Integer(3),Integer(2),Integer(1)]).corners_residue(Integer(2),␣
↪Integer(3))
[(0, 2)]
```

remove_cell(*i*, *j=None*)

Return the partition obtained by removing a cell at the end of row `i` of `self`.

EXAMPLES:

```
sage: Partition([2,2]).remove_cell(1)
[2, 1]
sage: Partition([2,2,1]).remove_cell(2)
[2, 2]
sage: #Partition([2,2]).remove_cell(0)
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2)]).remove_cell(Integer(1))
[2, 1]
>>> Partition([Integer(2),Integer(2),Integer(1)]).remove_cell(Integer(2))
[2, 2]
>>> #Partition([2,2]).remove_cell(0)
```

```
sage: Partition([2,2]).remove_cell(1,1)
[2, 1]
sage: #Partition([2,2]).remove_cell(1,0)
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2)]).remove_cell(Integer(1),Integer(1))
[2, 1]
>>> #Partition([2,2]).remove_cell(1,0)
```

remove_horizontal_border_strip(*k*)

Return the partitions obtained from `self` by removing an horizontal border strip of length `k`.

EXAMPLES:

```
sage: Partition([5,3,1]).remove_horizontal_border_strip(0).list()
[[5, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(1).list()
[[5, 3], [5, 2, 1], [4, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(2).list()
[[5, 2], [5, 1, 1], [4, 3], [4, 2, 1], [3, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(3).list()
[[5, 1], [4, 2], [4, 1, 1], [3, 3], [3, 2, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(4).list()
[[4, 1], [3, 2], [3, 1, 1]]
```

(continues on next page)

(continued from previous page)

```
sage: Partition([5,3,1]).remove_horizontal_border_strip(5).list()
[[3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(6).list()
[]
```

```
>>> from sage.all import *
>>> Partition([Integer(5),Integer(3),Integer(1)]).remove_horizontal_border_
↪strip(Integer(0)).list()
[[5, 3, 1]]
>>> Partition([Integer(5),Integer(3),Integer(1)]).remove_horizontal_border_
↪strip(Integer(1)).list()
[[5, 3], [5, 2, 1], [4, 3, 1]]
>>> Partition([Integer(5),Integer(3),Integer(1)]).remove_horizontal_border_
↪strip(Integer(2)).list()
[[5, 2], [5, 1, 1], [4, 3], [4, 2, 1], [3, 3, 1]]
>>> Partition([Integer(5),Integer(3),Integer(1)]).remove_horizontal_border_
↪strip(Integer(3)).list()
[[5, 1], [4, 2], [4, 1, 1], [3, 3], [3, 2, 1]]
>>> Partition([Integer(5),Integer(3),Integer(1)]).remove_horizontal_border_
↪strip(Integer(4)).list()
[[4, 1], [3, 2], [3, 1, 1]]
>>> Partition([Integer(5),Integer(3),Integer(1)]).remove_horizontal_border_
↪strip(Integer(5)).list()
[[3, 1]]
>>> Partition([Integer(5),Integer(3),Integer(1)]).remove_horizontal_border_
↪strip(Integer(6)).list()
[]
```

The result is returned as an instance of *Partitions_with_constraints*:

```
sage: Partition([5,3,1]).remove_horizontal_border_strip(5)
The subpartitions of [5, 3, 1] obtained by removing a horizontal border strip␣
↪of length 5
```

```
>>> from sage.all import *
>>> Partition([Integer(5),Integer(3),Integer(1)]).remove_horizontal_border_
↪strip(Integer(5))
The subpartitions of [5, 3, 1] obtained by removing a horizontal border strip␣
↪of length 5
```

residue(r, c, l)

Return the `l`-residue of the cell at row `r` and column `c`.

The ℓ-residue of a cell is $c - r$ modulo ℓ.

This does not strictly depend upon the partition, however, this method is included because it is often useful in the context of partitions.

EXAMPLES:

```
sage: Partition([2,1]).residue(1, 0, 3)
2
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(1)]).residue(Integer(1), Integer(0),␣
↪Integer(3))
2
```

rim()

Return the rim of `self`.

The rim of a partition λ is defined as the cells which belong to λ and which are adjacent to cells not in λ.

EXAMPLES:

The rim of the partition $[5, 5, 2, 1]$ consists of the cells marked with `#` below:

```
****#
*####
##
#

sage: Partition([5,5,2,1]).rim()
[(3, 0), (2, 0), (2, 1), (1, 1), (1, 2), (1, 3), (1, 4), (0, 4)]

sage: Partition([2,2,1]).rim()
[(2, 0), (1, 0), (1, 1), (0, 1)]
sage: Partition([2,2]).rim()
[(1, 0), (1, 1), (0, 1)]
sage: Partition([6,3,3,1,1]).rim()
[(4, 0), (3, 0), (2, 0), (2, 1), (2, 2), (1, 2), (0, 2), (0, 3), (0, 4), (0,␣
↪5)]
sage: Partition([]).rim()
[]
```

row_standard_tableaux()

Return the *row standard tableaux* of shape `self`.

EXAMPLES:

```
sage: Partition([3,2,2,1]).row_standard_tableaux()
Row standard tableaux of shape [3, 2, 2, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(2),Integer(1)]).row_standard_
↪tableaux()
Row standard tableaux of shape [3, 2, 2, 1]
```

sign()

Return the sign of any permutation with cycle type `self`.

This function corresponds to a homomorphism from the symmetric group S_n into the cyclic group of order 2, whose kernel is exactly the alternating group A_n. Partitions of sign 1 are called even partitions while partitions of sign -1 are called odd.

EXAMPLES:

```
sage: Partition([5,3]).sign()
1
sage: Partition([5,2]).sign()
-1
```

```
>>> from sage.all import *
>>> Partition([Integer(5),Integer(3)]).sign()
1
>>> Partition([Integer(5),Integer(2)]).sign()
-1
```

Zolotarev's lemma states that the Legendre symbol $\left(\frac{a}{p}\right)$ for an integer $a \pmod p$ (p a prime number), can be computed as sign(p_a), where sign denotes the sign of a permutation and p_a the permutation of the residue classes $\pmod p$ induced by modular multiplication by a, provided p does not divide a.

We verify this in some examples.

```
sage: F = GF(11)                                                    #␣
↪needs sage.rings.finite_rings
sage: a = F.multiplicative_generator();a                            #␣
↪needs sage.rings.finite_rings
2
sage: plist = [int(a*F(x)) for x in range(1,11)]; plist             #␣
↪needs sage.rings.finite_rings
[2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
```

```
>>> from sage.all import *
>>> F = GF(Integer(11))                                              ␣
↪     # needs sage.rings.finite_rings
>>> a = F.multiplicative_generator();a                            #␣
↪needs sage.rings.finite_rings
2
>>> plist = [int(a*F(x)) for x in range(Integer(1),Integer(11))]; plist     ␣
↪              # needs sage.rings.finite_rings
[2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
```

This corresponds to the permutation (1, 2, 4, 8, 5, 10, 9, 7, 3, 6) (acting the set $\{1, 2, ..., 10\}$) and to the partition [10].

```
sage: p = PermutationGroupElement('(1, 2, 4, 8, 5, 10, 9, 7, 3, 6)')   #␣
↪needs sage.groups
sage: p.sign()                                                      #␣
↪needs sage.groups
-1
sage: Partition([10]).sign()
-1
sage: kronecker_symbol(11,2)
-1
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement('(1, 2, 4, 8, 5, 10, 9, 7, 3, 6)')   #␣
↪needs sage.groups
>>> p.sign()                                                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
-1
>>> Partition([Integer(10)]).sign()
-1
>>> kronecker_symbol(Integer(11),Integer(2))
-1
```

Now replace 2 by 3:

```
sage: plist = [int(F(3*x)) for x in range(1,11)]; plist                     #␣
↪needs sage.rings.finite_rings
[3, 6, 9, 1, 4, 7, 10, 2, 5, 8]
sage: list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: p = PermutationGroupElement('(3,4,8,7,9)')                             #␣
↪needs sage.groups
sage: p.sign()                                                               #␣
↪needs sage.groups
1
sage: kronecker_symbol(3,11)
1
sage: Partition([5,1,1,1,1,1]).sign()
1
```

```
>>> from sage.all import *
>>> plist = [int(F(Integer(3)*x)) for x in range(Integer(1),Integer(11))];␣
↪plist                      # needs sage.rings.finite_rings
[3, 6, 9, 1, 4, 7, 10, 2, 5, 8]
>>> list(range(Integer(1), Integer(11)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> p = PermutationGroupElement('(3,4,8,7,9)')                             #␣
↪needs sage.groups
>>> p.sign()                                                               #␣
↪needs sage.groups
1
>>> kronecker_symbol(Integer(3),Integer(11))
1
>>> Partition([Integer(5),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1)]).sign()
1
```

In both cases, Zolotarev holds.

REFERENCES:

- Wikipedia article Zolotarev%27s_lemma

simple_module_dimension(*base_ring=None*)

Return the dimension of the simple module corresponding to `self`.

When the base ring is a field of characteristic 0, this is equal to the dimension of the Specht module.

INPUT:

- `base_ring` – (default: $\mathbf{Q}$) the base ring

EXAMPLES:

```
sage: Partition([2,2,1]).simple_module_dimension()
5
sage: Partition([2,2,1]).specht_module_dimension(GF(3))                    #␣
↪needs sage.rings.finite_rings
5
sage: Partition([2,2,1]).simple_module_dimension(GF(3))                    #␣
↪needs sage.rings.finite_rings
4

sage: for la in Partitions(6, regular=3):
....:     print(la, la.specht_module_dimension(), la.simple_module_
↪dimension(GF(3)))
[6] 1 1
[5, 1] 5 4
[4, 2] 9 9
[4, 1, 1] 10 6
[3, 3] 5 1
[3, 2, 1] 16 4
[2, 2, 1, 1] 9 9
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).simple_module_dimension()
5
>>> Partition([Integer(2),Integer(2),Integer(1)]).specht_module_
↪dimension(GF(Integer(3)))                          # needs sage.rings.finite_
↪rings
5
>>> Partition([Integer(2),Integer(2),Integer(1)]).simple_module_
↪dimension(GF(Integer(3)))                          # needs sage.rings.finite_
↪rings
4

>>> for la in Partitions(Integer(6), regular=Integer(3)):
...     print(la, la.specht_module_dimension(), la.simple_module_
↪dimension(GF(Integer(3))))
[6] 1 1
[5, 1] 5 4
[4, 2] 9 9
[4, 1, 1] 10 6
[3, 3] 5 1
[3, 2, 1] 16 4
[2, 2, 1, 1] 9 9
```

size()

Return the size of `self`.

EXAMPLES:

```
sage: Partition([2,2]).size()
4
sage: Partition([3,2,1]).size()
6
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2)]).size()
4
>>> Partition([Integer(3),Integer(2),Integer(1)]).size()
6
```

specht_module(*base_ring=None*)

Return the Specht module corresponding to `self`.

EXAMPLES:

```
sage: SM = Partition([2,2,1]).specht_module(QQ); SM                       #␣
↪needs sage.modules
Specht module of [2, 2, 1] over Rational Field
sage: SM.frobenius_image()                                                #␣
↪needs sage.modules
s[2, 2, 1]
```

```
>>> from sage.all import *
>>> SM = Partition([Integer(2),Integer(2),Integer(1)]).specht_module(QQ); SM ␣
↪                         # needs sage.modules
Specht module of [2, 2, 1] over Rational Field
>>> SM.frobenius_image()                                                 #␣
↪needs sage.modules
s[2, 2, 1]
```

specht_module_dimension(*base_ring=None*)

Return the dimension of the Specht module corresponding to `self`.

This is equal to the number of standard tableaux of shape `self` when over a field of characteristic 0.

INPUT:

- `base_ring` – (default: $\mathbf{Q}$) the base ring

EXAMPLES:

```
sage: Partition([2,2,1]).specht_module_dimension()
5
sage: Partition([2,2,1]).specht_module_dimension(GF(2))                   #␣
↪needs sage.rings.finite_rings
5
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2),Integer(1)]).specht_module_dimension()
5
>>> Partition([Integer(2),Integer(2),Integer(1)]).specht_module_
↪dimension(GF(Integer(2)))                      # needs sage.rings.finite_
↪rings
5
```

standard_tableaux()

Return the *standard tableaux* of shape `self`.

EXAMPLES:

```
sage: Partition([3,2,2,1]).standard_tableaux()
Standard tableaux of shape [3, 2, 2, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(2),Integer(1)]).standard_
↪tableaux()
Standard tableaux of shape [3, 2, 2, 1]
```

stretch(*k*)

Return the partition obtained by multiplying each part with the given number.

EXAMPLES:

```
sage: p = Partition([4,2,2,1,1])
sage: p.stretch(3)
[12, 6, 6, 3, 3]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(4),Integer(2),Integer(2),Integer(1),Integer(1)])
>>> p.stretch(Integer(3))
[12, 6, 6, 3, 3]
```

suter_diagonal_slide(*n*, *exp=1*)

Return the image of `self` in Y_n under Suter's diagonal slide σ_n, where the notations used are those defined in [Sut2002].

The set Y_n is defined as the set of all partitions λ such that the hook length of the $(0, 0)$-cell (i.e. the northwestern most cell in English notation) of λ is less than n, including the empty partition.

The map σ_n sends a partition (with nonzero entries) $(\lambda_1, \lambda_2, \ldots, \lambda_m) \in Y_n$ to the partition $(\lambda_2 + 1, \lambda_3 + 1, \ldots, \lambda_m + 1, \underbrace{1, 1, \ldots, 1}_{n-m-\lambda_1 \text{ ones}})$. In other words, it pads the partition with trailing zeroes until it has length $n - \lambda_1$, then removes its first part, and finally adds 1 to each part.

By Theorem 2.1 of [Sut2002], the dihedral group D_n with $2n$ elements acts on Y_n by letting the primitive rotation act as σ_n and the reflection act as conjugation of partitions (`conjugate()`). This action is faithful if $n \geq 3$.

INPUT:

- `n` – nonnegative integer
- `exp` – (default: 1) how many times σ_n should be applied

OUTPUT:

The result of applying Suter's diagonal slide σ_n to `self`, assuming that `self` lies in Y_n. If the optional argument `exp` is set, then the slide σ_n is applied not just once, but `exp` times (note that `exp` is allowed to be negative, since the slide has finite order).

EXAMPLES:

```
sage: Partition([5,4,1]).suter_diagonal_slide(8)
[5, 2]
sage: Partition([5,4,1]).suter_diagonal_slide(9)
[5, 2, 1]
sage: Partition([]).suter_diagonal_slide(7)
```

(continues on next page)

(continued from previous page)

```
[1, 1, 1, 1, 1, 1]
sage: Partition([]).suter_diagonal_slide(1)
[]
sage: Partition([]).suter_diagonal_slide(7, exp=-1)
[6]
sage: Partition([]).suter_diagonal_slide(1, exp=-1)
[]
sage: P7 = Partitions(7)
sage: all( p == p.suter_diagonal_slide(9, exp=-1).suter_diagonal_slide(9)
....:      for p in P7 )
True
sage: all( p == p.suter_diagonal_slide(9, exp=3)
....:             .suter_diagonal_slide(9, exp=3)
....:             .suter_diagonal_slide(9, exp=3)
....:      for p in P7 )
True
sage: all( p == p.suter_diagonal_slide(9, exp=6)
....:             .suter_diagonal_slide(9, exp=6)
....:             .suter_diagonal_slide(9, exp=6)
....:      for p in P7 )
True
sage: all( p == p.suter_diagonal_slide(9, exp=-1)
....:             .suter_diagonal_slide(9, exp=1)
....:      for p in P7 )
True
```

```
>>> from sage.all import *
>>> Partition([Integer(5),Integer(4),Integer(1)]).suter_diagonal_
↪slide(Integer(8))
[5, 2]
>>> Partition([Integer(5),Integer(4),Integer(1)]).suter_diagonal_
↪slide(Integer(9))
[5, 2, 1]
>>> Partition([]).suter_diagonal_slide(Integer(7))
[1, 1, 1, 1, 1, 1]
>>> Partition([]).suter_diagonal_slide(Integer(1))
[]
>>> Partition([]).suter_diagonal_slide(Integer(7), exp=-Integer(1))
[6]
>>> Partition([]).suter_diagonal_slide(Integer(1), exp=-Integer(1))
[]
>>> P7 = Partitions(Integer(7))
>>> all( p == p.suter_diagonal_slide(Integer(9), exp=-Integer(1)).suter_
↪diagonal_slide(Integer(9))
...      for p in P7 )
True
>>> all( p == p.suter_diagonal_slide(Integer(9), exp=Integer(3))
...             .suter_diagonal_slide(Integer(9), exp=Integer(3))
...             .suter_diagonal_slide(Integer(9), exp=Integer(3))
...      for p in P7 )
True
>>> all( p == p.suter_diagonal_slide(Integer(9), exp=Integer(6))
```

(continues on next page)

(continued from previous page)

```
...             .suter_diagonal_slide(Integer(9), exp=Integer(6))
...             .suter_diagonal_slide(Integer(9), exp=Integer(6))
...       for p in P7 )
True
>>> all( p == p.suter_diagonal_slide(Integer(9), exp=-Integer(1))
...             .suter_diagonal_slide(Integer(9), exp=Integer(1))
...       for p in P7 )
True
```

Check of the assertion in [Sut2002] that $\sigma_n\big(\sigma_n(\lambda')'\big) = \lambda$:

```
sage: all( p.suter_diagonal_slide(8).conjugate()
....:       == p.conjugate().suter_diagonal_slide(8, exp=-1)
....:       for p in P7 )
True
```

```
>>> from sage.all import *
>>> all( p.suter_diagonal_slide(Integer(8)).conjugate()
...       == p.conjugate().suter_diagonal_slide(Integer(8), exp=-Integer(1))
...       for p in P7 )
True
```

Check of Claim 1 in [Sut2002]:

```
sage: P5 = Partitions(5)
sage: all( all( (p.suter_diagonal_slide(6) in q.suter_diagonal_slide(6).
↪down())
....:             or (q.suter_diagonal_slide(6) in p.suter_diagonal_slide(6).
↪down())
....:             for p in q.down() )
....:       for q in P5 )
True
```

```
>>> from sage.all import *
>>> P5 = Partitions(Integer(5))
>>> all( all( (p.suter_diagonal_slide(Integer(6)) in q.suter_diagonal_
↪slide(Integer(6)).down())
...             or (q.suter_diagonal_slide(Integer(6)) in p.suter_diagonal_
↪slide(Integer(6)).down())
...             for p in q.down() )
...       for q in P5 )
True
```

t_completion(*t*)

Return the t-completion of the partition self.

If $\lambda = (\lambda_1, \lambda_2, \lambda_3, \ldots)$ is a partition and t is an integer greater or equal to $|\lambda| + \lambda_1$, then the *t-completion of* λ is defined as the partition $(t - |\lambda|, \lambda_1, \lambda_2, \lambda_3, \ldots)$ of t. This partition is denoted by $\lambda[t]$ in [BOR2009], by $\lambda_{[t]}$ in [BdVO2012], and by $\lambda(t)$ in [CO2010].

EXAMPLES:

```
sage: Partition([]).t_completion(0)
[]
sage: Partition([]).t_completion(1)
[1]
sage: Partition([]).t_completion(2)
[2]
sage: Partition([]).t_completion(3)
[3]
sage: Partition([2, 1]).t_completion(5)
[2, 2, 1]
sage: Partition([2, 1]).t_completion(6)
[3, 2, 1]
sage: Partition([4, 2, 2, 1]).t_completion(13)
[4, 4, 2, 2, 1]
sage: Partition([4, 2, 2, 1]).t_completion(19)
[10, 4, 2, 2, 1]
sage: Partition([4, 2, 2, 1]).t_completion(10)
Traceback (most recent call last):
...
ValueError: 10-completion is not defined
sage: Partition([4, 2, 2, 1]).t_completion(5)
Traceback (most recent call last):
...
ValueError: 5-completion is not defined
```

```
>>> from sage.all import *
>>> Partition([]).t_completion(Integer(0))
[]
>>> Partition([]).t_completion(Integer(1))
[1]
>>> Partition([]).t_completion(Integer(2))
[2]
>>> Partition([]).t_completion(Integer(3))
[3]
>>> Partition([Integer(2), Integer(1)]).t_completion(Integer(5))
[2, 2, 1]
>>> Partition([Integer(2), Integer(1)]).t_completion(Integer(6))
[3, 2, 1]
>>> Partition([Integer(4), Integer(2), Integer(2), Integer(1)]).t_
↪completion(Integer(13))
[4, 4, 2, 2, 1]
>>> Partition([Integer(4), Integer(2), Integer(2), Integer(1)]).t_
↪completion(Integer(19))
[10, 4, 2, 2, 1]
>>> Partition([Integer(4), Integer(2), Integer(2), Integer(1)]).t_
↪completion(Integer(10))
Traceback (most recent call last):
...
ValueError: 10-completion is not defined
>>> Partition([Integer(4), Integer(2), Integer(2), Integer(1)]).t_
↪completion(Integer(5))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: 5-completion is not defined
```

tabloid_module(*base_ring=None*)

Return the tabloid module corresponding to `self`.

EXAMPLES:

```
sage: TM = Partition([2,2,1]).tabloid_module(QQ); TM
Tabloid module of [2, 2, 1] over Rational Field
sage: TM.frobenius_image()
s[2, 2, 1] + s[3, 1, 1] + 2*s[3, 2] + 2*s[4, 1] + s[5]
```

```
>>> from sage.all import *
>>> TM = Partition([Integer(2),Integer(2),Integer(1)]).tabloid_module(QQ); TM
Tabloid module of [2, 2, 1] over Rational Field
>>> TM.frobenius_image()
s[2, 2, 1] + s[3, 1, 1] + 2*s[3, 2] + 2*s[4, 1] + s[5]
```

to_core(*k*)

Map the k-bounded partition `self` to its corresponding $k+1$-core.

See also `k_skew()`.

EXAMPLES:

```
sage: p = Partition([4,3,2,2,1,1])
sage: c = p.to_core(4); c
[9, 5, 3, 2, 1, 1]
sage: type(c)
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: c.to_bounded_partition() == p
True
```

```
>>> from sage.all import *
>>> p = Partition([Integer(4),Integer(3),Integer(2),Integer(2),Integer(1),
↪Integer(1)])
>>> c = p.to_core(Integer(4)); c
[9, 5, 3, 2, 1, 1]
>>> type(c)
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
>>> c.to_bounded_partition() == p
True
```

to_dyck_word(*n=None*)

Return the `n`-Dyck word whose corresponding partition is `self` (or, if `n` is not specified, the n-Dyck word with smallest n to satisfy this property).

If w is an n-Dyck word (that is, a Dyck word with n open symbols and n close symbols), then the Dyck path corresponding to w can be regarded as a lattice path in the northeastern half of an $n \times n$-square. The region to the northeast of this Dyck path can be regarded as a partition. It is called the partition corresponding to the Dyck word w. (See `to_partition()`.)

For every partition λ and every nonnegative integer n, there exists at most one n-Dyck word w such that the partition corresponding to w is λ (in fact, such w exists if and only if $\lambda_i + i \leq n$ for every i, where λ is written in the form $(\lambda_1, \lambda_2, \ldots, \lambda_k)$ with $\lambda_k > 0$). This method computes this w for a given λ and n. If n is not specified, this method computes the w for the smallest possible n for which such an w exists. (The minimality of n means that the partition demarcated by the Dyck path touches the diagonal.)

EXAMPLES:

```
sage: Partition([2,2]).to_dyck_word()
[1, 1, 0, 0, 1, 1, 0, 0]
sage: Partition([2,2]).to_dyck_word(4)
[1, 1, 0, 0, 1, 1, 0, 0]
sage: Partition([2,2]).to_dyck_word(5)
[1, 1, 1, 0, 0, 1, 1, 0, 0, 0]
sage: Partition([6,3,1]).to_dyck_word()
[1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0]
sage: Partition([]).to_dyck_word()
[]
sage: Partition([]).to_dyck_word(3)
[1, 1, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2)]).to_dyck_word()
[1, 1, 0, 0, 1, 1, 0, 0]
>>> Partition([Integer(2),Integer(2)]).to_dyck_word(Integer(4))
[1, 1, 0, 0, 1, 1, 0, 0]
>>> Partition([Integer(2),Integer(2)]).to_dyck_word(Integer(5))
[1, 1, 1, 0, 0, 1, 1, 0, 0, 0]
>>> Partition([Integer(6),Integer(3),Integer(1)]).to_dyck_word()
[1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0]
>>> Partition([]).to_dyck_word()
[]
>>> Partition([]).to_dyck_word(Integer(3))
[1, 1, 1, 0, 0, 0]
```

The partition corresponding to `self.dyck_word()` is `self` indeed:

```
sage: all( p.to_dyck_word().to_partition() == p
....:      for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> all( p.to_dyck_word().to_partition() == p
...      for p in Partitions(Integer(5)) )
True
```

to_exp(*k=0*)

Return a list of the multiplicities of the parts of a partition. Use the optional parameter `k` to get a return list of length at least `k`.

EXAMPLES:

```
sage: Partition([3,2,2,1]).to_exp()
[1, 2, 1]
sage: Partition([3,2,2,1]).to_exp(5)
[1, 2, 1, 0, 0]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(2),Integer(1)]).to_exp()
[1, 2, 1]
>>> Partition([Integer(3),Integer(2),Integer(2),Integer(1)]).to_
↪exp(Integer(5))
[1, 2, 1, 0, 0]
```

to_exp_dict()

Return a dictionary containing the multiplicities of the parts of `self`.

EXAMPLES:

```
sage: p = Partition([4,2,2,1])
sage: d = p.to_exp_dict()
sage: d[4]
1
sage: d[2]
2
sage: d[1]
1
sage: 5 in d
False
```

```
>>> from sage.all import *
>>> p = Partition([Integer(4),Integer(2),Integer(2),Integer(1)])
>>> d = p.to_exp_dict()
>>> d[Integer(4)]
1
>>> d[Integer(2)]
2
>>> d[Integer(1)]
1
>>> Integer(5) in d
False
```

to_list()

Return `self` as a list.

EXAMPLES:

```
sage: p = Partition([2,1]).to_list(); p
[2, 1]
sage: type(p)
<class 'list'>
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(1)]).to_list(); p
[2, 1]
>>> type(p)
<class 'list'>
```

top_garnir_tableau(*e*, *cell*)

Return the most dominant *standard* tableau which dominates the corresponding Garnir tableau and has the same `e`-residue.

The Garnir tableau play an important role in integral and non-semisimple representation theory because they determine the "straightening" rules for the Specht modules. The *top Garnir tableaux* arise in the graded representation theory of the symmetric groups and higher level Hecke algebras. They were introduced in [KMR2012].

If the Garnir node is `cell=(r,c)` and m and M are the entries in the cells `(r,c)` and `(r+1,c)`, respectively, in the initial tableau then the top `e`-Garnir tableau is obtained by inserting the numbers $m, m+1, \ldots, M$ in order from left to right first in the cells in row `r+1` which are not in the `e`-Garnir belt, then in the cell in rows `r` and `r+1` which are in the Garnir belt and then, finally, in the remaining cells in row `r` which are not in the Garnir belt. All other entries in the tableau remain unchanged.

If `e = 0`, or if there are no `e`-bricks in either row `r` or `r+1`, then the top Garnir tableau is the corresponding Garnir tableau.

EXAMPLES:

```
sage: Partition([5,4,3,2]).top_garnir_tableau(2,(0,2)).pp()
   1  2  4  5  8
   3  6  7  9
  10 11 12
  13 14
sage: Partition([5,4,3,2]).top_garnir_tableau(3,(0,2)).pp()
   1  2  3  4  5
   6  7  8  9
  10 11 12
  13 14
sage: Partition([5,4,3,2]).top_garnir_tableau(4,(0,2)).pp()
   1  2  6  7  8
   3  4  5  9
  10 11 12
  13 14
sage: Partition([5,4,3,2]).top_garnir_tableau(0,(0,2)).pp()
   1  2  6  7  8
   3  4  5  9
  10 11 12
  13 14
```

```
>>> from sage.all import *
>>> Partition([Integer(5),Integer(4),Integer(3),Integer(2)]).top_garnir_
↪tableau(Integer(2),(Integer(0),Integer(2))).pp()
   1  2  4  5  8
   3  6  7  9
  10 11 12
  13 14
>>> Partition([Integer(5),Integer(4),Integer(3),Integer(2)]).top_garnir_
↪tableau(Integer(3),(Integer(0),Integer(2))).pp()
   1  2  3  4  5
   6  7  8  9
  10 11 12
  13 14
>>> Partition([Integer(5),Integer(4),Integer(3),Integer(2)]).top_garnir_
↪tableau(Integer(4),(Integer(0),Integer(2))).pp()
   1  2  6  7  8
   3  4  5  9
  10 11 12
```

(continues on next page)

(continued from previous page)

```
 13 14
>>> Partition([Integer(5),Integer(4),Integer(3),Integer(2)]).top_garnir_
↪tableau(Integer(0),(Integer(0),Integer(2))).pp()
  1  2  6  7  8
  3  4  5  9
 10 11 12
 13 14
```

REFERENCES:

- [KMR2012]

up()

Return a generator for partitions that can be obtained from `self` by adding a cell.

EXAMPLES:

```
sage: list(Partition([2,1,1]).up())
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
sage: list(Partition([3,2]).up())
[[4, 2], [3, 3], [3, 2, 1]]
sage: [p for p in Partition([]).up()]
[[1]]
```

```
>>> from sage.all import *
>>> list(Partition([Integer(2),Integer(1),Integer(1)]).up())
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
>>> list(Partition([Integer(3),Integer(2)]).up())
[[4, 2], [3, 3], [3, 2, 1]]
>>> [p for p in Partition([]).up()]
[[1]]
```

up_list()

Return a list of the partitions that can be formed from `self` by adding a cell.

EXAMPLES:

```
sage: Partition([2,1,1]).up_list()
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
sage: Partition([3,2]).up_list()
[[4, 2], [3, 3], [3, 2, 1]]
sage: Partition([]).up_list()
[[1]]
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(1),Integer(1)]).up_list()
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
>>> Partition([Integer(3),Integer(2)]).up_list()
[[4, 2], [3, 3], [3, 2, 1]]
>>> Partition([]).up_list()
[[1]]
```

upper_hook(*i*, *j*, *alpha*)

Return the upper hook length of the cell (i, j) in `self`. When `alpha = 1`, this is just the normal hook length.

The upper hook length of a cell (i, j) in a partition κ is defined by

$$h_\kappa^*(i, j) = \kappa_j' - i + \alpha(\kappa_i - j + 1).$$

EXAMPLES:

```
sage: p = Partition([2,1])
sage: p.upper_hook(0,0,1)
3
sage: p.hook_length(0,0)
3
sage: [ p.upper_hook(i,j,x) for i,j in p.cells() ]                          #␣
↪needs sage.symbolic
[2*x + 1, x, x]
```

```
>>> from sage.all import *
>>> p = Partition([Integer(2),Integer(1)])
>>> p.upper_hook(Integer(0),Integer(0),Integer(1))
3
>>> p.hook_length(Integer(0),Integer(0))
3
>>> [ p.upper_hook(i,j,x) for i,j in p.cells() ]                          #␣
↪needs sage.symbolic
[2*x + 1, x, x]
```

upper_hook_lengths(*alpha*)

Return a tableau of shape `self` with the cells filled in with the upper hook lengths. When `alpha = 1`, these are just the normal hook lengths.

The upper hook length of a cell (i, j) in a partition κ is defined by

$$h_\kappa^*(i, j) = \kappa_j' - i + \alpha(\kappa_i - j + 1).$$

EXAMPLES:

```
sage: Partition([3,2,1]).upper_hook_lengths(x)                               #␣
↪needs sage.symbolic
[[3*x + 2, 2*x + 1, x], [2*x + 1, x], [x]]
sage: Partition([3,2,1]).upper_hook_lengths(1)
[[5, 3, 1], [3, 1], [1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
```

```
>>> from sage.all import *
>>> Partition([Integer(3),Integer(2),Integer(1)]).upper_hook_lengths(x)      ␣
↪                       # needs sage.symbolic
[[3*x + 2, 2*x + 1, x], [2*x + 1, x], [x]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).upper_hook_
↪lengths(Integer(1))
[[5, 3, 1], [3, 1], [1]]
>>> Partition([Integer(3),Integer(2),Integer(1)]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
```

vertical_border_strip_cells(*k*)

Return a list of all the vertical border strips of length k which can be added to self, where each horizontal border strip is a generator of cells.

EXAMPLES:

```
sage: list(Partition([]).vertical_border_strip_cells(0))
[]
sage: list(Partition([3,2,1]).vertical_border_strip_cells(0))
[]
sage: list(Partition([]).vertical_border_strip_cells(2))
[[(0, 0), (1, 0)]]
sage: list(Partition([2,2]).vertical_border_strip_cells(2))
[[(0, 2), (1, 2)],
 [(0, 2), (2, 0)],
 [(2, 0), (3, 0)]]
sage: list(Partition([3,2,2]).vertical_border_strip_cells(2))
[[(0, 3), (1, 2)],
 [(0, 3), (3, 0)],
 [(1, 2), (2, 2)],
 [(1, 2), (3, 0)],
 [(3, 0), (4, 0)]]
```

```
>>> from sage.all import *
>>> list(Partition([]).vertical_border_strip_cells(Integer(0)))
[]
>>> list(Partition([Integer(3),Integer(2),Integer(1)]).vertical_border_strip_
↪cells(Integer(0)))
[]
>>> list(Partition([]).vertical_border_strip_cells(Integer(2)))
[[(0, 0), (1, 0)]]
>>> list(Partition([Integer(2),Integer(2)]).vertical_border_strip_
↪cells(Integer(2)))
[[(0, 2), (1, 2)],
 [(0, 2), (2, 0)],
 [(2, 0), (3, 0)]]
>>> list(Partition([Integer(3),Integer(2),Integer(2)]).vertical_border_strip_
↪cells(Integer(2)))
[[(0, 3), (1, 2)],
 [(0, 3), (3, 0)],
 [(1, 2), (2, 2)],
 [(1, 2), (3, 0)],
 [(3, 0), (4, 0)]]
```

weighted_size()

Return the weighted size of self.

The weighted size of a partition λ is

$$\sum_i i \cdot \lambda_i,$$

where $\lambda = (\lambda_0, \lambda_1, \lambda_2, \cdots)$.

This also the sum of the leg length of every cell in λ, or

$$\sum_i \binom{\lambda'_i}{2}$$

where λ' is the conjugate partition of λ.

EXAMPLES:

```
sage: Partition([2,2]).weighted_size()
2
sage: Partition([3,3,3]).weighted_size()
9
sage: Partition([5,2]).weighted_size()
2
sage: Partition([]).weighted_size()
0
```

```
>>> from sage.all import *
>>> Partition([Integer(2),Integer(2)]).weighted_size()
2
>>> Partition([Integer(3),Integer(3),Integer(3)]).weighted_size()
9
>>> Partition([Integer(5),Integer(2)]).weighted_size()
2
>>> Partition([]).weighted_size()
0
```

young_subgroup()

Return the corresponding Young, or parabolic, subgroup of the symmetric group.

The Young subgroup of a partition $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_\ell)$ of n is the group:

$$S_{\lambda_1} \times S_{\lambda_2} \times \cdots \times S_{\lambda_\ell}$$

embedded into S_n in the standard way (i.e., the S_{λ_i} factor acts on the numbers from $\lambda_1 + \lambda_2 + \cdots + \lambda_{i-1} + 1$ to $\lambda_1 + \lambda_2 + \cdots + \lambda_i$).

EXAMPLES:

```
sage: Partition([4,2]).young_subgroup()                                    #␣
↪needs sage.groups
Permutation Group with generators [(), (5,6), (3,4), (2,3), (1,2)]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(2)]).young_subgroup()                    ␣
↪               # needs sage.groups
Permutation Group with generators [(), (5,6), (3,4), (2,3), (1,2)]
```

young_subgroup_generators()

Return an indexing set for the generators of the corresponding Young subgroup. Here the generators correspond to the simple adjacent transpositions $s_i = (i\ i+1)$.

EXAMPLES:

```
sage: Partition([4,2]).young_subgroup_generators()
[1, 2, 3, 5]
sage: Partition([1,1,1]).young_subgroup_generators()
[]
sage: Partition([2,2]).young_subgroup_generators()
[1, 3]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(2)]).young_subgroup_generators()
[1, 2, 3, 5]
>>> Partition([Integer(1),Integer(1),Integer(1)]).young_subgroup_generators()
[]
>>> Partition([Integer(2),Integer(2)]).young_subgroup_generators()
[1, 3]
```

See also

young_subgroup()

zero_one_sequence()

Compute the finite $0-1$ sequence of the partition.

The full $0-1$ sequence is the sequence (infinite in both directions) indicating the steps taken when following the outer rim of the diagram of the partition. We use the convention that in English convention, a 1 corresponds to an East step, and a 0 corresponds to a North step.

Note that every full $0-1$ sequence starts with infinitely many 0s and ends with infinitely many 1s.

One place where these arise is in the affine symmetric group where one takes an affine permutation w and every i such that $w(i) \leq 0$ corresponds to a 1 and $w(i) > 0$ corresponds to a 0. See pages 24-25 of [LLMSSZ2013] for connections to affine Grassmannian elements (note there they use the French convention for their partitions).

These are also known as **path sequences**, **Maya diagrams**, **plus-minus diagrams**, **Comet code** [Sta-EC2], among others.

OUTPUT:

The finite $0-1$ sequence is obtained from the full $0-1$ sequence by omitting all heading 0s and trailing 1s. The output sequence is finite, starts with a 1 and ends with a 0 (unless it is empty, for the empty partition). Its length is the sum of the first part of the partition with the length of the partition.

EXAMPLES:

```
sage: Partition([5,4]).zero_one_sequence()
[1, 1, 1, 1, 0, 1, 0]
sage: Partition([]).zero_one_sequence()
[]
sage: Partition([2]).zero_one_sequence()
[1, 1, 0]
```

```
>>> from sage.all import *
>>> Partition([Integer(5),Integer(4)]).zero_one_sequence()
[1, 1, 1, 1, 0, 1, 0]
```

(continues on next page)

(continued from previous page)

```
>>> Partition([]).zero_one_sequence()
[]
>>> Partition([Integer(2)]).zero_one_sequence()
[1, 1, 0]
```

class sage.combinat.partition.**Partitions**(*is_infinite=False*)

Bases: UniqueRepresentation, Parent

`Partitions(n, **kwargs)` returns the combinatorial class of integer partitions of n subject to the constraints given by the keywords.

Valid keywords are: `starting`, `ending`, `min_part`, `max_part`, `max_length`, `min_length`, `length`, `max_slope`, `min_slope`, `inner`, `outer`, `parts_in`, `regular`, and `restricted`. They have the following meanings:

- `starting=p` specifies that the partitions should all be less than or equal to p in lex order. This argument cannot be combined with any other (see Issue #15467).
- `ending=p` specifies that the partitions should all be greater than or equal to p in lex order. This argument cannot be combined with any other (see Issue #15467).
- `length=k` specifies that the partitions have exactly k parts.
- `min_length=k` specifies that the partitions have at least k parts.
- `min_part=k` specifies that all parts of the partitions are at least k.
- `inner=p` specifies that the partitions must contain the partition p.
- `outer=p` specifies that the partitions be contained inside the partition p.
- `min_slope=k` specifies that the partitions have slope at least k; the slope at position i is the difference between the $(i+1)$-th part and the i-th part.
- `parts_in=S` specifies that the partitions have parts in the set S, which can be any sequence of pairwise distinct positive integers. This argument cannot be combined with any other (see Issue #15467).
- `regular=ell` specifies that the partitions are ℓ-regular, and can only be combined with the `max_length` or `max_part`, but not both, keywords if n is not specified
- `restricted=ell` specifies that the partitions are ℓ-restricted, and cannot be combined with any other keywords

The `max_*` versions, along with `inner` and `ending`, work analogously.

Right now, the `parts_in`, `starting`, `ending`, `regular`, and `restricted` keyword arguments are mutually exclusive, both of each other and of other keyword arguments. If you specify, say, `parts_in`, all other keyword arguments will be ignored; `starting`, `ending`, `regular`, and `restricted` work the same way.

EXAMPLES:

If no arguments are passed, then the combinatorial class of all integer partitions is returned:

```
sage: Partitions()
Partitions
sage: [2,1] in Partitions()
True
```

```
>>> from sage.all import *
>>> Partitions()
```

(continues on next page)

(continued from previous page)

```
Partitions
>>> [Integer(2),Integer(1)] in Partitions()
True
```

If an integer n is passed, then the combinatorial class of integer partitions of n is returned:

```
sage: Partitions(3)
Partitions of the integer 3
sage: Partitions(3).list()
[[3], [2, 1], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(3))
Partitions of the integer 3
>>> Partitions(Integer(3)).list()
[[3], [2, 1], [1, 1, 1]]
```

If `starting=p` is passed, then the combinatorial class of partitions greater than or equal to p in lexicographic order is returned:

```
sage: Partitions(3, starting=[2,1])
Partitions of the integer 3 starting with [2, 1]
sage: Partitions(3, starting=[2,1]).list()
[[2, 1], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(3), starting=[Integer(2),Integer(1)])
Partitions of the integer 3 starting with [2, 1]
>>> Partitions(Integer(3), starting=[Integer(2),Integer(1)]).list()
[[2, 1], [1, 1, 1]]
```

If `ending=p` is passed, then the combinatorial class of partitions at most p in lexicographic order is returned:

```
sage: Partitions(3, ending=[2,1])
Partitions of the integer 3 ending with [2, 1]
sage: Partitions(3, ending=[2,1]).list()
[[3], [2, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(3), ending=[Integer(2),Integer(1)])
Partitions of the integer 3 ending with [2, 1]
>>> Partitions(Integer(3), ending=[Integer(2),Integer(1)]).list()
[[3], [2, 1]]
```

Using `max_slope=-1` yields partitions into distinct parts – each part differs from the next by at least 1. Use a different `max_slope` to get parts that differ by, say, 2:

```
sage: Partitions(7, max_slope=-1).list()
[[7], [6, 1], [5, 2], [4, 3], [4, 2, 1]]
sage: Partitions(15, max_slope=-1).cardinality()
27
```

```
>>> from sage.all import *
>>> Partitions(Integer(7), max_slope=-Integer(1)).list()
[[7], [6, 1], [5, 2], [4, 3], [4, 2, 1]]
>>> Partitions(Integer(15), max_slope=-Integer(1)).cardinality()
27
```

The number of partitions of n into odd parts equals the number of partitions into distinct parts. Let's test that for n from 10 to 20:

```
sage: def test(n):
....:     return (Partitions(n, max_slope=-1).cardinality()
....:             == Partitions(n, parts_in=[1,3..n]).cardinality())
sage: all(test(n) for n in [10..20])                                        #␣
↪needs sage.libs.gap
True
```

```
>>> from sage.all import *
>>> def test(n):
...     return (Partitions(n, max_slope=-Integer(1)).cardinality()
...             == Partitions(n, parts_in=(ellipsis_range(Integer(1),Integer(3),
↪Ellipsis,n))).cardinality())
>>> all(test(n) for n in (ellipsis_range(Integer(10),Ellipsis,Integer(20))))     ␣
↪                                   # needs sage.libs.gap
True
```

The number of partitions of n into distinct parts that differ by at least 2 equals the number of partitions into parts that equal 1 or 4 modulo 5; this is one of the Rogers-Ramanujan identities:

```
sage: def test(n):
....:     return (Partitions(n, max_slope=-2).cardinality()
....:             == Partitions(n, parts_in=([1,6..n] + [4,9..n])).cardinality())
sage: all(test(n) for n in [10..20])                                        #␣
↪needs sage.libs.gap
True
```

```
>>> from sage.all import *
>>> def test(n):
...     return (Partitions(n, max_slope=-Integer(2)).cardinality()
...             == Partitions(n, parts_in=((ellipsis_range(Integer(1),Integer(6),
↪Ellipsis,n)) + (ellipsis_range(Integer(4),Integer(9),Ellipsis,n)))).
↪cardinality())
>>> all(test(n) for n in (ellipsis_range(Integer(10),Ellipsis,Integer(20))))     ␣
↪                                   # needs sage.libs.gap
True
```

Here are some more examples illustrating `min_part`, `max_part`, and `length`:

```
sage: Partitions(5, min_part=2)
Partitions of 5 whose parts are at least 2
sage: Partitions(5, min_part=2).list()
[[5], [3, 2]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(5), min_part=Integer(2))
Partitions of 5 whose parts are at least 2
>>> Partitions(Integer(5), min_part=Integer(2)).list()
[[5], [3, 2]]
```

```
sage: Partitions(3, max_length=2).list()
[[3], [2, 1]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(3), max_length=Integer(2)).list()
[[3], [2, 1]]
```

```
sage: Partitions(10, min_part=2, length=3).list()
[[6, 2, 2], [5, 3, 2], [4, 4, 2], [4, 3, 3]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(10), min_part=Integer(2), length=Integer(3)).list()
[[6, 2, 2], [5, 3, 2], [4, 4, 2], [4, 3, 3]]
```

Some examples using the `regular` keyword:

```
sage: Partitions(regular=4)
4-Regular Partitions
sage: Partitions(regular=4, max_length=3)
4-Regular Partitions with max length 3
sage: Partitions(regular=4, max_part=3)
4-Regular 3-Bounded Partitions
sage: Partitions(3, regular=4)
4-Regular Partitions of the integer 3
```

```
>>> from sage.all import *
>>> Partitions(regular=Integer(4))
4-Regular Partitions
>>> Partitions(regular=Integer(4), max_length=Integer(3))
4-Regular Partitions with max length 3
>>> Partitions(regular=Integer(4), max_part=Integer(3))
4-Regular 3-Bounded Partitions
>>> Partitions(Integer(3), regular=Integer(4))
4-Regular Partitions of the integer 3
```

Some examples using the `restricted` keyword:

```
sage: Partitions(restricted=4)
4-Restricted Partitions
sage: Partitions(3, restricted=4)
4-Restricted Partitions of the integer 3
```

```
>>> from sage.all import *
>>> Partitions(restricted=Integer(4))
4-Restricted Partitions
```

(continues on next page)

(continued from previous page)

```
>>> Partitions(Integer(3), restricted=Integer(4))
4-Restricted Partitions of the integer 3
```

Here are some further examples using various constraints:

```
sage: [x for x in Partitions(4)]
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(4, length=2)]
[[3, 1], [2, 2]]
sage: [x for x in Partitions(4, min_length=2)]
[[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(4, max_length=2)]
[[4], [3, 1], [2, 2]]
sage: [x for x in Partitions(4, min_length=2, max_length=2)]
[[3, 1], [2, 2]]
sage: [x for x in Partitions(4, max_part=2)]
[[2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(4, min_part=2)]
[[4], [2, 2]]
sage: [x for x in Partitions(4, outer=[3,1,1])]
[[3, 1], [2, 1, 1]]
sage: [x for x in Partitions(4, outer=[infinity, 1, 1])]
[[4], [3, 1], [2, 1, 1]]
sage: [x for x in Partitions(4, inner=[1,1,1])]
[[2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(4, max_slope=-1)]
[[4], [3, 1]]
sage: [x for x in Partitions(4, min_slope=-1)]
[[4], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(11, max_slope=-1, min_slope=-3, min_length=2, max_
↪length=4)]
[[7, 4], [6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2], [5, 3, 2, 1]]
sage: [x for x in Partitions(11, max_slope=-1, min_slope=-3, min_length=2, max_
↪length=4, outer=[6,5,2])]
[[6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2]]
```

```
>>> from sage.all import *
>>> [x for x in Partitions(Integer(4))]
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
>>> [x for x in Partitions(Integer(4), length=Integer(2))]
[[3, 1], [2, 2]]
>>> [x for x in Partitions(Integer(4), min_length=Integer(2))]
[[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
>>> [x for x in Partitions(Integer(4), max_length=Integer(2))]
[[4], [3, 1], [2, 2]]
>>> [x for x in Partitions(Integer(4), min_length=Integer(2), max_
↪length=Integer(2))]
[[3, 1], [2, 2]]
>>> [x for x in Partitions(Integer(4), max_part=Integer(2))]
[[2, 2], [2, 1, 1], [1, 1, 1, 1]]
>>> [x for x in Partitions(Integer(4), min_part=Integer(2))]
[[4], [2, 2]]
```

(continues on next page)

(continued from previous page)

```
>>> [x for x in Partitions(Integer(4), outer=[Integer(3),Integer(1),Integer(1)])]
[[3, 1], [2, 1, 1]]
>>> [x for x in Partitions(Integer(4), outer=[infinity, Integer(1), Integer(1)])]
[[4], [3, 1], [2, 1, 1]]
>>> [x for x in Partitions(Integer(4), inner=[Integer(1),Integer(1),Integer(1)])]
[[2, 1, 1], [1, 1, 1, 1]]
>>> [x for x in Partitions(Integer(4), max_slope=-Integer(1))]
[[4], [3, 1]]
>>> [x for x in Partitions(Integer(4), min_slope=-Integer(1))]
[[4], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
>>> [x for x in Partitions(Integer(11), max_slope=-Integer(1), min_slope=-
↪Integer(3), min_length=Integer(2), max_length=Integer(4))]
[[7, 4], [6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2], [5, 3, 2, 1]]
>>> [x for x in Partitions(Integer(11), max_slope=-Integer(1), min_slope=-
↪Integer(3), min_length=Integer(2), max_length=Integer(4), outer=[Integer(6),
↪Integer(5),Integer(2)])]
[[6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2]]
```

Note that if you specify `min_part=0`, then it will treat the minimum part as being 1 (see Issue #13605):

```
sage: [x for x in Partitions(4, length=3, min_part=0)]
[[2, 1, 1]]
sage: [x for x in Partitions(4, min_length=3, min_part=0)]
[[2, 1, 1], [1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> [x for x in Partitions(Integer(4), length=Integer(3), min_part=Integer(0))]
[[2, 1, 1]]
>>> [x for x in Partitions(Integer(4), min_length=Integer(3), min_
↪part=Integer(0))]
[[2, 1, 1], [1, 1, 1, 1]]
```

Except for very special cases, counting is done by brute force iteration through all the partitions. However the iteration itself has a reasonable complexity (see *IntegerListsLex*), which allows for manipulating large partitions:

```
sage: Partitions(1000, max_length=1).list()
[[1000]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(1000), max_length=Integer(1)).list()
[[1000]]
```

In particular, getting the first element is also constant time:

```
sage: Partitions(30, max_part=29).first()
[29, 1]
```

```
>>> from sage.all import *
>>> Partitions(Integer(30), max_part=Integer(29)).first()
[29, 1]
```

Element

alias of *Partition*

options = Current options for Partitions - convention: English - diagram_str: * - display: list - latex: young_diagram - latex_diagram_str: \ast

subset(**args*, ***kwargs*)

Return `self` if no arguments are given.

Otherwise, it raises a `ValueError`.

EXAMPLES:

```
sage: P = Partitions(5, starting=[3,1]); P
Partitions of the integer 5 starting with [3, 1]
sage: P.subset()
Partitions of the integer 5 starting with [3, 1]
sage: P.subset(ending=[3,1])
Traceback (most recent call last):
...
ValueError: invalid combination of arguments
```

```
>>> from sage.all import *
>>> P = Partitions(Integer(5), starting=[Integer(3),Integer(1)]); P
Partitions of the integer 5 starting with [3, 1]
>>> P.subset()
Partitions of the integer 5 starting with [3, 1]
>>> P.subset(ending=[Integer(3),Integer(1)])
Traceback (most recent call last):
...
ValueError: invalid combination of arguments
```

class sage.combinat.partition.**PartitionsGreatestEQ**(*n*, *k*)

Bases: `UniqueRepresentation`, *IntegerListsLex*

The class of all (unordered) "restricted" partitions of the integer n having all its greatest parts equal to the integer k.

EXAMPLES:

```
sage: PartitionsGreatestEQ(10, 2)
Partitions of 10 having greatest part equal to 2
sage: PartitionsGreatestEQ(10, 2).list()
[[2, 2, 2, 2, 2],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1]]

sage: [4,3,2,1] in PartitionsGreatestEQ(10, 2)
False
sage: [2,2,2,2,2] in PartitionsGreatestEQ(10, 2)
True
```

```
>>> from sage.all import *
>>> PartitionsGreatestEQ(Integer(10), Integer(2))
Partitions of 10 having greatest part equal to 2
>>> PartitionsGreatestEQ(Integer(10), Integer(2)).list()
```

(continues on next page)

(continued from previous page)

```
[[2, 2, 2, 2, 2],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1]]

>>> [Integer(4),Integer(3),Integer(2),Integer(1)] in␣
↪PartitionsGreatestEQ(Integer(10), Integer(2))
False
>>> [Integer(2),Integer(2),Integer(2),Integer(2),Integer(2)] in␣
↪PartitionsGreatestEQ(Integer(10), Integer(2))
True
```

The empty partition has no maximal part, but it is contained in the set of partitions with any specified maximal part:

```
sage: PartitionsGreatestEQ(0, 2).list()
[[]]
```

```
>>> from sage.all import *
>>> PartitionsGreatestEQ(Integer(0), Integer(2)).list()
[[]]
```

Element

alias of *Partition*

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: PartitionsGreatestEQ(10, 2).cardinality()
5
```

```
>>> from sage.all import *
>>> PartitionsGreatestEQ(Integer(10), Integer(2)).cardinality()
5
```

options = Current options for Partitions - convention: English - diagram_str: * - display: list - latex: young_diagram - latex_diagram_str: \ast

class sage.combinat.partition.**PartitionsGreatestLE**(n, k)

Bases: `UniqueRepresentation`, *IntegerListsLex*

The class of all (unordered) "restricted" partitions of the integer n having parts less than or equal to the integer k.

EXAMPLES:

```
sage: PartitionsGreatestLE(10, 2)
Partitions of 10 having parts less than or equal to 2
sage: PartitionsGreatestLE(10, 2).list()
[[2, 2, 2, 2, 2],
 [2, 2, 2, 2, 1, 1],
```

(continues on next page)

(continued from previous page)

```
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

sage: [4,3,2,1] in PartitionsGreatestLE(10, 2)
False
sage: [2,2,2,2,2] in PartitionsGreatestLE(10, 2)
True
sage: PartitionsGreatestLE(10, 2).first().parent()
Partitions...
```

```
>>> from sage.all import *
>>> PartitionsGreatestLE(Integer(10), Integer(2))
Partitions of 10 having parts less than or equal to 2
>>> PartitionsGreatestLE(Integer(10), Integer(2)).list()
[[2, 2, 2, 2, 2],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

>>> [Integer(4),Integer(3),Integer(2),Integer(1)] in␣
↪PartitionsGreatestLE(Integer(10), Integer(2))
False
>>> [Integer(2),Integer(2),Integer(2),Integer(2),Integer(2)] in␣
↪PartitionsGreatestLE(Integer(10), Integer(2))
True
>>> PartitionsGreatestLE(Integer(10), Integer(2)).first().parent()
Partitions...
```

Element

alias of *Partition*

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: PartitionsGreatestLE(9, 5).cardinality()                            #␣
↪needs sage.libs.gap
23
```

```
>>> from sage.all import *
>>> PartitionsGreatestLE(Integer(9), Integer(5)).cardinality()              ␣
↪               # needs sage.libs.gap
23
```

options = Current options for Partitions - convention: English - diagram_str: * - display: list - latex: young_diagram - latex_diagram_str: \ast

class sage.combinat.partition.**PartitionsInBox**(*h*, *w*)

Bases: *Partitions*

All partitions which fit in an $h \times w$ box.

EXAMPLES:

```
sage: PartitionsInBox(2, 2)
Integer partitions which fit in a 2 x 2 box
sage: PartitionsInBox(2, 2).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2]]

sage: Partitions(max_part=2, max_length=3)
Integer partitions which fit in a 3 x 2 box
```

```
>>> from sage.all import *
>>> PartitionsInBox(Integer(2), Integer(2))
Integer partitions which fit in a 2 x 2 box
>>> PartitionsInBox(Integer(2), Integer(2)).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2]]

>>> Partitions(max_part=Integer(2), max_length=Integer(3))
Integer partitions which fit in a 3 x 2 box
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: PartitionsInBox(2, 3).cardinality()
10
```

```
>>> from sage.all import *
>>> PartitionsInBox(Integer(2), Integer(3)).cardinality()
10
```

list()

Return a list of all the partitions inside a box of height h and width w.

EXAMPLES:

```
sage: PartitionsInBox(2,2).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2]]
sage: PartitionsInBox(2,3).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2], [3], [3, 1], [3, 2], [3, 3]]
```

```
>>> from sage.all import *
>>> PartitionsInBox(Integer(2),Integer(2)).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2]]
>>> PartitionsInBox(Integer(2),Integer(3)).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2], [3], [3, 1], [3, 2], [3, 3]]
```

class sage.combinat.partition.**Partitions_all**

Bases: *Partitions*

Class of all partitions.

from_beta_numbers(*beta*)

Return a partition corresponding to a sequence of beta numbers.

A sequence of beta numbers is a strictly increasing sequence $0 \leq b_1 < \cdots < b_k$ of nonnegative integers. The corresponding partition $\mu = (\mu_k, \ldots, \mu_1)$ is given by $\mu_i = [1, i) \setminus \{b_1, \ldots, b_i\}$. This gives a bijection from the set of partitions with at most k nonzero parts to the set of strictly increasing sequences of nonnegative integers of length k.

EXAMPLES:

```
sage: Partitions().from_beta_numbers([0,1,2,4,5,8])
[3, 1, 1]
sage: Partitions().from_beta_numbers([0,2,3,6])
[3, 1, 1]
```

```
>>> from sage.all import *
>>> Partitions().from_beta_numbers([Integer(0),Integer(1),Integer(2),
↪Integer(4),Integer(5),Integer(8)])
[3, 1, 1]
>>> Partitions().from_beta_numbers([Integer(0),Integer(2),Integer(3),
↪Integer(6)])
[3, 1, 1]
```

from_core_and_quotient(*core*, *quotient*)

Return a partition from its core and quotient.

Algorithm from mupad-combinat.

EXAMPLES:

```
sage: Partitions().from_core_and_quotient([2,1], [[2,1],[3],[1,1,1]])
[11, 5, 5, 3, 2, 2, 2]
```

```
>>> from sage.all import *
>>> Partitions().from_core_and_quotient([Integer(2),Integer(1)], [[Integer(2),
↪Integer(1)],[Integer(3)],[Integer(1),Integer(1),Integer(1)]])
[11, 5, 5, 3, 2, 2, 2]
```

from_exp(*exp*)

Return a partition from its list of multiplicities.

EXAMPLES:

```
sage: Partitions().from_exp([2,2,1])
[3, 2, 2, 1, 1]
```

```
>>> from sage.all import *
>>> Partitions().from_exp([Integer(2),Integer(2),Integer(1)])
[3, 2, 2, 1, 1]
```

from_frobenius_coordinates(*frobenius_coordinates*)

Return a partition from a pair of sequences of Frobenius coordinates.

EXAMPLES:

```
sage: Partitions().from_frobenius_coordinates(([],[]))
[]
sage: Partitions().from_frobenius_coordinates(([0],[0]))
[1]
sage: Partitions().from_frobenius_coordinates(([1],[1]))
[2, 1]
sage: Partitions().from_frobenius_coordinates(([6,3,2],[4,1,0]))
[7, 5, 5, 1, 1]
```

```
>>> from sage.all import *
>>> Partitions().from_frobenius_coordinates(([],[]))
[]
>>> Partitions().from_frobenius_coordinates(([Integer(0)],[Integer(0)]))
[1]
>>> Partitions().from_frobenius_coordinates(([Integer(1)],[Integer(1)]))
[2, 1]
>>> Partitions().from_frobenius_coordinates(([Integer(6),Integer(3),
→Integer(2)],[Integer(4),Integer(1),Integer(0)]))
[7, 5, 5, 1, 1]
```

from_zero_one(*seq*)

Return a partition from its $0-1$ sequence.

The full $0-1$ sequence is the sequence (infinite in both directions) indicating the steps taken when following the outer rim of the diagram of the partition. We use the convention that in English convention, a 1 corresponds to an East step, and a 0 corresponds to a North step.

Note that every full $0-1$ sequence starts with infinitely many 0s and ends with infinitely many 1s.

See also

Partition.zero_one_sequence()

INPUT:

The input should be a finite sequence of 0s and 1s. The heading 0s and trailing 1s will be discarded.

EXAMPLES:

```
sage: Partitions().from_zero_one([])
[]
sage: Partitions().from_zero_one([1,0])
[1]
sage: Partitions().from_zero_one([1, 1, 1, 1, 0, 1, 0])
[5, 4]
```

```
>>> from sage.all import *
>>> Partitions().from_zero_one([])
[]
>>> Partitions().from_zero_one([Integer(1),Integer(0)])
[1]
>>> Partitions().from_zero_one([Integer(1), Integer(1), Integer(1),
→Integer(1), Integer(0), Integer(1), Integer(0)])
[5, 4]
```

Heading 0s and trailing 1s are correctly handled:

```
sage: Partitions().from_zero_one([0,0,1,1,1,1,0,1,0,1,1,1])
[5, 4]
```

```
>>> from sage.all import *
>>> Partitions().from_zero_one([Integer(0),Integer(0),Integer(1),Integer(1),
↪Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(1),Integer(1)])
[5, 4]
```

subset(*size=None*, ***kwargs*)

Return the subset of partitions of a given size and additional keyword arguments.

EXAMPLES:

```
sage: P = Partitions()
sage: P.subset(4)
Partitions of the integer 4
```

```
>>> from sage.all import *
>>> P = Partitions()
>>> P.subset(Integer(4))
Partitions of the integer 4
```

class sage.combinat.partition.**Partitions_all_bounded**(*k*)

Bases: *Partitions*

Partitions whose parts do not exceed a given bound.

class sage.combinat.partition.**Partitions_all_constrained**(***kwargs*)

Bases: *Partitions*

class sage.combinat.partition.**Partitions_constraints**(**args*, ***kwds*)

Bases: *IntegerListsLex*

For unpickling old constrained `Partitions_constraints` objects created with sage <= 3.4.1. See *Partitions*.

class sage.combinat.partition.**Partitions_ending**(*n*, *ending_partition*)

Bases: *Partitions*

All partitions with a given ending.

first()

Return the first partition in `self`.

EXAMPLES:

```
sage: Partitions(4, ending=[1,1,1,1]).first()
[4]
sage: Partitions(4, ending=[5]).first() is None
True
```

```
>>> from sage.all import *
>>> Partitions(Integer(4), ending=[Integer(1),Integer(1),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)]).first()
[4]
>>> Partitions(Integer(4), ending=[Integer(5)]).first() is None
True
```

next (*part*)

Return the next partition after `part` in `self`.

EXAMPLES::

sage: Partitions(4, ending=[1,1,1,1,1]).next(Partition([4])) [3, 1] sage: Partitions(4, ending=[3,2]).next(Partition([3,1])) is None True sage: Partitions(4, ending=[1,1,1,1]).next(Partition([4])) [3, 1] sage: Partitions(4, ending=[1,1,1,1]).next(Partition([1,1,1,1])) is None True sage: Partitions(4, ending=[3]).next(Partition([3,1])) is None True

class sage.combinat.partition.**Partitions_length_and_parts_constrained**(*n*, *min_length*, *max_length*, *min_part*, *max_part*)

Bases: *Partitions*

The class of all integer partitions having parts and length in a given range.

This class is strictly more general than *PartitionsGreatestLE*, except that we insist that the size of the partition is positive and that neither the constraints on the parts nor on the length are contradictory.

INPUT:

- `n` – the size of the partition, positive
- `min_length` – the lower bound on the number of parts, between 1 and `n`
- `max_length` – the upper bound on the number of parts, between `min_length` and `n`
- `min_part` – the bound on the smallest part, between 1 and `n`
- `max_part` – the bound on the largest part, between `min_part` and `n`

EXAMPLES:

```
sage: from sage.combinat.partition import Partitions_length_and_parts_constrained
sage: Partitions_length_and_parts_constrained(10, 1, 10, 2, 5)
Partitions of 10 whose parts are between 2 and 5
sage: list(Partitions_length_and_parts_constrained(9, 3, 4, 2, 4))
[[4, 3, 2], [3, 3, 3], [3, 2, 2, 2]]

sage: [4,3,2,1] in Partitions_length_and_parts_constrained(10, 1, 10, 2, 10)
False
sage: [2,2,2,2,2] in Partitions_length_and_parts_constrained(10, 1, 10, 2, 10)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition import Partitions_length_and_parts_constrained
>>> Partitions_length_and_parts_constrained(Integer(10), Integer(1), Integer(10),␣
↪Integer(2), Integer(5))
Partitions of 10 whose parts are between 2 and 5
>>> list(Partitions_length_and_parts_constrained(Integer(9), Integer(3),␣
↪Integer(4), Integer(2), Integer(4)))
[[4, 3, 2], [3, 3, 3], [3, 2, 2, 2]]
```

(continues on next page)

(continued from previous page)

```
>>> [Integer(4),Integer(3),Integer(2),Integer(1)] in Partitions_length_and_parts_
↪constrained(Integer(10), Integer(1), Integer(10), Integer(2), Integer(10))
False
>>> [Integer(2),Integer(2),Integer(2),Integer(2),Integer(2)] in Partitions_length_
↪and_parts_constrained(Integer(10), Integer(1), Integer(10), Integer(2),␣
↪Integer(10))
True
```

> **Warning**
>
> If `min_length` and `min_part` equal 1 and `max_length` and `max_part` equal `n`, this class contains the same partitions as *Partitions*, but is different from that class.

```
sage: Partitions_length_and_parts_constrained(9, 1, 9, 1, 9)
Partitions of 9
sage: Partitions_length_and_parts_constrained(9, 1, 9, 1, 9) == Partitions(9)
False
```

```
>>> from sage.all import *
>>> Partitions_length_and_parts_constrained(Integer(9), Integer(1), Integer(9),␣
↪Integer(1), Integer(9))
Partitions of 9
>>> Partitions_length_and_parts_constrained(Integer(9), Integer(1), Integer(9),␣
↪Integer(1), Integer(9)) == Partitions(Integer(9))
False
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: from sage.combinat.partition import Partitions_length_and_parts_
↪constrained
sage: list(Partitions_length_and_parts_constrained(9, 1, 2, 3, 9))
[[9], [6, 3], [5, 4]]
sage: Partitions_length_and_parts_constrained(9, 1, 2, 3, 9).cardinality()
3
```

```
>>> from sage.all import *
>>> from sage.combinat.partition import Partitions_length_and_parts_
↪constrained
>>> list(Partitions_length_and_parts_constrained(Integer(9), Integer(1),␣
↪Integer(2), Integer(3), Integer(9)))
[[9], [6, 3], [5, 4]]
>>> Partitions_length_and_parts_constrained(Integer(9), Integer(1),␣
↪Integer(2), Integer(3), Integer(9)).cardinality()
3
```

class sage.combinat.partition.**Partitions_n**(*n*)

Bases: *Partitions*

Partitions of the integer n.

cardinality(*algorithm='flint'*)

Return the number of partitions of the specified size.

INPUT:

- `algorithm` – (default: `'flint'`)
  - `'flint'` – use FLINT (currently the fastest)
  - `'gap'` – use GAP (VERY *slow*)
  - `'pari'` – use PARI. Speed seems the same as GAP until n is in the thousands, in which case PARI is faster

It is possible to associate with every partition of the integer n a conjugacy class of permutations in the symmetric group on n points and vice versa. Therefore the number of partitions p_n is the number of conjugacy classes of the symmetric group on n points.

EXAMPLES:

```
sage: v = Partitions(5).list(); v
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: len(v)
7
sage: Partitions(5).cardinality(algorithm='gap')                          #␣
↪needs sage.libs.gap
7
```

```
>>> from sage.all import *
>>> v = Partitions(Integer(5)).list(); v
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
>>> len(v)
7
>>> Partitions(Integer(5)).cardinality(algorithm='gap')                   ␣
↪     # needs sage.libs.gap
7
```

```
sage: # needs sage.libs.flint
sage: Partitions(3).cardinality()
3
sage: number_of_partitions(5, algorithm='flint')
7
sage: Partitions(10).cardinality()
42
sage: Partitions(40).cardinality()
37338
sage: Partitions(100).cardinality()
190569292
```

```
>>> from sage.all import *
>>> # needs sage.libs.flint
>>> Partitions(Integer(3)).cardinality()
3
>>> number_of_partitions(Integer(5), algorithm='flint')
```

(continues on next page)

(continued from previous page)

```
7
>>> Partitions(Integer(10)).cardinality()
42
>>> Partitions(Integer(40)).cardinality()
37338
>>> Partitions(Integer(100)).cardinality()
190569292
```

```
sage: # needs sage.libs.pari
sage: Partitions(3).cardinality(algorithm='pari')
3
sage: Partitions(5).cardinality(algorithm='pari')
7
sage: Partitions(10).cardinality(algorithm='pari')
42
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> Partitions(Integer(3)).cardinality(algorithm='pari')
3
>>> Partitions(Integer(5)).cardinality(algorithm='pari')
7
>>> Partitions(Integer(10)).cardinality(algorithm='pari')
42
```

A generating function for p_n is given by the reciprocal of Euler's function:

$$\sum_{n=0}^{\infty} p_n x^n = \prod_{k=1}^{\infty} \frac{1}{1-x^k}.$$

We use Sage to verify that the first several coefficients do indeed agree:

```
sage: q = PowerSeriesRing(QQ, 'q', default_prec=9).gen()
sage: prod([(1-q^k)^(-1) for k in range(1,9)])  # partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
sage: [Partitions(k).cardinality() for k in range(2,10)]                    #␣
↪needs sage.libs.flint
[2, 3, 5, 7, 11, 15, 22, 30]
```

```
>>> from sage.all import *
>>> q = PowerSeriesRing(QQ, 'q', default_prec=Integer(9)).gen()
>>> prod([(Integer(1)-q**k)**(-Integer(1)) for k in range(Integer(1),
↪Integer(9))])  # partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
>>> [Partitions(k).cardinality() for k in range(Integer(2),Integer(10))]     ␣
↪              # needs sage.libs.flint
[2, 3, 5, 7, 11, 15, 22, 30]
```

Another consistency test for `n` up to 500:

```
sage: len([n for n in [1..500]                                              #␣
↪needs sage.libs.flint sage.libs.pari
```

(continues on next page)

(continued from previous page)

```
....:     if Partitions(n).cardinality() != Partitions(n).
↪cardinality(algorithm='pari')])
0
```

```
>>> from sage.all import *
>>> len([n for n in (ellipsis_range(Integer(1),Ellipsis,Integer(500)))     ␣
↪                                       # needs sage.libs.flint sage.libs.pari
...     if Partitions(n).cardinality() != Partitions(n).cardinality(algorithm=
↪'pari')])
0
```

For negative inputs, the result is zero (the algorithm is ignored):

```
sage: Partitions(-5).cardinality()
0
```

```
>>> from sage.all import *
>>> Partitions(-Integer(5)).cardinality()
0
```

REFERENCES:

- Wikipedia article Partition_(number_theory)

first()

Return the lexicographically first partition of a positive integer n. This is the partition `[n]`.

EXAMPLES:

```
sage: Partitions(4).first()
[4]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4)).first()
[4]
```

last()

Return the lexicographically last partition of the positive integer n. This is the all-ones partition.

EXAMPLES:

```
sage: Partitions(4).last()
[1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Partitions(Integer(4)).last()
[1, 1, 1, 1]
```

next(p)

Return the lexicographically next partition after the partition `p`.

EXAMPLES:

```
sage: Partitions(4).next([4])
[3, 1]
sage: Partitions(4).next([1,1,1,1]) is None
True
```

```
>>> from sage.all import *
>>> Partitions(Integer(4)).next([Integer(4)])
[3, 1]
>>> Partitions(Integer(4)).next([Integer(1),Integer(1),Integer(1),
↪Integer(1)]) is None
True
```

prev (*p*)

Return the lexicographically previous partition before partition `p`.

EXAMPLES:

```
sage: Partitions(4).prev([3, 1])
[4]
sage: Partitions(4).prev([4]) is None
True
```

```
>>> from sage.all import *
>>> Partitions(Integer(4)).prev([Integer(3), Integer(1)])
[4]
>>> Partitions(Integer(4)).prev([Integer(4)]) is None
True
```

random_element (*measure='uniform'*)

Return a random partitions of n for the specified measure.

INPUT:

- `measure` – `'uniform'` or `'Plancherel'` (default: `'uniform'`)

See also

- *random_element_uniform()*
- *random_element_plancherel()*

EXAMPLES:

```
sage: Partitions(5).random_element()   # random                          #␣
↪needs sage.libs.flint
[2, 1, 1, 1]
sage: Partitions(5).random_element(measure='Plancherel')   # random      #␣
↪needs sage.libs.flint
[2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Partitions(Integer(5)).random_element()   # random                   ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.libs.flint
[2, 1, 1, 1]
>>> Partitions(Integer(5)).random_element(measure='Plancherel')   # random    ␣
↪     # needs sage.libs.flint
[2, 1, 1, 1]
```

random_element_plancherel()

Return a random partition of n (for the Plancherel measure).

This probability distribution comes from the uniform distribution on permutations via the Robinson-Schensted correspondence.

See Wikipedia article Plancherel_measure and *Partition.plancherel_measure()*.

EXAMPLES:

```
sage: Partitions(5).random_element_plancherel()    # random
[2, 1, 1, 1]
sage: Partitions(20).random_element_plancherel()   # random
[9, 3, 3, 2, 2, 1]
```

```
>>> from sage.all import *
>>> Partitions(Integer(5)).random_element_plancherel()    # random
[2, 1, 1, 1]
>>> Partitions(Integer(20)).random_element_plancherel()   # random
[9, 3, 3, 2, 2, 1]
```

ALGORITHM:

- insert by Robinson-Schensted a uniform random permutations of n and returns the shape of the resulting tableau. The complexity is $O(n\ln(n))$ which is likely optimal. However, the implementation could be optimized.

AUTHOR:

- Florent Hivert (2009-11-23)

random_element_uniform()

Return a random partition of n with uniform probability.

EXAMPLES:

```
sage: Partitions(5).random_element_uniform()   # random                      #␣
↪needs sage.libs.flint
[2, 1, 1, 1]
sage: Partitions(20).random_element_uniform()   # random                     #␣
↪needs sage.libs.flint
[9, 3, 3, 2, 2, 1]
```

```
>>> from sage.all import *
>>> Partitions(Integer(5)).random_element_uniform()   # random                ␣
↪     # needs sage.libs.flint
[2, 1, 1, 1]
>>> Partitions(Integer(20)).random_element_uniform()   # random               ␣
↪     # needs sage.libs.flint
[9, 3, 3, 2, 2, 1]
```

ALGORITHM:

- It is a python Implementation of RANDPAR, see [NW1978]. The complexity is unknown, there may be better algorithms.

> **Todo**
>
> Check in Knuth AOCP4.

- There is also certainly a lot of room for optimizations, see comments in the code.

AUTHOR:

- Florent Hivert (2009-11-23)

subset(***kwargs*)

Return a subset of `self` with the additional optional arguments.

EXAMPLES:

```
sage: P = Partitions(5); P
Partitions of the integer 5
sage: P.subset(starting=[3,1])
Partitions of the integer 5 starting with [3, 1]
```

```
>>> from sage.all import *
>>> P = Partitions(Integer(5)); P
Partitions of the integer 5
>>> P.subset(starting=[Integer(3),Integer(1)])
Partitions of the integer 5 starting with [3, 1]
```

class sage.combinat.partition.**Partitions_nk**(*n*, *k*)

Bases: *Partitions*

Partitions of the integer n of length equal to k.

cardinality(*algorithm='hybrid'*)

Return the number of partitions of the specified size with the specified length.

INPUT:

- `algorithm` – (default: `'hybrid'`) the algorithm to compute the cardinality and can be one of the following:
  - `'hybrid'` – use a hybrid algorithm which uses heuristics to reduce the complexity
  - `'gap'` – use GAP

EXAMPLES:

```
sage: v = Partitions(5, length=2).list(); v
[[4, 1], [3, 2]]
sage: len(v)
2
sage: Partitions(5, length=2).cardinality()
2
```

```
>>> from sage.all import *
>>> v = Partitions(Integer(5), length=Integer(2)).list(); v
[[4, 1], [3, 2]]
>>> len(v)
2
>>> Partitions(Integer(5), length=Integer(2)).cardinality()
2
```

More generally, the number of partitions of n of length 2 is $\left\lfloor \frac{n}{2} \right\rfloor$:

```
sage: all( Partitions(n, length=2).cardinality()
....:      == n // 2 for n in range(10) )
True
```

```
>>> from sage.all import *
>>> all( Partitions(n, length=Integer(2)).cardinality()
...      == n // Integer(2) for n in range(Integer(10)) )
True
```

The number of partitions of n of length 1 is 1 for n positive:

```
sage: all( Partitions(n, length=1).cardinality() == 1
....:      for n in range(1, 10) )
True
```

```
>>> from sage.all import *
>>> all( Partitions(n, length=Integer(1)).cardinality() == Integer(1)
...      for n in range(Integer(1), Integer(10)) )
True
```

Further examples:

```
sage: # needs sage.libs.flint
sage: Partitions(5, length=3).cardinality()
2
sage: Partitions(6, length=3).cardinality()
3
sage: Partitions(8, length=4).cardinality()
5
sage: Partitions(8, length=5).cardinality()
3
sage: Partitions(15, length=6).cardinality()
26
sage: Partitions(0, length=0).cardinality()
1
sage: Partitions(0, length=1).cardinality()
0
sage: Partitions(1, length=0).cardinality()
0
sage: Partitions(1, length=4).cardinality()
0
```

```
>>> from sage.all import *
>>> # needs sage.libs.flint
>>> Partitions(Integer(5), length=Integer(3)).cardinality()
2
>>> Partitions(Integer(6), length=Integer(3)).cardinality()
3
>>> Partitions(Integer(8), length=Integer(4)).cardinality()
5
>>> Partitions(Integer(8), length=Integer(5)).cardinality()
3
>>> Partitions(Integer(15), length=Integer(6)).cardinality()
26
>>> Partitions(Integer(0), length=Integer(0)).cardinality()
1
>>> Partitions(Integer(0), length=Integer(1)).cardinality()
0
>>> Partitions(Integer(1), length=Integer(0)).cardinality()
0
>>> Partitions(Integer(1), length=Integer(4)).cardinality()
0
```

subset(***kwargs*)

Return a subset of `self` with the additional optional arguments.

EXAMPLES:

```
sage: P = Partitions(5, length=2); P
Partitions of the integer 5 of length 2
sage: P.subset(max_part=3)
Partitions of 5 having length 2 and whose parts are at most 3
```

```
>>> from sage.all import *
>>> P = Partitions(Integer(5), length=Integer(2)); P
Partitions of the integer 5 of length 2
>>> P.subset(max_part=Integer(3))
Partitions of 5 having length 2 and whose parts are at most 3
```

class sage.combinat.partition.**Partitions_parts_in**(*n*, *parts*)

Bases: `Partitions`

Partitions of n with parts in a given set S.

This is invoked indirectly when calling `Partitions(n, parts_in=parts)`, where `parts` is a list of pairwise distinct integers.

cardinality()

Return the number of partitions with parts in `self`. Wraps GAP's `NrRestrictedPartitions`.

EXAMPLES:

```
sage: Partitions(15, parts_in=[2,3,7]).cardinality()                    #␣
↪needs sage.libs.gap
5
```

```
>>> from sage.all import *
>>> Partitions(Integer(15), parts_in=[Integer(2),Integer(3),Integer(7)]).
↪cardinality()                          # needs sage.libs.gap
5
```

If you can use all parts 1 through n, we'd better get $p(n)$:

```
sage: (Partitions(20, parts_in=[1..20]).cardinality()                    #␣
↪needs sage.libs.gap
....:    == Partitions(20).cardinality())
True
```

```
>>> from sage.all import *
>>> (Partitions(Integer(20), parts_in=(ellipsis_range(Integer(1),Ellipsis,
↪Integer(20)))).cardinality()                         # needs sage.libs.gap
...    == Partitions(Integer(20)).cardinality())
True
```

first()

Return the lexicographically first partition of a positive integer n with the specified parts, or None if no such partition exists.

EXAMPLES:

```
sage: Partitions(9, parts_in=[3,4]).first()
[3, 3, 3]
sage: Partitions(6, parts_in=[1..6]).first()
[6]
sage: Partitions(30, parts_in=[4,7,8,10,11]).first()
[11, 11, 8]
```

```
>>> from sage.all import *
>>> Partitions(Integer(9), parts_in=[Integer(3),Integer(4)]).first()
[3, 3, 3]
>>> Partitions(Integer(6), parts_in=(ellipsis_range(Integer(1),Ellipsis,
↪Integer(6)))).first()
[6]
>>> Partitions(Integer(30), parts_in=[Integer(4),Integer(7),Integer(8),
↪Integer(10),Integer(11)]).first()
[11, 11, 8]
```

last()

Return the lexicographically last partition of the positive integer n with the specified parts, or None if no such partition exists.

EXAMPLES:

```
sage: Partitions(15, parts_in=[2,3]).last()
[3, 2, 2, 2, 2, 2, 2]
sage: Partitions(30, parts_in=[4,7,8,10,11]).last()
[7, 7, 4, 4, 4, 4]
sage: Partitions(10, parts_in=[3,6]).last() is None
True
```

(continues on next page)

(continued from previous page)

```
sage: Partitions(50, parts_in=[11,12,13]).last()
[13, 13, 12, 12]
sage: Partitions(30, parts_in=[4,7,8,10,11]).last()
[7, 7, 4, 4, 4, 4]
```

```
>>> from sage.all import *
>>> Partitions(Integer(15), parts_in=[Integer(2),Integer(3)]).last()
[3, 2, 2, 2, 2, 2, 2]
>>> Partitions(Integer(30), parts_in=[Integer(4),Integer(7),Integer(8),
↪Integer(10),Integer(11)]).last()
[7, 7, 4, 4, 4, 4]
>>> Partitions(Integer(10), parts_in=[Integer(3),Integer(6)]).last() is None
True
>>> Partitions(Integer(50), parts_in=[Integer(11),Integer(12),Integer(13)]).
↪last()
[13, 13, 12, 12]
>>> Partitions(Integer(30), parts_in=[Integer(4),Integer(7),Integer(8),
↪Integer(10),Integer(11)]).last()
[7, 7, 4, 4, 4, 4]
```

class sage.combinat.partition.**Partitions_starting**(*n*, *starting_partition*)

Bases: *Partitions*

All partitions with a given start.

first()

Return the first partition in self.

EXAMPLES:

```
sage: Partitions(3, starting=[2,1]).first()
[2, 1]
sage: Partitions(3, starting=[1,1,1]).first()
[1, 1, 1]
sage: Partitions(3, starting=[1,1]).first()
False
sage: Partitions(3, starting=[3,1]).first()
[3]
sage: Partitions(3, starting=[2,2]).first()
[2, 1]
```

```
>>> from sage.all import *
>>> Partitions(Integer(3), starting=[Integer(2),Integer(1)]).first()
[2, 1]
>>> Partitions(Integer(3), starting=[Integer(1),Integer(1),Integer(1)]).
↪first()
[1, 1, 1]
>>> Partitions(Integer(3), starting=[Integer(1),Integer(1)]).first()
False
>>> Partitions(Integer(3), starting=[Integer(3),Integer(1)]).first()
[3]
>>> Partitions(Integer(3), starting=[Integer(2),Integer(2)]).first()
[2, 1]
```

next (*part*)

Return the next partition after `part` in `self`.

EXAMPLES:

```
sage: Partitions(3, starting=[2,1]).next(Partition([2,1]))
[1, 1, 1]
```

```
>>> from sage.all import *
>>> Partitions(Integer(3), starting=[Integer(2),Integer(1)]).
↪next(Partition([Integer(2),Integer(1)]))
[1, 1, 1]
```

class sage.combinat.partition.**Partitions_with_constraints**(**args*, ***kwds*)

Bases: *IntegerListsLex*

Partitions which satisfy a set of constraints.

EXAMPLES:

```
sage: P = Partitions(6, inner=[1,1], max_slope=-1)
sage: list(P)
[[5, 1], [4, 2], [3, 2, 1]]
```

```
>>> from sage.all import *
>>> P = Partitions(Integer(6), inner=[Integer(1),Integer(1)], max_slope=-
↪Integer(1))
>>> list(P)
[[5, 1], [4, 2], [3, 2, 1]]
```

Element

alias of *Partition*

options = Current options for Partitions - convention: English - diagram_str: * - display: list - latex: young_diagram - latex_diagram_str: \ast

class sage.combinat.partition.**RegularPartitions**(*ell*, *is_infinite=False*)

Bases: *Partitions*

Base class for ℓ-regular partitions.

Let ℓ be a positive integer. A partition λ is *ℓ-regular* if $m_i < \ell$ for all i, where m_i is the multiplicity of i in λ.

> **Note**
>
> This is conjugate to the notion of *ℓ-restricted* partitions, where the difference between any two consecutive parts is $< \ell$.

INPUT:

- `ell` – the positive integer ℓ
- `is_infinite` – boolean; if the subset of ℓ-regular partitions is infinite

ell()

Return the value ℓ.

EXAMPLES:

```
sage: P = Partitions(regular=2)
sage: P.ell()
2
```

```
>>> from sage.all import *
>>> P = Partitions(regular=Integer(2))
>>> P.ell()
2
```

class sage.combinat.partition.**RegularPartitions_all**(*ell*)

Bases: *RegularPartitions*

The class of all ℓ-regular partitions.

INPUT:

- ell – the positive integer ℓ

See also

RegularPartitions

class sage.combinat.partition.**RegularPartitions_bounded**(*ell*, *k*)

Bases: *RegularPartitions*

The class of ℓ-regular k-bounded partitions.

INPUT:

- ell – the integer ℓ
- k – integer; the value k

See also

RegularPartitions

class sage.combinat.partition.**RegularPartitions_n**(*n*, *ell*)

Bases: *RegularPartitions*, *Partitions_n*

The class of ℓ-regular partitions of n.

INPUT:

- n – the integer n to partition
- ell – the integer ℓ

See also

RegularPartitions

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Partitions(5, regular=3)
sage: P.cardinality()                                                   #␣
↪needs sage.libs.flint
5
sage: P = Partitions(5, regular=6)
sage: P.cardinality()                                                   #␣
↪needs sage.libs.flint
7
sage: P.cardinality() == Partitions(5).cardinality()                   #␣
↪needs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> P = Partitions(Integer(5), regular=Integer(3))
>>> P.cardinality()                                                   #␣
↪needs sage.libs.flint
5
>>> P = Partitions(Integer(5), regular=Integer(6))
>>> P.cardinality()                                                   #␣
↪needs sage.libs.flint
7
>>> P.cardinality() == Partitions(Integer(5)).cardinality()            ␣
↪       # needs sage.libs.flint
True
```

class `sage.combinat.partition.`**RegularPartitions_truncated**(*ell*, *max_len*)

Bases: *RegularPartitions*

The class of ℓ-regular partitions with max length k.

INPUT:

- `ell` – the integer ℓ
- `max_len` – integer; the maximum length

See also

RegularPartitions

max_length()

Return the maximum length of the partitions of `self`.

EXAMPLES:

```
sage: P = Partitions(regular=4, max_length=3)
sage: P.max_length()
3
```

```
>>> from sage.all import *
>>> P = Partitions(regular=Integer(4), max_length=Integer(3))
>>> P.max_length()
3
```

class sage.combinat.partition.**RestrictedPartitions_all**(*ell*)

Bases: *RestrictedPartitions_generic*

The class of all ℓ-restricted partitions.

INPUT:

- ell – the positive integer ℓ

See also

RestrictedPartitions_generic

class sage.combinat.partition.**RestrictedPartitions_generic**(*ell*, *is_infinite=False*)

Bases: *Partitions*

Base class for ℓ-restricted partitions.

Let ℓ be a positive integer. A partition λ is *ℓ-restricted* if $\lambda_i - \lambda_{i+1} < \ell$ for all i, including rows of length 0.

Note

This is conjugate to the notion of *ℓ-regular* partitions, where the multiplicity of any parts is at most ℓ.

INPUT:

- ell – the positive integer ℓ
- is_infinite – boolean; if the subset of ℓ-restricted partitions is infinite

ell()

Return the value ℓ.

EXAMPLES:

```
sage: P = Partitions(restricted=2)
sage: P.ell()
2
```

```
>>> from sage.all import *
>>> P = Partitions(restricted=Integer(2))
>>> P.ell()
2
```

class sage.combinat.partition.**RestrictedPartitions_n**(*n*, *ell*)

Bases: *RestrictedPartitions_generic*, *Partitions_n*

The class of ℓ-restricted partitions of n.

INPUT:

- n – the integer n to partition

- `ell` – the integer ℓ

See also

RestrictedPartitions_generic

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Partitions(5, restricted=3)
sage: P.cardinality()                                                    #␣
↪needs sage.libs.flint
5
sage: P = Partitions(5, restricted=6)
sage: P.cardinality()                                                    #␣
↪needs sage.libs.flint
7
sage: P.cardinality() == Partitions(5).cardinality()                     #␣
↪needs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> P = Partitions(Integer(5), restricted=Integer(3))
>>> P.cardinality()                                                    #␣
↪needs sage.libs.flint
5
>>> P = Partitions(Integer(5), restricted=Integer(6))
>>> P.cardinality()                                                    #␣
↪needs sage.libs.flint
7
>>> P.cardinality() == Partitions(Integer(5)).cardinality()              ␣
↪      # needs sage.libs.flint
True
```

sage.combinat.partition.**number_of_partitions**(*n*, *algorithm='default'*)

Return the number of partitions of n with, optionally, at most k parts.

The options of *number_of_partitions()* are being deprecated Issue #13072 in favour of *Partitions_n.cardinality()* so that *number_of_partitions()* can become a stripped down version of the fastest algorithm available (currently this is using FLINT).

INPUT:

- `n` – integer
- **`algorithm` – (default: `'default'`)**
  [Will be deprecated except in Partition().cardinality()]
  - **`'default'` – if `k` is not `None`, then use Gap (very slow);**
    if `k` is `None`, use FLINT
  - `'flint'` – use FLINT

EXAMPLES:

```
sage: v = Partitions(5).list(); v
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: len(v)
7
```

```
>>> from sage.all import *
>>> v = Partitions(Integer(5)).list(); v
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
>>> len(v)
7
```

The input must be a nonnegative integer or a `ValueError` is raised.

```
sage: number_of_partitions(-5)
Traceback (most recent call last):
...
ValueError: n (=-5) must be a nonnegative integer
```

```
>>> from sage.all import *
>>> number_of_partitions(-Integer(5))
Traceback (most recent call last):
...
ValueError: n (=-5) must be a nonnegative integer
```

```
sage: # needs sage.libs.flint
sage: number_of_partitions(10)
42
sage: number_of_partitions(3)
3
sage: number_of_partitions(10)
42
sage: number_of_partitions(40)
37338
sage: number_of_partitions(100)
190569292
sage: number_of_partitions(100000)
27493510569775696512677516320986352688173429315980054758203125984302147328114964173
↪305550507416607366215901578447742962489404930630702004617927644930335101160793424
↪571901557189435097253124661084520063695589344642487168287898321823450092628538314
↪045970213071306745106244192273112389997022844086093709355316296978515695698921961
↪08480158600569421098519
```

```
>>> from sage.all import *
>>> # needs sage.libs.flint
>>> number_of_partitions(Integer(10))
42
>>> number_of_partitions(Integer(3))
3
>>> number_of_partitions(Integer(10))
42
>>> number_of_partitions(Integer(40))
37338
```

(continues on next page)

(continued from previous page)

```
>>> number_of_partitions(Integer(100))
190569292
>>> number_of_partitions(Integer(100000))
27493510569775696512677516320986352688173429315980054758203125984302147328114964173
↪30555074166073662159015784477429624894049306307020046179276449303351011607934245
↪71901557189435097253124661084520063695589344642487168287898321823450092628538314
↪04597021307130674510624419227311238999702284408609370935531629697851569569892196
↪108480158600569421098519
```

A generating function for the number of partitions p_n is given by the reciprocal of Euler's function:

$$\sum_{n=0}^{\infty} p_n x^n = \prod_{k=1}^{\infty} \left(\frac{1}{1-x^k} \right).$$

We use Sage to verify that the first several coefficients do instead agree:

```
sage: q = PowerSeriesRing(QQ, 'q', default_prec=9).gen()
sage: prod([(1-q^k)^(-1) for k in range(1,9)])  # partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
sage: [number_of_partitions(k) for k in range(2,10)]                          #␣
↪needs sage.libs.flint
[2, 3, 5, 7, 11, 15, 22, 30]
```

```
>>> from sage.all import *
>>> q = PowerSeriesRing(QQ, 'q', default_prec=Integer(9)).gen()
>>> prod([(Integer(1)-q**k)**(-Integer(1)) for k in range(Integer(1),
↪Integer(9))])  # partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
>>> [number_of_partitions(k) for k in range(Integer(2),Integer(10))]          ␣
↪              # needs sage.libs.flint
[2, 3, 5, 7, 11, 15, 22, 30]
```

REFERENCES:

- Wikipedia article Partition_(number_theory)

sage.combinat.partition.**number_of_partitions_length**(*n*, *k*, *algorithm='hybrid'*)

Return the number of partitions of n with length k.

This is a wrapper for GAP's `NrPartitions` function.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: from sage.combinat.partition import number_of_partitions_length
sage: number_of_partitions_length(5, 2)
2
sage: number_of_partitions_length(10, 2)
5
sage: number_of_partitions_length(10, 4)
9
sage: number_of_partitions_length(10, 0)
0
sage: number_of_partitions_length(10, 1)
```

(continues on next page)

(continued from previous page)

```
1
sage: number_of_partitions_length(0, 0)
1
sage: number_of_partitions_length(0, 1)
0
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> from sage.combinat.partition import number_of_partitions_length
>>> number_of_partitions_length(Integer(5), Integer(2))
2
>>> number_of_partitions_length(Integer(10), Integer(2))
5
>>> number_of_partitions_length(Integer(10), Integer(4))
9
>>> number_of_partitions_length(Integer(10), Integer(0))
0
>>> number_of_partitions_length(Integer(10), Integer(1))
1
>>> number_of_partitions_length(Integer(0), Integer(0))
1
>>> number_of_partitions_length(Integer(0), Integer(1))
0
```

sage.combinat.partition.**number_of_partitions_max_length_max_part**(k, b)

Return the number of partitions of n with at most k parts, all of which are at most b.

EXAMPLES:

This could also be computed using the q-binomial coefficient:

```
sage: from sage.combinat.partition import number_of_partitions_max_length_max_
↪part as f
sage: all(f(n, k, b) == q_binomial(k + b, b)[n] for n in range(5) for k in␣
↪range(n+1) for b in range(n+1))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition import number_of_partitions_max_length_max_part␣
↪as f
>>> all(f(n, k, b) == q_binomial(k + b, b)[n] for n in range(Integer(5)) for k in␣
↪range(n+Integer(1)) for b in range(n+Integer(1)))
True
```

However, although the q-binomial coefficient is faster for individual invocations, it seems that the caching we use here is essential for some computations:

```
sage: def A(n):
....:     s1 = number_of_partitions(n)
....:     s2 = sum(Partitions(m, max_part=l, length=k).cardinality()
....:              * Partitions(n-m-l^2, min_length=k+2*l).cardinality()
....:              for l in range(1, (n+1).isqrt())
....:              for m in range((n-l^2-2*l)*l//(l+1)+1)
```

(continues on next page)

(continued from previous page)

```
....:                for k in range(ceil(m/l), min(m, n-m-l^2-2*l)+1))
....:      return s1 + s2

sage: A(100)
10934714090
```

```
>>> from sage.all import *
>>> def A(n):
...     s1 = number_of_partitions(n)
...     s2 = sum(Partitions(m, max_part=l, length=k).cardinality()
...                * Partitions(n-m-l**Integer(2), min_length=k+Integer(2)*l).
↪cardinality()
...                for l in range(Integer(1), (n+Integer(1)).isqrt())
...                for m in range((n-l**Integer(2)-Integer(2)*l)*l//
↪(l+Integer(1))+Integer(1))
...                for k in range(ceil(m/l), min(m, n-m-l**Integer(2)-
↪Integer(2)*l)+Integer(1)))
...     return s1 + s2

>>> A(Integer(100))
10934714090
```

5.1.166 Partition/Diagram Algebras

class sage.combinat.partition_algebra.**PartitionAlgebraElement_ak**

Bases: *PartitionAlgebraElement_generic*

class sage.combinat.partition_algebra.**PartitionAlgebraElement_bk**

Bases: *PartitionAlgebraElement_generic*

class sage.combinat.partition_algebra.**PartitionAlgebraElement_generic**

Bases: IndexedFreeModuleElement

class sage.combinat.partition_algebra.**PartitionAlgebraElement_pk**

Bases: *PartitionAlgebraElement_generic*

class sage.combinat.partition_algebra.**PartitionAlgebraElement_prk**

Bases: *PartitionAlgebraElement_generic*

class sage.combinat.partition_algebra.**PartitionAlgebraElement_rk**

Bases: *PartitionAlgebraElement_generic*

class sage.combinat.partition_algebra.**PartitionAlgebraElement_sk**

Bases: *PartitionAlgebraElement_generic*

class sage.combinat.partition_algebra.**PartitionAlgebraElement_tk**

Bases: *PartitionAlgebraElement_generic*

class sage.combinat.partition_algebra.**PartitionAlgebra_ak**(*R*, *k*, *n*, *name=None*)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_ak(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_ak(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_bk**(*R*, *k*, *n*, *name=None*)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_bk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_bk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_generic**(*R*, *cclass*, *n*, *k*, *name=None*, *prefix=None*)

Bases: *CombinatorialFreeModule*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: s = PartitionAlgebra_sk(QQ, 3, 1)
sage: TestSuite(s).run()
sage: s == loads(dumps(s))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> s = PartitionAlgebra_sk(QQ, Integer(3), Integer(1))
>>> TestSuite(s).run()
>>> s == loads(dumps(s))
True
```

one_basis()

Return the basis index for the unit of the algebra.

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: s = PartitionAlgebra_sk(ZZ, 3, 1)
```

(continues on next page)

(continued from previous page)

```
sage: len(s.one().support())    # indirect doctest
1
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> s = PartitionAlgebra_sk(ZZ, Integer(3), Integer(1))
>>> len(s.one().support())    # indirect doctest
1
```

product_on_basis(*left*, *right*)

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: s = PartitionAlgebra_sk(QQ, 3, 1)
sage: t12 = s(Set([Set([1,-2]),Set([2,-1]),Set([3,-3])]))
sage: t12^2 == s(1) #indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> s = PartitionAlgebra_sk(QQ, Integer(3), Integer(1))
>>> t12 = s(Set([Set([Integer(1),-Integer(2)]),Set([Integer(2),-Integer(1)]),
↪Set([Integer(3),-Integer(3)])]))
>>> t12**Integer(2) == s(Integer(1)) #indirect doctest
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_pk**(*R*, *k*, *n*, *name=None*)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_pk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_pk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_prk**(*R*, *k*, *n*, *name=None*)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_prk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_prk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_rk**(*R*, *k*, *n*, *name=None*)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_rk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_rk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_sk**(*R*, *k*, *n*, *name=None*)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_sk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_sk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_tk**(*R*, *k*, *n*, *name=None*)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_tk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_tk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

sage.combinat.partition_algebra.**SetPartitionsAk**(k)

Return the combinatorial class of set partitions of type A_k.

EXAMPLES:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}

sage: A3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: A3.random_element()  #random
{{1, 3, -3, -1}, {2, -2}}

sage: A3.cardinality()
203

sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5.cardinality()
52

sage: A2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: A2p5.random_element() #random
{{-1}, {-2}, {3, -3}, {1, 2}}
```

```
>>> from sage.all import *
>>> A3 = SetPartitionsAk(Integer(3)); A3
Set partitions of {1, ..., 3, -1, ..., -3}

>>> A3.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> A3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
>>> A3.random_element()  #random
{{1, 3, -3, -1}, {2, -2}}

>>> A3.cardinality()
203

>>> A2p5 = SetPartitionsAk(RealNumber('2.5')); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
>>> A2p5.cardinality()
52

>>> A2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> A2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
```

(continues on next page)

(continued from previous page)

```
>>> A2p5.random_element() #random
{{-1}, {-2}, {3, -3}, {1, 2}}
```

class sage.combinat.partition_algebra.**SetPartitionsAk_k**(*k*)

Bases: *SetPartitions_set*

Element

alias of *SetPartitionsXkElement*

class sage.combinat.partition_algebra.**SetPartitionsAkhalf_k**(*k*)

Bases: *SetPartitions_set*

Element

alias of *SetPartitionsXkElement*

sage.combinat.partition_algebra.**SetPartitionsBk**(*k*)

Return the combinatorial class of set partitions of type B_k.

These are the set partitions where every block has size 2.

EXAMPLES:

```
sage: B3 = SetPartitionsBk(3); B3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2

sage: B3.first() #random
{{2, -2}, {1, -3}, {3, -1}}
sage: B3.last() #random
{{1, 2}, {3, -2}, {-3, -1}}
sage: B3.random_element() #random
{{2, -1}, {1, -3}, {3, -2}}

sage: B3.cardinality()
15

sage: B2p5 = SetPartitionsBk(2.5); B2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↪with block size 2

sage: B2p5.first() #random
{{2, -1}, {3, -3}, {1, -2}}
sage: B2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
sage: B2p5.random_element() #random
{{2, -2}, {3, -3}, {1, -1}}

sage: B2p5.cardinality()
3
```

```
>>> from sage.all import *
>>> B3 = SetPartitionsBk(Integer(3)); B3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2

>>> B3.first() #random
```

(continues on next page)

(continued from previous page)

```
{{2, -2}, {1, -3}, {3, -1}}
>>> B3.last() #random
{{1, 2}, {3, -2}, {-3, -1}}
>>> B3.random_element() #random
{{2, -1}, {1, -3}, {3, -2}}

>>> B3.cardinality()
15

>>> B2p5 = SetPartitionsBk(RealNumber('2.5')); B2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↪with block size 2

>>> B2p5.first() #random
{{2, -1}, {3, -3}, {1, -2}}
>>> B2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
>>> B2p5.random_element() #random
{{2, -2}, {3, -3}, {1, -1}}

>>> B2p5.cardinality()
3
```

class sage.combinat.partition_algebra.**SetPartitionsBk_k**(*k*)

Bases: *SetPartitionsAk_k*

cardinality()

Return the number of set partitions in B_k where k is an integer.

This is given by (2k)!! = (2k-1)*(2k-3)*…*5*3*1.

EXAMPLES:

```
sage: SetPartitionsBk(3).cardinality()
15
sage: SetPartitionsBk(2).cardinality()
3
sage: SetPartitionsBk(1).cardinality()
1
sage: SetPartitionsBk(4).cardinality()
105
sage: SetPartitionsBk(5).cardinality()
945
```

```
>>> from sage.all import *
>>> SetPartitionsBk(Integer(3)).cardinality()
15
>>> SetPartitionsBk(Integer(2)).cardinality()
3
>>> SetPartitionsBk(Integer(1)).cardinality()
1
>>> SetPartitionsBk(Integer(4)).cardinality()
105
```

(continues on next page)

(continued from previous page)

```
>>> SetPartitionsBk(Integer(5)).cardinality()
945
```

class sage.combinat.partition_algebra.**SetPartitionsBkhalf_k**(*k*)

Bases: *SetPartitionsAkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsIk**(*k*)

Return the combinatorial class of set partitions of type I_k.

These are set partitions with a propagating number of less than k. Note that the identity set partition $\{\{1, -1\}, \ldots, \{k, -k\}\}$ is not in I_k.

EXAMPLES:

```
sage: I3 = SetPartitionsIk(3); I3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number < 3
sage: I3.cardinality()
197

sage: I3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: I3.random_element() #random
{{-1}, {-3, -2}, {2, 3}, {1}}

sage: I2p5 = SetPartitionsIk(2.5); I2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and propagating number < 3
sage: I2p5.cardinality()
50

sage: I2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: I2p5.random_element() #random
{{-1}, {-2}, {1, 3, -3}, {2}}
```

```
>>> from sage.all import *
>>> I3 = SetPartitionsIk(Integer(3)); I3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number < 3
>>> I3.cardinality()
197

>>> I3.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> I3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
>>> I3.random_element() #random
{{-1}, {-3, -2}, {2, 3}, {1}}
```

(continues on next page)

(continued from previous page)

```
>>> I2p5 = SetPartitionsIk(RealNumber('2.5')); I2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↪propagating number < 3
>>> I2p5.cardinality()
50

>>> I2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> I2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
>>> I2p5.random_element() #random
{{-1}, {-2}, {1, 3, -3}, {2}}
```

class sage.combinat.partition_algebra.**SetPartitionsIk_k**(*k*)

Bases: *SetPartitionsAk_k*

cardinality()

class sage.combinat.partition_algebra.**SetPartitionsIkhalf_k**(*k*)

Bases: *SetPartitionsAkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsPRk**(*k*)

Return the combinatorial class of set partitions of type PR_k.

EXAMPLES:

```
sage: SetPartitionsPRk(3)
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive
 and negative entry in each block and that are planar
```

```
>>> from sage.all import *
>>> SetPartitionsPRk(Integer(3))
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive
 and negative entry in each block and that are planar
```

class sage.combinat.partition_algebra.**SetPartitionsPRk_k**(*k*)

Bases: *SetPartitionsRk_k*

cardinality()

class sage.combinat.partition_algebra.**SetPartitionsPRkhalf_k**(*k*)

Bases: *SetPartitionsRkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsPk**(*k*)

Return the combinatorial class of set partitions of type P_k.

These are the planar set partitions.

EXAMPLES:

```
sage: P3 = SetPartitionsPk(3); P3
Set partitions of {1, ..., 3, -1, ..., -3} that are planar
sage: P3.cardinality()
132

sage: P3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: P3.random_element() #random
{{1, 2, -1}, {-3}, {3, -2}}

sage: P2p5 = SetPartitionsPk(2.5); P2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↪that are planar
sage: P2p5.cardinality()
42

sage: P2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: P2p5.random_element() #random
{{1, 2, 3, -3}, {-1, -2}}
```

```
>>> from sage.all import *
>>> P3 = SetPartitionsPk(Integer(3)); P3
Set partitions of {1, ..., 3, -1, ..., -3} that are planar
>>> P3.cardinality()
132

>>> P3.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> P3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
>>> P3.random_element() #random
{{1, 2, -1}, {-3}, {3, -2}}

>>> P2p5 = SetPartitionsPk(RealNumber('2.5')); P2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↪that are planar
>>> P2p5.cardinality()
42

>>> P2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> P2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
>>> P2p5.random_element() #random
{{1, 2, 3, -3}, {-1, -2}}
```

class sage.combinat.partition_algebra.**SetPartitionsPk_k**(k)

Bases: *SetPartitionsAk_k*

cardinality()

class sage.combinat.partition_algebra.**SetPartitionsPkhalf_k**(*k*)

Bases: *SetPartitionsAkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsRk**(*k*)

Return the combinatorial class of set partitions of type R_k.

EXAMPLES:

```
sage: SetPartitionsRk(3)
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive
 and negative entry in each block
```

```
>>> from sage.all import *
>>> SetPartitionsRk(Integer(3))
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive
 and negative entry in each block
```

class sage.combinat.partition_algebra.**SetPartitionsRk_k**(*k*)

Bases: *SetPartitionsAk_k*

cardinality()

class sage.combinat.partition_algebra.**SetPartitionsRkhalf_k**(*k*)

Bases: *SetPartitionsAkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsSk**(*k*)

Return the combinatorial class of set partitions of type S_k.

There is a bijection between these set partitions and the permutations of $1, \ldots, k$.

EXAMPLES:

```
sage: S3 = SetPartitionsSk(3); S3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number 3
sage: S3.cardinality()
6

sage: S3.list()  #random
[{{2, -2}, {3, -3}, {1, -1}},
 {{1, -1}, {2, -3}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}},
 {{1, -2}, {2, -3}, {3, -1}},
 {{1, -3}, {2, -1}, {3, -2}},
 {{1, -3}, {2, -2}, {3, -1}}]
sage: S3.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: S3.last() #random
{{1, -3}, {2, -2}, {3, -1}}
sage: S3.random_element() #random
{{1, -3}, {2, -1}, {3, -2}}
```

(continues on next page)

(continued from previous page)

```
sage: S3p5 = SetPartitionsSk(3.5); S3p5
Set partitions of {1, ..., 4, -1, ..., -4} with 4 and -4 in the same block and␣
↪propagating number 4
sage: S3p5.cardinality()
6

sage: S3p5.list() #random
[{{2, -2}, {3, -3}, {1, -1}, {4, -4}},
 {{2, -3}, {1, -1}, {4, -4}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}, {4, -4}},
 {{2, -3}, {1, -2}, {4, -4}, {3, -1}},
 {{1, -3}, {2, -1}, {4, -4}, {3, -2}},
 {{1, -3}, {2, -2}, {4, -4}, {3, -1}}]
sage: S3p5.first() #random
{{2, -2}, {3, -3}, {1, -1}, {4, -4}}
sage: S3p5.last() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
sage: S3p5.random_element() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
```

```
>>> from sage.all import *
>>> S3 = SetPartitionsSk(Integer(3)); S3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number 3
>>> S3.cardinality()
6

>>> S3.list()  #random
[{{2, -2}, {3, -3}, {1, -1}},
 {{1, -1}, {2, -3}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}},
 {{1, -2}, {2, -3}, {3, -1}},
 {{1, -3}, {2, -1}, {3, -2}},
 {{1, -3}, {2, -2}, {3, -1}}]
>>> S3.first() #random
{{2, -2}, {3, -3}, {1, -1}}
>>> S3.last() #random
{{1, -3}, {2, -2}, {3, -1}}
>>> S3.random_element() #random
{{1, -3}, {2, -1}, {3, -2}}

>>> S3p5 = SetPartitionsSk(RealNumber('3.5')); S3p5
Set partitions of {1, ..., 4, -1, ..., -4} with 4 and -4 in the same block and␣
↪propagating number 4
>>> S3p5.cardinality()
6

>>> S3p5.list() #random
[{{2, -2}, {3, -3}, {1, -1}, {4, -4}},
 {{2, -3}, {1, -1}, {4, -4}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}, {4, -4}},
 {{2, -3}, {1, -2}, {4, -4}, {3, -1}},
```

(continues on next page)

(continued from previous page)

```
 {{1, -3}, {2, -1}, {4, -4}, {3, -2}},
 {{1, -3}, {2, -2}, {4, -4}, {3, -1}}]
>>> S3p5.first() #random
{{2, -2}, {3, -3}, {1, -1}, {4, -4}}
>>> S3p5.last() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
>>> S3p5.random_element() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
```

class sage.combinat.partition_algebra.**SetPartitionsSk_k**(*k*)

Bases: *SetPartitionsAk_k*

cardinality()

Return k!.

class sage.combinat.partition_algebra.**SetPartitionsSkhalf_k**(*k*)

Bases: *SetPartitionsAkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsTk**(*k*)

Return the combinatorial class of set partitions of type T_k.

These are planar set partitions where every block is of size 2.

EXAMPLES:

```
sage: T3 = SetPartitionsTk(3); T3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2 and that are planar
sage: T3.cardinality()
5

sage: T3.first() #random
{{1, -3}, {2, 3}, {-1, -2}}
sage: T3.last() #random
{{1, 2}, {3, -1}, {-3, -2}}
sage: T3.random_element() #random
{{1, -3}, {2, 3}, {-1, -2}}

sage: T2p5 = SetPartitionsTk(2.5); T2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and␣
↪with block size 2 and that are planar
sage: T2p5.cardinality()
2

sage: T2p5.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: T2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
```

```
>>> from sage.all import *
>>> T3 = SetPartitionsTk(Integer(3)); T3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2 and that are planar
>>> T3.cardinality()
```

(continues on next page)

(continued from previous page)

```
5

>>> T3.first() #random
{{1, -3}, {2, 3}, {-1, -2}}
>>> T3.last() #random
{{1, 2}, {3, -1}, {-3, -2}}
>>> T3.random_element() #random
{{1, -3}, {2, 3}, {-1, -2}}

>>> T2p5 = SetPartitionsTk(RealNumber('2.5')); T2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and␣
↪with block size 2 and that are planar
>>> T2p5.cardinality()
2

>>> T2p5.first() #random
{{2, -2}, {3, -3}, {1, -1}}
>>> T2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
```

class sage.combinat.partition_algebra.**SetPartitionsTk_k**(*k*)

Bases: *SetPartitionsBk_k*

cardinality()

class sage.combinat.partition_algebra.**SetPartitionsTkhalf_k**(*k*)

Bases: *SetPartitionsBkhalf_k*

cardinality()

class sage.combinat.partition_algebra.**SetPartitionsXkElement**(*parent*, *s*, *check=True*)

Bases: *SetPartition*

An element for the classes of `SetPartitionXk` where `X` is some letter.

check()

Check to make sure this is a set partition.

EXAMPLES:

```
sage: A2p5 = SetPartitionsAk(2.5)
sage: x = A2p5.first(); x
{{-3, -2, -1, 1, 2, 3}}
sage: x.check()
sage: y = A2p5.next(x); y
{{-3, 3}, {-2, -1, 1, 2}}
sage: y.check()
```

```
>>> from sage.all import *
>>> A2p5 = SetPartitionsAk(RealNumber('2.5'))
>>> x = A2p5.first(); x
{{-3, -2, -1, 1, 2, 3}}
>>> x.check()
>>> y = A2p5.next(x); y
```

(continues on next page)

(continued from previous page)

```
{{-3, 3}, {-2, -1, 1, 2}}
>>> y.check()
```

sage.combinat.partition_algebra.**identity**(*k*)

Return the identity set partition $1, -1, \ldots, k, -k$.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.identity(2)
{{2, -2}, {1, -1}}
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> pa.identity(Integer(2))
{{2, -2}, {1, -1}}
```

sage.combinat.partition_algebra.**is_planar**(*sp*)

Return True if the diagram corresponding to the set partition is planar; otherwise, it returns False.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.is_planar( pa.to_set_partition([[1,-2],[2,-1]]))
False
sage: pa.is_planar( pa.to_set_partition([[1,-1],[2,-2]]))
True
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> pa.is_planar( pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-
↪Integer(1)]]))
False
>>> pa.is_planar( pa.to_set_partition([[Integer(1),-Integer(1)],[Integer(2),-
↪Integer(2)]]))
True
```

sage.combinat.partition_algebra.**pair_to_graph**(*sp1*, *sp2*)

Return a graph consisting of the disjoint union of the graphs of set partitions sp1 and sp2 along with edges joining the bottom row (negative numbers) of sp1 to the top row (positive numbers) of sp2.

The vertices of the graph sp1 appear in the result as pairs (k, 1), whereas the vertices of the graph sp2 appear as pairs (k, 2).

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: g = pa.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> sp1 = pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> sp2 = pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> g = pa.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices
```

```
sage: g.vertices(sort=False) #random
[(1, 2), (-1, 1), (-2, 2), (-1, 2), (-2, 1), (2, 1), (2, 2), (1, 1)]
sage: g.edges(sort=False) #random
[((1, 2), (-1, 1), None),
 ((1, 2), (-2, 2), None),
 ((-1, 1), (2, 1), None),
 ((-1, 2), (2, 2), None),
 ((-2, 1), (1, 1), None),
 ((-2, 1), (2, 2), None)]
```

```
>>> from sage.all import *
>>> g.vertices(sort=False) #random
[(1, 2), (-1, 1), (-2, 2), (-1, 2), (-2, 1), (2, 1), (2, 2), (1, 1)]
>>> g.edges(sort=False) #random
[((1, 2), (-1, 1), None),
 ((1, 2), (-2, 2), None),
 ((-1, 1), (2, 1), None),
 ((-1, 2), (2, 2), None),
 ((-2, 1), (1, 1), None),
 ((-2, 1), (2, 2), None)]
```

Another example which used to be wrong until Issue #15958:

```
sage: sp3 = pa.to_set_partition([[1, -1], [2], [-2]])
sage: sp4 = pa.to_set_partition([[1], [-1], [2], [-2]])
sage: g = pa.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

sage: g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges(sort=True)
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

```
>>> from sage.all import *
>>> sp3 = pa.to_set_partition([[Integer(1), -Integer(1)], [Integer(2)], [-
↪Integer(2)]])
>>> sp4 = pa.to_set_partition([[Integer(1)], [-Integer(1)], [Integer(2)], [-
↪Integer(2)]])
>>> g = pa.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

>>> g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
>>> g.edges(sort=True)
```

(continues on next page)

(continued from previous page)

```
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

sage.combinat.partition_algebra.**propagating_number**(*sp*)

Return the propagating number of the set partition sp.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,2],[-2,-1]])
sage: pa.propagating_number(sp1)
2
sage: pa.propagating_number(sp2)
0
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> sp1 = pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> sp2 = pa.to_set_partition([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])
>>> pa.propagating_number(sp1)
2
>>> pa.propagating_number(sp2)
0
```

sage.combinat.partition_algebra.**set_partition_composition**(*sp1*, *sp2*)

Return a tuple consisting of the composition of the set partitions sp1 and sp2 and the number of components removed from the middle rows of the graph.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: pa.set_partition_composition(sp1, sp2) == (pa.identity(2), 0)
True
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> sp1 = pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> sp2 = pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> pa.set_partition_composition(sp1, sp2) == (pa.identity(Integer(2)),␣
↪Integer(0))
True
```

sage.combinat.partition_algebra.**to_graph**(*sp*)

Return a graph representing the set partition sp.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: g = pa.to_graph( pa.to_set_partition([[1,-2],[2,-1]])); g
Graph on 4 vertices

sage: g.vertices(sort=False) #random
[1, 2, -2, -1]
sage: g.edges(sort=False) #random
[(1, -2, None), (2, -1, None)]
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> g = pa.to_graph( pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-
↪Integer(1)]])); g
Graph on 4 vertices

>>> g.vertices(sort=False) #random
[1, 2, -2, -1]
>>> g.edges(sort=False) #random
[(1, -2, None), (2, -1, None)]
```

sage.combinat.partition_algebra.**to_set_partition**(*l*, *k=None*)

Convert a list of a list of numbers to a set partitions.

Each list of numbers in the outer list specifies the numbers contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of 1, …, k, -1, …, -k. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.to_set_partition([[1,-1],[2,-2]]) == pa.identity(2)
True
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> pa.to_set_partition([[Integer(1),-Integer(1)],[Integer(2),-Integer(2)]]) ==␣
↪pa.identity(Integer(2))
True
```

5.1.167 Kleshchev partitions

A partition (tuple) μ is Kleshchev if it can be recursively obtained by adding a sequence of good nodes to the empty `PartitionTuple` of the same `level()` and *multicharge*. In this way, the set of Kleshchev multipartitions becomes a realization of a Kashiwara crystal `sage.combinat.crystals.crystals` for a irreducible integral highest weight representation of $U_q(\widehat{\mathfrak{sl}}_e)$.

The Kleshchev multipartitions first appeared in the work of Ariki and Mathas [AM2000] where it was shown that they index the irreducible representations of the cyclotomic Hecke algebras of type A [AK1994]. Soon afterwards Ariki [Ariki2001] showed that the set of Kleshchev multipartitions naturally label the irreducible representations of these algebras. As a far reaching generalization of these ideas the Ariki-Brundan-Kleshchev categorification theorem [Ariki1996] [BK2009] says that these algebras categorify the irreducible integral highest weight representations of the quantum group

$U_q(\widehat{\mathfrak{sl}}_e)$ of the affine special linear group. Under this categorification, q corresponds to the grading shift on the cyclotomic Hecke algebras, where the grading from the Brundan-Kleshchev graded isomorphism theorem to the *KLR algebras* of type A [BK2009].

The group algebras of the symmetric group in characteristic p are an important special case of the cyclotomic Hecke algebras of type A. In this case, depending on your prefer convention, the set of Kleshchev partitions is the set of *`p`-regular* or *`p`-restricted* `Partitions`. In this case, Kleshchev [Kle1995] proved that the *modular branching rules* were given by adding and removing *good nodes*; see `good_cells()`. Lascoux, Leclerc and Thibon [LLT1996] noticed that Kleshchev's branching rules coincided with Kashiwara's crystal operators for the fundamental representation of $L(\Lambda_0)$ of $U_q(\widehat{\mathfrak{sl}}_p)$ and their celebrated *LLT conjecture* said that decomposition matrices of the `sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra` of the symmetric group should be computable using the canonical basis of $L(\Lambda_0)$. This was proved and generalised to all cyclotomic Hecke algebras of type A by Ariki [Ariki1996] and then further generalized to the graded setting by Brundan and Kleshchev [BK2009].

The main class for accessing Kleshchev partition (tuples) is `KleshchevPartitions`. Unfortunately, just as with the symmetric group, different authors use different conventions when defining Kleshchev partitions, which depends on whether you read components from left to right, or right to left, and whether you read the nodes in the partition in each component from top to bottom or bottom to top. The `KleshchevPartitions` class supports these four different conventions:

```
sage: KleshchevPartitions(2, [0,0], size=2, convention='left regular')[:]
[([1], [1]), ([2], [])]
sage: KleshchevPartitions(2, [0,0], size=2, convention='left restricted')[:]
[([1], [1]), ([], [1, 1])]
sage: KleshchevPartitions(2, [0,0], size=2, convention='right regular')[:]
[([1], [1]), ([], [2])]
sage: KleshchevPartitions(2, [0,0], size=2, convention='right restricted')[:]
[([1], [1]), ([1, 1], [])]
```

```
>>> from sage.all import *
>>> KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='left regular')[:]
[([1], [1]), ([2], [])]
>>> KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='left restricted')[:]
[([1], [1]), ([], [1, 1])]
>>> KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='right regular')[:]
[([1], [1]), ([], [2])]
>>> KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='right restricted')[:]
[([1], [1]), ([1, 1], [])]
```

By default, the `left restricted` convention is used. As a shorthand, `LG`, `LS`, `RG` and `RS`, respectively, can be used to specify the `convention` With the `left` convention the partition tuples should be ordered with the most dominant partitions in the partition tuple on the left and with the `right` convention the most dominant partition is on the right.

The `KleshchevPartitions` class can automatically convert between these four different conventions:

```
sage: KPlg = KleshchevPartitions(2, [0,0], size=2, convention='left regular')
sage: KPls = KleshchevPartitions(2, [0,0], size=2, convention='left restricted')
sage: [KPlg(mu) for mu in KPls] # indirect doc test
[([1], [1]), ([2], [])]
```

```
>>> from sage.all import *
>>> KPlg = KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='left regular')
>>> KPls = KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='left restricted')
>>> [KPlg(mu) for mu in KPls] # indirect doc test
[([1], [1]), ([2], [])]
```

AUTHORS:

- Andrew Mathas and Travis Scrimshaw (2018-05-1): Initial version

class sage.combinat.partition_kleshchev.**KleshchevCrystalMixin**

Bases: object

Mixin class for the crystal structure of a Kleshchev partition.

Epsilon()

Return ε of self.

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, [0,2], convention="left regular")
sage: x = C([[5,4,1],[3,2,1,1]])
sage: x.Epsilon()
3*Lambda[1]
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="left regular")
>>> x = C([[Integer(5),Integer(4),Integer(1)],[Integer(3),Integer(2),
↪Integer(1),Integer(1)]])
>>> x.Epsilon()
3*Lambda[1]
```

Phi()

Return ϕ of self.

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, [0,2], convention="left regular")
sage: x = C([[5,4,1],[3,2,1,1]])
sage: x.Phi()
3*Lambda[0] + 2*Lambda[1]
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="left regular")
>>> x = C([[Integer(5),Integer(4),Integer(1)],[Integer(3),Integer(2),
↪Integer(1),Integer(1)]])
>>> x.Phi()
3*Lambda[0] + 2*Lambda[1]
```

epsilon(i)

Return the Kashiwara crystal operator ε_i applied to self.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, [0,2], convention="left regular")
sage: x = C([[5,4,1],[3,2,1,1]])
sage: [x.epsilon(i) for i in C.index_set()]
[0, 3, 0]
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="left regular")
>>> x = C([[Integer(5),Integer(4),Integer(1)],[Integer(3),Integer(2),
↪Integer(1),Integer(1)]])
>>> [x.epsilon(i) for i in C.index_set()]
[0, 3, 0]
```

phi(*i*)

Return the Kashiwara crystal operator φ_i applied to `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, [0,2], convention="left regular")
sage: x = C([[5,4,1],[3,2,1,1]])
sage: [x.phi(i) for i in C.index_set()]
[3, 2, 0]
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="left regular")
>>> x = C([[Integer(5),Integer(4),Integer(1)],[Integer(3),Integer(2),
↪Integer(1),Integer(1)]])
>>> [x.phi(i) for i in C.index_set()]
[3, 2, 0]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, [0,2], convention="left regular")
sage: x = C([[5,4,1], [3,2,1,1]])
sage: x.weight()
3*Lambda[0] - Lambda[1] - 5*delta
sage: x.Phi() - x.Epsilon()
3*Lambda[0] - Lambda[1]

sage: C = crystals.KleshchevPartitions(3, [0,2], convention="right regular")
sage: y = C([[5,1,1], [4,2,2,1,1]])
sage: y.weight()
```

(continues on next page)

(continued from previous page)

```
6*Lambda[0] - 4*Lambda[1] - 4*delta
sage: y.Phi() - y.Epsilon()
6*Lambda[0] - 4*Lambda[1]

sage: C = crystals.KleshchevPartitions(3, [0,2], convention="left regular")
sage: y = C([[5,1,1], [4,2,2,1,1]])
sage: y.weight()
6*Lambda[0] - 4*Lambda[1] - 4*delta
sage: y.Phi() - y.Epsilon()
6*Lambda[0] - 4*Lambda[1]
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="left regular")
>>> x = C([[Integer(5),Integer(4),Integer(1)], [Integer(3),Integer(2),
↪Integer(1),Integer(1)]])
>>> x.weight()
3*Lambda[0] - Lambda[1] - 5*delta
>>> x.Phi() - x.Epsilon()
3*Lambda[0] - Lambda[1]

>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="right regular")
>>> y = C([[Integer(5),Integer(1),Integer(1)], [Integer(4),Integer(2),
↪Integer(2),Integer(1),Integer(1)]])
>>> y.weight()
6*Lambda[0] - 4*Lambda[1] - 4*delta
>>> y.Phi() - y.Epsilon()
6*Lambda[0] - 4*Lambda[1]

>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="left regular")
>>> y = C([[Integer(5),Integer(1),Integer(1)], [Integer(4),Integer(2),
↪Integer(2),Integer(1),Integer(1)]])
>>> y.weight()
6*Lambda[0] - 4*Lambda[1] - 4*delta
>>> y.Phi() - y.Epsilon()
6*Lambda[0] - 4*Lambda[1]
```

class sage.combinat.partition_kleshchev.**KleshchevPartition**(*parent*, *mu*)

Bases: *Partition*

Abstract base class for Kleshchev partitions. See *KleshchevPartitions*.

cogood_cells(*i=None*)

Return a list of the cells of `self` that are cogood.

The cogood i-cell is the 'last' conormal i-cell. As with the conormal cells we can choose to read either up or down the partition as specified by *convention()*.

INPUT:

- `i` – (optional) a residue

OUTPUT:

If no residue `i` is specified then a dictionary of cogood cells is returned, which gives the cogood cells for `0 <= i < e`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, convention='regular')
sage: KP([5,4,4,3,2]).cogood_cells()
{0: (1, 4), 1: (4, 2)}
sage: KP([5,4,4,3,2]).cogood_cells(0)
(1, 4)
sage: KP([5,4,4,3,2]).cogood_cells(1)
(4, 2)
sage: KP = KleshchevPartitions(4, convention='restricted')
sage: KP([5,4,4,3,2]).cogood_cells()
{1: (0, 5), 2: (4, 2), 3: (1, 4)}
sage: KP([5,4,4,3,2]).cogood_cells(0)
sage: KP([5,4,4,3,2]).cogood_cells(2)
(4, 2)
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), convention='regular')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).cogood_
↪cells()
{0: (1, 4), 1: (4, 2)}
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).cogood_
↪cells(Integer(0))
(1, 4)
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).cogood_
↪cells(Integer(1))
(4, 2)
>>> KP = KleshchevPartitions(Integer(4), convention='restricted')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).cogood_
↪cells()
{1: (0, 5), 2: (4, 2), 3: (1, 4)}
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).cogood_
↪cells(Integer(0))
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).cogood_
↪cells(Integer(2))
(4, 2)
```

conormal_cells(*i=None*)

Return a dictionary of the cells of `self` which are conormal.

Following [Kle1995], the *conormal* cells are computed by reading up (or down) the rows of the partition and marking all of the addable and removable cells of e-residue i and then recursively removing all adjacent pairs of removable and addable cells (in that order) from this list. The addable i-cells that remain at the end of the this process are the conormal i-cells.

When computing conormal cells you can either read the cells in order from top to bottom (this corresponds to labeling the simple modules of the symmetric group by regular partitions) or from bottom to top (corresponding to labeling the simples by restricted partitions). By default we read down the partition but this can be changed by setting `convention = 'RS'`.

INPUT:

- `i` – (optional) a residue

OUTPUT:

If no residue `i` is specified then a dictionary of conormal cells is returned, which gives the conormal cells for `0 <= i < e`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, convention='regular')
sage: KP([5,4,4,3,2]).conormal_cells()
{0: [(1, 4)], 1: [(5, 0), (4, 2)]}
sage: KP([5,4,4,3,2]).conormal_cells(0)
[(1, 4)]
sage: KP([5,4,4,3,2]).conormal_cells(1)
[(5, 0), (4, 2)]
sage: KP = KleshchevPartitions(3, convention='restricted')
sage: KP([5,4,4,3,2]).conormal_cells()
{0: [(1, 4), (3, 3)], 2: [(0, 5)]}
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), convention='regular')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).conormal_
↪cells()
{0: [(1, 4)], 1: [(5, 0), (4, 2)]}
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).conormal_
↪cells(Integer(0))
[(1, 4)]
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).conormal_
↪cells(Integer(1))
[(5, 0), (4, 2)]
>>> KP = KleshchevPartitions(Integer(3), convention='restricted')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).conormal_
↪cells()
{0: [(1, 4), (3, 3)], 2: [(0, 5)]}
```

good_cell_sequence()

Return a sequence of good nodes from the empty partition to `self`, or `None` if no such sequence exists.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, convention='regular')
sage: KP([5,4,4,3,2]).good_cell_sequence()
[(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2),
 (3, 0), (2, 1), (1, 2), (3, 1), (0, 3), (1, 3),
 (2, 2), (3, 2), (4, 0), (4, 1), (0, 4), (2, 3)]
sage: KP = KleshchevPartitions(3, convention='restricted')
sage: KP([5,4,4,3,2]).good_cell_sequence()
[(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0),
 (0, 3), (2, 1), (1, 2), (1, 3), (3, 0), (3, 1),
 (2, 2), (4, 0), (2, 3), (3, 2), (0, 4), (4, 1)]
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), convention='regular')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_cell_
↪sequence()
```

(continues on next page)

(continued from previous page)

```
[(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2),
 (3, 0), (2, 1), (1, 2), (3, 1), (0, 3), (1, 3),
 (2, 2), (3, 2), (4, 0), (4, 1), (0, 4), (2, 3)]
>>> KP = KleshchevPartitions(Integer(3), convention='restricted')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_cell_
↪sequence()
[(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0),
 (0, 3), (2, 1), (1, 2), (1, 3), (3, 0), (3, 1),
 (2, 2), (4, 0), (2, 3), (3, 2), (0, 4), (4, 1)]
```

good_cells(*i=None*)

Return a list of the cells of `self` that are good.

The good i-cell is the 'first' normal i-cell. As with the normal cells we can choose to read either up or down the partition as specified by *convention()*.

INPUT:

- `i` – (optional) a residue

OUTPUT:

If no residue `i` is specified then a dictionary of good cells is returned, which gives the good cells for `0 <= i < e`.

EXAMPLES:

```
sage: KP3 = KleshchevPartitions(3, convention='regular')
sage: KP3([5,4,4,3,2]).good_cells()
{1: (2, 3)}
sage: KP3([5,4,4,3,2]).good_cells(1)
(2, 3)
sage: KP4 = KleshchevPartitions(4, convention='restricted')
sage: KP4([5,4,4,3,2]).good_cells()
{1: (2, 3)}
sage: KP4([5,4,4,3,2]).good_cells(0)
sage: KP4([5,4,4,3,2]).good_cells(1)
(2, 3)
```

```
>>> from sage.all import *
>>> KP3 = KleshchevPartitions(Integer(3), convention='regular')
>>> KP3([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_cells()
{1: (2, 3)}
>>> KP3([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_
↪cells(Integer(1))
(2, 3)
>>> KP4 = KleshchevPartitions(Integer(4), convention='restricted')
>>> KP4([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_cells()
{1: (2, 3)}
>>> KP4([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_
↪cells(Integer(0))
>>> KP4([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_
↪cells(Integer(1))
(2, 3)
```

good_residue_sequence()

Return a sequence of good nodes from the empty partition to `self`, or `None` if no such sequence exists.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, convention='regular')
sage: KP([5,4,4,3,2]).good_residue_sequence()
[0, 2, 1, 1, 0, 2, 0, 2, 1, 1, 0, 2, 0, 2, 2, 0, 1, 1]
sage: KP = KleshchevPartitions(3, convention='restricted')
sage: KP([5,4,4,3,2]).good_residue_sequence()
[0, 1, 2, 2, 0, 1, 0, 2, 1, 2, 0, 1, 0, 2, 1, 2, 1, 0]
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), convention='regular')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_residue_
↪sequence()
[0, 2, 1, 1, 0, 2, 0, 2, 1, 1, 0, 2, 0, 2, 2, 0, 1, 1]
>>> KP = KleshchevPartitions(Integer(3), convention='restricted')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).good_residue_
↪sequence()
[0, 1, 2, 2, 0, 1, 0, 2, 1, 2, 0, 1, 0, 2, 1, 2, 1, 0]
```

is_regular()

Return `True` if `self` is a e-regular partition tuple.

A partition tuple is e-regular if we can get to the empty partition tuple by successively removing a sequence of good cells in the down direction. Equivalently, all partitions are 0-regular and if $e > 0$ then a partition is e-regular if no e nonzero parts of `self` are equal.

EXAMPLES:

```
sage: KP = KleshchevPartitions(2)
sage: KP([2,1,1]).is_regular()
False
sage: KP = KleshchevPartitions(3)
sage: KP([2,1,1]).is_regular()
True
sage: KP([]).is_regular()
True
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(2))
>>> KP([Integer(2),Integer(1),Integer(1)]).is_regular()
False
>>> KP = KleshchevPartitions(Integer(3))
>>> KP([Integer(2),Integer(1),Integer(1)]).is_regular()
True
>>> KP([]).is_regular()
True
```

is_restricted()

Return `True` if `self` is an e-restricted partition tuple.

A partition tuple is e-restricted if we can get to the empty partition tuple by successively removing a sequence of good cells in the up direction. Equivalently, all partitions are 0-restricted and if $e > 0$ then a partition is

e-restricted if the difference of successive parts of `self` are always strictly less than e.

EXAMPLES:

```
sage: KP = KleshchevPartitions(2, convention='regular')
sage: KP([3,1]).is_restricted()
False
sage: KP = KleshchevPartitions(3, convention='regular')
sage: KP([3,1]).is_restricted()
True
sage: KP([]).is_restricted()
True
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(2), convention='regular')
>>> KP([Integer(3),Integer(1)]).is_restricted()
False
>>> KP = KleshchevPartitions(Integer(3), convention='regular')
>>> KP([Integer(3),Integer(1)]).is_restricted()
True
>>> KP([]).is_restricted()
True
```

mullineux_conjugate()

Return the partition tuple that is the Mullineux conjugate of `self`.

It follows from results in [BK2009], [Mat2015] that if ν is the Mullineux conjugate of the Kleshchev partition tuple μ then the simple module $D^\nu = (D^\mu)^{\text{sgn}}$ is obtained from D^μ by twisting by the sgn-automorphism with is the Iwahori-Hecke algebra analogue of tensoring with the one dimensional sign representation.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, convention='regular')
sage: KP([5,4,4,3,2]).mullineux_conjugate()
[9, 7, 1, 1]
sage: KP = KleshchevPartitions(3, convention='restricted')
sage: KP([5,4,4,3,2]).mullineux_conjugate()
[3, 2, 2, 2, 2, 2, 2, 1, 1, 1]
sage: KP = KleshchevPartitions(3, [2], convention='regular')
sage: mc = KP([5,4,4,3,2]).mullineux_conjugate(); mc
[9, 7, 1, 1]
sage: mc.parent().multicharge()
(1,)
sage: KP = KleshchevPartitions(3, [2], convention='restricted')
sage: mc = KP([5,4,4,3,2]).mullineux_conjugate(); mc
[3, 2, 2, 2, 2, 2, 2, 1, 1, 1]
sage: mc.parent().multicharge()
(1,)
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), convention='regular')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).mullineux_
↪conjugate()
[9, 7, 1, 1]
```

(continues on next page)

(continued from previous page)

```
>>> KP = KleshchevPartitions(Integer(3), convention='restricted')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).mullineux_
↪conjugate()
[3, 2, 2, 2, 2, 2, 2, 1, 1, 1]
>>> KP = KleshchevPartitions(Integer(3), [Integer(2)], convention='regular')
>>> mc = KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).
↪mullineux_conjugate(); mc
[9, 7, 1, 1]
>>> mc.parent().multicharge()
(1,)
>>> KP = KleshchevPartitions(Integer(3), [Integer(2)], convention='restricted
↪')
>>> mc = KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).
↪mullineux_conjugate(); mc
[3, 2, 2, 2, 2, 2, 2, 1, 1, 1]
>>> mc.parent().multicharge()
(1,)
```

normal_cells(*i=None*)

Return a dictionary of the cells of the partition that are normal.

Following [Kle1995], the *normal* cells are computed by reading up (or down) the rows of the partition and marking all of the addable and removable cells of e-residue i and then recursively removing all adjacent pairs of removable and addable cells (in that order) from this list. The removable i-cells that remain at the end of the this process are the normal i-cells.

When computing normal cells you can either read the cells in order from top to bottom (this corresponds to labeling the simple modules of the symmetric group by regular partitions) or from bottom to top (corresponding to labeling the simples by restricted partitions). By default we read down the partition but this can be changed by setting `convention = 'RS'`.

INPUT:

- `i` – (optional) a residue

OUTPUT:

If no residue `i` is specified then a dictionary of normal cells is returned, which gives the normal cells for `0 <= i < e`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, convention='regular')
sage: KP([5,4,4,3,2]).normal_cells()
{1: [(2, 3), (0, 4)]}
sage: KP([5,4,4,3,2]).normal_cells(1)
[(2, 3), (0, 4)]
sage: KP = KleshchevPartitions(3, convention='restricted')
sage: KP([5,4,4,3,2]).normal_cells()
{0: [(4, 1)], 2: [(3, 2)]}
sage: KP([5,4,4,3,2]).normal_cells(2)
[(3, 2)]
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), convention='regular')
```

(continues on next page)

(continued from previous page)

```
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).normal_
↪cells()
{1: [(2, 3), (0, 4)]}
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).normal_
↪cells(Integer(1))
[(2, 3), (0, 4)]
>>> KP = KleshchevPartitions(Integer(3), convention='restricted')
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).normal_
↪cells()
{0: [(4, 1)], 2: [(3, 2)]}
>>> KP([Integer(5),Integer(4),Integer(4),Integer(3),Integer(2)]).normal_
↪cells(Integer(2))
[(3, 2)]
```

class sage.combinat.partition_kleshchev.**KleshchevPartitionCrystal**(*parent*, *mu*)

Bases: *KleshchevPartition*, *KleshchevCrystalMixin*

Kleshchev partition with the crystal structure.

e(*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, convention="left regular")
sage: x = C([5,4,1])
sage: x.e(0)
sage: x.e(1)
[5, 4]
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), convention="left regular")
>>> x = C([Integer(5),Integer(4),Integer(1)])
>>> x.e(Integer(0))
>>> x.e(Integer(1))
[5, 4]
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, convention="left regular")
sage: x = C([5,4,1])
sage: x.f(0)
[5, 5, 1]
sage: x.f(1)
```

(continues on next page)

(continued from previous page)

```
sage: x.f(2)
[5, 4, 2]
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), convention="left regular")
>>> x = C([Integer(5),Integer(4),Integer(1)])
>>> x.f(Integer(0))
[5, 5, 1]
>>> x.f(Integer(1))
>>> x.f(Integer(2))
[5, 4, 2]
```

class sage.combinat.partition_kleshchev.**KleshchevPartitionTuple**(*parent*, *mu*)

Bases: *PartitionTuple*

Abstract base class for Kleshchev partition tuples. See *KleshchevPartitions*.

cogood_cells(*i=None*)

Return a list of the cells of the partition that are cogood.

The cogood i-cell is the 'last' conormal i-cell. As with the conormal cells we can choose to read either up or down the partition as specified by *convention()*.

INPUT:

- `i` – (optional) a residue

OUTPUT:

If no residue `i` is specified then a dictionary of cogood cells is returned, which gives the cogood cells for `0 <= i < e`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, [0,1])
sage: pt = KP([[4, 2], [5, 3, 1]])
sage: pt.cogood_cells()
{0: (1, 2, 1), 1: (1, 3, 0)}
sage: pt.cogood_cells(0)
(1, 2, 1)
sage: KP = KleshchevPartitions(4, [0,1], convention="left regular")
sage: pt = KP([[5, 2, 2], [6, 1, 1]])
sage: pt.cogood_cells()
{1: (0, 0, 5), 2: (1, 3, 0)}
sage: pt.cogood_cells(0) is None
True
sage: pt.cogood_cells(1) is None
False
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)])
>>> pt = KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]])
>>> pt.cogood_cells()
{0: (1, 2, 1), 1: (1, 3, 0)}
>>> pt.cogood_cells(Integer(0))
```

(continues on next page)

(continued from previous page)

```
(1, 2, 1)
>>> KP = KleshchevPartitions(Integer(4), [Integer(0),Integer(1)], convention=
→"left regular")
>>> pt = KP([[Integer(5), Integer(2), Integer(2)], [Integer(6), Integer(1),␣
→Integer(1)]])
>>> pt.cogood_cells()
{1: (0, 0, 5), 2: (1, 3, 0)}
>>> pt.cogood_cells(Integer(0)) is None
True
>>> pt.cogood_cells(Integer(1)) is None
False
```

conormal_cells(*i=None*)

Return a dictionary of the cells of the partition that are conormal.

Following [Kle1995], the *conormal* cells are computed by reading up (or down) the rows of the partition and marking all of the addable and removable cells of e-residue i and then recursively removing all adjacent pairs of removable and addable cells (in that order) from this list. The addable i-cells that remain at the end of the this process are the conormal i-cells.

When computing conormal cells you can either read the cells in order from top to bottom (this corresponds to labeling the simple modules of the symmetric group by regular partitions) or from bottom to top (corresponding to labeling the simples by restricted partitions). By default we read down the partition but this can be changed by setting `convention = 'RS'`.

INPUT:

- `i` – (optional) a residue

OUTPUT:

If no residue `i` is specified then a dictionary of conormal cells is returned, which gives the conormal cells for `0 <= i < e`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, [0,1], convention="left regular")
sage: KP([[4, 2], [5, 3, 1]]).conormal_cells()
{0: [(1, 2, 1), (1, 1, 3), (1, 0, 5)],
1: [(1, 3, 0), (0, 2, 0), (0, 1, 2), (0, 0, 4)]}
sage: KP([[4, 2], [5, 3, 1]]).conormal_cells(1)
[(1, 3, 0), (0, 2, 0), (0, 1, 2), (0, 0, 4)]
sage: KP([[4, 2], [5, 3, 1]]).conormal_cells(2)
[]
sage: KP = KleshchevPartitions(3, [0,1], convention="right restricted")
sage: KP([[4, 2], [5, 3, 1]]).conormal_cells(0)
[(1, 0, 5), (1, 1, 3), (1, 2, 1)]
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)], convention=
→"left regular")
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
→conormal_cells()
{0: [(1, 2, 1), (1, 1, 3), (1, 0, 5)],
1: [(1, 3, 0), (0, 2, 0), (0, 1, 2), (0, 0, 4)]}
```

(continues on next page)

(continued from previous page)

```
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪conormal_cells(Integer(1))
[(1, 3, 0), (0, 2, 0), (0, 1, 2), (0, 0, 4)]
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪conormal_cells(Integer(2))
[]
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)], convention=
↪"right restricted")
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪conormal_cells(Integer(0))
[(1, 0, 5), (1, 1, 3), (1, 2, 1)]
```

good_cell_sequence()

Return a sequence of good nodes from the empty partition to `self`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3,[0,1])
sage: KP([[4, 2], [5, 3, 1]]).good_cell_sequence()
[(0, 0, 0), (1, 0, 0), (1, 0, 1), (0, 0, 1), (0, 1, 0),
 (1, 1, 0), (1, 1, 1), (1, 0, 2), (1, 2, 0), (0, 0, 2),
 (0, 1, 1), (1, 0, 3), (0, 0, 3), (1, 1, 2), (1, 0, 4)]
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3),[Integer(0),Integer(1)])
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).good_
↪cell_sequence()
[(0, 0, 0), (1, 0, 0), (1, 0, 1), (0, 0, 1), (0, 1, 0),
 (1, 1, 0), (1, 1, 1), (1, 0, 2), (1, 2, 0), (0, 0, 2),
 (0, 1, 1), (1, 0, 3), (0, 0, 3), (1, 1, 2), (1, 0, 4)]
```

good_cells(*i=None*)

Return a list of the cells of the partition tuple which are good.

The good i-cell is the 'first' normal i-cell. As with the normal cells we can choose to read either up or down the partition as specified by *convention()*.

INPUT:

- `i` – (optional) a residue

OUTPUT:

If no residue `i` is specified then a dictionary of good cells is returned, which gives the good cells for `0 <= i < e`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, [0,1])
sage: pt = KP([[4, 2], [5, 3, 1]])
sage: pt.good_cells()
{2: (1, 0, 4)}
sage: pt.good_cells(2)
(1, 0, 4)
sage: KP = KleshchevPartitions(4, [0,1], convention="left regular")
```

(continues on next page)

(continued from previous page)

```
sage: pt = KP([[5, 2, 2], [6, 2, 1]])
sage: pt.good_cells()
{0: (0, 0, 4), 2: (1, 0, 5), 3: (0, 2, 1)}
sage: pt.good_cells(1) is None
True
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)])
>>> pt = KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]])
>>> pt.good_cells()
{2: (1, 0, 4)}
>>> pt.good_cells(Integer(2))
(1, 0, 4)
>>> KP = KleshchevPartitions(Integer(4), [Integer(0),Integer(1)], convention=
↪"left regular")
>>> pt = KP([[Integer(5), Integer(2), Integer(2)], [Integer(6), Integer(2),␣
↪Integer(1)]])
>>> pt.good_cells()
{0: (0, 0, 4), 2: (1, 0, 5), 3: (0, 2, 1)}
>>> pt.good_cells(Integer(1)) is None
True
```

good_residue_sequence()

Return a sequence of good nodes from the empty partition to `self`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, [0,1])
sage: KP([[4, 2], [5, 3, 1]]).good_residue_sequence()
[0, 1, 2, 1, 2, 0, 1, 0, 2, 2, 0, 1, 0, 2, 2]
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)])
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).good_
↪residue_sequence()
[0, 1, 2, 1, 2, 0, 1, 0, 2, 2, 0, 1, 0, 2, 2]
```

is_regular()

Return `True` if `self` is a e-regular partition tuple.

A partition tuple is e-regular if we can get to the empty partition tuple by successively removing a sequence of good cells in the down direction.

EXAMPLES:

```
sage: KP = KleshchevPartitions(2, [0,2], convention="right restricted")
sage: KP([[3,2,1], [2,1,1]]).is_regular()
False
sage: KP = KleshchevPartitions(4, [0,2], convention="right restricted")
sage: KP([[3,2,1], [2,1,1]]).is_regular()
True
sage: KP([[], []]).is_regular()
True
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(2), [Integer(0),Integer(2)], convention=
↪"right restricted")
>>> KP([[Integer(3),Integer(2),Integer(1)], [Integer(2),Integer(1),
↪Integer(1)]]).is_regular()
False
>>> KP = KleshchevPartitions(Integer(4), [Integer(0),Integer(2)], convention=
↪"right restricted")
>>> KP([[Integer(3),Integer(2),Integer(1)], [Integer(2),Integer(1),
↪Integer(1)]]).is_regular()
True
>>> KP([[], []]).is_regular()
True
```

is_restricted()

Return True if self is an e-restricted partition tuple.

A partition tuple is e-restricted if we can get to the empty partition tuple by successively removing a sequence of good cells in the up direction.

EXAMPLES:

```
sage: KP = KleshchevPartitions(2, [0,2], convention="left regular")
sage: KP([[3,2,1], [3,1]]).is_restricted()
False
sage: KP = KleshchevPartitions(3, [0,2], convention="left regular")
sage: KP([[3,2,1], [3,1]]).is_restricted()
True
sage: KP([[], []]).is_restricted()
True
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(2), [Integer(0),Integer(2)], convention=
↪"left regular")
>>> KP([[Integer(3),Integer(2),Integer(1)], [Integer(3),Integer(1)]]).is_
↪restricted()
False
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(2)], convention=
↪"left regular")
>>> KP([[Integer(3),Integer(2),Integer(1)], [Integer(3),Integer(1)]]).is_
↪restricted()
True
>>> KP([[], []]).is_restricted()
True
```

mullineux_conjugate()

Return the partition that is the Mullineux conjugate of self.

It follows from results in [Kle1996] [Bru1998] that if ν is the Mullineux conjugate of the Kleshchev partition tuple μ then the simple module $D^\nu = (D^\mu)^{\text{sgn}}$ is obtained from D^μ by twisting by the sgn-automorphism with is the Hecke algebra analogue of tensoring with the one dimensional sign representation.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, [0,1])
sage: mc = KP([[4, 2], [5, 3, 1]]).mullineux_conjugate(); mc
([2, 2, 1, 1], [3, 2, 2, 1, 1])
sage: mc.parent()
Kleshchev partitions with e=3 and multicharge=(0,2)
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)])
>>> mc = KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪mullineux_conjugate(); mc
([2, 2, 1, 1], [3, 2, 2, 1, 1])
>>> mc.parent()
Kleshchev partitions with e=3 and multicharge=(0,2)
```

normal_cells(*i=None*)

Return a dictionary of the removable cells of the partition that are normal.

Following [Kle1995], the *normal* cells are computed by reading up (or down) the rows of the partition and marking all of the addable and removable cells of e-residue i and then recursively removing all adjacent pairs of removable and addable cells (in that order) from this list. The removable i-cells that remain at the end of the this process are the normal i-cells.

When computing normal cells you can either read the cells in order from top to bottom (this corresponds to labeling the simple modules of the symmetric group by regular partitions) or from bottom to top (corresponding to labeling the simples by restricted partitions). By default we read down the partition but this can be changed by setting `convention = 'RS'`.

INPUT:

- `i` – (optional) a residue

OUTPUT:

If no residue `i` is specified then a dictionary of normal cells is returned, which gives the normal cells for `0 <= i < e`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(3, [0,1], convention="left restricted")
sage: KP([[4, 2], [5, 3, 1]]).normal_cells()
{2: [(1, 0, 4), (1, 1, 2), (1, 2, 0)]}
sage: KP([[4, 2], [5, 3, 1]]).normal_cells(1)
[]
sage: KP = KleshchevPartitions(3, [0,1], convention="left regular")
sage: KP([[4, 2], [5, 3, 1]]).normal_cells()
{0: [(0, 1, 1), (0, 0, 3)], 2: [(1, 2, 0), (1, 1, 2), (1, 0, 4)]}
sage: KP = KleshchevPartitions(3, [0,1], convention="right regular")
sage: KP([[4, 2], [5, 3, 1]]).normal_cells()
{2: [(1, 2, 0), (1, 1, 2), (1, 0, 4)]}
sage: KP = KleshchevPartitions(3, [0,1], convention="right restricted")
sage: KP([[4, 2], [5, 3, 1]]).normal_cells()
{0: [(0, 0, 3), (0, 1, 1)], 2: [(1, 0, 4), (1, 1, 2), (1, 2, 0)]}
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)], convention=
```

(continues on next page)

(continued from previous page)

```
↪"left restricted")
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪normal_cells()
{2: [(1, 0, 4), (1, 1, 2), (1, 2, 0)]}
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪normal_cells(Integer(1))
[]
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)], convention=
↪"left regular")
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪normal_cells()
{0: [(0, 1, 1), (0, 0, 3)], 2: [(1, 2, 0), (1, 1, 2), (1, 0, 4)]}
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)], convention=
↪"right regular")
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪normal_cells()
{2: [(1, 2, 0), (1, 1, 2), (1, 0, 4)]}
>>> KP = KleshchevPartitions(Integer(3), [Integer(0),Integer(1)], convention=
↪"right restricted")
>>> KP([[Integer(4), Integer(2)], [Integer(5), Integer(3), Integer(1)]]).
↪normal_cells()
{0: [(0, 0, 3), (0, 1, 1)], 2: [(1, 0, 4), (1, 1, 2), (1, 2, 0)]}
```

class sage.combinat.partition_kleshchev.**KleshchevPartitionTupleCrystal**(*parent*, *mu*)

Bases: *KleshchevPartitionTuple*, *KleshchevCrystalMixin*

Kleshchev partition tuple with the crystal structure.

e(*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, [0,2], convention="left regular")
sage: x = C([[5,4,1],[3,2,1,1]])
sage: x.e(0)
sage: x.e(1)
([5, 4, 1], [2, 2, 1, 1])
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="left regular")
>>> x = C([[Integer(5),Integer(4),Integer(1)],[Integer(3),Integer(2),
↪Integer(1),Integer(1)]])
>>> x.e(Integer(0))
>>> x.e(Integer(1))
([5, 4, 1], [2, 2, 1, 1])
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: C = crystals.KleshchevPartitions(3, [0,2], convention="left regular")
sage: x = C([[5,4,1],[3,2,1,1]])
sage: x.f(0)
([5, 5, 1], [3, 2, 1, 1])
sage: x.f(1)
([5, 4, 1], [3, 2, 2, 1])
sage: x.f(2)
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="left regular")
>>> x = C([[Integer(5),Integer(4),Integer(1)],[Integer(3),Integer(2),
↪Integer(1),Integer(1)]])
>>> x.f(Integer(0))
([5, 5, 1], [3, 2, 1, 1])
>>> x.f(Integer(1))
([5, 4, 1], [3, 2, 2, 1])
>>> x.f(Integer(2))
```

class sage.combinat.partition_kleshchev.**KleshchevPartitions**

Bases: *PartitionTuples*

Kleshchev partitions.

A partition (tuple) μ is Kleshchev if it can be recursively obtained by adding a sequence of good nodes to the empty *PartitionTuple* of the same *level()* and multicharge.

There are four different conventions that are used in the literature for Kleshchev partitions, depending on whether we read partitions from top to bottom (regular) or bottom to top (restricted) and whether we read partition tuples from left to right or right to left. All of these conventions are supported:

```
sage: KleshchevPartitions(2, [0,0], size=2, convention='left regular')[:]
[([1], [1]), ([2], [])]
sage: KleshchevPartitions(2, [0,0], size=2, convention='left restricted')[:]
[([1], [1]), ([], [1, 1])]
sage: KleshchevPartitions(2, [0,0], size=2, convention='right regular')[:]
[([1], [1]), ([], [2])]
sage: KleshchevPartitions(2, [0,0], size=2, convention='right restricted')[:]
[([1], [1]), ([1, 1], [])]
```

```
>>> from sage.all import *
>>> KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='left regular')[:]
[([1], [1]), ([2], [])]
>>> KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='left restricted')[:]
[([1], [1]), ([], [1, 1])]
>>> KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='right regular')[:]
```

(continues on next page)

(continued from previous page)

```
[([1], [1]), ([], [2])]
>>> KleshchevPartitions(Integer(2), [Integer(0),Integer(0)], size=Integer(2),␣
↪convention='right restricted')[:]
[([1], [1]), ([1, 1], [])]
```

By default, the `left restricted` convention is used. As a shorthand, `LG`, `LS`, `RG` and `RS`, respectively, can be used to specify the `convention`. With the `left` convention the partition tuples should be ordered with the most dominant partitions in the partition tuple on the left and with the `right` convention the most dominant partition is on the right.

The *KleshchevPartitions* class will automatically convert between these four different conventions:

```
sage: KPlg = KleshchevPartitions(2, [0,0], size=2, convention='left regular')
sage: KPls = KleshchevPartitions(2, [0,0], size=2, convention='left restricted')
sage: [KPlg(mu) for mu in KPls]
[([1], [1]), ([2], [])]
```

```
>>> from sage.all import *
>>> KPlg = KleshchevPartitions(Integer(2), [Integer(0),Integer(0)],␣
↪size=Integer(2), convention='left regular')
>>> KPls = KleshchevPartitions(Integer(2), [Integer(0),Integer(0)],␣
↪size=Integer(2), convention='left restricted')
>>> [KPlg(mu) for mu in KPls]
[([1], [1]), ([2], [])]
```

EXAMPLES:

```
sage: sorted(KleshchevPartitions(5,[3,2,1],1, convention='RS'))
[([], [], [1]), ([], [1], []), ([1], [], [])]
sage: sorted(KleshchevPartitions(5, [3,2,1], 1, convention='LS'))
[([], [], [1]), ([], [1], []), ([1], [], [])]
sage: sorted(KleshchevPartitions(5, [3,2,1], 3))
[([], [], [1, 1, 1]),
 ([], [], [2, 1]),
 ([], [], [3]),
 ([], [1], [1, 1]),
 ([], [1], [2]),
 ([], [1, 1], [1]),
 ([], [2], [1]),
 ([], [3], []),
 ([1], [], [1, 1]),
 ([1], [], [2]),
 ([1], [1], [1]),
 ([1], [2], []),
 ([1, 1], [1], []),
 ([2], [], [1]),
 ([2], [1], []),
 ([3], [], [])]
sage: sorted(KleshchevPartitions(5, [3,2,1], 3, convention="left regular"))
[([], [], [1, 1, 1]),
 ([], [1], [1, 1]),
 ([], [1], [2]),
 ([], [1, 1], [1]),
```

(continues on next page)

(continued from previous page)

```
([], [1, 1, 1], []),
 ([1], [], [1, 1]),
 ([1], [1], [1]),
 ([1], [1, 1], []),
 ([1], [2], []),
 ([1, 1], [], [1]),
 ([1, 1], [1], []),
 ([1, 1, 1], [], []),
 ([2], [], [1]),
 ([2], [1], []),
 ([2, 1], [], []),
 ([3], [], [])]
```

```
>>> from sage.all import *
>>> sorted(KleshchevPartitions(Integer(5),[Integer(3),Integer(2),Integer(1)],
↪Integer(1), convention='RS'))
[([], [], [1]), ([], [1], []), ([1], [], [])]
>>> sorted(KleshchevPartitions(Integer(5), [Integer(3),Integer(2),Integer(1)],␣
↪Integer(1), convention='LS'))
[([], [], [1]), ([], [1], []), ([1], [], [])]
>>> sorted(KleshchevPartitions(Integer(5), [Integer(3),Integer(2),Integer(1)],␣
↪Integer(3)))
[([], [], [1, 1, 1]),
 ([], [], [2, 1]),
 ([], [], [3]),
 ([], [1], [1, 1]),
 ([], [1], [2]),
 ([], [1, 1], [1]),
 ([], [2], [1]),
 ([], [3], []),
 ([1], [], [1, 1]),
 ([1], [], [2]),
 ([1], [1], [1]),
 ([1], [2], []),
 ([1, 1], [1], []),
 ([2], [], [1]),
 ([2], [1], []),
 ([3], [], [])]
>>> sorted(KleshchevPartitions(Integer(5), [Integer(3),Integer(2),Integer(1)],␣
↪Integer(3), convention="left regular"))
[([], [], [1, 1, 1]),
 ([], [1], [1, 1]),
 ([], [1], [2]),
 ([], [1, 1], [1]),
 ([], [1, 1, 1], []),
 ([1], [], [1, 1]),
 ([1], [1], [1]),
 ([1], [1, 1], []),
 ([1], [2], []),
 ([1, 1], [], [1]),
 ([1, 1], [1], []),
 ([1, 1, 1], [], []),
```

(continues on next page)

(continued from previous page)

```
([2], [], [1]),
([2], [1], []),
([2, 1], [], []),
([3], [], [])]
```

REFERENCES:

- [AM2000]
- [Ariki2001]
- [BK2009]
- [Kle2009]

convention()

Return the convention of `self`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(4)
sage: KP.convention()
'restricted'
sage: KP = KleshchevPartitions(6, [4], 3, convention="right regular")
sage: KP.convention()
'regular'
sage: KP = KleshchevPartitions(5, [3,0,1], 1)
sage: KP.convention()
'left restricted'
sage: KP = KleshchevPartitions(5, [3,0,1], 1, convention='right regular')
sage: KP.convention()
'right regular'
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(4))
>>> KP.convention()
'restricted'
>>> KP = KleshchevPartitions(Integer(6), [Integer(4)], Integer(3), convention=
→"right regular")
>>> KP.convention()
'regular'
>>> KP = KleshchevPartitions(Integer(5), [Integer(3),Integer(0),Integer(1)],␣
→Integer(1))
>>> KP.convention()
'left restricted'
>>> KP = KleshchevPartitions(Integer(5), [Integer(3),Integer(0),Integer(1)],␣
→Integer(1), convention='right regular')
>>> KP.convention()
'right regular'
```

multicharge()

Return the multicharge of `self`.

EXAMPLES:

```
sage: KP = KleshchevPartitions(6, [2])
sage: KP.multicharge()
(2,)
sage: KP = KleshchevPartitions(5, [3,0,1], 1, convention='LS')
sage: KP.multicharge()
(3, 0, 1)
```

```
>>> from sage.all import *
>>> KP = KleshchevPartitions(Integer(6), [Integer(2)])
>>> KP.multicharge()
(2,)
>>> KP = KleshchevPartitions(Integer(5), [Integer(3),Integer(0),Integer(1)],␣
↪Integer(1), convention='LS')
>>> KP.multicharge()
(3, 0, 1)
```

class sage.combinat.partition_kleshchev.**KleshchevPartitions_all**(*e*, *multicharge*, *convention*)

Bases: *KleshchevPartitions*

Class of all Kleshchev partitions.

Crystal structure

We consider type $A_{e-1}^{(1)}$ crystals, and let $r = (r_i | r_i \in \mathbf{Z}/e\mathbf{Z})$ be a finite sequence of length k, which is the *level*, and $\lambda = \sum_i \Lambda_{r_i}$. We will model the highest weight $U_q(\mathfrak{g})$-crystal $B(\lambda)$ by a particular subset of partition tuples of level k.

Consider a partition tuple μ with multicharge r. We define $e_i(\mu)$ as the partition tuple obtained after the deletion of the i-`good cell` to μ and 0 if there is no i-good cell. We define $f_i(\mu)$ as the partition tuple obtained by the addition of the i-`cogood cell` to μ and 0 if there is no i-good cell.

The crystal $B(\lambda)$ is the crystal generated by the empty partition tuple. We can compute the weight of an element μ by taking $\lambda - \sum_{i=0}^{n} c_i \alpha_i$ where c_i is the number of cells of n-residue i in μ. Partition tuples in the crystal are known as *Kleshchev partitions*.

Note: We can describe normal (not restricted) Kleshchev partition tuples in $B(\lambda)$ as partition tuples μ such that $\mu^{(t)}_{r_t - r_{t+1} + x} < \mu^{(t+1)}_x$ for all $x \geq 1$ and $1 \leq t \leq k-1$.

INPUT:

- `e` – for type $A_{e-1}^{(1)}$ or 0
- `multicharge` – the multicharge sequence r
- `convention` – (default: `'LS'`) the reading convention

EXAMPLES:

We first do an example of a level 1 crystal:

```
sage: C = crystals.KleshchevPartitions(3, [0], convention="left restricted")
sage: C
Kleshchev partitions with e=3
```

(continues on next page)

(continued from previous page)

```
sage: mg = C.highest_weight_vector()
sage: mg
[]
sage: mg.f(0)
[1]
sage: mg.f(1)
sage: mg.f(2)
sage: mg.f_string([0,2,1,0])
[1, 1, 1, 1]
sage: mg.f_string([0,1,2,0])
[2, 2]
sage: GC = C.subcrystal(max_depth=5).digraph()
sage: B = crystals.LSPaths(['A',2,1], [1,0,0])
sage: GB = B.subcrystal(max_depth=5).digraph()
sage: GC.is_isomorphic(GB, edge_labels=True)
True
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0)], convention="left␣
↪restricted")
>>> C
Kleshchev partitions with e=3
>>> mg = C.highest_weight_vector()
>>> mg
[]
>>> mg.f(Integer(0))
[1]
>>> mg.f(Integer(1))
>>> mg.f(Integer(2))
>>> mg.f_string([Integer(0),Integer(2),Integer(1),Integer(0)])
[1, 1, 1, 1]
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(0)])
[2, 2]
>>> GC = C.subcrystal(max_depth=Integer(5)).digraph()
>>> B = crystals.LSPaths(['A',Integer(2),Integer(1)], [Integer(1),Integer(0),
↪Integer(0)])
>>> GB = B.subcrystal(max_depth=Integer(5)).digraph()
>>> GC.is_isomorphic(GB, edge_labels=True)
True
```

Now a higher level crystal:

```
sage: C = crystals.KleshchevPartitions(3, [0,2], convention="right restricted")
sage: mg = C.highest_weight_vector()
sage: mg
([], [])
sage: mg.f(0)
([1], [])
sage: mg.f(2)
([], [1])
sage: mg.f_string([0,1,2,0])
([2, 2], [])
```

(continues on next page)

(continued from previous page)

```
sage: mg.f_string([0,2,1,0])
([1, 1, 1, 1], [])
sage: mg.f_string([2,0,1,0])
([2], [2])
sage: GC = C.subcrystal(max_depth=5).digraph()
sage: B = crystals.LSPaths(['A',2,1], [1,0,1])
sage: GB = B.subcrystal(max_depth=5).digraph()
sage: GC.is_isomorphic(GB, edge_labels=True)
True
```

```
>>> from sage.all import *
>>> C = crystals.KleshchevPartitions(Integer(3), [Integer(0),Integer(2)],␣
↪convention="right restricted")
>>> mg = C.highest_weight_vector()
>>> mg
([], [])
>>> mg.f(Integer(0))
([1], [])
>>> mg.f(Integer(2))
([], [1])
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(0)])
([2, 2], [])
>>> mg.f_string([Integer(0),Integer(2),Integer(1),Integer(0)])
([1, 1, 1, 1], [])
>>> mg.f_string([Integer(2),Integer(0),Integer(1),Integer(0)])
([2], [2])
>>> GC = C.subcrystal(max_depth=Integer(5)).digraph()
>>> B = crystals.LSPaths(['A',Integer(2),Integer(1)], [Integer(1),Integer(0),
↪Integer(1)])
>>> GB = B.subcrystal(max_depth=Integer(5)).digraph()
>>> GC.is_isomorphic(GB, edge_labels=True)
True
```

The ordering of the residues gives a different representation of the higher level crystals (but it is still isomorphic):

```
sage: C2 = crystals.KleshchevPartitions(3, [2,0], convention="right restricted")
sage: mg2 = C2.highest_weight_vector()
sage: mg2.f_string([0,1,2,0])
([2], [2])
sage: mg2.f_string([0,2,1,0])
([1, 1, 1], [1])
sage: mg2.f_string([2,0,1,0])
([2, 1], [1])
sage: GC2 = C2.subcrystal(max_depth=5).digraph()
sage: GC.is_isomorphic(GC2, edge_labels=True)
True
```

```
>>> from sage.all import *
>>> C2 = crystals.KleshchevPartitions(Integer(3), [Integer(2),Integer(0)],␣
↪convention="right restricted")
>>> mg2 = C2.highest_weight_vector()
>>> mg2.f_string([Integer(0),Integer(1),Integer(2),Integer(0)])
```

(continues on next page)

(continued from previous page)

```
([2], [2])
>>> mg2.f_string([Integer(0),Integer(2),Integer(1),Integer(0)])
([1, 1, 1], [1])
>>> mg2.f_string([Integer(2),Integer(0),Integer(1),Integer(0)])
([2, 1], [1])
>>> GC2 = C2.subcrystal(max_depth=Integer(5)).digraph()
>>> GC.is_isomorphic(GC2, edge_labels=True)
True
```

REFERENCES:

- [Ariki1996]
- [Ariki2001]
- [Tingley2007]
- [TingleyLN]
- [Vazirani2002]

class sage.combinat.partition_kleshchev.**KleshchevPartitions_size**(*e*, *multicharge=(0,)*, *size=0*, *convention='RS'*)

Bases: *KleshchevPartitions*

Kleshchev partitions of a fixed size.

Element

alias of *KleshchevPartitionTuple*

5.1.168 Partition Shifting Algebras

This module contains families of operators that act on partitions or, more generally, integer sequences. In particular, this includes Young's raising operators R_{ij}, which act on integer sequences by adding 1 to the i-th entry and subtracting 1 to the j-th entry. A special case is acting on partitions.

AUTHORS:

- Matthew Lancellotti, George H. Seelinger (2018): Initial version

class sage.combinat.partition_shifting_algebras.**ShiftingOperatorAlgebra**(*base_ring=Univariate Polynomial Ring in t over Rational Field*, *prefix='S'*)

Bases: *CombinatorialFreeModule*

An algebra of shifting operators.

Let R be a commutative ring. The algebra of shifting operators is isomorphic as an R-algebra to the Laurent polynomial ring $R[x_1^\pm, x_2^\pm, x_3^\pm, \ldots]$. Moreover, the monomials of the shifting operator algebra act on any integer sequence $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_\ell)$ as follows. Let S be our algebra of shifting operators. Then, for any monomial $s = x_1^{a_1} x_2^{a_2} \cdots x_r^{a_r} \in S$ where $a_i \in \mathbf{Z}$ and $r \geq \ell$, we get that $s.\lambda = (\lambda_1 + a_1, \lambda_2 + a_2, \ldots, \lambda_r + a_r)$ where we pad λ with $r - \ell$ zeros. In particular, we can recover Young's raising operator, R_{ij}, for $i < j$, acting on partitions by having $\frac{x_i}{x_j}$ act on a partition λ.

One can extend the action of these shifting operators to a basis of symmetric functions, but at the expense of no longer actually having a well-defined operator. Formally, to extend the action of the shifting operators on a symmetric function basis $B = \{b_\lambda\}_\lambda$, we define an R-module homomorphism $\phi : R[x_1^\pm, x_2^\pm, \ldots] \to B$. Then we compute $x_1^{a_1} \cdots x_r^{a_r}.b_\lambda$ by first computing $(x_1^{a_1} \cdots x_r^{a_r})x_1^{\lambda_1} \cdots x_\ell^{\lambda_\ell}$ and then applying ϕ to the result. For examples of what this looks like with specific bases, see below.

This implementation is consistent with how many references work formally with raising operators. For instance, see exposition surrounding [BMPS2018] Equation (4.1).

We follow the following convention for creating elements: `S(1, 0, -1, 2)` is the shifting operator that raises the first part by 1, lowers the third part by 1, and raises the fourth part by 2.

In addition to acting on partitions (or any integer sequence), the shifting operators can also act on symmetric functions in a basis B when a conversion to B has been registered, preferably using *build_and_register_conversion()*.

For a definition of raising operators, see [BMPS2018] Definition 2.1. See *ij()* to create operators using the notation in [BMPS2018].

INPUT:

- `base_ring` – (default: `QQ['t']`) the base ring
- `prefix` – (default: `'S'`) the label for the shifting operators

EXAMPLES:

```
sage: S = ShiftingOperatorAlgebra()

sage: elm = S[1, -1, 2]; elm
S(1, -1, 2)
sage: elm([5, 4])
[([6, 3, 2], 1)]
```

```
>>> from sage.all import *
>>> S = ShiftingOperatorAlgebra()

>>> elm = S[Integer(1), -Integer(1), Integer(2)]; elm
S(1, -1, 2)
>>> elm([Integer(5), Integer(4)])
[([6, 3, 2], 1)]
```

The shifting operator monomials can act on a complete homogeneous symmetric function or a Schur function:

```
sage: s = SymmetricFunctions(QQ['t']).s()
sage: h = SymmetricFunctions(QQ['t']).h()

sage: elm(s[5, 4])
s[6, 3, 2]
sage: elm(h[5, 4])
h[6, 3, 2]

sage: S[1, -1](s[5, 4])
s[6, 3]
sage: S[1, -1](h[5, 4])
h[6, 3]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ['t']).s()
>>> h = SymmetricFunctions(QQ['t']).h()
```

(continues on next page)

(continued from previous page)

```
>>> elm(s[Integer(5), Integer(4)])
s[6, 3, 2]
>>> elm(h[Integer(5), Integer(4)])
h[6, 3, 2]

>>> S[Integer(1), -Integer(1)](s[Integer(5), Integer(4)])
s[6, 3]
>>> S[Integer(1), -Integer(1)](h[Integer(5), Integer(4)])
h[6, 3]
```

In fact, we can extend this action by linearity:

```
sage: elm = (1 - S[1,-1]) * (1 - S[4])
sage: elm == S([]) - S([1, -1]) - S([4]) + S([5, -1])
True
sage: elm(s[2, 2, 1])
s[2, 2, 1] - s[3, 1, 1] - s[6, 2, 1] + s[7, 1, 1]

sage: elm = (1 - S[1,-1]) * (1 - S[0,1,-1])
sage: elm == 1 - S[0,1,-1] - S[1,-1] + S[1,0,-1]
True
sage: elm(s[2, 2, 1])
s[2, 2, 1] - s[3, 1, 1] + s[3, 2]
```

```
>>> from sage.all import *
>>> elm = (Integer(1) - S[Integer(1),-Integer(1)]) * (Integer(1) - S[Integer(4)])
>>> elm == S([]) - S([Integer(1), -Integer(1)]) - S([Integer(4)]) + S([Integer(5),
↪ -Integer(1)])
True
>>> elm(s[Integer(2), Integer(2), Integer(1)])
s[2, 2, 1] - s[3, 1, 1] - s[6, 2, 1] + s[7, 1, 1]

>>> elm = (Integer(1) - S[Integer(1),-Integer(1)]) * (Integer(1) - S[Integer(0),
↪Integer(1),-Integer(1)])
>>> elm == Integer(1) - S[Integer(0),Integer(1),-Integer(1)] - S[Integer(1),-
↪Integer(1)] + S[Integer(1),Integer(0),-Integer(1)]
True
>>> elm(s[Integer(2), Integer(2), Integer(1)])
s[2, 2, 1] - s[3, 1, 1] + s[3, 2]
```

The algebra also comes equipped with homomorphisms to various symmetric function bases; these homomorphisms are how the action of `S` on the specific symmetric function bases is implemented:

```
sage: elm = S([3,1,2]); elm
S(3, 1, 2)
sage: h(elm)
h[3, 2, 1]
sage: s(elm)
0
```

```
>>> from sage.all import *
>>> elm = S([Integer(3),Integer(1),Integer(2)]); elm
S(3, 1, 2)
>>> h(elm)
h[3, 2, 1]
>>> s(elm)
0
```

However, not all homomorphisms are equivalent, so the action is basis dependent:

```
sage: elm = S([3,2,1]); elm
S(3, 2, 1)
sage: h(elm)
h[3, 2, 1]
sage: s(elm)
s[3, 2, 1]
sage: s(elm) == s(h(elm))
False
```

```
>>> from sage.all import *
>>> elm = S([Integer(3),Integer(2),Integer(1)]); elm
S(3, 2, 1)
>>> h(elm)
h[3, 2, 1]
>>> s(elm)
s[3, 2, 1]
>>> s(elm) == s(h(elm))
False
```

We can also use raising operators to implement the Jacobi-Trudi identity:

```
sage: op = (1-S[(1,-1)]) * (1-S[(1,0,-1)]) * (1-S[(0,1,-1)])
sage: s(op(h[3,2,1]))
s[3, 2, 1]
```

```
>>> from sage.all import *
>>> op = (Integer(1)-S[(Integer(1),-Integer(1))]) * (Integer(1)-S[(Integer(1),
↪Integer(0),-Integer(1))]) * (Integer(1)-S[(Integer(0),Integer(1),-Integer(1))])
>>> s(op(h[Integer(3),Integer(2),Integer(1)]))
s[3, 2, 1]
```

class Element

Bases: `IndexedFreeModuleElement`

An element of a *ShiftingOperatorAlgebra*.

build_and_register_conversion(*support_map*, *codomain*)

Build a module homomorphism from a map sending integer sequences to `codomain` and registers the result into Sage's conversion model.

The intended use is to define a morphism from `self` to a basis B of symmetric functions that will be used by *ShiftingOperatorAlgebra* to define the action of the operators on B.

Note

The actions on the complete homogeneous symmetric functions and on the Schur functions by morphisms are already registered.

Warning

Because `ShiftingOperatorAlgebra` inherits from `UniqueRepresentation`, once you register a conversion, this will apply to all instances of `ShiftingOperatorAlgebra` over the same base ring with the same prefix.

INPUT:

- `support_map` – a map from integer sequences to `codomain`
- `codomain` – the codomain of `support_map`, usually a basis of symmetric functions

EXAMPLES:

```
sage: S = ShiftingOperatorAlgebra(QQ)
sage: sym = SymmetricFunctions(QQ)
sage: p = sym.p()
sage: zero_map = lambda part: p.zero()
sage: S.build_and_register_conversion(zero_map, p)
sage: p(2*S([1,0,-1]) + S([2,1,0]) - 3*S([0,1,3]))
0
sage: op = S((1, -1))
sage: op(2*p[4,3] + 5*p[2,2] + 7*p[2]) == p.zero()
True
```

```
>>> from sage.all import *
>>> S = ShiftingOperatorAlgebra(QQ)
>>> sym = SymmetricFunctions(QQ)
>>> p = sym.p()
>>> zero_map = lambda part: p.zero()
>>> S.build_and_register_conversion(zero_map, p)
>>> p(Integer(2)*S([Integer(1),Integer(0),-Integer(1)]) + S([Integer(2),
↪Integer(1),Integer(0)]) - Integer(3)*S([Integer(0),Integer(1),Integer(3)]))
0
>>> op = S((Integer(1), -Integer(1)))
>>> op(Integer(2)*p[Integer(4),Integer(3)] + Integer(5)*p[Integer(2),
↪Integer(2)] + Integer(7)*p[Integer(2)]) == p.zero()
True
```

For a more illustrative example, we can implement a simple (but not mathematically justified!) conversion on the monomial basis:

```
sage: S = ShiftingOperatorAlgebra(QQ)
sage: sym = SymmetricFunctions(QQ)
sage: m = sym.m()
sage: def supp_map(gamma):
....:     gsort = sorted(gamma, reverse=True)
```

(continues on next page)

(continued from previous page)

```
....:     return m(gsort) if gsort in Partitions() else m.zero()
sage: S.build_and_register_conversion(supp_map, m)
sage: op = S.ij(0, 1)
sage: op(2*m[4,3] + 5*m[2,2] + 7*m[2]) == 2*m[5, 2] + 5*m[3, 1]
True
```

```
>>> from sage.all import *
>>> S = ShiftingOperatorAlgebra(QQ)
>>> sym = SymmetricFunctions(QQ)
>>> m = sym.m()
>>> def supp_map(gamma):
...     gsort = sorted(gamma, reverse=True)
...     return m(gsort) if gsort in Partitions() else m.zero()
>>> S.build_and_register_conversion(supp_map, m)
>>> op = S.ij(Integer(0), Integer(1))
>>> op(Integer(2)*m[Integer(4),Integer(3)] + Integer(5)*m[Integer(2),
↪Integer(2)] + Integer(7)*m[Integer(2)]) == Integer(2)*m[Integer(5),␣
↪Integer(2)] + Integer(5)*m[Integer(3), Integer(1)]
True
```

ij(*i*, *j*)

Return the raising operator R_{ij} as notated in [BMPS2018] Definition 2.1.

Shorthand element constructor that allows you to create raising operators using the familiar R_{ij} notation found in [BMPS2018] Definition 2.1, with the exception that indices here are 0-based, not 1-based.

EXAMPLES:

Create the raising operator which raises part 0 and lowers part 2 (indices are 0-based):

```
sage: R = ShiftingOperatorAlgebra()
sage: R.ij(0, 2)
S(1, 0, -1)
```

```
>>> from sage.all import *
>>> R = ShiftingOperatorAlgebra()
>>> R.ij(Integer(0), Integer(2))
S(1, 0, -1)
```

one_basis()

Return the index of the basis element for 1.

EXAMPLES:

```
sage: S = ShiftingOperatorAlgebra()
sage: S.one_basis()
()
```

```
>>> from sage.all import *
>>> S = ShiftingOperatorAlgebra()
>>> S.one_basis()
()
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by x and y.

EXAMPLES:

```
sage: S = ShiftingOperatorAlgebra()
sage: S.product_on_basis((0, 5, 2), (3, 2, -2, 5))
S(3, 7, 0, 5)
sage: S.product_on_basis((1, -2, 0, 3, -6), (-1, 2, 2))
S(0, 0, 2, 3, -6)
sage: S.product_on_basis((1, -2, -2), (-1, 2, 2))
S()
```

```
>>> from sage.all import *
>>> S = ShiftingOperatorAlgebra()
>>> S.product_on_basis((Integer(0), Integer(5), Integer(2)), (Integer(3),␣
↪Integer(2), -Integer(2), Integer(5)))
S(3, 7, 0, 5)
>>> S.product_on_basis((Integer(1), -Integer(2), Integer(0), Integer(3), -
↪Integer(6)), (-Integer(1), Integer(2), Integer(2)))
S(0, 0, 2, 3, -6)
>>> S.product_on_basis((Integer(1), -Integer(2), -Integer(2)), (-Integer(1),␣
↪Integer(2), Integer(2)))
S()
```

class sage.combinat.partition_shifting_algebras.**ShiftingSequenceSpace**

Bases: Singleton, Parent

A helper for *ShiftingOperatorAlgebra* that contains all tuples with entries in $\mathbf{Z}$ of finite support with no trailing 0's.

EXAMPLES:

```
sage: from sage.combinat.partition_shifting_algebras import ShiftingSequenceSpace
sage: S = ShiftingSequenceSpace()
sage: (1, -1) in S
True
sage: (1, -1, 0, 9) in S
True
sage: [1, -1] in S
False
sage: (0.5, 1) in S
False
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_shifting_algebras import ShiftingSequenceSpace
>>> S = ShiftingSequenceSpace()
>>> (Integer(1), -Integer(1)) in S
True
>>> (Integer(1), -Integer(1), Integer(0), Integer(9)) in S
True
>>> [Integer(1), -Integer(1)] in S
False
>>> (RealNumber('0.5'), Integer(1)) in S
False
```

check (*seq*)

Verify that `seq` is a valid shifting sequence.

If it is not, raise a `ValueError`.

EXAMPLES:

```
sage: from sage.combinat.partition_shifting_algebras import ShiftingSequenceSpace
sage: S = ShiftingSequenceSpace()
sage: S.check((1, -1))
sage: S.check((1, -1, 0, 9))
sage: S.check([1, -1])
Traceback (most recent call last):
...
ValueError: invalid index [1, -1]
sage: S.check((0.5, 1))
Traceback (most recent call last):
...
ValueError: invalid index (0.500000000000000, 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_shifting_algebras import ShiftingSequenceSpace
>>> S = ShiftingSequenceSpace()
>>> S.check((Integer(1), -Integer(1)))
>>> S.check((Integer(1), -Integer(1), Integer(0), Integer(9)))
>>> S.check([Integer(1), -Integer(1)])
Traceback (most recent call last):
...
ValueError: invalid index [1, -1]
>>> S.check((RealNumber('0.5'), Integer(1)))
Traceback (most recent call last):
...
ValueError: invalid index (0.500000000000000, 1)
```

5.1.169 Partition tuples

A *PartitionTuple* is a tuple of partitions. That is, an ordered k-tuple of partitions $\mu = (\mu^{(1)}, \mu^{(2)}, ..., \mu^{(k)})$. If

$$n = |\mu| = |\mu^{(1)}| + |\mu^{(2)}| + \cdots + |\mu^{(k)}|$$

then we say that μ is a k-partition of n.

In representation theory partition tuples arise as the natural indexing set for the ordinary irreducible representations of:

- the wreath products of cyclic groups with symmetric groups,
- the Ariki-Koike algebras, or the cyclotomic Hecke algebras of the complex reflection groups of type $G(r, 1, n)$,
- the degenerate cyclotomic Hecke algebras of type $G(r, 1, n)$.

When these algebras are not semisimple, partition tuples index an important class of modules for the algebras, which are generalisations of the Specht modules of the symmetric groups.

Tuples of partitions also index the standard basis of the higher level combinatorial Fock spaces. As a consequence, the combinatorics of partition tuples encapsulates the canonical bases of crystal graphs for the irreducible integrable highest weight modules of the (quantized) affine special linear groups and the (quantized) affine general linear groups. By the categorification theorems of Ariki, Varagnolo-Vasserot, Stroppel-Webster and others, in characteristic zero the degenerate and non-degenerate cyclotomic Hecke algebras, via their Khovanov-Lauda-Rouquier grading, categorify the canonical bases of the quantum affine special and general linear groups.

Partitions are naturally in bijection with 1-tuples of partitions. Most of the combinatorial operations defined on partitions extend to partition tuples in a meaningful way. For example, the semisimple branching rules for the Specht modules are described by adding and removing cells from partition tuples and the modular branching rules correspond to adding and removing good and cogood nodes, which is the underlying combinatorics for the associated crystal graphs.

A `PartitionTuple` belongs to `PartitionTuples` and its derived classes. `PartitionTuples` is the parent class for all partitions tuples. Four different classes of tuples of partitions are currently supported:

- `PartitionTuples(level=k,size=n)` are k-tuple of partitions of n.
- `PartitionTuples(level=k)` are k-tuple of partitions.
- `PartitionTuples(size=n)` are tuples of partitions of n.
- `PartitionTuples()` are tuples of partitions.

Note

As with `Partitions`, in sage the cells, or nodes, of partition tuples are 0-based. For example, the (lexicographically) first cell in any non-empty partition tuple is $[0, 0, 0]$.

EXAMPLES:

```
sage: PartitionTuple([[2,2],[1,1],[2]]).cells()
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 1, 0), (2, 0, 0), (2, 0,
→ 1)]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(1),Integer(1)],[Integer(2)]]).
→cells()
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 1, 0), (2, 0, 0), (2, 0,
→ 1)]
```

Note

Many `PartitionTuple` methods take the individual coordinates (k, r, c) as their arguments, here k is the component, r is the row index and c is the column index. If your coordinates are in the form `(k,r,c)` then use Python's *-operator.

EXAMPLES:

```
sage: mu=PartitionTuple([[1,1],[2],[2,1]])
sage: [ mu.arm_length(*c) for c in mu.cells()]
[0, 0, 1, 0, 1, 0, 0]
```

```
>>> from sage.all import *
>>> mu=PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
→Integer(1)]])
>>> [ mu.arm_length(*c) for c in mu.cells()]
[0, 0, 1, 0, 1, 0, 0]
```

Warning

In sage, if `mu` is a partition tuple then `mu[k]` most naturally refers to the k-th component of `mu`, so we use the convention of the (k, r, c)-th cell in a partition tuple refers to the cell in component k, row r, and column c. In the literature, the cells of a partition tuple are usually written in the form (r, c, k), where r is the row index, c is the column index, and k is the component index.

REFERENCES:

- [DJM1998]
- [BK2009]

AUTHORS:

- Andrew Mathas (2012-06-01): Initial classes.

EXAMPLES:

First is a finite enumerated set and the remaining classes are infinite enumerated sets:

```
sage: PartitionTuples().an_element()
([1, 1, 1, 1], [2, 1, 1], [3, 1], [4])
sage: PartitionTuples(4).an_element()
([], [1], [2], [3])
sage: PartitionTuples(size=5).an_element()
([1], [1], [1], [1], [1])
sage: PartitionTuples(4,5).an_element()
([1], [], [], [4])
sage: PartitionTuples(3,2)[:]                                               #␣
↪needs sage.libs.flint
[([2], [], []),
 ([1, 1], [], []),
 ([1], [1], []),
 ([1], [], [1]),
 ([], [2], []),
 ([], [1, 1], []),
 ([], [1], [1]),
 ([], [], [2]),
 ([], [], [1, 1])]
sage: PartitionTuples(2,3).list()                                           #␣
↪needs sage.libs.flint
[([3], []),
 ([2, 1], []),
 ([1, 1, 1], []),
 ([2], [1]),
 ([1, 1], [1]),
 ([1], [2]),
 ([1], [1, 1]),
 ([], [3]),
 ([], [2, 1]),
 ([], [1, 1, 1])]
```

```
>>> from sage.all import *
>>> PartitionTuples().an_element()
```

(continues on next page)

(continued from previous page)

```
([1, 1, 1, 1], [2, 1, 1], [3, 1], [4])
>>> PartitionTuples(Integer(4)).an_element()
([], [1], [2], [3])
>>> PartitionTuples(size=Integer(5)).an_element()
([1], [1], [1], [1], [1])
>>> PartitionTuples(Integer(4),Integer(5)).an_element()
([1], [], [], [4])
>>> PartitionTuples(Integer(3),Integer(2))[:]                                    ␣
↪              # needs sage.libs.flint
[([2], [], []),
 ([1, 1], [], []),
 ([1], [1], []),
 ([1], [], [1]),
 ([], [2], []),
 ([], [1, 1], []),
 ([], [1], [1]),
 ([], [], [2]),
 ([], [], [1, 1])]
>>> PartitionTuples(Integer(2),Integer(3)).list()                                 ␣
↪              # needs sage.libs.flint
[([3], []),
 ([2, 1], []),
 ([1, 1, 1], []),
 ([2], [1]),
 ([1, 1], [1]),
 ([1], [2]),
 ([1], [1, 1]),
 ([], [3]),
 ([], [2, 1]),
 ([], [1, 1, 1])]
```

One tuples of partitions are naturally in bijection with partitions and, as far as possible, partition tuples attempts to identify one tuples with partitions:

```
sage: Partition([4,3]) == PartitionTuple([[4,3]])
True
sage: Partition([4,3]) == PartitionTuple([4,3])
True
sage: PartitionTuple([4,3])
[4, 3]
sage: Partition([4,3]) in PartitionTuples()
True
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(3)]) == PartitionTuple([[Integer(4),Integer(3)]])
True
>>> Partition([Integer(4),Integer(3)]) == PartitionTuple([Integer(4),Integer(3)])
True
>>> PartitionTuple([Integer(4),Integer(3)])
[4, 3]
>>> Partition([Integer(4),Integer(3)]) in PartitionTuples()
True
```

Partition tuples come equipped with many of the corresponding methods for partitions. For example, it is possible to add and remove cells, to conjugate partition tuples, to work with their diagrams, compare partition tuples in dominance and so:

```
sage: PartitionTuple([[4,1],[],[2,2,1],[3]]).pp()
   ****   -   **   ***
   *          **
              *
sage: PartitionTuple([[4,1],[],[2,2,1],[3]]).conjugate()
([1, 1, 1], [3, 2], [], [2, 1, 1, 1])
sage: PartitionTuple([[4,1],[],[2,2,1],[3]]).conjugate().pp()
   *   ***   -   **
   *   **        *
   *             *
                 *
sage: lam = PartitionTuples(3)([[3,2],[],[1,1,1,1]]); lam
([3, 2], [], [1, 1, 1, 1])
sage: lam.level()
3
sage: lam.size()
9
sage: lam.category()
Category of elements of Partition tuples of level 3
sage: lam.parent()
Partition tuples of level 3
sage: lam[0]
[3, 2]
sage: lam[1]
[]
sage: lam[2]
[1, 1, 1, 1]
sage: lam.pp()
   ***   -   *
   **        *
             *
             *
sage: lam.removable_cells()
[(0, 0, 2), (0, 1, 1), (2, 3, 0)]
sage: lam.down_list()
[([2, 2], [], [1, 1, 1, 1]),
 ([3, 1], [], [1, 1, 1, 1]),
 ([3, 2], [], [1, 1, 1])]
sage: lam.addable_cells()
[(0, 0, 3), (0, 1, 2), (0, 2, 0), (1, 0, 0), (2, 0, 1), (2, 4, 0)]
sage: lam.up_list()
[([4, 2], [], [1, 1, 1, 1]),
 ([3, 3], [], [1, 1, 1, 1]),
 ([3, 2, 1], [], [1, 1, 1, 1]),
 ([3, 2], [1], [1, 1, 1, 1]),
 ([3, 2], [], [2, 1, 1, 1]),
 ([3, 2], [], [1, 1, 1, 1, 1])]
sage: lam.conjugate()
([4], [], [2, 2, 1])
```

(continues on next page)

(continued from previous page)

```
sage: lam.dominates( PartitionTuple([[3],[1],[2,2,1]]) )
False
sage: lam.dominates( PartitionTuple([[3],[2],[1,1,1]]))
True
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(4),Integer(1)],[],[Integer(2),Integer(2),Integer(1)],
↪[Integer(3)]]).pp()
   ****   -   **   ***
   *          **
              *
>>> PartitionTuple([[Integer(4),Integer(1)],[],[Integer(2),Integer(2),Integer(1)],
↪[Integer(3)]]).conjugate()
([1, 1, 1], [3, 2], [], [2, 1, 1, 1])
>>> PartitionTuple([[Integer(4),Integer(1)],[],[Integer(2),Integer(2),Integer(1)],
↪[Integer(3)]]).conjugate().pp()
   *   ***   -   **
   *   **        *
   *             *
                 *
>>> lam = PartitionTuples(Integer(3))([[Integer(3),Integer(2)],[],[Integer(1),
↪Integer(1),Integer(1),Integer(1)]]); lam
([3, 2], [], [1, 1, 1, 1])
>>> lam.level()
3
>>> lam.size()
9
>>> lam.category()
Category of elements of Partition tuples of level 3
>>> lam.parent()
Partition tuples of level 3
>>> lam[Integer(0)]
[3, 2]
>>> lam[Integer(1)]
[]
>>> lam[Integer(2)]
[1, 1, 1, 1]
>>> lam.pp()
   ***   -   *
   **        *
             *
             *
>>> lam.removable_cells()
[(0, 0, 2), (0, 1, 1), (2, 3, 0)]
>>> lam.down_list()
[([2, 2], [], [1, 1, 1, 1]),
 ([3, 1], [], [1, 1, 1, 1]),
 ([3, 2], [], [1, 1, 1])]
>>> lam.addable_cells()
[(0, 0, 3), (0, 1, 2), (0, 2, 0), (1, 0, 0), (2, 0, 1), (2, 4, 0)]
>>> lam.up_list()
[([4, 2], [], [1, 1, 1, 1]),
```

(continues on next page)

(continued from previous page)

```
 ([3, 3], [], [1, 1, 1, 1]),
 ([3, 2, 1], [], [1, 1, 1, 1]),
 ([3, 2], [1], [1, 1, 1, 1]),
 ([3, 2], [], [2, 1, 1, 1]),
 ([3, 2], [], [1, 1, 1, 1, 1])]
>>> lam.conjugate()
([4], [], [2, 2, 1])
>>> lam.dominates( PartitionTuple([[Integer(3)],[Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]) )
False
>>> lam.dominates( PartitionTuple([[Integer(3)],[Integer(2)],[Integer(1),Integer(1),
↪Integer(1)]]))
True
```

Every partition tuple behaves every much like a tuple of partitions:

```
sage: mu = PartitionTuple([[4,1],[],[2,2,1],[3]])
sage: [ nu for nu in mu ]
[[4, 1], [], [2, 2, 1], [3]]
sage: Set([ type(nu) for nu in mu ])
{<class 'sage.combinat.partition.Partitions_all_with_category.element_class'>}
sage: mu[2][2]
1
sage: mu[3]
[3]
sage: mu.components()
[[4, 1], [], [2, 2, 1], [3]]
sage: mu.components() == [ nu for nu in mu ]
True
sage: mu[0]
[4, 1]
sage: mu[1]
[]
sage: mu[2]
[2, 2, 1]
sage: mu[2][0]
2
sage: mu[2][1]
2
sage: mu.level()
4
sage: len(mu)
4
sage: mu.cells()
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 1, 0), (2, 0, 0), (2, 0, 1),
 (2, 1, 0), (2, 1, 1), (2, 2, 0), (3, 0, 0), (3, 0, 1), (3, 0, 2)]
sage: mu.addable_cells()
[(0, 0, 4), (0, 1, 1), (0, 2, 0), (1, 0, 0), (2, 0, 2), (2, 2, 1),
 (2, 3, 0), (3, 0, 3), (3, 1, 0)]
sage: mu.removable_cells()
[(0, 0, 3), (0, 1, 0), (2, 1, 1), (2, 2, 0), (3, 0, 2)]
```

```
>>> from sage.all import *
>>> mu = PartitionTuple([[Integer(4),Integer(1)],[],[Integer(2),Integer(2),
↪Integer(1)],[Integer(3)]])
>>> [ nu for nu in mu ]
[[4, 1], [], [2, 2, 1], [3]]
>>> Set([ type(nu) for nu in mu ])
{<class 'sage.combinat.partition.Partitions_all_with_category.element_class'>}
>>> mu[Integer(2)][Integer(2)]
1
>>> mu[Integer(3)]
[3]
>>> mu.components()
[[4, 1], [], [2, 2, 1], [3]]
>>> mu.components() == [ nu for nu in mu ]
True
>>> mu[Integer(0)]
[4, 1]
>>> mu[Integer(1)]
[]
>>> mu[Integer(2)]
[2, 2, 1]
>>> mu[Integer(2)][Integer(0)]
2
>>> mu[Integer(2)][Integer(1)]
2
>>> mu.level()
4
>>> len(mu)
4
>>> mu.cells()
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 1, 0), (2, 0, 0), (2, 0, 1),
 (2, 1, 0), (2, 1, 1), (2, 2, 0), (3, 0, 0), (3, 0, 1), (3, 0, 2)]
>>> mu.addable_cells()
[(0, 0, 4), (0, 1, 1), (0, 2, 0), (1, 0, 0), (2, 0, 2), (2, 2, 1),
 (2, 3, 0), (3, 0, 3), (3, 1, 0)]
>>> mu.removable_cells()
[(0, 0, 3), (0, 1, 0), (2, 1, 1), (2, 2, 0), (3, 0, 2)]
```

Attached to a partition tuple is the corresponding Young, or parabolic, subgroup:

```
sage: mu.young_subgroup()                                                       #␣
↪needs sage.groups
Permutation Group with generators
 [(), (12,13), (11,12), (8,9), (6,7), (3,4), (2,3), (1,2)]
sage: mu.young_subgroup_generators()                                            #␣
↪needs sage.groups
[1, 2, 3, 6, 8, 11, 12]
```

```
>>> from sage.all import *
>>> mu.young_subgroup()                                                         #␣
↪needs sage.groups
Permutation Group with generators
 [(), (12,13), (11,12), (8,9), (6,7), (3,4), (2,3), (1,2)]
```

(continues on next page)

(continued from previous page)

```
>>> mu.young_subgroup_generators()                                          #␣
↪needs sage.groups
[1, 2, 3, 6, 8, 11, 12]
```

class sage.combinat.partition_tuple.**PartitionTuple**(*parent*, *mu*)

Bases: *CombinatorialElement*

A tuple of *Partition*.

A tuple of partition comes equipped with many of methods available to partitions. The `level` of the PartitionTuple is the length of the tuple.

This is an ordered k-tuple of partitions $\mu = (\mu^{(1)}, \mu^{(2)}, ..., \mu^{(k)})$. If

$$n = |\mu| = |\mu^{(1)}| + |\mu^{(2)}| + \cdots + |\mu^{(k)}|$$

then μ is a k-partition of n.

In representation theory PartitionTuples arise as the natural indexing set for the ordinary irreducible representations of:

- the wreath products of cyclic groups with symmetric groups
- the Ariki-Koike algebras, or the cyclotomic Hecke algebras of the complex reflection groups of type $G(r, 1, n)$
- the degenerate cyclotomic Hecke algebras of type $G(r, 1, n)$

When these algebras are not semisimple, partition tuples index an important class of modules for the algebras which are generalisations of the Specht modules of the symmetric groups.

Tuples of partitions also index the standard basis of the higher level combinatorial Fock spaces. As a consequence, the combinatorics of partition tuples encapsulates the canonical bases of crystal graphs for the irreducible integrable highest weight modules of the (quantized) affine special linear groups and the (quantized) affine general linear groups. By the categorification theorems of Ariki, Varagnolo-Vasserot, Stroppel-Webster and others, in characteristic zero the degenerate and non-degenerate cyclotomic Hecke algebras, via their Khovanov-Lauda-Rouquier grading, categorify the canonical bases of the quantum affine special and general linear groups.

Partitions are naturally in bijection with 1-tuples of partitions. Most of the combinatorial operations defined on partitions extend to PartitionTuples in a meaningful way. For example, the semisimple branching rules for the Specht modules are described by adding and removing cells from partition tuples and the modular branching rules correspond to adding and removing good and cogood nodes, which is the underlying combinatorics for the associated crystal graphs.

Warning

In the literature, the cells of a partition tuple are usually written in the form (r, c, k), where r is the row index, c is the column index, and k is the component index. In sage, if `mu` is a partition tuple then `mu[k]` most naturally refers to the k-th component of `mu`, so we use the convention of the (k, r, c)-th cell in a partition tuple refers to the cell in component k, row r, and column c.

INPUT:

Anything which can reasonably be interpreted as a tuple of partitions. That is, a list or tuple of partitions or valid input to *Partition*.

EXAMPLES:

```
sage: mu=PartitionTuple( [[3,2],[2,1],[],[1,1,1,1]] ); mu
([3, 2], [2, 1], [], [1, 1, 1, 1])
sage: nu=PartitionTuple( ([3,2],[2,1],[],[1,1,1,1]) ); nu
([3, 2], [2, 1], [], [1, 1, 1, 1])
sage: mu == nu
True
sage: mu is nu
False
sage: mu in PartitionTuples()
True
sage: mu.parent()
Partition tuples

sage: lam=PartitionTuples(3)([[3,2],[],[1,1,1,1]]); lam
([3, 2], [], [1, 1, 1, 1])
sage: lam.level()
3
sage: lam.size()
9
sage: lam.category()
Category of elements of Partition tuples of level 3
sage: lam.parent()
Partition tuples of level 3
sage: lam[0]
[3, 2]
sage: lam[1]
[]
sage: lam[2]
[1, 1, 1, 1]
sage: lam.pp()
   ***   -   *
   **        *
             *
             *
sage: lam.removable_cells()
[(0, 0, 2), (0, 1, 1), (2, 3, 0)]
sage: lam.down_list()
[([2, 2], [], [1, 1, 1, 1]), ([3, 1], [], [1, 1, 1, 1]), ([3, 2], [], [1, 1, 1])]
sage: lam.addable_cells()
[(0, 0, 3), (0, 1, 2), (0, 2, 0), (1, 0, 0), (2, 0, 1), (2, 4, 0)]
sage: lam.up_list()
[([4, 2], [], [1, 1, 1, 1]), ([3, 3], [], [1, 1, 1, 1]), ([3, 2, 1], [], [1, 1, 1,
↪ 1]), ([3, 2], [1], [1, 1, 1, 1]), ([3, 2], [], [2, 1, 1, 1]), ([3, 2], [], [1,␣
↪1, 1, 1, 1])]
sage: lam.conjugate()
([4], [], [2, 2, 1])
sage: lam.dominates( PartitionTuple([[3],[1],[2,2,1]]) )
False
sage: lam.dominates( PartitionTuple([[3],[2],[1,1,1]]))
True
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> mu=PartitionTuple( [[Integer(3),Integer(2)],[Integer(2),Integer(1)],[],
↪[Integer(1),Integer(1),Integer(1),Integer(1)]] ); mu
([3, 2], [2, 1], [], [1, 1, 1, 1])
>>> nu=PartitionTuple( ([Integer(3),Integer(2)],[Integer(2),Integer(1)],[],
↪[Integer(1),Integer(1),Integer(1),Integer(1)]) ); nu
([3, 2], [2, 1], [], [1, 1, 1, 1])
>>> mu == nu
True
>>> mu is nu
False
>>> mu in PartitionTuples()
True
>>> mu.parent()
Partition tuples

>>> lam=PartitionTuples(Integer(3))([[Integer(3),Integer(2)],[],[Integer(1),
↪Integer(1),Integer(1),Integer(1)]]); lam
([3, 2], [], [1, 1, 1, 1])
>>> lam.level()
3
>>> lam.size()
9
>>> lam.category()
Category of elements of Partition tuples of level 3
>>> lam.parent()
Partition tuples of level 3
>>> lam[Integer(0)]
[3, 2]
>>> lam[Integer(1)]
[]
>>> lam[Integer(2)]
[1, 1, 1, 1]
>>> lam.pp()
   ***   -   *
   **        *
             *
             *
>>> lam.removable_cells()
[(0, 0, 2), (0, 1, 1), (2, 3, 0)]
>>> lam.down_list()
[([2, 2], [], [1, 1, 1, 1]), ([3, 1], [], [1, 1, 1, 1]), ([3, 2], [], [1, 1, 1])]
>>> lam.addable_cells()
[(0, 0, 3), (0, 1, 2), (0, 2, 0), (1, 0, 0), (2, 0, 1), (2, 4, 0)]
>>> lam.up_list()
[([4, 2], [], [1, 1, 1, 1]), ([3, 3], [], [1, 1, 1, 1]), ([3, 2, 1], [], [1, 1, 1,
↪ 1]), ([3, 2], [1], [1, 1, 1, 1]), ([3, 2], [], [2, 1, 1, 1]), ([3, 2], [], [1,␣
↪1, 1, 1, 1])]
>>> lam.conjugate()
([4], [], [2, 2, 1])
>>> lam.dominates( PartitionTuple([[Integer(3)],[Integer(1)],[Integer(2),
↪Integer(2),Integer(1)]]) )
False
```

(continues on next page)

(continued from previous page)

```
>>> lam.dominates( PartitionTuple([[Integer(3)],[Integer(2)],[Integer(1),
↪Integer(1),Integer(1)]]))
True
```

See also

- *PartitionTuples*
- *Partitions*

Element

alias of *Partition*

add_cell(*k*, *r*, *c*)

Return the partition tuple obtained by adding a cell in row r, column c, and component k.

This does not change self.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).add_cell(0,0,1)
([2, 1], [4, 3], [2, 1, 1])
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(4),Integer(3)],
↪[Integer(2),Integer(1),Integer(1)]]).add_cell(Integer(0),Integer(0),
↪Integer(1))
([2, 1], [4, 3], [2, 1, 1])
```

addable_cells()

Return a list of the removable cells of this partition tuple.

All indices are of the form (k, r, c), where r is the row-index, c is the column index and k is the component.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 2), (1, 1, 0), (2, 0, 2), (2, 1, 1), (2, 2, 0)]
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 4), (1, 1, 3), (1, 2, 0), (2, 0, 2), (2, 1, 1),
↪(2, 3, 0)]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
↪Integer(1)]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 2), (1, 1, 0), (2, 0, 2), (2, 1, 1), (2, 2, 0)]
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(4),Integer(3)],
↪[Integer(2),Integer(1),Integer(1)]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 4), (1, 1, 3), (1, 2, 0), (2, 0, 2), (2, 1, 1),
↪(2, 3, 0)]
```

arm_length(*k*, *r*, *c*)

Return the length of the arm of cell `(k, r, c)` in `self`.

INPUT:

- `k` – the component
- `r` – the row
- `c` – the cell

OUTPUT: the arm length as an integer

The arm of cell `(k, r, c)` is the number of cells in the `k`-th component which are to the right of the cell in row `r` and column `c`.

EXAMPLES:

```
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).arm_length(2,0,0)
1
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).arm_length(2,0,1)
0
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).arm_length(2,2,0)
0
```

```
>>> from sage.all import *
>>> PartitionTuple([[],[Integer(2),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)],[Integer(3)]]).arm_length(Integer(2),Integer(0),Integer(0))
1
>>> PartitionTuple([[],[Integer(2),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)],[Integer(3)]]).arm_length(Integer(2),Integer(0),Integer(1))
0
>>> PartitionTuple([[],[Integer(2),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)],[Integer(3)]]).arm_length(Integer(2),Integer(2),Integer(0))
0
```

block(*e*, *multicharge*)

Return a dictionary β that determines the block associated to the partition `self` and the *`quantum_characteristic()`* `e`.

INPUT:

- `e` – the quantum characteristic
- `multicharge` – the multicharge (default: $(0,)$)

OUTPUT:

- a dictionary giving the multiplicities of the residues in the partition tuple `self`

In more detail, the value `beta[i]` is equal to the number of nodes of residue `i`. This corresponds to the positive root

$$\sum_{i\in I}\beta_i\alpha_i\in Q^+,$$

a element of the positive root lattice of the corresponding Kac-Moody algebra. See [DJM1998] and [BK2009] for more details.

This is a useful statistics because two Specht modules for a cyclotomic Hecke algebra of type A belong to the same block if and only if they correspond to same element β of the root lattice, given above.

We return a dictionary because when the quantum characteristic is 0, the Cartan type is A_∞, in which case the simple roots are indexed by the integers.

EXAMPLES:

```
sage: PartitionTuple([[2,2],[2,2]]).block(0,(0,0))
{-1: 2, 0: 4, 1: 2}
sage: PartitionTuple([[2,2],[2,2]]).block(2,(0,0))
{0: 4, 1: 4}
sage: PartitionTuple([[2,2],[2,2]]).block(2,(0,1))
{0: 4, 1: 4}
sage: PartitionTuple([[2,2],[2,2]]).block(3,(0,2))
{0: 3, 1: 2, 2: 3}
sage: PartitionTuple([[2,2],[2,2]]).block(3,(0,2))
{0: 3, 1: 2, 2: 3}
sage: PartitionTuple([[2,2],[2,2]]).block(3,(3,2))
{0: 3, 1: 2, 2: 3}
sage: PartitionTuple([[2,2],[2,2]]).block(4,(0,0))
{0: 4, 1: 2, 3: 2}
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
↪block(Integer(0),(Integer(0),Integer(0)))
{-1: 2, 0: 4, 1: 2}
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
↪block(Integer(2),(Integer(0),Integer(0)))
{0: 4, 1: 4}
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
↪block(Integer(2),(Integer(0),Integer(1)))
{0: 4, 1: 4}
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
↪block(Integer(3),(Integer(0),Integer(2)))
{0: 3, 1: 2, 2: 3}
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
↪block(Integer(3),(Integer(0),Integer(2)))
{0: 3, 1: 2, 2: 3}
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
↪block(Integer(3),(Integer(3),Integer(2)))
{0: 3, 1: 2, 2: 3}
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
↪block(Integer(4),(Integer(0),Integer(0)))
{0: 4, 1: 2, 3: 2}
```

cells()

Return the coordinates of the cells of `self`. Coordinates are given as (component index, row index, column index) and are 0 based.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[1],[1,1,1]]).cells()
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (2, 0, 0), (2, 1, 0), (2, 2, 0)]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(1)],[Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(1)]]).cells()
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (2, 0, 0), (2, 1, 0), (2, 2, 0)]
```

components()

Return a list containing the shape of this partition.

This function exists in order to give a uniform way of iterating over the "components" of partition tuples of level 1 (partitions) and for higher levels.

EXAMPLES:

```
sage: for t in PartitionTuple([[2,1],[3,2],[3]]).components():
....:     print('%s\n' % t.ferrers_diagram())
**
*

***
**

***

sage: for t in PartitionTuple([3,2]).components():
....:     print('%s\n' % t.ferrers_diagram())
***
**
```

```
>>> from sage.all import *
>>> for t in PartitionTuple([[Integer(2),Integer(1)],[Integer(3),Integer(2)],
↪[Integer(3)]]).components():
...     print('%s\n' % t.ferrers_diagram())
**
*
<BLANKLINE>
***
**
<BLANKLINE>
***
<BLANKLINE>
>>> for t in PartitionTuple([Integer(3),Integer(2)]).components():
...     print('%s\n' % t.ferrers_diagram())
***
**
```

conjugate()

Return the conjugate partition tuple of `self`.

The conjugate partition tuple is obtained by reversing the order of the components and then swapping the rows and columns in each component.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[1],[1,1,1]]).conjugate()
([3], [1], [2, 1])
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(1)],[Integer(1),
↪Integer(1),Integer(1)]]).conjugate()
([3], [1], [2, 1])
```

contains(*mu*)

Return `True` if this partition tuple contains μ.

If $\lambda = (\lambda^{(1)}, \ldots, \lambda^{(l)})$ and $\mu = (\mu^{(1)}, \ldots, \mu^{(m)})$ are two partition tuples then λ contains μ if $m \leq l$ and $\mu^{(i)}_r \leq \lambda^{(i)}_r$ for $1 \leq i \leq m$ and $r \geq 0$.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).contains( PartitionTuple([[1,1],[2],
↪[2,1]]) )
True
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
↪Integer(1)]]).contains( PartitionTuple([[Integer(1),Integer(1)],
↪[Integer(2)],[Integer(2),Integer(1)]]) )
True
```

content(*k*, *r*, *c*, *multicharge*)

Return the content of the cell.

Let m_k = `multicharge[k]`, then the content of a cell is $m_k + c - r$.

If the `multicharge` is a list of integers then it simply offsets the values of the contents in each component. On the other hand, if the `multicharge` belongs to $\mathbf{Z}/e\mathbf{Z}$ then the corresponding e-residue is returned (that is, the content mod e).

As with the content method for partitions, the content of a cell does not technically depend on the partition tuple, but this method is included because it is often useful.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content(0,1,0, [0,0,0])
-1
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content(0,1,0, [1,0,0])
0
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content(2,1,0, [0,0,0])
-1
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2)],[Integer(1),
↪Integer(1),Integer(1)]]).content(Integer(0),Integer(1),Integer(0),␣
↪[Integer(0),Integer(0),Integer(0)])
-1
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2)],[Integer(1),
↪Integer(1),Integer(1)]]).content(Integer(0),Integer(1),Integer(0),␣
↪[Integer(1),Integer(0),Integer(0)])
0
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2)],[Integer(1),
↪Integer(1),Integer(1)]]).content(Integer(2),Integer(1),Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(0),Integer(0),Integer(0)])
-1
```

and now we return the 3-residue of a cell:

```
sage: multicharge = [IntegerModRing(3)(c) for c in [0,0,0]]
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content(0,1,0, multicharge)
2
```

```
>>> from sage.all import *
>>> multicharge = [IntegerModRing(Integer(3))(c) for c in [Integer(0),
↪Integer(0),Integer(0)]]
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2)],[Integer(1),
↪Integer(1),Integer(1)]]).content(Integer(0),Integer(1),Integer(0),
↪multicharge)
2
```

content_tableau(*multicharge*)

Return the tableau which has (k,r,c)th entry equal to the content `multicharge[k]-r+c` of this cell.

As with the content function, by setting the `multicharge` appropriately the tableau containing the residues is returned.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content_tableau([0,0,0])
([[0, 1], [-1]], [[0, 1]], [[0], [-1], [-2]])
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content_tableau([0,0,1]).pp()
    0  1     0  1     1
   -1                 0
                     -1
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2)],[Integer(1),
↪Integer(1),Integer(1)]]).content_tableau([Integer(0),Integer(0),Integer(0)])
([[0, 1], [-1]], [[0, 1]], [[0], [-1], [-2]])
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2)],[Integer(1),
↪Integer(1),Integer(1)]]).content_tableau([Integer(0),Integer(0),
↪Integer(1)]).pp()
    0  1     0  1     1
   -1                 0
                     -1
```

as with the content function the multicharge can be used to return the tableau containing the residues of the cells:

```
sage: multicharge=[ IntegerModRing(3)(c) for c in [0,0,1] ]
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content_tableau(multicharge).pp()
    0  1     0  1     1
    2                 0
                      2
```

```
>>> from sage.all import *
>>> multicharge=[ IntegerModRing(Integer(3))(c) for c in [Integer(0),
↪Integer(0),Integer(1)] ]
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2)],[Integer(1),
↪Integer(1),Integer(1)]]).content_tableau(multicharge).pp()
    0  1      0  1      1
    2                   0
                        2
```

corners()

Return a list of the removable cells of this partition tuple.

All indices are of the form `(k, r, c)`, where `r` is the row-index, `c` is the column index and `k` is the component.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).removable_cells()
[(0, 1, 0), (1, 0, 1), (2, 0, 1), (2, 1, 0)]
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).removable_cells()
[(0, 1, 0), (1, 0, 3), (1, 1, 2), (2, 0, 1), (2, 2, 0)]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
↪Integer(1)]]).removable_cells()
[(0, 1, 0), (1, 0, 1), (2, 0, 1), (2, 1, 0)]
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(4),Integer(3)],
↪[Integer(2),Integer(1),Integer(1)]]).removable_cells()
[(0, 1, 0), (1, 0, 3), (1, 1, 2), (2, 0, 1), (2, 2, 0)]
```

defect(*e*, *multicharge*)

Return the `e`-defect or the `e`-weight `self`.

The e-defect is the number of (connected) e-rim hooks that can be removed from the partition.

The defect of a partition tuple is given by

$$\text{defect}(\beta) = (\Lambda, \beta) - \tfrac{1}{2}(\beta, \beta),$$

where $\Lambda = \sum_r \Lambda_{\kappa_r}$ for the multicharge $(\kappa_1, \ldots, \kappa_\ell)$ and $\beta = \sum_{(r,c)} \alpha_{(c-r) \pmod e}$, with the sum being over the cells in the partition.

INPUT:

- `e` – the quantum characteristic
- `multicharge` – the multicharge (default: $(0,)$)

OUTPUT: a nonnegative integer, which is the defect of the block containing the partition tuple `self`

EXAMPLES:

```
sage: PartitionTuple([[2,2],[2,2]]).defect(0,(0,0))
0
sage: PartitionTuple([[2,2],[2,2]]).defect(2,(0,0))
8
sage: PartitionTuple([[2,2],[2,2]]).defect(2,(0,1))
```

(continues on next page)

(continued from previous page)

```
8
sage: PartitionTuple([[2,2],[2,2]]).defect(3,(0,2))
5
sage: PartitionTuple([[2,2],[2,2]]).defect(3,(0,2))
5
sage: PartitionTuple([[2,2],[2,2]]).defect(3,(3,2))
2
sage: PartitionTuple([[2,2],[2,2]]).defect(4,(0,0))
0
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
→defect(Integer(0),(Integer(0),Integer(0)))
0
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
→defect(Integer(2),(Integer(0),Integer(0)))
8
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
→defect(Integer(2),(Integer(0),Integer(1)))
8
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
→defect(Integer(3),(Integer(0),Integer(2)))
5
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
→defect(Integer(3),(Integer(0),Integer(2)))
5
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
→defect(Integer(3),(Integer(3),Integer(2)))
2
>>> PartitionTuple([[Integer(2),Integer(2)],[Integer(2),Integer(2)]]).
→defect(Integer(4),(Integer(0),Integer(0)))
0
```

degree(*e*)

Return the e-th degree of `self`.

The e-th degree is the sum of the degrees of the standard tableaux of shape λ. The e-th degree is the exponent of $\Phi_e(q)$ in the Gram determinant of the Specht module for a semisimple cyclotomic Hecke algebra of type A with parameter q.

For this calculation the multicharge $(\kappa_1, \ldots, \kappa_l)$ is chosen so that $\kappa_{r+1} - \kappa_r > n$, where n is the `size()` of λ as this ensures that the Hecke algebra is semisimple.

INPUT:

- `e` – integer $e > 1$

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: PartitionTuple([[2,1],[2,2]]).degree(2)
532
sage: PartitionTuple([[2,1],[2,2]]).degree(3)
259
```

(continues on next page)

(continued from previous page)

```
sage: PartitionTuple([[2,1],[2,2]]).degree(4)
196
sage: PartitionTuple([[2,1],[2,2]]).degree(5)
105
sage: PartitionTuple([[2,1],[2,2]]).degree(6)
105
sage: PartitionTuple([[2,1],[2,2]]).degree(7)
0
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).
↪degree(Integer(2))
532
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).
↪degree(Integer(3))
259
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).
↪degree(Integer(4))
196
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).
↪degree(Integer(5))
105
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).
↪degree(Integer(6))
105
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).
↪degree(Integer(7))
0
```

Therefore, the Gram determinant of $S(2,1|2,2)$ when the Hecke parameter q is "generic" is

$$q^N \Phi_2(q)^{532} \Phi_3(q)^{259} \Phi_4(q)^{196} \Phi_5(q)^{105} \Phi_6(q)^{105}$$

for some integer N. Compare with `prime_degree()`.

diagram()

Return a string for the Ferrers diagram of `self`.

EXAMPLES:

```
sage: print(PartitionTuple([[2,1],[3,2],[1,1,1]]).diagram())
   **   ***   *
   *    **    *
              *
sage: print(PartitionTuple([[3,2],[2,1],[],[1,1,1,1]]).diagram())
   ***   **   -   *
   **    *        *
                  *
                  *
sage: PartitionTuples.options(convention='french')
sage: print(PartitionTuple([[3,2],[2,1],[],[1,1,1,1]]).diagram())
                  *
                  *
```

(continues on next page)

(continued from previous page)

```
   **    *        *
   ***   **   _   *
sage: PartitionTuples.options._reset()
```

```
>>> from sage.all import *
>>> print(PartitionTuple([[Integer(2),Integer(1)],[Integer(3),Integer(2)],
↪[Integer(1),Integer(1),Integer(1)]]).diagram())
   **   ***   *
   *    **    *
              *
>>> print(PartitionTuple([[Integer(3),Integer(2)],[Integer(2),Integer(1)],[],
↪[Integer(1),Integer(1),Integer(1),Integer(1)]]).diagram())
   ***   **   _   *
   **    *        *
                  *
                  *
>>> PartitionTuples.options(convention='french')
>>> print(PartitionTuple([[Integer(3),Integer(2)],[Integer(2),Integer(1)],[],
↪[Integer(1),Integer(1),Integer(1),Integer(1)]]).diagram())
                  *
                  *
   **    *        *
   ***   **   _   *
>>> PartitionTuples.options._reset()
```

dominates(*mu*)

Return `True` if the PartitionTuple dominates or equals μ and `False` otherwise.

Given partition tuples $\mu = (\mu^{(1)}, ..., \mu^{(m)})$ and $\nu = (\nu^{(1)}, ..., \nu^{(n)})$ then μ dominates ν if

$$\sum_{k=1}^{l-1} |\mu^{(k)}| + \sum_{r\geq 1} \mu_r^{(l)} \geq \sum_{k=1}^{l-1} |\nu^{(k)}| + \sum_{r\geq 1} \nu_r^{(l)}$$

EXAMPLES:

```
sage: mu=PartitionTuple([[1,1],[2],[2,1]])
sage: nu=PartitionTuple([[1,1],[1,1],[2,1]])
sage: mu.dominates(mu)
True
sage: mu.dominates(nu)
True
sage: nu.dominates(mu)
False
sage: tau=PartitionTuple([[],[2,1],[]])
sage: tau.dominates([[2,1],[],[]])
False
sage: tau.dominates([[],[],[2,1]])
True
```

```
>>> from sage.all import *
>>> mu=PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
↪Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> nu=PartitionTuple([[Integer(1),Integer(1)],[Integer(1),Integer(1)],
↪[Integer(2),Integer(1)]])
>>> mu.dominates(mu)
True
>>> mu.dominates(nu)
True
>>> nu.dominates(mu)
False
>>> tau=PartitionTuple([[],[Integer(2),Integer(1)],[]])
>>> tau.dominates([[Integer(2),Integer(1)],[],[]])
False
>>> tau.dominates([[],[],[Integer(2),Integer(1)]])
True
```

down()

Generator (iterator) for the partition tuples that are obtained from `self` by removing a cell.

EXAMPLES:

```
sage: [mu for mu in PartitionTuple([[],[3,1],[1,1]]).down()]
[([], [2, 1], [1, 1]), ([], [3], [1, 1]), ([], [3, 1], [1])]
sage: [mu for mu in PartitionTuple([[],[],[]]).down()]
[]
```

```
>>> from sage.all import *
>>> [mu for mu in PartitionTuple([[],[Integer(3),Integer(1)],[Integer(1),
↪Integer(1)]]).down()]
[([], [2, 1], [1, 1]), ([], [3], [1, 1]), ([], [3, 1], [1])]
>>> [mu for mu in PartitionTuple([[],[],[]]).down()]
[]
```

down_list()

Return a list of the partition tuples that can be formed from `self` by removing a cell.

EXAMPLES:

```
sage: PartitionTuple([[],[3,1],[1,1]]).down_list()
[([], [2, 1], [1, 1]), ([], [3], [1, 1]), ([], [3, 1], [1])]
sage: PartitionTuple([[],[],[]]).down_list()
[]
```

```
>>> from sage.all import *
>>> PartitionTuple([[],[Integer(3),Integer(1)],[Integer(1),Integer(1)]]).down_
↪list()
[([], [2, 1], [1, 1]), ([], [3], [1, 1]), ([], [3, 1], [1])]
>>> PartitionTuple([[],[],[]]).down_list()
[]
```

ferrers_diagram()

Return a string for the Ferrers diagram of `self`.

EXAMPLES:

```
sage: print(PartitionTuple([[2,1],[3,2],[1,1,1]]).diagram())
   **   ***   *
   *    **    *
              *
sage: print(PartitionTuple([[3,2],[2,1],[],[1,1,1,1]]).diagram())
   ***   **   -   *
   **    *        *
                  *
                  *
sage: PartitionTuples.options(convention='french')
sage: print(PartitionTuple([[3,2],[2,1],[],[1,1,1,1]]).diagram())
                  *
                  *
   **    *        *
   ***   **   -   *
sage: PartitionTuples.options._reset()
```

```
>>> from sage.all import *
>>> print(PartitionTuple([[Integer(2),Integer(1)],[Integer(3),Integer(2)],
↪[Integer(1),Integer(1),Integer(1)]]).diagram())
   **   ***   *
   *    **    *
              *
>>> print(PartitionTuple([[Integer(3),Integer(2)],[Integer(2),Integer(1)],[],
↪[Integer(1),Integer(1),Integer(1),Integer(1)]]).diagram())
   ***   **   -   *
   **    *        *
                  *
                  *
>>> PartitionTuples.options(convention='french')
>>> print(PartitionTuple([[Integer(3),Integer(2)],[Integer(2),Integer(1)],[],
↪[Integer(1),Integer(1),Integer(1),Integer(1)]]).diagram())
                  *
                  *
   **    *        *
   ***   **   -   *
>>> PartitionTuples.options._reset()
```

garnir_tableau(**cell*)

Return the Garnir tableau of shape `self` corresponding to the cell `cell`.

If `cell` $= (k, a, c)$ then $(k, a + 1, c)$ must belong to the diagram of the *PartitionTuple*. If this is not the case then we return `False`.

> **Note**
>
> The function also sets `g._garnir_cell` equal to `cell` which is used by some other functions.

The Garnir tableaux play an important role in integral and non-semisimple representation theory because they determine the "straightening" rules for the Specht modules over an arbitrary ring.

The Garnir tableau are the "first" non-standard tableaux which arise when you act by simple transpositions. If (k, a, c) is a cell in the Young diagram of a partition, which is not at the bottom of its column, then the corresponding Garnir tableau has the integers $1, 2, \ldots, n$ entered in order from left to right along the rows of the diagram up to the cell $(k, a, c-1)$, then along the cells $(k, a+1, 1)$ to $(k, a+1, c)$, then (k, a, c) until the end of row a and then continuing from left to right in the remaining positions. The examples below probably make this clearer!

EXAMPLES:

```
sage: PartitionTuple([[5,3],[2,2],[4,3]]).garnir_tableau((0,0,2)).pp()
     1  2  6  7  8      9 10    13 14 15 16
     3  4  5           11 12    17 18 19
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((0,0,2)).pp()
     1  2  6  7  8     12 13    16 17 18 19
     3  4  5           14 15    20 21 22
     9 10 11
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((0,1,2)).pp()
     1  2  3  4  5     12 13    16 17 18 19
     6  7 11           14 15    20 21 22
     8  9 10
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((1,0,0)).pp()
     1  2  3  4  5     13 14    16 17 18 19
     6  7  8           12 15    20 21 22
     9 10 11
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((1,0,1)).pp()
     1  2  3  4  5     12 15    16 17 18 19
     6  7  8           13 14    20 21 22
     9 10 11
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((2,0,1)).pp()
     1  2  3  4  5     12 13    16 19 20 21
     6  7  8           14 15    17 18 22
     9 10 11
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((2,1,1)).pp()
Traceback (most recent call last):
...
ValueError: (comp, row+1, col) must be inside the diagram
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(5),Integer(3)],[Integer(2),Integer(2)],
↪[Integer(4),Integer(3)]]).garnir_tableau((Integer(0),Integer(0),
↪Integer(2))).pp()
     1  2  6  7  8      9 10    13 14 15 16
     3  4  5           11 12    17 18 19
>>> PartitionTuple([[Integer(5),Integer(3),Integer(3)],[Integer(2),
↪Integer(2)],[Integer(4),Integer(3)]]).garnir_tableau((Integer(0),Integer(0),
↪Integer(2))).pp()
     1  2  6  7  8     12 13    16 17 18 19
     3  4  5           14 15    20 21 22
     9 10 11
>>> PartitionTuple([[Integer(5),Integer(3),Integer(3)],[Integer(2),
↪Integer(2)],[Integer(4),Integer(3)]]).garnir_tableau((Integer(0),Integer(1),
↪Integer(2))).pp()
     1  2  3  4  5     12 13    16 17 18 19
     6  7 11           14 15    20 21 22
     8  9 10
>>> PartitionTuple([[Integer(5),Integer(3),Integer(3)],[Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)],[Integer(4),Integer(3)]]).garnir_tableau((Integer(1),Integer(0),
↪Integer(0))).pp()
     1  2  3  4  5     13 14     16 17 18 19
     6  7  8           12 15     20 21 22
     9 10 11
>>> PartitionTuple([[Integer(5),Integer(3),Integer(3)],[Integer(2),
↪Integer(2)],[Integer(4),Integer(3)]]).garnir_tableau((Integer(1),Integer(0),
↪Integer(1))).pp()
     1  2  3  4  5     12 15     16 17 18 19
     6  7  8           13 14     20 21 22
     9 10 11
>>> PartitionTuple([[Integer(5),Integer(3),Integer(3)],[Integer(2),
↪Integer(2)],[Integer(4),Integer(3)]]).garnir_tableau((Integer(2),Integer(0),
↪Integer(1))).pp()
     1  2  3  4  5     12 13     16 19 20 21
     6  7  8           14 15     17 18 22
     9 10 11
>>> PartitionTuple([[Integer(5),Integer(3),Integer(3)],[Integer(2),
↪Integer(2)],[Integer(4),Integer(3)]]).garnir_tableau((Integer(2),Integer(1),
↪Integer(1))).pp()
Traceback (most recent call last):
...
ValueError: (comp, row+1, col) must be inside the diagram
```

See also

- *top_garnir_tableau()*

hook_length(*k*, *r*, *c*)

Return the length of the hook of cell `(k, r, c)` in the partition.

The hook of cell `(k, r, c)` is defined as the cells to the right or below (in the English convention). If your coordinates are in the form `(k,r,c)`, use Python's *-operator.

EXAMPLES:

```
sage: mu=PartitionTuple([[1,1],[2],[2,1]])
sage: [ mu.hook_length(*c) for c in mu.cells()]
[2, 1, 2, 1, 3, 1, 1]
```

```
>>> from sage.all import *
>>> mu=PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
↪Integer(1)]])
>>> [ mu.hook_length(*c) for c in mu.cells()]
[2, 1, 2, 1, 3, 1, 1]
```

initial_column_tableau()

Return the initial column tableau of shape `self`.

The initial column tableau of shape λ is the standard tableau that has the numbers 1 to n, where n is the *size()* of λ, entered in order from top to bottom, and then left to right, down the columns of each component, starting from the rightmost component and working to the left.

EXAMPLES:

```
sage: PartitionTuple([ [3,1],[3,2] ]).initial_column_tableau()
([[6, 8, 9], [7]], [[1, 3, 5], [2, 4]])
```

```
>>> from sage.all import *
>>> PartitionTuple([ [Integer(3),Integer(1)],[Integer(3),Integer(2)] ]).
↪initial_column_tableau()
([[6, 8, 9], [7]], [[1, 3, 5], [2, 4]])
```

initial_tableau()

Return the *StandardTableauTuple* which has the numbers $1, 2, \ldots, n$, where n is the *size()* of `self`, entered in order from left to right along the rows of each component, where the components are ordered from left to right.

EXAMPLES:

```
sage: PartitionTuple([ [2,1],[3,2] ]).initial_tableau()
([[1, 2], [3]], [[4, 5, 6], [7, 8]])
```

```
>>> from sage.all import *
>>> PartitionTuple([ [Integer(2),Integer(1)],[Integer(3),Integer(2)] ]).
↪initial_tableau()
([[1, 2], [3]], [[4, 5, 6], [7, 8]])
```

leg_length(k, r, c)

Return the length of the leg of cell `(k, r, c)` in `self`.

INPUT:

- `k` – the component
- `r` – the row
- `c` – the cell

OUTPUT: the leg length as an integer

The leg of cell `(k, r, c)` is the number of cells in the `k`-th component which are below the node in row `r` and column `c`.

EXAMPLES:

```
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).leg_length(2,0,0)
2
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).leg_length(2,0,1)
1
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).leg_length(2,2,0)
0
```

```
>>> from sage.all import *
>>> PartitionTuple([[],[Integer(2),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)],[Integer(3)]]).leg_length(Integer(2),Integer(0),Integer(0))
2
>>> PartitionTuple([[],[Integer(2),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)],[Integer(3)]]).leg_length(Integer(2),Integer(0),Integer(1))
```

(continues on next page)

(continued from previous page)

```
1
>>> PartitionTuple([[],[Integer(2),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)],[Integer(3)]]).leg_length(Integer(2),Integer(2),Integer(0))
0
```

level()

Return the level of this partition tuple.

The level is the length of the tuple.

EXAMPLES:

```
sage: PartitionTuple([[2,1,1,0],[2,1]]).level()
2
sage: PartitionTuple([[],[],[2,1,1]]).level()
3
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1),Integer(1),Integer(0)],[Integer(2),
↪Integer(1)]]).level()
2
>>> PartitionTuple([[],[],[Integer(2),Integer(1),Integer(1)]]).level()
3
```

outside_corners()

Return a list of the removable cells of this partition tuple.

All indices are of the form `(k, r, c)`, where `r` is the row-index, `c` is the column index and `k` is the component.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 2), (1, 1, 0), (2, 0, 2), (2, 1, 1), (2, 2, 0)]
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 4), (1, 1, 3), (1, 2, 0), (2, 0, 2), (2, 1, 1),
↪(2, 3, 0)]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
↪Integer(1)]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 2), (1, 1, 0), (2, 0, 2), (2, 1, 1), (2, 2, 0)]
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(4),Integer(3)],
↪[Integer(2),Integer(1),Integer(1)]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 4), (1, 1, 3), (1, 2, 0), (2, 0, 2), (2, 1, 1),
↪(2, 3, 0)]
```

pp()

Pretty print this partition tuple. See `diagram()`.

EXAMPLES:

```
sage: PartitionTuple([[5,5,2,1],[3,2]]).pp()
*****   ***
```

(continues on next page)

(continued from previous page)

```
*****   **
**
*
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(5),Integer(5),Integer(2),Integer(1)],[Integer(3),
↪Integer(2)]]).pp()
*****   ***
*****   **
**
*
```

prime_degree(*p*)

Return the p-th prime degree of self.

The degree of a partition λ is the sum of the e-degrees` of the standard tableaux of shape λ (see *degree()*), for e a power of the prime p. The prime degree gives the exponent of p in the Gram determinant of the integral Specht module of the symmetric group.

The p-th degree is the sum of the degrees of the standard tableaux of shape λ. The p-th degree is the exponent of p in the Gram determinant of a semisimple cyclotomic Hecke algebra of type A with parameter $q = 1$.

As with *degree()*, for this calculation the multicharge $(\kappa_1, \ldots, \kappa_l)$ is chosen so that $\kappa_{r+1} - \kappa_r > n$, where n is the *size()* of λ as this ensures that the Hecke algebra is semisimple.

INPUT:

- e – an integer $e > 1$
- multicharge – an l-tuple of integers, where l is the *level()* of self

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: PartitionTuple([[2,1],[2,2]]).prime_degree(2)
728
sage: PartitionTuple([[2,1],[2,2]]).prime_degree(3)
259
sage: PartitionTuple([[2,1],[2,2]]).prime_degree(5)
105
sage: PartitionTuple([[2,1],[2,2]]).prime_degree(7)
0
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).prime_
↪degree(Integer(2))
728
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).prime_
↪degree(Integer(3))
259
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).prime_
↪degree(Integer(5))
105
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(2),Integer(2)]]).prime_
```

(continues on next page)

(continued from previous page)

```
↪degree(Integer(7))
0
```

Therefore, the Gram determinant of $S(2,1|2,2)$ when $q = 1$ is $2^{728}3^{259}5^{105}$. Compare with `degree()`.

removable_cells()

Return a list of the removable cells of this partition tuple.

All indices are of the form `(k, r, c)`, where `r` is the row-index, `c` is the column index and `k` is the component.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).removable_cells()
[(0, 1, 0), (1, 0, 1), (2, 0, 1), (2, 1, 0)]
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).removable_cells()
[(0, 1, 0), (1, 0, 3), (1, 1, 2), (2, 0, 1), (2, 2, 0)]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
↪Integer(1)]]).removable_cells()
[(0, 1, 0), (1, 0, 1), (2, 0, 1), (2, 1, 0)]
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(4),Integer(3)],
↪[Integer(2),Integer(1),Integer(1)]]).removable_cells()
[(0, 1, 0), (1, 0, 3), (1, 1, 2), (2, 0, 1), (2, 2, 0)]
```

remove_cell(k, r, c)

Return the partition tuple obtained by removing a cell in row `r`, column `c`, and component `k`.

This does not change `self`.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).remove_cell(0,1,0)
([1], [4, 3], [2, 1, 1])
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(4),Integer(3)],
↪[Integer(2),Integer(1),Integer(1)]]).remove_cell(Integer(0),Integer(1),
↪Integer(0))
([1], [4, 3], [2, 1, 1])
```

row_standard_tableaux()

Return the *row standard tableau tuples* of shape `self`.

EXAMPLES:

```
sage: PartitionTuple([[],[3,2,2,1],[2,2,1],[3]]).row_standard_tableaux()
Row standard tableau tuples of shape ([], [3, 2, 2, 1], [2, 2, 1], [3])
```

```
>>> from sage.all import *
>>> PartitionTuple([[],[Integer(3),Integer(2),Integer(2),Integer(1)],
↪[Integer(2),Integer(2),Integer(1)],[Integer(3)]]).row_standard_tableaux()
Row standard tableau tuples of shape ([], [3, 2, 2, 1], [2, 2, 1], [3])
```

size()

Return the size of a partition tuple.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[],[2,2]]).size()
7
sage: PartitionTuple([[],[],[1],[3,2,1]]).size()
7
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[],[Integer(2),Integer(2)]]).
↪size()
7
>>> PartitionTuple([[],[],[Integer(1)],[Integer(3),Integer(2),Integer(1)]]).
↪size()
7
```

standard_tableaux()

Return the *standard tableau tuples* of shape `self`.

EXAMPLES:

```
sage: PartitionTuple([[],[3,2,2,1],[2,2,1],[3]]).standard_tableaux()
Standard tableau tuples of shape ([], [3, 2, 2, 1], [2, 2, 1], [3])
```

```
>>> from sage.all import *
>>> PartitionTuple([[],[Integer(3),Integer(2),Integer(2),Integer(1)],
↪[Integer(2),Integer(2),Integer(1)],[Integer(3)]]).standard_tableaux()
Standard tableau tuples of shape ([], [3, 2, 2, 1], [2, 2, 1], [3])
```

to_exp(*k=0*)

Return a tuple of the multiplicities of the parts of a partition.

Use the optional parameter `k` to get a return list of length at least `k`.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).to_exp()
([2], [0, 1], [1, 1])
sage: PartitionTuple([[1,1],[2,2,2,2],[2,1]]).to_exp()
([2], [0, 4], [1, 1])
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(2)],[Integer(2),
↪Integer(1)]]).to_exp()
([2], [0, 1], [1, 1])
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(2),Integer(2),Integer(2),
↪Integer(2)],[Integer(2),Integer(1)]]).to_exp()
([2], [0, 4], [1, 1])
```

to_list()

Return `self` as a list of lists.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).to_list()
[[1, 1], [4, 3], [2, 1, 1]]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(1),Integer(1)],[Integer(4),Integer(3)],
↪[Integer(2),Integer(1),Integer(1)]]).to_list()
[[1, 1], [4, 3], [2, 1, 1]]
```

top_garnir_tableau(*e*, *cell*)

Return the most dominant *standard* tableau which dominates the corresponding Garnir tableau and has the same residue that has shape `self` and is determined by `e` and `cell`.

The Garnir tableau play an important role in integral and non-semisimple representation theory because they determine the "straightening" rules for the Specht modules over an arbitrary ring. The *top Garnir tableaux* arise in the graded representation theory of the symmetric groups and higher level Hecke algebras. They were introduced in [KMR2012].

If the Garnir node is `cell=(k,r,c)` and m and M are the entries in the cells `(k,r,c)` and `(k,r+1,c)`, respectively, in the initial tableau then the top `e`-Garnir tableau is obtained by inserting the numbers $m, m+1, \ldots, M$ in order from left to right first in the cells in row `r+1` which are not in the `e`-Garnir belt, then in the cell in rows `r` and `r+1` which are in the Garnir belt and then, finally, in the remaining cells in row `r` which are not in the Garnir belt. All other entries in the tableau remain unchanged.

If `e = 0`, or if there are no `e`-bricks in either row `r` or `r+1`, then the top Garnir tableau is the corresponding Garnir tableau.

EXAMPLES:

```
sage: PartitionTuple([[3,3,2],[5,4,3,2]]).top_garnir_tableau(2,(1,0,2)).pp()
    1  2  3      9 10 12 13 16
    4  5  6     11 14 15 17
    7  8        18 19 20
                21 22
sage: PartitionTuple([[3,3,2],[5,4,3,2]]).top_garnir_tableau(2,(1,0,1)).pp()
    1  2  3      9 10 11 12 13
    4  5  6     14 15 16 17
    7  8        18 19 20
                21 22
sage: PartitionTuple([[3,3,2],[5,4,3,2]]).top_garnir_tableau(3,(1,0,1)).pp()
    1  2  3      9 12 13 14 15
    4  5  6     10 11 16 17
    7  8        18 19 20
                21 22

sage: PartitionTuple([[3,3,2],[5,4,3,2]]).top_garnir_tableau(3,(3,0,1)).pp()
Traceback (most recent call last):
...
ValueError: (comp, row+1, col) must be inside the diagram
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(3),Integer(3),Integer(2)],[Integer(5),Integer(4),
↪Integer(3),Integer(2)]]).top_garnir_tableau(Integer(2),(Integer(1),
↪Integer(0),Integer(2))).pp()
    1  2  3      9 10 12 13 16
```

(continues on next page)

(continued from previous page)

```
    4  5  6     11 14 15 17
    7  8        18 19 20
                21 22
>>> PartitionTuple([[Integer(3),Integer(3),Integer(2)],[Integer(5),Integer(4),
↪Integer(3),Integer(2)]]).top_garnir_tableau(Integer(2),(Integer(1),
↪Integer(0),Integer(1))).pp()
    1  2  3      9 10 11 12 13
    4  5  6     14 15 16 17
    7  8        18 19 20
                21 22
>>> PartitionTuple([[Integer(3),Integer(3),Integer(2)],[Integer(5),Integer(4),
↪Integer(3),Integer(2)]]).top_garnir_tableau(Integer(3),(Integer(1),
↪Integer(0),Integer(1))).pp()
    1  2  3      9 12 13 14 15
    4  5  6     10 11 16 17
    7  8        18 19 20
                21 22

>>> PartitionTuple([[Integer(3),Integer(3),Integer(2)],[Integer(5),Integer(4),
↪Integer(3),Integer(2)]]).top_garnir_tableau(Integer(3),(Integer(3),
↪Integer(0),Integer(1))).pp()
Traceback (most recent call last):
...
ValueError: (comp, row+1, col) must be inside the diagram
```

See also:

- `garnir_tableau()`

up()

Generator (iterator) for the partition tuples that are obtained from `self` by adding a cell.

EXAMPLES:

```
sage: [mu for mu in PartitionTuple([[],[3,1],[1,1]]).up()]
[([1], [3, 1], [1, 1]), ([], [4, 1], [1, 1]), ([], [3, 2], [1, 1]), ([], [3,␣
↪1, 1], [1, 1]), ([], [3, 1], [2, 1]), ([], [3, 1], [1, 1, 1])]
sage: [mu for mu in PartitionTuple([[],[],[],[]]).up()]
[([1], [], [], []), ([], [1], [], []), ([], [], [1], []), ([], [], [], [1])]
```

```
>>> from sage.all import *
>>> [mu for mu in PartitionTuple([[],[Integer(3),Integer(1)],[Integer(1),
↪Integer(1)]]).up()]
[([1], [3, 1], [1, 1]), ([], [4, 1], [1, 1]), ([], [3, 2], [1, 1]), ([], [3,␣
↪1, 1], [1, 1]), ([], [3, 1], [2, 1]), ([], [3, 1], [1, 1, 1])]
>>> [mu for mu in PartitionTuple([[],[],[],[]]).up()]
[([1], [], [], []), ([], [1], [], []), ([], [], [1], []), ([], [], [], [1])]
```

up_list()

Return a list of the partition tuples that can be formed from `self` by adding a cell.

EXAMPLES:

```
sage: PartitionTuple([[],[3,1],[1,1]]).up_list()
[([1], [3, 1], [1, 1]), ([], [4, 1], [1, 1]), ([], [3, 2], [1, 1]), ([], [3,␣
↪1, 1], [1, 1]), ([], [3, 1], [2, 1]), ([], [3, 1], [1, 1, 1])]
sage: PartitionTuple([[],[],[],[]]).up_list()
[([1], [], [], []), ([], [1], [], []), ([], [], [1], []), ([], [], [], [1])]
```

```
>>> from sage.all import *
>>> PartitionTuple([[],[Integer(3),Integer(1)],[Integer(1),Integer(1)]]).up_
↪list()
[([1], [3, 1], [1, 1]), ([], [4, 1], [1, 1]), ([], [3, 2], [1, 1]), ([], [3,␣
↪1, 1], [1, 1]), ([], [3, 1], [2, 1]), ([], [3, 1], [1, 1, 1])]
>>> PartitionTuple([[],[],[],[]]).up_list()
[([1], [], [], []), ([], [1], [], []), ([], [], [1], []), ([], [], [], [1])]
```

young_subgroup()

Return the corresponding Young, or parabolic, subgroup of the symmetric group.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[4,2],[1]]).young_subgroup()                     #␣
↪needs sage.groups
Permutation Group with generators [(), (8,9), (6,7), (5,6), (4,5), (1,2)]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(4),Integer(2)],
↪[Integer(1)]]).young_subgroup()                 # needs sage.groups
Permutation Group with generators [(), (8,9), (6,7), (5,6), (4,5), (1,2)]
```

young_subgroup_generators()

Return an indexing set for the generators of the corresponding Young subgroup.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[4,2],[1]]).young_subgroup_generators()
[1, 4, 5, 6, 8]
```

```
>>> from sage.all import *
>>> PartitionTuple([[Integer(2),Integer(1)],[Integer(4),Integer(2)],
↪[Integer(1)]]).young_subgroup_generators()
[1, 4, 5, 6, 8]
```

class sage.combinat.partition_tuple.**PartitionTuples**

Bases: UniqueRepresentation, Parent

Class of all partition tuples.

For more information about partition tuples, see *PartitionTuple*.

This is a factory class which returns the appropriate parent based on the values of `level`, `size`, and `regular`

INPUT:

- `level` – the length of the tuple
- `size` – the total number of cells

- `regular` – positive integer or a tuple of nonnegative integers; if an integer, the highest multiplicity an entry may have in a component plus 1

If a level k is specified and `regular` is a tuple of integers $\ell_1, \ldots, \ell_k$, then this specifies partition tuples μ such that μ_i is ℓ_i-regular, where 0 here represents ∞-regular partitions (equivalently, partitions without restrictions). If `regular` is an integer ℓ, then we set $\ell_i = \ell$ for all i.

Element

alias of *PartitionTuple*

level()

Return the level or `None` if it is not defined.

EXAMPLES:

```
sage: PartitionTuples().level() is None
True
sage: PartitionTuples(7).level()
7
```

```
>>> from sage.all import *
>>> PartitionTuples().level() is None
True
>>> PartitionTuples(Integer(7)).level()
7
```

options = Current options for Partitions - convention: English - diagram_str: * - display: list - latex: young_diagram - latex_diagram_str: \ast

size()

Return the size or `None` if it is not defined.

EXAMPLES:

```
sage: PartitionTuples().size() is None
True
sage: PartitionTuples(size=7).size()
7
```

```
>>> from sage.all import *
>>> PartitionTuples().size() is None
True
>>> PartitionTuples(size=Integer(7)).size()
7
```

class sage.combinat.partition_tuple.**PartitionTuples_all**

Bases: *PartitionTuples*

Class of partition tuples of a arbitrary level and arbitrary sum.

class sage.combinat.partition_tuple.**PartitionTuples_level**(*level*, *category=None*)

Bases: *PartitionTuples*

Class of partition tuples of a fixed level, but summing to an arbitrary integer.

class sage.combinat.partition_tuple.**PartitionTuples_level_size**(*level*, *size*)

Bases: *PartitionTuples*

Class of partition tuples with a fixed level and a fixed size.

cardinality()

Return the number of `level`-tuples of partitions of size `n`.

Wraps a pari function call using pari:eta.

EXAMPLES:

```
sage: PartitionTuples(2,3).cardinality()                                # needs sage.libs.pari
10
sage: PartitionTuples(2,8).cardinality()                                # needs sage.libs.pari
185
```

```
>>> from sage.all import *
>>> PartitionTuples(Integer(2),Integer(3)).cardinality()                # needs sage.libs.pari
10
>>> PartitionTuples(Integer(2),Integer(8)).cardinality()                # needs sage.libs.pari
185
```

class sage.combinat.partition_tuple.**PartitionTuples_size**(*size*)

Bases: *PartitionTuples*

Class of partition tuples of a fixed size, but arbitrary level.

class sage.combinat.partition_tuple.**RegularPartitionTuples**(*regular*, ***kwds*)

Bases: *PartitionTuples*

Abstract base class for ℓ-regular partition tuples.

class sage.combinat.partition_tuple.**RegularPartitionTuples_all**(*regular*)

Bases: *RegularPartitionTuples*

Class of ℓ-regular partition tuples.

class sage.combinat.partition_tuple.**RegularPartitionTuples_level**(*level*, *regular*)

Bases: *PartitionTuples_level*

Regular Partition tuples of a fixed level.

INPUT:

- `level` – nonnegative integer; the level
- `regular` – positive integer or a tuple of nonnegative integers; if an integer, the highest multiplicity an entry may have in a component plus 1 with 0 representing ∞-regular (equivalently, partitions without restrictions)

`regular` is a tuple of integers $(\ell_1, \ldots, \ell_k)$ that specifies partition tuples μ such that μ_i is ℓ_i-regular. If `regular` is an integer ℓ, then we set $\ell_i = \ell$ for all i.

EXAMPLES:

```
sage: RPT = PartitionTuples(level=4, regular=(2,3,0,2))
sage: RPT[:24]                                                                  #␣
↪needs sage.libs.flint
[([], [], [], []),
 ([1], [], [], []),
 ([], [1], [], []),
 ([], [], [1], []),
 ([], [], [], [1]),
 ([2], [], [], []),
 ([1], [1], [], []),
 ([1], [], [1], []),
 ([1], [], [], [1]),
 ([], [2], [], []),
 ([], [1, 1], [], []),
 ([], [1], [1], []),
 ([], [1], [], [1]),
 ([], [], [2], []),
 ([], [], [1, 1], []),
 ([], [], [1], [1]),
 ([], [], [], [2]),
 ([3], [], [], []),
 ([2, 1], [], [], []),
 ([2], [1], [], []),
 ([2], [], [1], []),
 ([2], [], [], [1]),
 ([1], [2], [], []),
 ([1], [1, 1], [], [])]
sage: [[1,1],[3],[5,5,5],[7,2]] in RPT
False
sage: [[3,1],[3],[5,5,5],[7,2]] in RPT
True
sage: [[3,1],[3],[5,5,5]] in RPT
False
```

```
>>> from sage.all import *
>>> RPT = PartitionTuples(level=Integer(4), regular=(Integer(2),Integer(3),
↪Integer(0),Integer(2)))
>>> RPT[:Integer(24)]                                                           ␣
↪     # needs sage.libs.flint
[([], [], [], []),
 ([1], [], [], []),
 ([], [1], [], []),
 ([], [], [1], []),
 ([], [], [], [1]),
 ([2], [], [], []),
 ([1], [1], [], []),
 ([1], [], [1], []),
 ([1], [], [], [1]),
 ([], [2], [], []),
 ([], [1, 1], [], []),
 ([], [1], [1], []),
 ([], [1], [], [1]),
```

(continues on next page)

(continued from previous page)

```
 ([], [], [2], []),
 ([], [], [1, 1], []),
 ([], [], [1], [1]),
 ([], [], [], [2]),
 ([3], [], [], []),
 ([2, 1], [], [], []),
 ([2], [1], [], []),
 ([2], [], [1], []),
 ([2], [], [], [1]),
 ([1], [2], [], []),
 ([1], [1, 1], [], [])]
>>> [[Integer(1),Integer(1)],[Integer(3)],[Integer(5),Integer(5),Integer(5)],
↪[Integer(7),Integer(2)]] in RPT
False
>>> [[Integer(3),Integer(1)],[Integer(3)],[Integer(5),Integer(5),Integer(5)],
↪[Integer(7),Integer(2)]] in RPT
True
>>> [[Integer(3),Integer(1)],[Integer(3)],[Integer(5),Integer(5),Integer(5)]] in␣
↪RPT
False
```

class sage.combinat.partition_tuple.**RegularPartitionTuples_level_size**(*level*, *size*, *regular*)

Bases: *PartitionTuples_level_size*

Class of ℓ-regular partition tuples with a fixed level and a fixed size.

INPUT:

- `level` – nonnegative integer; the level
- `size` – nonnegative integer; the size
- `regular` – positive integer or a tuple of nonnegative integers; if an integer, the highest multiplicity an entry may have in a component plus 1 with 0 representing ∞-regular (equivalently, partitions without restrictions)

`regular` is a tuple of integers $(\ell_1, \ldots, \ell_k)$ that specifies partition tuples μ such that μ_i is ℓ_i-regular. If `regular` is an integer ℓ, then we set $\ell_i = \ell$ for all i.

EXAMPLES:

```
sage: PartitionTuples(level=3, size=7, regular=(2,1,3))[0:24]                 #␣
↪needs sage.libs.flint
[([7], [], []),
 ([6, 1], [], []),
 ([5, 2], [], []),
 ([4, 3], [], []),
 ([4, 2, 1], [], []),
 ([6], [], [1]),
 ([5, 1], [], [1]),
 ([4, 2], [], [1]),
 ([3, 2, 1], [], [1]),
 ([5], [], [2]),
 ([5], [], [1, 1]),
 ([4, 1], [], [2]),
 ([4, 1], [], [1, 1]),
```

(continues on next page)

(continued from previous page)

```
 ([3, 2], [], [2]),
 ([3, 2], [], [1, 1]),
 ([4], [], [3]),
 ([4], [], [2, 1]),
 ([3, 1], [], [3]),
 ([3, 1], [], [2, 1]),
 ([3], [], [4]),
 ([3], [], [3, 1]),
 ([3], [], [2, 2]),
 ([3], [], [2, 1, 1]),
 ([2, 1], [], [4])]
```

```
>>> from sage.all import *
>>> PartitionTuples(level=Integer(3), size=Integer(7), regular=(Integer(2),
↪Integer(1),Integer(3)))[Integer(0):Integer(24)]                # needs sage.
↪libs.flint
[([7], [], []),
 ([6, 1], [], []),
 ([5, 2], [], []),
 ([4, 3], [], []),
 ([4, 2, 1], [], []),
 ([6], [], [1]),
 ([5, 1], [], [1]),
 ([4, 2], [], [1]),
 ([3, 2, 1], [], [1]),
 ([5], [], [2]),
 ([5], [], [1, 1]),
 ([4, 1], [], [2]),
 ([4, 1], [], [1, 1]),
 ([3, 2], [], [2]),
 ([3, 2], [], [1, 1]),
 ([4], [], [3]),
 ([4], [], [2, 1]),
 ([3, 1], [], [3]),
 ([3, 1], [], [2, 1]),
 ([3], [], [4]),
 ([3], [], [3, 1]),
 ([3], [], [2, 2]),
 ([3], [], [2, 1, 1]),
 ([2, 1], [], [4])]
```

class sage.combinat.partition_tuple.**RegularPartitionTuples_size**(*size*, *regular*)

Bases: *RegularPartitionTuples*

Class of ℓ-regular partition tuples with a fixed size.

5.1.170 Iterators over the partitions of an integer

The iterators generate partitions in either increasing or decreasing lexicographic orders and a partition P is represented either in ascending ($P_i \leq P_{i+1}$) or descending ($P_i \geq P_{i+1}$) orders:

```
sage: from sage.combinat.partitions import ZS1_iterator
sage: for p in ZS1_iterator(4):
....:     print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
sage: from sage.combinat.partitions import AccelDesc_iterator
sage: for p in AccelDesc_iterator(4):
....:     print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
sage: from sage.combinat.partitions import ZS2_iterator
sage: for p in ZS2_iterator(4):
....:     print(p)
[1, 1, 1, 1]
[2, 1, 1]
[2, 2]
[3, 1]
[4]
sage: from sage.combinat.partitions import AccelAsc_iterator
sage: for p in AccelAsc_iterator(4):
....:     print(p)
[1, 1, 1, 1]
[1, 1, 2]
[1, 3]
[2, 2]
[4]
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS1_iterator
>>> for p in ZS1_iterator(Integer(4)):
...     print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
>>> from sage.combinat.partitions import AccelDesc_iterator
>>> for p in AccelDesc_iterator(Integer(4)):
...     print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
>>> from sage.combinat.partitions import ZS2_iterator
>>> for p in ZS2_iterator(Integer(4)):
```

(continues on next page)

(continued from previous page)

```
...     print(p)
[1, 1, 1, 1]
[2, 1, 1]
[2, 2]
[3, 1]
[4]
>>> from sage.combinat.partitions import AccelAsc_iterator
>>> for p in AccelAsc_iterator(Integer(4)):
...     print(p)
[1, 1, 1, 1]
[1, 1, 2]
[1, 3]
[2, 2]
[4]
```

For each of these iterators, this module also provides a `next` method that takes a partition as input and return the next partition in the corresponding ordering:

```
sage: from sage.combinat.partitions import ZS1_next
sage: ZS1_next([2, 2])
[2, 1, 1]
sage: from sage.combinat.partitions import AccelDesc_next
sage: AccelDesc_next([2, 2])
[2, 1, 1]
sage: from sage.combinat.partitions import ZS2_next
sage: ZS2_next([2, 2])
[3, 1]
sage: from sage.combinat.partitions import AccelAsc_next
sage: AccelAsc_next([2, 2])
[4]
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS1_next
>>> ZS1_next([Integer(2), Integer(2)])
[2, 1, 1]
>>> from sage.combinat.partitions import AccelDesc_next
>>> AccelDesc_next([Integer(2), Integer(2)])
[2, 1, 1]
>>> from sage.combinat.partitions import ZS2_next
>>> ZS2_next([Integer(2), Integer(2)])
[3, 1]
>>> from sage.combinat.partitions import AccelAsc_next
>>> AccelAsc_next([Integer(2), Integer(2)])
[4]
```

It is also possible to iterate over the partitions of bounded length:

```
sage: from sage.combinat.partitions import ZS1_iterator_nk
sage: for p in ZS1_iterator_nk(6, 3):
....:     print(p)
[6]
[5, 1]
```

(continues on next page)

(continued from previous page)

```
[4, 2]
[4, 1, 1]
[3, 3]
[3, 2, 1]
[2, 2, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS1_iterator_nk
>>> for p in ZS1_iterator_nk(Integer(6), Integer(3)):
...     print(p)
[6]
[5, 1]
[4, 2]
[4, 1, 1]
[3, 3]
[3, 2, 1]
[2, 2, 2]
```

AUTHOR:

- William Stein (2007-07-28): initial version
- Jonathan Bober (2007-07-28): wrote the program `partitions_c.cc` that does all the actual heavy lifting.
- David Coudert (2024-06-01): reshape method `ZS1_iterator()` to ease the implementation of `ZS1_next()` and add iterators and next methods based on `ZS2`, `AccelAsc` and `AccelDesc` from [ZS1998] and [KS2012].

`sage.combinat.partitions.`**`AccelAsc_iterator`**`(n)`

Return an iterator over the partitions of `n`.

The partitions are generated in the increasing lexicographic order and each partition is represented as a list in ascending order (i.e., $p_i \leq p_{i+1}$).

This is an implementation of the `AccelAsc` algorithm found in [KS2012].

See also

`sage.combinat.partitions.ZS1_iterator()`

EXAMPLES:

```
sage: from sage.combinat.partitions import AccelAsc_iterator
sage: for p in AccelAsc_iterator(4):
....:     print(p)
[1, 1, 1, 1]
[1, 1, 2]
[1, 3]
[2, 2]
[4]
sage: next(AccelAsc_iterator(4))
[1, 1, 1, 1]
sage: type(_)
<class 'list'>
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import AccelAsc_iterator
>>> for p in AccelAsc_iterator(Integer(4)):
...     print(p)
[1, 1, 1, 1]
[1, 1, 2]
[1, 3]
[2, 2]
[4]
>>> next(AccelAsc_iterator(Integer(4)))
[1, 1, 1, 1]
>>> type(_)
<class 'list'>
```

sage.combinat.partitions.**AccelAsc_next**(P)

Return the partition after `P` in the ordering of the `AccelAsc` algorithm.

INPUT:

- `P` – list encoding a partition of an integer n in ascending order (i.e., $P_i \leq P_{i+1}$)

EXAMPLES:

```
sage: from sage.combinat.partitions import AccelAsc_next
sage: P = [1, 1, 1, 1]
sage: while P:
....:     print(P)
....:     P = AccelAsc_next(P)
[1, 1, 1, 1]
[1, 1, 2]
[1, 3]
[2, 2]
[4]
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import AccelAsc_next
>>> P = [Integer(1), Integer(1), Integer(1), Integer(1)]
>>> while P:
...     print(P)
...     P = AccelAsc_next(P)
[1, 1, 1, 1]
[1, 1, 2]
[1, 3]
[2, 2]
[4]
```

sage.combinat.partitions.**AccelDesc_iterator**(n)

Return an iterator over the partitions of `n`.

The partitions are generated in the increasing lexicographic order and each partition is represented as a list in descending order (i.e., $p_i \geq p_{i+1}$).

This is an implementation of the AccelDesc algorithm found in [KS2012].

See also

sage.combinat.partitions.ZS1_iterator()

EXAMPLES:

```
sage: from sage.combinat.partitions import AccelDesc_iterator
sage: for p in AccelDesc_iterator(4):
....:     print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
sage: next(AccelDesc_iterator(4))
[4]
sage: type(_)
<class 'list'>
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import AccelDesc_iterator
>>> for p in AccelDesc_iterator(Integer(4)):
...     print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
>>> next(AccelDesc_iterator(Integer(4)))
[4]
>>> type(_)
<class 'list'>
```

Check that *ZS1_iterator()* and *AccelDesc_iterator()* generate partitions in the same order:

```
sage: from sage.combinat.partitions import ZS1_iterator
sage: from sage.misc.prandom import randint
sage: n = randint(1, 50)
sage: all(p == q for p, q in zip(ZS1_iterator(n), AccelDesc_iterator(n)))   # long time
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS1_iterator
>>> from sage.misc.prandom import randint
>>> n = randint(Integer(1), Integer(50))
>>> all(p == q for p, q in zip(ZS1_iterator(n), AccelDesc_iterator(n)))   # long time
True
```

sage.combinat.partitions.**AccelDesc_next**(*P*)

Return the partition after P in the ordering of the AccelDesc algorithm.

INPUT:

- `P` – list encoding a partition of an integer n in descending order (i.e., $P_i \geq P_{i+1}$)

EXAMPLES:

```
sage: from sage.combinat.partitions import AccelDesc_iterator, AccelDesc_next
sage: P = [4]
sage: while P:
....:     print(P)
....:     P = AccelDesc_next(P)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
sage: A = [list(p) for p in Partitions(7)]
sage: all(AccelDesc_next(p) == q for p, q in zip(A, A[1:]))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import AccelDesc_iterator, AccelDesc_next
>>> P = [Integer(4)]
>>> while P:
...     print(P)
...     P = AccelDesc_next(P)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
>>> A = [list(p) for p in Partitions(Integer(7))]
>>> all(AccelDesc_next(p) == q for p, q in zip(A, A[Integer(1):]))
True
```

sage.combinat.partitions.**ZS1_iterator**(*n*)

Return an iterator over the partitions of `n`.

The partitions are generated in the decreasing lexicographic order and each partition is represented as a list `P` in descending order (i.e., $P_i \geq P_{i+1}$). The method yields lists and not objects of type *Partition*.

This is an implementation of the ZS1 algorithm found in [ZS1998].

See also

- *sage.combinat.partitions.ZS2_iterator()*
- *sage.combinat.partitions.AccelDesc_iterator()*

EXAMPLES:

```
sage: from sage.combinat.partitions import ZS1_iterator
sage: for p in ZS1_iterator(4):
....:     print(p)
```

(continues on next page)

(continued from previous page)

```
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
sage: next(ZS1_iterator(4))
[4]
sage: type(_)
<class 'list'>
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS1_iterator
>>> for p in ZS1_iterator(Integer(4)):
...     print(p)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
>>> next(ZS1_iterator(Integer(4)))
[4]
>>> type(_)
<class 'list'>
```

sage.combinat.partitions.**ZS1_iterator_nk**(n, k)

An iterator for the partitions of n of length at most k (in the decreasing lexicographic order) which returns lists and not objects of type *Partition*.

The algorithm is a mild variation on *ZS1_iterator()*; I would not vow for its speed.

EXAMPLES:

```
sage: from sage.combinat.partitions import ZS1_iterator_nk
sage: it = ZS1_iterator_nk(4, 3)
sage: next(it)
[4]
sage: type(_)
<class 'list'>
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS1_iterator_nk
>>> it = ZS1_iterator_nk(Integer(4), Integer(3))
>>> next(it)
[4]
>>> type(_)
<class 'list'>
```

sage.combinat.partitions.**ZS1_next**(P)

Return the partition after `P` in the ordering of the ZS1 algorithm.

INPUT:

- `P` – list encoding a partition of an integer n in descending order (i.e., $P_i \geq P_{i+1}$)

EXAMPLES:

```
sage: from sage.combinat.partitions import ZS1_iterator, ZS1_next
sage: P = [4]
sage: while P:
....:     print(P)
....:     P = ZS1_next(P)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
sage: A = [list(p) for p in Partitions(7)]
sage: all(ZS1_next(p) == q for p, q in zip(A, A[1:]))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS1_iterator, ZS1_next
>>> P = [Integer(4)]
>>> while P:
...     print(P)
...     P = ZS1_next(P)
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
>>> A = [list(p) for p in Partitions(Integer(7))]
>>> all(ZS1_next(p) == q for p, q in zip(A, A[Integer(1):]))
True
```

sage.combinat.partitions.**ZS2_iterator**(*n*)

Return an iterator over the partitions of n.

The partitions are generated in the increasing lexicographic order and each partition is represented as a list in descending order (i.e., $p_i \geq p_{i+1}$).

This is an implementation of the ZS2 algorithm found in [ZS1998].

See also

sage.combinat.partitions.ZS1_iterator()

EXAMPLES:

```
sage: from sage.combinat.partitions import ZS2_iterator
sage: for p in ZS2_iterator(4):
....:     print(p)
[1, 1, 1, 1]
[2, 1, 1]
[2, 2]
[3, 1]
[4]
sage: next(ZS2_iterator(4))
```

(continues on next page)

(continued from previous page)

```
[1, 1, 1, 1]
sage: type(_)
<class 'list'>
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS2_iterator
>>> for p in ZS2_iterator(Integer(4)):
...     print(p)
[1, 1, 1, 1]
[2, 1, 1]
[2, 2]
[3, 1]
[4]
>>> next(ZS2_iterator(Integer(4)))
[1, 1, 1, 1]
>>> type(_)
<class 'list'>
```

sage.combinat.partitions.**ZS2_next**(P)

Return the partition after P in the ordering of the ZS2 algorithm.

INPUT:

- P – list encoding a partition of an integer n in descending order (i.e., $P_i \geq P_{i+1}$)

EXAMPLES:

```
sage: from sage.combinat.partitions import ZS2_iterator, ZS2_next
sage: P = [1, 1, 1, 1]
sage: while P:
....:     print(P)
....:     P = ZS2_next(P)
[1, 1, 1, 1]
[2, 1, 1]
[2, 2]
[3, 1]
[4]
```

```
>>> from sage.all import *
>>> from sage.combinat.partitions import ZS2_iterator, ZS2_next
>>> P = [Integer(1), Integer(1), Integer(1), Integer(1)]
>>> while P:
...     print(P)
...     P = ZS2_next(P)
[1, 1, 1, 1]
[2, 1, 1]
[2, 2]
[3, 1]
[4]
```

5.1.171 Perfect matchings

A perfect matching of a set S is a partition into 2-element sets. If S is the set $\{1, ..., n\}$, it is equivalent to fixpoint-free involutions. These simple combinatorial objects appear in different domains such as combinatorics of orthogonal polynomials and of the hyperoctaedral groups (see [MV], [McD] and also [CM]):

AUTHOR:

- Valentin Feray, 2010 : initial version
- Martin Rubey, 2017: inherit from SetPartition, move crossings and nestings to SetPartition

EXAMPLES:

Create a perfect matching:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')]);m
[('a', 'e'), ('b', 'c'), ('d', 'f')]
```

```
>>> from sage.all import *
>>> m = PerfectMatching([('a','e'),('b','c'),('d','f')]);m
[('a', 'e'), ('b', 'c'), ('d', 'f')]
```

Count its crossings, if the ground set is totally ordered:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.number_of_crossings()
1
```

```
>>> from sage.all import *
>>> n = PerfectMatching([Integer(3),Integer(8),Integer(1),Integer(7),Integer(6),
→Integer(5),Integer(4),Integer(2)]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
>>> n.number_of_crossings()
1
```

List the perfect matchings of a given ground set:

```
sage: PerfectMatchings(4).list()
[[(1, 2), (3, 4)], [(1, 3), (2, 4)], [(1, 4), (2, 3)]]
```

```
>>> from sage.all import *
>>> PerfectMatchings(Integer(4)).list()
[[(1, 2), (3, 4)], [(1, 3), (2, 4)], [(1, 4), (2, 3)]]
```

REFERENCES:

class sage.combinat.perfect_matching.**PerfectMatching**(*parent*, *s*, *check=True*, *sort=True*)

Bases: *SetPartition*

A perfect matching.

A *perfect matching* of a set X is a set partition of X where all parts have size 2.

A perfect matching can be created from a list of pairs or from a fixed point-free involution as follows:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')]);m
[('a', 'e'), ('b', 'c'), ('d', 'f')]
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]);n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: isinstance(m,PerfectMatching)
True
```

```
>>> from sage.all import *
>>> m = PerfectMatching([('a','e'),('b','c'),('d','f')]);m
[('a', 'e'), ('b', 'c'), ('d', 'f')]
>>> n = PerfectMatching([Integer(3),Integer(8),Integer(1),Integer(7),Integer(6),
↪Integer(5),Integer(4),Integer(2)]);n
[(1, 3), (2, 8), (4, 7), (5, 6)]
>>> isinstance(m,PerfectMatching)
True
```

The parent, which is the set of perfect matchings of the ground set, is automatically created:

```
sage: n.parent()
Perfect matchings of {1, 2, 3, 4, 5, 6, 7, 8}
```

```
>>> from sage.all import *
>>> n.parent()
Perfect matchings of {1, 2, 3, 4, 5, 6, 7, 8}
```

If the ground set is ordered, one can, for example, ask if the matching is non crossing:

```
sage: PerfectMatching([(1, 4), (2, 3), (5, 6)]).is_noncrossing()
True
```

```
>>> from sage.all import *
>>> PerfectMatching([(Integer(1), Integer(4)), (Integer(2), Integer(3)),␣
↪(Integer(5), Integer(6))]).is_noncrossing()
True
```

Weingarten_function(*d*, *other=None*)

Return the Weingarten function of two pairings.

This function is the value of some integrals over the orthogonal groups O_N. With the convention of [CM], the method returns $Wg^{O(d)}(other, self)$.

EXAMPLES:

```
sage: var('N')                                                          #␣
↪needs sage.symbolic
N
sage: m = PerfectMatching([(1,3),(2,4)])
sage: n = PerfectMatching([(1,2),(3,4)])
sage: factor(m.Weingarten_function(N, n))                               #␣
↪needs sage.symbolic
-1/((N + 2)*(N - 1)*N)
```

```
>>> from sage.all import *
>>> var('N')                                                          #␣
↪needs sage.symbolic
N
>>> m = PerfectMatching([(Integer(1),Integer(3)),(Integer(2),Integer(4))])
>>> n = PerfectMatching([(Integer(1),Integer(2)),(Integer(3),Integer(4))])
>>> factor(m.Weingarten_function(N, n))                               #␣
↪needs sage.symbolic
-1/((N + 2)*(N - 1)*N)
```

loop_type(*other=None*)

Return the loop type of `self`.

INPUT:

- `other` – a perfect matching of the same set of `self` (if the second argument is empty, the method `an_element()` is called on the parent of the first)

OUTPUT:

If we draw the two perfect matchings simultaneously as edges of a graph, the graph obtained is a union of cycles of even lengths. The function returns the ordered list of the semi-length of these cycles (considered as a partition)

EXAMPLES:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')])
sage: n = PerfectMatching([('a','b'),('d','f'),('e','c')])
sage: m.loop_type(n)
[2, 1]
```

```
>>> from sage.all import *
>>> m = PerfectMatching([('a','e'),('b','c'),('d','f')])
>>> n = PerfectMatching([('a','b'),('d','f'),('e','c')])
>>> m.loop_type(n)
[2, 1]
```

loops(*other=None*)

Return the loops of `self`.

INPUT:

- `other` – a perfect matching of the same set of `self` (if the second argument is empty, the method `an_element()` is called on the parent of the first)

OUTPUT:

If we draw the two perfect matchings simultaneously as edges of a graph, the graph obtained is a union of cycles of even lengths. The function returns the list of these cycles (each cycle is given as a list).

EXAMPLES:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')])
sage: n = PerfectMatching([('a','b'),('d','f'),('e','c')])
sage: loops = m.loops(n)
sage: loops # random
[['a', 'e', 'c', 'b'], ['d', 'f']]
```

(continues on next page)

(continued from previous page)

```
sage: o = PerfectMatching([(1, 7), (2, 4), (3, 8), (5, 6)])
sage: p = PerfectMatching([(1, 6), (2, 7), (3, 4), (5, 8)])
sage: o.loops(p)
[[1, 7, 2, 4, 3, 8, 5, 6]]
```

```
>>> from sage.all import *
>>> m = PerfectMatching([('a','e'),('b','c'),('d','f')])
>>> n = PerfectMatching([('a','b'),('d','f'),('e','c')])
>>> loops = m.loops(n)
>>> loops # random
[['a', 'e', 'c', 'b'], ['d', 'f']]

>>> o = PerfectMatching([(Integer(1), Integer(7)), (Integer(2), Integer(4)),␣
↪(Integer(3), Integer(8)), (Integer(5), Integer(6))])
>>> p = PerfectMatching([(Integer(1), Integer(6)), (Integer(2), Integer(7)),␣
↪(Integer(3), Integer(4)), (Integer(5), Integer(8))])
>>> o.loops(p)
[[1, 7, 2, 4, 3, 8, 5, 6]]
```

loops_iterator(*other=None*)

Iterate through the loops of `self`.

INPUT:

- `other` – a perfect matching of the same set of `self` (if the second argument is empty, the method `an_element()` is called on the parent of the first)

OUTPUT:

If we draw the two perfect matchings simultaneously as edges of a graph, the graph obtained is a union of cycles of even lengths. The function returns an iterator for these cycles (each cycle is given as a list).

EXAMPLES:

```
sage: o = PerfectMatching([(1, 7), (2, 4), (3, 8), (5, 6)])
sage: p = PerfectMatching([(1, 6), (2, 7), (3, 4), (5, 8)])
sage: it = o.loops_iterator(p)
sage: next(it)
[1, 7, 2, 4, 3, 8, 5, 6]
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

```
>>> from sage.all import *
>>> o = PerfectMatching([(Integer(1), Integer(7)), (Integer(2), Integer(4)),␣
↪(Integer(3), Integer(8)), (Integer(5), Integer(6))])
>>> p = PerfectMatching([(Integer(1), Integer(6)), (Integer(2), Integer(7)),␣
↪(Integer(3), Integer(4)), (Integer(5), Integer(8))])
>>> it = o.loops_iterator(p)
>>> next(it)
[1, 7, 2, 4, 3, 8, 5, 6]
>>> next(it)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
StopIteration
```

number_of_loops(*other=None*)

Return the number of loops of `self`.

INPUT:

- `other` – a perfect matching of the same set of `self` (if the second argument is empty, the method `an_element()` is called on the parent of the first)

OUTPUT:

If we draw the two perfect matchings simultaneously as edges of a graph, the graph obtained is a union of cycles of even lengths. The function returns their numbers.

EXAMPLES:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')])
sage: n = PerfectMatching([('a','b'),('d','f'),('e','c')])
sage: m.number_of_loops(n)
2
```

```
>>> from sage.all import *
>>> m = PerfectMatching([('a','e'),('b','c'),('d','f')])
>>> n = PerfectMatching([('a','b'),('d','f'),('e','c')])
>>> m.number_of_loops(n)
2
```

partner(*x*)

Return the element in the same pair than `x` in the matching `self`.

EXAMPLES:

```
sage: m = PerfectMatching([(-3, 1), (2, 4), (-2, 7)])
sage: m.partner(4)
2
sage: n = PerfectMatching([('c','b'),('d','f'),('e','a')])
sage: n.partner('c')
'b'
```

```
>>> from sage.all import *
>>> m = PerfectMatching([(-Integer(3), Integer(1)), (Integer(2), Integer(4)),␣
↪(-Integer(2), Integer(7))])
>>> m.partner(Integer(4))
2
>>> n = PerfectMatching([('c','b'),('d','f'),('e','a')])
>>> n.partner('c')
'b'
```

standardization()

Return the standardization of `self`.

See *SetPartition.standardization()* for details.

EXAMPLES:

```
sage: n = PerfectMatching([('c','b'),('d','f'),('e','a')])
sage: n.standardization()
[(1, 5), (2, 3), (4, 6)]
```

```
>>> from sage.all import *
>>> n = PerfectMatching([('c','b'),('d','f'),('e','a')])
>>> n.standardization()
[(1, 5), (2, 3), (4, 6)]
```

to_graph()

Return the graph corresponding to the perfect matching.

OUTPUT: the realization of `self` as a graph

EXAMPLES:

```
sage: PerfectMatching([[1,3], [4,2]]).to_graph().edges(sort=True,              #␣
↪needs sage.graphs
....:                                                    labels=False)
[(1, 3), (2, 4)]
sage: PerfectMatching([[1,4], [3,2]]).to_graph().edges(sort=True,              #␣
↪needs sage.graphs
....:                                                    labels=False)
[(1, 4), (2, 3)]
sage: PerfectMatching([]).to_graph().edges(sort=True, labels=False)            #␣
↪needs sage.graphs
[]
```

```
>>> from sage.all import *
>>> PerfectMatching([[Integer(1),Integer(3)], [Integer(4),Integer(2)]]).to_
↪graph().edges(sort=True,            # needs sage.graphs
...                                                    labels=False)
[(1, 3), (2, 4)]
>>> PerfectMatching([[Integer(1),Integer(4)], [Integer(3),Integer(2)]]).to_
↪graph().edges(sort=True,            # needs sage.graphs
...                                                    labels=False)
[(1, 4), (2, 3)]
>>> PerfectMatching([]).to_graph().edges(sort=True, labels=False)          #␣
↪needs sage.graphs
[]
```

to_noncrossing_set_partition()

Return the noncrossing set partition (on half as many elements) corresponding to the perfect matching if the perfect matching is noncrossing, and otherwise gives an error.

OUTPUT: the realization of `self` as a noncrossing set partition

EXAMPLES:

```
sage: PerfectMatching([[1,3], [4,2]]).to_noncrossing_set_partition()
Traceback (most recent call last):
...
ValueError: matching must be non-crossing
```

(continues on next page)

(continued from previous page)

```
sage: PerfectMatching([[1,4], [3,2]]).to_noncrossing_set_partition()
{{1, 2}}
sage: PerfectMatching([]).to_noncrossing_set_partition()
{}
```

```
>>> from sage.all import *
>>> PerfectMatching([[Integer(1),Integer(3)], [Integer(4),Integer(2)]]).to_
↪noncrossing_set_partition()
Traceback (most recent call last):
...
ValueError: matching must be non-crossing
>>> PerfectMatching([[Integer(1),Integer(4)], [Integer(3),Integer(2)]]).to_
↪noncrossing_set_partition()
{{1, 2}}
>>> PerfectMatching([]).to_noncrossing_set_partition()
{}
```

class sage.combinat.perfect_matching.**PerfectMatchings**(*s*)

Bases: *SetPartitions_set*

Perfect matchings of a ground set.

INPUT:

- s – an iterable of hashable objects or an integer

EXAMPLES:

If the argument s is an integer n, it will be transformed into the set $\{1, \ldots, n\}$:

```
sage: M = PerfectMatchings(6); M
Perfect matchings of {1, 2, 3, 4, 5, 6}
sage: PerfectMatchings([-1, -3, 1, 2])
Perfect matchings of {1, 2, -3, -1}
```

```
>>> from sage.all import *
>>> M = PerfectMatchings(Integer(6)); M
Perfect matchings of {1, 2, 3, 4, 5, 6}
>>> PerfectMatchings([-Integer(1), -Integer(3), Integer(1), Integer(2)])
Perfect matchings of {1, 2, -3, -1}
```

One can ask for the list, the cardinality or an element of a set of perfect matching:

```
sage: PerfectMatchings(4).list()
[[(1, 2), (3, 4)], [(1, 3), (2, 4)], [(1, 4), (2, 3)]]
sage: PerfectMatchings(8).cardinality()
105
sage: M = PerfectMatchings(('a', 'e', 'b', 'f', 'c', 'd'))
sage: x = M.an_element()
sage: x # random
[('a', 'c'), ('b', 'e'), ('d', 'f')]
sage: all(PerfectMatchings(i).an_element() in PerfectMatchings(i)
....:     for i in range(2,11,2))
True
```

```
>>> from sage.all import *
>>> PerfectMatchings(Integer(4)).list()
[[(1, 2), (3, 4)], [(1, 3), (2, 4)], [(1, 4), (2, 3)]]
>>> PerfectMatchings(Integer(8)).cardinality()
105
>>> M = PerfectMatchings(('a', 'e', 'b', 'f', 'c', 'd'))
>>> x = M.an_element()
>>> x # random
[('a', 'c'), ('b', 'e'), ('d', 'f')]
>>> all(PerfectMatchings(i).an_element() in PerfectMatchings(i)
...     for i in range(Integer(2),Integer(11),Integer(2)))
True
```

Element

alias of *PerfectMatching*

Weingarten_matrix(N)

Return the Weingarten matrix corresponding to the set of PerfectMatchings `self`.

It is a useful theoretical tool to compute polynomial integrals over the orthogonal group O_N (see [CM]).

EXAMPLES:

```
sage: M = PerfectMatchings(4).Weingarten_matrix(var('N'))                    #␣
↪needs sage.symbolic
sage: N*(N-1)*(N+2)*M.apply_map(factor)                                      #␣
↪needs sage.symbolic
[N + 1    -1    -1]
[   -1 N + 1    -1]
[   -1    -1 N + 1]
```

```
>>> from sage.all import *
>>> M = PerfectMatchings(Integer(4)).Weingarten_matrix(var('N'))             ␣
↪     # needs sage.symbolic
>>> N*(N-Integer(1))*(N+Integer(2))*M.apply_map(factor)                      ␣
↪                # needs sage.symbolic
[N + 1    -1    -1]
[   -1 N + 1    -1]
[   -1    -1 N + 1]
```

base_set()

Return the base set of `self`.

EXAMPLES:

```
sage: PerfectMatchings(3).base_set()
{1, 2, 3}
```

```
>>> from sage.all import *
>>> PerfectMatchings(Integer(3)).base_set()
{1, 2, 3}
```

base_set_cardinality()

Return the cardinality of the base set of `self`.

EXAMPLES:

```
sage: PerfectMatchings(3).base_set_cardinality()
3
```

```
>>> from sage.all import *
>>> PerfectMatchings(Integer(3)).base_set_cardinality()
3
```

cardinality()

Return the cardinality of the set of perfect matchings `self`.

This is $1 * 3 * 5 * ... * (2n - 1)$, where $2n$ is the size of the ground set.

EXAMPLES:

```
sage: PerfectMatchings(8).cardinality()
105
sage: PerfectMatchings([1,2,3,4]).cardinality()
3
sage: PerfectMatchings(3).cardinality()
0
sage: PerfectMatchings([]).cardinality()
1
```

```
>>> from sage.all import *
>>> PerfectMatchings(Integer(8)).cardinality()
105
>>> PerfectMatchings([Integer(1),Integer(2),Integer(3),Integer(4)]).
↪cardinality()
3
>>> PerfectMatchings(Integer(3)).cardinality()
0
>>> PerfectMatchings([]).cardinality()
1
```

random_element()

Return a random element of `self`.

EXAMPLES:

```
sage: M = PerfectMatchings(('a', 'e', 'b', 'f', 'c', 'd'))
sage: x = M.random_element()
sage: x # random
[('a', 'b'), ('c', 'd'), ('e', 'f')]
```

```
>>> from sage.all import *
>>> M = PerfectMatchings(('a', 'e', 'b', 'f', 'c', 'd'))
>>> x = M.random_element()
>>> x # random
[('a', 'b'), ('c', 'd'), ('e', 'f')]
```

5.1.172 Permutations

The Permutations module. Use `Permutation?` to get information about the Permutation class, and `Permutations?` to get information about the combinatorial class of permutations.

> **Warning**
>
> This file defined *Permutation* which depends upon *CombinatorialElement* despite it being deprecated (see Issue #13742). This is dangerous. In particular, the `Permutation._left_to_right_multiply_on_right()` method (which can be called through multiplication) disables the input checks (see *Permutation()*). This should not happen. Do not trust the results.

What does this file define ?

The main part of this file consists in the definition of permutation objects, i.e. the *Permutation()* method and the *Permutation* class. Global options for elements of the permutation class can be set through the *Permutations.options()* object.

Below are listed all methods and classes defined in this file.

Methods of Permutations objects

left_action_product()	Return the product of `self` with another permutation, in which the other permutation is applied first.
right_action_product()	Return the product of `self` with another permutation, in which `self` is applied first.
size()	Return the size of the permutation `self`.
cycle_string()	Return the disjoint-cycles representation of `self` as string.
next()	Return the permutation that follows `self` in lexicographic order (in the same symmetric group as `self`).
prev()	Return the permutation that comes directly before `self` in lexicographic order (in the same symmetric group as `self`).
to_tableau_by_shape()	Return a tableau of shape `shape` with the entries in `self`.
to_cycles()	Return the permutation `self` as a list of disjoint cycles.
forget_cycles()	Return `self` under the forget cycle map.
to_permutation_group_element()	Return a `PermutationGroupElement` equal to `self`.
signature()	Return the signature of the permutation `sef`.
is_even()	Return `True` if the permutation `self` is even, and `False` otherwise.
to_matrix()	Return a matrix representing the permutation `self`.
rank()	Return the rank of `self` in lexicographic ordering (on the symmetric group containing `self`).
to_inversion_vector()	Return the inversion vector of a permutation `self`.
inversions()	Return a list of the inversions of permutation `self`.
stack_sort()	Return the permutation obtained by sorting `self` through one stack.
to_digraph()	Return a digraph representation of `self`.
show()	Display the permutation as a drawing.
number_of_inversions()	Return the number of inversions in the permutation `self`.
noninversions()	Return the `k`-noninversions in the permutation `self`.
number_of_noninversions()	Return the number of `k`-noninversions in the permutation `self`.
length()	Return the Coxeter length of a permutation `self`.
inverse()	Return the inverse of a permutation `self`.

continues on next page

Table 3 – continued from previous page

ishift()	Return the i-shift of self.
iswitch()	Return the i-switch of self.
runs()	Return a list of the runs in the permutation self.
longest_increasing_subsequence_length()	Return the length of the longest increasing subsequences of self.
longest_increasing_subsequences()	Return the list of the longest increasing subsequences of self.
longest_increasing_subsequences_number()	Return the number of longest increasing subsequences
cycle_type()	Return the cycle type of self as a partition of len(self).
foata_bijection()	Return the image of the permutation self under the Foata bijection ϕ.
foata_bijection_inverse()	Return the image of the permutation self under the inverse of the Foata bijection ϕ.
fundamental_transformation()	Return the image of the permutation self under the Renyi-Foata-Schuetzenberger fundamental transformation.
fundamental_transformation_inverse()	Return the image of the permutation self under the inverse of the Renyi-Foata-Schuetzenberger fundamental transformation.
destandardize()	Return destandardization of self with respect to weight and ordered_alphabet.
to_lehmer_code()	Return the Lehmer code of the permutation self.
to_lehmer_cocode()	Return the Lehmer cocode of self.
reduced_word()	Return the reduced word of the permutation self.
reduced_words()	Return a list of the reduced words of the permutation self.
reduced_words_iterator()	An iterator for the reduced words of the permutation self.
reduced_word_lexmin()	Return a lexicographically minimal reduced word of a permutation self.
fixed_points()	Return a list of the fixed points of the permutation self.
is_derangement()	Return True if the permutation self is a derangement, and False otherwise.
is_simple()	Return True if the permutation self is simple, and False otherwise.
number_of_fixed_points()	Return the number of fixed points of the permutation self.
recoils()	Return the list of the positions of the recoils of the permutation self.
number_of_recoils()	Return the number of recoils of the permutation self.
recoils_composition()	Return the composition corresponding to the recoils of self.
descents()	Return the list of the descents of the permutation self.
idescents()	Return a list of the idescents of self.
idescents_signature()	Return the list obtained by mapping each position in self to -1 if it is an idescent and 1 if it is not an idescent.
number_of_descents()	Return the number of descents of the permutation self.
number_of_idescents()	Return the number of idescents of the permutation self.
descents_composition()	Return the composition corresponding to the descents of self.
descent_polynomial()	Return the descent polynomial of the permutation self.
major_index()	Return the major index of the permutation self.
imajor_index()	Return the inverse major index of the permutation self.
to_major_code()	Return the major code of the permutation self.
peaks()	Return a list of the peaks of the permutation self.
number_of_peaks()	Return the number of peaks of the permutation self.
saliances()	Return a list of the saliances of the permutation self.
number_of_saliances()	Return the number of saliances of the permutation self.
bruhat_lequal()	Return True if self is less or equal to p2 in the Bruhat order.
weak_excedences()	Return all the numbers self[i] such that self[i] >= i+1.

continues on next page

Table 3 – continued from previous page

bruhat_inversions()	Return the list of inversions of self such that the application of this inversion to self decrements its number of inversions.
bruhat_inversions_iterator()	Return an iterator over Bruhat inversions of self.
bruhat_succ()	Return a list of the permutations covering self in the Bruhat order.
bruhat_succ_iterator()	An iterator for the permutations covering self in the Bruhat order.
bruhat_pred()	Return a list of the permutations covered by self in the Bruhat order.
bruhat_pred_iterator()	An iterator for the permutations covered by self in the Bruhat order.
bruhat_smaller()	Return the combinatorial class of permutations smaller than or equal to self in the Bruhat order.
bruhat_greater()	Return the combinatorial class of permutations greater than or equal to self in the Bruhat order.
permutohedron_lequal()	Return True if self is less or equal to p2 in the permutohedron order.
permutohedron_succ()	Return a list of the permutations covering self in the permutohedron order.
permutohedron_pred()	Return a list of the permutations covered by self in the permutohedron order.
permutohedron_smaller()	Return a list of permutations smaller than or equal to self in the permutohedron order.
permutohedron_greater()	Return a list of permutations greater than or equal to self in the permutohedron order.
right_permutohedron_interval_iterator()	Return an iterator over permutations in an interval of the permutohedron order.
right_permutohedron_interval()	Return a list of permutations in an interval of the permutohedron order.
has_pattern()	Test whether the permutation self matches the pattern.
avoids()	Test whether the permutation self avoids the pattern.
pattern_positions()	Return the list of positions where the pattern patt appears in self.
reverse()	Return the permutation obtained by reversing the 1-line notation of self.
complement()	Return the complement of the permutation which is obtained by replacing each value x in the 1-line notation of self with $n - x + 1$.
permutation_poset()	Return the permutation poset of self.
dict()	Return a dictionary corresponding to the permutation self.
action()	Return the action of the permutation self on a list.
robinson_schensted()	Return the pair of standard tableaux obtained by running the Robinson-Schensted Algorithm on self.
left_tableau()	Return the left standard tableau after performing the RSK algorithm.
right_tableau()	Return the right standard tableau after performing the RSK algorithm.
increasing_tree()	Return the increasing tree of self.
increasing_tree_shape()	Return the shape of the increasing tree of self.
binary_search_tree()	Return the binary search tree of self.
sylvester_class()	Iterate over the equivalence class of self under sylvester congruence
RS_partition()	Return the shape of the tableaux obtained by the RSK algorithm.
remove_extra_fixed_points()	Return the permutation obtained by removing any fixed points at the end of self.
retract_plain()	Return the plain retract of self to a smaller symmetric group S_m.
retract_direct_product()	Return the direct-product retract of self to a smaller symmetric group S_m.
retract_okounkov_vershik()	Return the Okounkov-Vershik retract of self to a smaller symmetric group S_m.
hyperoctahedral_double_coset_type()	Return the coset-type of self as a partition.
binary_search_tree_shape()	Return the shape of the binary search tree of self (a non labelled binary tree).

continues on next page

Table 3 – continued from previous page

shifted_concatenation()	Return the right (or left) shifted concatenation of `self` with a permutation `other`.
shifted_shuffle()	Return the shifted shuffle of `self` with a permutation `other`.

Other classes defined in this file

Permutations
Permutations_nk
Permutations_mset
Permutations_set
Permutations_msetk
Permutations_setk
Arrangements
Arrangements_msetk
Arrangements_setk
StandardPermutations_all
StandardPermutations_n_abstract
StandardPermutations_n
StandardPermutations_descents
StandardPermutations_recoilsfiner
StandardPermutations_recoilsfatter
StandardPermutations_recoils
StandardPermutations_bruhat_smaller
StandardPermutations_bruhat_greater
CyclicPermutations
CyclicPermutationsOfPartition
StandardPermutations_avoiding_12
StandardPermutations_avoiding_21
StandardPermutations_avoiding_132
StandardPermutations_avoiding_123
StandardPermutations_avoiding_321
StandardPermutations_avoiding_231
StandardPermutations_avoiding_312
StandardPermutations_avoiding_213
StandardPermutations_avoiding_generic
PatternAvoider

Functions defined in this file

from_major_code()	Return the permutation corresponding to major code `mc`.
from_permutation_group_element()	Return a Permutation give a `PermutationGroupElement` `pge`.
from_rank()	Return the permutation with the specified lexicographic rank.
from_inversion_vector()	Return the permutation corresponding to inversion vector `iv`.
from_cycles()	Return the permutation with given disjoint-cycle representation `cycles`.
from_lehmer_code()	Return the permutation with Lehmer code `lehmer`.
from_reduced_word()	Return the permutation corresponding to the reduced word `rw`.
bistochastic_as_sum_of_permutations()	Return a given bistochastic matrix as a nonnegative linear combination of permutations.
bounded_affine_permutation()	Return a partial permutation representing the bounded affine permutation of a matrix.
descents_composition_list()	Return a list of all the permutations in a given descent class (i. e., having a given descents composition).
descents_composition_first()	Return the smallest element of a descent class.
descents_composition_last()	Return the largest element of a descent class.
bruhat_lequal()	Return `True` if `p1` is less or equal to `p2` in the Bruhat order.
permutohedron_lequal()	Return `True` if `p1` is less or equal to `p2` in the permutohedron order.
to_standard()	Return a standard permutation corresponding to the permutation `self`.

AUTHORS:

- Mike Hansen
- Dan Drake (2008-04-07): allow Permutation() to take lists of tuples
- Sébastien Labbé (2009-03-17): added robinson_schensted_inverse
- Travis Scrimshaw:
  - (2012-08-16): `to_standard()` no longer modifies input
  - (2013-01-19): Removed RSK implementation and moved to *rsk*.
  - (2013-07-13): Removed `CombinatorialClass` and moved permutations to the category framework.
- Darij Grinberg (2013-09-07): added methods; ameliorated Issue #14885 by exposing and documenting methods for global-independent multiplication.
- Travis Scrimshaw (2014-02-05): Made *StandardPermutations_n* a finite Weyl group to make it more uniform with `SymmetricGroup`. Added ability to compute the conjugacy classes.
- Trevor K. Karn (2022-08-05): Add `Permutation.n_reduced_words()`
- Amrutha P, Shriya M, Divya Aggarwal (2022-08-16): Added Multimajor Index.

Classes and methods

class sage.combinat.permutation.**Arrangements**

Bases: *Permutations*

An arrangement of a multiset `mset` is an ordered selection without repetitions. It is represented by a list that contains only elements from `mset`, but maybe in a different order.

`Arrangements` returns the combinatorial class of arrangements of the multiset `mset` that contain `k` elements.

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: Arrangements(mset, 2).list()                                              #␣
↪needs sage.libs.gap
[[1, 1],
 [1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [2, 1],
 [2, 3],
 [2, 4],
 [2, 5],
 [3, 1],
 [3, 2],
 [3, 4],
 [3, 5],
 [4, 1],
 [4, 2],
 [4, 3],
 [4, 4],
 [4, 5],
 [5, 1],
 [5, 2],
 [5, 3],
 [5, 4]]
 sage: Arrangements(mset, 2).cardinality()                                      #␣
↪needs sage.libs.gap
 22
 sage: Arrangements( ["c","a","t"], 2 ).list()
 [['c', 'a'], ['c', 't'], ['a', 'c'], ['a', 't'], ['t', 'c'], ['t', 'a']]
 sage: Arrangements( ["c","a","t"], 3 ).list()
 [['c', 'a', 't'],
  ['c', 't', 'a'],
  ['a', 'c', 't'],
  ['a', 't', 'c'],
  ['t', 'c', 'a'],
  ['t', 'a', 'c']]
```

```
>>> from sage.all import *
>>> mset = [Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(4),
↪Integer(5)]
>>> Arrangements(mset, Integer(2)).list()                                        ␣
↪     # needs sage.libs.gap
[[1, 1],
 [1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [2, 1],
 [2, 3],
 [2, 4],
```

(continues on next page)

(continued from previous page)

```
 [2, 5],
 [3, 1],
 [3, 2],
 [3, 4],
 [3, 5],
 [4, 1],
 [4, 2],
 [4, 3],
 [4, 4],
 [4, 5],
 [5, 1],
 [5, 2],
 [5, 3],
 [5, 4]]
>>> Arrangements(mset, Integer(2)).cardinality()                              ␣
↪    # needs sage.libs.gap
22
>>> Arrangements( ["c","a","t"], Integer(2) ).list()
[['c', 'a'], ['c', 't'], ['a', 'c'], ['a', 't'], ['t', 'c'], ['t', 'a']]
>>> Arrangements( ["c","a","t"], Integer(3) ).list()
[['c', 'a', 't'],
 ['c', 't', 'a'],
 ['a', 'c', 't'],
 ['a', 't', 'c'],
 ['t', 'c', 'a'],
 ['t', 'a', 'c']]
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: A = Arrangements([1,1,2,3,4,4,5], 2)
sage: A.cardinality()                                                          #␣
↪needs sage.libs.gap
22
```

```
>>> from sage.all import *
>>> A = Arrangements([Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(4),Integer(5)], Integer(2))
>>> A.cardinality()                                                          #␣
↪needs sage.libs.gap
22
```

class sage.combinat.permutation.**Arrangements_msetk**(*mset*, *k*)

Bases: *Arrangements*, *Permutations_msetk*

Arrangements of length k of a multiset M.

class sage.combinat.permutation.**Arrangements_setk**(*s*, *k*)

Bases: *Arrangements*, *Permutations_setk*

Arrangements of length k of a set S.

class sage.combinat.permutation.**CyclicPermutations**(*mset*)

Bases: *Permutations_mset*

Return the class of all cyclic permutations of mset in cycle notation. These are the same as necklaces.

INPUT:

- mset – a multiset

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list()                                  #␣
↪needs sage.combinat
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
sage: CyclicPermutations([1,1,1]).list()                                   #␣
↪needs sage.combinat
[[1, 1, 1]]
```

```
>>> from sage.all import *
>>> CyclicPermutations(range(Integer(4))).list()                           ␣
↪     # needs sage.combinat
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
>>> CyclicPermutations([Integer(1),Integer(1),Integer(1)]).list()          ␣
↪                        # needs sage.combinat
[[1, 1, 1]]
```

iterator(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list()  # indirect doctest             #␣
↪needs sage.combinat
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
 sage: CyclicPermutations([1,1,1]).list()                                  #␣
↪needs sage.combinat
 [[1, 1, 1]]
 sage: CyclicPermutations([1,1,1]).list(distinct=True)                     #␣
↪needs sage.combinat
 [[1, 1, 1], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> CyclicPermutations(range(Integer(4))).list()  # indirect doctest       ␣
↪     # needs sage.combinat
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
>>> CyclicPermutations([Integer(1),Integer(1),Integer(1)]).list()          ␣
↪                        # needs sage.combinat
 [[1, 1, 1]]
>>> CyclicPermutations([Integer(1),Integer(1),Integer(1)]).
↪list(distinct=True)                    # needs sage.combinat
 [[1, 1, 1], [1, 1, 1]]
```

list(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list()                                 #␣
↪needs sage.combinat
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> CyclicPermutations(range(Integer(4))).list()                          ␣
↪     # needs sage.combinat
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
```

class sage.combinat.permutation.**CyclicPermutationsOfPartition**(*partition*)

Bases: *Permutations*

Combinations of cyclic permutations of each cell of a given partition.

This is the same as a Cartesian product of necklaces.

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],[5,6,7]]).list()           #␣
↪needs sage.combinat
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
```

(continues on next page)

(continued from previous page)

```
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3),Integer(4)],
↪[Integer(5),Integer(6),Integer(7)]]).list()                    # needs sage.
↪combinat
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],[4,4,4]]).list()                #␣
↪needs sage.combinat
[[[1, 2, 3, 4], [4, 4, 4]],
 [[1, 2, 4, 3], [4, 4, 4]],
 [[1, 3, 2, 4], [4, 4, 4]],
 [[1, 3, 4, 2], [4, 4, 4]],
 [[1, 4, 2, 3], [4, 4, 4]],
 [[1, 4, 3, 2], [4, 4, 4]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3),Integer(4)],
↪[Integer(4),Integer(4),Integer(4)]]).list()                    # needs sage.
↪combinat
[[[1, 2, 3, 4], [4, 4, 4]],
 [[1, 2, 4, 3], [4, 4, 4]],
 [[1, 3, 2, 4], [4, 4, 4]],
 [[1, 3, 4, 2], [4, 4, 4]],
 [[1, 4, 2, 3], [4, 4, 4]],
 [[1, 4, 3, 2], [4, 4, 4]]]
```

```
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()                  #␣
↪needs sage.combinat
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3)],[Integer(4),
```

(continues on next page)

(continued from previous page)

```
↪Integer(4),Integer(4)]]).list()                    # needs sage.combinat
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]
```

```
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)       #␣
↪needs sage.combinat
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(4),Integer(4)]]).list(distinct=True)       # needs sage.combinat
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

class Element

Bases: `ClonableArray`

A cyclic permutation of a partition.

check()

Check that `self` is a valid element.

EXAMPLES:

```
sage: CP = CyclicPermutationsOfPartition([[1,2,3,4],[5,6,7]])
sage: elt = CP[0]                                                           #␣
↪needs sage.combinat
sage: elt.check()                                                           #␣
↪needs sage.combinat
```

```
>>> from sage.all import *
>>> CP = CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3),
↪Integer(4)],[Integer(5),Integer(6),Integer(7)]])
>>> elt = CP[Integer(0)]                                                    ␣
↪     # needs sage.combinat
>>> elt.check()                                                          #␣
↪needs sage.combinat
```

iterator(*distinct=False*)

AUTHORS:

- Robert Miller

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],        # indirect doctest    #␣
↪needs sage.combinat
....:                                [5,6,7]]).list()
[[[1, 2, 3, 4], [5, 6, 7]],
```

(continues on next page)

(continued from previous page)

```
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3),
↪Integer(4)],          # indirect doctest    # needs sage.combinat
...                                  [Integer(5),Integer(6),Integer(7)]]).list()
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],[4,4,4]]).list()                  #␣
↪needs sage.combinat
[[[1, 2, 3, 4], [4, 4, 4]],
 [[1, 2, 4, 3], [4, 4, 4]],
 [[1, 3, 2, 4], [4, 4, 4]],
 [[1, 3, 4, 2], [4, 4, 4]],
 [[1, 4, 2, 3], [4, 4, 4]],
 [[1, 4, 3, 2], [4, 4, 4]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3),
↪Integer(4)],[Integer(4),Integer(4),Integer(4)]]).list()          # needs␣
↪sage.combinat
[[[1, 2, 3, 4], [4, 4, 4]],
 [[1, 2, 4, 3], [4, 4, 4]],
 [[1, 3, 2, 4], [4, 4, 4]],
 [[1, 3, 4, 2], [4, 4, 4]],
 [[1, 4, 2, 3], [4, 4, 4]],
 [[1, 4, 3, 2], [4, 4, 4]]]
```

```
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()                    #␣
↪needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3)],
↪[Integer(4),Integer(4),Integer(4)]]).list()                  # needs sage.
↪combinat
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]
```

```
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)   #␣
↪needs sage.combinat
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3)],
↪[Integer(4),Integer(4),Integer(4)]]).list(distinct=True)   # needs sage.
↪combinat
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

list(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()                #␣
↪needs sage.combinat
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)   #␣
↪needs sage.combinat
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

```
>>> from sage.all import *
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3)],
↪[Integer(4),Integer(4),Integer(4)]]).list()                  # needs sage.
↪combinat
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]
>>> CyclicPermutationsOfPartition([[Integer(1),Integer(2),Integer(3)],
↪[Integer(4),Integer(4),Integer(4)]]).list(distinct=True)   # needs sage.
↪combinat
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

class sage.combinat.permutation.**PatternAvoider**(*parent*, *patterns*)

Bases: *GenericBacktracker*

EXAMPLES:

```
sage: from sage.combinat.permutation import PatternAvoider
sage: P = Permutations(4)
sage: p = PatternAvoider(P, [[1,2,3]])
sage: loads(dumps(p))
<sage.combinat.permutation.PatternAvoider object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.combinat.permutation import PatternAvoider
>>> P = Permutations(Integer(4))
>>> p = PatternAvoider(P, [[Integer(1),Integer(2),Integer(3)]])
>>> loads(dumps(p))
<sage.combinat.permutation.PatternAvoider object at 0x...>
```

class sage.combinat.permutation.**Permutation**(*parent*, *l*, *algorithm='lex'*, *sjt=None*, *check=True*)

Bases: `CombinatorialElement`

A permutation.

Converts `l` to a permutation on $\{1, 2, \ldots, n\}$.

INPUT:

- `l` – can be any one of the following:
  - an instance of `Permutation`,
  - list of integers, viewed as one-line permutation notation. The construction checks that you give an acceptable entry. To avoid the check, use the `check` option.
  - string, expressing the permutation in cycle notation.
  - list of tuples of integers, expressing the permutation in cycle notation.
  - a `PermutationGroupElement`
  - a pair of two standard tableaux of the same shape. This yields the permutation obtained from the pair using the inverse of the Robinson-Schensted algorithm.
- `check` – boolean (default: `True`); whether to check that input is correct; slows the function down, but ensures that nothing bad happens
- `algorithm` – string (default: `'lex'`); the algorithm used to generate the permutations. Supported algorithms are:
  - `'lex'`: lexicographic order generation, this is the default algorithm
  - `'sjt'`: Steinhaus-Johnson-Trotter algorithm to generate permutations using only transposition of two elements in the list. It is highly recommended to set `check=True` (default value).
- `sjt` – SJT (default: `None`); the `SJT` object holding the permutation internal state. This should only be specified when initializing with non-identity permutation.

Warning

Since Issue #13742 the input is checked for correctness : it is not accepted unless it actually is a permutation on $\{1, \ldots, n\}$. It means that some `Permutation()` objects cannot be created anymore without setting `check=False`, as there is no certainty that its functions can handle them, and this should be fixed in a much better way ASAP (the functions should be rewritten to handle those cases, and new tests be added).

Warning

There are two possible conventions for multiplying permutations, and the one currently enabled in Sage by default is the one which has $(pq)(i) = q(p(i))$ for any permutations $p \in S_n$ and $q \in S_n$ and any $1 \leq i \leq n$. (This equation looks less strange when the action of permutations on numbers is written from the right: then it takes the form $i^{pq} = (i^p)^q$, which is an associativity law). There is an alternative convention, which has $(pq)(i) = p(q(i))$ instead. The conventions can be switched at runtime using `sage.combinat.permutation.Permutations.options()`. It is best for code not to rely on this setting being set to a particular standard, but rather use the methods `left_action_product()` and `right_action_product()` for multiplying permutations (these methods don't depend on the setting). See Issue #14885 for more details.

Note

The `bruhat*` methods refer to the *strong* Bruhat order. To use the *weak* Bruhat order, look under `permutohedron*`.

EXAMPLES:

```
sage: Permutation([2,1])
[2, 1]
sage: Permutation([2, 1, 4, 5, 3])
[2, 1, 4, 5, 3]
sage: Permutation('(1,2)')
[2, 1]
sage: Permutation('(1,2)(3,4,5)')
[2, 1, 4, 5, 3]
sage: Permutation( ((1,2),(3,4,5)) )
[2, 1, 4, 5, 3]
sage: Permutation( [(1,2),(3,4,5)] )
[2, 1, 4, 5, 3]
sage: Permutation( ((1,2)) )
[2, 1]
sage: Permutation( (1,2) )
[2, 1]
sage: Permutation( ((1,2),) )
[2, 1]
sage: Permutation( ((1,),) )
[1]
sage: Permutation( (1,) )
[1]
sage: Permutation( () )
[]
sage: Permutation( ((),) )
[]
sage: p = Permutation((1, 2, 5)); p
[2, 5, 3, 4, 1]
sage: type(p)
<class 'sage.combinat.permutation.StandardPermutations_n_with_category.element_
→class'>
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1)])
[2, 1]
>>> Permutation([Integer(2), Integer(1), Integer(4), Integer(5), Integer(3)])
[2, 1, 4, 5, 3]
>>> Permutation('(1,2)')
[2, 1]
>>> Permutation('(1,2)(3,4,5)')
[2, 1, 4, 5, 3]
>>> Permutation( ((Integer(1),Integer(2)),(Integer(3),Integer(4),Integer(5))) )
[2, 1, 4, 5, 3]
>>> Permutation( [(Integer(1),Integer(2)),(Integer(3),Integer(4),Integer(5))] )
[2, 1, 4, 5, 3]
>>> Permutation( ((Integer(1),Integer(2))) )
[2, 1]
>>> Permutation( (Integer(1),Integer(2)) )
[2, 1]
>>> Permutation( ((Integer(1),Integer(2)),) )
[2, 1]
>>> Permutation( ((Integer(1),),) )
[1]
>>> Permutation( (Integer(1),) )
[1]
>>> Permutation( () )
[]
>>> Permutation( ((),) )
[]
>>> p = Permutation((Integer(1), Integer(2), Integer(5))); p
[2, 5, 3, 4, 1]
>>> type(p)
<class 'sage.combinat.permutation.StandardPermutations_n_with_category.element_
↪class'>
```

Generate permutations using the Steinhaus-Johnson Trotter algorithm. The output is not in lexicographic order:

```
sage: p = Permutation([1, 2, 3, 4], algorithm='sjt'); p
[1, 2, 3, 4]
sage: p = p.next(); p
[1, 2, 4, 3]
sage: p = p.next(); p
[1, 4, 2, 3]
sage: p = Permutation([1, 2, 3], algorithm='sjt')
sage: for _ in range(6):
....:     p = p.next()
sage: p
False

sage: Permutation([1, 3, 2, 4], algorithm='sjt')
Traceback (most recent call last):
...
ValueError: no internal state directions were given for non-identity
starting permutation for Steinhaus-Johnson-Trotter algorithm
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(1), Integer(2), Integer(3), Integer(4)], algorithm=
↪'sjt'); p
[1, 2, 3, 4]
>>> p = p.next(); p
[1, 2, 4, 3]
>>> p = p.next(); p
[1, 4, 2, 3]
>>> p = Permutation([Integer(1), Integer(2), Integer(3)], algorithm='sjt')
>>> for _ in range(Integer(6)):
...     p = p.next()
>>> p
False

>>> Permutation([Integer(1), Integer(3), Integer(2), Integer(4)], algorithm='sjt')
Traceback (most recent call last):
...
ValueError: no internal state directions were given for non-identity
starting permutation for Steinhaus-Johnson-Trotter algorithm
```

Construction from a string in cycle notation:

```
sage: p = Permutation( '(4,5)' ); p
[1, 2, 3, 5, 4]
```

```
>>> from sage.all import *
>>> p = Permutation( '(4,5)' ); p
[1, 2, 3, 5, 4]
```

The size of the permutation is the maximum integer appearing; add a 1-cycle to increase this:

```
sage: p2 = Permutation( '(4,5)(10)' ); p2
[1, 2, 3, 5, 4, 6, 7, 8, 9, 10]
sage: len(p); len(p2)
5
10
```

```
>>> from sage.all import *
>>> p2 = Permutation( '(4,5)(10)' ); p2
[1, 2, 3, 5, 4, 6, 7, 8, 9, 10]
>>> len(p); len(p2)
5
10
```

We construct a *Permutation* from a `PermutationGroupElement`:

```
sage: g = PermutationGroupElement([2,1,3])                                    #␣
↪needs sage.groups
sage: Permutation(g)                                                          #␣
↪needs sage.groups
[2, 1, 3]
```

```
>>> from sage.all import *
>>> g = PermutationGroupElement([Integer(2),Integer(1),Integer(3)])              ␣
↪                          # needs sage.groups
>>> Permutation(g)                                                              #␣
↪needs sage.groups
[2, 1, 3]
```

From a pair of tableaux of the same shape. This uses the inverse of the Robinson-Schensted algorithm:

```
sage: # needs sage.combinat
sage: p = [[1, 4, 7], [2, 5], [3], [6]]
sage: q = [[1, 2, 5], [3, 6], [4], [7]]
sage: P = Tableau(p)
sage: Q = Tableau(q)
sage: Permutation( (p, q) )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( [p, q] )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( (P, Q) )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( [P, Q] )
[3, 6, 5, 2, 7, 4, 1]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> p = [[Integer(1), Integer(4), Integer(7)], [Integer(2), Integer(5)],␣
↪[Integer(3)], [Integer(6)]]
>>> q = [[Integer(1), Integer(2), Integer(5)], [Integer(3), Integer(6)],␣
↪[Integer(4)], [Integer(7)]]
>>> P = Tableau(p)
>>> Q = Tableau(q)
>>> Permutation( (p, q) )
[3, 6, 5, 2, 7, 4, 1]
>>> Permutation( [p, q] )
[3, 6, 5, 2, 7, 4, 1]
>>> Permutation( (P, Q) )
[3, 6, 5, 2, 7, 4, 1]
>>> Permutation( [P, Q] )
[3, 6, 5, 2, 7, 4, 1]
```

RS_partition()

Return the shape of the tableaux obtained by applying the RSK algorithm to `self`.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).RS_partition()                                     #␣
↪needs sage.combinat
[2, 1, 1]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).RS_partition()␣
↪                                  # needs sage.combinat
[2, 1, 1]
```

absolute_length()

Return the absolute length of `self`

The absolute length is the length of the shortest expression of the element as a product of reflections.

For permutations in the symmetric groups, the absolute length is the size minus the number of its disjoint cycles.

EXAMPLES:

```
sage: Permutation([4,2,3,1]).absolute_length()                              #␣
↪needs sage.combinat
1
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(2),Integer(3),Integer(1)]).absolute_
↪length()                                       # needs sage.combinat
1
```

action(*a*)

Return the action of the permutation `self` on a list `a`.

The action of a permutation $p \in S_n$ on an n-element list $(a_1, a_2, \ldots, a_n)$ is defined to be $(a_{p(1)}, a_{p(2)}, \ldots, a_{p(n)})$.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: a = list(range(3))
sage: p.action(a)
[1, 0, 2]
sage: b = [1,2,3,4]
sage: p.action(b)
Traceback (most recent call last):
...
ValueError: len(a) must equal len(self)

sage: q = Permutation([2,3,1])
sage: a = list(range(3))
sage: q.action(a)
[1, 2, 0]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2),Integer(1),Integer(3)])
>>> a = list(range(Integer(3)))
>>> p.action(a)
[1, 0, 2]
>>> b = [Integer(1),Integer(2),Integer(3),Integer(4)]
>>> p.action(b)
Traceback (most recent call last):
...
ValueError: len(a) must equal len(self)

>>> q = Permutation([Integer(2),Integer(3),Integer(1)])
>>> a = list(range(Integer(3)))
```

(continues on next page)

(continued from previous page)

```
>>> q.action(a)
[1, 2, 0]
```

avoids (*patt*)

Test whether the permutation `self` avoids the pattern `patt`.

EXAMPLES:

```
sage: Permutation([6,2,5,4,3,1]).avoids([4,2,3,1])                          #␣
↪needs sage.combinat
False
sage: Permutation([6,1,2,5,4,3]).avoids([4,2,3,1])                          #␣
↪needs sage.combinat
True
sage: Permutation([6,1,2,5,4,3]).avoids([3,4,1,2])                          #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> Permutation([Integer(6),Integer(2),Integer(5),Integer(4),Integer(3),
↪Integer(1)]).avoids([Integer(4),Integer(2),Integer(3),Integer(1)])          ␣
↪                 # needs sage.combinat
False
>>> Permutation([Integer(6),Integer(1),Integer(2),Integer(5),Integer(4),
↪Integer(3)]).avoids([Integer(4),Integer(2),Integer(3),Integer(1)])          ␣
↪                 # needs sage.combinat
True
>>> Permutation([Integer(6),Integer(1),Integer(2),Integer(5),Integer(4),
↪Integer(3)]).avoids([Integer(3),Integer(4),Integer(1),Integer(2)])          ␣
↪                 # needs sage.combinat
True
```

binary_search_tree (*left_to_right=True*)

Return the binary search tree associated to `self`.

If w is a word, then the binary search tree associated to w is defined as the result of starting with an empty binary tree, and then inserting the letters of w one by one into this tree. Here, the insertion is being done according to the method *binary_search_insert()*, and the word w is being traversed from left to right.

A permutation is regarded as a word (using one-line notation), and thus a binary search tree associated to a permutation is defined.

If the optional keyword variable `left_to_right` is set to `False`, the word w is being traversed from right to left instead.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).binary_search_tree()                            #␣
↪needs sage.graphs
1[., 4[3[2[., .], .], .]]
sage: Permutation([4,1,3,2]).binary_search_tree()                            #␣
↪needs sage.graphs
4[1[., 3[2[., .], .]], .]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).binary_search_
↪tree()                                 # needs sage.graphs
1[., 4[3[2[., .], .], .]]
>>> Permutation([Integer(4),Integer(1),Integer(3),Integer(2)]).binary_search_
↪tree()                                 # needs sage.graphs
4[1[., 3[2[., .], .]], .]
```

By passing the option `left_to_right=False` one can have the insertion going from right to left:

```
sage: Permutation([1,4,3,2]).binary_search_tree(False)                     #␣
↪needs sage.graphs
2[1[., .], 3[., 4[., .]]]
sage: Permutation([4,1,3,2]).binary_search_tree(False)                     #␣
↪needs sage.graphs
2[1[., .], 3[., 4[., .]]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).binary_search_
↪tree(False)                            # needs sage.graphs
2[1[., .], 3[., 4[., .]]]
>>> Permutation([Integer(4),Integer(1),Integer(3),Integer(2)]).binary_search_
↪tree(False)                            # needs sage.graphs
2[1[., .], 3[., 4[., .]]]
```

binary_search_tree_shape(*left_to_right=True*)

Return the shape of the binary search tree of the permutation (a non labelled binary tree).

EXAMPLES:

```
sage: Permutation([1,4,3,2]).binary_search_tree_shape()                    #␣
↪needs sage.graphs
[., [[[., .], .], .]]
sage: Permutation([4,1,3,2]).binary_search_tree_shape()                    #␣
↪needs sage.graphs
[[., [[., .], .]], .]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).binary_search_
↪tree_shape()                           # needs sage.graphs
[., [[[., .], .], .]]
>>> Permutation([Integer(4),Integer(1),Integer(3),Integer(2)]).binary_search_
↪tree_shape()                           # needs sage.graphs
[[., [[., .], .]], .]
```

By passing the option `left_to_right=False` one can have the insertion going from right to left:

```
sage: Permutation([1,4,3,2]).binary_search_tree_shape(False)               #␣
↪needs sage.graphs
[[., .], [., [., .]]]
sage: Permutation([4,1,3,2]).binary_search_tree_shape(False)               #␣
↪needs sage.graphs
[[., .], [., [., .]]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).binary_search_
↪tree_shape(False)                    # needs sage.graphs
[[., .], [., [., .]]]
>>> Permutation([Integer(4),Integer(1),Integer(3),Integer(2)]).binary_search_
↪tree_shape(False)                    # needs sage.graphs
[[., .], [., [., .]]]
```

bruhat_greater()

Return the combinatorial class of permutations greater than or equal to `self` in the Bruhat order (on the symmetric group containing `self`).

See *bruhat_lequal()* for the definition of the Bruhat order.

EXAMPLES:

```
sage: Permutation([4,1,2,3]).bruhat_greater().list()
[[4, 1, 2, 3],
 [4, 1, 3, 2],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3)]).bruhat_
↪greater().list()
[[4, 1, 2, 3],
 [4, 1, 3, 2],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
```

bruhat_inversions()

Return the list of inversions of `self` such that the application of this inversion to `self` decreases its number of inversions by exactly 1.

Equivalently, it returns the list of pairs (i, j) such that $i < j$, such that $p(i) > p(j)$ and such that there exists no k (strictly) between i and j satisfying $p(i) > p(k) > p(j)$.

EXAMPLES:

```
sage: Permutation([5,2,3,4,1]).bruhat_inversions()
[[0, 1], [0, 2], [0, 3], [1, 4], [2, 4], [3, 4]]
sage: Permutation([6,1,4,5,2,3]).bruhat_inversions()
[[0, 1], [0, 2], [0, 3], [2, 4], [2, 5], [3, 4], [3, 5]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(5),Integer(2),Integer(3),Integer(4),Integer(1)]).
↪bruhat_inversions()
[[0, 1], [0, 2], [0, 3], [1, 4], [2, 4], [3, 4]]
>>> Permutation([Integer(6),Integer(1),Integer(4),Integer(5),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)]).bruhat_inversions()
[[0, 1], [0, 2], [0, 3], [2, 4], [2, 5], [3, 4], [3, 5]]
```

bruhat_inversions_iterator()

Return the iterator for the inversions of `self` such that the application of this inversion to `self` decreases its number of inversions by exactly 1.

EXAMPLES:

```
sage: list(Permutation([5,2,3,4,1]).bruhat_inversions_iterator())
[[0, 1], [0, 2], [0, 3], [1, 4], [2, 4], [3, 4]]
sage: list(Permutation([6,1,4,5,2,3]).bruhat_inversions_iterator())
[[0, 1], [0, 2], [0, 3], [2, 4], [2, 5], [3, 4], [3, 5]]
```

```
>>> from sage.all import *
>>> list(Permutation([Integer(5),Integer(2),Integer(3),Integer(4),
↪Integer(1)]).bruhat_inversions_iterator())
[[0, 1], [0, 2], [0, 3], [1, 4], [2, 4], [3, 4]]
>>> list(Permutation([Integer(6),Integer(1),Integer(4),Integer(5),Integer(2),
↪Integer(3)]).bruhat_inversions_iterator())
[[0, 1], [0, 2], [0, 3], [2, 4], [2, 5], [3, 4], [3, 5]]
```

bruhat_lequal(*p2*)

Return `True` if `self` is less or equal to `p2` in the Bruhat order.

The Bruhat order (also called strong Bruhat order or Chevalley order) on the symmetric group S_n is the partial order on S_n determined by the following condition: If p is a permutation, and i and j are two indices satisfying $p(i) > p(j)$ and $i < j$ (that is, (i, j) is an inversion of p with $i < j$), then $p \circ (i, j)$ (the permutation obtained by first switching i with j and then applying p) is smaller than p in the Bruhat order.

One can show that a permutation $p \in S_n$ is less or equal to a permutation $q \in S_n$ in the Bruhat order if and only if for every $i \in \{0, 1, \cdots, n\}$ and $j \in \{1, 2, \cdots, n\}$, the number of the elements among $p(1), p(2), \cdots, p(j)$ that are greater than i is $\leq$ to the number of the elements among $q(1), q(2), \cdots, q(j)$ that are greater than i.

This method assumes that `self` and `p2` are permutations of the same integer n.

EXAMPLES:

```
sage: Permutation([2,4,3,1]).bruhat_lequal(Permutation([3,4,2,1]))
True

sage: Permutation([2,1,3]).bruhat_lequal(Permutation([2,3,1]))
True
sage: Permutation([2,1,3]).bruhat_lequal(Permutation([3,1,2]))
True
sage: Permutation([2,1,3]).bruhat_lequal(Permutation([1,2,3]))
False
sage: Permutation([1,3,2]).bruhat_lequal(Permutation([2,1,3]))
False
sage: Permutation([1,3,2]).bruhat_lequal(Permutation([2,3,1]))
True
sage: Permutation([2,3,1]).bruhat_lequal(Permutation([1,3,2]))
False
```

(continues on next page)

(continued from previous page)

```
sage: sorted( [len([b for b in Permutations(3) if a.bruhat_lequal(b)])
....:           for a in Permutations(3)] )
[1, 2, 2, 4, 4, 6]

sage: Permutation([]).bruhat_lequal(Permutation([]))
True
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(4),Integer(3),Integer(1)]).bruhat_
↪lequal(Permutation([Integer(3),Integer(4),Integer(2),Integer(1)]))
True

>>> Permutation([Integer(2),Integer(1),Integer(3)]).bruhat_
↪lequal(Permutation([Integer(2),Integer(3),Integer(1)]))
True
>>> Permutation([Integer(2),Integer(1),Integer(3)]).bruhat_
↪lequal(Permutation([Integer(3),Integer(1),Integer(2)]))
True
>>> Permutation([Integer(2),Integer(1),Integer(3)]).bruhat_
↪lequal(Permutation([Integer(1),Integer(2),Integer(3)]))
False
>>> Permutation([Integer(1),Integer(3),Integer(2)]).bruhat_
↪lequal(Permutation([Integer(2),Integer(1),Integer(3)]))
False
>>> Permutation([Integer(1),Integer(3),Integer(2)]).bruhat_
↪lequal(Permutation([Integer(2),Integer(3),Integer(1)]))
True
>>> Permutation([Integer(2),Integer(3),Integer(1)]).bruhat_
↪lequal(Permutation([Integer(1),Integer(3),Integer(2)]))
False
>>> sorted( [len([b for b in Permutations(Integer(3)) if a.bruhat_lequal(b)])
...           for a in Permutations(Integer(3))] )
[1, 2, 2, 4, 4, 6]

>>> Permutation([]).bruhat_lequal(Permutation([]))
True
```

bruhat_pred()

Return a list of the permutations strictly smaller than `self` in the Bruhat order (on the symmetric group containing `self`) such that there is no permutation between one of those and `self`.

See *bruhat_lequal()* for the definition of the Bruhat order.

EXAMPLES:

```
sage: Permutation([6,1,4,5,2,3]).bruhat_pred()
[[1, 6, 4, 5, 2, 3],
 [4, 1, 6, 5, 2, 3],
 [5, 1, 4, 6, 2, 3],
 [6, 1, 2, 5, 4, 3],
 [6, 1, 3, 5, 2, 4],
 [6, 1, 4, 2, 5, 3],
 [6, 1, 4, 3, 2, 5]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(6),Integer(1),Integer(4),Integer(5),Integer(2),
↪Integer(3)]).bruhat_pred()
[[1, 6, 4, 5, 2, 3],
 [4, 1, 6, 5, 2, 3],
 [5, 1, 4, 6, 2, 3],
 [6, 1, 2, 5, 4, 3],
 [6, 1, 3, 5, 2, 4],
 [6, 1, 4, 2, 5, 3],
 [6, 1, 4, 3, 2, 5]]
```

bruhat_pred_iterator()

An iterator for the permutations strictly smaller than `self` in the Bruhat order (on the symmetric group containing `self`) such that there is no permutation between one of those and `self`.

See *bruhat_lequal()* for the definition of the Bruhat order.

EXAMPLES:

```
sage: [x for x in Permutation([6,1,4,5,2,3]).bruhat_pred_iterator()]
[[1, 6, 4, 5, 2, 3],
 [4, 1, 6, 5, 2, 3],
 [5, 1, 4, 6, 2, 3],
 [6, 1, 2, 5, 4, 3],
 [6, 1, 3, 5, 2, 4],
 [6, 1, 4, 2, 5, 3],
 [6, 1, 4, 3, 2, 5]]
```

```
>>> from sage.all import *
>>> [x for x in Permutation([Integer(6),Integer(1),Integer(4),Integer(5),
↪Integer(2),Integer(3)]).bruhat_pred_iterator()]
[[1, 6, 4, 5, 2, 3],
 [4, 1, 6, 5, 2, 3],
 [5, 1, 4, 6, 2, 3],
 [6, 1, 2, 5, 4, 3],
 [6, 1, 3, 5, 2, 4],
 [6, 1, 4, 2, 5, 3],
 [6, 1, 4, 3, 2, 5]]
```

bruhat_smaller()

Return the combinatorial class of permutations smaller than or equal to `self` in the Bruhat order (on the symmetric group containing `self`).

See *bruhat_lequal()* for the definition of the Bruhat order.

EXAMPLES:

```
sage: Permutation([4,1,2,3]).bruhat_smaller().list()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 3, 2, 4],
 [1, 4, 2, 3],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
```

(continues on next page)

(continued from previous page)

```
 [3, 1, 2, 4],
 [4, 1, 2, 3]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3)]).bruhat_
↪smaller().list()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 3, 2, 4],
 [1, 4, 2, 3],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [3, 1, 2, 4],
 [4, 1, 2, 3]]
```

bruhat_succ()

Return a list of the permutations strictly greater than `self` in the Bruhat order (on the symmetric group containing `self`) such that there is no permutation between one of those and `self`.

See *bruhat_lequal()* for the definition of the Bruhat order.

EXAMPLES:

```
sage: Permutation([6,1,4,5,2,3]).bruhat_succ()
[[6, 4, 1, 5, 2, 3],
 [6, 2, 4, 5, 1, 3],
 [6, 1, 5, 4, 2, 3],
 [6, 1, 4, 5, 3, 2]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(6),Integer(1),Integer(4),Integer(5),Integer(2),
↪Integer(3)]).bruhat_succ()
[[6, 4, 1, 5, 2, 3],
 [6, 2, 4, 5, 1, 3],
 [6, 1, 5, 4, 2, 3],
 [6, 1, 4, 5, 3, 2]]
```

bruhat_succ_iterator()

An iterator for the permutations that are strictly greater than `self` in the Bruhat order (on the symmetric group containing `self`) such that there is no permutation between one of those and `self`.

See *bruhat_lequal()* for the definition of the Bruhat order.

EXAMPLES:

```
sage: [x for x in Permutation([6,1,4,5,2,3]).bruhat_succ_iterator()]
[[6, 4, 1, 5, 2, 3],
 [6, 2, 4, 5, 1, 3],
 [6, 1, 5, 4, 2, 3],
 [6, 1, 4, 5, 3, 2]]
```

```
>>> from sage.all import *
>>> [x for x in Permutation([Integer(6),Integer(1),Integer(4),Integer(5),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(3)]).bruhat_succ_iterator()]
[[6, 4, 1, 5, 2, 3],
 [6, 2, 4, 5, 1, 3],
 [6, 1, 5, 4, 2, 3],
 [6, 1, 4, 5, 3, 2]]
```

complement()

Return the complement of the permutation `self`.

The complement of a permutation $w \in S_n$ is defined as the permutation in S_n sending each i to $n+1-w(i)$.

EXAMPLES:

```
sage: Permutation([1,2,3]).complement()
[3, 2, 1]
sage: Permutation([1, 3, 2]).complement()
[3, 1, 2]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(3)]).complement()
[3, 2, 1]
>>> Permutation([Integer(1), Integer(3), Integer(2)]).complement()
[3, 1, 2]
```

cycle_string(*singletons=False*)

Return a string of the permutation in cycle notation.

If `singletons=True`, it includes 1-cycles in the string.

EXAMPLES:

```
sage: Permutation([1,2,3]).cycle_string()
'()'
sage: Permutation([2,1,3]).cycle_string()
'(1,2)'
sage: Permutation([2,3,1]).cycle_string()
'(1,2,3)'
sage: Permutation([2,1,3]).cycle_string(singletons=True)
'(1,2)(3)'
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(3)]).cycle_string()
'()'
>>> Permutation([Integer(2),Integer(1),Integer(3)]).cycle_string()
'(1,2)'
>>> Permutation([Integer(2),Integer(3),Integer(1)]).cycle_string()
'(1,2,3)'
>>> Permutation([Integer(2),Integer(1),Integer(3)]).cycle_
↪string(singletons=True)
'(1,2)(3)'
```

cycle_tuples(*singletons=True*, *use_min=True*)

Return the permutation `self` as a list of disjoint cycles.

The cycles are returned in the order of increasing smallest elements, and each cycle is returned as a tuple which starts with its smallest element.

If `singletons=False` is given, the list does not contain the singleton cycles.

If `use_min=False` is given, the cycles are returned in the order of increasing *largest* (not smallest) elements, and each cycle starts with its largest element.

EXAMPLES:

```
sage: Permutation([2,1,3,4]).to_cycles()
[(1, 2), (3,), (4,)]
sage: Permutation([2,1,3,4]).to_cycles(singletons=False)
[(1, 2)]
sage: Permutation([2,1,3,4]).to_cycles(use_min=True)
[(1, 2), (3,), (4,)]
sage: Permutation([2,1,3,4]).to_cycles(use_min=False)
[(4,), (3,), (2, 1)]
sage: Permutation([2,1,3,4]).to_cycles(singletons=False, use_min=False)
[(2, 1)]

sage: Permutation([4,1,5,2,6,3]).to_cycles()
[(1, 4, 2), (3, 5, 6)]
sage: Permutation([4,1,5,2,6,3]).to_cycles(use_min=False)
[(6, 3, 5), (4, 2, 1)]

sage: Permutation([6, 4, 5, 2, 3, 1]).to_cycles()
[(1, 6), (2, 4), (3, 5)]
sage: Permutation([6, 4, 5, 2, 3, 1]).to_cycles(use_min=False)
[(6, 1), (5, 3), (4, 2)]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_cycles()
[(1, 2), (3,), (4,)]
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_
→cycles(singletons=False)
[(1, 2)]
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_cycles(use_
→min=True)
[(1, 2), (3,), (4,)]
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_cycles(use_
→min=False)
[(4,), (3,), (2, 1)]
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_
→cycles(singletons=False, use_min=False)
[(2, 1)]

>>> Permutation([Integer(4),Integer(1),Integer(5),Integer(2),Integer(6),
→Integer(3)]).to_cycles()
[(1, 4, 2), (3, 5, 6)]
>>> Permutation([Integer(4),Integer(1),Integer(5),Integer(2),Integer(6),
→Integer(3)]).to_cycles(use_min=False)
[(6, 3, 5), (4, 2, 1)]

>>> Permutation([Integer(6), Integer(4), Integer(5), Integer(2), Integer(3),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)]).to_cycles()
[(1, 6), (2, 4), (3, 5)]
>>> Permutation([Integer(6), Integer(4), Integer(5), Integer(2), Integer(3),␣
↪Integer(1)]).to_cycles(use_min=False)
[(6, 1), (5, 3), (4, 2)]
```

The algorithm is of complexity $O(n)$ where n is the size of the given permutation.

cycle_type()

Return a partition of `len(self)` corresponding to the cycle type of `self`.

This is a non-increasing sequence of the cycle lengths of `self`.

EXAMPLES:

```
sage: Permutation([3,1,2,4]).cycle_type()                                #␣
↪needs sage.combinat
[3, 1]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(1),Integer(2),Integer(4)]).cycle_type()  ␣
↪                                 # needs sage.combinat
[3, 1]
```

decreasing_runs(*as_tuple=False*)

Decreasing runs of the permutation.

INPUT:

- `as_tuple` – boolean (default: `False`); choice of output format

OUTPUT: list of lists or a tuple of tuples

See also

runs()

EXAMPLES:

```
sage: s = Permutation([2,8,3,9,6,4,5,1,7])
sage: s.decreasing_runs()
[[2], [8, 3], [9, 6, 4], [5, 1], [7]]
sage: s.decreasing_runs(as_tuple=True)
((2,), (8, 3), (9, 6, 4), (5, 1), (7,))
```

```
>>> from sage.all import *
>>> s = Permutation([Integer(2),Integer(8),Integer(3),Integer(9),Integer(6),
↪Integer(4),Integer(5),Integer(1),Integer(7)])
>>> s.decreasing_runs()
[[2], [8, 3], [9, 6, 4], [5, 1], [7]]
>>> s.decreasing_runs(as_tuple=True)
((2,), (8, 3), (9, 6, 4), (5, 1), (7,))
```

descent_polynomial()

Return the descent polynomial of the permutation self.

The descent polynomial of a permutation p is the product of all the z[p(i)] where i ranges over the descents of p.

A descent of a permutation p is an integer i such that p(i) > p(i+1).

REFERENCES:

- [GS1984]

EXAMPLES:

```
sage: Permutation([2,1,3]).descent_polynomial()
z2
sage: Permutation([4,3,2,1]).descent_polynomial()
z2*z3*z4
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3)]).descent_polynomial()
z2
>>> Permutation([Integer(4),Integer(3),Integer(2),Integer(1)]).descent_
↪polynomial()
z2*z3*z4
```

Todo

This docstring needs to be fixed. This is not defined in [GS1984] (the descent monomial in their (7.23) is different).

descents(*final_descent=False*, *side='right'*, *positive=False*, *from_zero=False*, *index_set=None*)

Return the list of the descents of self.

A descent of a permutation p is an integer i such that $p(i) > p(i+1)$.

Warning

By default, the descents are returned as elements in the index set, i.e., starting at 1. If you want them to start at 0, set the keyword from_zero to True.

INPUT:

- final_descent – boolean (default: False); if True, the last position of a non-empty permutation is also considered as a descent
- side – 'right' (default) or 'left'; if 'left', return the descents of the inverse permutation
- positive – boolean (default: False); if True, return the positions that are not descents
- from_zero – boolean (default: False); if True, return the positions starting from 0
- index_set – list (default: [1, ..., n-1] where self is a permutation of n); the index set to check for descents

EXAMPLES:

```
sage: Permutation([3,1,2]).descents()
[1]
sage: Permutation([1,4,3,2]).descents()
[2, 3]
sage: Permutation([1,4,3,2]).descents(final_descent=True)
[2, 3, 4]
sage: Permutation([1,4,3,2]).descents(index_set=[1,2])
[2]
sage: Permutation([1,4,3,2]).descents(from_zero=True)
[1, 2]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(1),Integer(2)]).descents()
[1]
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).descents()
[2, 3]
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).descents(final_
↪descent=True)
[2, 3, 4]
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).descents(index_
↪set=[Integer(1),Integer(2)])
[2]
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).descents(from_
↪zero=True)
[1, 2]
```

descents_composition()

Return the descent composition of `self`.

The descent composition of a permutation $p \in S_n$ is defined as the composition of n whose descent set equals the descent set of p. Here, the descent set of p is defined as the set of all $i \in \{1, 2, \ldots, n-1\}$ satisfying $p(i) > p(i+1)$. The descent set of a composition $c = (i_1, i_2, \ldots, i_k)$ is defined as the set $\{i_1, i_1 + i_2, i_1 + i_2 + i_3, \ldots, i_1 + i_2 + \cdots + i_{k-1}\}$.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).descents_composition()
[2, 2]
sage: Permutation([4,1,6,7,2,3,8,5]).descents_composition()
[1, 3, 3, 1]
sage: Permutation([]).descents_composition()
[]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(4)]).descents_
↪composition()
[2, 2]
>>> Permutation([Integer(4),Integer(1),Integer(6),Integer(7),Integer(2),
↪Integer(3),Integer(8),Integer(5)]).descents_composition()
[1, 3, 3, 1]
>>> Permutation([]).descents_composition()
[]
```

destandardize(*weight*, *ordered_alphabet=None*)

Return destandardization of `self` with respect to `weight` and `ordered_alphabet`.

INPUT:

- `weight` – list or tuple of nonnegative integers that sum to n if `self` is a permutation in S_n
- `ordered_alphabet` – (default: `None`) a list or tuple specifying the ordered alphabet the destandardized word is over

OUTPUT: word over the `ordered_alphabet` which standardizes to `self`

Let $weight = (w_1, w_2, \ldots, w_\ell)$. Then this methods looks for an increasing sequence of $1, 2, \ldots, w_1$ and labels all letters in it by 1, then an increasing sequence of $w_1 + 1, w_1 + 2, \ldots, w_1 + w_2$ and labels all these letters by 2, etc.. If an increasing sequence for the specified `weight` does not exist, an error is returned. The output is a word `w` over the specified ordered alphabet with evaluation `weight` such that `w.standard_permutation()` is `self`.

EXAMPLES:

```
sage: p = Permutation([1,2,5,3,6,4])
sage: p.destandardize([3,1,2])                                          #␣
↪needs sage.combinat
word: 113132
sage: p = Permutation([2,1,3])
sage: p.destandardize([2,1])
Traceback (most recent call last):
...
ValueError: Standardization with weight [2, 1] is not possible!
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(1),Integer(2),Integer(5),Integer(3),Integer(6),
↪Integer(4)])
>>> p.destandardize([Integer(3),Integer(1),Integer(2)])                 ␣
↪                       # needs sage.combinat
word: 113132
>>> p = Permutation([Integer(2),Integer(1),Integer(3)])
>>> p.destandardize([Integer(2),Integer(1)])
Traceback (most recent call last):
...
ValueError: Standardization with weight [2, 1] is not possible!
```

dict()

Return a dictionary corresponding to the permutation.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: d = p.dict()
sage: d[1]
2
sage: d[2]
1
sage: d[3]
3
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2),Integer(1),Integer(3)])
>>> d = p.dict()
>>> d[Integer(1)]
2
>>> d[Integer(2)]
1
>>> d[Integer(3)]
3
```

fixed_points()

Return a list of the fixed points of `self`.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).fixed_points()
[1, 4]
sage: Permutation([1,2,3,4]).fixed_points()
[1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(4)]).fixed_points()
[1, 4]
>>> Permutation([Integer(1),Integer(2),Integer(3),Integer(4)]).fixed_points()
[1, 2, 3, 4]
```

foata_bijection()

Return the image of the permutation `self` under the Foata bijection ϕ.

The bijection shows that maj (the major index) and inv (the number of inversions) are equidistributed: if $\phi(P) = Q$, then $\mathrm{maj}(P) = \mathrm{inv}(Q)$.

The Foata bijection ϕ is a bijection on the set of words with no two equal letters. It can be defined by induction on the size of the word: Given a word $w_1 w_2 \cdots w_n$, start with $\phi(w_1) = w_1$. At the i-th step, if $\phi(w_1 w_2 \cdots w_i) = v_1 v_2 \cdots v_i$, we define $\phi(w_1 w_2 \cdots w_i w_{i+1})$ by placing w_{i+1} on the end of the word $v_1 v_2 \cdots v_i$ and breaking the word up into blocks as follows. If $w_{i+1} > v_i$, place a vertical line to the right of each v_k for which $w_{i+1} > v_k$. Otherwise, if $w_{i+1} < v_i$, place a vertical line to the right of each v_k for which $w_{i+1} < v_k$. In either case, place a vertical line at the start of the word as well. Now, within each block between vertical lines, cyclically shift the entries one place to the right.

For instance, to compute $\phi([1, 4, 2, 5, 3])$, the sequence of words is

- 1,
- $|1|4 \to 14$,
- $|14|2 \to 412$,
- $|4|1|2|5 \to 4125$,
- $|4|125|3 \to 45123$.

So $\phi([1, 4, 2, 5, 3]) = [4, 5, 1, 2, 3]$.

See section 2 of [FS1978], and the proof of Proposition 1.4.6 in [EnumComb1].

See also

foata_bijection_inverse() for the inverse map.

EXAMPLES:

```
sage: Permutation([1,2,4,3]).foata_bijection()
[4, 1, 2, 3]
sage: Permutation([2,5,1,3,4]).foata_bijection()
[2, 1, 3, 5, 4]

sage: P = Permutation([2,5,1,3,4])
sage: P.major_index() == P.foata_bijection().number_of_inversions()
True

sage: all( P.major_index() == P.foata_bijection().number_of_inversions()
....:      for P in Permutations(4) )
True
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(4),Integer(3)]).foata_
↪bijection()
[4, 1, 2, 3]
>>> Permutation([Integer(2),Integer(5),Integer(1),Integer(3),Integer(4)]).
↪foata_bijection()
[2, 1, 3, 5, 4]

>>> P = Permutation([Integer(2),Integer(5),Integer(1),Integer(3),Integer(4)])
>>> P.major_index() == P.foata_bijection().number_of_inversions()
True

>>> all( P.major_index() == P.foata_bijection().number_of_inversions()
...      for P in Permutations(Integer(4)) )
True
```

The example from [FS1978]:

```
sage: Permutation([7,4,9,2,6,1,5,8,3]).foata_bijection()
[4, 7, 2, 6, 1, 9, 5, 8, 3]
```

```
>>> from sage.all import *
>>> Permutation([Integer(7),Integer(4),Integer(9),Integer(2),Integer(6),
↪Integer(1),Integer(5),Integer(8),Integer(3)]).foata_bijection()
[4, 7, 2, 6, 1, 9, 5, 8, 3]
```

Border cases:

```
sage: Permutation([]).foata_bijection()
[]
sage: Permutation([1]).foata_bijection()
[1]
```

```
>>> from sage.all import *
>>> Permutation([]).foata_bijection()
[]
>>> Permutation([Integer(1)]).foata_bijection()
[1]
```

foata_bijection_inverse()

Return the image of the permutation `self` under the inverse of the Foata bijection ϕ.

See *foata_bijection()* for the definition of the Foata bijection.

EXAMPLES:

```
sage: Permutation([4, 1, 2, 3]).foata_bijection()
[1, 2, 4, 3]
```

```
>>> from sage.all import *
>>> Permutation([Integer(4), Integer(1), Integer(2), Integer(3)]).foata_
↪bijection()
[1, 2, 4, 3]
```

forget_cycles()

Return the image of `self` under the map which forgets cycles.

Consider a permutation σ written in standard cyclic form:

$$\sigma = (a_{1,1}, \ldots, a_{1,k_1})(a_{2,1}, \ldots, a_{2,k_2}) \cdots (a_{m,1}, \ldots, a_{m,k_m}),$$

where $a_{1,1} < a_{2,1} < \cdots < a_{m,1}$ and $a_{j,1} < a_{j,i}$ for all $1 \leq j \leq m$ and $2 \leq i \leq k_j$ where we include cycles of length 1 as well. The image of the forget cycle map ϕ is given by

$$\phi(\sigma) = [a_{1,1}, \ldots, a_{1,k_1}, a_{2,1}, \ldots, a_{2,k_2}, \ldots, a_{m,1}, \ldots, a_{m,k_m}],$$

considered as a permutation in 1-line notation.

See also

fundamental_transformation(), which is a similar map that is defined by instead taking $a_{j,1} > a_{j,i}$ and is bijective.

EXAMPLES:

```
sage: P = Permutations(5)
sage: x = P([1, 5, 3, 4, 2])
sage: x.forget_cycles()
[1, 2, 5, 3, 4]
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(5))
>>> x = P([Integer(1), Integer(5), Integer(3), Integer(4), Integer(2)])
>>> x.forget_cycles()
[1, 2, 5, 3, 4]
```

We select all permutations with a cycle composition of $[2, 3, 1]$ in S_6:

```
sage: P = Permutations(6)
sage: l = [p for p in P if [len(t) for t in p.to_cycles()] == [1,3,2]]
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(6))
>>> l = [p for p in P if [len(t) for t in p.to_cycles()] == [Integer(1),
↪Integer(3),Integer(2)]]
```

Next we apply ϕ and then take the inverse, and then view the results as a poset under the Bruhat order:

```
sage: l = [p.forget_cycles().inverse() for p in l]
sage: B = Poset([l, lambda x,y: x.bruhat_lequal(y)])                          #␣
↪needs sage.combinat sage.graphs
sage: R.<q> = QQ[]
sage: sum(q^B.rank_function()(x) for x in B)                                  #␣
↪needs sage.combinat sage.graphs
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
```

```
>>> from sage.all import *
>>> l = [p.forget_cycles().inverse() for p in l]
>>> B = Poset([l, lambda x,y: x.bruhat_lequal(y)])                          #␣
↪needs sage.combinat sage.graphs
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> sum(q**B.rank_function()(x) for x in B)                                 #␣
↪needs sage.combinat sage.graphs
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
```

We check the statement in [CC2013] that the posets $C_{[1,3,1,1]}$ and $C_{[1,3,2]}$ are isomorphic:

```
sage: l2 = [p for p in P if [len(t) for t in p.to_cycles()] == [1,3,1,1]]
sage: l2 = [p.forget_cycles().inverse() for p in l2]
sage: B2 = Poset([l2, lambda x,y: x.bruhat_lequal(y)])                        #␣
↪needs sage.combinat sage.graphs
sage: B.is_isomorphic(B2)                                                     #␣
↪needs sage.combinat sage.graphs
True
```

```
>>> from sage.all import *
>>> l2 = [p for p in P if [len(t) for t in p.to_cycles()] == [Integer(1),
↪Integer(3),Integer(1),Integer(1)]]
>>> l2 = [p.forget_cycles().inverse() for p in l2]
>>> B2 = Poset([l2, lambda x,y: x.bruhat_lequal(y)])                        #␣
↪needs sage.combinat sage.graphs
>>> B.is_isomorphic(B2)                                                     #␣
↪needs sage.combinat sage.graphs
True
```

See also:

fundamental_transformation().

fundamental_transformation()

Return the image of the permutation `self` under the Renyi-Foata-Schuetzenberger fundamental transformation.

The fundamental transformation is a bijection from the set of all permutations of $\{1, 2, \ldots, n\}$ to itself, which transforms any such permutation w as follows: Write w in cycle form, with each cycle starting with its highest element, and the cycles being sorted in increasing order of their highest elements. Drop the parentheses in the resulting expression, thus reading it as a one-line notation of a new permutation u. Then, u is the image of w under the fundamental transformation.

See [EnumComb1], Proposition 1.3.1.

See also

fundamental_transformation_inverse() for the inverse map.

forget_cycles() for a similar (but non-bijective) map where each cycle is starting from its lowest element.

EXAMPLES:

```
sage: Permutation([5, 1, 3, 4, 2]).fundamental_transformation()
[3, 4, 5, 2, 1]
sage: Permutations(5)([1, 5, 3, 4, 2]).fundamental_transformation()
[1, 3, 4, 5, 2]
sage: Permutation([8, 4, 7, 2, 9, 6, 5, 1, 3]).fundamental_transformation()
[4, 2, 6, 8, 1, 9, 3, 7, 5]
```

```
>>> from sage.all import *
>>> Permutation([Integer(5), Integer(1), Integer(3), Integer(4), Integer(2)]).
↪fundamental_transformation()
[3, 4, 5, 2, 1]
>>> Permutations(Integer(5))([Integer(1), Integer(5), Integer(3), Integer(4),␣
↪Integer(2)]).fundamental_transformation()
[1, 3, 4, 5, 2]
>>> Permutation([Integer(8), Integer(4), Integer(7), Integer(2), Integer(9),␣
↪Integer(6), Integer(5), Integer(1), Integer(3)]).fundamental_
↪transformation()
[4, 2, 6, 8, 1, 9, 3, 7, 5]
```

Comparison with *forget_cycles()*:

```
sage: P = Permutation([(1,3,4),(2,5)])
sage: P
[3, 5, 4, 1, 2]
sage: P.forget_cycles()
[1, 3, 4, 2, 5]
sage: P.fundamental_transformation()
[4, 1, 3, 5, 2]
```

```
>>> from sage.all import *
>>> P = Permutation([(Integer(1),Integer(3),Integer(4)),(Integer(2),
↪Integer(5))])
```

(continues on next page)

(continued from previous page)

```
>>> P
[3, 5, 4, 1, 2]
>>> P.forget_cycles()
[1, 3, 4, 2, 5]
>>> P.fundamental_transformation()
[4, 1, 3, 5, 2]
```

fundamental_transformation_inverse()

Return the image of the permutation `self` under the inverse of the Renyi-Foata-Schuetzenberger fundamental transformation.

The inverse of the fundamental transformation is a bijection from the set of all permutations of $\{1, 2, \ldots, n\}$ to itself, which transforms any such permutation w as follows: Let $I = \{i_1 < i_2 < \cdots < i_k\}$ be the set of all left-to-right maxima of w (that is, of all indices j such that $w(j)$ is bigger than each of $w(1), w(2), \ldots, w(j-1)$). The image of w under the inverse of the fundamental transformation is the permutation u that sends $w(i-1)$ to $w(i)$ for all $i \notin I$ (notice that this makes sense, since $1 \in I$ whenever $n > 0$), while sending each $w(i_p - 1)$ (with $p \geq 2$) to $w(i_{p-1})$. Here, we set $i_{k+1} = n + 1$.

See [EnumComb1], Proposition 1.3.1.

See also

fundamental_transformation() for the inverse map.

EXAMPLES:

```
sage: Permutation([3, 4, 5, 2, 1]).fundamental_transformation_inverse()
[5, 1, 3, 4, 2]
sage: Permutation([4, 2, 6, 8, 1, 9, 3, 7, 5]).fundamental_transformation_
↪inverse()
[8, 4, 7, 2, 9, 6, 5, 1, 3]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3), Integer(4), Integer(5), Integer(2), Integer(1)]).
↪fundamental_transformation_inverse()
[5, 1, 3, 4, 2]
>>> Permutation([Integer(4), Integer(2), Integer(6), Integer(8), Integer(1),␣
↪Integer(9), Integer(3), Integer(7), Integer(5)]).fundamental_transformation_
↪inverse()
[8, 4, 7, 2, 9, 6, 5, 1, 3]
```

grade()

Return the size of `self`.

EXAMPLES:

```
sage: Permutation([3,4,1,2,5]).size()
5
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(4),Integer(1),Integer(2),Integer(5)]).
```

(continues on next page)

(continued from previous page)

```
↪size()
5
```

has_nth_root(*n*)

Check if `self` has n-th roots.

An n-th root of the permutation σ is a permutation γ such that $\gamma^n = \sigma$.

Note that the number of n-th roots only depends on the cycle type of `self`.

EXAMPLES:

```
sage: # needs sage.combinat
sage: sigma = Permutations(5).identity()
sage: sigma.has_nth_root(3)
True
sage: sigma = Permutation('(1, 3)')
sage: sigma.has_nth_root(2)
False
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> sigma = Permutations(Integer(5)).identity()
>>> sigma.has_nth_root(Integer(3))
True
>>> sigma = Permutation('(1, 3)')
>>> sigma.has_nth_root(Integer(2))
False
```

See also

- *nth_roots()*
- *number_of_nth_roots()*

has_pattern(*patt*)

Test whether the permutation `self` contains the pattern `patt`.

EXAMPLES:

```
sage: Permutation([3,5,1,4,6,2]).has_pattern([1,3,2])                          #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(5),Integer(1),Integer(4),Integer(6),
↪Integer(2)]).has_pattern([Integer(1),Integer(3),Integer(2)])                  ␣
↪       # needs sage.combinat
True
```

hyperoctahedral_double_coset_type()

Return the coset-type of `self` as a partition.

self must be a permutation of even size $2n$. The coset-type determines the double class of the permutation, that is its image in $H_n \backslash S_{2n} / H_n$, where H_n is the n-th hyperoctahedral group.

The coset-type is determined as follows. Consider the perfect matching $\{\{1, 2\}, \{3, 4\}, \ldots, \{2n-1, 2n\}\}$ and its image by self, and draw them simultaneously as edges of a graph whose vertices are labeled by $1, 2, \ldots, 2n$. The coset-type is the ordered sequence of the semi-lengths of the cycles of this graph (see Chapter VII of [Mac1995] for more details, particularly Section VII.2).

EXAMPLES:

```
sage: # needs sage.combinat
sage: p = Permutation([3, 4, 6, 1, 5, 7, 2, 8])
sage: p.hyperoctahedral_double_coset_type()
[3, 1]
sage: all(p.hyperoctahedral_double_coset_type() ==
....:     p.inverse().hyperoctahedral_double_coset_type()
....:     for p in Permutations(4))
True
sage: Permutation([]).hyperoctahedral_double_coset_type()
[]
sage: Permutation([3,1,2]).hyperoctahedral_double_coset_type()
Traceback (most recent call last):
...
ValueError: [3, 1, 2] is a permutation of odd size and has no coset-type
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> p = Permutation([Integer(3), Integer(4), Integer(6), Integer(1),
↪Integer(5), Integer(7), Integer(2), Integer(8)])
>>> p.hyperoctahedral_double_coset_type()
[3, 1]
>>> all(p.hyperoctahedral_double_coset_type() ==
...     p.inverse().hyperoctahedral_double_coset_type()
...     for p in Permutations(Integer(4)))
True
>>> Permutation([]).hyperoctahedral_double_coset_type()
[]
>>> Permutation([Integer(3),Integer(1),Integer(2)]).hyperoctahedral_double_
↪coset_type()
Traceback (most recent call last):
...
ValueError: [3, 1, 2] is a permutation of odd size and has no coset-type
```

idescents(*final_descent=False*, *from_zero=False*)

Return a list of the idescents of self, that is the list of the descents of self's inverse.

A descent of a permutation p is an integer i such that p(i) > p(i+1).

Warning: By default, the descents are returned as elements in the index set, i.e., starting at 1. If you want them to start at 0, set the keyword from_zero to True.

INPUT:

- `final_descent` – boolean (default: `False`); if `True`, the last position of a non-empty permutation is also considered as a descent
- `from_zero` – boolean (default: `False`); if `False`, return the positions starting from 1

EXAMPLES:

```
sage: Permutation([2,3,1]).idescents()
[1]
sage: Permutation([1,4,3,2]).idescents()
[2, 3]
sage: Permutation([1,4,3,2]).idescents(final_descent=True)
[2, 3, 4]
sage: Permutation([1,4,3,2]).idescents(from_zero=True)
[1, 2]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(3),Integer(1)]).idescents()
[1]
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).idescents()
[2, 3]
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).
↪idescents(final_descent=True)
[2, 3, 4]
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).idescents(from_
↪zero=True)
[1, 2]
```

idescents_signature(*final_descent=False*)

Return the list obtained as follows: Each position in `self` is mapped to -1 if it is an idescent and 1 if it is not an idescent.

See *idescents()* for a definition of idescents.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).idescents()
[2, 3]
sage: Permutation([1,4,3,2]).idescents_signature()
[1, -1, -1, 1]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).idescents()
[2, 3]
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).idescents_
↪signature()
[1, -1, -1, 1]
```

imajor_index(*final_descent=False*)

Return the inverse major index of the permutation `self`, which is the major index of the inverse of `self`.

The major index of a permutation p is the sum of the descents of p. Since our permutation indices are 0-based, we need to add the number of descents.

With the final_descent option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([2,1,3]).imajor_index()
1
sage: Permutation([3,4,1,2]).imajor_index()
2
sage: Permutation([4,3,2,1]).imajor_index()
6
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3)]).imajor_index()
1
>>> Permutation([Integer(3),Integer(4),Integer(1),Integer(2)]).imajor_index()
2
>>> Permutation([Integer(4),Integer(3),Integer(2),Integer(1)]).imajor_index()
6
```

increasing_tree(*compare=<built-in function min>*)

Return the increasing tree associated to self.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).increasing_tree()                               #␣
↪needs sage.graphs
1[., 2[3[4[., .], .], .]]
sage: Permutation([4,1,3,2]).increasing_tree()                               #␣
↪needs sage.graphs
1[4[., .], 2[3[., .], .]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).increasing_
↪tree()                                # needs sage.graphs
1[., 2[3[4[., .], .], .]]
>>> Permutation([Integer(4),Integer(1),Integer(3),Integer(2)]).increasing_
↪tree()                                # needs sage.graphs
1[4[., .], 2[3[., .], .]]
```

By passing the option compare=max one can have the decreasing tree instead:

```
sage: Permutation([2,3,4,1]).increasing_tree(max)                            #␣
↪needs sage.graphs
4[3[2[., .], .], 1[., .]]
sage: Permutation([2,3,1,4]).increasing_tree(max)                            #␣
↪needs sage.graphs
4[3[2[., .], 1[., .]], .]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(3),Integer(4),Integer(1)]).increasing_
↪tree(max)                             # needs sage.graphs
4[3[2[., .], .], 1[., .]]
>>> Permutation([Integer(2),Integer(3),Integer(1),Integer(4)]).increasing_
```

(continues on next page)

(continued from previous page)

```
↪tree(max)                              # needs sage.graphs
4[3[2[., .], 1[., .]], .]
```

increasing_tree_shape(*compare=<built-in function min>*)

Return the shape of the increasing tree associated with the permutation.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).increasing_tree_shape()                   #␣
↪needs sage.graphs
[., [[[., .], .], .]]
sage: Permutation([4,1,3,2]).increasing_tree_shape()                   #␣
↪needs sage.graphs
[[., .], [[., .], .]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).increasing_
↪tree_shape()                           # needs sage.graphs
[., [[[., .], .], .]]
>>> Permutation([Integer(4),Integer(1),Integer(3),Integer(2)]).increasing_
↪tree_shape()                           # needs sage.graphs
[[., .], [[., .], .]]
```

By passing the option `compare=max` one can have the decreasing tree instead:

```
sage: Permutation([2,3,4,1]).increasing_tree_shape(max)                #␣
↪needs sage.graphs
[[[., .], .], [., .]]
sage: Permutation([2,3,1,4]).increasing_tree_shape(max)                #␣
↪needs sage.graphs
[[[., .], [., .]], .]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(3),Integer(4),Integer(1)]).increasing_
↪tree_shape(max)                        # needs sage.graphs
[[[., .], .], [., .]]
>>> Permutation([Integer(2),Integer(3),Integer(1),Integer(4)]).increasing_
↪tree_shape(max)                        # needs sage.graphs
[[[., .], [., .]], .]
```

inverse()

Return the inverse of `self`.

EXAMPLES:

```
sage: Permutation([3,8,5,10,9,4,6,1,7,2]).inverse()
[8, 10, 1, 6, 3, 7, 9, 2, 5, 4]
sage: Permutation([2, 4, 1, 5, 3]).inverse()
[3, 1, 5, 2, 4]
sage: ~Permutation([2, 4, 1, 5, 3])
[3, 1, 5, 2, 4]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(8),Integer(5),Integer(10),Integer(9),
↪Integer(4),Integer(6),Integer(1),Integer(7),Integer(2)]).inverse()
[8, 10, 1, 6, 3, 7, 9, 2, 5, 4]
>>> Permutation([Integer(2), Integer(4), Integer(1), Integer(5), Integer(3)]).
↪inverse()
[3, 1, 5, 2, 4]
>>> ~Permutation([Integer(2), Integer(4), Integer(1), Integer(5), Integer(3)])
[3, 1, 5, 2, 4]
```

inversions()

Return a list of the inversions of `self`.

An inversion of a permutation p is a pair (i, j) such that $i < j$ and $p(i) > p(j)$.

EXAMPLES:

```
sage: Permutation([3,2,4,1,5]).inversions()
[(1, 2), (1, 4), (2, 4), (3, 4)]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(2),Integer(4),Integer(1),Integer(5)]).
↪inversions()
[(1, 2), (1, 4), (2, 4), (3, 4)]
```

is_derangement()

Return whether `self` is a derangement.

A permutation σ is a derangement if σ has no fixed points.

EXAMPLES:

```
sage: P = Permutation([1,4,2,3])
sage: P.is_derangement()
False
sage: P = Permutation([2,3,1])
sage: P.is_derangement()
True
```

```
>>> from sage.all import *
>>> P = Permutation([Integer(1),Integer(4),Integer(2),Integer(3)])
>>> P.is_derangement()
False
>>> P = Permutation([Integer(2),Integer(3),Integer(1)])
>>> P.is_derangement()
True
```

is_even()

Return `True` if the permutation `self` is even and `False` otherwise.

EXAMPLES:

```
sage: Permutation([1,2,3]).is_even()
True
```

(continues on next page)

(continued from previous page)

```
sage: Permutation([2,1,3]).is_even()
False
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(3)]).is_even()
True
>>> Permutation([Integer(2),Integer(1),Integer(3)]).is_even()
False
```

is_simple()

Return whether `self` is simple.

A permutation is simple if it does not send any proper sub-interval to a sub-interval.

For instance, `[6,1,3,5,2,4]` is not simple because it maps the interval `[3,4,5,6]` to `[2,3,4,5]`, whereas `[2,6,3,5,1,4]` is simple.

See OEIS sequence A111111

EXAMPLES:

```
sage: g = Permutation([4,2,3,1])
sage: g.is_simple()
False

sage: g = Permutation([6,1,3,5,2,4])
sage: g.is_simple()
False

sage: g = Permutation([2,6,3,5,1,4])
sage: g.is_simple()
True

sage: [len([pi for pi in Permutations(n) if pi.is_simple()])
....:  for n in range(6)]
[1, 1, 2, 0, 2, 6]
```

```
>>> from sage.all import *
>>> g = Permutation([Integer(4),Integer(2),Integer(3),Integer(1)])
>>> g.is_simple()
False

>>> g = Permutation([Integer(6),Integer(1),Integer(3),Integer(5),Integer(2),
↪Integer(4)])
>>> g.is_simple()
False

>>> g = Permutation([Integer(2),Integer(6),Integer(3),Integer(5),Integer(1),
↪Integer(4)])
>>> g.is_simple()
True

>>> [len([pi for pi in Permutations(n) if pi.is_simple()])
```

(continues on next page)

(continued from previous page)

```
...     for n in range(Integer(6))]
[1, 1, 2, 0, 2, 6]
```

ishift (*i*)

Return the i-shift of self. If an i-shift of self can't be performed, then self is returned.

An i-shift can be applied when i is not inbetween $i-1$ and $i+1$. The i-shift moves i to the other side, and leaves the relative positions of $i-1$ and $i+1$ in place. All other entries of the permutations are also left in place.

EXAMPLES:

Here, 2 is to the left of both 1 and 3. A 2-shift can be applied which moves the 2 to the right and leaves 1 and 3 in their same relative order:

```
sage: Permutation([2,1,3]).ishift(2)
[1, 3, 2]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3)]).ishift(Integer(2))
[1, 3, 2]
```

All entries other than i, $i-1$ and $i+1$ are unchanged:

```
sage: Permutation([2,4,1,3]).ishift(2)
[1, 4, 3, 2]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(4),Integer(1),Integer(3)]).
→ishift(Integer(2))
[1, 4, 3, 2]
```

Since 2 is between 1 and 3 in [1,2,3], a 2-shift cannot be applied to [1,2,3]

```
sage: Permutation([1,2,3]).ishift(2)
[1, 2, 3]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(3)]).ishift(Integer(2))
[1, 2, 3]
```

iswitch (*i*)

Return the i-switch of self. If an i-switch of self can't be performed, then self is returned.

An i-switch can be applied when the subsequence of self formed by the entries $i-1$, i and $i+1$ is neither increasing nor decreasing. In this case, this subsequence is reversed (i. e., its leftmost element and its rightmost element switch places), while all other letters of self are kept in place.

EXAMPLES:

Here, 2 is to the left of both 1 and 3. A 2-switch can be applied which moves the 2 to the right and switches the relative order between 1 and 3:

```
sage: Permutation([2,1,3]).iswitch(2)
[3, 1, 2]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3)]).iswitch(Integer(2))
[3, 1, 2]
```

All entries other than $i-1$, i and $i+1$ are unchanged:

```
sage: Permutation([2,4,1,3]).iswitch(2)
[3, 4, 1, 2]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(4),Integer(1),Integer(3)]).
↪iswitch(Integer(2))
[3, 4, 1, 2]
```

Since 2 is between 1 and 3 in `[1,2,3]`, a 2-switch cannot be applied to `[1,2,3]`

```
sage: Permutation([1,2,3]).iswitch(2)
[1, 2, 3]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(3)]).iswitch(Integer(2))
[1, 2, 3]
```

left_action_product(*lp*)

Return the permutation obtained by composing `self` with `lp` in such an order that `lp` is applied first and `self` is applied afterwards.

This is usually denoted by either `self * lp` or `lp * self` depending on the conventions used by the author. If the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by $p(i)$, then this should be denoted by `self * lp` in order to have associativity (i.e., in order to have $(p \cdot q)(i) = p(q(i))$ for all p, q and i). If, on the other hand, the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by i^p, then this should be denoted by `lp * self` in order to have associativity (i.e., in order to have $i^{p \cdot q} = (i^p)^q$ for all p, q and i).

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: q = Permutation([3,1,2])
sage: p.left_action_product(q)
[3, 2, 1]
sage: q.left_action_product(p)
[1, 3, 2]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2),Integer(1),Integer(3)])
>>> q = Permutation([Integer(3),Integer(1),Integer(2)])
>>> p.left_action_product(q)
[3, 2, 1]
>>> q.left_action_product(p)
[1, 3, 2]
```

left_tableau()

Return the left standard tableau after performing the RSK algorithm on `self`.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).left_tableau()                                    #␣
↪needs sage.combinat
[[1, 2], [3], [4]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).left_tableau()␣
↪                               # needs sage.combinat
[[1, 2], [3], [4]]
```

length()

Return the Coxeter length of `self`.

The length of a permutation p is given by the number of inversions of p.

EXAMPLES:

```
sage: Permutation([5, 1, 3, 4, 2]).length()
6
```

```
>>> from sage.all import *
>>> Permutation([Integer(5), Integer(1), Integer(3), Integer(4), Integer(2)]).
↪length()
6
```

longest_increasing_subsequence_length()

Return the length of the longest increasing subsequences of `self`.

EXAMPLES:

```
sage: Permutation([2,3,1,4]).longest_increasing_subsequence_length()
3
sage: all(i.longest_increasing_subsequence_length() == len(RSK(i)[0][0])     #␣
↪needs sage.combinat
....:     for i in Permutations(5))
True
sage: Permutation([]).longest_increasing_subsequence_length()
0
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(3),Integer(1),Integer(4)]).longest_
↪increasing_subsequence_length()
3
>>> all(i.longest_increasing_subsequence_length() ==␣
↪len(RSK(i)[Integer(0)][Integer(0)])     # needs sage.combinat
...     for i in Permutations(Integer(5)))
True
>>> Permutation([]).longest_increasing_subsequence_length()
0
```

longest_increasing_subsequences()

Return the list of the longest increasing subsequences of `self`.

A theorem of Schensted ([Sch1961]) states that an increasing subsequence of length i ends with the value entered in the i-th column of the p-tableau. The algorithm records which column of the p-tableau each value of the permutation is entered into, creates a digraph to record all increasing subsequences, and reads the paths from a source to a sink; these are the longest increasing subsequences.

EXAMPLES:

```
sage: Permutation([2,3,4,1]).longest_increasing_subsequences()              #␣
↪needs sage.graphs
[[2, 3, 4]]
sage: Permutation([5, 7, 1, 2, 6, 4, 3]).longest_increasing_subsequences()  #␣
↪needs sage.graphs
[[1, 2, 6], [1, 2, 4], [1, 2, 3]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(3),Integer(4),Integer(1)]).longest_
↪increasing_subsequences()              # needs sage.graphs
[[2, 3, 4]]
>>> Permutation([Integer(5), Integer(7), Integer(1), Integer(2), Integer(6),␣
↪Integer(4), Integer(3)]).longest_increasing_subsequences()  # needs sage.
↪graphs
[[1, 2, 6], [1, 2, 4], [1, 2, 3]]
```

Note

This algorithm could be made faster using a balanced search tree for each column instead of sorted lists. See discussion on Issue #31451.

longest_increasing_subsequences_number()

Return the number of increasing subsequences of maximal length in `self`.

The list of longest increasing subsequences of a permutation is given by *longest_increasing_subsequences()*, and the length of these subsequences is given by *longest_increasing_subsequence_length()*.

The algorithm is similar to *longest_increasing_subsequences()*. Namely, the longest increasing subsequences are encoded as increasing sequences in a ranked poset from a smallest to a largest element. Their number can be obtained via dynamic programming: for each v in the poset we compute the number of paths from a smallest element to v.

EXAMPLES:

```
sage: sum(p.longest_increasing_subsequences_number()
....:     for p in Permutations(8))
120770

sage: p = Permutations(50).random_element()
sage: (len(p.longest_increasing_subsequences()) ==                          #␣
↪needs sage.graphs
....:  p.longest_increasing_subsequences_number())
True
```

```
>>> from sage.all import *
>>> sum(p.longest_increasing_subsequences_number()
```

(continues on next page)

(continued from previous page)

```
...     for p in Permutations(Integer(8)))
120770

>>> p = Permutations(Integer(50)).random_element()
>>> (len(p.longest_increasing_subsequences()) ==                    #␣
↪needs sage.graphs
...  p.longest_increasing_subsequences_number())
True
```

major_index(*final_descent=False*)

Return the major index of `self`.

The major index of a permutation p is the sum of the descents of p. Since our permutation indices are 0-based, we need to add the number of descents.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([2,1,3]).major_index()
1
sage: Permutation([3,4,1,2]).major_index()
2
sage: Permutation([4,3,2,1]).major_index()
6
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3)]).major_index()
1
>>> Permutation([Integer(3),Integer(4),Integer(1),Integer(2)]).major_index()
2
>>> Permutation([Integer(4),Integer(3),Integer(2),Integer(1)]).major_index()
6
```

multi_major_index(*composition*)

Return the multimajor index of this permutation with respect to `composition`.

INPUT:

- `composition` – a composition of the *size()* of this permutation

EXAMPLES:

```
sage: p = Permutation([5, 6, 2, 1, 3, 7, 4])
sage: p.multi_major_index([3, 2, 2])
[2, 0, 1]
sage: p.multi_major_index([7]) == [p.major_index()]
True
sage: p.multi_major_index([1]*7)
[0, 0, 0, 0, 0, 0, 0]
sage: Permutation([]).multi_major_index([])
[]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(5), Integer(6), Integer(2), Integer(1),␣
↪Integer(3), Integer(7), Integer(4)])
>>> p.multi_major_index([Integer(3), Integer(2), Integer(2)])
[2, 0, 1]
>>> p.multi_major_index([Integer(7)]) == [p.major_index()]
True
>>> p.multi_major_index([Integer(1)]*Integer(7))
[0, 0, 0, 0, 0, 0, 0]
>>> Permutation([]).multi_major_index([])
[]
```

REFERENCES:

- [JS2000]

next()

Return the permutation that follows `self` on the symmetric group containing `self`. If `self` is the last permutation, then `next` returns `False`. If the `algorithm` parameter is specified, the permutations will be generated according to it. Supported algorithms are:

- `lex`: lexicographic order generation, this is the default algorithm.
- `sjt`: Steinhaus-Johnson-Trotter algorithm to generate permutations using only transposition of two elements in the list. It is highly recommended to set `check=True` (default value).

EXAMPLES:

```
sage: p = Permutation([1, 3, 2])
sage: next(p)
[2, 1, 3]
sage: p = Permutation([4,3,2,1])
sage: next(p)
False
sage: p = Permutation([1, 2, 3], algorithm='sjt')
sage: p = next(p); p
[1, 3, 2]
sage: p = next(p); p
[3, 1, 2]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(1), Integer(3), Integer(2)])
>>> next(p)
[2, 1, 3]
>>> p = Permutation([Integer(4),Integer(3),Integer(2),Integer(1)])
>>> next(p)
False
>>> p = Permutation([Integer(1), Integer(2), Integer(3)], algorithm='sjt')
>>> p = next(p); p
[1, 3, 2]
>>> p = next(p); p
[3, 1, 2]
```

noninversions(k)

Return the list of all `k`-noninversions in `self`.

If k is an integer and $p \in S_n$ is a permutation, then a k-noninversion in p is defined as a strictly increasing sequence $(i_1, i_2, \ldots, i_k)$ of elements of $\{1, 2, \ldots, n\}$ satisfying $p(i_1) < p(i_2) < \cdots < p(i_k)$. (In other words, a k-noninversion in p can be regarded as a k-element subset of $\{1, 2, \ldots, n\}$ on which p restricts to an increasing map.)

EXAMPLES:

```
sage: p = Permutation([3, 2, 4, 1, 5])
sage: p.noninversions(1)
[[3], [2], [4], [1], [5]]
sage: p.noninversions(2)
[[3, 4], [3, 5], [2, 4], [2, 5], [4, 5], [1, 5]]
sage: p.noninversions(3)
[[3, 4, 5], [2, 4, 5]]
sage: p.noninversions(4)
[]
sage: p.noninversions(5)
[]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3), Integer(2), Integer(4), Integer(1),␣
↪Integer(5)])
>>> p.noninversions(Integer(1))
[[3], [2], [4], [1], [5]]
>>> p.noninversions(Integer(2))
[[3, 4], [3, 5], [2, 4], [2, 5], [4, 5], [1, 5]]
>>> p.noninversions(Integer(3))
[[3, 4, 5], [2, 4, 5]]
>>> p.noninversions(Integer(4))
[]
>>> p.noninversions(Integer(5))
[]
```

nth_roots(*n*)

Return all n-th roots of `self` (as a generator).

An n-th root of the permutation σ is a permutation γ such that $\gamma^n = \sigma$.

Note that the number of n-th roots only depends on the cycle type of `self`.

EXAMPLES:

```
sage: # needs sage.combinat
sage: sigma = Permutations(5).identity()
sage: list(sigma.nth_roots(3))
[[1, 4, 3, 5, 2], [1, 5, 3, 2, 4], [1, 2, 4, 5, 3], [1, 2, 5, 3, 4],
 [4, 2, 3, 5, 1], [5, 2, 3, 1, 4], [3, 2, 5, 4, 1], [5, 2, 1, 4, 3],
 [2, 5, 3, 4, 1], [5, 1, 3, 4, 2], [2, 3, 1, 4, 5], [3, 1, 2, 4, 5],
 [2, 4, 3, 1, 5], [4, 1, 3, 2, 5], [3, 2, 4, 1, 5], [4, 2, 1, 3, 5],
 [1, 3, 4, 2, 5], [1, 4, 2, 3, 5], [1, 3, 5, 4, 2], [1, 5, 2, 4, 3],
 [1, 2, 3, 4, 5]]
sage: sigma = Permutation('(1, 3)')
sage: list(sigma.nth_roots(2))
[]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> sigma = Permutations(Integer(5)).identity()
>>> list(sigma.nth_roots(Integer(3)))
[[1, 4, 3, 5, 2], [1, 5, 3, 2, 4], [1, 2, 4, 5, 3], [1, 2, 5, 3, 4],
 [4, 2, 3, 5, 1], [5, 2, 3, 1, 4], [3, 2, 5, 4, 1], [5, 2, 1, 4, 3],
 [2, 5, 3, 4, 1], [5, 1, 3, 4, 2], [2, 3, 1, 4, 5], [3, 1, 2, 4, 5],
 [2, 4, 3, 1, 5], [4, 1, 3, 2, 5], [3, 2, 4, 1, 5], [4, 2, 1, 3, 5],
 [1, 3, 4, 2, 5], [1, 4, 2, 3, 5], [1, 3, 5, 4, 2], [1, 5, 2, 4, 3],
 [1, 2, 3, 4, 5]]
>>> sigma = Permutation('(1, 3)')
>>> list(sigma.nth_roots(Integer(2)))
[]
```

For $n \geq 6$, this algorithm begins to be more efficient than naive search (look at all permutations and test their n-th power).

See also

- *has_nth_root()*
- *number_of_nth_roots()*

number_of_descents(*final_descent=False*)

Return the number of descents of `self`.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_descents()
2
sage: Permutation([1,4,3,2]).number_of_descents(final_descent=True)
3
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).number_of_
↪descents()
2
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).number_of_
↪descents(final_descent=True)
3
```

number_of_fixed_points()

Return the number of fixed points of `self`.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).number_of_fixed_points()
2
sage: Permutation([1,2,3,4]).number_of_fixed_points()
4
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(4)]).number_of_
↪fixed_points()
2
>>> Permutation([Integer(1),Integer(2),Integer(3),Integer(4)]).number_of_
↪fixed_points()
4
```

number_of_idescents(*final_descent=False*)

Return the number of idescents of `self`.

See `idescents()` for a definition of idescents.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_idescents()
2
sage: Permutation([1,4,3,2]).number_of_idescents(final_descent=True)
3
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).number_of_
↪idescents()
2
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).number_of_
↪idescents(final_descent=True)
3
```

number_of_inversions()

Return the number of inversions in `self`.

An inversion of a permutation is a pair of elements (i, j) with $i < j$ and $p(i) > p(j)$.

REFERENCES:

- http://mathworld.wolfram.com/PermutationInversion.html

EXAMPLES:

```
sage: Permutation([3, 2, 4, 1, 5]).number_of_inversions()
4
sage: Permutation([1, 2, 6, 4, 7, 3, 5]).number_of_inversions()
6
```

```
>>> from sage.all import *
>>> Permutation([Integer(3), Integer(2), Integer(4), Integer(1), Integer(5)]).
↪number_of_inversions()
4
>>> Permutation([Integer(1), Integer(2), Integer(6), Integer(4), Integer(7),␣
↪Integer(3), Integer(5)]).number_of_inversions()
6
```

number_of_noninversions(*k*)

Return the number of `k`-noninversions in `self`.

If k is an integer and $p \in S_n$ is a permutation, then a k-noninversion in p is defined as a strictly increasing sequence $(i_1, i_2, \ldots, i_k)$ of elements of $\{1, 2, \ldots, n\}$ satisfying $p(i_1) < p(i_2) < \cdots < p(i_k)$. (In other words, a k-noninversion in p can be regarded as a k-element subset of $\{1, 2, \ldots, n\}$ on which p restricts to an increasing map.)

The number of k-noninversions in p has been denoted by $\mathrm{noninv}_k(p)$ in [RSW2011], where conjectures and results regarding this number have been stated.

EXAMPLES:

```
sage: p = Permutation([3, 2, 4, 1, 5])
sage: p.number_of_noninversions(1)
5
sage: p.number_of_noninversions(2)
6
sage: p.number_of_noninversions(3)
2
sage: p.number_of_noninversions(4)
0
sage: p.number_of_noninversions(5)
0
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3), Integer(2), Integer(4), Integer(1),␣
↪Integer(5)])
>>> p.number_of_noninversions(Integer(1))
5
>>> p.number_of_noninversions(Integer(2))
6
>>> p.number_of_noninversions(Integer(3))
2
>>> p.number_of_noninversions(Integer(4))
0
>>> p.number_of_noninversions(Integer(5))
0
```

The number of 2-noninversions of a permutation $p \in S_n$ is $\binom{n}{2}$ minus its number of inversions:

```
sage: b = binomial(5, 2)                                                    #␣
↪needs sage.symbolic
sage: all( x.number_of_noninversions(2) == b - x.number_of_inversions()     #␣
↪needs sage.symbolic
....:      for x in Permutations(5) )
True
```

```
>>> from sage.all import *
>>> b = binomial(Integer(5), Integer(2))                                     ␣
↪              # needs sage.symbolic
>>> all( x.number_of_noninversions(Integer(2)) == b - x.number_of_
↪inversions()     # needs sage.symbolic
...      for x in Permutations(Integer(5)) )
True
```

We also check some corner cases:

```
sage: all( x.number_of_noninversions(1) == 5 for x in Permutations(5) )
True
sage: all( x.number_of_noninversions(0) == 1 for x in Permutations(5) )
True
sage: Permutation([]).number_of_noninversions(1)
0
sage: Permutation([]).number_of_noninversions(0)
1
sage: Permutation([2, 1]).number_of_noninversions(3)
0
```

```
>>> from sage.all import *
>>> all( x.number_of_noninversions(Integer(1)) == Integer(5) for x in␣
↪Permutations(Integer(5)) )
True
>>> all( x.number_of_noninversions(Integer(0)) == Integer(1) for x in␣
↪Permutations(Integer(5)) )
True
>>> Permutation([]).number_of_noninversions(Integer(1))
0
>>> Permutation([]).number_of_noninversions(Integer(0))
1
>>> Permutation([Integer(2), Integer(1)]).number_of_noninversions(Integer(3))
0
```

number_of_nth_roots(n)

Return the number of n-th roots of `self`.

An n-th root of the permutation σ is a permutation γ such that $\gamma^n = \sigma$.

Note that the number of n-th roots only depends on the cycle type of `self`.

EXAMPLES:

```
sage: # needs sage.combinat
sage: Sigma = Permutations(5).identity()
sage: Sigma.number_of_nth_roots(3)
21
sage: Sigma = Permutation('(1, 3)')
sage: Sigma.number_of_nth_roots(2)
0
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> Sigma = Permutations(Integer(5)).identity()
>>> Sigma.number_of_nth_roots(Integer(3))
21
>>> Sigma = Permutation('(1, 3)')
>>> Sigma.number_of_nth_roots(Integer(2))
0
```

See also

- `nth_roots()`
- `has_nth_root()`

number_of_peaks()

Return the number of peaks of the permutation `self`.

A peak of a permutation p is an integer i such that $p(i-1) < p(i)$ and $p(i) > p(i+1)$.

EXAMPLES:

```
sage: Permutation([1,3,2,4,5]).number_of_peaks()
1
sage: Permutation([4,1,3,2,6,5]).number_of_peaks()
2
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(4),Integer(5)]).
↪number_of_peaks()
1
>>> Permutation([Integer(4),Integer(1),Integer(3),Integer(2),Integer(6),
↪Integer(5)]).number_of_peaks()
2
```

number_of_recoils()

Return the number of recoils of the permutation `self`.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_recoils()
2
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).number_of_
↪recoils()
2
```

number_of_reduced_words()

Return the number of reduced words of `self` without explicitly computing them all.

EXAMPLES:

```
sage: p = Permutation([6,4,2,5,1,8,3,7])
sage: len(p.reduced_words()) == p.number_of_reduced_words()                #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(6),Integer(4),Integer(2),Integer(5),Integer(1),
↪Integer(8),Integer(3),Integer(7)])
>>> len(p.reduced_words()) == p.number_of_reduced_words()                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
True
```

number_of_saliances()

Return the number of saliances of `self`.

A saliance of a permutation p is an integer i such that $p(i) > p(j)$ for all $j > i$.

EXAMPLES:

```
sage: Permutation([2,3,1,5,4]).number_of_saliances()
2
sage: Permutation([5,4,3,2,1]).number_of_saliances()
5
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(3),Integer(1),Integer(5),Integer(4)]).
↪number_of_saliances()
2
>>> Permutation([Integer(5),Integer(4),Integer(3),Integer(2),Integer(1)]).
↪number_of_saliances()
5
```

order()

Return the order of `self`.

EXAMPLES:

```
sage: sigma = Permutation([3,4,1,2,5])
sage: sigma.order()
2
sage: sigma * sigma
[1, 2, 3, 4, 5]
```

```
>>> from sage.all import *
>>> sigma = Permutation([Integer(3),Integer(4),Integer(1),Integer(2),
↪Integer(5)])
>>> sigma.order()
2
>>> sigma * sigma
[1, 2, 3, 4, 5]
```

pattern_positions(*patt*)

Return the list of positions where the pattern `patt` appears in the permutation `self`.

EXAMPLES:

```
sage: Permutation([3,5,1,4,6,2]).pattern_positions([1,3,2])                #␣
↪needs sage.combinat
[[0, 1, 3], [2, 3, 5], [2, 4, 5]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(5),Integer(1),Integer(4),Integer(6),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)]).pattern_positions([Integer(1),Integer(3),Integer(2)])              ␣
↪        # needs sage.combinat
[[0, 1, 3], [2, 3, 5], [2, 4, 5]]
```

peaks()

Return a list of the peaks of the permutation `self`.

A peak of a permutation p is an integer i such that $p(i-1) < p(i)$ and $p(i) > p(i+1)$.

EXAMPLES:

```
sage: Permutation([1,3,2,4,5]).peaks()
[1]
sage: Permutation([4,1,3,2,6,5]).peaks()
[2, 4]
sage: Permutation([]).peaks()
[]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(4),Integer(5)]).
↪peaks()
[1]
>>> Permutation([Integer(4),Integer(1),Integer(3),Integer(2),Integer(6),
↪Integer(5)]).peaks()
[2, 4]
>>> Permutation([]).peaks()
[]
```

permutation_poset()

Return the permutation poset of `self`.

The permutation poset of a permutation p is the poset with vertices $(i, p(i))$ for $i = 1, 2, \ldots, n$ (where n is the size of p) and order inherited from $\mathbf{Z} \times \mathbf{Z}$.

EXAMPLES:

```
sage: # needs sage.combinat sage.graphs
sage: Permutation([3,1,5,4,2]).permutation_poset().cover_relations()
[[(2, 1), (5, 2)],
 [(2, 1), (3, 5)],
 [(2, 1), (4, 4)],
 [(1, 3), (3, 5)],
 [(1, 3), (4, 4)]]
sage: Permutation([]).permutation_poset().cover_relations()
[]
sage: Permutation([1,3,2]).permutation_poset().cover_relations()
[[(1, 1), (2, 3)], [(1, 1), (3, 2)]]
sage: Permutation([1,2]).permutation_poset().cover_relations()
[[(1, 1), (2, 2)]]
sage: P = Permutation([1,5,2,4,3])
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.graphs
```

(continues on next page)

(continued from previous page)

```
>>> Permutation([Integer(3),Integer(1),Integer(5),Integer(4),Integer(2)]).
→permutation_poset().cover_relations()
[[(2, 1), (5, 2)],
 [(2, 1), (3, 5)],
 [(2, 1), (4, 4)],
 [(1, 3), (3, 5)],
 [(1, 3), (4, 4)]]
>>> Permutation([]).permutation_poset().cover_relations()
[]
>>> Permutation([Integer(1),Integer(3),Integer(2)]).permutation_poset().cover_
→relations()
[[(1, 1), (2, 3)], [(1, 1), (3, 2)]]
>>> Permutation([Integer(1),Integer(2)]).permutation_poset().cover_relations()
[[(1, 1), (2, 2)]]
>>> P = Permutation([Integer(1),Integer(5),Integer(2),Integer(4),Integer(3)])
```

This should hold for any P:

```
sage: P.permutation_poset().greene_shape() == P.RS_partition()              #␣
→needs sage.combinat sage.graphs
True
```

```
>>> from sage.all import *
>>> P.permutation_poset().greene_shape() == P.RS_partition()              #␣
→needs sage.combinat sage.graphs
True
```

permutohedron_greater(*side='right'*)

Return a list of permutations greater than or equal to `self` in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

See *permutohedron_lequal()* for the definition of the permutohedron orders.

EXAMPLES:

```
sage: Permutation([4,2,1,3]).permutohedron_greater()
[[4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1]]
sage: Permutation([4,2,1,3]).permutohedron_greater(side='left')
[[4, 2, 1, 3], [4, 3, 1, 2], [4, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(2),Integer(1),Integer(3)]).permutohedron_
→greater()
[[4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1]]
>>> Permutation([Integer(4),Integer(2),Integer(1),Integer(3)]).permutohedron_
→greater(side='left')
[[4, 2, 1, 3], [4, 3, 1, 2], [4, 3, 2, 1]]
```

permutohedron_join(*other*, *side='right'*)

Return the join of the permutations `self` and `other` in the right permutohedron order (or, if `side` is set to `'left'`, in the left permutohedron order).

The permutohedron orders (see `permutohedron_lequal()`) are lattices; the join operation refers to this lattice structure. In more elementary terms, the join of two permutations π and ψ in the symmetric group S_n is the permutation in S_n whose set of inversion is the transitive closure of the union of the set of inversions of π with the set of inversions of ψ.

See also

`permutohedron_lequal()`, `permutohedron_meet()`.

ALGORITHM:

It is enough to construct the join of any two permutations π and ψ in S_n with respect to the right weak order. (The join of π and ψ with respect to the left weak order is the inverse of the join of π^{-1} and ψ^{-1} with respect to the right weak order.) Start with an empty list l (denoted `xs` in the actual code). For $i = 1, 2, \ldots, n$ (in this order), we insert i into this list in the rightmost possible position such that any letter in $\{1, 2, ..., i-1\}$ which appears further right than i in either π or ψ (or both) must appear further right than i in the resulting list. After all numbers are inserted, we are left with a list which is precisely the join of π and ψ (in one-line notation). This algorithm is due to Markowsky, [Mar1994] (Theorem 1 (a)).

AUTHORS:

Viviane Pons and Darij Grinberg, 18 June 2014.

EXAMPLES:

```
sage: p = Permutation([3,1,2])
sage: q = Permutation([1,3,2])
sage: p.permutohedron_join(q)
[3, 1, 2]
sage: r = Permutation([2,1,3])
sage: r.permutohedron_join(p)
[3, 2, 1]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3),Integer(1),Integer(2)])
>>> q = Permutation([Integer(1),Integer(3),Integer(2)])
>>> p.permutohedron_join(q)
[3, 1, 2]
>>> r = Permutation([Integer(2),Integer(1),Integer(3)])
>>> r.permutohedron_join(p)
[3, 2, 1]
```

```
sage: p = Permutation([3,2,4,1])
sage: q = Permutation([4,2,1,3])
sage: p.permutohedron_join(q)
[4, 3, 2, 1]
sage: r = Permutation([3,1,2,4])
sage: p.permutohedron_join(r)
[3, 2, 4, 1]
sage: q.permutohedron_join(r)
[4, 3, 2, 1]
sage: s = Permutation([1,4,2,3])
sage: s.permutohedron_join(r)
[4, 3, 1, 2]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3),Integer(2),Integer(4),Integer(1)])
>>> q = Permutation([Integer(4),Integer(2),Integer(1),Integer(3)])
>>> p.permutohedron_join(q)
[4, 3, 2, 1]
>>> r = Permutation([Integer(3),Integer(1),Integer(2),Integer(4)])
>>> p.permutohedron_join(r)
[3, 2, 4, 1]
>>> q.permutohedron_join(r)
[4, 3, 2, 1]
>>> s = Permutation([Integer(1),Integer(4),Integer(2),Integer(3)])
>>> s.permutohedron_join(r)
[4, 3, 1, 2]
```

The universal property of the join operation is satisfied:

```
sage: def test_uni_join(p, q):
....:     j = p.permutohedron_join(q)
....:     if not p.permutohedron_lequal(j):
....:         return False
....:     if not q.permutohedron_lequal(j):
....:         return False
....:     for r in p.permutohedron_greater():
....:         if q.permutohedron_lequal(r) and not j.permutohedron_lequal(r):
....:             return False
....:     return True
sage: all( test_uni_join(p, q) for p in Permutations(3) for q in␣
↪Permutations(3) )
True
sage: test_uni_join(Permutation([6, 4, 7, 3, 2, 5, 8, 1]), Permutation([7, 3,␣
↪1, 2, 5, 4, 6, 8]))
True
```

```
>>> from sage.all import *
>>> def test_uni_join(p, q):
...     j = p.permutohedron_join(q)
...     if not p.permutohedron_lequal(j):
...         return False
...     if not q.permutohedron_lequal(j):
...         return False
...     for r in p.permutohedron_greater():
...         if q.permutohedron_lequal(r) and not j.permutohedron_lequal(r):
...             return False
...     return True
>>> all( test_uni_join(p, q) for p in Permutations(Integer(3)) for q in␣
↪Permutations(Integer(3)) )
True
>>> test_uni_join(Permutation([Integer(6), Integer(4), Integer(7), Integer(3),
↪ Integer(2), Integer(5), Integer(8), Integer(1)]), Permutation([Integer(7),␣
↪Integer(3), Integer(1), Integer(2), Integer(5), Integer(4), Integer(6),␣
↪Integer(8)]))
True
```

Border cases:

```
sage: p = Permutation([])
sage: p.permutohedron_join(p)
[]
sage: p = Permutation([1])
sage: p.permutohedron_join(p)
[1]
```

```
>>> from sage.all import *
>>> p = Permutation([])
>>> p.permutohedron_join(p)
[]
>>> p = Permutation([Integer(1)])
>>> p.permutohedron_join(p)
[1]
```

The left permutohedron:

```
sage: p = Permutation([3,1,2])
sage: q = Permutation([1,3,2])
sage: p.permutohedron_join(q, side='left')
[3, 2, 1]
sage: r = Permutation([2,1,3])
sage: r.permutohedron_join(p, side='left')
[3, 1, 2]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3),Integer(1),Integer(2)])
>>> q = Permutation([Integer(1),Integer(3),Integer(2)])
>>> p.permutohedron_join(q, side='left')
[3, 2, 1]
>>> r = Permutation([Integer(2),Integer(1),Integer(3)])
>>> r.permutohedron_join(p, side='left')
[3, 1, 2]
```

permutohedron_lequal(*p2*, *side='right'*)

Return `True` if `self` is less or equal to `p2` in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

For every nonnegative integer n, the right (resp. left) permutohedron order (also called the right (resp. left) weak order, or the right (resp. left) weak Bruhat order) is a partial order on the symmetric group S_n. It can be defined in various ways, including the following ones:

- Two permutations u and v in S_n satisfy $u \leq v$ in the right (resp. left) permutohedron order if and only if the (Coxeter) length of the permutation $v^{-1} \circ u$ (resp. of the permutation $u \circ v^{-1}$) equals the length of v minus the length of u. Here, $p \circ q$ means the permutation obtained by applying q first and then p. (Recall that the Coxeter length of a permutation is its number of inversions.)
- Two permutations u and v in S_n satisfy $u \leq v$ in the right (resp. left) permutohedron order if and only if every pair (i, j) of elements of $\{1, 2, \cdots, n\}$ such that $i < j$ and $u^{-1}(i) > u^{-1}(j)$ (resp. $u(i) > u(j)$) also satisfies $v^{-1}(i) > v^{-1}(j)$ (resp. $v(i) > v(j)$).
- A permutation $v \in S_n$ covers a permutation $u \in S_n$ in the right (resp. left) permutohedron order if and

only if we have $v = u \circ (i, i+1)$ (resp. $v = (i, i+1) \circ u$) for some $i \in \{1, 2, \cdots, n-1\}$ satisfying $u(i) < u(i+1)$ (resp. $u^{-1}(i) < u^{-1}(i+1)$). Here, again, $p \circ q$ means the permutation obtained by applying q first and then p.

The right and the left permutohedron order are mutually isomorphic, with the isomorphism being the map sending every permutation to its inverse. Each of these orders endows the symmetric group S_n with the structure of a graded poset (the rank function being the Coxeter length).

Warning

The permutohedron order is not to be mistaken for the strong Bruhat order (`bruhat_lequal()`), despite both orders being occasionally referred to as the Bruhat order.

EXAMPLES:

```
sage: p = Permutation([3,2,1,4])
sage: p.permutohedron_lequal(Permutation([4,2,1,3]))
False
sage: p.permutohedron_lequal(Permutation([4,2,1,3]), side='left')
True
sage: p.permutohedron_lequal(p)
True

sage: Permutation([2,1,3]).permutohedron_lequal(Permutation([2,3,1]))
True
sage: Permutation([2,1,3]).permutohedron_lequal(Permutation([3,1,2]))
False
sage: Permutation([2,1,3]).permutohedron_lequal(Permutation([1,2,3]))
False
sage: Permutation([1,3,2]).permutohedron_lequal(Permutation([2,1,3]))
False
sage: Permutation([1,3,2]).permutohedron_lequal(Permutation([2,3,1]))
False
sage: Permutation([2,3,1]).permutohedron_lequal(Permutation([1,3,2]))
False
sage: Permutation([2,1,3]).permutohedron_lequal(Permutation([2,3,1]), side=
↪'left')
False
sage: sorted( [len([b for b in Permutations(3) if a.permutohedron_lequal(b)])
....:              for a in Permutations(3)] )
[1, 2, 2, 3, 3, 6]
sage: sorted( [len([b for b in Permutations(3) if a.permutohedron_lequal(b,␣
↪side='left')])
....:              for a in Permutations(3)] )
[1, 2, 2, 3, 3, 6]

sage: Permutation([]).permutohedron_lequal(Permutation([]))
True
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3),Integer(2),Integer(1),Integer(4)])
>>> p.permutohedron_lequal(Permutation([Integer(4),Integer(2),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)]))
False
>>> p.permutohedron_lequal(Permutation([Integer(4),Integer(2),Integer(1),
↪Integer(3)]), side='left')
True
>>> p.permutohedron_lequal(p)
True

>>> Permutation([Integer(2),Integer(1),Integer(3)]).permutohedron_
↪lequal(Permutation([Integer(2),Integer(3),Integer(1)]))
True
>>> Permutation([Integer(2),Integer(1),Integer(3)]).permutohedron_
↪lequal(Permutation([Integer(3),Integer(1),Integer(2)]))
False
>>> Permutation([Integer(2),Integer(1),Integer(3)]).permutohedron_
↪lequal(Permutation([Integer(1),Integer(2),Integer(3)]))
False
>>> Permutation([Integer(1),Integer(3),Integer(2)]).permutohedron_
↪lequal(Permutation([Integer(2),Integer(1),Integer(3)]))
False
>>> Permutation([Integer(1),Integer(3),Integer(2)]).permutohedron_
↪lequal(Permutation([Integer(2),Integer(3),Integer(1)]))
False
>>> Permutation([Integer(2),Integer(3),Integer(1)]).permutohedron_
↪lequal(Permutation([Integer(1),Integer(3),Integer(2)]))
False
>>> Permutation([Integer(2),Integer(1),Integer(3)]).permutohedron_
↪lequal(Permutation([Integer(2),Integer(3),Integer(1)]), side='left')
False
>>> sorted( [len([b for b in Permutations(Integer(3)) if a.permutohedron_
↪lequal(b)])
...           for a in Permutations(Integer(3))] )
[1, 2, 2, 3, 3, 6]
>>> sorted( [len([b for b in Permutations(Integer(3)) if a.permutohedron_
↪lequal(b, side='left')])
...           for a in Permutations(Integer(3))] )
[1, 2, 2, 3, 3, 6]

>>> Permutation([]).permutohedron_lequal(Permutation([]))
True
```

permutohedron_meet(*other*, *side='right'*)

Return the meet of the permutations `self` and `other` in the right permutohedron order (or, if `side` is set to `'left'`, in the left permutohedron order).

The permutohedron orders (see *permutohedron_lequal()*) are lattices; the meet operation refers to this lattice structure. It is connected to the join operation by the following simple symmetry property: If π and ψ are two permutations π and ψ in the symmetric group S_n, and if w_0 denotes the permutation $(n, n-1, \ldots, 1) \in S_n$, then

$$\pi \wedge \psi = w_0 \circ ((w_0 \circ \pi) \vee (w_0 \circ \psi)) = ((\pi \circ w_0) \vee (\psi \circ w_0)) \circ w_0$$

and

$$\pi \vee \psi = w_0 \circ ((w_0 \circ \pi) \wedge (w_0 \circ \psi)) = ((\pi \circ w_0) \wedge (\psi \circ w_0)) \circ w_0,$$

where $\wedge$ means meet and $\vee$ means join.

See also

permutohedron_lequal(), *permutohedron_join()*.

AUTHORS:

Viviane Pons and Darij Grinberg, 18 June 2014.

EXAMPLES:

```
sage: p = Permutation([3,1,2])
sage: q = Permutation([1,3,2])
sage: p.permutohedron_meet(q)
[1, 3, 2]
sage: r = Permutation([2,1,3])
sage: r.permutohedron_meet(p)
[1, 2, 3]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3),Integer(1),Integer(2)])
>>> q = Permutation([Integer(1),Integer(3),Integer(2)])
>>> p.permutohedron_meet(q)
[1, 3, 2]
>>> r = Permutation([Integer(2),Integer(1),Integer(3)])
>>> r.permutohedron_meet(p)
[1, 2, 3]
```

```
sage: p = Permutation([3,2,4,1])
sage: q = Permutation([4,2,1,3])
sage: p.permutohedron_meet(q)
[2, 1, 3, 4]
sage: r = Permutation([3,1,2,4])
sage: p.permutohedron_meet(r)
[3, 1, 2, 4]
sage: q.permutohedron_meet(r)
[1, 2, 3, 4]
sage: s = Permutation([1,4,2,3])
sage: s.permutohedron_meet(r)
[1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3),Integer(2),Integer(4),Integer(1)])
>>> q = Permutation([Integer(4),Integer(2),Integer(1),Integer(3)])
>>> p.permutohedron_meet(q)
[2, 1, 3, 4]
>>> r = Permutation([Integer(3),Integer(1),Integer(2),Integer(4)])
>>> p.permutohedron_meet(r)
```

(continues on next page)

(continued from previous page)

```
[3, 1, 2, 4]
>>> q.permutohedron_meet(r)
[1, 2, 3, 4]
>>> s = Permutation([Integer(1),Integer(4),Integer(2),Integer(3)])
>>> s.permutohedron_meet(r)
[1, 2, 3, 4]
```

The universal property of the meet operation is satisfied:

```
sage: def test_uni_meet(p, q):
....:     m = p.permutohedron_meet(q)
....:     if not m.permutohedron_lequal(p):
....:         return False
....:     if not m.permutohedron_lequal(q):
....:         return False
....:     for r in p.permutohedron_smaller():
....:         if r.permutohedron_lequal(q) and not r.permutohedron_lequal(m):
....:             return False
....:     return True
sage: all( test_uni_meet(p, q) for p in Permutations(3) for q in␣
↪Permutations(3) )
True
sage: test_uni_meet(Permutation([6, 4, 7, 3, 2, 5, 8, 1]), Permutation([7, 3,␣
↪1, 2, 5, 4, 6, 8]))
True
```

```
>>> from sage.all import *
>>> def test_uni_meet(p, q):
...     m = p.permutohedron_meet(q)
...     if not m.permutohedron_lequal(p):
...         return False
...     if not m.permutohedron_lequal(q):
...         return False
...     for r in p.permutohedron_smaller():
...         if r.permutohedron_lequal(q) and not r.permutohedron_lequal(m):
...             return False
...     return True
>>> all( test_uni_meet(p, q) for p in Permutations(Integer(3)) for q in␣
↪Permutations(Integer(3)) )
True
>>> test_uni_meet(Permutation([Integer(6), Integer(4), Integer(7), Integer(3),
↪ Integer(2), Integer(5), Integer(8), Integer(1)]), Permutation([Integer(7),␣
↪Integer(3), Integer(1), Integer(2), Integer(5), Integer(4), Integer(6),␣
↪Integer(8)]))
True
```

Border cases:

```
sage: p = Permutation([])
sage: p.permutohedron_meet(p)
[]
sage: p = Permutation([1])
```

(continues on next page)

(continued from previous page)

```
sage: p.permutohedron_meet(p)
[1]
```

```
>>> from sage.all import *
>>> p = Permutation([])
>>> p.permutohedron_meet(p)
[]
>>> p = Permutation([Integer(1)])
>>> p.permutohedron_meet(p)
[1]
```

The left permutohedron:

```
sage: p = Permutation([3,1,2])
sage: q = Permutation([1,3,2])
sage: p.permutohedron_meet(q, side='left')
[1, 2, 3]
sage: r = Permutation([2,1,3])
sage: r.permutohedron_meet(p, side='left')
[2, 1, 3]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3),Integer(1),Integer(2)])
>>> q = Permutation([Integer(1),Integer(3),Integer(2)])
>>> p.permutohedron_meet(q, side='left')
[1, 2, 3]
>>> r = Permutation([Integer(2),Integer(1),Integer(3)])
>>> r.permutohedron_meet(p, side='left')
[2, 1, 3]
```

permutohedron_pred(*side='right'*)

Return a list of the permutations strictly smaller than `self` in the permutohedron order such that there is no permutation between any of those and `self`.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

See *permutohedron_lequal()* for the definition of the permutohedron orders.

EXAMPLES:

```
sage: p = Permutation([4,2,1,3])
sage: p.permutohedron_pred()
[[2, 4, 1, 3], [4, 1, 2, 3]]
sage: p.permutohedron_pred(side='left')
[[4, 1, 2, 3], [3, 2, 1, 4]]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(4),Integer(2),Integer(1),Integer(3)])
>>> p.permutohedron_pred()
[[2, 4, 1, 3], [4, 1, 2, 3]]
>>> p.permutohedron_pred(side='left')
[[4, 1, 2, 3], [3, 2, 1, 4]]
```

permutohedron_smaller(*side='right'*)

Return a list of permutations smaller than or equal to `self` in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

See *permutohedron_lequal()* for the definition of the permutohedron orders.

EXAMPLES:

```
sage: Permutation([4,2,1,3]).permutohedron_smaller()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 4, 2, 3],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [2, 4, 1, 3],
 [4, 1, 2, 3],
 [4, 2, 1, 3]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(2),Integer(1),Integer(3)]).permutohedron_
↪smaller()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 4, 2, 3],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [2, 4, 1, 3],
 [4, 1, 2, 3],
 [4, 2, 1, 3]]
```

```
sage: Permutation([4,2,1,3]).permutohedron_smaller(side='left')
[[1, 2, 3, 4],
 [1, 3, 2, 4],
 [2, 1, 3, 4],
 [2, 3, 1, 4],
 [3, 1, 2, 4],
 [3, 2, 1, 4],
 [4, 1, 2, 3],
 [4, 2, 1, 3]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(2),Integer(1),Integer(3)]).permutohedron_
↪smaller(side='left')
[[1, 2, 3, 4],
 [1, 3, 2, 4],
 [2, 1, 3, 4],
 [2, 3, 1, 4],
 [3, 1, 2, 4],
 [3, 2, 1, 4],
 [4, 1, 2, 3],
 [4, 2, 1, 3]]
```

permutohedron_succ(*side='right'*)

Return a list of the permutations strictly greater than `self` in the permutohedron order such that there is no permutation between any of those and `self`.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

See `permutohedron_lequal()` for the definition of the permutohedron orders.

EXAMPLES:

```
sage: p = Permutation([4,2,1,3])
sage: p.permutohedron_succ()
[[4, 2, 3, 1]]
sage: p.permutohedron_succ(side='left')
[[4, 3, 1, 2]]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(4),Integer(2),Integer(1),Integer(3)])
>>> p.permutohedron_succ()
[[4, 2, 3, 1]]
>>> p.permutohedron_succ(side='left')
[[4, 3, 1, 2]]
```

prev()

Return the permutation that comes directly before `self` in lexicographic order on the symmetric group containing `self`. If `self` is the first permutation, then it returns `False`. Does not support the Steinhaus-Johnson-Trotter algorithm for the moment.

EXAMPLES:

```
sage: p = Permutation([1,2,3])
sage: p.prev()
False
sage: p = Permutation([1,3,2])
sage: p.prev()
[1, 2, 3]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(1),Integer(2),Integer(3)])
>>> p.prev()
False
>>> p = Permutation([Integer(1),Integer(3),Integer(2)])
>>> p.prev()
[1, 2, 3]
```

Todo

Implement the previous permutation for the Steinhaus-Johnson-Trotter algorithm.

rank()

Return the rank of `self` in the lexicographic ordering on the symmetric group to which `self` belongs.

EXAMPLES:

```
sage: Permutation([1,2,3]).rank()
0
sage: Permutation([1, 2, 4, 6, 3, 5]).rank()
10
sage: perms = Permutations(6).list()
sage: [p.rank() for p in perms] == list(range(factorial(6)))
True
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(3)]).rank()
0
>>> Permutation([Integer(1), Integer(2), Integer(4), Integer(6), Integer(3),␣
↪Integer(5)]).rank()
10
>>> perms = Permutations(Integer(6)).list()
>>> [p.rank() for p in perms] == list(range(factorial(Integer(6))))
True
```

rank_matrix()

Return the rank matrix of `self`.

Let $P = [P_{ij}]_{i,j=1}^n$ be the permutation matrix of $w \in S_n$. The rank matrix is the $n \times n$ matrix with entries

$$r_w(i,j) = \sum_{a=1}^{i}\sum_{b=1}^{j} P_{ij}.$$

EXAMPLES:

```
sage: w = Permutation([2, 1, 5, 4, 3])
sage: w.to_matrix()
[0 1 0 0 0]
[1 0 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
[0 0 1 0 0]
sage: w.rank_matrix()
[0 1 1 1 1]
[1 2 2 2 2]
[1 2 2 2 3]
[1 2 2 3 4]
[1 2 3 4 5]
```

```
>>> from sage.all import *
>>> w = Permutation([Integer(2), Integer(1), Integer(5), Integer(4),␣
↪Integer(3)])
>>> w.to_matrix()
[0 1 0 0 0]
[1 0 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
[0 0 1 0 0]
>>> w.rank_matrix()
```

(continues on next page)

(continued from previous page)

```
[0 1 1 1 1]
[1 2 2 2 2]
[1 2 2 2 3]
[1 2 2 3 4]
[1 2 3 4 5]
```

recoils()

Return the list of the positions of the recoils of `self`.

A recoil of a permutation p is an integer i such that $i + 1$ appears to the left of i in p. Here, the positions are being counted starting at 0. (Note that it is the positions, not the recoils themselves, which are being listed.)

EXAMPLES:

```
sage: Permutation([1,4,3,2]).recoils()
[2, 3]
sage: Permutation([]).recoils()
[]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).recoils()
[2, 3]
>>> Permutation([]).recoils()
[]
```

recoils_composition()

Return the recoils composition of `self`.

The recoils composition of a permutation $p \in S_n$ is the composition of n whose descent set is the set of the recoils of p (not their positions). In other words, this is the descents composition of p^{-1}.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).recoils_composition()
[2, 2]
sage: Permutation([]).recoils_composition()
[]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(4)]).recoils_
↪composition()
[2, 2]
>>> Permutation([]).recoils_composition()
[]
```

reduced_word()

Return a reduced word of the permutation `self`.

See *reduced_words()* for the definition of reduced words and a way to compute them all.

> **Warning**
>
> This does not respect the multiplication convention.

EXAMPLES:

```
sage: Permutation([3,5,4,6,2,1]).reduced_word()
[2, 1, 4, 3, 2, 4, 3, 5, 4, 5]

Permutation([1]).reduced_word_lexmin()
[]
Permutation([]).reduced_word_lexmin()
[]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(5),Integer(4),Integer(6),Integer(2),
↪Integer(1)]).reduced_word()
[2, 1, 4, 3, 2, 4, 3, 5, 4, 5]

Permutation([1]).reduced_word_lexmin()
[]
Permutation([]).reduced_word_lexmin()
[]
```

reduced_word_lexmin()

Return a lexicographically minimal reduced word of the permutation `self`.

See *reduced_words()* for the definition of reduced words and a way to compute them all.

EXAMPLES:

```
sage: Permutation([3,4,2,1]).reduced_word_lexmin()
[1, 2, 1, 3, 2]

Permutation([1]).reduced_word_lexmin()
[]
Permutation([]).reduced_word_lexmin()
[]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(4),Integer(2),Integer(1)]).reduced_word_
↪lexmin()
[1, 2, 1, 3, 2]

Permutation([1]).reduced_word_lexmin()
[]
Permutation([]).reduced_word_lexmin()
[]
```

reduced_words()

Return a list of the reduced words of `self`.

The notion of a reduced word is based on the well-known fact that every permutation can be written as a product of adjacent transpositions. In more detail: If n is a nonnegative integer, we can define the transpositions $s_i = (i, i+1) \in S_n$ for all $i \in \{1, 2, \ldots, n-1\}$, and every $p \in S_n$ can then be written as a product $s_{i_1} s_{i_2} \cdots s_{i_k}$ for some sequence $(i_1, i_2, \ldots, i_k)$ of elements of $\{1, 2, \ldots, n-1\}$ (here $\{1, 2, \ldots, n-1\}$ denotes the empty set when $n \leq 1$). Fixing a p, the sequences $(i_1, i_2, \ldots, i_k)$ of smallest length satisfying $p = s_{i_1} s_{i_2} \cdots s_{i_k}$ are called the reduced words of p. (Their length is the Coxeter length of p, and can be computed using *length()*.)

Note that the product of permutations is defined here in such a way that $(pq)(i) = p(q(i))$ for all permutations p and q and each $i \in \{1, 2, \ldots, n\}$ (this is the same convention as in `left_action_product()`, but not the default semantics of the $*$ operator on permutations in Sage). Thus, for instance, $s_2 s_1$ is the permutation obtained by first transposing 1 with 2 and then transposing 2 with 3.

See also:

`reduced_word()`, `reduced_word_lexmin()`

EXAMPLES:

```
sage: Permutation([2,1,3]).reduced_words()
[[1]]
sage: Permutation([3,1,2]).reduced_words()
[[2, 1]]
sage: Permutation([3,2,1]).reduced_words()
[[1, 2, 1], [2, 1, 2]]
sage: Permutation([3,2,4,1]).reduced_words()
[[1, 2, 3, 1], [1, 2, 1, 3], [2, 1, 2, 3]]

Permutation([1]).reduced_words()
[[]]
Permutation([]).reduced_words()
[[]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3)]).reduced_words()
[[1]]
>>> Permutation([Integer(3),Integer(1),Integer(2)]).reduced_words()
[[2, 1]]
>>> Permutation([Integer(3),Integer(2),Integer(1)]).reduced_words()
[[1, 2, 1], [2, 1, 2]]
>>> Permutation([Integer(3),Integer(2),Integer(4),Integer(1)]).reduced_words()
[[1, 2, 3, 1], [1, 2, 1, 3], [2, 1, 2, 3]]

Permutation([1]).reduced_words()
[[]]
Permutation([]).reduced_words()
[[]]
```

reduced_words_iterator()

Return an iterator for the reduced words of `self`.

EXAMPLES:

```
sage: next(Permutation([5,2,3,4,1]).reduced_words_iterator())
[1, 2, 3, 4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> next(Permutation([Integer(5),Integer(2),Integer(3),Integer(4),
↪Integer(1)]).reduced_words_iterator())
[1, 2, 3, 4, 3, 2, 1]
```

remove_extra_fixed_points()

Return the permutation obtained by removing any fixed points at the end of `self`.

However, return `[1]` rather than `[]` if `self` is the identity permutation.

This is mostly a helper method for *sage.combinat.schubert_polynomial*, where it is used to normalize finitary permutations of $\{1, 2, 3, \ldots\}$.

EXAMPLES:

```
sage: Permutation([2,1,3]).remove_extra_fixed_points()
[2, 1]
sage: Permutation([1,2,3,4]).remove_extra_fixed_points()
[1]
sage: Permutation([2,1]).remove_extra_fixed_points()
[2, 1]
sage: Permutation([]).remove_extra_fixed_points()
[1]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3)]).remove_extra_fixed_
→points()
[2, 1]
>>> Permutation([Integer(1),Integer(2),Integer(3),Integer(4)]).remove_extra_
→fixed_points()
[1]
>>> Permutation([Integer(2),Integer(1)]).remove_extra_fixed_points()
[2, 1]
>>> Permutation([]).remove_extra_fixed_points()
[1]
```

See also

retract_plain()

retract_direct_product(*m*)

Return the direct-product retract of the permutation `self` $\in S_n$ to S_m, where $m \leq n$. If this retract is undefined, then `None` is returned.

If $p \in S_n$ is a permutation, and m is a nonnegative integer less or equal to n, then the direct-product retract of p to S_m is defined only if $p([m]) = [m]$, where $[m]$ denotes the interval $\{1, 2, \ldots, m\}$. In this case, it is defined as the permutation written $(p(1), p(2), \ldots, p(m))$ in one-line notation.

EXAMPLES:

```
sage: Permutation([4,1,2,3,5]).retract_direct_product(4)
[4, 1, 2, 3]
sage: Permutation([4,1,2,3,5]).retract_direct_product(3)

sage: Permutation([1,4,2,3,6,5]).retract_direct_product(5)
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(4)
[1, 4, 2, 3]
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(3)
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(2)
```

(continues on next page)

(continued from previous page)

```
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(1)
[1]
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(0)
[]

sage: all( p.retract_direct_product(3) == p for p in Permutations(3) )
True
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
↪retract_direct_product(Integer(4))
[4, 1, 2, 3]
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
↪retract_direct_product(Integer(3))

>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_direct_product(Integer(5))
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_direct_product(Integer(4))
[1, 4, 2, 3]
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_direct_product(Integer(3))
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_direct_product(Integer(2))
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_direct_product(Integer(1))
[1]
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_direct_product(Integer(0))
[]

>>> all( p.retract_direct_product(Integer(3)) == p for p in␣
↪Permutations(Integer(3)) )
True
```

See also

retract_plain(), *retract_okounkov_vershik()*

retract_okounkov_vershik(*m*)

Return the Okounkov-Vershik retract of the permutation `self` $\in S_n$ to S_m, where $m \leq n$.

If $p \in S_n$ is a permutation, and m is a nonnegative integer less or equal to n, then the Okounkov-Vershik retract of p to S_m is defined as the permutation in S_m which sends every $i \in \{1, 2, \ldots, m\}$ to $p^{k_i}(i)$, where k_i is the smallest positive integer k satisfying $p^k(i) \leq m$.

In other words, the Okounkov-Vershik retract of p is the permutation whose disjoint cycle decomposition is obtained by removing all letters strictly greater than m from the decomposition of p into disjoint cycles (and removing all cycles which are emptied in the process).

When $m = n-1$, the Okounkov-Vershik retract (as a map $S_n \to S_{n-1}$) is the map $\widetilde{p}_n$ introduced in Section 7 of [VO2005], and appears as (3.20) in [CST2010]. In the general case, the Okounkov-Vershik retract of a permutation in S_n to S_m can be obtained by first taking its Okounkov-Vershik retract to S_{n-1}, then that of the resulting permutation to S_{n-2}, etc. until arriving in S_m.

EXAMPLES:

```
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(4)
[4, 1, 2, 3]
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(3)
[3, 1, 2]
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(2)
[2, 1]
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(1)
[1]
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(0)
[]

sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(5)
[1, 4, 2, 3, 5]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(4)
[1, 4, 2, 3]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(3)
[1, 3, 2]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(2)
[1, 2]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(1)
[1]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(0)
[]

sage: Permutation([6,5,4,3,2,1]).retract_okounkov_vershik(5)
[1, 5, 4, 3, 2]
sage: Permutation([6,5,4,3,2,1]).retract_okounkov_vershik(4)
[1, 2, 4, 3]

sage: Permutation([1,5,2,6,3,7,4,8]).retract_okounkov_vershik(4)
[1, 3, 2, 4]

sage: all( p.retract_direct_product(3) == p for p in Permutations(3) )
True
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
↪retract_okounkov_vershik(Integer(4))
[4, 1, 2, 3]
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
↪retract_okounkov_vershik(Integer(3))
[3, 1, 2]
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
↪retract_okounkov_vershik(Integer(2))
[2, 1]
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
↪retract_okounkov_vershik(Integer(1))
[1]
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
```

(continues on next page)

(continued from previous page)

```
↪retract_okounkov_vershik(Integer(0))
[]

>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_okounkov_vershik(Integer(5))
[1, 4, 2, 3, 5]
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_okounkov_vershik(Integer(4))
[1, 4, 2, 3]
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_okounkov_vershik(Integer(3))
[1, 3, 2]
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_okounkov_vershik(Integer(2))
[1, 2]
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_okounkov_vershik(Integer(1))
[1]
>>> Permutation([Integer(1),Integer(4),Integer(2),Integer(3),Integer(6),
↪Integer(5)]).retract_okounkov_vershik(Integer(0))
[]

>>> Permutation([Integer(6),Integer(5),Integer(4),Integer(3),Integer(2),
↪Integer(1)]).retract_okounkov_vershik(Integer(5))
[1, 5, 4, 3, 2]
>>> Permutation([Integer(6),Integer(5),Integer(4),Integer(3),Integer(2),
↪Integer(1)]).retract_okounkov_vershik(Integer(4))
[1, 2, 4, 3]

>>> Permutation([Integer(1),Integer(5),Integer(2),Integer(6),Integer(3),
↪Integer(7),Integer(4),Integer(8)]).retract_okounkov_vershik(Integer(4))
[1, 3, 2, 4]

>>> all( p.retract_direct_product(Integer(3)) == p for p in
↪Permutations(Integer(3)) )
True
```

See also:

retract_plain(), *retract_direct_product()*

retract_plain(m)

Return the plain retract of the permutation `self` in S_n to S_m, where $m \leq n$. If this retract is undefined, then `None` is returned.

If $p \in S_n$ is a permutation, and m is a nonnegative integer less or equal to n, then the plain retract of p to S_m is defined only if every $i > m$ satisfies $p(i) = i$. In this case, it is defined as the permutation written $(p(1), p(2), \ldots, p(m))$ in one-line notation.

EXAMPLES:

```
sage: Permutation([4,1,2,3,5]).retract_plain(4)
[4, 1, 2, 3]
sage: Permutation([4,1,2,3,5]).retract_plain(3)

sage: Permutation([1,3,2,4,5,6]).retract_plain(3)
[1, 3, 2]
sage: Permutation([1,3,2,4,5,6]).retract_plain(2)

sage: Permutation([1,2,3,4,5]).retract_plain(1)
[1]
sage: Permutation([1,2,3,4,5]).retract_plain(0)
[]

sage: all( p.retract_plain(3) == p for p in Permutations(3) )
True
```

```
>>> from sage.all import *
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
↪retract_plain(Integer(4))
[4, 1, 2, 3]
>>> Permutation([Integer(4),Integer(1),Integer(2),Integer(3),Integer(5)]).
↪retract_plain(Integer(3))

>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(4),Integer(5),
↪Integer(6)]).retract_plain(Integer(3))
[1, 3, 2]
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(4),Integer(5),
↪Integer(6)]).retract_plain(Integer(2))

>>> Permutation([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]).
↪retract_plain(Integer(1))
[1]
>>> Permutation([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]).
↪retract_plain(Integer(0))
[]

>>> all( p.retract_plain(Integer(3)) == p for p in Permutations(Integer(3)) )
True
```

See also

retract_direct_product(), *retract_okounkov_vershik()*, *remove_extra_fixed_points()*

reverse()

Return the permutation obtained by reversing the list.

EXAMPLES:

```
sage: Permutation([3,4,1,2]).reverse()
[2, 1, 4, 3]
```

(continues on next page)

(continued from previous page)

```
sage: Permutation([1,2,3,4,5]).reverse()
[5, 4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(4),Integer(1),Integer(2)]).reverse()
[2, 1, 4, 3]
>>> Permutation([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]).
↪reverse()
[5, 4, 3, 2, 1]
```

right_action_product(*rp*)

Return the permutation obtained by composing `self` with `rp` in such an order that `self` is applied first and `rp` is applied afterwards.

This is usually denoted by either `self * rp` or `rp * self` depending on the conventions used by the author. If the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by $p(i)$, then this should be denoted by `rp * self` in order to have associativity (i.e., in order to have $(p \cdot q)(i) = p(q(i))$ for all p, q and i). If, on the other hand, the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by i^p, then this should be denoted by `self * rp` in order to have associativity (i.e., in order to have $i^{p \cdot q} = (i^p)^q$ for all p, q and i).

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: q = Permutation([3,1,2])
sage: p.right_action_product(q)
[1, 3, 2]
sage: q.right_action_product(p)
[3, 2, 1]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2),Integer(1),Integer(3)])
>>> q = Permutation([Integer(3),Integer(1),Integer(2)])
>>> p.right_action_product(q)
[1, 3, 2]
>>> q.right_action_product(p)
[3, 2, 1]
```

right_permutohedron_interval(*other*)

Return the list of the permutations belonging to the right permutohedron interval where `self` is the minimal element and `other` the maximal element.

See *permutohedron_lequal()* for the definition of the permutohedron orders.

EXAMPLES:

```
sage: p = Permutation([2, 1, 4, 5, 3]); q = Permutation([2, 5, 4, 1, 3])
sage: p.right_permutohedron_interval(q)                                    #␣
↪needs sage.graphs sage.modules
[[2, 4, 5, 1, 3], [2, 4, 1, 5, 3], [2, 1, 4, 5, 3],
 [2, 1, 5, 4, 3], [2, 5, 1, 4, 3], [2, 5, 4, 1, 3]]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2), Integer(1), Integer(4), Integer(5),␣
↪Integer(3)]); q = Permutation([Integer(2), Integer(5), Integer(4),␣
↪Integer(1), Integer(3)])
>>> p.right_permutohedron_interval(q)                                    #␣
↪needs sage.graphs sage.modules
[[2, 4, 5, 1, 3], [2, 4, 1, 5, 3], [2, 1, 4, 5, 3],
 [2, 1, 5, 4, 3], [2, 5, 1, 4, 3], [2, 5, 4, 1, 3]]
```

right_permutohedron_interval_iterator(*other*)

Return an iterator on the permutations (represented as integer lists) belonging to the right permutohedron interval where `self` is the minimal element and `other` the maximal element.

See `permutohedron_lequal()` for the definition of the permutohedron orders.

EXAMPLES:

```
sage: p = Permutation([2, 1, 4, 5, 3]); q = Permutation([2, 5, 4, 1, 3])
sage: p.right_permutohedron_interval(q)  # indirect doctest              #␣
↪needs sage.graphs sage.modules
[[2, 4, 5, 1, 3], [2, 4, 1, 5, 3], [2, 1, 4, 5, 3],
 [2, 1, 5, 4, 3], [2, 5, 1, 4, 3], [2, 5, 4, 1, 3]]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2), Integer(1), Integer(4), Integer(5),␣
↪Integer(3)]); q = Permutation([Integer(2), Integer(5), Integer(4),␣
↪Integer(1), Integer(3)])
>>> p.right_permutohedron_interval(q)  # indirect doctest                #␣
↪needs sage.graphs sage.modules
[[2, 4, 5, 1, 3], [2, 4, 1, 5, 3], [2, 1, 4, 5, 3],
 [2, 1, 5, 4, 3], [2, 5, 1, 4, 3], [2, 5, 4, 1, 3]]
```

right_tableau()

Return the right standard tableau after performing the RSK algorithm on `self`.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).right_tableau()                             #␣
↪needs sage.combinat
[[1, 2], [3], [4]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2)]).right_
↪tableau()                                # needs sage.combinat
[[1, 2], [3], [4]]
```

robinson_schensted()

Return the pair of standard tableaux obtained by running the Robinson-Schensted algorithm on `self`.

This can also be done by running `RSK()` on `self` (with the optional argument `check_standard=True` to return standard Young tableaux).

EXAMPLES:

```
sage: Permutation([6,2,3,1,7,5,4]).robinson_schensted()                    #␣
↪needs sage.combinat
[[[1, 3, 4], [2, 5], [6, 7]], [[1, 3, 5], [2, 6], [4, 7]]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(6),Integer(2),Integer(3),Integer(1),Integer(7),
↪Integer(5),Integer(4)]).robinson_schensted()                     # needs␣
↪sage.combinat
[[[1, 3, 4], [2, 5], [6, 7]], [[1, 3, 5], [2, 6], [4, 7]]]
```

rothe_diagram()

Return the Rothe diagram of `self`.

EXAMPLES:

```
sage: p = Permutation([4,2,1,3])
sage: D = p.rothe_diagram(); D                                             #␣
↪needs sage.combinat
[(0, 0), (0, 1), (0, 2), (1, 0)]
sage: D.pp()                                                               #␣
↪needs sage.combinat
O O O .
O . . .
. . . .
. . . .
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(4),Integer(2),Integer(1),Integer(3)])
>>> D = p.rothe_diagram(); D                                             #␣
↪needs sage.combinat
[(0, 0), (0, 1), (0, 2), (1, 0)]
>>> D.pp()                                                               #␣
↪needs sage.combinat
O O O .
O . . .
. . . .
. . . .
```

runs(*as_tuple=False*)

Return a list of the runs in the nonempty permutation `self`.

A run in a permutation is defined to be a maximal (with respect to inclusion) nonempty increasing substring (i.e., contiguous subsequence). For instance, the runs in the permutation `[6,1,7,3,4,5,2]` are `[6]`, `[1,7]`, `[3,4,5]` and `[2]`.

Runs in an empty permutation are not defined.

INPUT:

- `as_tuple` – boolean (default: `False`); choice of output format

OUTPUT: list of lists or a tuple of tuples

REFERENCES:

- http://mathworld.wolfram.com/PermutationRun.html

EXAMPLES:

```
sage: Permutation([1,2,3,4]).runs()
[[1, 2, 3, 4]]
sage: Permutation([4,3,2,1]).runs()
[[4], [3], [2], [1]]
sage: Permutation([2,4,1,3]).runs()
[[2, 4], [1, 3]]
sage: Permutation([1]).runs()
[[1]]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(3),Integer(4)]).runs()
[[1, 2, 3, 4]]
>>> Permutation([Integer(4),Integer(3),Integer(2),Integer(1)]).runs()
[[4], [3], [2], [1]]
>>> Permutation([Integer(2),Integer(4),Integer(1),Integer(3)]).runs()
[[2, 4], [1, 3]]
>>> Permutation([Integer(1)]).runs()
[[1]]
```

The example from above:

```
sage: Permutation([6,1,7,3,4,5,2]).runs()
[[6], [1, 7], [3, 4, 5], [2]]
sage: Permutation([6,1,7,3,4,5,2]).runs(as_tuple=True)
((6,), (1, 7), (3, 4, 5), (2,))
```

```
>>> from sage.all import *
>>> Permutation([Integer(6),Integer(1),Integer(7),Integer(3),Integer(4),
↪Integer(5),Integer(2)]).runs()
[[6], [1, 7], [3, 4, 5], [2]]
>>> Permutation([Integer(6),Integer(1),Integer(7),Integer(3),Integer(4),
↪Integer(5),Integer(2)]).runs(as_tuple=True)
((6,), (1, 7), (3, 4, 5), (2,))
```

The number of runs in a nonempty permutation equals its number of descents plus 1:

```
sage: all( len(p.runs()) == p.number_of_descents() + 1
....:      for p in Permutations(6) )
True
```

```
>>> from sage.all import *
>>> all( len(p.runs()) == p.number_of_descents() + Integer(1)
...      for p in Permutations(Integer(6)) )
True
```

saliances()

Return a list of the saliances of the permutation `self`.

A saliance of a permutation p is an integer i such that $p(i) > p(j)$ for all $j > i$.

EXAMPLES:

```
sage: Permutation([2,3,1,5,4]).saliances()
[3, 4]
sage: Permutation([5,4,3,2,1]).saliances()
[0, 1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(3),Integer(1),Integer(5),Integer(4)]).
↪saliances()
[3, 4]
>>> Permutation([Integer(5),Integer(4),Integer(3),Integer(2),Integer(1)]).
↪saliances()
[0, 1, 2, 3, 4]
```

schubert_determinant_ideal()

Return the Schubert determinant ideal of `self`.

From Lemma 3.10 [Ful1992], the matrix Schubert variety of a permutation $w \in S_n$ is defined by the ideal

$$I_w = \sum_{(i,j)} I_{r_w(i,j)+1}(i,j),$$

where the sum is over the *essential set*, $[r_w(i,j)]_{i,j}$ is the *rank_matrix()* of w, and $I_k(i,j)$ is the ideal generated by the $k \times k$ minors of the $i \times j$ matrix $[z_{ab}]_{a,b}$.

These ideals are known to be prime of codimension equal to $\ell(w)$ (the length of w) such that R/I_w is Cohen-Macaulay, where R is the polynomial ring $\mathbf{Q}[z_{ab} | 1 \leq a, b \leq n]$ from Prop. 3.10 of [Ful1992].

EXAMPLES:

```
sage: w = Permutation([3, 1, 4, 2])
sage: Iw = w.schubert_determinant_ideal()
sage: Iw.gens()
[z00, z01, -z01*z10 + z00*z11, -z01*z20 + z00*z21, -z11*z20 + z10*z21]
sage: Iw.dimension()
13
sage: Iw.dimension() + w.length()
16
sage: Iw.is_prime()
True

sage: w = Permutation([2, 1, 5, 4, 3])
sage: Iw = w.schubert_determinant_ideal()
sage: Iw.dimension()
21
sage: w.length() + Iw.dimension()
25
sage: Iw.is_prime()
True
```

```
>>> from sage.all import *
>>> w = Permutation([Integer(3), Integer(1), Integer(4), Integer(2)])
>>> Iw = w.schubert_determinant_ideal()
>>> Iw.gens()
[z00, z01, -z01*z10 + z00*z11, -z01*z20 + z00*z21, -z11*z20 + z10*z21]
```

(continues on next page)

(continued from previous page)

```
>>> Iw.dimension()
13
>>> Iw.dimension() + w.length()
16
>>> Iw.is_prime()
True

>>> w = Permutation([Integer(2), Integer(1), Integer(5), Integer(4),␣
→Integer(3)])
>>> Iw = w.schubert_determinant_ideal()
>>> Iw.dimension()
21
>>> w.length() + Iw.dimension()
25
>>> Iw.is_prime()
True
```

shifted_concatenation(*other*, *side='right'*)

Return the right (or left) shifted concatenation of `self` with a permutation `other`. These operations are also known as the Loday-Ronco over and under operations.

INPUT:

- `other` – a permutation, a list, a tuple, or any iterable representing a permutation
- `side` – string (default: `'right'`); `'left'` or `'right'`

OUTPUT:

If `side` is `'right'`, the method returns the permutation obtained by concatenating `self` with the letters of `other` incremented by the size of `self`. This is what is called `side / other` in [LR0102066], and denoted as the "over" operation. Otherwise, i. e., when `side` is `'left'`, the method returns the permutation obtained by concatenating the letters of `other` incremented by the size of `self` with `self`. This is what is called `side \ other` in [LR0102066] (which seems to use the $(\sigma\pi)(i) = \pi(\sigma(i))$ convention for the product of permutations).

EXAMPLES:

```
sage: Permutation([]).shifted_concatenation(Permutation([]), "right")
[]
sage: Permutation([]).shifted_concatenation(Permutation([]), "left")
[]
sage: Permutation([2, 4, 1, 3]).shifted_concatenation(Permutation([3, 1, 2]),
→"right")
[2, 4, 1, 3, 7, 5, 6]
sage: Permutation([2, 4, 1, 3]).shifted_concatenation(Permutation([3, 1, 2]),
→"left")
[7, 5, 6, 2, 4, 1, 3]
```

```
>>> from sage.all import *
>>> Permutation([]).shifted_concatenation(Permutation([]), "right")
[]
>>> Permutation([]).shifted_concatenation(Permutation([]), "left")
[]
>>> Permutation([Integer(2), Integer(4), Integer(1), Integer(3)]).shifted_
```

(continues on next page)

(continued from previous page)

```
↪concatenation(Permutation([Integer(3), Integer(1), Integer(2)]), "right")
[2, 4, 1, 3, 7, 5, 6]
>>> Permutation([Integer(2), Integer(4), Integer(1), Integer(3)]).shifted_
↪concatenation(Permutation([Integer(3), Integer(1), Integer(2)]), "left")
[7, 5, 6, 2, 4, 1, 3]
```

shifted_shuffle(*other*)

Return the shifted shuffle of two permutations `self` and `other`.

INPUT:

- `other` – a permutation, a list, a tuple, or any iterable representing a permutation

OUTPUT:

The list of the permutations appearing in the shifted shuffle of the permutations `self` and `other`.

EXAMPLES:

```
sage: # needs sage.graphs sage.modules
sage: Permutation([]).shifted_shuffle(Permutation([]))
[[]]
sage: Permutation([1, 2, 3]).shifted_shuffle(Permutation([1]))
[[4, 1, 2, 3], [1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3]]
sage: Permutation([1, 2]).shifted_shuffle(Permutation([2, 1]))
[[4, 1, 3, 2], [4, 3, 1, 2], [1, 4, 3, 2],
 [1, 4, 2, 3], [1, 2, 4, 3], [4, 1, 2, 3]]
sage: Permutation([1]).shifted_shuffle([1])
[[2, 1], [1, 2]]
sage: p = Permutation([3, 1, 5, 4, 2])
sage: len(p.shifted_shuffle(Permutation([2, 1, 4, 3])))
126
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.modules
>>> Permutation([]).shifted_shuffle(Permutation([]))
[[]]
>>> Permutation([Integer(1), Integer(2), Integer(3)]).shifted_
↪shuffle(Permutation([Integer(1)]))
[[4, 1, 2, 3], [1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3]]
>>> Permutation([Integer(1), Integer(2)]).shifted_
↪shuffle(Permutation([Integer(2), Integer(1)]))
[[4, 1, 3, 2], [4, 3, 1, 2], [1, 4, 3, 2],
 [1, 4, 2, 3], [1, 2, 4, 3], [4, 1, 2, 3]]
>>> Permutation([Integer(1)]).shifted_shuffle([Integer(1)])
[[2, 1], [1, 2]]
>>> p = Permutation([Integer(3), Integer(1), Integer(5), Integer(4),␣
↪Integer(2)])
>>> len(p.shifted_shuffle(Permutation([Integer(2), Integer(1), Integer(4),␣
↪Integer(3)])))
126
```

The shifted shuffle product is associative. We can test this on an admittedly toy example:

```
sage: all( all( all( sorted(flatten([abs.shifted_shuffle(c)                 #␣
↪needs sage.graphs sage.modules
....:                                 for abs in a.shifted_shuffle(b)]))
....:                  == sorted(flatten([a.shifted_shuffle(bcs)
....:                                    for bcs in b.shifted_shuffle(c)]))
....:                  for c in Permutations(2) )
....:             for b in Permutations(2) )
....:        for a in Permutations(2) )
True
```

```
>>> from sage.all import *
>>> all( all( all( sorted(flatten([abs.shifted_shuffle(c)                 #␣
↪needs sage.graphs sage.modules
...                                 for abs in a.shifted_shuffle(b)]))
...                  == sorted(flatten([a.shifted_shuffle(bcs)
...                                    for bcs in b.shifted_shuffle(c)]))
...                  for c in Permutations(Integer(2)) )
...             for b in Permutations(Integer(2)) )
...        for a in Permutations(Integer(2)) )
True
```

The shifted_shuffle method on permutations gives the same permutations as the shifted_shuffle method on words (but is faster):

```
sage: all( all( sorted(p1.shifted_shuffle(p2))                              #␣
↪needs sage.combinat sage.graphs sage.modules sage.rings.finite_rings
....:             == sorted([Permutation(p) for p in
....:                         Word(p1).shifted_shuffle(Word(p2))])
....:             for p2 in Permutations(3) )
....:        for p1 in Permutations(2) )
True
```

```
>>> from sage.all import *
>>> all( all( sorted(p1.shifted_shuffle(p2))                              #␣
↪needs sage.combinat sage.graphs sage.modules sage.rings.finite_rings
...             == sorted([Permutation(p) for p in
...                         Word(p1).shifted_shuffle(Word(p2))])
...             for p2 in Permutations(Integer(3)) )
...        for p1 in Permutations(Integer(2)) )
True
```

show(*representation='cycles'*, *orientation='landscape'*, ***args*)

Display the permutation as a drawing.

INPUT:

- representation – different kinds of drawings are available
  - 'cycles' – default; the permutation is displayed as a collection of directed cycles
  - 'braid' – the permutation is displayed as segments linking each element $1, ..., n$ to its image on a parallel line

    When using this drawing, it is also possible to display the permutation horizontally (orientation = "landscape", default option) or vertically (orientation = "portrait").

- 'chord-diagram' – the permutation is displayed as a directed graph, all of its vertices being located on a circle

All additional arguments are forwarded to the show subcalls.

EXAMPLES:

```
sage: P20 = Permutations(20)
sage: P20.random_element().show(representation='cycles')                    #␣
↪needs sage.graphs sage.plot
sage: P20.random_element().show(representation='chord-diagram')             #␣
↪needs sage.graphs sage.plot
sage: P20.random_element().show(representation='braid')                     #␣
↪needs sage.plot
sage: P20.random_element().show(representation='braid',                     #␣
↪needs sage.plot
....:                              orientation='portrait')
```

```
>>> from sage.all import *
>>> P20 = Permutations(Integer(20))
>>> P20.random_element().show(representation='cycles')                      #␣
↪needs sage.graphs sage.plot
>>> P20.random_element().show(representation='chord-diagram')               #␣
↪needs sage.graphs sage.plot
>>> P20.random_element().show(representation='braid')                       #␣
↪needs sage.plot
>>> P20.random_element().show(representation='braid',                       #␣
↪needs sage.plot
...                              orientation='portrait')
```

sign()

Return the signature of the permutation self. This is $(-1)^l$, where l is the number of inversions of self.

> **Note**
>
> *sign()* can be used as an alias for *signature()*.

EXAMPLES:

```
sage: Permutation([4, 2, 3, 1, 5]).signature()
-1
sage: Permutation([1,3,2,5,4]).sign()
1
sage: Permutation([]).sign()
1
```

```
>>> from sage.all import *
>>> Permutation([Integer(4), Integer(2), Integer(3), Integer(1), Integer(5)]).
↪signature()
-1
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(5),Integer(4)]).
↪sign()
1
```

(continues on next page)

(continued from previous page)

```
>>> Permutation([]).sign()
1
```

signature()

Return the signature of the permutation `self`. This is $(-1)^l$, where l is the number of inversions of `self`.

Note

sign() can be used as an alias for *signature()*.

EXAMPLES:

```
sage: Permutation([4, 2, 3, 1, 5]).signature()
-1
sage: Permutation([1,3,2,5,4]).sign()
1
sage: Permutation([]).sign()
1
```

```
>>> from sage.all import *
>>> Permutation([Integer(4), Integer(2), Integer(3), Integer(1), Integer(5)]).
↪signature()
-1
>>> Permutation([Integer(1),Integer(3),Integer(2),Integer(5),Integer(4)]).
↪sign()
1
>>> Permutation([]).sign()
1
```

simion_schmidt(*avoid=[1, 2, 3]*)

Implement the Simion-Schmidt map which sends an arbitrary permutation to a pattern avoiding permutation, where the permutation pattern is one of four length-three patterns. This method also implements the bijection between (for example) `[1,2,3]`- and `[1,3,2]`-avoiding permutations.

INPUT:

- `avoid` – one of the patterns `[1,2,3]`, `[1,3,2]`, `[3,1,2]`, `[3,2,1]`

EXAMPLES:

```
sage: P = Permutations(6)
sage: p = P([4,5,1,6,3,2])
sage: pl = [ [1,2,3], [1,3,2], [3,1,2], [3,2,1] ]
sage: for q in pl:                                                          #␣
↪needs sage.combinat
....:     s = p.simion_schmidt(q)
....:     print("{} {}".format(s, s.has_pattern(q)))
[4, 6, 1, 5, 3, 2] False
[4, 2, 1, 3, 5, 6] False
[4, 5, 3, 6, 2, 1] False
[4, 5, 1, 6, 2, 3] False
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(6))
>>> p = P([Integer(4),Integer(5),Integer(1),Integer(6),Integer(3),Integer(2)])
>>> pl = [ [Integer(1),Integer(2),Integer(3)], [Integer(1),Integer(3),
↪Integer(2)], [Integer(3),Integer(1),Integer(2)], [Integer(3),Integer(2),
↪Integer(1)] ]
>>> for q in pl:                                                          #␣
↪needs sage.combinat
...     s = p.simion_schmidt(q)
...     print("{} {}".format(s, s.has_pattern(q)))
[4, 6, 1, 5, 3, 2] False
[4, 2, 1, 3, 5, 6] False
[4, 5, 3, 6, 2, 1] False
[4, 5, 1, 6, 2, 3] False
```

size()

Return the size of `self`.

EXAMPLES:

```
sage: Permutation([3,4,1,2,5]).size()
5
```

```
>>> from sage.all import *
>>> Permutation([Integer(3),Integer(4),Integer(1),Integer(2),Integer(5)]).
↪size()
5
```

stack_sort()

Return the stack sort of a permutation.

This is another permutation obtained through the process of sorting using one stack. If the result is the identity permutation, the original permutation is *stack-sortable*.

See Wikipedia article Stack-sortable_permutation

EXAMPLES:

```
sage: p = Permutation([2,1,5,3,4,9,7,8,6])
sage: p.stack_sort()
[1, 2, 3, 4, 5, 7, 6, 8, 9]

sage: S5 = Permutations(5)
sage: len([1 for s in S5 if s.stack_sort() == S5.one()])
42
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2),Integer(1),Integer(5),Integer(3),Integer(4),
↪Integer(9),Integer(7),Integer(8),Integer(6)])
>>> p.stack_sort()
[1, 2, 3, 4, 5, 7, 6, 8, 9]

>>> S5 = Permutations(Integer(5))
```

(continues on next page)

(continued from previous page)

```
>>> len([Integer(1) for s in S5 if s.stack_sort() == S5.one()])
42
```

sylvester_class(*left_to_right=False*)

Iterate over the equivalence class of the permutation `self` under sylvester congruence.

Sylvester congruence is an equivalence relation on the set S_n of all permutations of n. It is defined as the smallest equivalence relation such that every permutation of the form $uacvbw$ with u, v and w being words and a, b and c being letters satisfying $a \leq b < c$ is equivalent to the permutation $ucavbw$. (Here, permutations are regarded as words by way of one-line notation.) This definition comes from [HNT2005], Definition 8, where it is more generally applied to arbitrary words.

The equivalence class of a permutation $p \in S_n$ under sylvester congruence is called the *sylvester class* of p. It is an interval in the right permutohedron order (see `permutohedron_lequal()`) on S_n.

This is related to the `sylvester_class()` method in that the equivalence class of a permutation π under sylvester congruence is the sylvester class of the right-to-left binary search tree of π. However, the present method yields permutations, while the method on labelled binary trees yields plain lists.

If the variable `left_to_right` is set to `True`, the method instead iterates over the equivalence class of `self` with respect to the *left* sylvester congruence. The left sylvester congruence is easiest to define by saying that two permutations are equivalent under it if and only if their reverses (`reverse()`) are equivalent under (standard) sylvester congruence.

EXAMPLES:

The sylvester class of a permutation in S_5:

```
sage: p = Permutation([3, 5, 1, 2, 4])
sage: sorted(p.sylvester_class())                                       #␣
↪needs sage.combinat sage.graphs
[[1, 3, 2, 5, 4],
 [1, 3, 5, 2, 4],
 [1, 5, 3, 2, 4],
 [3, 1, 2, 5, 4],
 [3, 1, 5, 2, 4],
 [3, 5, 1, 2, 4],
 [5, 1, 3, 2, 4],
 [5, 3, 1, 2, 4]]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3), Integer(5), Integer(1), Integer(2),␣
↪Integer(4)])
>>> sorted(p.sylvester_class())                                       #␣
↪needs sage.combinat sage.graphs
[[1, 3, 2, 5, 4],
 [1, 3, 5, 2, 4],
 [1, 5, 3, 2, 4],
 [3, 1, 2, 5, 4],
 [3, 1, 5, 2, 4],
 [3, 5, 1, 2, 4],
 [5, 1, 3, 2, 4],
 [5, 3, 1, 2, 4]]
```

The sylvester class of a permutation p contains p:

```
sage: all(p in p.sylvester_class() for p in Permutations(4))                #␣
↪needs sage.combinat sage.graphs
True
```

```
>>> from sage.all import *
>>> all(p in p.sylvester_class() for p in Permutations(Integer(4)))         ␣
↪     # needs sage.combinat sage.graphs
True
```

Small cases:

```
sage: list(Permutation([]).sylvester_class())                               #␣
↪needs sage.combinat sage.graphs
[[]]

sage: list(Permutation([1]).sylvester_class())                              #␣
↪needs sage.combinat sage.graphs
[[1]]
```

```
>>> from sage.all import *
>>> list(Permutation([]).sylvester_class())                               #␣
↪needs sage.combinat sage.graphs
[[]]

>>> list(Permutation([Integer(1)]).sylvester_class())                       ␣
↪     # needs sage.combinat sage.graphs
[[1]]
```

The sylvester classes in S_3:

```
sage: [sorted(p.sylvester_class()) for p in Permutations(3)]                #␣
↪needs sage.combinat sage.graphs
[[[1, 2, 3]],
 [[1, 3, 2], [3, 1, 2]],
 [[2, 1, 3]],
 [[2, 3, 1]],
 [[1, 3, 2], [3, 1, 2]],
 [[3, 2, 1]]]
```

```
>>> from sage.all import *
>>> [sorted(p.sylvester_class()) for p in Permutations(Integer(3))]         ␣
↪     # needs sage.combinat sage.graphs
[[[1, 2, 3]],
 [[1, 3, 2], [3, 1, 2]],
 [[2, 1, 3]],
 [[2, 3, 1]],
 [[1, 3, 2], [3, 1, 2]],
 [[3, 2, 1]]]
```

The left sylvester classes in S_3:

```
sage: [sorted(p.sylvester_class(left_to_right=True))                        #␣
↪needs sage.combinat sage.graphs
```

(continues on next page)

(continued from previous page)

```
....:  for p in Permutations(3)]
[[[1, 2, 3]],
 [[1, 3, 2]],
 [[2, 1, 3], [2, 3, 1]],
 [[2, 1, 3], [2, 3, 1]],
 [[3, 1, 2]],
 [[3, 2, 1]]]
```

```
>>> from sage.all import *
>>> [sorted(p.sylvester_class(left_to_right=True))                         #␣
↪needs sage.combinat sage.graphs
...  for p in Permutations(Integer(3))]
[[[1, 2, 3]],
 [[1, 3, 2]],
 [[2, 1, 3], [2, 3, 1]],
 [[2, 1, 3], [2, 3, 1]],
 [[3, 1, 2]],
 [[3, 2, 1]]]
```

A left sylvester class in S_5:

```
sage: p = Permutation([4, 2, 1, 5, 3])
sage: sorted(p.sylvester_class(left_to_right=True))                         #␣
↪needs sage.combinat sage.graphs
[[4, 2, 1, 3, 5],
 [4, 2, 1, 5, 3],
 [4, 2, 3, 1, 5],
 [4, 2, 3, 5, 1],
 [4, 2, 5, 1, 3],
 [4, 2, 5, 3, 1],
 [4, 5, 2, 1, 3],
 [4, 5, 2, 3, 1]]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(4), Integer(2), Integer(1), Integer(5),␣
↪Integer(3)])
>>> sorted(p.sylvester_class(left_to_right=True))                          #␣
↪needs sage.combinat sage.graphs
[[4, 2, 1, 3, 5],
 [4, 2, 1, 5, 3],
 [4, 2, 3, 1, 5],
 [4, 2, 3, 5, 1],
 [4, 2, 5, 1, 3],
 [4, 2, 5, 3, 1],
 [4, 5, 2, 1, 3],
 [4, 5, 2, 3, 1]]
```

to_alternating_sign_matrix()

Return a matrix representing the permutation in the *AlternatingSignMatrix* class.

EXAMPLES:

```
sage: m = Permutation([1,2,3]).to_alternating_sign_matrix(); m                  #␣
↪needs sage.combinat sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
sage: parent(m)                                                                  #␣
↪needs sage.combinat sage.modules
Alternating sign matrices of size 3
```

```
>>> from sage.all import *
>>> m = Permutation([Integer(1),Integer(2),Integer(3)]).to_alternating_sign_
↪matrix(); m              # needs sage.combinat sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
>>> parent(m)                                                                   #␣
↪needs sage.combinat sage.modules
Alternating sign matrices of size 3
```

to_cycles(*singletons=True*, *use_min=True*)

Return the permutation `self` as a list of disjoint cycles.

The cycles are returned in the order of increasing smallest elements, and each cycle is returned as a tuple which starts with its smallest element.

If `singletons=False` is given, the list does not contain the singleton cycles.

If `use_min=False` is given, the cycles are returned in the order of increasing *largest* (not smallest) elements, and each cycle starts with its largest element.

EXAMPLES:

```
sage: Permutation([2,1,3,4]).to_cycles()
[(1, 2), (3,), (4,)]
sage: Permutation([2,1,3,4]).to_cycles(singletons=False)
[(1, 2)]
sage: Permutation([2,1,3,4]).to_cycles(use_min=True)
[(1, 2), (3,), (4,)]
sage: Permutation([2,1,3,4]).to_cycles(use_min=False)
[(4,), (3,), (2, 1)]
sage: Permutation([2,1,3,4]).to_cycles(singletons=False, use_min=False)
[(2, 1)]

sage: Permutation([4,1,5,2,6,3]).to_cycles()
[(1, 4, 2), (3, 5, 6)]
sage: Permutation([4,1,5,2,6,3]).to_cycles(use_min=False)
[(6, 3, 5), (4, 2, 1)]

sage: Permutation([6, 4, 5, 2, 3, 1]).to_cycles()
[(1, 6), (2, 4), (3, 5)]
sage: Permutation([6, 4, 5, 2, 3, 1]).to_cycles(use_min=False)
[(6, 1), (5, 3), (4, 2)]
```

```
>>> from sage.all import *
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_cycles()
[(1, 2), (3,), (4,)]
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_
↪cycles(singletons=False)
[(1, 2)]
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_cycles(use_
↪min=True)
[(1, 2), (3,), (4,)]
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_cycles(use_
↪min=False)
[(4,), (3,), (2, 1)]
>>> Permutation([Integer(2),Integer(1),Integer(3),Integer(4)]).to_
↪cycles(singletons=False, use_min=False)
[(2, 1)]

>>> Permutation([Integer(4),Integer(1),Integer(5),Integer(2),Integer(6),
↪Integer(3)]).to_cycles()
[(1, 4, 2), (3, 5, 6)]
>>> Permutation([Integer(4),Integer(1),Integer(5),Integer(2),Integer(6),
↪Integer(3)]).to_cycles(use_min=False)
[(6, 3, 5), (4, 2, 1)]

>>> Permutation([Integer(6), Integer(4), Integer(5), Integer(2), Integer(3),␣
↪Integer(1)]).to_cycles()
[(1, 6), (2, 4), (3, 5)]
>>> Permutation([Integer(6), Integer(4), Integer(5), Integer(2), Integer(3),␣
↪Integer(1)]).to_cycles(use_min=False)
[(6, 1), (5, 3), (4, 2)]
```

The algorithm is of complexity $O(n)$ where n is the size of the given permutation.

to_digraph()

Return a digraph representation of `self`.

EXAMPLES:

```
sage: d = Permutation([3, 1, 2]).to_digraph()                              #␣
↪needs sage.graphs
sage: d.edges(sort=True, labels=False)                                     #␣
↪needs sage.graphs
[(1, 3), (2, 1), (3, 2)]
sage: P = Permutations(range(1, 10))
sage: d = Permutation(P.random_element()).to_digraph()                     #␣
↪needs sage.graphs
sage: all(c.is_cycle()                                                     #␣
↪needs sage.graphs
....:     for c in d.strongly_connected_components_subgraphs())
True
```

```
>>> from sage.all import *
>>> d = Permutation([Integer(3), Integer(1), Integer(2)]).to_digraph()      ␣
↪                        # needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
>>> d.edges(sort=True, labels=False)                                        #␣
↪needs sage.graphs
[(1, 3), (2, 1), (3, 2)]
>>> P = Permutations(range(Integer(1), Integer(10)))
>>> d = Permutation(P.random_element()).to_digraph()                        #␣
↪needs sage.graphs
>>> all(c.is_cycle()                                                        #␣
↪needs sage.graphs
...     for c in d.strongly_connected_components_subgraphs())
True
```

to_inversion_vector()

Return the inversion vector of `self`.

The inversion vector of a permutation $p \in S_n$ is defined as the vector $(v_1, v_2, \ldots, v_n)$, where v_i is the number of elements larger than i that appear to the left of i in the permutation p.

The algorithm is of complexity $O(n \log(n))$ where n is the size of the given permutation.

EXAMPLES:

```
sage: Permutation([5,9,1,8,2,6,4,7,3]).to_inversion_vector()
[2, 3, 6, 4, 0, 2, 2, 1, 0]
sage: Permutation([8,7,2,1,9,4,6,5,10,3]).to_inversion_vector()
[3, 2, 7, 3, 4, 3, 1, 0, 0, 0]
sage: Permutation([3,2,4,1,5]).to_inversion_vector()
[3, 1, 0, 0, 0]
```

```
>>> from sage.all import *
>>> Permutation([Integer(5),Integer(9),Integer(1),Integer(8),Integer(2),
↪Integer(6),Integer(4),Integer(7),Integer(3)]).to_inversion_vector()
[2, 3, 6, 4, 0, 2, 2, 1, 0]
>>> Permutation([Integer(8),Integer(7),Integer(2),Integer(1),Integer(9),
↪Integer(4),Integer(6),Integer(5),Integer(10),Integer(3)]).to_inversion_
↪vector()
[3, 2, 7, 3, 4, 3, 1, 0, 0, 0]
>>> Permutation([Integer(3),Integer(2),Integer(4),Integer(1),Integer(5)]).to_
↪inversion_vector()
[3, 1, 0, 0, 0]
```

to_lehmer_cocode()

Return the Lehmer cocode of the permutation `self`.

The Lehmer cocode of a permutation p is defined as the list $(c_1, c_2, \ldots, c_n)$, where c_i is the number of $j < i$ such that $p(j) > p(i)$.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: p.to_lehmer_cocode()
[0, 1, 0]
sage: q = Permutation([3,1,2])
sage: q.to_lehmer_cocode()
[0, 1, 1]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2),Integer(1),Integer(3)])
>>> p.to_lehmer_cocode()
[0, 1, 0]
>>> q = Permutation([Integer(3),Integer(1),Integer(2)])
>>> q.to_lehmer_cocode()
[0, 1, 1]
```

to_lehmer_code()

Return the Lehmer code of the permutation `self`.

The Lehmer code of a permutation p is defined as the list $[c[1], c[2], ..., c[n]]$, where $c[i]$ is the number of $j > i$ such that $p(j) < p(i)$.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: p.to_lehmer_code()
[1, 0, 0]
sage: q = Permutation([3,1,2])
sage: q.to_lehmer_code()
[2, 0, 0]

sage: Permutation([1]).to_lehmer_code()
[0]
sage: Permutation([]).to_lehmer_code()
[]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(2),Integer(1),Integer(3)])
>>> p.to_lehmer_code()
[1, 0, 0]
>>> q = Permutation([Integer(3),Integer(1),Integer(2)])
>>> q.to_lehmer_code()
[2, 0, 0]

>>> Permutation([Integer(1)]).to_lehmer_code()
[0]
>>> Permutation([]).to_lehmer_code()
[]
```

to_major_code(*final_descent=False*)

Return the major code of the permutation `self`.

The major code of a permutation p is defined as the sequence $(m_1 - m_2, m_2 - m_3, \ldots, m_n)$, where m_i is the major index of the permutation obtained by erasing all letters smaller than i from p.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent. This has an effect on the computation of major indices.

REFERENCES:

- Carlitz, L. *q-Bernoulli and Eulerian Numbers*. Trans. Amer. Math. Soc. 76 (1954) 332-350. http://www.ams.org/journals/tran/1954-076-02/S0002-9947-1954-0060538-2/
- Skandera, M. *An Eulerian Partner for Inversions*. Sém. Lothar. Combin. 46 (2001) B46d. http://www.lehigh.edu/~mas906/papers/partner.ps

EXAMPLES:

```
sage: Permutation([9,3,5,7,2,1,4,6,8]).to_major_code()
[5, 0, 1, 0, 1, 2, 0, 1, 0]
sage: Permutation([2,8,4,3,6,7,9,5,1]).to_major_code()
[8, 3, 3, 1, 4, 0, 1, 0, 0]
```

```
>>> from sage.all import *
>>> Permutation([Integer(9),Integer(3),Integer(5),Integer(7),Integer(2),
↪Integer(1),Integer(4),Integer(6),Integer(8)]).to_major_code()
[5, 0, 1, 0, 1, 2, 0, 1, 0]
>>> Permutation([Integer(2),Integer(8),Integer(4),Integer(3),Integer(6),
↪Integer(7),Integer(9),Integer(5),Integer(1)]).to_major_code()
[8, 3, 3, 1, 4, 0, 1, 0, 0]
```

to_matrix()

Return a matrix representing the permutation.

EXAMPLES:

```
sage: Permutation([1,2,3]).to_matrix()                                   #␣
↪needs sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(2),Integer(3)]).to_matrix()            ␣
↪                         # needs sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
```

Alternatively:

```
sage: matrix(Permutation([1,3,2]))                                        #␣
↪needs sage.modules
[1 0 0]
[0 0 1]
[0 1 0]
```

```
>>> from sage.all import *
>>> matrix(Permutation([Integer(1),Integer(3),Integer(2)]))               ␣
↪                         # needs sage.modules
[1 0 0]
[0 0 1]
[0 1 0]
```

Notice that matrix multiplication corresponds to permutation multiplication only when the permutation option mult='r2l'

```
sage: Permutations.options.mult='r2l'
sage: p = Permutation([2,1,3])
```

(continues on next page)

(continued from previous page)

```
sage: q = Permutation([3,1,2])
sage: (p*q).to_matrix()                                                    #␣
↪needs sage.modules
[0 0 1]
[0 1 0]
[1 0 0]
sage: p.to_matrix()*q.to_matrix()                                          #␣
↪needs sage.modules
[0 0 1]
[0 1 0]
[1 0 0]
sage: Permutations.options.mult='l2r'
sage: (p*q).to_matrix()                                                    #␣
↪needs sage.modules
[1 0 0]
[0 0 1]
[0 1 0]
```

```
>>> from sage.all import *
>>> Permutations.options.mult='r2l'
>>> p = Permutation([Integer(2),Integer(1),Integer(3)])
>>> q = Permutation([Integer(3),Integer(1),Integer(2)])
>>> (p*q).to_matrix()                                                    #␣
↪needs sage.modules
[0 0 1]
[0 1 0]
[1 0 0]
>>> p.to_matrix()*q.to_matrix()                                          #␣
↪needs sage.modules
[0 0 1]
[0 1 0]
[1 0 0]
>>> Permutations.options.mult='l2r'
>>> (p*q).to_matrix()                                                    #␣
↪needs sage.modules
[1 0 0]
[0 0 1]
[0 1 0]
```

to_permutation_group_element()

Return a PermutationGroupElement equal to `self`.

EXAMPLES:

```
sage: Permutation([2,1,4,3]).to_permutation_group_element()                #␣
↪needs sage.groups
(1,2)(3,4)
sage: Permutation([1,2,3]).to_permutation_group_element()                  #␣
↪needs sage.groups
()
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> Permutation([Integer(2),Integer(1),Integer(4),Integer(3)]).to_permutation_
↪group_element()                    # needs sage.groups
(1,2)(3,4)
>>> Permutation([Integer(1),Integer(2),Integer(3)]).to_permutation_group_
↪element()                    # needs sage.groups
()
```

to_tableau_by_shape(*shape*)

Return a tableau of shape `shape` with the entries in `self`.

The tableau is such that the reading word (i. e., the word obtained by reading the tableau row by row, starting from the top row in English notation, with each row being read from left to right) is `self`.

EXAMPLES:

```
sage: T = Permutation([3,4,1,2,5]).to_tableau_by_shape([3,2]); T              #␣
↪needs sage.combinat
[[1, 2, 5], [3, 4]]
sage: T.reading_word_permutation()                                           #␣
↪needs sage.combinat
[3, 4, 1, 2, 5]
```

```
>>> from sage.all import *
>>> T = Permutation([Integer(3),Integer(4),Integer(1),Integer(2),Integer(5)]).
↪to_tableau_by_shape([Integer(3),Integer(2)]); T              # needs sage.
↪combinat
[[1, 2, 5], [3, 4]]
>>> T.reading_word_permutation()                                            #␣
↪needs sage.combinat
[3, 4, 1, 2, 5]
```

weak_excedences()

Return all the numbers `self[i]` such that `self[i] >= i+1`.

EXAMPLES:

```
sage: Permutation([1,4,3,2,5]).weak_excedences()
[1, 4, 3, 5]
```

```
>>> from sage.all import *
>>> Permutation([Integer(1),Integer(4),Integer(3),Integer(2),Integer(5)]).
↪weak_excedences()
[1, 4, 3, 5]
```

class sage.combinat.permutation.**Permutations**

Bases: `UniqueRepresentation`, `Parent`

Permutations.

`Permutations(n)` returns the class of permutations of `n`, if `n` is an integer, list, set, or string.

`Permutations(n, k)` returns the class of length-`k` partial permutations of `n` (where `n` is any of the above things); `k` must be a nonnegative integer. A length-k partial permutation of n is defined as a k-tuple of pairwise distinct elements of $\{1, 2, \ldots, n\}$.

Valid keyword arguments are: 'descents', 'bruhat_smaller', 'bruhat_greater', 'recoils_finer', 'recoils_fatter', 'recoils', and 'avoiding'. With the exception of 'avoiding', you cannot specify `n` or `k` along with a keyword.

`Permutations(descents=(list,n))` returns the class of permutations of n with descents in the positions specified by `list`. This uses the slightly nonstandard convention that the images of $1, 2, ..., n$ under the permutation are regarded as positions $0, 1, ..., n-1$, so for example the presence of 1 in `list` signifies that the permutations π should satisfy $\pi(2) > \pi(3)$. Note that `list` is supposed to be a list of positions of the descents, not the descents composition. It does *not* return the class of permutations with descents composition `list`.

`Permutations(bruhat_smaller=p)` and `Permutations(bruhat_greater=p)` return the class of permutations smaller-or-equal or greater-or-equal, respectively, than the given permutation `p` in the Bruhat order. (The Bruhat order is defined in `bruhat_lequal()`. It is also referred to as the *strong* Bruhat order.)

`Permutations(recoils=p)` returns the class of permutations whose recoils composition is `p`. Unlike the `descents=(list, n)` syntax, this actually takes a *composition* as input.

`Permutations(recoils_fatter=p)` and `Permutations(recoils_finer=p)` return the class of permutations whose recoils composition is fatter or finer, respectively, than the given composition `p`.

`Permutations(n, avoiding=P)` returns the class of permutations of `n` avoiding `P`. Here `P` may be a single permutation or a list of permutations; the returned class will avoid all patterns in `P`.

EXAMPLES:

```
sage: p = Permutations(3); p
Standard permutations of 3
sage: p.list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

```
>>> from sage.all import *
>>> p = Permutations(Integer(3)); p
Standard permutations of 3
>>> p.list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

```
sage: p = Permutations(3, 2); p
Permutations of {1,...,3} of length 2
sage: p.list()
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

```
>>> from sage.all import *
>>> p = Permutations(Integer(3), Integer(2)); p
Permutations of {1,...,3} of length 2
>>> p.list()
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

```
sage: p = Permutations(['c', 'a', 't']); p
Permutations of the set ['c', 'a', 't']
sage: p.list()
[['c', 'a', 't'],
 ['c', 't', 'a'],
 ['a', 'c', 't'],
 ['a', 't', 'c'],
 ['t', 'c', 'a'],
 ['t', 'a', 'c']]
```

```
>>> from sage.all import *
>>> p = Permutations(['c', 'a', 't']); p
Permutations of the set ['c', 'a', 't']
>>> p.list()
[['c', 'a', 't'],
 ['c', 't', 'a'],
 ['a', 'c', 't'],
 ['a', 't', 'c'],
 ['t', 'c', 'a'],
 ['t', 'a', 'c']]
```

```
sage: p = Permutations(['c', 'a', 't'], 2); p
Permutations of the set ['c', 'a', 't'] of length 2
sage: p.list()
[['c', 'a'], ['c', 't'], ['a', 'c'], ['a', 't'], ['t', 'c'], ['t', 'a']]
```

```
>>> from sage.all import *
>>> p = Permutations(['c', 'a', 't'], Integer(2)); p
Permutations of the set ['c', 'a', 't'] of length 2
>>> p.list()
[['c', 'a'], ['c', 't'], ['a', 'c'], ['a', 't'], ['t', 'c'], ['t', 'a']]
```

```
sage: p = Permutations([1,1,2]); p
Permutations of the multi-set [1, 1, 2]
sage: p.list()
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

```
>>> from sage.all import *
>>> p = Permutations([Integer(1),Integer(1),Integer(2)]); p
Permutations of the multi-set [1, 1, 2]
>>> p.list()
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

```
sage: p = Permutations([1,1,2], 2); p
Permutations of the multi-set [1, 1, 2] of length 2
sage: p.list()                                                      #␣
↪needs sage.libs.gap
[[1, 1], [1, 2], [2, 1]]
```

```
>>> from sage.all import *
>>> p = Permutations([Integer(1),Integer(1),Integer(2)], Integer(2)); p
Permutations of the multi-set [1, 1, 2] of length 2
>>> p.list()                                                        #␣
↪needs sage.libs.gap
[[1, 1], [1, 2], [2, 1]]
```

```
sage: p = Permutations(descents=([1], 4)); p
Standard permutations of 4 with descents [1]
sage: p.list()                                                      #␣
↪needs sage.graphs sage.modules
[[2, 4, 1, 3], [3, 4, 1, 2], [1, 4, 2, 3], [1, 3, 2, 4], [2, 3, 1, 4]]
```

```
>>> from sage.all import *
>>> p = Permutations(descents=([Integer(1)], Integer(4))); p
Standard permutations of 4 with descents [1]
>>> p.list()                                                            #␣
↪needs sage.graphs sage.modules
[[2, 4, 1, 3], [3, 4, 1, 2], [1, 4, 2, 3], [1, 3, 2, 4], [2, 3, 1, 4]]
```

```
sage: p = Permutations(bruhat_smaller=[1,3,2,4]); p
Standard permutations that are less than or equal
to [1, 3, 2, 4] in the Bruhat order
sage: p.list()
[[1, 2, 3, 4], [1, 3, 2, 4]]
```

```
>>> from sage.all import *
>>> p = Permutations(bruhat_smaller=[Integer(1),Integer(3),Integer(2),
↪Integer(4)]); p
Standard permutations that are less than or equal
to [1, 3, 2, 4] in the Bruhat order
>>> p.list()
[[1, 2, 3, 4], [1, 3, 2, 4]]
```

```
sage: p = Permutations(bruhat_greater=[4,2,3,1]); p
Standard permutations that are greater than or equal
to [4, 2, 3, 1] in the Bruhat order
sage: p.list()
[[4, 2, 3, 1], [4, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> p = Permutations(bruhat_greater=[Integer(4),Integer(2),Integer(3),
↪Integer(1)]); p
Standard permutations that are greater than or equal
to [4, 2, 3, 1] in the Bruhat order
>>> p.list()
[[4, 2, 3, 1], [4, 3, 2, 1]]
```

```
sage: p = Permutations(recoils_finer=[2,1]); p
Standard permutations whose recoils composition is finer than [2, 1]
sage: p.list()                                                          #␣
↪needs sage.graphs sage.modules
[[3, 1, 2], [1, 2, 3], [1, 3, 2]]
```

```
>>> from sage.all import *
>>> p = Permutations(recoils_finer=[Integer(2),Integer(1)]); p
Standard permutations whose recoils composition is finer than [2, 1]
>>> p.list()                                                            #␣
↪needs sage.graphs sage.modules
[[3, 1, 2], [1, 2, 3], [1, 3, 2]]
```

```
sage: p = Permutations(recoils_fatter=[2,1]); p
Standard permutations whose recoils composition is fatter than [2, 1]
sage: p.list()                                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs sage.modules
[[3, 1, 2], [3, 2, 1], [1, 3, 2]]
```

```
>>> from sage.all import *
>>> p = Permutations(recoils_fatter=[Integer(2),Integer(1)]); p
Standard permutations whose recoils composition is fatter than [2, 1]
>>> p.list()                                                                    #␣
↪needs sage.graphs sage.modules
[[3, 1, 2], [3, 2, 1], [1, 3, 2]]
```

```
sage: p = Permutations(recoils=[2,1]); p
Standard permutations whose recoils composition is [2, 1]
sage: p.list()                                                                  #␣
↪needs sage.graphs sage.modules
[[3, 1, 2], [1, 3, 2]]
```

```
>>> from sage.all import *
>>> p = Permutations(recoils=[Integer(2),Integer(1)]); p
Standard permutations whose recoils composition is [2, 1]
>>> p.list()                                                                    #␣
↪needs sage.graphs sage.modules
[[3, 1, 2], [1, 3, 2]]
```

```
sage: p = Permutations(4, avoiding=[1,3,2]); p
Standard permutations of 4 avoiding [[1, 3, 2]]
sage: p.list()
[[4, 1, 2, 3],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1],
 [3, 4, 1, 2],
 [3, 4, 2, 1],
 [2, 3, 4, 1],
 [3, 2, 4, 1],
 [1, 2, 3, 4],
 [2, 1, 3, 4],
 [2, 3, 1, 4],
 [3, 1, 2, 4],
 [3, 2, 1, 4]]
```

```
>>> from sage.all import *
>>> p = Permutations(Integer(4), avoiding=[Integer(1),Integer(3),Integer(2)]); p
Standard permutations of 4 avoiding [[1, 3, 2]]
>>> p.list()
[[4, 1, 2, 3],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1],
 [3, 4, 1, 2],
```

(continues on next page)

(continued from previous page)

```
[3, 4, 2, 1],
 [2, 3, 4, 1],
 [3, 2, 4, 1],
 [1, 2, 3, 4],
 [2, 1, 3, 4],
 [2, 3, 1, 4],
 [3, 1, 2, 4],
 [3, 2, 1, 4]]
```

```
sage: p = Permutations(5, avoiding=[[3,4,1,2], [4,2,3,1]]); p
Standard permutations of 5 avoiding [[3, 4, 1, 2], [4, 2, 3, 1]]
sage: p.cardinality()                                                    #␣
↪needs sage.combinat
88
sage: p.random_element().parent() is p                                   #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> p = Permutations(Integer(5), avoiding=[[Integer(3),Integer(4),Integer(1),
↪Integer(2)], [Integer(4),Integer(2),Integer(3),Integer(1)]]); p
Standard permutations of 5 avoiding [[3, 4, 1, 2], [4, 2, 3, 1]]
>>> p.cardinality()                                                    #␣
↪needs sage.combinat
88
>>> p.random_element().parent() is p                                   #␣
↪needs sage.combinat
True
```

Element

alias of `Permutation`

options = Current options for Permutations - display: list - generator_name: s - latex: list - latex_empty_str: 1 - mult: l2r

class sage.combinat.permutation.**PermutationsNK**(*s*, *k*)

Bases: `Permutations_setk`

This exists solely for unpickling `PermutationsNK` objects created with Sage <= 6.3.

class sage.combinat.permutation.**Permutations_mset**(*mset*)

Bases: `Permutations`

Permutations of a multiset M.

A permutation of a multiset M is represented by a list that contains exactly the same elements as M (with the same multiplicities), but possibly in different order. If M is a proper set there are $|M|!$ such permutations. Otherwise, if the first element appears k_1 times, the second element appears k_2 times and so on, the number of permutations is $|M|!/(k_1!k_2!\ldots)$, which is sometimes called a multinomial coefficient.

EXAMPLES:

```
sage: mset = [1,1,2,2,2]
sage: from sage.combinat.permutation import Permutations_mset
```

(continues on next page)

(continued from previous page)

```
sage: P = Permutations_mset(mset); P
Permutations of the multi-set [1, 1, 2, 2, 2]
sage: sorted(P)
[[1, 1, 2, 2, 2],
 [1, 2, 1, 2, 2],
 [1, 2, 2, 1, 2],
 [1, 2, 2, 2, 1],
 [2, 1, 1, 2, 2],
 [2, 1, 2, 1, 2],
 [2, 1, 2, 2, 1],
 [2, 2, 1, 1, 2],
 [2, 2, 1, 2, 1],
 [2, 2, 2, 1, 1]]

sage: # needs sage.modules
sage: MS = MatrixSpace(GF(2), 2, 2)
sage: A = MS([1,0,1,1])
sage: rows = A.rows()
sage: rows[0].set_immutable()
sage: rows[1].set_immutable()
sage: P = Permutations_mset(rows); P
Permutations of the multi-set [(1, 0), (1, 1)]
sage: sorted(P)
[[(1, 0), (1, 1)], [(1, 1), (1, 0)]]
```

```
>>> from sage.all import *
>>> mset = [Integer(1),Integer(1),Integer(2),Integer(2),Integer(2)]
>>> from sage.combinat.permutation import Permutations_mset
>>> P = Permutations_mset(mset); P
Permutations of the multi-set [1, 1, 2, 2, 2]
>>> sorted(P)
[[1, 1, 2, 2, 2],
 [1, 2, 1, 2, 2],
 [1, 2, 2, 1, 2],
 [1, 2, 2, 2, 1],
 [2, 1, 1, 2, 2],
 [2, 1, 2, 1, 2],
 [2, 1, 2, 2, 1],
 [2, 2, 1, 1, 2],
 [2, 2, 1, 2, 1],
 [2, 2, 2, 1, 1]]

>>> # needs sage.modules
>>> MS = MatrixSpace(GF(Integer(2)), Integer(2), Integer(2))
>>> A = MS([Integer(1),Integer(0),Integer(1),Integer(1)])
>>> rows = A.rows()
>>> rows[Integer(0)].set_immutable()
>>> rows[Integer(1)].set_immutable()
>>> P = Permutations_mset(rows); P
Permutations of the multi-set [(1, 0), (1, 1)]
>>> sorted(P)
[[(1, 0), (1, 1)], [(1, 1), (1, 0)]]
```

class Element

Bases: `ClonableArray`

A permutation of an arbitrary multiset.

check()

Verify that `self` is a valid permutation of the underlying multiset.

EXAMPLES:

```
sage: S = Permutations(['c','a','c'])
sage: elt = S(['c','c','a'])
sage: elt.check()
```

```
>>> from sage.all import *
>>> S = Permutations(['c','a','c'])
>>> elt = S(['c','c','a'])
>>> elt.check()
```

cardinality()

Return the cardinality of the set.

EXAMPLES:

```
sage: Permutations([1,2,2]).cardinality()
3
sage: Permutations([1,1,2,2,2]).cardinality()
10
```

```
>>> from sage.all import *
>>> Permutations([Integer(1),Integer(2),Integer(2)]).cardinality()
3
>>> Permutations([Integer(1),Integer(1),Integer(2),Integer(2),Integer(2)]).
↪cardinality()
10
```

rank(p)

Return the rank of `p` in lexicographic order.

INPUT:

- `p` – a permutation of M

ALGORITHM:

The algorithm uses the recurrence from the solution to exercise 4 in [Knu2011], Section 7.2.1.2:

$$\mathrm{rank}(p_1 \dots p_n) = \mathrm{rank}(p_2 \dots p_n) + \frac{1}{n}\binom{n}{n_1, \dots, n_t} \sum_{j=1}^{t} n_j \left[x_j < p_1\right],$$

where x_j, n_j are the distinct elements of p with their multiplicities, n is the sum of $n_1, \dots, n_t$, $\binom{n}{n_1,\dots,n_t}$ is the multinomial coefficient $\frac{n!}{n_1!\dots n_t!}$, and $\sum_{j=1}^{t} n_j \left[x_j < p_1\right]$ means "the number of elements to the right of the first element that are less than the first element".

EXAMPLES:

```
sage: mset = [1, 1, 2, 3, 4, 5, 5, 6, 9]
sage: p = Permutations(mset)
sage: p.rank(list(sorted(mset)))
0
sage: p.rank(list(reversed(sorted(mset)))) == p.cardinality() - 1
True
sage: p.rank([3, 1, 4, 1, 5, 9, 2, 6, 5])
30991
```

```
>>> from sage.all import *
>>> mset = [Integer(1), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5), Integer(5), Integer(6), Integer(9)]
>>> p = Permutations(mset)
>>> p.rank(list(sorted(mset)))
0
>>> p.rank(list(reversed(sorted(mset)))) == p.cardinality() - Integer(1)
True
>>> p.rank([Integer(3), Integer(1), Integer(4), Integer(1), Integer(5),␣
↪Integer(9), Integer(2), Integer(6), Integer(5)])
30991
```

unrank(*r*)

Return the permutation of M having lexicographic rank `r`.

INPUT:

- `r` – integer between `0` and `self.cardinality()-1` inclusive

ALGORITHM:

The algorithm is adapted from the solution to exercise 4 in [Knu2011], Section 7.2.1.2.

EXAMPLES:

```
sage: mset = [1, 1, 2, 3, 4, 5, 5, 6, 9]
sage: p = Permutations(mset)
sage: p.unrank(30991)
[3, 1, 4, 1, 5, 9, 2, 6, 5]
sage: p.rank(p.unrank(10))
10
sage: p.unrank(0) == list(sorted(mset))
True
sage: p.unrank(p.cardinality()-1) == list(reversed(sorted(mset)))
True
```

```
>>> from sage.all import *
>>> mset = [Integer(1), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5), Integer(5), Integer(6), Integer(9)]
>>> p = Permutations(mset)
>>> p.unrank(Integer(30991))
[3, 1, 4, 1, 5, 9, 2, 6, 5]
>>> p.rank(p.unrank(Integer(10)))
10
>>> p.unrank(Integer(0)) == list(sorted(mset))
True
```

(continues on next page)

(continued from previous page)

```
>>> p.unrank(p.cardinality()-Integer(1)) == list(reversed(sorted(mset)))
True
```

class sage.combinat.permutation.**Permutations_msetk**(*mset*, *k*)

Bases: *Permutations_mset*

Length-k partial permutations of a multiset.

A length-k partial permutation of a multiset M is represented by a list of length k whose entries are elements of M, appearing in the list with a multiplicity not higher than their respective multiplicity in M.

cardinality()

Return the cardinality of the set.

EXAMPLES:

```
sage: Permutations([1,2,2], 2).cardinality()                              #␣
↪needs sage.libs.gap
3
```

```
>>> from sage.all import *
>>> Permutations([Integer(1),Integer(2),Integer(2)], Integer(2)).
↪cardinality()                                  # needs sage.libs.gap
3
```

class sage.combinat.permutation.**Permutations_nk**(*n*, *k*)

Bases: *Permutations*

Length-k partial permutations of $\{1, 2, \ldots, n\}$.

class Element

Bases: ClonableArray

A length-k partial permutation of $[n]$.

check()

Verify that self is a valid length-k partial permutation of $[n]$.

EXAMPLES:

```
sage: S = Permutations(4, 2)
sage: elt = S([3, 1])
sage: elt.check()
```

```
>>> from sage.all import *
>>> S = Permutations(Integer(4), Integer(2))
>>> elt = S([Integer(3), Integer(1)])
>>> elt.check()
```

cardinality()

EXAMPLES:

```
sage: Permutations(3,0).cardinality()
1
sage: Permutations(3,1).cardinality()
```

(continues on next page)

(continued from previous page)

```
3
sage: Permutations(3,2).cardinality()
6
sage: Permutations(3,3).cardinality()
6
sage: Permutations(3,4).cardinality()
0
```

```
>>> from sage.all import *
>>> Permutations(Integer(3),Integer(0)).cardinality()
1
>>> Permutations(Integer(3),Integer(1)).cardinality()
3
>>> Permutations(Integer(3),Integer(2)).cardinality()
6
>>> Permutations(Integer(3),Integer(3)).cardinality()
6
>>> Permutations(Integer(3),Integer(4)).cardinality()
0
```

random_element()

EXAMPLES:

```
sage: s = Permutations(3,2).random_element()
sage: s in Permutations(3,2)
True
```

```
>>> from sage.all import *
>>> s = Permutations(Integer(3),Integer(2)).random_element()
>>> s in Permutations(Integer(3),Integer(2))
True
```

class sage.combinat.permutation.**Permutations_set**(*s*)

Bases: *Permutations*

Permutations of an arbitrary given finite set.

Here, a "permutation of a finite set S" means a list of the elements of S in which every element of S occurs exactly once. This is not to be confused with bijections from S to S, which are also often called permutations in literature.

class Element

Bases: ClonableArray

A permutation of an arbitrary set.

check()

Verify that `self` is a valid permutation of the underlying set.

EXAMPLES:

```
sage: S = Permutations(['c','a','t'])
sage: elt = S(['t','c','a'])
sage: elt.check()
```

```
>>> from sage.all import *
>>> S = Permutations(['c','a','t'])
>>> elt = S(['t','c','a'])
>>> elt.check()
```

cardinality()

Return the cardinality of the set.

EXAMPLES:

```
sage: Permutations([1,2,3]).cardinality()
6
```

```
>>> from sage.all import *
>>> Permutations([Integer(1),Integer(2),Integer(3)]).cardinality()
6
```

random_element()

EXAMPLES:

```
sage: s = Permutations([1,2,3]).random_element()
sage: s.parent() is Permutations([1,2,3])
True
```

```
>>> from sage.all import *
>>> s = Permutations([Integer(1),Integer(2),Integer(3)]).random_element()
>>> s.parent() is Permutations([Integer(1),Integer(2),Integer(3)])
True
```

class sage.combinat.permutation.**Permutations_setk**(*s*, *k*)

Bases: *Permutations_set*

Length-k partial permutations of an arbitrary given finite set.

Here, a "length-k partial permutation of a finite set S" means a list of length k whose entries are pairwise distinct and all belong to S.

random_element()

EXAMPLES:

```
sage: s = Permutations([1,2,4], 2).random_element()
sage: s in Permutations([1,2,4], 2)
True
```

```
>>> from sage.all import *
>>> s = Permutations([Integer(1),Integer(2),Integer(4)], Integer(2)).random_
↪element()
>>> s in Permutations([Integer(1),Integer(2),Integer(4)], Integer(2))
True
```

class sage.combinat.permutation.**StandardPermutations_all**

Bases: *Permutations*

All standard permutations.

graded_component(*n*)

Return the graded component.

EXAMPLES:

```
sage: P = Permutations()
sage: P.graded_component(4) == Permutations(4)
True
```

```
>>> from sage.all import *
>>> P = Permutations()
>>> P.graded_component(Integer(4)) == Permutations(Integer(4))
True
```

class sage.combinat.permutation.**StandardPermutations_all_avoiding**(*a*)

Bases: *StandardPermutations_all*

All standard permutations avoiding a set of patterns.

patterns()

Return the patterns avoided by this class of permutations.

EXAMPLES:

```
sage: P = Permutations(avoiding=[[2,1,3],[1,2,3]])
sage: P.patterns()
([2, 1, 3], [1, 2, 3])
```

```
>>> from sage.all import *
>>> P = Permutations(avoiding=[[Integer(2),Integer(1),Integer(3)],[Integer(1),
↪Integer(2),Integer(3)]])
>>> P.patterns()
([2, 1, 3], [1, 2, 3])
```

class sage.combinat.permutation.**StandardPermutations_avoiding_12**(*n*)

Bases: *StandardPermutations_avoiding_generic*

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Permutations(3, avoiding=[1, 2])
sage: P.cardinality()
1
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(3), avoiding=[Integer(1), Integer(2)])
>>> P.cardinality()
1
```

class sage.combinat.permutation.**StandardPermutations_avoiding_123**(*n*)

Bases: *StandardPermutations_avoiding_generic*

cardinality()

EXAMPLES:

```
sage: Permutations(5, avoiding=[1, 2, 3]).cardinality()
42
sage: len( Permutations(5, avoiding=[1, 2, 3]).list() )
42
```

```
>>> from sage.all import *
>>> Permutations(Integer(5), avoiding=[Integer(1), Integer(2), Integer(3)]).
→cardinality()
42
>>> len( Permutations(Integer(5), avoiding=[Integer(1), Integer(2),␣
→Integer(3)]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_132**(*n*)

Bases: *StandardPermutations_avoiding_generic*

cardinality()

EXAMPLES:

```
sage: Permutations(5, avoiding=[1, 3, 2]).cardinality()
42
sage: len( Permutations(5, avoiding=[1, 3, 2]).list() )
42
```

```
>>> from sage.all import *
>>> Permutations(Integer(5), avoiding=[Integer(1), Integer(3), Integer(2)]).
→cardinality()
42
>>> len( Permutations(Integer(5), avoiding=[Integer(1), Integer(3),␣
→Integer(2)]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_21**(*n*)

Bases: *StandardPermutations_avoiding_generic*

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Permutations(3, avoiding=[2, 1])
sage: P.cardinality()
1
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(3), avoiding=[Integer(2), Integer(1)])
>>> P.cardinality()
1
```

class sage.combinat.permutation.**StandardPermutations_avoiding_213**(*n*)

Bases: *StandardPermutations_avoiding_generic*

cardinality()

EXAMPLES:

```
sage: Permutations(5, avoiding=[2, 1, 3]).cardinality()
42
sage: len( Permutations(5, avoiding=[2, 1, 3]).list() )
42
```

```
>>> from sage.all import *
>>> Permutations(Integer(5), avoiding=[Integer(2), Integer(1), Integer(3)]).
↪cardinality()
42
>>> len( Permutations(Integer(5), avoiding=[Integer(2), Integer(1),␣
↪Integer(3)]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_231**(*n*)

Bases: *StandardPermutations_avoiding_generic*

cardinality()

EXAMPLES:

```
sage: Permutations(5, avoiding=[2, 3, 1]).cardinality()
42
sage: len( Permutations(5, avoiding=[2, 3, 1]).list() )
42
```

```
>>> from sage.all import *
>>> Permutations(Integer(5), avoiding=[Integer(2), Integer(3), Integer(1)]).
↪cardinality()
42
>>> len( Permutations(Integer(5), avoiding=[Integer(2), Integer(3),␣
↪Integer(1)]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_312**(*n*)

Bases: *StandardPermutations_avoiding_generic*

cardinality()

EXAMPLES:

```
sage: Permutations(5, avoiding=[3, 1, 2]).cardinality()
42
sage: len( Permutations(5, avoiding=[3, 1, 2]).list() )
42
```

```
>>> from sage.all import *
>>> Permutations(Integer(5), avoiding=[Integer(3), Integer(1), Integer(2)]).
↪cardinality()
42
>>> len( Permutations(Integer(5), avoiding=[Integer(3), Integer(1),␣
↪Integer(2)]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_321**(*n*)

Bases: *StandardPermutations_avoiding_generic*

cardinality()

EXAMPLES:

```
sage: Permutations(5, avoiding=[3, 2, 1]).cardinality()
42
sage: len( Permutations(5, avoiding=[3, 2, 1]).list() )
42
```

```
>>> from sage.all import *
>>> Permutations(Integer(5), avoiding=[Integer(3), Integer(2), Integer(1)]).
↪cardinality()
42
>>> len( Permutations(Integer(5), avoiding=[Integer(3), Integer(2),␣
↪Integer(1)]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_generic**(*n*, *a*)

Bases: *StandardPermutations_n_abstract*

Generic class for subset of permutations avoiding a set of patterns.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Permutations(3, avoiding=[[2, 1, 3],[1,2,3]])
sage: P.cardinality()                                                    #␣
↪needs sage.combinat
4
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(3), avoiding=[[Integer(2), Integer(1),␣
↪Integer(3)],[Integer(1),Integer(2),Integer(3)]])
>>> P.cardinality()                                                    #␣
↪needs sage.combinat
4
```

patterns()

Return the patterns avoided by this class of permutations.

EXAMPLES:

```
sage: P = Permutations(3, avoiding=[[2,1,3],[1,2,3]])
sage: P.patterns()
([2, 1, 3], [1, 2, 3])
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(3), avoiding=[[Integer(2),Integer(1),Integer(3)],
↪[Integer(1),Integer(2),Integer(3)]])
>>> P.patterns()
([2, 1, 3], [1, 2, 3])
```

class sage.combinat.permutation.**StandardPermutations_bruhat_greater**(*p*)

Bases: *Permutations*

Permutations of $\{1, \ldots, n\}$ that are greater than or equal to a permutation p in the Bruhat order.

class sage.combinat.permutation.**StandardPermutations_bruhat_smaller**(*p*)

Bases: *Permutations*

Permutations of $\{1, \ldots, n\}$ that are less than or equal to a permutation p in the Bruhat order.

class sage.combinat.permutation.**StandardPermutations_descents**(*d*, *n*)

Bases: *StandardPermutations_n_abstract*

Permutations of $\{1, \ldots, n\}$ with a fixed set of descents.

cardinality()

Return the cardinality of `self`.

ALGORITHM:

The algorithm described in [Vie1979] is implemented naively.

EXAMPLES:

```
sage: P = Permutations(descents=([1,0,2], 5))
sage: P.cardinality()
4
```

```
>>> from sage.all import *
>>> P = Permutations(descents=([Integer(1),Integer(0),Integer(2)],␣
↪Integer(5)))
>>> P.cardinality()
4
```

first()

Return the first permutation with descents d.

EXAMPLES:

```
sage: Permutations(descents=([1,0,4,8],12)).first()
[3, 2, 1, 4, 6, 5, 7, 8, 10, 9, 11, 12]
```

```
>>> from sage.all import *
>>> Permutations(descents=([Integer(1),Integer(0),Integer(4),Integer(8)],
↪Integer(12))).first()
[3, 2, 1, 4, 6, 5, 7, 8, 10, 9, 11, 12]
```

last()

Return the last permutation with descents d.

EXAMPLES:

```
sage: Permutations(descents=([1,0,4,8],12)).last()
[12, 11, 8, 9, 10, 4, 5, 6, 7, 1, 2, 3]
```

```
>>> from sage.all import *
>>> Permutations(descents=([Integer(1),Integer(0),Integer(4),Integer(8)],
↪Integer(12))).last()
[12, 11, 8, 9, 10, 4, 5, 6, 7, 1, 2, 3]
```

class sage.combinat.permutation.**StandardPermutations_n**(*n*)

Bases: *StandardPermutations_n_abstract*

Permutations of the set $\{1, 2, \ldots, n\}$.

These are also called permutations of size n, or the elements of the n-th symmetric group.

Todo

Have a `reduced_word()` which works in both multiplication conventions.

class Element(*parent*, *l*, *algorithm='lex'*, *sjt=None*, *check=True*)

Bases: *Permutation*

apply_simple_reflection_left(*i*)

Return `self` multiplied by the simple reflection `s[i]` on the left.

This acts by switching the entries in positions i and $i + 1$.

Warning

This ignores the multiplication convention in order to be consistent with other Coxeter operations in permutations (e.g., computing `reduced_word()`).

EXAMPLES:

```
sage: W = Permutations(3)
sage: w = W([2,3,1])
sage: w.apply_simple_reflection_left(1)
[1, 3, 2]
sage: w.apply_simple_reflection_left(2)
[3, 2, 1]
```

```
>>> from sage.all import *
>>> W = Permutations(Integer(3))
>>> w = W([Integer(2),Integer(3),Integer(1)])
>>> w.apply_simple_reflection_left(Integer(1))
[1, 3, 2]
>>> w.apply_simple_reflection_left(Integer(2))
[3, 2, 1]
```

apply_simple_reflection_right(*i*)

Return `self` multiplied by the simple reflection `s[i]` on the right.

This acts by switching the entries i and $i + 1$.

> **Warning:** This ignores the multiplication convention in order to be consistent with other Coxeter operations in permutations (e.g., computing `reduced_word()`).

EXAMPLES:

```
sage: W = Permutations(3)
sage: w = W([2,3,1])
sage: w.apply_simple_reflection_right(1)
[3, 2, 1]
sage: w.apply_simple_reflection_right(2)
[2, 1, 3]
```

```
>>> from sage.all import *
>>> W = Permutations(Integer(3))
>>> w = W([Integer(2),Integer(3),Integer(1)])
>>> w.apply_simple_reflection_right(Integer(1))
[3, 2, 1]
>>> w.apply_simple_reflection_right(Integer(2))
[2, 1, 3]
```

has_left_descent(i)

Check if `i` is a left descent of `self`.

A *left descent* of a permutation $\pi \in S_n$ means an index $i \in \{1, 2, \ldots, n-1\}$ such that $s_i \circ \pi$ has smaller length than π. Thus, a left descent of π is an index $i \in \{1, 2, \ldots, n-1\}$ satisfying $\pi^{-1}(i) > \pi^{-1}(i+1)$.

> **Warning:** The methods `descents()` and `idescents()` behave differently than their Weyl group counterparts. In particular, the indexing is 0-based. This could lead to errors. Instead, construct the descent set as in the example.

> **Warning:** This ignores the multiplication convention in order to be consistent with other Coxeter operations in permutations (e.g., computing `reduced_word()`).

EXAMPLES:

```
sage: P = Permutations(4)
sage: x = P([3, 2, 4, 1])
sage: (~x).descents()
[1, 2]
sage: [i for i in P.index_set() if x.has_left_descent(i)]
[1, 2]
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> x = P([Integer(3), Integer(2), Integer(4), Integer(1)])
>>> (~x).descents()
[1, 2]
>>> [i for i in P.index_set() if x.has_left_descent(i)]
[1, 2]
```

has_right_descent(*i*, *mult=None*)

Check if `i` is a right descent of `self`.

A *right descent* of a permutation $\pi \in S_n$ means an index $i \in \{1, 2, \ldots, n-1\}$ such that $\pi \circ s_i$ has smaller length than π. Thus, a right descent of π is an index $i \in \{1, 2, \ldots, n-1\}$ satisfying $\pi(i) > \pi(i+1)$.

> **Warning**
>
> The methods `descents()` and `idescents()` behave differently than their Weyl group counterparts. In particular, the indexing is 0-based. This could lead to errors. Instead, construct the descent set as in the example.

> **Warning**
>
> This ignores the multiplication convention in order to be consistent with other Coxeter operations in permutations (e.g., computing `reduced_word()`).

EXAMPLES:

```
sage: P = Permutations(4)
sage: x = P([3, 2, 4, 1])
sage: x.descents()
[1, 3]
sage: [i for i in P.index_set() if x.has_right_descent(i)]
[1, 3]
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
>>> x = P([Integer(3), Integer(2), Integer(4), Integer(1)])
>>> x.descents()
[1, 3]
>>> [i for i in P.index_set() if x.has_right_descent(i)]
[1, 3]
```

inverse()

Return the inverse of `self`.

EXAMPLES:

```
sage: P = Permutations(4)
sage: w0 = P([4,3,2,1])
sage: w0.inverse() == w0
True
sage: w0.inverse().parent() is P
```

(continues on next page)

(continued from previous page)

```
True
sage: P([3,2,4,1]).inverse()
[4, 2, 1, 3]
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
>>> w0 = P([Integer(4),Integer(3),Integer(2),Integer(1)])
>>> w0.inverse() == w0
True
>>> w0.inverse().parent() is P
True
>>> P([Integer(3),Integer(2),Integer(4),Integer(1)]).inverse()
[4, 2, 1, 3]
```

algebra(*base_ring*, *category=None*)

Return the symmetric group algebra associated to `self`.

INPUT:

- `base_ring` – a ring
- `category` – a category (default: the category of `self`)

EXAMPLES:

```
sage: # needs sage.groups sage.modules
sage: P = Permutations(4)
sage: A = P.algebra(QQ); A
Symmetric group algebra of order 4 over Rational Field
sage: A.category()
Join of Category of Coxeter group algebras over Rational Field
    and Category of finite group algebras over Rational Field
    and Category of finite dimensional cellular algebras
        with basis over Rational Field
sage: A = P.algebra(QQ, category=Monoids())
sage: A.category()
Category of finite dimensional cellular monoid algebras over Rational Field
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> P = Permutations(Integer(4))
>>> A = P.algebra(QQ); A
Symmetric group algebra of order 4 over Rational Field
>>> A.category()
Join of Category of Coxeter group algebras over Rational Field
    and Category of finite group algebras over Rational Field
    and Category of finite dimensional cellular algebras
        with basis over Rational Field
>>> A = P.algebra(QQ, category=Monoids())
>>> A.category()
Category of finite dimensional cellular monoid algebras over Rational Field
```

as_permutation_group()

Return `self` as a permutation group.

EXAMPLES:

```
sage: P = Permutations(4)
sage: PG = P.as_permutation_group(); PG                                    #␣
↪needs sage.groups
Symmetric group of order 4! as a permutation group

sage: G = SymmetricGroup(4)                                                #␣
↪needs sage.groups
sage: PG is G                                                              #␣
↪needs sage.groups
True
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
>>> PG = P.as_permutation_group(); PG                                    #␣
↪needs sage.groups
Symmetric group of order 4! as a permutation group

>>> G = SymmetricGroup(Integer(4))                                         ␣
↪     # needs sage.groups
>>> PG is G                                                              #␣
↪needs sage.groups
True
```

cardinality()

Return the number of permutations of size n, which is $n!$.

EXAMPLES:

```
sage: Permutations(0).cardinality()
1
sage: Permutations(3).cardinality()
6
sage: Permutations(4).cardinality()
24
```

```
>>> from sage.all import *
>>> Permutations(Integer(0)).cardinality()
1
>>> Permutations(Integer(3)).cardinality()
6
>>> Permutations(Integer(4)).cardinality()
24
```

cartan_type()

Return the Cartan type of `self`.

The symmetric group S_n is a Coxeter group of type A_{n-1}.

EXAMPLES:

```
sage: A = SymmetricGroup([2,3,7]); A.cartan_type()                         #␣
↪needs sage.combinat sage.groups
```

(continues on next page)

(continued from previous page)

```
['A', 2]
sage: A = SymmetricGroup([]); A.cartan_type()                                   #␣
↪needs sage.combinat sage.groups
['A', 0]
```

```
>>> from sage.all import *
>>> A = SymmetricGroup([Integer(2),Integer(3),Integer(7)]); A.cartan_type()    ␣
↪                       # needs sage.combinat sage.groups
['A', 2]
>>> A = SymmetricGroup([]); A.cartan_type()                                   #␣
↪needs sage.combinat sage.groups
['A', 0]
```

codegrees()

Return the codegrees of `self`.

EXAMPLES:

```
sage: Permutations(3).codegrees()
(0, 1)
sage: Permutations(7).codegrees()
(0, 1, 2, 3, 4, 5)
```

```
>>> from sage.all import *
>>> Permutations(Integer(3)).codegrees()
(0, 1)
>>> Permutations(Integer(7)).codegrees()
(0, 1, 2, 3, 4, 5)
```

conjugacy_class(*g*)

Return the conjugacy class of `g` in `self`.

INPUT:

- `g` – a partition or an element of `self`

EXAMPLES:

```
sage: G = Permutations(5)
sage: g = G([2,3,4,1,5])
sage: G.conjugacy_class(g)                                                      #␣
↪needs sage.combinat sage.graphs sage.groups
Conjugacy class of cycle type [4, 1] in Standard permutations of 5
sage: G.conjugacy_class(Partition([2, 1, 1, 1]))                                #␣
↪needs sage.combinat sage.graphs sage.groups
Conjugacy class of cycle type [2, 1, 1, 1] in Standard permutations of 5
```

```
>>> from sage.all import *
>>> G = Permutations(Integer(5))
>>> g = G([Integer(2),Integer(3),Integer(4),Integer(1),Integer(5)])
>>> G.conjugacy_class(g)                                                      #␣
↪needs sage.combinat sage.graphs sage.groups
Conjugacy class of cycle type [4, 1] in Standard permutations of 5
```

(continues on next page)

(continued from previous page)

```
>>> G.conjugacy_class(Partition([Integer(2), Integer(1), Integer(1),␣
↪Integer(1)]))                                  # needs sage.combinat sage.graphs␣
↪sage.groups
Conjugacy class of cycle type [2, 1, 1, 1] in Standard permutations of 5
```

conjugacy_classes()

Return a list of the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = Permutations(4)
sage: G.conjugacy_classes()                                                    #␣
↪needs sage.combinat sage.graphs sage.groups
[Conjugacy class of cycle type [1, 1, 1, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [2, 1, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [2, 2] in Standard permutations of 4,
 Conjugacy class of cycle type [3, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [4] in Standard permutations of 4]
```

```
>>> from sage.all import *
>>> G = Permutations(Integer(4))
>>> G.conjugacy_classes()                                                    #␣
↪needs sage.combinat sage.graphs sage.groups
[Conjugacy class of cycle type [1, 1, 1, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [2, 1, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [2, 2] in Standard permutations of 4,
 Conjugacy class of cycle type [3, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [4] in Standard permutations of 4]
```

conjugacy_classes_iterator()

Iterate over the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = Permutations(4)
sage: list(G.conjugacy_classes_iterator()) == G.conjugacy_classes()          #␣
↪needs sage.combinat sage.graphs sage.groups
True
```

```
>>> from sage.all import *
>>> G = Permutations(Integer(4))
>>> list(G.conjugacy_classes_iterator()) == G.conjugacy_classes()          #␣
↪needs sage.combinat sage.graphs sage.groups
True
```

conjugacy_classes_representatives()

Return a complete list of representatives of conjugacy classes in `self`.

Let S_n be the symmetric group on n letters. The conjugacy classes are indexed by partitions λ of n. The ordering of the conjugacy classes is reverse lexicographic order of the partitions.

EXAMPLES:

```
sage: G = Permutations(5)
sage: G.conjugacy_classes_representatives()                                 #␣
↪needs sage.combinat sage.libs.flint
[[1, 2, 3, 4, 5],
 [2, 1, 3, 4, 5],
 [2, 1, 4, 3, 5],
 [2, 3, 1, 4, 5],
 [2, 3, 1, 5, 4],
 [2, 3, 4, 1, 5],
 [2, 3, 4, 5, 1]]
```

```
>>> from sage.all import *
>>> G = Permutations(Integer(5))
>>> G.conjugacy_classes_representatives()                                 #␣
↪needs sage.combinat sage.libs.flint
[[1, 2, 3, 4, 5],
 [2, 1, 3, 4, 5],
 [2, 1, 4, 3, 5],
 [2, 3, 1, 4, 5],
 [2, 3, 1, 5, 4],
 [2, 3, 4, 1, 5],
 [2, 3, 4, 5, 1]]
```

degree()

Return the degree of `self`.

This is the cardinality n of the set `self` acts on.

EXAMPLES:

```
sage: Permutations(0).degree()
0
sage: Permutations(1).degree()
1
sage: Permutations(5).degree()
5
```

```
>>> from sage.all import *
>>> Permutations(Integer(0)).degree()
0
>>> Permutations(Integer(1)).degree()
1
>>> Permutations(Integer(5)).degree()
5
```

degrees()

Return the degrees of `self`.

These are the degrees of the fundamental invariants of the ring of polynomial invariants.

EXAMPLES:

```
sage: Permutations(3).degrees()
(2, 3)
```

(continues on next page)

(continued from previous page)

```
sage: Permutations(7).degrees()
(2, 3, 4, 5, 6, 7)
```

```
>>> from sage.all import *
>>> Permutations(Integer(3)).degrees()
(2, 3)
>>> Permutations(Integer(7)).degrees()
(2, 3, 4, 5, 6, 7)
```

element_in_conjugacy_classes(*nu*)

Return a permutation with cycle type `nu`.

If the size of `nu` is smaller than the size of permutations in `self`, then some fixed points are added.

EXAMPLES:

```
sage: PP = Permutations(5)
sage: PP.element_in_conjugacy_classes([2,2])                                    #␣
↪needs sage.combinat
[2, 1, 4, 3, 5]
sage: PP.element_in_conjugacy_classes([5, 5])                                   #␣
↪needs sage.combinat
Traceback (most recent call last):
...
ValueError: the size of the partition (=10) should be at most the size of the␣
↪permutations (=5)
```

```
>>> from sage.all import *
>>> PP = Permutations(Integer(5))
>>> PP.element_in_conjugacy_classes([Integer(2),Integer(2)])                     ␣
↪               # needs sage.combinat
[2, 1, 4, 3, 5]
>>> PP.element_in_conjugacy_classes([Integer(5), Integer(5)])                    ␣
↪               # needs sage.combinat
Traceback (most recent call last):
...
ValueError: the size of the partition (=10) should be at most the size of the␣
↪permutations (=5)
```

gens()

Return a set of generators for `self` as a group.

EXAMPLES:

```
sage: P4 = Permutations(4)
sage: P4.gens()
([2, 1, 3, 4], [1, 3, 2, 4], [1, 2, 4, 3])
```

```
>>> from sage.all import *
>>> P4 = Permutations(Integer(4))
>>> P4.gens()
([2, 1, 3, 4], [1, 3, 2, 4], [1, 2, 4, 3])
```

identity()

Return the identity permutation of size n.

EXAMPLES:

```
sage: Permutations(4).identity()
[1, 2, 3, 4]
sage: Permutations(0).identity()
[]
```

```
>>> from sage.all import *
>>> Permutations(Integer(4)).identity()
[1, 2, 3, 4]
>>> Permutations(Integer(0)).identity()
[]
```

index_set()

Return the index set for the descents of the symmetric group `self`.

This is $\{1, 2, \ldots, n-1\}$, where `self` is S_n.

EXAMPLES:

```
sage: P = Permutations(8)
sage: P.index_set()
(1, 2, 3, 4, 5, 6, 7)
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(8))
>>> P.index_set()
(1, 2, 3, 4, 5, 6, 7)
```

one()

Return the identity permutation of size n.

EXAMPLES:

```
sage: Permutations(4).identity()
[1, 2, 3, 4]
sage: Permutations(0).identity()
[]
```

```
>>> from sage.all import *
>>> Permutations(Integer(4)).identity()
[1, 2, 3, 4]
>>> Permutations(Integer(0)).identity()
[]
```

random_element()

EXAMPLES:

```
sage: s = Permutations(4).random_element(); s  # random
[1, 2, 4, 3]
sage: s in Permutations(4)
True
```

```
>>> from sage.all import *
>>> s = Permutations(Integer(4)).random_element(); s   # random
[1, 2, 4, 3]
>>> s in Permutations(Integer(4))
True
```

rank(*p=None*)

Return the rank of `self` or `p` depending on input.

If a permutation `p` is given, return the rank of `p` in `self`. Otherwise return the dimension of the underlying vector space spanned by the (simple) roots.

EXAMPLES:

```
sage: P = Permutations(5)
sage: P.rank()
4

sage: SP3 = Permutations(3)
sage: list(map(SP3.rank, SP3))
[0, 1, 2, 3, 4, 5]
sage: SP0 = Permutations(0)
sage: list(map(SP0.rank, SP0))
[0]
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(5))
>>> P.rank()
4

>>> SP3 = Permutations(Integer(3))
>>> list(map(SP3.rank, SP3))
[0, 1, 2, 3, 4, 5]
>>> SP0 = Permutations(Integer(0))
>>> list(map(SP0.rank, SP0))
[0]
```

reflection(*i*)

Return the reflection indexed by `i` of `self`.

This returns the permutation with cycle $i = (a, b)$.

See also

- `reflections_index_set()`
- `reflections()`

EXAMPLES:

```
sage: P = Permutations(4)
sage: for i in P.reflection_index_set():
....:     print('%s %s'%(i, P.reflection(i)))
```

(continues on next page)

(continued from previous page)

```
(1, 2) [2, 1, 3, 4]
(1, 3) [3, 2, 1, 4]
(1, 4) [4, 2, 3, 1]
(2, 3) [1, 3, 2, 4]
(2, 4) [1, 4, 3, 2]
(3, 4) [1, 2, 4, 3]
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
>>> for i in P.reflection_index_set():
...     print('%s %s'%(i, P.reflection(i)))
(1, 2) [2, 1, 3, 4]
(1, 3) [3, 2, 1, 4]
(1, 4) [4, 2, 3, 1]
(2, 3) [1, 3, 2, 4]
(2, 4) [1, 4, 3, 2]
(3, 4) [1, 2, 4, 3]
```

reflection_index_set()

Return the index set of the reflections of `self`.

See also

- *reflection()*
- reflections()

EXAMPLES:

```
sage: P = Permutations(4)
sage: P.reflection_index_set()
((1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4))
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
>>> P.reflection_index_set()
((1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4))
```

simple_reflection(i)

For i in the index set of `self` (that is, for i in $\{1, 2, \ldots, n-1\}$, where `self` is S_n), this returns the elementary transposition $s_i = (i, i + 1)$.

EXAMPLES:

```
sage: P = Permutations(4)
sage: P.simple_reflection(2)
[1, 3, 2, 4]
sage: P.simple_reflections()
Finite family {1: [2, 1, 3, 4], 2: [1, 3, 2, 4], 3: [1, 2, 4, 3]}
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(4))
>>> P.simple_reflection(Integer(2))
[1, 3, 2, 4]
>>> P.simple_reflections()
Finite family {1: [2, 1, 3, 4], 2: [1, 3, 2, 4], 3: [1, 2, 4, 3]}
```

unrank(*r*)

EXAMPLES:

```
sage: SP3 = Permutations(3)
sage: l = list(map(SP3.unrank, range(6)))
sage: l == SP3.list()
True
sage: SP0 = Permutations(0)
sage: l = list(map(SP0.unrank, range(1)))
sage: l == SP0.list()
True
```

```
>>> from sage.all import *
>>> SP3 = Permutations(Integer(3))
>>> l = list(map(SP3.unrank, range(Integer(6))))
>>> l == SP3.list()
True
>>> SP0 = Permutations(Integer(0))
>>> l = list(map(SP0.unrank, range(Integer(1))))
>>> l == SP0.list()
True
```

class sage.combinat.permutation.**StandardPermutations_n_abstract**(*n*, *category=None*)

Bases: *Permutations*

Abstract base class for subsets of permutations of the set $\{1, 2, \ldots, n\}$.

> **Warning**
>
> Anything inheriting from this class should override the `__contains__` method.

class sage.combinat.permutation.**StandardPermutations_recoils**(*recoils*)

Bases: *Permutations*

Permutations of $\{1, \ldots, n\}$ with a fixed recoils composition.

class sage.combinat.permutation.**StandardPermutations_recoilsfatter**(*recoils*)

Bases: *Permutations*

class sage.combinat.permutation.**StandardPermutations_recoilsfiner**(*recoils*)

Bases: *Permutations*

sage.combinat.permutation.**bistochastic_as_sum_of_permutations**(*M*, *check=True*)

Return the positive sum of permutations corresponding to the bistochastic matrix `M`.

A stochastic matrix is a matrix with nonnegative real entries such that the sum of the elements of any row is equal to 1. A bistochastic matrix is a stochastic matrix whose transpose matrix is also stochastic (there are conditions both on the rows and on the columns).

According to the Birkhoff-von Neumann Theorem, any bistochastic matrix can be written as a convex combination of permutation matrices, which also means that the polytope of bistochastic matrices is integer.

As a non-bistochastic matrix can obviously not be written as a convex combination of permutations, this theorem is an equivalence.

This function, given a bistochastic matrix, returns the corresponding decomposition.

INPUT:

- `M` – a bistochastic matrix
- `check` – boolean; set to `True` (default) to check that the matrix is indeed bistochastic

OUTPUT:

- An element of `CombinatorialFreeModule`, which is a free F-module (where F is the ground ring of the given matrix) whose basis is indexed by the permutations.

Note

- In this function, we just assume 1 to be any constant : for us a matrix M is bistochastic if there exists $c > 0$ such that M/c is bistochastic.
- You can obtain a sequence of pairs `(permutation,coeff)`, where `permutation` is a Sage `Permutation` instance, and `coeff` its corresponding coefficient from the result of this function by applying the `list` function.
- If you are interested in the matrix corresponding to a `Permutation` you will be glad to learn about the `Permutation.to_matrix()` method.
- The base ring of the matrix can be anything that can be coerced to `RR`.

See also

- `as_sum_of_permutations()` to use this method through the `Matrix` class.

EXAMPLES:

We create a bistochastic matrix from a convex sum of permutations, then try to deduce the decomposition from the matrix:

```
sage: from sage.combinat.permutation import bistochastic_as_sum_of_permutations

sage: # needs networkx sage.graphs sage.modules
sage: L = []
sage: L.append((9,Permutation([4, 1, 3, 5, 2])))
sage: L.append((6,Permutation([5, 3, 4, 1, 2])))
sage: L.append((3,Permutation([3, 1, 4, 2, 5])))
sage: L.append((2,Permutation([1, 4, 2, 3, 5])))
sage: M = sum([c * p.to_matrix() for (c,p) in L])
sage: decomp = bistochastic_as_sum_of_permutations(M)
sage: print(decomp)
```

(continues on next page)

(continued from previous page)

```
2*B[[1, 4, 2, 3, 5]] + 3*B[[3, 1, 4, 2, 5]]
 + 9*B[[4, 1, 3, 5, 2]] + 6*B[[5, 3, 4, 1, 2]]
```

```
>>> from sage.all import *
>>> from sage.combinat.permutation import bistochastic_as_sum_of_permutations

>>> # needs networkx sage.graphs sage.modules
>>> L = []
>>> L.append((Integer(9),Permutation([Integer(4), Integer(1), Integer(3),␣
↪Integer(5), Integer(2)])))
>>> L.append((Integer(6),Permutation([Integer(5), Integer(3), Integer(4),␣
↪Integer(1), Integer(2)])))
>>> L.append((Integer(3),Permutation([Integer(3), Integer(1), Integer(4),␣
↪Integer(2), Integer(5)])))
>>> L.append((Integer(2),Permutation([Integer(1), Integer(4), Integer(2),␣
↪Integer(3), Integer(5)])))
>>> M = sum([c * p.to_matrix() for (c,p) in L])
>>> decomp = bistochastic_as_sum_of_permutations(M)
>>> print(decomp)
2*B[[1, 4, 2, 3, 5]] + 3*B[[3, 1, 4, 2, 5]]
 + 9*B[[4, 1, 3, 5, 2]] + 6*B[[5, 3, 4, 1, 2]]
```

An exception is raised when the matrix is not positive and bistochastic:

```
sage: # needs sage.modules
sage: M = Matrix([[2,3],[2,2]])
sage: decomp = bistochastic_as_sum_of_permutations(M)
Traceback (most recent call last):
...
ValueError: The matrix is not bistochastic
sage: bistochastic_as_sum_of_permutations(Matrix(GF(7), 2, [2,1,1,2]))
Traceback (most recent call last):
...
ValueError: The base ring of the matrix must have a coercion map to RR
sage: bistochastic_as_sum_of_permutations(Matrix(ZZ, 2, [2,-1,-1,2]))
Traceback (most recent call last):
...
ValueError: The matrix should have nonnegative entries
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> M = Matrix([[Integer(2),Integer(3)],[Integer(2),Integer(2)]])
>>> decomp = bistochastic_as_sum_of_permutations(M)
Traceback (most recent call last):
...
ValueError: The matrix is not bistochastic
>>> bistochastic_as_sum_of_permutations(Matrix(GF(Integer(7)), Integer(2),␣
↪[Integer(2),Integer(1),Integer(1),Integer(2)]))
Traceback (most recent call last):
...
ValueError: The base ring of the matrix must have a coercion map to RR
>>> bistochastic_as_sum_of_permutations(Matrix(ZZ, Integer(2), [Integer(2),-
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),-Integer(1),Integer(2)]))
Traceback (most recent call last):
...
ValueError: The matrix should have nonnegative entries
```

sage.combinat.permutation.**bounded_affine_permutation**(*A*)

Return the bounded affine permutation of a matrix.

The *bounded affine permutation* of a matrix A with entries in R is a partial permutation of length n, where n is the number of columns of A. The entry in position i is the smallest value j such that column i is in the span of columns $i+1,\ldots,j$, over R, where column indices are taken modulo n. If column i is the zero vector, then the permutation has a fixed point at i.

INPUT:

- `A` – matrix with entries in a ring R

EXAMPLES:

```
sage: from sage.combinat.permutation import bounded_affine_permutation
sage: A = Matrix(ZZ, [[1,0,0,0], [0,1,0,0]])                                    #␣
↪needs sage.modules
sage: bounded_affine_permutation(A)                                             #␣
↪needs sage.libs.flint sage.modules
[5, 6, 3, 4]

sage: A = Matrix(ZZ, [[0,1,0,1,0], [0,0,1,1,0]])                                #␣
↪needs sage.modules
sage: bounded_affine_permutation(A)                                             #␣
↪needs sage.libs.flint sage.modules
[1, 4, 7, 8, 5]
```

```
>>> from sage.all import *
>>> from sage.combinat.permutation import bounded_affine_permutation
>>> A = Matrix(ZZ, [[Integer(1),Integer(0),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0),Integer(0)]])                                   # needs␣
↪sage.modules
>>> bounded_affine_permutation(A)                                             #␣
↪needs sage.libs.flint sage.modules
[5, 6, 3, 4]

>>> A = Matrix(ZZ, [[Integer(0),Integer(1),Integer(0),Integer(1),Integer(0)],␣
↪[Integer(0),Integer(0),Integer(1),Integer(1),Integer(0)]])                    ␣
↪          # needs sage.modules
>>> bounded_affine_permutation(A)                                             #␣
↪needs sage.libs.flint sage.modules
[1, 4, 7, 8, 5]
```

REFERENCES:

- [KLS2013]

sage.combinat.permutation.**bruhat_lequal**(*p1*, *p2*)

Return `True` if `p1` is less than `p2` in the Bruhat order.

Algorithm from mupad-combinat.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.bruhat_lequal([2,4,3,1],[3,4,2,1])
True
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.bruhat_lequal([Integer(2),Integer(4),Integer(3),Integer(1)],
↪[Integer(3),Integer(4),Integer(2),Integer(1)])
True
```

sage.combinat.permutation.**descents_composition_first**(*dc*)

Compute the smallest element of a descent class having a descent composition dc.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_first([1,1,3,4,3])
[3, 2, 1, 4, 6, 5, 7, 8, 10, 9, 11, 12]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.descents_composition_first([Integer(1),Integer(1),Integer(3),
↪Integer(4),Integer(3)])
[3, 2, 1, 4, 6, 5, 7, 8, 10, 9, 11, 12]
```

sage.combinat.permutation.**descents_composition_last**(*dc*)

Return the largest element of a descent class having a descent composition dc.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_last([1,1,3,4,3])
[12, 11, 8, 9, 10, 4, 5, 6, 7, 1, 2, 3]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.descents_composition_last([Integer(1),Integer(1),Integer(3),
↪Integer(4),Integer(3)])
[12, 11, 8, 9, 10, 4, 5, 6, 7, 1, 2, 3]
```

sage.combinat.permutation.**descents_composition_list**(*dc*)

Return a list of all the permutations that have the descent composition dc.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_list([1,2,2])                    #␣
↪needs sage.graphs sage.modules
[[5, 2, 4, 1, 3],
 [5, 3, 4, 1, 2],
 [4, 3, 5, 1, 2],
 [4, 2, 5, 1, 3],
```

(continues on next page)

(continued from previous page)

```
 [3, 2, 5, 1, 4],
 [2, 1, 5, 3, 4],
 [3, 1, 5, 2, 4],
 [4, 1, 5, 2, 3],
 [5, 1, 4, 2, 3],
 [5, 1, 3, 2, 4],
 [4, 1, 3, 2, 5],
 [3, 1, 4, 2, 5],
 [2, 1, 4, 3, 5],
 [3, 2, 4, 1, 5],
 [4, 2, 3, 1, 5],
 [5, 2, 3, 1, 4]]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.descents_composition_list([Integer(1),Integer(2),Integer(2)])    ␣
↪                      # needs sage.graphs sage.modules
[[5, 2, 4, 1, 3],
 [5, 3, 4, 1, 2],
 [4, 3, 5, 1, 2],
 [4, 2, 5, 1, 3],
 [3, 2, 5, 1, 4],
 [2, 1, 5, 3, 4],
 [3, 1, 5, 2, 4],
 [4, 1, 5, 2, 3],
 [5, 1, 4, 2, 3],
 [5, 1, 3, 2, 4],
 [4, 1, 3, 2, 5],
 [3, 1, 4, 2, 5],
 [2, 1, 4, 3, 5],
 [3, 2, 4, 1, 5],
 [4, 2, 3, 1, 5],
 [5, 2, 3, 1, 4]]
```

sage.combinat.permutation.**from_cycles**(*n*, *cycles*, *parent=None*)

Return the permutation in the n-th symmetric group whose decomposition into disjoint cycles is `cycles`.

This function checks that its input is correct (i.e. that the cycles are disjoint and their elements integers among $1...n$). It raises an exception otherwise.

Warning: It assumes that the elements are of `int` type.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_cycles(4, [[1,2]])
[2, 1, 3, 4]
sage: permutation.from_cycles(4, [[1,2,4]])
[2, 4, 3, 1]
```

(continues on next page)

(continued from previous page)

```
sage: permutation.from_cycles(10, [[3,1],[4,5],[6,8,9]])
[3, 2, 1, 5, 4, 8, 7, 9, 6, 10]
sage: permutation.from_cycles(10, ((2, 5), (6, 1, 3)))
[3, 5, 6, 4, 2, 1, 7, 8, 9, 10]
sage: permutation.from_cycles(4, [])
[1, 2, 3, 4]
sage: permutation.from_cycles(4, [[]])
[1, 2, 3, 4]
sage: permutation.from_cycles(0, [])
[]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.from_cycles(Integer(4), [[Integer(1),Integer(2)]])
[2, 1, 3, 4]
>>> permutation.from_cycles(Integer(4), [[Integer(1),Integer(2),Integer(4)]])
[2, 4, 3, 1]
>>> permutation.from_cycles(Integer(10), [[Integer(3),Integer(1)],[Integer(4),
↪Integer(5)],[Integer(6),Integer(8),Integer(9)]])
[3, 2, 1, 5, 4, 8, 7, 9, 6, 10]
>>> permutation.from_cycles(Integer(10), ((Integer(2), Integer(5)), (Integer(6),␣
↪Integer(1), Integer(3))))
[3, 5, 6, 4, 2, 1, 7, 8, 9, 10]
>>> permutation.from_cycles(Integer(4), [])
[1, 2, 3, 4]
>>> permutation.from_cycles(Integer(4), [[]])
[1, 2, 3, 4]
>>> permutation.from_cycles(Integer(0), [])
[]
```

Bad input (see Issue #13742):

```
sage: Permutation("(-12,2)(3,4)")
Traceback (most recent call last):
...
ValueError: all elements should be strictly positive integers, but I found -12
sage: Permutation("(1,2)(2,4)")
Traceback (most recent call last):
...
ValueError: the element 2 appears more than once in the input
sage: permutation.from_cycles(4, [[1,18]])
Traceback (most recent call last):
...
ValueError: you claimed that this is a permutation on 1...4, but it contains 18
```

```
>>> from sage.all import *
>>> Permutation("(-12,2)(3,4)")
Traceback (most recent call last):
...
ValueError: all elements should be strictly positive integers, but I found -12
>>> Permutation("(1,2)(2,4)")
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: the element 2 appears more than once in the input
>>> permutation.from_cycles(Integer(4), [[Integer(1),Integer(18)]])
Traceback (most recent call last):
...
ValueError: you claimed that this is a permutation on 1...4, but it contains 18
```

sage.combinat.permutation.**from_inversion_vector**(*iv*, *parent=None*)

Return the permutation corresponding to inversion vector iv.

See $sage.combinat.permutation.Permutation.to_inversion_vector$ for a definition of the inversion vector of a permutation.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_inversion_vector([3,1,0,0,0])
[3, 2, 4, 1, 5]
sage: permutation.from_inversion_vector([2,3,6,4,0,2,2,1,0])
[5, 9, 1, 8, 2, 6, 4, 7, 3]
sage: permutation.from_inversion_vector([0])
[1]
sage: permutation.from_inversion_vector([])
[]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.from_inversion_vector([Integer(3),Integer(1),Integer(0),
↪Integer(0),Integer(0)])
[3, 2, 4, 1, 5]
>>> permutation.from_inversion_vector([Integer(2),Integer(3),Integer(6),
↪Integer(4),Integer(0),Integer(2),Integer(2),Integer(1),Integer(0)])
[5, 9, 1, 8, 2, 6, 4, 7, 3]
>>> permutation.from_inversion_vector([Integer(0)])
[1]
>>> permutation.from_inversion_vector([])
[]
```

sage.combinat.permutation.**from_lehmer_cocode**(*lehmer*, *parent=Standard permutations*)

Return the permutation with Lehmer cocode lehmer.

The Lehmer cocode of a permutation p is defined as the list $(c_1, c_2, \ldots, c_n)$, where c_i is the number of $j < i$ such that $p(j) > p(i)$.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: lcc = Permutation([2,1,5,4,3]).to_lehmer_cocode(); lcc
[0, 1, 0, 1, 2]
sage: permutation.from_lehmer_cocode(lcc)
[2, 1, 5, 4, 3]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> lcc = Permutation([Integer(2),Integer(1),Integer(5),Integer(4),Integer(3)]).
↪to_lehmer_cocode(); lcc
[0, 1, 0, 1, 2]
>>> permutation.from_lehmer_cocode(lcc)
[2, 1, 5, 4, 3]
```

sage.combinat.permutation.**from_lehmer_code**(*lehmer*, *parent=None*)

Return the permutation with Lehmer code `lehmer`.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: lc = Permutation([2,1,5,4,3]).to_lehmer_code(); lc
[1, 0, 2, 1, 0]
sage: permutation.from_lehmer_code(lc)
[2, 1, 5, 4, 3]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> lc = Permutation([Integer(2),Integer(1),Integer(5),Integer(4),Integer(3)]).to_
↪lehmer_code(); lc
[1, 0, 2, 1, 0]
>>> permutation.from_lehmer_code(lc)
[2, 1, 5, 4, 3]
```

sage.combinat.permutation.**from_major_code**(*mc*, *final_descent=False*)

Return the permutation with major code `mc`.

The major code of a permutation is defined in `to_major_code()`.

Warning

This function creates illegal permutations (i.e. `Permutation([9])`, and this is dangerous as the `Permutation()` class is only designed to handle permutations on $1...n$. This will have to be changed when Sage permutations will be able to handle anything, but right now this should be fixed. Be careful with the results.

Warning

If `mc` is not a major index of a permutation, then the return value of this method can be anything. Garbage in, garbage out!

REFERENCES:

- Skandera, M. *An Eulerian Partner for Inversions*. Sem. Lothar. Combin. 46 (2001) B46d.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_major_code([5, 0, 1, 0, 1, 2, 0, 1, 0])
[9, 3, 5, 7, 2, 1, 4, 6, 8]
```

(continues on next page)

(continued from previous page)

```
sage: permutation.from_major_code([8, 3, 3, 1, 4, 0, 1, 0, 0])
[2, 8, 4, 3, 6, 7, 9, 5, 1]
sage: Permutation([2,1,6,4,7,3,5]).to_major_code()
[3, 2, 0, 2, 2, 0, 0]
sage: permutation.from_major_code([3, 2, 0, 2, 2, 0, 0])
[2, 1, 6, 4, 7, 3, 5]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.from_major_code([Integer(5), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(0), Integer(1), Integer(0)])
[9, 3, 5, 7, 2, 1, 4, 6, 8]
>>> permutation.from_major_code([Integer(8), Integer(3), Integer(3), Integer(1),␣
↪Integer(4), Integer(0), Integer(1), Integer(0), Integer(0)])
[2, 8, 4, 3, 6, 7, 9, 5, 1]
>>> Permutation([Integer(2),Integer(1),Integer(6),Integer(4),Integer(7),
↪Integer(3),Integer(5)]).to_major_code()
[3, 2, 0, 2, 2, 0, 0]
>>> permutation.from_major_code([Integer(3), Integer(2), Integer(0), Integer(2),␣
↪Integer(2), Integer(0), Integer(0)])
[2, 1, 6, 4, 7, 3, 5]
```

sage.combinat.permutation.**from_permutation_group_element**(*pge*, *parent=None*)

Return a *Permutation* given a PermutationGroupElement pge.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: pge = PermutationGroupElement([(1,2),(3,4)])                          #␣
↪needs sage.groups
sage: permutation.from_permutation_group_element(pge)                       #␣
↪needs sage.groups
[2, 1, 4, 3]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> pge = PermutationGroupElement([(Integer(1),Integer(2)),(Integer(3),
↪Integer(4))])                              # needs sage.groups
>>> permutation.from_permutation_group_element(pge)                       #␣
↪needs sage.groups
[2, 1, 4, 3]
```

sage.combinat.permutation.**from_rank**(*n*, *rank*)

Return the permutation of the set $\{1, ..., n\}$ with lexicographic rank rank. This is the permutation whose Lehmer code is the factoradic representation of rank. In particular, the permutation with rank 0 is the identity permutation.

The permutation is computed without iterating through all of the permutations with lower rank. This makes it efficient for large permutations.

Note

The variable rank is not checked for being in the interval from 0 to $n! - 1$. When outside this interval, it acts as its residue modulo $n!$.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: Permutation([3, 6, 5, 4, 2, 1]).rank()
359
sage: [permutation.from_rank(3, i) for i in range(6)]
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
sage: Permutations(6)[10]
[1, 2, 4, 6, 3, 5]
sage: permutation.from_rank(6,10)
[1, 2, 4, 6, 3, 5]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> Permutation([Integer(3), Integer(6), Integer(5), Integer(4), Integer(2),␣
↪Integer(1)]).rank()
359
>>> [permutation.from_rank(Integer(3), i) for i in range(Integer(6))]
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
>>> Permutations(Integer(6))[Integer(10)]
[1, 2, 4, 6, 3, 5]
>>> permutation.from_rank(Integer(6),Integer(10))
[1, 2, 4, 6, 3, 5]
```

sage.combinat.permutation.**from_reduced_word**(*rw*, *parent=None*)

Return the permutation corresponding to the reduced word `rw`.

See *reduced_words()* for a definition of reduced words and the convention on the order of multiplication used.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_reduced_word([3,2,3,1,2,3,1])
[3, 4, 2, 1]
sage: permutation.from_reduced_word([])
[]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.from_reduced_word([Integer(3),Integer(2),Integer(3),Integer(1),
↪Integer(2),Integer(3),Integer(1)])
[3, 4, 2, 1]
>>> permutation.from_reduced_word([])
[]
```

sage.combinat.permutation.**permutohedron_lequal**(*p1*, *p2*, *side='right'*)

Return `True` if `p1` is less than or equal to `p2` in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.permutohedron_lequal(Permutation([3,2,1,4]),Permutation([4,2,1,
↪3]))
False
sage: permutation.permutohedron_lequal(Permutation([3,2,1,4]),Permutation([4,2,1,
↪3]), side='left')
True
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation as permutation
>>> permutation.permutohedron_lequal(Permutation([Integer(3),Integer(2),
↪Integer(1),Integer(4)]),Permutation([Integer(4),Integer(2),Integer(1),
↪Integer(3)]))
False
>>> permutation.permutohedron_lequal(Permutation([Integer(3),Integer(2),
↪Integer(1),Integer(4)]),Permutation([Integer(4),Integer(2),Integer(1),
↪Integer(3)]), side='left')
True
```

sage.combinat.permutation.**to_standard**(*p*, *key=None*)

Return a standard permutation corresponding to the iterable `p`.

INPUT:

- `p` – an iterable
- `key` – (optional) a comparison key for the element `x` of `p`

EXAMPLES:

```
sage: # needs sage.combinat
sage: import sage.combinat.permutation as permutation
sage: permutation.to_standard([4,2,7])
[2, 1, 3]
sage: permutation.to_standard([1,2,3])
[1, 2, 3]
sage: permutation.to_standard([])
[]
sage: permutation.to_standard([1,2,3], key=lambda x: -x)
[3, 2, 1]
sage: permutation.to_standard([5,8,2,5], key=lambda x: -x)
[2, 1, 4, 3]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> import sage.combinat.permutation as permutation
>>> permutation.to_standard([Integer(4),Integer(2),Integer(7)])
[2, 1, 3]
>>> permutation.to_standard([Integer(1),Integer(2),Integer(3)])
[1, 2, 3]
>>> permutation.to_standard([])
[]
>>> permutation.to_standard([Integer(1),Integer(2),Integer(3)], key=lambda x: -x)
[3, 2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> permutation.to_standard([Integer(5),Integer(8),Integer(2),Integer(5)],␣
↪key=lambda x: -x)
[2, 1, 4, 3]
```

5.1.173 Permutations (Cython file)

This is a nearly-straightforward implementation of what Knuth calls "Algorithm P" in TAOCP 7.2.1.2. The intent is to be able to enumerate permutation by "plain changes", or multiplication by adjacent transpositions, as a generator. This is useful when a class of objects is inherently enumerated by permutations, but it is faster to swap items in a permutation than construct the next object directly from the next permutation in a list. The backtracking algorithm in sage/graphs/genus.pyx is an example of this.

The lowest level is implemented as a struct with auxiliary methods. This is because Cython does not allow pointers to class instances, so a list of these objects is inherently slower than a list of structs. The author prefers ugly code to slow code.

For those willing to sacrifice a (very small) amount of speed, we provide a class that wraps our struct.

`sage.combinat.permutation_cython.`**`left_action_product`**`(S, lp)`

Return the permutation obtained by composing a permutation `S` with a permutation `lp` in such an order that `lp` is applied first and `S` is applied afterwards.

See also

sage.combinat.permutation.Permutation.left_action_product()

EXAMPLES:

```
sage: p = [2,1,3,4]
sage: q = [3,1,2]
sage: from sage.combinat.permutation_cython import left_action_product
sage: left_action_product(p, q)
[3, 2, 1, 4]
sage: left_action_product(q, p)
[1, 3, 2, 4]
sage: q
[3, 1, 2]
```

```
>>> from sage.all import *
>>> p = [Integer(2),Integer(1),Integer(3),Integer(4)]
>>> q = [Integer(3),Integer(1),Integer(2)]
>>> from sage.combinat.permutation_cython import left_action_product
>>> left_action_product(p, q)
[3, 2, 1, 4]
>>> left_action_product(q, p)
[1, 3, 2, 4]
>>> q
[3, 1, 2]
```

`sage.combinat.permutation_cython.`**`left_action_same_n`**`(S, lp)`

Return the permutation obtained by composing a permutation `S` with a permutation `lp` in such an order that `lp` is applied first and `S` is applied afterwards and `S` and `lp` are of the same length.

See also

sage.combinat.permutation.Permutation.left_action_product()

EXAMPLES:

```
sage: p = [2,1,3]
sage: q = [3,1,2]
sage: from sage.combinat.permutation_cython import left_action_same_n
sage: left_action_same_n(p, q)
[3, 2, 1]
sage: left_action_same_n(q, p)
[1, 3, 2]
```

```
>>> from sage.all import *
>>> p = [Integer(2),Integer(1),Integer(3)]
>>> q = [Integer(3),Integer(1),Integer(2)]
>>> from sage.combinat.permutation_cython import left_action_same_n
>>> left_action_same_n(p, q)
[3, 2, 1]
>>> left_action_same_n(q, p)
[1, 3, 2]
```

sage.combinat.permutation_cython.**map_to_list**(*l*, *values*, *n*)

Build a list by mapping the array `l` using `values`.

Warning

There is no check of the input data at any point. Using wrong types or values with wrong length is likely to result in a Sage crash.

INPUT:

- `l` – array of unsigned int (i.e., type `'I'`)
- `values` – tuple; the values of the permutation
- `n` – integer; the length of the array `l`

OUTPUT: list representing the permutation

EXAMPLES:

```
sage: from array import array
sage: from sage.combinat.permutation_cython import map_to_list
sage: l = array('I', [0, 1, 0, 3, 3, 0, 1])
sage: map_to_list(l, ('a', 'b', 'c', 'd'), 7)
['a', 'b', 'a', 'd', 'd', 'a', 'b']
```

```
>>> from sage.all import *
>>> from array import array
>>> from sage.combinat.permutation_cython import map_to_list
>>> l = array('I', [Integer(0), Integer(1), Integer(0), Integer(3), Integer(3),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0), Integer(1)])
>>> map_to_list(l, ('a', 'b', 'c', 'd'), Integer(7))
['a', 'b', 'a', 'd', 'd', 'a', 'b']
```

sage.combinat.permutation_cython.**next_perm**(*l*)

Obtain the next permutation under lex order of `l` by mutating `l`.

Algorithm based on: http://marknelson.us/2002/03/01/next-permutation/

INPUT:

- `l` – array of unsigned int (i.e., type `'I'`)

Warning: This method mutates the array `l`.

OUTPUT: boolean; whether another permutation was obtained

EXAMPLES:

```
sage: from sage.combinat.permutation_cython import next_perm
sage: from array import array
sage: L = array('I', [1, 1, 2, 3])
sage: while next_perm(L):
....:     print(L)
array('I', [1, 1, 3, 2])
array('I', [1, 2, 1, 3])
array('I', [1, 2, 3, 1])
array('I', [1, 3, 1, 2])
array('I', [1, 3, 2, 1])
array('I', [2, 1, 1, 3])
array('I', [2, 1, 3, 1])
array('I', [2, 3, 1, 1])
array('I', [3, 1, 1, 2])
array('I', [3, 1, 2, 1])
array('I', [3, 2, 1, 1])
```

```
>>> from sage.all import *
>>> from sage.combinat.permutation_cython import next_perm
>>> from array import array
>>> L = array('I', [Integer(1), Integer(1), Integer(2), Integer(3)])
>>> while next_perm(L):
...     print(L)
array('I', [1, 1, 3, 2])
array('I', [1, 2, 1, 3])
array('I', [1, 2, 3, 1])
array('I', [1, 3, 1, 2])
array('I', [1, 3, 2, 1])
array('I', [2, 1, 1, 3])
array('I', [2, 1, 3, 1])
array('I', [2, 3, 1, 1])
array('I', [3, 1, 1, 2])
```

(continues on next page)

(continued from previous page)

```
array('I', [3, 1, 2, 1])
array('I', [3, 2, 1, 1])
```

sage.combinat.permutation_cython.**permutation_iterator_transposition_list**(*n*)

Return a list of transposition indices to enumerate the permutations on n letters by adjacent transpositions. Assumes zero-based lists. We artificially limit the argument to $n < 12$ to avoid overflowing 32-bit pointers. While the algorithm works for larger n, the user is encouraged to avoid filling anything more than 4GB of memory with the output of this function.

EXAMPLES:

```
sage: import sage.combinat.permutation_cython
sage: from sage.combinat.permutation_cython import permutation_iterator_
↪transposition_list
sage: permutation_iterator_transposition_list(4)
[2, 1, 0, 2, 0, 1, 2, 0, 2, 1, 0, 2, 0, 1, 2, 0, 2, 1, 0, 2, 0, 1, 2]
sage: permutation_iterator_transposition_list(200)
Traceback (most recent call last):
...
ValueError: Cowardly refusing to enumerate the permutations on more than 12␣
↪letters.
sage: permutation_iterator_transposition_list(1)
[]

sage: # Generate the permutations of [1,2,3,4] fixing 4.
sage: Q = [1,2,3,4]
sage: L = [copy(Q)]
sage: for t in permutation_iterator_transposition_list(3):
....:     Q[t], Q[t+1] = Q[t+1], Q[t]
....:     L.append(copy(Q))
sage: print(L)
[[1, 2, 3, 4], [1, 3, 2, 4], [3, 1, 2, 4], [3, 2, 1, 4], [2, 3, 1, 4], [2, 1, 3,␣
↪4]]
```

```
>>> from sage.all import *
>>> import sage.combinat.permutation_cython
>>> from sage.combinat.permutation_cython import permutation_iterator_
↪transposition_list
>>> permutation_iterator_transposition_list(Integer(4))
[2, 1, 0, 2, 0, 1, 2, 0, 2, 1, 0, 2, 0, 1, 2, 0, 2, 1, 0, 2, 0, 1, 2]
>>> permutation_iterator_transposition_list(Integer(200))
Traceback (most recent call last):
...
ValueError: Cowardly refusing to enumerate the permutations on more than 12␣
↪letters.
>>> permutation_iterator_transposition_list(Integer(1))
[]

>>> # Generate the permutations of [1,2,3,4] fixing 4.
>>> Q = [Integer(1),Integer(2),Integer(3),Integer(4)]
>>> L = [copy(Q)]
>>> for t in permutation_iterator_transposition_list(Integer(3)):
```

(continues on next page)

(continued from previous page)

```
...     Q[t], Q[t+Integer(1)] = Q[t+Integer(1)], Q[t]
...     L.append(copy(Q))
>>> print(L)
[[1, 2, 3, 4], [1, 3, 2, 4], [3, 1, 2, 4], [3, 2, 1, 4], [2, 3, 1, 4], [2, 1, 3,␣
↪4]]
```

sage.combinat.permutation_cython.**right_action_product**(*S*, *rp*)

Return the permutation obtained by composing a permutation `S` with a permutation `rp` in such an order that `S` is applied first and `rp` is applied afterwards.

See also

sage.combinat.permutation.Permutation.right_action_product()

EXAMPLES:

```
sage: p = [2,1,3,4]
sage: q = [3,1,2]
sage: from sage.combinat.permutation_cython import right_action_product
sage: right_action_product(p, q)
[1, 3, 2, 4]
sage: right_action_product(q, p)
[3, 2, 1, 4]
sage: q
[3, 1, 2]
```

```
>>> from sage.all import *
>>> p = [Integer(2),Integer(1),Integer(3),Integer(4)]
>>> q = [Integer(3),Integer(1),Integer(2)]
>>> from sage.combinat.permutation_cython import right_action_product
>>> right_action_product(p, q)
[1, 3, 2, 4]
>>> right_action_product(q, p)
[3, 2, 1, 4]
>>> q
[3, 1, 2]
```

sage.combinat.permutation_cython.**right_action_same_n**(*S*, *rp*)

Return the permutation obtained by composing a permutation `S` with a permutation `rp` in such an order that `S` is applied first and `rp` is applied afterwards and `S` and `rp` are of the same length.

See also

sage.combinat.permutation.Permutation.right_action_product()

EXAMPLES:

```
sage: p = [2,1,3]
sage: q = [3,1,2]
sage: from sage.combinat.permutation_cython import right_action_same_n
```

(continues on next page)

(continued from previous page)

```
sage: right_action_same_n(p, q)
[1, 3, 2]
sage: right_action_same_n(q, p)
[3, 2, 1]
```

```
>>> from sage.all import *
>>> p = [Integer(2),Integer(1),Integer(3)]
>>> q = [Integer(3),Integer(1),Integer(2)]
>>> from sage.combinat.permutation_cython import right_action_same_n
>>> right_action_same_n(p, q)
[1, 3, 2]
>>> right_action_same_n(q, p)
[3, 2, 1]
```

5.1.174 Posets

Common posets can be accessed through `posets.<tab>` and are listed in the posets catalog:

- *Catalog of posets and lattices*

Poset-related classes:

- *Finite posets*
- *Finite lattices and semilattices*
- *Linear Extensions of Posets*
- *D-Complete Posets*
- *Forest Posets*
- *Mobile posets*
- *Incidence Algebras*
- *Cartesian products of Posets*
- *Möbius Algebras*
- *Generalized Tamari lattices*
- *Tamari Interval-posets*
- *Shard intersection order*

If you are looking for Poset-related `categories`, see `Posets`, `FinitePosets`, `LatticePosets` and `FiniteLatticePosets`.

5.1.175 Cartesian products of Posets

AUTHORS:

- Daniel Krenn (2015)

class `sage.combinat.posets.cartesian_product.`**`CartesianProductPoset`**(*sets*, *category*, *order=None*, ***kwargs*)

Bases: `CartesianProduct`

A class implementing Cartesian products of posets (and elements thereof). Compared to `CartesianProduct` you are able to specify an order for comparison of the elements.

INPUT:

- `sets` – tuple of parents
- `category` – a subcategory of `Sets().CartesianProducts() & Posets()`
- `order` – string or function specifying an order less or equal; it can be one of the following:
  - `'native'` – elements are ordered by their native ordering, i.e., the order the wrapped elements (tuples) provide
  - `'lex'` – elements are ordered lexicographically
  - `'product'` – an element is less or equal to another element, if less or equal is true for all its components (Cartesian projections)
  - a function which performs the comparison $\leq$; it takes two input arguments and outputs a boolean

Other keyword arguments (`kwargs`) are passed to the constructor of `CartesianProduct`.

EXAMPLES:

```
sage: P = Poset((srange(3), lambda left, right: left <= right))
sage: Cl = cartesian_product((P, P), order='lex')
sage: Cl((1, 1)) <= Cl((2, 0))
True
sage: Cp = cartesian_product((P, P), order='product')
sage: Cp((1, 1)) <= Cp((2, 0))
False
sage: def le_sum(left, right):
....:     return (sum(left) < sum(right) or
....:             sum(left) == sum(right) and left[0] <= right[0])
sage: Cs = cartesian_product((P, P), order=le_sum)
sage: Cs((1, 1)) <= Cs((2, 0))
True
```

```
>>> from sage.all import *
>>> P = Poset((srange(Integer(3)), lambda left, right: left <= right))
>>> Cl = cartesian_product((P, P), order='lex')
>>> Cl((Integer(1), Integer(1))) <= Cl((Integer(2), Integer(0)))
True
>>> Cp = cartesian_product((P, P), order='product')
>>> Cp((Integer(1), Integer(1))) <= Cp((Integer(2), Integer(0)))
False
>>> def le_sum(left, right):
...     return (sum(left) < sum(right) or
...             sum(left) == sum(right) and left[Integer(0)] <= right[Integer(0)])
>>> Cs = cartesian_product((P, P), order=le_sum)
>>> Cs((Integer(1), Integer(1))) <= Cs((Integer(2), Integer(0)))
True
```

See also:

`CartesianProduct`

class Element

Bases: `Element`

le(*left*, *right*)

Test whether `left` is less than or equal to `right`.

INPUT:

- `left` – an element
- `right` – an element

OUTPUT: boolean

Note

This method uses the order defined on creation of this Cartesian product. See `CartesianProductPoset`.

EXAMPLES:

```
sage: P = posets.ChainPoset(10)
sage: def le_sum(left, right):
....:     return (sum(left) < sum(right) or
....:             sum(left) == sum(right) and left[0] <= right[0])
sage: C = cartesian_product((P, P), order=le_sum)
sage: C.le(C((1, 6)), C((6, 1)))
True
sage: C.le(C((6, 1)), C((1, 6)))
False
sage: C.le(C((1, 6)), C((6, 6)))
True
sage: C.le(C((6, 6)), C((1, 6)))
False
```

```
>>> from sage.all import *
>>> P = posets.ChainPoset(Integer(10))
>>> def le_sum(left, right):
...     return (sum(left) < sum(right) or
...             sum(left) == sum(right) and left[Integer(0)] <=␣
↪right[Integer(0)])
>>> C = cartesian_product((P, P), order=le_sum)
>>> C.le(C((Integer(1), Integer(6))), C((Integer(6), Integer(1))))
True
>>> C.le(C((Integer(6), Integer(1))), C((Integer(1), Integer(6))))
False
>>> C.le(C((Integer(1), Integer(6))), C((Integer(6), Integer(6))))
True
>>> C.le(C((Integer(6), Integer(6))), C((Integer(1), Integer(6))))
False
```

le_lex(*left*, *right*)

Test whether `left` is lexicographically smaller or equal to `right`.

INPUT:

- `left` – an element
- `right` – an element

OUTPUT: boolean

EXAMPLES:

```
sage: P = Poset((srange(2), lambda left, right: left <= right))
sage: Q = cartesian_product((P, P), order='lex')
sage: T = [Q((0, 0)), Q((1, 1)), Q((0, 1)), Q((1, 0))]
sage: for a in T:
....:     for b in T:
....:         assert Q.le(a, b) == (a <= b)
....:         print('%s <= %s = %s' % (a, b, a <= b))
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = True
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

```
>>> from sage.all import *
>>> P = Poset((srange(Integer(2)), lambda left, right: left <= right))
>>> Q = cartesian_product((P, P), order='lex')
>>> T = [Q((Integer(0), Integer(0))), Q((Integer(1), Integer(1))),
→Q((Integer(0), Integer(1))), Q((Integer(1), Integer(0)))]
>>> for a in T:
...     for b in T:
...         assert Q.le(a, b) == (a <= b)
...         print('%s <= %s = %s' % (a, b, a <= b))
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = True
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

le_native(*left*, *right*)

Test whether `left` is smaller or equal to `right` in the order provided by the elements themselves.

INPUT:

- `left` – an element
- `right` – an element

OUTPUT: boolean

EXAMPLES:

```
sage: P = Poset((srange(2), lambda left, right: left <= right))
sage: Q = cartesian_product((P, P), order='native')
sage: T = [Q((0, 0)), Q((1, 1)), Q((0, 1)), Q((1, 0))]
sage: for a in T:
....:     for b in T:
....:         assert Q.le(a, b) == (a <= b)
....:         print('%s <= %s = %s' % (a, b, a <= b))
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = True
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

```
>>> from sage.all import *
>>> P = Poset((srange(Integer(2)), lambda left, right: left <= right))
>>> Q = cartesian_product((P, P), order='native')
>>> T = [Q((Integer(0), Integer(0))), Q((Integer(1), Integer(1))),
↪Q((Integer(0), Integer(1))), Q((Integer(1), Integer(0)))]
>>> for a in T:
...     for b in T:
...         assert Q.le(a, b) == (a <= b)
...         print('%s <= %s = %s' % (a, b, a <= b))
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
```

(continues on next page)

(continued from previous page)

```
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = True
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

le_product (*left*, *right*)

Test whether `left` is component-wise smaller or equal to `right`.

INPUT:

- `left` – an element
- `right` – an element

OUTPUT: boolean

The comparison is `True` if the result of the comparison in each component is `True`.

EXAMPLES:

```
sage: P = Poset((srange(2), lambda left, right: left <= right))
sage: Q = cartesian_product((P, P), order='product')
sage: T = [Q((0, 0)), Q((1, 1)), Q((0, 1)), Q((1, 0))]
sage: for a in T:
....:     for b in T:
....:         assert Q.le(a, b) == (a <= b)
....:         print('%s <= %s = %s' % (a, b, a <= b))
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = False
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

```
>>> from sage.all import *
>>> P = Poset((srange(Integer(2)), lambda left, right: left <= right))
>>> Q = cartesian_product((P, P), order='product')
>>> T = [Q((Integer(0), Integer(0))), Q((Integer(1), Integer(1))),
↪Q((Integer(0), Integer(1))), Q((Integer(1), Integer(0)))]
>>> for a in T:
...     for b in T:
...         assert Q.le(a, b) == (a <= b)
...         print('%s <= %s = %s' % (a, b, a <= b))
```

(continues on next page)

(continued from previous page)

```
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = False
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

5.1.176 D-Complete Posets

AUTHORS:

- Stefan Grosser (06-2020): initial implementation

class sage.combinat.posets.d_complete.**DCompletePoset**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: *FiniteJoinSemilattice*

A d-complete poset.

D-complete posets are a class of posets introduced by Proctor in [Proc1999]. It includes common families such as shapes, shifted shapes, and rooted forests. Proctor showed in [PDynk1999] that d-complete posets have decompositions in *irreducible* posets, and showed in [Proc2014] that d-complete posets admit a hook-length formula (see Wikipedia article Hook_length_formula). A complete proof of the hook-length formula can be found in [KY2019].

EXAMPLES:

```
sage: from sage.combinat.posets.poset_examples import Posets
sage: P = Posets.DoubleTailedDiamond(2)
sage: TestSuite(P).run()
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.poset_examples import Posets
>>> P = Posets.DoubleTailedDiamond(Integer(2))
>>> TestSuite(P).run()
```

get_hook(*elmt*)

Return the hook length of the element `elmt`.

EXAMPLES:

```
sage: from sage.combinat.posets.d_complete import DCompletePoset
sage: P = DCompletePoset(DiGraph({0: [1], 1: [2]}))
sage: P.get_hook(1)
2
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.d_complete import DCompletePoset
>>> P = DCompletePoset(DiGraph({Integer(0): [Integer(1)], Integer(1):␣
→[Integer(2)]}))
>>> P.get_hook(Integer(1))
2
```

get_hooks()

Return all the hook lengths as a dictionary.

EXAMPLES:

```
sage: from sage.combinat.posets.d_complete import DCompletePoset
sage: P = DCompletePoset(DiGraph({0: [1, 2], 1: [3], 2: [3], 3: []}))
sage: P.get_hooks()
{0: 1, 1: 2, 2: 2, 3: 3}
sage: from sage.combinat.posets.poset_examples import Posets
sage: YDP321 = Posets.YoungDiagramPoset(Partition([3,2,1]))
sage: P = DCompletePoset(YDP321._hasse_diagram.reverse())
sage: P.get_hooks()
{0: 5, 1: 3, 2: 1, 3: 3, 4: 1, 5: 1}
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.d_complete import DCompletePoset
>>> P = DCompletePoset(DiGraph({Integer(0): [Integer(1), Integer(2)],␣
→Integer(1): [Integer(3)], Integer(2): [Integer(3)], Integer(3): []}))
>>> P.get_hooks()
{0: 1, 1: 2, 2: 2, 3: 3}
>>> from sage.combinat.posets.poset_examples import Posets
>>> YDP321 = Posets.YoungDiagramPoset(Partition([Integer(3),Integer(2),
→Integer(1)]))
>>> P = DCompletePoset(YDP321._hasse_diagram.reverse())
>>> P.get_hooks()
{0: 5, 1: 3, 2: 1, 3: 3, 4: 1, 5: 1}
```

hook_product()

Return the hook product for the poset.

5.1.177 Mobile posets

class sage.combinat.posets.mobile.**MobilePoset**(*hasse_diagram*, *elements*, *category*, *facade*, *key*, *ribbon=None*, *check=True*)

Bases: *FinitePoset*

A mobile poset.

Mobile posets are an extension of d-complete posets which permit a determinant formula for counting linear extensions. They are formed by having a ribbon poset with d-complete posets 'hanging' below it and at most one d-complete poset above it, known as the anchor. See [GGMM2020] for the definition.

EXAMPLES:

```
sage: P = posets.MobilePoset(posets.RibbonPoset(7, [1,3]),                    #␣
→needs sage.combinat sage.modules
```

(continues on next page)

(continued from previous page)

```
....:                          {1: [posets.YoungDiagramPoset([3, 2], dual=True)],
....:                           3: [posets.DoubleTailedDiamond(6)]},
....:                          anchor=(4, 2, posets.ChainPoset(6)))
sage: len(P._ribbon)                                                          #␣
↪needs sage.combinat sage.modules
8
sage: P._anchor                                                               #␣
↪needs sage.combinat sage.modules
(4, 5)
```

```
>>> from sage.all import *
>>> P = posets.MobilePoset(posets.RibbonPoset(Integer(7), [Integer(1),
↪Integer(3)]),                          # needs sage.combinat sage.modules
...                        {Integer(1): [posets.YoungDiagramPoset([Integer(3),␣
↪Integer(2)], dual=True)],
...                         Integer(3): [posets.DoubleTailedDiamond(Integer(6))]},
...                        anchor=(Integer(4), Integer(2), posets.
↪ChainPoset(Integer(6))))
>>> len(P._ribbon)                                                          #␣
↪needs sage.combinat sage.modules
8
>>> P._anchor                                                               #␣
↪needs sage.combinat sage.modules
(4, 5)
```

This example is Example 5.9 in [GGMM2020]:

```
sage: P1 = posets.MobilePoset(posets.RibbonPoset(8, [2,3,4]),
....:                         {4: [posets.ChainPoset(1)]},
....:                         anchor=(3, 0, posets.ChainPoset(1)))
sage: sorted([P1._element_to_vertex(i) for i in P1._ribbon])
[0, 1, 2, 6, 7, 9]
sage: P1._anchor
(3, 2)

sage: P2 = posets.MobilePoset(posets.RibbonPoset(15, [1,3,5,7,9,11,13]),
....:                         {}, anchor=(8, 0, posets.ChainPoset(1)))
sage: sorted(P2._ribbon)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
sage: P2._anchor
(8, (8, 0))
sage: P2.linear_extensions().cardinality()                                    #␣
↪needs sage.modules
21399440939

sage: EP = posets.MobilePoset(posets.ChainPoset(0), {})
Traceback (most recent call last):
...
ValueError: the empty poset is not a mobile poset
```

```
>>> from sage.all import *
>>> P1 = posets.MobilePoset(posets.RibbonPoset(Integer(8), [Integer(2),Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(4)]),
...                             {Integer(4): [posets.ChainPoset(Integer(1))]},
...                             anchor=(Integer(3), Integer(0), posets.
↪ChainPoset(Integer(1))))
>>> sorted([P1._element_to_vertex(i) for i in P1._ribbon])
[0, 1, 2, 6, 7, 9]
>>> P1._anchor
(3, 2)

>>> P2 = posets.MobilePoset(posets.RibbonPoset(Integer(15), [Integer(1),
↪Integer(3),Integer(5),Integer(7),Integer(9),Integer(11),Integer(13)]),
...                             {}, anchor=(Integer(8), Integer(0), posets.
↪ChainPoset(Integer(1))))
>>> sorted(P2._ribbon)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> P2._anchor
(8, (8, 0))
>>> P2.linear_extensions().cardinality()                                    #␣
↪needs sage.modules
21399440939

>>> EP = posets.MobilePoset(posets.ChainPoset(Integer(0)), {})
Traceback (most recent call last):
...
ValueError: the empty poset is not a mobile poset
```

anchor()

Return the anchor of the mobile poset.

EXAMPLES:

```
sage: from sage.combinat.posets.mobile import MobilePoset
sage: M2 = MobilePoset(Poset([[0,1,2,3,4,5,6,7,8],
....:           [(1,0),(3,0),(2,1),(2,3),(4,3),(5,4),(7,4),(7,8)]]))
sage: M2.anchor()
(4, 3)
sage: M3 = MobilePoset(Posets.RibbonPoset(5, [1,2]))
sage: M3.anchor() is None
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.mobile import MobilePoset
>>> M2 = MobilePoset(Poset([[Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(6),Integer(7),Integer(8)],
...           [(Integer(1),Integer(0)),(Integer(3),Integer(0)),(Integer(2),
↪Integer(1)),(Integer(2),Integer(3)),(Integer(4),Integer(3)),(Integer(5),
↪Integer(4)),(Integer(7),Integer(4)),(Integer(7),Integer(8))]]))
>>> M2.anchor()
(4, 3)
>>> M3 = MobilePoset(Posets.RibbonPoset(Integer(5), [Integer(1),Integer(2)]))
>>> M3.anchor() is None
True
```

ribbon()

Return the ribbon of the mobile poset.

EXAMPLES:

```
sage: from sage.combinat.posets.mobile import MobilePoset
sage: M3 = MobilePoset(Posets.RibbonPoset(5, [1,2]))
sage: sorted(M3.ribbon())
[1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.mobile import MobilePoset
>>> M3 = MobilePoset(Posets.RibbonPoset(Integer(5), [Integer(1),Integer(2)]))
>>> sorted(M3.ribbon())
[1, 2, 3, 4]
```

5.1.178 Elements of posets, lattices, semilattices, etc.

class sage.combinat.posets.elements.**JoinSemilatticeElement**(*poset*, *element*, *vertex*)

Bases: *PosetElement*

class sage.combinat.posets.elements.**LatticePosetElement**(*poset*, *element*, *vertex*)

Bases: *MeetSemilatticeElement*, *JoinSemilatticeElement*

class sage.combinat.posets.elements.**MeetSemilatticeElement**(*poset*, *element*, *vertex*)

Bases: *PosetElement*

class sage.combinat.posets.elements.**PosetElement**(*poset*, *element*, *vertex*)

Bases: Element

Establish the parent-child relationship between `poset` and `element`, where `element` is associated to the vertex `vertex` of the Hasse diagram of the poset.

INPUT:

- `poset` – a poset object
- `element` – any object
- `vertex` – a vertex of the Hasse diagram of the poset

5.1.179 Forest Posets

AUTHORS:

- Stefan Grosser (06-2020): initial implementation

class sage.combinat.posets.forest.**ForestPoset**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: *FinitePoset*

A forest poset is a poset where the underlying Hasse diagram and is directed acyclic graph.

5.1.180 Hasse diagrams of posets

antichains()	Return all antichains of `self`, organized as a prefix tree.
antichains_iterator()	Return an iterator over the antichains of the poset.

continues on next page

Table 4 – continued from previous page

are_comparable()	Return whether i and j are comparable in the poset.
are_incomparable()	Return whether i and j are incomparable in the poset.
atoms_of_congruence_lattice()	Return atoms of the congruence lattice.
bottom_moebius_function()	Return the value of the Möbius function of the poset on the elements zero and j, where zero is self.bottom(), the unique minimal element of the poset.
cardinality()	Return the number of elements in the poset.
chain_polynomial()	Return the chain polynomial of the poset.
chains()	Return all chains of self, organized as a prefix tree.
closed_interval()	Return a list of the elements z of self such that $x \leq z \leq y$.
common_lower_covers()	Return the list of all common lower covers of vertices.
common_upper_covers()	Return the list of all common upper covers of vertices.
congruence()	Return the congruence start "extended" by parts.
congruences_iterator()	Return an iterator over all congruences of the lattice.
cover_relations()	Return the list of cover relations.
cover_relations_iterator()	Iterate over cover relations.
covers()	Return True if y covers x and False otherwise.
diamonds()	Return the list of diamonds of self.
dual()	Return a poset that is dual to the given poset.
find_nonsemidistributive_elements()	Check if the lattice is semidistributive or not.
find_nonsemimodular_pair()	Return pair of elements showing the lattice is not modular.
find_nontrivial_congruence()	Return a pair that generates non-trivial congruence or None if there is not any.
frattini_sublattice()	Return the list of elements of the Frattini sublattice of the lattice.
greedy_linear_extensions_iterator()	Return an iterator over greedy linear extensions of the Hasse diagram.
has_bottom()	Return True if the poset has a unique minimal element.
has_top()	Return True if the poset contains a unique maximal element, and False otherwise.
interval_iterator()	Return an iterator of the elements z of self such that $x \leq z \leq y$.
is_antichain_of_poset()	Return True if elms is an antichain of the Hasse diagram and False otherwise.
is_bounded()	Return True if the poset contains a unique maximal element and a unique minimal element, and False otherwise.
is_chain()	Return True if the poset is totally ordered, and False otherwise.
is_complemented()	Return an element of the lattice that has no complement.
is_congruence_normal()	Return True if the lattice can be constructed from the one-element lattice with Day doubling constructions of convex subsets.
is_convex_subset()	Return True if S is a convex subset of the poset, and False otherwise.
is_gequal()	Return True if x is greater than or equal to y, and False otherwise.
is_greater_than()	Return True if x is greater than but not equal to y, and False otherwise.
is_join_semilattice()	Return True if self has a join operation, and False otherwise.
is_lequal()	Return True if i is less than or equal to j in the poset, and False otherwise.
is_less_than()	Return True if x is less than but not equal to y in the poset, and False otherwise.
is_linear_extension()	Test if an ordering is a linear extension.
is_linear_interval()	Return whether the interval [t_min, t_max] is linear.
is_meet_semilattice()	Return True if self has a meet operation, and False otherwise.
is_ranked()	Return True if the poset is ranked, and False otherwise.
join_matrix()	Return the matrix of joins of self, when self is a join-semilattice; raise an error otherwise.

continues on next page

Table 4 – continued from previous page

`kappa()`	Return the maximum element greater than the element covered by `a` but not greater than `a`.
`kappa_dual()`	Return the minimum element smaller than the element covering `a` but not smaller than `a`.
`lequal_matrix()`	Return a matrix whose `(i,j)` entry is 1 if `i` is less than `j` in the poset, and 0 otherwise; and redefines `__lt__` to use the boolean version of this matrix.
`linear_extension()`	Return a linear extension.
`linear_extensions()`	Return an iterator over all linear extensions.
`lower_covers_iterator()`	Return the list of elements that are covered by `element`.
`maximal_elements()`	Return a list of the maximal elements of the poset.
`maximal_sublattices()`	Return maximal sublattices of the lattice.
`meet_matrix()`	Return the matrix of meets of `self`, when `self` is a meet-semilattice; raise an error otherwise.
`minimal_elements()`	Return a list of the minimal elements of the poset.
`moebius_function()`	Return the value of the Möbius function of the poset on the elements `i` and `j`.
`moebius_function_matrix()`	Return the matrix of the Möbius function of this poset.
`neutral_elements()`	Return the list of neutral elements of the lattice.
`open_interval()`	Return a list of the elements z of `self` such that $x < z < y$.
`order_filter()`	Return the order filter generated by a list of elements.
`order_ideal()`	Return the order ideal generated by a list of elements.
`order_ideal_cardinality()`	Return the cardinality of the order ideal generated by `elements`.
`orthocomplementations_iterator()`	Return an iterator over orthocomplementations of the lattice.
`prime_elements()`	Return the join-prime and meet-prime elements of the bounded poset.
`principal_congruences_poset()`	Return the poset of join-irreducibles of the congruence lattice.
`principal_order_filter()`	Return the order filter generated by `i`.
`principal_order_ideal()`	Return the order ideal generated by i.
`pseudocomplement()`	Return the pseudocomplement of `element`, if it exists.
`rank()`	Return the rank of `element`, or the rank of the poset if `element` is `None`. (The rank of a poset is the length of the longest chain of elements of the poset.)
`rank_function()`	Return the (normalized) rank function of the poset, if it exists.
`skeleton()`	Return the skeleton of the lattice.
`sublattices_iterator()`	Return an iterator over sublattices of the Hasse diagram.
`supergreedy_linear_extensions_iterator()`	Return an iterator over supergreedy linear extensions of the Hasse diagram.
`top()`	Return the top element of the poset, if it exists.
`upper_covers_iterator()`	Return the list of elements that cover `element`.
`vertical_decomposition()`	Return vertical decomposition of the lattice.

class `sage.combinat.posets.hasse_diagram.`**`HasseDiagram`**(*data=None*, *pos=None*, *loops=None*, *format=None*, *weighted=None*, *data_structure='sparse'*, *vertex_labels=True*, *name=None*, *multiedges=None*, *convert_empty_dict_labels_to_None=None*, *sparse=True*, *immutable=False*, *hash_labels=None*)

Bases: `DiGraph`

The Hasse diagram of a poset. This is just a transitively-reduced, directed, acyclic graph without loops or multiple edges.

Note: We assume that `range(n)` is a linear extension of the poset. That is, `range(n)` is the vertex set and a topological sort of the digraph.

This should not be called directly, use Poset instead; all type checking happens there.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]}); H
Hasse diagram of a poset containing 4 elements
sage: TestSuite(H).run()
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(2)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[]}); H
Hasse diagram of a poset containing 4 elements
>>> TestSuite(H).run()
```

antichains(*element_class=<class 'list'>*)

Return all antichains of `self`, organized as a prefix tree.

INPUT:

- `element_class` – (default: `list`) an iterable type

EXAMPLES:

```
sage: # needs sage.modules
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: A = H.antichains()
sage: list(A)
[[], [0], [1], [1, 2], [1, 3], [2], [3], [4]]
sage: A.cardinality()
8
sage: [1,3] in A
True
sage: [1,4] in A
False
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> P = posets.PentagonPoset()
>>> H = P._hasse_diagram
>>> A = H.antichains()
>>> list(A)
[[], [0], [1], [1, 2], [1, 3], [2], [3], [4]]
>>> A.cardinality()
```

(continues on next page)

(continued from previous page)

```
8
>>> [Integer(1),Integer(3)] in A
True
>>> [Integer(1),Integer(4)] in A
False
```

antichains_iterator()

Return an iterator over the antichains of the poset.

Note

The algorithm is based on Freese-Jezek-Nation p. 226. It does a depth first search through the set of all antichains organized in a prefix tree.

EXAMPLES:

```
sage: # needs sage.modules
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: H.antichains_iterator()
<generator object ...antichains_iterator at ...>
sage: list(H.antichains_iterator())
[[], [4], [3], [2], [1], [1, 3], [1, 2], [0]]
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[4],2:[3],3:[4]})
sage: list(H.antichains_iterator())
[[], [4], [3], [2], [1], [1, 3], [1, 2], [0]]
sage: H = HasseDiagram({0:[],1:[],2:[]})
sage: list(H.antichains_iterator())
[[], [2], [1], [1, 2], [0], [0, 2], [0, 1], [0, 1, 2]]
sage: H = HasseDiagram({0:[1],1:[2],2:[3],3:[4]})
sage: list(H.antichains_iterator())
[[], [4], [3], [2], [1], [0]]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> P = posets.PentagonPoset()
>>> H = P._hasse_diagram
>>> H.antichains_iterator()
<generator object ...antichains_iterator at ...>
>>> list(H.antichains_iterator())
[[], [4], [3], [2], [1], [1, 3], [1, 2], [0]]
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(3)],Integer(3):[Integer(4)]})
>>> list(H.antichains_iterator())
[[], [4], [3], [2], [1], [1, 3], [1, 2], [0]]
>>> H = HasseDiagram({Integer(0):[],Integer(1):[],Integer(2):[]})
>>> list(H.antichains_iterator())
[[], [2], [1], [1, 2], [0], [0, 2], [0, 1], [0, 1, 2]]
>>> H = HasseDiagram({Integer(0):[Integer(1)],Integer(1):[Integer(2)],
```

(continues on next page)

(continued from previous page)

```
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)]})
>>> list(H.antichains_iterator())
[[], [4], [3], [2], [1], [0]]
```

are_comparable(*i*, *j*)

Return whether i and j are comparable in the poset.

INPUT:

- i, j – vertices of this Hasse diagram

EXAMPLES:

```
sage: # needs sage.modules
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: H.are_comparable(1,2)
False
sage: V = H.vertices(sort=True)
sage: [ (i,j) for i in V for j in V if H.are_comparable(i,j)]
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1), (1, 4),
 (2, 0), (2, 2), (2, 3), (2, 4), (3, 0), (3, 2), (3, 3), (3, 4),
 (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> P = posets.PentagonPoset()
>>> H = P._hasse_diagram
>>> H.are_comparable(Integer(1),Integer(2))
False
>>> V = H.vertices(sort=True)
>>> [ (i,j) for i in V for j in V if H.are_comparable(i,j)]
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1), (1, 4),
 (2, 0), (2, 2), (2, 3), (2, 4), (3, 0), (3, 2), (3, 3), (3, 4),
 (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
```

are_incomparable(*i*, *j*)

Return whether i and j are incomparable in the poset.

INPUT:

- i, j – vertices of this Hasse diagram

EXAMPLES:

```
sage: # needs sage.modules
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: H.are_incomparable(1,2)
True
sage: V = H.vertices(sort=True)
sage: [ (i,j) for i in V for j in V if H.are_incomparable(i,j)]
[(1, 2), (1, 3), (2, 1), (3, 1)]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> P = posets.PentagonPoset()
>>> H = P._hasse_diagram
>>> H.are_incomparable(Integer(1),Integer(2))
True
>>> V = H.vertices(sort=True)
>>> [ (i,j) for i in V for j in V if H.are_incomparable(i,j)]
[(1, 2), (1, 3), (2, 1), (3, 1)]
```

atoms_of_congruence_lattice()

Return atoms of the congruence lattice.

In other words, return "minimal non-trivial" congruences: A congruence is minimal if the only finer (as a partition of set of elements) congruence is the trivial congruence where every block contains only one element.

> **See also**
>
> *congruence()*

OUTPUT:

List of congruences, every congruence as *sage.combinat.set_partition.SetPartition*

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: N5 = HasseDiagram({0: [1, 2], 1: [4], 2: [3], 3:[4]})
sage: N5.atoms_of_congruence_lattice()                                      #␣
↪needs sage.combinat sage.modules
[{{0}, {1}, {2, 3}, {4}}]
sage: Hex = HasseDiagram({0: [1, 2], 1: [3], 2: [4], 3: [5], 4: [5]})
sage: Hex.atoms_of_congruence_lattice()                                     #␣
↪needs sage.combinat sage.modules
[{{0}, {1}, {2, 4}, {3}, {5}}, {{0}, {1, 3}, {2}, {4}, {5}}]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> N5 = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(4)], Integer(2): [Integer(3)], Integer(3):[Integer(4)]})
>>> N5.atoms_of_congruence_lattice()                                      #␣
↪needs sage.combinat sage.modules
[{{0}, {1}, {2, 3}, {4}}]
>>> Hex = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(4)], Integer(3): [Integer(5)],␣
↪Integer(4): [Integer(5)]})
>>> Hex.atoms_of_congruence_lattice()                                     #␣
↪needs sage.combinat sage.modules
[{{0}, {1}, {2, 4}, {3}, {5}}, {{0}, {1, 3}, {2}, {4}, {5}}]
```

ALGORITHM:

Every atom is a join-irreducible. Every join-irreducible of $\mathrm{Con}(L)$ is a principal congruence generated by a meet-irreducible element and the only element covering it (and also by a join-irreducible element and the only element covered by it). Hence we check those principal congruences to find the minimal ones.

bottom()

Return the bottom element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.bottom() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.bottom()
0
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)],Integer(4):[]})
>>> P.bottom() is None
True
>>> Q = Poset({Integer(0):[Integer(1)],Integer(1):[]})
>>> Q.bottom()
0
```

bottom_moebius_function(*j*)

Return the value of the Möbius function of the poset on the elements `zero` and `j`, where `zero` is `self.bottom()`, the unique minimal element of the poset.

EXAMPLES:

```
sage: P = Poset({0: [1,2]})
sage: hasse = P._hasse_diagram
sage: hasse.bottom_moebius_function(1)
-1
sage: hasse.bottom_moebius_function(2)
-1
sage: P = Poset({0: [1,3], 1:[2], 2:[4], 3:[4]})
sage: hasse = P._hasse_diagram
sage: for i in range(5):
....:   print(hasse.bottom_moebius_function(i))
1
-1
0
-1
1
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0): [Integer(1),Integer(2)]})
>>> hasse = P._hasse_diagram
>>> hasse.bottom_moebius_function(Integer(1))
-1
>>> hasse.bottom_moebius_function(Integer(2))
-1
>>> P = Poset({Integer(0): [Integer(1),Integer(3)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(4)], Integer(3):[Integer(4)]})
```

(continues on next page)

(continued from previous page)

```
>>> hasse = P._hasse_diagram
>>> for i in range(Integer(5)):
...     print(hasse.bottom_moebius_function(i))
1
-1
0
-1
1
```

cardinality()

Return the number of elements in the poset.

EXAMPLES:

```
sage: Poset([[1,2,3],[4],[4],[4],[]]).cardinality()
5
```

```
>>> from sage.all import *
>>> Poset([[Integer(1),Integer(2),Integer(3)],[Integer(4)],[Integer(4)],
↪[Integer(4)],[]]).cardinality()
5
```

chain_polynomial()

Return the chain polynomial of the poset.

The coefficient of q^k is the number of chains of k elements in the poset. List of coefficients of this polynomial is also called a *f-vector* of the poset.

EXAMPLES:

```
sage: P = posets.ChainPoset(3)
sage: H = P._hasse_diagram
sage: t = H.chain_polynomial(); t
q^3 + 3*q^2 + 3*q + 1
```

```
>>> from sage.all import *
>>> P = posets.ChainPoset(Integer(3))
>>> H = P._hasse_diagram
>>> t = H.chain_polynomial(); t
q^3 + 3*q^2 + 3*q + 1
```

chains(*element_class=<class 'list'>*, *exclude=None*, *conversion=None*)

Return all chains of `self`, organized as a prefix tree.

INPUT:

- `element_class` – (default: `list`) an iterable type
- `exclude` – elements of the poset to be excluded (default: `None`)
- `conversion` – (default: `None`) used to pass the list of elements of the poset in their fixed order

OUTPUT:

The enumerated set (with a forest structure given by prefix ordering) consisting of all chains of `self`, each of which is given as an `element_class`.

If `conversion` is given, then the chains are converted to chain of elements of this list.

EXAMPLES:

```
sage: # needs sage.modules
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: A = H.chains()
sage: list(A)
[[], [0], [0, 1], [0, 1, 4], [0, 2], [0, 2, 3], [0, 2, 3, 4], [0, 2, 4],
 [0, 3], [0, 3, 4], [0, 4], [1], [1, 4], [2], [2, 3], [2, 3, 4], [2, 4],
 [3], [3, 4], [4]]
sage: A.cardinality()
20
sage: [1,3] in A
False
sage: [1,4] in A
True
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> P = posets.PentagonPoset()
>>> H = P._hasse_diagram
>>> A = H.chains()
>>> list(A)
[[], [0], [0, 1], [0, 1, 4], [0, 2], [0, 2, 3], [0, 2, 3, 4], [0, 2, 4],
 [0, 3], [0, 3, 4], [0, 4], [1], [1, 4], [2], [2, 3], [2, 3, 4], [2, 4],
 [3], [3, 4], [4]]
>>> A.cardinality()
20
>>> [Integer(1),Integer(3)] in A
False
>>> [Integer(1),Integer(4)] in A
True
```

One can exclude some vertices:

```
sage: # needs sage.modules
sage: list(H.chains(exclude=[4, 3]))
[[], [0], [0, 1], [0, 2], [1], [2]]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> list(H.chains(exclude=[Integer(4), Integer(3)]))
[[], [0], [0, 1], [0, 2], [1], [2]]
```

The `element_class` keyword determines how the chains are being returned:

```
sage: P = Poset({1: [2, 3], 2: [4]})
sage: list(P._hasse_diagram.chains(element_class=tuple))
[(), (0,), (0, 1), (0, 1, 2), (0, 2), (0, 3), (1,), (1, 2), (2,), (3,)]
sage: list(P._hasse_diagram.chains())
[[], [0], [0, 1], [0, 1, 2], [0, 2], [0, 3], [1], [1, 2], [2], [3]]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(2): [Integer(4)]}
↪)
>>> list(P._hasse_diagram.chains(element_class=tuple))
[(), (0,), (0, 1), (0, 1, 2), (0, 2), (0, 3), (1,), (1, 2), (2,), (3,)]
>>> list(P._hasse_diagram.chains())
[[], [0], [0, 1], [0, 1, 2], [0, 2], [0, 3], [1], [1, 2], [2], [3]]
```

(Note that taking the Hasse diagram has renamed the vertices.)

```
sage: list(P._hasse_diagram.chains(element_class=tuple, exclude=[0]))
[(), (1,), (1, 2), (2,), (3,)]
```

```
>>> from sage.all import *
>>> list(P._hasse_diagram.chains(element_class=tuple, exclude=[Integer(0)]))
[(), (1,), (1, 2), (2,), (3,)]
```

See also

antichains()

closed_interval(*x*, *y*)

Return a list of the elements z of `self` such that $x \leq z \leq y$.

The order is that induced by the ordering in `self.linear_extension`.

INPUT:

- `x` – any element of the poset
- `y` – any element of the poset

Note

The method `_precompute_intervals()` creates a cache which is used if available, making the function very fast.

See also

interval_iterator()

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: I = set([2,5,6,4,7])
sage: I == set(H.interval(2,7))
True
```

```
>>> from sage.all import *
>>> uc = [[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]]
>>> dag = DiGraph(dict(zip(range(len(uc)),uc)))
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram(dag)
>>> I = set([Integer(2),Integer(5),Integer(6),Integer(4),Integer(7)])
>>> I == set(H.interval(Integer(2),Integer(7)))
True
```

common_lower_covers(*vertices*)

Return the list of all common lower covers of `vertices`.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1,2], 1: [3], 2: [3], 3: []})
sage: H.common_lower_covers([1, 2])
[0]

sage: from sage.combinat.posets.poset_examples import Posets
sage: H = Posets.YoungDiagramPoset(Partition([3, 2, 2]))._hasse_diagram     #␣
↪needs sage.combinat sage.modules
sage: H.common_lower_covers([4, 5])                                          #␣
↪needs sage.combinat sage.modules
[3]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1),Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(3)], Integer(3): []})
>>> H.common_lower_covers([Integer(1), Integer(2)])
[0]

>>> from sage.combinat.posets.poset_examples import Posets
>>> H = Posets.YoungDiagramPoset(Partition([Integer(3), Integer(2),␣
↪Integer(2)]))._hasse_diagram     # needs sage.combinat sage.modules
>>> H.common_lower_covers([Integer(4), Integer(5)])                          ␣
↪              # needs sage.combinat sage.modules
[3]
```

common_upper_covers(*vertices*)

Return the list of all common upper covers of `vertices`.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1,2], 1: [3], 2: [3], 3: []})
sage: H.common_upper_covers([1, 2])
[3]

sage: from sage.combinat.posets.poset_examples import Posets
```

(continues on next page)

(continued from previous page)

```
sage: H = Posets.YoungDiagramPoset(Partition([3, 2, 2]))._hasse_diagram     #␣
↪needs sage.combinat sage.modules
sage: H.common_upper_covers([4, 5])                                          #␣
↪needs sage.combinat sage.modules
[6]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1),Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(3)], Integer(3): []})
>>> H.common_upper_covers([Integer(1), Integer(2)])
[3]

>>> from sage.combinat.posets.poset_examples import Posets
>>> H = Posets.YoungDiagramPoset(Partition([Integer(3), Integer(2),␣
↪Integer(2)]))._hasse_diagram     # needs sage.combinat sage.modules
>>> H.common_upper_covers([Integer(4), Integer(5)])                          ␣
↪              # needs sage.combinat sage.modules
[6]
```

congruence(*parts*, *start=None*, *stop_pairs=None*)

Return the congruence `start` "extended" by `parts`.

`start` is assumed to be a valid congruence of the lattice, and this is *not* checked.

INPUT:

- `parts` – list of lists; congruences to add
- `start` – a disjoint set; already computed congruence (or `None`)
- `stop_pairs` – list of pairs; list of pairs for stopping computation

OUTPUT:

`None`, if the congruence generated by `start` and `parts` together contains a block that has elements a, b so that `(a, b)` is in the list `stop_pairs`. Otherwise the least congruence that contains a block whose subset is p for every p in `parts` or `start`, given as `sage.sets.disjoint_set.DisjointSet_class`.

ALGORITHM:

Use the quadrilateral argument from page 120 of [Dav1997].

Basically we take one block from todo-list, search quadrilateral blocks up and down against the block, and then complete them to closed intervals and add to todo-list.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [3], 2: [4], 3: [4]})
sage: cong = H.congruence([[0, 1]]); cong                                    #␣
↪needs sage.modules
{{0, 1, 3}, {2, 4}}
sage: H.congruence([[0, 2]], start=cong)                                     #␣
↪needs sage.modules
{{0, 1, 2, 3, 4}}
```

(continues on next page)

(continued from previous page)

```
sage: H.congruence([[0, 1]], stop_pairs=[(1, 3)]) is None                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(4)], Integer(3): [Integer(4)]})
>>> cong = H.congruence([[Integer(0), Integer(1)]]); cong                     ␣
↪               # needs sage.modules
{{0, 1, 3}, {2, 4}}
>>> H.congruence([[Integer(0), Integer(2)]], start=cong)                      ␣
↪               # needs sage.modules
{{0, 1, 2, 3, 4}}

>>> H.congruence([[Integer(0), Integer(1)]], stop_pairs=[(Integer(1),␣
↪Integer(3))]) is None                          # needs sage.modules
True
```

congruences_iterator()

Return an iterator over all congruences of the lattice.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram('GY@OQ?OW@?O?')
sage: it = H.congruences_iterator(); it
<generator object ...>
sage: sorted([cong.number_of_subsets() for cong in it])                       #␣
↪needs sage.combinat sage.modules
[1, 2, 2, 2, 4, 4, 4, 8]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram('GY@OQ?OW@?O?')
>>> it = H.congruences_iterator(); it
<generator object ...>
>>> sorted([cong.number_of_subsets() for cong in it])                       #␣
↪needs sage.combinat sage.modules
[1, 2, 2, 2, 4, 4, 4, 8]
```

cover_relations()

Return the list of cover relations.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: H.cover_relations()
[(0, 2), (0, 3), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(2),Integer(3)],
↪Integer(1):[Integer(3),Integer(4)], Integer(2):[Integer(5)],
↪Integer(3):[Integer(5)], Integer(4):[Integer(5)]})
>>> H.cover_relations()
[(0, 2), (0, 3), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)]
```

cover_relations_iterator()

Iterate over cover relations.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: list(H.cover_relations_iterator())
[(0, 2), (0, 3), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(2),Integer(3)],
↪Integer(1):[Integer(3),Integer(4)], Integer(2):[Integer(5)],
↪Integer(3):[Integer(5)], Integer(4):[Integer(5)]})
>>> list(H.cover_relations_iterator())
[(0, 2), (0, 3), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)]
```

covers(*x*, *y*)

Return `True` if y covers x and `False` otherwise.

EXAMPLES:

```
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.covers(Q(1),Q(6))
True
sage: Q.covers(Q(1),Q(4))
False
```

```
>>> from sage.all import *
>>> Q = Poset([[Integer(1),Integer(5)],[Integer(2),Integer(6)],[Integer(3)],
↪[Integer(4)],[],[Integer(6),Integer(3)],[Integer(4)]])
>>> Q.covers(Q(Integer(1)),Q(Integer(6)))
True
>>> Q.covers(Q(Integer(1)),Q(Integer(4)))
False
```

coxeter_transformation(*algorithm='cython'*)

Return the matrix of the Auslander-Reiten translation acting on the Grothendieck group of the derived category of modules on the poset, in the basis of simple modules.

INPUT:

- `algorithm` – `'cython'` (default) or `'matrix'`

This uses either a specific matrix code in Cython, or generic matrices.

See also

lequal_matrix(), *moebius_function_matrix()*

EXAMPLES:

```
sage: # needs sage.libs.flint sage.modules
sage: P = posets.PentagonPoset()._hasse_diagram
sage: M = P.coxeter_transformation(); M
[ 0  0  0  0 -1]
[ 0  0  0  1 -1]
[ 0  1  0  0 -1]
[-1  1  1  0 -1]
[-1  1  0  1 -1]
sage: P.__dict__['coxeter_transformation'].clear_cache()
sage: P.coxeter_transformation(algorithm='matrix') == M
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.flint sage.modules
>>> P = posets.PentagonPoset()._hasse_diagram
>>> M = P.coxeter_transformation(); M
[ 0  0  0  0 -1]
[ 0  0  0  1 -1]
[ 0  1  0  0 -1]
[-1  1  1  0 -1]
[-1  1  0  1 -1]
>>> P.__dict__['coxeter_transformation'].clear_cache()
>>> P.coxeter_transformation(algorithm='matrix') == M
True
```

diamonds()

Return the list of diamonds of `self`.

A diamond is the following subgraph of the Hasse diagram:

```
  z
 / \
x   y
 \ /
  w
```

Thus each edge represents a cover relation in the Hasse diagram. We represent his as the tuple (w, x, y, z).

OUTPUT: a tuple with

- a list of all diamonds in the Hasse Diagram,
- a boolean checking that every w, x, y that form a `V`, there is a unique element z, which completes the diamond.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1,2], 1: [3], 2: [3], 3: []})
```

(continues on next page)

(continued from previous page)

```
sage: H.diamonds()
([(0, 1, 2, 3)], True)

sage: P = posets.YoungDiagramPoset(Partition([3, 2, 2]))                    #␣
↪needs sage.combinat sage.modules
sage: H = P._hasse_diagram                                                   #␣
↪needs sage.combinat sage.modules
sage: H.diamonds()                                                           #␣
↪needs sage.combinat sage.modules
([(0, 1, 3, 4), (3, 4, 5, 6)], False)
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1),Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(3)], Integer(3): []})
>>> H.diamonds()
([(0, 1, 2, 3)], True)

>>> P = posets.YoungDiagramPoset(Partition([Integer(3), Integer(2),␣
↪Integer(2)]))                    # needs sage.combinat sage.modules
>>> H = P._hasse_diagram                                                    #␣
↪needs sage.combinat sage.modules
>>> H.diamonds()                                                            #␣
↪needs sage.combinat sage.modules
([(0, 1, 3, 4), (3, 4, 5, 6)], False)
```

dual()

Return a poset that is dual to the given poset.

This means that it has the same elements but opposite order. The elements are renumbered to ensure that `range(n)` is a linear extension.

EXAMPLES:

```
sage: P = posets.IntegerPartitions(4)                                       #␣
↪needs sage.combinat
sage: H = P._hasse_diagram; H                                               #␣
↪needs sage.combinat
Hasse diagram of a poset containing 5 elements
sage: H.dual()                                                              #␣
↪needs sage.combinat
Hasse diagram of a poset containing 5 elements
```

```
>>> from sage.all import *
>>> P = posets.IntegerPartitions(Integer(4))                                 ␣
↪      # needs sage.combinat
>>> H = P._hasse_diagram; H                                                 #␣
↪needs sage.combinat
Hasse diagram of a poset containing 5 elements
>>> H.dual()                                                                #␣
↪needs sage.combinat
Hasse diagram of a poset containing 5 elements
```

find_nonsemidistributive_elements(*meet_or_join*)

Check if the lattice is semidistributive or not.

INPUT:

- `meet_or_join` – string `'meet'` or `'join'` to decide if to check for join-semidistributivity or meet-semidistributivity

OUTPUT:

- `None` if the lattice is semidistributive OR
- tuple `(u, e, x, y)` such that $u = e \vee x = e \vee y$ but $u \neq e \vee (x \wedge y)$ if `meet_or_join=='join'` and $u = e \wedge x = e \wedge y$ but $u \neq e \wedge (x \vee y)$ if `meet_or_join=='meet'`

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1, 2], 1:[3, 4], 2:[4, 5], 3:[6],
....:                   4:[6], 5:[6]})
sage: H.find_nonsemidistributive_elements('join') is None                   #␣
↪needs sage.modules
False
sage: H.find_nonsemidistributive_elements('meet') is None                   #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1), Integer(2)],␣
↪Integer(1):[Integer(3), Integer(4)], Integer(2):[Integer(4), Integer(5)],␣
↪Integer(3):[Integer(6)],
...                   Integer(4):[Integer(6)], Integer(5):[Integer(6)]})
>>> H.find_nonsemidistributive_elements('join') is None                   #␣
↪needs sage.modules
False
>>> H.find_nonsemidistributive_elements('meet') is None                   #␣
↪needs sage.modules
True
```

find_nonsemimodular_pair(*upper*)

Return pair of elements showing the lattice is not modular.

INPUT:

- `upper` – boolean; if `True`, test whether the lattice is upper semimodular. Otherwise test whether the lattice is lower semimodular.

OUTPUT:

`None`, if the lattice is semimodular. Pair (a, b) violating semimodularity otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1, 2], 1:[3, 4], 2:[4, 5], 3:[6], 4:[6], 5:[6]})
sage: H.find_nonsemimodular_pair(upper=True) is None
True
```

(continues on next page)

(continued from previous page)

```
sage: H.find_nonsemimodular_pair(upper=False)
(5, 3)

sage: H_ = HasseDiagram(H.reverse().relabel(lambda x: 6-x, inplace=False))
sage: H_.find_nonsemimodular_pair(upper=True)
(3, 1)
sage: H_.find_nonsemimodular_pair(upper=False) is None
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1), Integer(2)],
↪Integer(1):[Integer(3), Integer(4)], Integer(2):[Integer(4), Integer(5)],
↪Integer(3):[Integer(6)], Integer(4):[Integer(6)], Integer(5):[Integer(6)]})
>>> H.find_nonsemimodular_pair(upper=True) is None
True
>>> H.find_nonsemimodular_pair(upper=False)
(5, 3)

>>> H_ = HasseDiagram(H.reverse().relabel(lambda x: Integer(6)-x,
↪inplace=False))
>>> H_.find_nonsemimodular_pair(upper=True)
(3, 1)
>>> H_.find_nonsemimodular_pair(upper=False) is None
True
```

find_nontrivial_congruence()

Return a pair that generates non-trivial congruence or `None` if there is not any.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [5], 2: [3, 4], 3: [5], 4: [5]})
sage: H.find_nontrivial_congruence()                                    #
↪needs sage.modules
{{0, 1}, {2, 3, 4, 5}}

sage: H = HasseDiagram({0: [1, 2, 3], 1: [4], 2: [4], 3: [4]})
sage: H.find_nontrivial_congruence() is None                            #
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):
↪[Integer(5)], Integer(2): [Integer(3), Integer(4)], Integer(3):
↪[Integer(5)], Integer(4): [Integer(5)]})
>>> H.find_nontrivial_congruence()                                    #
↪needs sage.modules
{{0, 1}, {2, 3, 4, 5}}

>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2), Integer(3)],
```

(continues on next page)

(continued from previous page)

```
↪Integer(1): [Integer(4)], Integer(2): [Integer(4)], Integer(3):␣
↪[Integer(4)]})
>>> H.find_nontrivial_congruence() is None                              #␣
↪needs sage.modules
True
```

ALGORITHM:

See https://www.math.hawaii.edu/~ralph/Preprints/conlat.pdf:

If Θ is a join irreducible element of a Con(L), then there is at least one join-irreducible j and one meet-irreducible m such that Θ is both the principal congruence generated by (j^*, j), where j^* is the unique lower cover of j, and the principal congruence generated by (m, m^*), where m^* is the unique upper cover of m.

So, we only check join irreducibles or meet irreducibles, whichever is a smaller set. To optimize more we stop computation whenever it finds a pair that we know to generate one-element congruence.

frattini_sublattice()

Return the list of elements of the Frattini sublattice of the lattice.

EXAMPLES:

```
sage: H = posets.PentagonPoset()._hasse_diagram                          #␣
↪needs sage.modules
sage: H.frattini_sublattice()                                            #␣
↪needs sage.modules
[0, 4]
```

```
>>> from sage.all import *
>>> H = posets.PentagonPoset()._hasse_diagram                           #␣
↪needs sage.modules
>>> H.frattini_sublattice()                                             #␣
↪needs sage.modules
[0, 4]
```

greedy_linear_extensions_iterator()

Return an iterator over greedy linear extensions of the Hasse diagram.

A linear extension $[e_1, e_2, \ldots, e_n]$ is *greedy* if for every i either e_{i+1} covers e_i or all upper covers of e_i have at least one lower cover that is not in $[e_1, e_2, \ldots, e_i]$.

Informally said a linear extension is greedy if it "always goes up when possible" and so has no unnecessary jumps.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: N5 = HasseDiagram({0: [1, 2], 2: [3], 1: [4], 3: [4]})
sage: for l in N5.greedy_linear_extensions_iterator():
....:     print(l)
[0, 1, 2, 3, 4]
[0, 2, 3, 1, 4]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> N5 = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(2):␣
↪[Integer(3)], Integer(1): [Integer(4)], Integer(3): [Integer(4)]})
>>> for l in N5.greedy_linear_extensions_iterator():
...     print(l)
[0, 1, 2, 3, 4]
[0, 2, 3, 1, 4]
```

has_bottom()

Return True if the poset has a unique minimal element.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.has_bottom()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.has_bottom()
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)],Integer(4):[]})
>>> P.has_bottom()
False
>>> Q = Poset({Integer(0):[Integer(1)],Integer(1):[]})
>>> Q.has_bottom()
True
```

has_top()

Return True if the poset contains a unique maximal element, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.has_top()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.has_top()
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4),Integer(5)],Integer(4):[],
↪Integer(5):[]})
>>> P.has_top()
False
>>> Q = Poset({Integer(0):[Integer(1)],Integer(1):[]})
>>> Q.has_top()
True
```

interval(x, y)

Return a list of the elements z of self such that $x \leq z \leq y$.

The order is that induced by the ordering in `self.linear_extension`.

INPUT:

- `x` – any element of the poset
- `y` – any element of the poset

Note: The method `_precompute_intervals()` creates a cache which is used if available, making the function very fast.

See also:

interval_iterator()

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: I = set([2,5,6,4,7])
sage: I == set(H.interval(2,7))
True
```

```
>>> from sage.all import *
>>> uc = [[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]]
>>> dag = DiGraph(dict(zip(range(len(uc)),uc)))
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram(dag)
>>> I = set([Integer(2),Integer(5),Integer(6),Integer(4),Integer(7)])
>>> I == set(H.interval(Integer(2),Integer(7)))
True
```

interval_iterator(x, y)

Return an iterator of the elements z of `self` such that $x \leq z \leq y$.

INPUT:

- `x` – any element of the poset
- `y` – any element of the poset

See also:

interval()

Note

This becomes much faster when first calling `_leq_storage()`, which precomputes the principal upper ideals.

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: I = set([2,5,6,4,7])
sage: I == set(H.interval_iterator(2,7))
True
```

```
>>> from sage.all import *
>>> uc = [[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]]
>>> dag = DiGraph(dict(zip(range(len(uc)),uc)))
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram(dag)
>>> I = set([Integer(2),Integer(5),Integer(6),Integer(4),Integer(7)])
>>> I == set(H.interval_iterator(Integer(2),Integer(7)))
True
```

is_antichain_of_poset(*elms*)

Return `True` if `elms` is an antichain of the Hasse diagram and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2, 3], 1: [4], 2: [4], 3: [4]})
sage: H.is_antichain_of_poset([1, 2, 3])
True
sage: H.is_antichain_of_poset([0, 2, 3])
False
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2), Integer(3)],␣
↪Integer(1): [Integer(4)], Integer(2): [Integer(4)], Integer(3):␣
↪[Integer(4)]})
>>> H.is_antichain_of_poset([Integer(1), Integer(2), Integer(3)])
True
>>> H.is_antichain_of_poset([Integer(0), Integer(2), Integer(3)])
False
```

is_bounded()

Return `True` if the poset contains a unique maximal element and a unique minimal element, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.is_bounded()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.is_bounded()
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4),Integer(5)],Integer(4):[],
↪Integer(5):[]})
>>> P.is_bounded()
False
>>> Q = Poset({Integer(0):[Integer(1)],Integer(1):[]})
>>> Q.is_bounded()
True
```

is_chain()

Return `True` if the poset is totally ordered, and `False` otherwise.

EXAMPLES:

```
sage: L = Poset({0:[1],1:[2],2:[3],3:[4]})
sage: L.is_chain()
True
sage: V = Poset({0:[1,2]})
sage: V.is_chain()
False
```

```
>>> from sage.all import *
>>> L = Poset({Integer(0):[Integer(1)],Integer(1):[Integer(2)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)]})
>>> L.is_chain()
True
>>> V = Poset({Integer(0):[Integer(1),Integer(2)]})
>>> V.is_chain()
False
```

is_complemented()

Return an element of the lattice that has no complement.

If the lattice is complemented, return `None`.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram

sage: H = HasseDiagram({0:[1, 2], 1:[3], 2:[3], 3:[4]})
sage: H.is_complemented()                                                    #␣
↪needs sage.modules
1

sage: H = HasseDiagram({0:[1, 2, 3], 1:[4], 2:[4], 3:[4]})
sage: H.is_complemented() is None                                            #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram

>>> H = HasseDiagram({Integer(0):[Integer(1), Integer(2)],␣
↪Integer(1):[Integer(3)], Integer(2):[Integer(3)], Integer(3):[Integer(4)]})
>>> H.is_complemented()                                              #␣
↪needs sage.modules
1

>>> H = HasseDiagram({Integer(0):[Integer(1), Integer(2), Integer(3)],␣
↪Integer(1):[Integer(4)], Integer(2):[Integer(4)], Integer(3):[Integer(4)]})
>>> H.is_complemented() is None                                      #␣
↪needs sage.modules
True
```

is_congruence_normal()

Return `True` if the lattice can be constructed from the one-element lattice with Day doubling constructions of convex subsets.

Subsets to double does not need to be lower nor upper pseudo-intervals. On the other hand they must be convex, i.e. doubling a non-convex but municipal subset will give a lattice that returns `False` from this function.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram('IX?Q@?AG?OG?W?O@??')
sage: H.is_congruence_normal()                                         #␣
↪needs sage.combinat sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram('IX?Q@?AG?OG?W?O@??')
>>> H.is_congruence_normal()                                         #␣
↪needs sage.combinat sage.modules
True
```

The 5-element diamond is the smallest non-example:

```
sage: H = HasseDiagram({0: [1, 2, 3], 1: [4], 2: [4], 3: [4]})
sage: H.is_congruence_normal()                                         #␣
↪needs sage.combinat sage.modules
False
```

```
>>> from sage.all import *
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2), Integer(3)],␣
↪Integer(1): [Integer(4)], Integer(2): [Integer(4)], Integer(3):␣
↪[Integer(4)]})
```

(continues on next page)

(continued from previous page)

```
>>> H.is_congruence_normal()                                                    #␣
↪needs sage.combinat sage.modules
False
```

This is done by doubling a non-convex subset:

```
sage: H = HasseDiagram('OQC?a?@CO?G_C@?GA?O??_??@?BO?A_?G??C??_?@???')
sage: H.is_congruence_normal()                                                  #␣
↪needs sage.combinat sage.modules
False
```

```
>>> from sage.all import *
>>> H = HasseDiagram('OQC?a?@CO?G_C@?GA?O??_??@?BO?A_?G??C??_?@???')
>>> H.is_congruence_normal()                                                    #␣
↪needs sage.combinat sage.modules
False
```

ALGORITHM:

See http://www.math.hawaii.edu/~jb/inflation.pdf

is_convex_subset (S)

Return `True` if S is a convex subset of the poset, and `False` otherwise.

A subset S is *convex* in the poset if $b \in S$ whenever $a, c \in S$ and $a \le b \le c$.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: B3 = HasseDiagram({0: [1, 2, 4], 1: [3, 5], 2: [3, 6],
....:                    3: [7], 4: [5, 6], 5: [7], 6: [7]})
sage: B3.is_convex_subset([1, 3, 5, 4])  # Also connected
True
sage: B3.is_convex_subset([1, 3, 4])  # Not connected
True

sage: B3.is_convex_subset([0, 1, 2, 3, 6])  # No, 0 < 4 < 6
False
sage: B3.is_convex_subset([0, 1, 2, 7])  # No, 1 < 3 < 7.
False
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> B3 = HasseDiagram({Integer(0): [Integer(1), Integer(2), Integer(4)],␣
↪Integer(1): [Integer(3), Integer(5)], Integer(2): [Integer(3), Integer(6)],
...                    Integer(3): [Integer(7)], Integer(4): [Integer(5),␣
↪Integer(6)], Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> B3.is_convex_subset([Integer(1), Integer(3), Integer(5), Integer(4)])  #␣
↪Also connected
True
>>> B3.is_convex_subset([Integer(1), Integer(3), Integer(4)])  # Not connected
True

>>> B3.is_convex_subset([Integer(0), Integer(1), Integer(2), Integer(3),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(6)])  # No, 0 < 4 < 6
False
>>> B3.is_convex_subset([Integer(0), Integer(1), Integer(2), Integer(7)])  #␣
↪No, 1 < 3 < 7.
False
```

is_gequal(*x*, *y*)

Return `True` if `x` is greater than or equal to `y`, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: Q = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0,1,4
sage: Q.is_gequal(x,y)
False
sage: Q.is_gequal(y,x)
False
sage: Q.is_gequal(x,z)
False
sage: Q.is_gequal(z,x)
True
sage: Q.is_gequal(z,y)
True
sage: Q.is_gequal(z,z)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> Q = HasseDiagram({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> x,y,z = Integer(0),Integer(1),Integer(4)
>>> Q.is_gequal(x,y)
False
>>> Q.is_gequal(y,x)
False
>>> Q.is_gequal(x,z)
False
>>> Q.is_gequal(z,x)
True
>>> Q.is_gequal(z,y)
True
>>> Q.is_gequal(z,z)
True
```

is_greater_than(*x*, *y*)

Return `True` if `x` is greater than but not equal to `y`, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: Q = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0,1,4
```

(continues on next page)

(continued from previous page)

```
sage: Q.is_greater_than(x,y)
False
sage: Q.is_greater_than(y,x)
False
sage: Q.is_greater_than(x,z)
False
sage: Q.is_greater_than(z,x)
True
sage: Q.is_greater_than(z,y)
True
sage: Q.is_greater_than(z,z)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> Q = HasseDiagram({Integer(0):[Integer(2)], Integer(1):[Integer(2)],
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> x,y,z = Integer(0),Integer(1),Integer(4)
>>> Q.is_greater_than(x,y)
False
>>> Q.is_greater_than(y,x)
False
>>> Q.is_greater_than(x,z)
False
>>> Q.is_greater_than(z,x)
True
>>> Q.is_greater_than(z,y)
True
>>> Q.is_greater_than(z,z)
False
```

is_join_semilattice()

Return `True` if `self` has a join operation, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],
↪7:[]})
sage: H.is_join_semilattice()                                          #␣
↪needs sage.modules
True
sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.is_join_semilattice()                                          #␣
↪needs sage.modules
False
sage: H = HasseDiagram({0:[2,3],1:[2,3],2:[4],3:[4]})
sage: H.is_join_semilattice()                                          #␣
↪needs sage.modules
False
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(3),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(4),Integer(5),Integer(6)],
↪Integer(3):[Integer(6)],Integer(4):[Integer(7)],Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)],Integer(7):[]})
>>> H.is_join_semilattice()                                             #␣
↪needs sage.modules
True
>>> H = HasseDiagram({Integer(0):[Integer(2),Integer(3)],
↪Integer(1):[Integer(2),Integer(3)]})
>>> H.is_join_semilattice()                                             #␣
↪needs sage.modules
False
>>> H = HasseDiagram({Integer(0):[Integer(2),Integer(3)],
↪Integer(1):[Integer(2),Integer(3)],Integer(2):[Integer(4)],
↪Integer(3):[Integer(4)]})
>>> H.is_join_semilattice()                                             #␣
↪needs sage.modules
False
```

is_lequal(*i*, *j*)

Return `True` if i is less than or equal to j in the poset, and `False` otherwise.

> **Note**
>
> If the *lequal_matrix()* has been computed, then this method is redefined to use the cached data (see `_alternate_is_lequal()`).

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0, 1, 4
sage: H.is_lequal(x,y)
False
sage: H.is_lequal(y,x)
False
sage: H.is_lequal(x,z)
True
sage: H.is_lequal(y,z)
True
sage: H.is_lequal(z,z)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> x,y,z = Integer(0), Integer(1), Integer(4)
>>> H.is_lequal(x,y)
False
```

(continues on next page)

(continued from previous page)

```
>>> H.is_lequal(y,x)
False
>>> H.is_lequal(x,z)
True
>>> H.is_lequal(y,z)
True
>>> H.is_lequal(z,z)
True
```

is_less_than(*x*, *y*)

Return `True` if `x` is less than but not equal to `y` in the poset, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0, 1, 4
sage: H.is_less_than(x,y)
False
sage: H.is_less_than(y,x)
False
sage: H.is_less_than(x,z)
True
sage: H.is_less_than(y,z)
True
sage: H.is_less_than(z,z)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> x,y,z = Integer(0), Integer(1), Integer(4)
>>> H.is_less_than(x,y)
False
>>> H.is_less_than(y,x)
False
>>> H.is_less_than(x,z)
True
>>> H.is_less_than(y,z)
True
>>> H.is_less_than(z,z)
False
```

is_linear_extension(*lin_ext=None*)

Test if an ordering is a linear extension.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.is_linear_extension(list(range(4)))
True
```

(continues on next page)

(continued from previous page)

```
sage: H.is_linear_extension([3,2,1,0])
False
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(2)],
↪Integer(1):[Integer(3)],Integer(2):[Integer(3)],Integer(3):[]})
>>> H.is_linear_extension(list(range(Integer(4))))
True
>>> H.is_linear_extension([Integer(3),Integer(2),Integer(1),Integer(0)])
False
```

is_linear_interval (*t_min*, *t_max*)

Return whether the interval `[t_min, t_max]` is linear.

This means that this interval is a total order.

EXAMPLES::

sage: # needs sage.modules sage: P = posets.PentagonPoset() sage: H = P._hasse_diagram sage: H.is_linear_interval(0, 4) False sage: H.is_linear_interval(0, 3) True sage: H.is_linear_interval(1, 3) False sage: H.is_linear_interval(1, 1) True

is_meet_semilattice ()

Return `True` if `self` has a meet operation, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],
↪7:[]})
sage: H.is_meet_semilattice()                                          #␣
↪needs sage.modules
True

sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.is_meet_semilattice()                                          #␣
↪needs sage.modules
True

sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.is_meet_semilattice()                                          #␣
↪needs sage.modules
False

sage: H = HasseDiagram({0:[1,2],1:[3,4],2:[3,4]})
sage: H.is_meet_semilattice()                                          #␣
↪needs sage.modules
False
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(3),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(4),Integer(5),Integer(6)],
```

(continues on next page)

(continued from previous page)

```
↪Integer(3):[Integer(6)],Integer(4):[Integer(7)],Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)],Integer(7):[]})
>>> H.is_meet_semilattice()                                             #␣
↪needs sage.modules
True

>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(2)],
↪Integer(1):[Integer(3)],Integer(2):[Integer(3)],Integer(3):[]})
>>> H.is_meet_semilattice()                                             #␣
↪needs sage.modules
True

>>> H = HasseDiagram({Integer(0):[Integer(2),Integer(3)],
↪Integer(1):[Integer(2),Integer(3)]})
>>> H.is_meet_semilattice()                                             #␣
↪needs sage.modules
False

>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(2)],
↪Integer(1):[Integer(3),Integer(4)],Integer(2):[Integer(3),Integer(4)]})
>>> H.is_meet_semilattice()                                             #␣
↪needs sage.modules
False
```

is_ranked()

Return `True` if the poset is ranked, and `False` otherwise.

A poset is *ranked* if it admits a rank function. For more information about the rank function, see *rank_function()* and `is_graded()`.

EXAMPLES:

```
sage: P = Poset([[1],[2],[3],[4],[]])
sage: P.is_ranked()
True
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.is_ranked()
False
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1)],[Integer(2)],[Integer(3)],[Integer(4)],[]])
>>> P.is_ranked()
True
>>> Q = Poset([[Integer(1),Integer(5)],[Integer(2),Integer(6)],[Integer(3)],
↪[Integer(4)],[],[Integer(6),Integer(3)],[Integer(4)]])
>>> Q.is_ranked()
False
```

join_matrix()

Return the matrix of joins of `self`, when `self` is a join-semilattice; raise an error otherwise.

The `(x,y)`-entry of this matrix is the join of `x` and `y` in `self`.

This algorithm is modelled after the algorithm of Freese-Jezek-Nation (p217). It can also be found on page

140 of [Gec81].

Note

If `self` is a join-semilattice, then the return of this method is the same as `_join()`. Once the matrix has been computed, it is stored in `_join()`. Delete this attribute if you want to recompute the matrix.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],
↪7:[]})
sage: H.join_matrix()                                                   #␣
↪needs sage.modules
[0 1 2 3 4 5 6 7]
[1 1 4 7 4 7 7 7]
[2 4 2 6 4 5 6 7]
[3 7 6 3 7 7 6 7]
[4 4 4 7 4 7 7 7]
[5 7 5 7 7 5 7 7]
[6 7 6 6 7 7 6 7]
[7 7 7 7 7 7 7 7]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(3),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(4),Integer(5),Integer(6)],
↪Integer(3):[Integer(6)],Integer(4):[Integer(7)],Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)],Integer(7):[]})
>>> H.join_matrix()                                                     #␣
↪needs sage.modules
[0 1 2 3 4 5 6 7]
[1 1 4 7 4 7 7 7]
[2 4 2 6 4 5 6 7]
[3 7 6 3 7 7 6 7]
[4 4 4 7 4 7 7 7]
[5 7 5 7 7 5 7 7]
[6 7 6 6 7 7 6 7]
[7 7 7 7 7 7 7 7]
```

kappa(*a*)

Return the maximum element greater than the element covered by `a` but not greater than `a`.

Define $\kappa(a)$ as the maximum element of $(\uparrow a_*) \setminus (\uparrow a)$, where a_* is the element covered by a. It is always a meet-irreducible element, if it exists.

Note

Element `a` is expected to be join-irreducible, and this is *not* checked.

INPUT:

- `a` – a join-irreducible element of the lattice

OUTPUT:

The element $\kappa(a)$ or `None` if there is not a unique greatest element with given constraints.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2, 3], 1: [4], 2: [4, 5], 3: [5], 4: [6], 5:␣
↪[6]})
sage: H.kappa(1)
5
sage: H.kappa(2) is None
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2), Integer(3)],␣
↪Integer(1): [Integer(4)], Integer(2): [Integer(4), Integer(5)], Integer(3):␣
↪[Integer(5)], Integer(4): [Integer(6)], Integer(5): [Integer(6)]})
>>> H.kappa(Integer(1))
5
>>> H.kappa(Integer(2)) is None
True
```

kappa_dual(*a*)

Return the minimum element smaller than the element covering `a` but not smaller than `a`.

Define $\kappa^*(a)$ as the minimum element of $(\downarrow a_*) \setminus (\downarrow a)$, where a_* is the element covering a. It is always a join-irreducible element, if it exists.

> **Note**
>
> Element `a` is expected to be meet-irreducible, and this is *not* checked.

INPUT:

- `a` – a join-irreducible element of the lattice

OUTPUT:

The element $\kappa^*(a)$ or `None` if there is not a unique smallest element with given constraints.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [3, 4], 2: [4, 5], 3: [6], 4: [6], 5:␣
↪[6]})
sage: H.kappa_dual(3)
2
sage: H.kappa_dual(4) is None
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(3), Integer(4)], Integer(2): [Integer(4), Integer(5)], Integer(3):␣
↪[Integer(6)], Integer(4): [Integer(6)], Integer(5): [Integer(6)]})
>>> H.kappa_dual(Integer(3))
2
>>> H.kappa_dual(Integer(4)) is None
True
```

lequal_matrix(*boolean=False*)

Return a matrix whose `(i,j)` entry is 1 if `i` is less than `j` in the poset, and 0 otherwise; and redefines `__lt__` to use the boolean version of this matrix.

INPUT:

- `boolean` – flag (default: `False`); whether to return a matrix with coefficients in $\mathbf{F}_(2)$ or in $\mathbf{Z}$

See also

moebius_function_matrix(), *coxeter_transformation()*

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: H = P._hasse_diagram
sage: M = H.lequal_matrix(); M                                          #␣
↪needs sage.modules
[1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 0 1]
[0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 1]
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1]
sage: M.base_ring()                                                     #␣
↪needs sage.modules
Integer Ring

sage: P = posets.DiamondPoset(6)
sage: H = P._hasse_diagram
sage: M = H.lequal_matrix(boolean=True)                                 #␣
↪needs sage.modules
sage: M.base_ring()                                                     #␣
↪needs sage.modules
Finite Field of size 2
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]])
>>> H = P._hasse_diagram
>>> M = H.lequal_matrix(); M                                          #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
[1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 0 1]
[0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 1]
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1]
>>> M.base_ring()                                                          #␣
↪needs sage.modules
Integer Ring

>>> P = posets.DiamondPoset(Integer(6))
>>> H = P._hasse_diagram
>>> M = H.lequal_matrix(boolean=True)                                      #␣
↪needs sage.modules
>>> M.base_ring()                                                          #␣
↪needs sage.modules
Finite Field of size 2
```

linear_extension()

Return a linear extension.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.linear_extension()
[0, 1, 2, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(2)],
↪Integer(1):[Integer(3)],Integer(2):[Integer(3)],Integer(3):[]})
>>> H.linear_extension()
[0, 1, 2, 3]
```

linear_extensions()

Return an iterator over all linear extensions.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: list(H.linear_extensions())                                          #␣
↪needs sage.modules
[[0, 1, 2, 3], [0, 2, 1, 3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(2)],
↪Integer(1):[Integer(3)],Integer(2):[Integer(3)],Integer(3):[]})
```

(continues on next page)

(continued from previous page)

```
>>> list(H.linear_extensions())                                        #␣
↪needs sage.modules
[[0, 1, 2, 3], [0, 2, 1, 3]]
```

lower_covers_iterator(*element*)

Return the list of elements that are covered by `element`.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],
↪7:[]})
sage: list(H.lower_covers_iterator(0))
[]
sage: list(H.lower_covers_iterator(4))
[1, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(3),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(4),Integer(5),Integer(6)],
↪Integer(3):[Integer(6)],Integer(4):[Integer(7)],Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)],Integer(7):[]})
>>> list(H.lower_covers_iterator(Integer(0)))
[]
>>> list(H.lower_covers_iterator(Integer(4)))
[1, 2]
```

maximal_elements()

Return a list of the maximal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.maximal_elements()
[4]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)],Integer(4):[]})
>>> P.maximal_elements()
[4]
```

maximal_sublattices()

Return maximal sublattices of the lattice.

EXAMPLES:

```
sage: L = posets.PentagonPoset()                                        #␣
↪needs sage.modules
sage: ms = L._hasse_diagram.maximal_sublattices()                       #␣
↪needs sage.modules
sage: sorted(ms, key=sorted)                                            #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
[{0, 1, 2, 4}, {0, 1, 3, 4}, {0, 2, 3, 4}]
```

```
>>> from sage.all import *
>>> L = posets.PentagonPoset()                                              #␣
↪needs sage.modules
>>> ms = L._hasse_diagram.maximal_sublattices()                             #␣
↪needs sage.modules
>>> sorted(ms, key=sorted)                                                  #␣
↪needs sage.modules
[{0, 1, 2, 4}, {0, 1, 3, 4}, {0, 2, 3, 4}]
```

meet_matrix()

Return the matrix of meets of `self`, when `self` is a meet-semilattice; raise an error otherwise.

The `(x,y)`-entry of this matrix is the meet of `x` and `y` in `self`.

This algorithm is modelled after the algorithm of Freese-Jezek-Nation (p217). It can also be found on page 140 of [Gec81].

Note

If `self` is a meet-semilattice, then the return of this method is the same as `_meet()`. Once the matrix has been computed, it is stored in `_meet()`. Delete this attribute if you want to recompute the matrix.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],
↪7:[]})
sage: H.meet_matrix()                                                         #␣
↪needs sage.modules
[0 0 0 0 0 0 0 0]
[0 1 0 0 1 0 0 1]
[0 0 2 0 2 2 2 2]
[0 0 0 3 0 0 3 3]
[0 1 2 0 4 2 2 4]
[0 0 2 0 2 5 2 5]
[0 0 2 3 2 2 6 6]
[0 1 2 3 4 5 6 7]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(3),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(4),Integer(5),Integer(6)],
↪Integer(3):[Integer(6)],Integer(4):[Integer(7)],Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)],Integer(7):[]})
>>> H.meet_matrix()                                                         #␣
↪needs sage.modules
[0 0 0 0 0 0 0 0]
[0 1 0 0 1 0 0 1]
```

(continues on next page)

(continued from previous page)

```
[0 0 2 0 2 2 2 2]
[0 0 0 3 0 0 3 3]
[0 1 2 0 4 2 2 4]
[0 0 2 0 2 5 2 5]
[0 0 2 3 2 2 6 6]
[0 1 2 3 4 5 6 7]
```

REFERENCE:

minimal_elements()

Return a list of the minimal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P(0) in P.minimal_elements()
True
sage: P(1) in P.minimal_elements()
True
sage: P(2) in P.minimal_elements()
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)],Integer(4):[]})
>>> P(Integer(0)) in P.minimal_elements()
True
>>> P(Integer(1)) in P.minimal_elements()
True
>>> P(Integer(2)) in P.minimal_elements()
True
```

moebius_function(i, j)

Return the value of the Möbius function of the poset on the elements `i` and `j`.

EXAMPLES:

```
sage: P = Poset([[1,2,3],[4],[4],[4],[]])
sage: H = P._hasse_diagram
sage: H.moebius_function(0,4)
2
sage: for u,v in P.cover_relations_iterator():
....:     if P.moebius_function(u,v) != -1:
....:         print("Bug in moebius_function!")
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(2),Integer(3)],[Integer(4)],[Integer(4)],
↪[Integer(4)],[]])
>>> H = P._hasse_diagram
>>> H.moebius_function(Integer(0),Integer(4))
2
>>> for u,v in P.cover_relations_iterator():
```

(continues on next page)

(continued from previous page)

```
...     if P.moebius_function(u,v) != -Integer(1):
...         print("Bug in moebius_function!")
```

moebius_function_matrix(*algorithm='cython'*)

Return the matrix of the Möbius function of this poset.

This returns the matrix over **Z** whose `(x, y)` entry is the value of the Möbius function of `self` evaluated on `x` and `y`, and redefines *moebius_function()* to use it.

INPUT:

- `algorithm` – `'recursive'`, `'matrix'` or `'cython'` (default)

This uses either the recursive formula, a generic matrix inversion or a specific matrix inversion coded in Cython.

OUTPUT: a dense matrix for the algorithm `cython`, a sparse matrix otherwise

Note: The result is cached in `_moebius_function_matrix()`.

See also:

lequal_matrix(), *coxeter_transformation()*

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],
↪7:[]})
sage: H.moebius_function_matrix()                                    #␣
↪needs sage.libs.flint sage.modules
[ 1 -1 -1 -1  1  0  1  0]
[ 0  1  0  0 -1  0  0  0]
[ 0  0  1  0 -1 -1 -1  2]
[ 0  0  0  1  0  0 -1  0]
[ 0  0  0  0  1  0  0 -1]
[ 0  0  0  0  0  1  0 -1]
[ 0  0  0  0  0  0  1 -1]
[ 0  0  0  0  0  0  0  1]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(3),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(4),Integer(5),Integer(6)],
↪Integer(3):[Integer(6)],Integer(4):[Integer(7)],Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)],Integer(7):[]})
>>> H.moebius_function_matrix()                                    #␣
↪needs sage.libs.flint sage.modules
[ 1 -1 -1 -1  1  0  1  0]
[ 0  1  0  0 -1  0  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  1  0 -1 -1 -1  2]
[ 0  0  0  1  0  0 -1  0]
[ 0  0  0  0  1  0  0 -1]
[ 0  0  0  0  0  1  0 -1]
[ 0  0  0  0  0  0  1 -1]
[ 0  0  0  0  0  0  0  1]
```

neutral_elements()

Return the list of neutral elements of the lattice.

An element a in a lattice is neutral if the sublattice generated by a, x and y is distributive for every x, y in the lattice.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [4], 2: [3], 3: [4, 5],
....:                    4: [6], 5: [6]})
sage: sorted(H.neutral_elements())                                          #␣
↪needs sage.modules
[0, 4, 6]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(4)], Integer(2): [Integer(3)], Integer(3): [Integer(4),␣
↪Integer(5)],
...                    Integer(4): [Integer(6)], Integer(5): [Integer(6)]})
>>> sorted(H.neutral_elements())                                          #␣
↪needs sage.modules
[0, 4, 6]
```

ALGORITHM:

Basically we just check the distributivity against all element pairs x, y to see if element a is neutral or not.

If we found that a, x, y is not a distributive triple, we add all three to list of non-neutral elements. If we found a to be neutral, we add it to list of neutral elements. When testing we skip already found neutral elements, as they can't be our x or y.

We skip a, x, y as trivial if it is a chain. We do that by letting x to be a non-comparable to a; y can be any element.

We first try to found x and y from elements not yet tested, so that we could get three birds with one stone.

And last, the top and bottom elements are always neutral and need not be tested.

open_interval(x, y)

Return a list of the elements z of `self` such that $x < z < y$.

The order is that induced by the ordering in `self.linear_extension`.

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
```

(continues on next page)

(continued from previous page)

```
sage: H = HasseDiagram(dag)
sage: set([5,6,4]) == set(H.open_interval(2,7))
True
sage: H.open_interval(7,2)
[]
```

```
>>> from sage.all import *
>>> uc = [[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]]
>>> dag = DiGraph(dict(zip(range(len(uc)),uc)))
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram(dag)
>>> set([Integer(5),Integer(6),Integer(4)]) == set(H.open_interval(Integer(2),
↪Integer(7)))
True
>>> H.open_interval(Integer(7),Integer(2))
[]
```

order_filter(*elements*)

Return the order filter generated by a list of elements.

I is an order filter if, for any x in I and y such that $y \geq x$, then y is in I.

EXAMPLES:

```
sage: H = posets.BooleanLattice(4)._hasse_diagram
sage: H.order_filter([3,8])
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

```
>>> from sage.all import *
>>> H = posets.BooleanLattice(Integer(4))._hasse_diagram
>>> H.order_filter([Integer(3),Integer(8)])
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

order_ideal(*elements*)

Return the order ideal generated by a list of elements.

I is an order ideal if, for any x in I and y such that $y \leq x$, then y is in I.

EXAMPLES:

```
sage: H = posets.BooleanLattice(4)._hasse_diagram
sage: H.order_ideal([7,10])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

```
>>> from sage.all import *
>>> H = posets.BooleanLattice(Integer(4))._hasse_diagram
>>> H.order_ideal([Integer(7),Integer(10)])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

order_ideal_cardinality(*elements*)

Return the cardinality of the order ideal generated by `elements`.

I is an order ideal if, for any x in I and y such that $y \le x$, then y is in I.

EXAMPLES:

```
sage: H = posets.BooleanLattice(4)._hasse_diagram
sage: H.order_ideal_cardinality([7,10])
10
```

```
>>> from sage.all import *
>>> H = posets.BooleanLattice(Integer(4))._hasse_diagram
>>> H.order_ideal_cardinality([Integer(7),Integer(10)])
10
```

orthocomplementations_iterator()

Return an iterator over orthocomplementations of the lattice.

OUTPUT: an iterator that gives plain list of integers

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2], 1:[3,4], 3:[5], 4:[5], 2:[6,7],
....:                   6:[8], 7:[8], 5:[9], 8:[9]})
sage: list(H.orthocomplementations_iterator())                          #␣
↪needs sage.groups
[[9, 8, 5, 6, 7, 2, 3, 4, 1, 0], [9, 8, 5, 7, 6, 2, 4, 3, 1, 0]]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(2)],␣
↪Integer(1):[Integer(3),Integer(4)], Integer(3):[Integer(5)],␣
↪Integer(4):[Integer(5)], Integer(2):[Integer(6),Integer(7)],
...                   Integer(6):[Integer(8)], Integer(7):[Integer(8)],␣
↪Integer(5):[Integer(9)], Integer(8):[Integer(9)]})
>>> list(H.orthocomplementations_iterator())                          #␣
↪needs sage.groups
[[9, 8, 5, 6, 7, 2, 3, 4, 1, 0], [9, 8, 5, 7, 6, 2, 4, 3, 1, 0]]
```

ALGORITHM:

As `DiamondPoset(2*n+2)` has $(2n)!/(n!2^n)$ different orthocomplementations, the complexity of listing all of them is necessarily $O(n!)$.

An orthocomplemented lattice is self-dual, so that for example orthocomplement of an atom is a coatom. This function basically just computes list of possible orthocomplementations for every element (i.e. they must be complements and "duals"), and then tries to fit them all.

prime_elements()

Return the join-prime and meet-prime elements of the bounded poset.

An element x of a poset P is join-prime if the subposet induced by $\{y \in P \mid y \not\geq x\}$ has a top element. Meet-prime is defined dually.

> **Note**
>
> The poset is expected to be bounded, and this is *not* checked.

OUTPUT:

A pair (j, m) where j is a list of join-prime elements and m is a list of meet-prime elements.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [3], 2: [4], 3: [4]})
sage: H.prime_elements()
([1, 2], [2, 3])
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(4)], Integer(3): [Integer(4)]})
>>> H.prime_elements()
([1, 2], [2, 3])
```

principal_congruences_poset()

Return the poset of join-irreducibles of the congruence lattice.

OUTPUT:

A pair (P, D) where P is a poset and D is a dictionary.

Elements of P are pairs (x, y) such that x is an element of the lattice and y is an element covering it. In the poset (a, b) is less than (c, d) iff the principal congruence generated by (a, b) is refinement of the principal congruence generated by (c, d).

D is a dictionary from pairs (x, y) to the congruence (given as DisjointSet) generated by the pair.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: N5 = HasseDiagram({0: [1, 2], 1: [4], 2: [3], 3: [4]})
sage: P, D = N5.principal_congruences_poset()                              #␣
↪needs sage.combinat sage.modules
sage: P                                                                    #␣
↪needs sage.combinat sage.modules
Finite poset containing 3 elements
sage: P.bottom()                                                           #␣
↪needs sage.combinat sage.modules
(2, 3)
sage: D[(2, 3)]                                                            #␣
↪needs sage.combinat sage.modules
{{0}, {1}, {2, 3}, {4}}
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> N5 = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(4)], Integer(2): [Integer(3)], Integer(3): [Integer(4)]})
>>> P, D = N5.principal_congruences_poset()                              #␣
↪needs sage.combinat sage.modules
>>> P                                                                    #␣
↪needs sage.combinat sage.modules
Finite poset containing 3 elements
```

(continues on next page)

(continued from previous page)

```
>>> P.bottom()                                                                    #␣
↪needs sage.combinat sage.modules
(2, 3)
>>> D[(Integer(2), Integer(3))]                                                    ␣
↪              # needs sage.combinat sage.modules
{{0}, {1}, {2, 3}, {4}}
```

principal_order_filter(*i*)

Return the order filter generated by `i`.

EXAMPLES:

```
sage: H = posets.BooleanLattice(4)._hasse_diagram
sage: H.principal_order_filter(2)
[2, 3, 6, 7, 10, 11, 14, 15]
```

```
>>> from sage.all import *
>>> H = posets.BooleanLattice(Integer(4))._hasse_diagram
>>> H.principal_order_filter(Integer(2))
[2, 3, 6, 7, 10, 11, 14, 15]
```

principal_order_ideal(*i*)

Return the order ideal generated by i.

EXAMPLES:

```
sage: H = posets.BooleanLattice(4)._hasse_diagram
sage: H.principal_order_ideal(6)
[0, 2, 4, 6]
```

```
>>> from sage.all import *
>>> H = posets.BooleanLattice(Integer(4))._hasse_diagram
>>> H.principal_order_ideal(Integer(6))
[0, 2, 4, 6]
```

pseudocomplement(*element*)

Return the pseudocomplement of `element`, if it exists.

The pseudocomplement is the greatest element whose meet with given element is the bottom element. It may not exist, and then the function returns `None`.

INPUT:

- `element` – an element of the lattice

OUTPUT:

An element of the Hasse diagram, i.e. an integer, or `None` if the pseudocomplement does not exist.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [3], 2: [4], 3: [4]})
sage: H.pseudocomplement(2)                                                        #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
3

sage: H = HasseDiagram({0: [1, 2, 3], 1: [4], 2: [4], 3: [4]})
sage: H.pseudocomplement(2) is None                                   #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(4)], Integer(3): [Integer(4)]})
>>> H.pseudocomplement(Integer(2))                                     ␣
↪     # needs sage.modules
3

>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2), Integer(3)],␣
↪Integer(1): [Integer(4)], Integer(2): [Integer(4)], Integer(3):␣
↪[Integer(4)]})
>>> H.pseudocomplement(Integer(2)) is None                             ␣
↪     # needs sage.modules
True
```

rank(*element=None*)

Return the rank of `element`, or the rank of the poset if `element` is `None`. (The rank of a poset is the length of the longest chain of elements of the poset.)

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],
↪7:[]})
sage: H.rank(5)
2
sage: H.rank()
3
sage: Q = HasseDiagram({0:[1,2],1:[3],2:[],3:[]})
sage: Q.rank()
2
sage: Q.rank(1)
1
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(3),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(4),Integer(5),Integer(6)],
↪Integer(3):[Integer(6)],Integer(4):[Integer(7)],Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)],Integer(7):[]})
>>> H.rank(Integer(5))
2
>>> H.rank()
3
>>> Q = HasseDiagram({Integer(0):[Integer(1),Integer(2)],
```

(continues on next page)

(continued from previous page)

```
↪Integer(1):[Integer(3)],Integer(2):[],Integer(3):[]})
>>> Q.rank()
2
>>> Q.rank(Integer(1))
1
```

rank_function()

Return the (normalized) rank function of the poset, if it exists.

A *rank function* of a poset P is a function r that maps elements of P to integers and satisfies: $r(x) = r(y) + 1$ if x covers y. The function r is normalized such that its minimum value on every connected component of the Hasse diagram of P is 0. This determines the function r uniquely (when it exists).

OUTPUT:

- a lambda function, if the poset admits a rank function
- `None`, if the poset does not admit a rank function

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: P.rank_function() is not None
True
sage: P = Poset(([1,2,3,4],[[1,4],[2,3],[3,4]]), facade = True)
sage: P.rank_function() is not None
True
sage: P = Poset(([1,2,3,4,5],[[1,2],[2,3],[3,4],[1,5],[5,4]]), facade = True)
sage: P.rank_function() is not None
False
sage: P = Poset(([1,2,3,4,5,6,7,8],[[1,4],[2,3],[3,4],[5,7],[6,7]]), facade =␣
↪True)
sage: f = P.rank_function(); f is not None
True
sage: f(5)
0
sage: f(2)
0
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]])
>>> P.rank_function() is not None
True
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)],[[Integer(1),
↪Integer(4)],[Integer(2),Integer(3)],[Integer(3),Integer(4)]]), facade =␣
↪True)
>>> P.rank_function() is not None
True
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],
↪[[Integer(1),Integer(2)],[Integer(2),Integer(3)],[Integer(3),Integer(4)],
↪[Integer(1),Integer(5)],[Integer(5),Integer(4)]]), facade = True)
>>> P.rank_function() is not None
```

(continues on next page)

(continued from previous page)

```
False
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7),Integer(8)],[[Integer(1),Integer(4)],[Integer(2),
↪Integer(3)],[Integer(3),Integer(4)],[Integer(5),Integer(7)],[Integer(6),
↪Integer(7)]]), facade = True)
>>> f = P.rank_function(); f is not None
True
>>> f(Integer(5))
0
>>> f(Integer(2))
0
```

skeleton()

Return the skeleton of the lattice.

The lattice is expected to be pseudocomplemented and non-empty.

The skeleton of the lattice is the subposet induced by those elements that are the pseudocomplement to at least one element.

OUTPUT:

List of elements such that the subposet induced by them is the skeleton of the lattice.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [3, 4], 2: [4],
....:                   3: [5], 4: [5]})
sage: H.skeleton()                                                    #␣
↪needs sage.modules
[5, 2, 0, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(3), Integer(4)], Integer(2): [Integer(4)],
...                   Integer(3): [Integer(5)], Integer(4): [Integer(5)]})
>>> H.skeleton()                                                      #␣
↪needs sage.modules
[5, 2, 0, 3]
```

sublattices_iterator(*elms*, *min_e*)

Return an iterator over sublattices of the Hasse diagram.

INPUT:

- `elms` – elements already in sublattice; use set() at start
- `min_e` – smallest new element to add for new sublattices

OUTPUT: list of sublattices as sets of integers

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1:[3], 2:[3]})
```

(continues on next page)

(continued from previous page)

```
sage: it = H.sublattices_iterator(set(), 0); it
<generator object ...sublattices_iterator at ...>
sage: next(it)                                                              #␣
↪needs sage.modules
set()
sage: next(it)                                                              #␣
↪needs sage.modules
{0}
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)],␣
↪Integer(1):[Integer(3)], Integer(2):[Integer(3)]})
>>> it = H.sublattices_iterator(set(), Integer(0)); it
<generator object ...sublattices_iterator at ...>
>>> next(it)                                                              #␣
↪needs sage.modules
set()
>>> next(it)                                                              #␣
↪needs sage.modules
{0}
```

supergreedy_linear_extensions_iterator()

Return an iterator over supergreedy linear extensions of the Hasse diagram.

A linear extension $[e_1, e_2, \ldots, e_n]$ is *supergreedy* if, for every i and j where $i > j$, e_i covers e_j if for every $i > k > j$ at least one lower cover of e_k is not in $[e_1, e_2, \ldots, e_k]$.

Informally said a linear extension is supergreedy if it "always goes as high possible, and withdraw so less as possible". These are also called depth-first linear extensions.

EXAMPLES:

We show the difference between "only greedy" and supergreedy extensions:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 2: [3, 4]})
sage: G_ext = list(H.greedy_linear_extensions_iterator())
sage: SG_ext = list(H.supergreedy_linear_extensions_iterator())
sage: [0, 2, 3, 1, 4] in G_ext
True
sage: [0, 2, 3, 1, 4] in SG_ext
False

sage: len(SG_ext)
4
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0): [Integer(1), Integer(2)], Integer(2):␣
↪[Integer(3), Integer(4)]})
>>> G_ext = list(H.greedy_linear_extensions_iterator())
>>> SG_ext = list(H.supergreedy_linear_extensions_iterator())
```

(continues on next page)

(continued from previous page)

```
>>> [Integer(0), Integer(2), Integer(3), Integer(1), Integer(4)] in G_ext
True
>>> [Integer(0), Integer(2), Integer(3), Integer(1), Integer(4)] in SG_ext
False

>>> len(SG_ext)
4
```

top()

Return the top element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.top() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.top()
1
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4),Integer(5)],Integer(4):[],
↪Integer(5):[]})
>>> P.top() is None
True
>>> Q = Poset({Integer(0):[Integer(1)],Integer(1):[]})
>>> Q.top()
1
```

upper_covers_iterator(*element*)

Return the list of elements that cover `element`.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],
↪7:[]})
sage: list(H.upper_covers_iterator(0))
[1, 2, 3]
sage: list(H.upper_covers_iterator(7))
[]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.hasse_diagram import HasseDiagram
>>> H = HasseDiagram({Integer(0):[Integer(1),Integer(3),Integer(2)],
↪Integer(1):[Integer(4)],Integer(2):[Integer(4),Integer(5),Integer(6)],
↪Integer(3):[Integer(6)],Integer(4):[Integer(7)],Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)],Integer(7):[]})
>>> list(H.upper_covers_iterator(Integer(0)))
[1, 2, 3]
>>> list(H.upper_covers_iterator(Integer(7)))
[]
```

vertical_decomposition(*return_list=False*)

Return vertical decomposition of the lattice.

This is the backend function for vertical decomposition functions of lattices.

The property of being vertically decomposable is defined for lattices. This is *not* checked, and the function works with any bounded poset.

INPUT:

- `return_list` – boolean (default: `False`); if `False` (the default), return an element that is not the top neither the bottom element of the lattice, but is comparable to all elements of the lattice, if the lattice is vertically decomposable and `None` otherwise. If `True`, return list of decomposition elements.

EXAMPLES:

```
sage: H = posets.BooleanLattice(4)._hasse_diagram
sage: H.vertical_decomposition() is None
True
sage: P = Poset( ([1,2,3,6,12,18,36], attrcall("divides")) )
sage: P._hasse_diagram.vertical_decomposition()
3
sage: P._hasse_diagram.vertical_decomposition(return_list=True)
[3]
```

```
>>> from sage.all import *
>>> H = posets.BooleanLattice(Integer(4))._hasse_diagram
>>> H.vertical_decomposition() is None
True
>>> P = Poset( ([Integer(1),Integer(2),Integer(3),Integer(6),Integer(12),
↪Integer(18),Integer(36)], attrcall("divides")) )
>>> P._hasse_diagram.vertical_decomposition()
3
>>> P._hasse_diagram.vertical_decomposition(return_list=True)
[3]
```

exception sage.combinat.posets.hasse_diagram.**LatticeError**(*fail*, *x*, *y*)

Bases: `ValueError`

Helper exception class to forward elements without meet or join to upper level, so that the user will see “No meet for a and b” instead of “No meet for 1 and 2”.

5.1.181 Incidence Algebras

class sage.combinat.posets.incidence_algebras.**IncidenceAlgebra**(*R*, *P*, *prefix='I'*)

Bases: *CombinatorialFreeModule*

The incidence algebra of a poset.

Let P be a poset and R be a commutative unital associative ring. The *incidence algebra* I_P is the algebra of functions $\alpha\colon P \times P \to R$ such that $\alpha(x, y) = 0$ if $x \not\leq y$ where multiplication is given by convolution:

$$(\alpha * \beta)(x, y) = \sum_{x \leq k \leq y} \alpha(x, k)\beta(k, y).$$

This has a natural basis given by indicator functions for the interval $[a, b]$, i.e. $X_{a,b}(x, y) = \delta_{ax}\delta_{by}$. The incidence algebra is a unital algebra with the identity given by the Kronecker delta $\delta(x, y) = \delta_{xy}$. The Möbius function of P is another element of I_p whose inverse is the ζ function of the poset (so $\zeta(x, y) = 1$ for every interval $[x, y]$).

Todo

Implement the incidence coalgebra.

REFERENCES:

- Wikipedia article Incidence_algebra

class Element

Bases: `IndexedFreeModuleElement`

An element of an incidence algebra.

is_unit()

Return if `self` is a unit.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: mu = I.moebius()
sage: mu.is_unit()
True
sage: zeta = I.zeta()
sage: zeta.is_unit()
True
sage: x = mu - I.zeta() + I[2,2]
sage: x.is_unit()
False
sage: y = I.moebius() + I.zeta()
sage: y.is_unit()
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> I = P.incidence_algebra(QQ)
>>> mu = I.moebius()
>>> mu.is_unit()
True
>>> zeta = I.zeta()
>>> zeta.is_unit()
True
>>> x = mu - I.zeta() + I[Integer(2),Integer(2)]
>>> x.is_unit()
False
>>> y = I.moebius() + I.zeta()
>>> y.is_unit()
True
```

This depends on the base ring:

```
sage: I = P.incidence_algebra(ZZ)
sage: y = I.moebius() + I.zeta()
sage: y.is_unit()
False
```

```
>>> from sage.all import *
>>> I = P.incidence_algebra(ZZ)
>>> y = I.moebius() + I.zeta()
>>> y.is_unit()
False
```

to_matrix()

Return self as a matrix.

We define a matrix $M_{xy} = \alpha(x, y)$ for some element $\alpha \in I_P$ in the incidence algebra I_P and we order the elements $x, y \in P$ by some linear extension of P. This defines an algebra (iso)morphism; in particular, multiplication in the incidence algebra goes to matrix multiplication.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: I.moebius().to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
sage: I.zeta().to_matrix()
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> I = P.incidence_algebra(QQ)
>>> I.moebius().to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
>>> I.zeta().to_matrix()
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

delta()

Return the element 1 in self (which is the Kronecker delta $\delta(x, y)$).

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
 + I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
 + I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
 + I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
 + I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

moebius()

Return the Möbius function of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: I.moebius()
I[0, 0] - I[0, 1] - I[0, 2] + I[0, 3] + I[1, 1]
 - I[1, 3] + I[2, 2] - I[2, 3] + I[3, 3]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> I = P.incidence_algebra(QQ)
>>> I.moebius()
I[0, 0] - I[0, 1] - I[0, 2] + I[0, 3] + I[1, 1]
 - I[1, 3] + I[2, 2] - I[2, 3] + I[3, 3]
```

one()

Return the element 1 in `self` (which is the Kronecker delta $\delta(x, y)$).

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
 + I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
 + I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
 + I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
 + I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

poset()

Return the defining poset of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
```

(continues on next page)

(continued from previous page)

```
sage: I.poset()
Finite lattice containing 16 elements
sage: I.poset() == P
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.poset()
Finite lattice containing 16 elements
>>> I.poset() == P
True
```

product_on_basis(*A*, *B*)

Return the product of basis elements indexed by `A` and `B`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.product_on_basis((1, 3), (3, 11))
I[1, 11]
sage: I.product_on_basis((1, 3), (2, 2))
0
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.product_on_basis((Integer(1), Integer(3)), (Integer(3), Integer(11)))
I[1, 11]
>>> I.product_on_basis((Integer(1), Integer(3)), (Integer(2), Integer(2)))
0
```

reduced_subalgebra(*prefix='R'*)

Return the reduced incidence subalgebra.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.reduced_subalgebra()
Reduced incidence algebra of Finite lattice containing 16 elements
 over Rational Field
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.reduced_subalgebra()
Reduced incidence algebra of Finite lattice containing 16 elements
 over Rational Field
```

some_elements()

Return a list of elements of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(1)
sage: I = P.incidence_algebra(QQ)
sage: Ielts = I.some_elements(); Ielts # random
[2*I[0, 0] + 2*I[0, 1] + 3*I[1, 1],
 I[0, 0] - I[0, 1] + I[1, 1],
 I[0, 0] + I[0, 1] + I[1, 1]]
sage: [a in I for a in Ielts]
[True, True, True]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(1))
>>> I = P.incidence_algebra(QQ)
>>> Ielts = I.some_elements(); Ielts # random
[2*I[0, 0] + 2*I[0, 1] + 3*I[1, 1],
 I[0, 0] - I[0, 1] + I[1, 1],
 I[0, 0] + I[0, 1] + I[1, 1]]
>>> [a in I for a in Ielts]
[True, True, True]
```

zeta()

Return the ζ function in `self`.

The ζ function on a poset P is given by

$$\zeta(x, y) = \begin{cases} 1 & x \leq y, \\ 0 & x \not\leq y. \end{cases}$$

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.zeta() * I.moebius() == I.one()
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.zeta() * I.moebius() == I.one()
True
```

class sage.combinat.posets.incidence_algebras.**ReducedIncidenceAlgebra**(*I*, *prefix='R'*)

Bases: *CombinatorialFreeModule*

The reduced incidence algebra of a poset.

The reduced incidence algebra R_P is a subalgebra of the incidence algebra I_P where $\alpha(x, y) = \alpha(x', y')$ when $[x, y]$ is isomorphic to $[x', y']$ as posets. Thus the delta, Möbius, and zeta functions are all elements of R_P.

class Element

Bases: `IndexedFreeModuleElement`

An element of a reduced incidence algebra.

is_unit()

Return if `self` is a unit.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: x = R.an_element()
sage: x.is_unit()
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> x = R.an_element()
>>> x.is_unit()
True
```

lift()

Return the lift of `self` to the ambient space.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: R = I.reduced_subalgebra()
sage: x = R.an_element(); x
2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
sage: x.lift()
2*I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + 2*I[1, 1]
 + 2*I[1, 3] + 2*I[2, 2] + 2*I[2, 3] + 2*I[3, 3]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> I = P.incidence_algebra(QQ)
>>> R = I.reduced_subalgebra()
>>> x = R.an_element(); x
2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
>>> x.lift()
2*I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + 2*I[1, 1]
 + 2*I[1, 3] + 2*I[2, 2] + 2*I[2, 3] + 2*I[3, 3]
```

to_matrix()

Return `self` as a matrix.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: mu = R.moebius()
sage: mu.to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> mu = R.moebius()
>>> mu.to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
```

delta()

Return the Kronecker delta function in `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.delta()
R[(0, 0)]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.delta()
R[(0, 0)]
```

lift()

Return the lift morphism from `self` to the ambient space.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.lift
Generic morphism:
  From: Reduced incidence algebra of Finite lattice containing 4 elements␣
↪over Rational Field
  To:   Incidence algebra of Finite lattice containing 4 elements over␣
↪Rational Field
sage: R.an_element() - R.one()
R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
sage: R.lift(R.an_element() - R.one())
I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + I[1, 1]
 + 2*I[1, 3] + I[2, 2] + 2*I[2, 3] + I[3, 3]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.lift
Generic morphism:
  From: Reduced incidence algebra of Finite lattice containing 4 elements␣
↪over Rational Field
  To:   Incidence algebra of Finite lattice containing 4 elements over␣
```

(continues on next page)

(continued from previous page)

```
↪Rational Field
>>> R.an_element() - R.one()
R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
>>> R.lift(R.an_element() - R.one())
I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + I[1, 1]
 + 2*I[1, 3] + I[2, 2] + 2*I[2, 3] + I[3, 3]
```

moebius()

Return the Möbius function of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.moebius()
R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.moebius()
R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)]
```

one_basis()

Return the index of the element 1 in `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.one_basis()
(0, 0)
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.one_basis()
(0, 0)
```

poset()

Return the defining poset of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.poset()
Finite lattice containing 16 elements
sage: R.poset() == P
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.poset()
Finite lattice containing 16 elements
>>> R.poset() == P
True
```

some_elements()

Return a list of elements of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.some_elements()
[2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)],
 R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)],
 R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.some_elements()
[2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)],
 R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)],
 R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]]
```

zeta()

Return the ζ function in `self`.

The ζ function on a poset P is given by

$$\zeta(x, y) = \begin{cases} 1 & x \leq y, \\ 0 & x \not\leq y. \end{cases}$$

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.zeta()
R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.zeta()
R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]
```

5.1.182 Finite lattices and semilattices

This module implements finite (semi)lattices. It defines:

LatticePoset()	Construct a lattice.
MeetSemilattice()	Construct a meet semi-lattice.
JoinSemilattice()	Construct a join semi-lattice.
FiniteLatticePoset	A class for finite lattices.
FiniteMeetSemilattice	A class for finite meet semilattices.
FiniteJoinSemilattice	A class for finite join semilattices.

List of (semi)lattice methods

Meet and join

meet()	Return the meet of given elements.
join()	Return the join of given elements.
meet_matrix()	Return the matrix of meets of all elements of the meet semi-lattice.
join_matrix()	Return the matrix of joins of all elements of the join semi-lattice.

Properties of the lattice

is_distributive()	Return `True` if the lattice is distributive.
is_modular()	Return `True` if the lattice is modular.
is_lower_semimodular()	Return `True` if all elements with common upper cover have a common lower cover.
is_upper_semimodular()	Return `True` if all elements with common lower cover have a common upper cover.
is_semidistributive()	Return `True` if the lattice is both join- and meet-semidistributive.
is_join_semidistributive()	Return `True` if the lattice is join-semidistributive.
is_meet_semidistributive()	Return `True` if the lattice is meet-semidistributive.
is_join_distributive()	Return `True` if the lattice is join-distributive.
is_meet_distributive()	Return `True` if the lattice is meet-distributive.
is_atomic()	Return `True` if every element of the lattice can be written as a join of atoms.
is_coatomic()	Return `True` if every element of the lattice can be written as a meet of coatoms.
is_geometric()	Return `True` if the lattice is atomic and upper semimodular.
is_extremal()	Return `True` if the lattice is extremal.
is_complemented()	Return `True` if every element of the lattice has at least one complement.
is_sectionally_complemented()	Return `True` if every interval from the bottom is complemented.
is_cosectionally_complemented()	Return `True` if every interval to the top is complemented.
is_relatively_complemented()	Return `True` if every interval of the lattice is complemented.
is_pseudocomplemented()	Return `True` if every element of the lattice has a (meet-)pseudocomplement.
is_join_pseudocomplemented()	Return `True` if every element of the lattice has a join-pseudocomplement.
is_orthocomplemented()	Return `True` if the lattice has an orthocomplementation.
is_supersolvable()	Return `True` if the lattice is supersolvable.
is_planar()	Return `True` if the lattice has an upward planar drawing.
is_dismantlable()	Return `True` if the lattice is dismantlable.

continues on next page

Table 5 – continued from previous page

is_interval_dismantlable()	Return `True` if the lattice is interval dismantlable.
is_sublattice_dismantlable()	Return `True` if the lattice is sublattice dismantlable.
is_stone()	Return `True` if the lattice is a Stone lattice.
is_trim()	Return `True` if the lattice is a trim lattice.
is_vertically_decomposable()	Return `True` if the lattice is vertically decomposable.
is_simple()	Return `True` if the lattice has no nontrivial congruences.
is_isoform()	Return `True` if all congruences of the lattice consists of isoform blocks.
is_uniform()	Return `True` if all congruences of the lattice consists of equal-sized blocks.
is_regular()	Return `True` if all congruences of lattice are determined by any of the congruence blocks.
is_subdirectly_reducible()	Return `True` if the lattice is a sublattice of the product of smaller lattices.
is_constructible_by_doublings()	Return `True` if the lattice is constructible by doublings from the one-element lattice.
breadth()	Return the breadth of the lattice.

Specific elements

atoms()	Return elements covering the bottom element.
coatoms()	Return elements covered by the top element.
double_irreducibles()	Return double irreducible elements.
join_primes()	Return the join prime elements.
meet_primes()	Return the meet prime elements.
complements()	Return the list of complements of an element, or the dictionary of complements for all elements.
pseudocomplement()	Return the pseudocomplement of an element.
is_modular_element()	Return `True` if given element is modular in the lattice.
is_left_modular_element()	Return `True` if given element is left modular in the lattice.
neutral_elements()	Return neutral elements of the lattice.
canonical_joinands()	Return the canonical joinands of an element.
canonical_meetands()	Return the canonical meetands of an element.

Sublattices

sublattice()	Return sublattice generated by list of elements.
submeetsemilattice()	Return meet-subsemilattice generated by list of elements.
subjoinsemilattice()	Return join-subsemilattice generated by list of elements.
is_sublattice()	Return `True` if the lattice is a sublattice of given lattice.
sublattices()	Return all sublattices of the lattice.
sublattices_lattice()	Return the lattice of sublattices.
isomorphic_sublattices_iterator()	Return an iterator over the sublattices isomorphic to given lattice.
maximal_sublattices()	Return maximal sublattices of the lattice.
frattini_sublattice()	Return the intersection of maximal sublattices of the lattice.
skeleton()	Return the skeleton of the lattice.
center()	Return the sublattice of complemented neutral elements.
vertical_decomposition()	Return the vertical decomposition of the lattice.

Miscellaneous

moebius_algebra()	Return the Möbius algebra of the lattice.
quantum_moebius_algebra()	Return the quantum Möbius algebra of the lattice.
vertical_composition()	Return ordinal sum of lattices with top/bottom element unified.
day_doubling()	Return the lattice with Alan Day's doubling construction of a subset.
adjunct()	Return the adjunct with other lattice.
subdirect_decomposition()	Return the subdirect decomposition of the lattice.
congruence()	Return the congruence generated by lists of elements.
quotient()	Return the quotient lattice by a congruence.
congruences_lattice()	Return the lattice of congruences.

class sage.combinat.posets.lattices.**FiniteJoinSemilattice**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: *FinitePoset*

We assume that the argument passed to FiniteJoinSemilattice is the poset of a join-semilattice (i.e. a poset with least upper bound for each pair of elements).

Element

alias of *JoinSemilatticeElement*

coatoms()

Return the list of co-atoms of this (semi)lattice.

A *co-atom* of a lattice is an element covered by the top element.

EXAMPLES:

```
sage: L = posets.DivisorLattice(60)
sage: sorted(L.coatoms())
[12, 20, 30]
```

```
>>> from sage.all import *
>>> L = posets.DivisorLattice(Integer(60))
```

(continues on next page)

(continued from previous page)

```
>>> sorted(L.coatoms())
[12, 20, 30]
```

See also

- Dual function: `atoms()`

join(*x*, *y=None*)

Return the join of given elements in the lattice.

INPUT:

- `x`, `y` – two elements of the (semi)lattice OR
- `x` – list or tuple of elements

EXAMPLES:

```
sage: D = posets.DiamondPoset(5)
sage: D.join(1, 2)
4
sage: D.join(1, 1)
1
sage: D.join(1, 4)
4
sage: D.join(1, 0)
1
```

```
>>> from sage.all import *
>>> D = posets.DiamondPoset(Integer(5))
>>> D.join(Integer(1), Integer(2))
4
>>> D.join(Integer(1), Integer(1))
1
>>> D.join(Integer(1), Integer(4))
4
>>> D.join(Integer(1), Integer(0))
1
```

Using list of elements as an argument. Join of empty list is the bottom element:

```
sage: B4=posets.BooleanLattice(4)
sage: B4.join([2,4,8])
14
sage: B4.join([])
0
```

```
>>> from sage.all import *
>>> B4=posets.BooleanLattice(Integer(4))
>>> B4.join([Integer(2),Integer(4),Integer(8)])
14
```

(continues on next page)

(continued from previous page)

```
>>> B4.join([])
0
```

For non-facade lattices operator + works for join:

```
sage: L = posets.PentagonPoset(facade=False)
sage: L(1)+L(2)
4
```

```
>>> from sage.all import *
>>> L = posets.PentagonPoset(facade=False)
>>> L(Integer(1))+L(Integer(2))
4
```

See also

- Dual function: *meet()*

join_matrix()

Return a matrix whose `(i,j)` entry is `k`, where `self.linear_extension()[k]` is the join (least upper bound) of `self.linear_extension()[i]` and `self.linear_extension()[j]`.

EXAMPLES:

```
sage: P = LatticePoset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]], facade =
↪False)
sage: J = P.join_matrix(); J
[0 1 2 3 4 5 6 7]
[1 1 3 3 7 7 7 7]
[2 3 2 3 4 6 6 7]
[3 3 3 3 7 7 7 7]
[4 7 4 7 4 7 7 7]
[5 7 6 7 7 5 6 7]
[6 7 6 7 7 6 6 7]
[7 7 7 7 7 7 7 7]
sage: J[P(4).vertex,P(3).vertex] == P(7).vertex
True
sage: J[P(5).vertex,P(2).vertex] == P(5).vertex
True
sage: J[P(5).vertex,P(2).vertex] == P(2).vertex
False
```

```
>>> from sage.all import *
>>> P = LatticePoset([[Integer(1),Integer(3),Integer(2)],[Integer(4)],
↪[Integer(4),Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],
↪[Integer(7)],[]], facade = False)
>>> J = P.join_matrix(); J
[0 1 2 3 4 5 6 7]
[1 1 3 3 7 7 7 7]
[2 3 2 3 4 6 6 7]
```

(continues on next page)

(continued from previous page)

```
[3 3 3 3 7 7 7 7]
[4 7 4 7 4 7 7 7]
[5 7 6 7 7 5 6 7]
[6 7 6 7 7 6 6 7]
[7 7 7 7 7 7 7 7]
>>> J[P(Integer(4)).vertex,P(Integer(3)).vertex] == P(Integer(7)).vertex
True
>>> J[P(Integer(5)).vertex,P(Integer(2)).vertex] == P(Integer(5)).vertex
True
>>> J[P(Integer(5)).vertex,P(Integer(2)).vertex] == P(Integer(2)).vertex
False
```

class sage.combinat.posets.lattices.**FiniteLatticePoset**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: *FiniteMeetSemilattice*, *FiniteJoinSemilattice*

We assume that the argument passed to FiniteLatticePoset is the poset of a lattice (i.e. a poset with greatest lower bound and least upper bound for each pair of elements).

Element

alias of *LatticePosetElement*

adjunct(*other*, *a*, *b*)

Return the adjunct of the lattice by `other` on the pair (a, b).

It is assumed that $a < b$ but b does not cover a.

The adjunct of a lattice K to L with respect to pair (a, b) of L is defined such that $x < y$ if

- $x, y \in K$ and $x < y$ in K,
- $x, y \in L$ and $x < y$ in L,
- $x \in L$, $y \in K$ and $x \leq a$ in L, or
- $x \in K$, $y \in L$ and $b \leq y$ in L.

Informally this can be seen as attaching the lattice K to L as a new block between a and b. Dismantlable lattices are exactly those that can be created from chains with this function.

Mathematically, it is only defined when L and K have no common element; here we force that by giving them different names in the resulting lattice.

EXAMPLES:

```
sage: Pnum = posets.PentagonPoset()
sage: Palp = Pnum.relabel(lambda x: chr(ord('a')+x))
sage: PP = Pnum.adjunct(Palp, 0, 3)
sage: PP.atoms()
[(0, 1), (0, 2), (1, 'a')]
sage: PP.coatoms()
[(0, 1), (0, 3)]
```

```
>>> from sage.all import *
>>> Pnum = posets.PentagonPoset()
>>> Palp = Pnum.relabel(lambda x: chr(ord('a')+x))
>>> PP = Pnum.adjunct(Palp, Integer(0), Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> PP.atoms()
[(0, 1), (0, 2), (1, 'a')]
>>> PP.coatoms()
[(0, 1), (0, 3)]
```

breadth(*certificate=False*)

Return the breadth of the lattice.

The breadth of a lattice is the largest integer n such that any join of elements $x_1, x_2, \ldots, x_{n+1}$ is join of a proper subset of x_i.

This can be also characterized by subposets: a lattice of breadth at least n contains a subposet isomorphic to the Boolean lattice of 2^n elements.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return the pair (b, a) where b is the breadth and a is an antichain such that the join of a differs from the join of any proper subset of a. If `certificate=False` return just the breadth.

EXAMPLES:

```
sage: D10 = posets.DiamondPoset(10)
sage: D10.breadth()
2

sage: B3 = posets.BooleanLattice(3)
sage: B3.breadth()
3
sage: B3.breadth(certificate=True)
(3, [1, 2, 4])
```

```
>>> from sage.all import *
>>> D10 = posets.DiamondPoset(Integer(10))
>>> D10.breadth()
2

>>> B3 = posets.BooleanLattice(Integer(3))
>>> B3.breadth()
3
>>> B3.breadth(certificate=True)
(3, [1, 2, 4])
```

ALGORITHM:

For a lattice to have breadth at least n, it must have an n-element antichain A with join j. Element j must cover at least n elements. There must also be $n-2$ levels of elements between A and j. So we start by searching elements that could be our j and then just check possible antichains A.

Note

Prior to version 8.1 this function returned just an antichain with `certificate=True`.

canonical_joinands(*e*)

Return the canonical joinands of e.

The canonical joinands of an element e in the lattice L is the subset $S \subseteq L$ such that 1) the join of S is e, and 2) if the join of some other subset S' of is also e, then for every element $s \in S$ there is an element $s' \in S'$ such that $s \leq s'$.

Informally said this is the set of lowest possible elements with given join. It exists for every element if and only if the lattice is join-semidistributive. Canonical joinands are always join-irreducibles.

INPUT:

- `e` – an element of the lattice

OUTPUT: canonical joinands as a list, if it exists; if not, `None`

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3], 2: [4, 5], 3: [5], 4: [6],
....:                   5: [7], 6: [7]})
sage: L.canonical_joinands(7)
[3, 4]

sage: L = LatticePoset({1: [2, 3], 2: [4, 5], 3: [6], 4: [6],
....: 5: [6]})
sage: L.canonical_joinands(6) is None
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
→[Integer(4), Integer(5)], Integer(3): [Integer(5)], Integer(4):␣
→[Integer(6)],
...                   Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> L.canonical_joinands(Integer(7))
[3, 4]

>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
→[Integer(4), Integer(5)], Integer(3): [Integer(6)], Integer(4):␣
→[Integer(6)],
... Integer(5): [Integer(6)]})
>>> L.canonical_joinands(Integer(6)) is None
True
```

See also

canonical_meetands()

canonical_meetands(*e*)

Return the canonical meetands of e.

The canonical meetands of an element e in the lattice L is the subset $S \subseteq L$ such that 1) the meet of S is e, and 2) if the meet of some other subset S' of is also e, then for every element $s \in S$ there is an element $s' \in S'$ such that $s \geq s'$.

Informally said this is the set of greatest possible elements with given meet. It exists for every element if and only if the lattice is meet-semidistributive. Canonical meetands are always meet-irreducibles.

INPUT:

- `e` – an element of the lattice

OUTPUT: canonical meetands as a list, if it exists; if not, `None`

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3], 2: [4], 3: [5, 6], 4: [6],
....:                   5: [7], 6: [7]})
sage: L.canonical_meetands(1)
[5, 4]

sage: L = LatticePoset({1: [2, 3], 2: [4, 5], 3: [6], 4: [6],
....: 5: [6]})
sage: L.canonical_meetands(1) is None
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
→[Integer(4)], Integer(3): [Integer(5), Integer(6)], Integer(4):␣
→[Integer(6)],
...                   Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> L.canonical_meetands(Integer(1))
[5, 4]

>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
→[Integer(4), Integer(5)], Integer(3): [Integer(6)], Integer(4):␣
→[Integer(6)],
... Integer(5): [Integer(6)]})
>>> L.canonical_meetands(Integer(1)) is None
True
```

See also

canonical_joinands()

center()

Return the center of the lattice.

An element of a lattice is *central* if it is neutral and has a complement. The subposet induced by central elements is a *center* of the lattice. Actually it is a Boolean lattice.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [6, 7], 3: [8, 9, 7],
....:                   4: [5, 6], 5: [8, 10], 6: [10], 7: [13, 11],
....:                   8: [13, 12], 9: [11, 12], 10: [13],
....:                   11: [14], 12: [14], 13: [14]})
sage: C = L.center(); C
Finite lattice containing 4 elements
sage: C.cover_relations()
```

(continues on next page)

(continued from previous page)

```
[[1, 2], [1, 12], [2, 14], [12, 14]]

sage: L = posets.DivisorLattice(60)
sage: sorted(L.center().list())
[1, 3, 4, 5, 12, 15, 20, 60]
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],
↪Integer(2): [Integer(6), Integer(7)], Integer(3): [Integer(8), Integer(9),
↪Integer(7)],
...                     Integer(4): [Integer(5), Integer(6)], Integer(5):
↪[Integer(8), Integer(10)], Integer(6): [Integer(10)], Integer(7):
↪[Integer(13), Integer(11)],
...                     Integer(8): [Integer(13), Integer(12)], Integer(9):
↪[Integer(11), Integer(12)], Integer(10): [Integer(13)],
...                     Integer(11): [Integer(14)], Integer(12): [Integer(14)],
↪Integer(13): [Integer(14)]})
>>> C = L.center(); C
Finite lattice containing 4 elements
>>> C.cover_relations()
[[1, 2], [1, 12], [2, 14], [12, 14]]

>>> L = posets.DivisorLattice(Integer(60))
>>> sorted(L.center().list())
[1, 3, 4, 5, 12, 15, 20, 60]
```

See also

neutral_elements(), *complements()*

complements(*element=None*)

Return the list of complements of an element in the lattice, or the dictionary of complements for all elements.

Elements x and y are complements if their meet and join are respectively the bottom and the top element of the lattice.

INPUT:

- `element` – an element of the lattice whose complement is returned. If `None` (default) then dictionary of complements for all elements having at least one complement is returned.

EXAMPLES:

```
sage: L = LatticePoset({0:['a','b','c'],'a':[1],'b':[1],'c':[1]})
sage: C = L.complements()
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(0):['a','b','c'],'a':[Integer(1)],'b
↪':[Integer(1)],'c':[Integer(1)]})
>>> C = L.complements()
```

Let us check that 'a' and 'b' are complements of each other:

```
sage: 'a' in C['b']
True
sage: 'b' in C['a']
True
```

```
>>> from sage.all import *
>>> 'a' in C['b']
True
>>> 'b' in C['a']
True
```

Full list of complements:

```
sage: L.complements() # random order
{0: [1], 1: [0], 'a': ['b', 'c'], 'b': ['c', 'a'], 'c': ['b', 'a']}

sage: L = LatticePoset({0:[1,2],1:[3],2:[3],3:[4]})
sage: L.complements() # random order
{0: [4], 4: [0]}
sage: L.complements(1)
[]
```

```
>>> from sage.all import *
>>> L.complements() # random order
{0: [1], 1: [0], 'a': ['b', 'c'], 'b': ['c', 'a'], 'c': ['b', 'a']}

>>> L = LatticePoset({Integer(0):[Integer(1),Integer(2)],
↪Integer(1):[Integer(3)],Integer(2):[Integer(3)],Integer(3):[Integer(4)]})
>>> L.complements() # random order
{0: [4], 4: [0]}
>>> L.complements(Integer(1))
[]
```

See also:

is_complemented()

congruence(S)

Return the congruence generated by set of sets S.

A congruence of a lattice is an equivalence relation $\cong$ that is compatible with meet and join; i.e. if $a_1 \cong a_2$ and $b_1 \cong b_2$, then $(a_1$
$veeb_1) \cong (a_2$
$veeb_2)$ and $(a_1 \wedge b_1) \cong (a_2 \wedge b_2)$.

By the congruence generated by set of sets $\{S_1, \ldots, S_n\}$ we mean the least congruence $\cong$ such that for every $x, y \in S_i$ for some i we have $x \cong y$.

INPUT:

- `S` – list of lists; list of element blocks that the congruence will contain

OUTPUT:

Congruence of the lattice as a *sage.combinat.set_partition.SetPartition*.

EXAMPLES:

```
sage: # needs sage.combinat
sage: L = posets.DivisorLattice(12)
sage: cong = L.congruence([[1, 3]])
sage: sorted(sorted(c) for c in cong)
[[1, 3], [2, 6], [4, 12]]
sage: L.congruence([[1, 2], [6, 12]])
{{1, 2, 4}, {3, 6, 12}}

sage: L = LatticePoset({1: [2, 3], 2: [4], 3: [4], 4: [5]})
sage: L.congruence([[1, 2]])                                              #␣
↪needs sage.combinat
{{1, 2}, {3, 4}, {5}}

sage: L = LatticePoset({1: [2, 3], 2: [4, 5, 6], 4: [5], 5: [7, 8],
....:                   6: [8], 3: [9], 7: [10], 8: [10], 9:[10]})
sage: cong = L.congruence([[1, 2]])                                       #␣
↪needs sage.combinat
sage: cong[0]                                                             #␣
↪needs sage.combinat
frozenset({1, 2, 3, 4, 5, 6, 7, 8, 9, 10})
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> L = posets.DivisorLattice(Integer(12))
>>> cong = L.congruence([[Integer(1), Integer(3)]])
>>> sorted(sorted(c) for c in cong)
[[1, 3], [2, 6], [4, 12]]
>>> L.congruence([[Integer(1), Integer(2)], [Integer(6), Integer(12)]])
{{1, 2, 4}, {3, 6, 12}}

>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4)], Integer(3): [Integer(4)], Integer(4): [Integer(5)]})
>>> L.congruence([[Integer(1), Integer(2)]])                              ␣
↪               # needs sage.combinat
{{1, 2}, {3, 4}, {5}}

>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4), Integer(5), Integer(6)], Integer(4): [Integer(5)], Integer(5):␣
↪[Integer(7), Integer(8)],
...                   Integer(6): [Integer(8)], Integer(3): [Integer(9)],␣
↪Integer(7): [Integer(10)], Integer(8): [Integer(10)],␣
↪Integer(9):[Integer(10)]})
>>> cong = L.congruence([[Integer(1), Integer(2)]])                       ␣
↪               # needs sage.combinat
>>> cong[Integer(0)]                                                      ␣
↪      # needs sage.combinat
frozenset({1, 2, 3, 4, 5, 6, 7, 8, 9, 10})
```

See also

```
quotient()
```

congruences_lattice(*labels='congruence'*)

Return the lattice of congruences.

A congruence of a lattice is a partition of elements to classes compatible with both meet- and join-operation; see `congruence()`. Elements of the *congruence lattice* are congruences ordered by refinement; i.e. if every class of a congruence Θ is contained in some class of Φ, then $\Theta \leq \Phi$ in the congruence lattice.

INPUT:

- `labels` – string; the type of elements in the resulting lattice

OUTPUT: a distributive lattice

- If `labels='congruence'`, then elements of the result will be congruences given as `sage.combinat.set_partition.SetPartition`.
- If `labels='integers'`, result is a lattice on integers isomorphic to the congruence lattice.

EXAMPLES:

```
sage: # needs sage.combinat
sage: N5 = posets.PentagonPoset()
sage: CL = N5.congruences_lattice(); CL
Finite lattice containing 5 elements
sage: CL.atoms()
[{{0}, {1}, {2, 3}, {4}}]
sage: CL.coatoms()
[{{0, 1}, {2, 3, 4}}, {{0, 2, 3}, {1, 4}}]

sage: C4 = posets.ChainPoset(4)
sage: CL = C4.congruences_lattice(labels='integer')                    #␣
↪needs sage.combinat
sage: CL.is_isomorphic(posets.BooleanLattice(3))                       #␣
↪needs sage.combinat
True
```

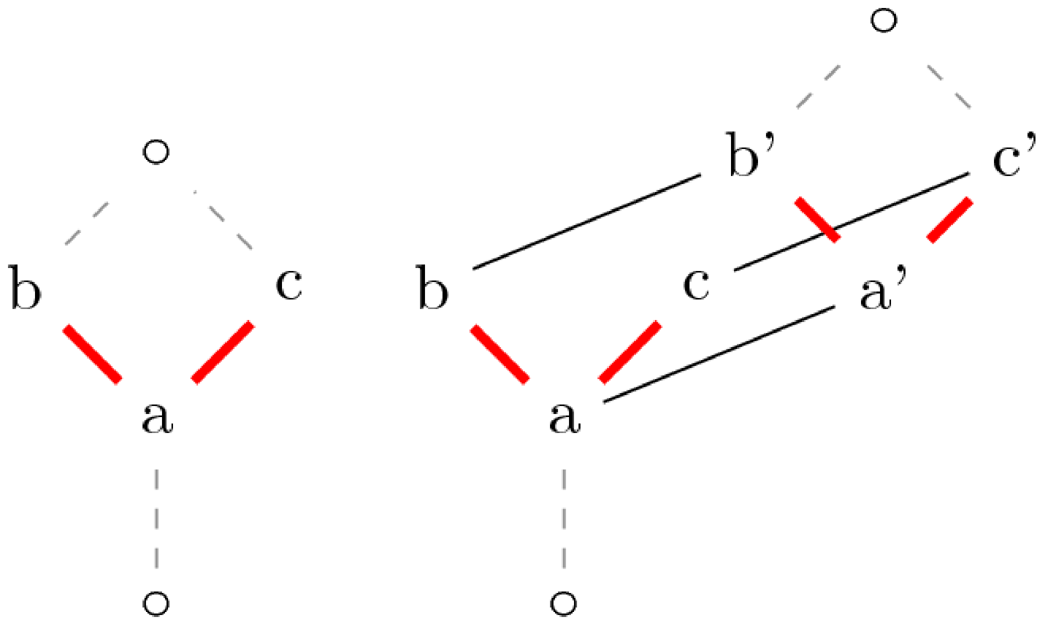
```
>>> from sage.all import *
>>> # needs sage.combinat
>>> N5 = posets.PentagonPoset()
>>> CL = N5.congruences_lattice(); CL
Finite lattice containing 5 elements
>>> CL.atoms()
[{{0}, {1}, {2, 3}, {4}}]
>>> CL.coatoms()
[{{0, 1}, {2, 3, 4}}, {{0, 2, 3}, {1, 4}}]

>>> C4 = posets.ChainPoset(Integer(4))
>>> CL = C4.congruences_lattice(labels='integer')                    #␣
↪needs sage.combinat
>>> CL.is_isomorphic(posets.BooleanLattice(Integer(3)))                ␣
↪     # needs sage.combinat
True
```

day_doubling(S)

Return the lattice with Alan Day's doubling construction of subset S.

The subset S is assumed to be convex (i.e. if $a, c \in S$ and $a < b < c$ in the lattice, then $b \in S$) and connected (i.e. if $a, b \in S$ then there is a chain $a = e_1, e_2, \ldots, e_n = b$ such that e_i either covers or is covered by e_{i+1}).



Alan Day's doubling construction is a specific extension of the lattice. Here we formulate it in a format more suitable for computation.

Let L be a lattice and S a convex subset of it. The resulting lattice $L[S]$ has elements $(e, 0)$ for each $e \in L$ and $(e, 1)$ for each $e \in S$. If $x \leq y$ in L, then in the new lattice we have

- $(x, 0), (x, 1) \leq (y, 0), (y, 1)$
- $(x, 0) \leq (x, 1)$

INPUT:

- `S` – a subset of the lattice

EXAMPLES:

```
sage: L = LatticePoset({1: ['a', 'b', 2], 'a': ['c'], 'b': ['c', 'd'],
....:                   2: [3], 'c': [4], 'd': [4], 3: [4]})
sage: L2 = L.day_doubling(['a', 'b', 'c', 'd']); L2
Finite lattice containing 12 elements
sage: set(L2.upper_covers((1, 0))) == set([(2, 0), ('a', 0), ('b', 0)])
True
sage: set(L2.upper_covers(('b', 0))) == set([('d', 0), ('b', 1), ('c', 0)])
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): ['a', 'b', Integer(2)], 'a': ['c'], 'b': ['c
↪', 'd'],
```

(continues on next page)

(continued from previous page)

```
...                    Integer(2): [Integer(3)], 'c': [Integer(4)], 'd':␣
↪[Integer(4)], Integer(3): [Integer(4)]})
>>> L2 = L.day_doubling(['a', 'b', 'c', 'd']); L2
Finite lattice containing 12 elements
>>> set(L2.upper_covers((Integer(1), Integer(0)))) == set([(Integer(2),␣
↪Integer(0)), ('a', Integer(0)), ('b', Integer(0))])
True
>>> set(L2.upper_covers(('b', Integer(0)))) == set([('d', Integer(0)), ('b',␣
↪Integer(1)), ('c', Integer(0))])
True
```

See also:

is_constructible_by_doublings()

double_irreducibles()

Return the list of double irreducible elements of this lattice.

A *double irreducible* element of a lattice is an element covering and covered by exactly one element. In other words it is neither a meet nor a join of any elements.

EXAMPLES:

```
sage: L = posets.DivisorLattice(12)
sage: sorted(L.double_irreducibles())
[3, 4]

sage: L = posets.BooleanLattice(3)
sage: L.double_irreducibles()
[]
```

```
>>> from sage.all import *
>>> L = posets.DivisorLattice(Integer(12))
>>> sorted(L.double_irreducibles())
[3, 4]

>>> L = posets.BooleanLattice(Integer(3))
>>> L.double_irreducibles()
[]
```

See also:

meet_irreducibles(), join_irreducibles()

feichtner_yuzvinsky_ring(*G*, *use_defining=False*, *base_ring=None*)

Return the Feichtner-Yuzvinsky ring of `self` and `G`.

Let R be a commutative ring, L a lattice, and $G \subseteq L$. The *Feichtner-Yuzvinsky ring* is the quotient of the polynomial ring $R[h_g \mid g \in G]$ by the ideal generated by

- h_a for every atom $a \in L \cap G$ and

- for every antichain A of the subposet G such that $g := \bigvee A \in G$ (with the join taken in L)

$$\prod_{a \in A} (h_g - h_a).$$

This was originally described for G such that (L, G) is a built lattice in the sense of [FY2004] (which has a geometric motivation), but this has been extended to G being an arbitrary subset.

This is not the original definition, which uses the nested subsets of G (see [FY2004] for the definition). However, the original construction, which we call the *defining* presentation and use the variables $\{x_g \mid g \in G\}$, can be recovered by setting $h_g = \sum_{g' \geq g} x_{g'}$ (where $g' \in G$).

INPUT:

- `G` – a subset of elements of `self`
- `use_defining` – boolean (default: `False`); whether or not to use the defining presentation in x_g
- `base_ring` – (default: $\mathbf{Q}$) the base ring

The order on the variables is equal to the ordering of the elements in `G`.

EXAMPLES:

```
sage: B2 = posets.BooleanLattice(2)
sage: FY = B2.feichtner_yuzvinsky_ring(B2[1:])
sage: FY
Quotient of Multivariate Polynomial Ring in h0, h1, h2 over Rational Field
 by the ideal (h0, h1, h0*h1 - h0*h2 - h1*h2 + h2^2)

sage: FY = B2.feichtner_yuzvinsky_ring(B2[1:], use_defining=True)
sage: FY
Quotient of Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
 by the ideal (x0 + x2, x1 + x2, x0*x1)
```

```
>>> from sage.all import *
>>> B2 = posets.BooleanLattice(Integer(2))
>>> FY = B2.feichtner_yuzvinsky_ring(B2[Integer(1):])
>>> FY
Quotient of Multivariate Polynomial Ring in h0, h1, h2 over Rational Field
 by the ideal (h0, h1, h0*h1 - h0*h2 - h1*h2 + h2^2)

>>> FY = B2.feichtner_yuzvinsky_ring(B2[Integer(1):], use_defining=True)
>>> FY
Quotient of Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
 by the ideal (x0 + x2, x1 + x2, x0*x1)
```

We reproduce the example from Section 5 of [Coron2023]:

```
sage: # needs sage.geometry.polyhedron
sage: H.<a,b,c,d> = HyperplaneArrangements(QQ)
sage: Arr = H(a-b, b-c, c-d, d-a)
sage: P = LatticePoset(Arr.intersection_poset())
sage: FY = P.feichtner_yuzvinsky_ring([P.top(),5,1,2,3,4])
sage: FY.defining_ideal().groebner_basis()                                    #␣
↪needs sage.libs.singular
[h0^2 - h0*h1, h1^2, h2, h3, h4, h5]
```

```
>>> from sage.all import *
>>> # needs sage.geometry.polyhedron
>>> H = HyperplaneArrangements(QQ, names=('a', 'b', 'c', 'd',)); (a, b, c, d,
↪) = H._first_ngens(4)
>>> Arr = H(a-b, b-c, c-d, d-a)
>>> P = LatticePoset(Arr.intersection_poset())
>>> FY = P.feichtner_yuzvinsky_ring([P.top(),Integer(5),Integer(1),Integer(2),
↪Integer(3),Integer(4)])
>>> FY.defining_ideal().groebner_basis()                              #␣
↪needs sage.libs.singular
[h0^2 - h0*h1, h1^2, h2, h3, h4, h5]
```

frattini_sublattice()

Return the Frattini sublattice of the lattice.

The Frattini sublattice $\Phi(L)$ is the intersection of all proper maximal sublattices of L. It is also the set of "non-generators" - if the sublattice generated by set S of elements is whole lattice, then also $S \setminus \Phi(L)$ generates whole lattice.

EXAMPLES:

```
sage: L = LatticePoset(( [], [[1,2],[1,17],[1,8],[2,3],[2,22],
....:                          [2,5],[2,7],[17,22],[17,13],[8,7],
....:                          [8,13],[3,16],[3,9],[22,16],[22,18],
....:                          [22,10],[5,18],[5,14],[7,9],[7,14],
....:                          [7,10],[13,10],[16,6],[16,19],[9,19],
....:                          [18,6],[18,33],[14,33],[10,19],
....:                          [10,33],[6,4],[19,4],[33,4]] ))
sage: sorted(L.frattini_sublattice().list())
[1, 2, 4, 10, 19, 22, 33]
```

```
>>> from sage.all import *
>>> L = LatticePoset(( [], [[Integer(1),Integer(2)],[Integer(1),Integer(17)],
↪[Integer(1),Integer(8)],[Integer(2),Integer(3)],[Integer(2),Integer(22)],
...                          [Integer(2),Integer(5)],[Integer(2),Integer(7)],
↪[Integer(17),Integer(22)],[Integer(17),Integer(13)],[Integer(8),Integer(7)],
...                          [Integer(8),Integer(13)],[Integer(3),Integer(16)],
↪[Integer(3),Integer(9)],[Integer(22),Integer(16)],[Integer(22),Integer(18)],
...                          [Integer(22),Integer(10)],[Integer(5),
↪Integer(18)],[Integer(5),Integer(14)],[Integer(7),Integer(9)],[Integer(7),
↪Integer(14)],
...                          [Integer(7),Integer(10)],[Integer(13),
↪Integer(10)],[Integer(16),Integer(6)],[Integer(16),Integer(19)],[Integer(9),
↪Integer(19)],
...                          [Integer(18),Integer(6)],[Integer(18),
↪Integer(33)],[Integer(14),Integer(33)],[Integer(10),Integer(19)],
...                          [Integer(10),Integer(33)],[Integer(6),Integer(4)],
↪[Integer(19),Integer(4)],[Integer(33),Integer(4)]] ))
>>> sorted(L.frattini_sublattice().list())
[1, 2, 4, 10, 19, 22, 33]
```

is_atomic(*certificate=False*)

Return `True` if the lattice is atomic, and `False` otherwise.

A lattice is atomic if every element can be written as a join of atoms.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, e)`, where e is a join-irreducible element that is not an atom. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5], 3:[5], 4:[6], 5:[6]})
sage: L.is_atomic()
True

sage: L = LatticePoset({0: [1, 2], 1: [3], 2: [3], 3:[4]})
sage: L.is_atomic()
False
sage: L.is_atomic(certificate=True)
(False, 4)
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5)], Integer(3):[Integer(5)], Integer(4):[Integer(6)],␣
↪Integer(5):[Integer(6)]})
>>> L.is_atomic()
True

>>> L = LatticePoset({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(3)], Integer(3):[Integer(4)]})
>>> L.is_atomic()
False
>>> L.is_atomic(certificate=True)
(False, 4)
```

Note: See [EnumComb1], Section 3.3 for a discussion of atomic lattices.

See also:

- Dual property: *is_coatomic()*
- Stronger properties: *is_sectionally_complemented()*
- Mutually exclusive properties: *is_vertically_decomposable()*

is_coatomic(*certificate=False*)

Return `True` if the lattice is coatomic, and `False` otherwise.

A lattice is coatomic if every element can be written as a meet of coatoms; i.e. if the dual of the lattice is atomic.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, e)`, where e is a meet-irreducible element that is not a coatom. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(3)
sage: L.is_coatomic()
True

sage: L = LatticePoset({1: [2], 2: [3, 4], 3: [5], 4:[5]})
sage: L.is_coatomic()
False
sage: L.is_coatomic(certificate=True)
(False, 1)
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(3))
>>> L.is_coatomic()
True

>>> L = LatticePoset({Integer(1): [Integer(2)], Integer(2): [Integer(3),␣
↪Integer(4)], Integer(3): [Integer(5)], Integer(4):[Integer(5)]})
>>> L.is_coatomic()
False
>>> L.is_coatomic(certificate=True)
(False, 1)
```

See also:

- Dual property: *is_atomic()*
- Stronger properties: *is_cosectionally_complemented()*
- Mutually exclusive properties: *is_vertically_decomposable()*

is_complemented(*certificate=False*)

Return `True` if the lattice is complemented, and `False` otherwise.

A lattice is complemented if every element has at least one complement.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, e)`, where `e` is an element without a complement. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({0: [1, 2, 3], 1: [4], 2: [4], 3: [4]})
sage: L.is_complemented()
```

(continues on next page)

(continued from previous page)

```
True

sage: L = LatticePoset({1: [2, 3, 4], 2: [5, 6], 3: [5], 4: [6],
....:                   5: [7], 6: [7]})
sage: L.is_complemented()
False
sage: L.is_complemented(certificate=True)
(False, 2)
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(0): [Integer(1), Integer(2), Integer(3)],␣
↪Integer(1): [Integer(4)], Integer(2): [Integer(4)], Integer(3):␣
↪[Integer(4)]})
>>> L.is_complemented()
True

>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5), Integer(6)], Integer(3): [Integer(5)], Integer(4):␣
↪[Integer(6)],
...                   Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> L.is_complemented()
False
>>> L.is_complemented(certificate=True)
(False, 2)
```

See also:

- Stronger properties: *is_sectionally_complemented()*, *is_cosectionally_complemented()*, *is_orthocomplemented()*
- Other: *complements()*

is_congruence_uniform()

Return whether `self` is congruence uniform.

This is equivalent to being constructible by doubling intervals.

See also:

is_constructible_by_doublings()

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.is_congruence_uniform()
True
sage: P = posets.DiamondPoset(5)
sage: P.is_congruence_uniform()
False
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.is_congruence_uniform()
True
>>> P = posets.DiamondPoset(Integer(5))
>>> P.is_congruence_uniform()
False
```

REFERENCES:

- [Day1979]

is_constructible_by_doublings(*type*)

Return `True` if the lattice is constructible by doublings, and `False` otherwise.

We call a lattice doubling constructible if it can be constructed from the one element lattice by a sequence of Alan Day's doubling constructions.

Lattices constructible by interval doubling are also called *bounded*. Lattices constructible by lower and upper pseudo-interval are called *lower bounded* and *upper bounded*. Lattices constructible by any convex set doubling are called *congruence normal*.

INPUT:

- `type` – string; can be one of the following:
  - `'interval'` – allow only doublings of an interval
  - `'lower'` – allow doublings of lower pseudo-interval; that is, a subset of the lattice with a unique minimal element
  - `'upper'` – allow doublings of upper pseudo-interval; that is, a subset of the lattice with a unique maximal element
  - `'convex'` – allow doubling of any convex set
  - `'any'` – allow doubling of any set

EXAMPLES:

The pentagon can be constructed by doubling intervals; the 5-element diamond can not be constructed by any doublings:

```
sage: posets.PentagonPoset().is_constructible_by_doublings('interval')
True

sage: posets.DiamondPoset(5).is_constructible_by_doublings('any')            #␣
↪needs sage.combinat
False
```

```
>>> from sage.all import *
>>> posets.PentagonPoset().is_constructible_by_doublings('interval')
True

>>> posets.DiamondPoset(Integer(5)).is_constructible_by_doublings('any')      ␣
↪     # needs sage.combinat
False
```

After doubling both upper and lower pseudo-interval a lattice is constructible by convex subset doubling:

```
sage: L = posets.BooleanLattice(2)
sage: L = L.day_doubling([0, 1, 2])  # A lower pseudo-interval
sage: L.is_constructible_by_doublings('interval')
False
sage: L.is_constructible_by_doublings('lower')
True
sage: L = L.day_doubling([(3,0), (1,1), (2,1)])  # An upper pseudo-interval
sage: L.is_constructible_by_doublings('upper')
False
sage: L.is_constructible_by_doublings('convex')                    #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(2))
>>> L = L.day_doubling([Integer(0), Integer(1), Integer(2)])  # A lower␣
↪pseudo-interval
>>> L.is_constructible_by_doublings('interval')
False
>>> L.is_constructible_by_doublings('lower')
True
>>> L = L.day_doubling([(Integer(3),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(1))])  # An upper pseudo-interval
>>> L.is_constructible_by_doublings('upper')
False
>>> L.is_constructible_by_doublings('convex')                    #␣
↪needs sage.combinat
True
```

An example of a lattice that can be constructed by doublings of a non-convex subsets:

```
sage: L = LatticePoset(DiGraph('OQC?a?@CO?G_C@?GA?O??_??@?BO?A_?G??C??_?@???
↪'))
sage: L.is_constructible_by_doublings('convex')
False
sage: L.is_constructible_by_doublings('any')
True
```

```
>>> from sage.all import *
>>> L = LatticePoset(DiGraph('OQC?a?@CO?G_C@?GA?O??_??@?BO?A_?G??C??_?@???'))
>>> L.is_constructible_by_doublings('convex')
False
>>> L.is_constructible_by_doublings('any')
True
```

See also:

- Stronger properties: `is_distributive()` (doubling by interval), `is_join_semidistributive()` (doubling by lower pseudo-intervals), `is_meet_semidistributive()` (doubling by upper pseudo-intervals)
- Mutually exclusive properties: `is_simple()` (doubling by any set)

- Other: *day_doubling()*

ALGORITHM:

According to [HOLM2016] a lattice L is lower bounded if and only if $|\text{Ji}(L)| = |\text{Ji}(\text{Con } L)|$, and so dually $|\text{Mi}(L)| = |\text{Mi}(\text{Con } L)|$ in upper bounded lattices. The same reference gives a test for being constructible by convex or by any subset.

is_cosectionally_complemented(*certificate=False*)

Return `True` if the lattice is cosectionally complemented, and `False` otherwise.

A lattice is *cosectionally complemented* if all intervals to the top element interpreted as sublattices are complemented lattices.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not cosectionally complemented

OUTPUT:

- If `certificate=False` return `True` or `False`. If `certificate=True` return either `(True, None)` or `(False, (b, e))`, where b is an element so that in the sublattice from b to the top element has no complement for element e.

EXAMPLES:

The smallest sectionally but not cosectionally complemented lattice:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5], 3: [5], 4: [6], 5: [6]})
sage: L.is_sectionally_complemented(), L.is_cosectionally_complemented()
(True, False)
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],
↪Integer(2): [Integer(5)], Integer(3): [Integer(5)], Integer(4):
↪[Integer(6)], Integer(5): [Integer(6)]})
>>> L.is_sectionally_complemented(), L.is_cosectionally_complemented()
(True, False)
```

A sectionally and cosectionally but not relatively complemented lattice:

```
sage: L = LatticePoset(DiGraph('MYi@O?P??D?OG?@?O_?C?Q??O?W?@??O??'))
sage: L.is_sectionally_complemented() and L.is_cosectionally_complemented()
True
sage: L.is_relatively_complemented()
False
```

```
>>> from sage.all import *
>>> L = LatticePoset(DiGraph('MYi@O?P??D?OG?@?O_?C?Q??O?W?@??O??'))
>>> L.is_sectionally_complemented() and L.is_cosectionally_complemented()
True
>>> L.is_relatively_complemented()
False
```

Getting a certificate:

```
sage: L = LatticePoset(DiGraph('HW?@D?Q?GE?G@??'))
sage: L.is_cosectionally_complemented(certificate=True)
(False, (2, 7))
```

```
>>> from sage.all import *
>>> L = LatticePoset(DiGraph('HW?@D?Q?GE?G@??'))
>>> L.is_cosectionally_complemented(certificate=True)
(False, (2, 7))
```

See also

- Dual property: *is_sectionally_complemented()*
- Weaker properties: *is_complemented()*, *is_coatomic()*, *is_regular()*
- Stronger properties: *is_relatively_complemented()*

is_dismantlable(*certificate=False*)

Return True if the lattice is dismantlable, and False otherwise.

An n-element lattice L_n is dismantlable if there is a sublattice chain $L_{n-1} \supset L_{n-2}, \supset \cdots, \supset L_0$ so that every L_i is a sublattice of L_{i+1} with one element less, and L_0 is the empty lattice. In other words, a dismantlable lattice can be reduced to empty lattice removing doubly irreducible element one by one.

INPUT:

- certificate – boolean; whether to return a certificate
  - If certificate = False (default), returns True or False accordingly.
  - If certificate = True, returns:
    * (True, elms) when the lattice is dismantlable, where elms is elements listed in a possible removing order.
    * (False, crown) when the lattice is not dismantlable, where crown is a subposet of $2k$ elements $a_1, \ldots, a_k, b_1, \ldots, b_k$ with covering relations $a_i \lessdot b_i$ and $a_i \lessdot b_{i+1}$ for $i \in [1, \ldots, k-1]$, and $a_k \lessdot b_1$.

EXAMPLES:

```
sage: DL12 = LatticePoset((divisors(12), attrcall("divides")))
sage: DL12.is_dismantlable()
True
sage: DL12.is_dismantlable(certificate=True)
(True, [4, 2, 1, 3, 6, 12])

sage: B3 = posets.BooleanLattice(3)
sage: B3.is_dismantlable()
False
sage: B3.is_dismantlable(certificate=True)
(False, Finite poset containing 6 elements)
```

```
>>> from sage.all import *
>>> DL12 = LatticePoset((divisors(Integer(12)), attrcall("divides")))
```

(continues on next page)

(continued from previous page)

```
>>> DL12.is_dismantlable()
True
>>> DL12.is_dismantlable(certificate=True)
(True, [4, 2, 1, 3, 6, 12])

>>> B3 = posets.BooleanLattice(Integer(3))
>>> B3.is_dismantlable()
False
>>> B3.is_dismantlable(certificate=True)
(False, Finite poset containing 6 elements)
```

Every planar lattice is dismantlable. Converse is not true:

```
sage: L = LatticePoset( ([], [[0, 1], [0, 2], [0, 3], [0, 4],
....:                          [1, 7], [2, 6], [3, 5], [4, 5],
....:                          [4, 6], [4, 7], [5, 8], [6, 8],
....:                          [7, 8]]) )
sage: L.is_dismantlable()
True
sage: L.is_planar()
False
```

```
>>> from sage.all import *
>>> L = LatticePoset( ([], [[Integer(0), Integer(1)], [Integer(0),␣
↪Integer(2)], [Integer(0), Integer(3)], [Integer(0), Integer(4)],
...                        [Integer(1), Integer(7)], [Integer(2),␣
↪Integer(6)], [Integer(3), Integer(5)], [Integer(4), Integer(5)],
...                        [Integer(4), Integer(6)], [Integer(4),␣
↪Integer(7)], [Integer(5), Integer(8)], [Integer(6), Integer(8)],
...                        [Integer(7), Integer(8)]]) )
>>> L.is_dismantlable()
True
>>> L.is_planar()
False
```

See also:

- Stronger properties: *is_planar()*
- Weaker properties: *is_sublattice_dismantlable()*

is_distributive(*certificate=False*)

Return `True` if the lattice is distributive, and `False` otherwise.

A lattice $(L, \vee, \wedge)$ is distributive if meet distributes over join: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ for every $x, y, z \in L$ just like $x \cdot (y + z) = x \cdot y + x \cdot z$ in normal arithmetic. For duality in lattices it follows that then also join distributes over meet.

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (x, y, z))`, where x, y and z

are elements of the lattice such that $x \wedge (y \vee z) \neq (x \wedge y) \vee (x \wedge z)$. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3], 2: [4], 3: [4], 4: [5]})
sage: L.is_distributive()
True
sage: L = LatticePoset({1: [2, 3, 4], 2: [5], 3: [6], 4: [6], 5: [6]})
sage: L.is_distributive()
False
sage: L.is_distributive(certificate=True)
(False, (5, 3, 2))
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4)], Integer(3): [Integer(4)], Integer(4): [Integer(5)]})
>>> L.is_distributive()
True
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5)], Integer(3): [Integer(6)], Integer(4):␣
↪[Integer(6)], Integer(5): [Integer(6)]})
>>> L.is_distributive()
False
>>> L.is_distributive(certificate=True)
(False, (5, 3, 2))
```

See also:

- Weaker properties: *is_modular()*, *is_semidistributive()*, *is_join_distributive()*, *is_meet_distributive()*, *is_subdirectly_reducible()*, *is_trim()*, *is_congruence_uniform()*, *is_extremal()*
- Stronger properties: *is_stone()*

is_extremal()

Return `True` if the lattice is extremal, and `False` otherwise.

A lattice is *extremal* if the number of join-irreducibles is equal to the number of meet-irreducibles and to the number of cover relations in the longest chains.

EXAMPLES:

```
sage: posets.PentagonPoset().is_extremal()
True

sage: P = LatticePoset(posets.SymmetricGroupWeakOrderPoset(3))
sage: P.is_extremal()
False
```

```
>>> from sage.all import *
>>> posets.PentagonPoset().is_extremal()
True
```

(continues on next page)

(continued from previous page)

```
>>> P = LatticePoset(posets.SymmetricGroupWeakOrderPoset(Integer(3)))
>>> P.is_extremal()
False
```

See also

- Stronger properties: `is_distributive()`, `is_trim()`

REFERENCES:

- [Mark1992]

is_geometric()

Return `True` if the lattice is geometric, and `False` otherwise.

A lattice is geometric if it is both atomic and upper semimodular.

EXAMPLES:

Canonical example is the lattice of partitions of finite set ordered by refinement:

```
sage: L = posets.SetPartitions(4)                                   #␣
↪needs sage.combinat
sage: L.is_geometric()                                              #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> L = posets.SetPartitions(Integer(4))                            ␣
↪     # needs sage.combinat
>>> L.is_geometric()                                              #␣
↪needs sage.combinat
True
```

Smallest example of geometric lattice that is not modular:

```
sage: L = LatticePoset(DiGraph('K]?@g@S?q?M?@?@?@?@?@?@??'))
sage: L.is_geometric()
True
sage: L.is_modular()
False
```

```
>>> from sage.all import *
>>> L = LatticePoset(DiGraph('K]?@g@S?q?M?@?@?@?@?@?@??'))
>>> L.is_geometric()
True
>>> L.is_modular()
False
```

Two non-examples:

```
sage: L = LatticePoset({1:[2, 3, 4], 2:[5, 6], 3:[5], 4:[6], 5:[7], 6:[7]})
sage: L.is_geometric()  # Graded, but not upper semimodular
False
sage: L = posets.ChainPoset(3)
sage: L.is_geometric()  # Modular, but not atomic
False
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1):[Integer(2), Integer(3), Integer(4)],␣
↪Integer(2):[Integer(5), Integer(6)], Integer(3):[Integer(5)],␣
↪Integer(4):[Integer(6)], Integer(5):[Integer(7)], Integer(6):[Integer(7)]})
>>> L.is_geometric()  # Graded, but not upper semimodular
False
>>> L = posets.ChainPoset(Integer(3))
>>> L.is_geometric()  # Modular, but not atomic
False
```

See also

- Weaker properties: *is_upper_semimodular()*, *is_relatively_complemented()*

is_interval_dismantlable(*certificate=False*)

Return `True` if the lattice is interval dismantlable, and `False` otherwise.

An interval dismantling is a subdivision of a lattice to a principal upper set and a principal lower set. A lattice is *interval dismantlable* if it can be decomposed into 1-element lattices by consecutive interval distmantlings.

A lattice is *minimally interval non-dismantlable* if it is not interval dismantlable, but all of its sublattices are interval dismantlable.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- if `certificate=False`, return only `True` or `False`
- if `certificate=True`, return either
  - `(True, list)` where `list` is a nested list showing the decomposition; for example `list[1][0]` is a lower part of upper part of the lattice when decomposed twice.
  - `(False, M)` where M is a minimally interval non-dismantlable sublattice of the lattice.

EXAMPLES:

```
sage: L1 = LatticePoset({1: [2, 3], 3: [4, 5], 2: [6], 4: [6], 5: [6]})
sage: L1.is_interval_dismantlable()
True

sage: L2 = LatticePoset({1: [2, 3, 4, 5], 2: [6], 3: [6], 4: [6],
....:                    5: [6, 7], 6: [8], 7: [9, 10], 8:[10], 9:[10]})
sage: L2.is_interval_dismantlable()
False
```

```
>>> from sage.all import *
>>> L1 = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(3):␣
↪[Integer(4), Integer(5)], Integer(2): [Integer(6)], Integer(4):␣
↪[Integer(6)], Integer(5): [Integer(6)]})
>>> L1.is_interval_dismantlable()
True

>>> L2 = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4),␣
↪Integer(5)], Integer(2): [Integer(6)], Integer(3): [Integer(6)],␣
↪Integer(4): [Integer(6)],
...                     Integer(5): [Integer(6), Integer(7)], Integer(6):␣
↪[Integer(8)], Integer(7): [Integer(9), Integer(10)],␣
↪Integer(8):[Integer(10)], Integer(9):[Integer(10)]})
>>> L2.is_interval_dismantlable()
False
```

To get certificates:

```
sage: L1.is_interval_dismantlable(certificate=True)
(True, [[[1], [2]], [[[3], [5]], [[4], [6]]]])
sage: L2.is_interval_dismantlable(certificate=True)
(False, Finite lattice containing 5 elements)
```

```
>>> from sage.all import *
>>> L1.is_interval_dismantlable(certificate=True)
(True, [[[1], [2]], [[[3], [5]], [[4], [6]]]])
>>> L2.is_interval_dismantlable(certificate=True)
(False, Finite lattice containing 5 elements)
```

See also:

- Stronger properties: *is_join_semidistributive()*, *is_meet_semidistributive()*
- Weaker properties: *is_sublattice_dismantlable()*

is_isoform(*certificate=False*)

Return `True` if the lattice is isoform and `False` otherwise.

A congruence is *isoform* (or *isotype*) if all blocks are isomorphic sublattices. A lattice is isoform if it has only isoform congruences.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not isoform

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, C)`, where C is a non-isoform congruence as a *sage.combinat.set_partition.SetPartition*. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5, 6], 3: [6, 7],
....:                    4: [7], 5: [8], 6: [8], 7: [8]})
sage: L.is_isoform()                                                    #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5), Integer(6)], Integer(3): [Integer(6), Integer(7)],
...                    Integer(4): [Integer(7)], Integer(5): [Integer(8)],␣
↪Integer(6): [Integer(8)], Integer(7): [Integer(8)]})
>>> L.is_isoform()                                                      #␣
↪needs sage.combinat
True
```

Every isoform lattice is (trivially) uniform, but the converse is not true:

```
sage: L = LatticePoset({1: [2, 3, 6], 2: [4, 5], 3: [5], 4: [9, 8],
....:                    5: [7, 8], 6: [9], 7: [10], 8: [10], 9: [10]})
sage: L.is_isoform(), L.is_uniform()                                    #␣
↪needs sage.combinat
(False, True)

sage: L.is_isoform(certificate=True)                                    #␣
↪needs sage.combinat
(False, {{1, 2, 4, 6, 9}, {3, 5, 7, 8, 10}})
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(6)],␣
↪Integer(2): [Integer(4), Integer(5)], Integer(3): [Integer(5)], Integer(4):␣
↪[Integer(9), Integer(8)],
...                    Integer(5): [Integer(7), Integer(8)], Integer(6):␣
↪[Integer(9)], Integer(7): [Integer(10)], Integer(8): [Integer(10)],␣
↪Integer(9): [Integer(10)]})
>>> L.is_isoform(), L.is_uniform()                                      #␣
↪needs sage.combinat
(False, True)

>>> L.is_isoform(certificate=True)                                      #␣
↪needs sage.combinat
(False, {{1, 2, 4, 6, 9}, {3, 5, 7, 8, 10}})
```

See also:

- Weaker properties: `is_uniform()`
- Stronger properties: `is_simple()`, `is_relatively_complemented()`
- Other: `congruence()`

is_join_distributive(*certificate=False*)

Return `True` if the lattice is join-distributive and `False` otherwise.

A lattice is *join-distributive* if every interval from an element to the join of the element's upper covers is a distributive lattice. Actually this distributive sublattice is then a Boolean lattice.

They are also called as *Dilworth's lattices* and *upper locally distributive lattices*. They can be characterized in many other ways, see [Dil1940].

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, e)`, where e is an element such that the interval from e to the meet of upper covers of e is not distributive. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5, 6], 3: [5, 7],
....:                   4: [6, 7], 5: [8, 9], 6: [9], 7: [9, 10],
....:                   8: [11], 9: [11], 10: [11]})
sage: L.is_join_distributive()
True

sage: L = LatticePoset({1: [2], 2: [3, 4], 3: [5], 4: [6],
....:                   5: [7], 6: [7]})
sage: L.is_join_distributive()
False
sage: L.is_join_distributive(certificate=True)
(False, 2)
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],
↪Integer(2): [Integer(5), Integer(6)], Integer(3): [Integer(5), Integer(7)],
...                   Integer(4): [Integer(6), Integer(7)], Integer(5):
↪[Integer(8), Integer(9)], Integer(6): [Integer(9)], Integer(7): [Integer(9),
↪ Integer(10)],
...                   Integer(8): [Integer(11)], Integer(9): [Integer(11)],
↪Integer(10): [Integer(11)]})
>>> L.is_join_distributive()
True

>>> L = LatticePoset({Integer(1): [Integer(2)], Integer(2): [Integer(3),
↪Integer(4)], Integer(3): [Integer(5)], Integer(4): [Integer(6)],
...                   Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> L.is_join_distributive()
False
>>> L.is_join_distributive(certificate=True)
(False, 2)
```

See also:

- Dual property: *is_meet_distributive()*
- Weaker properties: *is_meet_semidistributive()*, *is_upper_semimodular()*

- Stronger properties: *is_distributive()*

is_join_pseudocomplemented(*certificate=False*)

Return True if the lattice is join-pseudocomplemented, and False otherwise.

A lattice is join-pseudocomplemented if every element e has a join-pseudocomplement e', i.e. the least element such that the join of e and e' is the top element.

INPUT:

- certificate – boolean (default: False); whether to return a certificate

OUTPUT:

- If certificate=True return either (True, None) or (False, e), where e is an element without a join-pseudocomplement. If certificate=False return True or False.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 5], 2: [3, 6], 3: [4], 4: [7],
....:                   5: [6], 6: [7]})
sage: L.is_join_pseudocomplemented()
True

sage: L = LatticePoset({1: [2, 3], 2: [4, 5, 6], 3: [6], 4: [7],
....:                   5: [7], 6: [7]})
sage: L.is_join_pseudocomplemented()
False
sage: L.is_join_pseudocomplemented(certificate=True)
(False, 4)
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(5)], Integer(2):␣
↪[Integer(3), Integer(6)], Integer(3): [Integer(4)], Integer(4):␣
↪[Integer(7)],
...                   Integer(5): [Integer(6)], Integer(6): [Integer(7)]})
>>> L.is_join_pseudocomplemented()
True

>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4), Integer(5), Integer(6)], Integer(3): [Integer(6)], Integer(4):␣
↪[Integer(7)],
...                   Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> L.is_join_pseudocomplemented()
False
>>> L.is_join_pseudocomplemented(certificate=True)
(False, 4)
```

See also

- Dual property: *is_pseudocomplemented()*
- Stronger properties: *is_join_semidistributive()*

is_join_semidistributive(*certificate=False*)

Return `True` if the lattice is join-semidistributive, and `False` otherwise.

A lattice is join-semidistributive if for all elements e, x, y in the lattice we have

$$e \vee x = e \vee y \implies e \vee x = e \vee (x \wedge y)$$

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (e, x, y))` such that $e \vee x = e \vee y$ but $e \vee x \neq e \vee (x \wedge y)$. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: T4 = posets.TamariLattice(4)
sage: T4.is_join_semidistributive()
True
sage: L = LatticePoset({1:[2, 3], 2:[4, 5], 3:[5, 6],
....:                    4:[7], 5:[7], 6:[7]})
sage: L.is_join_semidistributive()
False
sage: L.is_join_semidistributive(certificate=True)
(False, (5, 4, 6))
```

```
>>> from sage.all import *
>>> T4 = posets.TamariLattice(Integer(4))
>>> T4.is_join_semidistributive()
True
>>> L = LatticePoset({Integer(1):[Integer(2), Integer(3)],
↪Integer(2):[Integer(4), Integer(5)], Integer(3):[Integer(5), Integer(6)],
...                   Integer(4):[Integer(7)], Integer(5):[Integer(7)],
↪Integer(6):[Integer(7)]})
>>> L.is_join_semidistributive()
False
>>> L.is_join_semidistributive(certificate=True)
(False, (5, 4, 6))
```

See also

- Dual property: *is_meet_semidistributive()*
- Weaker properties: *is_join_pseudocomplemented()*, *is_interval_dismantlable()*
- Stronger properties: *is_semidistributive()*, *is_meet_distributive()*, *is_constructible_by_doublings()* (by lower pseudo-intervals)

is_left_modular_element(*x*)

Return `True` if `x` is a left modular element and `False` otherwise.

INPUT:

- `x` – an element of the lattice

An element x in a lattice L is *left modular* if

$$(y \vee x) \wedge z = y \vee (x \wedge z)$$

for every $y \leq z \in L$.

It is enough to check this condition on all cover relations $y < z$.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: [i for i in P if P.is_left_modular_element(i)]
[0, 2, 3, 4]
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> [i for i in P if P.is_left_modular_element(i)]
[0, 2, 3, 4]
```

See also

- Stronger properties: `is_modular_element()`

is_lower_semimodular(*certificate=False*)

Return `True` if the lattice is lower semimodular and `False` otherwise.

A lattice is lower semimodular if any pair of elements with a common upper cover have also a common lower cover.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not lower semimodular

OUTPUT:

- If `certificate=False` return `True` or `False`. If `certificate=True` return either `(True, None)` or `(False, (a, b))`, where a and b are covered by their join but do no cover their meet.

See Wikipedia article Semimodular_lattice

EXAMPLES:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_lower_semimodular()
True

sage: L = posets.PentagonPoset()
sage: L.is_lower_semimodular()
False

sage: L = posets.ChainPoset(6)
sage: L.is_lower_semimodular()
True

sage: L = LatticePoset(DiGraph('IS?`?AAOE_@?C?_@??'))
```

(continues on next page)

(continued from previous page)

```
sage: L.is_lower_semimodular(certificate=True)
(False, (4, 2))
```

```
>>> from sage.all import *
>>> L = posets.DiamondPoset(Integer(5))
>>> L.is_lower_semimodular()
True

>>> L = posets.PentagonPoset()
>>> L.is_lower_semimodular()
False

>>> L = posets.ChainPoset(Integer(6))
>>> L.is_lower_semimodular()
True

>>> L = LatticePoset(DiGraph('IS?`?AAOE_@?C?_@??'))
>>> L.is_lower_semimodular(certificate=True)
(False, (4, 2))
```

See also

- Dual property: *is_upper_semimodular()*
- Weaker properties: *is_graded()*
- Stronger properties: *is_modular()*, *is_meet_distributive()*

is_meet_distributive(*certificate=False*)

Return `True` if the lattice is meet-distributive and `False` otherwise.

A lattice is *meet-distributive* if every interval to an element from the meet of the element's lower covers is a distributive lattice. Actually this distributive sublattice is then a Boolean lattice.

They are also called as *lower locally distributive lattices*. They can be characterized in many other ways, see [Dil1940].

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, e)`, where e is an element such that the interval to e from the meet of lower covers of e is not distributive. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5], 3: [5, 6, 7],
....:                   4: [7], 5: [9, 8], 6: [10, 8], 7:
....:                   [9, 10], 8: [11], 9: [11], 10: [11]})
sage: L.is_meet_distributive()
True
```

(continues on next page)

(continued from previous page)

```
sage: L = LatticePoset({1: [2, 3], 2: [4], 3: [5], 4: [6],
....:                   5: [6], 6: [7]})
sage: L.is_meet_distributive()
False
sage: L.is_meet_distributive(certificate=True)
(False, 6)
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5)], Integer(3): [Integer(5), Integer(6), Integer(7)],
...                   Integer(4): [Integer(7)], Integer(5): [Integer(9),␣
↪Integer(8)], Integer(6): [Integer(10), Integer(8)], Integer(7):
...                   [Integer(9), Integer(10)], Integer(8): [Integer(11)],␣
↪Integer(9): [Integer(11)], Integer(10): [Integer(11)]})
>>> L.is_meet_distributive()
True

>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4)], Integer(3): [Integer(5)], Integer(4): [Integer(6)],
...                   Integer(5): [Integer(6)], Integer(6): [Integer(7)]})
>>> L.is_meet_distributive()
False
>>> L.is_meet_distributive(certificate=True)
(False, 6)
```

See also

- Dual property: *is_join_distributive()*
- Weaker properties: *is_join_semidistributive()*, *is_lower_semimodular()*
- Stronger properties: *is_distributive()*

is_meet_semidistributive(*certificate=False*)

Return `True` if the lattice is meet-semidistributive, and `False` otherwise.

A lattice is meet-semidistributive if for all elements e, x, y in the lattice we have

$$e \wedge x = e \wedge y \implies e \wedge x = e \wedge (x \vee y)$$

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (e, x, y))` such that $e \wedge x = e \wedge y$ but $e \wedge x \neq e \wedge (x \vee y)$. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1:[2, 3, 4], 2:[4, 5], 3:[5, 6],
....:                   4:[7], 5:[7], 6:[7]})
```

(continues on next page)

(continued from previous page)

```
sage: L.is_meet_semidistributive()
True
sage: L_ = L.dual()
sage: L_.is_meet_semidistributive()
False
sage: L_.is_meet_semidistributive(certificate=True)
(False, (5, 4, 6))
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1):[Integer(2), Integer(3), Integer(4)],␣
↪Integer(2):[Integer(4), Integer(5)], Integer(3):[Integer(5), Integer(6)],
...                   Integer(4):[Integer(7)], Integer(5):[Integer(7)],␣
↪Integer(6):[Integer(7)]})
>>> L.is_meet_semidistributive()
True
>>> L_ = L.dual()
>>> L_.is_meet_semidistributive()
False
>>> L_.is_meet_semidistributive(certificate=True)
(False, (5, 4, 6))
```

See also

- Dual property: `is_join_semidistributive()`
- Weaker properties: `is_pseudocomplemented()`, `is_interval_dismantlable()`
- Stronger properties: `is_semidistributive()`, `is_join_distributive()`, `is_constructible_by_doublings()` (by upper pseudo-intervals)

is_modular(*L=None*, *certificate=False*)

Return `True` if the lattice is modular and `False` otherwise.

An element b of a lattice is *modular* if

$$x \vee (a \wedge b) = (x \vee a) \wedge b$$

for every element $x \leq b$ and a. A lattice is modular if every element is modular. There are other equivalent definitions, see Wikipedia article Modular_lattice.

With the parameter `L` this can be used to check that some subset of elements are all modular.

INPUT:

- `L` – (default: `None`) a list of elements to check being modular, if `L` is `None`, then this checks the entire lattice
- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (x, a, b))`, where a, b and x are elements of the lattice such that $x < b$ but $x \vee (a \wedge b) \neq (x \vee a) \wedge b$. If also L is given then b in the certificate will be an element of L. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_modular()
True

sage: L = posets.PentagonPoset()
sage: L.is_modular()
False

sage: L = LatticePoset({1:[2,3],2:[4,5],3:[5,6],4:[7],5:[7],6:[7]})
sage: L.is_modular(certificate=True)
(False, (2, 6, 4))
sage: [L.is_modular([x]) for x in L]
[True, True, False, True, True, False, True]
```

```
>>> from sage.all import *
>>> L = posets.DiamondPoset(Integer(5))
>>> L.is_modular()
True

>>> L = posets.PentagonPoset()
>>> L.is_modular()
False

>>> L = LatticePoset({Integer(1):[Integer(2),Integer(3)],
↪Integer(2):[Integer(4),Integer(5)],Integer(3):[Integer(5),Integer(6)],
↪Integer(4):[Integer(7)],Integer(5):[Integer(7)],Integer(6):[Integer(7)]})
>>> L.is_modular(certificate=True)
(False, (2, 6, 4))
>>> [L.is_modular([x]) for x in L]
[True, True, False, True, True, False, True]
```

See also:

- Weaker properties: *is_upper_semimodular()*, *is_lower_semimodular()*, *is_supersolvable()*
- Stronger properties: *is_distributive()*
- Other: *is_modular_element()*

is_modular_element(*x*)

Return `True` if `x` is a modular element and `False` otherwise.

INPUT:

- `x` – an element of the lattice

An element x in a lattice L is *modular* if $x \leq b$ implies

$$x \vee (a \wedge b) = (x \vee a) \wedge b$$

for every $a, b \in L$.

EXAMPLES:

```
sage: L = LatticePoset({1:[2,3],2:[4,5],3:[5,6],4:[7],5:[7],6:[7]})
sage: L.is_modular()
False
sage: [L.is_modular_element(x) for x in L]
[True, True, False, True, True, False, True]
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1):[Integer(2),Integer(3)],
↪Integer(2):[Integer(4),Integer(5)],Integer(3):[Integer(5),Integer(6)],
↪Integer(4):[Integer(7)],Integer(5):[Integer(7)],Integer(6):[Integer(7)]})
>>> L.is_modular()
False
>>> [L.is_modular_element(x) for x in L]
[True, True, False, True, True, False, True]
```

See also

- Weaker properties: `is_left_modular_element()`
- Other: `is_modular()` to check modularity for the full lattice or some set of elements

is_orthocomplemented(*unique=False*)

Return `True` if the lattice admits an orthocomplementation, and `False` otherwise.

An orthocomplementation of a lattice is a function defined for every element e and marked as $e^{\perp}$ such that 1) they are complements, i.e. $e \vee e^{\perp}$ is the top element and $e \wedge e^{\perp}$ is the bottom element, 2) it is involution, i.e. $(e^{\perp})^{\perp} = e$, and 3) it is order-reversing, i.e. if $a < b$ then $b^{\perp} < a^{\perp}$.

INPUT:

- `unique` – boolean; if `True`, return `True` only if the lattice has exactly one orthocomplementation. If `False` (the default), return `True` when the lattice has at least one orthocomplementation.

EXAMPLES:

```
sage: D5 = posets.DiamondPoset(5)
sage: D5.is_orthocomplemented()
False

sage: D6 = posets.DiamondPoset(6)
sage: D6.is_orthocomplemented()                                              #␣
↪needs sage.groups
True
sage: D6.is_orthocomplemented(unique=True)                                   #␣
↪needs sage.groups
False

sage: hexagon = LatticePoset({0: [1, 2], 1: [3], 2: [4], 3:[5], 4: [5]})
sage: hexagon.is_orthocomplemented(unique=True)                              #␣
↪needs sage.groups
True
```

```
>>> from sage.all import *
>>> D5 = posets.DiamondPoset(Integer(5))
>>> D5.is_orthocomplemented()
False

>>> D6 = posets.DiamondPoset(Integer(6))
>>> D6.is_orthocomplemented()                                          #␣
↪needs sage.groups
True
>>> D6.is_orthocomplemented(unique=True)                               #␣
↪needs sage.groups
False

>>> hexagon = LatticePoset({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(3)], Integer(2): [Integer(4)], Integer(3):[Integer(5)],␣
↪Integer(4): [Integer(5)]})
>>> hexagon.is_orthocomplemented(unique=True)                          #␣
↪needs sage.groups
True
```

See also

- Weaker properties: *is_complemented()*, is_self_dual()

is_planar()

Return True if the lattice is *upward* planar, and False otherwise.

A lattice is upward planar if its Hasse diagram has a planar drawing in the $\mathbb{R}^2$ plane, in such a way that x is strictly below y (on the vertical axis) whenever $x < y$ in the lattice.

Note that the scientific literature on posets often omits "upward" and shortens it to "planar lattice" (e.g. [GW2014]), which can cause confusion with the notion of graph planarity in graph theory.

Note

Not all lattices which are planar – in the sense of graph planarity – admit such a planar drawing (see example below).

ALGORITHM:

Using the result from [Platt1976], this method returns its result by testing that the Hasse diagram of the lattice is planar (in the sense of graph theory) when an edge is added between the top and bottom elements.

EXAMPLES:

The Boolean lattice of 2^3 elements is not upward planar, even if its covering relations graph is planar:

```
sage: B3 = posets.BooleanLattice(3)
sage: B3.is_planar()
False
sage: G = B3.cover_relations_graph()
```

(continues on next page)

(continued from previous page)

```
sage: G.is_planar()
True
```

```
>>> from sage.all import *
>>> B3 = posets.BooleanLattice(Integer(3))
>>> B3.is_planar()
False
>>> G = B3.cover_relations_graph()
>>> G.is_planar()
True
```

Ordinal product of planar lattices is obviously planar. Same does not apply to Cartesian products:

```
sage: P = posets.PentagonPoset()
sage: Pc = P.product(P)
sage: Po = P.ordinal_product(P)
sage: Pc.is_planar()
False
sage: Po.is_planar()
True
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> Pc = P.product(P)
>>> Po = P.ordinal_product(P)
>>> Pc.is_planar()
False
>>> Po.is_planar()
True
```

See also

- Weaker properties: `is_dismantlable()`

is_pseudocomplemented(*certificate=False*)

Return `True` if the lattice is pseudocomplemented, and `False` otherwise.

A lattice is (meet-)pseudocomplemented if every element e has a pseudocomplement $e^\star$, i.e. the greatest element such that the meet of e and $e^\star$ is the bottom element.

See Wikipedia article Pseudocomplement.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, e)`, where `e` is an element without a pseudocomplement. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 5], 2: [3, 6], 3: [4], 4: [7],
....:                   5: [6], 6: [7]})
sage: L.is_pseudocomplemented()
True

sage: L = LatticePoset({1: [2, 3], 2: [4, 5, 6], 3: [6], 4: [7],
....:                   5: [7], 6: [7]})
sage: L.is_pseudocomplemented()
False
sage: L.is_pseudocomplemented(certificate=True)
(False, 3)
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(5)], Integer(2):␣
↪[Integer(3), Integer(6)], Integer(3): [Integer(4)], Integer(4):␣
↪[Integer(7)],
...                   Integer(5): [Integer(6)], Integer(6): [Integer(7)]})
>>> L.is_pseudocomplemented()
True

>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4), Integer(5), Integer(6)], Integer(3): [Integer(6)], Integer(4):␣
↪[Integer(7)],
...                   Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> L.is_pseudocomplemented()
False
>>> L.is_pseudocomplemented(certificate=True)
(False, 3)
```

See also

- Dual property: *is_join_pseudocomplemented()*
- Stronger properties: *is_meet_semidistributive()*
- Other: *pseudocomplement()*.

ALGORITHM:

According to [Cha92] a lattice is pseudocomplemented if and only if every atom has a pseudocomplement. So we only check those.

is_regular(*certificate=False*)

Return `True` if the lattice is regular and `False` otherwise.

A congruence of a lattice is *regular* if it is generated by any of its parts. A lattice is regular if it has only regular congruences.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not regular

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (C, p))`, where C is a non-regular congruence as a *sage.combinat.set_partition.SetPartition* and p is a congruence class of C such that the congruence generated by p is not C. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5, 6], 3: [8, 7], 4: [6, 7],
....:                   5: [8], 6: [9], 7: [9], 8: [9]})
sage: L.is_regular()                                                    #␣
↪needs sage.combinat
True

sage: N5 = posets.PentagonPoset()
sage: N5.is_regular()                                                   #␣
↪needs sage.combinat
False
sage: N5.is_regular(certificate=True)                                   #␣
↪needs sage.combinat
(False, ({{0}, {1}, {2, 3}, {4}}, [0]))
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5), Integer(6)], Integer(3): [Integer(8), Integer(7)],␣
↪Integer(4): [Integer(6), Integer(7)],
...                   Integer(5): [Integer(8)], Integer(6): [Integer(9)],␣
↪Integer(7): [Integer(9)], Integer(8): [Integer(9)]})
>>> L.is_regular()                                                      #␣
↪needs sage.combinat
True

>>> N5 = posets.PentagonPoset()
>>> N5.is_regular()                                                     #␣
↪needs sage.combinat
False
>>> N5.is_regular(certificate=True)                                     #␣
↪needs sage.combinat
(False, ({{0}, {1}, {2, 3}, {4}}, [0]))
```

See also:

- Stronger properties: *is_uniform()*, *is_sectionally_complemented()*, *is_cosectionally_complemented()*
- Mutually exclusive properties: *is_vertically_decomposable()*
- Other: *congruence()*

is_relatively_complemented(*certificate=False*)

Return `True` if the lattice is relatively complemented, and `False` otherwise.

A lattice is relatively complemented if every interval of it is a complemented lattice.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not relatively complemented

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (a, b, c))`, where b is the only element that covers a and is covered by c. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4, 8], 2: [5, 6], 3: [5, 7],
....:                   4: [6, 7], 5: [9], 6: [9], 7: [9], 8: [9]})
sage: L.is_relatively_complemented()
True

sage: L = posets.PentagonPoset()
sage: L.is_relatively_complemented()
False
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4),␣
↪Integer(8)], Integer(2): [Integer(5), Integer(6)], Integer(3): [Integer(5),␣
↪Integer(7)],
...                   Integer(4): [Integer(6), Integer(7)], Integer(5):␣
↪[Integer(9)], Integer(6): [Integer(9)], Integer(7): [Integer(9)],␣
↪Integer(8): [Integer(9)]})
>>> L.is_relatively_complemented()
True

>>> L = posets.PentagonPoset()
>>> L.is_relatively_complemented()
False
```

Relatively complemented lattice must be both atomic and coatomic. Implication to other direction does not hold:

```
sage: L = LatticePoset({0: [1, 2, 3, 4, 5], 1: [6, 7], 2: [6, 8],
....:                   3: [7, 8, 9], 4: [9, 11], 5: [9, 10],
....:                   6: [10, 11], 7: [12], 8: [12], 9: [12],
....:                   10: [12], 11: [12]})
sage: L.is_atomic() and L.is_coatomic()
True
sage: L.is_relatively_complemented()
False
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(0): [Integer(1), Integer(2), Integer(3),␣
↪Integer(4), Integer(5)], Integer(1): [Integer(6), Integer(7)], Integer(2):␣
↪[Integer(6), Integer(8)],
...                   Integer(3): [Integer(7), Integer(8), Integer(9)],␣
↪Integer(4): [Integer(9), Integer(11)], Integer(5): [Integer(9),␣
↪Integer(10)],
...                   Integer(6): [Integer(10), Integer(11)], Integer(7):␣
↪[Integer(12)], Integer(8): [Integer(12)], Integer(9): [Integer(12)],
...                   Integer(10): [Integer(12)], Integer(11): [Integer(12)]})
>>> L.is_atomic() and L.is_coatomic()
True
```

(continues on next page)

(continued from previous page)

```
>>> L.is_relatively_complemented()
False
```

We can also get a non-complemented 3-element interval:

```
sage: L.is_relatively_complemented(certificate=True)
(False, (1, 6, 11))
```

```
>>> from sage.all import *
>>> L.is_relatively_complemented(certificate=True)
(False, (1, 6, 11))
```

See also:

- Weaker properties: *is_sectionally_complemented()*, *is_cosectionally_complemented()*, *is_isoform()*
- Stronger properties: *is_geometric()*

is_sectionally_complemented(*certificate=False*)

Return `True` if the lattice is sectionally complemented, and `False` otherwise.

A lattice is sectionally complemented if all intervals from the bottom element interpreted as sublattices are complemented lattices.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not sectionally complemented

OUTPUT:

- If `certificate=False` return `True` or `False`. If `certificate=True` return either `(True, None)` or `(False, (t, e))`, where t is an element so that in the sublattice from the bottom element to t has no complement for element e.

EXAMPLES:

Smallest examples of a complemented but not sectionally complemented lattice and a sectionally complemented but not relatively complemented lattice:

```
sage: L = posets.PentagonPoset()
sage: L.is_complemented()
True
sage: L.is_sectionally_complemented()
False

sage: L = LatticePoset({0: [1, 2, 3], 1: [4], 2: [4], 3: [5], 4: [5]})
sage: L.is_sectionally_complemented()
True
sage: L.is_relatively_complemented()
False
```

```
>>> from sage.all import *
>>> L = posets.PentagonPoset()
>>> L.is_complemented()
True
>>> L.is_sectionally_complemented()
False

>>> L = LatticePoset({Integer(0): [Integer(1), Integer(2), Integer(3)],␣
↪Integer(1): [Integer(4)], Integer(2): [Integer(4)], Integer(3):␣
↪[Integer(5)], Integer(4): [Integer(5)]})
>>> L.is_sectionally_complemented()
True
>>> L.is_relatively_complemented()
False
```

Getting a certificate:

```
sage: L = LatticePoset(DiGraph('HYOgC?C@?C?G@??'))
sage: L.is_sectionally_complemented(certificate=True)
(False, (6, 1))
```

```
>>> from sage.all import *
>>> L = LatticePoset(DiGraph('HYOgC?C@?C?G@??'))
>>> L.is_sectionally_complemented(certificate=True)
(False, (6, 1))
```

See also

- Dual property: `is_cosectionally_complemented()`
- Weaker properties: `is_complemented()`, `is_atomic()`, `is_regular()`
- Stronger properties: `is_relatively_complemented()`

is_semidistributive()

Return `True` if the lattice is both join- and meet-semidistributive, and `False` otherwise.

EXAMPLES:

Tamari lattices are typical examples of semidistributive but not distributive (and hence not modular) lattices:

```
sage: T4 = posets.TamariLattice(4)
sage: T4.is_semidistributive(), T4.is_distributive()
(True, False)
```

```
>>> from sage.all import *
>>> T4 = posets.TamariLattice(Integer(4))
>>> T4.is_semidistributive(), T4.is_distributive()
(True, False)
```

Smallest non-selfdual example:

```
sage: L = LatticePoset({1: [2, 3], 2: [4, 5], 3: [5], 4: [6], 5: [7], 6: [7]})
sage: L.is_semidistributive()
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4), Integer(5)], Integer(3): [Integer(5)], Integer(4):␣
↪[Integer(6)], Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> L.is_semidistributive()
True
```

The diamond is not semidistributive:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_semidistributive()
False
```

```
>>> from sage.all import *
>>> L = posets.DiamondPoset(Integer(5))
>>> L.is_semidistributive()
False
```

See also

- Weaker properties: *is_join_semidistributive()*, *is_meet_semidistributive()*
- Stronger properties: *is_distributive()*, *is_congruence_uniform()*

is_simple(*certificate=False*)

Return `True` if the lattice is simple and `False` otherwise.

A lattice is *simple* if it has no nontrivial congruences; in other words, for every two distinct elements a and b the principal congruence generated by (a, b) has only one component, i.e. the whole lattice.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not simple

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, c)`, where c is a nontrivial congruence as a *sage.combinat.set_partition.SetPartition*. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: # needs sage.combinat
sage: posets.DiamondPoset(5).is_simple()   # Smallest nontrivial example
True
sage: L = LatticePoset({1: [2, 3], 2: [4, 5], 3: [6], 4: [6], 5: [6]})
sage: L.is_simple()
False
sage: L.is_simple(certificate=True)
(False, {{1, 3}, {2, 4, 5, 6}})
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> posets.DiamondPoset(Integer(5)).is_simple()   # Smallest nontrivial example
True
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4), Integer(5)], Integer(3): [Integer(6)], Integer(4):␣
↪[Integer(6)], Integer(5): [Integer(6)]})
>>> L.is_simple()
False
>>> L.is_simple(certificate=True)
(False, {{1, 3}, {2, 4, 5, 6}})
```

Two more examples. First is a non-simple lattice without any 2-element congruences:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5], 3: [5], 4: [6, 7],
....:                   5: [8], 6: [8], 7: [8]})
sage: L.is_simple()                                                        #␣
↪needs sage.combinat
False
sage: L = LatticePoset({1: [2, 3], 2: [4, 5], 3: [6, 7], 4: [8],
....:                   5: [8], 6: [8], 7: [8]})
sage: L.is_simple()                                                        #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5)], Integer(3): [Integer(5)], Integer(4): [Integer(6),
↪ Integer(7)],
...                   Integer(5): [Integer(8)], Integer(6): [Integer(8)],␣
↪Integer(7): [Integer(8)]})
>>> L.is_simple()                                                        #␣
↪needs sage.combinat
False
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4), Integer(5)], Integer(3): [Integer(6), Integer(7)], Integer(4):␣
↪[Integer(8)],
...                   Integer(5): [Integer(8)], Integer(6): [Integer(8)],␣
↪Integer(7): [Integer(8)]})
>>> L.is_simple()                                                        #␣
↪needs sage.combinat
True
```

See also:

- Weaker properties: `is_isoform()`
- Mutually exclusive properties: `is_constructible_by_doublings()` (by any set)
- Other: `congruence()`

is_stone(*certificate=False*)

Return `True` if the lattice is a Stone lattice, and `False` otherwise.

The lattice is expected to be distributive (and hence pseudocomplemented).

A pseudocomplemented lattice is a Stone lattice if

$$e^* \vee e^{**} = \top$$

for every element e of the lattice, where * is the pseudocomplement and $\top$ is the top element of the lattice.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, e)` such that $e^* \vee e^{**} \neq \top$. If `certificate=False` return `True` or `False`.

EXAMPLES:

Divisor lattices are canonical example:

```
sage: D72 = posets.DivisorLattice(72)
sage: D72.is_stone()
True
```

```
>>> from sage.all import *
>>> D72 = posets.DivisorLattice(Integer(72))
>>> D72.is_stone()
True
```

A non-example:

```
sage: L = LatticePoset({1: [2, 3], 2: [4], 3: [4], 4: [5]})
sage: L.is_stone()
False
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):
↪[Integer(4)], Integer(3): [Integer(4)], Integer(4): [Integer(5)]})
>>> L.is_stone()
False
```

See also:

- Weaker properties: *is_distributive()*

is_subdirectly_reducible(*certificate=False*)

Return `True` if the lattice is subdirectly reducible.

A lattice M is a *subdirect product* of K and L if it is a sublattice of $K \times L$. Lattice M is *subdirectly reducible* if there exists such lattices K and L so that M is not a sublattice of either.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- if `certificate=False`, return only `True` or `False`
- if `certificate=True`, return either
  - `(True, (K, L))` such that the lattice is isomorphic to a sublattice of $K \times L$.
  - `(False, (a, b))`, where a and b are elements that are in the same congruence class for every nontrivial congruence of the lattice. Special case: If the lattice has zero or one element, return `(False, None)`.

EXAMPLES:

```
sage: N5 = posets.PentagonPoset()
sage: N5.is_subdirectly_reducible()                                         #␣
↪needs sage.combinat
False

sage: hex = LatticePoset({1: [2, 3], 2: [4], 3: [5], 4: [6], 5: [6]})
sage: hex.is_subdirectly_reducible()                                        #␣
↪needs sage.combinat
True

sage: hex.is_subdirectly_reducible(certificate=True)                        #␣
↪needs sage.combinat
(True,
 (Finite lattice containing 5 elements, Finite lattice containing 5 elements))

sage: N5.is_subdirectly_reducible(certificate=True)                         #␣
↪needs sage.combinat
(False, (2, 3))
sage: res, cert = hex.is_subdirectly_reducible(certificate=True)            #␣
↪needs sage.combinat
sage: cert[0].is_isomorphic(N5)                                             #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> N5 = posets.PentagonPoset()
>>> N5.is_subdirectly_reducible()                                       #␣
↪needs sage.combinat
False

>>> hex = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4)], Integer(3): [Integer(5)], Integer(4): [Integer(6)],␣
↪Integer(5): [Integer(6)]})
>>> hex.is_subdirectly_reducible()                                      #␣
↪needs sage.combinat
True

>>> hex.is_subdirectly_reducible(certificate=True)                      #␣
↪needs sage.combinat
(True,
 (Finite lattice containing 5 elements, Finite lattice containing 5 elements))
```

(continues on next page)

(continued from previous page)

```
>>> N5.is_subdirectly_reducible(certificate=True)                    #␣
↪needs sage.combinat
(False, (2, 3))
>>> res, cert = hex.is_subdirectly_reducible(certificate=True)       #␣
↪needs sage.combinat
>>> cert[Integer(0)].is_isomorphic(N5)                                ␣
↪     # needs sage.combinat
True
```

See also

- Stronger properties: *is_distributive()*, *is_vertically_decomposable()*
- Other: *subdirect_decomposition()*

is_sublattice(*other*)

Return `True` if the lattice is a sublattice of `other`, and `False` otherwise.

Lattice K is a sublattice of L if K is an (induced) subposet of L and closed under meet and join of L.

Note

This method does not check whether the lattice is a *isomorphic* (i.e., up to relabeling) sublattice of `other`, but only if `other` directly contains the lattice as an sublattice.

EXAMPLES:

A pentagon sublattice in a non-modular lattice:

```
sage: L = LatticePoset({1: [2, 3], 2: [4, 5], 3: [5, 6], 4: [7], 5: [7], 6:␣
↪[7]})
sage: N5 = LatticePoset({1: [2, 6], 2: [4], 4: [7], 6: [7]})
sage: N5.is_sublattice(L)
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4), Integer(5)], Integer(3): [Integer(5), Integer(6)], Integer(4):␣
↪[Integer(7)], Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> N5 = LatticePoset({Integer(1): [Integer(2), Integer(6)], Integer(2):␣
↪[Integer(4)], Integer(4): [Integer(7)], Integer(6): [Integer(7)]})
>>> N5.is_sublattice(L)
True
```

This pentagon is a subposet but not closed under join, hence not a sublattice:

```
sage: N5_ = LatticePoset({1: [2, 3], 2: [4], 3: [7], 4: [7]})
sage: N5_.is_induced_subposet(L)
True
sage: N5_.is_sublattice(L)
False
```

```
>>> from sage.all import *
>>> N5_ = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4)], Integer(3): [Integer(7)], Integer(4): [Integer(7)]})
>>> N5_.is_induced_subposet(L)
True
>>> N5_.is_sublattice(L)
False
```

See also

isomorphic_sublattices_iterator()

is_sublattice_dismantlable()

Return `True` if the lattice is sublattice dismantlable, and `False` otherwise.

A sublattice dismantling is a subdivision of a lattice into two non-empty sublattices. A lattice is *sublattice dismantlable* if it can be decomposed into 1-element lattices by consecutive sublattice dismantlings.

EXAMPLES:

The smallest non-example is this (and the dual):

```
sage: P = Poset({1: [11, 12, 13], 2: [11, 14, 15],
....:            3: [12, 14, 16], 4: [13, 15, 16]})
sage: L = LatticePoset(P.with_bounds())
sage: L.is_sublattice_dismantlable()
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1): [Integer(11), Integer(12), Integer(13)],␣
↪Integer(2): [Integer(11), Integer(14), Integer(15)],
...            Integer(3): [Integer(12), Integer(14), Integer(16)],␣
↪Integer(4): [Integer(13), Integer(15), Integer(16)]})
>>> L = LatticePoset(P.with_bounds())
>>> L.is_sublattice_dismantlable()
False
```

Here we adjoin a (double-irreducible-)dismantlable lattice as a part to an interval-dismantlable lattice:

```
sage: B3 = posets.BooleanLattice(3)
sage: N5 = posets.PentagonPoset()
sage: L = B3.adjunct(N5, 1, 7)
sage: L.is_dismantlable(), L.is_interval_dismantlable()
(False, False)
sage: L.is_sublattice_dismantlable()
True
```

```
>>> from sage.all import *
>>> B3 = posets.BooleanLattice(Integer(3))
>>> N5 = posets.PentagonPoset()
>>> L = B3.adjunct(N5, Integer(1), Integer(7))
>>> L.is_dismantlable(), L.is_interval_dismantlable()
```

(continues on next page)

(continued from previous page)

```
(False, False)
>>> L.is_sublattice_dismantlable()
True
```

See also:

- Stronger properties: *is_dismantlable()*, *is_interval_dismantlable()*

Todo: Add a certificate-option.

is_supersolvable(*certificate=False*)

Return `True` if the lattice is supersolvable, and `False` otherwise.

A lattice L is *supersolvable* if there exists a maximal chain C such that every $x \in C$ is a modular element in L. Equivalent definition is that the sublattice generated by C and any other chain is distributive.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(False, None)` or `(True, C)`, where `C` is a maximal chain of modular elements. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_supersolvable()
True

sage: L = posets.PentagonPoset()
sage: L.is_supersolvable()
False

sage: L = LatticePoset({1:[2,3],2:[4,5],3:[5,6],4:[7],5:[7],6:[7]})
sage: L.is_supersolvable()
True
sage: L.is_supersolvable(certificate=True)
(True, [1, 2, 5, 7])
sage: L.is_modular()
False

sage: L = LatticePoset({0: [1, 2, 3, 4], 1: [5, 6, 7],
....:                   2: [5, 8, 9], 3: [6, 8, 10], 4: [7, 9, 10],
....:                   5: [11], 6: [11], 7: [11], 8: [11],
....:                   9: [11], 10: [11]})
sage: L.is_supersolvable()
False
```

```
>>> from sage.all import *
>>> L = posets.DiamondPoset(Integer(5))
>>> L.is_supersolvable()
True

>>> L = posets.PentagonPoset()
>>> L.is_supersolvable()
False

>>> L = LatticePoset({Integer(1):[Integer(2),Integer(3)],
↪Integer(2):[Integer(4),Integer(5)],Integer(3):[Integer(5),Integer(6)],
↪Integer(4):[Integer(7)],Integer(5):[Integer(7)],Integer(6):[Integer(7)]})
>>> L.is_supersolvable()
True
>>> L.is_supersolvable(certificate=True)
(True, [1, 2, 5, 7])
>>> L.is_modular()
False

>>> L = LatticePoset({Integer(0): [Integer(1), Integer(2), Integer(3),␣
↪Integer(4)], Integer(1): [Integer(5), Integer(6), Integer(7)],
...                  Integer(2): [Integer(5), Integer(8), Integer(9)],␣
↪Integer(3): [Integer(6), Integer(8), Integer(10)], Integer(4): [Integer(7),␣
↪Integer(9), Integer(10)],
...                  Integer(5): [Integer(11)], Integer(6): [Integer(11)],␣
↪Integer(7): [Integer(11)], Integer(8): [Integer(11)],
...                  Integer(9): [Integer(11)], Integer(10): [Integer(11)]})
>>> L.is_supersolvable()
False
```

See also

- Weaker properties: *is_graded()*
- Stronger properties: *is_modular()*

is_trim(*certificate=False*)

Return whether a lattice is trim.

A lattice is trim if it is extremal and left modular.

This notion is defined in [Thom2006].

INPUT:

- `certificate` – boolean (default: `False`); whether to return instead a maximum chain of left modular elements

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.is_trim()
True
```

(continues on next page)

(continued from previous page)

```
sage: Q = LatticePoset(posets.SymmetricGroupWeakOrderPoset(3))
sage: Q.is_trim()
False
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.is_trim()
True

>>> Q = LatticePoset(posets.SymmetricGroupWeakOrderPoset(Integer(3)))
>>> Q.is_trim()
False
```

See also

- Weaker properties: *is_extremal()*
- Stronger properties: *is_distributive()*

REFERENCES:

is_uniform(*certificate=False*)

Return `True` if the lattice is uniform and `False` otherwise.

A congruence is *uniform* if all blocks have equal number of elements. A lattice is uniform if it has only uniform congruences.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not uniform

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, C)`, where C is a non-uniform congruence as a *sage.combinat.set_partition.SetPartition*. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [6, 7], 3: [5], 4: [5],
....:                   5: [9, 8], 6: [9], 7: [10], 8: [10], 9: [10]})
sage: L.is_uniform()                                                    #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(6), Integer(7)], Integer(3): [Integer(5)], Integer(4):␣
↪[Integer(5)],
...                   Integer(5): [Integer(9), Integer(8)], Integer(6):␣
↪[Integer(9)], Integer(7): [Integer(10)], Integer(8): [Integer(10)],␣
↪Integer(9): [Integer(10)]})
>>> L.is_uniform()                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
True
```

Every uniform lattice is regular, but the converse is not true:

```
sage: N6 = LatticePoset({1: [2, 3, 5], 2: [4], 3: [4], 5: [6], 4: [6]})
sage: N6.is_uniform(), N6.is_regular()
(False, True)

sage: N6.is_uniform(certificate=True)                                   #␣
↪needs sage.combinat
(False, {{1, 2, 3, 4}, {5, 6}})
```

```
>>> from sage.all import *
>>> N6 = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(5)],␣
↪Integer(2): [Integer(4)], Integer(3): [Integer(4)], Integer(5):␣
↪[Integer(6)], Integer(4): [Integer(6)]})
>>> N6.is_uniform(), N6.is_regular()
(False, True)

>>> N6.is_uniform(certificate=True)                                    #␣
↪needs sage.combinat
(False, {{1, 2, 3, 4}, {5, 6}})
```

See also

- Weaker properties: *is_regular()*
- Stronger properties: *is_isoform()*
- Other: *congruence()*

is_upper_semimodular(*certificate=False*)

Return `True` if the lattice is upper semimodular and `False` otherwise.

A lattice is upper semimodular if any pair of elements with a common lower cover have also a common upper cover.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate if the lattice is not upper semimodular

OUTPUT:

- If `certificate=False` return `True` or `False`. If `certificate=True` return either `(True, None)` or `(False, (a, b))`, where a and b covers their meet but are not covered by their join.

See Wikipedia article Semimodular_lattice

EXAMPLES:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_upper_semimodular()
```

(continues on next page)

(continued from previous page)

```
True

sage: L = posets.PentagonPoset()
sage: L.is_upper_semimodular()
False

sage: L = LatticePoset(posets.IntegerPartitions(4))                       #␣
↪needs sage.combinat
sage: L.is_upper_semimodular()                                            #␣
↪needs sage.combinat
True

sage: L = LatticePoset({1:[2, 3, 4], 2: [5], 3:[5, 6], 4:[6], 5:[7], 6:[7]})
sage: L.is_upper_semimodular(certificate=True)
(False, (4, 2))
```

```
>>> from sage.all import *
>>> L = posets.DiamondPoset(Integer(5))
>>> L.is_upper_semimodular()
True

>>> L = posets.PentagonPoset()
>>> L.is_upper_semimodular()
False

>>> L = LatticePoset(posets.IntegerPartitions(Integer(4)))                 ␣
↪     # needs sage.combinat
>>> L.is_upper_semimodular()                                             #␣
↪needs sage.combinat
True

>>> L = LatticePoset({Integer(1):[Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5)], Integer(3):[Integer(5), Integer(6)],␣
↪Integer(4):[Integer(6)], Integer(5):[Integer(7)], Integer(6):[Integer(7)]})
>>> L.is_upper_semimodular(certificate=True)
(False, (4, 2))
```

See also:

- Dual property: `is_lower_semimodular()`
- Weaker properties: `is_graded()`
- Stronger properties: `is_modular()`, `is_join_distributive()`, `is_geometric()`

is_vertically_decomposable(*certificate=False*)

Return `True` if the lattice is vertically decomposable, and `False` otherwise.

A lattice is vertically decomposable if it has an element that is comparable to all elements and is neither the bottom nor the top element.

Informally said, a lattice is vertically decomposable if it can be seen as two lattices "glued" by unifying the top element of first lattice to the bottom element of second one.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(False, None)` or `(True, e)`, where e is an element that is comparable to all other elements and is neither the bottom nor the top element. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: posets.TamariLattice(4).is_vertically_decomposable()
False
sage: L = LatticePoset( ([1, 2, 3, 6, 12, 18, 36],
....:     attrcall("divides")) )
sage: L.is_vertically_decomposable()
True
sage: L.is_vertically_decomposable(certificate=True)
(True, 6)
```

```
>>> from sage.all import *
>>> posets.TamariLattice(Integer(4)).is_vertically_decomposable()
False
>>> L = LatticePoset( ([Integer(1), Integer(2), Integer(3), Integer(6),␣
↪Integer(12), Integer(18), Integer(36)],
...     attrcall("divides")) )
>>> L.is_vertically_decomposable()
True
>>> L.is_vertically_decomposable(certificate=True)
(True, 6)
```

See also:

- Weaker properties: *is_subdirectly_reducible()*
- Mutually exclusive properties: *is_atomic()*, *is_coatomic()*, *is_regular()*
- Other: *vertical_decomposition()*

isomorphic_sublattices_iterator(*other*)

Return an iterator over the sublattices of the lattice isomorphic to `other`.

INPUT:

- `other` – a finite lattice

EXAMPLES:

A non-modular lattice contains a pentagon sublattice:

```
sage: L = LatticePoset({1: [2, 3], 2: [4, 5], 3: [5, 6], 4: [7], 5: [7], 6:␣
↪[7]})
sage: L.is_modular()
False
```

(continues on next page)

(continued from previous page)

```
sage: N5 = posets.PentagonPoset()
sage: N5_in_L = next(L.isomorphic_sublattices_iterator(N5)); N5_in_L
Finite lattice containing 5 elements
sage: N5_in_L.list()
[1, 3, 6, 4, 7]
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):
↪[Integer(4), Integer(5)], Integer(3): [Integer(5), Integer(6)], Integer(4):
↪[Integer(7)], Integer(5): [Integer(7)], Integer(6): [Integer(7)]})
>>> L.is_modular()
False
>>> N5 = posets.PentagonPoset()
>>> N5_in_L = next(L.isomorphic_sublattices_iterator(N5)); N5_in_L
Finite lattice containing 5 elements
>>> N5_in_L.list()
[1, 3, 6, 4, 7]
```

A divisor lattice is modular, hence does not contain the pentagon as sublattice, even if it has the pentagon subposet:

```
sage: D12 = posets.DivisorLattice(12)
sage: D12.has_isomorphic_subposet(N5)
True
sage: list(D12.isomorphic_sublattices_iterator(N5))
[]
```

```
>>> from sage.all import *
>>> D12 = posets.DivisorLattice(Integer(12))
>>> D12.has_isomorphic_subposet(N5)
True
>>> list(D12.isomorphic_sublattices_iterator(N5))
[]
```

See also

sage.combinat.posets.posets.FinitePoset.isomorphic_subposets_iterator()

Warning

This function will return same sublattice as many times as there are automorphism on it. This is due to `subgraph_search_iterator()` returning labelled subgraphs.

join_primes()

Return the join-prime elements of the lattice.

An element x of a lattice L is *join-prime* if $x \leq a \vee b$ implies $x \leq a$ or $x \leq b$ for every $a, b \in L$.

These are also called *coprime* in some books. Every join-prime is join-irreducible; converse holds if and only if the lattice is distributive.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5, 6], 3: [5],
....:                   4: [6], 5: [7], 6: [7]})
sage: L.join_primes()
[3, 4]

sage: D12 = posets.DivisorLattice(12)  # Distributive lattice
sage: D12.join_irreducibles() == D12.join_primes()
True
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5), Integer(6)], Integer(3): [Integer(5)],
...                   Integer(4): [Integer(6)], Integer(5): [Integer(7)],␣
↪Integer(6): [Integer(7)]})
>>> L.join_primes()
[3, 4]

>>> D12 = posets.DivisorLattice(Integer(12))  # Distributive lattice
>>> D12.join_irreducibles() == D12.join_primes()
True
```

See also

- Dual function: *meet_primes()*
- Other: join_irreducibles()

maximal_sublattices()

Return maximal (proper) sublattices of the lattice.

EXAMPLES:

```
sage: L = LatticePoset(( [], [[1,2],[1,17],[1,8],[2,3],[2,22],
....:                         [2,5],[2,7],[17,22],[17,13],[8,7],
....:                         [8,13],[3,16],[3,9],[22,16],[22,18],
....:                         [22,10],[5,18],[5,14],[7,9],[7,14],
....:                         [7,10],[13,10],[16,6],[16,19],[9,19],
....:                         [18,6],[18,33],[14,33],[10,19],
....:                         [10,33],[6,4],[19,4],[33,4]] ))
sage: maxs = L.maximal_sublattices()
sage: len(maxs)
7
sage: sorted(maxs[0].list())
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 16, 18, 19, 22, 33]
```

```
>>> from sage.all import *
>>> L = LatticePoset(( [], [[Integer(1),Integer(2)],[Integer(1),Integer(17)],
↪[Integer(1),Integer(8)],[Integer(2),Integer(3)],[Integer(2),Integer(22)],
...                         [Integer(2),Integer(5)],[Integer(2),Integer(7)],
↪[Integer(17),Integer(22)],[Integer(17),Integer(13)],[Integer(8),Integer(7)],
```

(continues on next page)

(continued from previous page)

```
...                          [Integer(8),Integer(13)],[Integer(3),Integer(16)],
↪[Integer(3),Integer(9)],[Integer(22),Integer(16)],[Integer(22),Integer(18)],
...                          [Integer(22),Integer(10)],[Integer(5),
↪Integer(18)],[Integer(5),Integer(14)],[Integer(7),Integer(9)],[Integer(7),
↪Integer(14)],
...                          [Integer(7),Integer(10)],[Integer(13),
↪Integer(10)],[Integer(16),Integer(6)],[Integer(16),Integer(19)],[Integer(9),
↪Integer(19)],
...                          [Integer(18),Integer(6)],[Integer(18),
↪Integer(33)],[Integer(14),Integer(33)],[Integer(10),Integer(19)],
...                          [Integer(10),Integer(33)],[Integer(6),Integer(4)],
↪[Integer(19),Integer(4)],[Integer(33),Integer(4)]] ))
>>> maxs = L.maximal_sublattices()
>>> len(maxs)
7
>>> sorted(maxs[Integer(0)].list())
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 16, 18, 19, 22, 33]
```

meet_primes()

Return the meet-prime elements of the lattice.

An element x of a lattice L is *meet-prime* if $x \geq a \wedge b$ implies $x \geq a$ or $x \geq b$ for every $a, b \in L$.

These are also called just *prime* in some books. Every meet-prime is meet-irreducible; converse holds if and only if the lattice is distributive.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2: [5, 6], 3: [5],
....:                   4: [6], 5: [7], 6: [7]})
sage: L.meet_primes()
[6, 5]

sage: D12 = posets.DivisorLattice(12)
sage: sorted(D12.meet_primes())
[3, 4, 6]
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2): [Integer(5), Integer(6)], Integer(3): [Integer(5)],
...                   Integer(4): [Integer(6)], Integer(5): [Integer(7)],␣
↪Integer(6): [Integer(7)]})
>>> L.meet_primes()
[6, 5]

>>> D12 = posets.DivisorLattice(Integer(12))
>>> sorted(D12.meet_primes())
[3, 4, 6]
```

See also

- Dual function: *join_primes()*

- Other: `meet_irreducibles()`

moebius_algebra(*R*)

Return the Möbius algebra of `self` over `R`.

OUTPUT: an instance of *sage.combinat.posets.moebius_algebra.MoebiusAlgebra*

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: L.moebius_algebra(QQ)
Moebius algebra of Finite lattice containing 16 elements over Rational Field
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> L.moebius_algebra(QQ)
Moebius algebra of Finite lattice containing 16 elements over Rational Field
```

neutral_elements()

Return the list of neutral elements of the lattice.

An element e of the lattice L is *neutral* if the sublattice generated by e, x and y is distributive for all $x, y \in L$. It can also be characterized as an element of intersection of maximal distributive sublattices.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3], 2: [6], 3: [4, 5, 6], 4: [8],
....:                   5: [7], 6: [7], 7: [8, 9], 8: [10], 9: [10]})
sage: L.neutral_elements()
[1, 3, 8, 10]
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
→[Integer(6)], Integer(3): [Integer(4), Integer(5), Integer(6)], Integer(4):␣
→[Integer(8)],
...                   Integer(5): [Integer(7)], Integer(6): [Integer(7)],␣
→Integer(7): [Integer(8), Integer(9)], Integer(8): [Integer(10)],␣
→Integer(9): [Integer(10)]})
>>> L.neutral_elements()
[1, 3, 8, 10]
```

quantum_moebius_algebra(*q=None*)

Return the quantum Möbius algebra of `self` with parameter `q`.

INPUT:

- `q` – (optional) the deformation parameter q

OUTPUT: an instance of *sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra*

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: L.quantum_moebius_algebra()
Quantum Moebius algebra of Finite lattice containing 16 elements
 with q=q over Univariate Laurent Polynomial Ring in q over Integer Ring
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> L.quantum_moebius_algebra()
Quantum Moebius algebra of Finite lattice containing 16 elements
 with q=q over Univariate Laurent Polynomial Ring in q over Integer Ring
```

quotient(*congruence*, *labels='tuple'*)

Return the quotient lattice by `congruence`.

Let L be a lattice and Θ be a congruence of L with congruence classes $\Theta_1, \Theta_2, \ldots$. The quotient lattice L/Θ is the lattice with elements $\{\Theta_1, \Theta_2, \ldots\}$ and meet and join given by the original lattice. Explicitly, if $e_1 \in \Theta_1$ and $e_2 \in \Theta_2$, such that $e_1 \vee e_2 \in \Theta_3$ then $\Theta_1 \vee \Theta_2 = \Theta_3$ in L/Θ and similarly for meets.

INPUT:

- `congruence` – list of lists; a congruence
- `labels` – string; the elements of the resulting lattice and can be one of the following:
  - `'tuple'` – elements are tuples of elements of the original lattice
  - `'lattice'` – elements are sublattices of the original lattice
  - `'integer'` – elements are labeled by integers

Warning

`congruence` is expected to be a valid congruence of the lattice. This is *not* checked.

EXAMPLES:

```
sage: # needs sage.combinat
sage: L = posets.PentagonPoset()
sage: c = L.congruence([[0, 1]])
sage: I = L.quotient(c); I
Finite lattice containing 2 elements
sage: I.top()
(2, 3, 4)
sage: I = L.quotient(c, labels='lattice')
sage: I.top()
Finite lattice containing 3 elements

sage: # needs sage.combinat
sage: B3 = posets.BooleanLattice(3)
sage: c = B3.congruence([[0,1]])
sage: B2 = B3.quotient(c, labels='integer')
sage: B2.is_isomorphic(posets.BooleanLattice(2))
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> L = posets.PentagonPoset()
>>> c = L.congruence([[Integer(0), Integer(1)]])
>>> I = L.quotient(c); I
Finite lattice containing 2 elements
```

(continues on next page)

(continued from previous page)

```
>>> I.top()
(2, 3, 4)
>>> I = L.quotient(c, labels='lattice')
>>> I.top()
Finite lattice containing 3 elements

>>> # needs sage.combinat
>>> B3 = posets.BooleanLattice(Integer(3))
>>> c = B3.congruence([[Integer(0),Integer(1)]])
>>> B2 = B3.quotient(c, labels='integer')
>>> B2.is_isomorphic(posets.BooleanLattice(Integer(2)))
True
```

See also

congruence()

skeleton()

Return the skeleton of the lattice.

The lattice is expected to be pseudocomplemented.

The *skeleton* of a pseudocomplemented lattice L, where * is the pseudocomplementation operation, is the subposet induced by $\{e^* \mid e \in L\}$. Actually this poset is a Boolean lattice.

EXAMPLES:

```
sage: D12 = posets.DivisorLattice(12)
sage: S = D12.skeleton(); S
Finite lattice containing 4 elements
sage: S.cover_relations()
[[1, 3], [1, 4], [3, 12], [4, 12]]

sage: T4 = posets.TamariLattice(4)
sage: T4.skeleton().is_isomorphic(posets.BooleanLattice(3))
True
```

```
>>> from sage.all import *
>>> D12 = posets.DivisorLattice(Integer(12))
>>> S = D12.skeleton(); S
Finite lattice containing 4 elements
>>> S.cover_relations()
[[1, 3], [1, 4], [3, 12], [4, 12]]

>>> T4 = posets.TamariLattice(Integer(4))
>>> T4.skeleton().is_isomorphic(posets.BooleanLattice(Integer(3)))
True
```

See also

sage.combinat.posets.lattices.FiniteMeetSemilattice.pseudocomplement().

subdirect_decomposition()

Return the subdirect decomposition of the lattice.

The subdirect decomposition of a lattice L is the list of smaller lattices $L_1, \dots, L_n$ such that L is a sublattice of $L_1 \times \dots \times L_n$, none of L_i can be decomposed further and L is not a sublattice of any L_i. (Except when the list has only one element, i.e. when the lattice is subdirectly irreducible.)

EXAMPLES:

```
sage: posets.ChainPoset(3).subdirect_decomposition()                          #␣
↪needs sage.combinat
[Finite lattice containing 2 elements, Finite lattice containing 2 elements]

sage: # needs sage.combinat
sage: L = LatticePoset({1: [2, 4], 2: [3], 3: [6, 7], 4: [5, 7],
....:                   5: [9, 8], 6: [9], 7: [9], 8: [10], 9: [10]})
sage: Ldecomp = L.subdirect_decomposition()
sage: [fac.cardinality() for fac in Ldecomp]
[2, 5, 7]
sage: Ldecomp[1].is_isomorphic(posets.PentagonPoset())
True
```

```
>>> from sage.all import *
>>> posets.ChainPoset(Integer(3)).subdirect_decomposition()                   ␣
↪     # needs sage.combinat
[Finite lattice containing 2 elements, Finite lattice containing 2 elements]

>>> # needs sage.combinat
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(4)], Integer(2):␣
↪[Integer(3)], Integer(3): [Integer(6), Integer(7)], Integer(4): [Integer(5),
↪ Integer(7)],
...                   Integer(5): [Integer(9), Integer(8)], Integer(6):␣
↪[Integer(9)], Integer(7): [Integer(9)], Integer(8): [Integer(10)],␣
↪Integer(9): [Integer(10)]})
>>> Ldecomp = L.subdirect_decomposition()
>>> [fac.cardinality() for fac in Ldecomp]
[2, 5, 7]
>>> Ldecomp[Integer(1)].is_isomorphic(posets.PentagonPoset())
True
```

sublattice(*elms*)

Return the smallest sublattice containing elements on the given list.

INPUT:

- `elms` – list of elements of the lattice

EXAMPLES:

```
sage: L = LatticePoset(([], [[1,2],[1,17],[1,8],[2,3],[2,22],[2,5],[2,7],[17,
↪22],[17,13],[8,7],[8,13],[3,16],[3,9],[22,16],[22,18],[22,10],[5,18],[5,14],
↪[7,9],[7,14],[7,10],[13,10],[16,6],[16,19],[9,19],[18,6],[18,33],[14,33],
↪[10,19],[10,33],[6,4],[19,4],[33,4]]))
sage: L.sublattice([14, 13, 22]).list()
[1, 2, 8, 7, 14, 17, 13, 22, 10, 33]
```

(continues on next page)

(continued from previous page)

```
sage: L = posets.BooleanLattice(3)
sage: L.sublattice([3,5,6,7])
Finite lattice containing 8 elements
```

```
>>> from sage.all import *
>>> L = LatticePoset(([], [[Integer(1),Integer(2)],[Integer(1),Integer(17)],
↪[Integer(1),Integer(8)],[Integer(2),Integer(3)],[Integer(2),Integer(22)],
↪[Integer(2),Integer(5)],[Integer(2),Integer(7)],[Integer(17),Integer(22)],
↪[Integer(17),Integer(13)],[Integer(8),Integer(7)],[Integer(8),Integer(13)],
↪[Integer(3),Integer(16)],[Integer(3),Integer(9)],[Integer(22),Integer(16)],
↪[Integer(22),Integer(18)],[Integer(22),Integer(10)],[Integer(5),
↪Integer(18)],[Integer(5),Integer(14)],[Integer(7),Integer(9)],[Integer(7),
↪Integer(14)],[Integer(7),Integer(10)],[Integer(13),Integer(10)],
↪[Integer(16),Integer(6)],[Integer(16),Integer(19)],[Integer(9),Integer(19)],
↪[Integer(18),Integer(6)],[Integer(18),Integer(33)],[Integer(14),
↪Integer(33)],[Integer(10),Integer(19)],[Integer(10),Integer(33)],
↪[Integer(6),Integer(4)],[Integer(19),Integer(4)],[Integer(33),Integer(4)]]))
>>> L.sublattice([Integer(14), Integer(13), Integer(22)]).list()
[1, 2, 8, 7, 14, 17, 13, 22, 10, 33]

>>> L = posets.BooleanLattice(Integer(3))
>>> L.sublattice([Integer(3),Integer(5),Integer(6),Integer(7)])
Finite lattice containing 8 elements
```

sublattices()

Return all sublattices of the lattice.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 3, 4], 2:[5], 3:[5, 6], 4:[6],
....:                   5:[7], 6:[7]})
sage: sublats = L.sublattices(); len(sublats)
54
sage: sublats[3]
Finite lattice containing 4 elements
sage: sublats[3].list()
[1, 2, 3, 5]
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3), Integer(4)],␣
↪Integer(2):[Integer(5)], Integer(3):[Integer(5), Integer(6)],␣
↪Integer(4):[Integer(6)],
...                   Integer(5):[Integer(7)], Integer(6):[Integer(7)]})
>>> sublats = L.sublattices(); len(sublats)
54
>>> sublats[Integer(3)]
Finite lattice containing 4 elements
>>> sublats[Integer(3)].list()
[1, 2, 3, 5]
```

sublattices_lattice(*labels='lattice'*)

Return the lattice of sublattices.

Every element of the returned lattice is a sublattice and they are ordered by containment; that is, atoms are one-element lattices, coatoms are maximal sublattices of the original lattice and so on.

INPUT:

- `labels` – string; can be one of the following:
  - `'lattice'` (default) elements of the lattice will be lattices that correspond to sublattices of the original lattice
  - `'tuple'` – elements are tuples of elements of the sublattices of the original lattice
  - `'integer'` – elements are plain integers

EXAMPLES:

```
sage: D4 = posets.DiamondPoset(4)
sage: sll = D4.sublattices_lattice(labels='tuple')
sage: sll.coatoms()  # = maximal sublattices of the original lattice
[(0, 1, 3), (0, 2, 3)]

sage: L = posets.DivisorLattice(12)
sage: sll = L.sublattices_lattice()
sage: L.is_dismantlable() == (len(sll.atoms()) == sll.rank())
True
```

```
>>> from sage.all import *
>>> D4 = posets.DiamondPoset(Integer(4))
>>> sll = D4.sublattices_lattice(labels='tuple')
>>> sll.coatoms()  # = maximal sublattices of the original lattice
[(0, 1, 3), (0, 2, 3)]

>>> L = posets.DivisorLattice(Integer(12))
>>> sll = L.sublattices_lattice()
>>> L.is_dismantlable() == (len(sll.atoms()) == sll.rank())
True
```

vertical_composition(*other*, *labels='pairs'*)

Return the vertical composition of the lattice with `other`.

Let L and K be lattices and b_K the bottom element of K. The vertical composition of L and K is the ordinal sum of L and $K \setminus \{b_K\}$. Informally said this is lattices "glued" together with a common element.

Mathematically, it is only defined when L and K have no common element; here we force that by giving them different names in the resulting poset.

INPUT:

- `other` – a lattice
- `labels` – string (default: `'pairs'`); can be one of the following:
  - `'pairs'` – each element `v` in this poset will be named `(0, v)` and each element `u` in `other` will be named `(1, u)` in the result
  - `'integers'` – the elements of the result will be relabeled with consecutive integers

EXAMPLES:

```
sage: L = LatticePoset({'a': ['b', 'c'], 'b': ['d'], 'c': ['d']})
sage: K = LatticePoset({'e': ['f', 'g'], 'f': ['h'], 'g': ['h']})
sage: M = L.vertical_composition(K)
sage: M.list()
[(0, 'a'), (0, 'b'), (0, 'c'), (0, 'd'), (1, 'f'), (1, 'g'), (1, 'h')]
sage: M.upper_covers((0, 'd'))
[(1, 'f'), (1, 'g')]

sage: C2 = posets.ChainPoset(2)
sage: M3 = posets.DiamondPoset(5)
sage: L = C2.vertical_composition(M3, labels='integers')
sage: L.cover_relations()
[[0, 1], [1, 2], [1, 3], [1, 4], [2, 5], [3, 5], [4, 5]]
```

```
>>> from sage.all import *
>>> L = LatticePoset({'a': ['b', 'c'], 'b': ['d'], 'c': ['d']})
>>> K = LatticePoset({'e': ['f', 'g'], 'f': ['h'], 'g': ['h']})
>>> M = L.vertical_composition(K)
>>> M.list()
[(0, 'a'), (0, 'b'), (0, 'c'), (0, 'd'), (1, 'f'), (1, 'g'), (1, 'h')]
>>> M.upper_covers((Integer(0), 'd'))
[(1, 'f'), (1, 'g')]

>>> C2 = posets.ChainPoset(Integer(2))
>>> M3 = posets.DiamondPoset(Integer(5))
>>> L = C2.vertical_composition(M3, labels='integers')
>>> L.cover_relations()
[[0, 1], [1, 2], [1, 3], [1, 4], [2, 5], [3, 5], [4, 5]]
```

See also

vertical_decomposition(), *sage.combinat.posets.posets.FinitePoset.ordinal_sum()*

vertical_decomposition(*elements_only=False*)

Return sublattices from the vertical decomposition of the lattice.

Let $d_1, \ldots, d_n$ be elements (excluding the top and bottom elements) comparable to every element of the lattice. Let b be the bottom element and t be the top element. This function returns either a list $d_1, \ldots, d_n$, or the list of intervals $[b, d_1], [d_1, d_2], \ldots, [d_{n-1}, d_n], [d_n, t]$ as lattices.

Informally said, this returns the lattice split into parts at every single-element "cutting point".

INPUT:

- `elements_only` – if `True`, return the list of decomposing elements as defined above; if `False` (the default), return the list of sublattices so that the lattice is a vertical composition of them.

EXAMPLES:

Number 6 is divided by 1, 2, and 3, and it divides 12, 18 and 36:

```
sage: L = LatticePoset( ([1, 2, 3, 6, 12, 18, 36],
....:     attrcall("divides")) )
```

(continues on next page)

(continued from previous page)

```
sage: parts = L.vertical_decomposition()
sage: [lat.list() for lat in parts]
[[1, 2, 3, 6], [6, 12, 18, 36]]
sage: L.vertical_decomposition(elements_only=True)
[6]
```

```
>>> from sage.all import *
>>> L = LatticePoset( ([Integer(1), Integer(2), Integer(3), Integer(6),␣
↪Integer(12), Integer(18), Integer(36)],
...     attrcall("divides")) )
>>> parts = L.vertical_decomposition()
>>> [lat.list() for lat in parts]
[[1, 2, 3, 6], [6, 12, 18, 36]]
>>> L.vertical_decomposition(elements_only=True)
[6]
```

See also

vertical_composition(), *is_vertically_decomposable()*

class sage.combinat.posets.lattices.**FiniteMeetSemilattice**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: *FinitePoset*

Note

We assume that the argument passed to MeetSemilattice is the poset of a meet-semilattice (i.e. a poset with greatest lower bound for each pair of elements).

Element

alias of *MeetSemilatticeElement*

atoms()

Return the list atoms of this (semi)lattice.

An *atom* of a lattice is an element covering the bottom element.

EXAMPLES:

```
sage: L = posets.DivisorLattice(60)
sage: sorted(L.atoms())
[2, 3, 5]
```

```
>>> from sage.all import *
>>> L = posets.DivisorLattice(Integer(60))
>>> sorted(L.atoms())
[2, 3, 5]
```

See also

- Dual function: `coatoms()`

meet(*x*, *y=None*)

Return the meet of given elements in the lattice.

INPUT:

- `x`, `y` – two elements of the (semi)lattice OR
- `x` – list or tuple of elements

EXAMPLES:

```
sage: D = posets.DiamondPoset(5)
sage: D.meet(1, 2)
0
sage: D.meet(1, 1)
1
sage: D.meet(1, 0)
0
sage: D.meet(1, 4)
1
```

```
>>> from sage.all import *
>>> D = posets.DiamondPoset(Integer(5))
>>> D.meet(Integer(1), Integer(2))
0
>>> D.meet(Integer(1), Integer(1))
1
>>> D.meet(Integer(1), Integer(0))
0
>>> D.meet(Integer(1), Integer(4))
1
```

Using list of elements as an argument. Meet of empty list is the bottom element:

```
sage: B4=posets.BooleanLattice(4)
sage: B4.meet([3,5,6])
0
sage: B4.meet([])
15
```

```
>>> from sage.all import *
>>> B4=posets.BooleanLattice(Integer(4))
>>> B4.meet([Integer(3),Integer(5),Integer(6)])
0
>>> B4.meet([])
15
```

For non-facade lattices operator `*` works for meet:

```
sage: L = posets.PentagonPoset(facade=False)
sage: L(1)*L(2)
0
```

```
>>> from sage.all import *
>>> L = posets.PentagonPoset(facade=False)
>>> L(Integer(1))*L(Integer(2))
0
```

See also

- Dual function: *join()*

meet_matrix()

Return a matrix whose `(i,j)` entry is `k`, where `self.linear_extension()[k]` is the meet (greatest lower bound) of `self.linear_extension()[i]` and `self.linear_extension()[j]`.

EXAMPLES:

```
sage: P = LatticePoset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]], facade =␣
↪False)
sage: M = P.meet_matrix(); M
[0 0 0 0 0 0 0 0]
[0 1 0 1 0 0 0 1]
[0 0 2 2 2 0 2 2]
[0 1 2 3 2 0 2 3]
[0 0 2 2 4 0 2 4]
[0 0 0 0 0 5 5 5]
[0 0 2 2 2 5 6 6]
[0 1 2 3 4 5 6 7]
sage: M[P(4).vertex,P(3).vertex] == P(0).vertex
True
sage: M[P(5).vertex,P(2).vertex] == P(2).vertex
True
sage: M[P(5).vertex,P(2).vertex] == P(5).vertex
False
```

```
>>> from sage.all import *
>>> P = LatticePoset([[Integer(1),Integer(3),Integer(2)],[Integer(4)],
↪[Integer(4),Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],
↪[Integer(7)],[]], facade = False)
>>> M = P.meet_matrix(); M
[0 0 0 0 0 0 0 0]
[0 1 0 1 0 0 0 1]
[0 0 2 2 2 0 2 2]
[0 1 2 3 2 0 2 3]
[0 0 2 2 4 0 2 4]
[0 0 0 0 0 5 5 5]
[0 0 2 2 2 5 6 6]
[0 1 2 3 4 5 6 7]
>>> M[P(Integer(4)).vertex,P(Integer(3)).vertex] == P(Integer(0)).vertex
```

(continues on next page)

(continued from previous page)

```
True
>>> M[P(Integer(5)).vertex,P(Integer(2)).vertex] == P(Integer(2)).vertex
True
>>> M[P(Integer(5)).vertex,P(Integer(2)).vertex] == P(Integer(5)).vertex
False
```

pseudocomplement(*element*)

Return the pseudocomplement of `element`, if it exists.

The (meet-)pseudocomplement is the greatest element whose meet with given element is the bottom element. I.e. in a meet-semilattice with bottom element $\hat{0}$ the pseudocomplement of an element e is the element $e^\star$ such that $e \wedge e^\star = \hat{0}$ and $e' \leq e^\star$ if $e \wedge e' = \hat{0}$.

See Wikipedia article Pseudocomplement.

INPUT:

- `element` – an element of the lattice

OUTPUT:

An element of the lattice or `None` if the pseudocomplement does not exist.

EXAMPLES:

The pseudocomplement's pseudocomplement is not always the original element:

```
sage: L = LatticePoset({1: [2, 3], 2: [4], 3: [5], 4: [6], 5: [6]})
sage: L.pseudocomplement(2)
5
sage: L.pseudocomplement(5)
4
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4)], Integer(3): [Integer(5)], Integer(4): [Integer(6)],␣
↪Integer(5): [Integer(6)]})
>>> L.pseudocomplement(Integer(2))
5
>>> L.pseudocomplement(Integer(5))
4
```

An element can have complements but no pseudocomplement, or vice versa:

```
sage: L = LatticePoset({0: [1, 2], 1: [3, 4, 5], 2: [5], 3: [6],
....:                   4: [6], 5: [6]})
sage: L.complements(1), L.pseudocomplement(1)
([], 2)
sage: L.complements(2), L.pseudocomplement(2)
([3, 4], None)
```

```
>>> from sage.all import *
>>> L = LatticePoset({Integer(0): [Integer(1), Integer(2)], Integer(1):␣
↪[Integer(3), Integer(4), Integer(5)], Integer(2): [Integer(5)], Integer(3):␣
↪[Integer(6)],
...                   Integer(4): [Integer(6)], Integer(5): [Integer(6)]})
```

(continues on next page)

(continued from previous page)

```
>>> L.complements(Integer(1)), L.pseudocomplement(Integer(1))
([], 2)
>>> L.complements(Integer(2)), L.pseudocomplement(Integer(2))
([3, 4], None)
```

See also

is_pseudocomplemented()

subjoinsemilattice(*elms*)

Return the smallest join-subsemilattice containing elements on the given list.

INPUT:

- elms – list of elements of the lattice

EXAMPLES:

```
sage: L = posets.DivisorLattice(1000)
sage: L_ = L.subjoinsemilattice([2, 25, 125]); L_
Finite join-semilattice containing 5 elements
sage: sorted(L_.list())
[2, 25, 50, 125, 250]
```

```
>>> from sage.all import *
>>> L = posets.DivisorLattice(Integer(1000))
>>> L_ = L.subjoinsemilattice([Integer(2), Integer(25), Integer(125)]); L_
Finite join-semilattice containing 5 elements
>>> sorted(L_.list())
[2, 25, 50, 125, 250]
```

See also

- Dual function: *submeetsemilattice()*

submeetsemilattice(*elms*)

Return the smallest meet-subsemilattice containing elements on the given list.

INPUT:

- elms – list of elements of the lattice

EXAMPLES:

```
sage: L = posets.DivisorLattice(1000)
sage: L_ = L.submeetsemilattice([200, 250, 125]); L_
Finite meet-semilattice containing 5 elements
sage: L_.list()
[25, 50, 200, 125, 250]
```

```
>>> from sage.all import *
>>> L = posets.DivisorLattice(Integer(1000))
>>> L_ = L.submeetsemilattice([Integer(200), Integer(250), Integer(125)]); L_
Finite meet-semilattice containing 5 elements
>>> L_.list()
[25, 50, 200, 125, 250]
```

See also

- Dual function: `subjoinsemilattice()`

`sage.combinat.posets.lattices.JoinSemilattice(data=None, *args, **options)`

Construct a join semi-lattice from various forms of input data.

INPUT:

- `data`, `*args`, `**options` – data and options that will be passed down to `Poset()` to construct a poset that is also a join semilattice

See also

`Poset()`, `MeetSemilattice()`, `LatticePoset()`

EXAMPLES:

Using data that defines a poset:

```
sage: JoinSemilattice([[1,2],[3],[3]])
Finite join-semilattice containing 3 elements

sage: JoinSemilattice([[1,2],[3],[3]], cover_relations = True)
Finite join-semilattice containing 3 elements
```

```
>>> from sage.all import *
>>> JoinSemilattice([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]])
Finite join-semilattice containing 3 elements

>>> JoinSemilattice([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]], cover_
↪relations = True)
Finite join-semilattice containing 3 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: J = JoinSemilattice(P); J
Finite join-semilattice containing 3 elements
sage: type(J)
<class 'sage.combinat.posets.lattices.FiniteJoinSemilattice_with_category'>
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]])
```

(continues on next page)

(continued from previous page)

```
>>> J = JoinSemilattice(P); J
Finite join-semilattice containing 3 elements
>>> type(J)
<class 'sage.combinat.posets.lattices.FiniteJoinSemilattice_with_category'>
```

If the data is not a lattice, then an error is raised:

```
sage: JoinSemilattice({'a': ['b', 'c'], 'b': ['d', 'e'],
....:                  'c': ['d', 'e'], 'd': ['f'], 'e': ['f']})
Traceback (most recent call last):
...
LatticeError: no join for b and c
```

```
>>> from sage.all import *
>>> JoinSemilattice({'a': ['b', 'c'], 'b': ['d', 'e'],
...                  'c': ['d', 'e'], 'd': ['f'], 'e': ['f']})
Traceback (most recent call last):
...
LatticeError: no join for b and c
```

sage.combinat.posets.lattices.**LatticePoset**(*data=None*, **args*, ***options*)

Construct a lattice from various forms of input data.

INPUT:

- `data`, `*args`, `**options` – data and options that will be passed down to *Poset()* to construct a poset that is also a lattice.

OUTPUT: an instance of *FiniteLatticePoset*

See also

Posets, FiniteLatticePosets, JoinSemiLattice(), MeetSemiLattice()

EXAMPLES:

Using data that defines a poset:

```
sage: LatticePoset([[1,2],[3],[3]])
Finite lattice containing 3 elements

sage: LatticePoset([[1,2],[3],[3]], cover_relations = True)
Finite lattice containing 3 elements
```

```
>>> from sage.all import *
>>> LatticePoset([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]])
Finite lattice containing 3 elements

>>> LatticePoset([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]], cover_
↪relations = True)
Finite lattice containing 3 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: L = LatticePoset(P); L
Finite lattice containing 3 elements
sage: type(L)
<class 'sage.combinat.posets.lattices.FiniteLatticePoset_with_category'>
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]])
>>> L = LatticePoset(P); L
Finite lattice containing 3 elements
>>> type(L)
<class 'sage.combinat.posets.lattices.FiniteLatticePoset_with_category'>
```

If the data is not a lattice, then an error is raised:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: LatticePoset((elms, rels))
Traceback (most recent call last):
...
ValueError: not a meet-semilattice: no bottom element
```

```
>>> from sage.all import *
>>> elms = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7)]
>>> rels = [[Integer(1),Integer(2)],[Integer(3),Integer(4)],[Integer(4),
↪Integer(5)],[Integer(2),Integer(5)]]
>>> LatticePoset((elms, rels))
Traceback (most recent call last):
...
ValueError: not a meet-semilattice: no bottom element
```

Creating a facade lattice:

```
sage: L = LatticePoset([[1,2],[3],[3]], facade = True)
sage: L.category()
Category of facade finite enumerated lattice posets
sage: parent(L[0])
Integer Ring
sage: TestSuite(L).run(skip = ['_test_an_element']) # is_parent_of is not yet␣
↪implemented
```

```
>>> from sage.all import *
>>> L = LatticePoset([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]], facade␣
↪= True)
>>> L.category()
Category of facade finite enumerated lattice posets
>>> parent(L[Integer(0)])
Integer Ring
>>> TestSuite(L).run(skip = ['_test_an_element']) # is_parent_of is not yet␣
↪implemented
```

sage.combinat.posets.lattices.**MeetSemilattice**(*data=None*, **args*, ***options*)

Construct a meet semi-lattice from various forms of input data.

INPUT:

- `data`, `*args`, `**options` – data and options that will be passed down to `Poset()` to construct a poset that is also a meet semilattice.

See also

`Poset()`, `JoinSemilattice()`, `LatticePoset()`

EXAMPLES:

Using data that defines a poset:

```
sage: MeetSemilattice([[1,2],[3],[3]])
Finite meet-semilattice containing 3 elements

sage: MeetSemilattice([[1,2],[3],[3]], cover_relations = True)
Finite meet-semilattice containing 3 elements
```

```
>>> from sage.all import *
>>> MeetSemilattice([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]])
Finite meet-semilattice containing 3 elements

>>> MeetSemilattice([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]], cover_
→relations = True)
Finite meet-semilattice containing 3 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: L = MeetSemilattice(P); L
Finite meet-semilattice containing 3 elements
sage: type(L)
<class 'sage.combinat.posets.lattices.FiniteMeetSemilattice_with_category'>
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]])
>>> L = MeetSemilattice(P); L
Finite meet-semilattice containing 3 elements
>>> type(L)
<class 'sage.combinat.posets.lattices.FiniteMeetSemilattice_with_category'>
```

If the data is not a lattice, then an error is raised:

```
sage: MeetSemilattice({'a': ['b', 'c'], 'b': ['d', 'e'],
....:                  'c': ['d', 'e'], 'd': ['f'], 'e': ['f']})
Traceback (most recent call last):
...
LatticeError: no meet for e and d
```

```
>>> from sage.all import *
>>> MeetSemilattice({'a': ['b', 'c'], 'b': ['d', 'e'],
...                  'c': ['d', 'e'], 'd': ['f'], 'e': ['f']})
Traceback (most recent call last):
...
LatticeError: no meet for e and d
```

5.1.183 Linear Extensions of Posets

This module defines two classes:

- *LinearExtensionOfPoset*
- *LinearExtensionsOfPoset*
- *LinearExtensionsOfPosetWithHooks*
- *LinearExtensionsOfForest*

Classes and methods

class sage.combinat.posets.linear_extensions.**LinearExtensionOfPoset**

Bases: ClonableArray

A linear extension of a finite poset P of size n is a total ordering $\pi := \pi_0\pi_1\dots\pi_{n-1}$ of its elements such that $i < j$ whenever $\pi_i < \pi_j$ in the poset P.

When the elements of P are indexed by $\{1, 2, \dots, n\}$, π denotes a permutation of the elements of P in one-line notation.

INPUT:

- linear_extension – list of the elements of P
- poset – the underlying poset P

See also

Poset, *LinearExtensionsOfPoset*

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]),
....:           linear_extension=True, facade=False)
sage: p = P.linear_extension([1,4,2,3]); p
[1, 4, 2, 3]
sage: p.parent()
The set of all linear extensions of
 Finite poset containing 4 elements with distinguished linear extension
sage: p[0], p[1], p[2], p[3]
(1, 4, 2, 3)
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)], [[Integer(1),
↪Integer(3)],[Integer(1),Integer(4)],[Integer(2),Integer(3)]]),
...           linear_extension=True, facade=False)
```

(continues on next page)

(continued from previous page)

```
>>> p = P.linear_extension([Integer(1),Integer(4),Integer(2),Integer(3)]); p
[1, 4, 2, 3]
>>> p.parent()
The set of all linear extensions of
 Finite poset containing 4 elements with distinguished linear extension
>>> p[Integer(0)], p[Integer(1)], p[Integer(2)], p[Integer(3)]
(1, 4, 2, 3)
```

Following Schützenberger and later Haiman and Malvenuto-Reutenauer, Stanley [Stan2009] defined a promotion and evacuation operator on any finite poset P using operators τ_i on the linear extensions of P:

```
sage: p.promotion()
[1, 2, 3, 4]
sage: Q = p.promotion().to_poset()
sage: Q.cover_relations()
[[1, 3], [1, 4], [2, 3]]
sage: Q == P
True

sage: p.promotion(3)
[1, 4, 2, 3]
sage: Q = p.promotion(3).to_poset()
sage: Q == P
False
sage: Q.cover_relations()
[[1, 2], [1, 4], [3, 4]]
```

```
>>> from sage.all import *
>>> p.promotion()
[1, 2, 3, 4]
>>> Q = p.promotion().to_poset()
>>> Q.cover_relations()
[[1, 3], [1, 4], [2, 3]]
>>> Q == P
True

>>> p.promotion(Integer(3))
[1, 4, 2, 3]
>>> Q = p.promotion(Integer(3)).to_poset()
>>> Q == P
False
>>> Q.cover_relations()
[[1, 2], [1, 4], [3, 4]]
```

check()

Check whether `self` is indeed a linear extension of the underlying poset.

evacuation()

Compute evacuation on the linear extension of a poset.

Evacuation on a linear extension π of length n is defined as $\pi(\tau_1 \cdots \tau_{n-1})(\tau_1 \cdots \tau_{n-2}) \cdots (\tau_1)$. For more details see [Stan2009].

See also

`tau()`, `promotion()`

EXAMPLES:

```
sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,
↪6]]))
sage: p = P.linear_extension([1,2,3,4,5,6,7])
sage: p.evacuation()
[1, 4, 2, 3, 7, 5, 6]
sage: p.evacuation().evacuation() == p
True
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7)], [[Integer(1),Integer(2)],[Integer(1),Integer(4)],
↪[Integer(2),Integer(3)],[Integer(2),Integer(5)],[Integer(3),Integer(6)],
↪[Integer(4),Integer(7)],[Integer(5),Integer(6)]]))
>>> p = P.linear_extension([Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5),Integer(6),Integer(7)])
>>> p.evacuation()
[1, 4, 2, 3, 7, 5, 6]
>>> p.evacuation().evacuation() == p
True
```

is_greedy()

Return `True` if the linear extension is greedy.

A linear extension $[e_1, e_2, \ldots, e_n]$ is *greedy* if for every i either e_{i+1} covers e_i or all upper covers of e_i have at least one lower cover that is not in $[e_1, e_2, \ldots, e_i]$.

Informally said a linear extension is greedy if it "always goes up when possible" and so has no unnecessary jumps.

EXAMPLES:

```
sage: P = posets.PentagonPoset()                                        #␣
↪needs sage.modules
sage: for l in P.linear_extensions():                                   #␣
↪needs sage.modules
....:     if not l.is_greedy():
....:         print(l)
[0, 2, 1, 3, 4]
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()                                        #␣
↪needs sage.modules
>>> for l in P.linear_extensions():                                   #␣
↪needs sage.modules
...     if not l.is_greedy():
...         print(l)
[0, 2, 1, 3, 4]
```

is_supergreedy()

Return `True` if the linear extension is supergreedy.

A linear extension of a poset P with elements $\{x_1, x_2, ..., x_t\}$ is *super greedy*, if it can be obtained using the following algorithm: choose x_1 to be a minimal element of P; suppose $X = \{x_1, ..., x_i\}$ have been chosen; let M be the set of minimal elements of $P \setminus X$. If there is an element of M which covers an element x_j in X, then let x_{i+1} be one of these such that j is maximal; otherwise, choose x_{i+1} to be any element of M.

Informally, a linear extension is supergreedy if it "always goes up and receedes the least"; in other words, supergreedy linear extensions are depth-first linear extensions. For more details see [KTZ1987].

EXAMPLES:

```
sage: X = [0,1,2,3,4,5,6]
sage: Y = [[0,5],[1,4],[1,5],[3,6],[4,3],[5,6],[6,2]]
sage: P = Poset((X,Y), cover_relations=True, facade=False)
sage: for l in P.linear_extensions():                                    #␣
↪needs sage.modules
....:     if l.is_supergreedy():
....:         print(l)
[1, 4, 3, 0, 5, 6, 2]
[0, 1, 4, 3, 5, 6, 2]
[0, 1, 5, 4, 3, 6, 2]

sage: Q = posets.PentagonPoset()                                         #␣
↪needs sage.modules
sage: for l in Q.linear_extensions():                                    #␣
↪needs sage.modules sage.rings.finite_rings
....:     if not l.is_supergreedy():
....:         print(l)
[0, 2, 1, 3, 4]
```

```
>>> from sage.all import *
>>> X = [Integer(0),Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6)]
>>> Y = [[Integer(0),Integer(5)],[Integer(1),Integer(4)],[Integer(1),
↪Integer(5)],[Integer(3),Integer(6)],[Integer(4),Integer(3)],[Integer(5),
↪Integer(6)],[Integer(6),Integer(2)]]
>>> P = Poset((X,Y), cover_relations=True, facade=False)
>>> for l in P.linear_extensions():                                    #␣
↪needs sage.modules
...     if l.is_supergreedy():
...         print(l)
[1, 4, 3, 0, 5, 6, 2]
[0, 1, 4, 3, 5, 6, 2]
[0, 1, 5, 4, 3, 6, 2]

>>> Q = posets.PentagonPoset()                                         #␣
↪needs sage.modules
>>> for l in Q.linear_extensions():                                    #␣
↪needs sage.modules sage.rings.finite_rings
...     if not l.is_supergreedy():
...         print(l)
[0, 2, 1, 3, 4]
```

jump_count()

Return the number of jumps in the linear extension.

A *jump* in a linear extension $[e_1, e_2, \dots, e_n]$ is a pair (e_i, e_{i+1}) such that e_{i+1} does not cover e_i.

See also

- *sage.combinat.posets.posets.FinitePoset.jump_number()*

EXAMPLES:

```
sage: B3 = posets.BooleanLattice(3)
sage: l1 = B3.linear_extension((0, 1, 2, 3, 4, 5, 6, 7))
sage: l1.jump_count()
3
sage: l2 = B3.linear_extension((0, 1, 2, 4, 3, 5, 6, 7))
sage: l2.jump_count()
5
```

```
>>> from sage.all import *
>>> B3 = posets.BooleanLattice(Integer(3))
>>> l1 = B3.linear_extension((Integer(0), Integer(1), Integer(2), Integer(3),␣
↪Integer(4), Integer(5), Integer(6), Integer(7)))
>>> l1.jump_count()
3
>>> l2 = B3.linear_extension((Integer(0), Integer(1), Integer(2), Integer(4),␣
↪Integer(3), Integer(5), Integer(6), Integer(7)))
>>> l2.jump_count()
5
```

poset()

Return the underlying original poset.

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,2],[2,3],[1,4]]))
sage: p = P.linear_extension([1,2,4,3])
sage: p.poset()
Finite poset containing 4 elements
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)], [[Integer(1),
↪Integer(2)],[Integer(2),Integer(3)],[Integer(1),Integer(4)]]))
>>> p = P.linear_extension([Integer(1),Integer(2),Integer(4),Integer(3)])
>>> p.poset()
Finite poset containing 4 elements
```

promotion(*i=1*)

Compute the (generalized) promotion on the linear extension of a poset.

INPUT:

- i – (default: 1) an integer between 1 and $n - 1$, where n is the cardinality of the poset

The i-th generalized promotion operator ∂_i on a linear extension π is defined as $\pi\tau_i\tau_{i+1}\cdots\tau_{n-1}$, where n is the size of the linear extension (or size of the underlying poset).

For more details see [Stan2009].

See also

`tau()`, `evacuation()`

EXAMPLES:

```
sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,
↪6]]))
sage: p = P.linear_extension([1,2,3,4,5,6,7])
sage: q = p.promotion(4); q
[1, 2, 3, 5, 6, 4, 7]
sage: p.to_poset() == q.to_poset()
False
sage: p.to_poset().is_isomorphic(q.to_poset())
True
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7)], [[Integer(1),Integer(2)],[Integer(1),Integer(4)],
↪[Integer(2),Integer(3)],[Integer(2),Integer(5)],[Integer(3),Integer(6)],
↪[Integer(4),Integer(7)],[Integer(5),Integer(6)]]))
>>> p = P.linear_extension([Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5),Integer(6),Integer(7)])
>>> q = p.promotion(Integer(4)); q
[1, 2, 3, 5, 6, 4, 7]
>>> p.to_poset() == q.to_poset()
False
>>> p.to_poset().is_isomorphic(q.to_poset())
True
```

tau(*i*)

Return the operator τ_i on linear extensions `self` of a poset.

INPUT:

- `i` – integer between 1 and $n-1$, where n is the cardinality of the poset

The operator τ_i on a linear extension π of a poset P interchanges positions i and $i+1$ if the result is again a linear extension of P, and otherwise acts trivially. For more details, see [Stan2009].

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension=True)
sage: L = P.linear_extensions()
sage: l = L.an_element(); l
[1, 2, 3, 4]
sage: l.tau(1)
[2, 1, 3, 4]
sage: for p in L:                                                          #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
....:     for i in range(1,4):
....:         print("{} {} {}".format(i, p, p.tau(i)))
1 [1, 2, 3, 4] [2, 1, 3, 4]
2 [1, 2, 3, 4] [1, 2, 3, 4]
3 [1, 2, 3, 4] [1, 2, 4, 3]
1 [2, 1, 3, 4] [1, 2, 3, 4]
2 [2, 1, 3, 4] [2, 1, 3, 4]
3 [2, 1, 3, 4] [2, 1, 4, 3]
1 [2, 1, 4, 3] [1, 2, 4, 3]
2 [2, 1, 4, 3] [2, 1, 4, 3]
3 [2, 1, 4, 3] [2, 1, 3, 4]
1 [1, 4, 2, 3] [1, 4, 2, 3]
2 [1, 4, 2, 3] [1, 2, 4, 3]
3 [1, 4, 2, 3] [1, 4, 2, 3]
1 [1, 2, 4, 3] [2, 1, 4, 3]
2 [1, 2, 4, 3] [1, 4, 2, 3]
3 [1, 2, 4, 3] [1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)], [[Integer(1),
↪Integer(3)],[Integer(1),Integer(4)],[Integer(2),Integer(3)]]), linear_
↪extension=True)
>>> L = P.linear_extensions()
>>> l = L.an_element(); l
[1, 2, 3, 4]
>>> l.tau(Integer(1))
[2, 1, 3, 4]
>>> for p in L:                                                          #␣
↪needs sage.modules
...     for i in range(Integer(1),Integer(4)):
...         print("{} {} {}".format(i, p, p.tau(i)))
1 [1, 2, 3, 4] [2, 1, 3, 4]
2 [1, 2, 3, 4] [1, 2, 3, 4]
3 [1, 2, 3, 4] [1, 2, 4, 3]
1 [2, 1, 3, 4] [1, 2, 3, 4]
2 [2, 1, 3, 4] [2, 1, 3, 4]
3 [2, 1, 3, 4] [2, 1, 4, 3]
1 [2, 1, 4, 3] [1, 2, 4, 3]
2 [2, 1, 4, 3] [2, 1, 4, 3]
3 [2, 1, 4, 3] [2, 1, 3, 4]
1 [1, 4, 2, 3] [1, 4, 2, 3]
2 [1, 4, 2, 3] [1, 2, 4, 3]
3 [1, 4, 2, 3] [1, 4, 2, 3]
1 [1, 2, 4, 3] [2, 1, 4, 3]
2 [1, 2, 4, 3] [1, 4, 2, 3]
3 [1, 2, 4, 3] [1, 2, 3, 4]
```

to_poset()

Return the poset associated to the linear extension `self`.

This method returns the poset obtained from the original poset P by relabelling the i-th element of `self` to the i-th element of the original poset, while keeping the linear extension of the original poset.

For a poset with default linear extension $1, \dots, n$, `self` can be interpreted as a permutation, and the relabelling is done according to the inverse of this permutation.

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,2],[1,3],[3,4]]), linear_extension=True,␣
↪facade=False)
sage: p = P.linear_extension([1,3,4,2])
sage: Q = p.to_poset(); Q
Finite poset containing 4 elements with distinguished linear extension
sage: P == Q
False
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)], [[Integer(1),
↪Integer(2)],[Integer(1),Integer(3)],[Integer(3),Integer(4)]]), linear_
↪extension=True, facade=False)
>>> p = P.linear_extension([Integer(1),Integer(3),Integer(4),Integer(2)])
>>> Q = p.to_poset(); Q
Finite poset containing 4 elements with distinguished linear extension
>>> P == Q
False
```

The default linear extension remains the same:

```
sage: list(P)
[1, 2, 3, 4]
sage: list(Q)
[1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> list(P)
[1, 2, 3, 4]
>>> list(Q)
[1, 2, 3, 4]
```

But the relabelling can be seen on cover relations:

```
sage: P.cover_relations()
[[1, 2], [1, 3], [3, 4]]
sage: Q.cover_relations()
[[1, 2], [1, 4], [2, 3]]

sage: p = P.linear_extension([1,2,3,4])
sage: Q = p.to_poset()
sage: P == Q
True
```

```
>>> from sage.all import *
>>> P.cover_relations()
[[1, 2], [1, 3], [3, 4]]
>>> Q.cover_relations()
[[1, 2], [1, 4], [2, 3]]
```

(continues on next page)

(continued from previous page)

```
>>> p = P.linear_extension([Integer(1),Integer(2),Integer(3),Integer(4)])
>>> Q = p.to_poset()
>>> P == Q
True
```

class sage.combinat.posets.linear_extensions.**LinearExtensionsOfForest**(*poset*, *facade*)

Bases: *LinearExtensionsOfPoset*

Linear extensions such that the poset is a forest.

cardinality()

Use Atkinson's algorithm to compute the number of linear extensions.

EXAMPLES:

```
sage: from sage.combinat.posets.forest import ForestPoset
sage: from sage.combinat.posets.poset_examples import Posets
sage: P = Poset({0: [2], 1: [2], 2: [3, 4], 3: [], 4: []})
sage: P.linear_extensions().cardinality()                                   #␣
↪needs sage.modules
4

sage: Q = Poset({0: [1], 1: [2, 3], 2: [], 3: [], 4: [5, 6], 5: [], 6: []})
sage: Q.linear_extensions().cardinality()                                   #␣
↪needs sage.modules
140
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.forest import ForestPoset
>>> from sage.combinat.posets.poset_examples import Posets
>>> P = Poset({Integer(0): [Integer(2)], Integer(1): [Integer(2)],␣
↪Integer(2): [Integer(3), Integer(4)], Integer(3): [], Integer(4): []})
>>> P.linear_extensions().cardinality()                                 #␣
↪needs sage.modules
4

>>> Q = Poset({Integer(0): [Integer(1)], Integer(1): [Integer(2), Integer(3)],
↪ Integer(2): [], Integer(3): [], Integer(4): [Integer(5), Integer(6)],␣
↪Integer(5): [], Integer(6): []})
>>> Q.linear_extensions().cardinality()                                 #␣
↪needs sage.modules
140
```

class sage.combinat.posets.linear_extensions.**LinearExtensionsOfMobile**(*poset*, *facade*)

Bases: *LinearExtensionsOfPoset*

Linear extensions for a mobile poset.

cardinality()

Return the number of linear extensions by using the determinant formula for counting linear extensions of mobiles.

EXAMPLES:

```
sage: from sage.combinat.posets.mobile import MobilePoset
sage: M = MobilePoset(DiGraph([[0,1,2,3,4,5,6,7,8], [(1,0),(3,0),(2,1),(2,3),
↪(4,
....: 3), (5,4),(5,6),(7,4),(7,8)]]))
sage: M.linear_extensions().cardinality()                                      #␣
↪needs sage.modules
1098

sage: M1 = posets.RibbonPoset(6, [1,3])
sage: M1.linear_extensions().cardinality()                                     #␣
↪needs sage.modules
61

sage: P = posets.MobilePoset(posets.RibbonPoset(7, [1,3]),                     #␣
↪needs sage.combinat sage.modules
....:                           {1: [posets.YoungDiagramPoset([3, 2],␣
↪dual=True)],
....:                            3: [posets.DoubleTailedDiamond(6)]},
....:                           anchor=(4, 2, posets.ChainPoset(6)))
sage: P.linear_extensions().cardinality()                                      #␣
↪needs sage.combinat sage.modules
361628701868606400
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.mobile import MobilePoset
>>> M = MobilePoset(DiGraph([[Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(6),Integer(7),Integer(8)], [(Integer(1),
↪Integer(0)),(Integer(3),Integer(0)),(Integer(2),Integer(1)),(Integer(2),
↪Integer(3)),(Integer(4),
... Integer(3)), (Integer(5),Integer(4)),(Integer(5),Integer(6)),(Integer(7),
↪Integer(4)),(Integer(7),Integer(8))]]))
>>> M.linear_extensions().cardinality()                                      #␣
↪needs sage.modules
1098

>>> M1 = posets.RibbonPoset(Integer(6), [Integer(1),Integer(3)])
>>> M1.linear_extensions().cardinality()                                     #␣
↪needs sage.modules
61

>>> P = posets.MobilePoset(posets.RibbonPoset(Integer(7), [Integer(1),
↪Integer(3)]),                    # needs sage.combinat sage.modules
...                           {Integer(1): [posets.YoungDiagramPoset([Integer(3),
↪ Integer(2)], dual=True)],
...                            Integer(3): [posets.
↪DoubleTailedDiamond(Integer(6))]},
...                           anchor=(Integer(4), Integer(2), posets.
↪ChainPoset(Integer(6))))
>>> P.linear_extensions().cardinality()                                      #␣
↪needs sage.combinat sage.modules
361628701868606400
```

class sage.combinat.posets.linear_extensions.**LinearExtensionsOfPoset**(*poset*, *facade*)

Bases: `UniqueRepresentation`, `Parent`

The set of all linear extensions of a finite poset.

INPUT:

- `poset` – a poset P of size n
- `facade` – boolean (default: `False`)

See also

- `sage.combinat.posets.posets.FinitePoset.linear_extensions()`

EXAMPLES:

```
sage: elms = [1,2,3,4]
sage: rels = [[1,3],[1,4],[2,3]]
sage: P = Poset((elms, rels), linear_extension=True)
sage: L = P.linear_extensions(); L
The set of all linear extensions of
 Finite poset containing 4 elements with distinguished linear extension
sage: L.cardinality()
5
sage: L.list()                                                          #␣
↪needs sage.modules
[[1, 2, 3, 4], [2, 1, 3, 4], [2, 1, 4, 3], [1, 4, 2, 3], [1, 2, 4, 3]]
sage: L.an_element()
[1, 2, 3, 4]
sage: L.poset()
Finite poset containing 4 elements with distinguished linear extension
```

```
>>> from sage.all import *
>>> elms = [Integer(1),Integer(2),Integer(3),Integer(4)]
>>> rels = [[Integer(1),Integer(3)],[Integer(1),Integer(4)],[Integer(2),
↪Integer(3)]]
>>> P = Poset((elms, rels), linear_extension=True)
>>> L = P.linear_extensions(); L
The set of all linear extensions of
 Finite poset containing 4 elements with distinguished linear extension
>>> L.cardinality()
5
>>> L.list()                                                            #␣
↪needs sage.modules
[[1, 2, 3, 4], [2, 1, 3, 4], [2, 1, 4, 3], [1, 4, 2, 3], [1, 2, 4, 3]]
>>> L.an_element()
[1, 2, 3, 4]
>>> L.poset()
Finite poset containing 4 elements with distinguished linear extension
```

Element

alias of `LinearExtensionOfPoset`

cardinality()

Return the number of linear extensions.

EXAMPLES:

```
sage: N = Poset({0: [2, 3], 1: [3]})
sage: N.linear_extensions().cardinality()
5
```

```
>>> from sage.all import *
>>> N = Poset({Integer(0): [Integer(2), Integer(3)], Integer(1): [Integer(3)]}
↪)
>>> N.linear_extensions().cardinality()
5
```

markov_chain_digraph(*action='promotion'*, *labeling='identity'*)

Return the digraph of the action of generalized promotion or tau on `self`.

INPUT:

- `action` – 'promotion' or 'tau' (default: `'promotion'`)
- `labeling` – 'identity' or 'source' (default: `'identity'`)

Todo

- generalize this feature by accepting a family of operators as input
- move up in some appropriate category

This method creates a graph with vertices being the linear extensions of a given finite poset and an edge from π to π' if $\pi' = \pi\partial_i$ where ∂_i is the promotion operator (see `promotion()`) if `action` is set to `promotion` and τ_i (see `tau()`) if `action` is set to `tau`. The label of the edge is i (resp. π_i) if `labeling` is set to `identity` (resp. `source`).

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension=True)
sage: L = P.linear_extensions()
sage: G = L.markov_chain_digraph(); G
Looped multi-digraph on 5 vertices
sage: G.vertices(sort=True, key=repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
sage: G.edges(sort=True, key=repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 4), ([1, 2, 3, 4], [1, 2, 4, 3], 2), ([1, 2, 3,␣
↪4], [1, 2, 4, 3], 3),
 ([1, 2, 3, 4], [2, 1, 4, 3], 1), ([1, 2, 4, 3], [1, 2, 3, 4], 3), ([1, 2, 4,␣
↪3], [1, 2, 4, 3], 4),
 ([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4, 3], [2, 1, 3, 4], 1), ([1, 4, 2,␣
↪3], [1, 2, 3, 4], 1),
 ([1, 4, 2, 3], [1, 2, 3, 4], 2), ([1, 4, 2, 3], [1, 4, 2, 3], 3), ([1, 4, 2,␣
↪3], [1, 4, 2, 3], 4),
 ([2, 1, 3, 4], [1, 2, 4, 3], 1), ([2, 1, 3, 4], [2, 1, 3, 4], 4), ([2, 1, 3,␣
↪4], [2, 1, 4, 3], 2),
 ([2, 1, 3, 4], [2, 1, 4, 3], 3), ([2, 1, 4, 3], [1, 4, 2, 3], 1), ([2, 1, 4,␣
↪3], [2, 1, 3, 4], 2),
 ([2, 1, 4, 3], [2, 1, 3, 4], 3), ([2, 1, 4, 3], [2, 1, 4, 3], 4)]
```

(continues on next page)

(continued from previous page)

```
sage: G = L.markov_chain_digraph(labeling='source')
sage: G.vertices(sort=True, key=repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
sage: G.edges(sort=True, key=repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 4), ([1, 2, 3, 4], [1, 2, 4, 3], 2), ([1, 2, 3,
→4], [1, 2, 4, 3], 3),
 ([1, 2, 3, 4], [2, 1, 4, 3], 1), ([1, 2, 4, 3], [1, 2, 3, 4], 4), ([1, 2, 4,
→3], [1, 2, 4, 3], 3),
 ([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4, 3], [2, 1, 3, 4], 1), ([1, 4, 2,
→3], [1, 2, 3, 4], 1),
 ([1, 4, 2, 3], [1, 2, 3, 4], 4), ([1, 4, 2, 3], [1, 4, 2, 3], 2), ([1, 4, 2,
→3], [1, 4, 2, 3], 3),
 ([2, 1, 3, 4], [1, 2, 4, 3], 2), ([2, 1, 3, 4], [2, 1, 3, 4], 4), ([2, 1, 3,
→4], [2, 1, 4, 3], 1),
 ([2, 1, 3, 4], [2, 1, 4, 3], 3), ([2, 1, 4, 3], [1, 4, 2, 3], 2), ([2, 1, 4,
→3], [2, 1, 3, 4], 1),
 ([2, 1, 4, 3], [2, 1, 3, 4], 4), ([2, 1, 4, 3], [2, 1, 4, 3], 3)]
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)], [[Integer(1),
→Integer(3)],[Integer(1),Integer(4)],[Integer(2),Integer(3)]]), linear_
→extension=True)
>>> L = P.linear_extensions()
>>> G = L.markov_chain_digraph(); G
Looped multi-digraph on 5 vertices
>>> G.vertices(sort=True, key=repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
>>> G.edges(sort=True, key=repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 4), ([1, 2, 3, 4], [1, 2, 4, 3], 2), ([1, 2, 3,
→4], [1, 2, 4, 3], 3),
 ([1, 2, 3, 4], [2, 1, 4, 3], 1), ([1, 2, 4, 3], [1, 2, 3, 4], 3), ([1, 2, 4,
→3], [1, 2, 4, 3], 4),
 ([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4, 3], [2, 1, 3, 4], 1), ([1, 4, 2,
→3], [1, 2, 3, 4], 1),
 ([1, 4, 2, 3], [1, 2, 3, 4], 2), ([1, 4, 2, 3], [1, 4, 2, 3], 3), ([1, 4, 2,
→3], [1, 4, 2, 3], 4),
 ([2, 1, 3, 4], [1, 2, 4, 3], 1), ([2, 1, 3, 4], [2, 1, 3, 4], 4), ([2, 1, 3,
→4], [2, 1, 4, 3], 2),
 ([2, 1, 3, 4], [2, 1, 4, 3], 3), ([2, 1, 4, 3], [1, 4, 2, 3], 1), ([2, 1, 4,
→3], [2, 1, 3, 4], 2),
 ([2, 1, 4, 3], [2, 1, 3, 4], 3), ([2, 1, 4, 3], [2, 1, 4, 3], 4)]

>>> G = L.markov_chain_digraph(labeling='source')
>>> G.vertices(sort=True, key=repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
>>> G.edges(sort=True, key=repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 4), ([1, 2, 3, 4], [1, 2, 4, 3], 2), ([1, 2, 3,
→4], [1, 2, 4, 3], 3),
 ([1, 2, 3, 4], [2, 1, 4, 3], 1), ([1, 2, 4, 3], [1, 2, 3, 4], 4), ([1, 2, 4,
→3], [1, 2, 4, 3], 3),
 ([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4, 3], [2, 1, 3, 4], 1), ([1, 4, 2,
```

(continues on next page)

(continued from previous page)

```
↪3], [1, 2, 3, 4], 1),
 ([1, 4, 2, 3], [1, 2, 3, 4], 4), ([1, 4, 2, 3], [1, 4, 2, 3], 2), ([1, 4, 2,␣
↪3], [1, 4, 2, 3], 3),
 ([2, 1, 3, 4], [1, 2, 4, 3], 2), ([2, 1, 3, 4], [2, 1, 3, 4], 4), ([2, 1, 3,␣
↪4], [2, 1, 4, 3], 1),
 ([2, 1, 3, 4], [2, 1, 4, 3], 3), ([2, 1, 4, 3], [1, 4, 2, 3], 2), ([2, 1, 4,␣
↪3], [2, 1, 3, 4], 1),
 ([2, 1, 4, 3], [2, 1, 3, 4], 4), ([2, 1, 4, 3], [2, 1, 4, 3], 3)]
```

The edges of the graph are by default colored using blue for edge 1, red for edge 2, green for edge 3, and yellow for edge 4:

```
sage: view(G) # optional - dot2tex graphviz, not tested (opens external␣
↪window)
```

```
>>> from sage.all import *
>>> view(G) # optional - dot2tex graphviz, not tested (opens external window)
```

Alternatively, one may get the graph of the action of the `tau` operator:

```
sage: G = L.markov_chain_digraph(action='tau'); G
Looped multi-digraph on 5 vertices
sage: G.vertices(sort=True, key=repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
sage: G.edges(sort=True, key=repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 2), ([1, 2, 3, 4], [1, 2, 4, 3], 3), ([1, 2, 3,␣
↪4], [2, 1, 3, 4], 1),
 ([1, 2, 4, 3], [1, 2, 3, 4], 3), ([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4,␣
↪3], [2, 1, 4, 3], 1),
 ([1, 4, 2, 3], [1, 2, 4, 3], 2), ([1, 4, 2, 3], [1, 4, 2, 3], 1), ([1, 4, 2,␣
↪3], [1, 4, 2, 3], 3),
 ([2, 1, 3, 4], [1, 2, 3, 4], 1), ([2, 1, 3, 4], [2, 1, 3, 4], 2), ([2, 1, 3,␣
↪4], [2, 1, 4, 3], 3),
 ([2, 1, 4, 3], [1, 2, 4, 3], 1), ([2, 1, 4, 3], [2, 1, 3, 4], 3), ([2, 1, 4,␣
↪3], [2, 1, 4, 3], 2)]
sage: view(G) # optional - dot2tex graphviz, not tested (opens external␣
↪window)
```

```
>>> from sage.all import *
>>> G = L.markov_chain_digraph(action='tau'); G
Looped multi-digraph on 5 vertices
>>> G.vertices(sort=True, key=repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
>>> G.edges(sort=True, key=repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 2), ([1, 2, 3, 4], [1, 2, 4, 3], 3), ([1, 2, 3,␣
↪4], [2, 1, 3, 4], 1),
 ([1, 2, 4, 3], [1, 2, 3, 4], 3), ([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4,␣
↪3], [2, 1, 4, 3], 1),
 ([1, 4, 2, 3], [1, 2, 4, 3], 2), ([1, 4, 2, 3], [1, 4, 2, 3], 1), ([1, 4, 2,␣
↪3], [1, 4, 2, 3], 3),
 ([2, 1, 3, 4], [1, 2, 3, 4], 1), ([2, 1, 3, 4], [2, 1, 3, 4], 2), ([2, 1, 3,␣
↪4], [2, 1, 4, 3], 3),
```

(continues on next page)

(continued from previous page)

```
([2, 1, 4, 3], [1, 2, 4, 3], 1), ([2, 1, 4, 3], [2, 1, 3, 4], 3), ([2, 1, 4,␣
↪3], [2, 1, 4, 3], 2)]
>>> view(G) # optional - dot2tex graphviz, not tested (opens external window)
```

See also

markov_chain_transition_matrix(), promotion(), tau()

markov_chain_transition_matrix(*action='promotion'*, *labeling='identity'*)

Return the transition matrix of the Markov chain for the action of generalized promotion or tau on `self`.

INPUT:

- `action` – `'promotion'` or `'tau'` (default: `'promotion'`)
- `labeling` – `'identity'` or `'source'` (default: `'identity'`)

This method yields the transition matrix of the Markov chain defined by the action of the generalized promotion operator ∂_i (resp. τ_i) on the set of linear extensions of a finite poset. Here the transition from the linear extension π to π', where $\pi' = \pi\partial_i$ (resp. $\pi' = \pi\tau_i$) is counted with weight x_i (resp. x_{π_i} if `labeling` is set to `source`).

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension=True)
sage: L = P.linear_extensions()
sage: L.markov_chain_transition_matrix()                                    #␣
↪needs sage.modules
[-x0 - x1 - x2            x2       x0 + x1             0             0]
[      x1 + x2 -x0 - x1 - x2             0            x0             0]
[            0            x1      -x0 - x1             0            x0]
[            0            x0             0 -x0 - x1 - x2       x1 + x2]
[           x0             0             0       x1 + x2 -x0 - x1 - x2]

sage: L.markov_chain_transition_matrix(labeling='source')                   #␣
↪needs sage.modules
[-x0 - x1 - x2            x3       x0 + x3             0             0]
[      x1 + x2 -x0 - x1 - x3             0            x1             0]
[            0            x1      -x0 - x3             0            x1]
[            0            x0             0 -x0 - x1 - x2       x0 + x3]
[           x0             0             0       x0 + x2 -x0 - x1 - x3]

sage: L.markov_chain_transition_matrix(action='tau')                        #␣
↪needs sage.modules
[     -x0 - x2            x2             0            x0             0]
[           x2 -x0 - x1 - x2            x1             0            x0]
[            0            x1           -x1             0             0]
[           x0             0             0      -x0 - x2            x2]
[            0            x0             0            x2      -x0 - x2]

sage: L.markov_chain_transition_matrix(action='tau', labeling='source')      #␣
↪needs sage.modules
[     -x0 - x2            x3             0            x1             0]
```

(continues on next page)

(continued from previous page)

```
[            x2 -x0 - x1 - x3             x3             0             x1]
[             0             x1           -x3             0              0]
[            x0              0             0       -x1 - x2            x3]
[             0             x0             0            x2       -x1 - x3]
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)], [[Integer(1),
↪Integer(3)],[Integer(1),Integer(4)],[Integer(2),Integer(3)]]), linear_
↪extension=True)
>>> L = P.linear_extensions()
>>> L.markov_chain_transition_matrix()                                      #␣
↪needs sage.modules
[-x0 - x1 - x2             x2        x0 + x1             0              0]
[      x1 + x2 -x0 - x1 - x2              0             x0              0]
[             0             x1      -x0 - x1             0             x0]
[             0             x0             0 -x0 - x1 - x2        x1 + x2]
[            x0              0             0        x1 + x2 -x0 - x1 - x2]

>>> L.markov_chain_transition_matrix(labeling='source')                     #␣
↪needs sage.modules
[-x0 - x1 - x2             x3        x0 + x3             0              0]
[      x1 + x2 -x0 - x1 - x3              0             x1              0]
[             0             x1      -x0 - x3             0             x1]
[             0             x0             0 -x0 - x1 - x2        x0 + x3]
[            x0              0             0        x0 + x2 -x0 - x1 - x3]

>>> L.markov_chain_transition_matrix(action='tau')                          #␣
↪needs sage.modules
[     -x0 - x2             x2             0             x0              0]
[            x2 -x0 - x1 - x2             x1             0             x0]
[             0             x1           -x1             0              0]
[            x0              0             0       -x0 - x2            x2]
[             0             x0             0            x2       -x0 - x2]

>>> L.markov_chain_transition_matrix(action='tau', labeling='source')       #␣
↪needs sage.modules
[     -x0 - x2             x3             0             x1              0]
[            x2 -x0 - x1 - x3             x3             0             x1]
[             0             x1           -x3             0              0]
[            x0              0             0       -x1 - x2            x3]
[             0             x0             0            x2       -x1 - x3]
```

See also

markov_chain_digraph(), promotion(), tau()

poset()

Return the underlying original poset.

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,2],[2,3],[1,4]]))
sage: L = P.linear_extensions()
sage: L.poset()
Finite poset containing 4 elements
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)], [[Integer(1),
↪Integer(2)],[Integer(2),Integer(3)],[Integer(1),Integer(4)]]))
>>> L = P.linear_extensions()
>>> L.poset()
Finite poset containing 4 elements
```

class sage.combinat.posets.linear_extensions.**LinearExtensionsOfPosetWithHooks**(*poset*, *facade*)

Bases: *LinearExtensionsOfPoset*

Linear extensions such that the poset has well-defined hook lengths (i.e., d-complete).

cardinality()

Count the number of linear extensions using a hook-length formula.

EXAMPLES:

```
sage: from sage.combinat.posets.poset_examples import Posets
sage: P = Posets.YoungDiagramPoset(Partition([3,2]), dual=True)             #␣
↪needs sage.combinat sage.modules
sage: P.linear_extensions().cardinality()                                   #␣
↪needs sage.combinat sage.modules
5
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.poset_examples import Posets
>>> P = Posets.YoungDiagramPoset(Partition([Integer(3),Integer(2)]),␣
↪dual=True)              # needs sage.combinat sage.modules
>>> P.linear_extensions().cardinality()                                   #␣
↪needs sage.combinat sage.modules
5
```

5.1.184 Möbius Algebras

class sage.combinat.posets.moebius_algebra.**BasisAbstract**(*R*, *basis_keys=None*, *element_class=None*, *category=None*, *prefix=None*, *names=None*, ***kwds*)

Bases: *CombinatorialFreeModule*, BindableClass

Abstract base class for a basis.

class sage.combinat.posets.moebius_algebra.**MoebiusAlgebra**(*R*, *L*)

Bases: Parent, UniqueRepresentation

The Möbius algebra of a lattice.

Let L be a lattice. The *Möbius algebra* M_L was originally constructed by Solomon [Solomon67] and has a natural basis $\{E_x \mid x \in L\}$ with multiplication given by $E_x \cdot E_y = E_{x \vee y}$. Moreover this has a basis given by orthogonal idempotents $\{I_x \mid x \in L\}$ (so $I_x I_y = \delta_{xy} I_x$ where δ is the Kronecker delta) related to the natural basis by

$$I_x = \sum_{x \leq y} \mu_L(x, y) E_y,$$

where μ_L is the Möbius function of L.

Note

We use the join $\vee$ for our multiplication, whereas [Greene73] and [Etienne98] define the Möbius algebra using the meet $\wedge$. This is done for compatibility with `QuantumMoebiusAlgebra`.

REFERENCES:

class E(*M*, *prefix='E'*)

Bases: `BasisAbstract`

The natural basis of a Möbius algebra.

Let E_x and E_y be basis elements of M_L for some lattice L. Multiplication is given by $E_x E_y = E_{x \vee y}$.

one()

Return the element `1` of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.moebius_algebra(QQ).E()
sage: E.one()
E[0]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> E = L.moebius_algebra(QQ).E()
>>> E.one()
E[0]
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by `x` and `y`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.moebius_algebra(QQ).E()
sage: E.product_on_basis(5, 14)
E[15]
sage: E.product_on_basis(2, 8)
E[10]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> E = L.moebius_algebra(QQ).E()
>>> E.product_on_basis(Integer(5), Integer(14))
E[15]
>>> E.product_on_basis(Integer(2), Integer(8))
E[10]
```

class I(*M*, *prefix='I'*)

Bases: *BasisAbstract*

The (orthogonal) idempotent basis of a Möbius algebra.

Let I_x and I_y be basis elements of M_L for some lattice L. Multiplication is given by $I_x I_y = \delta_{xy} I_x$ where δ_{xy} is the Kronecker delta.

one()

Return the element 1 of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: I = L.moebius_algebra(QQ).I()
sage: I.one()
I[0] + I[1] + I[2] + I[3] + I[4] + I[5] + I[6] + I[7] + I[8]
 + I[9] + I[10] + I[11] + I[12] + I[13] + I[14] + I[15]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> I = L.moebius_algebra(QQ).I()
>>> I.one()
I[0] + I[1] + I[2] + I[3] + I[4] + I[5] + I[6] + I[7] + I[8]
 + I[9] + I[10] + I[11] + I[12] + I[13] + I[14] + I[15]
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by x and y.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: I = L.moebius_algebra(QQ).I()
sage: I.product_on_basis(5, 14)
0
sage: I.product_on_basis(2, 2)
I[2]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> I = L.moebius_algebra(QQ).I()
>>> I.product_on_basis(Integer(5), Integer(14))
0
>>> I.product_on_basis(Integer(2), Integer(2))
I[2]
```

a_realization()

Return a particular realization of self (the B-basis).

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.moebius_algebra(QQ)
sage: M.a_realization()
Moebius algebra of Finite lattice containing 16 elements
 over Rational Field in the natural basis
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> M = L.moebius_algebra(QQ)
>>> M.a_realization()
Moebius algebra of Finite lattice containing 16 elements
 over Rational Field in the natural basis
```

idempotent

alias of `I`

lattice()

Return the defining lattice of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.moebius_algebra(QQ)
sage: M.lattice()
Finite lattice containing 16 elements
sage: M.lattice() == L
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> M = L.moebius_algebra(QQ)
>>> M.lattice()
Finite lattice containing 16 elements
>>> M.lattice() == L
True
```

natural

alias of `E`

class sage.combinat.posets.moebius_algebra.**MoebiusAlgebraBases**(*parent_with_realization*)

Bases: `Category_realization_of_parent`

The category of bases of a Möbius algebra.

INPUT:

- `base` – a Möbius algebra

class ElementMethods

Bases: `object`

class ParentMethods

Bases: `object`

one()

Return the element `1` of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: C = L.quantum_moebius_algebra().C()
sage: all(C.one() * b == b for b in C.basis())
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> C = L.quantum_moebius_algebra().C()
>>> all(C.one() * b == b for b in C.basis())
True
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by x and y.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: C = L.quantum_moebius_algebra().C()
sage: C.product_on_basis(5, 14)
q^3*C[15]
sage: C.product_on_basis(2, 8)
q^4*C[10]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> C = L.quantum_moebius_algebra().C()
>>> C.product_on_basis(Integer(5), Integer(14))
q^3*C[15]
>>> C.product_on_basis(Integer(2), Integer(8))
q^4*C[10]
```

super_categories()

The super categories of self.

EXAMPLES:

```
sage: from sage.combinat.posets.moebius_algebra import MoebiusAlgebraBases
sage: M = posets.BooleanLattice(4).moebius_algebra(QQ)
sage: bases = MoebiusAlgebraBases(M)
sage: bases.super_categories()
[Category of finite dimensional commutative algebras with basis over Rational␣
↪Field,
 Category of realizations of Moebius algebra of Finite lattice
    containing 16 elements over Rational Field]
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.moebius_algebra import MoebiusAlgebraBases
>>> M = posets.BooleanLattice(Integer(4)).moebius_algebra(QQ)
>>> bases = MoebiusAlgebraBases(M)
>>> bases.super_categories()
[Category of finite dimensional commutative algebras with basis over Rational␣
↪Field,
 Category of realizations of Moebius algebra of Finite lattice
    containing 16 elements over Rational Field]
```

class sage.combinat.posets.moebius_algebra.**QuantumMoebiusAlgebra**(*L*, *q=None*)

Bases: Parent, UniqueRepresentation

The quantum Möbius algebra of a lattice.

Let L be a lattice, and we define the *quantum Möbius algebra* $M_L(q)$ as the algebra with basis $\{E_x \mid x \in L\}$ with multiplication given by

$$E_x E_y = \sum_{z \geq a \geq x \vee y} \mu_L(a, z) q^{\operatorname{crk} a} E_z,$$

where μ_L is the Möbius function of L and crk is the corank function (i.e., $\operatorname{crk} a = \operatorname{rank} L - \operatorname{rank}$ a). At $q = 1$, this reduces to the multiplication formula originally given by Solomon.

class C(*M*, *prefix='C'*)

Bases: `BasisAbstract`

The characteristic basis of a quantum Möbius algebra.

The characteristic basis $\{C_x \mid x \in L\}$ of M_L for some lattice L is defined by

$$C_x = \sum_{a \geq x} P(F^x; q) E_a,$$

where $F^x = \{y \in L \mid y \geq x\}$ is the principal order filter of x and $P(F^x; q)$ is the characteristic polynomial of the (sub)poset F^x.

class E(*M*, *prefix='E'*)

Bases: `BasisAbstract`

The natural basis of a quantum Möbius algebra.

Let E_x and E_y be basis elements of M_L for some lattice L. Multiplication is given by

$$E_x E_y = \sum_{z \geq a \geq x \vee y} \mu_L(a, z) q^{\operatorname{crk} a} E_z,$$

where μ_L is the Möbius function of L and crk is the corank function (i.e., $\operatorname{crk} a = \operatorname{rank} L - \operatorname{rank}$ a).

one()

Return the element `1` of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.quantum_moebius_algebra().E()
sage: all(E.one() * b == b for b in E.basis())
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> E = L.quantum_moebius_algebra().E()
>>> all(E.one() * b == b for b in E.basis())
True
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by `x` and `y`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.quantum_moebius_algebra().E()
sage: E.product_on_basis(5, 14)
```

(continues on next page)

(continued from previous page)

```
E[15]
sage: E.product_on_basis(2, 8)
q^2*E[10] + (q-q^2)*E[11] + (q-q^2)*E[14] + (1-2*q+q^2)*E[15]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> E = L.quantum_moebius_algebra().E()
>>> E.product_on_basis(Integer(5), Integer(14))
E[15]
>>> E.product_on_basis(Integer(2), Integer(8))
q^2*E[10] + (q-q^2)*E[11] + (q-q^2)*E[14] + (1-2*q+q^2)*E[15]
```

class KL(*M*, *prefix='KL'*)

Bases: `BasisAbstract`

The Kazhdan-Lusztig basis of a quantum Möbius algebra.

The Kazhdan-Lusztig basis $\{B_x \mid x \in L\}$ of M_L for some lattice L is defined by

$$B_x = \sum_{y \geq x} P_{x,y}(q) E_a,$$

where $P_{x,y}(q)$ is the Kazhdan-Lusztig polynomial of L, following the definition given in [EPW14].

EXAMPLES:

We construct some examples of Proposition 4.5 of [EPW14]:

```
sage: M = posets.BooleanLattice(4).quantum_moebius_algebra()
sage: KL = M.KL()
sage: KL[4] * KL[5]
(q^2+q^3)*KL[5] + (q+2*q^2+q^3)*KL[7] + (q+2*q^2+q^3)*KL[13]
 + (1+3*q+3*q^2+q^3)*KL[15]
sage: KL[4] * KL[15]
(1+3*q+3*q^2+q^3)*KL[15]
sage: KL[4] * KL[10]
(q+3*q^2+3*q^3+q^4)*KL[14] + (1+4*q+6*q^2+4*q^3+q^4)*KL[15]
```

```
>>> from sage.all import *
>>> M = posets.BooleanLattice(Integer(4)).quantum_moebius_algebra()
>>> KL = M.KL()
>>> KL[Integer(4)] * KL[Integer(5)]
(q^2+q^3)*KL[5] + (q+2*q^2+q^3)*KL[7] + (q+2*q^2+q^3)*KL[13]
 + (1+3*q+3*q^2+q^3)*KL[15]
>>> KL[Integer(4)] * KL[Integer(15)]
(1+3*q+3*q^2+q^3)*KL[15]
>>> KL[Integer(4)] * KL[Integer(10)]
(q+3*q^2+3*q^3+q^4)*KL[14] + (1+4*q+6*q^2+4*q^3+q^4)*KL[15]
```

a_realization()

Return a particular realization of `self` (the B-basis).

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.quantum_moebius_algebra()
sage: M.a_realization()
Quantum Moebius algebra of Finite lattice containing 16 elements
 with q=q over Univariate Laurent Polynomial Ring in q
 over Integer Ring in the natural basis
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> M = L.quantum_moebius_algebra()
>>> M.a_realization()
Quantum Moebius algebra of Finite lattice containing 16 elements
 with q=q over Univariate Laurent Polynomial Ring in q
 over Integer Ring in the natural basis
```

characteristic_basis

alias of `C`

kazhdan_lusztig

alias of `KL`

lattice()

Return the defining lattice of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.quantum_moebius_algebra()
sage: M.lattice()
Finite lattice containing 16 elements
sage: M.lattice() == L
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> M = L.quantum_moebius_algebra()
>>> M.lattice()
Finite lattice containing 16 elements
>>> M.lattice() == L
True
```

natural

alias of `E`

5.1.185 Catalog of posets and lattices

Some common posets can be accessed through the `posets.<tab>` object:

```
sage: posets.PentagonPoset()
Finite lattice containing 5 elements
```

```
>>> from sage.all import *
>>> posets.PentagonPoset()
Finite lattice containing 5 elements
```

Moreover, the set of all posets of order n is represented by `Posets(n)`:

```
sage: Posets(5)
Posets containing 5 elements
```

```
>>> from sage.all import *
>>> Posets(Integer(5))
Posets containing 5 elements
```

The infinite set of all posets can be used to find minimal examples:

```
sage: for P in Posets():
....:     if not P.is_series_parallel():
....:         break
sage: P
Finite poset containing 4 elements
```

```
>>> from sage.all import *
>>> for P in Posets():
...     if not P.is_series_parallel():
...         break
>>> P
Finite poset containing 4 elements
```

Catalog of common posets:

`AntichainPoset()`	Return an antichain on n elements.
`BooleanLattice()`	Return the Boolean lattice on 2^n elements.
`ChainPoset()`	Return a chain on n elements.
`Crown()`	Return the crown poset on $2n$ elements.
`DexterSemilattice()`	Return the Dexter semilattice.
`DiamondPoset()`	Return the lattice of rank two on n elements.
`DivisorLattice()`	Return the divisor lattice of an integer.
`DoubleTailedDiamond()`	Return the double tailed diamond poset on $2n + 2$ elements.
`IntegerCompositions()`	Return the poset of integer compositions of n.
`IntegerPartitions()`	Return the poset of integer partitions of `n`.
`IntegerPartitionsDominanceOrder()`	Return the lattice of integer partitions on the integer n ordered by dominance.
`MobilePoset()`	Return the mobile poset formed by the $ribbon$ with $hangers$ below and an $anchor$ above.
`NoncrossingPartitions()`	Return the poset of noncrossing partitions of a finite Coxeter group `W`.
`PentagonPoset()`	Return the Pentagon poset.
`PermutationPattern()`	Return the Permutation pattern poset.
`PermutationPatternInterval()`	Return an interval in the Permutation pattern poset.
`PermutationPatternOccurrenceInterval()`	Return the occurrence poset for a pair of comparable elements in the Permutation pattern poset.
`PowerPoset()`	Return a power poset.

continues on next page

Table 6 – continued from previous page

`ProductOfChains()`	Return a product of chain posets.
`RandomLattice()`	Return a random lattice on n elements.
`RandomPoset()`	Return a random poset on n elements.
`RibbonPoset()`	Return a ribbon on n elements with descents at $descents$.
`RestrictedIntegerPartitions()`	Return the poset of integer partitions of n, ordered by restricted refinement.
`SetPartitions()`	Return the poset of set partitions of the set $\{1, \dots, n\}$.
`ShardPoset()`	Return the shard intersection order.
`SSTPoset()`	Return the poset on semistandard tableaux of shape s and largest entry f that is ordered by componentwise comparison.
`StandardExample()`	Return the standard example of a poset with dimension n.
`SymmetricGroupAbsoluteOrderPoset()`	The poset of permutations with respect to absolute order.
`SymmetricGroupBruhatIntervalPoset()`	The poset of permutations with respect to Bruhat order.
`SymmetricGroupBruhatOrderPoset()`	The poset of permutations with respect to Bruhat order.
`SymmetricGroupWeakOrderPoset()`	The poset of permutations of $\{1, 2, \dots, n\}$ with respect to the weak order.
`TamariLattice()`	Return the Tamari lattice.
`TetrahedralPoset()`	Return the Tetrahedral poset with $n - 1$ layers based on the input colors.
`UpDownPoset()`	Return the up-down poset on n elements.
`YoungDiagramPoset()`	Return the poset of cells in the Young diagram of a partition.
`YoungsLattice()`	Return Young's Lattice up to rank n.
`YoungsLatticePrincipalOrderIdeal()`	Return the principal order ideal of the partition lam in Young's Lattice.
`YoungFibonacci()`	Return the Young-Fibonacci lattice up to rank n.

Other available posets:

`face_lattice()`	Return the face lattice of a polyhedron.
`face_lattice()`	Return the face lattice of a combinatorial polyhedron.

Constructions

class sage.combinat.posets.poset_examples.**Posets**

Bases: `object`

A collection of posets and lattices.

EXAMPLES:

```
sage: posets.BooleanLattice(3)
Finite lattice containing 8 elements
sage: posets.ChainPoset(3)
Finite lattice containing 3 elements
sage: posets.RandomPoset(17,.15)
Finite poset containing 17 elements
```

```
>>> from sage.all import *
>>> posets.BooleanLattice(Integer(3))
```

(continues on next page)

(continued from previous page)

```
Finite lattice containing 8 elements
>>> posets.ChainPoset(Integer(3))
Finite lattice containing 3 elements
>>> posets.RandomPoset(Integer(17),RealNumber('.15'))
Finite poset containing 17 elements
```

The category of all posets:

```
sage: Posets()
Category of posets
```

```
>>> from sage.all import *
>>> Posets()
Category of posets
```

The enumerated set of all posets on 3 elements, up to an isomorphism:

```
sage: Posets(3)
Posets containing 3 elements
```

```
>>> from sage.all import *
>>> Posets(Integer(3))
Posets containing 3 elements
```

See also:

`Posets`, `FinitePosets`, *`Poset()`*

static AntichainPoset(*n*, *facade=None*)

Return an antichain (a poset with no comparable elements) containing n elements.

INPUT:

- `n` – integer; number of elements
- `facade` – boolean; whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the *`Poset()`* constructor

EXAMPLES:

```
sage: A = posets.AntichainPoset(6); A
Finite poset containing 6 elements
```

```
>>> from sage.all import *
>>> A = posets.AntichainPoset(Integer(6)); A
Finite poset containing 6 elements
```

static BooleanLattice(*n*, *facade=None*, *use_subsets=False*)

Return the Boolean lattice containing 2^n elements.

- `n` – integer; number of elements will be 2^n
- `facade` – boolean; whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the *`Poset()`* constructor

- `use_subsets` – boolean (default: `False`); if `True`, then label the elements by subsets of $\{1, 2, \ldots, n\}$; otherwise label the elements by $0, 1, 2, \ldots, 2^n - 1$

EXAMPLES:

```
sage: posets.BooleanLattice(5)
Finite lattice containing 32 elements

sage: sorted(posets.BooleanLattice(2))
[0, 1, 2, 3]
sage: sorted(posets.BooleanLattice(2, use_subsets=True), key=list)
[{}, {1}, {1, 2}, {2}]
```

```
>>> from sage.all import *
>>> posets.BooleanLattice(Integer(5))
Finite lattice containing 32 elements

>>> sorted(posets.BooleanLattice(Integer(2)))
[0, 1, 2, 3]
>>> sorted(posets.BooleanLattice(Integer(2), use_subsets=True), key=list)
[{}, {1}, {1, 2}, {2}]
```

static ChainPoset(*n*, *facade=None*)

Return a chain (a totally ordered poset) containing n elements.

- `n` – integer; number of elements
- `facade` – boolean; whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the `Poset()` constructor

EXAMPLES:

```
sage: C = posets.ChainPoset(6); C
Finite lattice containing 6 elements
sage: C.linear_extension()
[0, 1, 2, 3, 4, 5]
```

```
>>> from sage.all import *
>>> C = posets.ChainPoset(Integer(6)); C
Finite lattice containing 6 elements
>>> C.linear_extension()
[0, 1, 2, 3, 4, 5]
```

static CoxeterGroupAbsoluteOrderPoset(*W*, *use_reduced_words=True*)

Return the poset of elements of a Coxeter group with respect to absolute order.

INPUT:

- `W` – a Coxeter group
- `use_reduced_words` – boolean (default: `True`); if `True`, then the elements are labeled by their lexicographically minimal reduced word

EXAMPLES:

```
sage: W = CoxeterGroup(['B', 3])                                        #␣
↪needs sage.groups
```

(continues on next page)

(continued from previous page)

```
sage: posets.CoxeterGroupAbsoluteOrderPoset(W)                              #␣
↪needs sage.groups
Finite poset containing 48 elements

sage: W = WeylGroup(['B', 2], prefix='s')                                   #␣
↪needs sage.groups
sage: posets.CoxeterGroupAbsoluteOrderPoset(W, False)                       #␣
↪needs sage.groups
Finite poset containing 8 elements
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['B', Integer(3)])                                     ␣
↪       # needs sage.groups
>>> posets.CoxeterGroupAbsoluteOrderPoset(W)                              #␣
↪needs sage.groups
Finite poset containing 48 elements

>>> W = WeylGroup(['B', Integer(2)], prefix='s')                            ␣
↪       # needs sage.groups
>>> posets.CoxeterGroupAbsoluteOrderPoset(W, False)                       #␣
↪needs sage.groups
Finite poset containing 8 elements
```

static Crown(*n*, *facade=None*)

Return the crown poset of $2n$ elements.

In this poset every element i for $0 \leq i \leq n-1$ is covered by elements $i+n$ and $i+n+1$, except that $n-1$ is covered by n and $n+1$.

INPUT:

- `n` – number of elements, an integer at least 2
- `facade` – boolean; whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the `Poset()` constructor

EXAMPLES:

```
sage: posets.Crown(3)
Finite poset containing 6 elements
```

```
>>> from sage.all import *
>>> posets.Crown(Integer(3))
Finite poset containing 6 elements
```

static DexterSemilattice(*n*)

Return the n-th Dexter meet-semilattice.

INPUT:

- `n` – nonnegative integer (the index)

OUTPUT: a finite meet-semilattice

The elements of the semilattice are `Dyck paths` in the $(n+1 \times n)$-rectangle.

EXAMPLES:

```
sage: posets.DexterSemilattice(3)
Finite meet-semilattice containing 5 elements

sage: P = posets.DexterSemilattice(4); P
Finite meet-semilattice containing 14 elements
sage: len(P.maximal_chains())
15
sage: len(P.maximal_elements())
4
sage: P.chain_polynomial()
q^5 + 19*q^4 + 47*q^3 + 42*q^2 + 14*q + 1
```

```
>>> from sage.all import *
>>> posets.DexterSemilattice(Integer(3))
Finite meet-semilattice containing 5 elements

>>> P = posets.DexterSemilattice(Integer(4)); P
Finite meet-semilattice containing 14 elements
>>> len(P.maximal_chains())
15
>>> len(P.maximal_elements())
4
>>> P.chain_polynomial()
q^5 + 19*q^4 + 47*q^3 + 42*q^2 + 14*q + 1
```

REFERENCES:

- [Cha18]

static DiamondPoset(*n*, *facade=None*)

Return the lattice of rank two containing `n` elements.

INPUT:

- `n` – number of elements, an integer at least 3
- `facade` – boolean; whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the `Poset()` constructor

EXAMPLES:

```
sage: posets.DiamondPoset(7)
Finite lattice containing 7 elements
```

```
>>> from sage.all import *
>>> posets.DiamondPoset(Integer(7))
Finite lattice containing 7 elements
```

static DivisorLattice(*n*, *facade=None*)

Return the divisor lattice of an integer.

Elements of the lattice are divisors of n and $x < y$ in the lattice if x divides y.

INPUT:

- `n` – integer

- facade – boolean; whether to make the returned poset a facade poset (see sage.categories.facade_sets); the default behaviour is the same as the default behaviour of the Poset() constructor

EXAMPLES:

```
sage: P = posets.DivisorLattice(12)
sage: sorted(P.cover_relations())
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]

sage: P = posets.DivisorLattice(10, facade=False)
sage: P(2) < P(5)
False
```

```
>>> from sage.all import *
>>> P = posets.DivisorLattice(Integer(12))
>>> sorted(P.cover_relations())
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]

>>> P = posets.DivisorLattice(Integer(10), facade=False)
>>> P(Integer(2)) < P(Integer(5))
False
```

static DoubleTailedDiamond(*n*)

Return a double-tailed diamond of $2n + 2$ elements.

INPUT:

- n – positive integer

EXAMPLES:

```
sage: P = posets.DoubleTailedDiamond(2); P
Finite d-complete poset containing 6 elements
sage: P.cover_relations()
[[1, 2], [2, 3], [2, 4], [3, 5], [4, 5], [5, 6]]
```

```
>>> from sage.all import *
>>> P = posets.DoubleTailedDiamond(Integer(2)); P
Finite d-complete poset containing 6 elements
>>> P.cover_relations()
[[1, 2], [2, 3], [2, 4], [3, 5], [4, 5], [5, 6]]
```

static IntegerCompositions(*n*)

Return the poset of integer compositions of the integer n.

A composition of a positive integer n is a list of positive integers that sum to n. The order is reverse refinement: $[p_1, p_2, ..., p_l] < [q_1, q_2, ..., q_m]$ if q consists of an integer composition of p_1, followed by an integer composition of p_2, and so on.

EXAMPLES:

```
sage: P = posets.IntegerCompositions(7); P
Finite poset containing 64 elements
sage: len(P.cover_relations())
192
```

```
>>> from sage.all import *
>>> P = posets.IntegerCompositions(Integer(7)); P
Finite poset containing 64 elements
>>> len(P.cover_relations())
192
```

static IntegerPartitions(*n*)

Return the poset of integer partitions on the integer `n`.

A partition of a positive integer n is a non-increasing list of positive integers that sum to n. If p and q are integer partitions of n, then p covers q if and only if q is obtained from p by joining two parts of p (and sorting, if necessary).

EXAMPLES:

```
sage: P = posets.IntegerPartitions(7); P
Finite poset containing 15 elements
sage: len(P.cover_relations())
28
```

```
>>> from sage.all import *
>>> P = posets.IntegerPartitions(Integer(7)); P
Finite poset containing 15 elements
>>> len(P.cover_relations())
28
```

static IntegerPartitionsDominanceOrder(*n*)

Return the lattice of integer partitions on the integer n ordered by dominance.

That is, if $p = (p_1, \ldots, p_i)$ and $q = (q_1, \ldots, q_j)$ are integer partitions of n, then p is greater than q if and only if $p_1 + \cdots + p_k > q_1 + \cdots + q_k$ for all k.

INPUT:

- `n` – positive integer

EXAMPLES:

```
sage: P = posets.IntegerPartitionsDominanceOrder(6); P
Finite lattice containing 11 elements
sage: P.cover_relations()
[[[1, 1, 1, 1, 1, 1], [2, 1, 1, 1, 1]],
 [[2, 1, 1, 1, 1], [2, 2, 1, 1]],
 [[2, 2, 1, 1], [2, 2, 2]],
 [[2, 2, 1, 1], [3, 1, 1, 1]],
 [[2, 2, 2], [3, 2, 1]],
 [[3, 1, 1, 1], [3, 2, 1]],
 [[3, 2, 1], [3, 3]],
 [[3, 2, 1], [4, 1, 1]],
 [[3, 3], [4, 2]],
 [[4, 1, 1], [4, 2]],
 [[4, 2], [5, 1]],
 [[5, 1], [6]]]
```

```
>>> from sage.all import *
>>> P = posets.IntegerPartitionsDominanceOrder(Integer(6)); P
Finite lattice containing 11 elements
>>> P.cover_relations()
[[[1, 1, 1, 1, 1, 1], [2, 1, 1, 1, 1]],
 [[2, 1, 1, 1, 1], [2, 2, 1, 1]],
 [[2, 2, 1, 1], [2, 2, 2]],
 [[2, 2, 1, 1], [3, 1, 1, 1]],
 [[2, 2, 2], [3, 2, 1]],
 [[3, 1, 1, 1], [3, 2, 1]],
 [[3, 2, 1], [3, 3]],
 [[3, 2, 1], [4, 1, 1]],
 [[3, 3], [4, 2]],
 [[4, 1, 1], [4, 2]],
 [[4, 2], [5, 1]],
 [[5, 1], [6]]]
```

static MobilePoset(*ribbon*, *hangers*, *anchor=None*)

Return a mobile poset with the ribbon `ribbon` and with hanging d-complete posets specified in `hangers` and a d-complete poset attached above, specified in `anchor`.

INPUT:

- `ribbon` – a finite poset that is a ribbon
- `hangers` – dictionary mapping an element on the ribbon to a list of d-complete posets that it covers
- `anchor` – (optional) a `tuple` (`ribbon_elmt`, `anchor_elmt`, `anchor_poset`), where `anchor_elmt` covers `ribbon_elmt`, and `anchor_elmt` is an acyclic element of `anchor_poset`

EXAMPLES:

```
sage: R = Posets.RibbonPoset(5, [1,2])
sage: H = Poset([[5, 6, 7], [(5, 6), (6,7)]])
sage: M = Posets.MobilePoset(R, {3: [H]})
sage: len(M.cover_relations())
7

sage: P = posets.MobilePoset(posets.RibbonPoset(7, [1,3]),
....:         {1: [posets.YoungDiagramPoset([3, 2], dual=True)],
....:          3: [posets.DoubleTailedDiamond(6)]},
....:         anchor=(4, 2, posets.ChainPoset(6)))
sage: len(P.cover_relations())
33
```

```
>>> from sage.all import *
>>> R = Posets.RibbonPoset(Integer(5), [Integer(1),Integer(2)])
>>> H = Poset([[Integer(5), Integer(6), Integer(7)], [(Integer(5),
↪Integer(6)), (Integer(6),Integer(7))]])
>>> M = Posets.MobilePoset(R, {Integer(3): [H]})
>>> len(M.cover_relations())
7

>>> P = posets.MobilePoset(posets.RibbonPoset(Integer(7), [Integer(1),
↪Integer(3)]),
```

(continues on next page)

(continued from previous page)

```
...         {Integer(1): [posets.YoungDiagramPoset([Integer(3), Integer(2)],␣
↪dual=True)],
...          Integer(3): [posets.DoubleTailedDiamond(Integer(6))]},
...         anchor=(Integer(4), Integer(2), posets.ChainPoset(Integer(6))))
>>> len(P.cover_relations())
33
```

static NoncrossingPartitions(*W*)

Return the lattice of noncrossing partitions.

INPUT:

- `W` – a finite Coxeter group or a Weyl group

EXAMPLES:

```
sage: W = CoxeterGroup(['A', 3])                                        #␣
↪needs sage.groups
sage: posets.NoncrossingPartitions(W)                                   #␣
↪needs sage.groups
Finite lattice containing 14 elements

sage: W = WeylGroup(['B', 2], prefix='s')                               #␣
↪needs sage.groups
sage: posets.NoncrossingPartitions(W)                                   #␣
↪needs sage.groups
Finite lattice containing 6 elements
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A', Integer(3)])                                  ␣
↪     # needs sage.groups
>>> posets.NoncrossingPartitions(W)                                   #␣
↪needs sage.groups
Finite lattice containing 14 elements

>>> W = WeylGroup(['B', Integer(2)], prefix='s')                         ␣
↪     # needs sage.groups
>>> posets.NoncrossingPartitions(W)                                   #␣
↪needs sage.groups
Finite lattice containing 6 elements
```

static PentagonPoset(*facade=None*)

Return the Pentagon poset.

INPUT:

- `facade` – boolean; whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the *Poset()* constructor

EXAMPLES:

```
sage: P = posets.PentagonPoset(); P
Finite lattice containing 5 elements
sage: P.cover_relations()
[[0, 1], [0, 2], [1, 4], [2, 3], [3, 4]]
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset(); P
Finite lattice containing 5 elements
>>> P.cover_relations()
[[0, 1], [0, 2], [1, 4], [2, 3], [3, 4]]
```

static PermutationPattern(*n*)

Return the poset of permutations under pattern containment up to rank n.

INPUT:

- n – positive integer

A permutation $u = u_1 \cdots u_n$ contains the pattern $v = v_1 \cdots v_m$ if there is a (not necessarily consecutive) subsequence of u of length m whose entries have the same relative order as v.

See Wikipedia article Permutation_pattern.

EXAMPLES:

```
sage: P4 = posets.PermutationPattern(4); P4
Finite poset containing 33 elements
sage: sorted(P4.lower_covers(Permutation([2,4,1,3])))
[[1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]
```

```
>>> from sage.all import *
>>> P4 = posets.PermutationPattern(Integer(4)); P4
Finite poset containing 33 elements
>>> sorted(P4.lower_covers(Permutation([Integer(2),Integer(4),Integer(1),
↪Integer(3)])))
[[1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]
```

See also

has_pattern()

static PermutationPatternInterval(*bottom*, *top*)

Return the poset consisting of an interval in the poset of permutations under pattern containment between bottom and top.

INPUT:

- bottom, top – permutations where top contains bottom as a pattern

A permutation $u = u_1 \cdots u_n$ contains the pattern $v = v_1 \cdots v_m$ if there is a (not necessarily consecutive) subsequence of u of length m whose entries have the same relative order as v.

See Wikipedia article Permutation_pattern.

EXAMPLES:

```
sage: t = Permutation([2,3,1])
sage: b = Permutation([4,6,2,3,5,1])
sage: R = posets.PermutationPatternInterval(t, b); R
Finite poset containing 14 elements
```

(continues on next page)

(continued from previous page)

```
sage: R.moebius_function(R.bottom(),R.top())
-4
```

```
>>> from sage.all import *
>>> t = Permutation([Integer(2),Integer(3),Integer(1)])
>>> b = Permutation([Integer(4),Integer(6),Integer(2),Integer(3),Integer(5),
↪Integer(1)])
>>> R = posets.PermutationPatternInterval(t, b); R
Finite poset containing 14 elements
>>> R.moebius_function(R.bottom(),R.top())
-4
```

See also

has_pattern(), *PermutationPattern()*

static PermutationPatternOccurrenceInterval(*bottom*, *top*, *pos*)

Return the poset consisting of an interval in the poset of permutations under pattern containment between `bottom` and `top`, where a specified instance of `bottom` in `top` must be maintained.

INPUT:

- **`bottom`, `top` – permutations where `top` contains**
  `bottom` as a pattern
- **`pos` – list of indices indicating a distinguished copy of**
  `bottom` inside `top` (indexed starting at 0)

For further information (and picture illustrating included example), see [ST2010] .

See Wikipedia article Permutation_pattern.

EXAMPLES:

```
sage: t = Permutation([3,2,1])
sage: b = Permutation([6,3,4,5,2,1])
sage: A = posets.PermutationPatternOccurrenceInterval(t, b, (0,2,4)); A
Finite poset containing 8 elements
```

```
>>> from sage.all import *
>>> t = Permutation([Integer(3),Integer(2),Integer(1)])
>>> b = Permutation([Integer(6),Integer(3),Integer(4),Integer(5),Integer(2),
↪Integer(1)])
>>> A = posets.PermutationPatternOccurrenceInterval(t, b, (Integer(0),
↪Integer(2),Integer(4))); A
Finite poset containing 8 elements
```

See also

has_pattern(), *PermutationPattern()*, *PermutationPatternInterval()*

static PowerPoset(*n*)

Return the power poset on n element posets.

Elements of the power poset are all posets on the set $\{0, 1, \ldots, n-1\}$ ordered by extension. That is, the antichain of n elements is the bottom and $P_a \leq P_b$ in the power poset if P_b is an extension of P_a.

These were studied in [Bru1994].

EXAMPLES:

```
sage: P3 = posets.PowerPoset(3); P3
Finite meet-semilattice containing 19 elements
sage: all(P.is_chain() for P in P3.maximal_elements())
True
```

```
>>> from sage.all import *
>>> P3 = posets.PowerPoset(Integer(3)); P3
Finite meet-semilattice containing 19 elements
>>> all(P.is_chain() for P in P3.maximal_elements())
True
```

static ProductOfChains(*chain_lengths*, *facade=None*)

Return a product of chains.

- `chain_lengths` – list of nonnegative integers; number of elements in each chain
- `facade` – boolean; whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the `Poset()` constructor

EXAMPLES:

```
sage: P = posets.ProductOfChains([2, 2]); P
Finite lattice containing 4 elements
sage: P.linear_extension()
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: P.upper_covers((0,0))
[(0, 1), (1, 0)]
sage: P.lower_covers((1,1))
[(0, 1), (1, 0)]
```

```
>>> from sage.all import *
>>> P = posets.ProductOfChains([Integer(2), Integer(2)]); P
Finite lattice containing 4 elements
>>> P.linear_extension()
[(0, 0), (0, 1), (1, 0), (1, 1)]
>>> P.upper_covers((Integer(0),Integer(0)))
[(0, 1), (1, 0)]
>>> P.lower_covers((Integer(1),Integer(1)))
[(0, 1), (1, 0)]
```

static RandomLattice(*n*, *p*, *properties=None*)

Return a random lattice on `n` elements.

INPUT:

- `n` – number of elements, a nonnegative integer
- `p` – a probability, a positive real number less than one

- `properties` – list of properties for the lattice. Currently implemented:
  - `None`, no restrictions for lattices to create
  - `'planar'`, the lattice has an upward planar drawing
  - `'dismantlable'` (implicated by `'planar'`)
  - `'distributive'` (implicated by `'stone'`)
  - `'stone'`

OUTPUT:

A lattice on n elements. When `properties` is `None`, the probability p roughly measures number of covering relations of the lattice. To create interesting examples, make the probability a little below one, for example 0.9.

Currently parameter `p` has no effect only when `properties` is not `None`.

Note

Results are reproducible in same Sage version only. Underlying algorithm may change in future versions.

EXAMPLES:

```
sage: set_random_seed(0)  # Results are reproducible
sage: L = posets.RandomLattice(8, 0.995); L
Finite lattice containing 8 elements
sage: L.cover_relations()
[[7, 6], [7, 3], [7, 1], ..., [5, 4], [2, 4], [1, 4], [0, 4]]
sage: L = posets.RandomLattice(10, 0, properties=['dismantlable'])
sage: L.is_dismantlable()
True
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))  # Results are reproducible
>>> L = posets.RandomLattice(Integer(8), RealNumber('0.995')); L
Finite lattice containing 8 elements
>>> L.cover_relations()
[[7, 6], [7, 3], [7, 1], ..., [5, 4], [2, 4], [1, 4], [0, 4]]
>>> L = posets.RandomLattice(Integer(10), Integer(0), properties=[
↪'dismantlable'])
>>> L.is_dismantlable()
True
```

See also

RandomPoset()

static RandomPoset(*n*, *p*)

Generate a random poset on `n` elements according to a probability `p`.

INPUT:

- `n` – number of elements, a nonnegative integer

- p – a probability, a real number between 0 and 1 (inclusive)

OUTPUT:

A poset on n elements. The probability p roughly measures width/height of the output: $p = 0$ always generates an antichain, $p = 1$ will return a chain. To create interesting examples, keep the probability small, perhaps on the order of $1/n$.

EXAMPLES:

```
sage: set_random_seed(0)  # Results are reproducible
sage: P = posets.RandomPoset(5, 0.3)
sage: P.cover_relations()
[[5, 4], [4, 2], [1, 2]]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))  # Results are reproducible
>>> P = posets.RandomPoset(Integer(5), RealNumber('0.3'))
>>> P.cover_relations()
[[5, 4], [4, 2], [1, 2]]
```

See also

RandomLattice()

static RestrictedIntegerPartitions(*n*)

Return the poset of integer partitions on the integer n ordered by restricted refinement.

That is, if p and q are integer partitions of n, then p covers q if and only if q is obtained from p by joining two distinct parts of p (and sorting, if necessary).

EXAMPLES:

```
sage: P = posets.RestrictedIntegerPartitions(7); P
Finite poset containing 15 elements
sage: len(P.cover_relations())
17
```

```
>>> from sage.all import *
>>> P = posets.RestrictedIntegerPartitions(Integer(7)); P
Finite poset containing 15 elements
>>> len(P.cover_relations())
17
```

static RibbonPoset(*n*, *descents*)

Return a ribbon poset on n vertices with descents at descents.

INPUT:

- n – the number of vertices
- descents – an iterable; the indices on the ribbon where $y > x$

EXAMPLES:

```
sage: R = Posets.RibbonPoset(5, [1,2])
sage: sorted(R.cover_relations())
[[0, 1], [2, 1], [3, 2], [3, 4]]
```

```
>>> from sage.all import *
>>> R = Posets.RibbonPoset(Integer(5), [Integer(1),Integer(2)])
>>> sorted(R.cover_relations())
[[0, 1], [2, 1], [3, 2], [3, 4]]
```

static SSTPoset(*s*, *f=None*)

The lattice poset on semistandard tableaux of shape `s` and largest entry `f` that is ordered by componentwise comparison of the entries.

INPUT:

- `s` – shape of the tableaux
- `f` – integer (default: `None`); the maximum fill number. By default (`None`), the method uses the number of cells in the shape.

Note

This is a basic implementation and most certainly not the most efficient.

EXAMPLES:

```
sage: posets.SSTPoset([2,1])
Finite lattice containing 8 elements

sage: posets.SSTPoset([2,1],4)
Finite lattice containing 20 elements

sage: posets.SSTPoset([2,1],2).cover_relations()
[[[[1, 1], [2]], [[1, 2], [2]]]]

sage: posets.SSTPoset([3,2]).bottom()        # long time (6s on sage.math,␣
↪2012)
[[1, 1, 1], [2, 2]]

sage: posets.SSTPoset([3,2],4).maximal_elements()
[[[3, 3, 4], [4, 4]]]
```

```
>>> from sage.all import *
>>> posets.SSTPoset([Integer(2),Integer(1)])
Finite lattice containing 8 elements

>>> posets.SSTPoset([Integer(2),Integer(1)],Integer(4))
Finite lattice containing 20 elements

>>> posets.SSTPoset([Integer(2),Integer(1)],Integer(2)).cover_relations()
[[[[1, 1], [2]], [[1, 2], [2]]]]

>>> posets.SSTPoset([Integer(3),Integer(2)]).bottom()        # long time (6s␣
```

(continues on next page)

(continued from previous page)

```
↪on sage.math, 2012)
[[1, 1, 1], [2, 2]]

>>> posets.SSTPoset([Integer(3),Integer(2)],Integer(4)).maximal_elements()
[[[3, 3, 4], [4, 4]]]
```

static SetPartitions(*n*)

Return the lattice of set partitions of the set $\{1, \ldots, n\}$ ordered by refinement.

INPUT:

- n – positive integer

EXAMPLES:

```
sage: posets.SetPartitions(4)
Finite lattice containing 15 elements
```

```
>>> from sage.all import *
>>> posets.SetPartitions(Integer(4))
Finite lattice containing 15 elements
```

static ShardPoset(*n*)

Return the shard intersection order on permutations of size n.

This is defined on the set of permutations. To every permutation, one can attach a pre-order, using the descending runs and their relative positions.

The shard intersection order is given by the implication (or refinement) order on the set of pre-orders defined from all permutations.

This can also be seen in a geometrical way. Every pre-order defines a cone in a vector space of dimension n. The shard poset is given by the inclusion of these cones.

See also:

shard_preorder_graph()

EXAMPLES:

```
sage: P = posets.ShardPoset(4); P  # indirect doctest
Finite poset containing 24 elements
sage: P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
sage: P.characteristic_polynomial()
q^3 - 11*q^2 + 23*q - 13
sage: P.zeta_polynomial()
17/3*q^3 - 6*q^2 + 4/3*q
sage: P.is_self_dual()
False
```

```
>>> from sage.all import *
>>> P = posets.ShardPoset(Integer(4)); P  # indirect doctest
```

(continues on next page)

(continued from previous page)

```
Finite poset containing 24 elements
>>> P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
>>> P.characteristic_polynomial()
q^3 - 11*q^2 + 23*q - 13
>>> P.zeta_polynomial()
17/3*q^3 - 6*q^2 + 4/3*q
>>> P.is_self_dual()
False
```

static StandardExample(*n*, *facade=None*)

Return the partially ordered set on $2n$ elements with dimension n.

Let P be the poset on $\{0, 1, 2, \ldots, 2n-1\}$ whose defining relations are that $i < j$ for every $0 \leq i < n \leq j < 2n$ except when $i + n = j$. The poset P is the so-called *standard example* of a poset with dimension n.

INPUT:

- `n` – integer ≥ 2; dimension of the constructed poset
- `facade` – boolean; whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the `Poset()` constructor

OUTPUT: the standard example of a poset of dimension n

EXAMPLES:

```
sage: A = posets.StandardExample(3); A
Finite poset containing 6 elements
sage: A.dimension()                                                    #␣
↪needs networkx
3
```

```
>>> from sage.all import *
>>> A = posets.StandardExample(Integer(3)); A
Finite poset containing 6 elements
>>> A.dimension()                                                    #␣
↪needs networkx
3
```

REFERENCES:

- [Gar2015]
- [Ros1999]

static SymmetricGroupAbsoluteOrderPoset(*n*, *labels='permutations'*)

Return the poset of permutations with respect to absolute order.

INPUT:

- `n` – a positive integer
- `label` – (default: `'permutations'`) a label for the elements of the poset returned by the function; the options are
  - `'permutations'` – labels the elements by their one-line notation
  - `'reduced_words'` – labels the elements by the lexicographically minimal reduced word

– 'cycles' – labels the elements by their expression as a product of cycles

EXAMPLES:

```
sage: posets.SymmetricGroupAbsoluteOrderPoset(4)                              #␣
↪needs sage.groups
Finite poset containing 24 elements
sage: posets.SymmetricGroupAbsoluteOrderPoset(3, labels='cycles')             #␣
↪needs sage.groups
Finite poset containing 6 elements
sage: posets.SymmetricGroupAbsoluteOrderPoset(3, labels='reduced_words')      #␣
↪needs sage.groups
Finite poset containing 6 elements
```

```
>>> from sage.all import *
>>> posets.SymmetricGroupAbsoluteOrderPoset(Integer(4))                        ␣
↪     # needs sage.groups
Finite poset containing 24 elements
>>> posets.SymmetricGroupAbsoluteOrderPoset(Integer(3), labels='cycles')       ␣
↪     # needs sage.groups
Finite poset containing 6 elements
>>> posets.SymmetricGroupAbsoluteOrderPoset(Integer(3), labels='reduced_words
↪')     # needs sage.groups
Finite poset containing 6 elements
```

static SymmetricGroupBruhatIntervalPoset(*start*, *end*)

The poset of permutations with respect to Bruhat order.

INPUT:

- start – list permutation
- end – list permutation (same n, of course)

Note

Must have start <= end.

EXAMPLES:

Any interval is rank symmetric if and only if it avoids these permutations:

```
sage: P1 = posets.SymmetricGroupBruhatIntervalPoset([1,2,3,4], [3,4,1,2])
sage: P2 = posets.SymmetricGroupBruhatIntervalPoset([1,2,3,4], [4,2,3,1])
sage: ranks1 = [P1.rank(v) for v in P1]
sage: ranks2 = [P2.rank(v) for v in P2]
sage: [ranks1.count(i) for i in sorted(set(ranks1))]
[1, 3, 5, 4, 1]
sage: [ranks2.count(i) for i in sorted(set(ranks2))]
[1, 3, 5, 6, 4, 1]
```

```
>>> from sage.all import *
>>> P1 = posets.SymmetricGroupBruhatIntervalPoset([Integer(1),Integer(2),
↪Integer(3),Integer(4)], [Integer(3),Integer(4),Integer(1),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> P2 = posets.SymmetricGroupBruhatIntervalPoset([Integer(1),Integer(2),
↪Integer(3),Integer(4)], [Integer(4),Integer(2),Integer(3),Integer(1)])
>>> ranks1 = [P1.rank(v) for v in P1]
>>> ranks2 = [P2.rank(v) for v in P2]
>>> [ranks1.count(i) for i in sorted(set(ranks1))]
[1, 3, 5, 4, 1]
>>> [ranks2.count(i) for i in sorted(set(ranks2))]
[1, 3, 5, 6, 4, 1]
```

static SymmetricGroupBruhatOrderPoset(*n*)

The poset of permutations with respect to Bruhat order.

EXAMPLES:

```
sage: posets.SymmetricGroupBruhatOrderPoset(4)
Finite poset containing 24 elements
```

```
>>> from sage.all import *
>>> posets.SymmetricGroupBruhatOrderPoset(Integer(4))
Finite poset containing 24 elements
```

static SymmetricGroupWeakOrderPoset(*n*, *labels='permutations'*, *side='right'*)

The poset of permutations of $\{1, 2, \ldots, n\}$ with respect to the weak order (also known as the permutohedron order, cf. `permutohedron_lequal()`).

The optional variable `labels` (default: `'permutations'`) determines the labelling of the elements if $n < 10$. The optional variable `side` (default: `'right'`) determines whether the right or the left permutohedron order is to be used.

EXAMPLES:

```
sage: posets.SymmetricGroupWeakOrderPoset(4)
Finite poset containing 24 elements
```

```
>>> from sage.all import *
>>> posets.SymmetricGroupWeakOrderPoset(Integer(4))
Finite poset containing 24 elements
```

static TamariLattice(*n*, *m=1*)

Return the n-th Tamari lattice.

Using the slope parameter m, one can also get the m-Tamari lattices.

INPUT:

- `n` – nonnegative integer (the index)
- `m` – nonnegative integer (the slope, default: 1)

OUTPUT: a finite lattice

In the usual case, the elements of the lattice are `Dyck paths` in the $(n + 1 \times n)$-rectangle. For a general slope m, the elements are Dyck paths in the $(mn + 1 \times n)$-rectangle.

See Tamari lattice for mathematical background.

EXAMPLES:

```
sage: posets.TamariLattice(3)
Finite lattice containing 5 elements

sage: posets.TamariLattice(3, 2)
Finite lattice containing 12 elements
```

```
>>> from sage.all import *
>>> posets.TamariLattice(Integer(3))
Finite lattice containing 5 elements

>>> posets.TamariLattice(Integer(3), Integer(2))
Finite lattice containing 12 elements
```

REFERENCES:

- [BMFPR2011]

static TetrahedralPoset(*n*, **colors*, ***labels*)

Return the tetrahedral poset based on the input colors.

This method will return the tetrahedral poset with $n - 1$ layers and covering relations based on the input colors of 'green', 'red', 'orange', 'silver', 'yellow' and 'blue' as defined in [Striker2011]. For particular color choices, the order ideals of the resulting tetrahedral poset will be isomorphic to known combinatorial objects.

For example, for the colors 'blue', 'yellow', 'orange', and 'green', the order ideals will be in bijection with alternating sign matrices. For the colors 'yellow', 'orange', and 'green', the order ideals will be in bijection with semistandard Young tableaux of staircase shape. For the colors 'red', 'orange', 'green', and optionally 'yellow', the order ideals will be in bijection with totally symmetric self-complementary plane partitions in a $2n \times 2n \times 2n$ box.

INPUT:

- `n` – defines the number (n-1) of layers in the poset
- `colors` – the colors that define the covering relations of the poset; colors used are 'green', 'red', 'yellow', 'orange', 'silver', and 'blue'
- `labels` – keyword variable used to determine whether the poset is labeled with integers or tuples. To label with integers, the method should be called with `labels='integers'`. Otherwise, the labeling will default to tuples.

EXAMPLES:

```
sage: posets.TetrahedralPoset(4,'green','red','yellow','silver','blue','orange
↪')
Finite poset containing 10 elements

sage: posets.TetrahedralPoset(4,'green','red','yellow','silver','blue','orange
↪',
....:                          labels='integers')
Finite poset containing 10 elements

sage: A = AlternatingSignMatrices(3)
sage: p = A.lattice()
sage: ji = p.join_irreducibles_poset()
sage: tet = posets.TetrahedralPoset(3, 'green','yellow','blue','orange')
```

(continues on next page)

(continued from previous page)

```
sage: ji.is_isomorphic(tet)
True
```

```
>>> from sage.all import *
>>> posets.TetrahedralPoset(Integer(4),'green','red','yellow','silver','blue',
↪'orange')
Finite poset containing 10 elements

>>> posets.TetrahedralPoset(Integer(4),'green','red','yellow','silver','blue',
↪'orange',
...                             labels='integers')
Finite poset containing 10 elements

>>> A = AlternatingSignMatrices(Integer(3))
>>> p = A.lattice()
>>> ji = p.join_irreducibles_poset()
>>> tet = posets.TetrahedralPoset(Integer(3), 'green','yellow','blue','orange
↪')
>>> ji.is_isomorphic(tet)
True
```

static UpDownPoset (*n*, *m=1*)

Return the up-down poset on n elements where every $(m+1)$ step is down and the rest are up.

The case where $m = 1$ is sometimes referred to as the zig-zag poset or the fence.

INPUT:

- `n` – nonnegative integer; number of elements in the poset
- `m` – nonnegative integer (default: 1); how frequently down steps occur

OUTPUT:

The partially ordered set on $\{0, 1, \ldots, n-1\}$ where i covers $i+1$ if m divides $i+1$, and $i+1$ covers i otherwise.

EXAMPLES:

```
sage: P = posets.UpDownPoset(7, 2); P
Finite poset containing 7 elements
sage: sorted(P.cover_relations())
[[0, 1], [1, 2], [3, 2], [3, 4], [4, 5], [6, 5]]
```

```
>>> from sage.all import *
>>> P = posets.UpDownPoset(Integer(7), Integer(2)); P
Finite poset containing 7 elements
>>> sorted(P.cover_relations())
[[0, 1], [1, 2], [3, 2], [3, 4], [4, 5], [6, 5]]
```

Fibonacci numbers as the number of antichains of a poset:

```
sage: [len(posets.UpDownPoset(n).antichains().list()) for n in range(6)]
[1, 2, 3, 5, 8, 13]
```

```
>>> from sage.all import *
>>> [len(posets.UpDownPoset(n).antichains().list()) for n in␣
↪range(Integer(6))]
[1, 2, 3, 5, 8, 13]
```

static YoungDiagramPoset(*lam*, *dual=False*)

Return the poset of cells in the Young diagram of a partition.

INPUT:

- `lam` – a partition
- `dual` – boolean (default: `False`); determines the orientation of the poset. If `True`, then it is a join semilattice, otherwise it is a meet semilattice.

EXAMPLES:

```
sage: P = posets.YoungDiagramPoset(Partition([2, 2])); P
Finite meet-semilattice containing 4 elements

sage: sorted(P.cover_relations())
[[(0, 0), (0, 1)], [(0, 0), (1, 0)], [(0, 1), (1, 1)], [(1, 0), (1, 1)]]

sage: posets.YoungDiagramPoset([3, 2], dual=True)
Finite join-semilattice containing 5 elements
```

```
>>> from sage.all import *
>>> P = posets.YoungDiagramPoset(Partition([Integer(2), Integer(2)])); P
Finite meet-semilattice containing 4 elements

>>> sorted(P.cover_relations())
[[(0, 0), (0, 1)], [(0, 0), (1, 0)], [(0, 1), (1, 1)], [(1, 0), (1, 1)]]

>>> posets.YoungDiagramPoset([Integer(3), Integer(2)], dual=True)
Finite join-semilattice containing 5 elements
```

static YoungFibonacci(*n*)

Return the Young-Fibonacci lattice up to rank n.

Elements of the (infinite) lattice are words with letters '1' and '2'. The covers of a word are the words with another '1' added somewhere not after the first occurrence of an existing '1' and, additionally, the words where the first '1' is replaced by a '2'. The lattice is truncated to have rank n.

See Wikipedia article Young-Fibonacci lattice.

EXAMPLES:

```
sage: Y5 = posets.YoungFibonacci(5); Y5
Finite meet-semilattice containing 20 elements
sage: sorted(Y5.upper_covers(Word('211')))
[word: 1211, word: 2111, word: 221]
```

```
>>> from sage.all import *
>>> Y5 = posets.YoungFibonacci(Integer(5)); Y5
Finite meet-semilattice containing 20 elements
```

(continues on next page)

(continued from previous page)

```
>>> sorted(Y5.upper_covers(Word('211')))
[word: 1211, word: 2111, word: 221]
```

static YoungsLattice(*n*)

Return Young's Lattice up to rank n.

In other words, the poset of partitions of size less than or equal to n ordered by inclusion.

INPUT:

- `n` – positive integer

EXAMPLES:

```
sage: P = posets.YoungsLattice(3); P
Finite meet-semilattice containing 7 elements
sage: P.cover_relations()
[[[], [1]],
 [[1], [1, 1]],
 [[1], [2]],
 [[1, 1], [1, 1, 1]],
 [[1, 1], [2, 1]],
 [[2], [2, 1]],
 [[2], [3]]]
```

```
>>> from sage.all import *
>>> P = posets.YoungsLattice(Integer(3)); P
Finite meet-semilattice containing 7 elements
>>> P.cover_relations()
[[[], [1]],
 [[1], [1, 1]],
 [[1], [2]],
 [[1, 1], [1, 1, 1]],
 [[1, 1], [2, 1]],
 [[2], [2, 1]],
 [[2], [3]]]
```

static YoungsLatticePrincipalOrderIdeal(*lam*)

Return the principal order ideal of the partition lam in Young's Lattice.

INPUT:

- `lam` – a partition

EXAMPLES:

```
sage: P = posets.YoungsLatticePrincipalOrderIdeal(Partition([2,2]))
sage: P
Finite lattice containing 6 elements
sage: P.cover_relations()
[[[], [1]],
 [[1], [1, 1]],
 [[1], [2]],
 [[1, 1], [2, 1]],
 [[2], [2, 1]],
 [[2, 1], [2, 2]]]
```

```
>>> from sage.all import *
>>> P = posets.YoungsLatticePrincipalOrderIdeal(Partition([Integer(2),
↪Integer(2)]))
>>> P
Finite lattice containing 6 elements
>>> P.cover_relations()
[[[], [1]],
 [[1], [1, 1]],
 [[1], [2]],
 [[1, 1], [2, 1]],
 [[2], [2, 1]],
 [[2, 1], [2, 2]]]
```

sage = <module 'sage' (namespace) from ['/home/user/sage-git/src/sage']>

sage.combinat.posets.poset_examples.**check_int**(*n*, *minimum=0*)

Check that n is an integer at least equal to minimum.

This is a boilerplate function ensuring input safety.

INPUT:

- n – anything
- minimum – an optional integer (default: 0)

EXAMPLES:

```
sage: from sage.combinat.posets.poset_examples import check_int
sage: check_int(6, 3)
6
sage: check_int(6)
6

sage: check_int(-1)
Traceback (most recent call last):
...
ValueError: number of elements must be a nonnegative integer, not -1

sage: check_int(1, 3)
Traceback (most recent call last):
...
ValueError: number of elements must be an integer at least 3, not 1

sage: check_int('junk')
Traceback (most recent call last):
...
ValueError: number of elements must be a nonnegative integer, not junk
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.poset_examples import check_int
>>> check_int(Integer(6), Integer(3))
6
>>> check_int(Integer(6))
6
```

(continues on next page)

(continued from previous page)

```
>>> check_int(-Integer(1))
Traceback (most recent call last):
...
ValueError: number of elements must be a nonnegative integer, not -1

>>> check_int(Integer(1), Integer(3))
Traceback (most recent call last):
...
ValueError: number of elements must be an integer at least 3, not 1

>>> check_int('junk')
Traceback (most recent call last):
...
ValueError: number of elements must be a nonnegative integer, not junk
```

`sage.combinat.posets.poset_examples.`**`posets`**

alias of *Posets*

5.1.186 Finite posets

This module implements finite partially ordered sets. It defines:

FinitePoset	A class for finite posets
FinitePosets_n	A class for finite posets up to isomorphism (i.e. unlabeled posets)
Poset()	Construct a finite poset from various forms of input data.
is_poset()	Return `True` if a directed graph is acyclic and transitively reduced.

List of Poset methods

Comparing, intervals and relations

is_less_than()	Return `True` if x is strictly less than y in the poset.
is_greater_than()	Return `True` if x is strictly greater than y in the poset.
is_lequal()	Return `True` if x is less than or equal to y in the poset.
is_gequal()	Return `True` if x is greater than or equal to y in the poset.
compare_elements()	Compare two element of the poset.
closed_interval()	Return the list of elements in a closed interval of the poset.
open_interval()	Return the list of elements in an open interval of the poset.
relations()	Return the list of relations in the poset.
relations_iterator()	Return an iterator over relations in the poset.
order_filter()	Return the upper set generated by elements.
order_ideal()	Return the lower set generated by elements.

Covering

covers()	Return True if y covers x.
lower_covers()	Return elements covered by given element.
upper_covers()	Return elements covering given element.
cover_relations()	Return the list of cover relations.
lower_covers_iterator()	Return an iterator over elements covered by given element.
upper_covers_iterator()	Return an iterator over elements covering given element.
cover_relations_iterator()	Return an iterator over cover relations of the poset.
common_upper_covers()	Return the list of all common upper covers of the given elements.
common_lower_covers()	Return the list of all common lower covers of the given elements.
meet()	Return the meet of given elements if it exists; None otherwise.
join()	Return the join of given elements if it exists; None otherwise.

Properties of the poset

cardinality()	Return the number of elements in the poset.
height()	Return the number of elements in a longest chain of the poset.
width()	Return the number of elements in a longest antichain of the poset.
relations_number()	Return the number of relations in the poset.
dimension()	Return the dimension of the poset.
jump_number()	Return the jump number of the poset.
magnitude()	Return the magnitude of the poset.
has_bottom()	Return True if the poset has a unique minimal element.
has_top()	Return True if the poset has a unique maximal element.
is_bounded()	Return True if the poset has both unique minimal and unique maximal element.
is_chain()	Return True if the poset is totally ordered.
is_connected()	Return True if the poset is connected.
is_graded()	Return True if all maximal chains of the poset has same length.
is_ranked()	Return True if the poset has a rank function.
is_rank_symmetric()	Return True if the poset is rank symmetric.
is_series_parallel()	Return True if the poset can be built by ordinal sums and disjoint unions.
is_greedy()	Return True if all greedy linear extensions have equal number of jumps.
is_jump_critical()	Return True if removal of any element reduces the jump number.
is_eulerian()	Return True if the poset is Eulerian.
is_incomparable_chain_free()	Return True if the poset is (m+n)-free.
is_slender()	Return True if the poset is slender.
is_sperner()	Return True if the poset is Sperner.
is_join_semilattice()	Return True if the poset has a join operation.
is_meet_semilattice()	Return True if the poset has a meet operation.

Minimal and maximal elements

bottom()	Return the bottom element of the poset, if it exists.
top()	Return the top element of the poset, if it exists.
maximal_elements()	Return the list of the maximal elements of the poset.
minimal_elements()	Return the list of the minimal elements of the poset.

New posets from old ones

disjoint_union()	Return the disjoint union of the poset with other poset.
ordinal_sum()	Return the ordinal sum of the poset with other poset.
product()	Return the Cartesian product of the poset with other poset.
ordinal_product()	Return the ordinal product of the poset with other poset.
rees_product()	Return the Rees product of the poset with other poset.
lexicographic_sum()	Return the lexicographic sum of posets.
star_product()	Return the star product of the poset with other poset.
with_bounds()	Return the poset with bottom and top element adjoined.
without_bounds()	Return the poset with bottom and top element removed.
dual()	Return the dual of the poset.
completion_by_cuts()	Return the Dedekind-MacNeille completion of the poset.
intervals_poset()	Return the poset of intervals of the poset.
connected_components()	Return the connected components of the poset as subposets.
factor()	Return the decomposition of the poset as a Cartesian product.
ordinal_summands()	Return the ordinal summands of the poset.
subposet()	Return the subposet containing elements with partial order induced by this poset.
random_subposet()	Return a random subposet that contains each element with given probability.
relabel()	Return a copy of this poset with its elements relabelled.
canonical_label()	Return copy of the poset canonically (re)labelled to integers.
slant_sum()	Return the slant sum poset of two posets.

Chains, antichains & linear intervals

is_chain_of_poset()	Return `True` if elements in the given list are comparable.
is_antichain_of_poset()	Return `True` if elements in the given list are incomparable.
is_linear_interval()	Return whether the given interval is a total order.
chains()	Return the chains of the poset.
antichains()	Return the antichains of the poset.
maximal_chains()	Return the maximal chains of the poset.
maximal_antichains()	Return the maximal antichains of the poset.
maximal_chains_iterator()	Return an iterator over the maximal chains of the poset.
maximal_chain_length()	Return the maximum length of maximal chains of the poset.
antichains_iterator()	Return an iterator over the antichains of the poset.
random_maximal_chain()	Return a random maximal chain.
random_maximal_antichain()	Return a random maximal antichain.
linear_intervals_count()	Return the enumeration of linear intervals in the poset.

Drawing

show()	Display the Hasse diagram of the poset.
plot()	Return a Graphic object corresponding the Hasse diagram of the poset.
graphviz_string()	Return a representation in the DOT language, ready to render in graphviz.

Comparing posets

is_isomorphic()	Return `True` if both posets are isomorphic.
is_induced_subposet()	Return `True` if given poset is an induced subposet of this poset.

Polynomials

chain_polynomial()	Return the chain polynomial of the poset.
characteristic_polynomial()	Return the characteristic polynomial of the poset.
f_polynomial()	Return the f-polynomial of the poset.
flag_f_polynomial()	Return the flag f-polynomial of the poset.
h_polynomial()	Return the h-polynomial of the poset.
flag_h_polynomial()	Return the flag h-polynomial of the poset.
order_polynomial()	Return the order polynomial of the poset.
zeta_polynomial()	Return the zeta polynomial of the poset.
apozeta_polynomial()	Return the apozeta polynomial of the poset.
M_triangle()	Return the M-triangle of the poset.
kazhdan_lusztig_polynomial()	Return the Kazhdan-Lusztig polynomial of the poset.
coxeter_polynomial()	Return the characteristic polynomial of the Coxeter transformation.
degree_polynomial()	Return the generating polynomial of degrees of vertices in the Hasse diagram.
p_partition_enumerator()	Return a P-partition enumerator of the poset.

Polytopes

chain_polytope()	Return the chain polytope of the poset.
order_polytope()	Return the order polytope of the poset.

Graphs

hasse_diagram()	Return the Hasse diagram of the poset as a directed graph.
cover_relations_graph()	Return the (undirected) graph of cover relations.
comparability_graph()	Return the comparability graph of the poset.
incomparability_graph()	Return the incomparability graph of the poset.
frank_network()	Return Frank's network of the poset.
linear_extensions_graph()	Return the linear extensions graph of the poset.

Linear extensions

is_linear_extension()	Return `True` if the given list is a linear extension of the poset.
linear_extension()	Return a linear extension of the poset.
linear_extensions()	Return the enumerated set of all the linear extensions of the poset.
promotion()	Return the (extended) promotion on the linear extension of the poset.
evacuation()	Return evacuation on the linear extension associated to the poset.
with_linear_extension()	Return a copy of `self` with a different default linear extension.
random_linear_extension()	Return a random linear extension.

Matrices

lequal_matrix()	Compute the matrix whose `(i,j)` entry is 1 if `self.linear_extension()[i] < self.linear_extension()[j]` and 0 otherwise.
moebius_function()	Return the value of Möbius function of given elements in the poset.
moebius_function_matrix()	Return a matrix whose `(i,j)` entry is the value of the Möbius function evaluated at `self.linear_extension()[i]` and `self.linear_extension()[j]`.
coxeter_transformation()	Return the matrix of the Auslander-Reiten translation acting on the Grothendieck group of the derived category of modules.
coxeter_smith_form()	Return the Smith form of the Coxeter transformation.

Miscellaneous

sorted()	Return given list sorted by the poset.
isomorphic_subposets()	Return all subposets isomorphic to another poset.
isomorphic_subposets_iterator()	Return an iterator over the subposets isomorphic to another poset.
has_isomorphic_subposet()	Return `True` if the poset contains a subposet isomorphic to another poset.
list()	List the elements of the poset.
cuts()	Return the cuts of the given poset.
dilworth_decomposition()	Return a partition of the points into the minimal number of chains.
greene_shape()	Compute the Greene-Kleitman partition aka Greene shape of the poset `self`.
incidence_algebra()	Return the incidence algebra of `self`.
is_EL_labelling()	Return whether `f` is an EL labelling of the poset.
isomorphic_subposets_iterator()	Return an iterator over the subposets isomorphic to another poset.
isomorphic_subposets()	Return all subposets isomorphic to another poset.
level_sets()	Return elements grouped by maximal number of cover relations from a minimal element.
order_complex()	Return the order complex associated to this poset.
random_order_ideal()	Return a random order ideal of `self` with uniform probability.
rank()	Return the rank of an element, or the rank of the poset.
rank_function()	Return a rank function of the poset, if it exists.
unwrap()	Unwrap an element of this poset.
atkinson()	Return the a-spectrum of a poset whose undirected Hasse diagram is a forest.
spectrum()	Return the a-spectrum of this poset.

Classes and functions

class `sage.combinat.posets.posets.`**`FinitePoset`**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: `UniqueRepresentation`, `Parent`

A (finite) n-element poset constructed from a directed acyclic graph.

INPUT:

- `hasse_diagram` – an instance of *FinitePoset*, or a `DiGraph` that is transitively-reduced, acyclic, loop-free, and multiedge-free.
- `elements` – an optional list of elements, with `element[i]` corresponding to vertex `i`. If `elements` is `None`, then it is set to be the vertex set of the digraph. Note that if this option is set, then `elements` is considered as a specified linear extension of the poset and the `linear_extension` attribute is set.

- `category` – `FinitePosets`, or a subcategory thereof
- `facade` – boolean or `None` (default); whether the *`FinitePoset`*'s elements should be wrapped to make them aware of the Poset they belong to.
  - If `facade = True`, the *`FinitePoset`*'s elements are exactly those given as input.
  - If `facade = False`, the *`FinitePoset`*'s elements will become `PosetElement` objects.
  - If `facade = None` (default) the expected behaviour is the behaviour of `facade = True`, unless the opposite can be deduced from the context (i.e. for instance if a *`FinitePoset`* is built from another *`FinitePoset`*, itself built with `facade = False`)
- `key` – any hashable value (default: `None`)

EXAMPLES:

```
sage: uc = [[2,3], [], [1], [1], [1], [3,4]]
sage: from sage.combinat.posets.posets import FinitePoset
sage: P = FinitePoset(DiGraph(dict([[i,uc[i]] for i in range(len(uc))])),
↪facade=False); P
Finite poset containing 6 elements
sage: P.cover_relations()
[[5, 4], [5, 3], [4, 1], [0, 2], [0, 3], [2, 1], [3, 1]]
sage: TestSuite(P).run()
sage: P.category()
Category of finite enumerated posets
sage: P.__class__
<class 'sage.combinat.posets.posets.FinitePoset_with_category'>

sage: Q = sage.combinat.posets.posets.FinitePoset(P, facade = False); Q
Finite poset containing 6 elements

sage: Q is P
True
```

```
>>> from sage.all import *
>>> uc = [[Integer(2),Integer(3)], [], [Integer(1)], [Integer(1)], [Integer(1)],
↪[Integer(3),Integer(4)]]
>>> from sage.combinat.posets.posets import FinitePoset
>>> P = FinitePoset(DiGraph(dict([[i,uc[i]] for i in range(len(uc))])),
↪facade=False); P
Finite poset containing 6 elements
>>> P.cover_relations()
[[5, 4], [5, 3], [4, 1], [0, 2], [0, 3], [2, 1], [3, 1]]
>>> TestSuite(P).run()
>>> P.category()
Category of finite enumerated posets
>>> P.__class__
<class 'sage.combinat.posets.posets.FinitePoset_with_category'>

>>> Q = sage.combinat.posets.posets.FinitePoset(P, facade = False); Q
Finite poset containing 6 elements

>>> Q is P
True
```

We keep the same underlying Hasse diagram, but change the elements:

```
sage: Q = sage.combinat.posets.posets.FinitePoset(P, elements=[1,2,3,4,5,6],
↪facade=False); Q
Finite poset containing 6 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 5], [2, 6], [3, 4], [3, 5], [4, 6], [5, 6]]
```

```
>>> from sage.all import *
>>> Q = sage.combinat.posets.posets.FinitePoset(P, elements=[Integer(1),
↪Integer(2),Integer(3),Integer(4),Integer(5),Integer(6)], facade=False); Q
Finite poset containing 6 elements with distinguished linear extension
>>> Q.cover_relations()
[[1, 2], [1, 5], [2, 6], [3, 4], [3, 5], [4, 6], [5, 6]]
```

We test the facade argument:

```
sage: P = Poset(DiGraph({'a':['b'],'b':['c'],'c':['d']}), facade=False)
sage: P.category()
Category of finite enumerated posets
sage: parent(P[0]) is P
True

sage: Q = Poset(DiGraph({'a':['b'],'b':['c'],'c':['d']}), facade=True)
sage: Q.category()
Category of facade finite enumerated posets
sage: parent(Q[0]) is str
True
sage: TestSuite(Q).run(skip = ['_test_an_element']) # is_parent_of is not yet
↪implemented
```

```
>>> from sage.all import *
>>> P = Poset(DiGraph({'a':['b'],'b':['c'],'c':['d']}), facade=False)
>>> P.category()
Category of finite enumerated posets
>>> parent(P[Integer(0)]) is P
True

>>> Q = Poset(DiGraph({'a':['b'],'b':['c'],'c':['d']}), facade=True)
>>> Q.category()
Category of facade finite enumerated posets
>>> parent(Q[Integer(0)]) is str
True
>>> TestSuite(Q).run(skip = ['_test_an_element']) # is_parent_of is not yet
↪implemented
```

Changing a non facade poset to a facade poset:

```
sage: PQ = Poset(P, facade=True)
sage: PQ.category()
Category of facade finite enumerated posets
sage: parent(PQ[0]) is str
True
```

(continues on next page)

(continued from previous page)

```
sage: PQ is Q
True
```

```
>>> from sage.all import *
>>> PQ = Poset(P, facade=True)
>>> PQ.category()
Category of facade finite enumerated posets
>>> parent(PQ[Integer(0)]) is str
True
>>> PQ is Q
True
```

Changing a facade poset to a non facade poset:

```
sage: QP = Poset(Q, facade = False)
sage: QP.category()
Category of finite enumerated posets
sage: parent(QP[0]) is QP
True
```

```
>>> from sage.all import *
>>> QP = Poset(Q, facade = False)
>>> QP.category()
Category of finite enumerated posets
>>> parent(QP[Integer(0)]) is QP
True
```

Conversion to some other software is possible:

```
sage: P = posets.TamariLattice(3)
sage: libgap(P)                                         # optional - gap_package_
↪qpa
<A poset on 5 points>

sage: P = Poset({1:[2],2:[]})
sage: macaulay2('needsPackage "Posets"')    # optional - macaulay2
Posets
sage: macaulay2(P)    # optional - macaulay2
Relation Matrix: | 1 1 |
                 | 0 1 |
```

```
>>> from sage.all import *
>>> P = posets.TamariLattice(Integer(3))
>>> libgap(P)                                         # optional - gap_package_qpa
<A poset on 5 points>

>>> P = Poset({Integer(1):[Integer(2)],Integer(2):[]})
>>> macaulay2('needsPackage "Posets"')    # optional - macaulay2
Posets
>>> macaulay2(P)    # optional - macaulay2
Relation Matrix: | 1 1 |
                 | 0 1 |
```

Note

A class that inherits from this class needs to define `Element`. This is the class of the elements that the inheriting class contains. For example, for this class, `FinitePoset`, `Element` is `PosetElement`. It can also define `_dual_class` which is the class of dual posets of this class. E.g. `FiniteMeetSemilattice._dual_class` is `FiniteJoinSemilattice`.

Element

alias of *PosetElement*

M_triangle()

Return the M-triangle of the poset.

The poset is expected to be graded.

OUTPUT: an *M_triangle*

The M-triangle is the generating polynomial of the Möbius numbers

$$M(x, y) = \sum_{a \leq b} \mu(a, b) x^{|a|} y^{|b|}.$$

EXAMPLES:

```
sage: P = posets.DiamondPoset(5)
sage: P.M_triangle()                                                    #␣
↪needs sage.combinat
M: x^2*y^2 - 3*x*y^2 + 3*x*y + 2*y^2 - 3*y + 1
```

```
>>> from sage.all import *
>>> P = posets.DiamondPoset(Integer(5))
>>> P.M_triangle()                                                      #␣
↪needs sage.combinat
M: x^2*y^2 - 3*x*y^2 + 3*x*y + 2*y^2 - 3*y + 1
```

antichains(*element_constructor=None*)

Return the antichains of the poset.

An *antichain* of a poset is a set of elements of the poset that are pairwise incomparable.

INPUT:

- `element_constructor` – a function taking an iterable as argument (default: `list`)

OUTPUT:

The enumerated set (of type *PairwiseCompatibleSubsets*) of all antichains of the poset, each of which is given as an `element_constructor`.

EXAMPLES:

```
sage: A = posets.PentagonPoset().antichains(); A
Set of antichains of Finite lattice containing 5 elements
sage: list(A)
[[], [0], [1], [1, 2], [1, 3], [2], [3], [4]]
sage: A.cardinality()
8
```

(continues on next page)

(continued from previous page)

```
sage: A[3]
[1, 2]
```

```
>>> from sage.all import *
>>> A = posets.PentagonPoset().antichains(); A
Set of antichains of Finite lattice containing 5 elements
>>> list(A)
[[], [0], [1], [1, 2], [1, 3], [2], [3], [4]]
>>> A.cardinality()
8
>>> A[Integer(3)]
[1, 2]
```

To get the antichains as, say, sets, one may use the `element_constructor` option:

```
sage: list(posets.ChainPoset(3).antichains(element_constructor=set))
[set(), {0}, {1}, {2}]
```

```
>>> from sage.all import *
>>> list(posets.ChainPoset(Integer(3)).antichains(element_constructor=set))
[set(), {0}, {1}, {2}]
```

To get the antichains of a given size one can currently use:

```
sage: list(A.elements_of_depth_iterator(2))
[[1, 2], [1, 3]]
```

```
>>> from sage.all import *
>>> list(A.elements_of_depth_iterator(Integer(2)))
[[1, 2], [1, 3]]
```

Eventually the following syntax will be accepted:

```
sage: A.subset(size=2)                # not implemented
```

```
>>> from sage.all import *
>>> A.subset(size=Integer(2))              # not implemented
```

Note

Internally, this uses *sage.combinat.subsets_pairwise.PairwiseCompatibleSubsets* and `RecursivelyEnumeratedSet_forest`. At this point, iterating through this set is about twice slower than using *antichains_iterator()* (tested on `posets.AntichainPoset(15)`). The algorithm is the same (depth first search through the tree), but *antichains_iterator()* manually inlines things which apparently avoids some infrastructure overhead.

On the other hand, this returns a full featured enumerated set, with containment testing, etc.

See also

maximal_antichains(), *chains()*

antichains_iterator()

Return an iterator over the antichains of the poset.

EXAMPLES:

```
sage: it = posets.PentagonPoset().antichains_iterator(); it
<generator object ...antichains_iterator at ...>
sage: next(it), next(it)
([], [4])
```

```
>>> from sage.all import *
>>> it = posets.PentagonPoset().antichains_iterator(); it
<generator object ...antichains_iterator at ...>
>>> next(it), next(it)
([], [4])
```

See also:

antichains()

apozeta_polynomial()

Return the apozeta polynomial of the poset `self`.

The poset is assumed to be graded.

The apozeta polynomial of a poset is the unique polynomial $Z^a(q)$ such that for every integer $m > 1$, $Z^a(m)$ is the number of weakly increasing sequences $x_1 \leq x_2 \leq \cdots \leq x_{m-1}$ of elements of the poset whose largest element belongs to the top level of the poset.

When the poset P has a unique maximal element, this is equal to $Z(q-1)$ where Z is the zeta polynomial of P.

The name comes from the greek radical `apo`.

See also:

zeta_polynomial()

EXAMPLES:

```
sage: P = posets.NoncrossingPartitions(SymmetricGroup(4))
sage: P.apozeta_polynomial()
8/3*q^3 - 10*q^2 + 37/3*q - 5

sage: P = Poset({"a": "bc", "b": "d", "c": "de"})
sage: P.apozeta_polynomial()
3/2*q^2 - 5/2*q + 1
sage: P.zeta_polynomial()
3/2*q^2 - 1/2*q
```

```
>>> from sage.all import *
>>> P = posets.NoncrossingPartitions(SymmetricGroup(Integer(4)))
>>> P.apozeta_polynomial()
8/3*q^3 - 10*q^2 + 37/3*q - 5

>>> P = Poset({"a": "bc", "b": "d", "c": "de"})
>>> P.apozeta_polynomial()
3/2*q^2 - 5/2*q + 1
>>> P.zeta_polynomial()
3/2*q^2 - 1/2*q
```

atkinson(*a*)

Return the a-spectrum of a poset whose Hasse diagram is cycle-free as an undirected graph.

Given an element a in a poset P, the a-spectrum is the list of integers whose i-th term contains the number of linear extensions of P with element a located in the i-th position.

INPUT:

- `self` – a poset whose Hasse diagram is a forest
- `a` – an element of the poset

OUTPUT: the a-spectrum of this poset, returned as a list

EXAMPLES:

```
sage: P = Poset({0: [2], 1: [2], 2: [3, 4], 3: [], 4: []})
sage: P.atkinson(0)
[2, 2, 0, 0, 0]

sage: P = Poset({0: [1], 1: [2, 3], 2: [], 3: [], 4: [5, 6], 5: [], 6: []})
sage: P.atkinson(5)
[0, 10, 18, 24, 28, 30, 30]

sage: P = posets.AntichainPoset(10)
sage: P.atkinson(0)
[362880, 362880, 362880, 362880, 362880, 362880, 362880, 362880, 362880,␣
↪362880]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0): [Integer(2)], Integer(1): [Integer(2)],␣
↪Integer(2): [Integer(3), Integer(4)], Integer(3): [], Integer(4): []})
>>> P.atkinson(Integer(0))
[2, 2, 0, 0, 0]

>>> P = Poset({Integer(0): [Integer(1)], Integer(1): [Integer(2), Integer(3)],
↪ Integer(2): [], Integer(3): [], Integer(4): [Integer(5), Integer(6)],␣
↪Integer(5): [], Integer(6): []})
>>> P.atkinson(Integer(5))
[0, 10, 18, 24, 28, 30, 30]

>>> P = posets.AntichainPoset(Integer(10))
>>> P.atkinson(Integer(0))
[362880, 362880, 362880, 362880, 362880, 362880, 362880, 362880, 362880,␣
↪362880]
```

Note

This function is the implementation of the algorithm from [At1990].

bottom()

Return the unique minimal element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.bottom() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.bottom()
0
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)],Integer(4):[]})
>>> P.bottom() is None
True
>>> Q = Poset({Integer(0):[Integer(1)],Integer(1):[]})
>>> Q.bottom()
0
```

See also

has_bottom(), *top()*

canonical_label(*algorithm=None*)

Return the unique poset on the labels $\{0, \ldots, n-1\}$ (where n is the number of elements in the poset) that is isomorphic to this poset and invariant in the isomorphism class.

INPUT:

- algorithm – string (optional); a parameter forwarded to underlying graph function to select the algorithm to use

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: Q = P.canonical_label()
sage: sorted(Q.list())
[0, 1, 2, 3, 4, 5]
sage: Q.is_isomorphic(P)
True
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")), linear_
↪extension=True)
```

(continues on next page)

(continued from previous page)

```
>>> P.list()
[1, 2, 3, 4, 6, 12]
>>> Q = P.canonical_label()
>>> sorted(Q.list())
[0, 1, 2, 3, 4, 5]
>>> Q.is_isomorphic(P)
True
```

Canonical labeling of (semi)lattice returns (semi)lattice:

```
sage: D = DiGraph({'a':['b','c']})
sage: P = Poset(D)
sage: ML = MeetSemilattice(D)
sage: P.canonical_label()
Finite poset containing 3 elements
sage: ML.canonical_label()
Finite meet-semilattice containing 3 elements
```

```
>>> from sage.all import *
>>> D = DiGraph({'a':['b','c']})
>>> P = Poset(D)
>>> ML = MeetSemilattice(D)
>>> P.canonical_label()
Finite poset containing 3 elements
>>> ML.canonical_label()
Finite meet-semilattice containing 3 elements
```

See also

- Canonical labeling of directed graphs: `canonical_label()`

cardinality()

Return the number of elements in the poset.

EXAMPLES:

```
sage: Poset([[1,2,3],[4],[4],[4],[]]).cardinality()
5
```

```
>>> from sage.all import *
>>> Poset([[Integer(1),Integer(2),Integer(3)],[Integer(4)],[Integer(4)],
↪[Integer(4)],[]]).cardinality()
5
```

See also

`degree_polynomial()` for a more refined invariant

chain_polynomial()

Return the chain polynomial of the poset.

The coefficient of q^k is the number of chains of k elements in the poset. List of coefficients of this polynomial is also called a *f-vector* of the poset.

Note

This is not what has been called the chain polynomial in [St1986]. The latter is identical with the order polynomial in SageMath (`order_polynomial()`).

See also

`f_polynomial()`, `order_polynomial()`

EXAMPLES:

```
sage: P = posets.ChainPoset(3)
sage: t = P.chain_polynomial(); t
q^3 + 3*q^2 + 3*q + 1
sage: t(1) == len(list(P.chains()))
True

sage: P = posets.BooleanLattice(3)
sage: P.chain_polynomial()
6*q^4 + 18*q^3 + 19*q^2 + 8*q + 1

sage: P = posets.AntichainPoset(5)
sage: P.chain_polynomial()
5*q + 1
```

```
>>> from sage.all import *
>>> P = posets.ChainPoset(Integer(3))
>>> t = P.chain_polynomial(); t
q^3 + 3*q^2 + 3*q + 1
>>> t(Integer(1)) == len(list(P.chains()))
True

>>> P = posets.BooleanLattice(Integer(3))
>>> P.chain_polynomial()
6*q^4 + 18*q^3 + 19*q^2 + 8*q + 1

>>> P = posets.AntichainPoset(Integer(5))
>>> P.chain_polynomial()
5*q + 1
```

chain_polytope()

Return the chain polytope of the poset `self`.

The chain polytope of a finite poset P is defined as the subset of $\mathbf{R}^P$ consisting of all maps $x : P \to \mathbf{R}$ satisfying

$$x(p) \geq 0 \text{ for all } p \in P,$$

and

$$x(p_1) + x(p_2) + \ldots + x(p_k) \leq 1 \text{ for all chains } p_1 < p_2 < \ldots < p_k \text{ in } P.$$

This polytope was defined and studied in [St1986].

EXAMPLES:

```
sage: P = posets.AntichainPoset(3)
sage: Q = P.chain_polytope(); Q                                              #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: P = posets.PentagonPoset()
sage: Q = P.chain_polytope(); Q                                              #␣
↪needs sage.geometry.polyhedron
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 8 vertices
```

```
>>> from sage.all import *
>>> P = posets.AntichainPoset(Integer(3))
>>> Q = P.chain_polytope(); Q                                               #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
>>> P = posets.PentagonPoset()
>>> Q = P.chain_polytope(); Q                                               #␣
↪needs sage.geometry.polyhedron
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 8 vertices
```

chains(*element_constructor=None*, *exclude=None*)

Return the chains of the poset.

A *chain* of a poset is an increasing sequence of distinct elements of the poset.

INPUT:

- `element_constructor` – a function taking an iterable as argument (default: `list`)
- `exclude` – elements of the poset to be excluded (default: `None`)

OUTPUT:

The enumerated set (of type *PairwiseCompatibleSubsets*) of all chains of the poset, each of which is given as an `element_constructor`.

EXAMPLES:

```
sage: C = posets.PentagonPoset().chains(); C
Set of chains of Finite lattice containing 5 elements
sage: list(C)
[[], [0], [0, 1], [0, 1, 4], [0, 2], [0, 2, 3], [0, 2, 3, 4], [0, 2, 4],
 [0, 3], [0, 3, 4], [0, 4], [1], [1, 4], [2], [2, 3], [2, 3, 4], [2, 4],
 [3], [3, 4], [4]]
```

```
>>> from sage.all import *
>>> C = posets.PentagonPoset().chains(); C
Set of chains of Finite lattice containing 5 elements
>>> list(C)
[[], [0], [0, 1], [0, 1, 4], [0, 2], [0, 2, 3], [0, 2, 3, 4], [0, 2, 4],
```

(continues on next page)

(continued from previous page)

```
[0, 3], [0, 3, 4], [0, 4], [1], [1, 4], [2], [2, 3], [2, 3, 4], [2, 4],
[3], [3, 4], [4]]
```

Exclusion of elements, tuple (instead of list) as constructor:

```
sage: P = Poset({1: [2, 3], 2: [4], 3: [4, 5]})
sage: list(P.chains(element_constructor=tuple, exclude=[3]))
[(), (1,), (1, 2), (1, 2, 4), (1, 4), (1, 5), (2,), (2, 4), (4,), (5,)]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(2): [Integer(4)],
↪ Integer(3): [Integer(4), Integer(5)]})
>>> list(P.chains(element_constructor=tuple, exclude=[Integer(3)]))
[(), (1,), (1, 2), (1, 2, 4), (1, 4), (1, 5), (2,), (2, 4), (4,), (5,)]
```

To get the chains of a given size one can currently use:

```
sage: list(C.elements_of_depth_iterator(2))
[[0, 1], [0, 2], [0, 3], [0, 4], [1, 4], [2, 3], [2, 4], [3, 4]]
```

```
>>> from sage.all import *
>>> list(C.elements_of_depth_iterator(Integer(2)))
[[0, 1], [0, 2], [0, 3], [0, 4], [1, 4], [2, 3], [2, 4], [3, 4]]
```

Eventually the following syntax will be accepted:

```
sage: C.subset(size=2)                # not implemented
```

```
>>> from sage.all import *
>>> C.subset(size=Integer(2))                # not implemented
```

See also

maximal_chains(), *antichains()*

characteristic_polynomial()

Return the characteristic polynomial of the poset.

The poset is expected to be graded and have a bottom element.

If P is a graded poset with rank n and a unique minimal element $\hat{0}$, then the characteristic polynomial of P is defined to be

$$\sum_{x\in P} \mu(\hat{0}, x) q^{n-\rho(x)} \in \mathbf{Z}[q],$$

where ρ is the rank function, and μ is the Möbius function of P.

See section 3.10 of [EnumComb1].

EXAMPLES:

```
sage: P = posets.DiamondPoset(5)
sage: P.characteristic_polynomial()
q^2 - 3*q + 2

sage: P = Poset({1: [2, 3], 2: [4], 3: [5], 4: [6], 5: [6], 6: [7]})
sage: P.characteristic_polynomial()
q^4 - 2*q^3 + q
```

```
>>> from sage.all import *
>>> P = posets.DiamondPoset(Integer(5))
>>> P.characteristic_polynomial()
q^2 - 3*q + 2

>>> P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(2): [Integer(4)],
→ Integer(3): [Integer(5)], Integer(4): [Integer(6)], Integer(5):␣
→[Integer(6)], Integer(6): [Integer(7)]})
>>> P.characteristic_polynomial()
q^4 - 2*q^3 + q
```

closed_interval(*x*, *y*)

Return the list of elements z such that $x \leq z \leq y$ in the poset.

EXAMPLES:

```
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.closed_interval(2, 100)
[2, 4, 10, 20, 50, 100]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(1000)), attrcall("divides")))
>>> P.closed_interval(Integer(2), Integer(100))
[2, 4, 10, 20, 50, 100]
```

See also

open_interval()

common_lower_covers(*elmts*)

Return all of the common lower covers of the elements `elmts`.

EXAMPLES:

```
sage: P = Poset({0: [1,2], 1: [3], 2: [3], 3: []})
sage: P.common_lower_covers([1, 2])
[0]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0): [Integer(1),Integer(2)], Integer(1): [Integer(3)],␣
→Integer(2): [Integer(3)], Integer(3): []})
>>> P.common_lower_covers([Integer(1), Integer(2)])
[0]
```

common_upper_covers(*elmts*)

Return all of the common upper covers of the elements `elmts`.

EXAMPLES:

```
sage: P = Poset({0: [1,2], 1: [3], 2: [3], 3: []})
sage: P.common_upper_covers([1, 2])
[3]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0): [Integer(1),Integer(2)], Integer(1): [Integer(3)],␣
↪Integer(2): [Integer(3)], Integer(3): []})
>>> P.common_upper_covers([Integer(1), Integer(2)])
[3]
```

comparability_graph()

Return the comparability graph of the poset.

The comparability graph is an undirected graph where vertices are the elements of the poset and there is an edge between two vertices if they are comparable in the poset.

See Wikipedia article Comparability_graph

EXAMPLES:

```
sage: Y = Poset({1: [2], 2: [3, 4]})
sage: g = Y.comparability_graph(); g
Comparability graph on 4 vertices
sage: Y.compare_elements(1, 3) is not None
True
sage: g.has_edge(1, 3)
True
```

```
>>> from sage.all import *
>>> Y = Poset({Integer(1): [Integer(2)], Integer(2): [Integer(3), Integer(4)]}
↪)
>>> g = Y.comparability_graph(); g
Comparability graph on 4 vertices
>>> Y.compare_elements(Integer(1), Integer(3)) is not None
True
>>> g.has_edge(Integer(1), Integer(3))
True
```

See also

incomparability_graph(), `sage.graphs.comparability`

compare_elements(*x*, *y*)

Compare x and y in the poset.

- If $x < y$, return `-1`.
- If $x = y$, return `0`.
- If $x > y$, return `1`.

- If x and y are not comparable, return None.

EXAMPLES:

```
sage: P = Poset([[1, 2], [4], [3], [4], []])
sage: P.compare_elements(0, 0)
0
sage: P.compare_elements(0, 4)
-1
sage: P.compare_elements(4, 0)
1
sage: P.compare_elements(1, 2) is None
True
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1), Integer(2)], [Integer(4)], [Integer(3)],␣
↪[Integer(4)], []])
>>> P.compare_elements(Integer(0), Integer(0))
0
>>> P.compare_elements(Integer(0), Integer(4))
-1
>>> P.compare_elements(Integer(4), Integer(0))
1
>>> P.compare_elements(Integer(1), Integer(2)) is None
True
```

completion_by_cuts()

Return the completion by cuts of self.

This is the smallest lattice containing the poset. This is also called the Dedekind-MacNeille completion.

See the Wikipedia article Dedekind-MacNeille completion.

OUTPUT: a finite lattice

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.completion_by_cuts().is_isomorphic(P)
True

sage: Y = Poset({1: [2], 2: [3, 4]})
sage: trafficsign = LatticePoset({1: [2], 2: [3, 4], 3: [5], 4: [5]})
sage: L = Y.completion_by_cuts()
sage: L.is_isomorphic(trafficsign)
True

sage: P = posets.SymmetricGroupBruhatOrderPoset(3)
sage: Q = P.completion_by_cuts(); Q
Finite lattice containing 7 elements
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.completion_by_cuts().is_isomorphic(P)
True
```

(continues on next page)

(continued from previous page)

```
>>> Y = Poset({Integer(1): [Integer(2)], Integer(2): [Integer(3), Integer(4)]}
↪)
>>> trafficsign = LatticePoset({Integer(1): [Integer(2)], Integer(2):␣
↪[Integer(3), Integer(4)], Integer(3): [Integer(5)], Integer(4):␣
↪[Integer(5)]})
>>> L = Y.completion_by_cuts()
>>> L.is_isomorphic(trafficsign)
True

>>> P = posets.SymmetricGroupBruhatOrderPoset(Integer(3))
>>> Q = P.completion_by_cuts(); Q
Finite lattice containing 7 elements
```

See also

cuts(), irreducibles_poset()

connected_components()

Return the connected components of the poset as subposets.

EXAMPLES:

```
sage: P = Poset({1: [2, 3], 3: [4, 5], 6: [7, 8]})
sage: parts = sorted(P.connected_components(), key=len); parts
[Finite poset containing 3 elements,
 Finite poset containing 5 elements]
sage: parts[0].cover_relations()
[[6, 7], [6, 8]]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(3): [Integer(4),␣
↪Integer(5)], Integer(6): [Integer(7), Integer(8)]})
>>> parts = sorted(P.connected_components(), key=len); parts
[Finite poset containing 3 elements,
 Finite poset containing 5 elements]
>>> parts[Integer(0)].cover_relations()
[[6, 7], [6, 8]]
```

See also

disjoint_union(), *is_connected()*

cover_relations()

Return the list of pairs `[x, y]` of elements of the poset such that `y` covers `x`.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.cover_relations()
```

(continues on next page)

(continued from previous page)

```
[[1, 2], [0, 2], [2, 3], [3, 4]]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.cover_relations()
[[1, 2], [0, 2], [2, 3], [3, 4]]
```

cover_relations_graph()

Return the (undirected) graph of cover relations.

EXAMPLES:

```
sage: P = Poset({0: [1, 2], 1: [3], 2: [3]})
sage: G = P.cover_relations_graph(); G
Graph on 4 vertices
sage: G.has_edge(3, 1), G.has_edge(3, 0)
(True, False)
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0): [Integer(1), Integer(2)], Integer(1): [Integer(3)],
↪ Integer(2): [Integer(3)]})
>>> G = P.cover_relations_graph(); G
Graph on 4 vertices
>>> G.has_edge(Integer(3), Integer(1)), G.has_edge(Integer(3), Integer(0))
(True, False)
```

See also

hasse_diagram()

cover_relations_iterator()

Return an iterator over the cover relations of the poset.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: type(P.cover_relations_iterator())
<class 'generator'>
sage: [z for z in P.cover_relations_iterator()]
[[1, 2], [0, 2], [2, 3], [3, 4]]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> type(P.cover_relations_iterator())
<class 'generator'>
>>> [z for z in P.cover_relations_iterator()]
[[1, 2], [0, 2], [2, 3], [3, 4]]
```

covers(x, y)

Return `True` if `y` covers `x` and `False` otherwise.

Element y covers x if $x < y$ and there is no z such that $x < z < y$.

EXAMPLES:

```
sage: P = Poset([[1,5], [2,6], [3], [4], [], [6,3], [4]])
sage: P.covers(1, 6)
True
sage: P.covers(1, 4)
False
sage: P.covers(1, 5)
False
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(5)], [Integer(2),Integer(6)], [Integer(3)],
↪ [Integer(4)], [], [Integer(6),Integer(3)], [Integer(4)]])
>>> P.covers(Integer(1), Integer(6))
True
>>> P.covers(Integer(1), Integer(4))
False
>>> P.covers(Integer(1), Integer(5))
False
```

coxeter_polynomial(*algorithm='sage'*)

Return the Coxeter polynomial of the poset.

INPUT:

- `algorithm` – optional (default: `"sage"`) ; the unique other option is `"magma"`

OUTPUT: a polynomial in one variable

The output is the characteristic polynomial of the Coxeter transformation. This polynomial only depends on the derived category of modules on the poset.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.coxeter_polynomial()                                          #␣
↪needs sage.libs.flint
x^5 + x^4 + x + 1

sage: p = posets.SymmetricGroupWeakOrderPoset(3)                      #␣
↪needs sage.groups
sage: p.coxeter_polynomial()                                          #␣
↪needs sage.groups sage.libs.flint
x^6 + x^5 - x^3 + x + 1
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.coxeter_polynomial()                                          #␣
↪needs sage.libs.flint
x^5 + x^4 + x + 1
```

(continues on next page)

(continued from previous page)

```
>>> p = posets.SymmetricGroupWeakOrderPoset(Integer(3))
↪     # needs sage.groups
>>> p.coxeter_polynomial()                                                  #␣
↪needs sage.groups sage.libs.flint
x^6 + x^5 - x^3 + x + 1
```

See also

coxeter_transformation(), *coxeter_smith_form()*

coxeter_smith_form(*algorithm='singular'*)

Return the Smith normal form of x minus the Coxeter transformation matrix.

INPUT:

- `algorithm` – (default: `'singular'`) possible values are `'singular'`, `'sage'`, `'gap'`, `'pari'`, `'maple'`, `'magma'`, `'fricas'`

Beware that speed depends very much on the choice of algorithm. Sage is rather slow, Singular is faster and Pari is fast at least for small sizes.

OUTPUT: list of polynomials in one variable, each one dividing the next one

The output list is a refinement of the characteristic polynomial of the Coxeter transformation, which is its product. This list of polynomials only depends on the derived category of modules on the poset.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.coxeter_smith_form()
↪# needs sage.libs.singular
[1, 1, 1, 1, x^5 + x^4 + x + 1]

sage: P = posets.DiamondPoset(7)
sage: prod(P.coxeter_smith_form()) == P.coxeter_polynomial()
↪# needs sage.libs.singular
True
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.coxeter_smith_form()                                                  #␣
↪needs sage.libs.singular
[1, 1, 1, 1, x^5 + x^4 + x + 1]

>>> P = posets.DiamondPoset(Integer(7))
>>> prod(P.coxeter_smith_form()) == P.coxeter_polynomial()                  #␣
↪needs sage.libs.singular
True
```

See also

coxeter_transformation(), `coxeter_matrix()`

coxeter_transformation()

Return the Coxeter transformation of the poset.

OUTPUT: a square matrix with integer coefficients

The output is the matrix of the Auslander-Reiten translation acting on the Grothendieck group of the derived category of modules on the poset, in the basis of simple modules. This matrix is usually called the Coxeter transformation.

EXAMPLES:

```
sage: posets.PentagonPoset().coxeter_transformation()                    #␣
↪needs sage.libs.flint
[ 0  0  0  0 -1]
[ 0  0  0  1 -1]
[ 0  1  0  0 -1]
[-1  1  1  0 -1]
[-1  1  0  1 -1]
```

```
>>> from sage.all import *
>>> posets.PentagonPoset().coxeter_transformation()                    #␣
↪needs sage.libs.flint
[ 0  0  0  0 -1]
[ 0  0  0  1 -1]
[ 0  1  0  0 -1]
[-1  1  1  0 -1]
[-1  1  0  1 -1]
```

See also

coxeter_polynomial(), *coxeter_smith_form()*

cuts()

Return the list of cuts of the poset `self`.

A cut is a subset A of `self` such that the set of lower bounds of the set of upper bounds of A is exactly A.

The cuts are computed as the smallest family of subsets of P containing its principal order filters, the whole set P and which is closed by intersection.

EXAMPLES:

```
sage: P = posets.AntichainPoset(3)
sage: Pc = P.cuts()
sage: Pc # random
[frozenset({2}),
 frozenset({1}),
 frozenset({0}),
 frozenset(),
 frozenset({0, 1, 2})]
sage: sorted(list(c) for c in Pc)
[[], [0], [0, 1, 2], [1], [2]]
```

```
>>> from sage.all import *
>>> P = posets.AntichainPoset(Integer(3))
>>> Pc = P.cuts()
>>> Pc # random
[frozenset({2}),
 frozenset({1}),
 frozenset({0}),
 frozenset(),
 frozenset({0, 1, 2})]
>>> sorted(list(c) for c in Pc)
[[], [0], [0, 1, 2], [1], [2]]
```

See also:

completion_by_cuts()

degree_polynomial()

Return the generating polynomial of degrees of vertices in `self`.

This is the sum

$$\sum_{v \in P} x^{\mathrm{in}(v)} y^{\mathrm{out}(v)},$$

where `in(v)` and `out(v)` are the number of incoming and outgoing edges at vertex v in the Hasse diagram of P.

Because this polynomial is multiplicative for Cartesian product of posets, it is useful to help see if the poset can be isomorphic to a Cartesian product.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.degree_polynomial()
x^2 + 3*x*y + y^2

sage: P = posets.BooleanLattice(4)
sage: P.degree_polynomial().factor()
(x + y)^4
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.degree_polynomial()
x^2 + 3*x*y + y^2

>>> P = posets.BooleanLattice(Integer(4))
>>> P.degree_polynomial().factor()
(x + y)^4
```

See also:

cardinality() for the value at $(x, y) = (1, 1)$

diamonds()

Return the list of diamonds of `self`.

A diamond is the following subgraph of the Hasse diagram:

```
  z
 / \
x   y
 \ /
  w
```

Thus each edge represents a cover relation in the Hasse diagram. We represent this as the tuple (w, x, y, z).

OUTPUT: a tuple with

- a list of all diamonds in the Hasse Diagram,
- a boolean checking that every w, x, y that form a `V`, there is a unique element z, which completes the diamond.

EXAMPLES:

```
sage: P = Poset({0: [1,2], 1: [3], 2: [3], 3: []})
sage: P.diamonds()
([(0, 1, 2, 3)], True)

sage: P = posets.YoungDiagramPoset(Partition([3, 2, 2]))                    #␣
↪needs sage.combinat
sage: P.diamonds()                                                           #␣
↪needs sage.combinat
([((0, 0), (0, 1), (1, 0), (1, 1)), ((1, 0), (1, 1), (2, 0), (2, 1))], False)
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0): [Integer(1),Integer(2)], Integer(1): [Integer(3)],␣
↪Integer(2): [Integer(3)], Integer(3): []})
>>> P.diamonds()
([(0, 1, 2, 3)], True)

>>> P = posets.YoungDiagramPoset(Partition([Integer(3), Integer(2),␣
↪Integer(2)]))                   # needs sage.combinat
>>> P.diamonds()                                                          #␣
↪needs sage.combinat
([((0, 0), (0, 1), (1, 0), (1, 1)), ((1, 0), (1, 1), (2, 0), (2, 1))], False)
```

dilworth_decomposition()

Return a partition of the points into the minimal number of chains.

According to Dilworth's theorem, the points of a poset can be partitioned into α chains, where α is the cardinality of its largest antichain. This method returns such a partition.

See Wikipedia article Dilworth%27s_theorem.

ALGORITHM:

We build a bipartite graph in which a vertex v of the poset is represented by two vertices v^-, v^+. For any two u, v such that $u < v$ in the poset we add an edge v^+u^-.

A matching in this graph is equivalent to a partition of the poset into chains: indeed, a chain $v_1...v_k$ gives rise to the matching $v_1^+v_2^-, v_2^+v_3^-, ...$, and from a matching one can build the union of chains.

According to Dilworth's theorem, the number of chains is equal to α (the posets' width).

EXAMPLES:

```
sage: p = posets.BooleanLattice(4)
sage: p.width()                                                         #␣
↪needs networkx
6
sage: p.dilworth_decomposition()  # random                              #␣
↪needs networkx
[[7, 6, 4], [11, 3], [12, 8, 0], [13, 9, 1], [14, 10, 2], [15, 5]]
```

```
>>> from sage.all import *
>>> p = posets.BooleanLattice(Integer(4))
>>> p.width()                                                         #␣
↪needs networkx
6
>>> p.dilworth_decomposition()  # random                              #␣
↪needs networkx
[[7, 6, 4], [11, 3], [12, 8, 0], [13, 9, 1], [14, 10, 2], [15, 5]]
```

See also

`level_sets()` to return elements grouped to antichains.

dimension(*certificate*, *solver*, *integrality_tolerance=False*)

Return the dimension of the Poset.

The (Dushnik-Miller) dimension of a poset is the minimal number of total orders so that the poset is their "intersection". More precisely, the dimension of a poset defined on a set X of points is the smallest integer n such that there exist linear extensions $P_1, ..., P_n$ of P satisfying:

$$u \leq_P v \text{ if and only if } \forall i, u \leq_{P_i} v$$

For more information, see the Wikipedia article Order_dimension.

INPUT:

- `certificate` – boolean (default: `False`); whether to return an integer (the dimension) or a certificate, i.e. a smallest set of linear extensions
- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`

Note

The speed of this function greatly improves when more efficient MILP solvers (e.g. Gurobi, CPLEX) are installed. See `MixedIntegerLinearProgram` for more information.

Note

Prior to version 8.3 this returned only realizer with `certificate=True`. Now it returns a pair having a realizer as the second element. See Issue #25588 for details.

ALGORITHM:

As explained [FT00], the dimension of a poset is equal to the (weak) chromatic number of a hypergraph. More precisely:

> Let $inc(P)$ be the set of (ordered) pairs of incomparable elements of P, i.e. all uv and vu such that $u \not\leq_P v$ and $v \not\leq_P u$. Any linear extension of P is a total order on X that can be seen as the union of relations from P along with some relations from $inc(P)$. Thus, the dimension of P is the smallest number of linear extensions of P which *cover* all points of $inc(P)$.
>
> Consequently, $dim(P)$ is equal to the chromatic number of the hypergraph $\mathcal{H}_{inc}$, where $\mathcal{H}_{inc}$ is the hypergraph defined on $inc(P)$ whose sets are all $S \subseteq inc(P)$ such that $P \cup S$ is not acyclic.

We solve this problem through a `Mixed Integer Linear Program`.

The problem is known to be NP-complete.

EXAMPLES:

We create a poset, compute a set of linear extensions and check that we get back the poset from them:

```
sage: P = Poset([[1,4], [3], [4,5,3], [6], [], [6], []])
sage: P.dimension()                                                             #␣
↪needs networkx
3
sage: dim, L = P.dimension(certificate=True)                                    #␣
↪needs sage.numerical.mip
sage: L  # random -- architecture-dependent                                     #␣
↪needs sage.numerical.mip
[[0, 2, 4, 5, 1, 3, 6], [2, 5, 0, 1, 3, 4, 6], [0, 1, 2, 3, 5, 6, 4]]
sage: Poset( (L[0], lambda x, y: all(l.index(x) < l.index(y) for l in L)) )␣
↪== P         # needs sage.numerical.mip
True
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(4)], [Integer(3)], [Integer(4),Integer(5),
↪Integer(3)], [Integer(6)], [], [Integer(6)], []])
>>> P.dimension()                                                             #␣
↪needs networkx
3
>>> dim, L = P.dimension(certificate=True)                                    #␣
↪needs sage.numerical.mip
>>> L  # random -- architecture-dependent                                     #␣
↪needs sage.numerical.mip
[[0, 2, 4, 5, 1, 3, 6], [2, 5, 0, 1, 3, 4, 6], [0, 1, 2, 3, 5, 6, 4]]
>>> Poset( (L[Integer(0)], lambda x, y: all(l.index(x) < l.index(y) for l in␣
↪L)) ) == P         # needs sage.numerical.mip
True
```

According to Schnyder's theorem, the incidence poset (of height 2) of a graph has dimension ≤ 3 if and only if the graph is planar:

```
sage: G = graphs.CompleteGraph(4)
sage: P = Poset(DiGraph({(u,v):[u,v] for u,v,_ in G.edges(sort=True)}))
sage: P.dimension()                                                     #␣
↪needs networkx
3

sage: G = graphs.CompleteBipartiteGraph(3,3)
sage: P = Poset(DiGraph({(u,v):[u,v] for u,v,_ in G.edges(sort=True)}))
sage: P.dimension()   # not tested (around 4s with CPLEX)
4
```

```
>>> from sage.all import *
>>> G = graphs.CompleteGraph(Integer(4))
>>> P = Poset(DiGraph({(u,v):[u,v] for u,v,_ in G.edges(sort=True)}))
>>> P.dimension()                                                     #␣
↪needs networkx
3

>>> G = graphs.CompleteBipartiteGraph(Integer(3),Integer(3))
>>> P = Poset(DiGraph({(u,v):[u,v] for u,v,_ in G.edges(sort=True)}))
>>> P.dimension()   # not tested (around 4s with CPLEX)
4
```

disjoint_union(*other*, *labels='pairs'*)

Return a poset isomorphic to disjoint union (also called direct sum) of the poset with `other`.

The disjoint union of P and Q is a poset that contains every element and relation from both P and Q, and where every element of P is incomparable to every element of Q.

Mathematically, it is only defined when P and Q have no common element; here we force that by giving them different names in the resulting poset.

INPUT:

- `other` – poset
- `labels` – (defaults to 'pairs') If set to 'pairs', each element `v` in this poset will be named `(0,v)` and each element `u` in `other` will be named `(1,u)` in the result. If set to 'integers', the elements of the result will be relabeled with consecutive integers.

EXAMPLES:

```
sage: P1 = Poset({'a': 'b'})
sage: P2 = Poset({'c': 'd'})
sage: P = P1.disjoint_union(P2); P
Finite poset containing 4 elements
sage: sorted(P.cover_relations())
[[(0, 'a'), (0, 'b')], [(1, 'c'), (1, 'd')]]
sage: P = P1.disjoint_union(P2, labels='integers')
sage: P.cover_relations()
[[2, 3], [0, 1]]

sage: N5 = posets.PentagonPoset(); N5
Finite lattice containing 5 elements
sage: N5.disjoint_union(N5)   # Union of lattices is not a lattice
Finite poset containing 10 elements
```

```
>>> from sage.all import *
>>> P1 = Poset({'a': 'b'})
>>> P2 = Poset({'c': 'd'})
>>> P = P1.disjoint_union(P2); P
Finite poset containing 4 elements
>>> sorted(P.cover_relations())
[[(0, 'a'), (0, 'b')], [(1, 'c'), (1, 'd')]]
>>> P = P1.disjoint_union(P2, labels='integers')
>>> P.cover_relations()
[[2, 3], [0, 1]]

>>> N5 = posets.PentagonPoset(); N5
Finite lattice containing 5 elements
>>> N5.disjoint_union(N5)  # Union of lattices is not a lattice
Finite poset containing 10 elements
```

We show how to get literally direct sum with elements untouched:

```
sage: P = P1.disjoint_union(P2).relabel(lambda x: x[1])
sage: sorted(P.cover_relations())
[['a', 'b'], ['c', 'd']]
```

```
>>> from sage.all import *
>>> P = P1.disjoint_union(P2).relabel(lambda x: x[Integer(1)])
>>> sorted(P.cover_relations())
[['a', 'b'], ['c', 'd']]
```

See also

connected_components()

dual()

Return the dual poset of the given poset.

In the dual of a poset P we have $x \leq y$ iff $y \leq x$ in P.

EXAMPLES:

```
sage: P = Poset({1: [2, 3], 3: [4]})
sage: P.cover_relations()
[[1, 2], [1, 3], [3, 4]]
sage: Q = P.dual()
sage: Q.cover_relations()
[[4, 3], [3, 1], [2, 1]]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(3): [Integer(4)]}
↪)
>>> P.cover_relations()
[[1, 2], [1, 3], [3, 4]]
>>> Q = P.dual()
```

(continues on next page)

(continued from previous page)

```
>>> Q.cover_relations()
[[4, 3], [3, 1], [2, 1]]
```

Dual of a lattice is a lattice; dual of a meet-semilattice is join-semilattice and vice versa. Also the dual of a (non-)facade poset is again (non-)facade:

```
sage: V = MeetSemilattice({1: [2, 3]}, facade=False)
sage: A = V.dual(); A
Finite join-semilattice containing 3 elements
sage: A(2) < A(1)
True
```

```
>>> from sage.all import *
>>> V = MeetSemilattice({Integer(1): [Integer(2), Integer(3)]}, facade=False)
>>> A = V.dual(); A
Finite join-semilattice containing 3 elements
>>> A(Integer(2)) < A(Integer(1))
True
```

See also

`is_self_dual()`

evacuation()

Compute evacuation on the linear extension associated to the poset `self`.

OUTPUT: an isomorphic poset, with the same default linear extension

Evacuation is defined on a poset `self` of size n by applying the evacuation operator $(\tau_1 \cdots \tau_{n-1})(\tau_1 \cdots \tau_{n-2}) \cdots (\tau_1)$, to the default linear extension π of `self` (see *evacuation()*), and relabeling `self` accordingly. For more details see [Stan2009].

EXAMPLES:

```
sage: P = Poset(([1,2], [[1,2]]), linear_extension=True, facade=False)
sage: P.evacuation()
Finite poset containing 2 elements with distinguished linear extension
sage: P.evacuation() == P
True

sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,
↪6]]), linear_extension=True, facade=False)
sage: P.list()
[1, 2, 3, 4, 5, 6, 7]
sage: Q = P.evacuation(); Q
Finite poset containing 7 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 3], [2, 5], [3, 4], [3, 6], [4, 7], [6, 7]]
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2)], [[Integer(1),Integer(2)]]), linear_
↪extension=True, facade=False)
```

(continues on next page)

(continued from previous page)

```
>>> P.evacuation()
Finite poset containing 2 elements with distinguished linear extension
>>> P.evacuation() == P
True

>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7)], [[Integer(1),Integer(2)],[Integer(1),Integer(4)],
↪[Integer(2),Integer(3)],[Integer(2),Integer(5)],[Integer(3),Integer(6)],
↪[Integer(4),Integer(7)],[Integer(5),Integer(6)]]), linear_extension=True,␣
↪facade=False)
>>> P.list()
[1, 2, 3, 4, 5, 6, 7]
>>> Q = P.evacuation(); Q
Finite poset containing 7 elements with distinguished linear extension
>>> Q.cover_relations()
[[1, 2], [1, 3], [2, 5], [3, 4], [3, 6], [4, 7], [6, 7]]
```

Note that the results depend on the linear extension associated to the poset:

```
sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,
↪6]]))
sage: P.list()
[1, 2, 3, 5, 6, 4, 7]
sage: Q = P.evacuation(); Q
Finite poset containing 7 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 5], [2, 3], [5, 6], [5, 4], [6, 7], [4, 7]]
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7)], [[Integer(1),Integer(2)],[Integer(1),Integer(4)],
↪[Integer(2),Integer(3)],[Integer(2),Integer(5)],[Integer(3),Integer(6)],
↪[Integer(4),Integer(7)],[Integer(5),Integer(6)]]))
>>> P.list()
[1, 2, 3, 5, 6, 4, 7]
>>> Q = P.evacuation(); Q
Finite poset containing 7 elements with distinguished linear extension
>>> Q.cover_relations()
[[1, 2], [1, 5], [2, 3], [5, 6], [5, 4], [6, 7], [4, 7]]
```

Here is an example of a poset where the elements are not labelled by $\{1, 2, \ldots, n\}$:

```
sage: P = Poset((divisors(15), attrcall("divides")), linear_extension = True)
sage: P.list()
[1, 3, 5, 15]
sage: Q = P.evacuation(); Q
Finite poset containing 4 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 3], [1, 5], [3, 15], [5, 15]]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(15)), attrcall("divides")), linear_extension␣
```

(continues on next page)

(continued from previous page)

```
↪= True)
>>> P.list()
[1, 3, 5, 15]
>>> Q = P.evacuation(); Q
Finite poset containing 4 elements with distinguished linear extension
>>> Q.cover_relations()
[[1, 3], [1, 5], [3, 15], [5, 15]]
```

See also

- `linear_extension()`
- `with_linear_extension()` and the `linear_extension` option of `Poset()`
- `evacuation()`
- `promotion()`

AUTHOR:

- Anne Schilling (2012-02-18)

f_polynomial()

Return the f-polynomial of the poset.

The poset is expected to be bounded.

This is the f-polynomial of the order complex of the poset minus its bounds.

The coefficient of q^i is the number of chains of $i + 1$ elements containing both bounds of the poset.

Note

This is slightly different from the `fPolynomial` method in Macaulay2.

EXAMPLES:

```
sage: P = posets.DiamondPoset(5)
sage: P.f_polynomial()
3*q^2 + q

sage: P = Poset({1: [2, 3], 2: [4], 3: [5], 4: [6], 5: [7], 6: [7]})
sage: P.f_polynomial()
q^4 + 4*q^3 + 5*q^2 + q
```

```
>>> from sage.all import *
>>> P = posets.DiamondPoset(Integer(5))
>>> P.f_polynomial()
3*q^2 + q

>>> P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(2): [Integer(4)],
↪ Integer(3): [Integer(5)], Integer(4): [Integer(6)], Integer(5):␣
↪[Integer(7)], Integer(6): [Integer(7)]})
```

(continues on next page)

(continued from previous page)

```
>>> P.f_polynomial()
q^4 + 4*q^3 + 5*q^2 + q
```

See also

is_bounded(), *h_polynomial()*, *order_complex()*, sage.topology.cell_complex.GenericCellComplex.f_vector()

factor()

Factor the poset as a Cartesian product of smaller posets.

This only works for connected posets.

The decomposition of a connected poset as a Cartesian product of posets (prime in the sense that they cannot be written as Cartesian products) is unique up to reordering and isomorphism.

OUTPUT: list of posets

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: Q = P*P
sage: Q.factor()
[Finite poset containing 5 elements,
 Finite poset containing 5 elements]

sage: P1 = posets.ChainPoset(3)
sage: P2 = posets.ChainPoset(7)
sage: P1.factor()
[Finite lattice containing 3 elements]
sage: (P1 * P2).factor()
[Finite poset containing 7 elements,
 Finite poset containing 3 elements]

sage: P = posets.TamariLattice(4)
sage: (P*P).factor()
[Finite poset containing 14 elements,
Finite poset containing 14 elements]
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> Q = P*P
>>> Q.factor()
[Finite poset containing 5 elements,
 Finite poset containing 5 elements]

>>> P1 = posets.ChainPoset(Integer(3))
>>> P2 = posets.ChainPoset(Integer(7))
>>> P1.factor()
[Finite lattice containing 3 elements]
>>> (P1 * P2).factor()
[Finite poset containing 7 elements,
```

(continues on next page)

(continued from previous page)

```
 Finite poset containing 3 elements]

>>> P = posets.TamariLattice(Integer(4))
>>> (P*P).factor()
[Finite poset containing 14 elements,
Finite poset containing 14 elements]
```

See also

product()

REFERENCES:

flag_f_polynomial()

Return the flag f-polynomial of the poset.

The poset is expected to be bounded and ranked.

This is the sum, over all chains containing both bounds, of a monomial encoding the ranks of the elements of the chain.

More precisely, if P is a bounded ranked poset, then the flag f-polynomial of P is defined as the polynomial

$$\sum_{\substack{p_0 < p_1 < \ldots < p_k, \\ p_0 = \min P,\ p_k = \max P}} x_{\rho(p_1)} x_{\rho(p_2)} \cdots x_{\rho(p_k)} \in \mathbf{Z}[x_1, x_2, \cdots, x_n],$$

where $\min P$ and $\max P$ are (respectively) the minimum and the maximum of P, where ρ is the rank function of P (normalized to satisfy $\rho(\min P) = 0$), and where n is the rank of $\max P$. (Note that the indeterminate x_0 does not actually appear in the polynomial.)

For technical reasons, the polynomial is returned in the slightly larger ring $\mathbf{Z}[x_0, x_1, x_2, \cdots, x_{n+1}]$ by this method.

See Wikipedia article h-vector.

EXAMPLES:

```
sage: P = posets.DiamondPoset(5)
sage: P.flag_f_polynomial()
3*x1*x2 + x2

sage: P = Poset({1: [2, 3], 2: [4], 3: [5], 4: [6], 5: [6]})
sage: fl = P.flag_f_polynomial(); fl
2*x1*x2*x3 + 2*x1*x3 + 2*x2*x3 + x3
sage: q = polygen(ZZ,'q')
sage: fl(q,q,q,q) == P.f_polynomial()
True

sage: P = Poset({1: [2, 3, 4], 2: [5], 3: [5], 4: [5], 5: [6]})
sage: P.flag_f_polynomial()
3*x1*x2*x3 + 3*x1*x3 + x2*x3 + x3
```

```
>>> from sage.all import *
>>> P = posets.DiamondPoset(Integer(5))
```

(continues on next page)

(continued from previous page)

```
>>> P.flag_f_polynomial()
3*x1*x2 + x2

>>> P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(2): [Integer(4)],
↪ Integer(3): [Integer(5)], Integer(4): [Integer(6)], Integer(5):␣
↪[Integer(6)]})
>>> fl = P.flag_f_polynomial(); fl
2*x1*x2*x3 + 2*x1*x3 + 2*x2*x3 + x3
>>> q = polygen(ZZ,'q')
>>> fl(q,q,q,q) == P.f_polynomial()
True

>>> P = Poset({Integer(1): [Integer(2), Integer(3), Integer(4)], Integer(2):␣
↪[Integer(5)], Integer(3): [Integer(5)], Integer(4): [Integer(5)],␣
↪Integer(5): [Integer(6)]})
>>> P.flag_f_polynomial()
3*x1*x2*x3 + 3*x1*x3 + x2*x3 + x3
```

See also

is_bounded(), *flag_h_polynomial()*

flag_h_polynomial()

Return the flag h-polynomial of the poset.

The poset is expected to be bounded and ranked.

If P is a bounded ranked poset whose maximal element has rank n (where the minimal element is set to have rank 0), then the flag h-polynomial of P is defined as the polynomial

$$\prod_{k=1}^{n}(1-x_k)\cdot f\left(\frac{x_1}{1-x_1},\frac{x_2}{1-x_2},\cdots,\frac{x_n}{1-x_n}\right)\in \mathbf{Z}[x_1,x_2,\cdots,x_n],$$

where f is the flag f-polynomial of P (see *flag_f_polynomial()*).

For technical reasons, the polynomial is returned in the slightly larger ring $\mathbf{Q}[x_0,x_1,x_2,\cdots,x_{n+1}]$ by this method.

See Wikipedia article h-vector.

EXAMPLES:

```
sage: P = posets.DiamondPoset(5)
sage: P.flag_h_polynomial()
2*x1*x2 + x2

sage: P = Poset({1: [2, 3], 2: [4], 3: [5], 4: [6], 5: [6]})
sage: fl = P.flag_h_polynomial(); fl
-x1*x2*x3 + x1*x3 + x2*x3 + x3
sage: q = polygen(ZZ,'q')
sage: fl(q,q,q,q) == P.h_polynomial()
True
```

(continues on next page)

(continued from previous page)

```
sage: P = Poset({1: [2, 3, 4], 2: [5], 3: [5], 4: [5], 5: [6]})
sage: P.flag_h_polynomial()
2*x1*x3 + x3

sage: P = posets.ChainPoset(4)
sage: P.flag_h_polynomial()
x3
```

```
>>> from sage.all import *
>>> P = posets.DiamondPoset(Integer(5))
>>> P.flag_h_polynomial()
2*x1*x2 + x2

>>> P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(2): [Integer(4)],
↪ Integer(3): [Integer(5)], Integer(4): [Integer(6)], Integer(5):␣
↪[Integer(6)]})
>>> fl = P.flag_h_polynomial(); fl
-x1*x2*x3 + x1*x3 + x2*x3 + x3
>>> q = polygen(ZZ,'q')
>>> fl(q,q,q,q) == P.h_polynomial()
True

>>> P = Poset({Integer(1): [Integer(2), Integer(3), Integer(4)], Integer(2):␣
↪[Integer(5)], Integer(3): [Integer(5)], Integer(4): [Integer(5)],␣
↪Integer(5): [Integer(6)]})
>>> P.flag_h_polynomial()
2*x1*x3 + x3

>>> P = posets.ChainPoset(Integer(4))
>>> P.flag_h_polynomial()
x3
```

See also

is_bounded(), *flag_f_polynomial()*

frank_network()

Return Frank's network of the poset.

This is defined in Section 8 of [BF1999].

OUTPUT:

A pair (G, e), where G is Frank's network of P encoded as a `DiGraph`, and e is the cost function on its edges encoded as a dictionary (indexed by these edges, which in turn are encoded as tuples of 2 vertices).

Note

Frank's network of P is a certain directed graph with $2|P|+2$ vertices, defined in Section 8 of [BF1999]. Its set of vertices consists of two vertices $(0, p)$ and $(1, p)$ for each element p of P, as well as two vertices $(-1, 0)$ and $(2, 0)$. (These notations are not the ones used in [BF1999]; see the table below for their relation.) The edges are:

- for each p in P, an edge from $(-1, 0)$ to $(0, p)$;
- for each p in P, an edge from $(1, p)$ to $(2, 0)$;
- for each p and q in P such that $p \geq q$, an edge from $(0, p)$ to $(1, q)$.

We make this digraph into a network in the sense of flow theory as follows: The vertex $(-1, 0)$ is considered as the source of this network, and the vertex $(2, 0)$ as the sink. The cost function is defined to be 1 on the edge from $(0, p)$ to $(1, p)$ for each $p \in P$, and to be 0 on every other edge. The capacity is 1 on each edge. Here is how to translate this notations into that used in [BF1999]:

```
our notations                     [BF1999]
   (-1, 0)                          s
   (0, p)                          x_p
   (1, p)                          y_p
   (2, 0)                           t
    a[e]                           a(e)
```

EXAMPLES:

```
sage: ps = [[16,12,14,-13],[[12,14],[14,-13],[12,16],[16,-13]]]
sage: G, e = Poset(ps).frank_network()
sage: G.edges(sort=True)
[((-1, 0), (0, -13), None), ((-1, 0), (0, 12), None),
 ((-1, 0), (0, 14), None), ((-1, 0), (0, 16), None),
 ((0, -13), (1, -13), None), ((0, -13), (1, 12), None),
 ((0, -13), (1, 14), None), ((0, -13), (1, 16), None),
 ((0, 12), (1, 12), None), ((0, 14), (1, 12), None),
 ((0, 14), (1, 14), None), ((0, 16), (1, 12), None),
 ((0, 16), (1, 16), None), ((1, -13), (2, 0), None),
 ((1, 12), (2, 0), None), ((1, 14), (2, 0), None),
 ((1, 16), (2, 0), None)]
sage: e
{((-1, 0), (0, -13)): 0,
 ((-1, 0), (0, 12)): 0,
 ((-1, 0), (0, 14)): 0,
 ((-1, 0), (0, 16)): 0,
 ((0, -13), (1, -13)): 1,
 ((0, -13), (1, 12)): 0,
 ((0, -13), (1, 14)): 0,
 ((0, -13), (1, 16)): 0,
 ((0, 12), (1, 12)): 1,
 ((0, 14), (1, 12)): 0,
 ((0, 14), (1, 14)): 1,
 ((0, 16), (1, 12)): 0,
 ((0, 16), (1, 16)): 1,
 ((1, -13), (2, 0)): 0,
 ((1, 12), (2, 0)): 0,
 ((1, 14), (2, 0)): 0,
 ((1, 16), (2, 0)): 0}
sage: qs = [[1,2,3,4,5,6,7,8,9],[[1,3],[3,4],[5,7],[1,9],[2,3]]]
sage: Poset(qs).frank_network()
(Digraph on 20 vertices,
```

(continues on next page)

(continued from previous page)

```
{((-1, 0), (0, 1)): 0,
 ((-1, 0), (0, 2)): 0,
 ((-1, 0), (0, 3)): 0,
 ((-1, 0), (0, 4)): 0,
 ((-1, 0), (0, 5)): 0,
 ((-1, 0), (0, 6)): 0,
 ((-1, 0), (0, 7)): 0,
 ((-1, 0), (0, 8)): 0,
 ((-1, 0), (0, 9)): 0,
 ((0, 1), (1, 1)): 1,
 ((0, 2), (1, 2)): 1,
 ((0, 3), (1, 1)): 0,
 ((0, 3), (1, 2)): 0,
 ((0, 3), (1, 3)): 1,
 ((0, 4), (1, 1)): 0,
 ((0, 4), (1, 2)): 0,
 ((0, 4), (1, 3)): 0,
 ((0, 4), (1, 4)): 1,
 ((0, 5), (1, 5)): 1,
 ((0, 6), (1, 6)): 1,
 ((0, 7), (1, 5)): 0,
 ((0, 7), (1, 7)): 1,
 ((0, 8), (1, 8)): 1,
 ((0, 9), (1, 1)): 0,
 ((0, 9), (1, 9)): 1,
 ((1, 1), (2, 0)): 0,
 ((1, 2), (2, 0)): 0,
 ((1, 3), (2, 0)): 0,
 ((1, 4), (2, 0)): 0,
 ((1, 5), (2, 0)): 0,
 ((1, 6), (2, 0)): 0,
 ((1, 7), (2, 0)): 0,
 ((1, 8), (2, 0)): 0,
 ((1, 9), (2, 0)): 0})
```

```
>>> from sage.all import *
>>> ps = [[Integer(16),Integer(12),Integer(14),-Integer(13)],[[Integer(12),
↪Integer(14)],[Integer(14),-Integer(13)],[Integer(12),Integer(16)],
↪[Integer(16),-Integer(13)]]]
>>> G, e = Poset(ps).frank_network()
>>> G.edges(sort=True)
[((-1, 0), (0, -13), None), ((-1, 0), (0, 12), None),
 ((-1, 0), (0, 14), None), ((-1, 0), (0, 16), None),
 ((0, -13), (1, -13), None), ((0, -13), (1, 12), None),
 ((0, -13), (1, 14), None), ((0, -13), (1, 16), None),
 ((0, 12), (1, 12), None), ((0, 14), (1, 12), None),
 ((0, 14), (1, 14), None), ((0, 16), (1, 12), None),
 ((0, 16), (1, 16), None), ((1, -13), (2, 0), None),
 ((1, 12), (2, 0), None), ((1, 14), (2, 0), None),
 ((1, 16), (2, 0), None)]
>>> e
{((-1, 0), (0, -13)): 0,
```

(continues on next page)

(continued from previous page)

```
((-1, 0), (0, 12)): 0,
((-1, 0), (0, 14)): 0,
((-1, 0), (0, 16)): 0,
((0, -13), (1, -13)): 1,
((0, -13), (1, 12)): 0,
((0, -13), (1, 14)): 0,
((0, -13), (1, 16)): 0,
((0, 12), (1, 12)): 1,
((0, 14), (1, 12)): 0,
((0, 14), (1, 14)): 1,
((0, 16), (1, 12)): 0,
((0, 16), (1, 16)): 1,
((1, -13), (2, 0)): 0,
((1, 12), (2, 0)): 0,
((1, 14), (2, 0)): 0,
((1, 16), (2, 0)): 0}
>>> qs = [[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(8),Integer(9)],[[Integer(1),Integer(3)],[Integer(3),
↪Integer(4)],[Integer(5),Integer(7)],[Integer(1),Integer(9)],[Integer(2),
↪Integer(3)]]]
>>> Poset(qs).frank_network()
(Digraph on 20 vertices,
 {((-1, 0), (0, 1)): 0,
  ((-1, 0), (0, 2)): 0,
  ((-1, 0), (0, 3)): 0,
  ((-1, 0), (0, 4)): 0,
  ((-1, 0), (0, 5)): 0,
  ((-1, 0), (0, 6)): 0,
  ((-1, 0), (0, 7)): 0,
  ((-1, 0), (0, 8)): 0,
  ((-1, 0), (0, 9)): 0,
  ((0, 1), (1, 1)): 1,
  ((0, 2), (1, 2)): 1,
  ((0, 3), (1, 1)): 0,
  ((0, 3), (1, 2)): 0,
  ((0, 3), (1, 3)): 1,
  ((0, 4), (1, 1)): 0,
  ((0, 4), (1, 2)): 0,
  ((0, 4), (1, 3)): 0,
  ((0, 4), (1, 4)): 1,
  ((0, 5), (1, 5)): 1,
  ((0, 6), (1, 6)): 1,
  ((0, 7), (1, 5)): 0,
  ((0, 7), (1, 7)): 1,
  ((0, 8), (1, 8)): 1,
  ((0, 9), (1, 1)): 0,
  ((0, 9), (1, 9)): 1,
  ((1, 1), (2, 0)): 0,
  ((1, 2), (2, 0)): 0,
  ((1, 3), (2, 0)): 0,
  ((1, 4), (2, 0)): 0,
  ((1, 5), (2, 0)): 0,
```

(continues on next page)

(continued from previous page)

```
((1, 6), (2, 0)): 0,
((1, 7), (2, 0)): 0,
((1, 8), (2, 0)): 0,
((1, 9), (2, 0)): 0})
```

AUTHOR:

- Darij Grinberg (2013-05-09)

ge (x, y)

Return `True` if x is greater than or equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_gequal(3, 1)
True
sage: P.is_gequal(2, 2)
True
sage: P.is_gequal(0, 1)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.is_gequal(Integer(3), Integer(1))
True
>>> P.is_gequal(Integer(2), Integer(2))
True
>>> P.is_gequal(Integer(0), Integer(1))
False
```

See also

is_greater_than(), *is_lequal()*.

graphviz_string (*graph_string='graph'*, *edge_string='--'*)

Return a representation in the DOT language, ready to render in graphviz.

See http://www.graphviz.org/doc/info/lang.html for more information about graphviz.

EXAMPLES:

```
sage: P = Poset({'a':['b'],'b':['d'],'c':['d'],'d':['f'],'e':['f'],'f':[]})
sage: print(P.graphviz_string())
graph {
"f";"d";"b";"a";"c";"e";
"f"--"e";"d"--"c";"b"--"a";"d"--"b";"f"--"d";
}
```

```
>>> from sage.all import *
>>> P = Poset({'a':['b'],'b':['d'],'c':['d'],'d':['f'],'e':['f'],'f':[]})
```

(continues on next page)

(continued from previous page)

```
>>> print(P.graphviz_string())
graph {
"f";"d";"b";"a";"c";"e";
"f"--"e";"d"--"c";"b"--"a";"d"--"b";"f"--"d";
}
```

greene_shape()

Return the Greene-Kleitman partition of `self`.

The Greene-Kleitman partition of a finite poset P is the partition $(c_1 - c_0, c_2 - c_1, c_3 - c_2, \ldots)$, where c_k is the maximum cardinality of a union of k chains of P. Equivalently, this is the conjugate of the partition $(a_1 - a_0, a_2 - a_1, a_3 - a_2, \ldots)$, where a_k is the maximum cardinality of a union of k antichains of P.

See many sources, e. g., [BF1999], for proofs of this equivalence.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = Poset([[3,2,1], [[3,1],[2,1]]])
sage: P.greene_shape()
[2, 1]
sage: P = Poset([[1,2,3,4], [[1,4],[2,4],[4,3]]])
sage: P.greene_shape()
[3, 1]
sage: P = Poset([[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22],
....:            [[1,4],[2,4],[4,3]]])
sage: P.greene_shape()
[3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: P = Poset([[],[]])
sage: P.greene_shape()
[]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = Poset([[Integer(3),Integer(2),Integer(1)], [[Integer(3),Integer(1)],
↪[Integer(2),Integer(1)]]])
>>> P.greene_shape()
[2, 1]
>>> P = Poset([[Integer(1),Integer(2),Integer(3),Integer(4)], [[Integer(1),
↪Integer(4)],[Integer(2),Integer(4)],[Integer(4),Integer(3)]]])
>>> P.greene_shape()
[3, 1]
>>> P = Poset([[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7),Integer(8),Integer(9),Integer(10),Integer(11),
↪Integer(12),Integer(13),Integer(14),Integer(15),Integer(16),Integer(17),
↪Integer(18),Integer(19),Integer(20),Integer(21),Integer(22)],
...            [[Integer(1),Integer(4)],[Integer(2),Integer(4)],[Integer(4),
↪Integer(3)]]])
>>> P.greene_shape()
[3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> P = Poset([[],[]])
>>> P.greene_shape()
[]
```

AUTHOR:

- Darij Grinberg (2013-05-09)

gt (x, y)

Return `True` if x is greater than but not equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_greater_than(3, 1)
True
sage: P.is_greater_than(1, 2)
False
sage: P.is_greater_than(3, 3)
False
sage: P.is_greater_than(0, 1)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.is_greater_than(Integer(3), Integer(1))
True
>>> P.is_greater_than(Integer(1), Integer(2))
False
>>> P.is_greater_than(Integer(3), Integer(3))
False
>>> P.is_greater_than(Integer(0), Integer(1))
False
```

For non-facade posets also > works:

```
sage: P = Poset({3: [1, 2]}, facade=False)
sage: P(2) > P(3)
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(3): [Integer(1), Integer(2)]}, facade=False)
>>> P(Integer(2)) > P(Integer(3))
True
```

See also

is_gequal(), *is_less_than()*.

h_polynomial()

Return the h-polynomial of a bounded poset `self`.

This is the h-polynomial of the order complex of the poset minus its bounds.

This is related to the f-polynomial by a simple change of variables:

$$h(q) = (1-q)^{\deg f} f\left(\frac{q}{1-q}\right),$$

where f and h denote the f-polynomial and the h-polynomial, respectively.

See Wikipedia article h-vector.

Warning

This is slightly different from the `hPolynomial` method in Macaulay2.

EXAMPLES:

```
sage: P = posets.AntichainPoset(3).order_ideals_lattice()
sage: P.h_polynomial()
q^3 + 4*q^2 + q
sage: P = posets.DiamondPoset(5)
sage: P.h_polynomial()
2*q^2 + q
sage: P = Poset({1: []})
sage: P.h_polynomial()
1
```

```
>>> from sage.all import *
>>> P = posets.AntichainPoset(Integer(3)).order_ideals_lattice()
>>> P.h_polynomial()
q^3 + 4*q^2 + q
>>> P = posets.DiamondPoset(Integer(5))
>>> P.h_polynomial()
2*q^2 + q
>>> P = Poset({Integer(1): []})
>>> P.h_polynomial()
1
```

See also

is_bounded(), *f_polynomial()*, *order_complex()*, `sage.topology.simplicial_complex.SimplicialComplex.h_vector()`

has_bottom()

Return `True` if the poset has a unique minimal element, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3], 1:[3], 2:[3], 3:[4], 4:[]})
sage: P.has_bottom()
False
sage: Q = Poset({0:[1], 1:[]})
sage: Q.has_bottom()
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)], Integer(1):[Integer(3)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
```

(continues on next page)

(continued from previous page)

```
>>> P.has_bottom()
False
>>> Q = Poset({Integer(0):[Integer(1)], Integer(1):[]})
>>> Q.has_bottom()
True
```

See also

- Dual Property: *has_top()*
- Stronger properties: *is_bounded()*
- Other: *bottom()*

has_isomorphic_subposet(*other*)

Return `True` if the poset contains a subposet isomorphic to `other`.

By subposet we mean that there exist a set `X` of elements such that `self.subposet(X)` is isomorphic to `other`.

INPUT:

- `other` – a finite poset

EXAMPLES:

```
sage: D = Poset({1:[2,3], 2:[4], 3:[4]})
sage: T = Poset({1:[2,3], 2:[4,5], 3:[6,7]})
sage: N5 = posets.PentagonPoset()

sage: N5.has_isomorphic_subposet(T)
False
sage: N5.has_isomorphic_subposet(D)
True

sage: len([P for P in Posets(5) if P.has_isomorphic_subposet(D)])
11
```

```
>>> from sage.all import *
>>> D = Poset({Integer(1):[Integer(2),Integer(3)], Integer(2):[Integer(4)],␣
↪Integer(3):[Integer(4)]})
>>> T = Poset({Integer(1):[Integer(2),Integer(3)], Integer(2):[Integer(4),
↪Integer(5)], Integer(3):[Integer(6),Integer(7)]})
>>> N5 = posets.PentagonPoset()

>>> N5.has_isomorphic_subposet(T)
False
>>> N5.has_isomorphic_subposet(D)
True

>>> len([P for P in Posets(Integer(5)) if P.has_isomorphic_subposet(D)])
11
```

has_top()

Return True if the poset has a unique maximal element, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3], 1:[3], 2:[3], 3:[4, 5], 4:[], 5:[]})
sage: P.has_top()
False
sage: Q = Poset({0:[3], 1:[3], 2:[3], 3:[4], 4:[]})
sage: Q.has_top()
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)], Integer(1):[Integer(3)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4), Integer(5)], Integer(4):[],
↪ Integer(5):[]})
>>> P.has_top()
False
>>> Q = Poset({Integer(0):[Integer(3)], Integer(1):[Integer(3)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> Q.has_top()
True
```

See also

- Dual Property: *has_bottom()*
- Stronger properties: *is_bounded()*
- Other: *top()*

hasse_diagram()

Return the Hasse diagram of the poset as a Sage DiGraph.

The Hasse diagram is a directed graph where vertices are the elements of the poset and there is an edge from u to v whenever v covers u in the poset.

If dot2tex is installed, then this sets the Hasse diagram's latex options to use the dot2tex formatting.

EXAMPLES:

```
sage: P = posets.DivisorLattice(12)
sage: H = P.hasse_diagram(); H
Digraph on 6 vertices
sage: P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
sage: H.edges(sort=True, labels=False)
[(1, 2), (1, 3), (2, 4), (2, 6), (3, 6), (4, 12), (6, 12)]
```

```
>>> from sage.all import *
>>> P = posets.DivisorLattice(Integer(12))
>>> H = P.hasse_diagram(); H
Digraph on 6 vertices
>>> P.cover_relations()
```

(continues on next page)

(continued from previous page)

```
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
>>> H.edges(sort=True, labels=False)
[(1, 2), (1, 3), (2, 4), (2, 6), (3, 6), (4, 12), (6, 12)]
```

height (*certificate=False*)

Return the height (number of elements in a longest chain) of the poset.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return `(h, c)`, where `h` is the height and `c` is a chain of maximum cardinality. If `certificate=False` return only the height.

EXAMPLES:

```
sage: P = Poset({0: [1], 2: [3, 4], 4: [5, 6]})
sage: P.height()
3
sage: posets.PentagonPoset().height(certificate=True)
(4, [0, 2, 3, 4])
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0): [Integer(1)], Integer(2): [Integer(3), Integer(4)],
↪ Integer(4): [Integer(5), Integer(6)]})
>>> P.height()
3
>>> posets.PentagonPoset().height(certificate=True)
(4, [0, 2, 3, 4])
```

incidence_algebra (*R*, *prefix='I'*)

Return the incidence algebra of `self` over `R`.

OUTPUT: an instance of *sage.combinat.posets.incidence_algebras.IncidenceAlgebra*

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: P.incidence_algebra(QQ)
Incidence algebra of Finite lattice containing 16 elements
 over Rational Field
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> P.incidence_algebra(QQ)
Incidence algebra of Finite lattice containing 16 elements
 over Rational Field
```

incomparability_graph ()

Return the incomparability graph of the poset.

This is the complement of the comparability graph, i.e. an undirected graph where vertices are the elements of the poset and there is an edge between vertices if they are not comparable in the poset.

EXAMPLES:

```
sage: Y = Poset({1: [2], 2: [3, 4]})
sage: g = Y.incomparability_graph(); g
Incomparability graph on 4 vertices
sage: Y.compare_elements(1, 3) is not None
True
sage: g.has_edge(1, 3)
False
```

```
>>> from sage.all import *
>>> Y = Poset({Integer(1): [Integer(2)], Integer(2): [Integer(3), Integer(4)]}
↪)
>>> g = Y.incomparability_graph(); g
Incomparability graph on 4 vertices
>>> Y.compare_elements(Integer(1), Integer(3)) is not None
True
>>> g.has_edge(Integer(1), Integer(3))
False
```

See also

comparability_graph()

interval(*x*, *y*)

Return a list of the elements z such that $x \leq z \leq y$.

INPUT:

- x – any element of the poset
- y – any element of the poset

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: P = Poset(dag)
sage: I = set(map(P,[2,5,6,4,7]))
sage: I == set(P.interval(2,7))
True
```

```
>>> from sage.all import *
>>> uc = [[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]]
>>> dag = DiGraph(dict(zip(range(len(uc)),uc)))
>>> P = Poset(dag)
>>> I = set(map(P,[Integer(2),Integer(5),Integer(6),Integer(4),Integer(7)]))
>>> I == set(P.interval(Integer(2),Integer(7)))
True
```

```
sage: dg = DiGraph({"a":["b","c"], "b":["d"], "c":["d"]})
sage: P = Poset(dg, facade = False)
```

(continues on next page)

(continued from previous page)

```
sage: P.interval("a","d")
[a, b, c, d]
```

```
>>> from sage.all import *
>>> dg = DiGraph({"a":["b","c"], "b":["d"], "c":["d"]})
>>> P = Poset(dg, facade = False)
>>> P.interval("a","d")
[a, b, c, d]
```

intervals_number()

Return the number of relations in the poset.

A relation is a pair of elements x and y such that $x \leq y$ in the poset.

Relations are also often called intervals. The number of intervals is the dimension of the incidence algebra.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.relations_number()
13

sage: posets.TamariLattice(4).relations_number()
68
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.relations_number()
13

>>> posets.TamariLattice(Integer(4)).relations_number()
68
```

See also

relations_iterator(), *relations()*

intervals_poset()

Return the natural partial order on the set of intervals of the poset.

OUTPUT: a finite poset

The poset of intervals of a poset P has the set of intervals $[x, y]$ in P as elements, endowed with the order relation defined by $[x_1, y_1] \leq [x_2, y_2]$ if and only if $x_1 \leq x_2$ and $y_1 \leq y_2$.

This is also called P to the power 2, meaning the poset of poset-morphisms from the 2-chain to P.

If P is a lattice, the result is also a lattice.

EXAMPLES:

```
sage: P = Poset({0:[1]})
sage: P.intervals_poset()
Finite poset containing 3 elements
```

(continues on next page)

(continued from previous page)

```
sage: P = posets.PentagonPoset()
sage: P.intervals_poset()
Finite lattice containing 13 elements
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(1)]})
>>> P.intervals_poset()
Finite poset containing 3 elements

>>> P = posets.PentagonPoset()
>>> P.intervals_poset()
Finite lattice containing 13 elements
```

is_EL_labelling(*f*, *return_raising_chains=False*)

Return `True` if `f` is an EL labelling of `self`.

A labelling f of the edges of the Hasse diagram of a poset is called an EL labelling (edge lexicographic labelling) if for any two elements u and v with $u \leq v$,

- there is a unique f-raising chain from u to v in the Hasse diagram, and this chain is lexicographically first among all chains from u to v.

For more details, see [Bj1980].

INPUT:

- `f` – a function taking two elements `a` and `b` in `self` such that `b` covers `a` and returning elements in a totally ordered set.
- `return_raising_chains` (optional; default: `False`) if `True`, returns the set of all raising chains in `self`, if possible.

EXAMPLES:

Let us consider a Boolean poset:

```
sage: P = Poset([[(0,0),(0,1),(1,0),(1,1)],[[(0,0),(0,1)],[(0,0),(1,0)],[(0,
↪1),(1,1)],[(1,0),(1,1)]]],facade=True)
sage: label = lambda a,b: min( i for i in [0,1] if a[i] != b[i] )
sage: P.is_EL_labelling(label)
True
sage: P.is_EL_labelling(label,return_raising_chains=True)
{((0, 0), (0, 1)): [1],
 ((0, 0), (1, 0)): [0],
 ((0, 0), (1, 1)): [0, 1],
 ((0, 1), (1, 1)): [0],
 ((1, 0), (1, 1)): [1]}
```

```
>>> from sage.all import *
>>> P = Poset([[(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(1),
↪Integer(0)),(Integer(1),Integer(1))],[[(Integer(0),Integer(0)),(Integer(0),
↪Integer(1))],[(Integer(0),Integer(0)),(Integer(1),Integer(0))],[(Integer(0),
↪Integer(1)),(Integer(1),Integer(1))],[(Integer(1),Integer(0)),(Integer(1),
↪Integer(1))]]],facade=True)
```

(continues on next page)

(continued from previous page)

```
>>> label = lambda a,b: min( i for i in [Integer(0),Integer(1)] if a[i] !=␣
↪b[i] )
>>> P.is_EL_labelling(label)
True
>>> P.is_EL_labelling(label,return_raising_chains=True)
{((0, 0), (0, 1)): [1],
 ((0, 0), (1, 0)): [0],
 ((0, 0), (1, 1)): [0, 1],
 ((0, 1), (1, 1)): [0],
 ((1, 0), (1, 1)): [1]}
```

is_antichain_of_poset(*elms*)

Return `True` if `elms` is an antichain of the poset and `False` otherwise.

Set of elements are an *antichain* of a poset if they are pairwise incomparable.

EXAMPLES:

```
sage: P = posets.BooleanLattice(5)
sage: P.is_antichain_of_poset([3, 5, 7])
False
sage: P.is_antichain_of_poset([3, 5, 14])
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(5))
>>> P.is_antichain_of_poset([Integer(3), Integer(5), Integer(7)])
False
>>> P.is_antichain_of_poset([Integer(3), Integer(5), Integer(14)])
True
```

is_bounded()

Return `True` if the poset is bounded, and `False` otherwise.

A poset is bounded if it contains both a unique maximal element and a unique minimal element.

EXAMPLES:

```
sage: P = Poset({0:[3], 1:[3], 2:[3], 3:[4, 5], 4:[], 5:[]})
sage: P.is_bounded()
False
sage: Q = posets.DiamondPoset(5)
sage: Q.is_bounded()
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)], Integer(1):[Integer(3)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4), Integer(5)], Integer(4):[],
↪ Integer(5):[]})
>>> P.is_bounded()
False
>>> Q = posets.DiamondPoset(Integer(5))
>>> Q.is_bounded()
True
```

See also

- Weaker properties: *has_bottom()*, *has_top()*
- Other: *with_bounds()*, *without_bounds()*

is_chain()

Return `True` if the poset is totally ordered ("chain"), and `False` otherwise.

EXAMPLES:

```
sage: I = Poset({0:[1], 1:[2], 2:[3], 3:[4]})
sage: I.is_chain()
True

sage: II = Poset({0:[1], 2:[3]})
sage: II.is_chain()
False

sage: V = Poset({0:[1, 2]})
sage: V.is_chain()
False
```

```
>>> from sage.all import *
>>> I = Poset({Integer(0):[Integer(1)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)]})
>>> I.is_chain()
True

>>> II = Poset({Integer(0):[Integer(1)], Integer(2):[Integer(3)]})
>>> II.is_chain()
False

>>> V = Poset({Integer(0):[Integer(1), Integer(2)]})
>>> V.is_chain()
False
```

is_chain_of_poset(*elms*, *ordered=False*)

Return `True` if `elms` is a chain of the poset, and `False` otherwise.

Set of elements are a *chain* of a poset if they are comparable to each other.

INPUT:

- `elms` – list or other iterable containing some elements of the poset
- `ordered` – boolean; if `True`, then return `True` only if elements in `elms` are strictly increasing in the poset. This makes no sense if `elms` is a set. If `False` (the default), then elements can be repeated and be in any order.

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")))
sage: sorted(P.list())
[1, 2, 3, 4, 6, 12]
```

(continues on next page)

(continued from previous page)

```
sage: P.is_chain_of_poset([12, 3])
True
sage: P.is_chain_of_poset({3, 4, 12})
False
sage: P.is_chain_of_poset([12, 3], ordered=True)
False
sage: P.is_chain_of_poset((1, 1, 3))
True
sage: P.is_chain_of_poset((1, 1, 3), ordered=True)
False
sage: P.is_chain_of_poset((1, 3), ordered=True)
True
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")))
>>> sorted(P.list())
[1, 2, 3, 4, 6, 12]
>>> P.is_chain_of_poset([Integer(12), Integer(3)])
True
>>> P.is_chain_of_poset({Integer(3), Integer(4), Integer(12)})
False
>>> P.is_chain_of_poset([Integer(12), Integer(3)], ordered=True)
False
>>> P.is_chain_of_poset((Integer(1), Integer(1), Integer(3)))
True
>>> P.is_chain_of_poset((Integer(1), Integer(1), Integer(3)), ordered=True)
False
>>> P.is_chain_of_poset((Integer(1), Integer(3)), ordered=True)
True
```

is_connected()

Return `True` if the poset is connected, and `False` otherwise.

A poset is connected if its Hasse diagram is connected.

If a poset is not connected, then it can be divided to parts S_1 and S_2 so that every element of S_1 is incomparable to every element of S_2.

EXAMPLES:

```
sage: P = Poset({1:[2, 3], 3:[4, 5]})
sage: P.is_connected()
True

sage: P = Poset({1:[2, 3], 3:[4, 5], 6:[7, 8]})
sage: P.is_connected()
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1):[Integer(2), Integer(3)], Integer(3):[Integer(4),␣
↪Integer(5)]})
>>> P.is_connected()
True
```

(continues on next page)

(continued from previous page)

```
>>> P = Poset({Integer(1):[Integer(2), Integer(3)], Integer(3):[Integer(4),␣
↪Integer(5)], Integer(6):[Integer(7), Integer(8)]})
>>> P.is_connected()
False
```

See also

connected_components()

is_d_complete()

Return `True` if a poset is d-complete and `False` otherwise.

See also

- *d_complete*

EXAMPLES:

```
sage: from sage.combinat.posets.posets import FinitePoset
sage: A = Poset({0: [1,2]})
sage: A.is_d_complete()
False

sage: from sage.combinat.posets.poset_examples import Posets
sage: B = Posets.DoubleTailedDiamond(3)
sage: B.is_d_complete()
True

sage: C = Poset({0: [2], 1: [2], 2: [3, 4], 3: [5], 4: [5], 5: [6]})
sage: C.is_d_complete()
False

sage: D = Poset({0: [1, 2], 1: [4], 2: [4], 3: [4]})
sage: D.is_d_complete()
False

sage: P = Posets.YoungDiagramPoset(Partition([3, 2, 2]), dual=True)          #␣
↪needs sage.combinat
sage: P.is_d_complete()                                                      #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.posets import FinitePoset
>>> A = Poset({Integer(0): [Integer(1),Integer(2)]})
>>> A.is_d_complete()
False
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.posets.poset_examples import Posets
>>> B = Posets.DoubleTailedDiamond(Integer(3))
>>> B.is_d_complete()
True

>>> C = Poset({Integer(0): [Integer(2)], Integer(1): [Integer(2)],␣
↪Integer(2): [Integer(3), Integer(4)], Integer(3): [Integer(5)], Integer(4):␣
↪[Integer(5)], Integer(5): [Integer(6)]})
>>> C.is_d_complete()
False

>>> D = Poset({Integer(0): [Integer(1), Integer(2)], Integer(1): [Integer(4)],
↪ Integer(2): [Integer(4)], Integer(3): [Integer(4)]})
>>> D.is_d_complete()
False

>>> P = Posets.YoungDiagramPoset(Partition([Integer(3), Integer(2),␣
↪Integer(2)]), dual=True)          # needs sage.combinat
>>> P.is_d_complete()                                                    #␣
↪needs sage.combinat
True
```

is_eulerian(*k=None*, *certificate=False*)

Return `True` if the poset is Eulerian, and `False` otherwise.

The poset is expected to be graded and bounded.

A poset is Eulerian if every non-trivial interval has the same number of elements of even rank as of odd rank. A poset is k-eulerian if every non-trivial interval up to rank k is Eulerian.

See Wikipedia article Eulerian_poset.

INPUT:

- `k` – integer; only check if the poset is k-eulerian. If `None` (the default), check if the poset is Eulerian.
- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `True, None` or `False, (a, b)`, where the interval `(a, b)` is not Eulerian. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: P = Poset({0: [1, 2, 3], 1: [4, 5], 2: [4, 6], 3: [5, 6],
....:            4: [7, 8], 5: [7, 8], 6: [7, 8], 7: [9], 8: [9]})
sage: P.is_eulerian()                                                    #␣
↪needs sage.libs.flint
True
sage: P = Poset({0: [1, 2, 3], 1: [4, 5, 6], 2: [4, 6], 3: [5,6],
....:            4: [7], 5:[7], 6:[7]})
sage: P.is_eulerian()                                                    #␣
↪needs sage.libs.flint
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0): [Integer(1), Integer(2), Integer(3)], Integer(1):␣
↪[Integer(4), Integer(5)], Integer(2): [Integer(4), Integer(6)], Integer(3):␣
↪[Integer(5), Integer(6)],
...             Integer(4): [Integer(7), Integer(8)], Integer(5): [Integer(7),␣
↪Integer(8)], Integer(6): [Integer(7), Integer(8)], Integer(7): [Integer(9)],
↪ Integer(8): [Integer(9)]})
>>> P.is_eulerian()                                                         #␣
↪needs sage.libs.flint
True
>>> P = Poset({Integer(0): [Integer(1), Integer(2), Integer(3)], Integer(1):␣
↪[Integer(4), Integer(5), Integer(6)], Integer(2): [Integer(4), Integer(6)],␣
↪Integer(3): [Integer(5),Integer(6)],
...             Integer(4): [Integer(7)], Integer(5):[Integer(7)],␣
↪Integer(6):[Integer(7)]})
>>> P.is_eulerian()                                                         #␣
↪needs sage.libs.flint
False
```

Canonical examples of Eulerian posets are the face lattices of convex polytopes:

```
sage: P = polytopes.cube().face_lattice()                                   #␣
↪needs sage.geometry.polyhedron
sage: P.is_eulerian()                                                       #␣
↪needs sage.geometry.polyhedron sage.libs.flint
True
```

```
>>> from sage.all import *
>>> P = polytopes.cube().face_lattice()                                     #␣
↪needs sage.geometry.polyhedron
>>> P.is_eulerian()                                                         #␣
↪needs sage.geometry.polyhedron sage.libs.flint
True
```

A poset that is 3- but not 4-eulerian:

```
sage: P = Poset(DiGraph('MWW@_?W?@_?W??@??O@_?W?@_?W?@??O??')); P
Finite poset containing 14 elements
sage: P.is_eulerian(k=3)                                                    #␣
↪needs sage.libs.flint
True
sage: P.is_eulerian(k=4)                                                    #␣
↪needs sage.libs.flint
False
```

```
>>> from sage.all import *
>>> P = Poset(DiGraph('MWW@_?W?@_?W??@??O@_?W?@_?W?@??O??')); P
Finite poset containing 14 elements
>>> P.is_eulerian(k=Integer(3))                                               ␣
↪      # needs sage.libs.flint
True
>>> P.is_eulerian(k=Integer(4))                                               ␣
```

(continues on next page)

(continued from previous page)

```
↪      # needs sage.libs.flint
False
```

Getting an interval that is not Eulerian:

```
sage: P = posets.DivisorLattice(12)
sage: P.is_eulerian(certificate=True)                                    #␣
↪needs sage.libs.flint
(False, (1, 4))
```

```
>>> from sage.all import *
>>> P = posets.DivisorLattice(Integer(12))
>>> P.is_eulerian(certificate=True)                                     #␣
↪needs sage.libs.flint
(False, (1, 4))
```

is_gequal(*x*, *y*)

Return `True` if x is greater than or equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_gequal(3, 1)
True
sage: P.is_gequal(2, 2)
True
sage: P.is_gequal(0, 1)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.is_gequal(Integer(3), Integer(1))
True
>>> P.is_gequal(Integer(2), Integer(2))
True
>>> P.is_gequal(Integer(0), Integer(1))
False
```

See also

is_greater_than(), *is_lequal()*.

is_graded()

Return `True` if the poset is graded, and `False` otherwise.

A poset is graded if all its maximal chains have the same length.

There are various competing definitions for graded posets (see Wikipedia article Graded_poset). This definition is from section 3.1 of Richard Stanley's *Enumerative Combinatorics, Vol. 1* [EnumComb1]. Some sources call these posets *tiered*.

Every graded poset is ranked. The converse is true for bounded posets, including lattices.

EXAMPLES:

```
sage: P = posets.PentagonPoset()  # Not even ranked
sage: P.is_graded()
False

sage: P = Poset({1:[2, 3], 3:[4]})  # Ranked, but not graded
sage: P.is_graded()
False

sage: P = Poset({1:[3, 4], 2:[3, 4], 5:[6]})
sage: P.is_graded()
True

sage: P = Poset([[1], [2], [], [4], []])
sage: P.is_graded()
False
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()  # Not even ranked
>>> P.is_graded()
False

>>> P = Poset({Integer(1):[Integer(2), Integer(3)], Integer(3):[Integer(4)]})␣
↪ # Ranked, but not graded
>>> P.is_graded()
False

>>> P = Poset({Integer(1):[Integer(3), Integer(4)], Integer(2):[Integer(3),␣
↪Integer(4)], Integer(5):[Integer(6)]})
>>> P.is_graded()
True

>>> P = Poset([[Integer(1)], [Integer(2)], [], [Integer(4)], []])
>>> P.is_graded()
False
```

See also:

is_ranked(), *level_sets()*

is_greater_than(x, y)

Return True if x is greater than but not equal to y in the poset, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_greater_than(3, 1)
True
sage: P.is_greater_than(1, 2)
False
sage: P.is_greater_than(3, 3)
```

(continues on next page)

(continued from previous page)

```
False
sage: P.is_greater_than(0, 1)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.is_greater_than(Integer(3), Integer(1))
True
>>> P.is_greater_than(Integer(1), Integer(2))
False
>>> P.is_greater_than(Integer(3), Integer(3))
False
>>> P.is_greater_than(Integer(0), Integer(1))
False
```

For non-facade posets also > works:

```
sage: P = Poset({3: [1, 2]}, facade=False)
sage: P(2) > P(3)
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(3): [Integer(1), Integer(2)]}, facade=False)
>>> P(Integer(2)) > P(Integer(3))
True
```

See also:

is_gequal(), *is_less_than()*.

is_greedy(*certificate=False*)

Return `True` if the poset is greedy, and `False` otherwise.

A poset is *greedy* if every greedy linear extension has the same number of jumps.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (A, B))` where A and B are greedy linear extension so that B has more jumps. If `certificate=False` return `True` or `False`.

EXAMPLES:

This is not a self-dual property:

```
sage: W = Poset({1: [3, 4], 2: [4, 5]})
sage: M = W.dual()
sage: W.is_greedy()
True
```

(continues on next page)

(continued from previous page)

```
sage: M.is_greedy()
False
```

```
>>> from sage.all import *
>>> W = Poset({Integer(1): [Integer(3), Integer(4)], Integer(2): [Integer(4),␣
↪Integer(5)]})
>>> M = W.dual()
>>> W.is_greedy()
True
>>> M.is_greedy()
False
```

Getting a certificate:

```
sage: N = Poset({1: [3], 2: [3, 4]})
sage: N.is_greedy(certificate=True)
(False, ([1, 2, 4, 3], [2, 4, 1, 3]))
```

```
>>> from sage.all import *
>>> N = Poset({Integer(1): [Integer(3)], Integer(2): [Integer(3), Integer(4)]}
↪)
>>> N.is_greedy(certificate=True)
(False, ([1, 2, 4, 3], [2, 4, 1, 3]))
```

is_incomparable_chain_free(*m*, *n=None*)

Return `True` if the poset is $(m + n)$-free, and `False` otherwise.

A poset is $(m + n)$-free if there is no incomparable chains of lengths m and n. Three cases have special name (see [EnumComb1], exercise 3.15):

- "interval order" is $(2 + 2)$-free
- "semiorder" (or "unit interval order") is $(1 + 3)$-free and $(2 + 2)$-free
- "weak order" is $(1 + 2)$-free.

INPUT:

- `m`, `n` – positive integers

It is also possible to give a list of integer pairs as argument. See below for an example.

EXAMPLES:

```
sage: B3 = posets.BooleanLattice(3)
sage: B3.is_incomparable_chain_free(1, 3)
True
sage: B3.is_incomparable_chain_free(2, 2)
False

sage: IP6 = posets.IntegerPartitions(6)                                    #␣
↪needs sage.combinat
sage: IP6.is_incomparable_chain_free(1, 3)                                 #␣
↪needs sage.combinat
False
sage: IP6.is_incomparable_chain_free(2, 2)                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> B3 = posets.BooleanLattice(Integer(3))
>>> B3.is_incomparable_chain_free(Integer(1), Integer(3))
True
>>> B3.is_incomparable_chain_free(Integer(2), Integer(2))
False

>>> IP6 = posets.IntegerPartitions(Integer(6))                              ␣
↪     # needs sage.combinat
>>> IP6.is_incomparable_chain_free(Integer(1), Integer(3))                  ␣
↪              # needs sage.combinat
False
>>> IP6.is_incomparable_chain_free(Integer(2), Integer(2))                  ␣
↪              # needs sage.combinat
True
```

A list of pairs as an argument:

```
sage: B3.is_incomparable_chain_free([[1, 3], [2, 2]])
False
```

```
>>> from sage.all import *
>>> B3.is_incomparable_chain_free([[Integer(1), Integer(3)], [Integer(2),␣
↪Integer(2)]])
False
```

We show how to get an incomparable chain pair:

```
sage: P = posets.PentagonPoset()
sage: chains_1_2 = Poset({0:[], 1:[2]})
sage: incomps = P.isomorphic_subposets(chains_1_2)[0]
sage: sorted(incomps.list()), incomps.cover_relations()
([1, 2, 3], [[2, 3]])
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> chains_1_2 = Poset({Integer(0):[], Integer(1):[Integer(2)]})
>>> incomps = P.isomorphic_subposets(chains_1_2)[Integer(0)]
>>> sorted(incomps.list()), incomps.cover_relations()
([1, 2, 3], [[2, 3]])
```

AUTHOR:

- Eric Rowland (2013-05-28)

is_induced_subposet(*other*)

Return `True` if the poset is an induced subposet of `other`, and `False` otherwise.

A poset P is an induced subposet of Q if every element of P is an element of Q, and $x \leq_P y$ iff $x \leq_Q y$. Note that “induced” here has somewhat different meaning compared to that of graphs.

INPUT:

- `other` – poset

> **Note**
>
> This method does not check whether the poset is a *isomorphic* (i.e., up to relabeling) subposet of `other`, but only if `other` directly contains the poset as an induced subposet. For isomorphic subposets see `has_isomorphic_subposet()`.

EXAMPLES:

```
sage: P = Poset({1:[2, 3]})
sage: Q = Poset({1:[2, 4], 2:[3]})
sage: P.is_induced_subposet(Q)
False
sage: R = Poset({0:[1], 1:[3, 4], 3:[5], 4:[2]})
sage: P.is_induced_subposet(R)
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1):[Integer(2), Integer(3)]})
>>> Q = Poset({Integer(1):[Integer(2), Integer(4)], Integer(2):[Integer(3)]})
>>> P.is_induced_subposet(Q)
False
>>> R = Poset({Integer(0):[Integer(1)], Integer(1):[Integer(3), Integer(4)],
↪Integer(3):[Integer(5)], Integer(4):[Integer(2)]})
>>> P.is_induced_subposet(R)
True
```

is_isomorphic(*other*, ***kwds*)

Return `True` if both posets are isomorphic.

EXAMPLES:

```
sage: P = Poset(([1,2,3],[[1,3],[2,3]]))
sage: Q = Poset(([4,5,6],[[4,6],[5,6]]))
sage: P.is_isomorphic(Q)
True
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3)],[[Integer(1),Integer(3)],
↪[Integer(2),Integer(3)]]))
>>> Q = Poset(([Integer(4),Integer(5),Integer(6)],[[Integer(4),Integer(6)],
↪[Integer(5),Integer(6)]]))
>>> P.is_isomorphic(Q)
True
```

is_join_semilattice(*certificate=False*)

Return `True` if the poset has a join operation, and `False` otherwise.

A join is the least upper bound for given elements, if it exists.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (a, b))` where elements a and b have no least upper bound. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: P = Poset([[1,3,2], [4], [4,5,6], [6], [7], [7], [7], []])
sage: P.is_join_semilattice()
True

sage: P = Poset({1:[3, 4], 2:[3, 4], 3:[5], 4:[5]})
sage: P.is_join_semilattice()
False
sage: P.is_join_semilattice(certificate=True)
(False, (2, 1))
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(3),Integer(2)], [Integer(4)], [Integer(4),
↪Integer(5),Integer(6)], [Integer(6)], [Integer(7)], [Integer(7)],␣
↪[Integer(7)], []])
>>> P.is_join_semilattice()
True

>>> P = Poset({Integer(1):[Integer(3), Integer(4)], Integer(2):[Integer(3),␣
↪Integer(4)], Integer(3):[Integer(5)], Integer(4):[Integer(5)]})
>>> P.is_join_semilattice()
False
>>> P.is_join_semilattice(certificate=True)
(False, (2, 1))
```

See also

- Dual property: *`is_meet_semilattice()`*
- Stronger properties: `is_lattice()`

is_jump_critical(*certificate=False*)

Return `True` if the poset is jump-critical, and `False` otherwise.

A poset P is *jump-critical* if every proper subposet has smaller jump number.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, e)` so that removing element e from the poset does not decrease the jump number. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: P = Poset({1: [3, 6], 2: [3, 4, 5], 4: [6, 7], 5: [7]})
sage: P.is_jump_critical()
True

sage: P = posets.PentagonPoset()
sage: P.is_jump_critical()
False
sage: P.is_jump_critical(certificate=True)
(False, 3)
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1): [Integer(3), Integer(6)], Integer(2): [Integer(3),
↪Integer(4), Integer(5)], Integer(4): [Integer(6), Integer(7)], Integer(5):
↪[Integer(7)]})
>>> P.is_jump_critical()
True

>>> P = posets.PentagonPoset()
>>> P.is_jump_critical()
False
>>> P.is_jump_critical(certificate=True)
(False, 3)
```

See also

jump_number()

is_lequal(*x*, *y*)

Return True if x is less than or equal to y in the poset, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_lequal(2, 4)
True
sage: P.is_lequal(2, 2)
True
sage: P.is_lequal(0, 1)
False
sage: P.is_lequal(3, 2)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.is_lequal(Integer(2), Integer(4))
True
>>> P.is_lequal(Integer(2), Integer(2))
True
>>> P.is_lequal(Integer(0), Integer(1))
False
```

(continues on next page)

(continued from previous page)

```
>>> P.is_lequal(Integer(3), Integer(2))
False
```

See also

is_less_than(), *is_gequal()*.

is_less_than(*x*, *y*)

Return True if x is less than but not equal to y in the poset, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_less_than(1, 3)
True
sage: P.is_less_than(0, 1)
False
sage: P.is_less_than(2, 2)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
→Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.is_less_than(Integer(1), Integer(3))
True
>>> P.is_less_than(Integer(0), Integer(1))
False
>>> P.is_less_than(Integer(2), Integer(2))
False
```

For non-facade posets also < works:

```
sage: P = Poset({3: [1, 2]}, facade=False)
sage: P(1) < P(2)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(3): [Integer(1), Integer(2)]}, facade=False)
>>> P(Integer(1)) < P(Integer(2))
False
```

See also

is_lequal(), *is_greater_than()*.

is_linear_extension(*l*)

Return whether l is a linear extension of self.

INPUT:

- l – list (or iterable) containing all of the elements of self exactly once

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), facade=True, linear_
↪extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: P.is_linear_extension([1, 2, 4, 3, 6, 12])
True
sage: P.is_linear_extension([1, 2, 4, 6, 3, 12])
False

sage: [p for p in Permutations(list(P)) if P.is_linear_extension(p)]
[[1, 2, 3, 4, 6, 12],
 [1, 2, 3, 6, 4, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12]]
sage: list(P.linear_extensions())
[[1, 2, 3, 4, 6, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12],
 [1, 2, 3, 6, 4, 12]]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")), facade=True,␣
↪linear_extension=True)
>>> P.list()
[1, 2, 3, 4, 6, 12]
>>> P.is_linear_extension([Integer(1), Integer(2), Integer(4), Integer(3),␣
↪Integer(6), Integer(12)])
True
>>> P.is_linear_extension([Integer(1), Integer(2), Integer(4), Integer(6),␣
↪Integer(3), Integer(12)])
False

>>> [p for p in Permutations(list(P)) if P.is_linear_extension(p)]
[[1, 2, 3, 4, 6, 12],
 [1, 2, 3, 6, 4, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12]]
>>> list(P.linear_extensions())
[[1, 2, 3, 4, 6, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12],
 [1, 2, 3, 6, 4, 12]]
```

Note

This is used and systematically tested in `LinearExtensionsOfPosets`

> **See also**
>
> *linear_extension()*, *linear_extensions()*

is_linear_interval(*x*, *y*)

Return whether the interval [x, y] is linear.

This means that this interval is a total order.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.is_linear_interval(0, 4)
False
sage: P.is_linear_interval(0, 3)
True
sage: P.is_linear_interval(1, 3)
False
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.is_linear_interval(Integer(0), Integer(4))
False
>>> P.is_linear_interval(Integer(0), Integer(3))
True
>>> P.is_linear_interval(Integer(1), Integer(3))
False
```

is_meet_semilattice(*certificate=False*)

Return True if the poset has a meet operation, and False otherwise.

A meet is the greatest lower bound for given elements, if it exists.

INPUT:

- certificate – boolean (default: False); whether to return a certificate

OUTPUT:

- If certificate=True return either (True, None) or (False, (a, b)) where elements a and b have no greatest lower bound. If certificate=False return True or False.

EXAMPLES:

```
sage: P = Poset({1:[2, 3, 4], 2:[5, 6], 3:[6], 4:[6, 7]})
sage: P.is_meet_semilattice()
True

sage: Q = P.dual()
sage: Q.is_meet_semilattice()
False

sage: V = posets.IntegerPartitions(5)                                  #␣
↪needs sage.combinat
sage: V.is_meet_semilattice(certificate=True)                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
(False, ((2, 2, 1), (3, 1, 1)))
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1):[Integer(2), Integer(3), Integer(4)],␣
↪Integer(2):[Integer(5), Integer(6)], Integer(3):[Integer(6)],␣
↪Integer(4):[Integer(6), Integer(7)]})
>>> P.is_meet_semilattice()
True

>>> Q = P.dual()
>>> Q.is_meet_semilattice()
False

>>> V = posets.IntegerPartitions(Integer(5))                                  ␣
↪     # needs sage.combinat
>>> V.is_meet_semilattice(certificate=True)                                 #␣
↪needs sage.combinat
(False, ((2, 2, 1), (3, 1, 1)))
```

See also:

- Dual property: *is_join_semilattice()*
- Stronger properties: `is_lattice()`

is_parent_of(*x*)

Return `True` if `x` is an element of the poset.

is_rank_symmetric()

Return `True` if the poset is rank symmetric, and `False` otherwise.

The poset is expected to be graded and connected.

A poset of rank h (maximal chains have $h + 1$ elements) is rank symmetric if the number of elements are equal in ranks i and $h - i$ for every i in $0, 1, \ldots, h$.

EXAMPLES:

```
sage: P = Poset({1:[3, 4, 5], 2:[3, 4, 5], 3:[6], 4:[7], 5:[7]})
sage: P.is_rank_symmetric()
True
sage: P = Poset({1:[2], 2:[3, 4], 3:[5], 4:[5]})
sage: P.is_rank_symmetric()
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1):[Integer(3), Integer(4), Integer(5)],␣
↪Integer(2):[Integer(3), Integer(4), Integer(5)], Integer(3):[Integer(6)],␣
↪Integer(4):[Integer(7)], Integer(5):[Integer(7)]})
>>> P.is_rank_symmetric()
True
```

(continues on next page)

(continued from previous page)

```
>>> P = Poset({Integer(1):[Integer(2)], Integer(2):[Integer(3), Integer(4)],
↪Integer(3):[Integer(5)], Integer(4):[Integer(5)]})
>>> P.is_rank_symmetric()
False
```

is_ranked()

Return `True` if the poset is ranked, and `False` otherwise.

A poset is ranked if there is a function r from poset elements to integers so that $r(x) = r(y) + 1$ when x covers y.

Informally said a ranked poset can be "levelized": every element is on a "level", and every cover relation goes only one level up.

EXAMPLES:

```
sage: P = Poset( ([1, 2, 3, 4], [[1, 2], [2, 4], [3, 4]] ))
sage: P.is_ranked()
True

sage: P = Poset([[1, 5], [2, 6], [3], [4],[], [6, 3], [4]])
sage: P.is_ranked()
False
```

```
>>> from sage.all import *
>>> P = Poset( ([Integer(1), Integer(2), Integer(3), Integer(4)],
↪[[Integer(1), Integer(2)], [Integer(2), Integer(4)], [Integer(3),
↪Integer(4)]] ))
>>> P.is_ranked()
True

>>> P = Poset([[Integer(1), Integer(5)], [Integer(2), Integer(6)],
↪[Integer(3)], [Integer(4)],[], [Integer(6), Integer(3)], [Integer(4)]])
>>> P.is_ranked()
False
```

See also

rank_function(), *rank()*, *is_graded()*

is_series_parallel()

Return `True` if the poset is series-parallel, and `False` otherwise.

A poset is *series-parallel* if it can be built up from one-element posets using the operations of disjoint union and ordinal sum. This is also called *N-free* property: every poset that is not series-parallel contains a subposet isomorphic to the 4-element N-shaped poset where $a > c, d$ and $b > d$.

Note

Some papers use the term N-free for posets having no N-shaped poset as a *cover-preserving subposet*. This definition is not used here.

See Wikipedia article Series-parallel partial order.

EXAMPLES:

```
sage: VA = Poset({1: [2, 3], 4: [5], 6: [5]})
sage: VA.is_series_parallel()
True
sage: big_N = Poset({1: [2, 4], 2: [3], 4:[7], 5:[6], 6:[7]})
sage: big_N.is_series_parallel()
False
```

```
>>> from sage.all import *
>>> VA = Poset({Integer(1): [Integer(2), Integer(3)], Integer(4):␣
↪[Integer(5)], Integer(6): [Integer(5)]})
>>> VA.is_series_parallel()
True
>>> big_N = Poset({Integer(1): [Integer(2), Integer(4)], Integer(2):␣
↪[Integer(3)], Integer(4):[Integer(7)], Integer(5):[Integer(6)],␣
↪Integer(6):[Integer(7)]})
>>> big_N.is_series_parallel()
False
```

is_slender(*certificate=False*)

Return `True` if the poset is slender, and `False` otherwise.

A finite graded poset is *slender* if every rank 2 interval contains three or four elements, as defined in [Stan2009]. (This notion of "slender" is unrelated to the eponymous notion defined by Graetzer and Kelly in "The Free $\mathfrak{m}$-Lattice on the Poset H", Order 1 (1984), 47–65.)

This function *does not* check if the poset is graded or not. Instead it just returns `True` if the poset does not contain 5 distinct elements x, y, a, b and c such that $x \lessdot a, b, c \lessdot y$ where $\lessdot$ is the covering relation.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return either `(True, None)` or `(False, (a, b))` so that the interval $[a, b]$ has at least five elements. If `certificate=False` return `True` or `False`.

EXAMPLES:

```
sage: P = Poset(([1, 2, 3, 4], [[1, 2], [1, 3], [2, 4], [3, 4]]))
sage: P.is_slender()
True
sage: P = Poset(([1,2,3,4,5],[[1,2],[1,3],[1,4],[2,5],[3,5],[4,5]]))
sage: P.is_slender()
False

sage: # needs sage.groups
sage: W = WeylGroup(['A', 2])
sage: G = W.bruhat_poset()
sage: G.is_slender()
True
sage: W = WeylGroup(['A', 3])
sage: G = W.bruhat_poset()
```

(continues on next page)

(continued from previous page)

```
sage: G.is_slender()
True

sage: P = posets.IntegerPartitions(6)                                         #␣
↪needs sage.combinat
sage: P.is_slender(certificate=True)                                          #␣
↪needs sage.combinat
(False, ((6,), (3, 2, 1)))
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1), Integer(2), Integer(3), Integer(4)], [[Integer(1),
↪ Integer(2)], [Integer(1), Integer(3)], [Integer(2), Integer(4)],␣
↪[Integer(3), Integer(4)]]))
>>> P.is_slender()
True
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],
↪[[Integer(1),Integer(2)],[Integer(1),Integer(3)],[Integer(1),Integer(4)],
↪[Integer(2),Integer(5)],[Integer(3),Integer(5)],[Integer(4),Integer(5)]]))
>>> P.is_slender()
False

>>> # needs sage.groups
>>> W = WeylGroup(['A', Integer(2)])
>>> G = W.bruhat_poset()
>>> G.is_slender()
True
>>> W = WeylGroup(['A', Integer(3)])
>>> G = W.bruhat_poset()
>>> G.is_slender()
True

>>> P = posets.IntegerPartitions(Integer(6))                                  ␣
↪     # needs sage.combinat
>>> P.is_slender(certificate=True)                                         #␣
↪needs sage.combinat
(False, ((6,), (3, 2, 1)))
```

is_sperner()

Return `True` if the poset is Sperner, and `False` otherwise.

The poset is expected to be ranked.

A poset is Sperner, if no antichain is larger than the largest rank level (one of the sets of elements of the same rank) in the poset.

See Wikipedia article Sperner_property_of_a_partially_ordered_set

See also

width(), *dilworth_decomposition()*

EXAMPLES:

```
sage: posets.SetPartitions(3).is_sperner()                                  #␣
↪needs sage.combinat
True

sage: P = Poset({0: [3,4,5], 1: [5], 2: [5]})
sage: P.is_sperner()                                                        #␣
↪needs networkx
False
```

```
>>> from sage.all import *
>>> posets.SetPartitions(Integer(3)).is_sperner()                           ␣
↪     # needs sage.combinat
True

>>> P = Poset({Integer(0): [Integer(3),Integer(4),Integer(5)], Integer(1):␣
↪[Integer(5)], Integer(2): [Integer(5)]})
>>> P.is_sperner()                                                        #␣
↪needs networkx
False
```

isomorphic_subposets(*other*)

Return a list of subposets of `self` isomorphic to `other`.

By subposet we mean `self.subposet(X)` which is isomorphic to `other` and where `X` is a subset of elements of `self`.

INPUT:

- `other` – a finite poset

EXAMPLES:

```
sage: C2 = Poset({0:[1]})
sage: C3 = Poset({'a':['b'], 'b':['c']})
sage: L = sorted(x.cover_relations() for x in C3.isomorphic_subposets(C2))
sage: for x in L: print(x)
[['a', 'b']]
[['a', 'c']]
[['b', 'c']]

sage: D = Poset({1:[2,3], 2:[4], 3:[4]})
sage: N5 = posets.PentagonPoset()
sage: len(N5.isomorphic_subposets(D))
2
```

```
>>> from sage.all import *
>>> C2 = Poset({Integer(0):[Integer(1)]})
>>> C3 = Poset({'a':['b'], 'b':['c']})
>>> L = sorted(x.cover_relations() for x in C3.isomorphic_subposets(C2))
>>> for x in L: print(x)
[['a', 'b']]
[['a', 'c']]
[['b', 'c']]
```

(continues on next page)

(continued from previous page)

```
>>> D = Poset({Integer(1):[Integer(2),Integer(3)], Integer(2):[Integer(4)],␣
↪Integer(3):[Integer(4)]})
>>> N5 = posets.PentagonPoset()
>>> len(N5.isomorphic_subposets(D))
2
```

Note: If this function takes too much time, try using `isomorphic_subposets_iterator()`.

isomorphic_subposets_iterator(*other*)

Return an iterator over the subposets of `self` isomorphic to `other`.

By subposet we mean `self.subposet(X)` which is isomorphic to `other` and where `X` is a subset of elements of `self`.

INPUT:

- `other` – a finite poset

EXAMPLES:

```
sage: D = Poset({1:[2,3], 2:[4], 3:[4]})
sage: N5 = posets.PentagonPoset()
sage: for P in N5.isomorphic_subposets_iterator(D):
....:     print(P.cover_relations())
[[0, 1], [0, 2], [1, 4], [2, 4]]
[[0, 1], [0, 3], [1, 4], [3, 4]]
[[0, 1], [0, 2], [1, 4], [2, 4]]
[[0, 1], [0, 3], [1, 4], [3, 4]]
```

```
>>> from sage.all import *
>>> D = Poset({Integer(1):[Integer(2),Integer(3)], Integer(2):[Integer(4)],␣
↪Integer(3):[Integer(4)]})
>>> N5 = posets.PentagonPoset()
>>> for P in N5.isomorphic_subposets_iterator(D):
...     print(P.cover_relations())
[[0, 1], [0, 2], [1, 4], [2, 4]]
[[0, 1], [0, 3], [1, 4], [3, 4]]
[[0, 1], [0, 2], [1, 4], [2, 4]]
[[0, 1], [0, 3], [1, 4], [3, 4]]
```

Warning: This function will return same subposet as many times as there are automorphism on it. This is due to `subgraph_search_iterator()` returning labelled subgraphs. On the other hand, this function does not eat memory like `isomorphic_subposets()` does.

See also:

> *sage.combinat.posets.lattices.FiniteLatticePoset.isomorphic_sublattices_iterator()*.

join(*x*, *y*)

Return the join of two elements `x, y` in the poset if the join exists; and `None` otherwise.

EXAMPLES:

```
sage: D = Poset({1:[2,3], 2:[4], 3:[4]})
sage: D.join(2, 3)
4
sage: P = Poset({'e':['b'], 'f':['b', 'c', 'd'], 'g':['c', 'd'],
....:            'b':['a'], 'c':['a']})
sage: P.join('a', 'b')
'a'
sage: P.join('e', 'a')
'a'
sage: P.join('c', 'b')
'a'
sage: P.join('e', 'f')
'b'
sage: P.join('e', 'g')
'a'
sage: P.join('c', 'd') is None
True
sage: P.join('g', 'f') is None
True
```

```
>>> from sage.all import *
>>> D = Poset({Integer(1):[Integer(2),Integer(3)], Integer(2):[Integer(4)],␣
↪Integer(3):[Integer(4)]})
>>> D.join(Integer(2), Integer(3))
4
>>> P = Poset({'e':['b'], 'f':['b', 'c', 'd'], 'g':['c', 'd'],
...            'b':['a'], 'c':['a']})
>>> P.join('a', 'b')
'a'
>>> P.join('e', 'a')
'a'
>>> P.join('c', 'b')
'a'
>>> P.join('e', 'f')
'b'
>>> P.join('e', 'g')
'a'
>>> P.join('c', 'd') is None
True
>>> P.join('g', 'f') is None
True
```

jump_number(*certificate=False*)

Return the jump number of the poset.

A *jump* in a linear extension $[e_1, \ldots, e_n]$ of a poset P is a pair (e_i, e_{i+1}) so that e_{i+1} does not cover e_i in P. The jump number of a poset is the minimal number of jumps in linear extensions of a poset.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- If `certificate=True` return a pair (n, l) where n is the jump number and l is a linear extension with n jumps. If `certificate=False` return only the jump number.

EXAMPLES:

```
sage: B3 = posets.BooleanLattice(3)
sage: B3.jump_number()
3

sage: N = Poset({1: [3, 4], 2: [3]})
sage: N.jump_number(certificate=True)
(1, [1, 4, 2, 3])
```

```
>>> from sage.all import *
>>> B3 = posets.BooleanLattice(Integer(3))
>>> B3.jump_number()
3

>>> N = Poset({Integer(1): [Integer(3), Integer(4)], Integer(2): [Integer(3)]}
↪)
>>> N.jump_number(certificate=True)
(1, [1, 4, 2, 3])
```

ALGORITHM:

It is known that every poset has a greedy linear extension – an extension $[e_1, e_2, \dots, e_n]$ where every e_{i+1} is an upper cover of e_i if that is possible – with the smallest possible number of jumps; see [Sys1987].

Hence it suffices to test only those. We do that by backtracking.

The problem is proven to be NP-complete.

See also

is_jump_critical()

kazhdan_lusztig_polynomial(*x=None*, *y=None*, *q=None*, *canonical_labels=None*)

Return the Kazhdan-Lusztig polynomial $P_{x,y}(q)$ of the poset.

The poset is expected to be ranked.

We follow the definition given in [EPW14]. Let G denote a graded poset with unique minimal and maximal elements and χ_G denote the characteristic polynomial of G. Let I_x and F^x denote the principal order ideal and filter of x respectively. Define the *Kazhdan-Lusztig polynomial* of G as the unique polynomial $P_G(q)$ satisfying the following:

1. If $\operatorname{rank} G = 0$, then $P_G(q) = 1$.
2. If $\operatorname{rank} G > 0$, then $\deg P_G(q) < \frac{1}{2} \operatorname{rank} G$.
3. We have

$$q^{\operatorname{rank} G} P_G(q^{-1}) = \sum_{x \in G} \chi_{I_x}(q) P_{F^x}(q).$$

We then extend this to $P_{x,y}(q)$ by considering the subposet corresponding to the (closed) interval $[x, y]$. We also define $P_{\emptyset}(q) = 0$ (so if $x \not\leq y$, then $P_{x,y}(q) = 0$).

INPUT:

- `q` – (default: $q \in \mathbf{Z}[q]$) the indeterminate q
- `x` – (default: the minimal element) the element x
- `y` – (default: the maximal element) the element y
- `canonical_labels` – (optional) for subposets, use the canonical labeling (this can limit recursive calls for posets with large amounts of symmetry, but producing the labeling takes time); if not specified, this is `True` if `x` and `y` are both not specified and `False` otherwise

EXAMPLES:

```
sage: L = posets.BooleanLattice(3)
sage: L.kazhdan_lusztig_polynomial()
1
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(3))
>>> L.kazhdan_lusztig_polynomial()
1
```

```
sage: L = posets.SymmetricGroupWeakOrderPoset(4)
sage: L.kazhdan_lusztig_polynomial()
1
sage: x = '2314'
sage: y = '3421'
sage: L.kazhdan_lusztig_polynomial(x, y)
-q + 1
sage: L.kazhdan_lusztig_polynomial(x, y, var('t'))                          #␣
↪needs sage.symbolic
-t + 1
```

```
>>> from sage.all import *
>>> L = posets.SymmetricGroupWeakOrderPoset(Integer(4))
>>> L.kazhdan_lusztig_polynomial()
1
>>> x = '2314'
>>> y = '3421'
>>> L.kazhdan_lusztig_polynomial(x, y)
-q + 1
>>> L.kazhdan_lusztig_polynomial(x, y, var('t'))                          #␣
↪needs sage.symbolic
-t + 1
```

AUTHORS:

- Travis Scrimshaw (27-12-2014)

le(*x*, *y*)

Return `True` if x is less than or equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_lequal(2, 4)
True
sage: P.is_lequal(2, 2)
True
sage: P.is_lequal(0, 1)
False
sage: P.is_lequal(3, 2)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.is_lequal(Integer(2), Integer(4))
True
>>> P.is_lequal(Integer(2), Integer(2))
True
>>> P.is_lequal(Integer(0), Integer(1))
False
>>> P.is_lequal(Integer(3), Integer(2))
False
```

See also

is_less_than(), *is_gequal()*.

lequal_matrix(*ring=Integer Ring*, *sparse=False*)

Compute the matrix whose `(i,j)` entry is 1 if `self.linear_extension()[i] < self.linear_extension()[j]` and 0 otherwise.

INPUT:

- `ring` – the ring of coefficients (default: `ZZ`)
- `sparse` – whether the returned matrix is sparse or not (default: `True`)

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]], facade=False)
sage: LEQM = P.lequal_matrix(); LEQM
[1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 0 1]
[0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 1]
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1]
sage: LEQM[1,3]
1
sage: P.linear_extension()[1] < P.linear_extension()[3]
True
sage: LEQM[2,5]
```

(continues on next page)

(continued from previous page)

```
0
sage: P.linear_extension()[2] < P.linear_extension()[5]
False
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]], facade=False)
>>> LEQM = P.lequal_matrix(); LEQM
[1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 0 1]
[0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 1]
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1]
>>> LEQM[Integer(1),Integer(3)]
1
>>> P.linear_extension()[Integer(1)] < P.linear_extension()[Integer(3)]
True
>>> LEQM[Integer(2),Integer(5)]
0
>>> P.linear_extension()[Integer(2)] < P.linear_extension()[Integer(5)]
False
```

We now demonstrate the usage of the optional parameters:

```
sage: P.lequal_matrix(ring=QQ, sparse=False).parent()                          #␣
↪needs sage.libs.flint
Full MatrixSpace of 8 by 8 dense matrices over Rational Field
```

```
>>> from sage.all import *
>>> P.lequal_matrix(ring=QQ, sparse=False).parent()                          #␣
↪needs sage.libs.flint
Full MatrixSpace of 8 by 8 dense matrices over Rational Field
```

level_sets()

Return elements grouped by maximal number of cover relations from a minimal element.

This returns a list of lists `l` such that `l[i]` is the set of minimal elements of the poset obtained by removing the elements in `l[0], l[1], ..., l[i-1]`. (In particular, `l[0]` is the set of minimal elements of `self`.)

Every level is an antichain of the poset.

EXAMPLES:

```
sage: P = Poset({0:[1,2],1:[3],2:[3],3:[]})
sage: P.level_sets()
[[0], [1, 2], [3]]

sage: Q = Poset({0:[1,2], 1:[3], 2:[4], 3:[4]})
```

(continues on next page)

(continued from previous page)

```
sage: Q.level_sets()
[[0], [1, 2], [3], [4]]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(1),Integer(2)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[]})
>>> P.level_sets()
[[0], [1, 2], [3]]

>>> Q = Poset({Integer(0):[Integer(1),Integer(2)], Integer(1):[Integer(3)],␣
↪Integer(2):[Integer(4)], Integer(3):[Integer(4)]})
>>> Q.level_sets()
[[0], [1, 2], [3], [4]]
```

See also

`dilworth_decomposition()` to return elements grouped to chains.

lexicographic_sum(P)

Return the lexicographic sum using this poset as index.

In the lexicographic sum of posets P_t by index poset T we have $x \leq y$ if either $x \leq y$ in P_t for some $t \in T$, or $x \in P_i$, $y \in P_j$ and $i \leq j$ in T.

Informally said we substitute every element of T by corresponding poset P_t.

Mathematically, it is only defined when all P_t have no common element; here we force that by giving them different names in the resulting poset.

`disjoint_union()` and `ordinal_sum()` are special cases of lexicographic sum where the index poset is an (anti)chain. `ordinal_product()` is a special case where every P_t is same poset.

INPUT:

- `P` – dictionary whose keys are elements of this poset, values are posets

EXAMPLES:

```
sage: N = Poset({1: [3, 4], 2: [4]})
sage: P = {1: posets.PentagonPoset(), 2: N,
....:      3: posets.ChainPoset(3), 4: posets.AntichainPoset(4)}
sage: NP = N.lexicographic_sum(P); NP
Finite poset containing 16 elements
sage: sorted(NP.minimal_elements())
[(1, 0), (2, 1), (2, 2)]
```

```
>>> from sage.all import *
>>> N = Poset({Integer(1): [Integer(3), Integer(4)], Integer(2): [Integer(4)]}
↪)
>>> P = {Integer(1): posets.PentagonPoset(), Integer(2): N,
...      Integer(3): posets.ChainPoset(Integer(3)), Integer(4): posets.
↪AntichainPoset(Integer(4))}
>>> NP = N.lexicographic_sum(P); NP
```

(continues on next page)

(continued from previous page)

```
Finite poset containing 16 elements
>>> sorted(NP.minimal_elements())
[(1, 0), (2, 1), (2, 2)]
```

linear_extension(*linear_extension=None*, *check=True*)

Return a linear extension of this poset.

A linear extension of a finite poset P of size n is a total ordering $\pi := \pi_0\pi_1\ldots\pi_{n-1}$ of its elements such that $i < j$ whenever $\pi_i < \pi_j$ in the poset P.

INPUT:

- `linear_extension` – (default: `None`) a list of the elements of `self`
- `check` – boolean (default: `True`); whether to check that `linear_extension` is indeed a linear extension of `self`

EXAMPLES:

```
sage: P = Poset((divisors(15), attrcall("divides")), facade=True)
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(15)), attrcall("divides")), facade=True)
```

Without optional argument, the default linear extension of the poset is returned, as a plain list:

```
sage: P.linear_extension()
[1, 3, 5, 15]
```

```
>>> from sage.all import *
>>> P.linear_extension()
[1, 3, 5, 15]
```

Otherwise, a full-featured linear extension is constructed as an element of `P.linear_extensions()`:

```
sage: l = P.linear_extension([1,5,3,15]); l
[1, 5, 3, 15]
sage: type(l)
<class 'sage.combinat.posets.linear_extensions.LinearExtensionsOfPoset_with_
↪category.element_class'>
sage: l.parent()
The set of all linear extensions of Finite poset containing 4 elements
```

```
>>> from sage.all import *
>>> l = P.linear_extension([Integer(1),Integer(5),Integer(3),Integer(15)]); l
[1, 5, 3, 15]
>>> type(l)
<class 'sage.combinat.posets.linear_extensions.LinearExtensionsOfPoset_with_
↪category.element_class'>
>>> l.parent()
The set of all linear extensions of Finite poset containing 4 elements
```

By default, the linear extension is checked for correctness:

```
sage: l = P.linear_extension([1,3,15,5])
Traceback (most recent call last):
...
ValueError: [1, 3, 15, 5] is not a linear extension of Finite poset␣
↪containing 4 elements
```

```
>>> from sage.all import *
>>> l = P.linear_extension([Integer(1),Integer(3),Integer(15),Integer(5)])
Traceback (most recent call last):
...
ValueError: [1, 3, 15, 5] is not a linear extension of Finite poset␣
↪containing 4 elements
```

This can be disabled (at your own risks!) with:

```
sage: P.linear_extension([1,3,15,5], check=False)
[1, 3, 15, 5]
```

```
>>> from sage.all import *
>>> P.linear_extension([Integer(1),Integer(3),Integer(15),Integer(5)],␣
↪check=False)
[1, 3, 15, 5]
```

See also

is_linear_extension(), *linear_extensions()*

Todo

- Is it acceptable to have those two features for a single method?
- In particular, we miss a short idiom to get the default linear extension

linear_extensions(*facade=False*)

Return the enumerated set of all the linear extensions of this poset.

INPUT:

- `facade` – boolean (default: `False`); whether to return the linear extensions as plain lists

Warning

The `facade` option is not yet fully functional:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_
↪extension=True)
sage: L = P.linear_extensions(facade=True); L
The set of all linear extensions of
 Finite poset containing 6 elements with distinguished linear extension
sage: L([1, 2, 3, 4, 6, 12])
Traceback (most recent call last):
```

```
...
TypeError: Cannot convert list to sage.structure.element.Element
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")), linear_
↪extension=True)
>>> L = P.linear_extensions(facade=True); L
The set of all linear extensions of
 Finite poset containing 6 elements with distinguished linear extension
>>> L([Integer(1), Integer(2), Integer(3), Integer(4), Integer(6),␣
↪Integer(12)])
Traceback (most recent call last):
...
TypeError: Cannot convert list to sage.structure.element.Element
```

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: L = P.linear_extensions(); L
The set of all linear extensions of
 Finite poset containing 6 elements with distinguished linear extension
sage: l = L.an_element(); l
[1, 2, 3, 4, 6, 12]
sage: L.cardinality()
5
sage: L.list()
[[1, 2, 3, 4, 6, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12],
 [1, 2, 3, 6, 4, 12]]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")), linear_
↪extension=True)
>>> P.list()
[1, 2, 3, 4, 6, 12]
>>> L = P.linear_extensions(); L
The set of all linear extensions of
 Finite poset containing 6 elements with distinguished linear extension
>>> l = L.an_element(); l
[1, 2, 3, 4, 6, 12]
>>> L.cardinality()
5
>>> L.list()
[[1, 2, 3, 4, 6, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12],
 [1, 2, 3, 6, 4, 12]]
```

Each element is aware that it is a linear extension of P:

```
sage: type(l.parent())
<class 'sage.combinat.posets.linear_extensions.LinearExtensionsOfPoset_with_
↪category'>
```

```
>>> from sage.all import *
>>> type(l.parent())
<class 'sage.combinat.posets.linear_extensions.LinearExtensionsOfPoset_with_
↪category'>
```

With `facade=True`, the elements of `L` are plain lists instead:

```
sage: L = P.linear_extensions(facade=True)
sage: l = L.an_element()
sage: type(l)
<class 'list'>
```

```
>>> from sage.all import *
>>> L = P.linear_extensions(facade=True)
>>> l = L.an_element()
>>> type(l)
<class 'list'>
```

Warning

In Sage <= 4.8, this function used to return a plain list of lists. To recover the previous functionality, please use:

```
sage: L = list(P.linear_extensions(facade=True)); L
[[1, 2, 3, 4, 6, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12],
 [1, 2, 3, 6, 4, 12]]
sage: type(L[0])
<class 'list'>
```

```
>>> from sage.all import *
>>> L = list(P.linear_extensions(facade=True)); L
[[1, 2, 3, 4, 6, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12],
 [1, 2, 3, 6, 4, 12]]
>>> type(L[Integer(0)])
<class 'list'>
```

See also

linear_extension(), *is_linear_extension()*

linear_extensions_graph()

Return the linear extensions graph of the poset.

Vertices of the graph are linear extensions of the poset. Two vertices are connected by an edge if the linear extensions differ by only one adjacent transposition.

EXAMPLES:

```
sage: N = Poset({1: [3, 4], 2: [4]})
sage: G = N.linear_extensions_graph(); G
Graph on 5 vertices
sage: G.neighbors(N.linear_extension([1,2,3,4]))
[[2, 1, 3, 4], [1, 3, 2, 4], [1, 2, 4, 3]]

sage: chevron = Poset({1: [2, 6], 2: [3], 4: [3, 5], 6: [5]})
sage: G = chevron.linear_extensions_graph(); G
Graph on 22 vertices
sage: G.size()
36
```

```
>>> from sage.all import *
>>> N = Poset({Integer(1): [Integer(3), Integer(4)], Integer(2): [Integer(4)]}
↪)
>>> G = N.linear_extensions_graph(); G
Graph on 5 vertices
>>> G.neighbors(N.linear_extension([Integer(1),Integer(2),Integer(3),
↪Integer(4)]))
[[2, 1, 3, 4], [1, 3, 2, 4], [1, 2, 4, 3]]

>>> chevron = Poset({Integer(1): [Integer(2), Integer(6)], Integer(2):␣
↪[Integer(3)], Integer(4): [Integer(3), Integer(5)], Integer(6):␣
↪[Integer(5)]})
>>> G = chevron.linear_extensions_graph(); G
Graph on 22 vertices
>>> G.size()
36
```

linear_intervals_count()

Return the enumeration of linear intervals w.r.t. their cardinality.

An interval is linear if it is a total order.

OUTPUT: list of integers

See also

is_linear_interval()

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.linear_intervals_count()
[5, 5, 2]
sage: P = posets.TamariLattice(4)
```

(continues on next page)

(continued from previous page)

```
sage: P.linear_intervals_count()
[14, 21, 12, 2]
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.linear_intervals_count()
[5, 5, 2]
>>> P = posets.TamariLattice(Integer(4))
>>> P.linear_intervals_count()
[14, 21, 12, 2]
```

list()

List the elements of the poset. This just returns the result of *linear_extension()*.

EXAMPLES:

```
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] }, facade = False)
sage: D.list()
[0, 1, 2, 3, 4]
sage: type(D.list()[0])
<class 'sage.combinat.posets.posets.FinitePoset_with_category.element_class'>
```

```
>>> from sage.all import *
>>> D = Poset({ Integer(0):[Integer(1),Integer(2)], Integer(1):[Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)] }, facade = False)
>>> D.list()
[0, 1, 2, 3, 4]
>>> type(D.list()[Integer(0)])
<class 'sage.combinat.posets.posets.FinitePoset_with_category.element_class'>
```

lower_covers(x)

Return the list of lower covers of the element x.

A lower cover of x is an element y such that $y < x$ and there is no element z so that $y < z < x$.

EXAMPLES:

```
sage: P = Poset([[1,5], [2,6], [3], [4], [], [6,3], [4]])
sage: P.lower_covers(3)
[2, 5]
sage: P.lower_covers(0)
[]
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(5)], [Integer(2),Integer(6)], [Integer(3)],
↪ [Integer(4)], [], [Integer(6),Integer(3)], [Integer(4)]])
>>> P.lower_covers(Integer(3))
[2, 5]
>>> P.lower_covers(Integer(0))
[]
```

See also

upper_covers()

lower_covers_iterator(*x*)

Return an iterator over the lower covers of the element `x`.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[]})
sage: l0 = P.lower_covers_iterator(3)
sage: type(l0)
<class 'generator'>
sage: next(l0)
2
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[]})
>>> l0 = P.lower_covers_iterator(Integer(3))
>>> type(l0)
<class 'generator'>
>>> next(l0)
2
```

lt(*x*, *y*)

Return `True` if x is less than but not equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_less_than(1, 3)
True
sage: P.is_less_than(0, 1)
False
sage: P.is_less_than(2, 2)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.is_less_than(Integer(1), Integer(3))
True
>>> P.is_less_than(Integer(0), Integer(1))
False
>>> P.is_less_than(Integer(2), Integer(2))
False
```

For non-facade posets also < works:

```
sage: P = Poset({3: [1, 2]}, facade=False)
sage: P(1) < P(2)
False
```

```
>>> from sage.all import *
>>> P = Poset({Integer(3): [Integer(1), Integer(2)]}, facade=False)
>>> P(Integer(1)) < P(Integer(2))
False
```

See also:

is_lequal(), *is_greater_than()*.

magnitude()

Return the magnitude of `self`.

The magnitude is an integer defined as the sum of all Möbius numbers, and can be seen as some kind of Euler characteristic of the poset. It is additive under disjoint union and multiplicative under Cartesian product.

REFERENCES:

- [Lein2008] Tom Leinster, *The Euler Characteristic of a Category*, Documenta Mathematica, Vol. 13 (2008), 21-49 https://www.math.uni-bielefeld.de/documenta/vol-13/02.html
- https://golem.ph.utexas.edu/category/2011/06/the_magnitude_of_an_enriched_c.html

EXAMPLES:

```
sage: # needs sage.groups sage.libs.flint
sage: P = posets.PentagonPoset()
sage: P.magnitude()
1
sage: W = SymmetricGroup(4)
sage: P = W.noncrossing_partition_lattice().without_bounds()
sage: P.magnitude()
-4
sage: P = posets.TamariLattice(4).without_bounds()
sage: P.magnitude()
0
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.libs.flint
>>> P = posets.PentagonPoset()
>>> P.magnitude()
1
>>> W = SymmetricGroup(Integer(4))
>>> P = W.noncrossing_partition_lattice().without_bounds()
>>> P.magnitude()
-4
>>> P = posets.TamariLattice(Integer(4)).without_bounds()
>>> P.magnitude()
0
```

See also:

order_complex()

maximal_antichains()

Return the maximal antichains of the poset.

An antichain a of poset P is *maximal* if there is no element $e \in P \setminus a$ such that $a \cup \{e\}$ is an antichain.

EXAMPLES:

```
sage: P = Poset({'a':['b', 'c'], 'b':['d','e']})
sage: [sorted(anti) for anti in P.maximal_antichains()]
[['a'], ['b', 'c'], ['c', 'd', 'e']]

sage: posets.PentagonPoset().maximal_antichains()
[[0], [1, 2], [1, 3], [4]]
```

```
>>> from sage.all import *
>>> P = Poset({'a':['b', 'c'], 'b':['d','e']})
>>> [sorted(anti) for anti in P.maximal_antichains()]
[['a'], ['b', 'c'], ['c', 'd', 'e']]

>>> posets.PentagonPoset().maximal_antichains()
[[0], [1, 2], [1, 3], [4]]
```

See also

antichains(), *maximal_chains()*

maximal_chain_length()

Return the maximum length of a maximal chain in the poset.

The length here is the number of vertices.

EXAMPLES:

```
sage: P = posets.TamariLattice(5)
sage: P.maximal_chain_length()
11
```

```
>>> from sage.all import *
>>> P = posets.TamariLattice(Integer(5))
>>> P.maximal_chain_length()
11
```

See also

maximal_chains(), *maximal_chains_iterator()*

maximal_chains(*partial=None*)

Return all maximal chains of this poset.

Each chain is listed in increasing order.

INPUT:

- `partial` – list (optional); if given, the list `partial` is assumed to be the start of a maximal chain, and the function will find all maximal chains starting with the elements in `partial`

This is used in constructing the order complex for the poset.

EXAMPLES:

```
sage: P = posets.BooleanLattice(3)
sage: P.maximal_chains()
[[0, 1, 3, 7], [0, 1, 5, 7], [0, 2, 3, 7],
 [0, 2, 6, 7], [0, 4, 5, 7], [0, 4, 6, 7]]
sage: P.maximal_chains(partial=[0,2])
[[0, 2, 3, 7], [0, 2, 6, 7]]
sage: Q = posets.ChainPoset(6)
sage: Q.maximal_chains()
[[0, 1, 2, 3, 4, 5]]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(3))
>>> P.maximal_chains()
[[0, 1, 3, 7], [0, 1, 5, 7], [0, 2, 3, 7],
 [0, 2, 6, 7], [0, 4, 5, 7], [0, 4, 6, 7]]
>>> P.maximal_chains(partial=[Integer(0),Integer(2)])
[[0, 2, 3, 7], [0, 2, 6, 7]]
>>> Q = posets.ChainPoset(Integer(6))
>>> Q.maximal_chains()
[[0, 1, 2, 3, 4, 5]]
```

See also

maximal_antichains(), *chains()*

maximal_chains_iterator(*partial=None*)

Return an iterator over maximal chains.

Each chain is listed in increasing order.

INPUT:

- `partial` – list (optional); if given, the list `partial` is assumed to be the start of a maximal chain, and the function will yield all maximal chains starting with the elements in `partial`

EXAMPLES:

```
sage: P = posets.BooleanLattice(3)
sage: it = P.maximal_chains_iterator()
sage: next(it)
[0, 1, 3, 7]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(3))
>>> it = P.maximal_chains_iterator()
>>> next(it)
[0, 1, 3, 7]
```

See also

antichains_iterator()

maximal_elements()

Return the list of the maximal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.maximal_elements()
[4]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)],Integer(4):[]})
>>> P.maximal_elements()
[4]
```

See also

minimal_elements().

meet(*x*, *y*)

Return the meet of two elements `x, y` in the poset if the meet exists; and `None` otherwise.

EXAMPLES:

```
sage: D = Poset({1:[2,3], 2:[4], 3:[4]})
sage: D.meet(2, 3)
1
sage: P = Poset({'a':['b', 'c'], 'b':['e', 'f'], 'c':['f', 'g'],
....:             'd':['f', 'g']})
sage: P.meet('a', 'b')
'a'
sage: P.meet('e', 'a')
'a'
sage: P.meet('c', 'b')
'a'
sage: P.meet('e', 'f')
'b'
sage: P.meet('e', 'g')
'a'
sage: P.meet('c', 'd') is None
True
sage: P.meet('g', 'f') is None
True
```

```
>>> from sage.all import *
>>> D = Poset({Integer(1):[Integer(2),Integer(3)], Integer(2):[Integer(4)],␣
↪Integer(3):[Integer(4)]})
```

(continues on next page)

(continued from previous page)

```
>>> D.meet(Integer(2), Integer(3))
1
>>> P = Poset({'a':['b', 'c'], 'b':['e', 'f'], 'c':['f', 'g'],
...             'd':['f', 'g']})
>>> P.meet('a', 'b')
'a'
>>> P.meet('e', 'a')
'a'
>>> P.meet('c', 'b')
'a'
>>> P.meet('e', 'f')
'b'
>>> P.meet('e', 'g')
'a'
>>> P.meet('c', 'd') is None
True
>>> P.meet('g', 'f') is None
True
```

minimal_elements()

Return the list of the minimal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P(0) in P.minimal_elements()
True
sage: P(1) in P.minimal_elements()
True
sage: P(2) in P.minimal_elements()
True
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4)],Integer(4):[]})
>>> P(Integer(0)) in P.minimal_elements()
True
>>> P(Integer(1)) in P.minimal_elements()
True
>>> P(Integer(2)) in P.minimal_elements()
True
```

See also

maximal_elements().

moebius_function(x, y)

Return the value of the Möbius function of the poset on the elements x and y.

EXAMPLES:

```
sage: P = Poset([[1,2,3],[4],[4],[4],[]])
sage: P.moebius_function(P(0),P(4))
2
sage: sum(P.moebius_function(P(0),v) for v in P)
0
sage: sum(abs(P.moebius_function(P(0),v))
....:     for v in P)
6
sage: for u,v in P.cover_relations_iterator():
....:     if P.moebius_function(u,v) != -1:
....:         print("Bug in moebius_function!")
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(2),Integer(3)],[Integer(4)],[Integer(4)],
↪[Integer(4)],[]])
>>> P.moebius_function(P(Integer(0)),P(Integer(4)))
2
>>> sum(P.moebius_function(P(Integer(0)),v) for v in P)
0
>>> sum(abs(P.moebius_function(P(Integer(0)),v))
...     for v in P)
6
>>> for u,v in P.cover_relations_iterator():
...     if P.moebius_function(u,v) != -Integer(1):
...         print("Bug in moebius_function!")
```

```
sage: Q = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: Q.moebius_function(Q(0),Q(7))
0
sage: Q.moebius_function(Q(0),Q(5))
0
sage: Q.moebius_function(Q(2),Q(7))
2
sage: Q.moebius_function(Q(3),Q(3))
1
sage: sum([Q.moebius_function(Q(0),v) for v in Q])
0
```

```
>>> from sage.all import *
>>> Q = Poset([[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]])
>>> Q.moebius_function(Q(Integer(0)),Q(Integer(7)))
0
>>> Q.moebius_function(Q(Integer(0)),Q(Integer(5)))
0
>>> Q.moebius_function(Q(Integer(2)),Q(Integer(7)))
2
>>> Q.moebius_function(Q(Integer(3)),Q(Integer(3)))
1
>>> sum([Q.moebius_function(Q(Integer(0)),v) for v in Q])
0
```

moebius_function_matrix(*ring=Integer Ring*, *sparse=False*)

Return a matrix whose `(i,j)` entry is the value of the Möbius function evaluated at `self.linear_extension()[i]` and `self.linear_extension()[j]`.

INPUT:

- `ring` – the ring of coefficients (default: `ZZ`)
- `sparse` – whether the returned matrix is sparse or not (default: `True`)

EXAMPLES:

```
sage: P = Poset([[4,2,3],[],[1],[1],[1]])
sage: x,y = (P.linear_extension()[0],P.linear_extension()[1])
sage: P.moebius_function(x,y)
-1
sage: M = P.moebius_function_matrix(); M                                    #␣
↪needs sage.libs.flint
[ 1 -1 -1 -1  2]
[ 0  1  0  0 -1]
[ 0  0  1  0 -1]
[ 0  0  0  1 -1]
[ 0  0  0  0  1]
sage: M[0,4]                                                                #␣
↪needs sage.libs.flint
2
sage: M[0,1]                                                                #␣
↪needs sage.libs.flint
-1
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(4),Integer(2),Integer(3)],[],[Integer(1)],
↪[Integer(1)],[Integer(1)]])
>>> x,y = (P.linear_extension()[Integer(0)],P.linear_extension()[Integer(1)])
>>> P.moebius_function(x,y)
-1
>>> M = P.moebius_function_matrix(); M                                    #␣
↪needs sage.libs.flint
[ 1 -1 -1 -1  2]
[ 0  1  0  0 -1]
[ 0  0  1  0 -1]
[ 0  0  0  1 -1]
[ 0  0  0  0  1]
>>> M[Integer(0),Integer(4)]                                                ␣
↪                # needs sage.libs.flint
2
>>> M[Integer(0),Integer(1)]                                                ␣
↪                # needs sage.libs.flint
-1
```

We now demonstrate the usage of the optional parameters:

```
sage: P.moebius_function_matrix(ring=QQ, sparse=False).parent()            #␣
↪needs sage.libs.flint
Full MatrixSpace of 5 by 5 dense matrices over Rational Field
```

```
>>> from sage.all import *
>>> P.moebius_function_matrix(ring=QQ, sparse=False).parent()            #␣
↪needs sage.libs.flint
Full MatrixSpace of 5 by 5 dense matrices over Rational Field
```

open_interval(*x*, *y*)

Return the list of elements z such that $x < z < y$ in the poset.

EXAMPLES:

```
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.open_interval(2, 100)
[4, 10, 20, 50]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(1000)), attrcall("divides")))
>>> P.open_interval(Integer(2), Integer(100))
[4, 10, 20, 50]
```

See also

closed_interval()

order_complex(*on_ints=False*)

Return the order complex associated to this poset.

The order complex is the simplicial complex with vertices equal to the elements of the poset, and faces given by the chains.

INPUT:

- `on_ints` – boolean (default: `False`)

OUTPUT: an order complex of type `SimplicialComplex`

EXAMPLES:

```
sage: P = posets.BooleanLattice(3)
sage: S = P.order_complex(); S
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6, 7) and 6 facets
sage: S.f_vector()
[1, 8, 19, 18, 6]
sage: S.homology()       # S is contractible
{0: 0, 1: 0, 2: 0, 3: 0}
sage: Q = P.subposet([1,2,3,4,5,6])
sage: Q.order_complex().homology()     # a circle
{0: 0, 1: Z}

sage: P = Poset((divisors(15), attrcall("divides")), facade=True)
sage: P.order_complex()
Simplicial complex with vertex set (1, 3, 5, 15) and
 facets {(1, 3, 15), (1, 5, 15)}
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(3))
>>> S = P.order_complex(); S
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6, 7) and 6 facets
>>> S.f_vector()
[1, 8, 19, 18, 6]
>>> S.homology()      # S is contractible
{0: 0, 1: 0, 2: 0, 3: 0}
>>> Q = P.subposet([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6)])
>>> Q.order_complex().homology()     # a circle
{0: 0, 1: Z}

>>> P = Poset((divisors(Integer(15)), attrcall("divides")), facade=True)
>>> P.order_complex()
Simplicial complex with vertex set (1, 3, 5, 15) and
 facets {(1, 3, 15), (1, 5, 15)}
```

If `on_ints`, then the elements of the poset are labelled $0, 1, \dots$ in the chain complex:

```
sage: P.order_complex(on_ints=True)
Simplicial complex with vertex set (0, 1, 2, 3) and
 facets {(0, 1, 3), (0, 2, 3)}
```

```
>>> from sage.all import *
>>> P.order_complex(on_ints=True)
Simplicial complex with vertex set (0, 1, 2, 3) and
 facets {(0, 1, 3), (0, 2, 3)}
```

order_filter(*elements*)

Return the order filter generated by the elements of an iterable `elements`.

I is an order filter if, for any x in I and y such that $y \geq x$, then y is in I. This is also called upper set or upset.

EXAMPLES:

```
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.order_filter([20, 25])
[20, 40, 25, 50, 100, 200, 125, 250, 500, 1000]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(1000)), attrcall("divides")))
>>> P.order_filter([Integer(20), Integer(25)])
[20, 40, 25, 50, 100, 200, 125, 250, 500, 1000]
```

See also

order_ideal(), `principal_order_filter()`.

order_ideal(*elements*)

Return the order ideal generated by the elements of an iterable `elements`.

I is an order ideal if, for any x in I and y such that $y \leq x$, then y is in I. This is also called lower set or downset.

EXAMPLES:

```
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.order_ideal([20, 25])
[1, 2, 4, 5, 10, 20, 25]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(1000)), attrcall("divides")))
>>> P.order_ideal([Integer(20), Integer(25)])
[1, 2, 4, 5, 10, 20, 25]
```

See also

order_filter(), principal_order_ideal().

order_ideal_cardinality(*elements*)

Return the cardinality of the order ideal generated by `elements`.

The elements I is an order ideal if, for any $x \in I$ and y such that $y \leq x$, then $y \in I$.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: P.order_ideal_cardinality([7,10])
10
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> P.order_ideal_cardinality([Integer(7),Integer(10)])
10
```

order_ideal_plot(*elements*)

Return a plot of the order ideal generated by the elements of an iterable `elements`.

I is an order ideal if, for any x in I and y such that $y \leq x$, then y is in I. This is also called lower set or downset.

EXAMPLES:

```
sage: # needs sage.plot
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.order_ideal_plot([20, 25])
Graphics object consisting of 41 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> P = Poset((divisors(Integer(1000)), attrcall("divides")))
>>> P.order_ideal_plot([Integer(20), Integer(25)])
Graphics object consisting of 41 graphics primitives
```

order_polynomial()

Return the order polynomial of the poset.

The order polynomial $\Omega_P(q)$ of a poset P is defined as the unique polynomial S such that for each integer $m \geq 1$, $S(m)$ is the number of order-preserving maps from P to $\{1, \ldots, m\}$.

See sections 3.12 and 3.15 of [EnumComb1], and also [St1986].

EXAMPLES:

```
sage: P = posets.AntichainPoset(3)
sage: P.order_polynomial()
q^3

sage: P = posets.ChainPoset(3)
sage: f = P.order_polynomial(); f
1/6*q^3 + 1/2*q^2 + 1/3*q
sage: [f(i) for i in range(4)]
[0, 1, 4, 10]
```

```
>>> from sage.all import *
>>> P = posets.AntichainPoset(Integer(3))
>>> P.order_polynomial()
q^3

>>> P = posets.ChainPoset(Integer(3))
>>> f = P.order_polynomial(); f
1/6*q^3 + 1/2*q^2 + 1/3*q
>>> [f(i) for i in range(Integer(4))]
[0, 1, 4, 10]
```

See also:

order_polytope()

order_polytope()

Return the order polytope of the poset `self`.

The order polytope of a finite poset P is defined as the subset of $\mathbf{R}^P$ consisting of all maps $x : P \to \mathbf{R}$ satisfying

$$0 \leq x(p) \leq 1 \text{ for all } p \in P,$$

and

$$x(p) \leq x(q) \text{ for all } p, q \in P \text{ satisfying } p < q.$$

This polytope was defined and studied in [St1986].

EXAMPLES:

```
sage: P = posets.AntichainPoset(3)
sage: Q = P.order_polytope(); Q                                          #␣
↪needs sage.geometry.polyhedron
```

(continues on next page)

(continued from previous page)

```
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: P = posets.PentagonPoset()
sage: Q = P.order_polytope(); Q                                      #␣
↪needs sage.geometry.polyhedron
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 8 vertices

sage: P = Poset([[1,2,3],[[1,2],[1,3]]])
sage: Q = P.order_polytope()                                         #␣
↪needs sage.geometry.polyhedron
sage: Q.contains((1,0,0))                                            #␣
↪needs sage.geometry.polyhedron
False
sage: Q.contains((0,1,1))                                            #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> P = posets.AntichainPoset(Integer(3))
>>> Q = P.order_polytope(); Q                                       #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
>>> P = posets.PentagonPoset()
>>> Q = P.order_polytope(); Q                                       #␣
↪needs sage.geometry.polyhedron
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 8 vertices

>>> P = Poset([[Integer(1),Integer(2),Integer(3)],[[Integer(1),Integer(2)],
↪[Integer(1),Integer(3)]]])
>>> Q = P.order_polytope()                                          #␣
↪needs sage.geometry.polyhedron
>>> Q.contains((Integer(1),Integer(0),Integer(0)))                   ␣
↪                         # needs sage.geometry.polyhedron
False
>>> Q.contains((Integer(0),Integer(1),Integer(1)))                   ␣
↪                         # needs sage.geometry.polyhedron
True
```

ordinal_product(*other*, *labels='pairs'*)

Return the ordinal product of `self` and `other`.

The ordinal product of two posets P and Q is a partial order on the Cartesian product of the underlying sets of P and Q, defined as follows (see [EnumComb1], p. 284).

In the ordinal product, $(p, q) \leq (p', q')$ if either $p \leq p'$ or $p = p'$ and $q \leq q'$.

This construction is not symmetric in P and Q. Informally said we put a copy of Q in place of every element of P.

INPUT:

- `other` – a poset
- `labels` – either `'integers'` or `'pairs'` (default); how the resulting poset will be labeled

EXAMPLES:

```
sage: P1 = Poset((['a', 'b'], [['a', 'b']]))
sage: P2 = Poset((['c', 'd'], [['c', 'd']]))
sage: P = P1.ordinal_product(P2); P
Finite poset containing 4 elements
sage: sorted(P.cover_relations())
[[('a', 'c'), ('a', 'd')], [('a', 'd'), ('b', 'c')],
[('b', 'c'), ('b', 'd')]]
```

```
>>> from sage.all import *
>>> P1 = Poset((['a', 'b'], [['a', 'b']]))
>>> P2 = Poset((['c', 'd'], [['c', 'd']]))
>>> P = P1.ordinal_product(P2); P
Finite poset containing 4 elements
>>> sorted(P.cover_relations())
[[('a', 'c'), ('a', 'd')], [('a', 'd'), ('b', 'c')],
[('b', 'c'), ('b', 'd')]]
```

See also

product(), *ordinal_sum()*

ordinal_sum(*other*, *labels='pairs'*)

Return a poset or (semi)lattice isomorphic to ordinal sum of the poset with `other`.

The ordinal sum of P and Q is a poset that contains every element and relation from both P and Q, and where every element of P is smaller than any element of Q.

Mathematically, it is only defined when P and Q have no common element; here we force that by giving them different names in the resulting poset.

The ordinal sum on lattices is a lattice; resp. for meet- and join-semilattices.

INPUT:

- `other` – poset
- `labels` – (defaults to 'pairs') If set to 'pairs', each element `v` in this poset will be named `(0,v)` and each element `u` in `other` will be named `(1,u)` in the result. If set to 'integers', the elements of the result will be relabeled with consecutive integers.

EXAMPLES:

```
sage: P1 = Poset( ([1, 2, 3, 4], [[1, 2], [1, 3], [1, 4]]) )
sage: P2 = Poset( ([1, 2, 3,], [[2,1], [3,1]]) )
sage: P3 = P1.ordinal_sum(P2); P3
Finite poset containing 7 elements
sage: len(P1.maximal_elements())*len(P2.minimal_elements())
6
sage: len(P1.cover_relations()+P2.cover_relations())
5
sage: len(P3.cover_relations()) # Every element of P2 is greater than␣
↪elements of P1.
11
sage: P3.list()  # random
```

(continues on next page)

(continued from previous page)

```
[(0, 1), (0, 2), (0, 4), (0, 3), (1, 2), (1, 3), (1, 1)]
sage: P4 = P1.ordinal_sum(P2, labels='integers')
sage: P4.list()  # random
[0, 1, 2, 3, 5, 6, 4]
```

```
>>> from sage.all import *
>>> P1 = Poset( ([Integer(1), Integer(2), Integer(3), Integer(4)],␣
↪[[Integer(1), Integer(2)], [Integer(1), Integer(3)], [Integer(1),␣
↪Integer(4)]]) )
>>> P2 = Poset( ([Integer(1), Integer(2), Integer(3),], [[Integer(2),
↪Integer(1)], [Integer(3),Integer(1)]]) )
>>> P3 = P1.ordinal_sum(P2); P3
Finite poset containing 7 elements
>>> len(P1.maximal_elements())*len(P2.minimal_elements())
6
>>> len(P1.cover_relations()+P2.cover_relations())
5
>>> len(P3.cover_relations()) # Every element of P2 is greater than elements␣
↪of P1.
11
>>> P3.list()  # random
[(0, 1), (0, 2), (0, 4), (0, 3), (1, 2), (1, 3), (1, 1)]
>>> P4 = P1.ordinal_sum(P2, labels='integers')
>>> P4.list()  # random
[0, 1, 2, 3, 5, 6, 4]
```

Return type depends on input types:

```
sage: P = Poset({1:[2]}); P
Finite poset containing 2 elements
sage: JL = JoinSemilattice({1:[2]}); JL
Finite join-semilattice containing 2 elements
sage: L = LatticePoset({1:[2]}); L
Finite lattice containing 2 elements
sage: P.ordinal_sum(L)
Finite poset containing 4 elements
sage: L.ordinal_sum(JL)
Finite join-semilattice containing 4 elements
sage: L.ordinal_sum(L)
Finite lattice containing 4 elements
```

```
>>> from sage.all import *
>>> P = Poset({Integer(1):[Integer(2)]}); P
Finite poset containing 2 elements
>>> JL = JoinSemilattice({Integer(1):[Integer(2)]}); JL
Finite join-semilattice containing 2 elements
>>> L = LatticePoset({Integer(1):[Integer(2)]}); L
Finite lattice containing 2 elements
>>> P.ordinal_sum(L)
Finite poset containing 4 elements
>>> L.ordinal_sum(JL)
Finite join-semilattice containing 4 elements
```

(continues on next page)

(continued from previous page)

```
>>> L.ordinal_sum(L)
Finite lattice containing 4 elements
```

See also:

ordinal_summands(), *disjoint_union()*, *sage.combinat.posets.lattices.FiniteLatticePoset.vertical_composition()*

ordinal_summands()

Return the ordinal summands of the poset as subposets.

The ordinal summands of a poset P is the longest list of non-empty subposets $P_1, \ldots, P_n$ whose ordinal sum is P. This decomposition is unique.

EXAMPLES:

```
sage: P = Poset({'a': ['c', 'd'], 'b': ['d'], 'c': ['x', 'y'],
....: 'd': ['x', 'y']})
sage: parts = P.ordinal_summands(); parts
[Finite poset containing 4 elements, Finite poset containing 2 elements]
sage: sorted(parts[0])
['a', 'b', 'c', 'd']
sage: Q = parts[0].ordinal_sum(parts[1])
sage: Q.is_isomorphic(P)
True
```

```
>>> from sage.all import *
>>> P = Poset({'a': ['c', 'd'], 'b': ['d'], 'c': ['x', 'y'],
... 'd': ['x', 'y']})
>>> parts = P.ordinal_summands(); parts
[Finite poset containing 4 elements, Finite poset containing 2 elements]
>>> sorted(parts[Integer(0)])
['a', 'b', 'c', 'd']
>>> Q = parts[Integer(0)].ordinal_sum(parts[Integer(1)])
>>> Q.is_isomorphic(P)
True
```

See also:

ordinal_sum()

ALGORITHM:

Suppose that a poset P is the ordinal sum of posets L and U. Then P contains maximal antichains l and u such that every element of u covers every element of l; they correspond to maximal elements of L and minimal elements of U.

We consider a linear extension $x_1, \ldots, x_n$ of the poset's elements.

We keep track of the maximal elements of subposet induced by elements $0, \ldots, x_i$ and minimal elements of subposet induced by elements $x_{i+1}, \ldots, x_n$, incrementing i one by one. We then check if l and u fit the previous description.

p_partition_enumerator(*tup*, *R*, *weights=None*, *check=False*)

Return a P-partition enumerator of `self`.

Given a total order $\prec$ on the elements of a finite poset P (the order of P and the total order $\prec$ can be unrelated; in particular, the latter does not have to extend the former), a P-partition enumerator is the quasisymmetric function $\sum_f \prod_{p \in P} x_{f(p)}$, where the first sum is taken over all P-partitions f.

A P-partition is a function $f : P \to \{1, 2, 3, ...\}$ satisfying the following properties for any two elements i and j of P satisfying $i <_P j$:

- if $i \prec j$ then $f(i) \leq f(j)$,
- if $j \prec i$ then $f(i) < f(j)$.

The optional argument `weights` allows constructing a generalized ("weighted") version of the P-partition enumerator. Namely, `weights` should be a dictionary whose keys are the elements of `P`. Then, the generalized P-partition enumerator corresponding to weights `weights` is $\sum_f \prod_{p \in P} x_{f(p)}^{w(p)}$, where the sum is again over all P-partitions f. Here, $w(p)$ is `weights[p]`. The classical P-partition enumerator is the particular case obtained when all p satisfy $w(p) = 1$.

In the language of [Grinb2016a], the generalized P-partition enumerator is the quasisymmetric function $\Gamma(\mathbf{E}, w)$, where $\mathbf{E}$ is the special double poset $(P, <_P, \prec)$, and where w is the dictionary `weights` (regarded as a function from P to the positive integers).

INPUT:

- `tup` – the tuple containing all elements of P (each of them exactly once), in the order dictated by the total order $\prec$
- `R` – a commutative ring
- `weights` – (optional) a dictionary of positive integers, indexed by elements of P; any missing item will be understood as 1

OUTPUT:

The P-partition enumerator of `self` according to `tup` in the algebra $QSym$ of quasisymmetric functions over the base ring R.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = Poset([[1,2,3,4],[[1,4],[2,4],[4,3]]])
sage: FP = P.p_partition_enumerator((3,1,2,4), QQ, check=True); FP
2*M[1, 1, 1, 1] + 2*M[1, 2, 1] + M[2, 1, 1] + M[3, 1]
sage: expansion = FP.expand(5)
sage: xs = expansion.parent().gens()
sage: expansion == sum(xs[a]*xs[b]*xs[c]*xs[d]
....:                  for a in range(5) for b in range(5)
....:                  for c in range(5) for d in range(5)
....:                  if a <= b and c <= b and b < d)
True

sage: P = Poset([[],[]])
sage: FP = P.p_partition_enumerator((), QQ, check=True); FP                     #␣
↪needs sage.combinat
M[]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = Poset([[Integer(1),Integer(2),Integer(3),Integer(4)],[[Integer(1),
↪Integer(4)],[Integer(2),Integer(4)],[Integer(4),Integer(3)]]])
>>> FP = P.p_partition_enumerator((Integer(3),Integer(1),Integer(2),
↪Integer(4)), QQ, check=True); FP
2*M[1, 1, 1, 1] + 2*M[1, 2, 1] + M[2, 1, 1] + M[3, 1]
>>> expansion = FP.expand(Integer(5))
>>> xs = expansion.parent().gens()
>>> expansion == sum(xs[a]*xs[b]*xs[c]*xs[d]
...                      for a in range(Integer(5)) for b in range(Integer(5))
...                      for c in range(Integer(5)) for d in range(Integer(5))
...                      if a <= b and c <= b and b < d)
True

>>> P = Poset([[],[]])
>>> FP = P.p_partition_enumerator((), QQ, check=True); FP                    #␣
↪needs sage.combinat
M[]
```

With the `weights` parameter:

```
sage: P = Poset([[1,2,3,4],[[1,4],[2,4],[4,3]]])
sage: FP = P.p_partition_enumerator((3,1,2,4), QQ,                              #␣
↪needs sage.combinat
....:             weights={1: 1, 2: 2, 3: 1, 4: 1}, check=True); FP
M[1, 2, 1, 1] + M[1, 3, 1] + M[2, 1, 1, 1] + M[2, 2, 1] + M[3, 1, 1] + M[4, 1]
sage: FP = P.p_partition_enumerator((3,1,2,4), QQ,                              #␣
↪needs sage.combinat
....:             weights={2: 2}, check=True); FP
M[1, 2, 1, 1] + M[1, 3, 1] + M[2, 1, 1, 1] + M[2, 2, 1] + M[3, 1, 1] + M[4, 1]

sage: P = Poset([['a','b','c'], [['a','b'], ['a','c']]])
sage: FP = P.p_partition_enumerator(('b','c','a'), QQ,                          #␣
↪needs sage.combinat
....:             weights={'a': 3, 'b': 5, 'c': 7}, check=True); FP
M[3, 5, 7] + M[3, 7, 5] + M[3, 12]

sage: P = Poset([['a','b','c'], [['a','c'], ['b','c']]])
sage: FP = P.p_partition_enumerator(('b','c','a'), QQ,                          #␣
↪needs sage.combinat
....:             weights={'a': 3, 'b': 5, 'c': 7}, check=True); FP
M[3, 5, 7] + M[3, 12] + M[5, 3, 7] + M[8, 7]
sage: FP = P.p_partition_enumerator(('a','b','c'), QQ,                          #␣
↪needs sage.combinat
....:             weights={'a': 3, 'b': 5, 'c': 7}, check=True); FP
M[3, 5, 7] + M[3, 12] + M[5, 3, 7] + M[5, 10] + M[8, 7] + M[15]
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(2),Integer(3),Integer(4)],[[Integer(1),
↪Integer(4)],[Integer(2),Integer(4)],[Integer(4),Integer(3)]]])
>>> FP = P.p_partition_enumerator((Integer(3),Integer(1),Integer(2),
↪Integer(4)), QQ,                          # needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
...           weights={Integer(1): Integer(1), Integer(2): Integer(2),␣
↪Integer(3): Integer(1), Integer(4): Integer(1)}, check=True); FP
M[1, 2, 1, 1] + M[1, 3, 1] + M[2, 1, 1, 1] + M[2, 2, 1] + M[3, 1, 1] + M[4, 1]
>>> FP = P.p_partition_enumerator((Integer(3),Integer(1),Integer(2),
↪Integer(4)), QQ,                     # needs sage.combinat
...           weights={Integer(2): Integer(2)}, check=True); FP
M[1, 2, 1, 1] + M[1, 3, 1] + M[2, 1, 1, 1] + M[2, 2, 1] + M[3, 1, 1] + M[4, 1]

>>> P = Poset([['a','b','c'], [['a','b'], ['a','c']]])
>>> FP = P.p_partition_enumerator(('b','c','a'), QQ,                      #␣
↪needs sage.combinat
...           weights={'a': Integer(3), 'b': Integer(5), 'c': Integer(7)},␣
↪check=True); FP
M[3, 5, 7] + M[3, 7, 5] + M[3, 12]

>>> P = Poset([['a','b','c'], [['a','c'], ['b','c']]])
>>> FP = P.p_partition_enumerator(('b','c','a'), QQ,                      #␣
↪needs sage.combinat
...           weights={'a': Integer(3), 'b': Integer(5), 'c': Integer(7)},␣
↪check=True); FP
M[3, 5, 7] + M[3, 12] + M[5, 3, 7] + M[8, 7]
>>> FP = P.p_partition_enumerator(('a','b','c'), QQ,                      #␣
↪needs sage.combinat
...           weights={'a': Integer(3), 'b': Integer(5), 'c': Integer(7)},␣
↪check=True); FP
M[3, 5, 7] + M[3, 12] + M[5, 3, 7] + M[5, 10] + M[8, 7] + M[15]
```

plot(*label_elements=True*, *element_labels=None*, *layout='acyclic'*, *cover_labels=None*, ***kwds*)

Return a Graphic object for the Hasse diagram of the poset.

If the poset is ranked, the plot uses the rank function for the heights of the elements.

INPUT:

- Options to change element look:
  - `element_colors` – dictionary where keys are colors and values are lists of elements
  - `element_color` – a color for elements not set in `element_colors`
  - `element_shape` – the shape of elements, like `'s'` for square; see https://matplotlib.org/api/markers_api.html for the list
  - `element_size` – (default: 200) the size of elements
  - `label_elements` – boolean (default: `True`); whether to display element labels
  - `element_labels` (default: `None`); a dictionary where keys are elements and values are labels to show
- Options to change cover relation look:
  - `cover_colors` – dictionary where keys are colors and values are lists of cover relations given as pairs of elements
  - `cover_color` – a color for elements not set in `cover_colors`
  - `cover_style` – style for cover relations: `'solid'`, `'dashed'`, `'dotted'` or `'dashdot'`

   - `cover_labels` – dictionary, list or function representing labels of the covers of the poset. When set to `None` (default) no label is displayed on the edges of the Hasse Diagram.
   - `cover_labels_background` – a background color for cover relations. The default is "white". To achieve a transparent background use "transparent".
- Options to change overall look:
   - `figsize` – (default: 8) size of the whole plot
   - `title` – a title for the plot
   - `fontsize` – fontsize for the title
   - `border` – (default: `False`) whether to draw a border over the plot

Note

All options of `GenericGraph.plot` are also available through this function.

EXAMPLES:

This function can be used without any parameters:

```
sage: # needs sage.plot
sage: D12 = posets.DivisorLattice(12)
sage: D12.plot()
Graphics object consisting of 14 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> D12 = posets.DivisorLattice(Integer(12))
>>> D12.plot()
Graphics object consisting of 14 graphics primitives
```

Just the abstract form of the poset; examples of relabeling:

```
sage: # needs sage.plot
sage: D12.plot(label_elements=False)
Graphics object consisting of 8 graphics primitives
sage: d = {1: 0, 2: 'a', 3: 'b', 4: 'c', 6: 'd', 12: 1}
sage: D12.plot(element_labels=d)
Graphics object consisting of 14 graphics primitives
sage: d = {i: str(factor(i)) for i in D12}
sage: D12.plot(element_labels=d)
Graphics object consisting of 14 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> D12.plot(label_elements=False)
Graphics object consisting of 8 graphics primitives
>>> d = {Integer(1): Integer(0), Integer(2): 'a', Integer(3): 'b',␣
↪Integer(4): 'c', Integer(6): 'd', Integer(12): Integer(1)}
>>> D12.plot(element_labels=d)
Graphics object consisting of 14 graphics primitives
```

(continues on next page)

(continued from previous page)

```
>>> d = {i: str(factor(i)) for i in D12}
>>> D12.plot(element_labels=d)
Graphics object consisting of 14 graphics primitives
```

Some settings for coverings:

```
sage: # needs sage.plot
sage: d = {(a, b): b/a for a, b in D12.cover_relations()}
sage: D12.plot(cover_labels=d, cover_color='gray', cover_style='dotted')
Graphics object consisting of 21 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> d = {(a, b): b/a for a, b in D12.cover_relations()}
>>> D12.plot(cover_labels=d, cover_color='gray', cover_style='dotted')
Graphics object consisting of 21 graphics primitives
```

To emphasize some elements and show some options:

```
sage: # needs sage.plot
sage: L = LatticePoset({0: [1, 2, 3, 4], 1: [12], 2: [6, 7],
....:                   3: [5, 9], 4: [5, 6, 10, 11], 5: [13],
....:                   6: [12], 7: [12, 8, 9], 8: [13], 9: [13],
....:                   10: [12], 11: [12], 12: [13]})
sage: F = L.frattini_sublattice()
sage: F_internal = [c for c in F.cover_relations()
....:               if c in L.cover_relations()]
sage: L.plot(figsize=12, border=True, element_shape='s',
....:        element_size=400, element_color='white',
....:        element_colors={'blue': F, 'green': L.double_irreducibles()},
....:        cover_color='lightgray', cover_colors={'black': F_internal},
....:        title='The Frattini\nsublattice in blue', fontsize=10)
Graphics object consisting of 39 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> L = LatticePoset({Integer(0): [Integer(1), Integer(2), Integer(3),␣
↪Integer(4)], Integer(1): [Integer(12)], Integer(2): [Integer(6),␣
↪Integer(7)],
...                   Integer(3): [Integer(5), Integer(9)], Integer(4):␣
↪[Integer(5), Integer(6), Integer(10), Integer(11)], Integer(5):␣
↪[Integer(13)],
...                   Integer(6): [Integer(12)], Integer(7): [Integer(12),␣
↪Integer(8), Integer(9)], Integer(8): [Integer(13)], Integer(9):␣
↪[Integer(13)],
...                   Integer(10): [Integer(12)], Integer(11): [Integer(12)],␣
↪Integer(12): [Integer(13)]})
>>> F = L.frattini_sublattice()
>>> F_internal = [c for c in F.cover_relations()
...               if c in L.cover_relations()]
>>> L.plot(figsize=Integer(12), border=True, element_shape='s',
...        element_size=Integer(400), element_color='white',
```

(continues on next page)

(continued from previous page)

```
...         element_colors={'blue': F, 'green': L.double_irreducibles()},
...         cover_color='lightgray', cover_colors={'black': F_internal},
...         title='The Frattini\nsublattice in blue', fontsize=Integer(10))
Graphics object consisting of 39 graphics primitives
```

product (*other*)

Return the Cartesian product of the poset with `other`.

The Cartesian (or 'direct') product of P and Q is defined by $(p, q) \leq (p', q')$ iff $p \leq p'$ in P and $q \leq q'$ in Q.

Product of (semi)lattices are returned as a (semi)lattice.

EXAMPLES:

```
sage: P = posets.ChainPoset(3)
sage: Q = posets.ChainPoset(4)
sage: PQ = P.product(Q) ; PQ
Finite lattice containing 12 elements
sage: len(PQ.cover_relations())
17
sage: Q.product(P).is_isomorphic(PQ)
True

sage: P = posets.BooleanLattice(2)
sage: Q = P.product(P)
sage: Q.is_isomorphic(posets.BooleanLattice(4))
True
```

```
>>> from sage.all import *
>>> P = posets.ChainPoset(Integer(3))
>>> Q = posets.ChainPoset(Integer(4))
>>> PQ = P.product(Q) ; PQ
Finite lattice containing 12 elements
>>> len(PQ.cover_relations())
17
>>> Q.product(P).is_isomorphic(PQ)
True

>>> P = posets.BooleanLattice(Integer(2))
>>> Q = P.product(P)
>>> Q.is_isomorphic(posets.BooleanLattice(Integer(4)))
True
```

One can also simply use $*$:

```
sage: P = posets.ChainPoset(2)
sage: Q = posets.ChainPoset(3)
sage: P*Q
Finite lattice containing 6 elements
```

```
>>> from sage.all import *
>>> P = posets.ChainPoset(Integer(2))
>>> Q = posets.ChainPoset(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> P*Q
Finite lattice containing 6 elements
```

See also:

CartesianProductPoset, *factor()*

promotion(*i=1*)

Compute the (extended) promotion on the linear extension of the poset `self`.

INPUT:

- `i` – integer between 1 and n (default: 1)

OUTPUT: an isomorphic poset, with the same default linear extension

The extended promotion is defined on a poset `self` of size n by applying the promotion operator $\tau_i \tau_{i+1} \cdots \tau_{n-1}$ to the default linear extension π of `self` (see *promotion()*), and relabeling `self` accordingly. For more details see [Stan2009].

When the elements of the poset `self` are labelled by $\{1, 2, \ldots, n\}$, the linear extension is the identity, and $i = 1$, the above algorithm corresponds to the promotion operator on posets defined by Schützenberger as follows. Remove 1 from `self` and replace it by the minimum j of all labels covering 1 in the poset. Then, remove j and replace it by the minimum of all labels covering j, and so on. This process ends when a label is a local maximum. Place the label $n + 1$ at this vertex. Finally, decrease all labels by 1.

EXAMPLES:

```
sage: P = Poset(([1,2], [[1,2]]), linear_extension=True, facade=False)
sage: P.promotion()
Finite poset containing 2 elements with distinguished linear extension
sage: P == P.promotion()
True

sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,
→6]]))
sage: P.list()
[1, 2, 3, 5, 6, 4, 7]
sage: Q = P.promotion(4); Q
Finite poset containing 7 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 6], [2, 3], [2, 5], [3, 7], [5, 7], [6, 4]]
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2)], [[Integer(1),Integer(2)]]), linear_
→extension=True, facade=False)
>>> P.promotion()
Finite poset containing 2 elements with distinguished linear extension
>>> P == P.promotion()
True

>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
→Integer(6),Integer(7)], [[Integer(1),Integer(2)],[Integer(1),Integer(4)],
```

(continues on next page)

(continued from previous page)

```
↪[Integer(2),Integer(3)],[Integer(2),Integer(5)],[Integer(3),Integer(6)],
↪[Integer(4),Integer(7)],[Integer(5),Integer(6)]]))
>>> P.list()
[1, 2, 3, 5, 6, 4, 7]
>>> Q = P.promotion(Integer(4)); Q
Finite poset containing 7 elements with distinguished linear extension
>>> Q.cover_relations()
[[1, 2], [1, 6], [2, 3], [2, 5], [3, 7], [5, 7], [6, 4]]
```

Note that if one wants to obtain the promotion defined by Schützenberger's algorithm directly on the poset, one needs to make sure the linear extension is the identity:

```
sage: P = P.with_linear_extension([1,2,3,4,5,6,7])
sage: P.list()
[1, 2, 3, 4, 5, 6, 7]
sage: Q = P.promotion(4); Q
Finite poset containing 7 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 6], [2, 3], [2, 4], [3, 5], [4, 5], [6, 7]]
sage: Q = P.promotion()
sage: Q.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 5], [3, 6], [4, 7], [5, 7]]
```

```
>>> from sage.all import *
>>> P = P.with_linear_extension([Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5),Integer(6),Integer(7)])
>>> P.list()
[1, 2, 3, 4, 5, 6, 7]
>>> Q = P.promotion(Integer(4)); Q
Finite poset containing 7 elements with distinguished linear extension
>>> Q.cover_relations()
[[1, 2], [1, 6], [2, 3], [2, 4], [3, 5], [4, 5], [6, 7]]
>>> Q = P.promotion()
>>> Q.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 5], [3, 6], [4, 7], [5, 7]]
```

Here is an example for a poset not labelled by $\{1, 2, \ldots, n\}$:

```
sage: P = Poset((divisors(30), attrcall("divides")), linear_extension=True)
sage: P.list()
[1, 2, 3, 5, 6, 10, 15, 30]
sage: P.cover_relations()
[[1, 2], [1, 3], [1, 5], [2, 6], [2, 10], [3, 6], [3, 15],
 [5, 10], [5, 15], [6, 30], [10, 30], [15, 30]]
sage: Q = P.promotion(4); Q
Finite poset containing 8 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 3], [1, 6], [2, 5], [2, 15], [3, 5], [3, 10],
 [5, 30], [6, 10], [6, 15], [10, 30], [15, 30]]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(30)), attrcall("divides")), linear_
```

(continues on next page)

(continued from previous page)

```
↪extension=True)
>>> P.list()
[1, 2, 3, 5, 6, 10, 15, 30]
>>> P.cover_relations()
[[1, 2], [1, 3], [1, 5], [2, 6], [2, 10], [3, 6], [3, 15],
 [5, 10], [5, 15], [6, 30], [10, 30], [15, 30]]
>>> Q = P.promotion(Integer(4)); Q
Finite poset containing 8 elements with distinguished linear extension
>>> Q.cover_relations()
[[1, 2], [1, 3], [1, 6], [2, 5], [2, 15], [3, 5], [3, 10],
 [5, 30], [6, 10], [6, 15], [10, 30], [15, 30]]
```

See also

- *linear_extension()*
- *with_linear_extension()* and the `linear_extension` option of *Poset()*
- *promotion()*
- *evacuation()*

AUTHOR:

- Anne Schilling (2012-02-18)

random_linear_extension()

Return a random linear extension of the poset.

The distribution is not uniform.

EXAMPLES:

```
sage: set_random_seed(0)  # results are reproducible
sage: P = posets.BooleanLattice(4)
sage: P.random_linear_extension()
[0, 4, 1, 2, 3, 8, 10, 5, 12, 9, 13, 11, 6, 14, 7, 15]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))  # results are reproducible
>>> P = posets.BooleanLattice(Integer(4))
>>> P.random_linear_extension()
[0, 4, 1, 2, 3, 8, 10, 5, 12, 9, 13, 11, 6, 14, 7, 15]
```

random_maximal_antichain()

Return a random maximal antichain of the poset.

The distribution is not uniform.

EXAMPLES:

```
sage: set_random_seed(0)  # results are reproducible
sage: P = posets.BooleanLattice(4)
sage: P.random_maximal_antichain()
[1, 8, 2, 4]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))  # results are reproducible
>>> P = posets.BooleanLattice(Integer(4))
>>> P.random_maximal_antichain()
[1, 8, 2, 4]
```

random_maximal_chain()

Return a random maximal chain of the poset.

The distribution is not uniform.

EXAMPLES:

```
sage: set_random_seed(0)  # results are reproducible
sage: P = posets.BooleanLattice(4)
sage: P.random_maximal_chain()
[0, 4, 5, 7, 15]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))  # results are reproducible
>>> P = posets.BooleanLattice(Integer(4))
>>> P.random_maximal_chain()
[0, 4, 5, 7, 15]
```

random_order_ideal(*direction='down'*)

Return a random order ideal with uniform probability.

INPUT:

- `direction` – `'up'`, `'down'` or `'antichain'` (default: `'down'`)

OUTPUT:

A randomly selected order ideal (or order filter if `direction='up'`, or antichain if `direction='antichain'`) where all order ideals have equal probability of occurring.

ALGORITHM:

Uses the coupling from the past algorithm described in [Propp1997].

EXAMPLES:

```
sage: P = posets.BooleanLattice(3)
sage: P.random_order_ideal()  # random
[0, 1, 2, 3, 4, 5, 6]
sage: P.random_order_ideal(direction='up')  # random
[6, 7]
sage: P.random_order_ideal(direction='antichain')  # random
[1, 2]

sage: P = posets.TamariLattice(5)
sage: a = P.random_order_ideal('antichain')
sage: P.is_antichain_of_poset(a)
True
sage: a = P.random_order_ideal('up')
sage: P.is_order_filter(a)
True
```

(continues on next page)

(continued from previous page)

```
sage: a = P.random_order_ideal('down')
sage: P.is_order_ideal(a)
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(3))
>>> P.random_order_ideal()  # random
[0, 1, 2, 3, 4, 5, 6]
>>> P.random_order_ideal(direction='up')  # random
[6, 7]
>>> P.random_order_ideal(direction='antichain')  # random
[1, 2]

>>> P = posets.TamariLattice(Integer(5))
>>> a = P.random_order_ideal('antichain')
>>> P.is_antichain_of_poset(a)
True
>>> a = P.random_order_ideal('up')
>>> P.is_order_filter(a)
True
>>> a = P.random_order_ideal('down')
>>> P.is_order_ideal(a)
True
```

random_subposet (*p*)

Return a random subposet that contains each element with probability `p`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(3)
sage: set_random_seed(0)  # Results are reproducible
sage: Q = P.random_subposet(0.5)
sage: Q.cover_relations()
[[0, 2], [0, 5], [2, 3], [3, 7], [5, 7]]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(3))
>>> set_random_seed(Integer(0))  # Results are reproducible
>>> Q = P.random_subposet(RealNumber('0.5'))
>>> Q.cover_relations()
[[0, 2], [0, 5], [2, 3], [3, 7], [5, 7]]
```

rank (*element=None*)

Return the rank of an element `element` in the poset `self`, or the rank of the poset if `element` is `None`.

(The rank of a poset is the length of the longest chain of elements of the poset. This is sometimes called the length of a poset.)

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]], facade = False)
sage: P.rank(5)
2
```

(continues on next page)

(continued from previous page)

```
sage: P.rank()
3
sage: Q = Poset([[1,2],[3],[],[]])

sage: P = posets.SymmetricGroupBruhatOrderPoset(4)
sage: [(v,P.rank(v)) for v in P]
[('1234', 0),
 ('1243', 1),
...
 ('4312', 5),
 ('4321', 6)]
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(3),Integer(2)],[Integer(4)],[Integer(4),
↪Integer(5),Integer(6)],[Integer(6)],[Integer(7)],[Integer(7)],[Integer(7)],
↪[]], facade = False)
>>> P.rank(Integer(5))
2
>>> P.rank()
3
>>> Q = Poset([[Integer(1),Integer(2)],[Integer(3)],[],[]])

>>> P = posets.SymmetricGroupBruhatOrderPoset(Integer(4))
>>> [(v,P.rank(v)) for v in P]
[('1234', 0),
 ('1243', 1),
...
 ('4312', 5),
 ('4321', 6)]
```

rank_function()

Return the (normalized) rank function of the poset, if it exists.

A *rank function* of a poset P is a function r that maps elements of P to integers and satisfies: $r(x) = r(y)+1$ if x covers y. The function r is normalized such that its minimum value on every connected component of the Hasse diagram of P is 0. This determines the function r uniquely (when it exists).

OUTPUT:

- a lambda function, if the poset admits a rank function
- `None`, if the poset does not admit a rank function

EXAMPLES:

```
sage: P = Poset(([1,2,3,4],[[1,4],[2,3],[3,4]]), facade=True)
sage: P.rank_function() is not None
True
sage: P = Poset(([1,2,3,4,5],[[1,2],[2,3],[3,4],[1,5],[5,4]]), facade=True)
sage: P.rank_function() is not None
False
sage: P = Poset(([1,2,3,4,5,6,7,8],[[1,4],[2,3],[3,4],[5,7],[6,7]]),␣
↪facade=True)
sage: f = P.rank_function(); f is not None
```

(continues on next page)

(continued from previous page)

```
True
sage: f(5)
0
sage: f(2)
0
```

```
>>> from sage.all import *
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4)],[[Integer(1),
↪Integer(4)],[Integer(2),Integer(3)],[Integer(3),Integer(4)]]), facade=True)
>>> P.rank_function() is not None
True
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],
↪[[Integer(1),Integer(2)],[Integer(2),Integer(3)],[Integer(3),Integer(4)],
↪[Integer(1),Integer(5)],[Integer(5),Integer(4)]]), facade=True)
>>> P.rank_function() is not None
False
>>> P = Poset(([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7),Integer(8)],[[Integer(1),Integer(4)],[Integer(2),
↪Integer(3)],[Integer(3),Integer(4)],[Integer(5),Integer(7)],[Integer(6),
↪Integer(7)]]), facade=True)
>>> f = P.rank_function(); f is not None
True
>>> f(Integer(5))
0
>>> f(Integer(2))
0
```

rees_product (*other*)

Return the Rees product of `self` and `other`.

This is only defined if both posets are graded.

The underlying set is the set of pairs (p, q) in the Cartesian product such that $\mathrm{rk}(p) \geq \mathrm{rk}(q)$.

This operation was defined by Björner and Welker in [BjWe2005]. Other references are [MBRe2011] and [LSW2012].

EXAMPLES:

```
sage: B3 = posets.BooleanLattice(3)
sage: B3t = B3.subposet(list(range(1,8)))
sage: C3 = posets.ChainPoset(3)
sage: D = B3t.rees_product(C3); D
Finite poset containing 12 elements
sage: sorted(D.minimal_elements())
[(1, 0), (2, 0), (4, 0)]
sage: sorted(D.maximal_elements())
[(7, 0), (7, 1), (7, 2)]
sage: D.with_bounds().moebius_function('bottom','top')
2
```

```
>>> from sage.all import *
>>> B3 = posets.BooleanLattice(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> B3t = B3.subposet(list(range(Integer(1),Integer(8))))
>>> C3 = posets.ChainPoset(Integer(3))
>>> D = B3t.rees_product(C3); D
Finite poset containing 12 elements
>>> sorted(D.minimal_elements())
[(1, 0), (2, 0), (4, 0)]
>>> sorted(D.maximal_elements())
[(7, 0), (7, 1), (7, 2)]
>>> D.with_bounds().moebius_function('bottom','top')
2
```

See also

product(), *ordinal_product()*, *star_product()*

relabel(*relabeling=None*)

Return a copy of this poset with its elements relabeled.

INPUT:

- `relabeling` – a function, dictionary, list or tuple

The given function or dictionary must map each (non-wrapped) element of `self` to some distinct object. The given list or tuple must be made of distinct objects.

When the input is a list or a tuple, the relabeling uses the total ordering of the elements of the poset given by `list(self)`.

If no relabeling is given, the poset is relabeled by integers from 0 to $n - 1$ according to one of its linear extensions. This means that $i < j$ as integers whenever $i < j$ in the relabeled poset.

EXAMPLES:

Relabeling using a function:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
sage: Q = P.relabel(lambda x: x+1)
sage: Q.list()
[2, 3, 4, 5, 7, 13]
sage: Q.cover_relations()
[[2, 3], [2, 4], [3, 5], [3, 7], [4, 7], [5, 13], [7, 13]]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")), linear_
↪extension=True)
>>> P.list()
[1, 2, 3, 4, 6, 12]
>>> P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
>>> Q = P.relabel(lambda x: x+Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> Q.list()
[2, 3, 4, 5, 7, 13]
>>> Q.cover_relations()
[[2, 3], [2, 4], [3, 5], [3, 7], [4, 7], [5, 13], [7, 13]]
```

Relabeling using a dictionary:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True,␣
↪facade=False)
sage: relabeling = {c.element:i for (i,c) in enumerate(P)}
sage: relabeling
{1: 0, 2: 1, 3: 2, 4: 3, 6: 4, 12: 5}
sage: Q = P.relabel(relabeling)
sage: Q.list()
[0, 1, 2, 3, 4, 5]
sage: Q.cover_relations()
[[0, 1], [0, 2], [1, 3], [1, 4], [2, 4], [3, 5], [4, 5]]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")), linear_
↪extension=True, facade=False)
>>> relabeling = {c.element:i for (i,c) in enumerate(P)}
>>> relabeling
{1: 0, 2: 1, 3: 2, 4: 3, 6: 4, 12: 5}
>>> Q = P.relabel(relabeling)
>>> Q.list()
[0, 1, 2, 3, 4, 5]
>>> Q.cover_relations()
[[0, 1], [0, 2], [1, 3], [1, 4], [2, 4], [3, 5], [4, 5]]
```

Mind the `c.element`; this is because the relabeling is applied to the elements of the poset without the wrapping. Thanks to this convention, the same relabeling function can be used both for facade or non facade posets.

Relabeling using a list:

```
sage: P = posets.PentagonPoset()
sage: list(P)
[0, 1, 2, 3, 4]
sage: P.cover_relations()
[[0, 1], [0, 2], [1, 4], [2, 3], [3, 4]]
sage: Q = P.relabel(list('abcde'))
sage: Q.cover_relations()
[['a', 'b'], ['a', 'c'], ['b', 'e'], ['c', 'd'], ['d', 'e']]
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> list(P)
[0, 1, 2, 3, 4]
>>> P.cover_relations()
[[0, 1], [0, 2], [1, 4], [2, 3], [3, 4]]
>>> Q = P.relabel(list('abcde'))
```

(continues on next page)

(continued from previous page)

```
>>> Q.cover_relations()
[['a', 'b'], ['a', 'c'], ['b', 'e'], ['c', 'd'], ['d', 'e']]
```

Default behaviour is increasing relabeling:

```
sage: a2 = posets.ChainPoset(2)
sage: P = a2 * a2
sage: Q = P.relabel()
sage: Q.cover_relations()
[[0, 1], [0, 2], [1, 3], [2, 3]]
```

```
>>> from sage.all import *
>>> a2 = posets.ChainPoset(Integer(2))
>>> P = a2 * a2
>>> Q = P.relabel()
>>> Q.cover_relations()
[[0, 1], [0, 2], [1, 3], [2, 3]]
```

Relabeling a (semi)lattice gives a (semi)lattice:

```
sage: P = JoinSemilattice({0: [1]})
sage: P.relabel(lambda n: n + 1)
Finite join-semilattice containing 2 elements
```

```
>>> from sage.all import *
>>> P = JoinSemilattice({Integer(0): [Integer(1)]})
>>> P.relabel(lambda n: n + Integer(1))
Finite join-semilattice containing 2 elements
```

Note

As can be seen in the above examples, the default linear extension of `Q` is that of `P` after relabeling. In particular, `P` and `Q` share the same internal Hasse diagram.

relations()

Return the list of all relations of the poset.

A relation is a pair of elements x and y such that $x \leq y$ in the poset.

The number of relations is the dimension of the incidence algebra.

OUTPUT:

A list of pairs (each pair is a list), where the first element of the pair is less than or equal to the second element.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.relations()
[[1, 1], [1, 2], [1, 3], [1, 4], [0, 0], [0, 2], [0, 3],
[0, 4], [2, 2], [2, 3], [2, 4], [3, 3], [3, 4], [4, 4]]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> P.relations()
[[1, 1], [1, 2], [1, 3], [1, 4], [0, 0], [0, 2], [0, 3],
[0, 4], [2, 2], [2, 3], [2, 4], [3, 3], [3, 4], [4, 4]]
```

See also:

relations_number(), *relations_iterator()*

AUTHOR:

- Rob Beezer (2011-05-04)

relations_iterator(*strict=False*)

Return an iterator for all the relations of the poset.

A relation is a pair of elements x and y such that $x \leq y$ in the poset.

INPUT:

- `strict` – boolean (default: `False`); if `True`, return an iterator over relations $x < y$, excluding all relations $x \leq x$

OUTPUT:

A generator that produces pairs (each pair is a list), where the first element of the pair is less than or equal to the second element.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: it = P.relations_iterator()
sage: type(it)
<class 'generator'>
sage: next(it), next(it)
([1, 1], [1, 2])

sage: P = posets.PentagonPoset()
sage: list(P.relations_iterator(strict=True))
[[0, 1], [0, 2], [0, 4], [0, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4)], Integer(4):[]})
>>> it = P.relations_iterator()
>>> type(it)
<class 'generator'>
>>> next(it), next(it)
([1, 1], [1, 2])

>>> P = posets.PentagonPoset()
>>> list(P.relations_iterator(strict=True))
[[0, 1], [0, 2], [0, 4], [0, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

See also:

relations_number(), *relations()*.

AUTHOR:

- Rob Beezer (2011-05-04)

relations_number()

Return the number of relations in the poset.

A relation is a pair of elements x and y such that $x \leq y$ in the poset.

Relations are also often called intervals. The number of intervals is the dimension of the incidence algebra.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.relations_number()
13

sage: posets.TamariLattice(4).relations_number()
68
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.relations_number()
13

>>> posets.TamariLattice(Integer(4)).relations_number()
68
```

See also:

relations_iterator(), *relations()*

show(*label_elements=True*, *element_labels=None*, *cover_labels=None*, ***kwds*)

Displays the Hasse diagram of the poset.

INPUT:

- `label_elements` – boolean (default: `True`); whether to display element labels
- `element_labels` – dictionary (default: `None`) of element labels
- `cover_labels` – dictionary, list or function representing labels of the covers of `self`; when set to `None` (default) no label is displayed on the edges of the Hasse Diagram

Note: This method also accepts:

- All options of `GenericGraph.plot`
- All options of `Graphics.show`

EXAMPLES:

```
sage: # needs sage.plot
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] })
sage: D.plot(label_elements=False)
Graphics object consisting of 6 graphics primitives
sage: D.show()
sage: elm_labs = {0:'a', 1:'b', 2:'c', 3:'d', 4:'e'}
sage: D.show(element_labels=elm_labs)
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> D = Poset({ Integer(0):[Integer(1),Integer(2)], Integer(1):[Integer(3)],␣
↪Integer(2):[Integer(3),Integer(4)] })
>>> D.plot(label_elements=False)
Graphics object consisting of 6 graphics primitives
>>> D.show()
>>> elm_labs = {Integer(0):'a', Integer(1):'b', Integer(2):'c', Integer(3):'d
↪', Integer(4):'e'}
>>> D.show(element_labels=elm_labs)
```

One more example with cover labels:

```
sage: # needs sage.plot
sage: P = posets.PentagonPoset()
sage: P.show(cover_labels=lambda a, b: a - b)
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> P = posets.PentagonPoset()
>>> P.show(cover_labels=lambda a, b: a - b)
```

slant_sum(*p*, *element*, *p_element*)

Return the slant sum poset of posets `self` and `p` by connecting them with a cover relation `(p_element, element)`.

Note

The element names of `self` and `p` must be distinct.

INPUT:

- `p` – the poset used for the slant sum
- `element` – the element of `self` that is the top of the new cover relation
- `p_element` – the element of `p` that is the bottom of the new cover relation

EXAMPLES:

```
sage: R = posets.RibbonPoset(5, [1,2])
sage: H = Poset([[5, 6, 7], [(5, 6), (6,7)]])
sage: SS = R.slant_sum(H, 3, 7)
sage: all(cr in SS.cover_relations() for cr in R.cover_relations())
```

(continues on next page)

(continued from previous page)

```
True
sage: all(cr in SS.cover_relations() for cr in H.cover_relations())
True
sage: SS.covers(7, 3)
True
```

```
>>> from sage.all import *
>>> R = posets.RibbonPoset(Integer(5), [Integer(1),Integer(2)])
>>> H = Poset([[Integer(5), Integer(6), Integer(7)], [(Integer(5),
↪Integer(6)), (Integer(6),Integer(7))]])
>>> SS = R.slant_sum(H, Integer(3), Integer(7))
>>> all(cr in SS.cover_relations() for cr in R.cover_relations())
True
>>> all(cr in SS.cover_relations() for cr in H.cover_relations())
True
>>> SS.covers(Integer(7), Integer(3))
True
```

sorted(*l*, *allow_incomparable=True*, *remove_duplicates=False*)

Return the list l sorted by the poset.

INPUT:

- `l` – list of elements of the poset
- `allow_incomparable` – boolean (default: `True`); if `True`, return incomparable elements in some order. If `False`, raise an error if `l` is not a chain of the poset.
- `remove_duplicates` – boolean (default: `False`); if `True`, remove duplicates from the output list

EXAMPLES:

```
sage: P = posets.DivisorLattice(36)
sage: P.sorted([1, 4, 1, 6, 2, 12])  # Random order for 4 and 6
[1, 1, 2, 4, 6, 12]
sage: P.sorted([1, 4, 1, 6, 2, 12], remove_duplicates=True)
[1, 2, 4, 6, 12]
sage: P.sorted([1, 4, 1, 6, 2, 12], allow_incomparable=False)
Traceback (most recent call last):
...
ValueError: the list contains incomparable elements

sage: P = Poset({7:[1, 5], 1:[2, 6], 5:[3], 6:[3, 4]})
sage: P.sorted([4, 1, 4, 5, 7])  # Random order for 1 and 5
[7, 1, 5, 4, 4]
sage: P.sorted([1, 4, 4, 7], remove_duplicates=True)
[7, 1, 4]
sage: P.sorted([4, 1, 4, 5, 7], allow_incomparable=False)
Traceback (most recent call last):
...
ValueError: the list contains incomparable elements
```

```
>>> from sage.all import *
>>> P = posets.DivisorLattice(Integer(36))
```

(continues on next page)

(continued from previous page)

```
>>> P.sorted([Integer(1), Integer(4), Integer(1), Integer(6), Integer(2),␣
↪Integer(12)])  # Random order for 4 and 6
[1, 1, 2, 4, 6, 12]
>>> P.sorted([Integer(1), Integer(4), Integer(1), Integer(6), Integer(2),␣
↪Integer(12)], remove_duplicates=True)
[1, 2, 4, 6, 12]
>>> P.sorted([Integer(1), Integer(4), Integer(1), Integer(6), Integer(2),␣
↪Integer(12)], allow_incomparable=False)
Traceback (most recent call last):
...
ValueError: the list contains incomparable elements

>>> P = Poset({Integer(7):[Integer(1), Integer(5)], Integer(1):[Integer(2),␣
↪Integer(6)], Integer(5):[Integer(3)], Integer(6):[Integer(3), Integer(4)]})
>>> P.sorted([Integer(4), Integer(1), Integer(4), Integer(5), Integer(7)])   #␣
↪Random order for 1 and 5
[7, 1, 5, 4, 4]
>>> P.sorted([Integer(1), Integer(4), Integer(4), Integer(7)], remove_
↪duplicates=True)
[7, 1, 4]
>>> P.sorted([Integer(4), Integer(1), Integer(4), Integer(5), Integer(7)],␣
↪allow_incomparable=False)
Traceback (most recent call last):
...
ValueError: the list contains incomparable elements
```

spectrum(a)

Return the a-spectrum of this poset.

The a-spectrum in a poset P is the list of integers whose i-th position contains the number of linear extensions of P that have a in the i-th location.

INPUT:

- a – an element of this poset

OUTPUT:

The a-spectrum of this poset, returned as a list.

EXAMPLES:

```
sage: P = posets.ChainPoset(5)
sage: P.spectrum(2)
[0, 0, 1, 0, 0]

sage: P = posets.BooleanLattice(3)
sage: P.spectrum(5)
[0, 0, 0, 4, 12, 16, 16, 0]

sage: P = posets.YoungDiagramPoset(Partition([3,2,1]))                        #␣
↪needs sage.combinat
sage: P.spectrum((0,1))                                                       #␣
↪needs sage.combinat
[0, 8, 6, 2, 0, 0]
```

(continues on next page)

(continued from previous page)

```
sage: P = posets.AntichainPoset(4)
sage: P.spectrum(3)
[6, 6, 6, 6]
```

```
>>> from sage.all import *
>>> P = posets.ChainPoset(Integer(5))
>>> P.spectrum(Integer(2))
[0, 0, 1, 0, 0]

>>> P = posets.BooleanLattice(Integer(3))
>>> P.spectrum(Integer(5))
[0, 0, 0, 4, 12, 16, 16, 0]

>>> P = posets.YoungDiagramPoset(Partition([Integer(3),Integer(2),
↪Integer(1)]))                          # needs sage.combinat
>>> P.spectrum((Integer(0),Integer(1)))                                    ␣
↪               # needs sage.combinat
[0, 8, 6, 2, 0, 0]

>>> P = posets.AntichainPoset(Integer(4))
>>> P.spectrum(Integer(3))
[6, 6, 6, 6]
```

star_product(*other*, *labels='pairs'*)

Return a poset isomorphic to the star product of the poset with `other`.

Both this poset and `other` are expected to be bounded and have at least two elements.

Let P be a poset with top element $\top_P$ and Q be a poset with bottom element $\bot_Q$. The star product of P and Q is the ordinal sum of $P \setminus \top_P$ and $Q \setminus \bot_Q$.

Mathematically, it is only defined when P and Q have no common elements; here we force that by giving them different names in the resulting poset.

INPUT:

- `other` – a poset
- `labels` – string (default: `'pairs'`); if set to 'pairs', each element `v` in this poset will be named `(0, v)` and each element `u` in `other` will be named `(1, u)` in the result. If set to 'integers', the elements of the result will be relabeled with consecutive integers.

EXAMPLES:

This is mostly used to combine two Eulerian posets to third one, and makes sense for graded posets only:

```
sage: B2 = posets.BooleanLattice(2)
sage: B3 = posets.BooleanLattice(3)
sage: P = B2.star_product(B3); P
Finite poset containing 10 elements
sage: P.is_eulerian()                                                     #␣
↪needs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> B2 = posets.BooleanLattice(Integer(2))
>>> B3 = posets.BooleanLattice(Integer(3))
>>> P = B2.star_product(B3); P
Finite poset containing 10 elements
>>> P.is_eulerian()                                                         #␣
↪needs sage.libs.flint
True
```

We can get elements as pairs or as integers:

```
sage: ABC = Poset({'a': ['b'], 'b': ['c']})
sage: XYZ = Poset({'x': ['y'], 'y': ['z']})
sage: ABC.star_product(XYZ).list()
[(0, 'a'), (0, 'b'), (1, 'y'), (1, 'z')]
sage: sorted(ABC.star_product(XYZ, labels='integers'))
[0, 1, 2, 3]
```

```
>>> from sage.all import *
>>> ABC = Poset({'a': ['b'], 'b': ['c']})
>>> XYZ = Poset({'x': ['y'], 'y': ['z']})
>>> ABC.star_product(XYZ).list()
[(0, 'a'), (0, 'b'), (1, 'y'), (1, 'z')]
>>> sorted(ABC.star_product(XYZ, labels='integers'))
[0, 1, 2, 3]
```

subposet (*elements*)

Return the poset containing given elements with partial order induced by this poset.

EXAMPLES:

```
sage: P = Poset({'a': ['c', 'd'], 'b': ['d','e'], 'c': ['f'],
....:            'd': ['f'], 'e': ['f']})
sage: Q = P.subposet(['a', 'b', 'f']); Q
Finite poset containing 3 elements
sage: Q.cover_relations()
[['b', 'f'], ['a', 'f']]
```

```
>>> from sage.all import *
>>> P = Poset({'a': ['c', 'd'], 'b': ['d','e'], 'c': ['f'],
...            'd': ['f'], 'e': ['f']})
>>> Q = P.subposet(['a', 'b', 'f']); Q
Finite poset containing 3 elements
>>> Q.cover_relations()
[['b', 'f'], ['a', 'f']]
```

A subposet of a non-facade poset is again a non-facade poset:

```
sage: P = posets.PentagonPoset(facade=False)
sage: Q = P.subposet([0, 1, 2, 4])
sage: Q(1) < Q(2)
False
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset(facade=False)
>>> Q = P.subposet([Integer(0), Integer(1), Integer(2), Integer(4)])
>>> Q(Integer(1)) < Q(Integer(2))
False
```

top()

Return the unique maximal element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.top() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.top()
1
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(3)],Integer(1):[Integer(3)],
↪Integer(2):[Integer(3)],Integer(3):[Integer(4),Integer(5)],Integer(4):[],
↪Integer(5):[]})
>>> P.top() is None
True
>>> Q = Poset({Integer(0):[Integer(1)],Integer(1):[]})
>>> Q.top()
1
```

See also

has_top(), *bottom()*

unwrap(*element*)

Return the element `element` of the poset `self` in unwrapped form.

INPUT:

- `element` – an element of `self`

EXAMPLES:

```
sage: P = Poset((divisors(15), attrcall("divides")), facade = False)
sage: x = P.an_element(); x
1
sage: x.parent()
Finite poset containing 4 elements
sage: P.unwrap(x)
1
sage: P.unwrap(x).parent()
Integer Ring
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(15)), attrcall("divides")), facade = False)
```

(continues on next page)

(continued from previous page)

```
>>> x = P.an_element(); x
1
>>> x.parent()
Finite poset containing 4 elements
>>> P.unwrap(x)
1
>>> P.unwrap(x).parent()
Integer Ring
```

For a non facade poset, this is equivalent to using the `.element` attribute:

```
sage: P.unwrap(x) is x.element
True
```

```
>>> from sage.all import *
>>> P.unwrap(x) is x.element
True
```

For a facade poset, this does nothing:

```
sage: P = Poset((divisors(15), attrcall("divides")), facade=True)
sage: x = P.an_element()
sage: P.unwrap(x) is x
True
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(15)), attrcall("divides")), facade=True)
>>> x = P.an_element()
>>> P.unwrap(x) is x
True
```

This method is useful in code where we do not know if `P` is a facade poset or not.

upper_covers(*x*)

Return the list of upper covers of the element `x`.

An upper cover of x is an element y such that $x < y$ and there is no element z so that $x < z < y$.

EXAMPLES:

```
sage: P = Poset([[1,5], [2,6], [3], [4], [], [6,3], [4]])
sage: P.upper_covers(1)
[2, 6]
```

```
>>> from sage.all import *
>>> P = Poset([[Integer(1),Integer(5)], [Integer(2),Integer(6)], [Integer(3)],
↪ [Integer(4)], [], [Integer(6),Integer(3)], [Integer(4)]])
>>> P.upper_covers(Integer(1))
[2, 6]
```

See also

lower_covers()

upper_covers_iterator(*x*)

Return an iterator over the upper covers of the element x.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[]})
sage: type(P.upper_covers_iterator(0))
<class 'generator'>
```

```
>>> from sage.all import *
>>> P = Poset({Integer(0):[Integer(2)], Integer(1):[Integer(2)],␣
→Integer(2):[Integer(3)], Integer(3):[]})
>>> type(P.upper_covers_iterator(Integer(0)))
<class 'generator'>
```

width(*certificate=False*)

Return the width of the poset (the size of its longest antichain).

It is computed through a matching in a bipartite graph; see Wikipedia article Dilworth%27s_theorem for more information. The width is also called Dilworth number.

INPUT:

- certificate – boolean (default: False); whether to return a certificate

OUTPUT:

- If certificate=True return (w, a), where w is the width of a poset and a is an antichain of maximum cardinality. If certificate=False return only width of the poset.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: P.width()                                                         #␣
→needs networkx
6

sage: w, max_achain = P.width(certificate=True)
sage: sorted(max_achain)
[3, 5, 6, 9, 10, 12]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> P.width()                                                         #␣
→needs networkx
6

>>> w, max_achain = P.width(certificate=True)
>>> sorted(max_achain)
[3, 5, 6, 9, 10, 12]
```

with_bounds(*labels=('bottom', 'top')*)

Return the poset with bottom and top elements adjoined.

This function adds top and bottom elements to the poset. It will always add elements, it does not check if the poset already has a bottom or a top element.

For lattices and semilattices this function returns a lattice.

INPUT:

- `labels` – a pair of elements to use as a bottom and top element of the poset. Default is strings `'bottom'` and `'top'`. Either of them can be `None`, and then a new bottom or top element will not be added.

EXAMPLES:

```
sage: V = Poset({0: [1, 2]})
sage: trafficsign = V.with_bounds(); trafficsign
Finite poset containing 5 elements
sage: trafficsign.list()
['bottom', 0, 1, 2, 'top']
sage: trafficsign = V.with_bounds(labels=(-1, -2))
sage: trafficsign.cover_relations()
[[-1, 0], [0, 1], [0, 2], [1, -2], [2, -2]]

sage: Y = V.with_bounds(labels=(-1, None))
sage: Y.cover_relations()
[[-1, 0], [0, 1], [0, 2]]

sage: P = posets.PentagonPoset()   # A lattice
sage: P.with_bounds()
Finite lattice containing 7 elements

sage: P = posets.PentagonPoset(facade=False)
sage: P.with_bounds()
Finite lattice containing 7 elements
```

```
>>> from sage.all import *
>>> V = Poset({Integer(0): [Integer(1), Integer(2)]})
>>> trafficsign = V.with_bounds(); trafficsign
Finite poset containing 5 elements
>>> trafficsign.list()
['bottom', 0, 1, 2, 'top']
>>> trafficsign = V.with_bounds(labels=(-Integer(1), -Integer(2)))
>>> trafficsign.cover_relations()
[[-1, 0], [0, 1], [0, 2], [1, -2], [2, -2]]

>>> Y = V.with_bounds(labels=(-Integer(1), None))
>>> Y.cover_relations()
[[-1, 0], [0, 1], [0, 2]]

>>> P = posets.PentagonPoset()   # A lattice
>>> P.with_bounds()
Finite lattice containing 7 elements

>>> P = posets.PentagonPoset(facade=False)
>>> P.with_bounds()
Finite lattice containing 7 elements
```

See also

`without_bounds()` for the reverse operation

with_linear_extension(*linear_extension*)

Return a copy of `self` with a different default linear extension.

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
sage: list(P)
[1, 2, 3, 4, 6, 12]
sage: Q = P.with_linear_extension([1,3,2,6,4,12])
sage: list(Q)
[1, 3, 2, 6, 4, 12]
sage: Q.cover_relations()
[[1, 3], [1, 2], [3, 6], [2, 6], [2, 4], [6, 12], [4, 12]]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")), linear_
↪extension=True)
>>> P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
>>> list(P)
[1, 2, 3, 4, 6, 12]
>>> Q = P.with_linear_extension([Integer(1),Integer(3),Integer(2),Integer(6),
↪Integer(4),Integer(12)])
>>> list(Q)
[1, 3, 2, 6, 4, 12]
>>> Q.cover_relations()
[[1, 3], [1, 2], [3, 6], [2, 6], [2, 4], [6, 12], [4, 12]]
```

Note

With the current implementation, this requires relabeling the internal `DiGraph` which is $O(n+m)$, where n is the number of elements and m the number of cover relations.

without_bounds()

Return the poset without its top and bottom elements.

This is useful as an input for the method *order_complex()*.

If there is either no top or no bottom element, this raises a `TypeError`.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: Q = P.without_bounds(); Q
Finite poset containing 3 elements
sage: Q.cover_relations()
[[2, 3]]

sage: P = posets.DiamondPoset(5)
sage: Q = P.without_bounds(); Q
Finite poset containing 3 elements
```

(continues on next page)

(continued from previous page)

```
sage: Q.cover_relations()
[]
```

```
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> Q = P.without_bounds(); Q
Finite poset containing 3 elements
>>> Q.cover_relations()
[[2, 3]]

>>> P = posets.DiamondPoset(Integer(5))
>>> Q = P.without_bounds(); Q
Finite poset containing 3 elements
>>> Q.cover_relations()
[]
```

See also

with_bounds() for the reverse operation

zeta_polynomial()

Return the zeta polynomial of the poset.

The zeta polynomial of a poset is the unique polynomial $Z(q)$ such that for every integer $m > 1$, $Z(m)$ is the number of weakly increasing sequences $x_1 \leq x_2 \leq \cdots \leq x_{m-1}$ of elements of the poset.

The polynomial $Z(q)$ is integral-valued, but generally does not have integer coefficients. It can be computed as

$$Z(q) = \sum_{k \geq 1} \binom{q-2}{k-1} c_k,$$

where c_k is the number of all chains of length k in the poset.

For more information, see section 3.12 of [EnumComb1].

In particular, $Z(2)$ is the number of vertices and $Z(3)$ is the number of intervals.

See also

apozeta_polynomial()

EXAMPLES:

```
sage: posets.ChainPoset(2).zeta_polynomial()
q
sage: posets.ChainPoset(3).zeta_polynomial()
1/2*q^2 + 1/2*q

sage: P = posets.PentagonPoset()
sage: P.zeta_polynomial()
1/6*q^3 + q^2 - 1/6*q
```

(continues on next page)

(continued from previous page)

```
sage: P = posets.DiamondPoset(5)
sage: P.zeta_polynomial()
3/2*q^2 - 1/2*q
```

```
>>> from sage.all import *
>>> posets.ChainPoset(Integer(2)).zeta_polynomial()
q
>>> posets.ChainPoset(Integer(3)).zeta_polynomial()
1/2*q^2 + 1/2*q

>>> P = posets.PentagonPoset()
>>> P.zeta_polynomial()
1/6*q^3 + q^2 - 1/6*q

>>> P = posets.DiamondPoset(Integer(5))
>>> P.zeta_polynomial()
3/2*q^2 - 1/2*q
```

class sage.combinat.posets.posets.**FinitePosets_n**(*n*)

Bases: UniqueRepresentation, Parent

The finite enumerated set of all posets on n elements, up to an isomorphism.

EXAMPLES:

```
sage: P = Posets(3)
sage: P.cardinality()
5
sage: for p in P: print(p.cover_relations())
[[1, 2], [0, 2]]
[[0, 1], [0, 2]]
[[0, 2]]
[[0, 1], [1, 2]]
[]
```

```
>>> from sage.all import *
>>> P = Posets(Integer(3))
>>> P.cardinality()
5
>>> for p in P: print(p.cover_relations())
[[1, 2], [0, 2]]
[[0, 1], [0, 2]]
[[0, 2]]
[[0, 1], [1, 2]]
[]
```

cardinality(*from_iterator=False*)

Return the cardinality of this object.

Note

By default, this returns pre-computed values obtained from the On-Line Encyclopedia of Integer Sequences (OEIS sequence A000112). To override this, pass the argument `from_iterator=True`.

EXAMPLES:

```
sage: P = Posets(3)
sage: P.cardinality()
5
sage: P.cardinality(from_iterator=True)
5
```

```
>>> from sage.all import *
>>> P = Posets(Integer(3))
>>> P.cardinality()
5
>>> P.cardinality(from_iterator=True)
5
```

`sage.combinat.posets.posets.Poset`(*data=None*, *element_labels=None*, *cover_relations=False*, *linear_extension=False*, *category=None*, *facade=None*, *key=None*)

Construct a finite poset from various forms of input data.

INPUT:

- `data` – different input are accepted by this constructor:
  1. A two-element list or tuple `(E, R)`, where `E` is a collection of elements of the poset and `R` is a collection of relations `x <= y`, each represented as a two-element list/tuple/iterable such as `[x, y]`. The poset is then the transitive closure of the provided relations. If `cover_relations=True`, then `R` is assumed to contain exactly the cover relations of the poset. If `E` is empty, then `E` is taken to be the set of elements appearing in the relations `R`.
  2. A two-element list or tuple `(E, f)`, where `E` is the set of elements of the poset and `f` is a function such that, for any pair `x, y` of elements of `E`, `f(x, y)` returns whether `x <= y`. If `cover_relations=True`, then `f(x, y)` should instead return whether `x` is covered by `y`.
  3. A dictionary of upper covers: `data[x]` is a list of the elements that cover the element x in the poset.
  4. A list or tuple of upper covers: `data[x]` is a list of the elements that cover the element x in the poset.

     If the set of elements is not a set of consecutive integers starting from zero, then:

     – every element must appear in the data, for example in its own entry.

     – `data` must be ordered in the same way as sorted elements.

     Warning

     If data is a list or tuple of length 2, then it is handled by the case 2 above.

  5. An acyclic, loop-free and multi-edge free `DiGraph`. If `cover_relations` is `True`, then the edges of the digraph are assumed to correspond to the cover relations of the poset. Otherwise, the cover relations are computed.
  6. A previously constructed poset (the poset itself is returned).
- `element_labels` – (default: `None`) an optional list or dictionary of objects that label the poset elements

- `cover_relations` – boolean (default: `False`); whether the data can be assumed to describe a directed acyclic graph whose arrows are cover relations. Otherwise, the cover relations are first computed.
- `linear_extension` – boolean (default: `False`); whether to use the provided list of elements as default linear extension for the poset. Otherwise a linear extension is computed. If the data is given as the pair `(E, f)`, then `E` is taken to be the linear extension.
- `facade` – boolean or `None` (default); whether the `Poset()`'s elements should be wrapped to make them aware of the Poset they belong to.
  - If `facade = True`, the `Poset()`'s elements are exactly those given as input.
  - If `facade = False`, the `Poset()`'s elements will become `PosetElement` objects.
  - If `facade = None` (default) the expected behaviour is the behaviour of `facade = True`, unless the opposite can be deduced from the context (i.e. for instance if a `Poset()` is built from another `Poset()`, itself built with `facade = False`)

OUTPUT:

`FinitePoset` – an instance of the `FinitePoset` class.

If `category` is specified, then the poset is created in this category instead of `FinitePosets`.

See also

`Posets`, `Posets`, `FinitePosets`

EXAMPLES:

1. Elements and cover relations:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: Poset((elms, rels), cover_relations = True, facade = False)
Finite poset containing 7 elements
```

```
>>> from sage.all import *
>>> elms = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7)]
>>> rels = [[Integer(1),Integer(2)],[Integer(3),Integer(4)],[Integer(4),
↪Integer(5)],[Integer(2),Integer(5)]]
>>> Poset((elms, rels), cover_relations = True, facade = False)
Finite poset containing 7 elements
```

Elements and non-cover relations:

```
sage: elms = [1,2,3,4]
sage: rels = [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
sage: P = Poset( [elms,rels] ,cover_relations=False); P
Finite poset containing 4 elements
sage: P.cover_relations()
[[1, 2], [2, 3], [3, 4]]
```

```
>>> from sage.all import *
>>> elms = [Integer(1),Integer(2),Integer(3),Integer(4)]
>>> rels = [[Integer(1),Integer(2)],[Integer(1),Integer(3)],[Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(4)],[Integer(2),Integer(3)],[Integer(2),Integer(4)],[Integer(3),
↪Integer(4)]]
>>> P = Poset( [elms,rels] ,cover_relations=False); P
Finite poset containing 4 elements
>>> P.cover_relations()
[[1, 2], [2, 3], [3, 4]]
```

2. Elements and function: the standard permutations of [1, 2, 3, 4] with the Bruhat order:

```
sage: elms = Permutations(4)
sage: fcn = lambda p,q : p.bruhat_lequal(q)
sage: Poset((elms, fcn))
Finite poset containing 24 elements
```

```
>>> from sage.all import *
>>> elms = Permutations(Integer(4))
>>> fcn = lambda p,q : p.bruhat_lequal(q)
>>> Poset((elms, fcn))
Finite poset containing 24 elements
```

With a function that identifies the cover relations: the set partitions of $\{1, 2, 3\}$ ordered by refinement:

```
sage: # needs sage.combinat
sage: elms = SetPartitions(3)
sage: def fcn(A, B):
....:     if len(A) != len(B)+1:
....:         return False
....:     for a in A:
....:         if not any(set(a).issubset(b) for b in B):
....:             return False
....:     return True
sage: Poset((elms, fcn), cover_relations=True)
Finite poset containing 5 elements
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> elms = SetPartitions(Integer(3))
>>> def fcn(A, B):
...     if len(A) != len(B)+Integer(1):
...         return False
...     for a in A:
...         if not any(set(a).issubset(b) for b in B):
...             return False
...     return True
>>> Poset((elms, fcn), cover_relations=True)
Finite poset containing 5 elements
```

3. A dictionary of upper covers:

```
sage: Poset({'a':['b','c'], 'b':['d'], 'c':['d'], 'd':[]})
Finite poset containing 4 elements
```

```
>>> from sage.all import *
>>> Poset({'a':['b','c'], 'b':['d'], 'c':['d'], 'd':[]})
Finite poset containing 4 elements
```

4. A list of upper covers, with range(5) as set of vertices:

```
sage: Poset([[1,2],[4],[3],[4],[]])
Finite poset containing 5 elements
```

```
>>> from sage.all import *
>>> Poset([[Integer(1),Integer(2)],[Integer(4)],[Integer(3)],[Integer(4)],[]])
Finite poset containing 5 elements
```

A list of upper covers, with letters as vertices:

```
sage: Poset([["a","b"],["b","c"],["c"]])
Finite poset containing 3 elements
```

```
>>> from sage.all import *
>>> Poset([["a","b"],["b","c"],["c"]])
Finite poset containing 3 elements
```

A list of upper covers and a dictionary of labels:

```
sage: elm_labs = {0:"a",1:"b",2:"c",3:"d",4:"e"}
sage: P = Poset([[1,2],[4],[3],[4],[]], elm_labs, facade=False)
sage: P.list()
[a, b, c, d, e]
```

```
>>> from sage.all import *
>>> elm_labs = {Integer(0):"a",Integer(1):"b",Integer(2):"c",Integer(3):"d",
↪Integer(4):"e"}
>>> P = Poset([[Integer(1),Integer(2)],[Integer(4)],[Integer(3)],[Integer(4)],
↪[]], elm_labs, facade=False)
>>> P.list()
[a, b, c, d, e]
```

Warning

The special case where the argument data is a list or tuple of length 2 is handled by the case 2. So you cannot use this method to input a 2-element poset.

5. An acyclic DiGraph.

```
sage: dag = DiGraph({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: Poset(dag)
Finite poset containing 6 elements
```

```
>>> from sage.all import *
>>> dag = DiGraph({Integer(0):[Integer(2),Integer(3)], Integer(1):[Integer(3),
↪Integer(4)], Integer(2):[Integer(5)], Integer(3):[Integer(5)],␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(4):[Integer(5)]})
>>> Poset(dag)
Finite poset containing 6 elements
```

Any directed acyclic graph without loops or multiple edges, as long as `cover_relations=False`:

```
sage: dig = DiGraph({0:[2,3], 1:[3,4,5], 2:[5], 3:[5], 4:[5]})
sage: dig.allows_multiple_edges()
False
sage: dig.allows_loops()
False
sage: dig.transitive_reduction() == dig
False
sage: Poset(dig, cover_relations=False)
Finite poset containing 6 elements
sage: Poset(dig, cover_relations=True)
Traceback (most recent call last):
...
ValueError: Hasse diagram is not transitively reduced
```

```
>>> from sage.all import *
>>> dig = DiGraph({Integer(0):[Integer(2),Integer(3)], Integer(1):[Integer(3),
↪Integer(4),Integer(5)], Integer(2):[Integer(5)], Integer(3):[Integer(5)],␣
↪Integer(4):[Integer(5)]})
>>> dig.allows_multiple_edges()
False
>>> dig.allows_loops()
False
>>> dig.transitive_reduction() == dig
False
>>> Poset(dig, cover_relations=False)
Finite poset containing 6 elements
>>> Poset(dig, cover_relations=True)
Traceback (most recent call last):
...
ValueError: Hasse diagram is not transitively reduced
```

Default Linear extension

Every poset P obtained with `Poset` comes equipped with a default linear extension, which is also used for enumerating its elements. By default, this linear extension is computed, and has no particular significance:

```
sage: P = Poset((divisors(12), attrcall("divides")))
sage: P.list()
[1, 2, 4, 3, 6, 12]
sage: P.linear_extension()
[1, 2, 4, 3, 6, 12]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")))
>>> P.list()
[1, 2, 4, 3, 6, 12]
```

(continues on next page)

(continued from previous page)

```
>>> P.linear_extension()
[1, 2, 4, 3, 6, 12]
```

You may enforce a specific linear extension using the `linear_extension` option:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: P.linear_extension()
[1, 2, 3, 4, 6, 12]
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")), linear_extension=True)
>>> P.list()
[1, 2, 3, 4, 6, 12]
>>> P.linear_extension()
[1, 2, 3, 4, 6, 12]
```

Depending on popular request, `Poset` might eventually get modified to always use the provided list of elements as default linear extension, when it is one.

See also

FinitePoset.linear_extensions()

Facade posets

When `facade = False`, the elements of a poset are wrapped so as to make them aware that they belong to that poset:

```
sage: P = Poset(DiGraph({'d':['c','b'],'c':['a'],'b':['a']}), facade = False)
sage: d,c,b,a = list(P)
sage: a.parent() is P
True
```

```
>>> from sage.all import *
>>> P = Poset(DiGraph({'d':['c','b'],'c':['a'],'b':['a']}), facade = False)
>>> d,c,b,a = list(P)
>>> a.parent() is P
True
```

This allows for comparing elements according to P:

```
sage: c < a
True
```

```
>>> from sage.all import *
>>> c < a
True
```

However, this may have surprising effects:

```
sage: my_elements = ['a','b','c','d']
sage: any(x in my_elements for x in P)
False
```

```
>>> from sage.all import *
>>> my_elements = ['a','b','c','d']
>>> any(x in my_elements for x in P)
False
```

and can be annoying when one wants to manipulate the elements of the poset:

```
sage: a + b
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Finite poset containing 4␣
↪elements' and 'Finite poset containing 4 elements'
sage: a.element + b.element
'ab'
```

```
>>> from sage.all import *
>>> a + b
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Finite poset containing 4␣
↪elements' and 'Finite poset containing 4 elements'
>>> a.element + b.element
'ab'
```

By default, facade posets are constructed instead:

```
sage: P = Poset(DiGraph({'d':['c','b'],'c':['a'],'b':['a']}))
```

```
>>> from sage.all import *
>>> P = Poset(DiGraph({'d':['c','b'],'c':['a'],'b':['a']}))
```

In this example, the elements of the poset remain plain strings:

```
sage: d,c,b,a = list(P)
sage: type(a)
<class 'str'>
```

```
>>> from sage.all import *
>>> d,c,b,a = list(P)
>>> type(a)
<class 'str'>
```

Of course, those strings are not aware of P. So to compare two such strings, one needs to query P:

```
sage: a < b
True
sage: P.lt(a,b)
False
```

```
>>> from sage.all import *
>>> a < b
True
>>> P.lt(a,b)
False
```

which models the usual mathematical notation $a <_P b$.

Most operations seem to still work, but at this point there is no guarantee whatsoever:

```
sage: P.list()
['d', 'c', 'b', 'a']
sage: P.principal_order_ideal('a')
['d', 'c', 'b', 'a']
sage: P.principal_order_ideal('b')
['d', 'b']
sage: P.principal_order_ideal('d')
['d']
sage: TestSuite(P).run()
```

```
>>> from sage.all import *
>>> P.list()
['d', 'c', 'b', 'a']
>>> P.principal_order_ideal('a')
['d', 'c', 'b', 'a']
>>> P.principal_order_ideal('b')
['d', 'b']
>>> P.principal_order_ideal('d')
['d']
>>> TestSuite(P).run()
```

Warning

`DiGraph` is used to construct the poset, and the vertices of a `DiGraph` are converted to plain Python `int`'s if they are `Integer`'s:

```
sage: G = DiGraph({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: type(G.vertices(sort=True)[0])
<class 'int'>
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(0):[Integer(2),Integer(3)], Integer(1):[Integer(3),
↪Integer(4)], Integer(2):[Integer(5)], Integer(3):[Integer(5)],
↪Integer(4):[Integer(5)]})
>>> type(G.vertices(sort=True)[Integer(0)])
<class 'int'>
```

This is worked around by systematically converting back the vertices of a poset to `Integer`'s if they are `int`'s:

```
sage: P = Poset((divisors(15), attrcall("divides")), facade = False)
sage: type(P.an_element().element)
<class 'sage.rings.integer.Integer'>

sage: P = Poset((divisors(15), attrcall("divides")), facade=True)
sage: type(P.an_element())
<class 'sage.rings.integer.Integer'>
```

```
>>> from sage.all import *
>>> P = Poset((divisors(Integer(15)), attrcall("divides")), facade = False)
>>> type(P.an_element().element)
<class 'sage.rings.integer.Integer'>

>>> P = Poset((divisors(Integer(15)), attrcall("divides")), facade=True)
>>> type(P.an_element())
<class 'sage.rings.integer.Integer'>
```

This may be abusive:

```
sage: P = Poset((range(5), operator.le), facade = True)
sage: P.an_element().parent()
Integer Ring
```

```
>>> from sage.all import *
>>> P = Poset((range(Integer(5)), operator.le), facade = True)
>>> P.an_element().parent()
Integer Ring
```

Unique representation

As most parents, *Poset* have unique representation (see `UniqueRepresentation`). Namely if two posets are created from two equal data, then they are not only equal but actually identical:

```
sage: data1 = [[1,2],[3],[3]]
sage: data2 = [[1,2],[3],[3]]
sage: P1 = Poset(data1)
sage: P2 = Poset(data2)
sage: P1 == P2
True
sage: P1 is P2
True
```

```
>>> from sage.all import *
>>> data1 = [[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]]
>>> data2 = [[Integer(1),Integer(2)],[Integer(3)],[Integer(3)]]
>>> P1 = Poset(data1)
>>> P2 = Poset(data2)
>>> P1 == P2
True
>>> P1 is P2
True
```

In situations where this behaviour is not desired, one can use the `key` option:

```
sage: P1 = Poset(data1, key = "foo")
sage: P2 = Poset(data2, key = "bar")
sage: P1 is P2
False
sage: P1 == P2
False
```

```
>>> from sage.all import *
>>> P1 = Poset(data1, key = "foo")
>>> P2 = Poset(data2, key = "bar")
>>> P1 is P2
False
>>> P1 == P2
False
```

key can be any hashable value and is passed down to UniqueRepresentation. It is otherwise ignored by the poset constructor.

sage.combinat.posets.posets.**is_poset**(*dig*)

Return True if a directed graph is acyclic and transitively reduced, and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.posets import is_poset
sage: dig = DiGraph({0:[2, 3], 1:[3, 4, 5], 2:[5], 3:[5], 4:[5]})
sage: is_poset(dig)
False
sage: is_poset(dig.transitive_reduction())
True
```

```
>>> from sage.all import *
>>> from sage.combinat.posets.posets import is_poset
>>> dig = DiGraph({Integer(0):[Integer(2), Integer(3)], Integer(1):[Integer(3),␣
↪Integer(4), Integer(5)], Integer(2):[Integer(5)], Integer(3):[Integer(5)],␣
↪Integer(4):[Integer(5)]})
>>> is_poset(dig)
False
>>> is_poset(dig.transitive_reduction())
True
```

5.1.187 q-Analogues

sage.combinat.q_analogues.**gaussian_binomial**(*n*, *k*, *q=None*, *algorithm='auto'*)

This is an alias of *q_binomial()*.

See *q_binomial()* for the full documentation.

EXAMPLES:

```
sage: gaussian_binomial(4,2)
q^4 + q^3 + 2*q^2 + q + 1
```

```
>>> from sage.all import *
>>> gaussian_binomial(Integer(4),Integer(2))
q^4 + q^3 + 2*q^2 + q + 1
```

sage.combinat.q_analogues.**gaussian_multinomial**(*seq*, *q=None*, *binomial_algorithm='auto'*)

Return the q-multinomial coefficient.

This is also known as the Gaussian multinomial coefficient, and is defined by

$$\binom{n}{k_1, k_2, \ldots, k_m}_q = \frac{[n]_q!}{[k_1]_q![k_2]_q! \cdots [k_m]_q!}$$

where $n = k_1 + k_2 + \cdots + k_m$.

If q is unspecified, then the variable is the generator q for a univariate polynomial ring over the integers.

INPUT:

- `seq` – an iterable of the values k_1 to k_m defined above
- `q` – (default: `None`) the variable q; if `None`, then use a default variable in $\mathbf{Z}[q]$
- `binomial_algorithm` – (default: `'auto'`) the algorithm to use in `q_binomial()`; see possible values there

ALGORITHM:

We use the equivalent formula

$$\binom{k_1 + \cdots + k_m}{k_1, \ldots, k_m}_q = \prod_{i=1}^{m} \binom{\sum_{j=1}^{i} k_j}{k_i}_q.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_multinomial
sage: q_multinomial([1,2,1])
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
sage: q_multinomial([1,2,1], q=1) == multinomial([1,2,1])
True
sage: q_multinomial((3,2)) == q_binomial(5,3)
True
sage: q_multinomial([])
1
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_multinomial
>>> q_multinomial([Integer(1),Integer(2),Integer(1)])
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
>>> q_multinomial([Integer(1),Integer(2),Integer(1)], q=Integer(1)) ==␣
↪multinomial([Integer(1),Integer(2),Integer(1)])
True
>>> q_multinomial((Integer(3),Integer(2))) == q_binomial(Integer(5),Integer(3))
True
>>> q_multinomial([])
1
```

`sage.combinat.q_analogues.`**`number_of_irreducible_polynomials`**`(`*n*, *q=None*, *m=1*`)`

Return the number of monic irreducible polynomials of degree `n` in `m` variables over the finite field with `q` elements.

If `q` is not given, the result is returned as an integer-valued polynomial in $\mathbf{Q}[q]$.

INPUT:

- `n` – positive integer
- `q` – `None` (default) or a prime power

- m – positive integer (default: 1)

OUTPUT: integer or integer-valued polynomial over **Q**

EXAMPLES:

```
sage: number_of_irreducible_polynomials(8, q=2)
30
sage: number_of_irreducible_polynomials(9, q=9)
43046640
sage: number_of_irreducible_polynomials(5, q=11, m=3)
2079650567184059145647246367401741345157369643207055703168
```

```
>>> from sage.all import *
>>> number_of_irreducible_polynomials(Integer(8), q=Integer(2))
30
>>> number_of_irreducible_polynomials(Integer(9), q=Integer(9))
43046640
>>> number_of_irreducible_polynomials(Integer(5), q=Integer(11), m=Integer(3))
2079650567184059145647246367401741345157369643207055703168
```

```
sage: poly = number_of_irreducible_polynomials(12); poly
1/12*q^12 - 1/12*q^6 - 1/12*q^4 + 1/12*q^2
sage: poly(5) == number_of_irreducible_polynomials(12, q=5)
True
sage: poly = number_of_irreducible_polynomials(5, m=3); poly
q^55 + q^54 + q^53 + q^52 + q^51 + q^50 + ... + 1/5*q^5 - 1/5*q^3 - 1/5*q^2 - 1/
↪5*q
sage: poly(11) == number_of_irreducible_polynomials(5, q=11, m=3)
True
```

```
>>> from sage.all import *
>>> poly = number_of_irreducible_polynomials(Integer(12)); poly
1/12*q^12 - 1/12*q^6 - 1/12*q^4 + 1/12*q^2
>>> poly(Integer(5)) == number_of_irreducible_polynomials(Integer(12),␣
↪q=Integer(5))
True
>>> poly = number_of_irreducible_polynomials(Integer(5), m=Integer(3)); poly
q^55 + q^54 + q^53 + q^52 + q^51 + q^50 + ... + 1/5*q^5 - 1/5*q^3 - 1/5*q^2 - 1/
↪5*q
>>> poly(Integer(11)) == number_of_irreducible_polynomials(Integer(5),␣
↪q=Integer(11), m=Integer(3))
True
```

This function is *much* faster than enumerating the polynomials:

```
sage: num = number_of_irreducible_polynomials(99, q=101)
sage: num.bit_length()
653
```

```
>>> from sage.all import *
>>> num = number_of_irreducible_polynomials(Integer(99), q=Integer(101))
>>> num.bit_length()
653
```

ALGORITHM:

In the univariate case, classical formula $\frac{1}{n}\sum_{d|n}\mu(n/d)q^d$ using the Möbius function μ; see `moebius()`.

In the multivariate case, formula from [Bodin2007], independently [Alekseyev2006].

`sage.combinat.q_analogues.q_binomial(n, k, q=None, algorithm='auto')`

Return the q-binomial coefficient.

This is also known as the Gaussian binomial coefficient, and is defined by

$$\binom{n}{k}_q = \frac{(1-q^n)(1-q^{n-1})\cdots(1-q^{n-k+1})}{(1-q)(1-q^2)\cdots(1-q^k)}.$$

See Wikipedia article Gaussian_binomial_coefficient.

If q is unspecified, then the variable is the generator q for a univariate polynomial ring over the integers.

INPUT:

- `n`, `k` – the values n and k defined above
- `q` – (default: `None`) the variable q; if `None`, then use a default variable in $\mathbf{Z}[q]$
- `algorithm` – (default: `'auto'`) the algorithm to use and can be one of the following:
  - `'auto'` – automatically choose the algorithm; see the algorithm section below
  - `'naive'` – use the naive algorithm
  - `'cyclotomic'` – use cyclotomic algorithm

ALGORITHM:

The naive algorithm uses the product formula. The cyclotomic algorithm uses a product of cyclotomic polynomials (cf. [CH2006]).

When the algorithm is set to `'auto'`, we choose according to the following rules:

- If `q` is a polynomial:

  When `n` is small or `k` is small with respect to `n`, one uses the naive algorithm. When both `n` and `k` are big, one uses the cyclotomic algorithm.
- If `q` is in the symbolic ring (or a symbolic subring), one uses the cyclotomic algorithm.
- Otherwise one uses the naive algorithm, unless `q` is a root of unity, then one uses the cyclotomic algorithm.

EXAMPLES:

By default, the variable is the generator of $\mathbf{Z}[q]$:

```
sage: from sage.combinat.q_analogues import q_binomial
sage: g = q_binomial(5,1) ; g
q^4 + q^3 + q^2 + q + 1
sage: g.parent()
Univariate Polynomial Ring in q over Integer Ring
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_binomial
>>> g = q_binomial(Integer(5),Integer(1)) ; g
q^4 + q^3 + q^2 + q + 1
>>> g.parent()
Univariate Polynomial Ring in q over Integer Ring
```

For $n \geq 0$, the q-binomial coefficient vanishes unless $0 \leq k \leq n$:

```
sage: q_binomial(4,5)
0
sage: q_binomial(5,-1)
0
```

```
>>> from sage.all import *
>>> q_binomial(Integer(4),Integer(5))
0
>>> q_binomial(Integer(5),-Integer(1))
0
```

For $k \geq 0$, the q-binomial coefficient is extended as a polynomial in n:

```
sage: q_binomial(-4,1)
-q^-4 - q^-3 - q^-2 - q^-1
sage: q_binomial(-2,3)
-q^-9 - q^-8 - q^-7 - q^-6
```

```
>>> from sage.all import *
>>> q_binomial(-Integer(4),Integer(1))
-q^-4 - q^-3 - q^-2 - q^-1
>>> q_binomial(-Integer(2),Integer(3))
-q^-9 - q^-8 - q^-7 - q^-6
```

Other variables can be used, given as third parameter:

```
sage: p = ZZ['p'].gen()
sage: q_binomial(4,2,p)
p^4 + p^3 + 2*p^2 + p + 1
```

```
>>> from sage.all import *
>>> p = ZZ['p'].gen()
>>> q_binomial(Integer(4),Integer(2),p)
p^4 + p^3 + 2*p^2 + p + 1
```

The third parameter can also be arbitrary values:

```
sage: q_binomial(5,1,2) == g.subs(q=2)
True
sage: q_binomial(5,1,1)
5
sage: q_binomial(4,2,-1)
2
sage: q_binomial(4,2,3.14)
152.030056160000
sage: R = GF((5, 2), 't')
sage: t = R.gen(0)
sage: q_binomial(6, 3, t)
2*t + 3
```

```
>>> from sage.all import *
>>> q_binomial(Integer(5),Integer(1),Integer(2)) == g.subs(q=Integer(2))
True
>>> q_binomial(Integer(5),Integer(1),Integer(1))
5
>>> q_binomial(Integer(4),Integer(2),-Integer(1))
2
>>> q_binomial(Integer(4),Integer(2),RealNumber('3.14'))
152.030056160000
>>> R = GF((Integer(5), Integer(2)), 't')
>>> t = R.gen(Integer(0))
>>> q_binomial(Integer(6), Integer(3), t)
2*t + 3
```

We can also do this for more complicated objects such as matrices or symmetric functions:

```
sage: q_binomial(4,2,matrix([[2,1],[-1,3]]))
[ -6  84]
[-84  78]
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: q_binomial(4,1, s[2]+s[1])
s[] + s[1] + s[1, 1] + s[1, 1, 1] + 2*s[2] + 4*s[2, 1] + 3*s[2, 1, 1]
 + 4*s[2, 2] + 3*s[2, 2, 1] + s[2, 2, 2] + 3*s[3] + 7*s[3, 1] + 3*s[3, 1, 1]
 + 6*s[3, 2] + 2*s[3, 2, 1] + s[3, 3] + 4*s[4] + 6*s[4, 1] + s[4, 1, 1]
 + 3*s[4, 2] + 3*s[5] + 2*s[5, 1] + s[6]
```

```
>>> from sage.all import *
>>> q_binomial(Integer(4),Integer(2),matrix([[Integer(2),Integer(1)],[-Integer(1),
↪Integer(3)]]))
[ -6  84]
[-84  78]
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.schur()
>>> q_binomial(Integer(4),Integer(1), s[Integer(2)]+s[Integer(1)])
s[] + s[1] + s[1, 1] + s[1, 1, 1] + 2*s[2] + 4*s[2, 1] + 3*s[2, 1, 1]
 + 4*s[2, 2] + 3*s[2, 2, 1] + s[2, 2, 2] + 3*s[3] + 7*s[3, 1] + 3*s[3, 1, 1]
 + 6*s[3, 2] + 2*s[3, 2, 1] + s[3, 3] + 4*s[4] + 6*s[4, 1] + s[4, 1, 1]
 + 3*s[4, 2] + 3*s[5] + 2*s[5, 1] + s[6]
```

REFERENCES:

AUTHORS:

- Frédéric Chapoton, David Joyner and William Stein

sage.combinat.q_analogues.**q_catalan_number**(*n*, *q=None*, *m=1*)

Return the q-Catalan number of index n.

INPUT:

- `q` – (optional) variable
- `m` – (optional) integer; to get instead the `m`-Fuss-Catalan numbers

If q is unspecified, then it defaults to using the generator q for a univariate polynomial ring over the integers.

There are several q-Catalan numbers. This procedure returns the one which can be written using the q-binomial coefficients.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_catalan_number
sage: q_catalan_number(4)
q^12 + q^10 + q^9 + 2*q^8 + q^7 + 2*q^6 + q^5 + 2*q^4 + q^3 + q^2 + 1

sage: p = ZZ['p'].0
sage: q_catalan_number(4, p)
p^12 + p^10 + p^9 + 2*p^8 + p^7 + 2*p^6 + p^5 + 2*p^4 + p^3 + p^2 + 1

sage: q_catalan_number(3, m=2)
q^12 + q^10 + q^9 + q^8 + q^7 + 2*q^6 + q^5 + q^4 + q^3 + q^2 + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_catalan_number
>>> q_catalan_number(Integer(4))
q^12 + q^10 + q^9 + 2*q^8 + q^7 + 2*q^6 + q^5 + 2*q^4 + q^3 + q^2 + 1

>>> p = ZZ['p'].gen(0)
>>> q_catalan_number(Integer(4), p)
p^12 + p^10 + p^9 + 2*p^8 + p^7 + 2*p^6 + p^5 + 2*p^4 + p^3 + p^2 + 1

>>> q_catalan_number(Integer(3), m=Integer(2))
q^12 + q^10 + q^9 + q^8 + q^7 + 2*q^6 + q^5 + q^4 + q^3 + q^2 + 1
```

sage.combinat.q_analogues.**q_factorial**(*n*, *q=None*)

Return the q-analogue of the factorial $n!$.

This is the product

$$[1]_q[2]_q \cdots [n]_q = 1 \cdot (1+q) \cdot (1+q+q^2) \cdots (1+q+q^2+\cdots+q^{n-1}).$$

If q is unspecified, then this function defaults to using the generator q for a univariate polynomial ring over the integers.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_factorial
sage: q_factorial(3)
q^3 + 2*q^2 + 2*q + 1
sage: p = ZZ['p'].0
sage: q_factorial(3, p)
p^3 + 2*p^2 + 2*p + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_factorial
>>> q_factorial(Integer(3))
q^3 + 2*q^2 + 2*q + 1
>>> p = ZZ['p'].gen(0)
>>> q_factorial(Integer(3), p)
p^3 + 2*p^2 + 2*p + 1
```

The q-analogue of $n!$ is only defined for n a nonnegative integer (Issue #11411):

```
sage: q_factorial(-2)
Traceback (most recent call last):
...
ValueError: argument (-2) must be a nonnegative integer
```

```
>>> from sage.all import *
>>> q_factorial(-Integer(2))
Traceback (most recent call last):
...
ValueError: argument (-2) must be a nonnegative integer
```

sage.combinat.q_analogues.**q_int**(*n*, *q=None*)

Return the q-analogue of the integer n.

The q-analogue of the integer n is given by

$$[n]_q = \begin{cases} 1 + q + \cdots + q^{n-1}, & \text{if } n \geq 0, \\ -q^{-n}[-n]_q, & \text{if } n \leq 0. \end{cases}$$

Consequently, if $q = 1$ then $[n]_1 = n$ and if $q \neq 1$ then $[n]_q = (q^n - 1)/(q - 1)$.

If the argument q is not specified then it defaults to the generator q of the univariate polynomial ring over the integers.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_int
sage: q_int(3)
q^2 + q + 1
sage: q_int(-3)
-q^-3 - q^-2 - q^-1
sage: p = ZZ['p'].0
sage: q_int(3,p)
p^2 + p + 1
sage: q_int(3/2)
Traceback (most recent call last):
...
ValueError: 3/2 must be an integer
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_int
>>> q_int(Integer(3))
q^2 + q + 1
>>> q_int(-Integer(3))
-q^-3 - q^-2 - q^-1
>>> p = ZZ['p'].gen(0)
>>> q_int(Integer(3),p)
p^2 + p + 1
>>> q_int(Integer(3)/Integer(2))
Traceback (most recent call last):
...
ValueError: 3/2 must be an integer
```

sage.combinat.q_analogues.**q_jordan**(*q=None*)

Return the q-Jordan number of t.

If q is the power of a prime number, the output is the number of complete flags in $\mathbf{F}_q^N$ (where N is the size of t) stable under a linear nilpotent endomorphism f_t whose Jordan type is given by t, i.e. such that for all i:

$$\dim(\ker f_t^i) = t[0] + \cdots + t[i-1]$$

If q is unspecified, then it defaults to using the generator q for a univariate polynomial ring over the integers.

The result is cached.

INPUT:

- t – integer partition, or an argument accepted by *Partition*
- q – (default: None) the variable q; if None, then use a default variable in $\mathbf{Z}[q]$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_jordan
sage: [q_jordan(mu, 2) for mu in Partitions(5)]
[9765, 1029, 213, 93, 29, 9, 1]
sage: [q_jordan(mu, 2) for mu in Partitions(6)]
[615195, 40635, 5643, 2331, 1491, 515, 147, 87, 47, 11, 1]
sage: q_jordan([3,2,1])
16*q^4 + 24*q^3 + 14*q^2 + 5*q + 1
sage: q_jordan([2,1], x)                                                    #␣
↪needs sage.symbolic
2*x + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_jordan
>>> [q_jordan(mu, Integer(2)) for mu in Partitions(Integer(5))]
[9765, 1029, 213, 93, 29, 9, 1]
>>> [q_jordan(mu, Integer(2)) for mu in Partitions(Integer(6))]
[615195, 40635, 5643, 2331, 1491, 515, 147, 87, 47, 11, 1]
>>> q_jordan([Integer(3),Integer(2),Integer(1)])
16*q^4 + 24*q^3 + 14*q^2 + 5*q + 1
>>> q_jordan([Integer(2),Integer(1)], x)                                    ␣
↪              # needs sage.symbolic
2*x + 1
```

If the partition is trivial (i.e. has only one part), we get the q-factorial (in this case, the nilpotent endomorphism is necessarily 0):

```
sage: from sage.combinat.q_analogues import q_factorial
sage: q_jordan([5]) == q_factorial(5)
True
sage: q_jordan([11], 5) == q_factorial(11, 5)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_factorial
>>> q_jordan([Integer(5)]) == q_factorial(Integer(5))
True
```

(continues on next page)

(continued from previous page)

```
>>> q_jordan([Integer(11)], Integer(5)) == q_factorial(Integer(11), Integer(5))
True
```

AUTHOR:

- Xavier Caruso (2012-06-29)

sage.combinat.q_analogues.**q_multinomial**(*seq*, *q=None*, *binomial_algorithm='auto'*)

Return the q-multinomial coefficient.

This is also known as the Gaussian multinomial coefficient, and is defined by

$$\binom{n}{k_1, k_2, \ldots, k_m}_q = \frac{[n]_q!}{[k_1]_q![k_2]_q! \cdots [k_m]_q!}$$

where $n = k_1 + k_2 + \cdots + k_m$.

If q is unspecified, then the variable is the generator q for a univariate polynomial ring over the integers.

INPUT:

- `seq` – an iterable of the values k_1 to k_m defined above
- `q` – (default: `None`) the variable q; if `None`, then use a default variable in $\mathbf{Z}[q]$
- `binomial_algorithm` – (default: `'auto'`) the algorithm to use in `q_binomial()`; see possible values there

ALGORITHM:

We use the equivalent formula

$$\binom{k_1 + \cdots + k_m}{k_1, \ldots, k_m}_q = \prod_{i=1}^{m} \binom{\sum_{j=1}^{i} k_j}{k_i}_q.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_multinomial
sage: q_multinomial([1,2,1])
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
sage: q_multinomial([1,2,1], q=1) == multinomial([1,2,1])
True
sage: q_multinomial((3,2)) == q_binomial(5,3)
True
sage: q_multinomial([])
1
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_multinomial
>>> q_multinomial([Integer(1),Integer(2),Integer(1)])
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
>>> q_multinomial([Integer(1),Integer(2),Integer(1)], q=Integer(1)) ==␣
↪multinomial([Integer(1),Integer(2),Integer(1)])
True
>>> q_multinomial((Integer(3),Integer(2))) == q_binomial(Integer(5),Integer(3))
True
>>> q_multinomial([])
1
```

sage.combinat.q_analogues.**q_pochhammer**(*n*, *a*, *q=None*)

Return the q-Pochhammer $(a;q)_n$.

The q-Pochhammer symbol is defined by

$$(a;q)_n = \prod_{k=0}^{n-1}(1 - aq^k)$$

with $(a;q)_0 = 1$ for all a, q and $n \in \mathbf{N}$. By using the identity

$$(a;q)_n = \frac{(a;q)_\infty}{(aq^n;q)_\infty},$$

we can extend the definition to $n < 0$ by

$$(a;q)_n = \frac{1}{(aq^n;q)_{-n}} = \prod_{k=1}^{-n}\frac{1}{1 - a/q^k}.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_pochhammer
sage: q_pochhammer(3, 1/7)
6/343*q^3 - 6/49*q^2 - 6/49*q + 6/7
sage: q_pochhammer(3, 3)
-18*q^3 + 6*q^2 + 6*q - 2
sage: q_pochhammer(3, 1)
0

sage: R.<q> = ZZ[]
sage: q_pochhammer(4, q)
q^10 - q^9 - q^8 + 2*q^5 - q^2 - q + 1
sage: q_pochhammer(4, q^2)
q^14 - q^12 - q^11 - q^10 + q^8 + 2*q^7 + q^6 - q^4 - q^3 - q^2 + 1
sage: q_pochhammer(-3, q)
1/(-q^9 + q^7 + q^6 + q^5 - q^4 - q^3 - q^2 + 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_pochhammer
>>> q_pochhammer(Integer(3), Integer(1)/Integer(7))
6/343*q^3 - 6/49*q^2 - 6/49*q + 6/7
>>> q_pochhammer(Integer(3), Integer(3))
-18*q^3 + 6*q^2 + 6*q - 2
>>> q_pochhammer(Integer(3), Integer(1))
0

>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> q_pochhammer(Integer(4), q)
q^10 - q^9 - q^8 + 2*q^5 - q^2 - q + 1
>>> q_pochhammer(Integer(4), q**Integer(2))
q^14 - q^12 - q^11 - q^10 + q^8 + 2*q^7 + q^6 - q^4 - q^3 - q^2 + 1
>>> q_pochhammer(-Integer(3), q)
1/(-q^9 + q^7 + q^6 + q^5 - q^4 - q^3 - q^2 + 1)
```

REFERENCES:

- Wikipedia article Q-Pochhammer_symbol

sage.combinat.q_analogues.**q_stirling_number1**(*k*, *q=None*)

Return the (unsigned) q-Stirling number of the first kind.

This is a q-analogue of `sage.combinat.combinat.stirling_number1()` .

INPUT:

- `n`, `k` – integers with $1 \leq k \leq n$
- `q` – variable (default: q)

OUTPUT: a polynomial in the variable q

These polynomials satisfy the recurrence

$$s_{n,k} = s_{n-1,k-1} + [n-1]_q s_{n-1,k}.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_stirling_number1
sage: q_stirling_number1(4,2)
q^3 + 3*q^2 + 4*q + 3

sage: all(stirling_number1(6,k) == q_stirling_number1(6,k)(1)                #␣
↪needs sage.libs.gap
....:     for k in range(1,6))
True

sage: x = polygen(QQ['q'],'x')
sage: S = sum(q_stirling_number1(5,k)*x**k for k in range(1, 6))
sage: factor(S)                                                               #␣
↪needs sage.libs.singular
x * (x + 1) * (x + q + 1) * (x + q^2 + q + 1) * (x + q^3 + q^2 + q + 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_stirling_number1
>>> q_stirling_number1(Integer(4),Integer(2))
q^3 + 3*q^2 + 4*q + 3

>>> all(stirling_number1(Integer(6),k) == q_stirling_number1(Integer(6),
↪k)(Integer(1))                     # needs sage.libs.gap
...     for k in range(Integer(1),Integer(6)))
True

>>> x = polygen(QQ['q'],'x')
>>> S = sum(q_stirling_number1(Integer(5),k)*x**k for k in range(Integer(1),␣
↪Integer(6)))
>>> factor(S)                                                               #␣
↪needs sage.libs.singular
x * (x + 1) * (x + q + 1) * (x + q^2 + q + 1) * (x + q^3 + q^2 + q + 1)
```

REFERENCES:

- [Ca1948]
- [Ca1954]

sage.combinat.q_analogues.**q_stirling_number2**(*k*, *q=None*)

Return the (unsigned) q-Stirling number of the second kind.

This is a q-analogue of *sage.combinat.combinat.stirling_number2()*.

INPUT:

- n, k – integers with $1 \leq k \leq n$
- q – variable (default: q)

OUTPUT: a polynomial in the variable q

These polynomials satisfy the recurrence

$$S_{n,k} = q^{k-1} S_{n-1,k-1} + [k]_q s_{n-1,k}.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_stirling_number2
sage: q_stirling_number2(4,2)
q^3 + 3*q^2 + 3*q

sage: all(stirling_number2(6,k) == q_stirling_number2(6,k)(1)
....:     for k in range(7))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_stirling_number2
>>> q_stirling_number2(Integer(4),Integer(2))
q^3 + 3*q^2 + 3*q

>>> all(stirling_number2(Integer(6),k) == q_stirling_number2(Integer(6),
↪k)(Integer(1))
...     for k in range(Integer(7)))
True
```

REFERENCES:

- [Mil1978]

sage.combinat.q_analogues.**q_subgroups_of_abelian_group**(*la*, *mu*, *q=None*, *algorithm='birkhoff'*)

Return the q-number of subgroups of type mu in a finite abelian group of type la.

INPUT:

- la – type of the ambient group as a *Partition*
- mu – type of the subgroup as a *Partition*
- q – (default: None) an indeterminate or a prime number; if None, this defaults to $q \in \mathbf{Z}[q]$
- algorithm – (default: 'birkhoff') the algorithm to use can be one of the following:
  - 'birkhoff – use the Birkhoff formula from [Bu87]
  - 'delsarte' – use the formula from [Delsarte48]

OUTPUT:

The number of subgroups of type mu in a group of type la as a polynomial in q.

ALGORITHM:

Let q be a prime number and $\lambda = (\lambda_1, \ldots, \lambda_l)$ be a partition. A finite abelian q-group is of type λ if it is isomorphic to

$$\mathbf{Z}/q^{\lambda_1}\mathbf{Z} \times \cdots \times \mathbf{Z}/q^{\lambda_l}\mathbf{Z}.$$

The formula from [Bu87] works as follows: Let λ and μ be partitions. Let λ' and μ' denote the conjugate partitions to λ and μ, respectively. The number of subgroups of type μ in a group of type λ is given by

$$\prod_{i=1}^{\mu_1} q^{\mu'_{i+1}(\lambda'_i - \mu'_i)} \binom{\lambda'_i - \mu'_{i+1}}{\mu'_i - \mu'_{i+1}}_q$$

The formula from [Delsarte48] works as follows: Let λ and μ be partitions. Let $(s_1, s_2, \ldots, s_l)$ and $(r_1, r_2, \ldots, r_k)$ denote the parts of the partitions conjugate to λ and μ respectively. Let

$$\mathfrak{F}(\xi_1, \ldots, \xi_k) = \xi_1^{r_2} \xi_2^{r_3} \cdots \xi_{k-1}^{r_k} \prod_{i_1=r_2}^{r_1-1} (\xi_1 - q^{i_1}) \prod_{i_2=r_3}^{r_2-1} (\xi_2 - q^{i_2}) \cdots \prod_{i_k=0}^{r_k-1} (\xi_k - q^{-i_k}).$$

Then the number of subgroups of type μ in a group of type λ is given by

$$\frac{\mathfrak{F}(q^{s_1}, q^{s_2}, \ldots, q^{s_k})}{\mathfrak{F}(q^{r_1}, q^{r_2}, \ldots, q^{r_k})}.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_subgroups_of_abelian_group
sage: q_subgroups_of_abelian_group([1,1], [1])
q + 1
sage: q_subgroups_of_abelian_group([3,3,2,1], [2,1])
q^6 + 2*q^5 + 3*q^4 + 2*q^3 + q^2
sage: R.<t> = QQ[]
sage: q_subgroups_of_abelian_group([5,3,1], [3,1], t)
t^4 + 2*t^3 + t^2
sage: q_subgroups_of_abelian_group([5,3,1], [3,1], 3)
144
sage: q_subgroups_of_abelian_group([1,1,1], [1]) == q_subgroups_of_abelian_
↪group([1,1,1], [1,1])
True
sage: q_subgroups_of_abelian_group([5], [3])
1
sage: q_subgroups_of_abelian_group([1], [2])
0
sage: q_subgroups_of_abelian_group([2], [1,1])
0
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import q_subgroups_of_abelian_group
>>> q_subgroups_of_abelian_group([Integer(1),Integer(1)], [Integer(1)])
q + 1
>>> q_subgroups_of_abelian_group([Integer(3),Integer(3),Integer(2),Integer(1)],␣
↪[Integer(2),Integer(1)])
q^6 + 2*q^5 + 3*q^4 + 2*q^3 + q^2
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> q_subgroups_of_abelian_group([Integer(5),Integer(3),Integer(1)], [Integer(3),
↪Integer(1)], t)
```

(continues on next page)

(continued from previous page)

```
t^4 + 2*t^3 + t^2
>>> q_subgroups_of_abelian_group([Integer(5),Integer(3),Integer(1)], [Integer(3),
↪Integer(1)], Integer(3))
144
>>> q_subgroups_of_abelian_group([Integer(1),Integer(1),Integer(1)],␣
↪[Integer(1)]) == q_subgroups_of_abelian_group([Integer(1),Integer(1),
↪Integer(1)], [Integer(1),Integer(1)])
True
>>> q_subgroups_of_abelian_group([Integer(5)], [Integer(3)])
1
>>> q_subgroups_of_abelian_group([Integer(1)], [Integer(2)])
0
>>> q_subgroups_of_abelian_group([Integer(2)], [Integer(1),Integer(1)])
0
```

REFERENCES:

AUTHORS:

- Amritanshu Prasad (2013-06-07): Implemented the Delsarte algorithm
- Tomer Bauer (2013, 2018): Implemented the Birkhoff algorithm and refactoring

sage.combinat.q_analogues.**qt_catalan_number**(n)

Return the q, t-Catalan number of index n.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import qt_catalan_number
sage: qt_catalan_number(1)
1
sage: qt_catalan_number(2)
q + t
sage: qt_catalan_number(3)
q^3 + q^2*t + q*t^2 + t^3 + q*t
sage: qt_catalan_number(4)
q^6 + q^5*t + q^4*t^2 + q^3*t^3 + q^2*t^4 + q*t^5 + t^6 + q^4*t + q^3*t^2 + q^2*t^
↪3 + q*t^4 + q^3*t + q^2*t^2 + q*t^3
```

```
>>> from sage.all import *
>>> from sage.combinat.q_analogues import qt_catalan_number
>>> qt_catalan_number(Integer(1))
1
>>> qt_catalan_number(Integer(2))
q + t
>>> qt_catalan_number(Integer(3))
q^3 + q^2*t + q*t^2 + t^3 + q*t
>>> qt_catalan_number(Integer(4))
q^6 + q^5*t + q^4*t^2 + q^3*t^3 + q^2*t^4 + q*t^5 + t^6 + q^4*t + q^3*t^2 + q^2*t^
↪3 + q*t^4 + q^3*t + q^2*t^2 + q*t^3
```

The q, t-Catalan number of index n is only defined for n a nonnegative integer (Issue #11411):

```
sage: qt_catalan_number(-2)
Traceback (most recent call last):
...
ValueError: argument (-2) must be a nonnegative integer
```

```
>>> from sage.all import *
>>> qt_catalan_number(-Integer(2))
Traceback (most recent call last):
...
ValueError: argument (-2) must be a nonnegative integer
```

5.1.188 q-Bernoulli Numbers and Polynomials

sage.combinat.q_bernoulli.**q_bernoulli**(*p=None*)

Compute Carlitz's q-analogue of the Bernoulli numbers.

For every nonnegative integer m, the q-Bernoulli number β_m is a rational function of the indeterminate q whose value at $q = 1$ is the usual Bernoulli number B_m.

INPUT:

- m – nonnegative integer
- p – (default: None) an optional value for q

OUTPUT:

A rational function of the indeterminate q (if p is None)

Otherwise, the rational function is evaluated at p.

EXAMPLES:

```
sage: from sage.combinat.q_bernoulli import q_bernoulli
sage: q_bernoulli(0)
1
sage: q_bernoulli(1)
-1/(q + 1)
sage: q_bernoulli(2)
q/(q^3 + 2*q^2 + 2*q + 1)
sage: all(q_bernoulli(i)(q=1) == bernoulli(i) for i in range(12))              #␣
↪needs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> from sage.combinat.q_bernoulli import q_bernoulli
>>> q_bernoulli(Integer(0))
1
>>> q_bernoulli(Integer(1))
-1/(q + 1)
>>> q_bernoulli(Integer(2))
q/(q^3 + 2*q^2 + 2*q + 1)
>>> all(q_bernoulli(i)(q=Integer(1)) == bernoulli(i) for i in range(Integer(12)))␣
↪              # needs sage.libs.flint
True
```

One can evaluate the rational function by giving a second argument:

```
sage: x = PolynomialRing(GF(2),'x').gen()
sage: q_bernoulli(5,x)
x/(x^6 + x^5 + x + 1)
```

```
>>> from sage.all import *
>>> x = PolynomialRing(GF(Integer(2)),'x').gen()
>>> q_bernoulli(Integer(5),x)
x/(x^6 + x^5 + x + 1)
```

The function does not accept negative arguments:

```
sage: q_bernoulli(-1)
Traceback (most recent call last):
...
ValueError: the argument must be a nonnegative integer
```

```
>>> from sage.all import *
>>> q_bernoulli(-Integer(1))
Traceback (most recent call last):
...
ValueError: the argument must be a nonnegative integer
```

REFERENCES:

sage.combinat.q_bernoulli.**q_bernoulli_polynomial**()

Compute Carlitz's q-analogue of the Bernoulli polynomials.

For every nonnegative integer m, the q-Bernoulli polynomial is a polynomial in one variable x with coefficients in $\mathbf{Q}(q)$ whose value at $q = 1$ is the usual Bernoulli polynomial $B_m(x)$.

The original q-Bernoulli polynomials introduced by Carlitz were polynomials in q^y with coefficients in $\mathbf{Q}(q)$. This function returns these polynomials but expressed in the variable $x = (q^y - 1)/(q - 1)$. This allows to let $q = 1$ to recover the classical Bernoulli polynomials.

INPUT:

- m – nonnegative integer

OUTPUT: a polynomial in one variable x

EXAMPLES:

```
sage: from sage.combinat.q_bernoulli import q_bernoulli_polynomial, q_bernoulli
sage: q_bernoulli_polynomial(0)
1
sage: q_bernoulli_polynomial(1)
(2/(q + 1))*x - 1/(q + 1)
sage: x = q_bernoulli_polynomial(1).parent().gen()
sage: all(q_bernoulli_polynomial(i)(q=1) == bernoulli_polynomial(x,i)          #␣
↪needs sage.libs.flint
....:     for i in range(12))
True
sage: all(q_bernoulli_polynomial(i)(x=0) == q_bernoulli(i)
....:     for i in range(12))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.q_bernoulli import q_bernoulli_polynomial, q_bernoulli
>>> q_bernoulli_polynomial(Integer(0))
1
>>> q_bernoulli_polynomial(Integer(1))
(2/(q + 1))*x - 1/(q + 1)
>>> x = q_bernoulli_polynomial(Integer(1)).parent().gen()
>>> all(q_bernoulli_polynomial(i)(q=Integer(1)) == bernoulli_polynomial(x,i)     ␣
↪     # needs sage.libs.flint
...     for i in range(Integer(12)))
True
>>> all(q_bernoulli_polynomial(i)(x=Integer(0)) == q_bernoulli(i)
...     for i in range(Integer(12)))
True
```

The function does not accept negative arguments:

```
sage: q_bernoulli_polynomial(-1)
Traceback (most recent call last):
...
ValueError: the argument must be a nonnegative integer
```

```
>>> from sage.all import *
>>> q_bernoulli_polynomial(-Integer(1))
Traceback (most recent call last):
...
ValueError: the argument must be a nonnegative integer
```

REFERENCES: [Ca1948], [Ca1954]

5.1.189 Combinatorics quickref

Integer Sequences:

```
sage: s = oeis([1,3,19,211]); s                              # optional - internet
0: A000275: ...
sage: s[0].programs()                                        # optional - internet
[('maple', ...),
 ('mathematica', ...),
 ('pari',
  0: {a(n) = if( n<0, 0, n!^2 * 4^n * polcoeff( 1 / besselj(0, x + x * O(x^(2*n))),␣
↪2*n))}; /* _Michael Somos_, May 17 2004 */)]
```

```
>>> from sage.all import *
>>> s = oeis([Integer(1),Integer(3),Integer(19),Integer(211)]); s                 ␣
↪     # optional - internet
0: A000275: ...
>>> s[Integer(0)].programs()                                 # optional - internet
[('maple', ...),
 ('mathematica', ...),
 ('pari',
  0: {a(n) = if( n<0, 0, n!^2 * 4^n * polcoeff( 1 / besselj(0, x + x * O(x^(2*n))),␣
↪2*n))}; /* _Michael Somos_, May 17 2004 */)]
```

Combinatorial objects:

```
sage: S = Subsets([1,2,3,4]); S.list(); S.<tab>                    # not tested
sage: P = Partitions(10000); P.cardinality()                               #␣
↪needs sage.libs.flint
3616...315650422081868605887952568754066420592310556052906916435144
sage: Combinations([1,3,7]).random_element()             # random
sage: Compositions(5, max_part=3).unrank(3)
[2, 2, 1]

sage: DyckWord([1,0,1,0,1,1,0,0]).to_binary_tree()                         #␣
↪needs sage.graphs
[., [., [[., .], .]]]
sage: Permutation([3,1,4,2]).robinson_schensted()
[[[1, 2], [3, 4]], [[1, 3], [2, 4]]]
sage: StandardTableau([[1, 4], [2, 5], [3]]).schuetzenberger_involution()
[[1, 3], [2, 4], [5]]
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(1),Integer(2),Integer(3),Integer(4)]); S.list(); S.<tab>    ␣
↪                 # not tested
>>> P = Partitions(Integer(10000)); P.cardinality()                                  ␣
↪     # needs sage.libs.flint
3616...315650422081868605887952568754066420592310556052906916435144
>>> Combinations([Integer(1),Integer(3),Integer(7)]).random_element()            #␣
↪random
>>> Compositions(Integer(5), max_part=Integer(3)).unrank(Integer(3))
[2, 2, 1]

>>> DyckWord([Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0)]).to_binary_tree()                            # needs␣
↪sage.graphs
[., [., [[., .], .]]]
>>> Permutation([Integer(3),Integer(1),Integer(4),Integer(2)]).robinson_schensted()
[[[1, 2], [3, 4]], [[1, 3], [2, 4]]]
>>> StandardTableau([[Integer(1), Integer(4)], [Integer(2), Integer(5)],␣
↪[Integer(3)]]).schuetzenberger_involution()
[[1, 3], [2, 4], [5]]
```

Constructions and Species:

```
sage: for (p, s) in cartesian_product([P,S]): print((p, s)) # not tested
sage: def IV_3(n):
....:     return IntegerVectors(n, 3)
sage: DisjointUnionEnumeratedSets(Family(IV_3, NonNegativeIntegers))  # not tested
```

```
>>> from sage.all import *
>>> for (p, s) in cartesian_product([P,S]): print((p, s)) # not tested
>>> def IV_3(n):
...     return IntegerVectors(n, Integer(3))
>>> DisjointUnionEnumeratedSets(Family(IV_3, NonNegativeIntegers))  # not tested
```

Words:

```
sage: Words('abc', 4).list()
[word: aaaa, ..., word: cccc]

sage: Word('aabcacbaa').is_palindrome()
True
sage: WordMorphism('a->ab,b->a').fixed_point('a')
word: abaababaabaababaababaabaababaabaababaaba...
```

```
>>> from sage.all import *
>>> Words('abc', Integer(4)).list()
[word: aaaa, ..., word: cccc]

>>> Word('aabcacbaa').is_palindrome()
True
>>> WordMorphism('a->ab,b->a').fixed_point('a')
word: abaababaabaababaababaabaababaabaababaaba...
```

Polytopes:

```
sage: points = random_matrix(ZZ, 6, 3, x=7).rows()                          #␣
↪needs sage.modules
sage: L = LatticePolytope(points)                                           #␣
↪needs sage.geometry.polyhedron sage.modules
sage: L.npoints(); L.plot3d()                           # random            #␣
↪needs sage.geometry.polyhedron sage.modules sage.plot
```

```
>>> from sage.all import *
>>> points = random_matrix(ZZ, Integer(6), Integer(3), x=Integer(7)).rows()          ␣
↪                   # needs sage.modules
>>> L = LatticePolytope(points)                                           #␣
↪needs sage.geometry.polyhedron sage.modules
>>> L.npoints(); L.plot3d()                           # random            #␣
↪needs sage.geometry.polyhedron sage.modules sage.plot
```

Root systems, Coxeter and Weyl groups:

```
sage: WeylGroup(["B",3]).bruhat_poset()                                     #␣
↪needs sage.graphs sage.modules
Finite poset containing 48 elements
sage: RootSystem(["A",2,1]).weight_lattice().plot()         # not tested    #␣
↪needs sage.graphs sage.modules sage.plot
```

```
>>> from sage.all import *
>>> WeylGroup(["B",Integer(3)]).bruhat_poset()                                      ␣
↪     # needs sage.graphs sage.modules
Finite poset containing 48 elements
>>> RootSystem(["A",Integer(2),Integer(1)]).weight_lattice().plot()         # not␣
↪tested            # needs sage.graphs sage.modules sage.plot
```

Crystals:

```
sage: CrystalOfTableaux(["A",3], shape=[3,2]).some_flashy_feature()   # not tested
```

```
>>> from sage.all import *
>>> CrystalOfTableaux(["A",Integer(3)], shape=[Integer(3),Integer(2)]).some_flashy_
↪feature()    # not tested
```

Symmetric functions and combinatorial Hopf algebras:

```
sage: Sym = SymmetricFunctions(QQ); Sym.inject_shorthands(verbose=False)          #␣
↪needs sage.sage.modules
sage: m( ( h[2,1] * (1 + 3 * p[2,1]) ) + s[2](s[3]) )                              #␣
↪needs sage.sage.modules
3*m[1, 1, 1] + ... + 10*m[5, 1] + 4*m[6]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ); Sym.inject_shorthands(verbose=False)          #␣
↪needs sage.sage.modules
>>> m( ( h[Integer(2),Integer(1)] * (Integer(1) + Integer(3) * p[Integer(2),
↪Integer(1)]) ) + s[Integer(2)](s[Integer(3)]) )                                #␣
↪needs sage.sage.modules
3*m[1, 1, 1] + ... + 10*m[5, 1] + 4*m[6]
```

Discrete groups, Permutation groups:

```
sage: S = SymmetricGroup(4)                                                        #␣
↪needs sage.groups
sage: M = PolynomialRing(QQ, 'x0,x1,x2,x3')
sage: M.an_element() * S.an_element()                                              #␣
↪needs sage.groups
x0
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(4))                                                 ␣
↪      # needs sage.groups
>>> M = PolynomialRing(QQ, 'x0,x1,x2,x3')
>>> M.an_element() * S.an_element()                                             #␣
↪needs sage.groups
x0
```

Graph theory, posets, lattices (Graph Theory, *Posets*):

```
sage: Poset({1: [2,3], 2: [4], 3: [4]}).linear_extensions().cardinality()          #␣
↪needs sage.graphs sage.modules
2
```

```
>>> from sage.all import *
>>> Poset({Integer(1): [Integer(2),Integer(3)], Integer(2): [Integer(4)], Integer(3):␣
↪[Integer(4)]}).linear_extensions().cardinality()          # needs sage.graphs sage.
↪modules
2
```

5.1.190 Rankers

`sage.combinat.ranker.from_list(l)`

Return a ranker from the list l.

INPUT:

- `l` – list

OUTPUT: `[rank, unrank]` – functions

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = [1,2,3]
sage: r,u = ranker.from_list(l)
sage: r(1)
0
sage: r(3)
2
sage: u(2)
3
sage: u(0)
1
```

```
>>> from sage.all import *
>>> import sage.combinat.ranker as ranker
>>> l = [Integer(1),Integer(2),Integer(3)]
>>> r,u = ranker.from_list(l)
>>> r(Integer(1))
0
>>> r(Integer(3))
2
>>> u(Integer(2))
3
>>> u(Integer(0))
1
```

`sage.combinat.ranker.on_fly()`

Return a pair of enumeration functions rank / unrank.

rank assigns on the fly an integer, starting from 0, to any object passed as argument. The object should be hashable. unrank is the inverse function; it returns None for indices that have not yet been assigned.

EXAMPLES:

```
sage: [rank, unrank] = sage.combinat.ranker.on_fly()
sage: rank('a')
0
sage: rank('b')
1
sage: rank('c')
2
sage: rank('a')
0
sage: unrank(2)
```

(continues on next page)

(continued from previous page)

```
'c'
sage: unrank(3)
sage: rank('d')
3
sage: unrank(3)
'd'
```

```
>>> from sage.all import *
>>> [rank, unrank] = sage.combinat.ranker.on_fly()
>>> rank('a')
0
>>> rank('b')
1
>>> rank('c')
2
>>> rank('a')
0
>>> unrank(Integer(2))
'c'
>>> unrank(Integer(3))
>>> rank('d')
3
>>> unrank(Integer(3))
'd'
```

Todo

add tests as in combinat::rankers

`sage.combinat.ranker.`**`rank_from_list`**`(`*l*`)`

Return a rank function for the elements of `l`.

INPUT:

- `l` – a duplicate free list (or iterable) of hashable objects

OUTPUT:

- a function from the elements of `l` to `0,...,len(l)`

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = ['a', 'b', 'c']
sage: r = ranker.rank_from_list(l)
sage: r('a')
0
sage: r('c')
2
```

```
>>> from sage.all import *
>>> import sage.combinat.ranker as ranker
```

(continues on next page)

(continued from previous page)

```
>>> l = ['a', 'b', 'c']
>>> r = ranker.rank_from_list(l)
>>> r('a')
0
>>> r('c')
2
```

For non elements a `ValueError` is raised, as with the usual `index` method of lists:

```
sage: r('blah')
Traceback (most recent call last):
...
ValueError: 'blah' is not in dict
```

```
>>> from sage.all import *
>>> r('blah')
Traceback (most recent call last):
...
ValueError: 'blah' is not in dict
```

Currently, the rank function is a `CallableDict`; but this is an implementation detail:

```
sage: type(r)
<class 'sage.misc.callable_dict.CallableDict'>
sage: r
{'a': 0, 'b': 1, 'c': 2}
```

```
>>> from sage.all import *
>>> type(r)
<class 'sage.misc.callable_dict.CallableDict'>
>>> r
{'a': 0, 'b': 1, 'c': 2}
```

With the current implementation, no error is issued in case of duplicate value in `l`. Instead, the rank function returns the position of some of the duplicates:

```
sage: r = ranker.rank_from_list(['a', 'b', 'a', 'c'])
sage: r('a')
2
```

```
>>> from sage.all import *
>>> r = ranker.rank_from_list(['a', 'b', 'a', 'c'])
>>> r('a')
2
```

Constructing the rank function itself is of complexity `O(len(l))`. Then, each call to the rank function consists of an essentially constant time dictionary lookup.

`sage.combinat.ranker.`**`unrank`**`(`L, i`)`

Return the i-th element of L.

INPUT:

- `L` – list, tuple, finite enumerated set, etc.

- `i` – integer

The purpose of this utility is to give a uniform idiom to recover the i-th element of an object `L`, whether `L` is a list, tuple (or more generally a `collections.abc.Sequence`), an enumerated set, some old parent of Sage still implementing unranking in the method `__getitem__`, or an iterable (see `collections.abc.Iterable`). See Issue #15919.

EXAMPLES:

Lists, tuples, and other `sequences`:

```
sage: from sage.combinat.ranker import unrank
sage: unrank(['a','b','c'], 2)
'c'
sage: unrank(('a','b','c'), 1)
'b'
sage: unrank(range(3,13,2), 1)
5
```

```
>>> from sage.all import *
>>> from sage.combinat.ranker import unrank
>>> unrank(['a','b','c'], Integer(2))
'c'
>>> unrank(('a','b','c'), Integer(1))
'b'
>>> unrank(range(Integer(3),Integer(13),Integer(2)), Integer(1))
5
```

Enumerated sets:

```
sage: unrank(GF(7), 2)
2
sage: unrank(IntegerModRing(29), 10)
10
```

```
>>> from sage.all import *
>>> unrank(GF(Integer(7)), Integer(2))
2
>>> unrank(IntegerModRing(Integer(29)), Integer(10))
10
```

An iterable:

```
sage: unrank(NN,4)
4
```

```
>>> from sage.all import *
>>> unrank(NN,Integer(4))
4
```

An iterator:

```
sage: unrank(('a{}'.format(i) for i in range(20)), 0)
'a0'
```

(continues on next page)

(continued from previous page)

```
sage: unrank(('a{}'.format(i) for i in range(20)), 2)
'a2'
```

```
>>> from sage.all import *
>>> unrank(('a{}'.format(i) for i in range(Integer(20))), Integer(0))
'a0'
>>> unrank(('a{}'.format(i) for i in range(Integer(20))), Integer(2))
'a2'
```

Warning: When unranking an iterator, it returns the `i`-th element beyond where it is currently at:

```
sage: from sage.combinat.ranker import unrank
sage: it = iter(range(20))
sage: unrank(it, 2)
2
sage: unrank(it, 2)
5
```

```
>>> from sage.all import *
>>> from sage.combinat.ranker import unrank
>>> it = iter(range(Integer(20)))
>>> unrank(it, Integer(2))
2
>>> unrank(it, Integer(2))
5
```

sage.combinat.ranker.**unrank_from_list**(*l*)

Return an unrank function from a list.

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = [1,2,3]
sage: u = ranker.unrank_from_list(l)
sage: u(2)
3
sage: u(0)
1
```

```
>>> from sage.all import *
>>> import sage.combinat.ranker as ranker
>>> l = [Integer(1),Integer(2),Integer(3)]
>>> u = ranker.unrank_from_list(l)
>>> u(Integer(2))
3
>>> u(Integer(0))
1
```

5.1.191 Recognizable Series

Let A be an alphabet and K a semiring. Then a formal series S with coefficients in K and indices in the words A^* is called recognizable if it has a linear representation, i.e., there exists

- a nonnegative integer n

and there exist

- two vectors *left* and *right* of dimension n and
- a morphism of monoids μ from A^* to $n \times n$ matrices over K

such that the coefficient corresponding to a word $w \in A^*$ equals

$$\mathit{left}\, \mu(w)\, \mathit{right}.$$

Note

Whenever a minimization (`minimized()`) of a series needs to be computed, it is required that K is a field. In particular, minimization is called before checking if a series is nonzero.

Various

See also

k-regular sequence, `sage.rings.cfinite_sequence`, `sage.combinat.binary_recurrence_sequences`.

AUTHORS:

- Daniel Krenn (2016, 2021)

ACKNOWLEDGEMENT:

- Daniel Krenn is supported by the Austrian Science Fund (FWF): P 24644-N26.

Classes and Methods

class `sage.combinat.recognizable_series.`**`PrefixClosedSet`**`(`*words*`)`

Bases: `object`

A prefix-closed set.

Creation of this prefix-closed set is interactive iteratively.

INPUT:

- `words` – a class of words (instance of `Words`)

EXAMPLES:

```
sage: from sage.combinat.recognizable_series import PrefixClosedSet
sage: P = PrefixClosedSet(Words([0, 1], infinite=False)); P
[word: ]

sage: P = PrefixClosedSet.create_by_alphabet([0, 1]); P
[word: ]
```

```
>>> from sage.all import *
>>> from sage.combinat.recognizable_series import PrefixClosedSet
>>> P = PrefixClosedSet(Words([Integer(0), Integer(1)], infinite=False)); P
[word: ]

>>> P = PrefixClosedSet.create_by_alphabet([Integer(0), Integer(1)]); P
[word: ]
```

See `iterate_possible_additions()` for further examples.

add(*w*, *check=True*)

Add a word to this prefix-closed set.

INPUT:

- `w` – a word
- `check` – boolean (default: `True`); if set, then it is verified whether all proper prefixes of `w` are already in this prefix-closed set

OUTPUT:

Nothing, but a RuntimeError is raised if the check fails.

EXAMPLES:

```
sage: from sage.combinat.recognizable_series import PrefixClosedSet
sage: P = PrefixClosedSet.create_by_alphabet([0, 1])
sage: W = P.words
sage: P.add(W([0])); P
[word: , word: 0]
sage: P.add(W([0, 1])); P
[word: , word: 0, word: 01]
sage: P.add(W([1, 1]))
Traceback (most recent call last):
...
ValueError: cannot add as not all prefixes of 11 are included yet
```

```
>>> from sage.all import *
>>> from sage.combinat.recognizable_series import PrefixClosedSet
>>> P = PrefixClosedSet.create_by_alphabet([Integer(0), Integer(1)])
>>> W = P.words
>>> P.add(W([Integer(0)])); P
[word: , word: 0]
>>> P.add(W([Integer(0), Integer(1)])); P
[word: , word: 0, word: 01]
>>> P.add(W([Integer(1), Integer(1)]))
Traceback (most recent call last):
...
ValueError: cannot add as not all prefixes of 11 are included yet
```

classmethod create_by_alphabet(*alphabet*)

A prefix-closed set.

This is a convenience method for the creation of prefix-closed sets by specifying an alphabet.

INPUT:

- `alphabet` – finite words over this `alphabet` will used

EXAMPLES:

```
sage: from sage.combinat.recognizable_series import PrefixClosedSet
sage: P = PrefixClosedSet.create_by_alphabet([0, 1]); P
[word: ]
```

```
>>> from sage.all import *
>>> from sage.combinat.recognizable_series import PrefixClosedSet
>>> P = PrefixClosedSet.create_by_alphabet([Integer(0), Integer(1)]); P
[word: ]
```

iterate_possible_additions()

Return an iterator over all elements including possible new elements.

OUTPUT: an iterator

EXAMPLES:

```
sage: from sage.combinat.recognizable_series import PrefixClosedSet
sage: P = PrefixClosedSet.create_by_alphabet([0, 1]); P
[word: ]
sage: for n, p in enumerate(P.iterate_possible_additions()):
....:     print('{}?'.format(p))
....:     if n in (0, 2, 3, 5):
....:         P.add(p)
....:         print('...added')
0?
...added
1?
00?
...added
01?
...added
000?
001?
...added
010?
011?
0010?
0011?
sage: P.elements
[word: , word: 0, word: 00, word: 01, word: 001]
```

```
>>> from sage.all import *
>>> from sage.combinat.recognizable_series import PrefixClosedSet
>>> P = PrefixClosedSet.create_by_alphabet([Integer(0), Integer(1)]); P
[word: ]
>>> for n, p in enumerate(P.iterate_possible_additions()):
...     print('{}?'.format(p))
...     if n in (Integer(0), Integer(2), Integer(3), Integer(5)):
...         P.add(p)
...         print('...added')
```

(continues on next page)

(continued from previous page)

```
0?
...added
1?
00?
...added
01?
...added
000?
001?
...added
010?
011?
0010?
0011?
>>> P.elements
[word: , word: 0, word: 00, word: 01, word: 001]
```

Calling the iterator once more, returns all elements:

```
sage: list(P.iterate_possible_additions())
[word: 0,
 word: 1,
 word: 00,
 word: 01,
 word: 000,
 word: 001,
 word: 010,
 word: 011,
 word: 0010,
 word: 0011]
```

```
>>> from sage.all import *
>>> list(P.iterate_possible_additions())
[word: 0,
 word: 1,
 word: 00,
 word: 01,
 word: 000,
 word: 001,
 word: 010,
 word: 011,
 word: 0010,
 word: 0011]
```

The method `iterate_possible_additions()` is roughly equivalent to

```
sage: list(p + a
....:      for p in P.elements
....:      for a in P.words.iterate_by_length(1))
[word: 0,
 word: 1,
 word: 00,
```

(continues on next page)

(continued from previous page)

```
 word: 01,
 word: 000,
 word: 001,
 word: 010,
 word: 011,
 word: 0010,
 word: 0011]
```

```
>>> from sage.all import *
>>> list(p + a
...      for p in P.elements
...      for a in P.words.iterate_by_length(Integer(1)))
[word: 0,
 word: 1,
 word: 00,
 word: 01,
 word: 000,
 word: 001,
 word: 010,
 word: 011,
 word: 0010,
 word: 0011]
```

However, the above does not allow to add elements during iteration, whereas `iterate_possible_additions()` does.

prefix_set()

Return the set of minimal (with respect to prefix ordering) elements of the complement of this prefix closed set.

See also Proposition 2.3.1 of [BR2010a].

OUTPUT: list

EXAMPLES:

```
sage: from sage.combinat.recognizable_series import PrefixClosedSet
sage: P = PrefixClosedSet.create_by_alphabet([0, 1]); P
[word: ]
sage: for n, p in enumerate(P.iterate_possible_additions()):
....:     if n in (0, 1, 2, 4, 6):
....:         P.add(p)
sage: P
[word: , word: 0, word: 1, word: 00, word: 10, word: 000]
sage: P.prefix_set()
[word: 01, word: 11, word: 001, word: 100,
 word: 101, word: 0000, word: 0001]
```

```
>>> from sage.all import *
>>> from sage.combinat.recognizable_series import PrefixClosedSet
>>> P = PrefixClosedSet.create_by_alphabet([Integer(0), Integer(1)]); P
[word: ]
>>> for n, p in enumerate(P.iterate_possible_additions()):
```

(continues on next page)

(continued from previous page)

```
...     if n in (Integer(0), Integer(1), Integer(2), Integer(4), Integer(6)):
...         P.add(p)
>>> P
[word: , word: 0, word: 1, word: 00, word: 10, word: 000]
>>> P.prefix_set()
[word: 01, word: 11, word: 001, word: 100,
 word: 101, word: 0000, word: 0001]
```

class sage.combinat.recognizable_series.**RecognizableSeries**(*parent*, *mu*, *left*, *right*)

Bases: ModuleElement

A recognizable series.

- parent – an instance of *RecognizableSeriesSpace*
- mu – a family of square matrices, all of which have the same dimension. The indices of this family are the elements of the alphabet. mu may be a list or tuple of the same cardinality as the alphabet as well. See also *mu*.
- left – a vector. When evaluating a coefficient, this vector is multiplied from the left to the matrix obtained from *mu* applying on a word. See also *left*.
- right – a vector. When evaluating a coefficient, this vector is multiplied from the right to the matrix obtained from *mu* applying on a word. See also *right*.

When created via the parent *RecognizableSeriesSpace*, then the following option is available.

EXAMPLES:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: S = Rec((Matrix([[3, 6], [0, 1]]), Matrix([[0, -6], [1, 5]])),
....:         vector([0, 1]), vector([1, 0])).transposed(); S
[1] + 3*[01] + [10] + 5*[11] + 9*[001] + 3*[010] + ...
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> S = Rec((Matrix([[Integer(3), Integer(6)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), -Integer(6)], [Integer(1), Integer(5)]])),
...         vector([Integer(0), Integer(1)]), vector([Integer(1), Integer(0)])).
↪transposed(); S
[1] + 3*[01] + [10] + 5*[11] + 9*[001] + 3*[010] + ...
```

We can access coefficients by

```
sage: W = Rec.indices()
sage: S[W([0, 0, 1])]
9
```

```
>>> from sage.all import *
>>> W = Rec.indices()
>>> S[W([Integer(0), Integer(0), Integer(1)])]
9
```

See also

recognizable series, `RecognizableSeriesSpace`.

coefficient_of_word(*w*, *multiply_left=True*, *multiply_right=True*)

Return the coefficient to word w of this series.

INPUT:

- `w` – a word over the parent's `alphabet()`
- `multiply_left` – boolean (default: `True`); if `False`, then multiplication by `left` is skipped
- `multiply_right` – boolean (default: `True`); if `False`, then multiplication by `right` is skipped

OUTPUT:

An element in the parent's `coefficient_ring()`

EXAMPLES:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: W = Rec.indices()
sage: S = Rec((Matrix([[1, 0], [0, 1]]), Matrix([[0, -1], [1, 2]])),
....:          left=vector([0, 1]), right=vector([1, 0]))
sage: S[W(7.digits(2))]  # indirect doctest
3
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> W = Rec.indices()
>>> S = Rec((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), -Integer(1)], [Integer(1), Integer(2)]])),
...          left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),␣
↪Integer(0)]))
>>> S[W(Integer(7).digits(Integer(2)))]  # indirect doctest
3
```

dimension()

Return the dimension of this recognizable series.

EXAMPLES:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: Rec((Matrix([[1, 0], [0, 1]]), Matrix([[1, 0], [0, 1]])),
....:     left=vector([0, 1]), right=vector([1, 0])).dimension()
2
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> Rec((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])),
...     left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),␣
↪Integer(0)])).dimension()
2
```

hadamard_product(**args*, ***kwds*)

Return the Hadamard product of this recognizable series and the `other` recognizable series, i.e., multiply the two series coefficient-wise.

INPUT:

- `other` – a *RecognizableSeries* with the same parent as this recognizable series
- `minimize` – (default: `None`) a boolean or `None`. If `True`, then *minimized()* is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a *RecognizableSeries*

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)

sage: E = Seq2((Matrix([[0, 1], [0, 1]]), Matrix([[0, 0], [0, 1]])),
....:          vector([1, 0]), vector([1, 1]))
sage: E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...

sage: O = Seq2((Matrix([[0, 0], [0, 1]]), Matrix([[0, 1], [0, 1]])),
....:          vector([1, 0]), vector([0, 1]))
sage: O
2-regular sequence 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, ...

sage: C = Seq2((Matrix([[2, 0], [2, 1]]), Matrix([[0, 1], [-2, 3]])),
....:          vector([1, 0]), vector([0, 1]))
sage: C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)

>>> E = Seq2((Matrix([[Integer(0), Integer(1)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(1)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(1),␣
↪Integer(1)]))
>>> E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...

>>> O = Seq2((Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(1)], [Integer(0), Integer(1)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(0),␣
↪Integer(1)]))
>>> O
2-regular sequence 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, ...

>>> C = Seq2((Matrix([[Integer(2), Integer(0)], [Integer(2), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(1)], [-Integer(2), Integer(3)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(0),␣
↪Integer(1)]))
>>> C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

```
sage: CE = C.hadamard_product(E)
sage: CE
2-regular sequence 0, 0, 2, 0, 4, 0, 6, 0, 8, 0, ...
sage: CE.linear_representation()
((1, 0, 0),
 Finite family {0: [0 1 0]
                   [0 2 0]
                   [0 2 1],
                1: [ 0  0  0]
                   [ 0  0  1]
                   [ 0 -2  3]},
 (0, 0, 2))

sage: Z = E.hadamard_product(O)
sage: Z
2-regular sequence 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
sage: Z.linear_representation()
((),
 Finite family {0: [],
                1: []},
 ())
```

```
>>> from sage.all import *
>>> CE = C.hadamard_product(E)
>>> CE
2-regular sequence 0, 0, 2, 0, 4, 0, 6, 0, 8, 0, ...
>>> CE.linear_representation()
((1, 0, 0),
 Finite family {0: [0 1 0]
                   [0 2 0]
                   [0 2 1],
                1: [ 0  0  0]
                   [ 0  0  1]
                   [ 0 -2  3]},
 (0, 0, 2))

>>> Z = E.hadamard_product(O)
>>> Z
2-regular sequence 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
>>> Z.linear_representation()
((),
 Finite family {0: [],
                1: []},
 ())
```

is_trivial_zero()

Return whether this recognizable series is trivially equal to zero (without any *minimization*).

EXAMPLES:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: Rec((Matrix([[1, 0], [0, 1]]), Matrix([[1, 0], [0, 1]])),
....:     left=vector([0, 1]), right=vector([1, 0])).is_trivial_zero()
```

(continues on next page)

(continued from previous page)

```
False
sage: Rec((Matrix([[1, 0], [0, 1]]), Matrix([[1, 0], [0, 1]])),
....:     left=vector([0, 0]), right=vector([1, 0])).is_trivial_zero()
True
sage: Rec((Matrix([[1, 0], [0, 1]]), Matrix([[1, 0], [0, 1]])),
....:     left=vector([0, 1]), right=vector([0, 0])).is_trivial_zero()
True
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> Rec((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),
↪Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])),
...     left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),
↪Integer(0)])).is_trivial_zero()
False
>>> Rec((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),
↪Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])),
...     left=vector([Integer(0), Integer(0)]), right=vector([Integer(1),
↪Integer(0)])).is_trivial_zero()
True
>>> Rec((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),
↪Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])),
...     left=vector([Integer(0), Integer(1)]), right=vector([Integer(0),
↪Integer(0)])).is_trivial_zero()
True
```

The following two differ in the coefficient of the empty word:

```
sage: Rec((Matrix([[0, 0], [0, 0]]), Matrix([[0, 0], [0, 0]])),
....:     left=vector([0, 1]), right=vector([1, 0])).is_trivial_zero()
True
sage: Rec((Matrix([[0, 0], [0, 0]]), Matrix([[0, 0], [0, 0]])),
....:     left=vector([1, 1]), right=vector([1, 1])).is_trivial_zero()
False
```

```
>>> from sage.all import *
>>> Rec((Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(0)]]),
↪Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(0)]])),
...     left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),
↪Integer(0)])).is_trivial_zero()
True
>>> Rec((Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(0)]]),
↪Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(0)]])),
...     left=vector([Integer(1), Integer(1)]), right=vector([Integer(1),
↪Integer(1)])).is_trivial_zero()
False
```

property left

When evaluating a coefficient, this vector is multiplied from the left to the matrix obtained from *mu* applied on a word.

linear_representation()

Return the linear representation of this series.

OUTPUT:

A triple `(left, mu, right)` containing the vectors `left` and `right`, and the family of matrices `mu`.

EXAMPLES:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: Rec((Matrix([[3, 6], [0, 1]]), Matrix([[0, -6], [1, 5]])),
....:      vector([0, 1]), vector([1, 0])
....:     ).transposed().linear_representation()
((1, 0),
 Finite family {0: [3 0]
                   [6 1],
                1: [ 0  1]
                   [-6  5]},
 (0, 1))
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> Rec((Matrix([[Integer(3), Integer(6)], [Integer(0), Integer(1)]]),
↪Matrix([[Integer(0), -Integer(6)], [Integer(1), Integer(5)]])),
...      vector([Integer(0), Integer(1)]), vector([Integer(1), Integer(0)])
...     ).transposed().linear_representation()
((1, 0),
 Finite family {0: [3 0]
                   [6 1],
                1: [ 0  1]
                   [-6  5]},
 (0, 1))
```

minimized()

Return a recognizable series equivalent to this series, but with a minimized linear representation.

The coefficients of the involved matrices need be in a field. If this is not the case, then the coefficients are automatically coerced to their fraction field.

OUTPUT: a `RecognizableSeries`

ALGORITHM:

This method implements the minimization algorithm presented in Chapter 2 of [BR2010a].

> **Note**
>
> Due to the algorithm, the left vector of the result is always $(1, 0, \ldots, 0)$, i.e., the first vector of the standard basis.

EXAMPLES:

```
sage: from itertools import islice
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])

sage: S = Rec((Matrix([[3, 6], [0, 1]]), Matrix([[0, -6], [1, 5]])),
....:          vector([0, 1]), vector([1, 0])).transposed()
sage: S
```

(continues on next page)

(continued from previous page)

```
[1] + 3*[01] + [10] + 5*[11] + 9*[001] + 3*[010]
    + 15*[011] + [100] + 11*[101] + 5*[110] + ...
sage: M = S.minimized()
sage: M.mu[0], M.mu[1], M.left, M.right
(
[3 0]  [ 0  1]
[6 1], [-6  5], (1, 0), (0, 1)
)
sage: M.left == vector([1, 0])
True
sage: all(c == d and v == w
....:     for (c, v), (d, w) in islice(zip(iter(S), iter(M)), 20))
True

sage: S = Rec((Matrix([[2, 0], [1, 1]]), Matrix([[2, 0], [2, 1]])),
....:         vector([1, 0]), vector([1, 1]))
sage: S
[] + 2*[0] + 2*[1] + 4*[00] + 4*[01] + 4*[10] + 4*[11]
   + 8*[000] + 8*[001] + 8*[010] + ...
sage: M = S.minimized()
sage: M.mu[0], M.mu[1], M.left, M.right
([2], [2], (1), (1))
sage: all(c == d and v == w
....:     for (c, v), (d, w) in islice(zip(iter(S), iter(M)), 20))
True
```

```
>>> from sage.all import *
>>> from itertools import islice
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])

>>> S = Rec((Matrix([[Integer(3), Integer(6)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), -Integer(6)], [Integer(1), Integer(5)]])),
...         vector([Integer(0), Integer(1)]), vector([Integer(1),␣
↪Integer(0)])).transposed()
>>> S
[1] + 3*[01] + [10] + 5*[11] + 9*[001] + 3*[010]
    + 15*[011] + [100] + 11*[101] + 5*[110] + ...
>>> M = S.minimized()
>>> M.mu[Integer(0)], M.mu[Integer(1)], M.left, M.right
(
[3 0]  [ 0  1]
[6 1], [-6  5], (1, 0), (0, 1)
)
>>> M.left == vector([Integer(1), Integer(0)])
True
>>> all(c == d and v == w
...     for (c, v), (d, w) in islice(zip(iter(S), iter(M)), Integer(20)))
True

>>> S = Rec((Matrix([[Integer(2), Integer(0)], [Integer(1), Integer(1)]]),␣
↪Matrix([[Integer(2), Integer(0)], [Integer(2), Integer(1)]])),
...         vector([Integer(1), Integer(0)]), vector([Integer(1),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)]))
>>> S
[] + 2*[0] + 2*[1] + 4*[00] + 4*[01] + 4*[10] + 4*[11]
   + 8*[000] + 8*[001] + 8*[010] + ...
>>> M = S.minimized()
>>> M.mu[Integer(0)], M.mu[Integer(1)], M.left, M.right
([2], [2], (1), (1))
>>> all(c == d and v == w
...     for (c, v), (d, w) in islice(zip(iter(S), iter(M)), Integer(20)))
True
```

property mu

When evaluating a coefficient, this is applied on each letter of a word; the result is a matrix. This extends *mu* to words over the parent's *alphabet()*.

property right

When evaluating a coefficient, this vector is multiplied from the right to the matrix obtained from *mu* applied on a word.

transposed()

Return the transposed series.

OUTPUT: a *RecognizableSeries*

Each of the matrices in *mu* is transposed. Additionally the vectors *left* and *right* are switched.

EXAMPLES:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: S = Rec((Matrix([[3, 6], [0, 1]]), Matrix([[0, -6], [1, 5]])),
....:          vector([0, 1]), vector([1, 0])).transposed()
sage: S
[1] + 3*[01] + [10] + 5*[11] + 9*[001] + 3*[010]
    + 15*[011] + [100] + 11*[101] + 5*[110] + ...
sage: S.mu[0], S.mu[1], S.left, S.right
(
[3 0]  [ 0  1]
[6 1], [-6  5], (1, 0), (0, 1)
)
sage: T = S.transposed()
sage: T
[1] + [01] + 3*[10] + 5*[11] + [001] + 3*[010]
    + 5*[011] + 9*[100] + 11*[101] + 15*[110] + ...
sage: T.mu[0], T.mu[1], T.left, T.right
(
[3 6]  [ 0 -6]
[0 1], [ 1  5], (0, 1), (1, 0)
)
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> S = Rec((Matrix([[Integer(3), Integer(6)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), -Integer(6)], [Integer(1), Integer(5)]])),
...         vector([Integer(0), Integer(1)]), vector([Integer(1),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)])).transposed()
>>> S
[1] + 3*[01] + [10] + 5*[11] + 9*[001] + 3*[010]
    + 15*[011] + [100] + 11*[101] + 5*[110] + ...
>>> S.mu[Integer(0)], S.mu[Integer(1)], S.left, S.right
(
[3 0]  [ 0  1]
[6 1], [-6  5], (1, 0), (0, 1)
)
>>> T = S.transposed()
>>> T
[1] + [01] + 3*[10] + 5*[11] + [001] + 3*[010]
    + 5*[011] + 9*[100] + 11*[101] + 15*[110] + ...
>>> T.mu[Integer(0)], T.mu[Integer(1)], T.left, T.right
(
[3 6]  [ 0 -6]
[0 1], [ 1  5], (0, 1), (1, 0)
)
```

class sage.combinat.recognizable_series.**RecognizableSeriesSpace**(*coefficient_ring*, *indices*, *category*, *minimize_results*)

Bases: UniqueRepresentation, Parent

The space of recognizable series on the given alphabet and with the given coefficients.

INPUT:

- `coefficient_ring` – a (semi-)ring
- `alphabet` – tuple, list or TotallyOrderedFiniteSet. If specified, then the `indices` are the finite words over this `alphabet`. `alphabet` and `indices` cannot be specified at the same time.
- `indices` – a SageMath-parent of finite words over an alphabet. `alphabet` and `indices` cannot be specified at the same time.
- `category` – (default: `None`) the category of this space

EXAMPLES:

We create a recognizable series that counts the number of ones in each word:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: Rec
Space of recognizable series on {0, 1} with coefficients in Integer Ring
sage: Rec((Matrix([[1, 0], [0, 1]]), Matrix([[1, 1], [0, 1]])),
....:     vector([1, 0]), vector([0, 1]))
[1] + [01] + [10] + 2*[11] + [001] + [010] + 2*[011] + [100] + 2*[101] + 2*[110]␣
↪+ ...
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> Rec
Space of recognizable series on {0, 1} with coefficients in Integer Ring
>>> Rec((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(1), Integer(1)], [Integer(0), Integer(1)]])),
...     vector([Integer(1), Integer(0)]), vector([Integer(0), Integer(1)]))
```

(continues on next page)

(continued from previous page)

```
[1] + [01] + [10] + 2*[11] + [001] + [010] + 2*[011] + [100] + 2*[101] + 2*[110]␣
↪+ ...
```

All of the following examples create the same space:

```
sage: Rec1 = RecognizableSeriesSpace(ZZ, [0, 1])
sage: Rec1
Space of recognizable series on {0, 1} with coefficients in Integer Ring
sage: Rec2 = RecognizableSeriesSpace(coefficient_ring=ZZ, alphabet=[0, 1])
sage: Rec2
Space of recognizable series on {0, 1} with coefficients in Integer Ring
sage: Rec3 = RecognizableSeriesSpace(ZZ, indices=Words([0, 1], infinite=False))
sage: Rec3
Space of recognizable series on {0, 1} with coefficients in Integer Ring
```

```
>>> from sage.all import *
>>> Rec1 = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> Rec1
Space of recognizable series on {0, 1} with coefficients in Integer Ring
>>> Rec2 = RecognizableSeriesSpace(coefficient_ring=ZZ, alphabet=[Integer(0),␣
↪Integer(1)])
>>> Rec2
Space of recognizable series on {0, 1} with coefficients in Integer Ring
>>> Rec3 = RecognizableSeriesSpace(ZZ, indices=Words([Integer(0), Integer(1)],␣
↪infinite=False))
>>> Rec3
Space of recognizable series on {0, 1} with coefficients in Integer Ring
```

See also

recognizable series, `RecognizableSeries`.

Element

alias of `RecognizableSeries`

alphabet()

Return the alphabet of this recognizable series space.

OUTPUT: a totally ordered set

EXAMPLES:

```
sage: RecognizableSeriesSpace(ZZ, [0, 1]).alphabet()
{0, 1}
```

```
>>> from sage.all import *
>>> RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)]).alphabet()
{0, 1}
```

coefficient_ring()

Return the coefficients of this recognizable series space.

OUTPUT:

A (semi-)ring

EXAMPLES:

```
sage: RecognizableSeriesSpace(ZZ, [0, 1]).coefficient_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)]).coefficient_ring()
Integer Ring
```

indices()

Return the indices of the recognizable series.

OUTPUT: the set of finite words over the alphabet

EXAMPLES:

```
sage: RecognizableSeriesSpace(ZZ, [0, 1]).indices()
Finite words over {0, 1}
```

```
>>> from sage.all import *
>>> RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)]).indices()
Finite words over {0, 1}
```

property minimize_results

A boolean indicating whether `RecognizableSeries.minimized()` is automatically called after performing operations.

one()

Return the one element of this `RecognizableSeriesSpace`, i.e. the embedding of the one of the coefficient ring into this `RecognizableSeriesSpace`.

EXAMPLES:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: O = Rec.one(); O
[] + ...
sage: O.linear_representation()
((1), Finite family {0: [0], 1: [0]}, (1))
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> O = Rec.one(); O
[] + ...
>>> O.linear_representation()
((1), Finite family {0: [0], 1: [0]}, (1))
```

one_hadamard()

Return the identity with respect to the `hadamard_product()`, i.e. the coefficient-wise multiplication.

OUTPUT: a `RecognizableSeries`

EXAMPLES:

```
sage: Rec = RecognizableSeriesSpace(ZZ, [0, 1])
sage: Rec.one_hadamard()
[] + [0] + [1] + [00] + [01] + [10]
   + [11] + [000] + [001] + [010] + ...
```

```
>>> from sage.all import *
>>> Rec = RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)])
>>> Rec.one_hadamard()
[] + [0] + [1] + [00] + [01] + [10]
   + [11] + [000] + [001] + [010] + ...
```

some_elements(***kwds*)

Return some elements of this recognizable series space.

See `TestSuite` for a typical use case.

INPUT:

- `kwds` are passed on to the element constructor

OUTPUT: an iterator

EXAMPLES:

```
sage: tuple(RecognizableSeriesSpace(ZZ, [0, 1]).some_elements())
([1] + [01] + [10] + 2*[11] + [001] + [010]
     + 2*[011] + [100] + 2*[101] + 2*[110] + ...,
 [] + [1] + [11] + [111] + [1111] + [11111] + [111111] + ...,
 [] + [0] + [1] + [00] + [10] + [11]
    + [000] - 1*[001] + [100] + [110] + ...,
 2*[] - 1*[1] + 2*[10] - 1*[101]
      + 2*[1010] - 1*[10101] + 2*[101010] + ...,
 [] + [1] + 6*[00] + [11] - 39*[000] + 5*[001] + 6*[100] + [111]
    + 288*[0000] - 33*[0001] + ...,
 -5*[] + ...,
 ...
 210*[] + ...,
 2210*[] - 170*[0] + 170*[1] + ...)
```

```
>>> from sage.all import *
>>> tuple(RecognizableSeriesSpace(ZZ, [Integer(0), Integer(1)]).some_
↪elements())
([1] + [01] + [10] + 2*[11] + [001] + [010]
     + 2*[011] + [100] + 2*[101] + 2*[110] + ...,
 [] + [1] + [11] + [111] + [1111] + [11111] + [111111] + ...,
 [] + [0] + [1] + [00] + [10] + [11]
    + [000] - 1*[001] + [100] + [110] + ...,
 2*[] - 1*[1] + 2*[10] - 1*[101]
      + 2*[1010] - 1*[10101] + 2*[101010] + ...,
 [] + [1] + 6*[00] + [11] - 39*[000] + 5*[001] + 6*[100] + [111]
    + 288*[0000] - 33*[0001] + ...,
 -5*[] + ...,
 ...
 210*[] + ...,
 2210*[] - 170*[0] + 170*[1] + ...)
```

`sage.combinat.recognizable_series.minimize_result(operation)`

A decorator for operations that enables control of automatic minimization on the result.

INPUT:

- `operation` – a method

OUTPUT: a method with the following additional argument:

- `minimize` – (default: `None`) a boolean or `None`. If `True`, then `minimized()` is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

Note

If the result of `operation` is `self`, then minimization is not applied unless `minimize=True` is explicitly set, in particular, independent of the parent's `minimize_results`.

5.1.192 k-regular sequences

An introduction and formal definition of k-regular sequences can be found, for example, on the Wikipedia article k-regular_sequence or in [AS2003].

```
sage: import logging
sage: logging.basicConfig()
```

```
>>> from sage.all import *
>>> import logging
>>> logging.basicConfig()
```

Examples

Binary sum of digits

The binary sum of digits $S(n)$ of a nonnegative integer n satisfies $S(2n) = S(n)$ and $S(2n+1) = S(n)+1$. We model this by the following:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: S = Seq2((Matrix([[1, 0], [0, 1]]), Matrix([[1, 0], [1, 1]])),
....:           left=vector([0, 1]), right=vector([1, 0]))
sage: S
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
sage: all(S[n] == sum(n.digits(2)) for n in srange(10))
True
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> S = Seq2((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),
↪Matrix([[Integer(1), Integer(0)], [Integer(1), Integer(1)]])),
...           left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),
↪Integer(0)]))
>>> S
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
```

(continues on next page)

(continued from previous page)

```
>>> all(S[n] == sum(n.digits(Integer(2))) for n in srange(Integer(10)))
True
```

Number of odd entries in Pascal's triangle

Let us consider the number of odd entries in the first n rows of Pascals's triangle:

```
sage: @cached_function
....: def u(n):
....:     if n <= 1:
....:         return n
....:     return 2 * u(n // 2) + u((n+1) // 2)
sage: tuple(u(n) for n in srange(10))
(0, 1, 3, 5, 9, 11, 15, 19, 27, 29)
```

```
>>> from sage.all import *
>>> @cached_function
... def u(n):
...     if n <= Integer(1):
...         return n
...     return Integer(2) * u(n // Integer(2)) + u((n+Integer(1)) // Integer(2))
>>> tuple(u(n) for n in srange(Integer(10)))
(0, 1, 3, 5, 9, 11, 15, 19, 27, 29)
```

There is a 2-regular sequence describing the numbers above as well:

```
sage: U = Seq2((Matrix([[3, 0], [2, 1]]), Matrix([[2, 1], [0, 3]])),
....:          left=vector([1, 0]), right=vector([0, 1]))
sage: all(U[n] == u(n) for n in srange(30))
True
```

```
>>> from sage.all import *
>>> U = Seq2((Matrix([[Integer(3), Integer(0)], [Integer(2), Integer(1)]]),
↪Matrix([[Integer(2), Integer(1)], [Integer(0), Integer(3)]])),
...          left=vector([Integer(1), Integer(0)]), right=vector([Integer(0),
↪Integer(1)]))
>>> all(U[n] == u(n) for n in srange(Integer(30)))
True
```

Various

See also:

recognizable series, *sage.rings.cfinite_sequence*, *sage.combinat.binary_recurrence_sequences*.

AUTHORS:

- Daniel Krenn (2016, 2021)
- Gabriel F. Lipnik (2021)

ACKNOWLEDGEMENT:

- Daniel Krenn is supported by the Austrian Science Fund (FWF): P 24644-N26.
- Gabriel F. Lipnik is supported by the Austrian Science Fund (FWF): W 1230.

Classes and Methods

exception sage.combinat.regular_sequence.**DegeneratedSequenceError**

Bases: RuntimeError

Exception raised if a degenerated sequence (see *is_degenerated()*) is detected.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: Seq2((Matrix([2]), Matrix([3])), vector([1]), vector([1]))
Traceback (most recent call last):
...
DegeneratedSequenceError: degenerated sequence: mu[0]*right != right.
Using such a sequence might lead to wrong results.
You can use 'allow_degenerated_sequence=True' followed
by a call of method .regenerated() for correcting this.
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> Seq2((Matrix([Integer(2)]), Matrix([Integer(3)])), vector([Integer(1)]),␣
↪vector([Integer(1)]))
Traceback (most recent call last):
...
DegeneratedSequenceError: degenerated sequence: mu[0]*right != right.
Using such a sequence might lead to wrong results.
You can use 'allow_degenerated_sequence=True' followed
by a call of method .regenerated() for correcting this.
```

class sage.combinat.regular_sequence.**RecurrenceParser**(*k*, *coefficient_ring*)

Bases: object

A parser for recurrence relations that allow the construction of a k-linear representation for the sequence satisfying these recurrence relations.

This is used by *RegularSequenceRing.from_recurrence()* to construct a *RegularSequence*.

ind(*M*, *m*, *ll*, *uu*)

Determine the index operator corresponding to the recursive sequence as defined in [HKL2022].

INPUT:

- M, m – parameters of the recursive sequences, see [HKL2022], Definition 3.1
- ll, uu – parameters of the resulting linear representation, see [HKL2022], Theorem A

OUTPUT:

A dictionary which maps both row numbers to subsequence parameters and vice versa, i.e.,

- ind[i] – a pair (j, d) representing the sequence $x(k^j n + d)$ in the i-th component (0-based) of the resulting linear representation
- ind[(j, d)] – the (0-based) row number of the sequence $x(k^j n + d)$ in the linear representation

EXAMPLES:

```
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: RP.ind(3, 1, -3, 3)
{(0, 0): 0, (1, -1): 3, (1, -2): 2, (1, -3): 1,
(1, 0): 4, (1, 1): 5, (1, 2): 6, (1, 3): 7, (2, -1): 10,
(2, -2): 9, (2, -3): 8, (2, 0): 11, (2, 1): 12, (2, 2): 13,
(2, 3): 14, (2, 4): 15, (2, 5): 16, 0: (0, 0), 1: (1, -3),
10: (2, -1), 11: (2, 0), 12: (2, 1), 13: (2, 2), 14: (2, 3),
15: (2, 4), 16: (2, 5), 2: (1, -2), 3: (1, -1), 4: (1, 0),
5: (1, 1), 6: (1, 2), 7: (1, 3), 8: (2, -3), 9: (2, -2)}
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import RecurrenceParser
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> RP.ind(Integer(3), Integer(1), -Integer(3), Integer(3))
{(0, 0): 0, (1, -1): 3, (1, -2): 2, (1, -3): 1,
(1, 0): 4, (1, 1): 5, (1, 2): 6, (1, 3): 7, (2, -1): 10,
(2, -2): 9, (2, -3): 8, (2, 0): 11, (2, 1): 12, (2, 2): 13,
(2, 3): 14, (2, 4): 15, (2, 5): 16, 0: (0, 0), 1: (1, -3),
10: (2, -1), 11: (2, 0), 12: (2, 1), 13: (2, 2), 14: (2, 3),
15: (2, 4), 16: (2, 5), 2: (1, -2), 3: (1, -1), 4: (1, 0),
5: (1, 1), 6: (1, 2), 7: (1, 3), 8: (2, -3), 9: (2, -2)}
```

See also

RegularSequenceRing.from_recurrence()

left(*recurrence_rules*)

Construct the vector `left` of the linear representation of recursive sequences.

INPUT:

- `recurrence_rules` – a namedtuple generated by *parameters()*; it only needs to contain a field `dim` (a positive integer)

OUTPUT: a vector

EXAMPLES:

```
sage: from collections import namedtuple
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: RRD = namedtuple('recurrence_rules_dim',
....:                  ['dim', 'inhomogeneities'])
sage: recurrence_rules = RRD(dim=5, inhomogeneities={})
sage: RP.left(recurrence_rules)
(1, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> from collections import namedtuple
>>> from sage.combinat.regular_sequence import RecurrenceParser
```

(continues on next page)

(continued from previous page)

```
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> RRD = namedtuple('recurrence_rules_dim',
...                  ['dim', 'inhomogeneities'])
>>> recurrence_rules = RRD(dim=Integer(5), inhomogeneities={})
>>> RP.left(recurrence_rules)
(1, 0, 0, 0, 0)
```

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: RRD = namedtuple('recurrence_rules_dim',
....:                  ['M', 'm', 'll', 'uu', 'dim', 'inhomogeneities'])
sage: recurrence_rules = RRD(M=3, m=2, ll=0, uu=9, dim=5,
....:                        inhomogeneities={0: Seq2.one_hadamard()})
sage: RP.left(recurrence_rules)
(1, 0, 0, 0, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> RRD = namedtuple('recurrence_rules_dim',
...                  ['M', 'm', 'll', 'uu', 'dim', 'inhomogeneities'])
>>> recurrence_rules = RRD(M=Integer(3), m=Integer(2), ll=Integer(0),␣
↪uu=Integer(9), dim=Integer(5),
...                        inhomogeneities={Integer(0): Seq2.one_hadamard()})
>>> RP.left(recurrence_rules)
(1, 0, 0, 0, 0, 0, 0, 0)
```

See also

RegularSequenceRing.from_recurrence()

matrix(*recurrence_rules*, *rem*, *correct_offset=True*)

Construct the matrix for remainder `rem` of the linear representation of the sequence represented by `recurrence_rules`.

INPUT:

- `recurrence_rules` – a namedtuple generated by *parameters()*
- `rem` – integer between 0 and $k - 1$
- `correct_offset` – boolean (default: `True`); if `True`, then the resulting linear representation has no offset. See [HKL2022] for more information.

OUTPUT: a matrix

EXAMPLES:

The following example illustrates how the coefficients in the right-hand sides of the recurrence relations correspond to the entries of the matrices.

```
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: var('n')
n
```

(continues on next page)

(continued from previous page)

```
sage: function('f')
f
sage: M, m, coeffs, initial_values = RP.parse_recurrence([
....:     f(8*n) == -1*f(2*n - 1) + 1*f(2*n + 1),
....:     f(8*n + 1) == -11*f(2*n - 1) + 10*f(2*n) + 11*f(2*n + 1),
....:     f(8*n + 2) == -21*f(2*n - 1) + 20*f(2*n) + 21*f(2*n + 1),
....:     f(8*n + 3) == -31*f(2*n - 1) + 30*f(2*n) + 31*f(2*n + 1),
....:     f(8*n + 4) == -41*f(2*n - 1) + 40*f(2*n) + 41*f(2*n + 1),
....:     f(8*n + 5) == -51*f(2*n - 1) + 50*f(2*n) + 51*f(2*n + 1),
....:     f(8*n + 6) == -61*f(2*n - 1) + 60*f(2*n) + 61*f(2*n + 1),
....:     f(8*n + 7) == -71*f(2*n - 1) + 70*f(2*n) + 71*f(2*n + 1),
....:     f(0) == 0, f(1) == 1, f(2) == 2, f(3) == 3, f(4) == 4,
....:     f(5) == 5, f(6) == 6, f(7) == 7], f, n)
sage: rules = RP.parameters(
....:     M, m, coeffs, initial_values, 0)
sage: RP.matrix(rules, 0, False)
[  0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0]
[  0 -51  50  51   0   0   0   0   0   0   0   0   0   0   0   0   0]
[  0 -61  60  61   0   0   0   0   0   0   0   0   0   0   0   0   0]
[  0 -71  70  71   0   0   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0  -1   0   1   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -11  10  11   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -21  20  21   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -31  30  31   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -41  40  41   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -51  50  51   0   0   0   0   0   0   0   0   0   0   0]
sage: RP.matrix(rules, 1, False)
[  0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1]
[  0   0   0 -11  10  11   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -21  20  21   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -31  30  31   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -41  40  41   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -51  50  51   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -61  60  61   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -71  70  71   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0  -1   0   1   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0 -11  10  11   0   0   0   0   0   0   0   0   0]
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import RecurrenceParser
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> var('n')
n
>>> function('f')
f
>>> M, m, coeffs, initial_values = RP.parse_recurrence([
...     f(Integer(8)*n) == -Integer(1)*f(Integer(2)*n - Integer(1)) +␣
↪Integer(1)*f(Integer(2)*n + Integer(1)),
...     f(Integer(8)*n + Integer(1)) == -Integer(11)*f(Integer(2)*n -␣
↪Integer(1)) + Integer(10)*f(Integer(2)*n) + Integer(11)*f(Integer(2)*n +␣
↪Integer(1)),
...     f(Integer(8)*n + Integer(2)) == -Integer(21)*f(Integer(2)*n -␣
↪Integer(1)) + Integer(20)*f(Integer(2)*n) + Integer(21)*f(Integer(2)*n +␣
↪Integer(1)),
...     f(Integer(8)*n + Integer(3)) == -Integer(31)*f(Integer(2)*n -␣
↪Integer(1)) + Integer(30)*f(Integer(2)*n) + Integer(31)*f(Integer(2)*n +␣
↪Integer(1)),
...     f(Integer(8)*n + Integer(4)) == -Integer(41)*f(Integer(2)*n -␣
↪Integer(1)) + Integer(40)*f(Integer(2)*n) + Integer(41)*f(Integer(2)*n +␣
↪Integer(1)),
...     f(Integer(8)*n + Integer(5)) == -Integer(51)*f(Integer(2)*n -␣
↪Integer(1)) + Integer(50)*f(Integer(2)*n) + Integer(51)*f(Integer(2)*n +␣
↪Integer(1)),
...     f(Integer(8)*n + Integer(6)) == -Integer(61)*f(Integer(2)*n -␣
↪Integer(1)) + Integer(60)*f(Integer(2)*n) + Integer(61)*f(Integer(2)*n +␣
↪Integer(1)),
...     f(Integer(8)*n + Integer(7)) == -Integer(71)*f(Integer(2)*n -␣
↪Integer(1)) + Integer(70)*f(Integer(2)*n) + Integer(71)*f(Integer(2)*n +␣
↪Integer(1)),
...     f(Integer(0)) == Integer(0), f(Integer(1)) == Integer(1),␣
↪f(Integer(2)) == Integer(2), f(Integer(3)) == Integer(3), f(Integer(4)) ==␣
↪Integer(4),
...     f(Integer(5)) == Integer(5), f(Integer(6)) == Integer(6),␣
↪f(Integer(7)) == Integer(7)], f, n)
>>> rules = RP.parameters(
...     M, m, coeffs, initial_values, Integer(0))
>>> RP.matrix(rules, Integer(0), False)
[  0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0]
[  0 -51  50  51   0   0   0   0   0   0   0   0   0   0   0   0   0]
[  0 -61  60  61   0   0   0   0   0   0   0   0   0   0   0   0   0]
[  0 -71  70  71   0   0   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0  -1   0   1   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -11  10  11   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -21  20  21   0   0   0   0   0   0   0   0   0   0   0]
```

(continues on next page)

(continued from previous page)

```
[  0   0   0 -31  30  31   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -41  40  41   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -51  50  51   0   0   0   0   0   0   0   0   0   0   0]
>>> RP.matrix(rules, Integer(1), False)
[  0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1]
[  0   0   0 -11  10  11   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -21  20  21   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -31  30  31   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -41  40  41   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -51  50  51   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -61  60  61   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0 -71  70  71   0   0   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0  -1   0   1   0   0   0   0   0   0   0   0   0]
[  0   0   0   0   0 -11  10  11   0   0   0   0   0   0   0   0   0]
```

Stern–Brocot Sequence:

```
sage: SB_rules = RP.parameters(
....:     1, 0, {(0, 0): 1, (1, 0): 1, (1, 1): 1},
....:     {0: 0, 1: 1, 2: 1}, 0)
sage: RP.matrix(SB_rules, 0)
[1 0 0]
[1 1 0]
[0 1 0]
sage: RP.matrix(SB_rules, 1)
[1 1 0]
[0 1 0]
[0 1 1]
```

```
>>> from sage.all import *
>>> SB_rules = RP.parameters(
...     Integer(1), Integer(0), {(Integer(0), Integer(0)): Integer(1),
↪(Integer(1), Integer(0)): Integer(1), (Integer(1), Integer(1)): Integer(1)},
...     {Integer(0): Integer(0), Integer(1): Integer(1), Integer(2):
↪Integer(1)}, Integer(0))
>>> RP.matrix(SB_rules, Integer(0))
[1 0 0]
[1 1 0]
[0 1 0]
>>> RP.matrix(SB_rules, Integer(1))
[1 1 0]
[0 1 0]
[0 1 1]
```

Number of Unbordered Factors in the Thue–Morse Sequence:

```
sage: M, m, coeffs, initial_values = RP.parse_recurrence([
....:     f(8*n) == 2*f(4*n),
....:     f(8*n + 1) == f(4*n + 1),
....:     f(8*n + 2) == f(4*n + 1) + f(4*n + 3),
....:     f(8*n + 3) == -f(4*n + 1) + f(4*n + 2),
....:     f(8*n + 4) == 2*f(4*n + 2),
....:     f(8*n + 5) == f(4*n + 3),
....:     f(8*n + 6) == -f(4*n + 1) + f(4*n + 2) + f(4*n + 3),
....:     f(8*n + 7) == 2*f(4*n + 1) + f(4*n + 3),
....:     f(0) == 1, f(1) == 2, f(2) == 2, f(3) == 4, f(4) == 2,
....:     f(5) == 4, f(6) == 6, f(7) == 0, f(8) == 4, f(9) == 4,
....:     f(10) == 4, f(11) == 4, f(12) == 12, f(13) == 0, f(14) == 4,
....:     f(15) == 4, f(16) == 8, f(17) == 4, f(18) == 8, f(19) == 0,
....:     f(20) == 8, f(21) == 4, f(22) == 4, f(23) == 8], f, n)
sage: UB_rules = RP.parameters(
....:     M, m, coeffs, initial_values, 3)
sage: RP.matrix(UB_rules, 0)
[ 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  2  0  0  0  0  0  0  0  0  0 -1  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  1  0  0  0  0  0  0 -4  0  0]
[ 0  0  0  0 -1  1  0  0  0  0  0  0  0  4  2  0]
[ 0  0  0  0  0  2  0  0  0  0  0  0  0 -2  0  0]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0 -1  1  1  0  0  0  0  0  0  2  2  0]
[ 0  0  0  0  2  0  1  0  0  0  0  0  0 -8 -4 -4]
[ 0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0]
sage: RP.matrix(UB_rules, 1)
[ 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  2  0  0  0  0  0  0  0 -2  0  0]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0 -1  1  1  0  0  0  0  0  0  2  2  0]
[ 0  0  0  0  2  0  1  0  0  0  0  0  0 -8 -4 -4]
[ 0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0 -1  1  0  0  0  2  0  0]
[ 0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> M, m, coeffs, initial_values = RP.parse_recurrence([
...     f(Integer(8)*n) == Integer(2)*f(Integer(4)*n),
...     f(Integer(8)*n + Integer(1)) == f(Integer(4)*n + Integer(1)),
...     f(Integer(8)*n + Integer(2)) == f(Integer(4)*n + Integer(1)) +␣
↪f(Integer(4)*n + Integer(3)),
...     f(Integer(8)*n + Integer(3)) == -f(Integer(4)*n + Integer(1)) +␣
↪f(Integer(4)*n + Integer(2)),
...     f(Integer(8)*n + Integer(4)) == Integer(2)*f(Integer(4)*n +␣
↪Integer(2)),
...     f(Integer(8)*n + Integer(5)) == f(Integer(4)*n + Integer(3)),
...     f(Integer(8)*n + Integer(6)) == -f(Integer(4)*n + Integer(1)) +␣
↪f(Integer(4)*n + Integer(2)) + f(Integer(4)*n + Integer(3)),
...     f(Integer(8)*n + Integer(7)) == Integer(2)*f(Integer(4)*n +␣
↪Integer(1)) + f(Integer(4)*n + Integer(3)),
...     f(Integer(0)) == Integer(1), f(Integer(1)) == Integer(2),␣
↪f(Integer(2)) == Integer(2), f(Integer(3)) == Integer(4), f(Integer(4)) ==␣
↪Integer(2),
...     f(Integer(5)) == Integer(4), f(Integer(6)) == Integer(6),␣
↪f(Integer(7)) == Integer(0), f(Integer(8)) == Integer(4), f(Integer(9)) ==␣
↪Integer(4),
...     f(Integer(10)) == Integer(4), f(Integer(11)) == Integer(4),␣
↪f(Integer(12)) == Integer(12), f(Integer(13)) == Integer(0), f(Integer(14))␣
↪== Integer(4),
...     f(Integer(15)) == Integer(4), f(Integer(16)) == Integer(8),␣
↪f(Integer(17)) == Integer(4), f(Integer(18)) == Integer(8), f(Integer(19))␣
↪== Integer(0),
...     f(Integer(20)) == Integer(8), f(Integer(21)) == Integer(4),␣
↪f(Integer(22)) == Integer(4), f(Integer(23)) == Integer(8)], f, n)
>>> UB_rules = RP.parameters(
...     M, m, coeffs, initial_values, Integer(3))
>>> RP.matrix(UB_rules, Integer(0))
[ 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  2  0  0  0  0  0  0  0  0  0 -1  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  1  0  0  0  0  0  0 -4  0  0]
[ 0  0  0  0 -1  1  0  0  0  0  0  0  0  4  2  0]
[ 0  0  0  0  0  2  0  0  0  0  0  0  0 -2  0  0]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0 -1  1  1  0  0  0  0  0  0  2  2  0]
[ 0  0  0  0  2  0  1  0  0  0  0  0  0 -8 -4 -4]
[ 0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0]
>>> RP.matrix(UB_rules, Integer(1))
[ 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  2  0  0  0  0  0  0  0 -2  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0 -1  1  1  0  0  0  0  0  0  2  2  0]
[ 0  0  0  0  2  0  1  0  0  0  0  0  0 -8 -4 -4]
[ 0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0 -1  1  0  0  0  2  0  0]
[ 0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
```

See also

RegularSequenceRing.from_recurrence()

parameters(*M*, *m*, *coeffs*, *initial_values*, *offset=0*, *inhomogeneities={}*)

Determine parameters from recurrence relations as admissible in *RegularSequenceRing.from_recurrence()*.

INPUT:

All parameters are explained in the high-level method *RegularSequenceRing.from_recurrence()*.

OUTPUT: a namedtuple `recurrence_rules` consisting of

- `M`, `m`, `l`, `u`, `offset` – parameters of the recursive sequences, see [HKL2022], Definition 3.1
- `ll`, `uu`, `n1`, `dim` – parameters and dimension of the resulting linear representation, see [HKL2022], Theorem A
- `coeffs` – dictionary mapping `(r, j)` to the coefficients $c_{r,j}$ as given in [HKL2022], Equation (3.1). If `coeffs[(r, j)]` is not given for some `r` and `j`, then it is assumed to be zero.
- `initial_values` – dictionary mapping integers `n` to the `n`-th value of the sequence
- `inhomogeneities` – dictionary mapping integers `r` to the inhomogeneity g_r as given in [HKL2022], Corollary D

EXAMPLES:

```
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: RP.parameters(2, 1,
....: {(0, -2): 3, (0, 0): 1, (0, 1): 2, (1, -2): 6, (1, 0): 4,
....: (1, 1): 5, (2, -2): 9, (2, 0): 7, (2, 1): 8, (3, -2): 12,
....: (3, 0): 10, (3, 1): 11}, {0: 1, 1: 2, 2: 1, 3: 4}, 0, {0: 1})
recurrence_rules(M=2, m=1, ll=-2, u=1, ll=-6, uu=3, dim=14,
coeffs={(0, -2): 3, (0, 0): 1, (0, 1): 2, (1, -2): 6, (1, 0): 4,
(1, 1): 5, (2, -2): 9, (2, 0): 7, (2, 1): 8, (3, -2): 12,
(3, 0): 10, (3, 1): 11}, initial_values={0: 1, 1: 2, 2: 1, 3: 4,
4: 13, 5: 30, 6: 48, 7: 66, 8: 77, 9: 208, 10: 340, 11: 472,
12: 220, 13: 600, -6: 0, -5: 0, -4: 0, -3: 0, -2: 0, -1: 0},
```

(continues on next page)

(continued from previous page)

```
offset=1, n1=3, inhomogeneities={0: 2-regular sequence 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, ...})
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import RecurrenceParser
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> RP.parameters(Integer(2), Integer(1),
... {(Integer(0), -Integer(2)): Integer(3), (Integer(0), Integer(0)):␣
↪Integer(1), (Integer(0), Integer(1)): Integer(2), (Integer(1), -
↪Integer(2)): Integer(6), (Integer(1), Integer(0)): Integer(4),
... (Integer(1), Integer(1)): Integer(5), (Integer(2), -Integer(2)):␣
↪Integer(9), (Integer(2), Integer(0)): Integer(7), (Integer(2), Integer(1)):␣
↪Integer(8), (Integer(3), -Integer(2)): Integer(12),
... (Integer(3), Integer(0)): Integer(10), (Integer(3), Integer(1)):␣
↪Integer(11)}, {Integer(0): Integer(1), Integer(1): Integer(2), Integer(2):␣
↪Integer(1), Integer(3): Integer(4)}, Integer(0), {Integer(0): Integer(1)})
recurrence_rules(M=2, m=1, l=-2, u=1, ll=-6, uu=3, dim=14,
coeffs={(0, -2): 3, (0, 0): 1, (0, 1): 2, (1, -2): 6, (1, 0): 4,
(1, 1): 5, (2, -2): 9, (2, 0): 7, (2, 1): 8, (3, -2): 12,
(3, 0): 10, (3, 1): 11}, initial_values={0: 1, 1: 2, 2: 1, 3: 4,
4: 13, 5: 30, 6: 48, 7: 66, 8: 77, 9: 208, 10: 340, 11: 472,
12: 220, 13: 600, -6: 0, -5: 0, -4: 0, -3: 0, -2: 0, -1: 0},
offset=1, n1=3, inhomogeneities={0: 2-regular sequence 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, ...})
```

See also

RegularSequenceRing.from_recurrence()

parse_direct_arguments(*M*, *m*, *coeffs*, *initial_values*)

Check whether the direct arguments as admissible in *RegularSequenceRing.from_recurrence()* are valid.

INPUT:

All parameters are explained in the high-level method *RegularSequenceRing.from_recurrence()*.

OUTPUT: a tuple consisting of the input parameters

EXAMPLES:

```
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: RP.parse_direct_arguments(2, 1,
....:     {(0, -2): 3, (0, 0): 1, (0, 1): 2,
....:      (1, -2): 6, (1, 0): 4, (1, 1): 5,
....:      (2, -2): 9, (2, 0): 7, (2, 1): 8,
....:      (3, -2): 12, (3, 0): 10, (3, 1): 11},
....:     {0: 1, 1: 2, 2: 1})
(2, 1, {(0, -2): 3, (0, 0): 1, (0, 1): 2,
(1, -2): 6, (1, 0): 4, (1, 1): 5,
(2, -2): 9, (2, 0): 7, (2, 1): 8,
```

(continues on next page)

(continued from previous page)

```
(3, -2): 12, (3, 0): 10, (3, 1): 11},
{0: 1, 1: 2, 2: 1})
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import RecurrenceParser
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> RP.parse_direct_arguments(Integer(2), Integer(1),
...     {(Integer(0), -Integer(2)): Integer(3), (Integer(0), Integer(0)):␣
↪Integer(1), (Integer(0), Integer(1)): Integer(2),
...      (Integer(1), -Integer(2)): Integer(6), (Integer(1), Integer(0)):␣
↪Integer(4), (Integer(1), Integer(1)): Integer(5),
...      (Integer(2), -Integer(2)): Integer(9), (Integer(2), Integer(0)):␣
↪Integer(7), (Integer(2), Integer(1)): Integer(8),
...      (Integer(3), -Integer(2)): Integer(12), (Integer(3), Integer(0)):␣
↪Integer(10), (Integer(3), Integer(1)): Integer(11)},
...     {Integer(0): Integer(1), Integer(1): Integer(2), Integer(2):␣
↪Integer(1)})
(2, 1, {(0, -2): 3, (0, 0): 1, (0, 1): 2,
(1, -2): 6, (1, 0): 4, (1, 1): 5,
(2, -2): 9, (2, 0): 7, (2, 1): 8,
(3, -2): 12, (3, 0): 10, (3, 1): 11},
{0: 1, 1: 2, 2: 1})
```

Stern–Brocot Sequence:

```
sage: RP.parse_direct_arguments(1, 0,
....:     {(0, 0): 1, (1, 0): 1, (1, 1): 1},
....:     {0: 0, 1: 1})
(1, 0, {(0, 0): 1, (1, 0): 1, (1, 1): 1}, {0: 0, 1: 1})
```

```
>>> from sage.all import *
>>> RP.parse_direct_arguments(Integer(1), Integer(0),
...     {(Integer(0), Integer(0)): Integer(1), (Integer(1), Integer(0)):␣
↪Integer(1), (Integer(1), Integer(1)): Integer(1)},
...     {Integer(0): Integer(0), Integer(1): Integer(1)})
(1, 0, {(0, 0): 1, (1, 0): 1, (1, 1): 1}, {0: 0, 1: 1})
```

See also

RegularSequenceRing.from_recurrence()

parse_recurrence(*equations*, *function*, *var*)

Parse recurrence relations as admissible in *RegularSequenceRing.from_recurrence()*.

INPUT:

All parameters are explained in the high-level method *RegularSequenceRing.from_recurrence()*.

OUTPUT: a tuple consisting of

- M, m – see *RegularSequenceRing.from_recurrence()*
- coeffs – see *RegularSequenceRing.from_recurrence()*

- `initial_values` – see *RegularSequenceRing.from_recurrence()*

EXAMPLES:

```
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: var('n')
n
sage: function('f')
f
sage: RP.parse_recurrence([
....:     f(4*n) == f(2*n) + 2*f(2*n + 1) + 3*f(2*n - 2),
....:     f(4*n + 1) == 4*f(2*n) + 5*f(2*n + 1) + 6*f(2*n - 2),
....:     f(4*n + 2) == 7*f(2*n) + 8*f(2*n + 1) + 9*f(2*n - 2),
....:     f(4*n + 3) == 10*f(2*n) + 11*f(2*n + 1) + 12*f(2*n - 2),
....:     f(0) == 1, f(1) == 2, f(2) == 1], f, n)
(2, 1, {(0, -2): 3, (0, 0): 1, (0, 1): 2, (1, -2): 6, (1, 0): 4,
(1, 1): 5, (2, -2): 9, (2, 0): 7, (2, 1): 8, (3, -2): 12, (3, 0): 10,
(3, 1): 11}, {0: 1, 1: 2, 2: 1})
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import RecurrenceParser
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> var('n')
n
>>> function('f')
f
>>> RP.parse_recurrence([
...     f(Integer(4)*n) == f(Integer(2)*n) + Integer(2)*f(Integer(2)*n +␣
↪Integer(1)) + Integer(3)*f(Integer(2)*n - Integer(2)),
...     f(Integer(4)*n + Integer(1)) == Integer(4)*f(Integer(2)*n) +␣
↪Integer(5)*f(Integer(2)*n + Integer(1)) + Integer(6)*f(Integer(2)*n -␣
↪Integer(2)),
...     f(Integer(4)*n + Integer(2)) == Integer(7)*f(Integer(2)*n) +␣
↪Integer(8)*f(Integer(2)*n + Integer(1)) + Integer(9)*f(Integer(2)*n -␣
↪Integer(2)),
...     f(Integer(4)*n + Integer(3)) == Integer(10)*f(Integer(2)*n) +␣
↪Integer(11)*f(Integer(2)*n + Integer(1)) + Integer(12)*f(Integer(2)*n -␣
↪Integer(2)),
...     f(Integer(0)) == Integer(1), f(Integer(1)) == Integer(2),␣
↪f(Integer(2)) == Integer(1)], f, n)
(2, 1, {(0, -2): 3, (0, 0): 1, (0, 1): 2, (1, -2): 6, (1, 0): 4,
(1, 1): 5, (2, -2): 9, (2, 0): 7, (2, 1): 8, (3, -2): 12, (3, 0): 10,
(3, 1): 11}, {0: 1, 1: 2, 2: 1})
```

Stern–Brocot Sequence:

```
sage: RP.parse_recurrence([
....:    f(2*n) == f(n), f(2*n + 1) == f(n) + f(n + 1),
....:    f(0) == 0, f(1) == 1], f, n)
(1, 0, {(0, 0): 1, (1, 0): 1, (1, 1): 1}, {0: 0, 1: 1})
```

```
>>> from sage.all import *
>>> RP.parse_recurrence([
```

(continues on next page)

(continued from previous page)

```
...     f(Integer(2)*n) == f(n), f(Integer(2)*n + Integer(1)) == f(n) + f(n +␣
↪Integer(1)),
...     f(Integer(0)) == Integer(0), f(Integer(1)) == Integer(1)], f, n)
(1, 0, {(0, 0): 1, (1, 0): 1, (1, 1): 1}, {0: 0, 1: 1})
```

See also:

RegularSequenceRing.from_recurrence()

right(*recurrence_rules*)

Construct the vector `right` of the linear representation of the sequence induced by `recurrence_rules`.

INPUT:

- `recurrence_rules` – a namedtuple generated by *parameters()*

OUTPUT: a vector

See also:

RegularSequenceRing.from_recurrence()

shifted_inhomogeneities(*recurrence_rules*)

Return a dictionary of all needed shifted inhomogeneities as described in the proof of Corollary D in [HKL2022].

INPUT:

- `recurrence_rules` – a namedtuple generated by *parameters()*

OUTPUT:

A dictionary mapping r to the regular sequence $\sum_i g_r(n+i)$ for g_r as given in [HKL2022], Corollary D, and i between $\lfloor \ell'/k^M \rfloor$ and $\lfloor (k^{M-1} - k^m + u')/k^M \rfloor + 1$; see [HKL2022], proof of Corollary D. The first blocks of the corresponding vector-valued sequence (obtained from its linear representation) correspond to the sequences $g_r(n+i)$ where i is as in the sum above; the remaining blocks consist of other shifts which are required for the regular sequence.

EXAMPLES:

```
sage: from collections import namedtuple
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: S = Seq2((Matrix([[1, 0], [0, 1]]), Matrix([[1, 0], [1, 1]])),
....:           left=vector([0, 1]), right=vector([1, 0]))
sage: S
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
sage: RR = namedtuple('recurrence_rules',
....:                  ['M', 'm', 'll', 'uu', 'inhomogeneities'])
sage: recurrence_rules = RR(M=3, m=0, ll=-14, uu=14,
....:                         inhomogeneities={0: S, 1: S})
sage: SI = RP.shifted_inhomogeneities(recurrence_rules)
```

(continues on next page)

(continued from previous page)

```
sage: SI
{0: 2-regular sequence 4, 5, 7, 9, 11, 11, 11, 12, 13, 13, ...,
 1: 2-regular sequence 4, 5, 7, 9, 11, 11, 11, 12, 13, 13, ...}
```

```
>>> from sage.all import *
>>> from collections import namedtuple
>>> from sage.combinat.regular_sequence import RecurrenceParser
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> S = Seq2((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(1), Integer(0)], [Integer(1), Integer(1)]])),
...           left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),␣
↪Integer(0)]))
>>> S
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
>>> RR = namedtuple('recurrence_rules',
...                 ['M', 'm', 'll', 'uu', 'inhomogeneities'])
>>> recurrence_rules = RR(M=Integer(3), m=Integer(0), ll=-Integer(14),␣
↪uu=Integer(14),
...                       inhomogeneities={Integer(0): S, Integer(1): S})
>>> SI = RP.shifted_inhomogeneities(recurrence_rules)
>>> SI
{0: 2-regular sequence 4, 5, 7, 9, 11, 11, 11, 12, 13, 13, ...,
 1: 2-regular sequence 4, 5, 7, 9, 11, 11, 11, 12, 13, 13, ...}
```

The first blocks of the corresponding vector-valued sequence correspond to the corresponding shifts of the inhomogeneity. In this particular case, there are no other blocks:

```
sage: lower = -2
sage: upper = 3
sage: SI[0].dimension() == S.dimension() * (upper - lower + 1)
True
sage: all(
....:     Seq2(
....:         SI[0].mu,
....:         vector((i - lower)*[0, 0] + list(S.left) + (upper - i)*[0, 0]),
....:         SI[0].right)
....:     == S.subsequence(1, i)
....:     for i in range(lower, upper+1))
True
```

```
>>> from sage.all import *
>>> lower = -Integer(2)
>>> upper = Integer(3)
>>> SI[Integer(0)].dimension() == S.dimension() * (upper - lower + Integer(1))
True
>>> all(
...     Seq2(
...         SI[Integer(0)].mu,
...         vector((i - lower)*[Integer(0), Integer(0)] + list(S.left) +␣
↪(upper - i)*[Integer(0), Integer(0)]),
...         SI[Integer(0)].right)
```

(continues on next page)

(continued from previous page)

```
...         == S.subsequence(Integer(1), i)
...         for i in range(lower, upper+Integer(1)))
True
```

See also:

RegularSequenceRing.from_recurrence()

v_eval_n (*recurrence_rules*, *n*)

Return the vector $v(n)$ as given in [HKL2022], Theorem A.

INPUT:

- `recurrence_rules` – a namedtuple generated by *parameters()*
- `n` – integer

OUTPUT: a vector

EXAMPLES:

Stern–Brocot Sequence:

```
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: SB_rules = RP.parameters(
....:     1, 0, {(0, 0): 1, (1, 0): 1, (1, 1): 1},
....:     {0: 0, 1: 1, 2: 1}, 0)
sage: RP.v_eval_n(SB_rules, 0)
(0, 1, 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import RecurrenceParser
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> SB_rules = RP.parameters(
...     Integer(1), Integer(0), {(Integer(0), Integer(0)): Integer(1),␣
↪(Integer(1), Integer(0)): Integer(1), (Integer(1), Integer(1)): Integer(1)},
...     {Integer(0): Integer(0), Integer(1): Integer(1), Integer(2):␣
↪Integer(1)}, Integer(0))
>>> RP.v_eval_n(SB_rules, Integer(0))
(0, 1, 1)
```

See also:

RegularSequenceRing.from_recurrence()

values (*M*, *m*, *l*, *u*, *ll*, *coeffs*, *initial_values*, *last_value_needed*, *offset*, *inhomogeneities*)

Determine enough values of the corresponding recursive sequence by applying the recurrence relations given in *RegularSequenceRing.from_recurrence()* to the values given in `initial_values`.

INPUT:

- `M`, `m`, `l`, `u`, `offset` – parameters of the recursive sequences, see [HKL2022], Definition 3.1

- `ll` – parameter of the resulting linear representation, see [HKL2022], Theorem A
- `coeffs` – dictionary where `coeffs[(r, j)]` is the coefficient $c_{r,j}$ as given in `RegularSequenceRing.from_recurrence()`. If `coeffs[(r, j)]` is not given for some `r` and `j`, then it is assumed to be zero.
- `initial_values` – dictionary mapping integers `n` to the `n`-th value of the sequence
- `last_value_needed` – last initial value which is needed to determine the linear representation
- `inhomogeneities` – dictionary mapping integers `r` to the inhomogeneity g_r as given in [HKL2022], Corollary D

OUTPUT:

A dictionary mapping integers `n` to the `n`-th value of the sequence for all `n` up to `last_value_needed`.

EXAMPLES:

Stern–Brocot Sequence:

```
sage: from sage.combinat.regular_sequence import RecurrenceParser
sage: RP = RecurrenceParser(2, ZZ)
sage: RP.values(M=1, m=0, l=0, u=1, ll=0,
....:     coeffs={(0, 0): 1, (1, 0): 1, (1, 1): 1},
....:     initial_values={0: 0, 1: 1, 2: 1}, last_value_needed=20,
....:     offset=0, inhomogeneities={})
{0: 0, 1: 1, 2: 1, 3: 2, 4: 1, 5: 3, 6: 2, 7: 3, 8: 1, 9: 4, 10: 3,
11: 5, 12: 2, 13: 5, 14: 3, 15: 4, 16: 1, 17: 5, 18: 4, 19: 7, 20: 3}
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import RecurrenceParser
>>> RP = RecurrenceParser(Integer(2), ZZ)
>>> RP.values(M=Integer(1), m=Integer(0), l=Integer(0), u=Integer(1),␣
↪ll=Integer(0),
...     coeffs={(Integer(0), Integer(0)): Integer(1), (Integer(1),␣
↪Integer(0)): Integer(1), (Integer(1), Integer(1)): Integer(1)},
...     initial_values={Integer(0): Integer(0), Integer(1): Integer(1),␣
↪Integer(2): Integer(1)}, last_value_needed=Integer(20),
...     offset=Integer(0), inhomogeneities={})
{0: 0, 1: 1, 2: 1, 3: 2, 4: 1, 5: 3, 6: 2, 7: 3, 8: 1, 9: 4, 10: 3,
11: 5, 12: 2, 13: 5, 14: 3, 15: 4, 16: 1, 17: 5, 18: 4, 19: 7, 20: 3}
```

See also

`RegularSequenceRing.from_recurrence()`

class `sage.combinat.regular_sequence.`**`RegularSequence`**(*parent*, *mu*, *left=None*, *right=None*)

Bases: `RecognizableSeries`

A k-regular sequence.

INPUT:

- `parent` – an instance of `RegularSequenceRing`
- `mu` – a family of square matrices, all of which have the same dimension. The indices of this family are $0, ..., k-1$. `mu` may be a list or tuple of cardinality k as well. See also `mu()`.

- `left` – (default: `None`) a vector. When evaluating the sequence, this vector is multiplied from the left to the matrix product. If `None`, then this multiplication is skipped.
- `right` – (default: `None`) a vector. When evaluating the sequence, this vector is multiplied from the right to the matrix product. If `None`, then this multiplication is skipped.

When created via the parent `RegularSequenceRing`, then the following option is available.

- `allow_degenerated_sequence` – boolean (default: `False`); if set, then there will be no check if the input is a degenerated sequence (see `is_degenerated()`). Otherwise the input is checked and a `DegeneratedSequenceError` is raised if such a sequence is detected.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: S = Seq2((Matrix([[3, 0], [6, 1]]), Matrix([[0, 1], [-6, 5]])),
....:          vector([1, 0]), vector([0, 1])); S
2-regular sequence 0, 1, 3, 5, 9, 11, 15, 19, 27, 29, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> S = Seq2((Matrix([[Integer(3), Integer(0)], [Integer(6), Integer(1)]]),
↪Matrix([[Integer(0), Integer(1)], [-Integer(6), Integer(5)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(0), Integer(1)]));
↪S
2-regular sequence 0, 1, 3, 5, 9, 11, 15, 19, 27, 29, ...
```

We can access the coefficients of a sequence by

```
sage: S[5]
11
```

```
>>> from sage.all import *
>>> S[Integer(5)]
11
```

or iterating over the first, say 10, by

```
sage: from itertools import islice
sage: list(islice(S, 10))
[0, 1, 3, 5, 9, 11, 15, 19, 27, 29]
```

```
>>> from sage.all import *
>>> from itertools import islice
>>> list(islice(S, Integer(10)))
[0, 1, 3, 5, 9, 11, 15, 19, 27, 29]
```

See also

k-regular sequence, `RegularSequenceRing`.

backward_differences(***kwds*)

Return the sequence of backward differences of this k-regular sequence.

INPUT:

- `minimize` – (default: `None`) a boolean or `None`. If `True`, then `minimized()` is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a `RegularSequence`

Note: The coefficient to the index -1 is 0.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: C = Seq2((Matrix([[2, 0], [2, 1]]), Matrix([[0, 1], [-2, 3]])),
....:           vector([1, 0]), vector([0, 1]))
sage: C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
sage: C.backward_differences()
2-regular sequence 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> C = Seq2((Matrix([[Integer(2), Integer(0)], [Integer(2), Integer(1)]]),
↪Matrix([[Integer(0), Integer(1)], [-Integer(2), Integer(3)]])),
...           vector([Integer(1), Integer(0)]), vector([Integer(0),
↪Integer(1)]))
>>> C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
>>> C.backward_differences()
2-regular sequence 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
```

```
sage: E = Seq2((Matrix([[0, 1], [0, 1]]), Matrix([[0, 0], [0, 1]])),
....:           vector([1, 0]), vector([1, 1]))
sage: E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
sage: E.backward_differences()
2-regular sequence 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, ...
```

```
>>> from sage.all import *
>>> E = Seq2((Matrix([[Integer(0), Integer(1)], [Integer(0), Integer(1)]]),
↪Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(1)]])),
...           vector([Integer(1), Integer(0)]), vector([Integer(1),
↪Integer(1)]))
>>> E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
>>> E.backward_differences()
2-regular sequence 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, ...
```

coefficient_of_n(*n*, ***kwds*)

Return the n-th entry of this sequence.

INPUT:

- `n` – nonnegative integer

OUTPUT: an element of the universe of the sequence

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: S = Seq2((Matrix([[1, 0], [0, 1]]), Matrix([[0, -1], [1, 2]])),
....:          left=vector([0, 1]), right=vector([1, 0]))
sage: S[7]
3
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> S = Seq2((Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), -Integer(1)], [Integer(1), Integer(2)]])),
...          left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),␣
↪Integer(0)]))
>>> S[Integer(7)]
3
```

This is equivalent to:

```
sage: S.coefficient_of_n(7)
3
```

```
>>> from sage.all import *
>>> S.coefficient_of_n(Integer(7))
3
```

convolution(**args*, ***kwds*)

Return the product of this k-regular sequence with `other`, where the multiplication is convolution of power series.

The operator $*$ is mapped to *convolution()*.

INPUT:

- `other` – a *RegularSequence*
- `minimize` – (default: `None`) a boolean or `None`. If `True`, then *minimized()* is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a *RegularSequence*

ALGORITHM:

See pdf attached to github pull request #35894 which contains a draft describing the details of the used algorithm.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: E = Seq2((Matrix([[0, 1], [0, 1]]), Matrix([[0, 0], [0, 1]])),
....:          vector([1, 0]), vector([1, 1]))
sage: E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> E = Seq2((Matrix([[Integer(0), Integer(1)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(1)]])),
...           vector([Integer(1), Integer(0)]), vector([Integer(1),␣
↪Integer(1)]))
>>> E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
```

We can build the convolution (in the sense of power-series) of E by itself via:

```
sage: E.convolution(E)
2-regular sequence 1, 0, 2, 0, 3, 0, 4, 0, 5, 0, ...
```

```
>>> from sage.all import *
>>> E.convolution(E)
2-regular sequence 1, 0, 2, 0, 3, 0, 4, 0, 5, 0, ...
```

This is the same as using multiplication operator:

```
sage: E * E
2-regular sequence 1, 0, 2, 0, 3, 0, 4, 0, 5, 0, ...
```

```
>>> from sage.all import *
>>> E * E
2-regular sequence 1, 0, 2, 0, 3, 0, 4, 0, 5, 0, ...
```

Building `partial_sums()` can also be seen as a convolution:

```
sage: o = Seq2.one_hadamard()
sage: E * o
2-regular sequence 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, ...
sage: E * o == E.partial_sums(include_n=True)
True
```

```
>>> from sage.all import *
>>> o = Seq2.one_hadamard()
>>> E * o
2-regular sequence 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, ...
>>> E * o == E.partial_sums(include_n=True)
True
```

forward_differences(***kwds*)

Return the sequence of forward differences of this k-regular sequence.

INPUT:

- `minimize` – (default: `None`) a boolean or `None`. If `True`, then `minimized()` is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a `RegularSequence`

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: C = Seq2((Matrix([[2, 0], [2, 1]]), Matrix([[0, 1], [-2, 3]])),
....:          vector([1, 0]), vector([0, 1]))
sage: C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
sage: C.forward_differences()
2-regular sequence 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> C = Seq2((Matrix([[Integer(2), Integer(0)], [Integer(2), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(1)], [-Integer(2), Integer(3)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(0),␣
↪Integer(1)]))
>>> C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
>>> C.forward_differences()
2-regular sequence 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
```

```
sage: E = Seq2((Matrix([[0, 1], [0, 1]]), Matrix([[0, 0], [0, 1]])),
....:          vector([1, 0]), vector([1, 1]))
sage: E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
sage: E.forward_differences()
2-regular sequence -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, ...
```

```
>>> from sage.all import *
>>> E = Seq2((Matrix([[Integer(0), Integer(1)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(1)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(1),␣
↪Integer(1)]))
>>> E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
>>> E.forward_differences()
2-regular sequence -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, ...
```

is_degenerated()

Return whether this k-regular sequence is degenerated, i.e., whether this k-regular sequence does not satisfy $\mu[0]\mathit{right} = \mathit{right}$.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: Seq2((Matrix([2]), Matrix([3])), vector([1]), vector([1]))  # indirect␣
↪doctest
Traceback (most recent call last):
...
DegeneratedSequenceError: degenerated sequence: mu[0]*right != right.
Using such a sequence might lead to wrong results.
You can use 'allow_degenerated_sequence=True' followed
by a call of method .regenerated() for correcting this.
sage: S = Seq2((Matrix([2]), Matrix([3])), vector([1]), vector([1]),
```

(continues on next page)

(continued from previous page)

```
....:           allow_degenerated_sequence=True)
sage: S
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
sage: S.is_degenerated()
True
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> Seq2((Matrix([Integer(2)]), Matrix([Integer(3)])), vector([Integer(1)]),␣
↪vector([Integer(1)]))  # indirect doctest
Traceback (most recent call last):
...
DegeneratedSequenceError: degenerated sequence: mu[0]*right != right.
Using such a sequence might lead to wrong results.
You can use 'allow_degenerated_sequence=True' followed
by a call of method .regenerated() for correcting this.
>>> S = Seq2((Matrix([Integer(2)]), Matrix([Integer(3)])),␣
↪vector([Integer(1)]), vector([Integer(1)]),
...          allow_degenerated_sequence=True)
>>> S
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
>>> S.is_degenerated()
True
```

```
sage: C = Seq2((Matrix([[2, 0], [2, 1]]), Matrix([[0, 1], [-2, 3]])),
....:           vector([1, 0]), vector([0, 1]))
sage: C.is_degenerated()
False
```

```
>>> from sage.all import *
>>> C = Seq2((Matrix([[Integer(2), Integer(0)], [Integer(2), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(1)], [-Integer(2), Integer(3)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(0),␣
↪Integer(1)]))
>>> C.is_degenerated()
False
```

partial_sums(**args*, ***kwds*)

Return the sequence of partial sums of this k-regular sequence. That is, the n-th entry of the result is the sum of the first n entries in the original sequence.

INPUT:

- `include_n` – boolean (default: `False`); if set, then the n-th entry of the result is the sum of the entries up to index n (included)
- `minimize` – (default: `None`) a boolean or `None`. If `True`, then `minimized()` is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a `RegularSequence`

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)

sage: E = Seq2((Matrix([[0, 1], [0, 1]]), Matrix([[0, 0], [0, 1]])),
....:           vector([1, 0]), vector([1, 1]))
sage: E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
sage: E.partial_sums()
2-regular sequence 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, ...
sage: E.partial_sums(include_n=True)
2-regular sequence 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)

>>> E = Seq2((Matrix([[Integer(0), Integer(1)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(0)], [Integer(0), Integer(1)]])),
...           vector([Integer(1), Integer(0)]), vector([Integer(1),␣
↪Integer(1)]))
>>> E
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
>>> E.partial_sums()
2-regular sequence 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, ...
>>> E.partial_sums(include_n=True)
2-regular sequence 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, ...
```

```
sage: C = Seq2((Matrix([[2, 0], [2, 1]]), Matrix([[0, 1], [-2, 3]])),
....:           vector([1, 0]), vector([0, 1]))
sage: C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
sage: C.partial_sums()
2-regular sequence 0, 0, 1, 3, 6, 10, 15, 21, 28, 36, ...
sage: C.partial_sums(include_n=True)
2-regular sequence 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, ...
```

```
>>> from sage.all import *
>>> C = Seq2((Matrix([[Integer(2), Integer(0)], [Integer(2), Integer(1)]]),␣
↪Matrix([[Integer(0), Integer(1)], [-Integer(2), Integer(3)]])),
...           vector([Integer(1), Integer(0)]), vector([Integer(0),␣
↪Integer(1)]))
>>> C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
>>> C.partial_sums()
2-regular sequence 0, 0, 1, 3, 6, 10, 15, 21, 28, 36, ...
>>> C.partial_sums(include_n=True)
2-regular sequence 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, ...
```

The following linear representation of S is chosen badly (is degenerated, see `is_degenerated()`), as $\mu(0)$ applied on *right* does not equal *right*:

```
sage: S = Seq2((Matrix([2]), Matrix([3])), vector([1]), vector([1]),
....:           allow_degenerated_sequence=True)
sage: S
```

(continues on next page)

(continued from previous page)

```
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
```

```
>>> from sage.all import *
>>> S = Seq2((Matrix([Integer(2)]), Matrix([Integer(3)])),␣
↪vector([Integer(1)]), vector([Integer(1)]),
...          allow_degenerated_sequence=True)
>>> S
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
```

Therefore, building partial sums produces a wrong result:

```
sage: H = S.partial_sums(include_n=True, minimize=False)
sage: H
2-regular sequence 1, 5, 16, 25, 62, 80, 98, 125, 274, 310, ...
sage: H = S.partial_sums(minimize=False)
sage: H
2-regular sequence 0, 2, 10, 16, 50, 62, 80, 98, 250, 274, ...
```

```
>>> from sage.all import *
>>> H = S.partial_sums(include_n=True, minimize=False)
>>> H
2-regular sequence 1, 5, 16, 25, 62, 80, 98, 125, 274, 310, ...
>>> H = S.partial_sums(minimize=False)
>>> H
2-regular sequence 0, 2, 10, 16, 50, 62, 80, 98, 250, 274, ...
```

We can `guess()` the correct representation:

```
sage: from itertools import islice
sage: L = []; ps = 0
sage: for s in islice(S, 110):
....:     ps += s
....:     L.append(ps)
sage: G = Seq2.guess(lambda n: L[n])
sage: G
2-regular sequence 1, 4, 10, 19, 31, 49, 67, 94, 118, 154, ...
sage: G.linear_representation()
((1, 0, 0, 0),
 Finite family {0: [  0   1   0   0]
                   [  0   0   0   1]
                   [ -5   5   1   0]
                   [ 10 -17   0   8],
                1: [  0   0   1   0]
                   [ -5   3   3   0]
                   [ -5   0   6   0]
                   [-30  21  10   0]},
 (1, 1, 4, 1))
sage: G.minimized().dimension() == G.dimension()
True
```

```
>>> from sage.all import *
>>> from itertools import islice
```

(continues on next page)

(continued from previous page)

```
>>> L = []; ps = Integer(0)
>>> for s in islice(S, Integer(110)):
...     ps += s
...     L.append(ps)
>>> G = Seq2.guess(lambda n: L[n])
>>> G
2-regular sequence 1, 4, 10, 19, 31, 49, 67, 94, 118, 154, ...
>>> G.linear_representation()
((1, 0, 0, 0),
 Finite family {0: [  0   1   0   0]
                   [  0   0   0   1]
                   [ -5   5   1   0]
                   [ 10 -17   0   8],
                1: [  0   0   1   0]
                   [ -5   3   3   0]
                   [ -5   0   6   0]
                   [-30  21  10   0]},
 (1, 1, 4, 1))
>>> G.minimized().dimension() == G.dimension()
True
```

Or we regenerate the sequence S first:

```
sage: S.regenerated().partial_sums(include_n=True, minimize=False)
2-regular sequence 1, 4, 10, 19, 31, 49, 67, 94, 118, 154, ...
sage: S.regenerated().partial_sums(minimize=False)
2-regular sequence 0, 1, 4, 10, 19, 31, 49, 67, 94, 118, ...
```

```
>>> from sage.all import *
>>> S.regenerated().partial_sums(include_n=True, minimize=False)
2-regular sequence 1, 4, 10, 19, 31, 49, 67, 94, 118, 154, ...
>>> S.regenerated().partial_sums(minimize=False)
2-regular sequence 0, 1, 4, 10, 19, 31, 49, 67, 94, 118, ...
```

regenerated(**args*, ***kwds*)

Return a k-regular sequence that satisfies $\mu[0] right = right$ with the same values as this sequence.

INPUT:

- `minimize` – (default: `None`) a boolean or `None`. If `True`, then *`minimized()`* is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a *`RegularSequence`*

ALGORITHM:

Theorem B of [HKL2022] with $n_0 = 1$.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
```

The following linear representation of S is chosen badly (is degenerated, see `is_degenerated()`), as $\mu(0)$ applied on *right* does not equal *right*:

```
sage: S = Seq2((Matrix([2]), Matrix([3])), vector([1]), vector([1]),
....:          allow_degenerated_sequence=True)
sage: S
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
sage: S.is_degenerated()
True
```

```
>>> from sage.all import *
>>> S = Seq2((Matrix([Integer(2)]), Matrix([Integer(3)])),␣
↪vector([Integer(1)]), vector([Integer(1)]),
...          allow_degenerated_sequence=True)
>>> S
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
>>> S.is_degenerated()
True
```

However, we can regenerate the sequence S:

```
sage: H = S.regenerated()
sage: H
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
sage: H.linear_representation()
((1, 0),
 Finite family {0: [ 0  1]
                   [-2  3],
                1: [3 0]
                   [6 0]},
 (1, 1))
sage: H.is_degenerated()
False
```

```
>>> from sage.all import *
>>> H = S.regenerated()
>>> H
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
>>> H.linear_representation()
((1, 0),
 Finite family {0: [ 0  1]
                   [-2  3],
                1: [3 0]
                   [6 0]},
 (1, 1))
>>> H.is_degenerated()
False
```

shift_left(*b=1*, ***kwds*)

Return the sequence obtained by shifting this k-regular sequence b steps to the left.

INPUT:

- `b` – integer

- `minimize` – (default: `None`) a boolean or `None`. If `True`, then *minimized()* is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a *RegularSequence*

Note

If b is negative (i.e., actually a right-shift), then the coefficients when accessing negative indices are 0.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: C = Seq2((Matrix([[2, 0], [0, 1]]), Matrix([[2, 1], [0, 1]])),
....:          vector([1, 0]), vector([0, 1])); C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

sage: C.shift_left()
2-regular sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
sage: C.shift_left(3)
2-regular sequence 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
sage: C.shift_left(-2)
2-regular sequence 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> C = Seq2((Matrix([[Integer(2), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(2), Integer(1)], [Integer(0), Integer(1)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(0),␣
↪Integer(1)])); C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

>>> C.shift_left()
2-regular sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
>>> C.shift_left(Integer(3))
2-regular sequence 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
>>> C.shift_left(-Integer(2))
2-regular sequence 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, ...
```

shift_right(*b=1*, ***kwds*)

Return the sequence obtained by shifting this k-regular sequence b steps to the right.

INPUT:

- `b` – integer
- `minimize` – (default: `None`) a boolean or `None`. If `True`, then *minimized()* is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a *RegularSequence*

Note

If b is positive (i.e., indeed a right-shift), then the coefficients when accessing negative indices are 0.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: C = Seq2((Matrix([[2, 0], [0, 1]]), Matrix([[2, 1], [0, 1]])),
....:          vector([1, 0]), vector([0, 1])); C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

sage: C.shift_right()
2-regular sequence 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, ...
sage: C.shift_right(3)
2-regular sequence 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, ...
sage: C.shift_right(-2)
2-regular sequence 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> C = Seq2((Matrix([[Integer(2), Integer(0)], [Integer(0), Integer(1)]]),
↪Matrix([[Integer(2), Integer(1)], [Integer(0), Integer(1)]])),
...          vector([Integer(1), Integer(0)]), vector([Integer(0),
↪Integer(1)])); C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

>>> C.shift_right()
2-regular sequence 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, ...
>>> C.shift_right(Integer(3))
2-regular sequence 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, ...
>>> C.shift_right(-Integer(2))
2-regular sequence 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
```

subsequence(**args*, ***kwds*)

Return the subsequence with indices $an + b$ of this k-regular sequence.

INPUT:

- `a` – nonnegative integer
- `b` – integer

  Alternatively, this is allowed to be a dictionary $b_j \mapsto c_j$. If so and applied on $f(n)$, the result will be the sum of all $c_j \cdot f(an + b_j)$.
- `minimize` – (default: `None`) a boolean or `None`. If `True`, then `minimized()` is called after the operation, if `False`, then not. If this argument is `None`, then the default specified by the parent's `minimize_results` is used.

OUTPUT: a `RegularSequence`

Note

If b is negative (i.e., right-shift), then the coefficients when accessing negative indices are 0.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
```

We consider the sequence C with $C(n) = n$ and the following linear representation corresponding to the vector $(n, 1)$:

```
sage: C = Seq2((Matrix([[2, 0], [0, 1]]), Matrix([[2, 1], [0, 1]])),
....:           vector([1, 0]), vector([0, 1])); C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

```
>>> from sage.all import *
>>> C = Seq2((Matrix([[Integer(2), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(2), Integer(1)], [Integer(0), Integer(1)]])),
...           vector([Integer(1), Integer(0)]), vector([Integer(0),␣
↪Integer(1)])); C
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

We now extract various subsequences of C:

```
sage: C.subsequence(2, 0)
2-regular sequence 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, ...

sage: S31 = C.subsequence(3, 1); S31
2-regular sequence 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, ...
sage: S31.linear_representation()
((1, 0),
 Finite family {0: [ 0  1]
                   [-2  3],
                1: [ 6 -2]
                   [10 -3]},
 (1, 1))

sage: C.subsequence(3, 2)
2-regular sequence 2, 5, 8, 11, 14, 17, 20, 23, 26, 29, ...
```

```
>>> from sage.all import *
>>> C.subsequence(Integer(2), Integer(0))
2-regular sequence 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, ...

>>> S31 = C.subsequence(Integer(3), Integer(1)); S31
2-regular sequence 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, ...
>>> S31.linear_representation()
((1, 0),
 Finite family {0: [ 0  1]
                   [-2  3],
                1: [ 6 -2]
                   [10 -3]},
 (1, 1))

>>> C.subsequence(Integer(3), Integer(2))
2-regular sequence 2, 5, 8, 11, 14, 17, 20, 23, 26, 29, ...
```

```
sage: Srs = C.subsequence(1, -1); Srs
2-regular sequence 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, ...
sage: Srs.linear_representation()
((1, 0, 0),
 Finite family {0: [ 0  1  0]
                   [-2  3  0]
                   [-4  4  1],
                1: [ -2   2   0]
                   [  0   0   1]
                   [ 12 -12   5]},
 (0, 0, 1))
```

```
>>> from sage.all import *
>>> Srs = C.subsequence(Integer(1), -Integer(1)); Srs
2-regular sequence 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, ...
>>> Srs.linear_representation()
((1, 0, 0),
 Finite family {0: [ 0  1  0]
                   [-2  3  0]
                   [-4  4  1],
                1: [ -2   2   0]
                   [  0   0   1]
                   [ 12 -12   5]},
 (0, 0, 1))
```

We can build *backward_differences()* manually by passing a dictionary for the parameter `b`:

```
sage: Sbd = C.subsequence(1, {0: 1, -1: -1}); Sbd
2-regular sequence 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
```

```
>>> from sage.all import *
>>> Sbd = C.subsequence(Integer(1), {Integer(0): Integer(1), -Integer(1): -
↪Integer(1)}); Sbd
2-regular sequence 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
```

transposed(*allow_degenerated_sequence=False*)

Return the transposed sequence.

INPUT:

- `allow_degenerated_sequence` – boolean (default: `False`); if set, then there will be no check if the transposed sequence is a degenerated sequence (see *is_degenerated()*). Otherwise the transposed sequence is checked and a *DegeneratedSequenceError* is raised if such a sequence is detected.

OUTPUT: a *RegularSequence*

Each of the matrices in `mu` is transposed. Additionally the vectors `left` and `right` are switched.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: U = Seq2((Matrix([[3, 2], [0, 1]]), Matrix([[2, 0], [1, 3]])),
....:          left=vector([0, 1]), right=vector([1, 0]),
....:          allow_degenerated_sequence=True)
sage: U.is_degenerated()
```

(continues on next page)

(continued from previous page)

```
True
sage: Ut = U.transposed()
sage: Ut.linear_representation()
((1, 0),
 Finite family {0: [3 0]
                   [2 1],
                1: [2 1]
                   [0 3]},
 (0, 1))
sage: Ut.is_degenerated()
False

sage: Ut.transposed()
Traceback (most recent call last):
...
DegeneratedSequenceError: degenerated sequence: mu[0]*right != right.
Using such a sequence might lead to wrong results.
You can use 'allow_degenerated_sequence=True' followed
by a call of method .regenerated() for correcting this.
sage: Utt = Ut.transposed(allow_degenerated_sequence=True)
sage: Utt.is_degenerated()
True
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> U = Seq2((Matrix([[Integer(3), Integer(2)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(2), Integer(0)], [Integer(1), Integer(3)]])),
...           left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),␣
↪Integer(0)]),
...           allow_degenerated_sequence=True)
>>> U.is_degenerated()
True
>>> Ut = U.transposed()
>>> Ut.linear_representation()
((1, 0),
 Finite family {0: [3 0]
                   [2 1],
                1: [2 1]
                   [0 3]},
 (0, 1))
>>> Ut.is_degenerated()
False

>>> Ut.transposed()
Traceback (most recent call last):
...
DegeneratedSequenceError: degenerated sequence: mu[0]*right != right.
Using such a sequence might lead to wrong results.
You can use 'allow_degenerated_sequence=True' followed
by a call of method .regenerated() for correcting this.
>>> Utt = Ut.transposed(allow_degenerated_sequence=True)
>>> Utt.is_degenerated()
```

(continues on next page)

(continued from previous page)

```
True
```

See also

RecognizableSeries.transposed

class sage.combinat.regular_sequence.**RegularSequenceRing**(*k*, **args*, ***kwds*)

Bases: *RecognizableSeriesSpace*

The space of k-regular Sequences over the given coefficient_ring.

INPUT:

- k – integer at least 2 specifying the base
- coefficient_ring – a (semi-)ring
- category – (default: None) the category of this space

EXAMPLES:

```
sage: RegularSequenceRing(2, ZZ)
Space of 2-regular sequences over Integer Ring
sage: RegularSequenceRing(3, ZZ)
Space of 3-regular sequences over Integer Ring
```

```
>>> from sage.all import *
>>> RegularSequenceRing(Integer(2), ZZ)
Space of 2-regular sequences over Integer Ring
>>> RegularSequenceRing(Integer(3), ZZ)
Space of 3-regular sequences over Integer Ring
```

See also

k-regular sequence, *RegularSequence*.

Element

alias of *RegularSequence*

from_recurrence(**args*, ***kwds*)

Construct the unique k-regular sequence which fulfills the given recurrence relations and initial values. The recurrence relations have to have the specific shape of k-recursive sequences as described in [HKL2022], and are either given as symbolic equations, e.g.,

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: var('n')
n
sage: function('f')
f
sage: Seq2.from_recurrence([
....:     f(2*n) == 2*f(n), f(2*n + 1) == 3*f(n) + 4*f(n - 1),
....:     f(0) == 0, f(1) == 1], f, n)
2-regular sequence 0, 0, 0, 1, 2, 3, 4, 10, 6, 17, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> var('n')
n
>>> function('f')
f
>>> Seq2.from_recurrence([
...     f(Integer(2)*n) == Integer(2)*f(n), f(Integer(2)*n + Integer(1)) ==␣
↪Integer(3)*f(n) + Integer(4)*f(n - Integer(1)),
...     f(Integer(0)) == Integer(0), f(Integer(1)) == Integer(1)], f, n)
2-regular sequence 0, 0, 0, 1, 2, 3, 4, 10, 6, 17, ...
```

or via the parameters of the k-recursive sequence as described in the input block below:

```
sage: Seq2.from_recurrence(M=1, m=0,
....:     coeffs={(0, 0): 2, (1, 0): 3, (1, -1): 4},
....:     initial_values={0: 0, 1: 1})
2-regular sequence 0, 0, 0, 1, 2, 3, 4, 10, 6, 17, ...
```

```
>>> from sage.all import *
>>> Seq2.from_recurrence(M=Integer(1), m=Integer(0),
...     coeffs={(Integer(0), Integer(0)): Integer(2), (Integer(1),␣
↪Integer(0)): Integer(3), (Integer(1), -Integer(1)): Integer(4)},
...     initial_values={Integer(0): Integer(0), Integer(1): Integer(1)})
2-regular sequence 0, 0, 0, 1, 2, 3, 4, 10, 6, 17, ...
```

INPUT:

Positional arguments:

If the recurrence relations are represented by symbolic equations, then the following arguments are required:

- `equations` – list of equations where the elements have either the form
  - $f(k^M n + r) = c_{r,l} f(k^m n + l) + c_{r,l+1} f(k^m n + l + 1) + ... + c_{r,u} f(k^m n + u)$ for some integers $0 \leq r < k^M$, $M > m \geq 0$ and $l \leq u$, and some coefficients $c_{r,j}$ from the (semi)ring `coefficients` of the corresponding `RegularSequenceRing`, valid for all integers $n \geq$ offset for some integer offset $\geq \max(-l/k^m, 0)$ (default: `0`), and there is an equation of this form (with the same parameters M and m) for all r

  or the form

  - `f(k) == t` for some integer `k` and some `t` from the (semi)ring `coefficient_ring`.

  The recurrence relations above uniquely determine a k-regular sequence; see [HKL2022] for further information.

- `function` – symbolic function `f` occurring in the equations
- `var` – symbolic variable (`n` in the above description of `equations`)

The following second representation of the recurrence relations is particularly useful for cases where `coefficient_ring` is not compatible with `sage.symbolic.ring.SymbolicRing`. Then the following arguments are required:

- `M` – parameter of the recursive sequences, see [HKL2022], Definition 3.1, as well as in the description of `equations` above

- `m` – parameter of the recursive sequences, see [HKL2022], Definition 3.1, as well as in the description of `equations` above
- `coeffs` – dictionary where `coeffs[(r, j)]` is the coefficient $c_{r,j}$ as given in the description of `equations` above. If `coeffs[(r, j)]` is not given for some `r` and `j`, then it is assumed to be zero.
- `initial_values` – dictionary mapping integers `n` to the `n`-th value of the sequence

Optional keyword-only argument:

- `offset` – integer (default: 0); see explanation of `equations` above
- `inhomogeneities` – (default: `{}`) a dictionary mapping integers `r` to the inhomogeneity g_r as given in [HKL2022], Corollary D. All inhomogeneities have to be regular sequences from `self` or elements of `coefficient_ring`.

OUTPUT: a *RegularSequence*

EXAMPLES:

Stern–Brocot Sequence:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: var('n')
n
sage: function('f')
f
sage: SB = Seq2.from_recurrence([
....:     f(2*n) == f(n), f(2*n + 1) == f(n) + f(n + 1),
....:     f(0) == 0, f(1) == 1], f, n)
sage: SB
2-regular sequence 0, 1, 1, 2, 1, 3, 2, 3, 1, 4, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> var('n')
n
>>> function('f')
f
>>> SB = Seq2.from_recurrence([
...     f(Integer(2)*n) == f(n), f(Integer(2)*n + Integer(1)) == f(n) + f(n +␣
↪Integer(1)),
...     f(Integer(0)) == Integer(0), f(Integer(1)) == Integer(1)], f, n)
>>> SB
2-regular sequence 0, 1, 1, 2, 1, 3, 2, 3, 1, 4, ...
```

Number of Odd Entries in Pascal's Triangle:

```
sage: Seq2.from_recurrence([
....:     f(2*n) == 3*f(n), f(2*n + 1) == 2*f(n) + f(n + 1),
....:     f(0) == 0, f(1) == 1], f, n)
2-regular sequence 0, 1, 3, 5, 9, 11, 15, 19, 27, 29, ...
```

```
>>> from sage.all import *
>>> Seq2.from_recurrence([
...     f(Integer(2)*n) == Integer(3)*f(n), f(Integer(2)*n + Integer(1)) ==␣
↪Integer(2)*f(n) + f(n + Integer(1)),
```

(continues on next page)

(continued from previous page)

```
...     f(Integer(0)) == Integer(0), f(Integer(1)) == Integer(1)], f, n)
2-regular sequence 0, 1, 3, 5, 9, 11, 15, 19, 27, 29, ...
```

Number of Unbordered Factors in the Thue–Morse Sequence:

```
sage: UB = Seq2.from_recurrence([
....:     f(8*n) == 2*f(4*n),
....:     f(8*n + 1) == f(4*n + 1),
....:     f(8*n + 2) == f(4*n + 1) + f(4*n + 3),
....:     f(8*n + 3) == -f(4*n + 1) + f(4*n + 2),
....:     f(8*n + 4) == 2*f(4*n + 2),
....:     f(8*n + 5) == f(4*n + 3),
....:     f(8*n + 6) == -f(4*n + 1) + f(4*n + 2) + f(4*n + 3),
....:     f(8*n + 7) == 2*f(4*n + 1) + f(4*n + 3),
....:     f(0) == 1, f(1) == 2, f(2) == 2, f(3) == 4, f(4) == 2,
....:     f(5) == 4, f(6) == 6, f(7) == 0, f(8) == 4, f(9) == 4,
....:     f(10) == 4, f(11) == 4, f(12) == 12, f(13) == 0, f(14) == 4,
....:     f(15) == 4, f(16) == 8, f(17) == 4, f(18) == 8, f(19) == 0,
....:     f(20) == 8, f(21) == 4, f(22) == 4, f(23) == 8], f, n, offset=3)
sage: UB
2-regular sequence 1, 2, 2, 4, 2, 4, 6, 0, 4, 4, ...
```

```
>>> from sage.all import *
>>> UB = Seq2.from_recurrence([
...     f(Integer(8)*n) == Integer(2)*f(Integer(4)*n),
...     f(Integer(8)*n + Integer(1)) == f(Integer(4)*n + Integer(1)),
...     f(Integer(8)*n + Integer(2)) == f(Integer(4)*n + Integer(1)) +␣
↪f(Integer(4)*n + Integer(3)),
...     f(Integer(8)*n + Integer(3)) == -f(Integer(4)*n + Integer(1)) +␣
↪f(Integer(4)*n + Integer(2)),
...     f(Integer(8)*n + Integer(4)) == Integer(2)*f(Integer(4)*n +␣
↪Integer(2)),
...     f(Integer(8)*n + Integer(5)) == f(Integer(4)*n + Integer(3)),
...     f(Integer(8)*n + Integer(6)) == -f(Integer(4)*n + Integer(1)) +␣
↪f(Integer(4)*n + Integer(2)) + f(Integer(4)*n + Integer(3)),
...     f(Integer(8)*n + Integer(7)) == Integer(2)*f(Integer(4)*n +␣
↪Integer(1)) + f(Integer(4)*n + Integer(3)),
...     f(Integer(0)) == Integer(1), f(Integer(1)) == Integer(2),␣
↪f(Integer(2)) == Integer(2), f(Integer(3)) == Integer(4), f(Integer(4)) ==␣
↪Integer(2),
...     f(Integer(5)) == Integer(4), f(Integer(6)) == Integer(6),␣
↪f(Integer(7)) == Integer(0), f(Integer(8)) == Integer(4), f(Integer(9)) ==␣
↪Integer(4),
...     f(Integer(10)) == Integer(4), f(Integer(11)) == Integer(4),␣
↪f(Integer(12)) == Integer(12), f(Integer(13)) == Integer(0), f(Integer(14))␣
↪== Integer(4),
...     f(Integer(15)) == Integer(4), f(Integer(16)) == Integer(8),␣
↪f(Integer(17)) == Integer(4), f(Integer(18)) == Integer(8), f(Integer(19))␣
↪== Integer(0),
...     f(Integer(20)) == Integer(8), f(Integer(21)) == Integer(4),␣
↪f(Integer(22)) == Integer(4), f(Integer(23)) == Integer(8)], f, n,␣
↪offset=Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> UB
2-regular sequence 1, 2, 2, 4, 2, 4, 6, 0, 4, 4, ...
```

Binary sum of digits $S(n)$, characterized by the recurrence relations $S(4n) = S(2n)$, $S(4n+1) = S(2n+1)$, $S(4n+2) = S(2n+1)$ and $S(4n+3) = -S(2n) + 2S(2n+1)$:

```
sage: S = Seq2.from_recurrence([
....:     f(4*n) == f(2*n),
....:     f(4*n + 1) == f(2*n + 1),
....:     f(4*n + 2) == f(2*n + 1),
....:     f(4*n + 3) == -f(2*n) + 2*f(2*n + 1),
....:     f(0) == 0, f(1) == 1], f, n)
sage: S
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
```

```
>>> from sage.all import *
>>> S = Seq2.from_recurrence([
...     f(Integer(4)*n) == f(Integer(2)*n),
...     f(Integer(4)*n + Integer(1)) == f(Integer(2)*n + Integer(1)),
...     f(Integer(4)*n + Integer(2)) == f(Integer(2)*n + Integer(1)),
...     f(Integer(4)*n + Integer(3)) == -f(Integer(2)*n) +␣
↪Integer(2)*f(Integer(2)*n + Integer(1)),
...     f(Integer(0)) == Integer(0), f(Integer(1)) == Integer(1)], f, n)
>>> S
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
```

In order to check if this sequence is indeed the binary sum of digits, we construct it directly via its linear representation and compare it with `S`:

```
sage: S2 = Seq2(
....:     (Matrix([[1, 0], [0, 1]]), Matrix([[1, 0], [1, 1]])),
....:     left=vector([0, 1]), right=vector([1, 0]))
sage: (S - S2).is_trivial_zero()
True
```

```
>>> from sage.all import *
>>> S2 = Seq2(
...     (Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(1)]]),␣
↪Matrix([[Integer(1), Integer(0)], [Integer(1), Integer(1)]])),
...     left=vector([Integer(0), Integer(1)]), right=vector([Integer(1),␣
↪Integer(0)]))
>>> (S - S2).is_trivial_zero()
True
```

Alternatively, we can also use the simpler but inhomogeneous recurrence relations $S(2n) = S(n)$ and $S(2n+1) = S(n) + 1$ via direct parameters:

```
sage: S3 = Seq2.from_recurrence(M=1, m=0,
....:     coeffs={(0, 0): 1, (1, 0): 1},
....:     initial_values={0: 0, 1: 1},
....:     inhomogeneities={1: 1})
sage: S3
```

(continues on next page)

(continued from previous page)

```
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
sage: (S3 - S2).is_trivial_zero()
True
```

```
>>> from sage.all import *
>>> S3 = Seq2.from_recurrence(M=Integer(1), m=Integer(0),
...     coeffs={(Integer(0), Integer(0)): Integer(1), (Integer(1),
↪Integer(0)): Integer(1)},
...     initial_values={Integer(0): Integer(0), Integer(1): Integer(1)},
...     inhomogeneities={Integer(1): Integer(1)})
>>> S3
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
>>> (S3 - S2).is_trivial_zero()
True
```

Number of Non-Zero Elements in the Generalized Pascal's Triangle (see [LRS2017]):

```
sage: Seq2 = RegularSequenceRing(2, QQ)
sage: P = Seq2.from_recurrence([
....:     f(4*n) == 5/3*f(2*n) - 1/3*f(2*n + 1),
....:     f(4*n + 1) == 4/3*f(2*n) + 1/3*f(2*n + 1),
....:     f(4*n + 2) == 1/3*f(2*n) + 4/3*f(2*n + 1),
....:     f(4*n + 3) == -1/3*f(2*n) + 5/3*f(2*n + 1),
....:     f(0) == 1, f(1) == 2], f, n)
sage: P
2-regular sequence 1, 2, 3, 3, 4, 5, 5, 4, 5, 7, ...
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), QQ)
>>> P = Seq2.from_recurrence([
...     f(Integer(4)*n) == Integer(5)/Integer(3)*f(Integer(2)*n) - Integer(1)/
↪Integer(3)*f(Integer(2)*n + Integer(1)),
...     f(Integer(4)*n + Integer(1)) == Integer(4)/Integer(3)*f(Integer(2)*n)
↪+ Integer(1)/Integer(3)*f(Integer(2)*n + Integer(1)),
...     f(Integer(4)*n + Integer(2)) == Integer(1)/Integer(3)*f(Integer(2)*n)
↪+ Integer(4)/Integer(3)*f(Integer(2)*n + Integer(1)),
...     f(Integer(4)*n + Integer(3)) == -Integer(1)/
↪Integer(3)*f(Integer(2)*n) + Integer(5)/Integer(3)*f(Integer(2)*n +
↪Integer(1)),
...     f(Integer(0)) == Integer(1), f(Integer(1)) == Integer(2)], f, n)
>>> P
2-regular sequence 1, 2, 3, 3, 4, 5, 5, 4, 5, 7, ...
```

Finally, the same sequence can also be obtained via direct parameters without symbolic equations:

```
sage: Seq2.from_recurrence(M=2, m=1,
....:     coeffs={(0, 0): 5/3, (0, 1): -1/3,
....:             (1, 0): 4/3, (1, 1): 1/3,
....:             (2, 0): 1/3, (2, 1): 4/3,
....:             (3, 0): -1/3, (3, 1): 5/3},
....:     initial_values={0: 1, 1: 2})
2-regular sequence 1, 2, 3, 3, 4, 5, 5, 4, 5, 7, ...
```

```
>>> from sage.all import *
>>> Seq2.from_recurrence(M=Integer(2), m=Integer(1),
...     coeffs={(Integer(0), Integer(0)): Integer(5)/Integer(3), (Integer(0),␣
↪Integer(1)): -Integer(1)/Integer(3),
...             (Integer(1), Integer(0)): Integer(4)/Integer(3), (Integer(1),␣
↪Integer(1)): Integer(1)/Integer(3),
...             (Integer(2), Integer(0)): Integer(1)/Integer(3), (Integer(2),␣
↪Integer(1)): Integer(4)/Integer(3),
...             (Integer(3), Integer(0)): -Integer(1)/Integer(3), (Integer(3),
↪ Integer(1)): Integer(5)/Integer(3)},
...     initial_values={Integer(0): Integer(1), Integer(1): Integer(2)})
2-regular sequence 1, 2, 3, 3, 4, 5, 5, 4, 5, 7, ...
```

guess (*f*, *n_verify=100*, *max_exponent=10*, *sequence=None*)

Guess a k-regular sequence whose first terms coincide with $(f(n))_{n\geq 0}$.

INPUT:

- `f` – a function (callable) which determines the sequence. It takes nonnegative integers as an input
- `n_verify` – (default: `100`) a positive integer. The resulting k-regular sequence coincides with f on the first `n_verify` terms.
- `max_exponent` – (default: `10`) a positive integer specifying the maximum exponent of k which is tried when guessing the sequence, i.e., relations between $f(k^t n + r)$ are used for $0 \leq t \leq$ `max_exponent` and $0 \leq r < k^j$
- `sequence` – (default: `None`) a k-regular sequence used for bootstrapping the guessing by adding information of the linear representation of `sequence` to the guessed representation

OUTPUT: a *RegularSequence*

ALGORITHM:

For the purposes of this description, the right vector valued sequence associated with a regular sequence consists of the corresponding matrix product multiplied by the right vector, but without the left vector of the regular sequence.

The algorithm maintains a right vector valued sequence consisting of the right vector valued sequence of the argument `sequence` (replaced by an empty tuple if `sequence` is `None`) plus several components of the shape $m \mapsto f(k^t \cdot m + r)$ for suitable `t` and `r`.

Implicitly, the algorithm also maintains a $d \times n_{\mathrm{verify}}$ matrix `A` (where `d` is the dimension of the right vector valued sequence) whose columns are the current right vector valued sequence evaluated at the nonnegative integers less than n_{verify} and ensures that this matrix has full row rank.

EXAMPLES:

Binary sum of digits:

```
sage: @cached_function
....: def s(n):
....:     if n == 0:
....:         return 0
....:     return s(n//2) + ZZ(is_odd(n))
sage: all(s(n) == sum(n.digits(2)) for n in srange(10))
True
sage: [s(n) for n in srange(10)]
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2]
```

```
>>> from sage.all import *
>>> @cached_function
... def s(n):
...     if n == Integer(0):
...         return Integer(0)
...     return s(n//Integer(2)) + ZZ(is_odd(n))
>>> all(s(n) == sum(n.digits(Integer(2))) for n in srange(Integer(10)))
True
>>> [s(n) for n in srange(Integer(10))]
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2]
```

Let us guess a 2-linear representation for $s(n)$:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: import logging
sage: logging.getLogger().setLevel(logging.INFO)
sage: S1 = Seq2.guess(s); S1
INFO:...:including f_{1*m+0}
INFO:...:including f_{2*m+1}
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
sage: S1.linear_representation()
((1, 0),
 Finite family {0: [1 0]
                   [0 1],
                1: [ 0  1]
                   [-1  2]},
 (0, 1))
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> import logging
>>> logging.getLogger().setLevel(logging.INFO)
>>> S1 = Seq2.guess(s); S1
INFO:...:including f_{1*m+0}
INFO:...:including f_{2*m+1}
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
>>> S1.linear_representation()
((1, 0),
 Finite family {0: [1 0]
                   [0 1],
                1: [ 0  1]
                   [-1  2]},
 (0, 1))
```

The `INFO` messages mean that the right vector valued sequence is the sequence $(s(n), s(2n+1))^\top$.

We guess again, but this time, we use a constant sequence for bootstrapping the guessing process:

```
sage: C = Seq2.one_hadamard(); C
2-regular sequence 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
sage: S2 = Seq2.guess(s, sequence=C); S2
INFO:...:including 2-regular sequence 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
INFO:...:including f_{1*m+0}
```

(continues on next page)

(continued from previous page)

```
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
sage: S2.linear_representation()
((0, 1),
 Finite family {0: [1 0]
                   [0 1],
                1: [1 0]
                   [1 1]},
 (1, 0))
sage: S1 == S2
True
```

```
>>> from sage.all import *
>>> C = Seq2.one_hadamard(); C
2-regular sequence 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
>>> S2 = Seq2.guess(s, sequence=C); S2
INFO:...:including 2-regular sequence 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
INFO:...:including f_{1*m+0}
2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...
>>> S2.linear_representation()
((0, 1),
 Finite family {0: [1 0]
                   [0 1],
                1: [1 0]
                   [1 1]},
 (1, 0))
>>> S1 == S2
True
```

The sequence of all natural numbers:

```
sage: S = Seq2.guess(lambda n: n); S
INFO:...:including f_{1*m+0}
INFO:...:including f_{2*m+1}
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
sage: S.linear_representation()
((1, 0),
 Finite family {0: [2 0]
                   [2 1],
                1: [ 0  1]
                   [-2  3]},
 (0, 1))
```

```
>>> from sage.all import *
>>> S = Seq2.guess(lambda n: n); S
INFO:...:including f_{1*m+0}
INFO:...:including f_{2*m+1}
2-regular sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
>>> S.linear_representation()
((1, 0),
 Finite family {0: [2 0]
                   [2 1],
                1: [ 0  1]
```

(continues on next page)

(continued from previous page)

```
                    [-2  3]},
 (0, 1))
```

The indicator function of the even integers:

```
sage: S = Seq2.guess(lambda n: ZZ(is_even(n))); S
INFO:...:including f_{1*m+0}
INFO:...:including f_{2*m+0}
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
sage: S.linear_representation()
((1, 0),
 Finite family {0: [0 1]
                   [0 1],
                1: [0 0]
                   [0 1]},
 (1, 1))
```

```
>>> from sage.all import *
>>> S = Seq2.guess(lambda n: ZZ(is_even(n))); S
INFO:...:including f_{1*m+0}
INFO:...:including f_{2*m+0}
2-regular sequence 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...
>>> S.linear_representation()
((1, 0),
 Finite family {0: [0 1]
                   [0 1],
                1: [0 0]
                   [0 1]},
 (1, 1))
```

The indicator function of the odd integers:

```
sage: S = Seq2.guess(lambda n: ZZ(is_odd(n))); S
INFO:...:including f_{1*m+0}
INFO:...:including f_{2*m+1}
2-regular sequence 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, ...
sage: S.linear_representation()
((1, 0),
 Finite family {0: [0 0]
                   [0 1],
                1: [0 1]
                   [0 1]},
 (0, 1))
sage: logging.getLogger().setLevel(logging.WARN)
```

```
>>> from sage.all import *
>>> S = Seq2.guess(lambda n: ZZ(is_odd(n))); S
INFO:...:including f_{1*m+0}
INFO:...:including f_{2*m+1}
2-regular sequence 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, ...
>>> S.linear_representation()
((1, 0),
```

(continues on next page)

(continued from previous page)

```
 Finite family {0: [0 0]
                   [0 1],
                1: [0 1]
                   [0 1]},
 (0, 1))
>>> logging.getLogger().setLevel(logging.WARN)
```

The following linear representation of S is chosen badly (is degenerated, see `is_degenerated()`), as $\mu(0)$ applied on *right* does not equal *right*:

```
sage: S = Seq2((Matrix([2]), Matrix([3])), vector([1]), vector([1]),
....:          allow_degenerated_sequence=True)
sage: S
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
sage: S.is_degenerated()
True
```

```
>>> from sage.all import *
>>> S = Seq2((Matrix([Integer(2)]), Matrix([Integer(3)])),␣
↪vector([Integer(1)]), vector([Integer(1)]),
...          allow_degenerated_sequence=True)
>>> S
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
>>> S.is_degenerated()
True
```

However, we can `guess()` a 2-regular sequence of dimension 2:

```
sage: G = Seq2.guess(lambda n: S[n])
sage: G
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
sage: G.linear_representation()
((1, 0),
 Finite family {0: [ 0  1]
                   [-2  3],
                1: [3 0]
                   [6 0]},
 (1, 1))

sage: G == S.regenerated()
True
```

```
>>> from sage.all import *
>>> G = Seq2.guess(lambda n: S[n])
>>> G
2-regular sequence 1, 3, 6, 9, 12, 18, 18, 27, 24, 36, ...
>>> G.linear_representation()
((1, 0),
 Finite family {0: [ 0  1]
                   [-2  3],
                1: [3 0]
                   [6 0]},
```

(continues on next page)

(continued from previous page)

```
 (1, 1))

>>> G == S.regenerated()
True
```

one()

Return the one element of this `RegularSequenceRing`, i.e. the unique neutral element for $*$ and also the embedding of the one of the coefficient ring into this `RegularSequenceRing`.

EXAMPLES:

```
sage: Seq2 = RegularSequenceRing(2, ZZ)
sage: O = Seq2.one(); O
2-regular sequence 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
sage: O.linear_representation()
((1), Finite family {0: [1], 1: [0]}, (1))
```

```
>>> from sage.all import *
>>> Seq2 = RegularSequenceRing(Integer(2), ZZ)
>>> O = Seq2.one(); O
2-regular sequence 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
>>> O.linear_representation()
((1), Finite family {0: [1], 1: [0]}, (1))
```

some_elements()

Return some elements of this k-regular sequence.

See `TestSuite` for a typical use case.

OUTPUT: an iterator

EXAMPLES:

```
sage: tuple(RegularSequenceRing(2, ZZ).some_elements())
(2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...,
 2-regular sequence 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, ...,
 2-regular sequence 1, 1, 0, 1, -1, 0, 0, 1, -2, -1, ...,
 2-regular sequence 2, -1, 0, 0, 0, -1, 0, 0, 0, 0, ...,
 2-regular sequence 1, 1, 0, 1, 5, 0, 0, 1, -33, 5, ...,
 2-regular sequence -5, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...,
 2-regular sequence -59, -20, 0, -20, 0, 0, 0, -20, 0, 0, ...,
 ...
 2-regular sequence 2210, 170, 0, 0, 0, 0, 0, 0, 0, 0, ...)
```

```
>>> from sage.all import *
>>> tuple(RegularSequenceRing(Integer(2), ZZ).some_elements())
(2-regular sequence 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...,
 2-regular sequence 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, ...,
 2-regular sequence 1, 1, 0, 1, -1, 0, 0, 1, -2, -1, ...,
 2-regular sequence 2, -1, 0, 0, 0, -1, 0, 0, 0, 0, ...,
 2-regular sequence 1, 1, 0, 1, 5, 0, 0, 1, -33, 5, ...,
 2-regular sequence -5, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...,
 2-regular sequence -59, -20, 0, -20, 0, 0, 0, -20, 0, 0, ...,
```

(continues on next page)

(continued from previous page)

```
...
2-regular sequence 2210, 170, 0, 0, 0, 0, 0, 0, 0, 0, ...)
```

sage.combinat.regular_sequence.**pad_right**(*T*, *length*, *zero=0*)

Pad `T` to the right by using `zero` to have at least the given `length`.

INPUT:

- `T` – tuple, list or other iterable
- `length` – nonnegative integer
- `zero` – (default: `0`) the elements to pad with

OUTPUT: an object of the same type as `T`

EXAMPLES:

```
sage: from sage.combinat.regular_sequence import pad_right
sage: pad_right((1, 2, 3), 10)
(1, 2, 3, 0, 0, 0, 0, 0, 0, 0)
sage: pad_right((1, 2, 3), 2)
(1, 2, 3)
sage: pad_right([(1, 2), (3, 4)], 4, (0, 0))
[(1, 2), (3, 4), (0, 0), (0, 0)]
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import pad_right
>>> pad_right((Integer(1), Integer(2), Integer(3)), Integer(10))
(1, 2, 3, 0, 0, 0, 0, 0, 0, 0)
>>> pad_right((Integer(1), Integer(2), Integer(3)), Integer(2))
(1, 2, 3)
>>> pad_right([(Integer(1), Integer(2)), (Integer(3), Integer(4))], Integer(4),
→(Integer(0), Integer(0)))
[(1, 2), (3, 4), (0, 0), (0, 0)]
```

sage.combinat.regular_sequence.**value**(*D*, *k*)

Return the value of the expansion with digits D in base k, i.e.

$$\sum_{0\leq j<\operatorname{len} D} D[j]k^j.$$

INPUT:

- `D` – tuple or other iterable
- `k` – the base

OUTPUT:

An element in the common parent of the base k and of the entries of D

EXAMPLES:

```
sage: from sage.combinat.regular_sequence import value
sage: value(42.digits(7), 7)
42
```

```
>>> from sage.all import *
>>> from sage.combinat.regular_sequence import value
>>> value(Integer(42).digits(Integer(7)), Integer(7))
42
```

5.1.193 Restricted growth arrays

These combinatorial objects are in bijection with set partitions.

class sage.combinat.restricted_growth.**RestrictedGrowthArrays**(*n*)

Bases: UniqueRepresentation, Parent

EXAMPLES:

```
sage: from sage.combinat.restricted_growth import RestrictedGrowthArrays
sage: R = RestrictedGrowthArrays(3)
sage: R == loads(dumps(R))
True
sage: TestSuite(R).run(skip=['_test_an_element',                              #␣
↪needs sage.libs.flint
....:   '_test_enumerated_set_contains', '_test_some_elements'])
```

```
>>> from sage.all import *
>>> from sage.combinat.restricted_growth import RestrictedGrowthArrays
>>> R = RestrictedGrowthArrays(Integer(3))
>>> R == loads(dumps(R))
True
>>> TestSuite(R).run(skip=['_test_an_element',                              # needs␣
↪sage.libs.flint
...   '_test_enumerated_set_contains', '_test_some_elements'])
```

cardinality()

EXAMPLES:

```
sage: from sage.combinat.restricted_growth import RestrictedGrowthArrays
sage: R = RestrictedGrowthArrays(6)
sage: R.cardinality()                                                        #␣
↪needs sage.libs.flint
203
```

```
>>> from sage.all import *
>>> from sage.combinat.restricted_growth import RestrictedGrowthArrays
>>> R = RestrictedGrowthArrays(Integer(6))
>>> R.cardinality()                                                      #␣
↪needs sage.libs.flint
203
```

5.1.194 Ribbons

5.1.195 Ribbon Shaped Tableaux

class sage.combinat.ribbon_shaped_tableau.**RibbonShapedTableau**(*parent*, *t*)

Bases: *SkewTableau*

A ribbon shaped tableau.

For the purposes of this class, a ribbon shaped tableau is a skew tableau whose shape is a skew partition which:

- has at least one cell in row 1;
- has at least one cell in column 1;
- has exactly one cell in each of q consecutive diagonals, for some nonnegative integer q.

A ribbon is given by a list of the rows from top to bottom.

EXAMPLES:

```
sage: x = RibbonShapedTableau([[None, None, None, 2, 3], [None, 1, 4, 5], [3,␣
↪2]]); x
[[None, None, None, 2, 3], [None, 1, 4, 5], [3, 2]]
sage: x.pp()
  .  .  .  2  3
  .  1  4  5
  3  2
sage: x.shape()
[5, 4, 2] / [3, 1]
```

```
>>> from sage.all import *
>>> x = RibbonShapedTableau([[None, None, None, Integer(2), Integer(3)], [None,␣
↪Integer(1), Integer(4), Integer(5)], [Integer(3), Integer(2)]]); x
[[None, None, None, 2, 3], [None, 1, 4, 5], [3, 2]]
>>> x.pp()
  .  .  .  2  3
  .  1  4  5
  3  2
>>> x.shape()
[5, 4, 2] / [3, 1]
```

The entries labeled by `None` correspond to the inner partition. Using `None` is optional; the entries will be shifted accordingly.

```
sage: x = RibbonShapedTableau([[2,3],[1,4,5],[3,2]]); x.pp()
  .  .  .  2  3
  .  1  4  5
  3  2
```

```
>>> from sage.all import *
>>> x = RibbonShapedTableau([[Integer(2),Integer(3)],[Integer(1),Integer(4),
↪Integer(5)],[Integer(3),Integer(2)]]); x.pp()
  .  .  .  2  3
  .  1  4  5
  3  2
```

height()

Return the height of `self`.

The height is given by the number of rows in the outer partition.

EXAMPLES:

```
sage: RibbonShapedTableau([[2,3],[1,4,5]]).height()
2
```

```
>>> from sage.all import *
>>> RibbonShapedTableau([[Integer(2),Integer(3)],[Integer(1),Integer(4),
↪Integer(5)]]).height()
2
```

spin()

Return the spin of `self`.

EXAMPLES:

```
sage: RibbonShapedTableau([[2,3],[1,4,5]]).spin()
1/2
```

```
>>> from sage.all import *
>>> RibbonShapedTableau([[Integer(2),Integer(3)],[Integer(1),Integer(4),
↪Integer(5)]]).spin()
1/2
```

width()

Return the width of the ribbon.

This is given by the length of the longest row in the outer partition.

EXAMPLES:

```
sage: RibbonShapedTableau([[2,3],[1,4,5]]).width()
4
sage: RibbonShapedTableau([]).width()
0
```

```
>>> from sage.all import *
>>> RibbonShapedTableau([[Integer(2),Integer(3)],[Integer(1),Integer(4),
↪Integer(5)]]).width()
4
>>> RibbonShapedTableau([]).width()
0
```

class sage.combinat.ribbon_shaped_tableau.**RibbonShapedTableaux**(*category=None*)

Bases: *SkewTableaux*

The set of all ribbon shaped tableaux.

Element

alias of *RibbonShapedTableau*

from_shape_and_word(*shape*, *word*)

Return the ribbon corresponding to the given ribbon shape and word.

EXAMPLES:

```
sage: RibbonShapedTableaux().from_shape_and_word([1,3],[1,3,3,7])
[[None, None, 1], [3, 3, 7]]
```

```
>>> from sage.all import *
>>> RibbonShapedTableaux().from_shape_and_word([Integer(1),Integer(3)],
↪[Integer(1),Integer(3),Integer(3),Integer(7)])
[[None, None, 1], [3, 3, 7]]
```

options = Current options for Tableaux - ascii_art: repr - convention: English - display: list - latex: diagram

class sage.combinat.ribbon_shaped_tableau.**Ribbon_class**(*parent*, *t*)

Bases: *RibbonShapedTableau*

This exists solely for unpickling `Ribbon_class` objects.

class sage.combinat.ribbon_shaped_tableau.**StandardRibbonShapedTableaux**(*category=None*)

Bases: *StandardSkewTableaux*

The set of all standard ribbon shaped tableaux.

INPUT:

- `shape` – (optional) the composition shape of the rows

Element

alias of *RibbonShapedTableau*

from_permutation(*p*)

Return a standard ribbon of size `len(p)` from a permutation `p`. The lengths of each row are given by the distance between the descents of the permutation `p`.

EXAMPLES:

```
sage: import sage.combinat.ribbon_shaped_tableau as rst
sage: [StandardRibbonShapedTableaux().from_permutation(p)
....:  for p in Permutations(3)]
[[[1, 2, 3]],
 [[None, 2], [1, 3]],
 [[1, 3], [2]],
 [[None, 1], [2, 3]],
 [[1, 2], [3]],
 [[1], [2], [3]]]
```

```
>>> from sage.all import *
>>> import sage.combinat.ribbon_shaped_tableau as rst
>>> [StandardRibbonShapedTableaux().from_permutation(p)
...  for p in Permutations(Integer(3))]
[[[1, 2, 3]],
 [[None, 2], [1, 3]],
 [[1, 3], [2]],
 [[None, 1], [2, 3]],
 [[1, 2], [3]],
 [[1], [2], [3]]]
```

from_shape_and_word(*shape*, *word*)

Return the ribbon corresponding to the given ribbon shape and word.

EXAMPLES:

```
sage: StandardRibbonShapedTableaux().from_shape_and_word([2,3],[1,2,3,4,5])
[[None, None, 1, 2], [3, 4, 5]]
```

```
>>> from sage.all import *
>>> StandardRibbonShapedTableaux().from_shape_and_word([Integer(2),
↪Integer(3)],[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
[[None, None, 1, 2], [3, 4, 5]]
```

options = Current options for Tableaux - ascii_art: repr - convention: English - display: list - latex: diagram

class sage.combinat.ribbon_shaped_tableau.**StandardRibbonShapedTableaux_shape**(*shape*)

Bases: *StandardRibbonShapedTableaux*

Class of standard ribbon shaped tableaux of ribbon shape `shape`.

EXAMPLES:

```
sage: StandardRibbonShapedTableaux([2,2])
Standard ribbon shaped tableaux of shape [2, 2]
sage: StandardRibbonShapedTableaux([2,2]).first()
[[None, 2, 4], [1, 3]]
sage: StandardRibbonShapedTableaux([2,2]).last()
[[None, 1, 2], [3, 4]]

sage: # needs sage.graphs sage.modules
sage: StandardRibbonShapedTableaux([2,2]).cardinality()
5
sage: StandardRibbonShapedTableaux([2,2]).list()
[[[None, 1, 3], [2, 4]],
 [[None, 1, 2], [3, 4]],
 [[None, 2, 3], [1, 4]],
 [[None, 2, 4], [1, 3]],
 [[None, 1, 4], [2, 3]]]
sage: StandardRibbonShapedTableaux([3,2,2]).cardinality()
155
```

```
>>> from sage.all import *
>>> StandardRibbonShapedTableaux([Integer(2),Integer(2)])
Standard ribbon shaped tableaux of shape [2, 2]
>>> StandardRibbonShapedTableaux([Integer(2),Integer(2)]).first()
[[None, 2, 4], [1, 3]]
>>> StandardRibbonShapedTableaux([Integer(2),Integer(2)]).last()
[[None, 1, 2], [3, 4]]

>>> # needs sage.graphs sage.modules
>>> StandardRibbonShapedTableaux([Integer(2),Integer(2)]).cardinality()
5
>>> StandardRibbonShapedTableaux([Integer(2),Integer(2)]).list()
[[[None, 1, 3], [2, 4]],
 [[None, 1, 2], [3, 4]],
 [[None, 2, 3], [1, 4]],
 [[None, 2, 4], [1, 3]],
```

(continues on next page)

(continued from previous page)

```
[[None, 1, 4], [2, 3]]
>>> StandardRibbonShapedTableaux([Integer(3),Integer(2),Integer(2)]).cardinality()
155
```

first()

Return the first standard ribbon of `self`.

EXAMPLES:

```
sage: StandardRibbonShapedTableaux([2,2]).first()
[[None, 2, 4], [1, 3]]
```

```
>>> from sage.all import *
>>> StandardRibbonShapedTableaux([Integer(2),Integer(2)]).first()
[[None, 2, 4], [1, 3]]
```

last()

Return the last standard ribbon of `self`.

EXAMPLES:

```
sage: StandardRibbonShapedTableaux([2,2]).last()
[[None, 1, 2], [3, 4]]
```

```
>>> from sage.all import *
>>> StandardRibbonShapedTableaux([Integer(2),Integer(2)]).last()
[[None, 1, 2], [3, 4]]
```

5.1.196 Ribbon Tableaux

class sage.combinat.ribbon_tableau.**MultiSkewTableau**(*parent*, **args*, ***kwds*)

Bases: *CombinatorialElement*

A multi skew tableau which is a tuple of skew tableaux.

EXAMPLES:

```
sage: s = MultiSkewTableau([ [[None,1],[2,3]], [[1,2],[2]] ])
sage: s.size()
6
sage: s.weight()
[2, 3, 1]
sage: s.shape()
[[2, 2] / [1], [2, 1] / []]
```

```
>>> from sage.all import *
>>> s = MultiSkewTableau([ [[None,Integer(1)],[Integer(2),Integer(3)]],
↪[[Integer(1),Integer(2)],[Integer(2)]] ])
>>> s.size()
6
>>> s.weight()
[2, 3, 1]
```

(continues on next page)

(continued from previous page)

```
>>> s.shape()
[[2, 2] / [1], [2, 1] / []]
```

inversion_pairs()

Return a list of the inversion pairs of `self`.

EXAMPLES:

```
sage: s = MultiSkewTableau([ [[2,3],[5,5]], [[1,1],[3,3]], [[2],[6]] ])
sage: s.inversion_pairs()
[((0, (0, 0)), (1, (0, 0))),
 ((0, (1, 0)), (1, (0, 1))),
 ((0, (1, 1)), (1, (0, 0))),
 ((0, (1, 1)), (1, (1, 1))),
 ((0, (1, 1)), (2, (0, 0))),
 ((1, (0, 1)), (2, (0, 0))),
 ((1, (1, 1)), (2, (0, 0)))]
```

```
>>> from sage.all import *
>>> s = MultiSkewTableau([ [[Integer(2),Integer(3)],[Integer(5),Integer(5)]],␣
↪[[Integer(1),Integer(1)],[Integer(3),Integer(3)]], [[Integer(2)],
↪[Integer(6)]] ])
>>> s.inversion_pairs()
[((0, (0, 0)), (1, (0, 0))),
 ((0, (1, 0)), (1, (0, 1))),
 ((0, (1, 1)), (1, (0, 0))),
 ((0, (1, 1)), (1, (1, 1))),
 ((0, (1, 1)), (2, (0, 0))),
 ((1, (0, 1)), (2, (0, 0))),
 ((1, (1, 1)), (2, (0, 0)))]
```

inversions()

Return the number of inversion pairs of `self`.

EXAMPLES:

```
sage: t1 = SkewTableau([[1]])
sage: t2 = SkewTableau([[2]])
sage: MultiSkewTableau([t1,t1]).inversions()
0
sage: MultiSkewTableau([t1,t2]).inversions()
0
sage: MultiSkewTableau([t2,t2]).inversions()
0
sage: MultiSkewTableau([t2,t1]).inversions()
1
sage: s = MultiSkewTableau([ [[2,3],[5,5]], [[1,1],[3,3]], [[2],[6]] ])
sage: s.inversions()
7
```

```
>>> from sage.all import *
>>> t1 = SkewTableau([[Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> t2 = SkewTableau([[Integer(2)]])
>>> MultiSkewTableau([t1,t1]).inversions()
0
>>> MultiSkewTableau([t1,t2]).inversions()
0
>>> MultiSkewTableau([t2,t2]).inversions()
0
>>> MultiSkewTableau([t2,t1]).inversions()
1
>>> s = MultiSkewTableau([ [[Integer(2),Integer(3)],[Integer(5),Integer(5)]],
→[[Integer(1),Integer(1)],[Integer(3),Integer(3)]], [[Integer(2)],
→[Integer(6)]] ])
>>> s.inversions()
7
```

shape()

Return the shape of `self`.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.shape()
[[2, 2] / [1], [2, 2] / [1], [2, 2] / [1]]
```

```
>>> from sage.all import *
>>> s = SemistandardSkewTableaux([[Integer(2),Integer(2)],[Integer(1)]]).
→list()
>>> a = MultiSkewTableau([s[Integer(0)],s[Integer(1)],s[Integer(2)]])
>>> a.shape()
[[2, 2] / [1], [2, 2] / [1], [2, 2] / [1]]
```

size()

Return the size of `self`.

This is the sum of the sizes of the skew tableaux in `self`.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.size()
9
```

```
>>> from sage.all import *
>>> s = SemistandardSkewTableaux([[Integer(2),Integer(2)],[Integer(1)]]).
→list()
>>> a = MultiSkewTableau([s[Integer(0)],s[Integer(1)],s[Integer(2)]])
>>> a.size()
9
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.weight()
[5, 3, 1]
```

```
>>> from sage.all import *
>>> s = SemistandardSkewTableaux([[Integer(2),Integer(2)],[Integer(1)]]).
↪list()
>>> a = MultiSkewTableau([s[Integer(0)],s[Integer(1)],s[Integer(2)]])
>>> a.weight()
[5, 3, 1]
```

class sage.combinat.ribbon_tableau.**MultiSkewTableaux**(*category=None*)

Bases: `UniqueRepresentation`, `Parent`

Multiskew tableaux.

Element

alias of *MultiSkewTableau*

class sage.combinat.ribbon_tableau.**RibbonTableau**(*parent*, *st*)

Bases: *SkewTableau*

A ribbon tableau.

A ribbon is a connected skew shape which does not contain any 2×2 boxes. A ribbon tableau is a skew tableau whose shape is partitioned into ribbons, each of which is filled with identical entries.

EXAMPLES:

```
sage: rt = RibbonTableau([[None, 1],[2,3]]); rt
[[None, 1], [2, 3]]
sage: rt.inner_shape()
[1]
sage: rt.outer_shape()
[2, 2]

sage: rt = RibbonTableau([[None, None, 0, 0, 0], [None, 0, 0, 2], [1, 0, 1]]); rt.
↪pp()
  .  .  0  0  0
  .  0  0  2
  1  0  1
```

```
>>> from sage.all import *
>>> rt = RibbonTableau([[None, Integer(1)],[Integer(2),Integer(3)]]); rt
[[None, 1], [2, 3]]
>>> rt.inner_shape()
[1]
>>> rt.outer_shape()
[2, 2]

>>> rt = RibbonTableau([[None, None, Integer(0), Integer(0), Integer(0)], [None,␣
↪Integer(0), Integer(0), Integer(2)], [Integer(1), Integer(0), Integer(1)]]); rt.
```

(continues on next page)

(continued from previous page)

```
↪pp()
  .  .  0  0  0
  .  0  0  2
  1  0  1
```

In the previous example, each ribbon is uniquely determined by a nonzero entry. The 0 entries are used to fill in the rest of the skew shape.

Note

Sanity checks are not performed; lists can contain any object.

```
sage: RibbonTableau(expr=[[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

```
>>> from sage.all import *
>>> RibbonTableau(expr=[[Integer(1),Integer(1)],[[Integer(5)],[Integer(3),
↪Integer(4)],[Integer(1),Integer(2)]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

length()

Return the length of the ribbons into a ribbon tableau.

EXAMPLES:

```
sage: RibbonTableau([[None, 1],[2,3]]).length()
1
sage: RibbonTableau([[1,0],[2,0]]).length()
2
```

```
>>> from sage.all import *
>>> RibbonTableau([[None, Integer(1)],[Integer(2),Integer(3)]]).length()
1
>>> RibbonTableau([[Integer(1),Integer(0)],[Integer(2),Integer(0)]]).length()
2
```

to_word()

Return a word obtained from a row reading of `self`.

Warning

Unlike the `to_word` method on skew tableaux (which are a superclass of this), this method does not filter out `None` entries.

EXAMPLES:

```
sage: R = RibbonTableau([[0, 0, 3, 0], [1, 1, 0], [2, 0, 4]])
sage: R.to_word()
word: 2041100030
```

```
>>> from sage.all import *
>>> R = RibbonTableau([[Integer(0), Integer(0), Integer(3), Integer(0)],
↪[Integer(1), Integer(1), Integer(0)], [Integer(2), Integer(0), Integer(4)]])
>>> R.to_word()
word: 2041100030
```

class sage.combinat.ribbon_tableau.**RibbonTableau_class**(*parent*, *st*)

Bases: *RibbonTableau*

This exists solely for unpickling `RibbonTableau_class` objects.

class sage.combinat.ribbon_tableau.**RibbonTableaux**

Bases: `UniqueRepresentation`, `Parent`

Ribbon tableaux.

A ribbon tableau is a skew tableau whose skew shape `shape` is tiled by ribbons of length `length`. The weight `weight` is calculated from the labels on the ribbons.

Note

Here we impose the condition that the ribbon tableaux are semistandard.

INPUT(Optional):

- `shape` – skew shape as a list of lists or an object of type SkewPartition
- `length` – integer; `shape` is partitioned into ribbons of length `length`
- `weight` – list of integers; computed from the values of nonzero entries labeling the ribbons

EXAMPLES:

```
sage: RibbonTableaux([[2,1],[]], [1,1,1], 1)
Ribbon tableaux of shape [2, 1] / [] and weight [1, 1, 1] with 1-ribbons

sage: R = RibbonTableaux([[5,4,3],[2,1]], [2,1], 3)
sage: for i in R: i.pp(); print("\n")
  .  .  0  0  0
  .  0  0  2
  1  0  1

  .  .  1  0  0
  .  0  0  0
  1  0  2

  .  .  0  0  0
  .  1  0  1
  2  0  0
```

```
>>> from sage.all import *
>>> RibbonTableaux([[Integer(2),Integer(1)],[]], [Integer(1),Integer(1),
↪Integer(1)], Integer(1))
Ribbon tableaux of shape [2, 1] / [] and weight [1, 1, 1] with 1-ribbons
```

(continues on next page)

(continued from previous page)

```
>>> R = RibbonTableaux([[Integer(5),Integer(4),Integer(3)],[Integer(2),
↪Integer(1)]], [Integer(2),Integer(1)], Integer(3))
>>> for i in R: i.pp(); print("\n")
  .  .  0  0  0
  .  0  0  2
  1  0  1
<BLANKLINE>
  .  .  1  0  0
  .  0  0  0
  1  0  2
<BLANKLINE>
  .  .  0  0  0
  .  1  0  1
  2  0  0
<BLANKLINE>
```

REFERENCES:

Element

alias of *RibbonTableau*

from_expr(*l*)

Return a *RibbonTableau* from a MuPAD-Combinat expr for a skew tableau. The first list in `expr` is the inner shape of the skew tableau. The second list are the entries in the rows of the skew tableau from bottom to top.

Provided primarily for compatibility with MuPAD-Combinat.

EXAMPLES:

```
sage: RibbonTableaux().from_expr([[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

```
>>> from sage.all import *
>>> RibbonTableaux().from_expr([[Integer(1),Integer(1)],[[Integer(5)],
↪[Integer(3),Integer(4)],[Integer(1),Integer(2)]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

options = Current options for Tableaux - ascii_art: repr - convention: English - display: list - latex: diagram

class sage.combinat.ribbon_tableau.**RibbonTableaux_shape_weight_length**(*shape*, *weight*, *length*)

Bases: *RibbonTableaux*

Ribbon tableaux of a given shape, weight, and length.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: RibbonTableaux([[2,1],[]],[1,1,1],1).cardinality()
2
sage: RibbonTableaux([[2,2],[]],[1,1],2).cardinality()
2
```

(continues on next page)

(continued from previous page)

```
sage: RibbonTableaux([[4,3,3],[]],[2,1,1,1],2).cardinality()
5
```

```
>>> from sage.all import *
>>> RibbonTableaux([[Integer(2),Integer(1)],[]],[Integer(1),Integer(1),
↪Integer(1)],Integer(1)).cardinality()
2
>>> RibbonTableaux([[Integer(2),Integer(2)],[]],[Integer(1),Integer(1)],
↪Integer(2)).cardinality()
2
>>> RibbonTableaux([[Integer(4),Integer(3),Integer(3)],[]],[Integer(2),
↪Integer(1),Integer(1),Integer(1)],Integer(2)).cardinality()
5
```

class sage.combinat.ribbon_tableau.**SemistandardMultiSkewTableaux**(*shape*, *weight*)

Bases: *MultiSkewTableaux*

Semistandard multi skew tableaux.

A multi skew tableau is a k-tuple of skew tableaux of given shape with a specified total weight.

EXAMPLES:

```
sage: S = SemistandardMultiSkewTableaux([ [[2,1],[]], [[2,2],[1]] ], [2,2,2]); S
Semistandard multi skew tableaux of shape [[2, 1] / [], [2, 2] / [1]] and weight␣
↪[2, 2, 2]
sage: S.list()
[[[[1, 1], [2]], [[None, 2], [3, 3]]],
 [[[1, 2], [2]], [[None, 1], [3, 3]]],
 [[[1, 3], [2]], [[None, 2], [1, 3]]],
 [[[1, 3], [2]], [[None, 1], [2, 3]]],
 [[[1, 1], [3]], [[None, 2], [2, 3]]],
 [[[1, 2], [3]], [[None, 2], [1, 3]]],
 [[[1, 2], [3]], [[None, 1], [2, 3]]],
 [[[2, 2], [3]], [[None, 1], [1, 3]]],
 [[[1, 3], [3]], [[None, 1], [2, 2]]],
 [[[2, 3], [3]], [[None, 1], [1, 2]]]]
```

```
>>> from sage.all import *
>>> S = SemistandardMultiSkewTableaux([ [[Integer(2),Integer(1)],[]],␣
↪[[Integer(2),Integer(2)],[Integer(1)]] ], [Integer(2),Integer(2),Integer(2)]); S
Semistandard multi skew tableaux of shape [[2, 1] / [], [2, 2] / [1]] and weight␣
↪[2, 2, 2]
>>> S.list()
[[[[1, 1], [2]], [[None, 2], [3, 3]]],
 [[[1, 2], [2]], [[None, 1], [3, 3]]],
 [[[1, 3], [2]], [[None, 2], [1, 3]]],
 [[[1, 3], [2]], [[None, 1], [2, 3]]],
 [[[1, 1], [3]], [[None, 2], [2, 3]]],
 [[[1, 2], [3]], [[None, 2], [1, 3]]],
 [[[1, 2], [3]], [[None, 1], [2, 3]]],
 [[[2, 2], [3]], [[None, 1], [1, 3]]],
 [[[1, 3], [3]], [[None, 1], [2, 2]]],
```

(continues on next page)

(continued from previous page)

```
[[[2, 3], [3]], [[None, 1], [1, 2]]]]
```

sage.combinat.ribbon_tableau.**cospin_polynomial**(*part*, *weight*, *length*)

Return the cospin polynomial associated to `part`, `weight`, and `length`.

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import cospin_polynomial
sage: cospin_polynomial([6,6,6],[4,2],3)
t^4 + t^3 + 2*t^2 + t + 1
sage: cospin_polynomial([3,3,3,2,1], [3,1], 3)
1
sage: cospin_polynomial([3,3,3,2,1], [2,2], 3)
t + 1
sage: cospin_polynomial([3,3,3,2,1], [2,1,1], 3)
t^2 + 2*t + 2
sage: cospin_polynomial([3,3,3,2,1], [1,1,1,1], 3)
t^3 + 3*t^2 + 5*t + 3
sage: cospin_polynomial([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^2 + 6*t + 2
sage: cospin_polynomial([[6]*6, [3,3]], [4,4,2], 3)
3*t^4 + 6*t^3 + 9*t^2 + 5*t + 3
```

```
>>> from sage.all import *
>>> from sage.combinat.ribbon_tableau import cospin_polynomial
>>> cospin_polynomial([Integer(6),Integer(6),Integer(6)],[Integer(4),Integer(2)],
↪Integer(3))
t^4 + t^3 + 2*t^2 + t + 1
>>> cospin_polynomial([Integer(3),Integer(3),Integer(3),Integer(2),Integer(1)],␣
↪[Integer(3),Integer(1)], Integer(3))
1
>>> cospin_polynomial([Integer(3),Integer(3),Integer(3),Integer(2),Integer(1)],␣
↪[Integer(2),Integer(2)], Integer(3))
t + 1
>>> cospin_polynomial([Integer(3),Integer(3),Integer(3),Integer(2),Integer(1)],␣
↪[Integer(2),Integer(1),Integer(1)], Integer(3))
t^2 + 2*t + 2
>>> cospin_polynomial([Integer(3),Integer(3),Integer(3),Integer(2),Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1)], Integer(3))
t^3 + 3*t^2 + 5*t + 3
>>> cospin_polynomial([Integer(5),Integer(4),Integer(3),Integer(2),Integer(1),
↪Integer(1),Integer(1)], [Integer(2),Integer(2),Integer(1)], Integer(3))
2*t^2 + 6*t + 2
>>> cospin_polynomial([[Integer(6)]*Integer(6), [Integer(3),Integer(3)]],␣
↪[Integer(4),Integer(4),Integer(2)], Integer(3))
3*t^4 + 6*t^3 + 9*t^2 + 5*t + 3
```

sage.combinat.ribbon_tableau.**count_rec**(*nexts*, *current*, *part*, *weight*, *length*)

INPUT:

- `nexts, current, part` – skew partitions
- `weight` – nonnegative integer list

- `length` – integer

`sage.combinat.ribbon_tableau.`**`graph_implementation_rec`**(*skp*, *weight*, *length*, *function*)

`sage.combinat.ribbon_tableau.`**`insertion_tableau`**(*skp*, *perm*, *evaluation*, *tableau*, *length*)

INPUT:

- `skp` – skew partitions
- `perm, evaluation` – nonnegative integers
- `tableau` – skew tableau
- `length` – integer

`sage.combinat.ribbon_tableau.`**`list_rec`**(*nexts*, *current*, *part*, *weight*, *length*)

INPUT:

- `nexts, current, part` – skew partitions
- `weight` – nonnegative integer list
- `length` – integer

`sage.combinat.ribbon_tableau.`**`spin_polynomial`**(*part*, *weight*, *length*)

Return the spin polynomial associated to `part`, `weight`, and `length`.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: from sage.combinat.ribbon_tableau import spin_polynomial
sage: spin_polynomial([6,6,6],[4,2],3)
t^6 + t^5 + 2*t^4 + t^3 + t^2
sage: spin_polynomial([6,6,6],[4,1,1],3)
t^6 + 2*t^5 + 3*t^4 + 2*t^3 + t^2
sage: spin_polynomial([3,3,3,2,1], [2,2], 3)
t^(7/2) + t^(5/2)
sage: spin_polynomial([3,3,3,2,1], [2,1,1], 3)
2*t^(7/2) + 2*t^(5/2) + t^(3/2)
sage: spin_polynomial([3,3,3,2,1], [1,1,1,1], 3)
3*t^(7/2) + 5*t^(5/2) + 3*t^(3/2) + sqrt(t)
sage: spin_polynomial([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^(9/2) + 6*t^(7/2) + 2*t^(5/2)
sage: spin_polynomial([[6]*6, [3,3]], [4,4,2], 3)
3*t^9 + 5*t^8 + 9*t^7 + 6*t^6 + 3*t^5
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> from sage.combinat.ribbon_tableau import spin_polynomial
>>> spin_polynomial([Integer(6),Integer(6),Integer(6)],[Integer(4),Integer(2)],
↪Integer(3))
t^6 + t^5 + 2*t^4 + t^3 + t^2
>>> spin_polynomial([Integer(6),Integer(6),Integer(6)],[Integer(4),Integer(1),
↪Integer(1)],Integer(3))
t^6 + 2*t^5 + 3*t^4 + 2*t^3 + t^2
>>> spin_polynomial([Integer(3),Integer(3),Integer(3),Integer(2),Integer(1)],␣
↪[Integer(2),Integer(2)], Integer(3))
t^(7/2) + t^(5/2)
```

(continues on next page)

(continued from previous page)

```
>>> spin_polynomial([Integer(3),Integer(3),Integer(3),Integer(2),Integer(1)],␣
↪[Integer(2),Integer(1),Integer(1)], Integer(3))
2*t^(7/2) + 2*t^(5/2) + t^(3/2)
>>> spin_polynomial([Integer(3),Integer(3),Integer(3),Integer(2),Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1),Integer(1)], Integer(3))
3*t^(7/2) + 5*t^(5/2) + 3*t^(3/2) + sqrt(t)
>>> spin_polynomial([Integer(5),Integer(4),Integer(3),Integer(2),Integer(1),
↪Integer(1),Integer(1)], [Integer(2),Integer(2),Integer(1)], Integer(3))
2*t^(9/2) + 6*t^(7/2) + 2*t^(5/2)
>>> spin_polynomial([[Integer(6)]*Integer(6), [Integer(3),Integer(3)]],␣
↪[Integer(4),Integer(4),Integer(2)], Integer(3))
3*t^9 + 5*t^8 + 9*t^7 + 6*t^6 + 3*t^5
```

sage.combinat.ribbon_tableau.**spin_polynomial_square**(*part*, *weight*, *length*)

Return the spin polynomial associated with `part`, `weight`, and `length`, with the substitution $t \to t^2$ made.

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import spin_polynomial_square
sage: spin_polynomial_square([6,6,6],[4,2],3)
t^12 + t^10 + 2*t^8 + t^6 + t^4
sage: spin_polynomial_square([6,6,6],[4,1,1],3)
t^12 + 2*t^10 + 3*t^8 + 2*t^6 + t^4
sage: spin_polynomial_square([3,3,3,2,1], [2,2], 3)
t^7 + t^5
sage: spin_polynomial_square([3,3,3,2,1], [2,1,1], 3)
2*t^7 + 2*t^5 + t^3
sage: spin_polynomial_square([3,3,3,2,1], [1,1,1,1], 3)
3*t^7 + 5*t^5 + 3*t^3 + t
sage: spin_polynomial_square([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^9 + 6*t^7 + 2*t^5
sage: spin_polynomial_square([[6]*6, [3,3]], [4,4,2], 3)
3*t^18 + 5*t^16 + 9*t^14 + 6*t^12 + 3*t^10
```

```
>>> from sage.all import *
>>> from sage.combinat.ribbon_tableau import spin_polynomial_square
>>> spin_polynomial_square([Integer(6),Integer(6),Integer(6)],[Integer(4),
↪Integer(2)],Integer(3))
t^12 + t^10 + 2*t^8 + t^6 + t^4
>>> spin_polynomial_square([Integer(6),Integer(6),Integer(6)],[Integer(4),
↪Integer(1),Integer(1)],Integer(3))
t^12 + 2*t^10 + 3*t^8 + 2*t^6 + t^4
>>> spin_polynomial_square([Integer(3),Integer(3),Integer(3),Integer(2),
↪Integer(1)], [Integer(2),Integer(2)], Integer(3))
t^7 + t^5
>>> spin_polynomial_square([Integer(3),Integer(3),Integer(3),Integer(2),
↪Integer(1)], [Integer(2),Integer(1),Integer(1)], Integer(3))
2*t^7 + 2*t^5 + t^3
>>> spin_polynomial_square([Integer(3),Integer(3),Integer(3),Integer(2),
↪Integer(1)], [Integer(1),Integer(1),Integer(1),Integer(1)], Integer(3))
3*t^7 + 5*t^5 + 3*t^3 + t
>>> spin_polynomial_square([Integer(5),Integer(4),Integer(3),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(1),Integer(1)], [Integer(2),Integer(2),Integer(1)],␣
↪Integer(3))
2*t^9 + 6*t^7 + 2*t^5
>>> spin_polynomial_square([[Integer(6)]*Integer(6), [Integer(3),Integer(3)]],␣
↪[Integer(4),Integer(4),Integer(2)], Integer(3))
3*t^18 + 5*t^16 + 9*t^14 + 6*t^12 + 3*t^10
```

sage.combinat.ribbon_tableau.**spin_rec**(*t*, *nexts*, *current*, *part*, *weight*, *length*)

Routine used for constructing the spin polynomial.

INPUT:

- `weight` – list of nonnegative integers
- `length` – the length of the ribbons we're tiling with
- `t` – the variable

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import spin_rec
sage: sp = SkewPartition
sage: t = ZZ['t'].gen()
sage: spin_rec(t, [], [[[], [3, 3]]], sp([[2, 2, 2], []]), [2], 3)
[t^4]
sage: spin_rec(t, [[0], [t^4]], [[[2, 1, 1, 1, 1], [0, 3]], [[2, 2, 2], [3, 0]]],␣
↪sp([[2, 2, 2, 2, 1], []]), [2, 1], 3)
[t^5]
sage: spin_rec(t, [], [[[], [3, 3, 0]]], sp([[3, 3], []]), [2], 3)
[t^2]
sage: spin_rec(t, [[t^4], [t^3], [t^2]], [[[2, 2, 2], [0, 0, 3]], [[3, 2, 1], [0,␣
↪3, 0]], [[3, 3], [3, 0, 0]]], sp([[3, 3, 3], []]), [2, 1], 3)
[t^6 + t^4 + t^2]
sage: spin_rec(t, [[t^5], [t^4], [t^6 + t^4 + t^2]], [[[2, 2, 2, 2, 1], [0, 0,␣
↪3]], [[3, 3, 1, 1, 1], [0, 3, 0]], [[3, 3, 3], [3, 0, 0]]], sp([[3, 3, 3, 2, 1],
↪ []]), [2, 1, 1], 3)
[2*t^7 + 2*t^5 + t^3]
```

```
>>> from sage.all import *
>>> from sage.combinat.ribbon_tableau import spin_rec
>>> sp = SkewPartition
>>> t = ZZ['t'].gen()
>>> spin_rec(t, [], [[[], [Integer(3), Integer(3)]]], sp([[Integer(2), Integer(2),
↪ Integer(2)], []]), [Integer(2)], Integer(3))
[t^4]
>>> spin_rec(t, [[Integer(0)], [t**Integer(4)]], [[[Integer(2), Integer(1),␣
↪Integer(1), Integer(1), Integer(1)], [Integer(0), Integer(3)]], [[Integer(2),␣
↪Integer(2), Integer(2)], [Integer(3), Integer(0)]]], sp([[Integer(2),␣
↪Integer(2), Integer(2), Integer(2), Integer(1)], []]), [Integer(2), Integer(1)],
↪ Integer(3))
[t^5]
>>> spin_rec(t, [], [[[], [Integer(3), Integer(3), Integer(0)]]], sp([[Integer(3),
↪ Integer(3)], []]), [Integer(2)], Integer(3))
[t^2]
```

(continues on next page)

(continued from previous page)

```
>>> spin_rec(t, [[t**Integer(4)], [t**Integer(3)], [t**Integer(2)]],␣
↪[[[Integer(2), Integer(2), Integer(2)], [Integer(0), Integer(0), Integer(3)]],␣
↪[[Integer(3), Integer(2), Integer(1)], [Integer(0), Integer(3), Integer(0)]],␣
↪[[Integer(3), Integer(3)], [Integer(3), Integer(0), Integer(0)]]],␣
↪sp([[Integer(3), Integer(3), Integer(3)], []]), [Integer(2), Integer(1)],␣
↪Integer(3))
[t^6 + t^4 + t^2]
>>> spin_rec(t, [[t**Integer(5)], [t**Integer(4)], [t**Integer(6) + t**Integer(4)␣
↪+ t**Integer(2)]], [[[Integer(2), Integer(2), Integer(2), Integer(2),␣
↪Integer(1)], [Integer(0), Integer(0), Integer(3)]], [[Integer(3), Integer(3),␣
↪Integer(1), Integer(1), Integer(1)], [Integer(0), Integer(3), Integer(0)]],␣
↪[[Integer(3), Integer(3), Integer(3)], [Integer(3), Integer(0), Integer(0)]]],␣
↪sp([[Integer(3), Integer(3), Integer(3), Integer(2), Integer(1)], []]),␣
↪[Integer(2), Integer(1), Integer(1)], Integer(3))
[2*t^7 + 2*t^5 + t^3]
```

5.1.197 Rigged configurations

Todo

Proofread / point to the main classes rather than the modules?

- *Crystal of Rigged Configurations*
- *Rigged Configurations of \mathcal{B}(\infty)*
- *Rigged Configurations*
- *Rigged Configuration Elements*
- *Tensor Product of Kirillov-Reshetikhin Tableaux*
- *Tensor Product of Kirillov-Reshetikhin Tableaux Elements*
- *Kirillov-Reshetikhin Tableaux*
- *Kleber Trees*
- *Rigged Partitions*

Bijections

- *Bijection between rigged configurations and KR tableaux*
- *Abstract classes for the rigged configuration bijections*
- *Bijection classes for type A_n^{(1)}*
- *Bijection classes for type B_n^{(1)}*
- *Bijection classes for type C_n^{(1)}*
- *Bijection classes for type D_n^{(1)}*
- *Bijection classes for type A_{2n-1}^{(2)}.*
- *Bijection classes for type A_{2n}^{(2)}*
- *Bijection classes for type A_{2n}^{(2)\dagger}*

- *Bijection classes for type D_{n+1}^{(2)}*
- *Bijection classes for type D_4^{(3)}*
- *Bijection between rigged configurations for B(\infty) and marginally large tableaux*

5.1.198 Abstract classes for the rigged configuration bijections

This file contains two sets of classes, one for the bijection from KR tableaux to rigged configurations and the other for the reverse bijection. We do this for two reasons, one is because we can store a state in the bijection locally, so we do not have to constantly pass it around between functions. The other is because it makes the code easier to read in the *_element.py files.

These classes are not meant to be used by the user and are only supposed to be used internally to perform the bijections between `TensorProductOfKirillovReshetikhinTableaux` and `RiggedConfigurations`.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version

class sage.combinat.rigged_configurations.bij_abstract_class.**KRTToRCBijectionAbstract**(*tp_krt*)

Bases: `object`

Root abstract class for the bijection from KR tableaux to rigged configurations.

This class holds the state of the bijection and generates the next state. This class should never be created directly.

next_state(*val*)

Build the next state in the bijection.

INPUT:

- `val` – the value we are adding

run(*verbose=False*)

Run the bijection from a tensor product of KR tableaux to a rigged configuration.

INPUT:

- `tp_krt` – a tensor product of KR tableaux
- `verbose` – (default: `False`) display each step in the bijection

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1],
→[[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import
→KRTToRCBijectionTypeA
sage: KRTToRCBijectionTypeA(KRT(pathlist=[[5,2]])).run()

-1[ ]-1

1[ ]1

0[ ]0

-1[ ]-1
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',␣
↪Integer(4), Integer(1)], [[Integer(2), Integer(1)]])
>>> from sage.combinat.rigged_configurations.bij_type_A import␣
↪KRTToRCBijectionTypeA
>>> KRTToRCBijectionTypeA(KRT(pathlist=[[Integer(5),Integer(2)]])).run()
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
1[ ]1
<BLANKLINE>
0[ ]0
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
```

class sage.combinat.rigged_configurations.bij_abstract_class.**RCToKRTBijectionAbstract**(*RC_element*)

Bases: object

Root abstract class for the bijection from rigged configurations to tensor product of Kirillov-Reshetikhin tableaux.

This class holds the state of the bijection and generates the next state. This class should never be created directly.

next_state(*height*)

Build the next state in the bijection.

run(*verbose=False*, *build_graph=False*)

Run the bijection from rigged configurations to tensor product of KR tableaux.

INPUT:

- verbose – boolean (default: False); display each step in the bijection
- build_graph – boolean (default: False); build the graph of each step of the bijection

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: x = RC(partition_list=[[1],[1],[1],[1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import␣
↪RCToKRTBijectionTypeA
sage: RCToKRTBijectionTypeA(x).run()
[[2], [5]]
sage: bij = RCToKRTBijectionTypeA(x)
sage: bij.run(build_graph=True)
[[2], [5]]
sage: bij._graph
Digraph on 3 vertices
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),␣
↪Integer(1)]])
>>> x = RC(partition_list=[[Integer(1)],[Integer(1)],[Integer(1)],
↪[Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.rigged_configurations.bij_type_A import␣
↪RCToKRTBijectionTypeA
>>> RCToKRTBijectionTypeA(x).run()
[[2], [5]]
>>> bij = RCToKRTBijectionTypeA(x)
>>> bij.run(build_graph=True)
[[2], [5]]
>>> bij._graph
Digraph on 3 vertices
```

5.1.199 Bijection between rigged configurations for $B(\infty)$ and marginally large tableaux

AUTHORS:

- Travis Scrimshaw (2015-07-01): Initial version

REFERENCES:

class sage.combinat.rigged_configurations.bij_infinity.**FromRCIsomorphism**

Bases: Morphism

Crystal isomorphism of $B(\infty)$ in the rigged configuration model to the tableau model.

class sage.combinat.rigged_configurations.bij_infinity.**FromTableauIsomorphism**

Bases: Morphism

Crystal isomorphism of $B(\infty)$ in the tableau model to the rigged configuration model.

class sage.combinat.rigged_configurations.bij_infinity.**MLTToRCBijectionTypeB**(*tp_krt*)

Bases: *KRTToRCBijectionTypeB*

run()

Run the bijection from a marginally large tableaux to a rigged configuration.

EXAMPLES:

```
sage: vct = CartanType(['B',4]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: T = crystals.infinity.Tableaux(['B',4])
sage: Psi = T.crystal_morphism({T.module_generators[0]: RC.module_
↪generators[0]})
sage: TS = [x.value for x in T.subcrystal(max_depth=4)]
sage: all(Psi(b) == RC(b) for b in TS) # long time # indirect doctest
True
```

```
>>> from sage.all import *
>>> vct = CartanType(['B',Integer(4)]).as_folding()
>>> RC = crystals.infinity.RiggedConfigurations(vct)
>>> T = crystals.infinity.Tableaux(['B',Integer(4)])
>>> Psi = T.crystal_morphism({T.module_generators[Integer(0)]: RC.module_
↪generators[Integer(0)]})
>>> TS = [x.value for x in T.subcrystal(max_depth=Integer(4))]
>>> all(Psi(b) == RC(b) for b in TS) # long time # indirect doctest
True
```

class sage.combinat.rigged_configurations.bij_infinity.**MLTToRCBijectionTypeD**(*tp_krt*)

Bases: *KRTToRCBijectionTypeD*

run()

Run the bijection from a marginally large tableaux to a rigged configuration.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['D',4])
sage: T = crystals.infinity.Tableaux(['D',4])
sage: Psi = T.crystal_morphism({T.module_generators[0]: RC.module_
↪generators[0]})
sage: TS = [x.value for x in T.subcrystal(max_depth=4)]
sage: all(Psi(b) == RC(b) for b in TS) # long time # indirect doctest
True
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['D',Integer(4)])
>>> T = crystals.infinity.Tableaux(['D',Integer(4)])
>>> Psi = T.crystal_morphism({T.module_generators[Integer(0)]: RC.module_
↪generators[Integer(0)]})
>>> TS = [x.value for x in T.subcrystal(max_depth=Integer(4))]
>>> all(Psi(b) == RC(b) for b in TS) # long time # indirect doctest
True
```

class sage.combinat.rigged_configurations.bij_infinity.**RCToMLTBijectionTypeB**(*RC_element*)

Bases: *RCToKRTBijectionTypeB*

run()

Run the bijection from rigged configurations to a marginally large tableau.

EXAMPLES:

```
sage: vct = CartanType(['B',4]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: T = crystals.infinity.Tableaux(['B',4])
sage: Psi = RC.crystal_morphism({RC.module_generators[0]: T.module_
↪generators[0]})
sage: RCS = [x.value for x in RC.subcrystal(max_depth=4)]
sage: all(Psi(nu) == T(nu) for nu in RCS) # long time # indirect doctest
True
```

```
>>> from sage.all import *
>>> vct = CartanType(['B',Integer(4)]).as_folding()
>>> RC = crystals.infinity.RiggedConfigurations(vct)
>>> T = crystals.infinity.Tableaux(['B',Integer(4)])
>>> Psi = RC.crystal_morphism({RC.module_generators[Integer(0)]: T.module_
↪generators[Integer(0)]})
>>> RCS = [x.value for x in RC.subcrystal(max_depth=Integer(4))]
>>> all(Psi(nu) == T(nu) for nu in RCS) # long time # indirect doctest
True
```

class sage.combinat.rigged_configurations.bij_infinity.**RCToMLTBijectionTypeD**(*RC_element*)

Bases: *RCToKRTBijectionTypeD*

run()

Run the bijection from rigged configurations to a marginally large tableau.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['D',4])
sage: T = crystals.infinity.Tableaux(['D',4])
sage: Psi = RC.crystal_morphism({RC.module_generators[0]: T.module_
↪generators[0]})
sage: RCS = [x.value for x in RC.subcrystal(max_depth=4)]
sage: all(Psi(nu) == T(nu) for nu in RCS) # long time # indirect doctest
True
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['D',Integer(4)])
>>> T = crystals.infinity.Tableaux(['D',Integer(4)])
>>> Psi = RC.crystal_morphism({RC.module_generators[Integer(0)]: T.module_
↪generators[Integer(0)]})
>>> RCS = [x.value for x in RC.subcrystal(max_depth=Integer(4))]
>>> all(Psi(nu) == T(nu) for nu in RCS) # long time # indirect doctest
True
```

5.1.200 Bijection classes for type $A_n^{(1)}$

Part of the (internal) classes which run the bijection between rigged configurations and tensor products of Kirillov-Reshetikhin tableaux of type $A_n^{(1)}$.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version

class sage.combinat.rigged_configurations.bij_type_A.**KRTToRCBijectionTypeA**(*tp_krt*)

Bases: *KRTToRCBijectionAbstract*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $A_n^{(1)}$.

next_state(*val*)

Build the next state for type $A_n^{(1)}$.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1],␣
↪[[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import␣
↪KRTToRCBijectionTypeA
sage: bijection = KRTToRCBijectionTypeA(KRT(pathlist=[[4,3]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [3])
sage: bijection.next_state(3)
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',␣
↪Integer(4), Integer(1)], [[Integer(2),Integer(1)]])
>>> from sage.combinat.rigged_configurations.bij_type_A import␣
```

(continues on next page)

(continued from previous page)

```
↪KRTToRCBijectionTypeA
>>> bijection = KRTToRCBijectionTypeA(KRT(pathlist=[[Integer(4),Integer(3)]]))
>>> bijection.cur_path.insert(Integer(0), [])
>>> bijection.cur_dims.insert(Integer(0), [Integer(0), Integer(1)])
>>> bijection.cur_path[Integer(0)].insert(Integer(0), [Integer(3)])
>>> bijection.next_state(Integer(3))
```

class sage.combinat.rigged_configurations.bij_type_A.**RCToKRTBijectionTypeA**(*RC_element*)

Bases: *RCToKRTBijectionAbstract*

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $A_n^{(1)}$.

next_state(*height*)

Build the next state for type $A_n^{(1)}$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import␣
↪RCToKRTBijectionTypeA
sage: bijection = RCToKRTBijectionTypeA(RC(partition_list=[[1],[1],[1],[1]]))
sage: bijection.next_state(1)
5
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),␣
↪Integer(1)]])
>>> from sage.combinat.rigged_configurations.bij_type_A import␣
↪RCToKRTBijectionTypeA
>>> bijection = RCToKRTBijectionTypeA(RC(partition_list=[[Integer(1)],
↪[Integer(1)],[Integer(1)],[Integer(1)]]))
>>> bijection.next_state(Integer(1))
5
```

5.1.201 Bijection classes for type $A_{2n}^{(2)\dagger}$

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $A_{2n}^{(2)\dagger}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

class sage.combinat.rigged_configurations.bij_type_A2_dual.**KRTToRCBijectionTypeA2Dual**(*tp_krt*)

Bases: *KRTToRCBijectionTypeC*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $A_{2n}^{(2)\dagger}$.

This inherits from type $C_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $A_{2n}^{(2)\dagger}$.

class sage.combinat.rigged_configurations.bij_type_A2_dual.**RCToKRTBijectionTypeA2Dual**(*RC_element*)

Bases: *RCToKRTBijectionTypeC*

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $A_{2n}^{(2)\dagger}$.

next_state(*height*)

Build the next state for type $A_{2n}^{(2)\dagger}$.

5.1.202 Bijection classes for type $A_{2n}^{(2)}$

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $A_{2n}^{(2)}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

class sage.combinat.rigged_configurations.bij_type_A2_even.**KRTToRCBijectionTypeA2Even**(*tp_krt*)

Bases: *KRTToRCBijectionTypeC*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $A_{2n}^{(2)}$.

This inherits from type $C_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $A_{2n}^{(2)}$.

class sage.combinat.rigged_configurations.bij_type_A2_even.**RCToKRTBijectionTypeA2Even**(*RC_element*)

Bases: *RCToKRTBijectionTypeC*

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $A_{2n}^{(2)}$.

next_state(*height*)

Build the next state for type $A_{2n}^{(2)}$.

5.1.203 Bijection classes for type $A_{2n-1}^{(2)}$.

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $A_{2n-1}^{(2)}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

class sage.combinat.rigged_configurations.bij_type_A2_odd.**KRTToRCBijectionTypeA2Odd**(*tp_krt*)

Bases: *KRTToRCBijectionTypeA*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $A_{2n-1}^{(2)}$.

This inherits from type $A_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $A_{2n-1}^{(2)}$.

class sage.combinat.rigged_configurations.bij_type_A2_odd.**RCToKRTBijectionTypeA2Odd**(*RC_element*)

Bases: *RCToKRTBijectionTypeA*

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $A_{2n-1}^{(2)}$.

next_state(*height*)

Build the next state for type $A_{2n-1}^{(2)}$.

5.1.204 Bijection classes for type $B_n^{(1)}$

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $B_n^{(1)}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

class sage.combinat.rigged_configurations.bij_type_B.**KRTToRCBijectionTypeB**(*tp_krt*)

Bases: *KRTToRCBijectionTypeC*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $B_n^{(1)}$.

next_state(*val*)

Build the next state for type $B_n^{(1)}$.

other_outcome(*rc*, *pos_val*, *width_n*)

Do the other case (QS) possibility.

This arises from the ambiguity when we found a singular string at the max width in $\nu^{(n)}$. We had first attempted case (S), and if that resulted in an invalid rigged configuration, we now finish the bijection using case (QS).

EXAMPLES:

```
sage: RC = RiggedConfigurations(['B',3,1], [[2,1],[1,2]])
sage: rc = RC(partition_list=[[2,1], [2,1,1], [5,1]])
sage: t = rc.to_tensor_product_of_kirillov_reshetikhin_tableaux()
sage: t.to_rigged_configuration() == rc # indirect doctest
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['B',Integer(3),Integer(1)], [[Integer(2),
→Integer(1)],[Integer(1),Integer(2)]])
>>> rc = RC(partition_list=[[Integer(2),Integer(1)], [Integer(2),Integer(1),
→Integer(1)], [Integer(5),Integer(1)]])
>>> t = rc.to_tensor_product_of_kirillov_reshetikhin_tableaux()
>>> t.to_rigged_configuration() == rc # indirect doctest
True
```

run(*verbose=False*)

Run the bijection from a tensor product of KR tableaux to a rigged configuration.

INPUT:

- tp_krt – a tensor product of KR tableaux

- `verbose` – (default: `False`) display each step in the bijection

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.bij_type_B import␣
↪KRTToRCBijectionTypeB
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['B', 3, 1],␣
↪[[2, 1]])
sage: KRTToRCBijectionTypeB(KRT(pathlist=[[0,3]])).run()

0[ ]0

-1[ ]-1
-1[ ]-1

0[]0

sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['B', 3, 1],␣
↪[[3, 1]])
sage: KRTToRCBijectionTypeB(KRT(pathlist=[[-2,3,1]])).run()

(/)

-1[ ]-1

0[]0
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.bij_type_B import␣
↪KRTToRCBijectionTypeB
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['B',␣
↪Integer(3), Integer(1)], [[Integer(2), Integer(1)]])
>>> KRTToRCBijectionTypeB(KRT(pathlist=[[Integer(0),Integer(3)]])).run()
<BLANKLINE>
0[ ]0
<BLANKLINE>
-1[ ]-1
-1[ ]-1
<BLANKLINE>
0[]0
<BLANKLINE>
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['B',␣
↪Integer(3), Integer(1)], [[Integer(3), Integer(1)]])
>>> KRTToRCBijectionTypeB(KRT(pathlist=[[-Integer(2),Integer(3),
↪Integer(1)]])).run()
<BLANKLINE>
(/)
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
0[]0
<BLANKLINE>
```

class `sage.combinat.rigged_configurations.bij_type_B.`**`RCToKRTBijectionTypeB`**(*RC_element*)

Bases: `RCToKRTBijectionTypeC`

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $B_n^{(1)}$.

next_state(*height*)

Build the next state for type $B_n^{(1)}$.

run(*verbose=False*, *build_graph=False*)

Run the bijection from rigged configurations to tensor product of KR tableaux for type $B_n^{(1)}$.

INPUT:

- `verbose` – boolean (default: `False`); display each step in the bijection
- `build_graph` – boolean (default: `False`); build the graph of each step of the bijection

EXAMPLES:

```
sage: RC = RiggedConfigurations(['B', 3, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_B import
↪RCToKRTBijectionTypeB
sage: RCToKRTBijectionTypeB(RC(partition_list=[[1],[1,1],[1]])).run()
[[3], [0]]

sage: RC = RiggedConfigurations(['B', 3, 1], [[3, 1]])
sage: x = RC(partition_list=[[],[1],[1]])
sage: RCToKRTBijectionTypeB(x).run()
[[1], [3], [-2]]
sage: bij = RCToKRTBijectionTypeB(x)
sage: bij.run(build_graph=True)
[[1], [3], [-2]]
sage: bij._graph
Digraph on 6 vertices
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['B', Integer(3), Integer(1)], [[Integer(2),
↪Integer(1)]])
>>> from sage.combinat.rigged_configurations.bij_type_B import
↪RCToKRTBijectionTypeB
>>> RCToKRTBijectionTypeB(RC(partition_list=[[Integer(1)],[Integer(1),
↪Integer(1)],[Integer(1)]])).run()
[[3], [0]]

>>> RC = RiggedConfigurations(['B', Integer(3), Integer(1)], [[Integer(3),
↪Integer(1)]])
>>> x = RC(partition_list=[[],[Integer(1)],[Integer(1)]])
>>> RCToKRTBijectionTypeB(x).run()
[[1], [3], [-2]]
>>> bij = RCToKRTBijectionTypeB(x)
>>> bij.run(build_graph=True)
[[1], [3], [-2]]
>>> bij._graph
Digraph on 6 vertices
```

5.1.205 Bijection classes for type $C_n^{(1)}$

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $C_n^{(1)}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

class sage.combinat.rigged_configurations.bij_type_C.**KRTToRCBijectionTypeC**(*tp_krt*)

Bases: *KRTToRCBijectionTypeA*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $C_n^{(1)}$.

This inherits from type $A_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $C_n^{(1)}$.

class sage.combinat.rigged_configurations.bij_type_C.**RCToKRTBijectionTypeC**(*RC_element*)

Bases: *RCToKRTBijectionTypeA*

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $C_n^{(1)}$.

next_state(*height*)

Build the next state for type $C_n^{(1)}$.

5.1.206 Bijection classes for type $D_n^{(1)}$

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $D_n^{(1)}$.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version

class sage.combinat.rigged_configurations.bij_type_D.**KRTToRCBijectionTypeD**(*tp_krt*)

Bases: *KRTToRCBijectionTypeA*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $D_n^{(1)}$.

This inherits from type $A_n^{(1)}$ because we use the same methods in some places.

doubling_map()

Perform the doubling map of the rigged configuration at the current state of the bijection.

This is the map $B(\Lambda) \hookrightarrow B(2\Lambda)$ which doubles each of the rigged partitions and updates the vacancy numbers accordingly.

halving_map()

Perform the halving map of the rigged configuration at the current state of the bijection.

This is the inverse map to $B(\Lambda) \hookrightarrow B(2\Lambda)$ which halves each of the rigged partitions and updates the vacancy numbers accordingly.

next_state(*val*)

Build the next state for type $D_n^{(1)}$.

run(*verbose=False*)

Run the bijection from a tensor product of KR tableaux to a rigged configuration for type $D_n^{(1)}$.

INPUT:

- `tp_krt` – a tensor product of KR tableaux
- `verbose` – (default: `False`) display each step in the bijection

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1],
↪[[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import
↪KRTToRCBijectionTypeD
sage: KRTToRCBijectionTypeD(KRT(pathlist=[[-3,2]])).run()

-1[ ]-1

2[ ]2

-1[ ]-1

-1[ ]-1
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',
↪Integer(4), Integer(1)], [[Integer(2),Integer(1)]])
>>> from sage.combinat.rigged_configurations.bij_type_D import
↪KRTToRCBijectionTypeD
>>> KRTToRCBijectionTypeD(KRT(pathlist=[[-Integer(3),Integer(2)]])).run()
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
2[ ]2
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
```

class sage.combinat.rigged_configurations.bij_type_D.**RCToKRTBijectionTypeD**(*RC_element*)

Bases: *RCToKRTBijectionTypeA*

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $D_n^{(1)}$.

doubling_map()

Perform the doubling map of the rigged configuration at the current state of the bijection.

This is the map $B(\Lambda) \hookrightarrow B(2\Lambda)$ which doubles each of the rigged partitions and updates the vacancy numbers accordingly.

halving_map()

Perform the halving map of the rigged configuration at the current state of the bijection.

This is the inverse map to $B(\Lambda) \hookrightarrow B(2\Lambda)$ which halves each of the rigged partitions and updates the vacancy numbers accordingly.

next_state(*height*)

Build the next state for type $D_n^{(1)}$.

run(*verbose=False*, *build_graph=False*)

Run the bijection from rigged configurations to tensor product of KR tableaux for type $D_n^{(1)}$.

INPUT:

- `verbose` – boolean (default: `False`); display each step in the bijection
- `build_graph` – boolean (default: `False`); build the graph of each step of the bijection

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 1]])
sage: x = RC(partition_list=[[1],[1],[1],[1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import␣
↪RCToKRTBijectionTypeD
sage: RCToKRTBijectionTypeD(x).run()
[[2], [-3]]
sage: bij = RCToKRTBijectionTypeD(x)
sage: bij.run(build_graph=True)
[[2], [-3]]
sage: bij._graph
Digraph on 3 vertices
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),␣
↪Integer(1)]])
>>> x = RC(partition_list=[[Integer(1)],[Integer(1)],[Integer(1)],
↪[Integer(1)]])
>>> from sage.combinat.rigged_configurations.bij_type_D import␣
↪RCToKRTBijectionTypeD
>>> RCToKRTBijectionTypeD(x).run()
[[2], [-3]]
>>> bij = RCToKRTBijectionTypeD(x)
>>> bij.run(build_graph=True)
[[2], [-3]]
>>> bij._graph
Digraph on 3 vertices
```

5.1.207 Bijection classes for type $D_{n+1}^{(2)}$

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $D_{n+1}^{(2)}$.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version

class sage.combinat.rigged_configurations.bij_type_D_twisted.**KRTToRCBijectionTypeDTwisted**(*tp_krt*)

Bases: *KRTToRCBijectionTypeD*, *KRTToRCBijectionTypeA2Even*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $D_{n+1}^{(2)}$.

This inherits from type $C_n^{(1)}$ and $D_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $D_{n+1}^{(2)}$.

run(*verbose=False*)

Run the bijection from a tensor product of KR tableaux to a rigged configuration for type $D_{n+1}^{(2)}$.

INPUT:

- `tp_krt` – a tensor product of KR tableaux
- `verbose` – (default: `False`) display each step in the bijection

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 2],
↪[[3,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_twisted import
↪KRTToRCBijectionTypeDTwisted
sage: KRTToRCBijectionTypeDTwisted(KRT(pathlist=[[-1,3,2]])).run()

-1[ ]-1

0[ ]0

1[ ]1
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',
↪Integer(4), Integer(2)], [[Integer(3),Integer(1)]])
>>> from sage.combinat.rigged_configurations.bij_type_D_twisted import
↪KRTToRCBijectionTypeDTwisted
>>> KRTToRCBijectionTypeDTwisted(KRT(pathlist=[[-Integer(1),Integer(3),
↪Integer(2)]])).run()
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
0[ ]0
<BLANKLINE>
1[ ]1
<BLANKLINE>
```

class sage.combinat.rigged_configurations.bij_type_D_twisted.**RCToKRTBijectionTypeDTwisted**(*RC_element*)

Bases: *RCToKRTBijectionTypeD*, *RCToKRTBijectionTypeA2Even*

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $D_{n+1}^{(2)}$.

next_state(*height*)

Build the next state for type $D_{n+1}^{(2)}$.

run(*verbose=False*, *build_graph=False*)

Run the bijection from rigged configurations to tensor product of KR tableaux for type $D_{n+1}^{(2)}$.

INPUT:

- `verbose` – boolean (default: `False`); display each step in the bijection
- `build_graph` – boolean (default: `False`); build the graph of each step of the bijection

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 2], [[3, 1]])
sage: x = RC(partition_list=[[],[1],[1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_twisted import␣
↪RCToKRTBijectionTypeDTwisted
sage: RCToKRTBijectionTypeDTwisted(x).run()
[[1], [3], [-2]]
sage: bij = RCToKRTBijectionTypeDTwisted(x)
sage: bij.run(build_graph=True)
[[1], [3], [-2]]
sage: bij._graph
Digraph on 6 vertices
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(2)], [[Integer(3),␣
↪Integer(1)]])
>>> x = RC(partition_list=[[],[Integer(1)],[Integer(1)]])
>>> from sage.combinat.rigged_configurations.bij_type_D_twisted import␣
↪RCToKRTBijectionTypeDTwisted
>>> RCToKRTBijectionTypeDTwisted(x).run()
[[1], [3], [-2]]
>>> bij = RCToKRTBijectionTypeDTwisted(x)
>>> bij.run(build_graph=True)
[[1], [3], [-2]]
>>> bij._graph
Digraph on 6 vertices
```

5.1.208 Bijection classes for type $D_4^{(3)}$

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $D_4^{(3)}$.

AUTHORS:

- Travis Scrimshaw (2014-09-10): Initial version

class `sage.combinat.rigged_configurations.bij_type_D_tri.`**`KRTToRCBijectionTypeDTri`**(*tp_krt*)

Bases: *KRTToRCBijectionTypeA*

Specific implementation of the bijection from KR tableaux to rigged configurations for type $D_4^{(3)}$.

This inherits from type $A_n^{(1)}$ because we use the same methods in some places.

`next_state`(*val*)

Build the next state for type $D_4^{(3)}$.

class `sage.combinat.rigged_configurations.bij_type_D_tri.`**`RCToKRTBijectionTypeDTri`**(*RC_element*)

Bases: *RCToKRTBijectionTypeA*

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $D_4^{(3)}$.

next_state(*height*)

Build the next state for type $D_4^{(3)}$.

5.1.209 Bijection between rigged configurations and KR tableaux

Functions which are big switch statements to create the bijection class of the correct type.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version
- Travis Scrimshaw (2012-12-21): Added all non-exceptional bijection types
- Travis Scrimshaw (2014-09-10): Added type $D_4^{(3)}$

sage.combinat.rigged_configurations.bijection.**KRTToRCBijection**(*tp_krt*)

Return the correct KR tableaux to rigged configuration bijection helper class.

sage.combinat.rigged_configurations.bijection.**RCToKRTBijection**(*rigged_configuration_elt*)

Return the correct rigged configuration to KR tableaux bijection helper class.

5.1.210 Kleber Trees

A Kleber tree is a tree of weights generated by Kleber's algorithm [Kleber1]. The nodes correspond to the weights in the positive Weyl chamber obtained by subtracting a (nonzero) positive root. The edges are labeled by the coefficients of the roots of the difference.

AUTHORS:

- Travis Scrimshaw (2011-05-03): Initial version
- Travis Scrimshaw (2013-02-13): Added support for virtual trees and improved LaTeX output

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KleberTree(['A', 3, 1], [[3,2], [2,1], [1,1], [1,1]])
Kleber tree of Cartan type ['A', 3, 1] and B = ((3, 2), (2, 1), (1, 1), (1, 1))
sage: KleberTree(['D', 4, 1], [[2,2]])
Kleber tree of Cartan type ['D', 4, 1] and B = ((2, 2),)
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KleberTree(['A', Integer(3), Integer(1)], [[Integer(3),Integer(2)], [Integer(2),
→Integer(1)], [Integer(1),Integer(1)], [Integer(1),Integer(1)]])
Kleber tree of Cartan type ['A', 3, 1] and B = ((3, 2), (2, 1), (1, 1), (1, 1))
>>> KleberTree(['D', Integer(4), Integer(1)], [[Integer(2),Integer(2)]])
Kleber tree of Cartan type ['D', 4, 1] and B = ((2, 2),)
```

class sage.combinat.rigged_configurations.kleber_tree.**KleberTree**(*cartan_type*, *B*, *classical_ct*)

Bases: UniqueRepresentation, Parent

The tree that is generated by Kleber's algorithm.

A Kleber tree is a tree of weights generated by Kleber's algorithm [Kleber1]. It is used to generate the set of all admissible rigged configurations for the simply-laced affine types $A_n^{(1)}$, $D_n^{(1)}$, $E_6^{(1)}$, $E_7^{(1)}$, and $E_8^{(1)}$.

See also

There is a modified version for non-simply-laced affine types at *VirtualKleberTree*.

The nodes correspond to the weights in the positive Weyl chamber obtained by subtracting a (nonzero) positive root. The edges are labeled by the coefficients of the roots, and X is a child of Y if Y is the root else if the edge label of Y to its parent Z is greater (in every component) than the label from X to Y.

For a Kleber tree, one needs to specify an affine (simply-laced) Cartan type and a sequence of pairs (r, s), where s is any positive integer and r is a node in the Dynkin diagram. Each (r, s) can be viewed as a rectangle of width s and height r.

INPUT:

- `cartan_type` – an affine simply-laced Cartan type
- `B` – list of dimensions of rectangles by $[r, c]$ where r is the number of rows and c is the number of columns

REFERENCES:

EXAMPLES:

Simply-laced example:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[3,2], [1,1]])
sage: KT.list()
[Kleber tree node with weight [1, 0, 2] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 1] and upwards edge root [1, 1, 1]]
sage: KT = KleberTree(['A', 3, 1], [[3,2], [2,1], [1,1], [1,1]])
sage: KT.cardinality()
10
sage: KT = KleberTree(['D', 4, 1], [[2,2]])
sage: KT.cardinality()
3
sage: KT = KleberTree(['D', 4, 1], [[4,5]])
sage: KT.cardinality()
1
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KT = KleberTree(['A', Integer(3), Integer(1)], [[Integer(3),Integer(2)],
↪[Integer(1),Integer(1)]])
>>> KT.list()
[Kleber tree node with weight [1, 0, 2] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 1] and upwards edge root [1, 1, 1]]
>>> KT = KleberTree(['A', Integer(3), Integer(1)], [[Integer(3),Integer(2)],
↪[Integer(2),Integer(1)], [Integer(1),Integer(1)], [Integer(1),Integer(1)]])
>>> KT.cardinality()
10
>>> KT = KleberTree(['D', Integer(4), Integer(1)], [[Integer(2),Integer(2)]])
>>> KT.cardinality()
3
>>> KT = KleberTree(['D', Integer(4), Integer(1)], [[Integer(4),Integer(5)]])
>>> KT.cardinality()
1
```

From [Kleber2]:

```
sage: KT = KleberTree(['E', 6, 1], [[4, 2]])  # long time (9s on sage.math, 2012)
sage: KT.cardinality()  # long time
12
```

```
>>> from sage.all import *
>>> KT = KleberTree(['E', Integer(6), Integer(1)], [[Integer(4), Integer(2)]])  #␣
↪long time (9s on sage.math, 2012)
>>> KT.cardinality()  # long time
12
```

We check that relabelled types work (Issue #16876):

```
sage: ct = CartanType(['A',3,1]).relabel(lambda x: x+2)
sage: kt = KleberTree(ct, [[3,1],[5,1]])
sage: list(kt)
[Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 1, 1]]
sage: kt = KleberTree(['A',3,1], [[1,1],[3,1]])
sage: list(kt)
[Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 1, 1]]
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(3),Integer(1)]).relabel(lambda x: x+Integer(2))
>>> kt = KleberTree(ct, [[Integer(3),Integer(1)],[Integer(5),Integer(1)]])
>>> list(kt)
[Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 1, 1]]
>>> kt = KleberTree(['A',Integer(3),Integer(1)], [[Integer(1),Integer(1)],
↪[Integer(3),Integer(1)]])
>>> list(kt)
[Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 1, 1]]
```

Element

alias of *KleberTreeNode*

breadth_first_iter()

Iterate over all nodes in the tree following a breadth-first traversal.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[2, 2], [2, 3]])
sage: for x in KT.breadth_first_iter(): x
Kleber tree node with weight [0, 5, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 3, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 3, 0] and upwards edge root [1, 2, 1]
Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 1, 0]
Kleber tree node with weight [1, 1, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KT = KleberTree(['A', Integer(3), Integer(1)], [[Integer(2), Integer(2)],
→[Integer(2), Integer(3)]])
>>> for x in KT.breadth_first_iter(): x
Kleber tree node with weight [0, 5, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 3, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 3, 0] and upwards edge root [1, 2, 1]
Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 1, 0]
Kleber tree node with weight [1, 1, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 2, 1]
```

cartan_type()

Return the Cartan type of this Kleber tree.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[1,1]])
sage: KT.cartan_type()
['A', 3, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KT = KleberTree(['A', Integer(3), Integer(1)], [[Integer(1),Integer(1)]])
>>> KT.cartan_type()
['A', 3, 1]
```

depth_first_iter()

Iterate (recursively) over the nodes in the tree following a depth-first traversal.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[2, 2], [2, 3]])
sage: for x in KT.depth_first_iter(): x
Kleber tree node with weight [0, 5, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 3, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 3, 0] and upwards edge root [1, 2, 1]
Kleber tree node with weight [1, 1, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KT = KleberTree(['A', Integer(3), Integer(1)], [[Integer(2), Integer(2)],
→[Integer(2), Integer(3)]])
>>> for x in KT.depth_first_iter(): x
Kleber tree node with weight [0, 5, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 3, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 3, 0] and upwards edge root [1, 2, 1]
```

(continues on next page)

(continued from previous page)

```
Kleber tree node with weight [1, 1, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 2, 1]
```

digraph()

Return a DiGraph representation of this Kleber tree.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['D', 4, 1], [[2, 2]])
sage: KT.digraph()
Digraph on 3 vertices
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KT = KleberTree(['D', Integer(4), Integer(1)], [[Integer(2), Integer(2)]])
>>> KT.digraph()
Digraph on 3 vertices
```

latex_options(***options*)

Return the current latex options if no arguments are passed, otherwise set the corresponding latex option.

OPTIONS:

- `hspace` – (default: 2.5) the horizontal spacing of the tree nodes
- `vspace` – (default: `x`) the vertical spacing of the tree nodes, here `x` is the minimum of -2.5 or $-.75n$ where n is the rank of the classical type
- `edge_labels` – boolean (default: `True`); display edge labels
- `use_vector_notation` – boolean (default: `False`); display edge labels using vector notation instead of a linear combination

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['D', 3, 1], [[2,1], [2,1]])
sage: KT.latex_options(vspace=-4, use_vector_notation=True)
sage: sorted(KT.latex_options().items())
[('edge_labels', True), ('hspace', 2.5), ('use_vector_notation', True), (
↪'vspace', -4)]
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KT = KleberTree(['D', Integer(3), Integer(1)], [[Integer(2),Integer(1)],␣
↪[Integer(2),Integer(1)]])
>>> KT.latex_options(vspace=-Integer(4), use_vector_notation=True)
>>> sorted(KT.latex_options().items())
[('edge_labels', True), ('hspace', 2.5), ('use_vector_notation', True), (
↪'vspace', -4)]
```

plot(***options*)

Return the plot of `self` as a directed graph.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['D', 4, 1], [[2, 2]])
sage: print(KT.plot())                                                  #␣
↪needs sage.plot
Graphics object consisting of 8 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KT = KleberTree(['D', Integer(4), Integer(1)], [[Integer(2), Integer(2)]])
>>> print(KT.plot())                                                    #␣
↪needs sage.plot
Graphics object consisting of 8 graphics primitives
```

class sage.combinat.rigged_configurations.kleber_tree.**KleberTreeNode**(*parent_obj*, *node_weight*, *dominant_root*, *parent_node=None*)

Bases: Element

A node in the Kleber tree.

This class is meant to be used internally by the Kleber tree class and should not be created directly by the user.

For more on the Kleber tree and the nodes, see *KleberTree*.

The dominating root is the up_root which is the difference between the parent node's weight and this node's weight.

INPUT:

- parent_obj – the parent object of this element
- node_weight – the weight of this node
- dominant_root – the dominating root
- parent_node – (default: None) the parent node of this node

depth()

Return the depth of this node in the tree.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: RS = RootSystem(['A', 2])
sage: WS = RS.weight_lattice()
sage: R = RS.root_lattice()
sage: KT = KleberTree(['A', 2, 1], [[1,1]])
sage: n = KT(WS.sum_of_terms([(1,5), (2,2)]), R.zero())
sage: n.depth
0
sage: n2 = KT(WS.sum_of_terms([(1,5), (2,2)]), R.zero(), n)
sage: n2.depth
1
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
```

(continues on next page)

(continued from previous page)

```
>>> RS = RootSystem(['A', Integer(2)])
>>> WS = RS.weight_lattice()
>>> R = RS.root_lattice()
>>> KT = KleberTree(['A', Integer(2), Integer(1)], [[Integer(1),Integer(1)]])
>>> n = KT(WS.sum_of_terms([(Integer(1),Integer(5)), (Integer(2),
↪Integer(2))]), R.zero())
>>> n.depth
0
>>> n2 = KT(WS.sum_of_terms([(Integer(1),Integer(5)), (Integer(2),
↪Integer(2))]), R.zero(), n)
>>> n2.depth
1
```

multiplicity()

Return the multiplicity of `self`.

The multiplicity of a node x of depth d weight λ in a simply-laced Kleber tree is equal to:

$$\prod_{i>0}\prod_{a\in\overline{I}}\binom{p_i^{(a)}+m_i^{(a)}}{p_i^{(a)}}$$

Recall that

$$\begin{aligned} m_i^{(a)} &= \left(\lambda^{(i-1)} - 2\lambda^{(i)} + \lambda^{(i+1)} \mid \overline{\Lambda}_a\right), \\ p_i^{(a)} &= \left(\alpha_a \mid \lambda^{(i)}\right) - \sum_{j>i}(j-i)L_j^{(a)}, \end{aligned}$$

where $\lambda^{(i)}$ is the weight node at depth i in the path to x from the root and we set $\lambda^{(j)} = \lambda$ for all $j \geq d$.

Note that $m_i^{(a)} = 0$ for all $i > d$.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A',3,1], [[3,2],[2,1],[1,1],[1,1]])
sage: for x in KT: x, x.multiplicity()
(Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 0, 0], 1)
(Kleber tree node with weight [3, 0, 1] and upwards edge root [0, 1, 1], 1)
(Kleber tree node with weight [0, 2, 2] and upwards edge root [1, 0, 0], 1)
(Kleber tree node with weight [1, 0, 3] and upwards edge root [1, 1, 0], 2)
(Kleber tree node with weight [1, 1, 1] and upwards edge root [1, 1, 1], 4)
(Kleber tree node with weight [0, 0, 2] and upwards edge root [2, 2, 1], 2)
(Kleber tree node with weight [2, 0, 0] and upwards edge root [0, 1, 1], 2)
(Kleber tree node with weight [0, 0, 2] and upwards edge root [1, 1, 0], 1)
(Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 1, 1], 2)
(Kleber tree node with weight [0, 1, 0] and upwards edge root [0, 0, 1], 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import KleberTree
>>> KT = KleberTree(['A',Integer(3),Integer(1)], [[Integer(3),Integer(2)],
↪[Integer(2),Integer(1)],[Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> for x in KT: x, x.multiplicity()
```

(continues on next page)

(continued from previous page)

```
(Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 0, 0], 1)
(Kleber tree node with weight [3, 0, 1] and upwards edge root [0, 1, 1], 1)
(Kleber tree node with weight [0, 2, 2] and upwards edge root [1, 0, 0], 1)
(Kleber tree node with weight [1, 0, 3] and upwards edge root [1, 1, 0], 2)
(Kleber tree node with weight [1, 1, 1] and upwards edge root [1, 1, 1], 4)
(Kleber tree node with weight [0, 0, 2] and upwards edge root [2, 2, 1], 2)
(Kleber tree node with weight [2, 0, 0] and upwards edge root [0, 1, 1], 2)
(Kleber tree node with weight [0, 0, 2] and upwards edge root [1, 1, 0], 1)
(Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 1, 1], 2)
(Kleber tree node with weight [0, 1, 0] and upwards edge root [0, 0, 1], 1)
```

class sage.combinat.rigged_configurations.kleber_tree.**KleberTreeTypeA2Even**(*cartan_type*, *B*)

Bases: *VirtualKleberTree*

Kleber tree for types $A_{2n}^{(2)}$ and $A_{2n}^{(2)\dagger}$.

Note that here for $A_{2n}^{(2)}$ we use $\tilde{\gamma}_a$ in place of γ_a in constructing the virtual Kleber tree, and so we end up selecting all nodes since $\tilde{\gamma}_a = 1$ for all $a \in \overline{I}$. For type $A_{2n}^{(2)\dagger}$, we have $\gamma_a = 1$ for all $a \in \overline{I}$.

See also

VirtualKleberTree

breadth_first_iter(*all_nodes=False*)

Iterate over all nodes in the tree following a breadth-first traversal.

INPUT:

- `all_nodes` – boolean (default: `False`); if `True`, output all nodes in the tree

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
sage: KT = VirtualKleberTree(['A', 4, 2], [[2,1]])
sage: for x in KT.breadth_first_iter(): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
sage: for x in KT.breadth_first_iter(True): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
>>> KT = VirtualKleberTree(['A', Integer(4), Integer(2)], [[Integer(2), Integer(1)]])
>>> for x in KT.breadth_first_iter(): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> for x in KT.breadth_first_iter(True): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
```

depth_first_iter(*all_nodes=False*)

Iterate (recursively) over the nodes in the tree following a depth-first traversal.

INPUT:

- `all_nodes` – boolean (default: `False`); if `True`, output all nodes in the tree

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
sage: KT = VirtualKleberTree(['A', 4, 2], [[2,1]])
sage: for x in KT.depth_first_iter(): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
sage: for x in KT.depth_first_iter(True): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
>>> KT = VirtualKleberTree(['A', Integer(4), Integer(2)], [[Integer(2),
↪Integer(1)]])
>>> for x in KT.depth_first_iter(): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
>>> for x in KT.depth_first_iter(True): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
```

class sage.combinat.rigged_configurations.kleber_tree.**VirtualKleberTree**(*cartan_type*, *B*)

Bases: *KleberTree*

A virtual Kleber tree.

We can use a modified version of the Kleber algorithm called the virtual Kleber algorithm [OSS03] to compute all admissible rigged configurations for non-simply-laced types. This uses the following embeddings into the simply-laced types:

$$\begin{aligned} C_n^{(1)}, A_{2n}^{(2)}, A_{2n}^{(2)\dagger}, D_{n+1}^{(2)} &\hookrightarrow A_{2n-1}^{(1)} \\ A_{2n-1}^{(2)}, B_n^{(1)} &\hookrightarrow D_{n+1}^{(1)} \\ E_6^{(2)}, F_4^{(1)} &\hookrightarrow E_6^{(1)} \\ D_4^{(3)}, G_2^{(1)} &\hookrightarrow D_4^{(1)} \end{aligned}$$

One then selects the subset of admissible nodes which are translates of the virtual requirements. In the graph, the selected nodes are indicated by brackets [].

Note: Because these are virtual nodes, all information is given in the corresponding simply-laced type.

See also:

For more on the Kleber algorithm, see `KleberTree`.

REFERENCES:

INPUT:

- `cartan_type` – an affine non-simply-laced Cartan type
- `B` – list of dimensions of rectangles by $[r, c]$ where r is the number of rows and c is the number of columns

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
sage: KT = VirtualKleberTree(['C', 4, 1], [[2,2]])
sage: KT.cardinality()
3
sage: KT.base_tree().cardinality()
6
sage: KT = VirtualKleberTree(['C', 4, 1], [[4,5]])
sage: KT.cardinality()
1
sage: KT = VirtualKleberTree(['D', 5, 2], [[2,1], [1,1]])
sage: KT.cardinality()
8
sage: KT = VirtualKleberTree(CartanType(['A', 4, 2]).dual(), [[1,1], [2,2]])
sage: KT.cardinality()
15
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
>>> KT = VirtualKleberTree(['C', Integer(4), Integer(1)], [[Integer(2),
↪Integer(2)]])
>>> KT.cardinality()
3
>>> KT.base_tree().cardinality()
6
>>> KT = VirtualKleberTree(['C', Integer(4), Integer(1)], [[Integer(4),
↪Integer(5)]])
>>> KT.cardinality()
1
>>> KT = VirtualKleberTree(['D', Integer(5), Integer(2)], [[Integer(2),
↪Integer(1)], [Integer(1),Integer(1)]])
>>> KT.cardinality()
```

(continues on next page)

(continued from previous page)

```
8
>>> KT = VirtualKleberTree(CartanType(['A', Integer(4), Integer(2)]).dual(),␣
↪[[Integer(1),Integer(1)], [Integer(2),Integer(2)]])
>>> KT.cardinality()
15
```

base_tree()

Return the underlying virtual Kleber tree associated to `self`.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
sage: KT = VirtualKleberTree(['C', 4, 1], [[2,2]])
sage: KT.base_tree()
Kleber tree of Cartan type ['A', 7, 1] and B = ((2, 2), (6, 2))
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
>>> KT = VirtualKleberTree(['C', Integer(4), Integer(1)], [[Integer(2),
↪Integer(2)]])
>>> KT.base_tree()
Kleber tree of Cartan type ['A', 7, 1] and B = ((2, 2), (6, 2))
```

breadth_first_iter(*all_nodes=False*)

Iterate over all nodes in the tree following a breadth-first traversal.

INPUT:

- `all_nodes` – boolean (default: `False`); if `True`, output all nodes in the tree

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
sage: KT = VirtualKleberTree(['C', 2, 1], [[1,1], [2,1]])
sage: for x in KT.breadth_first_iter(): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
sage: for x in KT.breadth_first_iter(True): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [0, 2, 0] and upwards edge root [1, 1, 1]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
>>> KT = VirtualKleberTree(['C', Integer(2), Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(2),Integer(1)]])
>>> for x in KT.breadth_first_iter(): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
```

(continues on next page)

(continued from previous page)

```
>>> for x in KT.breadth_first_iter(True): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [0, 2, 0] and upwards edge root [1, 1, 1]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
```

depth_first_iter(*all_nodes=False*)

Iterate (recursively) over the nodes in the tree following a depth-first traversal.

INPUT:

- `all_nodes` – boolean (default: `False`); if `True`, output all nodes in the tree

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
sage: KT = VirtualKleberTree(['C', 2, 1], [[1,1], [2,1]])
sage: for x in KT.depth_first_iter(): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
sage: for x in KT.depth_first_iter(True): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [0, 2, 0] and upwards edge root [1, 1, 1]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.rigged_configurations.kleber_tree import␣
↪VirtualKleberTree
>>> KT = VirtualKleberTree(['C', Integer(2), Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(2),Integer(1)]])
>>> for x in KT.depth_first_iter(): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
>>> for x in KT.depth_first_iter(True): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [0, 2, 0] and upwards edge root [1, 1, 1]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
```

5.1.211 Kirillov-Reshetikhin Tableaux

Kirillov-Reshetikhin tableaux are rectangular tableaux with r rows and s columns that naturally arise under the bijection between rigged configurations and tableaux [RigConBijection]. They are in bijection with the elements of the Kirillov-Reshetikhin crystal $B^{r,s}$ under the (inverse) filling map [OSS13] [SS2015]. They do not have to satisfy the semistandard row or column restrictions. These tensor products are the result from the bijection from rigged configurations [RigConBijection].

For more information, see *KirillovReshetikhinTableaux* and *TensorProductOfKirillovReshetikhinTableaux*.

AUTHORS:

- Travis Scrimshaw (2012-01-03): Initial version
- Travis Scrimshaw (2012-11-14): Added bijection to KR crystals

REFERENCES:

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxBn**(*cartan_type*, *r*, *s*)

Bases: *KRTableauxTypeHorizonal*

Kirillov-Reshetkhin tableaux $B^{n,s}$ of type $B_n^{(1)}$.

Element

alias of *KRTableauxSpinElement*

from_kirillov_reshetikhin_crystal(*krc*)

Construct an element of `self` from the Kirillov-Reshetikhin crystal element `krc`.

EXAMPLES:

```
sage: KR = crystals.KirillovReshetikhin(['B',3,1], 3, 3, model='KR')
sage: C = crystals.KirillovReshetikhin(['B',3,1], 3, 3, model='KN')
sage: krc = C.module_generators[1].f_string([3,2,3,1,3,3]); krc
[++-, [[2], [0], [-3]]]
sage: KR.from_kirillov_reshetikhin_crystal(krc)
[[1, 1, 2], [2, 2, -3], [-3, -3, -1]]
```

```
>>> from sage.all import *
>>> KR = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)], Integer(3),
→ Integer(3), model='KR')
>>> C = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)], Integer(3),␣
→Integer(3), model='KN')
>>> krc = C.module_generators[Integer(1)].f_string([Integer(3),Integer(2),
→Integer(3),Integer(1),Integer(3),Integer(3)]); krc
[++-, [[2], [0], [-3]]]
>>> KR.from_kirillov_reshetikhin_crystal(krc)
[[1, 1, 2], [2, 2, -3], [-3, -3, -1]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxDTwistedSpin**(*cartan_type*, *r*, *s*)

Bases: *KRTableauxRectangle*

Kirillov-Reshetikhin tableaux $B^{r,s}$ of type $D_n^{(2)}$ with $r = n$.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 2], 1, 1, model='KR')
sage: KRT.cardinality()
8
sage: KRC = crystals.KirillovReshetikhin(['D', 4, 2], 1, 1, model='KN')
sage: KRT.cardinality() == KRC.cardinality()
True
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)], Integer(1),␣
→Integer(1), model='KR')
>>> KRT.cardinality()
8
>>> KRC = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)], Integer(1),␣
→Integer(1), model='KN')
>>> KRT.cardinality() == KRC.cardinality()
True
```

Element

alias of *KRTableauxSpinElement*

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxRectangle**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinTableaux*

Kirillov-Reshetkhin tableaux $B^{r,s}$ whose module generator is a single $r \times s$ rectangle.

These are Kirillov-Reshetkhin tableaux $B^{r,s}$ of type:

- $A_n^{(1)}$ for all $1 \leq r \leq n$,
- $C_n^{(1)}$ when $r = n$.

from_kirillov_reshetikhin_crystal(*krc*)

Construct a *KirillovReshetikhinTableauxElement*.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: C = crystals.KirillovReshetikhin(['A',4,1], 2, 1, model='KN')
sage: krc = C(4,3); krc
[[3], [4]]
sage: KRT.from_kirillov_reshetikhin_crystal(krc)
[[3], [4]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)],␣
↪Integer(2), Integer(1), model='KR')
>>> C = crystals.KirillovReshetikhin(['A',Integer(4),Integer(1)], Integer(2),␣
↪Integer(1), model='KN')
>>> krc = C(Integer(4),Integer(3)); krc
[[3], [4]]
>>> KRT.from_kirillov_reshetikhin_crystal(krc)
[[3], [4]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxSpin**(*cartan_type*, *r*, *s*)

Bases: *KRTableauxRectangle*

Kirillov-Reshetikhin tableaux $B^{r,s}$ of type $D_n^{(1)}$ with $r = n, n-1$.

Element

alias of *KRTableauxSpinElement*

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxSpinElement**(*parent*, *list*, ***options*)

Bases: *KirillovReshetikhinTableauxElement*

Kirillov-Reshetikhin tableau for spinors.

Here we are in the embedding $B(\Lambda_n) \hookrightarrow B(2\Lambda_n)$, so e_i and f_i act by e_i^2 and f_i^2 respectively for all $i \neq 0$. We do this so our columns are full width (as opposed to half width and/or uses a $\pm$ representation).

classical_weight()

Return the classical weight of `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 4, 1, model='KR')
sage: KRT.module_generators[0].classical_weight()
(1/2, 1/2, 1/2, 1/2)
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(4), Integer(1), model='KR')
>>> KRT.module_generators[Integer(0)].classical_weight()
(1/2, 1/2, 1/2, 1/2)
```

e(*i*)

Calculate the action of e_i on `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 1, model='KR')
sage: KRT(-1, -4, 3, 2).e(1)
[[1], [3], [-4], [-2]]
sage: KRT(-1, -4, 3, 2).e(3)
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(4), Integer(1), model='KR')
>>> KRT(-Integer(1), -Integer(4), Integer(3), Integer(2)).e(Integer(1))
[[1], [3], [-4], [-2]]
>>> KRT(-Integer(1), -Integer(4), Integer(3), Integer(2)).e(Integer(3))
```

epsilon(*i*)

Compute ε_i of `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 1, model='KR')
sage: KRT(-1, -4, 3, 2).epsilon(1)
1
sage: KRT(-1, -4, 3, 2).epsilon(3)
0
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(4), Integer(1), model='KR')
>>> KRT(-Integer(1), -Integer(4), Integer(3), Integer(2)).epsilon(Integer(1))
1
>>> KRT(-Integer(1), -Integer(4), Integer(3), Integer(2)).epsilon(Integer(3))
0
```

f(*i*)

Calculate the action of f_i on `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 1, model='KR')
sage: KRT(-1, -4, 3, 2).f(1)
```

(continues on next page)

(continued from previous page)

```
sage: KRT(-1, -4, 3, 2).f(3)
[[2], [4], [-3], [-1]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(4), Integer(1), model='KR')
>>> KRT(-Integer(1), -Integer(4), Integer(3), Integer(2)).f(Integer(1))
>>> KRT(-Integer(1), -Integer(4), Integer(3), Integer(2)).f(Integer(3))
[[2], [4], [-3], [-1]]
```

left_split()

Return the image of `self` under the left column splitting map.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 4, 3, model='KR')
sage: elt = KRT(-3,-4,2,1,-3,-4,2,1,-2,-4,3,1); elt.pp()
  1  1  1
  2  2  3
 -4 -4 -4
 -3 -3 -2
sage: elt.left_split().pp()
  1 (X)   1  1
  2       2  3
 -4      -4 -4
 -3      -3 -2
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(4), Integer(3), model='KR')
>>> elt = KRT(-Integer(3),-Integer(4),Integer(2),Integer(1),-Integer(3),-
↪Integer(4),Integer(2),Integer(1),-Integer(2),-Integer(4),Integer(3),
↪Integer(1)); elt.pp()
  1  1  1
  2  2  3
 -4 -4 -4
 -3 -3 -2
>>> elt.left_split().pp()
  1 (X)   1  1
  2       2  3
 -4      -4 -4
 -3      -3 -2
```

phi(i)

Compute φ_i of `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 1, model='KR')
sage: KRT(-1, -4, 3, 2).phi(1)
0
sage: KRT(-1, -4, 3, 2).phi(3)
1
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(4), Integer(1), model='KR')
>>> KRT(-Integer(1), -Integer(4), Integer(3), Integer(2)).phi(Integer(1))
0
>>> KRT(-Integer(1), -Integer(4), Integer(3), Integer(2)).phi(Integer(3))
1
```

to_array(*rows=True*)

Return a 2-dimensional array representation of this Kirillov-Reshetikhin element.

If the output is in rows, then it outputs the top row first (in the English convention) from left to right.

For example: if the reading word is $[2, 1, 4, 3]$, so as a 2×2 tableau:

```
1 3
2 4
```

we output `[[1, 3], [2, 4]]`.

If the output is in columns, then it outputs the leftmost column first with the bottom element first. In other words this parses the reading word into its columns.

Continuing with the previous example, the output would be `[[2, 1], [4, 3]]`.

INPUT:

- `rows` – boolean (default: `True`); set to `True` if the resulting array is by row, otherwise it is by column

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 4, 3, model='KR')
sage: elt = KRT(-3,-4,2,1,-3,-4,2,1,-2,-4,3,1)
sage: elt.to_array()
[[1, 1, 1], [2, 2, 3], [-4, -4, -4], [-3, -3, -2]]
sage: elt.to_array(False)
[[-3, -4, 2, 1], [-3, -4, 2, 1], [-2, -4, 3, 1]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(4), Integer(3), model='KR')
>>> elt = KRT(-Integer(3),-Integer(4),Integer(2),Integer(1),-Integer(3),-
↪Integer(4),Integer(2),Integer(1),-Integer(2),-Integer(4),Integer(3),
↪Integer(1))
>>> elt.to_array()
[[1, 1, 1], [2, 2, 3], [-4, -4, -4], [-3, -3, -2]]
>>> elt.to_array(False)
[[-3, -4, 2, 1], [-3, -4, 2, 1], [-2, -4, 3, 1]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeBox**(*cartan_type*, *r*, *s*)

Bases: *KRTableauxTypeVertical*

Kirillov-Reshetikhin tableaux $B^{r,s}$ of type:

- $A_{2n}^{(2)}$ for all $r \leq n$,
- $D_{n+1}^{(2)}$ for all $r < n$,

- $D_4^{(3)}$ for $r = 1$.

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeFromRC**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinTableaux*

Kirillov-Reshetikhin tableaux $B^{r,s}$ constructed from rigged configurations under the bijection Φ.

Warning

The Kashiwara-Nakashima version is not implemented due to the non-trivial multiplicities of classical components, so classical_decomposition() does not work.

Element

alias of *KRTableauxTypeFromRCElement*

module_generators()

The module generators of self.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 1, model='KR')
sage: KRT.module_generators
([[1], [2]], [[1], [0]], [[1], [E]], [[E], [E]])
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],␣
↪Integer(2), Integer(1), model='KR')
>>> KRT.module_generators
([[1], [2]], [[1], [0]], [[1], [E]], [[E], [E]])
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeFromRCElement**(*parent*, *list*, ***options*)

Bases: *KirillovReshetikhinTableauxElement*

A Kirillov-Reshetikhin tableau constructed from rigged configurations under the bijection Φ.

e(*i*)

Perform the action of e_i on self.

Todo

Implement a direct action of e_0 without moving to rigged configurations.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 1, model='KR')
sage: KRT.module_generators[0].e(0)
[[2], [E]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],
↪Integer(2), Integer(1), model='KR')
>>> KRT.module_generators[Integer(0)].e(Integer(0))
[[2], [E]]
```

epsilon(*i*)

Compute ε_i of `self`.

Todo

Implement a direct action of ϵ_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 2, model='KR')
sage: KRT.module_generators[0].epsilon(0)
6
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],
↪Integer(2), Integer(2), model='KR')
>>> KRT.module_generators[Integer(0)].epsilon(Integer(0))
6
```

f(*i*)

Perform the action of f_i on `self`.

Todo

Implement a direct action of f_0 without moving to rigged configurations.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 1, model='KR')
sage: KRT.module_generators[0].f(0)
sage: KRT.module_generators[3].f(0)
[[1], [0]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],
↪Integer(2), Integer(1), model='KR')
>>> KRT.module_generators[Integer(0)].f(Integer(0))
>>> KRT.module_generators[Integer(3)].f(Integer(0))
[[1], [0]]
```

phi(*i*)

Compute φ_i of `self`.

Todo

Compute ϕ_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 2, model='KR')
sage: KRT.module_generators[0].phi(0)
0
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)],␣
↪Integer(2), Integer(2), model='KR')
>>> KRT.module_generators[Integer(0)].phi(Integer(0))
0
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeHorizonal**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinTableaux*

Kirillov-Reshetikhin tableaux $B^{r,s}$ of type:

- $C_n^{(1)}$ for $1 \leq r < n$,
- $A_{2n}^{(2)\dagger}$ for $1 \leq r \leq n$.

from_kirillov_reshetikhin_crystal(*krc*)

Construct an element of `self` from the Kirillov-Reshetikhin crystal element `krc`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['C',4,1], 2, 3, model='KR')
sage: C = crystals.KirillovReshetikhin(['C',4,1], 2, 3, model='KN')
sage: krc = C(4,3); krc
[[3], [4]]
sage: KRT.from_kirillov_reshetikhin_crystal(krc)
[[3, -2, 1], [4, -1, 2]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['C',Integer(4),Integer(1)],␣
↪Integer(2), Integer(3), model='KR')
>>> C = crystals.KirillovReshetikhin(['C',Integer(4),Integer(1)], Integer(2),␣
↪Integer(3), model='KN')
>>> krc = C(Integer(4),Integer(3)); krc
[[3], [4]]
>>> KRT.from_kirillov_reshetikhin_crystal(krc)
[[3, -2, 1], [4, -1, 2]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeVertical**(*cartan_type*, *r*, *s*)

Bases: *KirillovReshetikhinTableaux*

Kirillov-Reshetkihn tableaux $B^{r,s}$ of type:

- $D_n^{(1)}$ for all $1 \leq r < n-1$,

- $B_n^{(1)}$ for all $1 \leq r < n$,
- $A_{2n-1}^{(2)}$ for all $1 \leq r \leq n$.

from_kirillov_reshetikhin_crystal(*krc*)

Construct an element of `self` from the Kirillov-Reshetikhin crystal element `krc`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: C = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KN')
sage: krc = C(4,3); krc
[[3], [4]]
sage: KRT.from_kirillov_reshetikhin_crystal(krc)
[[3, -2, 1], [4, -1, 2]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(2), Integer(3), model='KR')
>>> C = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),␣
↪Integer(3), model='KN')
>>> krc = C(Integer(4),Integer(3)); krc
[[3], [4]]
>>> KRT.from_kirillov_reshetikhin_crystal(krc)
[[3, -2, 1], [4, -1, 2]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KirillovReshetikhinTableaux**(*cartan_type*, *r*, *s*)

Bases: `CrystalOfWords`

Kirillov-Reshetikhin tableaux.

Kirillov-Reshetikhin tableaux are rectangular tableaux with r rows and s columns that naturally arise under the bijection between rigged configurations and tableaux [RigConBijection]. They are in bijection with the elements of the Kirillov-Reshetikhin crystal $B^{r,s}$ under the (inverse) filling map.

Whenever $B^{r,s} \cong B(s\Lambda_r)$ as a classical crystal (which is the case for $B^{r,s}$ in type $A_n^{(1)}$, $B^{n,s}$ in type $C_n^{(1)}$ and $D_{n+1}^{(2)}$, $B^{n,s}$ and $B^{n-1,s}$ in type $D_n^{(1)}$) then the filling map is trivial.

For $B^{r,s}$ in:

- type $D_n^{(1)}$ when $r \leq n-2$,
- type $B_n^{(1)}$ when $r < n$,
- type $A_{2n-1}^{(2)}$ for all r,

the filling map is defined in [OSS2011].

For the spinor cases in type $D_n^{(1)}$, the crystal $B^{k,s}$ where $k = n-1, n$, is isomorphic as a classical crystal to $B(s\Lambda_k)$, and here we consider the Kirillov-Reshetikhin tableaux as living in $B(2s\Lambda_k)$ under the natural doubling map. In this case, the crystal operators e_i and f_i act as e_i^2 and f_i^2 respectively. See [BijectionDn].

For the spinor case in type $B_n^{(1)}$, the crystal $B^{n,s}$, we consider the images under the natural doubling map into $B^{n,2s}$. The classical components of this crystal are now given by removing 2×2 boxes. The filling map is the same as below (see the non-spin type $C_n^{(1)}$).

For $B^{r,s}$ in:

- type $C_n^{(1)}$ when $r < n$,
- type $A_{2n}^{(2)\dagger}$ for all r,

the filling map is given as follows. Suppose we are considering the (classically) highest weight element in the classical component $B(\lambda)$. Then we fill it in with the horizontal dominoes $[\bar{\imath}, i]$ in the i-th row from the top (in English notation) and reordering the columns so that they are increasing. Recall from above that $B^{n,s} \cong B(s\Lambda_n)$ in type $C_n^{(1)}$.

For $B^{r,s}$ in:

- type $A_{2n}^{(2)}$ for all r,
- type $D_{n+1}^{(2)}$ when $r < n$,
- type $D_4^{(3)}$ when $r = 1$,

the filling map is the same as given in [OSS2011] except for the rightmost column which is given by the column $[1, 2, \ldots, k, \emptyset, \ldots \emptyset]$ where $k = (r + x - 1)/2$ in Step 3 of [OSS2011].

For the spinor case in type $D_{n+1}^{(2)}$, the crystal $B^{n,s}$, we define the filling map in the same way as in type $D_n^{(1)}$.

Note: The filling map and classical decompositions in non-spinor cases can be classified by how the special node 0 connects with the corresponding classical diagram.

The classical crystal structure is given by the usual Kashiwara-Nakashima tableaux rules. That is to embed this into $B(\Lambda_1)^{\otimes ns}$ by using the reading word and then applying the classical crystal operator. The affine crystal structure is given by converting to the corresponding KR crystal element, performing the affine crystal operator, and pulling back to a KR tableau.

For more information about the bijection between rigged configurations and tensor products of Kirillov-Reshetikhin tableaux, see `TensorProductOfKirillovReshetikhinTableaux`.

Note: The tableaux for all non-simply-laced types are provably correct if the bijection with `rigged configurations` holds. Therefore this is currently only proven for $B^{r,1}$ or $B^{1,s}$ and in general for types $A_n^{(1)}$ and $D_n^{(1)}$.

INPUT:

- `cartan_type` – the Cartan type
- `r` – the Dynkin diagram index (typically the number of rows)
- `s` – the number of columns

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: elt = KRT(4, 3); elt
[[3], [4]]

sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 2, 1, model='KR')
sage: elt = KRT(-1, 1); elt
[[1], [-1]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)], Integer(2),␣
→Integer(1), model='KR')
>>> elt = KRT(Integer(4), Integer(3)); elt
[[3], [4]]

>>> KRT = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)], Integer(2),␣
→Integer(1), model='KR')
>>> elt = KRT(-Integer(1), Integer(1)); elt
[[1], [-1]]
```

We can create highest weight crystals from a given shape or weight:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR')
sage: KRT.module_generator(shape=[1,1])
[[1, 1], [2, -1]]
sage: KRT.module_generator(column_shape=[2])
[[1, 1], [2, -1]]
sage: WS = RootSystem(['D',4,1]).weight_space()
sage: KRT.module_generator(weight=WS.sum_of_terms([[0,-2],[2,1]]))
[[1, 1], [2, -1]]
sage: WSC = RootSystem(['D',4]).weight_space()
sage: KRT.module_generator(classical_weight=WSC.fundamental_weight(2))
[[1, 1], [2, -1]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)], Integer(2),␣
→Integer(2), model='KR')
>>> KRT.module_generator(shape=[Integer(1),Integer(1)])
[[1, 1], [2, -1]]
>>> KRT.module_generator(column_shape=[Integer(2)])
[[1, 1], [2, -1]]
>>> WS = RootSystem(['D',Integer(4),Integer(1)]).weight_space()
>>> KRT.module_generator(weight=WS.sum_of_terms([[Integer(0),-Integer(2)],
→[Integer(2),Integer(1)]]))
[[1, 1], [2, -1]]
>>> WSC = RootSystem(['D',Integer(4)]).weight_space()
>>> KRT.module_generator(classical_weight=WSC.fundamental_weight(Integer(2)))
[[1, 1], [2, -1]]
```

We can go between *KashiwaraNakashimaTableaux()* and *KirillovReshetikhinTableaux* elements:

```
sage: KRCrys = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR')
sage: elt = KRCrys(3, 2); elt
[[2], [3]]
sage: k = KRTab(elt); k
[[2, 1], [3, -1]]
sage: KRCrys(k)
[[2], [3]]
```

```
>>> from sage.all import *
>>> KRCrys = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(2), Integer(2), model='KN')
>>> KRTab = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> elt = KRCrys(Integer(3), Integer(2)); elt
[[2], [3]]
>>> k = KRTab(elt); k
[[2, 1], [3, -1]]
>>> KRCrys(k)
[[2], [3]]
```

We check that the classical weights in the classical decompositions agree in a few different type:

```
sage: KRCrys = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
sage: KRCrys = crystals.KirillovReshetikhin(['B', 3, 1], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['B', 3, 1], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
sage: KRCrys = crystals.KirillovReshetikhin(['C', 3, 1], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['C', 3, 1], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
sage: KRCrys = crystals.KirillovReshetikhin(['D', 4, 2], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['D', 4, 2], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
sage: KRCrys = crystals.KirillovReshetikhin(['A', 4, 2], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['A', 4, 2], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
```

```
>>> from sage.all import *
>>> KRCrys = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2), model='KN')
>>> KRTab = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
>>> KRCrys = crystals.KirillovReshetikhin(['B', Integer(3), Integer(1)],␣
↪Integer(2), Integer(2), model='KN')
>>> KRTab = crystals.KirillovReshetikhin(['B', Integer(3), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
>>> KRCrys = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)],␣
↪Integer(2), Integer(2), model='KN')
>>> KRTab = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
```

(continues on next page)

(continued from previous page)

```
>>> KRCrys = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)],␣
↪Integer(2), Integer(2), model='KN')
>>> KRTab = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)],␣
↪Integer(2), Integer(2), model='KR')
>>> all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
>>> KRCrys = crystals.KirillovReshetikhin(['A', Integer(4), Integer(2)],␣
↪Integer(2), Integer(2), model='KN')
>>> KRTab = crystals.KirillovReshetikhin(['A', Integer(4), Integer(2)],␣
↪Integer(2), Integer(2), model='KR')
>>> all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
```

Element

alias of *KirillovReshetikhinTableauxElement*

classical_decomposition()

Return the classical crystal decomposition of `self`.

EXAMPLES:

```
sage: crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR').classical_
↪decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[], [1, 1], [2, 2]]
```

```
>>> from sage.all import *
>>> crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)], Integer(2),␣
↪Integer(2), model='KR').classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[], [1, 1], [2, 2]]
```

from_kirillov_reshetikhin_crystal(*krc*)

Construct an element of `self` from the Kirillov-Reshetikhin crystal element `krc`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: C = crystals.KirillovReshetikhin(['A',4,1], 2, 1, model='KN')
sage: krc = C(4,3); krc
[[3], [4]]
sage: KRT.from_kirillov_reshetikhin_crystal(krc)
[[3], [4]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)],␣
↪Integer(2), Integer(1), model='KR')
>>> C = crystals.KirillovReshetikhin(['A',Integer(4),Integer(1)], Integer(2),␣
↪Integer(1), model='KN')
>>> krc = C(Integer(4),Integer(3)); krc
[[3], [4]]
>>> KRT.from_kirillov_reshetikhin_crystal(krc)
[[3], [4]]
```

kirillov_reshetikhin_crystal()

Return the corresponding KR crystal in the `Kashiwara-Nakashima model`.

EXAMPLES:

```
sage: crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR').kirillov_
↪reshetikhin_crystal()
Kirillov-Reshetikhin crystal of type ['A', 4, 1] with (r,s)=(2,1)
```

```
>>> from sage.all import *
>>> crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)], Integer(2),␣
↪Integer(1), model='KR').kirillov_reshetikhin_crystal()
Kirillov-Reshetikhin crystal of type ['A', 4, 1] with (r,s)=(2,1)
```

module_generator(*i=None*, ***options*)

Return the specified module generator.

INPUT:

- `i` – the index of the module generator

We can also get a module generator by using one of the following optional arguments:

- `shape` – the associated shape
- `column_shape` – the shape given as columns (a column of length k correspond to a classical weight ω_k)
- `weight` – the weight
- `classical_weight` – the classical weight

If no arguments are specified, then return the unique module generator of classical weight $s\Lambda_r$.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR')
sage: KRT.module_generator(1)
[[1, 1], [2, -1]]
sage: KRT.module_generator(shape=[1,1])
[[1, 1], [2, -1]]
sage: KRT.module_generator(column_shape=[2])
[[1, 1], [2, -1]]
sage: WS = RootSystem(['D',4,1]).weight_space()
sage: KRT.module_generator(weight=WS.sum_of_terms([[0,-2],[2,1]]))
[[1, 1], [2, -1]]
sage: WSC = RootSystem(['D',4]).weight_space()
sage: KRT.module_generator(classical_weight=WSC.fundamental_weight(2))
[[1, 1], [2, -1]]
sage: KRT.module_generator()
[[1, 1], [2, 2]]

sage: KRT = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: KRT.module_generator()
[[1, 1], [2, 2]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
```

(continues on next page)

(continued from previous page)

```
>>> KRT.module_generator(Integer(1))
[[1, 1], [2, -1]]
>>> KRT.module_generator(shape=[Integer(1),Integer(1)])
[[1, 1], [2, -1]]
>>> KRT.module_generator(column_shape=[Integer(2)])
[[1, 1], [2, -1]]
>>> WS = RootSystem(['D',Integer(4),Integer(1)]).weight_space()
>>> KRT.module_generator(weight=WS.sum_of_terms([[Integer(0),-Integer(2)],
↪[Integer(2),Integer(1)]]))
[[1, 1], [2, -1]]
>>> WSC = RootSystem(['D',Integer(4)]).weight_space()
>>> KRT.module_generator(classical_weight=WSC.fundamental_weight(Integer(2)))
[[1, 1], [2, -1]]
>>> KRT.module_generator()
[[1, 1], [2, 2]]

>>> KRT = crystals.KirillovReshetikhin(['A', Integer(3), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> KRT.module_generator()
[[1, 1], [2, 2]]
```

r()

Return the value r for this tableaux class which corresponds to the number of rows.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: KRT.r()
2
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)],␣
↪Integer(2), Integer(1), model='KR')
>>> KRT.r()
2
```

s()

Return the value s for this tableaux class which corresponds to the number of columns.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: KRT.s()
1
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)],␣
↪Integer(2), Integer(1), model='KR')
>>> KRT.s()
1
```

tensor(**crystals*, ***options*)

Return the tensor product of `self` with `crystals`.

If `crystals` is a list of (a tensor product of) KR tableaux, this returns a *TensorProductOfKirillovReshetikhinTableaux*.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: TP = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 3, 1],
→[[1,3],[3,1]])
sage: K.tensor(TP, K)
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1]
 and factor(s) ((2, 2), (1, 3), (3, 1), (2, 2))

sage: C = crystals.KirillovReshetikhin(['A',3,1], 3, 1, model='KN')
sage: K.tensor(K, C)
Full tensor product of the crystals
 [Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1)]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A', Integer(3), Integer(1)],
→Integer(2), Integer(2), model='KR')
>>> TP = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', Integer(3),
→ Integer(1)], [[Integer(1),Integer(3)],[Integer(3),Integer(1)]])
>>> K.tensor(TP, K)
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1]
 and factor(s) ((2, 2), (1, 3), (3, 1), (2, 2))

>>> C = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(3),
→Integer(1), model='KN')
>>> K.tensor(K, C)
Full tensor product of the crystals
 [Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1)]
```

class `sage.combinat.rigged_configurations.kr_tableaux.`**`KirillovReshetikhinTableauxElement`**(*parent*, *list*, ***options*)

Bases: *TensorProductOfRegularCrystalsElement*

A Kirillov-Reshetikhin tableau.

For more information, see *KirillovReshetikhinTableaux* and *TensorProductOfKirillovReshetikhinTableaux*.

`classical_weight()`

Return the classical weight of `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: elt = KRT(3,2,-1,1); elt
```

(continues on next page)

(continued from previous page)

```
[[2, 1], [3, -1]]
sage: elt.classical_weight()
(0, 1, 1, 0)
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],
↪Integer(2), Integer(2), model='KR')
>>> elt = KRT(Integer(3),Integer(2),-Integer(1),Integer(1)); elt
[[2, 1], [3, -1]]
>>> elt.classical_weight()
(0, 1, 1, 0)
```

e(i)

Perform the action of e_i on `self`.

Todo

Implement a direct action of e_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KRT.module_generators[0].e(0)
[[-2, 1], [-1, -1]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],
↪Integer(2), Integer(2), model='KR')
>>> KRT.module_generators[Integer(0)].e(Integer(0))
[[-2, 1], [-1, -1]]
```

epsilon(i)

Compute ε_i of `self`.

Todo

Implement a direct action of ε_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KRT.module_generators[0].epsilon(0)
2
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],
↪Integer(2), Integer(2), model='KR')
>>> KRT.module_generators[Integer(0)].epsilon(Integer(0))
2
```

f(*i*)

Perform the action of f_i on `self`.

Todo

Implement a direct action of f_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KRT.module_generators[0].f(0)
[[1, 1], [2, -1]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> KRT.module_generators[Integer(0)].f(Integer(0))
[[1, 1], [2, -1]]
```

left_split()

Return the image of `self` under the left column splitting map.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: mg = KRT.module_generators[1]; mg.pp()
  1 -2  1
  2 -1  2
sage: ls = mg.left_split(); ls.pp()
  1 (X)  -2  1
  2      -1  2
sage: ls.parent()
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and␣
↪factor(s) ((2, 1), (2, 2))
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(2), Integer(3), model='KR')
>>> mg = KRT.module_generators[Integer(1)]; mg.pp()
  1 -2  1
  2 -1  2
>>> ls = mg.left_split(); ls.pp()
  1 (X)  -2  1
  2      -1  2
>>> ls.parent()
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and␣
↪factor(s) ((2, 1), (2, 2))
```

phi(*i*)

Compute φ_i of `self`.

Todo

Compute φ_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KRT.module_generators[0].phi(0)
2
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> KRT.module_generators[Integer(0)].phi(Integer(0))
2
```

pp()

Pretty print self.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 2, model='KR')
sage: elt = KRT(2, 1, 4, 3); elt
[[1, 3], [2, 4]]
sage: elt.pp()
1  3
2  4
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> elt = KRT(Integer(2), Integer(1), Integer(4), Integer(3)); elt
[[1, 3], [2, 4]]
>>> elt.pp()
1  3
2  4
```

right_split()

Return the image of self under the right column splitting map.

Let $*$ denote the Lusztig involution, and ls as the *left splitting map*. The right splitting map is defined as $\mathrm{rs} := * \circ \mathrm{ls} \circ *$.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: mg = KRT.module_generators[1]; mg.pp()
  1 -2  1
  2 -1  2
sage: ls = mg.right_split(); ls.pp()
 -2  1 (X)    1
 -1  2        2
sage: ls.parent()
```

(continues on next page)

(continued from previous page)

```
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and␣
↪factor(s) ((2, 2), (2, 1))
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(2), Integer(3), model='KR')
>>> mg = KRT.module_generators[Integer(1)]; mg.pp()
  1 -2  1
  2 -1  2
>>> ls = mg.right_split(); ls.pp()
 -2  1 (X)   1
 -1  2       2
>>> ls.parent()
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and␣
↪factor(s) ((2, 2), (2, 1))
```

to_array(*rows=True*)

Return a 2-dimensional array representation of this Kirillov-Reshetikhin element.

If the output is in rows, then it outputs the top row first (in the English convention) from left to right.

For example: if the reading word is $[2, 1, 4, 3]$, so as a 2×2 tableau:

```
1 3
2 4
```

we output `[[1, 3], [2, 4]]`.

If the output is in columns, then it outputs the leftmost column first with the bottom element first. In other words this parses the reading word into its columns.

Continuing with the previous example, the output would be `[[2, 1], [4, 3]]`.

INPUT:

- `rows` – boolean (default: `True`); set to `True` if the resulting array is by row, otherwise it is by column

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 2, model='KR')
sage: elt = KRT(2, 1, 4, 3)
sage: elt.to_array()
[[1, 3], [2, 4]]
sage: elt.to_array(False)
[[2, 1], [4, 3]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> elt = KRT(Integer(2), Integer(1), Integer(4), Integer(3))
>>> elt.to_array()
[[1, 3], [2, 4]]
>>> elt.to_array(False)
[[2, 1], [4, 3]]
```

to_classical_highest_weight(*index_set=None*)

Return the classical highest weight element corresponding to `self`.

INPUT:

- `index_set` – (default: `None`) return the highest weight with respect to the index set; if `None` is passed in, then this uses the classical index set

OUTPUT:

A pair `[H, f_str]` where `H` is the highest weight element and `f_str` is a list of a_i of f_{a_i} needed to reach `H`.

EXAMPLES:

```
sage: KRTab = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: elt = KRTab(3,2,-1,1); elt
[[2, 1], [3, -1]]
sage: elt.to_classical_highest_weight()
[[[1, 1], [2, -1]], [1, 2]]
```

```
>>> from sage.all import *
>>> KRTab = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> elt = KRTab(Integer(3),Integer(2),-Integer(1),Integer(1)); elt
[[2, 1], [3, -1]]
>>> elt.to_classical_highest_weight()
[[[1, 1], [2, -1]], [1, 2]]
```

to_kirillov_reshetikhin_crystal()

Construct a `KashiwaraNakashimaTableaux()` element from `self`.

We construct the Kirillov-Reshetikhin crystal element as follows:

1. Determine the shape λ of the KR crystal from the weight.
2. Determine a path $e_{i_1} e_{i_2} \cdots e_{i_k}$ to the highest weight.
3. Apply $f_{i_k} \cdots f_{i_2} f_{i_1}$ to a highest weight KR crystal of shape λ.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: elt = KRT(3,2,-1,1); elt
[[2, 1], [3, -1]]
sage: elt.to_kirillov_reshetikhin_crystal()
[[2], [3]]
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> elt = KRT(Integer(3),Integer(2),-Integer(1),Integer(1)); elt
[[2, 1], [3, -1]]
>>> elt.to_kirillov_reshetikhin_crystal()
[[2], [3]]
```

to_tableau()

Return a *Tableau* object of `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 2, model='KR')
sage: elt = KRT(2, 1, 4, 3); elt
[[1, 3], [2, 4]]
sage: t = elt.to_tableau(); t
[[1, 3], [2, 4]]
sage: type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
```

```
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['A', Integer(4), Integer(1)],␣
↪Integer(2), Integer(2), model='KR')
>>> elt = KRT(Integer(2), Integer(1), Integer(4), Integer(3)); elt
[[1, 3], [2, 4]]
>>> t = elt.to_tableau(); t
[[1, 3], [2, 4]]
>>> type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: KR = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KR.module_generators[1].weight()
-2*Lambda[0] + Lambda[2]
```

```
>>> from sage.all import *
>>> KR = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)], Integer(2),
↪ Integer(2), model='KR')
>>> KR.module_generators[Integer(1)].weight()
-2*Lambda[0] + Lambda[2]
```

5.1.212 Crystal of Rigged Configurations

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

We only consider the highest weight crystal structure, not the Kirillov-Reshetikhin structure, and we extend this to symmetrizable types.

class sage.combinat.rigged_configurations.rc_crystal.**CrystalOfNonSimplyLacedRC**(*vct*, *wt*, *WLR*)

Bases: *CrystalOfRiggedConfigurations*

Highest weight crystal of rigged configurations in non-simply-laced type.

Element

alias of *RCHWNonSimplyLacedElement*

from_virtual(*vrc*)

Convert `vrc` in the virtual crystal into a rigged configuration of the original Cartan type.

INPUT:

- `vrc` – a virtual rigged configuration

EXAMPLES:

```
sage: La = RootSystem(['C', 3]).weight_lattice().fundamental_weights()
sage: vct = CartanType(['C', 3]).as_folding()
sage: RC = crystals.RiggedConfigurations(vct, La[2])
sage: elt = RC(partition_list=[[0], [1], [1]])
sage: elt == RC.from_virtual(RC.to_virtual(elt))
True
```

```
>>> from sage.all import *
>>> La = RootSystem(['C', Integer(3)]).weight_lattice().fundamental_weights()
>>> vct = CartanType(['C', Integer(3)]).as_folding()
>>> RC = crystals.RiggedConfigurations(vct, La[Integer(2)])
>>> elt = RC(partition_list=[[Integer(0)], [Integer(1)], [Integer(1)]])
>>> elt == RC.from_virtual(RC.to_virtual(elt))
True
```

to_virtual(*rc*)

Convert `rc` into a rigged configuration in the virtual crystal.

INPUT:

- `rc` – a rigged configuration element

EXAMPLES:

```
sage: La = RootSystem(['C', 3]).weight_lattice().fundamental_weights()
sage: vct = CartanType(['C', 3]).as_folding()
sage: RC = crystals.RiggedConfigurations(vct, La[2])
sage: elt = RC(partition_list=[[], [1], [1]]); elt

(/)

0[ ]0

-1[ ]-1

sage: RC.to_virtual(elt)

(/)

0[ ]0

-2[ ][ ]-2

0[ ]0

(/)
```

```
>>> from sage.all import *
>>> La = RootSystem(['C', Integer(3)]).weight_lattice().fundamental_weights()
>>> vct = CartanType(['C', Integer(3)]).as_folding()
>>> RC = crystals.RiggedConfigurations(vct, La[Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> elt = RC(partition_list=[[], [Integer(1)], [Integer(1)]]); elt
<BLANKLINE>
(/)
<BLANKLINE>
0[ ]0
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
>>> RC.to_virtual(elt)
<BLANKLINE>
(/)
<BLANKLINE>
0[ ]0
<BLANKLINE>
-2[ ][ ]-2
<BLANKLINE>
0[ ]0
<BLANKLINE>
(/)
<BLANKLINE>
```

virtual()

Return the corresponding virtual crystal.

EXAMPLES:

```
sage: La = RootSystem(['C', 2, 1]).weight_lattice().fundamental_weights()
sage: vct = CartanType(['C', 2, 1]).as_folding()
sage: RC = crystals.RiggedConfigurations(vct, La[0])
sage: RC
Crystal of rigged configurations of type ['C', 2, 1] and weight Lambda[0]
sage: RC.virtual
Crystal of rigged configurations of type ['A', 3, 1] and weight 2*Lambda[0]
```

```
>>> from sage.all import *
>>> La = RootSystem(['C', Integer(2), Integer(1)]).weight_lattice().
↪fundamental_weights()
>>> vct = CartanType(['C', Integer(2), Integer(1)]).as_folding()
>>> RC = crystals.RiggedConfigurations(vct, La[Integer(0)])
>>> RC
Crystal of rigged configurations of type ['C', 2, 1] and weight Lambda[0]
>>> RC.virtual
Crystal of rigged configurations of type ['A', 3, 1] and weight 2*Lambda[0]
```

class sage.combinat.rigged_configurations.rc_crystal.**CrystalOfRiggedConfigurations**(*wt*, *WLR*)

Bases: UniqueRepresentation, Parent

A highest weight crystal of rigged configurations.

The crystal structure for finite simply-laced types is given in [CrysStructSchilling06]. These were then shown to be the crystal operators in all finite types in [SS2015], all simply-laced and a large class of foldings of simply-laced types in [SS2015II], and all symmetrizable types (uniformly) in [SS2017].

INPUT:

- `cartan_type` – (optional) a Cartan type or a Cartan type given as a folding
- `wt` – the highest weight vector in the weight lattice

EXAMPLES:

For simplicity, we display the rigged configurations horizontally:

```
sage: RiggedConfigurations.options.display='horizontal'
```

```
>>> from sage.all import *
>>> RiggedConfigurations.options.display='horizontal'
```

We start with a simply-laced finite type:

```
sage: La = RootSystem(['A', 2]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[1] + La[2])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([1,2])
0[ ]0   0[ ]-1
sage: mg.f_string([1,2,2])
0[ ]0   -2[ ][ ]-2
sage: mg.f_string([1,2,2,2])
sage: mg.f_string([2,1,1,2])
-1[ ][ ]-1   -1[ ][ ]-1
sage: RC.cardinality()
8
sage: T = crystals.Tableaux(['A', 2], shape=[2,1])
sage: RC.digraph().is_isomorphic(T.digraph(), edge_labels=True)
True
```

```
>>> from sage.all import *
>>> La = RootSystem(['A', Integer(2)]).weight_lattice().fundamental_weights()
>>> RC = crystals.RiggedConfigurations(La[Integer(1)] + La[Integer(2)])
>>> mg = RC.highest_weight_vector()
>>> mg.f_string([Integer(1),Integer(2)])
0[ ]0   0[ ]-1
>>> mg.f_string([Integer(1),Integer(2),Integer(2)])
0[ ]0   -2[ ][ ]-2
>>> mg.f_string([Integer(1),Integer(2),Integer(2),Integer(2)])
>>> mg.f_string([Integer(2),Integer(1),Integer(1),Integer(2)])
-1[ ][ ]-1   -1[ ][ ]-1
>>> RC.cardinality()
8
>>> T = crystals.Tableaux(['A', Integer(2)], shape=[Integer(2),Integer(1)])
>>> RC.digraph().is_isomorphic(T.digraph(), edge_labels=True)
True
```

We construct a non-simply-laced affine type:

```
sage: La = RootSystem(['C', 3]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[2])
sage: mg = RC.highest_weight_vector()
```

(continues on next page)

(continued from previous page)

```
sage: mg.f_string([2,3])
(/)   1[ ]1   -1[ ]-1
sage: T = crystals.Tableaux(['C', 3], shape=[1,1])
sage: RC.digraph().is_isomorphic(T.digraph(), edge_labels=True)
True
```

```
>>> from sage.all import *
>>> La = RootSystem(['C', Integer(3)]).weight_lattice().fundamental_weights()
>>> RC = crystals.RiggedConfigurations(La[Integer(2)])
>>> mg = RC.highest_weight_vector()
>>> mg.f_string([Integer(2),Integer(3)])
(/)   1[ ]1   -1[ ]-1
>>> T = crystals.Tableaux(['C', Integer(3)], shape=[Integer(1),Integer(1)])
>>> RC.digraph().is_isomorphic(T.digraph(), edge_labels=True)
True
```

We can construct rigged configurations using a diagram folding of a simply-laced type. This yields an equivalent but distinct crystal:

```
sage: vct = CartanType(['C', 3]).as_folding()
sage: RC = crystals.RiggedConfigurations(vct, La[2])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([2,3])
(/)   0[ ]0   -1[ ]-1
sage: T = crystals.Tableaux(['C', 3], shape=[1,1])
sage: RC.digraph().is_isomorphic(T.digraph(), edge_labels=True)
True
```

```
>>> from sage.all import *
>>> vct = CartanType(['C', Integer(3)]).as_folding()
>>> RC = crystals.RiggedConfigurations(vct, La[Integer(2)])
>>> mg = RC.highest_weight_vector()
>>> mg.f_string([Integer(2),Integer(3)])
(/)   0[ ]0   -1[ ]-1
>>> T = crystals.Tableaux(['C', Integer(3)], shape=[Integer(1),Integer(1)])
>>> RC.digraph().is_isomorphic(T.digraph(), edge_labels=True)
True
```

We reset the global options:

```
sage: RiggedConfigurations.options._reset()
```

```
>>> from sage.all import *
>>> RiggedConfigurations.options._reset()
```

REFERENCES:

- [SS2015]
- [SS2015II]
- [SS2017]

Element

alias of *RCHighestWeightElement*

options = Current options for RiggedConfigurations - convention: English - display: vertical - element_ascii_art: True - half_width_boxes_type_B: True

weight_lattice_realization()

Return the weight lattice realization used to express the weights of elements in `self`.

EXAMPLES:

```
sage: La = RootSystem(['A', 2, 1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: RC = crystals.RiggedConfigurations(La[0])
sage: RC.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

```
>>> from sage.all import *
>>> La = RootSystem(['A', Integer(2), Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> RC = crystals.RiggedConfigurations(La[Integer(0)])
>>> RC.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

5.1.213 Rigged Configurations of $\mathcal{B}(\infty)$

AUTHORS:

- Travis Scrimshaw (2013-04-16): Initial version

class sage.combinat.rigged_configurations.rc_infinity.**InfinityCrystalOfNonSimplyLacedRC**(*vct*)

Bases: *InfinityCrystalOfRiggedConfigurations*

Rigged configurations for $\mathcal{B}(\infty)$ in non-simply-laced types.

class Element(*parent*, *rigged_partitions=[]*, ***options*)

Bases: *RCNonSimplyLacedElement*

A rigged configuration in $\mathcal{B}(\infty)$ in non-simply-laced types.

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: vct = CartanType(['C', 3]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: elt = RC(partition_list=[[1],[1,1],[1]], rigging_list=[[0],[-1,-1],
↪[0]])
sage: elt.weight()
(-1, -1, 0)

sage: vct = CartanType(['F', 4, 1]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: mg = RC.highest_weight_vector()
sage: elt = mg.f_string([1,0,3,4,2,2]); ascii_art(elt)
```

(continues on next page)

(continued from previous page)

```
-1[ ]-1   0[ ]1   -2[ ][ ]-2   0[ ]1   -1[ ]-1
sage: wt = elt.weight(); wt
-Lambda[0] + Lambda[1] - 2*Lambda[2] + 3*Lambda[3] - Lambda[4] - delta
sage: al = RC.weight_lattice_realization().simple_roots()
sage: wt == -(al[0] + al[1] + 2*al[2] + al[3] + al[4])
True
```

```
>>> from sage.all import *
>>> vct = CartanType(['C', Integer(3)]).as_folding()
>>> RC = crystals.infinity.RiggedConfigurations(vct)
>>> elt = RC(partition_list=[[Integer(1)],[Integer(1),Integer(1)],
↪[Integer(1)]], rigging_list=[[Integer(0)],[-Integer(1),-Integer(1)],
↪[Integer(0)]])
>>> elt.weight()
(-1, -1, 0)

>>> vct = CartanType(['F', Integer(4), Integer(1)]).as_folding()
>>> RC = crystals.infinity.RiggedConfigurations(vct)
>>> mg = RC.highest_weight_vector()
>>> elt = mg.f_string([Integer(1),Integer(0),Integer(3),Integer(4),
↪Integer(2),Integer(2)]); ascii_art(elt)
-1[ ]-1   0[ ]1   -2[ ][ ]-2   0[ ]1   -1[ ]-1
>>> wt = elt.weight(); wt
-Lambda[0] + Lambda[1] - 2*Lambda[2] + 3*Lambda[3] - Lambda[4] - delta
>>> al = RC.weight_lattice_realization().simple_roots()
>>> wt == -(al[Integer(0)] + al[Integer(1)] + Integer(2)*al[Integer(2)] +␣
↪al[Integer(3)] + al[Integer(4)])
True
```

from_virtual(*vrc*)

Convert `vrc` in the virtual crystal into a rigged configuration of the original Cartan type.

INPUT:

- `vrc` – a virtual rigged configuration

EXAMPLES:

```
sage: vct = CartanType(['C', 2]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: elt = RC(partition_list=[[3],[2]], rigging_list=[[-2],[0]])
sage: vrc_elt = RC.to_virtual(elt)
sage: ret = RC.from_virtual(vrc_elt); ret

-3[ ][ ][ ]-2

-1[ ][ ]0

sage: ret == elt
True
```

```
>>> from sage.all import *
>>> vct = CartanType(['C', Integer(2)]).as_folding()
```

(continues on next page)

(continued from previous page)

```
>>> RC = crystals.infinity.RiggedConfigurations(vct)
>>> elt = RC(partition_list=[[Integer(3)],[Integer(2)]], rigging_list=[[-
↪Integer(2)],[Integer(0)]])
>>> vrc_elt = RC.to_virtual(elt)
>>> ret = RC.from_virtual(vrc_elt); ret
<BLANKLINE>
-3[ ][ ][ ]-2
<BLANKLINE>
-1[ ][ ]0
<BLANKLINE>
>>> ret == elt
True
```

to_virtual(*rc*)

Convert rc into a rigged configuration in the virtual crystal.

INPUT:

- rc – a rigged configuration element

EXAMPLES:

```
sage: vct = CartanType(['C', 2]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: mg = RC.highest_weight_vector()
sage: elt = mg.f_string([1,2,2,1,1]); elt

-3[ ][ ][ ]-2

-1[ ][ ]0

sage: velt = RC.to_virtual(elt); velt

-3[ ][ ][ ]-2

-2[ ][ ][ ][ ]0

-3[ ][ ][ ]-2

sage: velt.parent()
The infinity crystal of rigged configurations of type ['A', 3]
```

```
>>> from sage.all import *
>>> vct = CartanType(['C', Integer(2)]).as_folding()
>>> RC = crystals.infinity.RiggedConfigurations(vct)
>>> mg = RC.highest_weight_vector()
>>> elt = mg.f_string([Integer(1),Integer(2),Integer(2),Integer(1),
↪Integer(1)]); elt
<BLANKLINE>
-3[ ][ ][ ]-2
<BLANKLINE>
-1[ ][ ]0
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
>>> velt = RC.to_virtual(elt); velt
<BLANKLINE>
-3[ ][ ][ ]-2
<BLANKLINE>
-2[ ][ ][ ][ ]0
<BLANKLINE>
-3[ ][ ][ ]-2
<BLANKLINE>
>>> velt.parent()
The infinity crystal of rigged configurations of type ['A', 3]
```

virtual()

Return the corresponding virtual crystal.

EXAMPLES:

```
sage: vct = CartanType(['C', 3]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: RC
The infinity crystal of rigged configurations of type ['C', 3]
sage: RC.virtual
The infinity crystal of rigged configurations of type ['A', 5]
```

```
>>> from sage.all import *
>>> vct = CartanType(['C', Integer(3)]).as_folding()
>>> RC = crystals.infinity.RiggedConfigurations(vct)
>>> RC
The infinity crystal of rigged configurations of type ['C', 3]
>>> RC.virtual
The infinity crystal of rigged configurations of type ['A', 5]
```

class sage.combinat.rigged_configurations.rc_infinity.**InfinityCrystalOfRiggedConfigurations**(*cartan_type*)

Bases: `UniqueRepresentation`, `Parent`

Rigged configuration model for $\mathcal{B}(\infty)$.

The crystal is generated by the empty rigged configuration with the same crystal structure given by the *highest weight model* except we remove the condition that the resulting rigged configuration needs to be valid when applying f_a.

INPUT:

- `cartan_type` – a Cartan type

EXAMPLES:

For simplicity, we display all of the rigged configurations horizontally:

```
sage: RiggedConfigurations.options(display='horizontal')
```

```
>>> from sage.all import *
>>> RiggedConfigurations.options(display='horizontal')
```

We begin with a simply-laced finite type:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3]); RC
The infinity crystal of rigged configurations of type ['A', 3]

sage: RC.options(display='horizontal')

sage: mg = RC.highest_weight_vector(); mg
(/)   (/)   (/)
sage: elt = mg.f_string([2,1,3,2]); elt
0[ ]0   -2[ ]-1   0[ ]0
        -2[ ]-1
sage: elt.e(1)
sage: elt.e(3)
sage: mg.f_string([2,1,3,2]).e(2)
-1[ ]-1  0[ ]1  -1[ ]-1
sage: mg.f_string([2,3,2,1,3,2])
0[ ]0  -3[ ][ ]-1  -1[ ][ ]-1
       -2[ ]-1
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['A', Integer(3)]); RC
The infinity crystal of rigged configurations of type ['A', 3]

>>> RC.options(display='horizontal')

>>> mg = RC.highest_weight_vector(); mg
(/)   (/)   (/)
>>> elt = mg.f_string([Integer(2),Integer(1),Integer(3),Integer(2)]); elt
0[ ]0   -2[ ]-1   0[ ]0
        -2[ ]-1
>>> elt.e(Integer(1))
>>> elt.e(Integer(3))
>>> mg.f_string([Integer(2),Integer(1),Integer(3),Integer(2)]).e(Integer(2))
-1[ ]-1  0[ ]1  -1[ ]-1
>>> mg.f_string([Integer(2),Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2)])
0[ ]0  -3[ ][ ]-1  -1[ ][ ]-1
       -2[ ]-1
```

Next we consider a non-simply-laced finite type:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C', 3])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([2,1,3,2])
0[ ]0   -1[ ]0    0[ ]0
        -1[ ]-1
sage: mg.f_string([2,3,2,1,3,2])
0[ ]-1   -1[ ][ ]-1   -1[ ][ ]0
         -1[ ]0
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['C', Integer(3)])
>>> mg = RC.highest_weight_vector()
```

(continues on next page)

(continued from previous page)

```
>>> mg.f_string([Integer(2),Integer(1),Integer(3),Integer(2)])
0[ ]0   -1[ ]0    0[ ]0
        -1[ ]-1
>>> mg.f_string([Integer(2),Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2)])
0[ ]-1   -1[ ][ ]-1   -1[ ][ ]0
         -1[ ]0
```

We can construct rigged configurations using a diagram folding of a simply-laced type. This yields an equivalent but distinct crystal:

```
sage: vct = CartanType(['C', 3]).as_folding()
sage: VRC = crystals.infinity.RiggedConfigurations(vct)
sage: mg = VRC.highest_weight_vector()
sage: mg.f_string([2,1,3,2])
0[ ]0   -2[ ]-1   0[ ]0
        -2[ ]-1
sage: mg.f_string([2,3,2,1,3,2])
-1[ ]-1  -2[ ][ ][ ]-1  -1[ ][ ]0

sage: G = RC.subcrystal(max_depth=5).digraph()
sage: VG = VRC.subcrystal(max_depth=5).digraph()
sage: G.is_isomorphic(VG, edge_labels=True)
True
```

```
>>> from sage.all import *
>>> vct = CartanType(['C', Integer(3)]).as_folding()
>>> VRC = crystals.infinity.RiggedConfigurations(vct)
>>> mg = VRC.highest_weight_vector()
>>> mg.f_string([Integer(2),Integer(1),Integer(3),Integer(2)])
0[ ]0   -2[ ]-1   0[ ]0
        -2[ ]-1
>>> mg.f_string([Integer(2),Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2)])
-1[ ]-1  -2[ ][ ][ ]-1  -1[ ][ ]0

>>> G = RC.subcrystal(max_depth=Integer(5)).digraph()
>>> VG = VRC.subcrystal(max_depth=Integer(5)).digraph()
>>> G.is_isomorphic(VG, edge_labels=True)
True
```

We can also construct $B(\infty)$ using rigged configurations in affine types:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3, 1])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([0,1,2,3,0,1,3])
-1[ ]0  -1[ ]-1  1[ ]1  -1[ ][ ]-1
-1[ ]0  -1[ ]-1

sage: RC = crystals.infinity.RiggedConfigurations(['C', 3, 1])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([1,2,3,0,1,2,3,3,0])
```

(continues on next page)

(continued from previous page)

```
-2[ ][ ]-1   0[ ]1   0[ ]0    -4[ ][ ][ ]-2
             0[ ]0   0[ ]-1

sage: RC = crystals.infinity.RiggedConfigurations(['A', 6, 2])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([1,2,3,0,1,2,3,3,0])
0[ ]-1   0[ ]1   0[ ]0    -4[ ][ ][ ]-2
0[ ]-1   0[ ]1   0[ ]-1
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['A', Integer(3), Integer(1)])
>>> mg = RC.highest_weight_vector()
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(3),Integer(0),
↪Integer(1),Integer(3)])
-1[ ]0  -1[ ]-1  1[ ]1  -1[ ][ ]-1
-1[ ]0  -1[ ]-1

>>> RC = crystals.infinity.RiggedConfigurations(['C', Integer(3), Integer(1)])
>>> mg = RC.highest_weight_vector()
>>> mg.f_string([Integer(1),Integer(2),Integer(3),Integer(0),Integer(1),
↪Integer(2),Integer(3),Integer(3),Integer(0)])
-2[ ][ ]-1   0[ ]1   0[ ]0    -4[ ][ ][ ]-2
             0[ ]0   0[ ]-1

>>> RC = crystals.infinity.RiggedConfigurations(['A', Integer(6), Integer(2)])
>>> mg = RC.highest_weight_vector()
>>> mg.f_string([Integer(1),Integer(2),Integer(3),Integer(0),Integer(1),
↪Integer(2),Integer(3),Integer(3),Integer(0)])
0[ ]-1   0[ ]1   0[ ]0    -4[ ][ ][ ]-2
0[ ]-1   0[ ]1   0[ ]-1
```

We reset the global options:

```
sage: RiggedConfigurations.options._reset()
```

```
>>> from sage.all import *
>>> RiggedConfigurations.options._reset()
```

class Element(*parent*, *rigged_partitions=[]*, ***options*)

Bases: `RiggedConfigurationElement`

A rigged configuration in $\mathcal{B}(\infty)$ in simply-laced types.

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3, 1])
sage: elt = RC(partition_list=[[1,1]]*4, rigging_list=[[1,1], [0,0], [0,
↪0], [-1,-1]])
sage: elt.weight()
-2*delta
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['A', Integer(3),␣
↪Integer(1)])
>>> elt = RC(partition_list=[[Integer(1),Integer(1)]]*Integer(4), rigging_
↪list=[[Integer(1),Integer(1)], [Integer(0),Integer(0)], [Integer(0),
↪Integer(0)], [-Integer(1),-Integer(1)]])
>>> elt.weight()
-2*delta
```

options = Current options for RiggedConfigurations - convention: English - display: vertical - element_ascii_art: True - half_width_boxes_type_B: True

weight_lattice_realization()

Return the weight lattice realization used to express the weights of elements in `self`.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 2, 1])
sage: RC.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['A', Integer(2), Integer(1)])
>>> RC.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

5.1.214 Rigged Configuration Elements

A rigged configuration element is a sequence of *RiggedPartition* objects.

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version
- Travis Scrimshaw (2012-10-25): Added virtual rigged configurations

class sage.combinat.rigged_configurations.rigged_configuration_element.**KRRCNonSimplyLacedElement**(*parent*, *rigged_partitions*, ***options*)

Bases: *KRRiggedConfigurationElement*, *RCNonSimplyLacedElement*

$U'_q(\mathfrak{g})$ rigged configurations in non-simply-laced types.

cc()

Compute the cocharge statistic.

Computes the cocharge statistic [OSS03] on this rigged configuration (ν, J) by computing the cocharge as a virtual rigged configuration $(\hat{\nu}, \hat{J})$ and then using the identity $cc(\hat{\nu}, \hat{J}) = \gamma_0 cc(\nu, J)$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C', 3, 1], [[2,1], [1,1]])
sage: RC(partition_list=[[1,1],[2,1],[1,1]]).cocharge()
1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C', Integer(3), Integer(1)], [[Integer(2),
↪Integer(1)], [Integer(1),Integer(1)]])
>>> RC(partition_list=[[Integer(1),Integer(1)],[Integer(2),Integer(1)],
↪[Integer(1),Integer(1)]]).cocharge()
1
```

cocharge()

Compute the cocharge statistic.

Computes the cocharge statistic [OSS03] on this rigged configuration (ν, J) by computing the cocharge as a virtual rigged configuration $(\hat{\nu}, \hat{J})$ and then using the identity $cc(\hat{\nu}, \hat{J}) = \gamma_0 cc(\nu, J)$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C', 3, 1], [[2,1], [1,1]])
sage: RC(partition_list=[[1,1],[2,1],[1,1]]).cocharge()
1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C', Integer(3), Integer(1)], [[Integer(2),
↪Integer(1)], [Integer(1),Integer(1)]])
>>> RC(partition_list=[[Integer(1),Integer(1)],[Integer(2),Integer(1)],
↪[Integer(1),Integer(1)]]).cocharge()
1
```

e(a)

Return the action of e_a on `self`.

This works by lifting into the virtual configuration, then applying

$$e_a^v = \prod_{j \in \iota(a)} \hat{e}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',6,2], [[1,1]]*7)
sage: elt = RC(partition_list=[[1]*5,[2,1,1],[3,2]])
sage: elt.e(3)

0[ ]0
0[ ]0
0[ ]0
0[ ]0
0[ ]0

0[ ][ ]0
1[ ]1
```

(continues on next page)

(continued from previous page)

```
1[ ]1

1[ ][ ]1
1[ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(6),Integer(2)], [[Integer(1),
↪Integer(1)]]*Integer(7))
>>> elt = RC(partition_list=[[Integer(1)]*Integer(5),[Integer(2),Integer(1),
↪Integer(1)],[Integer(3),Integer(2)]])
>>> elt.e(Integer(3))
<BLANKLINE>
0[ ]0
0[ ]0
0[ ]0
0[ ]0
0[ ]0
<BLANKLINE>
0[ ][ ]0
1[ ]1
1[ ]1
<BLANKLINE>
1[ ][ ]1
1[ ]0
<BLANKLINE>
```

f(*a*)

Return the action of f_a on `self`.

This works by lifting into the virtual configuration, then applying

$$f_a^v = \prod_{j \in \iota(a)} \hat{f}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',6,2], [[1,1]]*7)
sage: elt = RC(partition_list=[[1]*5,[2,1,1],[2,1]], rigging_list=[[0]*5,[0,1,
↪1],[1,0]])
sage: elt.f(3)

0[ ]0
0[ ]0
0[ ]0
0[ ]0
0[ ]0

1[ ][ ]1
1[ ]1
1[ ]1

```

(continues on next page)

(continued from previous page)

```
-1[ ][ ][ ]-1
 0[ ][ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(6),Integer(2)], [[Integer(1),
↪Integer(1)]]*Integer(7))
>>> elt = RC(partition_list=[[Integer(1)]*Integer(5),[Integer(2),Integer(1),
↪Integer(1)],[Integer(2),Integer(1)]], rigging_list=[[Integer(0)]*Integer(5),
↪[Integer(0),Integer(1),Integer(1)],[Integer(1),Integer(0)]])
>>> elt.f(Integer(3))
<BLANKLINE>
0[ ]0
0[ ]0
0[ ]0
0[ ]0
0[ ]0
<BLANKLINE>
1[ ][ ]1
1[ ]1
1[ ]1
<BLANKLINE>
-1[ ][ ][ ]-1
 0[ ][ ]0
<BLANKLINE>
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**KRRCSimplyLacedElement**(*parent*, *rigged_partitions=[]*, ***options*)

Bases: *KRRiggedConfigurationElement*

$U_q'(\mathfrak{g})$ rigged configurations in simply-laced types.

cc()

Compute the cocharge statistic of `self`.

Computes the cocharge statistic [CrysStructSchilling06] on this rigged configuration (ν, J). The cocharge statistic is defined as:

$$cc(\nu, J) = \frac{1}{2} \sum_{a,b \in I_0} \sum_{j,k>0} (\alpha_a \mid \alpha_b) \min(j,k) m_j^{(a)} m_k^{(b)} + \sum_{a \in I} \sum_{i>0} \left| J^{(a,i)} \right|.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [2,1], [1,1]])
sage: RC(partition_list=[[1], [1], []]).cocharge()
1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)], [Integer(2),Integer(1)], [Integer(1),Integer(1)]])
>>> RC(partition_list=[[Integer(1)], [Integer(1)], []]).cocharge()
1
```

charge()

Compute the charge statistic of `self`.

Let B denote a set of rigged configurations. The *charge* c of a rigged configuration b is computed as

$$c(b) = \max(cc(b) \mid b \in B) - cc(b).$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [2,1], [1,1]])
sage: RC(partition_list=[[],[],[]]).charge()
2
sage: RC(partition_list=[[1], [1], []]).charge()
1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),
↪Integer(2)], [Integer(2),Integer(1)], [Integer(1),Integer(1)]])
>>> RC(partition_list=[[],[],[]]).charge()
2
>>> RC(partition_list=[[Integer(1)], [Integer(1)], []]).charge()
1
```

cocharge()

Compute the cocharge statistic of `self`.

Computes the cocharge statistic [CrysStructSchilling06] on this rigged configuration (ν, J). The cocharge statistic is defined as:

$$cc(\nu, J) = \frac{1}{2} \sum_{a,b \in I_0} \sum_{j,k>0} (\alpha_a \mid \alpha_b) \min(j,k) m_j^{(a)} m_k^{(b)} + \sum_{a \in I} \sum_{i>0} \left| J^{(a,i)} \right|.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [2,1], [1,1]])
sage: RC(partition_list=[[1], [1], []]).cocharge()
1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),
↪Integer(2)], [Integer(2),Integer(1)], [Integer(1),Integer(1)]])
>>> RC(partition_list=[[Integer(1)], [Integer(1)], []]).cocharge()
1
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**KRRCTypeA2DualElement**(*parent*, *rigged_partitions=[]*, ***options*)

Bases: *KRRCNonSimplyLacedElement*

$U'_q(\mathfrak{g})$ rigged configurations in type $A_{2n}^{(2)\dagger}$.

cc()

Compute the cocharge statistic.

Computes the cocharge statistic [RigConBijection] on this rigged configuration (ν, J). The cocharge statistic is computed as:

$$cc(\nu, J) = \frac{1}{2} \sum_{a \in I_0} \sum_{i > 0} t_a^{\vee} m_i^{(a)} \left(\sum_{j > 0} \min(i, j) L_j^{(a)} - p_i^{(a)} \right) + \sum_{a \in I} t_a^{\vee} \sum_{i > 0} \left| J^{(a,i)} \right|.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1],[2,2]])
sage: sc = RC.cartan_type().as_folding().scaling_factors()
sage: all(mg.cocharge() * sc[0] == mg.to_virtual_configuration().cocharge()
....:     for mg in RC.module_generators)
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),
↪[[Integer(1),Integer(1)],[Integer(2),Integer(2)]])
>>> sc = RC.cartan_type().as_folding().scaling_factors()
>>> all(mg.cocharge() * sc[Integer(0)] == mg.to_virtual_configuration().
↪cocharge()
...     for mg in RC.module_generators)
True
```

cocharge()

Compute the cocharge statistic.

Computes the cocharge statistic [RigConBijection] on this rigged configuration (ν, J). The cocharge statistic is computed as:

$$cc(\nu, J) = \frac{1}{2} \sum_{a \in I_0} \sum_{i > 0} t_a^{\vee} m_i^{(a)} \left(\sum_{j > 0} \min(i, j) L_j^{(a)} - p_i^{(a)} \right) + \sum_{a \in I} t_a^{\vee} \sum_{i > 0} \left| J^{(a,i)} \right|.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1],[2,2]])
sage: sc = RC.cartan_type().as_folding().scaling_factors()
sage: all(mg.cocharge() * sc[0] == mg.to_virtual_configuration().cocharge()
....:     for mg in RC.module_generators)
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),
↪[[Integer(1),Integer(1)],[Integer(2),Integer(2)]])
>>> sc = RC.cartan_type().as_folding().scaling_factors()
>>> all(mg.cocharge() * sc[Integer(0)] == mg.to_virtual_configuration().
```

(continues on next page)

(continued from previous page)

```
↪cocharge()
...     for mg in RC.module_generators)
True
```

epsilon(*a*)

Return the value of ε_a of `self`.

Here we need to modify the usual definition by $\varepsilon_n' := 2\varepsilon_n$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1], [2,2]])
sage: def epsilon(x, i):
....:     x = x.e(i)
....:     eps = 0
....:     while x is not None:
....:         x = x.e(i)
....:         eps = eps + 1
....:     return eps
sage: all(epsilon(rc, 2) == rc.epsilon(2) for rc in RC)
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),␣
↪[[Integer(1),Integer(1)], [Integer(2),Integer(2)]])
>>> def epsilon(x, i):
...     x = x.e(i)
...     eps = Integer(0)
...     while x is not None:
...         x = x.e(i)
...         eps = eps + Integer(1)
...     return eps
>>> all(epsilon(rc, Integer(2)) == rc.epsilon(Integer(2)) for rc in RC)
True
```

phi(*a*)

Return the value of φ_a of `self`.

Here we need to modify the usual definition by $\varphi_n' := 2\varphi_n$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1], [2,2]])
sage: def phi(x, i):
....:     x = x.f(i)
....:     ph = 0
....:     while x is not None:
....:         x = x.f(i)
....:         ph = ph + 1
....:     return ph
sage: all(phi(rc, 2) == rc.phi(2) for rc in RC)
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),
↪[[Integer(1),Integer(1)], [Integer(2),Integer(2)]])
>>> def phi(x, i):
...     x = x.f(i)
...     ph = Integer(0)
...     while x is not None:
...         x = x.f(i)
...         ph = ph + Integer(1)
...     return ph
>>> all(phi(rc, Integer(2)) == rc.phi(Integer(2)) for rc in RC)
True
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**KRRiggedConfigurationElement**(p

Bases: *RiggedConfigurationElement*

$U_q'(\mathfrak{g})$ rigged configurations.

EXAMPLES:

We can go between *rigged configurations* and tensor products of *tensor products of KR tableaux*:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[1,1], [2,1]])
sage: rc_elt = RC(partition_list=[[1], [1,1], [1], [1]])
sage: tp_krtab = rc_elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); tp_
↪krtab
[[-2]] (X) [[1], [2]]
sage: tp_krcrys = rc_elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_
↪krcrys
[[[-2]], [[1], [2]]]
sage: tp_krcrys == tp_krtab.to_tensor_product_of_kirillov_reshetikhin_crystals()
True
sage: RC(tp_krcrys) == rc_elt
True
sage: RC(tp_krtab) == rc_elt
True
sage: tp_krtab.to_rigged_configuration() == rc_elt
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(2),Integer(1)]])
>>> rc_elt = RC(partition_list=[[Integer(1)], [Integer(1),Integer(1)],
↪[Integer(1)], [Integer(1)]])
>>> tp_krtab = rc_elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); tp_
↪krtab
[[-2]] (X) [[1], [2]]
>>> tp_krcrys = rc_elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_
```

(continues on next page)

(continued from previous page)

```
↪krcrys
[[[-2]], [[1], [2]]]
>>> tp_krcrys == tp_krtab.to_tensor_product_of_kirillov_reshetikhin_crystals()
True
>>> RC(tp_krcrys) == rc_elt
True
>>> RC(tp_krtab) == rc_elt
True
>>> tp_krtab.to_rigged_configuration() == rc_elt
True
```

check()

Make sure all of the riggings are less than or equal to the vacancy number.

classical_weight()

Return the classical weight of `self`.

The classical weight Λ of a rigged configuration is

$$\Lambda = \sum_{a \in \overline{I}} \sum_{i > 0} i L_i^{(a)} \Lambda_a - \sum_{a \in \overline{I}} \sum_{i > 0} i m_i^{(a)} \alpha_a.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D',4,1], [[2,2]])
sage: elt = RC(partition_list=[[2],[2,1],[1],[1]])
sage: elt.classical_weight()
(0, 1, 1, 0)
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D',Integer(4),Integer(1)], [[Integer(2),
↪Integer(2)]])
>>> elt = RC(partition_list=[[Integer(2)],[Integer(2),Integer(1)],
↪[Integer(1)],[Integer(1)]])
>>> elt.classical_weight()
(0, 1, 1, 0)
```

This agrees with the corresponding classical weight as KR tableaux:

```
sage: krt = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); krt
[[2, 1], [3, -1]]
sage: krt.classical_weight() == elt.classical_weight()
True
```

```
>>> from sage.all import *
>>> krt = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); krt
[[2, 1], [3, -1]]
>>> krt.classical_weight() == elt.classical_weight()
True
```

complement_rigging(*reverse_factors=False*)

Apply the complement rigging morphism θ to `self`.

Consider a highest weight rigged configuration (ν, J), the complement rigging morphism $\theta : RC(L) \to RC(L)$ is given by sending $(\nu, J) \mapsto (\nu, J')$, where J' is obtained by taking the coriggings $x' = p_i^{(a)} - x$, and then extending as a crystal morphism. (The name comes from taking the complement partition for the riggings in a $m_i^{(a)} \times p_i^{(a)}$ box.)

INPUT:

- `reverse_factors` – boolean (default: `False`); if `True`, then this returns an element in $RC(B')$ where B' is the tensor factors of `self` in reverse order

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D',4,1], [[1,1],[2,2]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
1[ ][ ]1  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
          0[ ][ ]0
sage: ascii_art(mg.complement_rigging())
1[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
          0[ ][ ]0

sage: lw = mg.to_lowest_weight([1,2,3,4])[0]
sage: ascii_art(lw)
-1[ ][ ]-1  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
-1[ ]-1     0[ ][ ]0  0[ ]0     0[ ]0
-1[ ]-1     0[ ]0
            0[ ]0
sage: ascii_art(lw.complement_rigging())
-1[ ][ ][ ]-1  0[ ][ ][ ]0  0[ ][ ][ ]0  0[ ][ ][ ]0
-1[ ]-1        0[ ][ ][ ]0
sage: lw.complement_rigging() == mg.complement_rigging().to_lowest_weight([1,
↪2,3,4])[0]
True

sage: mg.complement_rigging(True).parent()
Rigged configurations of type ['D', 4, 1] and factor(s) ((2, 2), (1, 1))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D',Integer(4),Integer(1)], [[Integer(1),
↪Integer(1)],[Integer(2),Integer(2)]])
>>> mg = RC.module_generators[-Integer(1)]
>>> ascii_art(mg)
1[ ][ ]1  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
          0[ ][ ]0
>>> ascii_art(mg.complement_rigging())
1[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
          0[ ][ ]0

>>> lw = mg.to_lowest_weight([Integer(1),Integer(2),Integer(3),
↪Integer(4)])[Integer(0)]
>>> ascii_art(lw)
-1[ ][ ]-1  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
-1[ ]-1     0[ ][ ]0  0[ ]0     0[ ]0
-1[ ]-1     0[ ]0
```

(continues on next page)

(continued from previous page)

```
            0[ ]0
>>> ascii_art(lw.complement_rigging())
-1[ ][ ][ ]-1  0[ ][ ][ ]0  0[ ][ ][ ]0  0[ ][ ][ ]0
-1[ ]-1        0[ ][ ][ ]0
>>> lw.complement_rigging() == mg.complement_rigging().to_lowest_
↪weight([Integer(1),Integer(2),Integer(3),Integer(4)])[Integer(0)]
True

>>> mg.complement_rigging(True).parent()
Rigged configurations of type ['D', 4, 1] and factor(s) ((2, 2), (1, 1))
```

We check that the Lusztig involution (under the modification of also mapping to the highest weight element) intertwines with the complement map θ (that reverses the tensor factors) under the bijection Φ:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2], [2, 1], [1, 2]])
sage: for mg in RC.module_generators: # long time
....:     y = mg.to_tensor_product_of_kirillov_reshetikhin_tableaux()
....:     hw = y.lusztig_involution().to_highest_weight([1,2,3,4])[0]
....:     c = mg.complement_rigging(True)
....:     hwc = c.to_tensor_product_of_kirillov_reshetikhin_tableaux()
....:     assert hw == hwc
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),␣
↪Integer(2)], [Integer(2), Integer(1)], [Integer(1), Integer(2)]])
>>> for mg in RC.module_generators: # long time
...     y = mg.to_tensor_product_of_kirillov_reshetikhin_tableaux()
...     hw = y.lusztig_involution().to_highest_weight([Integer(1),Integer(2),
↪Integer(3),Integer(4)])[Integer(0)]
...     c = mg.complement_rigging(True)
...     hwc = c.to_tensor_product_of_kirillov_reshetikhin_tableaux()
...     assert hw == hwc
```

delta(*return_b=False*)

Return the image of `self` under the left box removal map δ.

The map $\delta : RC(B^{r,1} \otimes B) \to RC(B^{r-1,1} \otimes B)$ (if $r = 1$, then we remove the left-most factor) is the basic map in the bijection Φ between rigged configurations and tensor products of Kirillov-Reshetikhin tableaux. For more information, see `to_tensor_product_of_kirillov_reshetikhin_tableaux()`. We can extend δ when the left-most factor is not a single column by precomposing with a `left_split()`.

Note

Due to the special nature of the bijection for the spinor cases in types $D_n^{(1)}$, $B_n^{(1)}$, and $A_{2n-1}^{(2)}$, this map is not defined in these cases.

INPUT:

- `return_b` – boolean (default: `False`); whether to return the resulting letter from δ

OUTPUT:

The resulting rigged configuration or if `return_b` is `True`, then a tuple of the resulting rigged configuration and the letter.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',4,1], [[3,2]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
sage: ascii_art(mg.left_box())
0[ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ][ ]0  0[ ]0
sage: x,b = mg.left_box(True)
sage: b
-1
sage: b.parent()
The crystal of letters for type ['C', 4]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(4),Integer(1)], [[Integer(3),
↪Integer(2)]])
>>> mg = RC.module_generators[-Integer(1)]
>>> ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
>>> ascii_art(mg.left_box())
0[ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ][ ]0  0[ ]0
>>> x,b = mg.left_box(True)
>>> b
-1
>>> b.parent()
The crystal of letters for type ['C', 4]
```

e(a)

Return the action of the crystal operator e_a on `self`.

For the classical operators, this implements the method defined in [CrysStructSchilling06]. For e_0, this converts the class to a tensor product of KR tableaux and does the corresponding e_0 and pulls back.

Todo

Implement e_0 without appealing to tensor product of KR tableaux.

INPUT:

- `a` – the index of the partition to remove a box

OUTPUT: the resulting rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2,1]])
sage: elt = RC(partition_list=[[1], [1], [1], [1]])
sage: elt.e(3)
sage: elt.e(1)

(/)

0[ ]0

0[ ]0

-1[ ]-1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),
↪Integer(1)]])
>>> elt = RC(partition_list=[[Integer(1)], [Integer(1)], [Integer(1)],
↪[Integer(1)]])
>>> elt.e(Integer(3))
>>> elt.e(Integer(1))
<BLANKLINE>
(/)
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
```

epsilon(*a*)

Return ε_a of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: I = RC.index_set()
sage: matrix([[mg.epsilon(i) for i in I] for mg in RC.module_generators])
[4 0 0 0 0]
[3 0 0 0 0]
[2 0 0 0 0]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),
↪Integer(2)]])
>>> I = RC.index_set()
>>> matrix([[mg.epsilon(i) for i in I] for mg in RC.module_generators])
[4 0 0 0 0]
[3 0 0 0 0]
[2 0 0 0 0]
```

f(*a*)

Return the action of the crystal operator f_a on `self`.

For the classical operators, this implements the method defined in [CrysStructSchilling06]. For f_0, this converts the class to a tensor product of KR tableaux and does the corresponding f_0 and pulls back.

Todo

Implement f_0 without appealing to tensor product of KR tableaux.

INPUT:

- `a` – the index of the partition to add a box

OUTPUT: the resulting rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2,1]])
sage: elt = RC(partition_list=[[1], [1], [1], [1]])
sage: elt.f(1)
sage: elt.f(2)

0[ ]0

-1[ ]-1
-1[ ]-1

1[ ]1

-1[ ]-1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),
↪Integer(1)]])
>>> elt = RC(partition_list=[[Integer(1)], [Integer(1)], [Integer(1)],
↪[Integer(1)]])
>>> elt.f(Integer(1))
>>> elt.f(Integer(2))
<BLANKLINE>
0[ ]0
<BLANKLINE>
-1[ ]-1
-1[ ]-1
<BLANKLINE>
1[ ]1
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
```

left_box(*return_b=False*)

Return the image of `self` under the left box removal map δ.

The map $\delta : RC(B^{r,1} \otimes B) \to RC(B^{r-1,1} \otimes B)$ (if $r = 1$, then we remove the left-most factor) is the basic map in the bijection Φ between rigged configurations and tensor products of Kirillov-Reshetikhin tableaux. For more information, see *to_tensor_product_of_kirillov_reshetikhin_tableaux()*. We can extend δ when the left-most factor is not a single column by precomposing with a *left_split()*.

Note

Due to the special nature of the bijection for the spinor cases in types $D_n^{(1)}$, $B_n^{(1)}$, and $A_{2n-1}^{(2)}$, this map is not defined in these cases.

INPUT:

- `return_b` – boolean (default: `False`); whether to return the resulting letter from δ

OUTPUT:

The resulting rigged configuration or if `return_b` is `True`, then a tuple of the resulting rigged configuration and the letter.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',4,1], [[3,2]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
sage: ascii_art(mg.left_box())
0[ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ][ ]0  0[ ]0
sage: x,b = mg.left_box(True)
sage: b
-1
sage: b.parent()
The crystal of letters for type ['C', 4]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(4),Integer(1)], [[Integer(3),
↪Integer(2)]])
>>> mg = RC.module_generators[-Integer(1)]
>>> ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
>>> ascii_art(mg.left_box())
0[ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ][ ]0  0[ ]0
>>> x,b = mg.left_box(True)
>>> b
-1
>>> b.parent()
The crystal of letters for type ['C', 4]
```

left_column_box()

Return the image of `self` under the left column box splitting map γ.

Consider the map $\gamma : RC(B^{r,1} \otimes B) \to RC(B^{1,1} \otimes B^{r-1,1} \otimes B)$ for $r > 1$, which is a natural strict classical crystal injection. On rigged configurations, the map γ adds a singular string of length 1 to $\nu^{(a)}$.

We can extend γ when the left-most factor is not a single column by precomposing with a *left_split()*.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',3,1], [[3,1], [2,1]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
sage: ascii_art(mg.left_column_box())
0[ ]0  0[ ][ ]0  0[ ]0
0[ ]0  0[ ]0     0[ ]0
       0[ ]0

sage: RC = RiggedConfigurations(['C',3,1], [[2,1], [1,1], [3,1]])
sage: mg = RC.module_generators[7]
sage: ascii_art(mg)
1[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
sage: ascii_art(mg.left_column_box())
1[ ]1  0[ ][ ]0  0[ ]0
1[ ]0  0[ ]0     0[ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(3),Integer(1)], [[Integer(3),
↪Integer(1)], [Integer(2),Integer(1)]])
>>> mg = RC.module_generators[-Integer(1)]
>>> ascii_art(mg)
0[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
>>> ascii_art(mg.left_column_box())
0[ ]0  0[ ][ ]0  0[ ]0
0[ ]0  0[ ]0     0[ ]0
       0[ ]0

>>> RC = RiggedConfigurations(['C',Integer(3),Integer(1)], [[Integer(2),
↪Integer(1)], [Integer(1),Integer(1)], [Integer(3),Integer(1)]])
>>> mg = RC.module_generators[Integer(7)]
>>> ascii_art(mg)
1[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
>>> ascii_art(mg.left_column_box())
1[ ]1  0[ ][ ]0  0[ ]0
1[ ]0  0[ ]0     0[ ]0
```

left_split()

Return the image of `self` under the left column splitting map β.

Consider the map $\beta : RC(B^{r,s} \otimes B) \to RC(B^{r,1} \otimes B^{r,s-1} \otimes B)$ for $s > 1$ which is a natural classical crystal injection. On rigged configurations, the map β does nothing (except possibly changing the vacancy numbers).

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',4,1], [[3,3]])
sage: mg = RC.module_generators[-1]
```

(continues on next page)

(continued from previous page)

```
sage: ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
sage: ascii_art(mg.left_split())
0[ ][ ]0  0[ ][ ]0  1[ ][ ]0  0[ ]0
          0[ ][ ]0  1[ ][ ]0  0[ ]0
                    1[ ][ ]0  0[ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(4),Integer(1)], [[Integer(3),
↪Integer(3)]])
>>> mg = RC.module_generators[-Integer(1)]
>>> ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
>>> ascii_art(mg.left_split())
0[ ][ ]0  0[ ][ ]0  1[ ][ ]0  0[ ]0
          0[ ][ ]0  1[ ][ ]0  0[ ]0
                    1[ ][ ]0  0[ ]0
```

phi(a)

Return φ_a of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: I = RC.index_set()
sage: matrix([[mg.phi(i) for i in I] for mg in RC.module_generators])
[0 0 2 0 0]
[1 0 1 0 0]
[2 0 0 0 0]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),
↪Integer(2)]])
>>> I = RC.index_set()
>>> matrix([[mg.phi(i) for i in I] for mg in RC.module_generators])
[0 0 2 0 0]
[1 0 1 0 0]
[2 0 0 0 0]
```

right_column_box()

Return the image of `self` under the right column box splitting map γ^*.

Consider the map $\gamma^* : RC(B \otimes B^{r,1}) \to RC(B \otimes B^{r-1,1} \otimes B^{1,1})$ for $r > 1$, which is a natural strict classical crystal injection. On rigged configurations, the map γ adds a string of length 1 with rigging 0 to $\nu^{(a)}$ for all $a < r$ to a classically highest weight element and extended as a classical crystal morphism.

We can extend γ^* when the right-most factor is not a single column by precomposing with a *right_split()*.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',3,1], [[2,1], [1,1], [3,1]])
sage: mg = RC.module_generators[7]
sage: ascii_art(mg)
1[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
sage: ascii_art(mg.right_column_box())
1[ ]0  0[ ][ ]0  0[ ]0
1[ ]0  0[ ]0     0[ ]0
       0[ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(3),Integer(1)], [[Integer(2),
↪Integer(1)], [Integer(1),Integer(1)], [Integer(3),Integer(1)]])
>>> mg = RC.module_generators[Integer(7)]
>>> ascii_art(mg)
1[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
>>> ascii_art(mg.right_column_box())
1[ ]0  0[ ][ ]0  0[ ]0
1[ ]0  0[ ]0     0[ ]0
       0[ ]0
```

right_split()

Return the image of `self` under the right column splitting map β^*.

Let θ denote the *complement rigging map* which reverses the tensor factors and β denote the *left splitting map*, we define the right splitting map by $\beta^* := \theta \circ \beta \circ \theta$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',4,1], [[3,3]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
sage: ascii_art(mg.right_split())
0[ ][ ]0  0[ ][ ]0  1[ ][ ]1  0[ ]0
          0[ ][ ]0  1[ ][ ]1  0[ ]0
                    1[ ][ ]1  0[ ]0

sage: RC = RiggedConfigurations(['D',4,1], [[2,2],[1,2]])
sage: elt = RC(partition_list=[[3,1], [2,2,1], [2,1], [2]])
sage: ascii_art(elt)
-1[ ][ ][ ]-1  0[ ][ ]0  -1[ ][ ]-1  1[ ][ ]1
 0[ ]0         0[ ][ ]0  -1[ ]-1
               0[ ]0
sage: ascii_art(elt.right_split())
-1[ ][ ][ ]-1  0[ ][ ]0  -1[ ][ ]-1  1[ ][ ]1
 1[ ]0         0[ ][ ]0  -1[ ]-1
               0[ ]0
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> RC = RiggedConfigurations(['C',Integer(4),Integer(1)], [[Integer(3),
↪Integer(3)]])
>>> mg = RC.module_generators[-Integer(1)]
>>> ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
>>> ascii_art(mg.right_split())
0[ ][ ]0  0[ ][ ]0  1[ ][ ]1  0[ ]0
          0[ ][ ]0  1[ ][ ]1  0[ ]0
                    1[ ][ ]1  0[ ]0

>>> RC = RiggedConfigurations(['D',Integer(4),Integer(1)], [[Integer(2),
↪Integer(2)],[Integer(1),Integer(2)]])
>>> elt = RC(partition_list=[[Integer(3),Integer(1)], [Integer(2),Integer(2),
↪Integer(1)], [Integer(2),Integer(1)], [Integer(2)]])
>>> ascii_art(elt)
-1[ ][ ][ ]-1  0[ ][ ]0  -1[ ][ ]-1  1[ ][ ]1
 0[ ]0         0[ ][ ]0  -1[ ]-1
               0[ ]0
>>> ascii_art(elt.right_split())
-1[ ][ ][ ]-1  0[ ][ ]0  -1[ ][ ]-1  1[ ][ ]1
 1[ ]0         0[ ][ ]0  -1[ ]-1
               0[ ]0
```

We check that the bijection commutes with the right splitting map:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[1,1], [2,2]])
sage: all(rc.right_split().to_tensor_product_of_kirillov_reshetikhin_
↪tableaux()
....:     == rc.to_tensor_product_of_kirillov_reshetikhin_tableaux().right_
↪split() for rc in RC)
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(2),Integer(2)]])
>>> all(rc.right_split().to_tensor_product_of_kirillov_reshetikhin_tableaux()
...     == rc.to_tensor_product_of_kirillov_reshetikhin_tableaux().right_
↪split() for rc in RC)
True
```

to_tensor_product_of_kirillov_reshetikhin_crystals(*display_steps=False*, *build_graph=False*)

Return the corresponding tensor product of Kirillov-Reshetikhin crystals.

This is a composition of the map to a tensor product of KR tableaux, and then to a tensor product of KR crystals.

INPUT:

- `display_steps` – boolean (default: `False`); indicates whether to print each step in the algorithm
- `build_graph` – boolean (default: `False`); indicates whether to construct and return a graph of the bijection whose vertices are rigged configurations obtained at each step and edges are labeled by either the return value of δ or the doubling/halving map

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: elt = RC(partition_list=[[2], [2,2], [1], [1]])
sage: krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); krc
[[[2, 3], [3, -2]]]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),␣
↪Integer(2)]])
>>> elt = RC(partition_list=[[Integer(2)], [Integer(2),Integer(2)],␣
↪[Integer(1)], [Integer(1)]])
>>> krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); krc
[[[2, 3], [3, -2]]]
```

We can recover the rigged configuration:

```
sage: ret = RC(krc); ret

0[ ][ ]0

-2[ ][ ]-2
-2[ ][ ]-2

0[ ]0

0[ ]0

sage: elt == ret
True
```

```
>>> from sage.all import *
>>> ret = RC(krc); ret
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
-2[ ][ ]-2
-2[ ][ ]-2
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
>>> elt == ret
True
```

We can also construct and display a graph of the bijection as follows:

```
sage: ret, G = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(build_
↪graph=True)
sage: view(G) # not tested
```

```
>>> from sage.all import *
>>> ret, G = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(build_
↪graph=True)
>>> view(G) # not tested
```

to_tensor_product_of_kirillov_reshetikhin_tableaux(*display_steps=False*, *build_graph=False*)

Perform the bijection from this rigged configuration to a tensor product of Kirillov-Reshetikhin tableaux given in [RigConBijection] for single boxes and with [BijectionLRT] and [BijectionDn] for multiple columns and rows.

Note: This is only proven to be a bijection in types $A_n^{(1)}$ and $D_n^{(1)}$, as well as $\bigotimes_i B^{r_i,1}$ and $\bigotimes_i B^{1,s_i}$ for general affine types.

INPUT:

- `display_steps` – boolean (default: `False`); indicates whether to print each step in the algorithm
- `build_graph` – boolean (default: `False`); indicates whether to construct and return a graph of the bijection whose vertices are rigged configurations obtained at each step and edges are labeled by either the return value of δ or the doubling/halving map

OUTPUT:

- The tensor product of KR tableaux element corresponding to this rigged configuration.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RC(partition_list=[[2], [2,2], [2], [2]]).to_tensor_product_of_kirillov_
↪reshetikhin_tableaux()
[[3, 3], [5, 5]]
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: elt = RC(partition_list=[[2], [2,2], [1], [1]])
sage: tp_krt = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); tp_
↪krt
[[2, 3], [3, -2]]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),␣
↪Integer(2)]])
>>> RC(partition_list=[[Integer(2)], [Integer(2),Integer(2)], [Integer(2)],␣
↪[Integer(2)]]).to_tensor_product_of_kirillov_reshetikhin_tableaux()
[[3, 3], [5, 5]]
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),␣
↪Integer(2)]])
>>> elt = RC(partition_list=[[Integer(2)], [Integer(2),Integer(2)],␣
↪[Integer(1)], [Integer(1)]])
>>> tp_krt = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); tp_krt
[[2, 3], [3, -2]]
```

This is invertible by calling *to_rigged_configuration()*:

```
sage: ret = tp_krt.to_rigged_configuration(); ret

0[ ][ ]0

-2[ ][ ]-2
-2[ ][ ]-2

0[ ]0

0[ ]0

sage: elt == ret
True
```

```
>>> from sage.all import *
>>> ret = tp_krt.to_rigged_configuration(); ret
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
-2[ ][ ]-2
-2[ ][ ]-2
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
>>> elt == ret
True
```

To view the steps of the bijection in the output, run with the `display_steps=True` option:

```
sage: elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(True)
====================
...
====================

0[ ]0

-2[ ][ ]-2
 0[ ]0

0[ ]0

0[ ]0

--------------------
[[3, 2]]
--------------------
...
[[2, 3], [3, -2]]
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(True)
====================
...
====================
<BLANKLINE>
0[ ]0
<BLANKLINE>
-2[ ][ ]-2
 0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
--------------------
[[3, 2]]
--------------------
...
[[2, 3], [3, -2]]
```

We can also construct and display a graph of the bijection as follows:

```
sage: ret, G = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(build_
→graph=True)
sage: view(G) # not tested
```

```
>>> from sage.all import *
>>> ret, G = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(build_
→graph=True)
>>> view(G) # not tested
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['E', 6, 1], [[2,2]])
sage: [x.weight() for x in RC.module_generators]
[-4*Lambda[0] + 2*Lambda[2], -2*Lambda[0] + Lambda[2], 0]
sage: KR = crystals.KirillovReshetikhin(['E',6,1], 2,2)
sage: [x.weight() for x in KR.module_generators]  # long time
[0, -2*Lambda[0] + Lambda[2], -4*Lambda[0] + 2*Lambda[2]]

sage: RC = RiggedConfigurations(['D', 6, 1], [[4,2]])
sage: [x.weight() for x in RC.module_generators]
[-4*Lambda[0] + 2*Lambda[4], -4*Lambda[0] + Lambda[2] + Lambda[4],
 -2*Lambda[0] + Lambda[4], -4*Lambda[0] + 2*Lambda[2],
 -2*Lambda[0] + Lambda[2], 0]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['E', Integer(6), Integer(1)], [[Integer(2),
→Integer(2)]])
```

(continues on next page)

(continued from previous page)

```
>>> [x.weight() for x in RC.module_generators]
[-4*Lambda[0] + 2*Lambda[2], -2*Lambda[0] + Lambda[2], 0]
>>> KR = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)], Integer(2),
↪Integer(2))
>>> [x.weight() for x in KR.module_generators]  # long time
[0, -2*Lambda[0] + Lambda[2], -4*Lambda[0] + 2*Lambda[2]]

>>> RC = RiggedConfigurations(['D', Integer(6), Integer(1)], [[Integer(4),
↪Integer(2)]])
>>> [x.weight() for x in RC.module_generators]
[-4*Lambda[0] + 2*Lambda[4], -4*Lambda[0] + Lambda[2] + Lambda[4],
 -2*Lambda[0] + Lambda[4], -4*Lambda[0] + 2*Lambda[2],
 -2*Lambda[0] + Lambda[2], 0]
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**RCHWNonSimplyLacedElement**(*parent*, *rigged_partitions*, ***options*)

Bases: *RCNonSimplyLacedElement*

Rigged configurations in highest weight crystals.

check()

Make sure all of the riggings are less than or equal to the vacancy number.

f(a)

Return the action of f_a on `self`.

This works by lifting into the virtual configuration, then applying

$$f_a^v = \prod_{j \in \iota(a)} \hat{f}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: La = RootSystem(['C',2,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: vct = CartanType(['C',2,1]).as_folding()
sage: RC = crystals.RiggedConfigurations(vct, La[0])
sage: elt = RC(partition_list=[[1,1],[2],[2]])
sage: elt.f(0)
sage: ascii_art(elt.f(1))
0[ ]0   0[ ][ ]0  -1[ ][ ]-1
0[ ]0  -1[ ]-1
sage: elt.f(2)
```

```
>>> from sage.all import *
>>> La = RootSystem(['C',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
```

(continues on next page)

(continued from previous page)

```
>>> vct = CartanType(['C',Integer(2),Integer(1)]).as_folding()
>>> RC = crystals.RiggedConfigurations(vct, La[Integer(0)])
>>> elt = RC(partition_list=[[Integer(1),Integer(1)],[Integer(2)],
↪[Integer(2)]])
>>> elt.f(Integer(0))
>>> ascii_art(elt.f(Integer(1)))
0[ ]0   0[ ][ ]0  -1[ ][ ]-1
0[ ]0  -1[ ]-1
>>> elt.f(Integer(2))
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: La = RootSystem(['C',2,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: vct = CartanType(['C',2,1]).as_folding()
sage: B = crystals.RiggedConfigurations(vct, La[0])
sage: mg = B.module_generators[0]
sage: mg.f_string([0,1,2]).weight()
2*Lambda[1] - Lambda[2] - delta
```

```
>>> from sage.all import *
>>> La = RootSystem(['C',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> vct = CartanType(['C',Integer(2),Integer(1)]).as_folding()
>>> B = crystals.RiggedConfigurations(vct, La[Integer(0)])
>>> mg = B.module_generators[Integer(0)]
>>> mg.f_string([Integer(0),Integer(1),Integer(2)]).weight()
2*Lambda[1] - Lambda[2] - delta
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**RCHighestWeightElement**(*parent*, *rigged_partitions=[]*, ***options*)

Bases: *RiggedConfigurationElement*

Rigged configurations in highest weight crystals.

check()

Make sure all of the riggings are less than or equal to the vacancy number.

f(*a*)

Return the action of the crystal operator f_a on `self`.

This implements the method defined in [CrysStructSchilling06] which finds the value k which is the length of the string with the smallest nonpositive rigging of largest length. Then it adds a box from a string of length k in the a-th rigged partition, keeping all colabels fixed and decreasing the new label by one. If no such string exists, then it adds a new string of length 1 with label -1. If any of the resulting vacancy numbers are larger than the labels (i.e. it is an invalid rigged configuration), then f_a is undefined.

INPUT:

- `a` – the index of the partition to add a box

OUTPUT: the resulting rigged configuration element

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: RC = crystals.RiggedConfigurations(['A',2,1], La[0])
sage: elt = RC(partition_list=[[1,1],[1],[2]])
sage: elt.f(0)

-2[ ][ ]-2
-1[ ]-1

1[ ]1

0[ ][ ]0

sage: elt.f(1)

0[ ]0
0[ ]0

-1[ ]-1
-1[ ]-1

0[ ][ ]0

sage: elt.f(2)
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> RC = crystals.RiggedConfigurations(['A',Integer(2),Integer(1)],␣
↪La[Integer(0)])
>>> elt = RC(partition_list=[[Integer(1),Integer(1)],[Integer(1)],
↪[Integer(2)]])
>>> elt.f(Integer(0))
<BLANKLINE>
-2[ ][ ]-2
-1[ ]-1
<BLANKLINE>
1[ ]1
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
>>> elt.f(Integer(1))
<BLANKLINE>
0[ ]0
0[ ]0
<BLANKLINE>
-1[ ]-1
```

(continues on next page)

(continued from previous page)

```
-1[ ]-1
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
>>> elt.f(Integer(2))
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_
↪weights()
sage: B = crystals.RiggedConfigurations(['A',2,1], La[0])
sage: mg = B.module_generators[0]
sage: mg.f_string([0,1,2,0]).weight()
-Lambda[0] + Lambda[1] + Lambda[2] - 2*delta
```

```
>>> from sage.all import *
>>> La = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True).fundamental_weights()
>>> B = crystals.RiggedConfigurations(['A',Integer(2),Integer(1)],␣
↪La[Integer(0)])
>>> mg = B.module_generators[Integer(0)]
>>> mg.f_string([Integer(0),Integer(1),Integer(2),Integer(0)]).weight()
-Lambda[0] + Lambda[1] + Lambda[2] - 2*delta
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**RCNonSimplyLacedElement**(*parent*, *rigged_partitions=[]*, ***options*)

Bases: *RiggedConfigurationElement*

Rigged configuration elements for non-simply-laced types.

e(a)

Return the action of e_a on `self`.

This works by lifting into the virtual configuration, then applying

$$e^v_a = \prod_{j \in \iota(a)} \hat{e}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: vct = CartanType(['C',2,1]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: elt = RC(partition_list=[[2],[1,1],[2]], rigging_list=[[-1],[-1,-1],[-
↪1]])
```

(continues on next page)

(continued from previous page)

```
sage: ascii_art(elt.e(0))
0[ ]0  -2[ ]-1  -2[ ][ ]-1
       -2[ ]-1
sage: ascii_art(elt.e(1))
-3[ ][ ]-2  0[ ]1  -3[ ][ ]-2
sage: ascii_art(elt.e(2))
-2[ ][ ]-1  -2[ ]-1  0[ ]0
            -2[ ]-1
```

```
>>> from sage.all import *
>>> vct = CartanType(['C',Integer(2),Integer(1)]).as_folding()
>>> RC = crystals.infinity.RiggedConfigurations(vct)
>>> elt = RC(partition_list=[[Integer(2)],[Integer(1),Integer(1)],
↪[Integer(2)]], rigging_list=[[-Integer(1)],[-Integer(1),-Integer(1)],[-
↪Integer(1)]])
>>> ascii_art(elt.e(Integer(0)))
0[ ]0  -2[ ]-1  -2[ ][ ]-1
       -2[ ]-1
>>> ascii_art(elt.e(Integer(1)))
-3[ ][ ]-2  0[ ]1  -3[ ][ ]-2
>>> ascii_art(elt.e(Integer(2)))
-2[ ][ ]-1  -2[ ]-1  0[ ]0
            -2[ ]-1
```

f(a)

Return the action of f_a on `self`.

This works by lifting into the virtual configuration, then applying

$$f_a^v = \prod_{j \in \iota(a)} \hat{f}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: vct = CartanType(['C',2,1]).as_folding()
sage: RC = crystals.infinity.RiggedConfigurations(vct)
sage: elt = RC(partition_list=[[2],[1,1],[2]], rigging_list=[[-1],[-1,-1],[-
↪1]])
sage: ascii_art(elt.f(0))
-4[ ][ ][ ]-2  -2[ ]-1  -2[ ][ ]-1
               -2[ ]-1
sage: ascii_art(elt.f(1))
-1[ ][ ]0  -2[ ][ ]-2  -1[ ][ ]0
           -2[ ]-1
sage: ascii_art(elt.f(2))
-2[ ][ ]-1  -2[ ]-1  -4[ ][ ][ ]-2
            -2[ ]-1
```

```
>>> from sage.all import *
>>> vct = CartanType(['C',Integer(2),Integer(1)]).as_folding()
>>> RC = crystals.infinity.RiggedConfigurations(vct)
```

(continues on next page)

(continued from previous page)

```
>>> elt = RC(partition_list=[[Integer(2)],[Integer(1),Integer(1)],
↪[Integer(2)]], rigging_list=[[-Integer(1)],[-Integer(1),-Integer(1)],[-
↪Integer(1)]])
>>> ascii_art(elt.f(Integer(0)))
-4[ ][ ][ ]-2  -2[ ]-1  -2[ ][ ]-1
               -2[ ]-1
>>> ascii_art(elt.f(Integer(1)))
-1[ ][ ]0  -2[ ][ ]-2  -1[ ][ ]0
           -2[ ]-1
>>> ascii_art(elt.f(Integer(2)))
-2[ ][ ]-1  -2[ ]-1  -4[ ][ ][ ]-2
            -2[ ]-1
```

to_virtual_configuration()

Return the corresponding rigged configuration in the virtual crystal.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: elt = RC(partition_list=[[3],[2]]); elt

0[ ][ ][ ]0

0[ ][ ]0
sage: elt.to_virtual_configuration()

0[ ][ ][ ]0

0[ ][ ][ ][ ]0

0[ ][ ][ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(2),Integer(1)], [[Integer(1),
↪Integer(2)],[Integer(1),Integer(1)],[Integer(2),Integer(1)]])
>>> elt = RC(partition_list=[[Integer(3)],[Integer(2)]]); elt
<BLANKLINE>
0[ ][ ][ ]0
<BLANKLINE>
0[ ][ ]0
>>> elt.to_virtual_configuration()
<BLANKLINE>
0[ ][ ][ ]0
<BLANKLINE>
0[ ][ ][ ][ ]0
<BLANKLINE>
0[ ][ ][ ]0
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**RiggedConfigurationElement**(*parent, rigged_partitions, **options*)

Bases: ClonableArray

A rigged configuration for simply-laced types.

For more information on rigged configurations, see *RiggedConfigurations*. For rigged configurations for non-simply-laced types, use *RCNonSimplyLacedElement*.

Typically to create a specific rigged configuration, the user will pass in the optional argument `partition_list` and if the user wants to specify the rigging values, give the optional argument `rigging_list` as well. If `rigging_list` is not passed, the rigging values are set to the corresponding vacancy numbers.

INPUT:

- `parent` – the parent of this element
- `rigged_partitions` – list of rigged partitions

There are two optional arguments to explicitly construct a rigged configuration. The first is `partition_list` which gives a list of partitions, and the second is `rigging_list` which is a list of corresponding lists of riggings. If only partition_list is specified, then it sets the rigging equal to the calculated vacancy numbers.

If we are constructing a rigged configuration from a rigged configuration (say of another type) and we don't want to recompute the vacancy numbers, we can use the `use_vacancy_numbers` to avoid the recomputation.

EXAMPLES:

Type $A_n^{(1)}$ examples:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RC(partition_list=[[2], [2, 2], [2], [2]])

0[ ][ ]0

-2[ ][ ]-2
-2[ ][ ]-2

2[ ][ ]2

-2[ ][ ]-2


sage: RC = RiggedConfigurations(['A', 4, 1], [[1, 1], [1, 1]])
sage: RC(partition_list=[[], [], [], []])

(/)

(/)

(/)

(/)
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),
↪Integer(2)]])
>>> RC(partition_list=[[Integer(2)], [Integer(2), Integer(2)], [Integer(2)],
↪[Integer(2)]])
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
-2[ ][ ]-2
-2[ ][ ]-2
<BLANKLINE>
2[ ][ ]2
<BLANKLINE>
-2[ ][ ]-2
<BLANKLINE>

>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(1), Integer(1)]])
>>> RC(partition_list=[[], [], [], []])
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
```

Type $D_n^{(1)}$ examples:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: RC(partition_list=[[3], [3,2], [4], [3]])

-1[ ][ ][ ]-1

1[ ][ ][ ]1
0[ ][ ]0

-3[ ][ ][ ][ ]-3

-1[ ][ ][ ]-1


sage: RC = RiggedConfigurations(['D', 4, 1], [[1, 1], [2, 1]])
sage: RC(partition_list=[[1], [1,1], [1], [1]])

1[ ]1

0[ ]0
0[ ]0

0[ ]0
```

(continues on next page)

(continued from previous page)

```
0[ ]0

sage: elt = RC(partition_list=[[1], [1,1], [1], [1]], rigging_list=[[0], [0,0],
↪[0], [0]]); elt

1[ ]0

0[ ]0
0[ ]0

0[ ]0

0[ ]0


sage: from sage.combinat.rigged_configurations.rigged_partition import
↪RiggedPartition
sage: RC2 = RiggedConfigurations(['D', 5, 1], [[2, 1], [3, 1]])
sage: l = [RiggedPartition()] + list(elt)
sage: ascii_art(RC2(*l))
(/)  1[ ]0  0[ ]0  0[ ]0  0[ ]0
            0[ ]0
sage: ascii_art(RC2(*l, use_vacancy_numbers=True))
(/)  1[ ]0  0[ ]0  0[ ]0  0[ ]0
            0[ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),
↪Integer(2)]])
>>> RC(partition_list=[[Integer(3)], [Integer(3),Integer(2)], [Integer(4)],
↪[Integer(3)]])
<BLANKLINE>
-1[ ][ ][ ]-1
<BLANKLINE>
1[ ][ ][ ]1
0[ ][ ]0
<BLANKLINE>
-3[ ][ ][ ][ ]-3
<BLANKLINE>
-1[ ][ ][ ]-1
<BLANKLINE>

>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(2), Integer(1)]])
>>> RC(partition_list=[[Integer(1)], [Integer(1),Integer(1)], [Integer(1)],
↪[Integer(1)]])
<BLANKLINE>
1[ ]1
<BLANKLINE>
0[ ]0
0[ ]0
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
>>> elt = RC(partition_list=[[Integer(1)], [Integer(1),Integer(1)], [Integer(1)],
→[Integer(1)]], rigging_list=[[Integer(0)], [Integer(0),Integer(0)],
→[Integer(0)], [Integer(0)]]); elt
<BLANKLINE>
1[ ]0
<BLANKLINE>
0[ ]0
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>

>>> from sage.combinat.rigged_configurations.rigged_partition import
→RiggedPartition
>>> RC2 = RiggedConfigurations(['D', Integer(5), Integer(1)], [[Integer(2),
→Integer(1)], [Integer(3), Integer(1)]])
>>> l = [RiggedPartition()] + list(elt)
>>> ascii_art(RC2(*l))
(/)  1[ ]0  0[ ]0  0[ ]0  0[ ]0
            0[ ]0
>>> ascii_art(RC2(*l, use_vacancy_numbers=True))
(/)  1[ ]0  0[ ]0  0[ ]0  0[ ]0
            0[ ]0
```

check()

Check the rigged configuration is properly defined.

There is nothing to check here.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 4])
sage: b = RC.module_generators[0].f_string([1,2,1,1,2,4,2,3,3,2])
sage: b.check()
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['A', Integer(4)])
>>> b = RC.module_generators[Integer(0)].f_string([Integer(1),Integer(2),
→Integer(1),Integer(1),Integer(2),Integer(4),Integer(2),Integer(3),
→Integer(3),Integer(2)])
>>> b.check()
```

e(a)

Return the action of the crystal operator e_a on `self`.

This implements the method defined in [CrysStructSchilling06] which finds the value k which is the length of the string with the smallest negative rigging of smallest length. Then it removes a box from a string of length

k in the a-th rigged partition, keeping all colabels fixed and increasing the new label by one. If no such string exists, then e_a is undefined.

This method can also be used when the underlying Cartan matrix is a Borcherds-Cartan matrix. In this case, then method of [SS2018] is used, where the new label is increased by half of the a-th diagonal entry of the underlying Borcherds-Cartan matrix. This method will also return `None` if a is imaginary and the smallest rigging in the a-th rigged partition is not exactly half of the a-th diagonal entry of the Borcherds-Cartan matrix.

INPUT:

- `a` – the index of the partition to remove a box

OUTPUT: the resulting rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2,1]])
sage: elt = RC(partition_list=[[1], [1], [1], [1]])
sage: elt.e(3)
sage: elt.e(1)

(/)

0[ ]0

0[ ]0

-1[ ]-1


sage: A = CartanMatrix([[-2,-1],[-1,-2]], borcherds=True)
sage: RC = crystals.infinity.RiggedConfigurations(A)
sage: nu0 = RC(partition_list=[[],[]])
sage: nu = nu0.f_string([1,0,0,0])
sage: ascii_art(nu.e(0))
5[ ]3  4[ ]3
5[ ]1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),
↪Integer(1)]])
>>> elt = RC(partition_list=[[Integer(1)], [Integer(1)], [Integer(1)],␣
↪[Integer(1)]])
>>> elt.e(Integer(3))
>>> elt.e(Integer(1))
<BLANKLINE>
(/)
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
>>> A = CartanMatrix([[-Integer(2),-Integer(1)],[-Integer(1),-Integer(2)]],
↪borcherds=True)
>>> RC = crystals.infinity.RiggedConfigurations(A)
>>> nu0 = RC(partition_list=[[],[]])
>>> nu = nu0.f_string([Integer(1),Integer(0),Integer(0),Integer(0)])
>>> ascii_art(nu.e(Integer(0)))
5[ ]3  4[ ]3
5[ ]1
```

epsilon(*a*)

Return ε_a of `self`.

Let x_ℓ be the smallest string of $\nu^{(a)}$ or 0 if $\nu^{(a)} = \emptyset$, then we have $\varepsilon_a = -\min(0, x_\ell)$.

EXAMPLES:

```
sage: La = RootSystem(['B',2]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[1]+La[2])
sage: I = RC.index_set()
sage: matrix([[rc.epsilon(i) for i in I] for rc in RC[:4]])
[0 0]
[1 0]
[0 1]
[0 2]
```

```
>>> from sage.all import *
>>> La = RootSystem(['B',Integer(2)]).weight_lattice().fundamental_weights()
>>> RC = crystals.RiggedConfigurations(La[Integer(1)]+La[Integer(2)])
>>> I = RC.index_set()
>>> matrix([[rc.epsilon(i) for i in I] for rc in RC[:Integer(4)]])
[0 0]
[1 0]
[0 1]
[0 2]
```

f(*a*)

Return the action of the crystal operator f_a on `self`.

This implements the method defined in [CrysStructSchilling06] which finds the value k which is the length of the string with the smallest nonpositive rigging of largest length. Then it adds a box from a string of length k in the a-th rigged partition, keeping all colabels fixed and decreasing the new label by one. If no such string exists, then it adds a new string of length 1 with label -1. However we need to modify the definition to work for $B(\infty)$ by removing the condition that the resulting rigged configuration is valid.

This method can also be used when the underlying Cartan matrix is a Borcherds-Cartan matrix. In this case, then method of [SS2018] is used, where the new label is decreased by half of the a-th diagonal entry of the underlying Borcherds-Cartan matrix.

INPUT:

- `a` – the index of the partition to add a box

OUTPUT: the resulting rigged configuration element

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3])
sage: nu0 = RC.module_generators[0]
sage: nu0.f(2)

(/)

-2[ ]-1

(/)


sage: A = CartanMatrix([[-2,-1],[-1,-2]], borcherds=True)
sage: RC = crystals.infinity.RiggedConfigurations(A)
sage: nu0 = RC(partition_list=[[],[]])
sage: nu = nu0.f_string([1,0,0,0])
sage: ascii_art(nu.f(0))
9[ ]7  6[ ]5
9[ ]5
9[ ]3
9[ ]1
```

```
>>> from sage.all import *
>>> RC = crystals.infinity.RiggedConfigurations(['A', Integer(3)])
>>> nu0 = RC.module_generators[Integer(0)]
>>> nu0.f(Integer(2))
<BLANKLINE>
(/)
<BLANKLINE>
-2[ ]-1
<BLANKLINE>
(/)
<BLANKLINE>

>>> A = CartanMatrix([[-Integer(2),-Integer(1)],[-Integer(1),-Integer(2)]],
↪borcherds=True)
>>> RC = crystals.infinity.RiggedConfigurations(A)
>>> nu0 = RC(partition_list=[[],[]])
>>> nu = nu0.f_string([Integer(1),Integer(0),Integer(0),Integer(0)])
>>> ascii_art(nu.f(Integer(0)))
9[ ]7  6[ ]5
9[ ]5
9[ ]3
9[ ]1
```

nu()

Return the list ν of rigged partitions of this rigged configuration element.

OUTPUT: the ν array as a list

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RC(partition_list=[[2], [2,2], [2], [2]]).nu()
```

(continues on next page)

(continued from previous page)

```
[0[ ][ ]0
 , -2[ ][ ]-2
 -2[ ][ ]-2
 , 2[ ][ ]2
 , -2[ ][ ]-2
 ]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),
→Integer(2)]])
>>> RC(partition_list=[[Integer(2)], [Integer(2),Integer(2)], [Integer(2)],
→[Integer(2)]]).nu()
[0[ ][ ]0
 , -2[ ][ ]-2
 -2[ ][ ]-2
 , 2[ ][ ]2
 , -2[ ][ ]-2
 ]
```

partition_rigging_lists()

Return the list of partitions and the associated list of riggings of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',3,1], [[1,2],[2,2]])
sage: rc = RC(partition_list=[[2],[1],[1]], rigging_list=[[-1],[0],[-1]]); rc

-1[ ][ ]-1

1[ ]0

-1[ ]-1

sage: rc.partition_rigging_lists()
[[[2], [1], [1]], [[-1], [0], [-1]]]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(3),Integer(1)], [[Integer(1),
→Integer(2)],[Integer(2),Integer(2)]])
>>> rc = RC(partition_list=[[Integer(2)],[Integer(1)],[Integer(1)]], rigging_
→list=[[-Integer(1)],[Integer(0)],[-Integer(1)]]); rc
<BLANKLINE>
-1[ ][ ]-1
<BLANKLINE>
1[ ]0
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
>>> rc.partition_rigging_lists()
[[[2], [1], [1]], [[-1], [0], [-1]]]
```

phi(*a*)

Return φ_a of `self`.

Let x_ℓ be the smallest string of $\nu^{(a)}$ or 0 if $\nu^{(a)} = \emptyset$, then we have $\varepsilon_a = p_\infty^{(a)} - \min(0, x_\ell)$.

EXAMPLES:

```
sage: La = RootSystem(['B',2]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[1]+La[2])
sage: I = RC.index_set()
sage: matrix([[rc.phi(i) for i in I] for rc in RC[:4]])
[1 1]
[0 3]
[0 2]
[1 1]
```

```
>>> from sage.all import *
>>> La = RootSystem(['B',Integer(2)]).weight_lattice().fundamental_weights()
>>> RC = crystals.RiggedConfigurations(La[Integer(1)]+La[Integer(2)])
>>> I = RC.index_set()
>>> matrix([[rc.phi(i) for i in I] for rc in RC[:Integer(4)]])
[1 1]
[0 3]
[0 2]
[1 1]
```

vacancy_number(a, i)

Return the vacancy number $p_i^{(a)}$.

INPUT:

- a – the index of the rigged partition
- i – the row of the rigged partition

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: elt = RC(partition_list=[[1], [2,1], [1], []])
sage: elt.vacancy_number(2, 3)
-2
sage: elt.vacancy_number(2, 2)
-2
sage: elt.vacancy_number(2, 1)
-1

sage: RC = RiggedConfigurations(['D',4,1], [[2,1], [2,1]])
sage: x = RC(partition_list=[[3], [3,1,1], [2], [3,1]]); ascii_art(x)
-1[ ][ ][ ]-1  1[ ][ ][ ]1  0[ ][ ]0  -3[ ][ ][ ]-3
               0[ ]0                  -1[ ]-1
               0[ ]0
sage: x.vacancy_number(2,2)
1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),␣
→Integer(2)]])
>>> elt = RC(partition_list=[[Integer(1)], [Integer(2),Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(1)], []])
>>> elt.vacancy_number(Integer(2), Integer(3))
-2
>>> elt.vacancy_number(Integer(2), Integer(2))
-2
>>> elt.vacancy_number(Integer(2), Integer(1))
-1

>>> RC = RiggedConfigurations(['D',Integer(4),Integer(1)], [[Integer(2),
↪Integer(1)], [Integer(2),Integer(1)]])
>>> x = RC(partition_list=[[Integer(3)], [Integer(3),Integer(1),Integer(1)],␣
↪[Integer(2)], [Integer(3),Integer(1)]]); ascii_art(x)
-1[ ][ ][ ]-1  1[ ][ ][ ]1  0[ ][ ]0  -3[ ][ ][ ]-3
               0[ ]0                  -1[ ]-1
               0[ ]0
>>> x.vacancy_number(Integer(2),Integer(2))
1
```

5.1.215 Rigged Configurations

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

`sage.combinat.rigged_configurations.rigged_configurations.`**`KirillovReshetikhinCrystal`**(*cartan_type*, *r*, *s*)

Return the KR crystal $B^{r,s}$ using *rigged configurations*.

This is the rigged configuration $RC(B^{r,s})$ or $RC(L)$ with $L = (L_i^{(a)})$ and $L_i^{(a)} = \delta_{a,r}\delta_{i,s}$.

EXAMPLES:

```
sage: K1 = crystals.kirillov_reshetikhin.RiggedConfigurations(['A',6,2], 2, 1)
sage: K2 = crystals.kirillov_reshetikhin.LSPaths(['A',6,2], 2, 1)
sage: K1.digraph().is_isomorphic(K2.digraph(), edge_labels=True)
True
```

```
>>> from sage.all import *
>>> K1 = crystals.kirillov_reshetikhin.RiggedConfigurations(['A',Integer(6),
↪Integer(2)], Integer(2), Integer(1))
>>> K2 = crystals.kirillov_reshetikhin.LSPaths(['A',Integer(6),Integer(2)],␣
↪Integer(2), Integer(1))
>>> K1.digraph().is_isomorphic(K2.digraph(), edge_labels=True)
True
```

class `sage.combinat.rigged_configurations.rigged_configurations.`**`RCNonSimplyLaced`**(*cartan_type*, *dims*)

Bases: *RiggedConfigurations*

Rigged configurations in non-simply-laced types.

These are rigged configurations which lift to virtual rigged configurations in a simply-laced type.

For more on rigged configurations, see *RiggedConfigurations*.

Element

alias of *KRRCNonSimplyLacedElement*

from_virtual(*vrc*)

Convert `vrc` in the virtual crystal into a rigged configuration of the original Cartan type.

INPUT:

- `vrc` – a virtual rigged configuration

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: elt = RC(partition_list=[[3],[2]])
sage: vrc_elt = RC.to_virtual(elt)
sage: ret = RC.from_virtual(vrc_elt); ret

0[ ][ ][ ]0

0[ ][ ]0
sage: ret == elt
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(2),Integer(1)], [[Integer(1),
↪Integer(2)],[Integer(1),Integer(1)],[Integer(2),Integer(1)]])
>>> elt = RC(partition_list=[[Integer(3)],[Integer(2)]])
>>> vrc_elt = RC.to_virtual(elt)
>>> ret = RC.from_virtual(vrc_elt); ret
<BLANKLINE>
0[ ][ ][ ]0
<BLANKLINE>
0[ ][ ]0
>>> ret == elt
True
```

kleber_tree()

Return the underlying (virtual) Kleber tree used to generate all highest weight rigged configurations.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',3,1], [[1,1], [2,1]])
sage: RC.kleber_tree()
Virtual Kleber tree of Cartan type ['C', 3, 1] and B = ((1, 1), (2, 1))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(3),Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(2),Integer(1)]])
>>> RC.kleber_tree()
Virtual Kleber tree of Cartan type ['C', 3, 1] and B = ((1, 1), (2, 1))
```

module_generators()

Module generators for this set of rigged configurations.

Iterate over the highest weight rigged configurations by moving through the `KleberTree` and then setting appropriate values of the partitions.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C', 3, 1], [[1,2]])
sage: for x in RC.module_generators: x

(/)

(/)

(/)


0[ ][ ]0

0[ ][ ]0

0[ ]0


sage: RC = RiggedConfigurations(['D',4,3], [[1,1]])
sage: RC.module_generators
(

     0[ ]0
(/)  0[ ]0

(/)  0[ ]0
   ,
)
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C', Integer(3), Integer(1)], [[Integer(1),
↪Integer(2)]])
>>> for x in RC.module_generators: x
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>

>>> RC = RiggedConfigurations(['D',Integer(4),Integer(3)], [[Integer(1),
↪Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> RC.module_generators
(
<BLANKLINE>
     0[ ]0
(/)  0[ ]0
<BLANKLINE>
(/)  0[ ]0
   ,
)
```

to_virtual(*rc*)

Convert `rc` into a rigged configuration in the virtual crystal.

INPUT:

- `rc` – a rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: elt = RC(partition_list=[[3],[2]]); elt

0[ ][ ][ ]0

0[ ][ ]0
sage: velt = RC.to_virtual(elt); velt

0[ ][ ][ ]0

0[ ][ ][ ][ ]0

0[ ][ ][ ]0
sage: velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1,␣
↪1), (3, 1), (2, 2))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(2),Integer(1)], [[Integer(1),
↪Integer(2)],[Integer(1),Integer(1)],[Integer(2),Integer(1)]])
>>> elt = RC(partition_list=[[Integer(3)],[Integer(2)]]); elt
<BLANKLINE>
0[ ][ ][ ]0
<BLANKLINE>
0[ ][ ]0
>>> velt = RC.to_virtual(elt); velt
<BLANKLINE>
0[ ][ ][ ]0
<BLANKLINE>
0[ ][ ][ ][ ]0
<BLANKLINE>
0[ ][ ][ ]0
>>> velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1,␣
```

(continues on next page)

(continued from previous page)

```
↪1), (3, 1), (2, 2))
```

virtual()

Return the corresponding virtual crystal.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: RC
Rigged configurations of type ['C', 2, 1] and factor(s) ((1, 2), (1, 1), (2,␣
↪1))
sage: RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1,␣
↪1), (3, 1), (2, 2))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['C',Integer(2),Integer(1)], [[Integer(1),
↪Integer(2)],[Integer(1),Integer(1)],[Integer(2),Integer(1)]])
>>> RC
Rigged configurations of type ['C', 2, 1] and factor(s) ((1, 2), (1, 1), (2,␣
↪1))
>>> RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1,␣
↪1), (3, 1), (2, 2))
```

class sage.combinat.rigged_configurations.rigged_configurations.**RCTypeA2Dual**(*cartan_type*, *dims*)

Bases: *RCTypeA2Even*

Rigged configurations of type $A_{2n}^{(2)\dagger}$.

For more on rigged configurations, see *RiggedConfigurations*.

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,2],[1,1],[2,1]])
sage: RC
Rigged configurations of type ['BC', 2, 2]^* and factor(s) ((1, 2), (1, 1), (2,␣
↪1))
sage: RC.cardinality()
750
sage: RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1, 1),␣
↪(3, 1), (2, 1), (2, 1))
sage: RC = RiggedConfigurations(CartanType(['A',2,2]).dual(), [[1,1]])
sage: RC.cardinality()
3
sage: RC = RiggedConfigurations(CartanType(['A',2,2]).dual(), [[1,2],[1,1]])
sage: TestSuite(RC).run() # long time
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[2,1]])
sage: TestSuite(RC).run() # long time
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),
↪[[Integer(1),Integer(2)],[Integer(1),Integer(1)],[Integer(2),Integer(1)]])
>>> RC
Rigged configurations of type ['BC', 2, 2]^* and factor(s) ((1, 2), (1, 1), (2,
↪1))
>>> RC.cardinality()
750
>>> RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1, 1),
↪(3, 1), (2, 1), (2, 1))
>>> RC = RiggedConfigurations(CartanType(['A',Integer(2),Integer(2)]).dual(),
↪[[Integer(1),Integer(1)]])
>>> RC.cardinality()
3
>>> RC = RiggedConfigurations(CartanType(['A',Integer(2),Integer(2)]).dual(),
↪[[Integer(1),Integer(2)],[Integer(1),Integer(1)]])
>>> TestSuite(RC).run() # long time
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),
↪[[Integer(2),Integer(1)]])
>>> TestSuite(RC).run() # long time
```

Element

alias of *KRRCTypeA2DualElement*

from_virtual(*vrc*)

Convert `vrc` in the virtual crystal into a rigged configuration of the original Cartan type.

INPUT:

- `vrc` – a virtual rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[2,2]])
sage: elt = RC(partition_list=[[1],[1]])
sage: velt = RC.to_virtual(elt)
sage: ret = RC.from_virtual(velt); ret

-1[ ]-1

1[ ]1

sage: ret == elt
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),
↪[[Integer(2),Integer(2)]])
>>> elt = RC(partition_list=[[Integer(1)],[Integer(1)]])
>>> velt = RC.to_virtual(elt)
>>> ret = RC.from_virtual(velt); ret
<BLANKLINE>
-1[ ]-1
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
1[ ]1
<BLANKLINE>
>>> ret == elt
True
```

module_generators()

Module generators for rigged configurations of type $A_{2n}^{(2)\dagger}$.

Iterate over the highest weight rigged configurations by moving through the `KleberTree` and then setting appropriate values of the partitions. This also skips rigged configurations where $P_i^{(n)} < 1$ when i is odd.

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A', 4, 2]).dual(), [[1,1]])
sage: for x in RC.module_generators: x

(/)

(/)
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(CartanType(['A', Integer(4), Integer(2)]).
↪dual(), [[Integer(1),Integer(1)]])
>>> for x in RC.module_generators: x
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
```

to_virtual(*rc*)

Convert `rc` into a rigged configuration in the virtual crystal.

INPUT:

- `rc` – a rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[2,2]])
sage: elt = RC(partition_list=[[1],[1]]); elt

-1[ ]-1

1[ ]1

sage: velt = RC.to_virtual(elt); velt

-1[ ]-1

2[ ]2

-1[ ]-1
```

(continues on next page)

(continued from previous page)

```
sage: velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 2), (2, 2))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),␣
↪[[Integer(2),Integer(2)]])
>>> elt = RC(partition_list=[[Integer(1)],[Integer(1)]]); elt
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
1[ ]1
<BLANKLINE>
>>> velt = RC.to_virtual(elt); velt
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
2[ ]2
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
>>> velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 2), (2, 2))
```

class sage.combinat.rigged_configurations.rigged_configurations.**RCTypeA2Even**(*cartan_type*, *dims*)

Bases: *RCNonSimplyLaced*

Rigged configurations for type $A_{2n}^{(2)}$.

For more on rigged configurations, see *RiggedConfigurations*.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[2,1], [1,2]])
sage: RC.cardinality()
150
sage: RC = RiggedConfigurations(['A',2,2], [[1,1]])
sage: RC.cardinality()
3
sage: RC = RiggedConfigurations(['A',2,2], [[1,2],[1,1]])
sage: TestSuite(RC).run() # long time
sage: RC = RiggedConfigurations(['A',4,2], [[2,1]])
sage: TestSuite(RC).run() # long time
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(4),Integer(2)], [[Integer(2),
↪Integer(1)], [Integer(1),Integer(2)]])
>>> RC.cardinality()
150
>>> RC = RiggedConfigurations(['A',Integer(2),Integer(2)], [[Integer(1),
↪Integer(1)]])
>>> RC.cardinality()
```

(continues on next page)

(continued from previous page)

```
3
>>> RC = RiggedConfigurations(['A',Integer(2),Integer(2)], [[Integer(1),
↪Integer(2)],[Integer(1),Integer(1)]])
>>> TestSuite(RC).run() # long time
>>> RC = RiggedConfigurations(['A',Integer(4),Integer(2)], [[Integer(2),
↪Integer(1)]])
>>> TestSuite(RC).run() # long time
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[1,1], [2,2]])
sage: RC.cardinality()
250
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(4),Integer(2)], [[Integer(1),
↪Integer(1)], [Integer(2),Integer(2)]])
>>> RC.cardinality()
250
```

from_virtual(*vrc*)

Convert `vrc` in the virtual crystal into a rigged configuration of the original Cartan type.

INPUT:

- `vrc` – a virtual rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[2,2]])
sage: elt = RC(partition_list=[[1],[1]])
sage: velt = RC.to_virtual(elt)
sage: ret = RC.from_virtual(velt); ret

-1[ ]-1

1[ ]1

sage: ret == elt
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(4),Integer(2)], [[Integer(2),
↪Integer(2)]])
>>> elt = RC(partition_list=[[Integer(1)],[Integer(1)]])
>>> velt = RC.to_virtual(elt)
>>> ret = RC.from_virtual(velt); ret
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
1[ ]1
<BLANKLINE>
>>> ret == elt
True
```

to_virtual(*rc*)

Convert `rc` into a rigged configuration in the virtual crystal.

INPUT:

- `rc` – a rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[2,2]])
sage: elt = RC(partition_list=[[1],[1]]); elt

-1[ ]-1

1[ ]1

sage: velt = RC.to_virtual(elt); velt

-1[ ]-1

2[ ]2

-1[ ]-1

sage: velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 2), (2, 2))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(4),Integer(2)], [[Integer(2),
↪Integer(2)]])
>>> elt = RC(partition_list=[[Integer(1)],[Integer(1)]]); elt
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
1[ ]1
<BLANKLINE>
>>> velt = RC.to_virtual(elt); velt
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
2[ ]2
<BLANKLINE>
-1[ ]-1
<BLANKLINE>
>>> velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 2), (2, 2))
```

virtual()

Return the corresponding virtual crystal.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[1,2],[1,1],[2,1]])
sage: RC
Rigged configurations of type ['BC', 2, 2] and factor(s) ((1, 2), (1, 1), (2,
↪1))
sage: RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1,
↪1), (3, 1), (2, 1), (2, 1))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(4),Integer(2)], [[Integer(1),
↪Integer(2)],[Integer(1),Integer(1)],[Integer(2),Integer(1)]])
>>> RC
Rigged configurations of type ['BC', 2, 2] and factor(s) ((1, 2), (1, 1), (2,
↪1))
>>> RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1,
↪1), (3, 1), (2, 1), (2, 1))
```

class sage.combinat.rigged_configurations.rigged_configurations.**RiggedConfigurations**(*cartan_type*, *B*)

Bases: UniqueRepresentation, Parent

Rigged configurations as $U_q'(\mathfrak{g})$-crystals.

Let $\overline{I}$ denote the classical index set associated to the Cartan type of the rigged configurations. A rigged configuration of multiplicity array $L_i^{(a)}$ and dominant weight Λ is a sequence of partitions $\{\nu^{(a)} \mid a \in \overline{I}\}$ such that

$$\sum_{\overline{I} \times \mathbf{Z}_{>0}} i m_i^{(a)} \alpha_a = \sum_{\overline{I} \times \mathbf{Z}_{>0}} i L_i^{(a)} \Lambda_a - \Lambda$$

where α_a is a simple root, Λ_a is a fundamental weight, and $m_i^{(a)}$ is the number of rows of length i in the partition $\nu^{(a)}$.

Each partition $\nu^{(a)}$, in the sequence also comes with a sequence of statistics $p_i^{(a)}$ called *vacancy numbers* and a weakly decreasing sequence $J_i^{(a)}$ of length $m_i^{(a)}$ called *riggings*. Vacancy numbers are computed based upon the partitions and $L_i^{(a)}$, and the riggings must satisfy $\max J_i^{(a)} \leq p_i^{(a)}$. We call such a partition a *rigged partition*. For more, see [RigConBijection] [CrysStructSchilling06] [BijectionLRT].

Rigged configurations form combinatorial objects first introduced by Kerov, Kirillov and Reshetikhin that arose from studies of statistical mechanical models using the Bethe Ansatz. They are sequences of rigged partitions. A rigged partition is a partition together with a label associated to each part that satisfy certain constraints. The labels are also called riggings.

Rigged configurations exist for all affine Kac-Moody Lie algebras. See for example [HKOTT2002]. In Sage they are specified by providing a Cartan type and a list of rectangular shapes B. The list of all (highest weight) rigged configurations for given B is computed via the (virtual) Kleber algorithm (see also *KleberTree* and *VirtualKleberTree*).

Rigged configurations in simply-laced types all admit a classical crystal structure [CrysStructSchilling06]. For non-simply-laced types, the crystal is given by using virtual rigged configurations [OSS03]. The highest weight rigged configurations are those where all riggings are nonnegative. The list of all rigged configurations is computed from the highest weight ones using the crystal operators.

Rigged configurations are conjecturally in bijection with `TensorProductOfKirillovReshetikhinTableaux` of non-exceptional affine types where the list B corresponds to the tensor factors $B^{r,s}$. The bijection has been proven in types $A_n^{(1)}$ and $D_n^{(1)}$ and when the only nonzero entries of $L_i^{(a)}$ are either only $L_1^{(a)}$ or only $L_i^{(1)}$ (corresponding to single columns or rows respectively) [RigConBijection], [BijectionLRT], [BijectionDn].

KR crystals are implemented in Sage, see `KirillovReshetikhinCrystal()`, however, in the bijection with rigged configurations a different realization of the elements in the crystal are obtained, which are coined KR tableaux, see `KirillovReshetikhinTableaux`. For more details see [OSS2011].

Note

All non-simply-laced rigged configurations have not been proven to give rise to aligned virtual crystals (i.e. have the correct crystal structure or isomorphic as affine crystals to the tensor product of KR tableaux).

INPUT:

- `cartan_type` – a Cartan type
- `B` – list of positive integer tuples (r, s) corresponding to the tensor factors in the bijection with tensor product of Kirillov-Reshetikhin tableaux or equivalently the sequence of width s and height r rectangles

REFERENCES:

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2], [1, 1]])
sage: RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((3, 2), (1, 2), (1, 1))

sage: RC = RiggedConfigurations(['A', 3, 1], [[2,1]]); RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 1),)
sage: RC.cardinality()
6
sage: len(RC.list()) == RC.cardinality()
True
sage: RC.list() # random
[

                                                0[ ]0
(/)   (/)       (/)       -1[ ]-1   -1[ ]-1
                                                -1[ ]-1
(/)   -1[ ]-1   0[ ]0     0[ ]0     1[ ]1     -1[ ]-1

(/)   (/)       -1[ ]-1   (/)       -1[ ]-1   0[ ]0
   ,         ,         ,         ,         ,
]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),␣
↪Integer(2)], [Integer(1), Integer(2)], [Integer(1), Integer(1)]])
>>> RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((3, 2), (1, 2), (1, 1))

>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)]]); RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 1),)
>>> RC.cardinality()
6
>>> len(RC.list()) == RC.cardinality()
True
>>> RC.list() # random
[
<BLANKLINE>
                                            0[ ]0
(/)   (/)       (/)       -1[ ]-1   -1[ ]-1
                                            -1[ ]-1
(/)   -1[ ]-1   0[ ]0     0[ ]0     1[ ]1     -1[ ]-1
<BLANKLINE>
(/)   (/)       -1[ ]-1   (/)       -1[ ]-1   0[ ]0
    ,         ,         ,         ,         ,
]
```

A rigged configuration element with all riggings equal to the vacancy numbers can be created as follows:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3,2], [2,1], [1,1], [1,1]]); RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((3, 2), (2, 1), (1, 1),␣
↪(1, 1))
sage: elt = RC(partition_list=[[1],[],[]]); elt

0[ ]0

(/)

(/)
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),
↪Integer(2)], [Integer(2),Integer(1)], [Integer(1),Integer(1)], [Integer(1),
↪Integer(1)]]); RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((3, 2), (2, 1), (1, 1),␣
↪(1, 1))
>>> elt = RC(partition_list=[[Integer(1)],[],[]]); elt
<BLANKLINE>
0[ ]0
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
```

If on the other hand we also want to specify the riggings, this can be achieved as follows:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2], [1, 1]])
sage: RC(partition_list=[[2],[2],[2]])

1[ ][ ]1
```

(continues on next page)

(continued from previous page)

```
0[ ][ ]0

0[ ][ ]0
sage: RC(partition_list=[[2],[2],[2]], rigging_list=[[0],[0],[0]])

1[ ][ ]0

0[ ][ ]0

0[ ][ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),␣
↪Integer(2)], [Integer(1), Integer(2)], [Integer(1), Integer(1)]])
>>> RC(partition_list=[[Integer(2)],[Integer(2)],[Integer(2)]])
<BLANKLINE>
1[ ][ ]1
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
0[ ][ ]0
>>> RC(partition_list=[[Integer(2)],[Integer(2)],[Integer(2)]], rigging_
↪list=[[Integer(0)],[Integer(0)],[Integer(0)]])
<BLANKLINE>
1[ ][ ]0
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
0[ ][ ]0
```

A larger example:

```
sage: RC = RiggedConfigurations(['D', 7, 1], [[3,3],[5,2],[4,3],[2,3],[4,4],[3,1],
↪[1,4],[2,2]])
sage: elt = RC(partition_list=[[2],[3,2,1],[2,2,1,1],[2,2,1,1,1,1],[3,2,1,1,1,1],
↪[2,1,1],[2,2]],
....:           rigging_list=[[2],[1,0,0],[4,1,2,1],[1,0,0,0,0,0],[0,1,0,0,0,0],[0,
↪0,0],[0,0]])
sage: elt

3[ ][ ]2

1[ ][ ][ ]1
2[ ][ ]0
1[ ]0

4[ ][ ]4
4[ ][ ]1
3[ ]2
3[ ]1
```

(continues on next page)

(continued from previous page)

```
2[ ][ ]1
2[ ][ ]0
0[ ]0
0[ ]0
0[ ]0
0[ ]0

0[ ][ ][ ]0
2[ ][ ]1
0[ ]0
0[ ]0
0[ ]0
0[ ]0

0[ ][ ]0
0[ ]0
0[ ]0

0[ ][ ]0
0[ ][ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(7), Integer(1)], [[Integer(3),
↪Integer(3)],[Integer(5),Integer(2)],[Integer(4),Integer(3)],[Integer(2),
↪Integer(3)],[Integer(4),Integer(4)],[Integer(3),Integer(1)],[Integer(1),
↪Integer(4)],[Integer(2),Integer(2)]])
>>> elt = RC(partition_list=[[Integer(2)],[Integer(3),Integer(2),Integer(1)],
↪[Integer(2),Integer(2),Integer(1),Integer(1)],[Integer(2),Integer(2),Integer(1),
↪Integer(1),Integer(1),Integer(1)],[Integer(3),Integer(2),Integer(1),Integer(1),
↪Integer(1),Integer(1)],[Integer(2),Integer(1),Integer(1)],[Integer(2),
↪Integer(2)]],
...          rigging_list=[[Integer(2)],[Integer(1),Integer(0),Integer(0)],
↪[Integer(4),Integer(1),Integer(2),Integer(1)],[Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(0)]])
>>> elt
<BLANKLINE>
3[ ][ ]2
<BLANKLINE>
1[ ][ ][ ]1
2[ ][ ]0
1[ ]0
<BLANKLINE>
4[ ][ ]4
4[ ][ ]1
3[ ]2
3[ ]1
<BLANKLINE>
2[ ][ ]1
2[ ][ ]0
0[ ]0
```

(continues on next page)

(continued from previous page)

```
0[ ]0
0[ ]0
0[ ]0
<BLANKLINE>
0[ ][ ][ ]0
2[ ][ ]1
0[ ]0
0[ ]0
0[ ]0
0[ ]0
<BLANKLINE>
0[ ][ ]0
0[ ]0
0[ ]0
<BLANKLINE>
0[ ][ ]0
0[ ][ ]0
<BLANKLINE>
```

To obtain the KR tableau under the bijection between rigged configurations and KR tableaux, we can type the following. This example was checked against Reiho Sakamoto's Mathematica program on rigged configurations:

```
sage: output = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); output
[[1, 1, 1], [2, 3, 3], [3, 4, -5]] (X) [[1, 1], [2, 2], [3, 3], [5, -6], [6, -5]]
↪(X)
[[1, 1, 2], [2, 2, 3], [3, 3, 7], [4, 4, -7]] (X) [[1, 1, 1], [2, 2, 2]] (X)
[[1, 1, 1, 3], [2, 2, 3, 4], [3, 3, 4, 5], [4, 4, 5, 6]] (X) [[1], [2], [3]] (X)
↪[[1, 1, 1, 1]] (X) [[1, 1], [2, 2]]
sage: elt.to_tensor_product_of_kirillov_reshetikhin_tableaux().to_rigged_
↪configuration() == elt
True
sage: output.to_rigged_configuration().to_tensor_product_of_kirillov_reshetikhin_
↪tableaux() == output
True
```

```
>>> from sage.all import *
>>> output = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); output
[[1, 1, 1], [2, 3, 3], [3, 4, -5]] (X) [[1, 1], [2, 2], [3, 3], [5, -6], [6, -5]]
↪(X)
[[1, 1, 2], [2, 2, 3], [3, 3, 7], [4, 4, -7]] (X) [[1, 1, 1], [2, 2, 2]] (X)
[[1, 1, 1, 3], [2, 2, 3, 4], [3, 3, 4, 5], [4, 4, 5, 6]] (X) [[1], [2], [3]] (X)
↪[[1, 1, 1, 1]] (X) [[1, 1], [2, 2]]
>>> elt.to_tensor_product_of_kirillov_reshetikhin_tableaux().to_rigged_
↪configuration() == elt
True
>>> output.to_rigged_configuration().to_tensor_product_of_kirillov_reshetikhin_
↪tableaux() == output
True
```

We can also convert between rigged configurations and tensor products of KR crystals:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 1]])
```

(continues on next page)

(continued from previous page)

```
sage: elt = RC(partition_list=[[1],[1,1],[1],[1]])
sage: tp_krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krc
[[]]
sage: ret = RC(tp_krc)
sage: ret == elt
True
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),
↪Integer(1)]])
>>> elt = RC(partition_list=[[Integer(1)],[Integer(1),Integer(1)],[Integer(1)],
↪[Integer(1)]])
>>> tp_krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krc
[[]]
>>> ret = RC(tp_krc)
>>> ret == elt
True
```

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[4,1], [3,3]])
sage: KR1 = crystals.KirillovReshetikhin(['D', 4, 1], 4, 1)
sage: KR2 = crystals.KirillovReshetikhin(['D', 4, 1], 3, 3)
sage: T = crystals.TensorProduct(KR1, KR2)
sage: t = T[1]; t
[[++++, []], [+++-, [[1], [2], [4], [-4]]]]
sage: ret = RC(t)
sage: ret.to_tensor_product_of_kirillov_reshetikhin_crystals()
[[++++, []], [+++-, [[1], [2], [4], [-4]]]]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(4),
↪Integer(1)], [Integer(3),Integer(3)]])
>>> KR1 = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)], Integer(4),
↪Integer(1))
>>> KR2 = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)], Integer(3),
↪Integer(3))
>>> T = crystals.TensorProduct(KR1, KR2)
>>> t = T[Integer(1)]; t
[[++++, []], [+++-, [[1], [2], [4], [-4]]]]
>>> ret = RC(t)
>>> ret.to_tensor_product_of_kirillov_reshetikhin_crystals()
[[++++, []], [+++-, [[1], [2], [4], [-4]]]]
```

Element

alias of *KRRCSimplyLacedElement*

classically_highest_weight_vectors()

Return the classically highest weight elements of `self`.

fermionic_formula(*q=None*, *only_highest_weight=False*, *weight=None*)

Return the fermionic formula associated to `self`.

Given a set of rigged configurations $RC(\lambda, L)$, the fermionic formula is defined as:

$$M(\lambda, L; q) = \sum_{(\nu, J)} q^{cc(\nu, J)}$$

where we sum over all (classically highest weight) rigged configurations of weight λ where cc is the `cocharge statistic`. This is known to reduce to

$$M(\lambda, L; q) = \sum_{\nu} q^{cc(\nu)} \prod_{(a,i) \in I \times \mathbf{Z}} \begin{bmatrix} p_i^{(a)} + m_i^{(a)} \\ m_i^{(a)} \end{bmatrix}_q .$$

The generating function of $M(\lambda, L; q)$ in the weight algebra subsumes all fermionic formulas:

$$M(L; q) = \sum_{\lambda \in P} M(\lambda, L; q) \lambda.$$

This is conjecturally equal to the `one dimensional configuration sum` of the corresponding tensor product of Kirillov-Reshetikhin crystals, see [HKOTT2002]. This has been proven in general for type $A_n^{(1)}$ [BijectionLRT], single factors $B^{r,s}$ in type $D_n^{(1)}$ [OSS2011] with the result from [Sakamoto13], as well as for a tensor product of single columns [OSS2003], [BijectionDn] or a tensor product of single rows [OSS03] for all non-exceptional types.

INPUT:

- `q` – the variable q
- `only_highest_weight` – use only the classically highest weight rigged configurations
- `weight` – return the fermionic formula $M(\lambda, L; q)$ where λ is the classical weight `weight`

REFERENCES:

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 2, 1], [[1,1], [1,1]])
sage: RC.fermionic_formula()
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
 + (q+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
sage: t = QQ['t'].gen(0)
sage: RC.fermionic_formula(t)
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]]
 + (t+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]
sage: La = RC.weight_lattice_realization().classical().fundamental_weights()
sage: RC.fermionic_formula(weight=La[2])
q + 1
sage: RC.fermionic_formula(only_highest_weight=True, weight=La[2])
q
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(2), Integer(1)], [[Integer(1),
→Integer(1)], [Integer(1),Integer(1)]])
>>> RC.fermionic_formula()
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
 + (q+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
```

(continues on next page)

(continued from previous page)

```
>>> t = QQ['t'].gen(Integer(0))
>>> RC.fermionic_formula(t)
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]]
 + (t+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]
>>> La = RC.weight_lattice_realization().classical().fundamental_weights()
>>> RC.fermionic_formula(weight=La[Integer(2)])
q + 1
>>> RC.fermionic_formula(only_highest_weight=True, weight=La[Integer(2)])
q
```

Only using the highest weight elements on other types:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3,1], [2,2]])
sage: RC.fermionic_formula(only_highest_weight=True)
q*B[Lambda[1] + Lambda[2]] + B[2*Lambda[2] + Lambda[3]]
sage: RC = RiggedConfigurations(['D', 4, 1], [[3,1], [4,1], [2,1]])
sage: RC.fermionic_formula(only_highest_weight=True)
(q^4+q^3+q^2)*B[Lambda[1]] + (q^2+q)*B[Lambda[1] + Lambda[2]]
 + q*B[Lambda[1] + 2*Lambda[3]] + q*B[Lambda[1] + 2*Lambda[4]]
 + B[Lambda[2] + Lambda[3] + Lambda[4]] + (q^3+2*q^2+q)*B[Lambda[3] +␣
↪Lambda[4]]
sage: RC = RiggedConfigurations(['E', 6, 1], [[2,2]])
sage: RC.fermionic_formula(only_highest_weight=True)
q^2*B[0] + q*B[Lambda[2]] + B[2*Lambda[2]]
sage: RC = RiggedConfigurations(['B', 3, 1], [[3,1], [2,2]])
sage: RC.fermionic_formula(only_highest_weight=True) # long time
q*B[Lambda[1] + Lambda[2] + Lambda[3]] + q^2*B[Lambda[1]
 + Lambda[3]] + (q^2+q)*B[Lambda[2] + Lambda[3]] + B[2*Lambda[2]
 + Lambda[3]] + (q^3+q^2)*B[Lambda[3]]
sage: RC = RiggedConfigurations(['C', 3, 1], [[3,1], [2,2]])
sage: RC.fermionic_formula(only_highest_weight=True) # long time
(q^3+q^2)*B[Lambda[1] + Lambda[2]] + q*B[Lambda[1] + 2*Lambda[2]]
 + (q^2+q)*B[2*Lambda[1] + Lambda[3]] + B[2*Lambda[2] + Lambda[3]]
 + (q^4+q^3+q^2)*B[Lambda[3]]
sage: RC = RiggedConfigurations(['D', 4, 2], [[3,1], [2,2]])
sage: RC.fermionic_formula(only_highest_weight=True) # long time
(q^2+q)*B[Lambda[1] + Lambda[2] + Lambda[3]] + (q^5+2*q^4+q^3)*B[Lambda[1]
 + Lambda[3]] + (q^3+q^2)*B[2*Lambda[1] + Lambda[3]] + (q^4+q^3+q^
↪2)*B[Lambda[2]
 + Lambda[3]] + B[2*Lambda[2] + Lambda[3]] + (q^6+q^5+q^4)*B[Lambda[3]]
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1],[2,2]])
sage: RC.fermionic_formula(only_highest_weight=True)
(q^3+q^2)*B[Lambda[1]] + (q^2+q)*B[Lambda[1] + 2*Lambda[2]]
 + B[Lambda[1] + 4*Lambda[2]] + q*B[3*Lambda[1]] + q*B[4*Lambda[2]]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),
↪Integer(1)], [Integer(2),Integer(2)]])
>>> RC.fermionic_formula(only_highest_weight=True)
q*B[Lambda[1] + Lambda[2]] + B[2*Lambda[2] + Lambda[3]]
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)], [Integer(4),Integer(1)], [Integer(2),Integer(1)]])
>>> RC.fermionic_formula(only_highest_weight=True)
(q^4+q^3+q^2)*B[Lambda[1]] + (q^2+q)*B[Lambda[1] + Lambda[2]]
 + q*B[Lambda[1] + 2*Lambda[3]] + q*B[Lambda[1] + 2*Lambda[4]]
 + B[Lambda[2] + Lambda[3] + Lambda[4]] + (q^3+2*q^2+q)*B[Lambda[3] +␣
↪Lambda[4]]
>>> RC = RiggedConfigurations(['E', Integer(6), Integer(1)], [[Integer(2),
↪Integer(2)]])
>>> RC.fermionic_formula(only_highest_weight=True)
q^2*B[0] + q*B[Lambda[2]] + B[2*Lambda[2]]
>>> RC = RiggedConfigurations(['B', Integer(3), Integer(1)], [[Integer(3),
↪Integer(1)], [Integer(2),Integer(2)]])
>>> RC.fermionic_formula(only_highest_weight=True) # long time
q*B[Lambda[1] + Lambda[2] + Lambda[3]] + q^2*B[Lambda[1]
 + Lambda[3]] + (q^2+q)*B[Lambda[2] + Lambda[3]] + B[2*Lambda[2]
 + Lambda[3]] + (q^3+q^2)*B[Lambda[3]]
>>> RC = RiggedConfigurations(['C', Integer(3), Integer(1)], [[Integer(3),
↪Integer(1)], [Integer(2),Integer(2)]])
>>> RC.fermionic_formula(only_highest_weight=True) # long time
(q^3+q^2)*B[Lambda[1] + Lambda[2]] + q*B[Lambda[1] + 2*Lambda[2]]
 + (q^2+q)*B[2*Lambda[1] + Lambda[3]] + B[2*Lambda[2] + Lambda[3]]
 + (q^4+q^3+q^2)*B[Lambda[3]]
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(2)], [[Integer(3),
↪Integer(1)], [Integer(2),Integer(2)]])
>>> RC.fermionic_formula(only_highest_weight=True) # long time
(q^2+q)*B[Lambda[1] + Lambda[2] + Lambda[3]] + (q^5+2*q^4+q^3)*B[Lambda[1]
 + Lambda[3]] + (q^3+q^2)*B[2*Lambda[1] + Lambda[3]] + (q^4+q^3+q^
↪2)*B[Lambda[2]
 + Lambda[3]] + B[2*Lambda[2] + Lambda[3]] + (q^6+q^5+q^4)*B[Lambda[3]]
>>> RC = RiggedConfigurations(CartanType(['A',Integer(4),Integer(2)]).dual(),␣
↪[[Integer(1),Integer(1)],[Integer(2),Integer(2)]])
>>> RC.fermionic_formula(only_highest_weight=True)
(q^3+q^2)*B[Lambda[1]] + (q^2+q)*B[Lambda[1] + 2*Lambda[2]]
 + B[Lambda[1] + 4*Lambda[2]] + q*B[3*Lambda[1]] + q*B[4*Lambda[2]]
```

kleber_tree()

Return the underlying Kleber tree used to generate all highest weight rigged configurations.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',3,1], [[1,1], [2,1]])
sage: RC.kleber_tree()
Kleber tree of Cartan type ['A', 3, 1] and B = ((1, 1), (2, 1))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A',Integer(3),Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(2),Integer(1)]])
>>> RC.kleber_tree()
Kleber tree of Cartan type ['A', 3, 1] and B = ((1, 1), (2, 1))
```

module_generators()

Module generators for this set of rigged configurations.

Iterate over the highest weight rigged configurations by moving through the `KleberTree` and then setting appropriate values of the partitions.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2,1]])
sage: for x in RC.module_generators: x

(/)

(/)

(/)

(/)


0[ ]0

0[ ]0
0[ ]0

0[ ]0

0[ ]0
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['D', Integer(4), Integer(1)], [[Integer(2),
↪Integer(1)]])
>>> for x in RC.module_generators: x
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
(/)
<BLANKLINE>
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
```

options = Current options for RiggedConfigurations - convention: English - display: vertical - element_ascii_art: True - half_width_boxes_type_B: True

tensor(**crystals*, ***options*)

Return the tensor product of `self` with `crystals`.

If `crystals` is a list of rigged configurations of the same Cartan type, then this returns a new *RiggedConfigurations*.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[2,1],[1,3]])
sage: RC2 = RiggedConfigurations(['A', 3, 1], [[1,1], [3,3]])
sage: RC.tensor(RC2, RC2)
Rigged configurations of type ['A', 3, 1]
 and factor(s) ((2, 1), (1, 3), (1, 1), (3, 3), (1, 1), (3, 3))

sage: K = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: RC.tensor(K)
Full tensor product of the crystals
 [Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 1), (1, 3)),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2)]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(2),
↪Integer(1)],[Integer(1),Integer(3)]])
>>> RC2 = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(1),
↪Integer(1)], [Integer(3),Integer(3)]])
>>> RC.tensor(RC2, RC2)
Rigged configurations of type ['A', 3, 1]
 and factor(s) ((2, 1), (1, 3), (1, 1), (3, 3), (1, 1), (3, 3))

>>> K = crystals.KirillovReshetikhin(['A', Integer(3), Integer(1)],
↪Integer(2), Integer(2), model='KR')
>>> RC.tensor(K)
Full tensor product of the crystals
 [Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 1), (1, 3)),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2)]
```

tensor_product_of_kirillov_reshetikhin_crystals()

Return the corresponding tensor product of Kirillov-Reshetikhin crystals.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3,1],[2,2]])
sage: RC.tensor_product_of_kirillov_reshetikhin_crystals()
Full tensor product of the crystals
[Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1),
 Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(2,2)]
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),
↪Integer(1)],[Integer(2),Integer(2)]])
>>> RC.tensor_product_of_kirillov_reshetikhin_crystals()
Full tensor product of the crystals
[Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1),
 Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(2,2)]
```

tensor_product_of_kirillov_reshetikhin_tableaux()

Return the corresponding tensor product of Kirillov-Reshetikhin tableaux.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2]])
sage: RC.tensor_product_of_kirillov_reshetikhin_tableaux()
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and
↪factor(s) ((3, 2), (1, 2))
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(3), Integer(1)], [[Integer(3),
↪Integer(2)], [Integer(1), Integer(2)]])
>>> RC.tensor_product_of_kirillov_reshetikhin_tableaux()
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and
↪factor(s) ((3, 2), (1, 2))
```

5.1.216 Rigged Partitions

Class and methods of the rigged partition which are used by the rigged configuration class. This is an internal class used by the rigged configurations and KR tableaux during the bijection, and is not to be used by the end-user.

We hold the partitions as a 1-dim array of positive integers where each value corresponds to the length of the row. This is the shape of the partition which can be accessed by the regular index.

The data for the vacancy number is also stored in a 1-dim array which each entry corresponds to the row of the tableau, and similarly for the partition values.

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

Todo

Convert this to using multiplicities m_i (perhaps with a dictionary?)?

class sage.combinat.rigged_configurations.rigged_partition.**RiggedPartition**

Bases: SageObject

The RiggedPartition class which is the data structure of a rigged (i.e. marked or decorated) Young diagram of a partition.

Note that this class as a stand-alone object does not make sense since the vacancy numbers are calculated using the entire rigged configuration. For more, see *RiggedConfigurations*.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RP = RC(partition_list=[[2],[2,2],[2,1],[2]])[2]
sage: RP
 0[ ][ ]0
-1[ ]-1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)]])
>>> RP = RC(partition_list=[[Integer(2)],[Integer(2),Integer(2)],[Integer(2),
↪Integer(1)],[Integer(2)]])[Integer(2)]
>>> RP
 0[ ][ ]0
-1[ ]-1
<BLANKLINE>
```

get_num_cells_to_column(*end_column*, *t=1*)

Get the number of cells in all columns before the `end_column`.

INPUT:

- `end_column` – the index of the column to end at
- `t` – the scaling factor

OUTPUT: the number of cells

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RP = RC(partition_list=[[2],[2,2],[2,1],[2]])[2]
sage: RP.get_num_cells_to_column(1)
2
sage: RP.get_num_cells_to_column(2)
3
sage: RP.get_num_cells_to_column(3)
3
sage: RP.get_num_cells_to_column(3, 2)
5
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),␣
↪Integer(2)]])
>>> RP = RC(partition_list=[[Integer(2)],[Integer(2),Integer(2)],[Integer(2),
↪Integer(1)],[Integer(2)]])[Integer(2)]
>>> RP.get_num_cells_to_column(Integer(1))
2
>>> RP.get_num_cells_to_column(Integer(2))
3
>>> RP.get_num_cells_to_column(Integer(3))
3
>>> RP.get_num_cells_to_column(Integer(3), Integer(2))
5
```

insert_cell(*max_width*)

Insert a cell given at a singular value as long as its less than the specified width.

Note that *insert_cell()* does not update riggings or vacancy numbers, but it does prepare the space for them. Returns the width of the row we inserted at.

INPUT:

- `max_width` – the maximum width (i.e. row length) that we can insert the cell at

OUTPUT: the width of the row we inserted at

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RP = RC(partition_list=[[2],[2,2],[2,1],[2]])[2]
sage: RP.insert_cell(2)
2
sage: RP
 0[ ][ ][ ]None
-1[ ]-1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),␣
→Integer(2)]])
>>> RP = RC(partition_list=[[Integer(2)],[Integer(2),Integer(2)],[Integer(2),
→Integer(1)],[Integer(2)]])[Integer(2)]
>>> RP.insert_cell(Integer(2))
2
>>> RP
 0[ ][ ][ ]None
-1[ ]-1
<BLANKLINE>
```

remove_cell(*row*, *num_cells=1*)

Removes a cell at the specified `row`.

Note that *remove_cell()* does not set/update the vacancy numbers or the riggings, but guarantees that the location has been allocated in the returned index.

INPUT:

- `row` – the row to remove the cell from
- `num_cells` – (default: 1) the number of cells to remove

OUTPUT:

- The location of the newly constructed row or `None` if unable to remove row or if deleted a row.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RP = RC(partition_list=[[2],[2,2],[2,1],[2]])[2]
sage: RP.remove_cell(0)
0
sage: RP
 None[ ]None
-1[ ]-1
```

```
>>> from sage.all import *
>>> RC = RiggedConfigurations(['A', Integer(4), Integer(1)], [[Integer(2),␣
→Integer(2)]])
>>> RP = RC(partition_list=[[Integer(2)],[Integer(2),Integer(2)],[Integer(2),
→Integer(1)],[Integer(2)]])[Integer(2)]
>>> RP.remove_cell(Integer(0))
0
```

(continues on next page)

(continued from previous page)

```
>>> RP
 None[ ]None
-1[ ]-1
<BLANKLINE>
```

rigging

vacancy_numbers

class sage.combinat.rigged_configurations.rigged_partition.**RiggedPartitionTypeB**

Bases: *RiggedPartition*

Rigged partitions for type $B_n^{(1)}$ which has special printing rules which comes from the fact that the n-th partition can have columns of width $\frac{1}{2}$.

5.1.217 Tensor Product of Kirillov-Reshetikhin Tableaux

A tensor product of *KirillovReshetikhinTableaux* which are tableaux of r rows and s columns which naturally arise in the bijection between rigged configurations and tableaux and which are in bijection with the elements of the Kirillov-Reshetikhin crystal $B^{r,s}$, see *KirillovReshetikhinCrystal()*.

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

EXAMPLES:

Type $A_n^{(1)}$ examples:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,1], [2,
↪1]])
sage: KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and factor(s) ((3,
↪ 1), (2, 1))
sage: KRT.cardinality()
24
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[1,1], [2,
↪1], [3,1]])
sage: KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and factor(s) ((1,
↪ 1), (2, 1), (3, 1))
sage: len(KRT.module_generators)
5
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[1,1], [2,
↪1], [3,1]])
sage: KRT.cardinality()
96
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(3),Integer(1)], [Integer(2),Integer(1)]])
>>> KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and factor(s) ((3,
↪ 1), (2, 1))
>>> KRT.cardinality()
```

(continues on next page)

(continued from previous page)

```
24
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(1),Integer(1)], [Integer(2),Integer(1)], [Integer(3),
↪Integer(1)]])
>>> KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and factor(s) ((1,
↪ 1), (2, 1), (3, 1))
>>> len(KRT.module_generators)
5
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(1),Integer(1)], [Integer(2),Integer(1)], [Integer(3),
↪Integer(1)]])
>>> KRT.cardinality()
96
```

Type $D_n^{(1)}$ examples:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[1, 1],
↪[2, 1], [1, 1]])
sage: KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and factor(s) ((1,
↪ 1), (2, 1), (1, 1))
sage: T = KRT(pathlist=[[1], [-2, 2], [1]])
sage: T
[[1]] (X) [[2], [-2]] (X) [[1]]
sage: T2 = KRT(pathlist=[[1], [2, -2], [1]])
sage: T2
[[1]] (X) [[-2], [2]] (X) [[1]]
sage: T == T2
False
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', Integer(4),
↪Integer(1)], [[Integer(1), Integer(1)], [Integer(2), Integer(1)], [Integer(1),
↪Integer(1)]])
>>> KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and factor(s) ((1,
↪ 1), (2, 1), (1, 1))
>>> T = KRT(pathlist=[[Integer(1)], [-Integer(2), Integer(2)], [Integer(1)]])
>>> T
[[1]] (X) [[2], [-2]] (X) [[1]]
>>> T2 = KRT(pathlist=[[Integer(1)], [Integer(2), -Integer(2)], [Integer(1)]])
>>> T2
[[1]] (X) [[-2], [2]] (X) [[1]]
>>> T == T2
False
```

class sage.combinat.rigged_configurations.tensor_product_kr_tableaux.**HighestWeightTensorKRT**(*tp_krt*)

Bases: UniqueRepresentation

Class so we do not have to build the module generators for *TensorProductOfKirillovReshetikhinTableaux* at initialization.

Warning

This class is for internal use only!

cardinality()

Return the cardinality of `self`, which is the number of highest weight elements.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',4,1],
↪[[2,2]])
sage: from sage.combinat.rigged_configurations.tensor_product_kr_tableaux
↪import HighestWeightTensorKRT
sage: HW = HighestWeightTensorKRT(KRT)
sage: HW.cardinality()
3
sage: len(HW)
3
sage: len(KRT.module_generators)
3
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',Integer(4),
↪Integer(1)], [[Integer(2),Integer(2)]])
>>> from sage.combinat.rigged_configurations.tensor_product_kr_tableaux
↪import HighestWeightTensorKRT
>>> HW = HighestWeightTensorKRT(KRT)
>>> HW.cardinality()
3
>>> len(HW)
3
>>> len(KRT.module_generators)
3
```

class sage.combinat.rigged_configurations.tensor_product_kr_tableaux.**TensorProductOfKirillovReshetikh**

Bases: *FullTensorProductOfRegularCrystals*

A tensor product of *KirillovReshetikhinTableaux*.

Through the bijection with rigged configurations, the tableaux that are produced in all nonexceptional types are all of rectangular shapes and do not necessarily obey the usual strict increase in columns and weak increase in rows. The relation between the elements of the Kirillov-Reshetikhin crystal, given by the Kashiwara-Nakashima tableaux, and the Kirillov-Reshetikhin tableaux is given by a filling map.

Note

The tableaux for all non-simply-laced types are provably correct if the bijection with *rigged configurations* holds. Therefore this is currently only proven for $B^{r,1}$ or $B^{1,s}$ and in general for types $A_n^{(1)}$ and $D_n^{(1)}$.

For more information see [OSS2011] and *KirillovReshetikhinTableaux*.

For more information on KR crystals, see `sage.combinat.crystals.kirillov_reshetikhin`.

INPUT:

- `cartan_type` – a Cartan type
- `B` – an (ordered) list of pairs (r, s) which give the dimension of a rectangle with r rows and s columns and corresponds to a Kirillov-Reshetikhin tableaux factor of $B^{r,s}$.

REFERENCES:

EXAMPLES:

We can go between tensor products of KR crystals and rigged configurations:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,1],
↪[2,2]])
sage: tp_krt = KRT(pathlist=[[3,2,1],[3,2,3,2]]); tp_krt
[[1], [2], [3]] (X) [[2, 2], [3, 3]]
sage: RC = RiggedConfigurations(['A',3,1], [[3,1],[2,2]])
sage: rc_elt = tp_krt.to_rigged_configuration(); rc_elt

-2[ ][ ]-2

0[ ][ ]0

(/)

sage: tp_krc = tp_krt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krc
[[[1], [2], [3]], [[2, 2], [3, 3]]]
sage: KRT(tp_krc) == tp_krt
True
sage: rc_elt == tp_krt.to_rigged_configuration()
True
sage: KR1 = crystals.KirillovReshetikhin(['A',3,1], 3,1)
sage: KR2 = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: T = crystals.TensorProduct(KR1, KR2)
sage: t = T(KR1(3,2,1), KR2(3,2,3,2))
sage: KRT(t) == tp_krt
True
sage: t == tp_krc
True
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(3),Integer(1)],[Integer(2),Integer(2)]])
>>> tp_krt = KRT(pathlist=[[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(2),Integer(3),Integer(2)]]); tp_krt
[[1], [2], [3]] (X) [[2, 2], [3, 3]]
>>> RC = RiggedConfigurations(['A',Integer(3),Integer(1)], [[Integer(3),
↪Integer(1)],[Integer(2),Integer(2)]])
>>> rc_elt = tp_krt.to_rigged_configuration(); rc_elt
<BLANKLINE>
-2[ ][ ]-2
<BLANKLINE>
0[ ][ ]0
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
(/)
<BLANKLINE>
>>> tp_krc = tp_krt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krc
[[[1], [2], [3]], [[2, 2], [3, 3]]]
>>> KRT(tp_krc) == tp_krt
True
>>> rc_elt == tp_krt.to_rigged_configuration()
True
>>> KR1 = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(3),
↪Integer(1))
>>> KR2 = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),
↪Integer(2))
>>> T = crystals.TensorProduct(KR1, KR2)
>>> t = T(KR1(Integer(3),Integer(2),Integer(1)), KR2(Integer(3),Integer(2),
↪Integer(3),Integer(2)))
>>> KRT(t) == tp_krt
True
>>> t == tp_krc
True
```

We can get the highest weight elements by using the attribute `module_generators`:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,1],
↪ [2,1]])
sage: list(KRT.module_generators)
[[[1], [2], [3]] (X) [[1], [2]], [[1], [3], [4]] (X) [[1], [2]]]
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(3),Integer(1)], [Integer(2),Integer(1)]])
>>> list(KRT.module_generators)
[[[1], [2], [3]] (X) [[1], [2]], [[1], [3], [4]] (X) [[1], [2]]]
```

To create elements directly (i.e. not passing in KR tableaux elements), there is the **pathlist** option will receive a list of lists which contain the reversed far-eastern reading word of the tableau. That is to say, in English notation, the word obtain from reading bottom-to-top, left-to-right.

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,2],
↪ [1,2], [2,1]])
sage: elt = KRT(pathlist=[[3, 2, 1, 4, 2, 1], [1, 3], [3, 1]])
sage: elt.pp()
  1  1 (X)   1  3 (X)   1
  2  2                  3
  3  4
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(3),Integer(2)], [Integer(1),Integer(2)], [Integer(2),
↪Integer(1)]])
>>> elt = KRT(pathlist=[[Integer(3), Integer(2), Integer(1), Integer(4),␣
↪Integer(2), Integer(1)], [Integer(1), Integer(3)], [Integer(3), Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> elt.pp()
  1  1 (X)   1  3 (X)   1
  2  2                  3
  3  4
```

One can still create elements in the same way as tensor product of crystals:

```
sage: K1 = crystals.KirillovReshetikhin(['A',3,1], 3, 2, model='KR')
sage: K2 = crystals.KirillovReshetikhin(['A',3,1], 1, 2, model='KR')
sage: K3 = crystals.KirillovReshetikhin(['A',3,1], 2, 1, model='KR')
sage: eltlong = KRT(K1(3, 2, 1, 4, 2, 1), K2(1, 3), K3(3, 1))
sage: eltlong == elt
True
```

```
>>> from sage.all import *
>>> K1 = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(3),␣
↪Integer(2), model='KR')
>>> K2 = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(1),␣
↪Integer(2), model='KR')
>>> K3 = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(2),␣
↪Integer(1), model='KR')
>>> eltlong = KRT(K1(Integer(3), Integer(2), Integer(1), Integer(4), Integer(2),␣
↪Integer(1)), K2(Integer(1), Integer(3)), K3(Integer(3), Integer(1)))
>>> eltlong == elt
True
```

Element

alias of *TensorProductOfKirillovReshetikhinTableauxElement*

rigged_configurations()

Return the corresponding set of rigged configurations.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1],␣
↪[[1,3], [2,1]])
sage: KRT.rigged_configurations()
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 3), (2, 1))
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(1),Integer(3)], [Integer(2),Integer(1)]])
>>> KRT.rigged_configurations()
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 3), (2, 1))
```

tensor(**crystals*, ***options*)

Return the tensor product of `self` with `crystals`.

If `crystals` is a list of (a tensor product of) KR tableaux, this returns a *TensorProductOfKirillovReshetikhinTableaux*.

EXAMPLES:

```
sage: TP = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 3, 1],
↪[[1,3],[3,1]])
sage: K = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: TP.tensor(K, TP)
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1]
 and factor(s) ((1, 3), (3, 1), (2, 2), (1, 3), (3, 1))

sage: C = crystals.KirillovReshetikhin(['A',3,1], 3, 1, model='KN')
sage: TP.tensor(K, C)
Full tensor product of the crystals
 [Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (1, 3),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (3, 1),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1)]
```

```
>>> from sage.all import *
>>> TP = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', Integer(3),
↪ Integer(1)], [[Integer(1),Integer(3)],[Integer(3),Integer(1)]])
>>> K = crystals.KirillovReshetikhin(['A', Integer(3), Integer(1)],
↪Integer(2), Integer(2), model='KR')
>>> TP.tensor(K, TP)
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1]
 and factor(s) ((1, 3), (3, 1), (2, 2), (1, 3), (3, 1))

>>> C = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], Integer(3),
↪Integer(1), model='KN')
>>> TP.tensor(K, C)
Full tensor product of the crystals
 [Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (1, 3),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (3, 1),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1)]
```

tensor_product_of_kirillov_reshetikhin_crystals()

Return the corresponding tensor product of Kirillov-Reshetikhin crystals.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1],
↪[[3,1],[2,2]])
sage: KRT.tensor_product_of_kirillov_reshetikhin_crystals()
Full tensor product of the crystals [Kirillov-Reshetikhin crystal of type ['A
↪', 3, 1] with (r,s)=(3,1),
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(2,2)]
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(3),Integer(1)],[Integer(2),Integer(2)]])
>>> KRT.tensor_product_of_kirillov_reshetikhin_crystals()
Full tensor product of the crystals [Kirillov-Reshetikhin crystal of type ['A
↪', 3, 1] with (r,s)=(3,1),
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(2,2)]
```

5.1.218 Tensor Product of Kirillov-Reshetikhin Tableaux Elements

A tensor product of *KirillovReshetikhinTableauxElement*.

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

class sage.combinat.rigged_configurations.tensor_product_kr_tableaux_element.**TensorProductOfKirillovR**

Bases: *TensorProductOfRegularCrystalsElement*

An element in a tensor product of Kirillov-Reshetikhin tableaux.

For more on tensor product of Kirillov-Reshetikhin tableaux, see *TensorProductOfKirillovReshetikhinTableaux*.

The most common way to construct an element is to specify the option `pathlist` which is a list of lists which will be used to generate the individual factors of *KirillovReshetikhinTableauxElement*.

EXAMPLES:

Type $A_n^{(1)}$ examples:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 3, 1], [[1,
→1], [2,1], [1,1], [2,1], [2,1], [2,1]])
sage: T = KRT(pathlist=[[2], [4,1], [3], [4,2], [3,1], [2,1]])
sage: T
[[2]] (X) [[1], [4]] (X) [[3]] (X) [[2], [4]] (X) [[1], [3]] (X) [[1], [2]]
sage: T.to_rigged_configuration()

0[ ][ ]0
1[ ]1

1[ ][ ]0
1[ ]0
1[ ]0

0[ ][ ]0

sage: T = KRT(pathlist=[[1], [2,1], [1], [4,1], [3,1], [2,1]])
sage: T
[[1]] (X) [[1], [2]] (X) [[1]] (X) [[1], [4]] (X) [[1], [3]] (X) [[1], [2]]
sage: T.to_rigged_configuration()

(/)

1[ ]0
1[ ]0

0[ ]0
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', Integer(3),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)], [[Integer(1),Integer(1)], [Integer(2),Integer(1)], [Integer(1),
↪Integer(1)], [Integer(2),Integer(1)], [Integer(2),Integer(1)], [Integer(2),
↪Integer(1)]])
>>> T = KRT(pathlist=[[Integer(2)], [Integer(4),Integer(1)], [Integer(3)],
↪[Integer(4),Integer(2)], [Integer(3),Integer(1)], [Integer(2),Integer(1)]])
>>> T
[[2]] (X) [[1], [4]] (X) [[3]] (X) [[2], [4]] (X) [[1], [3]] (X) [[1], [2]]
>>> T.to_rigged_configuration()
<BLANKLINE>
0[ ][ ]0
1[ ]1
<BLANKLINE>
1[ ][ ]0
1[ ]0
1[ ]0
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
>>> T = KRT(pathlist=[[Integer(1)], [Integer(2),Integer(1)], [Integer(1)],
↪[Integer(4),Integer(1)], [Integer(3),Integer(1)], [Integer(2),Integer(1)]])
>>> T
[[1]] (X) [[1], [2]] (X) [[1]] (X) [[1], [4]] (X) [[1], [3]] (X) [[1], [2]]
>>> T.to_rigged_configuration()
<BLANKLINE>
(/)
<BLANKLINE>
1[ ]0
1[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
```

Type $D_n^{(1)}$ examples:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[1,
↪1], [1,1], [1,1], [1,1]])
sage: T = KRT(pathlist=[[-1], [-1], [1], [1]])
sage: T
[[-1]] (X) [[-1]] (X) [[1]] (X) [[1]]
sage: T.to_rigged_configuration()

0[ ][ ]0
0[ ][ ]0

0[ ][ ]0
0[ ][ ]0

0[ ][ ]0

0[ ][ ]0

sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[2,
```

(continues on next page)

(continued from previous page)

```
↪1], [1,1], [1,1], [1,1]])
sage: T = KRT(pathlist=[[3,2], [1], [-1], [1]])
sage: T
[[2], [3]] (X) [[1]] (X) [[-1]] (X) [[1]]
sage: T.to_rigged_configuration()

0[ ]0
0[ ]0
0[ ]0

0[ ]0
0[ ]0
0[ ]0

1[ ]0

1[ ]0

sage: T.to_rigged_configuration().to_tensor_product_of_kirillov_reshetikhin_
↪tableaux()
[[2], [3]] (X) [[1]] (X) [[-1]] (X) [[1]]
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', Integer(4),␣
↪Integer(1)], [[Integer(1),Integer(1)], [Integer(1),Integer(1)], [Integer(1),
↪Integer(1)], [Integer(1),Integer(1)]])
>>> T = KRT(pathlist=[[-Integer(1)], [-Integer(1)], [Integer(1)], [Integer(1)]])
>>> T
[[-1]] (X) [[-1]] (X) [[1]] (X) [[1]]
>>> T.to_rigged_configuration()
<BLANKLINE>
0[ ][ ]0
0[ ][ ]0
<BLANKLINE>
0[ ][ ]0
0[ ][ ]0
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', Integer(4),␣
↪Integer(1)], [[Integer(2),Integer(1)], [Integer(1),Integer(1)], [Integer(1),
↪Integer(1)], [Integer(1),Integer(1)]])
>>> T = KRT(pathlist=[[Integer(3),Integer(2)], [Integer(1)], [-Integer(1)],␣
↪[Integer(1)]])
>>> T
[[2], [3]] (X) [[1]] (X) [[-1]] (X) [[1]]
>>> T.to_rigged_configuration()
<BLANKLINE>
0[ ]0
0[ ]0
```

(continues on next page)

(continued from previous page)

```
0[ ]0
<BLANKLINE>
0[ ]0
0[ ]0
0[ ]0
<BLANKLINE>
1[ ]0
<BLANKLINE>
1[ ]0
<BLANKLINE>
>>> T.to_rigged_configuration().to_tensor_product_of_kirillov_reshetikhin_
↪tableaux()
[[2], [3]] (X) [[1]] (X) [[-1]] (X) [[1]]
```

classical_weight()

Return the classical weight of `self`.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',4,1],␣
↪[[2,2]])
sage: elt = KRT(pathlist=[[3,2,-1,1]]); elt
[[2, 1], [3, -1]]
sage: elt.classical_weight()
(0, 1, 1, 0)
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1],␣
↪[[2,2],[1,3]])
sage: elt = KRT(pathlist=[[2,1,3,2],[1,4,4]]); elt
[[1, 2], [2, 3]] (X) [[1, 4, 4]]
sage: elt.classical_weight()
(2, 2, 1, 2)
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',Integer(4),
↪Integer(1)], [[Integer(2),Integer(2)]])
>>> elt = KRT(pathlist=[[Integer(3),Integer(2),-Integer(1),Integer(1)]]); elt
[[2, 1], [3, -1]]
>>> elt.classical_weight()
(0, 1, 1, 0)
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(2),Integer(2)],[Integer(1),Integer(3)]])
>>> elt = KRT(pathlist=[[Integer(2),Integer(1),Integer(3),Integer(2)],
↪[Integer(1),Integer(4),Integer(4)]]); elt
[[1, 2], [2, 3]] (X) [[1, 4, 4]]
>>> elt.classical_weight()
(2, 2, 1, 2)
```

left_split()

Return the image of `self` under the left column splitting map.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1],
↪[[2,2],[1,3]])
sage: elt = KRT(pathlist=[[2,1,3,2],[1,4,4]]); elt.pp()
  1  2 (X)   1  4  4
  2  3
sage: elt.left_split().pp()
  1 (X)   2 (X)   1  4  4
  2       3
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(2),Integer(2)],[Integer(1),Integer(3)]])
>>> elt = KRT(pathlist=[[Integer(2),Integer(1),Integer(3),Integer(2)],
↪[Integer(1),Integer(4),Integer(4)]]); elt.pp()
  1  2 (X)   1  4  4
  2  3
>>> elt.left_split().pp()
  1 (X)   2 (X)   1  4  4
  2       3
```

lusztig_involution()

Return the result of the classical Lusztig involution on `self`.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1],
↪[[2,2],[1,3]])
sage: elt = KRT(pathlist=[[2,1,3,2],[1,4,4]])
sage: li = elt.lusztig_involution(); li
[[1, 1, 4]] (X) [[2, 3], [3, 4]]
sage: li.parent()
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and
↪factor(s) ((1, 3), (2, 2))
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(2),Integer(2)],[Integer(1),Integer(3)]])
>>> elt = KRT(pathlist=[[Integer(2),Integer(1),Integer(3),Integer(2)],
↪[Integer(1),Integer(4),Integer(4)]])
>>> li = elt.lusztig_involution(); li
[[1, 1, 4]] (X) [[2, 3], [3, 4]]
>>> li.parent()
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and
↪factor(s) ((1, 3), (2, 2))
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: TPKRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',4,1],
↪[[2,2],[3,1],[3,3]])
sage: TPKRT.module_generators[0].pp()
```

(continues on next page)

(continued from previous page)

```
  1  1 (X)   1 (X)   1  1  1
  2  2       2       2  2  2
             3       3  3  3
```

```
>>> from sage.all import *
>>> TPKRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',
↪Integer(4),Integer(1)], [[Integer(2),Integer(2)],[Integer(3),Integer(1)],
↪[Integer(3),Integer(3)]])
>>> TPKRT.module_generators[Integer(0)].pp()
  1  1 (X)   1 (X)   1  1  1
  2  2       2       2  2  2
             3       3  3  3
```

right_split()

Return the image of `self` under the right column splitting map.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1],
↪[[2,2],[1,3]])
sage: elt = KRT(pathlist=[[2,1,3,2],[1,4,4]]); elt.pp()
  1  2 (X)   1  4  4
  2  3
sage: elt.right_split().pp()
  1  2 (X)   1  4 (X)   4
  2  3
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',Integer(3),
↪Integer(1)], [[Integer(2),Integer(2)],[Integer(1),Integer(3)]])
>>> elt = KRT(pathlist=[[Integer(2),Integer(1),Integer(3),Integer(2)],
↪[Integer(1),Integer(4),Integer(4)]]); elt.pp()
  1  2 (X)   1  4  4
  2  3
>>> elt.right_split().pp()
  1  2 (X)   1  4 (X)   4
  2  3
```

Let $*$ denote the `Lusztig involution`, we check that $* \circ \mathrm{ls} \circ * = \mathrm{rs}$:

```
sage: all(x.lusztig_involution().left_split().lusztig_involution() == x.right_
↪split() for x in KRT)
True
```

```
>>> from sage.all import *
>>> all(x.lusztig_involution().left_split().lusztig_involution() == x.right_
↪split() for x in KRT)
True
```

to_rigged_configuration(*display_steps=False*)

Perform the bijection from `self` to a `rigged configuration` which is described in [RigConBijection], [BijectionLRT], and [BijectionDn].

> **Note**
>
> This is only proven to be a bijection in types $A_n^{(1)}$ and $D_n^{(1)}$, as well as $\bigotimes_i B^{r_i,1}$ and $\bigotimes_i B^{1,s_i}$ for general affine types.

INPUT:

- `display_steps` – boolean (default: `False`); whether to output each step in the algorithm

OUTPUT: the rigged configuration corresponding to `self`

EXAMPLES:

Type $A_n^{(1)}$ example:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 3, 1],␣
↪[[2,1], [2,1], [2,1]])
sage: T = KRT(pathlist=[[4, 2], [3, 1], [2, 1]])
sage: T
[[2], [4]] (X) [[1], [3]] (X) [[1], [2]]
sage: T.to_rigged_configuration()

0[ ]0

1[ ]1
1[ ]0

0[ ]0
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',␣
↪Integer(3), Integer(1)], [[Integer(2),Integer(1)], [Integer(2),Integer(1)],␣
↪[Integer(2),Integer(1)]])
>>> T = KRT(pathlist=[[Integer(4), Integer(2)], [Integer(3), Integer(1)],␣
↪[Integer(2), Integer(1)]])
>>> T
[[2], [4]] (X) [[1], [3]] (X) [[1], [2]]
>>> T.to_rigged_configuration()
<BLANKLINE>
0[ ]0
<BLANKLINE>
1[ ]1
1[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
```

Type $D_n^{(1)}$ example:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1],␣
↪[[2,2]])
sage: T = KRT(pathlist=[[2,1,4,3]])
sage: T
[[1, 3], [2, 4]]
```

(continues on next page)

(continued from previous page)

```
sage: T.to_rigged_configuration()

0[ ]0

-1[ ]-1
-1[ ]-1

0[ ]0

(/)
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',␣
↪Integer(4), Integer(1)], [[Integer(2),Integer(2)]])
>>> T = KRT(pathlist=[[Integer(2),Integer(1),Integer(4),Integer(3)]])
>>> T
[[1, 3], [2, 4]]
>>> T.to_rigged_configuration()
<BLANKLINE>
0[ ]0
<BLANKLINE>
-1[ ]-1
-1[ ]-1
<BLANKLINE>
0[ ]0
<BLANKLINE>
(/)
```

Type $D_n^{(1)}$ spinor example:

```
sage: CP = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 5, 1],␣
↪[[5,1],[2,1],[1,1],[1,1],[1,1]])
sage: elt = CP(pathlist=[[-2,-5,4,3,1],[-1,2],[1],[1],[1]])
sage: elt
[[1], [3], [4], [-5], [-2]] (X) [[2], [-1]] (X) [[1]] (X) [[1]] (X) [[1]]
sage: elt.to_rigged_configuration()

2[ ][ ]1

0[ ][ ]0
0[ ]0

0[ ][ ]0
0[ ]0

0[ ]0

0[ ][ ]0
```

```
>>> from sage.all import *
>>> CP = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', Integer(5),
↪ Integer(1)], [[Integer(5),Integer(1)],[Integer(2),Integer(1)],[Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)],[Integer(1),Integer(1)],[Integer(1),Integer(1)]])
>>> elt = CP(pathlist=[[-Integer(2),-Integer(5),Integer(4),Integer(3),
↪Integer(1)],[-Integer(1),Integer(2)],[Integer(1)],[Integer(1)],
↪[Integer(1)]])
>>> elt
[[1], [3], [4], [-5], [-2]] (X) [[2], [-1]] (X) [[1]] (X) [[1]] (X) [[1]]
>>> elt.to_rigged_configuration()
<BLANKLINE>
2[ ][ ]1
<BLANKLINE>
0[ ][ ]0
0[ ]0
<BLANKLINE>
0[ ][ ]0
0[ ]0
<BLANKLINE>
0[ ]0
<BLANKLINE>
0[ ][ ]0
<BLANKLINE>
```

This is invertible by calling `to_tensor_product_of_kirillov_reshetikhin_tableaux()`:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1],␣
↪[[2,2]])
sage: T = KRT(pathlist=[[2,1,4,3]])
sage: rc = T.to_rigged_configuration()
sage: ret = rc.to_tensor_product_of_kirillov_reshetikhin_tableaux(); ret
[[1, 3], [2, 4]]
sage: ret == T
True
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',␣
↪Integer(4), Integer(1)], [[Integer(2),Integer(2)]])
>>> T = KRT(pathlist=[[Integer(2),Integer(1),Integer(4),Integer(3)]])
>>> rc = T.to_rigged_configuration()
>>> ret = rc.to_tensor_product_of_kirillov_reshetikhin_tableaux(); ret
[[1, 3], [2, 4]]
>>> ret == T
True
```

to_tensor_product_of_kirillov_reshetikhin_crystals()

Return a tensor product of Kirillov-Reshetikhin crystals corresponding to `self`.

This works by performing the filling map on each individual factor. For more on the filling map, see *KirillovReshetikhinTableaux*.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',4,1],␣
↪[[1,1],[2,2]])
sage: elt = KRT(pathlist=[[-1],[-1,2,-1,1]]); elt
```

(continues on next page)

(continued from previous page)

```
[[-1]] (X) [[2, 1], [-1, -1]]
sage: tp_krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_
↪krc
[[[-1]], [[2], [-1]]]
```

```
>>> from sage.all import *
>>> KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',Integer(4),
↪Integer(1)], [[Integer(1),Integer(1)],[Integer(2),Integer(2)]])
>>> elt = KRT(pathlist=[[-Integer(1)],[-Integer(1),Integer(2),-Integer(1),
↪Integer(1)]]); elt
[[-1]] (X) [[2, 1], [-1, -1]]
>>> tp_krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krc
[[[-1]], [[2], [-1]]]
```

We can recover the original tensor product of KR tableaux:

```
sage: ret = KRT(tp_krc); ret
[[-1]] (X) [[2, 1], [-1, -1]]
sage: ret == elt
True
```

```
>>> from sage.all import *
>>> ret = KRT(tp_krc); ret
[[-1]] (X) [[2, 1], [-1, -1]]
>>> ret == elt
True
```

5.1.219 Root Systems

Quickref

- `T = CartanType(["A", 3]), T.is_finite()` – Cartan types
- `T.dynkin_diagram(), DynkinDiagram(["G",2])` – Dynkin diagrams
- `T.cartan_matrix(), CartanMatrix(["F",4])` – Cartan matrices
- `RootSystem(T).weight_lattice()` – Root systems
- `WeylGroup(["B", 6, 1]).simple_reflections()` – Affine Weyl groups
- `WeylCharacterRing(["D", 4])` – Weyl character rings

Introductory material

- *Root Systems* – This overview
- `CartanType` – An introduction to Cartan types
- `RootSystem` – An introduction to root systems
- *Tutorial: visualizing root systems* – A root system visualization tutorial
- The Lie Methods and Related Combinatorics thematic tutorial

Related material

- *Crystals* – Crystals

Cartan datum

- *Cartan types*
- *Dynkin diagrams*
- *Cartan matrices*
- *Coxeter Matrices*
- *Coxeter Types*

Root systems

- *Root Systems*
- *Tutorial: visualizing root systems*
- *Root lattice realizations*
- *Group algebras of root lattice realizations*
- *Weight lattice realizations*
- *Root lattices and root spaces*
- *Weight lattices and weight spaces*
- *Ambient lattices and ambient spaces*

Coxeter groups

- *Coxeter Groups*
- *Weyl Groups*
- *Extended Affine Weyl Groups*
- *Fundamental Group of an Extended Affine Weyl Group*
- *Braid Move Calculator*
- *Braid Orbit*

See also

The categories `CoxeterGroups` and `WeylGroups`

Finite reflection groups

- *Finite complex reflection groups*
- *Finite real reflection groups*

See also

The category `ComplexReflectionGroups`

Representation theory

- *Weyl Character Rings*
- Fusion Rings
- *Integrable Representations of Affine Lie Algebras*
- *Branching Rules*
- *Hecke algebra representations*
- *Nonsymmetric Macdonald polynomials*

Root system data and code for specific families of Cartan types

- *Root system data for affine Cartan types*
- *Root system data for dual Cartan types*
- *Root system data for folded Cartan types*
- *Root system data for reducible Cartan types*
- *Root system data for relabelled Cartan types*
- *Root system data for Cartan types with marked nodes*

Root system data and code for specific Cartan types

- *Root system data for type A*
- *Root system data for type B*
- *Root system data for type C*
- *Root system data for type D*
- *Root system data for type E*
- *Root system data for type F*
- *Root system data for type G*
- *Root system data for type H*
- *Root system data for type I*
- *Root system data for (untwisted) type A affine*
- *Root system data for (untwisted) type B affine*
- *Root system data for (untwisted) type C affine*
- *Root system data for (untwisted) type D affine*
- *Root system data for (untwisted) type E affine*
- *Root system data for (untwisted) type F affine*
- *Root system data for (untwisted) type G affine*
- *Root system data for type BC affine*
- *Root system data for super type A*
- *Root system data for type A infinity*

5.1.220 Ambient lattices and ambient spaces

class sage.combinat.root_system.ambient_space.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *CombinatorialFreeModule*

Abstract class for ambient spaces.

All subclasses should implement a class method `smallest_base_ring` taking a Cartan type as input, and a method `dimension` working on a partially initialized instance with just `root_system` as attribute. There is no safe default implementation for the later, so none is provided.

EXAMPLES:

```
sage: AL = RootSystem(['A',2]).ambient_lattice()
```

```
>>> from sage.all import *
>>> AL = RootSystem(['A',Integer(2)]).ambient_lattice()
```

Note

This is only used so far for finite root systems.

Caveat: Most of the ambient spaces currently have a basis indexed by $0, \ldots, n$, unlike the usual mathematical convention:

```
sage: e = AL.basis()
sage: e[0], e[1], e[2]
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

```
>>> from sage.all import *
>>> e = AL.basis()
>>> e[Integer(0)], e[Integer(1)], e[Integer(2)]
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

This will be cleaned up!

See also

- *sage.combinat.root_system.type_A.AmbientSpace*
- *sage.combinat.root_system.type_B.AmbientSpace*
- *sage.combinat.root_system.type_C.AmbientSpace*
- *sage.combinat.root_system.type_D.AmbientSpace*
- *sage.combinat.root_system.type_E.AmbientSpace*
- *sage.combinat.root_system.type_F.AmbientSpace*
- *sage.combinat.root_system.type_G.AmbientSpace*
- *sage.combinat.root_system.type_dual.AmbientSpace*
- *sage.combinat.root_system.type_affine.AmbientSpace*

Element

alias of *AmbientSpaceElement*

coroot_lattice()

EXAMPLES:

```
sage: e = RootSystem(["A", 3]).ambient_lattice()
sage: e.coroot_lattice()
Ambient lattice of the Root system of type ['A', 3]
```

```
>>> from sage.all import *
>>> e = RootSystem(["A", Integer(3)]).ambient_lattice()
>>> e.coroot_lattice()
Ambient lattice of the Root system of type ['A', 3]
```

dimension()

Return the dimension of this ambient space.

EXAMPLES:

```
sage: from sage.combinat.root_system.ambient_space import AmbientSpace
sage: e = RootSystem(['F',4]).ambient_space()
sage: AmbientSpace.dimension(e)
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.ambient_space import AmbientSpace
>>> e = RootSystem(['F',Integer(4)]).ambient_space()
>>> AmbientSpace.dimension(e)
Traceback (most recent call last):
...
NotImplementedError
```

from_vector_notation(*weight*, *style='lattice'*)

INPUT:

- `weight` – a vector or tuple representing a weight

Returns an element of `self`. If the weight lattice is not of full rank, it coerces it into the weight lattice, or its ambient space by orthogonal projection. This arises in two cases: for SL(r+1), the weight lattice is contained in a hyperplane of codimension one in the ambient, space, and for types E6 and E7, the weight lattice is contained in a subspace of codimensions 2 or 3, respectively.

If style="coroots" and the data is a tuple of integers, it is assumed that the data represent a linear combination of fundamental weights. If style='coroots', and the root lattice is not of full rank in the ambient space, it is projected into the subspace corresponding to the semisimple derived group. This arises with Cartan type A, E6 and E7.

EXAMPLES:

```
sage: RootSystem("A2").ambient_space().from_vector_notation((1,0,0))
(1, 0, 0)
sage: RootSystem("A2").ambient_space().from_vector_notation([1,0,0])
```

(continues on next page)

(continued from previous page)

```
(1, 0, 0)
sage: RootSystem("A2").ambient_space().from_vector_notation((1,0),style=
↪'coroots')
(2/3, -1/3, -1/3)
```

```
>>> from sage.all import *
>>> RootSystem("A2").ambient_space().from_vector_notation((Integer(1),
↪Integer(0),Integer(0)))
(1, 0, 0)
>>> RootSystem("A2").ambient_space().from_vector_notation([Integer(1),
↪Integer(0),Integer(0)])
(1, 0, 0)
>>> RootSystem("A2").ambient_space().from_vector_notation((Integer(1),
↪Integer(0)),style='coroots')
(2/3, -1/3, -1/3)
```

fundamental_weight (*i*)

Return the fundamental weight Λ_i in `self`.

In several of the ambient spaces, it is more convenient to construct all fundamental weights at once. To support this, we provide this default implementation of `fundamental_weight` using the method `fundamental_weights`. Beware that this will cause a loop if neither `fundamental_weight` nor `fundamental_weights` is implemented.

EXAMPLES:

```
sage: e =  RootSystem(['F',4]).ambient_space()
sage: e.fundamental_weight(3)
(3/2, 1/2, 1/2, 1/2)

sage: e =  RootSystem(['G',2]).ambient_space()
sage: e.fundamental_weight(1)
(1, 0, -1)

sage: e =  RootSystem(['E',6]).ambient_space()
sage: e.fundamental_weight(3)
(-1/2, 1/2, 1/2, 1/2, 1/2, -5/6, -5/6, 5/6)
```

```
>>> from sage.all import *
>>> e =  RootSystem(['F',Integer(4)]).ambient_space()
>>> e.fundamental_weight(Integer(3))
(3/2, 1/2, 1/2, 1/2)

>>> e =  RootSystem(['G',Integer(2)]).ambient_space()
>>> e.fundamental_weight(Integer(1))
(1, 0, -1)

>>> e =  RootSystem(['E',Integer(6)]).ambient_space()
>>> e.fundamental_weight(Integer(3))
(-1/2, 1/2, 1/2, 1/2, 1/2, -5/6, -5/6, 5/6)
```

reflection (*root*, *coroot=None*)

EXAMPLES:

```
sage: e = RootSystem(["A", 3]).ambient_lattice()
sage: a = e.simple_root(0); a
(-1, 0, 0, 0)
sage: b = e.simple_root(1); b
(1, -1, 0, 0)
sage: s_a = e.reflection(a)
sage: s_a(b)
(0, -1, 0, 0)
```

```
>>> from sage.all import *
>>> e = RootSystem(["A", Integer(3)]).ambient_lattice()
>>> a = e.simple_root(Integer(0)); a
(-1, 0, 0, 0)
>>> b = e.simple_root(Integer(1)); b
(1, -1, 0, 0)
>>> s_a = e.reflection(a)
>>> s_a(b)
(0, -1, 0, 0)
```

simple_coroot(*i*)

Return the i-th simple coroot, as an element of this space.

EXAMPLES:

```
sage: R = RootSystem(["A",3])
sage: L = R.ambient_lattice()
sage: L.simple_coroot(1)
(1, -1, 0, 0)
sage: L.simple_coroot(2)
(0, 1, -1, 0)
sage: L.simple_coroot(3)
(0, 0, 1, -1)
```

```
>>> from sage.all import *
>>> R = RootSystem(["A",Integer(3)])
>>> L = R.ambient_lattice()
>>> L.simple_coroot(Integer(1))
(1, -1, 0, 0)
>>> L.simple_coroot(Integer(2))
(0, 1, -1, 0)
>>> L.simple_coroot(Integer(3))
(0, 0, 1, -1)
```

classmethod smallest_base_ring(*cartan_type=None*)

Return the smallest ground ring over which the ambient space can be realized.

This class method will get called with the Cartan type as input. This default implementation returns $\mathbf{Q}$; subclasses should override it as appropriate.

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.smallest_base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> e = RootSystem(['F',Integer(4)]).ambient_space()
>>> e.smallest_base_ring()
Rational Field
```

to_ambient_space_morphism()

Return the identity map on `self`.

This is present for uniformity of use; the corresponding method for abstract root and weight lattices/spaces, is not trivial.

EXAMPLES:

```
sage: P = RootSystem(['A',2]).ambient_space()
sage: f = P.to_ambient_space_morphism()
sage: p = P.an_element()
sage: p
(2, 2, 3)
sage: f(p)
(2, 2, 3)
sage: f(p)==p
True
```

```
>>> from sage.all import *
>>> P = RootSystem(['A',Integer(2)]).ambient_space()
>>> f = P.to_ambient_space_morphism()
>>> p = P.an_element()
>>> p
(2, 2, 3)
>>> f(p)
(2, 2, 3)
>>> f(p)==p
True
```

class sage.combinat.root_system.ambient_space.**AmbientSpaceElement**

Bases: `IndexedFreeModuleElement`

associated_coroot()

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: a = e.simple_root(0); a
(1/2, -1/2, -1/2, -1/2)
sage: a.associated_coroot()
(1, -1, -1, -1)
```

```
>>> from sage.all import *
>>> e = RootSystem(['F',Integer(4)]).ambient_space()
>>> a = e.simple_root(Integer(0)); a
(1/2, -1/2, -1/2, -1/2)
>>> a.associated_coroot()
(1, -1, -1, -1)
```

coerce_to_e6()

For type E7 or E8, orthogonally projects an element of the root lattice into the E6 root lattice. This operation on weights corresponds to intersection with the semisimple subgroup E6.

EXAMPLES:

```
sage: [b.coerce_to_e6() for b in RootSystem("E8").ambient_space().basis()]
[(1, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, 0), (0, 0, 1, 0, 0, 0, 0, 0),
(0, 0, 0, 1, 0, 0, 0, 0), (0, 0, 0, 0, 1, 0, 0, 0), (0, 0, 0, 0, 0, 1/3, 1/3,
↪-1/3),
(0, 0, 0, 0, 0, 1/3, 1/3, -1/3), (0, 0, 0, 0, 0, -1/3, -1/3, 1/3)]
```

```
>>> from sage.all import *
>>> [b.coerce_to_e6() for b in RootSystem("E8").ambient_space().basis()]
[(1, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, 0), (0, 0, 1, 0, 0, 0, 0, 0),
(0, 0, 0, 1, 0, 0, 0, 0), (0, 0, 0, 0, 1, 0, 0, 0), (0, 0, 0, 0, 0, 1/3, 1/3,
↪-1/3),
(0, 0, 0, 0, 0, 1/3, 1/3, -1/3), (0, 0, 0, 0, 0, -1/3, -1/3, 1/3)]
```

coerce_to_e7()

For type E8, this orthogonally projects the given element of the E8 root lattice into the E7 root lattice. This operation on weights corresponds to intersection with the semisimple subgroup E7.

EXAMPLES:

```
sage: [b.coerce_to_e7() for b in RootSystem("E8").ambient_space().basis()]
[(1, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, 0),
 (0, 0, 1, 0, 0, 0, 0, 0), (0, 0, 0, 1, 0, 0, 0, 0),
 (0, 0, 0, 0, 1, 0, 0, 0), (0, 0, 0, 0, 0, 1, 0, 0),
 (0, 0, 0, 0, 0, 0, 1/2, -1/2), (0, 0, 0, 0, 0, 0, -1/2, 1/2)]
```

```
>>> from sage.all import *
>>> [b.coerce_to_e7() for b in RootSystem("E8").ambient_space().basis()]
[(1, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, 0),
 (0, 0, 1, 0, 0, 0, 0, 0), (0, 0, 0, 1, 0, 0, 0, 0),
 (0, 0, 0, 0, 1, 0, 0, 0), (0, 0, 0, 0, 0, 1, 0, 0),
 (0, 0, 0, 0, 0, 0, 1/2, -1/2), (0, 0, 0, 0, 0, 0, -1/2, 1/2)]
```

coerce_to_sl()

For type ['A',r], this coerces the element of the ambient space into the root space by orthogonal projection. The root space has codimension one and corresponds to the Lie algebra of SL(r+1,CC), whereas the full weight space corresponds to the Lie algebra of GL(r+1,CC). So this operation corresponds to multiplication by a (possibly fractional) power of the determinant to give a weight determinant one.

EXAMPLES:

```
sage: [fw.coerce_to_sl() for fw in RootSystem("A2").ambient_space().
↪fundamental_weights()]
[(2/3, -1/3, -1/3), (1/3, 1/3, -2/3)]
sage: L = RootSystem("A2xA3").ambient_space()
sage: L([1,2,3,4,5,0,0]).coerce_to_sl()
(-1, 0, 1, 7/4, 11/4, -9/4, -9/4)
```

```
>>> from sage.all import *
>>> [fw.coerce_to_sl() for fw in RootSystem("A2").ambient_space().fundamental_
↪weights()]
[(2/3, -1/3, -1/3), (1/3, 1/3, -2/3)]
>>> L = RootSystem("A2xA3").ambient_space()
>>> L([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(0),
↪Integer(0)]).coerce_to_sl()
(-1, 0, 1, 7/4, 11/4, -9/4, -9/4)
```

dot_product(*lambdacheck*)

The scalar product with elements of the ambient space.

EXAMPLES:

```
sage: e = RootSystem(['A',2]).ambient_space()
sage: a = e.simple_root(0); a
(-1, 0, 0)
sage: a.inner_product(a)
2
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(2)]).ambient_space()
>>> a = e.simple_root(Integer(0)); a
(-1, 0, 0)
>>> a.inner_product(a)
2
```

inner_product(*lambdacheck*)

The scalar product with elements of the ambient space.

EXAMPLES:

```
sage: e = RootSystem(['A',2]).ambient_space()
sage: a = e.simple_root(0); a
(-1, 0, 0)
sage: a.inner_product(a)
2
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(2)]).ambient_space()
>>> a = e.simple_root(Integer(0)); a
(-1, 0, 0)
>>> a.inner_product(a)
2
```

is_positive_root()

EXAMPLES:

```
sage: R = RootSystem(['A',3]).ambient_space()
sage: r=R.simple_root(1)+R.simple_root(2)
sage: r.is_positive_root()
True
sage: r=R.simple_root(1)-R.simple_root(2)
```

(continues on next page)

(continued from previous page)

```
sage: r.is_positive_root()
False
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(3)]).ambient_space()
>>> r=R.simple_root(Integer(1))+R.simple_root(Integer(2))
>>> r.is_positive_root()
True
>>> r=R.simple_root(Integer(1))-R.simple_root(Integer(2))
>>> r.is_positive_root()
False
```

scalar(*lambdacheck*)

The scalar product with elements of the ambient space.

EXAMPLES:

```
sage: e = RootSystem(['A',2]).ambient_space()
sage: a = e.simple_root(0); a
(-1, 0, 0)
sage: a.inner_product(a)
2
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(2)]).ambient_space()
>>> a = e.simple_root(Integer(0)); a
(-1, 0, 0)
>>> a.inner_product(a)
2
```

to_ambient()

Map `self` to the ambient space.

This exists for uniformity. Its analogue for root and weight lattice realizations, is not trivial.

EXAMPLES:

```
sage: v = CartanType(['C',3]).root_system().ambient_space().an_element(); v
(2, 2, 3)
sage: v.to_ambient()
(2, 2, 3)
```

```
>>> from sage.all import *
>>> v = CartanType(['C',Integer(3)]).root_system().ambient_space().an_
↪element(); v
(2, 2, 3)
>>> v.to_ambient()
(2, 2, 3)
```

5.1.221 Associahedron

Todo

- fix adjacency matrix
- edit graph method to get proper vertex labellings
- UniqueRepresentation?

AUTHORS:

- Christian Stump

`sage.combinat.root_system.associahedron.`**`Associahedra`**(*base_ring*, *ambient_dim*, *backend='ppl'*)

Construct a parent class of Associahedra according to `backend`.

See also

Associahedra_base.

class `sage.combinat.root_system.associahedron.`**`Associahedra_base`**

Bases: `object`

Base class of parent of Associahedra of specified dimension

EXAMPLES:

```
sage: from sage.combinat.root_system.associahedron import Associahedra
sage: parent = Associahedra(QQ,2,'ppl');  parent
Polyhedra in QQ^2
sage: type(parent)
<class 'sage.combinat.root_system.associahedron.Associahedra_ppl_with_category'>
sage: parent(['A',2])
Generalized associahedron of type ['A', 2] with 5 vertices
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.associahedron import Associahedra
>>> parent = Associahedra(QQ,Integer(2),'ppl');  parent
Polyhedra in QQ^2
>>> type(parent)
<class 'sage.combinat.root_system.associahedron.Associahedra_ppl_with_category'>
>>> parent(['A',Integer(2)])
Generalized associahedron of type ['A', 2] with 5 vertices
```

Importantly, the parent knows the dimension of the ambient space. If you try to construct an associahedron of a different dimension, a `ValueError` is raised:

```
sage: parent(['A',3])
Traceback (most recent call last):
...
ValueError: V-representation data requires a list of length ambient_dim
```

```
>>> from sage.all import *
>>> parent(['A',Integer(3)])
Traceback (most recent call last):
...
ValueError: V-representation data requires a list of length ambient_dim
```

class sage.combinat.root_system.associahedron.**Associahedra_cdd**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Associahedra_base*, Polyhedra_QQ_cdd

Element

alias of *Associahedron_class_cdd*

class sage.combinat.root_system.associahedron.**Associahedra_field**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Associahedra_base*, Polyhedra_field

Element

alias of *Associahedron_class_field*

class sage.combinat.root_system.associahedron.**Associahedra_normaliz**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Associahedra_base*, Polyhedra_QQ_normaliz

Element

alias of *Associahedron_class_normaliz*

class sage.combinat.root_system.associahedron.**Associahedra_polymake**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Associahedra_base*, Polyhedra_polymake

Element

alias of *Associahedron_class_polymake*

class sage.combinat.root_system.associahedron.**Associahedra_ppl**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Associahedra_base*, Polyhedra_QQ_ppl

Element

alias of *Associahedron_class_ppl*

sage.combinat.root_system.associahedron.**Associahedron**(*cartan_type*, *backend='ppl'*)

Construct an associahedron.

The generalized associahedron is a polytopal complex with vertices in one-to-one correspondence with clusters in the cluster complex, and with edges between two vertices if and only if the associated two clusters intersect in codimension 1.

The associahedron of type A_n is one way to realize the classical associahedron as defined in the Wikipedia article Associahedron.

A polytopal realization of the associahedron can be found in [CFZ2002]. The implementation is based on [CFZ2002], Theorem 1.5, Remark 1.6, and Corollary 1.9.

INPUT:

- cartan_type – a cartan type according to *sage.combinat.root_system.cartan_type.CartanTypeFactory*

- backend – string ('ppl'); the backend to use; see sage.geometry.polyhedron.constructor.Polyhedron()

EXAMPLES:

```
sage: Asso = polytopes.associahedron(['A',2]); Asso
Generalized associahedron of type ['A', 2] with 5 vertices

sage: sorted(Asso.Hrepresentation(), key=repr)
[An inequality (-1, 0) x + 1 >= 0,
 An inequality (0, -1) x + 1 >= 0,
 An inequality (0, 1) x + 1 >= 0,
 An inequality (1, 0) x + 1 >= 0,
 An inequality (1, 1) x + 1 >= 0]

sage: Asso.Vrepresentation()
(A vertex at (1, -1), A vertex at (1, 1), A vertex at (-1, 1),
 A vertex at (-1, 0), A vertex at (0, -1))

sage: polytopes.associahedron(['B',2])
Generalized associahedron of type ['B', 2] with 6 vertices
```

```
>>> from sage.all import *
>>> Asso = polytopes.associahedron(['A',Integer(2)]); Asso
Generalized associahedron of type ['A', 2] with 5 vertices

>>> sorted(Asso.Hrepresentation(), key=repr)
[An inequality (-1, 0) x + 1 >= 0,
 An inequality (0, -1) x + 1 >= 0,
 An inequality (0, 1) x + 1 >= 0,
 An inequality (1, 0) x + 1 >= 0,
 An inequality (1, 1) x + 1 >= 0]

>>> Asso.Vrepresentation()
(A vertex at (1, -1), A vertex at (1, 1), A vertex at (-1, 1),
 A vertex at (-1, 0), A vertex at (0, -1))

>>> polytopes.associahedron(['B',Integer(2)])
Generalized associahedron of type ['B', 2] with 6 vertices
```

The two pictures of [CFZ2002] can be recovered with:

```
sage: Asso = polytopes.associahedron(['A',3]); Asso
Generalized associahedron of type ['A', 3] with 14 vertices
sage: Asso.plot()   # long time
Graphics3d Object

sage: Asso = polytopes.associahedron(['B',3]); Asso
Generalized associahedron of type ['B', 3] with 20 vertices
sage: Asso.plot()   # long time
Graphics3d Object
```

```
>>> from sage.all import *
>>> Asso = polytopes.associahedron(['A',Integer(3)]); Asso
```

(continues on next page)

(continued from previous page)

```
Generalized associahedron of type ['A', 3] with 14 vertices
>>> Asso.plot()  # long time
Graphics3d Object

>>> Asso = polytopes.associahedron(['B',Integer(3)]); Asso
Generalized associahedron of type ['B', 3] with 20 vertices
>>> Asso.plot()  # long time
Graphics3d Object
```

class sage.combinat.root_system.associahedron.**Associahedron_class_base**(*parent=None*, *Vrep=None*, *Hrep=None*, *cartan_type=None*, ***kwds*)

Bases: `object`

The base class of the Python class of an associahedron

You should use the *Associahedron()* convenience function to construct associahedra from the Cartan type.

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: polytopes.associahedron(['A',3]).cartan_type()
['A', 3]
```

```
>>> from sage.all import *
>>> polytopes.associahedron(['A',Integer(3)]).cartan_type()
['A', 3]
```

vertices_in_root_space()

Return the vertices of `self` as elements in the root space.

EXAMPLES:

```
sage: Asso = polytopes.associahedron(['A',2])
sage: Asso.vertices()
(A vertex at (1, -1), A vertex at (1, 1),
 A vertex at (-1, 1), A vertex at (-1, 0),
 A vertex at (0, -1))

sage: Asso.vertices_in_root_space()
(alpha[1] - alpha[2], alpha[1] + alpha[2], -alpha[1] + alpha[2],
-alpha[1], -alpha[2])
```

```
>>> from sage.all import *
>>> Asso = polytopes.associahedron(['A',Integer(2)])
>>> Asso.vertices()
(A vertex at (1, -1), A vertex at (1, 1),
 A vertex at (-1, 1), A vertex at (-1, 0),
 A vertex at (0, -1))
```

(continues on next page)

(continued from previous page)

```
>>> Asso.vertices_in_root_space()
(alpha[1] - alpha[2], alpha[1] + alpha[2], -alpha[1] + alpha[2],
-alpha[1], -alpha[2])
```

class sage.combinat.root_system.associahedron.**Associahedron_class_cdd**(*parent=None*, *Vrep=None*, *Hrep=None*, *cartan_type=None*, ***kwds*)

Bases: *Associahedron_class_base*, Polyhedron_QQ_cdd

class sage.combinat.root_system.associahedron.**Associahedron_class_field**(*parent=None*, *Vrep=None*, *Hrep=None*, *cartan_type=None*, ***kwds*)

Bases: *Associahedron_class_base*, Polyhedron_field

class sage.combinat.root_system.associahedron.**Associahedron_class_normaliz**(*parent=None*, *Vrep=None*, *Hrep=None*, *cartan_type=None*, ***kwds*)

Bases: *Associahedron_class_base*, Polyhedron_QQ_normaliz

class sage.combinat.root_system.associahedron.**Associahedron_class_polymake**(*parent=None*, *Vrep=None*, *Hrep=None*, *cartan_type=None*, ***kwds*)

Bases: *Associahedron_class_base*, Polyhedron_polymake

class sage.combinat.root_system.associahedron.**Associahedron_class_ppl**(*parent=None*, *Vrep=None*, *Hrep=None*, *cartan_type=None*, ***kwds*)

Bases: *Associahedron_class_base*, Polyhedron_QQ_ppl

5.1.222 Braid Move Calculator

AUTHORS:

- Dinakar Muthiah (2014-06-03): initial version

class sage.combinat.root_system.braid_move_calculator.**BraidMoveCalculator**(*coxeter_group*)

Bases: object

Helper class to compute braid moves.

chain_of_reduced_words(*start_word*, *end_word*)

Compute the chain of reduced words from `stard_word` to `end_word`.

INPUT:

- `start_word`, `end_word` – two reduced expressions for the long word

EXAMPLES:

```
sage: from sage.combinat.root_system.braid_move_calculator import␣
↪BraidMoveCalculator
sage: W = CoxeterGroup(['A',5])
sage: B = BraidMoveCalculator(W)
sage: B.chain_of_reduced_words((1,2,1,3,2,1,4,3,2,1,5,4,3,2,1), # not tested
....:                          (5,4,5,3,4,5,2,3,4,5,1,2,3,4,5))
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.braid_move_calculator import␣
↪BraidMoveCalculator
>>> W = CoxeterGroup(['A',Integer(5)])
>>> B = BraidMoveCalculator(W)
>>> B.chain_of_reduced_words((Integer(1),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(1),Integer(4),Integer(3),Integer(2),Integer(1),
↪Integer(5),Integer(4),Integer(3),Integer(2),Integer(1)), # not tested
...                          (Integer(5),Integer(4),Integer(5),Integer(3),
↪Integer(4),Integer(5),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)))
```

put_in_front(*k*, *input_word*)

Return a list of reduced words starting with `input_word` and ending with a reduced word whose first letter is `k`.

There still remains an issue with 0 indices.

EXAMPLES:

```
sage: from sage.combinat.root_system.braid_move_calculator import␣
↪BraidMoveCalculator
sage: W = CoxeterGroup(['C',3])
sage: B = BraidMoveCalculator(W)
sage: B.put_in_front(2, (3, 2, 3, 1, 2, 3, 1, 2, 1))
((3, 2, 3, 1, 2, 3, 1, 2, 1),
 (3, 2, 3, 1, 2, 1, 3, 2, 1),
 (3, 2, 3, 2, 1, 2, 3, 2, 1),
 (2, 3, 2, 3, 1, 2, 3, 2, 1))
sage: B.put_in_front(1, (3, 2, 3, 1, 2, 3, 1, 2, 1))
((3, 2, 3, 1, 2, 3, 1, 2, 1),
 (3, 2, 1, 3, 2, 3, 1, 2, 1),
 (3, 2, 1, 3, 2, 3, 2, 1, 2),
 (3, 2, 1, 2, 3, 2, 3, 1, 2),
 (3, 1, 2, 1, 3, 2, 3, 1, 2),
 (1, 3, 2, 1, 3, 2, 3, 1, 2))
sage: B.put_in_front(1, (1, 3, 2, 3, 2, 1, 2, 3, 2))
((1, 3, 2, 3, 2, 1, 2, 3, 2),)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.braid_move_calculator import␣
↪BraidMoveCalculator
>>> W = CoxeterGroup(['C',Integer(3)])
>>> B = BraidMoveCalculator(W)
>>> B.put_in_front(Integer(2), (Integer(3), Integer(2), Integer(3),␣
↪Integer(1), Integer(2), Integer(3), Integer(1), Integer(2), Integer(1)))
((3, 2, 3, 1, 2, 3, 1, 2, 1),
 (3, 2, 3, 1, 2, 1, 3, 2, 1),
 (3, 2, 3, 2, 1, 2, 3, 2, 1),
 (2, 3, 2, 3, 1, 2, 3, 2, 1))
>>> B.put_in_front(Integer(1), (Integer(3), Integer(2), Integer(3),␣
↪Integer(1), Integer(2), Integer(3), Integer(1), Integer(2), Integer(1)))
((3, 2, 3, 1, 2, 3, 1, 2, 1),
 (3, 2, 1, 3, 2, 3, 1, 2, 1),
 (3, 2, 1, 3, 2, 3, 2, 1, 2),
 (3, 2, 1, 2, 3, 2, 3, 1, 2),
 (3, 1, 2, 1, 3, 2, 3, 1, 2),
 (1, 3, 2, 1, 3, 2, 3, 1, 2))
>>> B.put_in_front(Integer(1), (Integer(1), Integer(3), Integer(2),␣
↪Integer(3), Integer(2), Integer(1), Integer(2), Integer(3), Integer(2)))
((1, 3, 2, 3, 2, 1, 2, 3, 2),)
```

5.1.223 Braid Orbit

Cython function to compute the orbit of the braid moves on a reduced word.

`sage.combinat.root_system.braid_orbit.`**`BraidOrbit`**`(`*word*, *rels*`)`

Return the orbit of `word` by all replacements given by `rels`.

INPUT:

- `word` – list of integers
- `rels` – list of pairs `(A, B)`, where `A` and `B` are lists of integers the same length

EXAMPLES:

```
sage: from sage.combinat.root_system.braid_orbit import BraidOrbit
sage: word = [1,2,1,3,2,1]
sage: rels = [[[2, 1, 2], [1, 2, 1]], [[3, 1], [1, 3]], [[3, 2, 3], [2, 3, 2]]]
sage: sorted(BraidOrbit(word, rels))
[(1, 2, 1, 3, 2, 1),
 (1, 2, 3, 1, 2, 1),
 (1, 2, 3, 2, 1, 2),
 (1, 3, 2, 1, 3, 2),
 (1, 3, 2, 3, 1, 2),
 (2, 1, 2, 3, 2, 1),
 (2, 1, 3, 2, 1, 3),
 (2, 1, 3, 2, 3, 1),
 (2, 3, 1, 2, 1, 3),
 (2, 3, 1, 2, 3, 1),
 (2, 3, 2, 1, 2, 3),
 (3, 1, 2, 1, 3, 2),
 (3, 1, 2, 3, 1, 2),
```

(continues on next page)

(continued from previous page)

```
 (3, 2, 1, 2, 3, 2),
 (3, 2, 1, 3, 2, 3),
 (3, 2, 3, 1, 2, 3)]
sage: len(_)
16
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.braid_orbit import BraidOrbit
>>> word = [Integer(1),Integer(2),Integer(1),Integer(3),Integer(2),Integer(1)]
>>> rels = [[[Integer(2), Integer(1), Integer(2)], [Integer(1), Integer(2),␣
↪Integer(1)]], [[Integer(3), Integer(1)], [Integer(1), Integer(3)]],␣
↪[[Integer(3), Integer(2), Integer(3)], [Integer(2), Integer(3), Integer(2)]]]
>>> sorted(BraidOrbit(word, rels))
[(1, 2, 1, 3, 2, 1),
 (1, 2, 3, 1, 2, 1),
 (1, 2, 3, 2, 1, 2),
 (1, 3, 2, 1, 3, 2),
 (1, 3, 2, 3, 1, 2),
 (2, 1, 2, 3, 2, 1),
 (2, 1, 3, 2, 1, 3),
 (2, 1, 3, 2, 3, 1),
 (2, 3, 1, 2, 1, 3),
 (2, 3, 1, 2, 3, 1),
 (2, 3, 2, 1, 2, 3),
 (3, 1, 2, 1, 3, 2),
 (3, 1, 2, 3, 1, 2),
 (3, 2, 1, 2, 3, 2),
 (3, 2, 1, 3, 2, 3),
 (3, 2, 3, 1, 2, 3)]
>>> len(_)
16
```

sage.combinat.root_system.braid_orbit.**is_fully_commutative**(*word*, *rels*)

Check if the braid orbit of `word` is using a braid relation.

INPUT:

- `word` – list of integers
- `rels` – list of pairs `(A, B)`, where `A` and `B` are lists of integers the same length

EXAMPLES:

```
sage: from sage.combinat.root_system.braid_orbit import is_fully_commutative
sage: rels = [[[2, 1, 2], [1, 2, 1]], [[3, 1], [1, 3]], [[3, 2, 3], [2, 3, 2]]]
sage: word = [1,2,1,3,2,1]
sage: is_fully_commutative(word, rels)
False
sage: word = [1,2,3]
sage: is_fully_commutative(word, rels)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.braid_orbit import is_fully_commutative
>>> rels = [[[Integer(2), Integer(1), Integer(2)], [Integer(1), Integer(2),␣
↪Integer(1)]], [[Integer(3), Integer(1)], [Integer(1), Integer(3)]],␣
↪[[Integer(3), Integer(2), Integer(3)], [Integer(2), Integer(3), Integer(2)]]]
>>> word = [Integer(1),Integer(2),Integer(1),Integer(3),Integer(2),Integer(1)]
>>> is_fully_commutative(word, rels)
False
>>> word = [Integer(1),Integer(2),Integer(3)]
>>> is_fully_commutative(word, rels)
True
```

5.1.224 Branching Rules

class sage.combinat.root_system.branching_rules.**BranchingRule**(*R*, *S*, *f*, *name='default'*, *intermediate_types=[]*, *intermediate_names=[]*)

Bases: `SageObject`

A class for branching rules.

Rtype()

In a branching rule R => S, returns the Cartan Type of the ambient group R.

EXAMPLES:

```
sage: branching_rule("A3","A2","levi").Rtype()
['A', 3]
```

```
>>> from sage.all import *
>>> branching_rule("A3","A2","levi").Rtype()
['A', 3]
```

Stype()

In a branching rule R => S, returns the Cartan Type of the subgroup S.

EXAMPLES:

```
sage: branching_rule("A3","A2","levi").Stype()
['A', 2]
```

```
>>> from sage.all import *
>>> branching_rule("A3","A2","levi").Stype()
['A', 2]
```

branch(*chi*, *style=None*)

INPUT:

- `chi` – a character of the WeylCharacterRing with Cartan type self.Rtype()

Returns the branched character.

EXAMPLES:

```
sage: G2=WeylCharacterRing("G2",style='coroots')
sage: chi=G2(1,1); chi.degree()
64
sage: b=G2.maximal_subgroup("A2"); b
extended branching rule G2 => A2
sage: b.branch(chi)
A2(0,1) + A2(1,0) + A2(0,2) + 2*A2(1,1) + A2(2,0) + A2(1,2) + A2(2,1)
sage: A2=WeylCharacterRing("A2",style='coroots'); A2
The Weyl Character Ring of Type A2 with Integer Ring coefficients
sage: chi.branch(A2,rule=b)
A2(0,1) + A2(1,0) + A2(0,2) + 2*A2(1,1) + A2(2,0) + A2(1,2) + A2(2,1)
```

```
>>> from sage.all import *
>>> G2=WeylCharacterRing("G2",style='coroots')
>>> chi=G2(Integer(1),Integer(1)); chi.degree()
64
>>> b=G2.maximal_subgroup("A2"); b
extended branching rule G2 => A2
>>> b.branch(chi)
A2(0,1) + A2(1,0) + A2(0,2) + 2*A2(1,1) + A2(2,0) + A2(1,2) + A2(2,1)
>>> A2=WeylCharacterRing("A2",style='coroots'); A2
The Weyl Character Ring of Type A2 with Integer Ring coefficients
>>> chi.branch(A2,rule=b)
A2(0,1) + A2(1,0) + A2(0,2) + 2*A2(1,1) + A2(2,0) + A2(1,2) + A2(2,1)
```

describe(*verbose=False*, *debug=False*, *no_r=False*)

Describe how extended roots restrict under `self`.

EXAMPLES:

```
sage: branching_rule("G2","A2","extended").describe()

3
O=<=O---O
1   2   0
G2~

root restrictions G2 => A2:

O---O
1   2
A2

0 => 2
2 => 1

For more detailed information use verbose=True
```

```
>>> from sage.all import *
>>> branching_rule("G2","A2","extended").describe()
<BLANKLINE>
3
O=<=O---O
```

(continues on next page)

(continued from previous page)

```
1   2   0
G2~
<BLANKLINE>
root restrictions G2 => A2:
<BLANKLINE>
O---O
1   2
A2
<BLANKLINE>
0 => 2
2 => 1
<BLANKLINE>
For more detailed information use verbose=True
```

In this example, 0 is the affine root, that is, the negative of the highest root, for "$G2$". If $i \geq j$ is printed, this means that the i-th simple (or affine) root of the ambient group restricts to the j-th simple root of the subgroup. For reference the Dynkin diagrams are also printed. The extended Dynkin diagram of the ambient group is printed if the affine root restricts to a simple root. More information is printed if the parameter $verbose$ is true.

`sage.combinat.root_system.branching_rules.`**`branch_weyl_character`**`(`*chi*, *R*, *S*, *rule='default'*`)`

A branching rule describes the restriction of representations from a Lie group or algebra G to a subgroup H. See for example, R. C. King, Branching rules for classical Lie groups using tensor and spinor methods. J. Phys. A 8 (1975), 429-449, Howe, Tan and Willenbring, Stable branching rules for classical symmetric pairs, Trans. Amer. Math. Soc. 357 (2005), no. 4, 1601-1626, McKay and Patera, Tables of Dimensions, Indices and Branching Rules for Representations of Simple Lie Algebras (Marcel Dekker, 1981), and Fauser, Jarvis, King and Wybourne, New branching rules induced by plethysm. J. Phys. A 39 (2006), no. 11, 2611–2655. If $H \subset G$ we will write $G \Rightarrow H$ to denote the branching rule, which is a homomorphism of WeylCharacterRings.

INPUT:

- `chi` – a character of G
- `R` – the Weyl Character Ring of G
- `S` – the Weyl Character Ring of H
- `rule` – an element of the `BranchingRule` class or one (most usually) a keyword such as:
  - `'levi'`
  - `'automorphic'`
  - `'symmetric'`
  - `'extended'`
  - `'orthogonal_sum'`
  - `'tensor'`
  - `'triality'`
  - `'miscellaneous'`

The *`BranchingRule`* class is a wrapper for functions from the weight lattice of G to the weight lattice of H. An instance of this class encodes an embedding of H into G. The usual way to specify an embedding is to supply a keyword, which tells Sage to use one of the built-in rules. We will discuss these first.

To explain the predefined rules, we survey the most important branching rules. These may be classified into several cases, and once this is understood, the detailed classification can be read off from the Dynkin diagrams. Dynkin classified the maximal subgroups of Lie groups in Mat. Sbornik N.S. 30(72):349-462 (1952).

We will list give predefined rules that cover most cases where the branching rule is to a maximal subgroup. For convenience, we also give some branching rules to subgroups that are not maximal. For example, a Levi subgroup may or may not be maximal.

For example, there is a "levi" branching rule defined from $SL(5)$ (with Cartan type A_4) to $SL(4)$ (with Cartan type A_3), so we may compute the branching rule as follows:

EXAMPLES:

```
sage: A3=WeylCharacterRing("A3",style='coroots')
sage: A2=WeylCharacterRing("A2",style='coroots')
sage: [A3(fw).branch(A2,rule='levi') for fw in A3.fundamental_weights()]
[A2(0,0) + A2(1,0), A2(0,1) + A2(1,0), A2(0,0) + A2(0,1)]
```

```
>>> from sage.all import *
>>> A3=WeylCharacterRing("A3",style='coroots')
>>> A2=WeylCharacterRing("A2",style='coroots')
>>> [A3(fw).branch(A2,rule='levi') for fw in A3.fundamental_weights()]
[A2(0,0) + A2(1,0), A2(0,1) + A2(1,0), A2(0,0) + A2(0,1)]
```

In this case the Levi branching rule is the default branching rule so we may omit the specification rule="levi".

If a subgroup is not maximal, you may specify a branching rule by finding a chain of intermediate subgroups. For this purpose, branching rules may be multiplied as in the following example.

EXAMPLES:

```
sage: A4=WeylCharacterRing("A4",style='coroots')
sage: A2=WeylCharacterRing("A2",style='coroots')
sage: br=branching_rule("A4","A3")*branching_rule("A3","A2")
sage: A4(1,0,0,0).branch(A2,rule=br)
2*A2(0,0) + A2(1,0)
```

```
>>> from sage.all import *
>>> A4=WeylCharacterRing("A4",style='coroots')
>>> A2=WeylCharacterRing("A2",style='coroots')
>>> br=branching_rule("A4","A3")*branching_rule("A3","A2")
>>> A4(Integer(1),Integer(0),Integer(0),Integer(0)).branch(A2,rule=br)
2*A2(0,0) + A2(1,0)
```

You may try omitting the rule if it is "obvious". Default rules are provided for the following cases:

$$\begin{aligned} A_{2s} &\Rightarrow B_s, \\ A_{2s-1} &\Rightarrow C_s, \\ A_{2*s-1} &\Rightarrow D_s. \end{aligned}$$

The above default rules correspond to embedding the group $SO(2s+1)$, $Sp(2s)$ or $SO(2s)$ into the corresponding general or special linear group by the standard representation. Default rules are also specified for the following cases:

$$\begin{aligned} B_{s+1} &\Rightarrow D_s, \\ D_s &\Rightarrow B_s. \end{aligned}$$

These correspond to the embedding of $O(n)$ into $O(n+1)$ where $n = 2s$ or $2s+1$. Finally, the branching rule for the embedding of a Levi subgroup is also implemented as a default rule.

EXAMPLES:

```
sage: A1 = WeylCharacterRing("A1", style='coroots')
sage: A2 = WeylCharacterRing("A2", style='coroots')
sage: D4 = WeylCharacterRing("D4", style='coroots')
sage: B3 = WeylCharacterRing("B3", style='coroots')
sage: B4 = WeylCharacterRing("B4", style='coroots')
sage: A6 = WeylCharacterRing("A6", style='coroots')
sage: A7 = WeylCharacterRing("A7", style='coroots')
sage: def try_default_rule(R, S): return [R(f).branch(S) for f in R.fundamental_
↪weights()]
sage: try_default_rule(A2,A1)
[A1(0) + A1(1), A1(0) + A1(1)]
sage: try_default_rule(D4,B3)
[B3(0,0,0) + B3(1,0,0), B3(1,0,0) + B3(0,1,0), B3(0,0,1), B3(0,0,1)]
sage: try_default_rule(B4,D4)
[D4(0,0,0,0) + D4(1,0,0,0), D4(1,0,0,0) + D4(0,1,0,0),
D4(0,1,0,0) + D4(0,0,1,1), D4(0,0,1,0) + D4(0,0,0,1)]
sage: try_default_rule(A7,D4)
[D4(1,0,0,0), D4(0,1,0,0), D4(0,0,1,1), D4(0,0,2,0) + D4(0,0,0,2),
D4(0,0,1,1),
D4(0,1,0,0),
D4(1,0,0,0)]
sage: try_default_rule(A6,B3)
[B3(1,0,0), B3(0,1,0), B3(0,0,2), B3(0,0,2), B3(0,1,0), B3(1,0,0)]
```

```
>>> from sage.all import *
>>> A1 = WeylCharacterRing("A1", style='coroots')
>>> A2 = WeylCharacterRing("A2", style='coroots')
>>> D4 = WeylCharacterRing("D4", style='coroots')
>>> B3 = WeylCharacterRing("B3", style='coroots')
>>> B4 = WeylCharacterRing("B4", style='coroots')
>>> A6 = WeylCharacterRing("A6", style='coroots')
>>> A7 = WeylCharacterRing("A7", style='coroots')
>>> def try_default_rule(R, S): return [R(f).branch(S) for f in R.fundamental_
↪weights()]
>>> try_default_rule(A2,A1)
[A1(0) + A1(1), A1(0) + A1(1)]
>>> try_default_rule(D4,B3)
[B3(0,0,0) + B3(1,0,0), B3(1,0,0) + B3(0,1,0), B3(0,0,1), B3(0,0,1)]
>>> try_default_rule(B4,D4)
[D4(0,0,0,0) + D4(1,0,0,0), D4(1,0,0,0) + D4(0,1,0,0),
D4(0,1,0,0) + D4(0,0,1,1), D4(0,0,1,0) + D4(0,0,0,1)]
>>> try_default_rule(A7,D4)
[D4(1,0,0,0), D4(0,1,0,0), D4(0,0,1,1), D4(0,0,2,0) + D4(0,0,0,2),
D4(0,0,1,1),
D4(0,1,0,0),
D4(1,0,0,0)]
>>> try_default_rule(A6,B3)
[B3(1,0,0), B3(0,1,0), B3(0,0,2), B3(0,0,2), B3(0,1,0), B3(1,0,0)]
```

If a default rule is not known, you may cue Sage as to what the Lie group embedding is by supplying a rule from the list of predefined rules. We will treat these next.

Levi Type

These can be read off from the Dynkin diagram. If removing a node from the Dynkin diagram produces another Dynkin diagram, there is a branching rule. A Levi subgroup may or may not be maximal. If it is maximal, there may or may not be a built-in branching rule for but you may obtain the Levi branching rule by first branching to a suitable maximal subgroup. For these rules use the option `rule="levi"`:

$$
\begin{aligned}
A_r &\Rightarrow A_{r-1} \\
B_r &\Rightarrow A_{r-1} \\
B_r &\Rightarrow B_{r-1} \\
C_r &\Rightarrow A_{r-1} \\
C_r &\Rightarrow C_{r-1} \\
D_r &\Rightarrow A_{r-1} \\
D_r &\Rightarrow D_{r-1} \\
E_r &\Rightarrow A_{r-1} \quad r = 7, 8 \\
E_r &\Rightarrow D_{r-1} \quad r = 6, 7, 8 \\
E_r &\Rightarrow E_{r-1} \\
F_4 &\Rightarrow B_3 \\
F_4 &\Rightarrow C_3 \\
G_2 &\Rightarrow A_1 (\text{short root})
\end{aligned}
$$

Not all Levi subgroups are maximal subgroups. If the Levi is not maximal there may or may not be a preprogrammed `rule="levi"` for it. If there is not, the branching rule may still be obtained by going through an intermediate subgroup that is maximal using rule="extended". Thus the other Levi branching rule from $G_2 \Rightarrow A_1$ corresponding to the long root is available by first branching $G_2 \Rightarrow A_2$ then $A_2 \Rightarrow A_1$. Similarly the branching rules to the Levi subgroup:

$$E_r \Rightarrow A_{r-1} \quad r = 6, 7, 8$$

may be obtained by first branching $E_6 \Rightarrow A_5 \times A_1$, $E_7 \Rightarrow A_7$ or $E_8 \Rightarrow A_8$.

EXAMPLES:

```
sage: A1 = WeylCharacterRing("A1")
sage: A2 = WeylCharacterRing("A2")
sage: A3 = WeylCharacterRing("A3")
sage: A4 = WeylCharacterRing("A4")
sage: A5 = WeylCharacterRing("A5")
sage: B2 = WeylCharacterRing("B2")
sage: B3 = WeylCharacterRing("B3")
sage: B4 = WeylCharacterRing("B4")
sage: C2 = WeylCharacterRing("C2")
sage: C3 = WeylCharacterRing("C3")
sage: D3 = WeylCharacterRing("D3")
sage: D4 = WeylCharacterRing("D4")
sage: G2 = WeylCharacterRing("G2")
sage: F4 = WeylCharacterRing("F4",style='coroots')
sage: E6=WeylCharacterRing("E6",style='coroots')
sage: E7=WeylCharacterRing("E7",style='coroots')
sage: D5=WeylCharacterRing("D5",style='coroots')
sage: D6=WeylCharacterRing("D6",style='coroots')
sage: [B3(w).branch(A2,rule='levi') for w in B3.fundamental_weights()]
```

(continues on next page)

(continued from previous page)

```
[A2(0,0,0) + A2(1,0,0) + A2(0,0,-1),
A2(0,0,0) + A2(1,0,0) + A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1) + A2(0,0,-1),
A2(-1/2,-1/2,-1/2) + A2(1/2,-1/2,-1/2) + A2(1/2,1/2,-1/2) + A2(1/2,1/2,1/2)]
```

```
>>> from sage.all import *
>>> A1 = WeylCharacterRing("A1")
>>> A2 = WeylCharacterRing("A2")
>>> A3 = WeylCharacterRing("A3")
>>> A4 = WeylCharacterRing("A4")
>>> A5 = WeylCharacterRing("A5")
>>> B2 = WeylCharacterRing("B2")
>>> B3 = WeylCharacterRing("B3")
>>> B4 = WeylCharacterRing("B4")
>>> C2 = WeylCharacterRing("C2")
>>> C3 = WeylCharacterRing("C3")
>>> D3 = WeylCharacterRing("D3")
>>> D4 = WeylCharacterRing("D4")
>>> G2 = WeylCharacterRing("G2")
>>> F4 = WeylCharacterRing("F4",style='coroots')
>>> E6=WeylCharacterRing("E6",style='coroots')
>>> E7=WeylCharacterRing("E7",style='coroots')
>>> D5=WeylCharacterRing("D5",style='coroots')
>>> D6=WeylCharacterRing("D6",style='coroots')
>>> [B3(w).branch(A2,rule='levi') for w in B3.fundamental_weights()]
[A2(0,0,0) + A2(1,0,0) + A2(0,0,-1),
A2(0,0,0) + A2(1,0,0) + A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1) + A2(0,0,-1),
A2(-1/2,-1/2,-1/2) + A2(1/2,-1/2,-1/2) + A2(1/2,1/2,-1/2) + A2(1/2,1/2,1/2)]
```

The last example must be understood as follows. The representation of B_3 being branched is spin, which is not a representation of $SO(7)$ but of its double cover spin(7). The group A_2 is really GL(3) and the double cover of $SO(7)$ induces a cover of $GL(3)$ that is trivial over $SL(3)$ but not over the center of $GL(3)$. The weight lattice for this $GL(3)$ consists of triples (a, b, c) of half integers such that $a - b$ and $b - c$ are in **Z**, and this is reflected in the last decomposition.

```
sage: [C3(w).branch(A2,rule='levi') for w in C3.fundamental_weights()]
[A2(1,0,0) + A2(0,0,-1),
A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1),
A2(-1,-1,-1) + A2(1,-1,-1) + A2(1,1,-1) + A2(1,1,1)]
sage: [D4(w).branch(A3,rule='levi') for w in D4.fundamental_weights()]
[A3(1,0,0,0) + A3(0,0,0,-1),
A3(0,0,0,0) + A3(1,1,0,0) + A3(1,0,0,-1) + A3(0,0,-1,-1),
A3(1/2,-1/2,-1/2,-1/2) + A3(1/2,1/2,1/2,-1/2),
A3(-1/2,-1/2,-1/2,-1/2) + A3(1/2,1/2,-1/2,-1/2) + A3(1/2,1/2,1/2,1/2)]
sage: [B3(w).branch(B2,rule='levi') for w in B3.fundamental_weights()]
[2*B2(0,0) + B2(1,0), B2(0,0) + 2*B2(1,0) + B2(1,1), 2*B2(1/2,1/2)]
sage: C3 = WeylCharacterRing(['C',3])
sage: [C3(w).branch(C2,rule='levi') for w in C3.fundamental_weights()]
[2*C2(0,0) + C2(1,0),
 C2(0,0) + 2*C2(1,0) + C2(1,1),
 C2(1,0) + 2*C2(1,1)]
sage: [D5(w).branch(D4,rule='levi') for w in D5.fundamental_weights()]
[2*D4(0,0,0,0) + D4(1,0,0,0),
```

(continues on next page)

(continued from previous page)

```
 D4(0,0,0,0) + 2*D4(1,0,0,0) + D4(1,1,0,0),
 D4(1,0,0,0) + 2*D4(1,1,0,0) + D4(1,1,1,0),
 D4(1/2,1/2,1/2,-1/2) + D4(1/2,1/2,1/2,1/2),
 D4(1/2,1/2,1/2,-1/2) + D4(1/2,1/2,1/2,1/2)]
sage: G2(1,0,-1).branch(A1,rule='levi')
A1(1,0) + A1(1,-1) + A1(0,-1)
sage: E6=WeylCharacterRing("E6",style='coroots')
sage: D5=WeylCharacterRing("D5",style='coroots')
sage: fw = E6.fundamental_weights()
sage: [E6(fw[i]).branch(D5,rule='levi') for i in [1,2,6]]
[D5(0,0,0,0,0) + D5(0,0,0,0,1) + D5(1,0,0,0,0),
 D5(0,0,0,0,0) + D5(0,0,0,1,0) + D5(0,0,0,0,1) + D5(0,1,0,0,0),
 D5(0,0,0,0,0) + D5(0,0,0,1,0) + D5(1,0,0,0,0)]
 sage: E7=WeylCharacterRing("E7",style='coroots')
 sage: A3xA3xA1=WeylCharacterRing("A3xA3xA1",style='coroots')
 sage: E7(1,0,0,0,0,0,0).branch(A3xA3xA1,rule='extended') # long time (0.7s)
 A3xA3xA1(0,0,1,0,0,1,1) + A3xA3xA1(0,1,0,0,1,0,0) + A3xA3xA1(1,0,0,1,0,0,1) +
  A3xA3xA1(1,0,1,0,0,0,0) + A3xA3xA1(0,0,0,1,0,1,0) + A3xA3xA1(0,0,0,0,0,0,2)
sage: fw = E7.fundamental_weights()
sage: [E7(fw[i]).branch(D6,rule='levi') for i in [1,2,7]] # long time (0.3s)
[3*D6(0,0,0,0,0,0) + 2*D6(0,0,0,0,1,0) + D6(0,1,0,0,0,0),
 3*D6(0,0,0,0,0,1) + 2*D6(1,0,0,0,0,0) + 2*D6(0,0,1,0,0,0) + D6(1,0,0,0,1,0),
 D6(0,0,0,0,0,1) + 2*D6(1,0,0,0,0,0)]
sage: D7=WeylCharacterRing("D7",style='coroots')
sage: E8=WeylCharacterRing("E8",style='coroots')
sage: D7=WeylCharacterRing("D7",style='coroots')
sage: E8(1,0,0,0,0,0,0,0).branch(D7,rule='levi') # long time (7s)
 3*D7(0,0,0,0,0,0,0) + 2*D7(0,0,0,0,0,1,0) + 2*D7(0,0,0,0,0,0,1) + 2*D7(1,0,0,0,0,
↪0,0)
 + D7(0,1,0,0,0,0,0) + 2*D7(0,0,1,0,0,0,0) + D7(0,0,0,1,0,0,0) + D7(1,0,0,0,0,1,
↪0) + D7(1,0,0,0,0,0,1) + D7(2,0,0,0,0,0,0)
sage: E8(0,0,0,0,0,0,0,1).branch(D7,rule='levi') # long time (0.6s)
 D7(0,0,0,0,0,0,0) + D7(0,0,0,0,0,1,0) + D7(0,0,0,0,0,0,1) + 2*D7(1,0,0,0,0,0,0)␣
↪+ D7(0,1,0,0,0,0,0)
sage: [F4(fw).branch(B3,rule='levi') for fw in F4.fundamental_weights()] # long␣
↪time (1s)
 [B3(0,0,0) + 2*B3(1/2,1/2,1/2) + 2*B3(1,0,0) + B3(1,1,0),
 B3(0,0,0) + 6*B3(1/2,1/2,1/2) + 5*B3(1,0,0) + 7*B3(1,1,0) + 3*B3(1,1,1)
 + 6*B3(3/2,1/2,1/2) + 2*B3(3/2,3/2,1/2) + B3(2,0,0) + 2*B3(2,1,0) + B3(2,1,1),
 3*B3(0,0,0) + 6*B3(1/2,1/2,1/2) + 4*B3(1,0,0) + 3*B3(1,1,0) + B3(1,1,1) + 2*B3(3/
↪2,1/2,1/2),
 3*B3(0,0,0) + 2*B3(1/2,1/2,1/2) + B3(1,0,0)]
sage: [F4(fw).branch(C3,rule='levi') for fw in F4.fundamental_weights()] # long␣
↪time (1s)
 [3*C3(0,0,0) + 2*C3(1,1,1) + C3(2,0,0),
 3*C3(0,0,0) + 6*C3(1,1,1) + 4*C3(2,0,0) + 2*C3(2,1,0) + 3*C3(2,2,0) + C3(2,2,2)␣
↪+ C3(3,1,0) + 2*C3(3,1,1),
 2*C3(1,0,0) + 3*C3(1,1,0) + C3(2,0,0) + 2*C3(2,1,0) + C3(2,1,1),
 2*C3(1,0,0) + C3(1,1,0)]
sage: A1xA1 = WeylCharacterRing("A1xA1")
sage: [A3(hwv).branch(A1xA1,rule='levi') for hwv in A3.fundamental_weights()]
[A1xA1(1,0,0,0) + A1xA1(0,0,1,0),
```

(continues on next page)

(continued from previous page)

```
A1xA1(1,1,0,0) + A1xA1(1,0,1,0) + A1xA1(0,0,1,1),
A1xA1(1,1,1,0) + A1xA1(1,0,1,1)]
sage: A1xB1=WeylCharacterRing("A1xB1",style='coroots')
sage: [B3(x).branch(A1xB1,rule='levi') for x in B3.fundamental_weights()]
[2*A1xB1(1,0) + A1xB1(0,2),
3*A1xB1(0,0) + 2*A1xB1(1,2) + A1xB1(2,0) + A1xB1(0,2),
A1xB1(1,1) + 2*A1xB1(0,1)]
```

```
>>> from sage.all import *
>>> [C3(w).branch(A2,rule='levi') for w in C3.fundamental_weights()]
[A2(1,0,0) + A2(0,0,-1),
A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1),
A2(-1,-1,-1) + A2(1,-1,-1) + A2(1,1,-1) + A2(1,1,1)]
>>> [D4(w).branch(A3,rule='levi') for w in D4.fundamental_weights()]
[A3(1,0,0,0) + A3(0,0,0,-1),
A3(0,0,0,0) + A3(1,1,0,0) + A3(1,0,0,-1) + A3(0,0,-1,-1),
A3(1/2,-1/2,-1/2,-1/2) + A3(1/2,1/2,1/2,-1/2),
A3(-1/2,-1/2,-1/2,-1/2) + A3(1/2,1/2,-1/2,-1/2) + A3(1/2,1/2,1/2,1/2)]
>>> [B3(w).branch(B2,rule='levi') for w in B3.fundamental_weights()]
[2*B2(0,0) + B2(1,0), B2(0,0) + 2*B2(1,0) + B2(1,1), 2*B2(1/2,1/2)]
>>> C3 = WeylCharacterRing(['C',Integer(3)])
>>> [C3(w).branch(C2,rule='levi') for w in C3.fundamental_weights()]
[2*C2(0,0) + C2(1,0),
 C2(0,0) + 2*C2(1,0) + C2(1,1),
 C2(1,0) + 2*C2(1,1)]
>>> [D5(w).branch(D4,rule='levi') for w in D5.fundamental_weights()]
[2*D4(0,0,0,0) + D4(1,0,0,0),
 D4(0,0,0,0) + 2*D4(1,0,0,0) + D4(1,1,0,0),
 D4(1,0,0,0) + 2*D4(1,1,0,0) + D4(1,1,1,0),
 D4(1/2,1/2,1/2,-1/2) + D4(1/2,1/2,1/2,1/2),
 D4(1/2,1/2,1/2,-1/2) + D4(1/2,1/2,1/2,1/2)]
>>> G2(Integer(1),Integer(0),-Integer(1)).branch(A1,rule='levi')
A1(1,0) + A1(1,-1) + A1(0,-1)
>>> E6=WeylCharacterRing("E6",style='coroots')
>>> D5=WeylCharacterRing("D5",style='coroots')
>>> fw = E6.fundamental_weights()
>>> [E6(fw[i]).branch(D5,rule='levi') for i in [Integer(1),Integer(2),Integer(6)]]
[D5(0,0,0,0,0) + D5(0,0,0,0,1) + D5(1,0,0,0,0),
 D5(0,0,0,0,0) + D5(0,0,0,1,0) + D5(0,0,0,0,1) + D5(0,1,0,0,0),
 D5(0,0,0,0,0) + D5(0,0,0,1,0) + D5(1,0,0,0,0)]
>>> E7=WeylCharacterRing("E7",style='coroots')
>>> A3xA3xA1=WeylCharacterRing("A3xA3xA1",style='coroots')
>>> E7(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0)).branch(A3xA3xA1,rule='extended') # long time (0.7s)
 A3xA3xA1(0,0,1,0,0,1,1) + A3xA3xA1(0,1,0,0,1,0,0) + A3xA3xA1(1,0,0,1,0,0,1) +
  A3xA3xA1(1,0,1,0,0,0,0) + A3xA3xA1(0,0,0,1,0,1,0) + A3xA3xA1(0,0,0,0,0,0,2)
>>> fw = E7.fundamental_weights()
>>> [E7(fw[i]).branch(D6,rule='levi') for i in [Integer(1),Integer(2),
↪Integer(7)]] # long time (0.3s)
[3*D6(0,0,0,0,0,0) + 2*D6(0,0,0,0,1,0) + D6(0,1,0,0,0,0),
 3*D6(0,0,0,0,0,1) + 2*D6(1,0,0,0,0,0) + 2*D6(0,0,1,0,0,0) + D6(1,0,0,0,1,0),
 D6(0,0,0,0,0,1) + 2*D6(1,0,0,0,0,0)]
```

(continues on next page)

(continued from previous page)

```
>>> D7=WeylCharacterRing("D7",style='coroots')
>>> E8=WeylCharacterRing("E8",style='coroots')
>>> D7=WeylCharacterRing("D7",style='coroots')
>>> E8(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0)).branch(D7,rule='levi') # long time (7s)
 3*D7(0,0,0,0,0,0,0) + 2*D7(0,0,0,0,0,1,0) + 2*D7(0,0,0,0,0,0,1) + 2*D7(1,0,0,0,0,
↪0,0)
 + D7(0,1,0,0,0,0,0) + 2*D7(0,0,1,0,0,0,0) + D7(0,0,0,1,0,0,0) + D7(1,0,0,0,0,1,
↪0) + D7(1,0,0,0,0,0,1) + D7(2,0,0,0,0,0,0)
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(D7,rule='levi') # long time (0.6s)
 D7(0,0,0,0,0,0,0) + D7(0,0,0,0,0,1,0) + D7(0,0,0,0,0,0,1) + 2*D7(1,0,0,0,0,0,0)␣
↪+ D7(0,1,0,0,0,0,0)
>>> [F4(fw).branch(B3,rule='levi') for fw in F4.fundamental_weights()] # long␣
↪time (1s)
 [B3(0,0,0) + 2*B3(1/2,1/2,1/2) + 2*B3(1,0,0) + B3(1,1,0),
 B3(0,0,0) + 6*B3(1/2,1/2,1/2) + 5*B3(1,0,0) + 7*B3(1,1,0) + 3*B3(1,1,1)
 + 6*B3(3/2,1/2,1/2) + 2*B3(3/2,3/2,1/2) + B3(2,0,0) + 2*B3(2,1,0) + B3(2,1,1),
 3*B3(0,0,0) + 6*B3(1/2,1/2,1/2) + 4*B3(1,0,0) + 3*B3(1,1,0) + B3(1,1,1) + 2*B3(3/
↪2,1/2,1/2),
 3*B3(0,0,0) + 2*B3(1/2,1/2,1/2) + B3(1,0,0)]
>>> [F4(fw).branch(C3,rule='levi') for fw in F4.fundamental_weights()] # long␣
↪time (1s)
 [3*C3(0,0,0) + 2*C3(1,1,1) + C3(2,0,0),
 3*C3(0,0,0) + 6*C3(1,1,1) + 4*C3(2,0,0) + 2*C3(2,1,0) + 3*C3(2,2,0) + C3(2,2,2)␣
↪+ C3(3,1,0) + 2*C3(3,1,1),
 2*C3(1,0,0) + 3*C3(1,1,0) + C3(2,0,0) + 2*C3(2,1,0) + C3(2,1,1),
 2*C3(1,0,0) + C3(1,1,0)]
>>> A1xA1 = WeylCharacterRing("A1xA1")
>>> [A3(hwv).branch(A1xA1,rule='levi') for hwv in A3.fundamental_weights()]
[A1xA1(1,0,0,0) + A1xA1(0,0,1,0),
A1xA1(1,1,0,0) + A1xA1(1,0,1,0) + A1xA1(0,0,1,1),
A1xA1(1,1,1,0) + A1xA1(1,0,1,1)]
>>> A1xB1=WeylCharacterRing("A1xB1",style='coroots')
>>> [B3(x).branch(A1xB1,rule='levi') for x in B3.fundamental_weights()]
[2*A1xB1(1,0) + A1xB1(0,2),
3*A1xB1(0,0) + 2*A1xB1(1,2) + A1xB1(2,0) + A1xB1(0,2),
A1xB1(1,1) + 2*A1xB1(0,1)]
```

Automorphic Type

If the Dynkin diagram has a symmetry, then there is an automorphism that is a special case of a branching rule. There is also an exotic "triality" automorphism of D_4 having order 3. Use `rule="automorphic"` (or for D_4 `rule="triality"`):

$$A_r \Rightarrow A_r$$
$$D_r \Rightarrow D_r$$
$$E_6 \Rightarrow E_6$$

EXAMPLES:

```
sage: [A3(chi).branch(A3,rule='automorphic') for chi in A3.fundamental_weights()]
[A3(0,0,0,-1), A3(0,0,-1,-1), A3(0,-1,-1,-1)]
```

(continues on next page)

(continued from previous page)

```
sage: [D4(chi).branch(D4,rule='automorphic') for chi in D4.fundamental_weights()]
[D4(1,0,0,0), D4(1,1,0,0), D4(1/2,1/2,1/2,1/2), D4(1/2,1/2,1/2,-1/2)]
```

```
>>> from sage.all import *
>>> [A3(chi).branch(A3,rule='automorphic') for chi in A3.fundamental_weights()]
[A3(0,0,0,-1), A3(0,0,-1,-1), A3(0,-1,-1,-1)]
>>> [D4(chi).branch(D4,rule='automorphic') for chi in D4.fundamental_weights()]
[D4(1,0,0,0), D4(1,1,0,0), D4(1/2,1/2,1/2,1/2), D4(1/2,1/2,1/2,-1/2)]
```

Here is an example with D_4 triality:

```
sage: [D4(chi).branch(D4,rule='triality') for chi in D4.fundamental_weights()]
[D4(1/2,1/2,1/2,-1/2), D4(1,1,0,0), D4(1/2,1/2,1/2,1/2), D4(1,0,0,0)]
```

```
>>> from sage.all import *
>>> [D4(chi).branch(D4,rule='triality') for chi in D4.fundamental_weights()]
[D4(1/2,1/2,1/2,-1/2), D4(1,1,0,0), D4(1/2,1/2,1/2,1/2), D4(1,0,0,0)]
```

Symmetric Type

Related to the automorphic type, when G admits an outer automorphism (usually of degree 2) we may consider the branching rule to the isotropy subgroup H. Outer automorphisms correspond to symmetries of the Dynkin diagram. For such isotropy subgroups use `rule="symmetric"`. We may thus obtain the following branching rules.

$$\begin{aligned} A_{2r} &\Rightarrow B_r \\ A_{2r-1} &\Rightarrow C_r \\ A_{2r-1} &\Rightarrow D_r \\ D_r &\Rightarrow B_{r-1} \\ E_6 &\Rightarrow F_4 \\ E_6 &\Rightarrow C_4 \\ D_4 &\Rightarrow G_2 \end{aligned}$$

The last branching rule, $D_4 \Rightarrow G_2$ is not to a maximal subgroup since $D_4 \Rightarrow B_3 \Rightarrow G_2$, but it is included for convenience.

In some cases, two outer automorphisms that differ by an inner automorphism may have different fixed subgroups. Thus, while the Dynkin diagram of E_6 has a single involutory automorphism, there are two involutions of the group (differing by an inner automorphism) with fixed subgroups F_4 and C_4. Similarly $SL(2r)$, of Cartan type A_{2r-1}, has subgroups $SO(2r)$ and $Sp(2r)$, both fixed subgroups of outer automorphisms that differ from each other by an inner automorphism.

In many cases the Dynkin diagram of H can be obtained by folding the Dynkin diagram of G.

EXAMPLES:

```
sage: [w.branch(B2,rule='symmetric') for w in [A4(1,0,0,0,0),A4(1,1,0,0,0),A4(1,1,
↪1,0,0),A4(2,0,0,0,0)]]
[B2(1,0), B2(1,1), B2(1,1), B2(0,0) + B2(2,0)]
sage: [A5(w).branch(C3,rule='symmetric') for w in A5.fundamental_weights()]
[C3(1,0,0), C3(0,0,0) + C3(1,1,0), C3(1,0,0) + C3(1,1,1), C3(0,0,0) + C3(1,1,0),␣
↪C3(1,0,0)]
```

(continues on next page)

(continued from previous page)

```
sage: [A5(w).branch(D3,rule='symmetric') for w in A5.fundamental_weights()]
[D3(1,0,0), D3(1,1,0), D3(1,1,-1) + D3(1,1,1), D3(1,1,0), D3(1,0,0)]
sage: [D4(x).branch(B3,rule='symmetric') for x in D4.fundamental_weights()]
[B3(0,0,0) + B3(1,0,0), B3(1,0,0) + B3(1,1,0), B3(1/2,1/2,1/2), B3(1/2,1/2,1/2)]
sage: [D4(x).branch(G2,rule='symmetric') for x in D4.fundamental_weights()]
[G2(0,0,0) + G2(1,0,-1), 2*G2(1,0,-1) + G2(2,-1,-1), G2(0,0,0) + G2(1,0,-1), G2(0,
↪0,0) + G2(1,0,-1)]
sage: [E6(fw).branch(F4,rule='symmetric') for fw in E6.fundamental_weights()] #␣
↪long time (4s)
[F4(0,0,0,0) + F4(0,0,0,1),
 F4(0,0,0,1) + F4(1,0,0,0),
 F4(0,0,0,1) + F4(1,0,0,0) + F4(0,0,1,0),
 F4(1,0,0,0) + 2*F4(0,0,1,0) + F4(1,0,0,1) + F4(0,1,0,0),
 F4(0,0,0,1) + F4(1,0,0,0) + F4(0,0,1,0),
 F4(0,0,0,0) + F4(0,0,0,1)]
sage: E6=WeylCharacterRing("E6",style='coroots')
sage: C4=WeylCharacterRing("C4",style='coroots')
sage: chi = E6(1,0,0,0,0,0); chi.degree()
27
sage: chi.branch(C4,rule='symmetric')
C4(0,1,0,0)
```

```
>>> from sage.all import *
>>> [w.branch(B2,rule='symmetric') for w in [A4(Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0)),A4(Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0)),A4(Integer(1),Integer(1),Integer(1),Integer(0),Integer(0)),
↪A4(Integer(2),Integer(0),Integer(0),Integer(0),Integer(0))]]
[B2(1,0), B2(1,1), B2(1,1), B2(0,0) + B2(2,0)]
>>> [A5(w).branch(C3,rule='symmetric') for w in A5.fundamental_weights()]
[C3(1,0,0), C3(0,0,0) + C3(1,1,0), C3(1,0,0) + C3(1,1,1), C3(0,0,0) + C3(1,1,0),␣
↪C3(1,0,0)]
>>> [A5(w).branch(D3,rule='symmetric') for w in A5.fundamental_weights()]
[D3(1,0,0), D3(1,1,0), D3(1,1,-1) + D3(1,1,1), D3(1,1,0), D3(1,0,0)]
>>> [D4(x).branch(B3,rule='symmetric') for x in D4.fundamental_weights()]
[B3(0,0,0) + B3(1,0,0), B3(1,0,0) + B3(1,1,0), B3(1/2,1/2,1/2), B3(1/2,1/2,1/2)]
>>> [D4(x).branch(G2,rule='symmetric') for x in D4.fundamental_weights()]
[G2(0,0,0) + G2(1,0,-1), 2*G2(1,0,-1) + G2(2,-1,-1), G2(0,0,0) + G2(1,0,-1), G2(0,
↪0,0) + G2(1,0,-1)]
>>> [E6(fw).branch(F4,rule='symmetric') for fw in E6.fundamental_weights()] #␣
↪long time (4s)
[F4(0,0,0,0) + F4(0,0,0,1),
 F4(0,0,0,1) + F4(1,0,0,0),
 F4(0,0,0,1) + F4(1,0,0,0) + F4(0,0,1,0),
 F4(1,0,0,0) + 2*F4(0,0,1,0) + F4(1,0,0,1) + F4(0,1,0,0),
 F4(0,0,0,1) + F4(1,0,0,0) + F4(0,0,1,0),
 F4(0,0,0,0) + F4(0,0,0,1)]
>>> E6=WeylCharacterRing("E6",style='coroots')
>>> C4=WeylCharacterRing("C4",style='coroots')
>>> chi = E6(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0));␣
↪chi.degree()
27
>>> chi.branch(C4,rule='symmetric')
```

(continues on next page)

(continued from previous page)

```
C4(0,1,0,0)
```

Extended Type

If removing a node from the extended Dynkin diagram results in a Dynkin diagram, then there is a branching rule. Use `rule="extended"` for these. We will also use this classification for some rules that are not of this type, mainly involving type B, such as $D_6 \Rightarrow B_3 \times B_3$.

Here is the extended Dynkin diagram for D_6:

```
    0       6
    O       O
    |       |
    |       |
O---O---O---O---O
1   2   3   4   6
```

Removing the node 3 results in an embedding $D_3 \times D_3 \Rightarrow D_6$. This corresponds to the embedding $SO(6) \times SO(6) \Rightarrow SO(12)$, and is of extended type. On the other hand the embedding $SO(5) \times SO(7) \Rightarrow SO(12)$ (e.g. $B_2 \times B_3 \Rightarrow D_6$) cannot be explained this way but for uniformity is implemented under `rule="extended"`.

The following rules are implemented as special cases of `rule="extended"`:

$$\begin{aligned}
E_6 &\Rightarrow A_5 \times A_1, A_2 \times A_2 \times A_2 \\
E_7 &\Rightarrow A_7, D_6 \times A_1, A_3 \times A_3 \times A_1 \\
E_8 &\Rightarrow A_8, D_8, E_7 \times A_1, A_4 \times A_4, D_5 \times A_3, E_6 \times A_2 \\
F_4 &\Rightarrow B_4, C_3 \times A_1, A_2 \times A_2, A_3 \times A_1 \\
G_2 &\Rightarrow A_1 \times A_1
\end{aligned}$$

Note that E_8 has only a limited number of representations of reasonably low degree.

EXAMPLES:

```
sage: [B3(x).branch(D3,rule='extended') for x in B3.fundamental_weights()]
[D3(0,0,0) + D3(1,0,0),
 D3(1,0,0) + D3(1,1,0),
 D3(1/2,1/2,-1/2) + D3(1/2,1/2,1/2)]
sage: [G2(w).branch(A2, rule='extended') for w in G2.fundamental_weights()]
[A2(0,0,0) + A2(1/3,1/3,-2/3) + A2(2/3,-1/3,-1/3),
 A2(1/3,1/3,-2/3) + A2(2/3,-1/3,-1/3) + A2(1,0,-1)]
sage: [F4(fw).branch(B4,rule='extended') for fw in F4.fundamental_weights()] #␣
↪long time (2s)
[B4(1/2,1/2,1/2,1/2) + B4(1,1,0,0),
 B4(1,1,0,0) + B4(1,1,1,0) + B4(3/2,1/2,1/2,1/2) + B4(3/2,3/2,1/2,1/2) + B4(2,1,1,
↪0),
 B4(1/2,1/2,1/2,1/2) + B4(1,0,0,0) + B4(1,1,0,0) + B4(1,1,1,0) + B4(3/2,1/2,1/2,1/
↪2),
 B4(0,0,0,0) + B4(1/2,1/2,1/2,1/2) + B4(1,0,0,0)]

sage: E6 = WeylCharacterRing("E6", style='coroots')
sage: A2xA2xA2 = WeylCharacterRing("A2xA2xA2",style='coroots')
sage: A5xA1 = WeylCharacterRing("A5xA1",style='coroots')
sage: G2 = WeylCharacterRing("G2", style='coroots')
```

(continues on next page)

(continued from previous page)

```
sage: A1xA1 = WeylCharacterRing("A1xA1", style='coroots')
sage: F4 = WeylCharacterRing("F4",style='coroots')
sage: A3xA1 = WeylCharacterRing("A3xA1", style='coroots')
sage: A2xA2 = WeylCharacterRing("A2xA2", style='coroots')
sage: A1xC3 = WeylCharacterRing("A1xC3",style='coroots')
sage: E6(1,0,0,0,0,0).branch(A5xA1,rule='extended') # (0.7s)
 A5xA1(0,0,0,1,0,0) + A5xA1(1,0,0,0,0,1)
sage: E6(1,0,0,0,0,0).branch(A2xA2xA2, rule='extended') # (0.7s)
A2xA2xA2(0,1,1,0,0,0) + A2xA2xA2(1,0,0,0,0,1) + A2xA2xA2(0,0,0,1,1,0)
sage: E7 = WeylCharacterRing("E7",style='coroots')
sage: A7 = WeylCharacterRing("A7",style='coroots')
sage: E7(1,0,0,0,0,0,0).branch(A7,rule='extended')
 A7(0,0,0,1,0,0,0) + A7(1,0,0,0,0,0,1)
sage: D6xA1 = WeylCharacterRing("D6xA1",style='coroots')
sage: E7(1,0,0,0,0,0,0).branch(D6xA1,rule='extended')
 D6xA1(0,0,0,0,1,0,1) + D6xA1(0,1,0,0,0,0,0) + D6xA1(0,0,0,0,0,0,2)
sage: A5xA2 = WeylCharacterRing("A5xA2",style='coroots')
sage: E7(1,0,0,0,0,0,0).branch(A5xA2,rule='extended')
A5xA2(0,0,0,1,0,1,0) + A5xA2(0,1,0,0,0,0,1) + A5xA2(1,0,0,0,1,0,0) + A5xA2(0,0,0,
↪0,0,1,1)
sage: E8 = WeylCharacterRing("E8",style='coroots')
sage: D8 = WeylCharacterRing("D8",style='coroots')
sage: A8 = WeylCharacterRing("A8",style='coroots')
sage: E8(0,0,0,0,0,0,0,1).branch(D8,rule='extended') # long time (0.56s)
 D8(0,0,0,0,0,0,1,0) + D8(0,1,0,0,0,0,0,0)
sage: E8(0,0,0,0,0,0,0,1).branch(A8,rule='extended') # long time (0.73s)
A8(0,0,0,0,0,1,0,0) + A8(0,0,1,0,0,0,0,0) + A8(1,0,0,0,0,0,0,1)
sage: F4(1,0,0,0).branch(A1xC3,rule='extended') # (0.05s)
 A1xC3(1,0,0,1) + A1xC3(2,0,0,0) + A1xC3(0,2,0,0)
sage: G2(0,1).branch(A1xA1, rule='extended')
 A1xA1(2,0) + A1xA1(3,1) + A1xA1(0,2)
sage: F4(0,0,0,1).branch(A2xA2, rule='extended') # (0.4s)
 A2xA2(0,1,0,1) + A2xA2(1,0,1,0) + A2xA2(0,0,1,1)
sage: F4(0,0,0,1).branch(A3xA1,rule='extended') # (0.34s)
 A3xA1(0,0,0,0) + A3xA1(0,0,1,1) + A3xA1(0,1,0,0) + A3xA1(1,0,0,1) + A3xA1(0,0,0,
↪2)
sage: D4=WeylCharacterRing("D4",style='coroots')
sage: D2xD2=WeylCharacterRing("D2xD2",style='coroots') # We get D4 => A1xA1xA1xA1␣
↪by remembering that A1xA1 = D2.
sage: [D4(fw).branch(D2xD2, rule='extended') for fw in D4.fundamental_weights()]
 [D2xD2(1,1,0,0) + D2xD2(0,0,1,1),
 D2xD2(2,0,0,0) + D2xD2(0,2,0,0) + D2xD2(1,1,1,1) + D2xD2(0,0,2,0) + D2xD2(0,0,0,
↪2),
 D2xD2(1,0,0,1) + D2xD2(0,1,1,0),
 D2xD2(1,0,1,0) + D2xD2(0,1,0,1)]
```

```
>>> from sage.all import *
>>> [B3(x).branch(D3,rule='extended') for x in B3.fundamental_weights()]
[D3(0,0,0) + D3(1,0,0),
 D3(1,0,0) + D3(1,1,0),
 D3(1/2,1/2,-1/2) + D3(1/2,1/2,1/2)]
>>> [G2(w).branch(A2, rule='extended') for w in G2.fundamental_weights()]
```

(continues on next page)

(continued from previous page)

```
[A2(0,0,0) + A2(1/3,1/3,-2/3) + A2(2/3,-1/3,-1/3),
 A2(1/3,1/3,-2/3) + A2(2/3,-1/3,-1/3) + A2(1,0,-1)]
>>> [F4(fw).branch(B4,rule='extended') for fw in F4.fundamental_weights()] # long␣
↪time (2s)
[B4(1/2,1/2,1/2,1/2) + B4(1,1,0,0),
 B4(1,1,0,0) + B4(1,1,1,0) + B4(3/2,1/2,1/2,1/2) + B4(3/2,3/2,1/2,1/2) + B4(2,1,1,
↪0),
 B4(1/2,1/2,1/2,1/2) + B4(1,0,0,0) + B4(1,1,0,0) + B4(1,1,1,0) + B4(3/2,1/2,1/2,1/
↪2),
 B4(0,0,0,0) + B4(1/2,1/2,1/2,1/2) + B4(1,0,0,0)]

>>> E6 = WeylCharacterRing("E6", style='coroots')
>>> A2xA2xA2 = WeylCharacterRing("A2xA2xA2",style='coroots')
>>> A5xA1 = WeylCharacterRing("A5xA1",style='coroots')
>>> G2 = WeylCharacterRing("G2", style='coroots')
>>> A1xA1 = WeylCharacterRing("A1xA1", style='coroots')
>>> F4 = WeylCharacterRing("F4",style='coroots')
>>> A3xA1 = WeylCharacterRing("A3xA1", style='coroots')
>>> A2xA2 = WeylCharacterRing("A2xA2", style='coroots')
>>> A1xC3 = WeylCharacterRing("A1xC3",style='coroots')
>>> E6(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)).
↪branch(A5xA1,rule='extended') # (0.7s)
 A5xA1(0,0,0,1,0,0) + A5xA1(1,0,0,0,0,1)
>>> E6(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)).
↪branch(A2xA2xA2, rule='extended') # (0.7s)
A2xA2xA2(0,1,1,0,0,0) + A2xA2xA2(1,0,0,0,0,1) + A2xA2xA2(0,0,0,1,1,0)
>>> E7 = WeylCharacterRing("E7",style='coroots')
>>> A7 = WeylCharacterRing("A7",style='coroots')
>>> E7(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0)).branch(A7,rule='extended')
 A7(0,0,0,1,0,0,0) + A7(1,0,0,0,0,0,1)
>>> D6xA1 = WeylCharacterRing("D6xA1",style='coroots')
>>> E7(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0)).branch(D6xA1,rule='extended')
 D6xA1(0,0,0,0,1,0,1) + D6xA1(0,1,0,0,0,0,0) + D6xA1(0,0,0,0,0,0,2)
>>> A5xA2 = WeylCharacterRing("A5xA2",style='coroots')
>>> E7(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0)).branch(A5xA2,rule='extended')
A5xA2(0,0,0,1,0,1,0) + A5xA2(0,1,0,0,0,0,1) + A5xA2(1,0,0,0,1,0,0) + A5xA2(0,0,0,
↪0,0,1,1)
>>> E8 = WeylCharacterRing("E8",style='coroots')
>>> D8 = WeylCharacterRing("D8",style='coroots')
>>> A8 = WeylCharacterRing("A8",style='coroots')
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(D8,rule='extended') # long time (0.56s)
 D8(0,0,0,0,0,0,1,0) + D8(0,1,0,0,0,0,0,0)
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(A8,rule='extended') # long time (0.73s)
A8(0,0,0,0,0,1,0,0) + A8(0,0,1,0,0,0,0,0) + A8(1,0,0,0,0,0,0,1)
>>> F4(Integer(1),Integer(0),Integer(0),Integer(0)).branch(A1xC3,rule='extended')
↪# (0.05s)
 A1xC3(1,0,0,1) + A1xC3(2,0,0,0) + A1xC3(0,2,0,0)
```

(continues on next page)

(continued from previous page)

```
>>> G2(Integer(0),Integer(1)).branch(A1xA1, rule='extended')
A1xA1(2,0) + A1xA1(3,1) + A1xA1(0,2)
>>> F4(Integer(0),Integer(0),Integer(0),Integer(1)).branch(A2xA2, rule='extended
↪') # (0.4s)
A2xA2(0,1,0,1) + A2xA2(1,0,1,0) + A2xA2(0,0,1,1)
>>> F4(Integer(0),Integer(0),Integer(0),Integer(1)).branch(A3xA1,rule='extended')
↪# (0.34s)
A3xA1(0,0,0,0) + A3xA1(0,0,1,1) + A3xA1(0,1,0,0) + A3xA1(1,0,0,1) + A3xA1(0,0,0,
↪2)
>>> D4=WeylCharacterRing("D4",style='coroots')
>>> D2xD2=WeylCharacterRing("D2xD2",style='coroots') # We get D4 => A1xA1xA1xA1␣
↪by remembering that A1xA1 = D2.
>>> [D4(fw).branch(D2xD2, rule='extended') for fw in D4.fundamental_weights()]
[D2xD2(1,1,0,0) + D2xD2(0,0,1,1),
D2xD2(2,0,0,0) + D2xD2(0,2,0,0) + D2xD2(1,1,1,1) + D2xD2(0,0,2,0) + D2xD2(0,0,0,
↪2),
D2xD2(1,0,0,1) + D2xD2(0,1,1,0),
D2xD2(1,0,1,0) + D2xD2(0,1,0,1)]
```

Orthogonal Sum

Using `rule="orthogonal_sum"`, for $n = a + b + c + \cdots$, you can get any branching rule

$$\begin{aligned} SO(n) &\Rightarrow SO(a) \times SO(b) \times SO(c) \times \cdots, \\ Sp(2n) &\Rightarrow Sp(2a) \times Sp(2b) \times Sp(2c)x \times \cdots, \end{aligned}$$

where $O(a)$ is type D_r for $a = 2r$ or B_r for $a = 2r + 1$ and $Sp(2r)$ is type C_r. In some cases these are also of extended type, as in the case $D_3 \times D_3 \Rightarrow D_6$ discussed above. But in other cases, for example $B_3 \times B_3 \Rightarrow D_7$, they are not of extended type.

Tensor

There are branching rules:

$$\begin{aligned} A_{rs-1} &\Rightarrow A_{r-1} \times A_{s-1}, \\ B_{2rs+r+s} &\Rightarrow B_r \times B_s, \\ D_{2rs+s} &\Rightarrow B_r \times D_s, \\ D_{2rs} &\Rightarrow D_r \times D_s, \\ D_{2rs} &\Rightarrow C_r \times C_s, \\ C_{2rs+s} &\Rightarrow B_r \times C_s, \\ C_{2rs} &\Rightarrow C_r \times D_s. \end{aligned}$$

corresponding to the tensor product homomorphism. For type A, the homomorphism is $GL(r) \times GL(s) \Rightarrow GL(rs)$. For the classical types, the relevant fact is that if V, W are orthogonal or symplectic spaces, that is, spaces endowed with symmetric or skew-symmetric bilinear forms, then $V \otimes W$ is also an orthogonal space (if V and W are both orthogonal or both symplectic) or symplectic (if one of V and W is orthogonal and the other symplectic).

The corresponding branching rules are obtained using `rule="tensor"`.

EXAMPLES:

```
sage: A5=WeylCharacterRing("A5", style='coroots')
sage: A2xA1=WeylCharacterRing("A2xA1", style='coroots')
sage: [A5(hwv).branch(A2xA1, rule='tensor') for hwv in A5.fundamental_weights()]
[A2xA1(1,0,1),
A2xA1(0,1,2) + A2xA1(2,0,0),
A2xA1(1,1,1) + A2xA1(0,0,3),
A2xA1(1,0,2) + A2xA1(0,2,0),
A2xA1(0,1,1)]
sage: B4=WeylCharacterRing("B4",style='coroots')
sage: B1xB1=WeylCharacterRing("B1xB1",style='coroots')
sage: [B4(f).branch(B1xB1,rule='tensor') for f in B4.fundamental_weights()]
[B1xB1(2,2),
B1xB1(2,0) + B1xB1(2,4) + B1xB1(4,2) + B1xB1(0,2),
B1xB1(2,0) + B1xB1(2,2) + B1xB1(2,4) + B1xB1(4,2) + B1xB1(4,4) + B1xB1(6,0) +
↪B1xB1(0,2) + B1xB1(0,6),
B1xB1(1,3) + B1xB1(3,1)]
sage: D4=WeylCharacterRing("D4",style='coroots')
sage: C2xC1=WeylCharacterRing("C2xC1",style='coroots')
sage: [D4(f).branch(C2xC1,rule='tensor') for f in D4.fundamental_weights()]
[C2xC1(1,0,1),
C2xC1(0,1,2) + C2xC1(2,0,0) + C2xC1(0,0,2),
C2xC1(1,0,1),
C2xC1(0,1,0) + C2xC1(0,0,2)]
sage: C3=WeylCharacterRing("C3",style='coroots')
sage: B1xC1=WeylCharacterRing("B1xC1",style='coroots')
sage: [C3(f).branch(B1xC1,rule='tensor') for f in C3.fundamental_weights()]
[B1xC1(2,1), B1xC1(2,2) + B1xC1(4,0), B1xC1(4,1) + B1xC1(0,3)]
```

```
>>> from sage.all import *
>>> A5=WeylCharacterRing("A5", style='coroots')
>>> A2xA1=WeylCharacterRing("A2xA1", style='coroots')
>>> [A5(hwv).branch(A2xA1, rule='tensor') for hwv in A5.fundamental_weights()]
[A2xA1(1,0,1),
A2xA1(0,1,2) + A2xA1(2,0,0),
A2xA1(1,1,1) + A2xA1(0,0,3),
A2xA1(1,0,2) + A2xA1(0,2,0),
A2xA1(0,1,1)]
>>> B4=WeylCharacterRing("B4",style='coroots')
>>> B1xB1=WeylCharacterRing("B1xB1",style='coroots')
>>> [B4(f).branch(B1xB1,rule='tensor') for f in B4.fundamental_weights()]
[B1xB1(2,2),
B1xB1(2,0) + B1xB1(2,4) + B1xB1(4,2) + B1xB1(0,2),
B1xB1(2,0) + B1xB1(2,2) + B1xB1(2,4) + B1xB1(4,2) + B1xB1(4,4) + B1xB1(6,0) +
↪B1xB1(0,2) + B1xB1(0,6),
B1xB1(1,3) + B1xB1(3,1)]
>>> D4=WeylCharacterRing("D4",style='coroots')
>>> C2xC1=WeylCharacterRing("C2xC1",style='coroots')
>>> [D4(f).branch(C2xC1,rule='tensor') for f in D4.fundamental_weights()]
[C2xC1(1,0,1),
C2xC1(0,1,2) + C2xC1(2,0,0) + C2xC1(0,0,2),
C2xC1(1,0,1),
C2xC1(0,1,0) + C2xC1(0,0,2)]
```

(continues on next page)

(continued from previous page)

```
>>> C3=WeylCharacterRing("C3",style='coroots')
>>> B1xC1=WeylCharacterRing("B1xC1",style='coroots')
>>> [C3(f).branch(B1xC1,rule='tensor') for f in C3.fundamental_weights()]
[B1xC1(2,1), B1xC1(2,2) + B1xC1(4,0), B1xC1(4,1) + B1xC1(0,3)]
```

Symmetric Power

The k-th symmetric and exterior power homomorphisms map

$$GL(n) \Rightarrow GL\left(\binom{n+k-1}{k}\right) \times GL\left(\binom{n}{k}\right).$$

The corresponding branching rules are not implemented but a special case is. The k-th symmetric power homomorphism $SL(2) \Rightarrow GL(k+1)$ has its image inside of $SO(2r+1)$ if $k = 2r$ and inside of $Sp(2r)$ if $k = 2r-1$. Hence there are branching rules:

$$B_r \Rightarrow A_1$$
$$C_r \Rightarrow A_1$$

and these may be obtained using the rule "symmetric_power".

EXAMPLES:

```
sage: A1=WeylCharacterRing("A1",style='coroots')
sage: B3=WeylCharacterRing("B3",style='coroots')
sage: C3=WeylCharacterRing("C3",style='coroots')
sage: [B3(fw).branch(A1,rule='symmetric_power') for fw in B3.fundamental_
↪weights()]
[A1(6), A1(2) + A1(6) + A1(10), A1(0) + A1(6)]
sage: [C3(fw).branch(A1,rule='symmetric_power') for fw in C3.fundamental_
↪weights()]
[A1(5), A1(4) + A1(8), A1(3) + A1(9)]
```

```
>>> from sage.all import *
>>> A1=WeylCharacterRing("A1",style='coroots')
>>> B3=WeylCharacterRing("B3",style='coroots')
>>> C3=WeylCharacterRing("C3",style='coroots')
>>> [B3(fw).branch(A1,rule='symmetric_power') for fw in B3.fundamental_weights()]
[A1(6), A1(2) + A1(6) + A1(10), A1(0) + A1(6)]
>>> [C3(fw).branch(A1,rule='symmetric_power') for fw in C3.fundamental_weights()]
[A1(5), A1(4) + A1(8), A1(3) + A1(9)]
```

Miscellaneous

Use `rule="miscellaneous"` for the following embeddings of maximal subgroups, all involving exceptional groups.

$$\begin{aligned}
B_3 &\Rightarrow G_2,\\
E_6 &\Rightarrow G_2,\\
E_6 &\Rightarrow A_2,\\
F_4 &\Rightarrow G_2 \times A_1,\\
E_6 &\Rightarrow G_2 \times A_2,\\
E_7 &\Rightarrow G_2 \times C_3,\\
E_7 &\Rightarrow F_4 \times A_1,\\
E_7 &\Rightarrow A_1 \times A_1,\\
E_7 &\Rightarrow G_2 \times A_1,\\
E_8 &\Rightarrow G_2 \times F_4.\\
E_8 &\Rightarrow A2 \times A_1.\\
E_8 &\Rightarrow B2.
\end{aligned}$$

Except for those embeddings available by `rule="extended"`, these are the only embeddings of these groups as maximal subgroups. There may be other embeddings besides these. For example, there are other more obvious embeddings of A_2 and G_2 into E_6. However the embeddings in this table are characterized as embeddings as maximal subgroups. Regarding the embeddings of A_2 and G_2 in E_6, the embeddings in question may be characterized by the condition that the 27-dimensional representations of E_6 restrict irreducibly to A_2 or G_2. Since G_2 has a subgroup isomorphic to A_2, it is worth mentioning that the composite branching rules:

```
branching_rule("E6","G2","miscellaneous")*branching_rule("G2","A2","extended")
branching_rule("E6","A2","miscellaneous")
```

are distinct.

These embeddings are described more completely (with references to the literature) in the thematic tutorial at:

https://doc.sagemath.org/html/en/thematic_tutorials/lie.html

EXAMPLES:

```
sage: G2 = WeylCharacterRing("G2")
sage: [fw1, fw2, fw3] = B3.fundamental_weights()
sage: B3(fw1+fw3).branch(G2, rule='miscellaneous')
G2(1,0,-1) + G2(2,-1,-1) + G2(2,0,-2)
sage: E6 = WeylCharacterRing("E6",style='coroots')
sage: G2 = WeylCharacterRing("G2",style='coroots')
sage: E6(1,0,0,0,0,0).branch(G2,"miscellaneous")
G2(2,0)
sage: A2=WeylCharacterRing("A2",style='coroots')
sage: E6(1,0,0,0,0,0).branch(A2,rule='miscellaneous')
A2(2,2)
sage: E6(0,1,0,0,0,0).branch(A2,rule='miscellaneous')
A2(1,1) + A2(1,4) + A2(4,1)
sage: E6(0,0,0,0,0,2).branch(G2,"miscellaneous") # long time (0.59s)
G2(0,0) + G2(2,0) + G2(1,1) + G2(0,2) + G2(4,0)
sage: F4=WeylCharacterRing("F4",style='coroots')
sage: G2xA1=WeylCharacterRing("G2xA1",style='coroots')
```

(continues on next page)

(continued from previous page)

```
sage: F4(0,0,1,0).branch(G2xA1,rule='miscellaneous')
G2xA1(1,0,0) + G2xA1(1,0,2) + G2xA1(1,0,4) + G2xA1(1,0,6) + G2xA1(0,1,4) +
↪G2xA1(2,0,2) + G2xA1(0,0,2) + G2xA1(0,0,6)
sage: E6 = WeylCharacterRing("E6",style='coroots')
sage: A2xG2 = WeylCharacterRing("A2xG2",style='coroots')
sage: E6(1,0,0,0,0,0).branch(A2xG2,rule='miscellaneous')
A2xG2(0,1,1,0) + A2xG2(2,0,0,0)
sage: E7=WeylCharacterRing("E7",style='coroots')
sage: G2xC3=WeylCharacterRing("G2xC3",style='coroots')
sage: E7(0,1,0,0,0,0,0).branch(G2xC3,rule='miscellaneous') # long time (1.84s)
G2xC3(1,0,1,0,0) + G2xC3(1,0,1,1,0) + G2xC3(0,1,0,0,1) + G2xC3(2,0,1,0,0) +
↪G2xC3(0,0,1,1,0)
sage: F4xA1=WeylCharacterRing("F4xA1",style='coroots')
sage: E7(0,0,0,0,0,0,1).branch(F4xA1,"miscellaneous")
F4xA1(0,0,0,1,1) + F4xA1(0,0,0,0,3)
sage: A1xA1=WeylCharacterRing("A1xA1",style='coroots')
sage: E7(0,0,0,0,0,0,1).branch(A1xA1,rule='miscellaneous')
A1xA1(2,5) + A1xA1(4,1) + A1xA1(6,3)
sage: A2=WeylCharacterRing("A2",style='coroots')
sage: E7(0,0,0,0,0,0,1).branch(A2,rule='miscellaneous')
A2(0,6) + A2(6,0)
sage: G2xA1=WeylCharacterRing("G2xA1",style='coroots')
sage: E7(1,0,0,0,0,0,0).branch(G2xA1,rule='miscellaneous')
G2xA1(1,0,4) + G2xA1(0,1,0) + G2xA1(2,0,2) + G2xA1(0,0,2)
sage: E8 = WeylCharacterRing("E8",style='coroots')
sage: G2xF4 = WeylCharacterRing("G2xF4",style='coroots')
sage: E8(0,0,0,0,0,0,0,1).branch(G2xF4,rule='miscellaneous') # long time (0.76s)
G2xF4(1,0,0,0,0,1) + G2xF4(0,1,0,0,0,0) + G2xF4(0,0,1,0,0,0)
sage: E8=WeylCharacterRing("E8",style='coroots')
sage: A1xA2=WeylCharacterRing("A1xA2",style='coroots')
sage: E8(0,0,0,0,0,0,0,1).branch(A1xA2,rule='miscellaneous') # long time (0.76s)
A1xA2(2,0,0) + A1xA2(2,2,2) + A1xA2(4,0,3) + A1xA2(4,3,0) + A1xA2(6,1,1) +
↪A1xA2(0,1,1)
sage: B2=WeylCharacterRing("B2",style='coroots')
sage: E8(0,0,0,0,0,0,0,1).branch(B2,rule='miscellaneous') # long time (0.53s)
B2(0,2) + B2(0,6) + B2(3,2)
```

```
>>> from sage.all import *
>>> G2 = WeylCharacterRing("G2")
>>> [fw1, fw2, fw3] = B3.fundamental_weights()
>>> B3(fw1+fw3).branch(G2, rule='miscellaneous')
G2(1,0,-1) + G2(2,-1,-1) + G2(2,0,-2)
>>> E6 = WeylCharacterRing("E6",style='coroots')
>>> G2 = WeylCharacterRing("G2",style='coroots')
>>> E6(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)).
↪branch(G2,"miscellaneous")
G2(2,0)
>>> A2=WeylCharacterRing("A2",style='coroots')
>>> E6(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)).
↪branch(A2,rule='miscellaneous')
A2(2,2)
>>> E6(Integer(0),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)).
```

(continues on next page)

(continued from previous page)

```
↪branch(A2,rule='miscellaneous')
A2(1,1) + A2(1,4) + A2(4,1)
>>> E6(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(2)).
↪branch(G2,"miscellaneous") # long time (0.59s)
G2(0,0) + G2(2,0) + G2(1,1) + G2(0,2) + G2(4,0)
>>> F4=WeylCharacterRing("F4",style='coroots')
>>> G2xA1=WeylCharacterRing("G2xA1",style='coroots')
>>> F4(Integer(0),Integer(0),Integer(1),Integer(0)).branch(G2xA1,rule=
↪'miscellaneous')
G2xA1(1,0,0) + G2xA1(1,0,2) + G2xA1(1,0,4) + G2xA1(1,0,6) + G2xA1(0,1,4) +␣
↪G2xA1(2,0,2) + G2xA1(0,0,2) + G2xA1(0,0,6)
>>> E6 = WeylCharacterRing("E6",style='coroots')
>>> A2xG2 = WeylCharacterRing("A2xG2",style='coroots')
>>> E6(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)).
↪branch(A2xG2,rule='miscellaneous')
A2xG2(0,1,1,0) + A2xG2(2,0,0,0)
>>> E7=WeylCharacterRing("E7",style='coroots')
>>> G2xC3=WeylCharacterRing("G2xC3",style='coroots')
>>> E7(Integer(0),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0)).branch(G2xC3,rule='miscellaneous') # long time (1.84s)
G2xC3(1,0,1,0,0) + G2xC3(1,0,1,1,0) + G2xC3(0,1,0,0,1) + G2xC3(2,0,1,0,0) +␣
↪G2xC3(0,0,1,1,0)
>>> F4xA1=WeylCharacterRing("F4xA1",style='coroots')
>>> E7(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1)).branch(F4xA1,"miscellaneous")
F4xA1(0,0,0,1,1) + F4xA1(0,0,0,0,3)
>>> A1xA1=WeylCharacterRing("A1xA1",style='coroots')
>>> E7(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1)).branch(A1xA1,rule='miscellaneous')
A1xA1(2,5) + A1xA1(4,1) + A1xA1(6,3)
>>> A2=WeylCharacterRing("A2",style='coroots')
>>> E7(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1)).branch(A2,rule='miscellaneous')
A2(0,6) + A2(6,0)
>>> G2xA1=WeylCharacterRing("G2xA1",style='coroots')
>>> E7(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0)).branch(G2xA1,rule='miscellaneous')
G2xA1(1,0,4) + G2xA1(0,1,0) + G2xA1(2,0,2) + G2xA1(0,0,2)
>>> E8 = WeylCharacterRing("E8",style='coroots')
>>> G2xF4 = WeylCharacterRing("G2xF4",style='coroots')
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(G2xF4,rule='miscellaneous') # long time (0.76s)
G2xF4(1,0,0,0,0,1) + G2xF4(0,1,0,0,0,0) + G2xF4(0,0,1,0,0,0)
>>> E8=WeylCharacterRing("E8",style='coroots')
>>> A1xA2=WeylCharacterRing("A1xA2",style='coroots')
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(A1xA2,rule='miscellaneous') # long time (0.76s)
A1xA2(2,0,0) + A1xA2(2,2,2) + A1xA2(4,0,3) + A1xA2(4,3,0) + A1xA2(6,1,1) +␣
↪A1xA2(0,1,1)
>>> B2=WeylCharacterRing("B2",style='coroots')
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(B2,rule='miscellaneous') # long time (0.53s)
```

(continues on next page)

(continued from previous page)

```
B2(0,2) + B2(0,6) + B2(3,2)
```

A1 maximal subgroups of exceptional groups

There are seven cases where the exceptional group G_2, F_4, E_7 or E_8 contains a maximal subgroup of type A_1. These are tabulated in Theorem 1 of Testerman, The construction of the maximal A1's in the exceptional algebraic groups, Proc. Amer. Math. Soc. 116 (1992), no. 3, 635-644. The names of these branching rules are roman numerals referring to the seven cases of her Theorem 1. Use these branching rules as in the following examples.

EXAMPLES:

```
sage: A1=WeylCharacterRing("A1",style='coroots')
sage: G2=WeylCharacterRing("G2",style='coroots')
sage: F4=WeylCharacterRing("F4",style='coroots')
sage: E7=WeylCharacterRing("E7",style='coroots')
sage: E8=WeylCharacterRing("E8",style='coroots')
sage: [G2(f).branch(A1,rule='i') for f in G2.fundamental_weights()]
[A1(6), A1(2) + A1(10)]
sage: F4(1,0,0,0).branch(A1,rule='ii')
A1(2) + A1(10) + A1(14) + A1(22)
sage: E7(0,0,0,0,0,0,1).branch(A1,rule='iii')
A1(9) + A1(17) + A1(27)
sage: E7(0,0,0,0,0,0,1).branch(A1,rule='iv')
A1(5) + A1(11) + A1(15) + A1(21)
sage: E8(0,0,0,0,0,0,0,1).branch(A1,rule='v') # long time (0.6s)
A1(2) + A1(14) + A1(22) + A1(26) + A1(34) + A1(38) + A1(46) + A1(58)
sage: E8(0,0,0,0,0,0,0,1).branch(A1,rule='vi') # long time (0.6s)
A1(2) + A1(10) + A1(14) + A1(18) + A1(22) + A1(26) + A1(28) + A1(34) + A1(38) +
↪A1(46)
sage: E8(0,0,0,0,0,0,0,1).branch(A1,rule='vii') # long time (0.6s)
A1(2) + A1(6) + A1(10) + A1(14) + A1(16) + A1(18) + 2*A1(22) + A1(26) + A1(28) +
↪A1(34) + A1(38)
```

```
>>> from sage.all import *
>>> A1=WeylCharacterRing("A1",style='coroots')
>>> G2=WeylCharacterRing("G2",style='coroots')
>>> F4=WeylCharacterRing("F4",style='coroots')
>>> E7=WeylCharacterRing("E7",style='coroots')
>>> E8=WeylCharacterRing("E8",style='coroots')
>>> [G2(f).branch(A1,rule='i') for f in G2.fundamental_weights()]
[A1(6), A1(2) + A1(10)]
>>> F4(Integer(1),Integer(0),Integer(0),Integer(0)).branch(A1,rule='ii')
A1(2) + A1(10) + A1(14) + A1(22)
>>> E7(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1)).branch(A1,rule='iii')
A1(9) + A1(17) + A1(27)
>>> E7(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(1)).branch(A1,rule='iv')
A1(5) + A1(11) + A1(15) + A1(21)
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(A1,rule='v') # long time (0.6s)
A1(2) + A1(14) + A1(22) + A1(26) + A1(34) + A1(38) + A1(46) + A1(58)
```

(continues on next page)

(continued from previous page)

```
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(A1,rule='vi') # long time (0.6s)
A1(2) + A1(10) + A1(14) + A1(18) + A1(22) + A1(26) + A1(28) + A1(34) + A1(38) +␣
↪A1(46)
>>> E8(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1)).branch(A1,rule='vii') # long time (0.6s)
A1(2) + A1(6) + A1(10) + A1(14) + A1(16) + A1(18) + 2*A1(22) + A1(26) + A1(28) +␣
↪A1(34) + A1(38)
```

Branching Rules From Plethysms

Nearly all branching rules $G \Rightarrow H$ where G is of type A, B, C or D are covered by the preceding rules. The function *branching_rule_from_plethysm()* covers the remaining cases.

This is a general rule that includes any branching rule from types A, B, C, or D as a special case. Thus it could be used in place of the above rules and would give the same results. However it is most useful when branching from G to a maximal subgroup H such that $\mathrm{rank}(H) < \mathrm{rank}(G) - 1$.

We consider a homomorphism $H \Rightarrow G$ where G is one of $SL(r+1)$, $SO(2r+1)$, $Sp(2r)$ or $SO(2r)$. The function *branching_rule_from_plethysm()* produces the corresponding branching rule. The main ingredient is the character χ of the representation of H that is the homomorphism to $GL(r+1)$, $GL(2r+1)$ or $GL(2r)$.

This rule is so powerful that it contains the other rules implemented above as special cases. First let us consider the symmetric fifth power representation of $SL(2)$.

```
sage: A1=WeylCharacterRing("A1",style='coroots')
sage: chi=A1([5])
sage: chi.degree()
 6
sage: chi.frobenius_schur_indicator()
-1
```

```
>>> from sage.all import *
>>> A1=WeylCharacterRing("A1",style='coroots')
>>> chi=A1([Integer(5)])
>>> chi.degree()
 6
>>> chi.frobenius_schur_indicator()
-1
```

This confirms that the character has degree 6 and is symplectic, so it corresponds to a homomorphism $SL(2) \Rightarrow Sp(6)$, and there is a corresponding branching rule $C_3 \Rightarrow A_1$.

```
sage: C3 = WeylCharacterRing("C3",style='coroots')
sage: sym5rule = branching_rule_from_plethysm(chi,"C3")
sage: [C3(hwv).branch(A1,rule=sym5rule) for hwv in C3.fundamental_weights()]
[A1(5), A1(4) + A1(8), A1(3) + A1(9)]
```

```
>>> from sage.all import *
>>> C3 = WeylCharacterRing("C3",style='coroots')
>>> sym5rule = branching_rule_from_plethysm(chi,"C3")
>>> [C3(hwv).branch(A1,rule=sym5rule) for hwv in C3.fundamental_weights()]
[A1(5), A1(4) + A1(8), A1(3) + A1(9)]
```

This is identical to the results we would obtain using `rule="symmetric_power"`. The next example gives a branching not available by other standard rules.

```
sage: G2 = WeylCharacterRing("G2",style='coroots')
sage: D7 = WeylCharacterRing("D7",style='coroots')
sage: ad=G2(0,1); ad.degree(); ad.frobenius_schur_indicator()
 14
 1
sage: spin = D7(0,0,0,0,0,1,0); spin.degree()
 64
sage: spin.branch(G2, rule=branching_rule_from_plethysm(ad, "D7"))
 G2(1,1)
```

```
>>> from sage.all import *
>>> G2 = WeylCharacterRing("G2",style='coroots')
>>> D7 = WeylCharacterRing("D7",style='coroots')
>>> ad=G2(Integer(0),Integer(1)); ad.degree(); ad.frobenius_schur_indicator()
 14
 1
>>> spin = D7(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(0)); spin.degree()
 64
>>> spin.branch(G2, rule=branching_rule_from_plethysm(ad, "D7"))
 G2(1,1)
```

We have confirmed that the adjoint representation of G_2 gives a homomorphism into $SO(14)$, and that the pullback of the one of the two 64 dimensional spin representations to $SO(14)$ is an irreducible representation of G_2.

We do not actually have to create the character or its parent WeylCharacterRing to create the branching rule:

```
sage: b = branching_rule("C7","C3(0,0,1)","plethysm"); b
plethysm (along C3(0,0,1)) branching rule C7 => C3
```

```
>>> from sage.all import *
>>> b = branching_rule("C7","C3(0,0,1)","plethysm"); b
plethysm (along C3(0,0,1)) branching rule C7 => C3
```

Isomorphic Type

Although not usually referred to as a branching rule, the effects of the accidental isomorphisms may be handled using `rule="isomorphic"`:

$$B_2 \Rightarrow C_2$$
$$C_2 \Rightarrow B_2$$
$$A_3 \Rightarrow D_3$$
$$D_3 \Rightarrow A_3$$
$$D_2 \Rightarrow A_1 \Rightarrow A_1$$
$$B_1 \Rightarrow A_1$$
$$C_1 \Rightarrow A_1$$

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2")
sage: C2 = WeylCharacterRing("C2")
sage: [B2(x).branch(C2, rule='isomorphic') for x in B2.fundamental_weights()]
[C2(1,1), C2(1,0)]
sage: [C2(x).branch(B2, rule='isomorphic') for x in C2.fundamental_weights()]
[B2(1/2,1/2), B2(1,0)]
sage: D3 = WeylCharacterRing("D3")
sage: A3 = WeylCharacterRing("A3")
sage: [A3(x).branch(D3,rule='isomorphic') for x in A3.fundamental_weights()]
[D3(1/2,1/2,1/2), D3(1,0,0), D3(1/2,1/2,-1/2)]
sage: [D3(x).branch(A3,rule='isomorphic') for x in D3.fundamental_weights()]
[A3(1/2,1/2,-1/2,-1/2), A3(1/4,1/4,1/4,-3/4), A3(3/4,-1/4,-1/4,-1/4)]
```

```
>>> from sage.all import *
>>> B2 = WeylCharacterRing("B2")
>>> C2 = WeylCharacterRing("C2")
>>> [B2(x).branch(C2, rule='isomorphic') for x in B2.fundamental_weights()]
[C2(1,1), C2(1,0)]
>>> [C2(x).branch(B2, rule='isomorphic') for x in C2.fundamental_weights()]
[B2(1/2,1/2), B2(1,0)]
>>> D3 = WeylCharacterRing("D3")
>>> A3 = WeylCharacterRing("A3")
>>> [A3(x).branch(D3,rule='isomorphic') for x in A3.fundamental_weights()]
[D3(1/2,1/2,1/2), D3(1,0,0), D3(1/2,1/2,-1/2)]
>>> [D3(x).branch(A3,rule='isomorphic') for x in D3.fundamental_weights()]
[A3(1/2,1/2,-1/2,-1/2), A3(1/4,1/4,1/4,-3/4), A3(3/4,-1/4,-1/4,-1/4)]
```

Here $A_3(x,y,z,w)$ can be understood as a representation of $SL(4)$. The weights x,y,z,w and $x+t,y+t,z+t,w+t$ represent the same representation of $SL(4)$ - though not of $GL(4)$ - since $A_3(x+t,y+t,z+t,w+t)$ is the same as $A_3(x,y,z,w)$ tensored with $\det^t$. So as a representation of $SL(4)$, `A3(1/4,1/4,1/4,-3/4)` is the same as `A3(1,1,1,0)`. The exterior square representation $SL(4) \Rightarrow GL(6)$ admits an invariant symmetric bilinear form, so is a representation $SL(4) \Rightarrow SO(6)$ that lifts to an isomorphism $SL(4) \Rightarrow \mathrm{Spin}(6)$. Conversely, there are two isomorphisms $SO(6) \Rightarrow SL(4)$, of which we've selected one.

In cases like this you might prefer `style="coroots"`:

```
sage: A3 = WeylCharacterRing("A3",style='coroots')
sage: D3 = WeylCharacterRing("D3",style='coroots')
sage: [D3(fw) for fw in D3.fundamental_weights()]
[D3(1,0,0), D3(0,1,0), D3(0,0,1)]
sage: [D3(fw).branch(A3,rule='isomorphic') for fw in D3.fundamental_weights()]
[A3(0,1,0), A3(0,0,1), A3(1,0,0)]
sage: D2 = WeylCharacterRing("D2", style='coroots')
sage: A1xA1 = WeylCharacterRing("A1xA1", style='coroots')
sage: [D2(fw).branch(A1xA1,rule='isomorphic') for fw in D2.fundamental_weights()]
[A1xA1(1,0), A1xA1(0,1)]
```

```
>>> from sage.all import *
>>> A3 = WeylCharacterRing("A3",style='coroots')
>>> D3 = WeylCharacterRing("D3",style='coroots')
>>> [D3(fw) for fw in D3.fundamental_weights()]
[D3(1,0,0), D3(0,1,0), D3(0,0,1)]
>>> [D3(fw).branch(A3,rule='isomorphic') for fw in D3.fundamental_weights()]
```

(continues on next page)

(continued from previous page)

```
[A3(0,1,0), A3(0,0,1), A3(1,0,0)]
>>> D2 = WeylCharacterRing("D2", style='coroots')
>>> A1xA1 = WeylCharacterRing("A1xA1", style='coroots')
>>> [D2(fw).branch(A1xA1,rule='isomorphic') for fw in D2.fundamental_weights()]
[A1xA1(1,0), A1xA1(0,1)]
```

Branching From a Reducible WeylCharacterRing

If the Cartan Type of R is reducible, we may project a character onto any of the components, or any combination of components. The rule to project on the first component is specified by the string `'proj1'`, the rule to project on the second component is `"proj2". To project on the first and third components, use ``'proj13'` and so on.

EXAMPLES:

```
sage: A2xG2=WeylCharacterRing("A2xG2",style='coroots')
sage: A2=WeylCharacterRing("A2",style='coroots')
sage: G2=WeylCharacterRing("G2",style='coroots')
sage: A2xG2(1,0,1,0).branch(A2,rule='proj1')
7*A2(1,0)
sage: A2xG2(1,0,1,0).branch(G2,rule='proj2')
3*G2(1,0)
sage: A2xA2xG2=WeylCharacterRing("A2xA2xG2",style='coroots')
sage: A2xA2xG2(0,1,1,1,0,1).branch(A2xG2,rule='proj13')
8*A2xG2(0,1,0,1)
```

```
>>> from sage.all import *
>>> A2xG2=WeylCharacterRing("A2xG2",style='coroots')
>>> A2=WeylCharacterRing("A2",style='coroots')
>>> G2=WeylCharacterRing("G2",style='coroots')
>>> A2xG2(Integer(1),Integer(0),Integer(1),Integer(0)).branch(A2,rule='proj1')
7*A2(1,0)
>>> A2xG2(Integer(1),Integer(0),Integer(1),Integer(0)).branch(G2,rule='proj2')
3*G2(1,0)
>>> A2xA2xG2=WeylCharacterRing("A2xA2xG2",style='coroots')
>>> A2xA2xG2(Integer(0),Integer(1),Integer(1),Integer(1),Integer(0),Integer(1)).
↪branch(A2xG2,rule='proj13')
8*A2xG2(0,1,0,1)
```

A more general way of specifying a branching rule from a reducible type is to supply a *list* of rules, one *component rule* for each component type in the root system. In the following example, we branch the fundamental representations of D_4 down to $A_1 \times A_1 \times A_1 \times A_1$ through the intermediate group $D_2 \times D_2$. We use multiplicative notation to compose the branching rules. There is no need to construct the intermediate WeylCharacterRing with type $D_2 \times D_2$.

EXAMPLES:

```
sage: D4 = WeylCharacterRing("D4",style='coroots')
sage: A1xA1xA1xA1 = WeylCharacterRing("A1xA1xA1xA1",style='coroots')
sage: b = branching_rule("D2","A1xA1","isomorphic")
sage: br = branching_rule("D4","D2xD2","extended")*branching_rule("D2xD2",
↪"A1xA1xA1xA1",[b,b])
sage: [D4(fw).branch(A1xA1xA1xA1,rule=br) for fw in D4.fundamental_weights()]
```

(continues on next page)

(continued from previous page)

```
[A1xA1xA1xA1(1,1,0,0) + A1xA1xA1xA1(0,0,1,1),
A1xA1xA1xA1(1,1,1,1) + A1xA1xA1xA1(2,0,0,0) + A1xA1xA1xA1(0,2,0,0) +␣
↪A1xA1xA1xA1(0,0,2,0) + A1xA1xA1xA1(0,0,0,2),
A1xA1xA1xA1(1,0,0,1) + A1xA1xA1xA1(0,1,1,0),
A1xA1xA1xA1(1,0,1,0) + A1xA1xA1xA1(0,1,0,1)]
```

```
>>> from sage.all import *
>>> D4 = WeylCharacterRing("D4",style='coroots')
>>> A1xA1xA1xA1 = WeylCharacterRing("A1xA1xA1xA1",style='coroots')
>>> b = branching_rule("D2","A1xA1","isomorphic")
>>> br = branching_rule("D4","D2xD2","extended")*branching_rule("D2xD2",
↪"A1xA1xA1xA1",[b,b])
>>> [D4(fw).branch(A1xA1xA1xA1,rule=br) for fw in D4.fundamental_weights()]
[A1xA1xA1xA1(1,1,0,0) + A1xA1xA1xA1(0,0,1,1),
A1xA1xA1xA1(1,1,1,1) + A1xA1xA1xA1(2,0,0,0) + A1xA1xA1xA1(0,2,0,0) +␣
↪A1xA1xA1xA1(0,0,2,0) + A1xA1xA1xA1(0,0,0,2),
A1xA1xA1xA1(1,0,0,1) + A1xA1xA1xA1(0,1,1,0),
A1xA1xA1xA1(1,0,1,0) + A1xA1xA1xA1(0,1,0,1)]
```

In the list of rules to be supplied in branching from a reducible root system, we may use two key words "omit" and "identity". The term "omit" means that we omit one factor, projecting onto the remaining factors. The term "identity" is supplied when the irreducible factor Cartan Types of both the target and the source are the same, and the component branching rule is to be the identity map. For example, we have projection maps from $A_3 \times A_2$ to A_3 and A_2, and the corresponding branching may be accomplished as follows. In this example the same could be accomplished using `rule="proj2"`.

EXAMPLES:

```
sage: A3xA2=WeylCharacterRing("A3xA2",style='coroots')
sage: A3=WeylCharacterRing("A3",style='coroots')
sage: chi = A3xA2(0,1,0,1,0)
sage: chi.branch(A3,rule=["identity","omit"])
3*A3(0,1,0)
sage: A2=WeylCharacterRing("A2",style='coroots')
sage: chi.branch(A2,rule=["omit","identity"])
6*A2(1,0)
```

```
>>> from sage.all import *
>>> A3xA2=WeylCharacterRing("A3xA2",style='coroots')
>>> A3=WeylCharacterRing("A3",style='coroots')
>>> chi = A3xA2(Integer(0),Integer(1),Integer(0),Integer(1),Integer(0))
>>> chi.branch(A3,rule=["identity","omit"])
3*A3(0,1,0)
>>> A2=WeylCharacterRing("A2",style='coroots')
>>> chi.branch(A2,rule=["omit","identity"])
6*A2(1,0)
```

Yet another way of branching from a reducible root system with repeated Cartan types is to embed along the diagonal. The branching rule is equivalent to the tensor product, as the example shows:

```
sage: G2=WeylCharacterRing("G2",style='coroots')
sage: G2xG2=WeylCharacterRing("G2xG2",style='coroots')
```

(continues on next page)

(continued from previous page)

```
sage: G2=WeylCharacterRing("G2",style='coroots')
sage: G2xG2(1,0,0,1).branch(G2,rule='diagonal')
G2(1,0) + G2(2,0) + G2(1,1)
sage: G2xG2(1,0,0,1).branch(G2,rule='diagonal') == G2(1,0)*G2(0,1)
True
```

```
>>> from sage.all import *
>>> G2=WeylCharacterRing("G2",style='coroots')
>>> G2xG2=WeylCharacterRing("G2xG2",style='coroots')
>>> G2=WeylCharacterRing("G2",style='coroots')
>>> G2xG2(Integer(1),Integer(0),Integer(0),Integer(1)).branch(G2,rule='diagonal')
G2(1,0) + G2(2,0) + G2(1,1)
>>> G2xG2(Integer(1),Integer(0),Integer(0),Integer(1)).branch(G2,rule='diagonal')␣
↪== G2(Integer(1),Integer(0))*G2(Integer(0),Integer(1))
True
```

Writing Your Own (Branching) Rules

Suppose you want to branch from a group G to a subgroup H. Arrange the embedding so that a Cartan subalgebra U of H is contained in a Cartan subalgebra T of G. There is thus a mapping from the weight spaces $\mathrm{Lie}(T)^* \Rightarrow \mathrm{Lie}(U)^*$. Two embeddings will produce identical branching rules if they differ by an element of the Weyl group of H.

The *rule* is this map $\mathrm{Lie}(T)^*$, which is `G.space()`, to $\mathrm{Lie}(U)^*$, which is `H.space()`, which you may implement as a function. As an example, let us consider how to implement the branching rule $A_3 \Rightarrow C_2$. Here $H = C_2 = Sp(4)$ embedded as a subgroup in $A_3 = GL(4)$. The Cartan subalgebra U consists of diagonal matrices with eigenvalues $u_1, u_2, -u_2, -u_1$. The `C2.space()` is the two dimensional vector spaces consisting of the linear functionals u_1 and u_2 on U. On the other hand $\mathrm{Lie}(T)$ is $\mathbf{R}^4$. A convenient way to see the restriction is to think of it as the adjoint of the map $(u_1, u_2) \mapsto (u_1, u_2, -u_2, -u_1)$, that is, $(x_0, x_1, x_2, x_3) \Rightarrow (x_0 - x_3, x_1 - x_2)$. Hence we may encode the rule as follows:

```
def rule(x):
    return [x[0]-x[3],x[1]-x[2]]
```

or simply:

```
rule = lambda x: [x[0]-x[3],x[1]-x[2]]
```

We may now make and use the branching rule as follows.

EXAMPLES:

```
sage: br = BranchingRule("A3", "C2", lambda x: [x[0]-x[3],x[1]-x[2]], "homemade");
↪ br
homemade branching rule A3 => C2
sage: [A3,C2]=[WeylCharacterRing(x,style='coroots') for x in ["A3","C2"]]
sage: A3(0,1,0).branch(C2,rule=br)
C2(0,0) + C2(0,1)
```

```
>>> from sage.all import *
>>> br = BranchingRule("A3", "C2", lambda x: [x[Integer(0)]-x[Integer(3)],
↪x[Integer(1)]-x[Integer(2)]], "homemade"); br
homemade branching rule A3 => C2
```

(continues on next page)

(continued from previous page)

```
>>> [A3,C2]=[WeylCharacterRing(x,style='coroots') for x in ["A3","C2"]]
>>> A3(Integer(0),Integer(1),Integer(0)).branch(C2,rule=br)
C2(0,0) + C2(0,1)
```

sage.combinat.root_system.branching_rules.**branching_rule**(*Rtype*, *Stype*, *rule='default'*)

Create a branching rule.

INPUT:

- R – the Weyl Character Ring of G
- S – the Weyl Character Ring of H
- rule – string describing the branching rule as a map from the weight space of S to the weight space of R

If the rule parameter is omitted, in some cases, a default rule is supplied. See *branch_weyl_character()*.

EXAMPLES:

```
sage: rule = branching_rule(CartanType("A3"),CartanType("C2"),"symmetric")
sage: [rule(x) for x in WeylCharacterRing("A3").fundamental_weights()]
[[1, 0], [1, 1], [1, 0]]
```

```
>>> from sage.all import *
>>> rule = branching_rule(CartanType("A3"),CartanType("C2"),"symmetric")
>>> [rule(x) for x in WeylCharacterRing("A3").fundamental_weights()]
[[1, 0], [1, 1], [1, 0]]
```

sage.combinat.root_system.branching_rules.**branching_rule_from_plethysm**(*chi*, *cartan_type*, *return_matrix=False*)

Create the branching rule of a plethysm.

INPUT:

- chi – the character of an irreducible representation π of a group G
- cartan_type – a classical Cartan type (A,`B`,`C` or D)

It is assumed that the image of the irreducible representation pi naturally has its image in the group G.

Returns a branching rule for this plethysm.

EXAMPLES:

The adjoint representation $SL(3) \to GL(8)$ factors through $SO(8)$. The branching rule in question will describe how representations of $SO(8)$ composed with this homomorphism decompose into irreducible characters of $SL(3)$:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2 = WeylCharacterRing("A2", style='coroots')
sage: ad = A2.adjoint_representation(); ad
A2(1,1)
sage: ad.degree()
8
sage: ad.frobenius_schur_indicator()
1
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> A2 = WeylCharacterRing("A2", style='coroots')
>>> ad = A2.adjoint_representation(); ad
A2(1,1)
>>> ad.degree()
8
>>> ad.frobenius_schur_indicator()
1
```

This confirms that ad has degree 8 and is orthogonal, hence factors through $SO(8)$ which is type D_4:

```
sage: br = branching_rule_from_plethysm(ad,"D4")
sage: D4 = WeylCharacterRing("D4")
sage: [D4(f).branch(A2,rule = br) for f in D4.fundamental_weights()]
[A2(1,1), A2(0,3) + A2(1,1) + A2(3,0), A2(1,1), A2(1,1)]
```

```
>>> from sage.all import *
>>> br = branching_rule_from_plethysm(ad,"D4")
>>> D4 = WeylCharacterRing("D4")
>>> [D4(f).branch(A2,rule = br) for f in D4.fundamental_weights()]
[A2(1,1), A2(0,3) + A2(1,1) + A2(3,0), A2(1,1), A2(1,1)]
```

sage.combinat.root_system.branching_rules.**get_branching_rule**(*Rtype*, *Stype*, *rule='default'*)

Create a branching rule.

INPUT:

- R – the Weyl Character Ring of G
- S – the Weyl Character Ring of H
- rule – string describing the branching rule as a map from the weight space of S to the weight space of R

If the rule parameter is omitted, in some cases, a default rule is supplied. See *branch_weyl_character()*.

EXAMPLES:

```
sage: rule = branching_rule(CartanType("A3"),CartanType("C2"),"symmetric")
sage: [rule(x) for x in WeylCharacterRing("A3").fundamental_weights()]
[[1, 0], [1, 1], [1, 0]]
```

```
>>> from sage.all import *
>>> rule = branching_rule(CartanType("A3"),CartanType("C2"),"symmetric")
>>> [rule(x) for x in WeylCharacterRing("A3").fundamental_weights()]
[[1, 0], [1, 1], [1, 0]]
```

sage.combinat.root_system.branching_rules.**maximal_subgroups**(*ct*, *mode='print_rules'*)

Given a classical Cartan type (of rank less than or equal to 8) this prints the Cartan types of maximal subgroups, with a method of obtaining the branching rule. The string to the right of the colon in the output is a command to create a branching rule.

INPUT:

- ct – a classical irreducible Cartan type

Returns a list of maximal subgroups of ct.

EXAMPLES:

```
sage: from sage.combinat.root_system.branching_rules import maximal_subgroups
sage: maximal_subgroups("D4")
B3:branching_rule("D4","B3","symmetric")
A2:branching_rule("D4","A2(1,1)","plethysm")
A1xC2:branching_rule("D4","C1xC2","tensor")*branching_rule("C1xC2","A1xC2",
↪[branching_rule("C1","A1","isomorphic"),"identity"])
A1xA1xA1xA1:branching_rule("D4","D2xD2","orthogonal_sum")*branching_rule("D2xD2",
↪"A1xA1xA1xA1",[branching_rule("D2","A1xA1","isomorphic"),branching_rule("D2",
↪"A1xA1","isomorphic")])
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.branching_rules import maximal_subgroups
>>> maximal_subgroups("D4")
B3:branching_rule("D4","B3","symmetric")
A2:branching_rule("D4","A2(1,1)","plethysm")
A1xC2:branching_rule("D4","C1xC2","tensor")*branching_rule("C1xC2","A1xC2",
↪[branching_rule("C1","A1","isomorphic"),"identity"])
A1xA1xA1xA1:branching_rule("D4","D2xD2","orthogonal_sum")*branching_rule("D2xD2",
↪"A1xA1xA1xA1",[branching_rule("D2","A1xA1","isomorphic"),branching_rule("D2",
↪"A1xA1","isomorphic")])
```

See also

`maximal_subgroups()`

5.1.225 Cartan matrices

AUTHORS:

- Travis Scrimshaw (2012-04-22): Nicolas M. Thiery moved matrix creation to *CartanType* to prepare `cartan_matrix()` for deprecation.
- Christian Stump, Travis Scrimshaw (2013-04-13): Created *CartanMatrix*.
- Ben Salisbury (2018-08-07): Added Borcherds-Cartan matrices.

class `sage.combinat.root_system.cartan_matrix.`**`CartanMatrix`**

Bases: `Matrix_integer_sparse`, *CartanType_abstract*

A (generalized) Cartan matrix.

A matrix $A = (a_{ij})_{i,j \in I}$ for some index set I is a generalized Cartan matrix if it satisfies the following properties:

- $a_{ii} = 2$ for all i,
- $a_{ij} \leq 0$ for all $i \neq j$,
- $a_{ij} = 0$ if and only if $a_{ji} = 0$ for all $i \neq j$.

Additionally some reference assume that a Cartan matrix is *symmetrizable* (see `is_symmetrizable()`). However following Kac, we do not make that assumption here.

An even, integral Borcherds–Cartan matrix is an integral matrix $A = (a_{ij})_{i,j \in I}$ for some countable index set I which satisfies the following properties:

- $a_{ii} \in \{2\} \cup 2\mathbf{Z}_{<0}$ for all i,

- $a_{ij} \leq 0$ for all $i \neq j$,
- $a_{ij} = 0$ if and only if $a_{ji} = 0$ for all $i \neq j$.

INPUT:

Can be anything which is accepted by `CartanType` or a matrix.

If given a matrix, one can also use the keyword `cartan_type` when giving a matrix to explicitly state the type. Otherwise this will try to check the input matrix against possible standard types of Cartan matrices. To disable this check, use the keyword `cartan_type_check = False`.

If one wants to initialize a Borcherds-Cartan matrix using matrix data, use the keyword `borcherds=True`. To specify the diagonal entries of corresponding to a Cartan type (a Cartan matrix is treated as matrix data), use `borcherds` with a list of the diagonal entries.

EXAMPLES:

```
sage: # needs sage.graphs
sage: CartanMatrix(['A', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
sage: CartanMatrix(['B', 6])
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0  0]
[ 0  0 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -2  2]
sage: CartanMatrix(['C', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
sage: CartanMatrix(['D', 6])
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0  0]
[ 0  0 -1  2 -1 -1]
[ 0  0  0 -1  2  0]
[ 0  0  0 -1  0  2]
sage: CartanMatrix(['E',6])
[ 2  0 -1  0  0  0]
[ 0  2  0 -1  0  0]
[-1  0  2 -1  0  0]
[ 0 -1 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -1  2]
sage: CartanMatrix(['E',7])
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2 -1  0  0  0]
[ 0 -1 -1  2 -1  0  0]
[ 0  0  0 -1  2 -1  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
sage: CartanMatrix(['E', 8])
[ 2  0 -1  0  0  0  0  0]
[ 0  2  0 -1  0  0  0  0]
[-1  0  2 -1  0  0  0  0]
[ 0 -1 -1  2 -1  0  0  0]
[ 0  0  0 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
sage: CartanMatrix(['F', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -2  2 -1]
[ 0  0 -1  2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CartanMatrix(['A', Integer(4)])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
>>> CartanMatrix(['B', Integer(6)])
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0  0]
[ 0  0 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -2  2]
>>> CartanMatrix(['C', Integer(4)])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
>>> CartanMatrix(['D', Integer(6)])
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0  0]
[ 0  0 -1  2 -1 -1]
[ 0  0  0 -1  2  0]
[ 0  0  0 -1  0  2]
>>> CartanMatrix(['E',Integer(6)])
[ 2  0 -1  0  0  0]
[ 0  2  0 -1  0  0]
[-1  0  2 -1  0  0]
[ 0 -1 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -1  2]
>>> CartanMatrix(['E',Integer(7)])
[ 2  0 -1  0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  2  0 -1  0  0  0]
[-1  0  2 -1  0  0  0]
[ 0 -1 -1  2 -1  0  0]
[ 0  0  0 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
>>> CartanMatrix(['E', Integer(8)])
[ 2  0 -1  0  0  0  0  0]
[ 0  2  0 -1  0  0  0  0]
[-1  0  2 -1  0  0  0  0]
[ 0 -1 -1  2 -1  0  0  0]
[ 0  0  0 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
>>> CartanMatrix(['F', Integer(4)])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -2  2 -1]
[ 0  0 -1  2]
```

This is different from MuPAD-Combinat, due to different node convention?

```
sage: # needs sage.graphs
sage: CartanMatrix(['G', 2])
[ 2 -3]
[-1  2]
sage: CartanMatrix(['A',1,1])
[ 2 -2]
[-2  2]
sage: CartanMatrix(['A', 3, 1])
[ 2 -1  0 -1]
[-1  2 -1  0]
[ 0 -1  2 -1]
[-1  0 -1  2]
sage: CartanMatrix(['B', 3, 1])
[ 2  0 -1  0]
[ 0  2 -1  0]
[-1 -1  2 -1]
[ 0  0 -2  2]
sage: CartanMatrix(['C', 3, 1])
[ 2 -1  0  0]
[-2  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
sage: CartanMatrix(['D', 4, 1])
[ 2  0 -1  0  0]
[ 0  2 -1  0  0]
[-1 -1  2 -1 -1]
[ 0  0 -1  2  0]
[ 0  0 -1  0  2]
sage: CartanMatrix(['E', 6, 1])
```

(continues on next page)

(continued from previous page)

```
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2  0 -1  0  0]
[ 0 -1  0  2 -1  0  0]
[ 0  0 -1 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
sage: CartanMatrix(['E', 7, 1])
[ 2 -1  0  0  0  0  0  0]
[-1  2  0 -1  0  0  0  0]
[ 0  0  2  0 -1  0  0  0]
[ 0 -1  0  2 -1  0  0  0]
[ 0  0 -1 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
sage: CartanMatrix(['E', 8, 1])
[ 2  0  0  0  0  0  0  0 -1]
[ 0  2  0 -1  0  0  0  0  0]
[ 0  0  2  0 -1  0  0  0  0]
[ 0 -1  0  2 -1  0  0  0  0]
[ 0  0 -1 -1  2 -1  0  0  0]
[ 0  0  0  0 -1  2 -1  0  0]
[ 0  0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0  0 -1  2 -1]
[-1  0  0  0  0  0  0 -1  2]
sage: CartanMatrix(['F', 4, 1])
[ 2 -1  0  0  0]
[-1  2 -1  0  0]
[ 0 -1  2 -1  0]
[ 0  0 -2  2 -1]
[ 0  0  0 -1  2]
sage: CartanMatrix(['G', 2, 1])
[ 2  0 -1]
[ 0  2 -3]
[-1 -1  2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CartanMatrix(['G', Integer(2)])
[ 2 -3]
[-1  2]
>>> CartanMatrix(['A',Integer(1),Integer(1)])
[ 2 -2]
[-2  2]
>>> CartanMatrix(['A', Integer(3), Integer(1)])
[ 2 -1  0 -1]
[-1  2 -1  0]
[ 0 -1  2 -1]
[-1  0 -1  2]
>>> CartanMatrix(['B', Integer(3), Integer(1)])
[ 2  0 -1  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  2 -1  0]
[-1 -1  2 -1]
[ 0  0 -2  2]
>>> CartanMatrix(['C', Integer(3), Integer(1)])
[ 2 -1  0  0]
[-2  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
>>> CartanMatrix(['D', Integer(4), Integer(1)])
[ 2  0 -1  0  0]
[ 0  2 -1  0  0]
[-1 -1  2 -1 -1]
[ 0  0 -1  2  0]
[ 0  0 -1  0  2]
>>> CartanMatrix(['E', Integer(6), Integer(1)])
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2  0 -1  0  0]
[ 0 -1  0  2 -1  0  0]
[ 0  0 -1 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
>>> CartanMatrix(['E', Integer(7), Integer(1)])
[ 2 -1  0  0  0  0  0  0]
[-1  2  0 -1  0  0  0  0]
[ 0  0  2  0 -1  0  0  0]
[ 0 -1  0  2 -1  0  0  0]
[ 0  0 -1 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
>>> CartanMatrix(['E', Integer(8), Integer(1)])
[ 2  0  0  0  0  0  0  0 -1]
[ 0  2  0 -1  0  0  0  0  0]
[ 0  0  2  0 -1  0  0  0  0]
[ 0 -1  0  2 -1  0  0  0  0]
[ 0  0 -1 -1  2 -1  0  0  0]
[ 0  0  0  0 -1  2 -1  0  0]
[ 0  0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0  0 -1  2 -1]
[-1  0  0  0  0  0  0 -1  2]
>>> CartanMatrix(['F', Integer(4), Integer(1)])
[ 2 -1  0  0  0]
[-1  2 -1  0  0]
[ 0 -1  2 -1  0]
[ 0  0 -2  2 -1]
[ 0  0  0 -1  2]
>>> CartanMatrix(['G', Integer(2), Integer(1)])
[ 2  0 -1]
[ 0  2 -3]
[-1 -1  2]
```

Examples of Borcherds-Cartan matrices:

```
sage: CartanMatrix([[2,-1],[-1,-2]], borcherds=True)                          #␣
↪needs sage.graphs
[ 2 -1]
[-1 -2]
sage: CartanMatrix('B3', borcherds=[-4,-6,2])                                 #␣
↪needs sage.graphs
[-4 -1  0]
[-1 -6 -1]
[ 0 -2  2]
```

```
>>> from sage.all import *
>>> CartanMatrix([[Integer(2),-Integer(1)],[-Integer(1),-Integer(2)]],␣
↪borcherds=True)                              # needs sage.graphs
[ 2 -1]
[-1 -2]
>>> CartanMatrix('B3', borcherds=[-Integer(4),-Integer(6),Integer(2)])        ␣
↪                          # needs sage.graphs
[-4 -1  0]
[-1 -6 -1]
[ 0 -2  2]
```

> **Note**
>
> Since this is a matrix, `row()` and `column()` will return the standard row and column respectively. To get the row with the indices as in Dynkin diagrams/Cartan types, use *row_with_indices()* and *column_with_indices()* respectively.

cartan_matrix()

Return the Cartan matrix of `self`.

EXAMPLES:

```
sage: CartanMatrix(['C',3]).cartan_matrix()                                   #␣
↪needs sage.graphs
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

```
>>> from sage.all import *
>>> CartanMatrix(['C',Integer(3)]).cartan_matrix()                            ␣
↪      # needs sage.graphs
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

cartan_type()

Return the Cartan type of `self` or `self` if unknown.

EXAMPLES:

```
sage: C = CartanMatrix(['A',4,1])                                             #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
sage: C.cartan_type()                                                   #␣
↪needs sage.graphs
['A', 4, 1]
```

```
>>> from sage.all import *
>>> C = CartanMatrix(['A',Integer(4),Integer(1)])                      ␣
↪               # needs sage.graphs
>>> C.cartan_type()                                                   #␣
↪needs sage.graphs
['A', 4, 1]
```

If the Cartan type is unknown:

```
sage: C = CartanMatrix([[2,-1,-2], [-1,2,-1], [-2,-1,2]])               #␣
↪needs sage.graphs
sage: C.cartan_type()                                                   #␣
↪needs sage.graphs
[ 2 -1 -2]
[-1  2 -1]
[-2 -1  2]
```

```
>>> from sage.all import *
>>> C = CartanMatrix([[Integer(2),-Integer(1),-Integer(2)], [-Integer(1),
↪Integer(2),-Integer(1)], [-Integer(2),-Integer(1),Integer(2)]])        ␣
↪       # needs sage.graphs
>>> C.cartan_type()                                                   #␣
↪needs sage.graphs
[ 2 -1 -2]
[-1  2 -1]
[-2 -1  2]
```

column_with_indices(j)

Return the j-th column $(a_{i,j})_i$ of `self` as a container (or iterator) of tuples $(i, a_{i,j})$

EXAMPLES:

```
sage: M = CartanMatrix(['B',4])                                         #␣
↪needs sage.graphs
sage: [ (i,a) for (i,a) in M.column_with_indices(3) ]                   #␣
↪needs sage.graphs
[(3, 2), (2, -1), (4, -2)]
```

```
>>> from sage.all import *
>>> M = CartanMatrix(['B',Integer(4)])                                  ␣
↪      # needs sage.graphs
>>> [ (i,a) for (i,a) in M.column_with_indices(Integer(3)) ]            ␣
↪      # needs sage.graphs
[(3, 2), (2, -1), (4, -2)]
```

coxeter_diagram()

Construct the Coxeter diagram of `self`.

See also:

CartanType_abstract.coxeter_diagram()

EXAMPLES:

```
sage: # needs sage.graphs
sage: cm = CartanMatrix([[2,-5,0],[-2,2,-1],[0,-1,2]])
sage: G = cm.coxeter_diagram(); G
Graph on 3 vertices
sage: G.edges(sort=True)
[(0, 1, +Infinity), (1, 2, 3)]
sage: ct = CartanType([['A',2,2], ['B',3]])
sage: ct.coxeter_diagram()
Graph on 5 vertices
sage: ct.cartan_matrix().coxeter_diagram() == ct.coxeter_diagram()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> cm = CartanMatrix([[Integer(2),-Integer(5),Integer(0)],[-Integer(2),
↪Integer(2),-Integer(1)],[Integer(0),-Integer(1),Integer(2)]])
>>> G = cm.coxeter_diagram(); G
Graph on 3 vertices
>>> G.edges(sort=True)
[(0, 1, +Infinity), (1, 2, 3)]
>>> ct = CartanType([['A',Integer(2),Integer(2)], ['B',Integer(3)]])
>>> ct.coxeter_diagram()
Graph on 5 vertices
>>> ct.cartan_matrix().coxeter_diagram() == ct.coxeter_diagram()
True
```

coxeter_matrix()

Return the Coxeter matrix for `self`.

See also:

CartanType_abstract.coxeter_matrix()

EXAMPLES:

```
sage: # needs sage.graphs
sage: cm = CartanMatrix([[2,-5,0],[-2,2,-1],[0,-1,2]])
sage: cm.coxeter_matrix()
[ 1 -1  2]
[-1  1  3]
[ 2  3  1]
sage: ct = CartanType([['A',2,2], ['B',3]])
sage: ct.coxeter_matrix()
[ 1 -1  2  2  2]
[-1  1  2  2  2]
```

(continues on next page)

(continued from previous page)

```
[ 2  2  1  3  2]
[ 2  2  3  1  4]
[ 2  2  2  4  1]
sage: ct.cartan_matrix().coxeter_matrix() == ct.coxeter_matrix()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> cm = CartanMatrix([[Integer(2),-Integer(5),Integer(0)],[-Integer(2),
↪Integer(2),-Integer(1)],[Integer(0),-Integer(1),Integer(2)]])
>>> cm.coxeter_matrix()
[ 1 -1  2]
[-1  1  3]
[ 2  3  1]
>>> ct = CartanType([['A',Integer(2),Integer(2)], ['B',Integer(3)]])
>>> ct.coxeter_matrix()
[ 1 -1  2  2  2]
[-1  1  2  2  2]
[ 2  2  1  3  2]
[ 2  2  3  1  4]
[ 2  2  2  4  1]
>>> ct.cartan_matrix().coxeter_matrix() == ct.coxeter_matrix()
True
```

dual()

Return the dual Cartan matrix of `self`, which is obtained by taking the transpose.

EXAMPLES:

```
sage: # needs sage.graphs
sage: ct = CartanType(['C',3])
sage: M = CartanMatrix(ct); M
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
sage: M.dual()
[ 2 -1  0]
[-1  2 -1]
[ 0 -2  2]
sage: M.dual() == CartanMatrix(ct.dual())
True
sage: M.dual().cartan_type() == ct.dual()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> ct = CartanType(['C',Integer(3)])
>>> M = CartanMatrix(ct); M
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
>>> M.dual()
```

(continues on next page)

(continued from previous page)

```
[ 2 -1  0]
[-1  2 -1]
[ 0 -2  2]
>>> M.dual() == CartanMatrix(ct.dual())
True
>>> M.dual().cartan_type() == ct.dual()
True
```

An example with arbitrary Cartan matrices:

```
sage: # needs sage.graphs
sage: cm = CartanMatrix([[2,-5], [-2, 2]]); cm
[ 2 -5]
[-2  2]
sage: cm.dual()
[ 2 -2]
[-5  2]
sage: cm.dual() == CartanMatrix(cm.transpose())
True
sage: cm.dual().dual() == cm
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> cm = CartanMatrix([[Integer(2),-Integer(5)], [-Integer(2), Integer(2)]]);
↪cm
[ 2 -5]
[-2  2]
>>> cm.dual()
[ 2 -2]
[-5  2]
>>> cm.dual() == CartanMatrix(cm.transpose())
True
>>> cm.dual().dual() == cm
True
```

dynkin_diagram()

Return the Dynkin diagram corresponding to `self`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: C = CartanMatrix(['A',2])
sage: C.dynkin_diagram()
O---O
1   2
A2
sage: C = CartanMatrix(['F',4,1])
sage: C.dynkin_diagram()
O---O---O=>=O---O
0   1   2   3   4
F4~
```

(continues on next page)

(continued from previous page)

```
sage: C = CartanMatrix([[2,-4],[-4,2]])
sage: C.dynkin_diagram()
Dynkin diagram of rank 2
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> C = CartanMatrix(['A',Integer(2)])
>>> C.dynkin_diagram()
O---O
1   2
A2
>>> C = CartanMatrix(['F',Integer(4),Integer(1)])
>>> C.dynkin_diagram()
O---O---O=>=O---O
0   1   2   3   4
F4~
>>> C = CartanMatrix([[Integer(2),-Integer(4)],[-Integer(4),Integer(2)]])
>>> C.dynkin_diagram()
Dynkin diagram of rank 2
```

indecomposable_blocks()

Return a tuple of all indecomposable blocks of `self`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = CartanMatrix(['A',2])
sage: M.indecomposable_blocks()
(
[ 2 -1]
[-1  2]
)
sage: M = CartanMatrix([['A',2,1],['A',3,1]])
sage: M.indecomposable_blocks()
(
[ 2 -1  0 -1]
[-1  2 -1  0]  [ 2 -1 -1]
[ 0 -1  2 -1]  [-1  2 -1]
[-1  0 -1  2], [-1 -1  2]
)
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = CartanMatrix(['A',Integer(2)])
>>> M.indecomposable_blocks()
(
[ 2 -1]
[-1  2]
)
>>> M = CartanMatrix([['A',Integer(2),Integer(1)],['A',Integer(3),
↪Integer(1)]])
>>> M.indecomposable_blocks()
```

(continues on next page)

(continued from previous page)

```
(
[ 2 -1  0 -1]
[-1  2 -1  0]  [ 2 -1 -1]
[ 0 -1  2 -1]  [-1  2 -1]
[-1  0 -1  2], [-1 -1  2]
)
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: C = CartanMatrix(['A',1,1])
sage: C.index_set()
(0, 1)
sage: C = CartanMatrix(['E',6])
sage: C.index_set()
(1, 2, 3, 4, 5, 6)
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> C = CartanMatrix(['A',Integer(1),Integer(1)])
>>> C.index_set()
(0, 1)
>>> C = CartanMatrix(['E',Integer(6)])
>>> C.index_set()
(1, 2, 3, 4, 5, 6)
```

is_affine()

Return `True` if `self` is an affine type or `False` otherwise.

A generalized Cartan matrix is affine if all of its indecomposable blocks are either finite (see *is_finite()*) or have zero determinant with all proper principal minors positive.

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = CartanMatrix(['C',4])
sage: M.is_affine()
False
sage: M = CartanMatrix(['D',4,1])
sage: M.is_affine()
True
sage: M = CartanMatrix([[2, -4], [-3, 2]])
sage: M.is_affine()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = CartanMatrix(['C',Integer(4)])
>>> M.is_affine()
False
```

(continues on next page)

(continued from previous page)

```
>>> M = CartanMatrix(['D',Integer(4),Integer(1)])
>>> M.is_affine()
True
>>> M = CartanMatrix([[Integer(2), -Integer(4)], [-Integer(3), Integer(2)]])
>>> M.is_affine()
False
```

is_crystallographic()

Implement *CartanType_abstract.is_crystallographic()*.

A Cartan matrix is crystallographic if it is symmetrizable.

EXAMPLES:

```
sage: CartanMatrix(['F',4]).is_crystallographic()                    #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> CartanMatrix(['F',Integer(4)]).is_crystallographic()              ␣
↪     # needs sage.graphs
True
```

is_finite()

Return True if self is a finite type or False otherwise.

A generalized Cartan matrix is finite if the determinant of all its principal submatrices (see *principal_submatrices()*) is positive. Such matrices have a positive definite symmetrized matrix. Note that a finite matrix may consist of multiple blocks of Cartan matrices each having finite Cartan type.

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = CartanMatrix(['C',4])
sage: M.is_finite()
True
sage: M = CartanMatrix(['D',4,1])
sage: M.is_finite()
False
sage: M = CartanMatrix([[2, -4], [-3, 2]])
sage: M.is_finite()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = CartanMatrix(['C',Integer(4)])
>>> M.is_finite()
True
>>> M = CartanMatrix(['D',Integer(4),Integer(1)])
>>> M.is_finite()
False
>>> M = CartanMatrix([[Integer(2), -Integer(4)], [-Integer(3), Integer(2)]])
>>> M.is_finite()
False
```

is_hyperbolic(*compact=False*)

Return if `True` if `self` is a (compact) hyperbolic type or `False` otherwise.

An indecomposable generalized Cartan matrix is hyperbolic if it has negative determinant and if any proper connected subdiagram of its Dynkin diagram is of finite or affine type. It is compact hyperbolic if any proper connected subdiagram has finite type.

INPUT:

- `compact` – if `True`, check if matrix is compact hyperbolic

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = CartanMatrix([[2,-2,0],[-2,2,-1],[0,-1,2]])
sage: M.is_hyperbolic()
True
sage: M.is_hyperbolic(compact=True)
False
sage: M = CartanMatrix([[2,-3],[-3,2]])
sage: M.is_hyperbolic()
True
sage: M = CartanMatrix(['C',4])
sage: M.is_hyperbolic()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = CartanMatrix([[Integer(2),-Integer(2),Integer(0)],[-Integer(2),
↪Integer(2),-Integer(1)],[Integer(0),-Integer(1),Integer(2)]])
>>> M.is_hyperbolic()
True
>>> M.is_hyperbolic(compact=True)
False
>>> M = CartanMatrix([[Integer(2),-Integer(3)],[-Integer(3),Integer(2)]])
>>> M.is_hyperbolic()
True
>>> M = CartanMatrix(['C',Integer(4)])
>>> M.is_hyperbolic()
False
```

is_indecomposable()

Return if `self` is an indecomposable matrix or `False` otherwise.

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = CartanMatrix(['A',5])
sage: M.is_indecomposable()
True
sage: M = CartanMatrix([[2,-1,0],[-1,2,0],[0,0,2]])
sage: M.is_indecomposable()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = CartanMatrix(['A',Integer(5)])
>>> M.is_indecomposable()
True
>>> M = CartanMatrix([[Integer(2),-Integer(1),Integer(0)],[-Integer(1),
↪Integer(2),Integer(0)],[Integer(0),Integer(0),Integer(2)]])
>>> M.is_indecomposable()
False
```

is_indefinite()

Return if `self` is an indefinite type or `False` otherwise.

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = CartanMatrix([[2,-3],[-3,2]])
sage: M.is_indefinite()
True
sage: M = CartanMatrix("A2")
sage: M.is_indefinite()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = CartanMatrix([[Integer(2),-Integer(3)],[-Integer(3),Integer(2)]])
>>> M.is_indefinite()
True
>>> M = CartanMatrix("A2")
>>> M.is_indefinite()
False
```

is_lorentzian()

Return `True` if `self` is a Lorentzian type or `False` otherwise.

A generalized Cartan matrix is Lorentzian if it has negative determinant and exactly one negative eigenvalue.

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = CartanMatrix([[2,-3],[-3,2]])
sage: M.is_lorentzian()
True
sage: M = CartanMatrix([[2,-1],[-1,2]])
sage: M.is_lorentzian()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = CartanMatrix([[Integer(2),-Integer(3)],[-Integer(3),Integer(2)]])
>>> M.is_lorentzian()
True
>>> M = CartanMatrix([[Integer(2),-Integer(1)],[-Integer(1),Integer(2)]])
```

(continues on next page)

(continued from previous page)

```
>>> M.is_lorentzian()
False
```

is_simply_laced()

Implement `CartanType_abstract.is_simply_laced()`.

A Cartan matrix is simply-laced if all non diagonal entries are 0 or -1.

EXAMPLES:

```
sage: cm = CartanMatrix([[2, -1, -1, -1], [-1, 2, -1, -1],                    #␣
↪needs sage.graphs
....:                    [-1, -1, 2, -1], [-1, -1, -1, 2]])
sage: cm.is_simply_laced()                                                      #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> cm = CartanMatrix([[Integer(2), -Integer(1), -Integer(1), -Integer(1)], [-
↪Integer(1), Integer(2), -Integer(1), -Integer(1)],                       # needs␣
↪sage.graphs
...                    [-Integer(1), -Integer(1), Integer(2), -Integer(1)], [-
↪Integer(1), -Integer(1), -Integer(1), Integer(2)]])
>>> cm.is_simply_laced()                                                        #␣
↪needs sage.graphs
True
```

matrix_space(*nrows=None*, *ncols=None*, *sparse=None*)

Return a matrix space over the integers.

INPUT:

- `nrows` – number of rows
- `ncols` – number of columns
- `sparse` – boolean

EXAMPLES:

```
sage: # needs sage.graphs
sage: cm = CartanMatrix(['A', 3])
sage: cm.matrix_space()
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
sage: cm.matrix_space(2, 2)
Full MatrixSpace of 2 by 2 sparse matrices over Integer Ring
sage: cm[:2,1:]    # indirect doctest
[-1  0]
[ 2 -1]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> cm = CartanMatrix(['A', Integer(3)])
>>> cm.matrix_space()
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
```

(continues on next page)

(continued from previous page)

```
>>> cm.matrix_space(Integer(2), Integer(2))
Full MatrixSpace of 2 by 2 sparse matrices over Integer Ring
>>> cm[:Integer(2),Integer(1):]    # indirect doctest
[-1  0]
[ 2 -1]
```

principal_submatrices(*proper=False*)

Return a list of all principal submatrices of `self`.

INPUT:

- `proper` – if `True`, return only proper submatrices

EXAMPLES:

```
sage: M = CartanMatrix(['A',2])                                               #␣
↪needs sage.graphs
sage: M.principal_submatrices()                                               #␣
↪needs sage.graphs
[
              [ 2 -1]
[], [2], [2], [-1  2]
]
sage: M.principal_submatrices(proper=True)                                    #␣
↪needs sage.graphs
[[], [2], [2]]
```

```
>>> from sage.all import *
>>> M = CartanMatrix(['A',Integer(2)])                                         ␣
↪     # needs sage.graphs
>>> M.principal_submatrices()                                                #␣
↪needs sage.graphs
[
              [ 2 -1]
[], [2], [2], [-1  2]
]
>>> M.principal_submatrices(proper=True)                                     #␣
↪needs sage.graphs
[[], [2], [2]]
```

rank()

Return the rank of `self`.

EXAMPLES:

```
sage: CartanMatrix(['C',3]).rank()                                            #␣
↪needs sage.graphs
3
sage: CartanMatrix(["A2","B2","F4"]).rank()                                   #␣
↪needs sage.graphs
8
```

```
>>> from sage.all import *
>>> CartanMatrix(['C',Integer(3)]).rank()                                          ␣
↪     # needs sage.graphs
3
>>> CartanMatrix(["A2","B2","F4"]).rank()                                        #␣
↪needs sage.graphs
8
```

reflection_group(*type='matrix'*)

Return the reflection group corresponding to `self`.

EXAMPLES:

```
sage: C = CartanMatrix(['A',3])                                                  #␣
↪needs sage.graphs
sage: C.reflection_group()                                                       #␣
↪needs sage.graphs sage.libs.gap
Weyl Group of type ['A', 3] (as a matrix group acting on the root space)
```

```
>>> from sage.all import *
>>> C = CartanMatrix(['A',Integer(3)])                                           ␣
↪     # needs sage.graphs
>>> C.reflection_group()                                                         #␣
↪needs sage.graphs sage.libs.gap
Weyl Group of type ['A', 3] (as a matrix group acting on the root space)
```

relabel(*relabelling*)

Return the relabelled Cartan matrix.

EXAMPLES:

```
sage: # needs sage.graphs
sage: CM = CartanMatrix(['C',3])
sage: R = CM.relabel({1:0, 2:4, 3:1}); R
[ 2  0 -1]
[ 0  2 -1]
[-1 -2  2]
sage: R.index_set()
(0, 1, 4)
sage: CM
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CM = CartanMatrix(['C',Integer(3)])
>>> R = CM.relabel({Integer(1):Integer(0), Integer(2):Integer(4),␣
↪Integer(3):Integer(1)}); R
[ 2  0 -1]
[ 0  2 -1]
[-1 -2  2]
>>> R.index_set()
```

(continues on next page)

(continued from previous page)

```
(0, 1, 4)
>>> CM
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

root_space()

Return the root space corresponding to `self`.

EXAMPLES:

```
sage: C = CartanMatrix(['A',3])                                                 #␣
↪needs sage.graphs
sage: C.root_space()                                                            #␣
↪needs sage.graphs
Root space over the Rational Field of the Root system of type ['A', 3]
```

```
>>> from sage.all import *
>>> C = CartanMatrix(['A',Integer(3)])                                          ␣
↪     # needs sage.graphs
>>> C.root_space()                                                            #␣
↪needs sage.graphs
Root space over the Rational Field of the Root system of type ['A', 3]
```

root_system()

Return the root system corresponding to `self`.

EXAMPLES:

```
sage: C = CartanMatrix(['A',3])                                                 #␣
↪needs sage.graphs
sage: C.root_system()                                                           #␣
↪needs sage.graphs
Root system of type ['A', 3]
```

```
>>> from sage.all import *
>>> C = CartanMatrix(['A',Integer(3)])                                          ␣
↪     # needs sage.graphs
>>> C.root_system()                                                           #␣
↪needs sage.graphs
Root system of type ['A', 3]
```

row_with_indices(i)

Return the i-th row $(a_{i,j})_j$ of `self` as a container (or iterator) of tuples $(j, a_{i,j})$

EXAMPLES:

```
sage: M = CartanMatrix(['C',4])                                                 #␣
↪needs sage.graphs
sage: [ (i,a) for (i,a) in M.row_with_indices(3) ]                              #␣
↪needs sage.graphs
[(3, 2), (2, -1), (4, -2)]
```

```
>>> from sage.all import *
>>> M = CartanMatrix(['C',Integer(4)])                                    ␣
↪     # needs sage.graphs
>>> [ (i,a) for (i,a) in M.row_with_indices(Integer(3)) ]                 ␣
↪     # needs sage.graphs
[(3, 2), (2, -1), (4, -2)]
```

subtype(*index_set*)

Return a subtype of `self` given by `index_set`.

A subtype can be considered the Dynkin diagram induced from the Dynkin diagram of `self` by `index_set`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: C = CartanMatrix(['F',4])
sage: S = C.subtype([1,2,3])
sage: S
[ 2 -1  0]
[-1  2 -1]
[ 0 -2  2]
sage: S.index_set()
(1, 2, 3)
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> C = CartanMatrix(['F',Integer(4)])
>>> S = C.subtype([Integer(1),Integer(2),Integer(3)])
>>> S
[ 2 -1  0]
[-1  2 -1]
[ 0 -2  2]
>>> S.index_set()
(1, 2, 3)
```

symmetrized_matrix()

Return the symmetrized matrix of `self` if symmetrizable.

EXAMPLES:

```
sage: cm = CartanMatrix(['B',4,1])                                         #␣
↪needs sage.graphs
sage: cm.symmetrized_matrix()                                              #␣
↪needs sage.graphs
[ 4  0 -2  0  0]
[ 0  4 -2  0  0]
[-2 -2  4 -2  0]
[ 0  0 -2  4 -2]
[ 0  0  0 -2  2]
```

```
>>> from sage.all import *
>>> cm = CartanMatrix(['B',Integer(4),Integer(1)])                        ␣
↪              # needs sage.graphs
>>> cm.symmetrized_matrix()                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[ 4  0 -2  0  0]
[ 0  4 -2  0  0]
[-2 -2  4 -2  0]
[ 0  0 -2  4 -2]
[ 0  0  0 -2  2]
```

symmetrizer()

Return the symmetrizer of `self`.

EXAMPLES:

```
sage: cm = CartanMatrix([[2,-5],[-2,2]])                                  #␣
↪needs sage.graphs
sage: cm.symmetrizer()                                                    #␣
↪needs sage.graphs
Finite family {0: 2, 1: 5}
```

```
>>> from sage.all import *
>>> cm = CartanMatrix([[Integer(2),-Integer(5)],[-Integer(2),Integer(2)]])  ␣
↪                                  # needs sage.graphs
>>> cm.symmetrizer()                                                     #␣
↪needs sage.graphs
Finite family {0: 2, 1: 5}
```

sage.combinat.root_system.cartan_matrix.**find_cartan_type_from_matrix**(*CM*)

Find a Cartan type by direct comparison of Dynkin diagrams given from the generalized Cartan matrix `CM` and return `None` if not found.

INPUT:

- `CM` – a generalized Cartan matrix

EXAMPLES:

```
sage: # needs sage.graphs
sage: from sage.combinat.root_system.cartan_matrix import find_cartan_type_from_
↪matrix
sage: CM = CartanMatrix([[2,-1,-1], [-1,2,-1], [-1,-1,2]])
sage: find_cartan_type_from_matrix(CM)
['A', 2, 1]
sage: CM = CartanMatrix([[2,-1,0], [-1,2,-2], [0,-1,2]])
sage: find_cartan_type_from_matrix(CM)
['C', 3] relabelled by {1: 0, 2: 1, 3: 2}
sage: CM = CartanMatrix([[2,-1,-2], [-1,2,-1], [-2,-1,2]])
sage: find_cartan_type_from_matrix(CM)
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> from sage.combinat.root_system.cartan_matrix import find_cartan_type_from_
↪matrix
>>> CM = CartanMatrix([[Integer(2),-Integer(1),-Integer(1)], [-Integer(1),
↪Integer(2),-Integer(1)], [-Integer(1),-Integer(1),Integer(2)]])
```

(continues on next page)

(continued from previous page)

```
>>> find_cartan_type_from_matrix(CM)
['A', 2, 1]
>>> CM = CartanMatrix([[Integer(2),-Integer(1),Integer(0)], [-Integer(1),
↪Integer(2),-Integer(2)], [Integer(0),-Integer(1),Integer(2)]])
>>> find_cartan_type_from_matrix(CM)
['C', 3] relabelled by {1: 0, 2: 1, 3: 2}
>>> CM = CartanMatrix([[Integer(2),-Integer(1),-Integer(2)], [-Integer(1),
↪Integer(2),-Integer(1)], [-Integer(2),-Integer(1),Integer(2)]])
>>> find_cartan_type_from_matrix(CM)
```

sage.combinat.root_system.cartan_matrix.**is_borcherds_cartan_matrix**(*M*)

Return `True` if `M` is an even, integral Borcherds-Cartan matrix. For a definition of such a matrix, see *CartanMatrix*.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_matrix import is_borcherds_cartan_
↪matrix
sage: M = Matrix([[2,-1],[-1,2]])
sage: is_borcherds_cartan_matrix(M)
True
sage: N = Matrix([[2,-1],[-1,0]])
sage: is_borcherds_cartan_matrix(N)
False
sage: O = Matrix([[2,-1],[-1,-2]])
sage: is_borcherds_cartan_matrix(O)
True
sage: O = Matrix([[2,-1],[-1,-3]])
sage: is_borcherds_cartan_matrix(O)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.cartan_matrix import is_borcherds_cartan_matrix
>>> M = Matrix([[Integer(2),-Integer(1)],[-Integer(1),Integer(2)]])
>>> is_borcherds_cartan_matrix(M)
True
>>> N = Matrix([[Integer(2),-Integer(1)],[-Integer(1),Integer(0)]])
>>> is_borcherds_cartan_matrix(N)
False
>>> O = Matrix([[Integer(2),-Integer(1)],[-Integer(1),-Integer(2)]])
>>> is_borcherds_cartan_matrix(O)
True
>>> O = Matrix([[Integer(2),-Integer(1)],[-Integer(1),-Integer(3)]])
>>> is_borcherds_cartan_matrix(O)
False
```

sage.combinat.root_system.cartan_matrix.**is_generalized_cartan_matrix**(*M*)

Return `True` if `M` is a generalized Cartan matrix. For a definition of a generalized Cartan matrix, see *CartanMatrix*.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_matrix import is_generalized_cartan_
↪matrix
sage: M = matrix([[2,-1,-2], [-1,2,-1], [-2,-1,2]])
sage: is_generalized_cartan_matrix(M)
True
sage: M = matrix([[2,-1,-2], [-1,2,-1], [0,-1,2]])
sage: is_generalized_cartan_matrix(M)
False
sage: M = matrix([[1,-1,-2], [-1,2,-1], [-2,-1,2]])
sage: is_generalized_cartan_matrix(M)
False
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.cartan_matrix import is_generalized_cartan_
↪matrix
>>> M = matrix([[Integer(2),-Integer(1),-Integer(2)], [-Integer(1),Integer(2),-
↪Integer(1)], [-Integer(2),-Integer(1),Integer(2)]])
>>> is_generalized_cartan_matrix(M)
True
>>> M = matrix([[Integer(2),-Integer(1),-Integer(2)], [-Integer(1),Integer(2),-
↪Integer(1)], [Integer(0),-Integer(1),Integer(2)]])
>>> is_generalized_cartan_matrix(M)
False
>>> M = matrix([[Integer(1),-Integer(1),-Integer(2)], [-Integer(1),Integer(2),-
↪Integer(1)], [-Integer(2),-Integer(1),Integer(2)]])
>>> is_generalized_cartan_matrix(M)
False
```

A non-symmetrizable example:

```
sage: M = matrix([[2,-1,-2], [-1,2,-1], [-1,-1,2]])
sage: is_generalized_cartan_matrix(M)
True
```

```
>>> from sage.all import *
>>> M = matrix([[Integer(2),-Integer(1),-Integer(2)], [-Integer(1),Integer(2),-
↪Integer(1)], [-Integer(1),-Integer(1),Integer(2)]])
>>> is_generalized_cartan_matrix(M)
True
```

5.1.226 Cartan types

Todo

Why does sphinx complain if I use sections here?

Introduction

Loosely speaking, Dynkin diagrams (or equivalently Cartan matrices) are graphs which are used to classify root systems, Coxeter and Weyl groups, Lie algebras, Lie groups, crystals, etc. up to an isomorphism. *Cartan types* are a standard set of names for those Dynkin diagrams (see Wikipedia article Dynkin_diagram).

Let us consider, for example, the Cartan type A_4:

```
sage: T = CartanType(['A', 4]); T
['A', 4]
```

```
>>> from sage.all import *
>>> T = CartanType(['A', Integer(4)]); T
['A', 4]
```

It is the name of the following Dynkin diagram:

```
sage: DynkinDiagram(T)                                                        #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

```
>>> from sage.all import *
>>> DynkinDiagram(T)                                                        #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

Note

For convenience, the following shortcuts are available:

```
sage: DynkinDiagram(['A',4])                                                 #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
sage: DynkinDiagram('A4')                                                    #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
sage: T.dynkin_diagram()                                                     #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

```
>>> from sage.all import *
>>> DynkinDiagram(['A',Integer(4)])                                            ␣
↪    # needs sage.graphs
O---O---O---O
1   2   3   4
A4
>>> DynkinDiagram('A4')                                                    #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
>>> T.dynkin_diagram()                                                     #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

See `DynkinDiagram` for how to further manipulate Dynkin diagrams.

From this data (the *Cartan datum*), one can construct the associated root system:

```
sage: RootSystem(T)
Root system of type ['A', 4]
```

```
>>> from sage.all import *
>>> RootSystem(T)
Root system of type ['A', 4]
```

The associated Weyl group of A_n is the symmetric group S_{n+1}:

```
sage: W = WeylGroup(T); W                                                  #␣
↪needs sage.libs.gap
Weyl Group of type ['A', 4] (as a matrix group acting on the ambient space)
sage: W.cardinality()                                                      #␣
↪needs sage.libs.gap
120
```

```
>>> from sage.all import *
>>> W = WeylGroup(T); W                                                  #␣
↪needs sage.libs.gap
Weyl Group of type ['A', 4] (as a matrix group acting on the ambient space)
>>> W.cardinality()                                                      #␣
↪needs sage.libs.gap
120
```

while the Lie algebra is sl_{n+1}, and the Lie group SL_{n+1} (TODO: illustrate this once this is implemented).

One may also construct crystals associated to various Dynkin diagrams. For example:

```
sage: C = crystals.Letters(T); C                                           #␣
↪needs sage.combinat
The crystal of letters for type ['A', 4]
sage: C.list()                                                             #␣
↪needs sage.combinat
[1, 2, 3, 4, 5]

sage: C = crystals.Tableaux(T, shape=[2]); C                               #␣
↪needs sage.combinat
The crystal of tableaux of type ['A', 4] and shape(s) [[2]]
sage: C.cardinality()                                                      #␣
↪needs sage.combinat
15
```

```
>>> from sage.all import *
>>> C = crystals.Letters(T); C                                           #␣
↪needs sage.combinat
The crystal of letters for type ['A', 4]
>>> C.list()                                                             #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
[1, 2, 3, 4, 5]

>>> C = crystals.Tableaux(T, shape=[Integer(2)]); C                              ␣
↪     # needs sage.combinat
The crystal of tableaux of type ['A', 4] and shape(s) [[2]]
>>> C.cardinality()                                                            #␣
↪needs sage.combinat
15
```

Here is a sample of all the finite irreducible crystallographic Cartan types:

```
sage: CartanType.samples(finite=True, crystallographic=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], ['D',
↪ 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2]]
```

```
>>> from sage.all import *
>>> CartanType.samples(finite=True, crystallographic=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], ['D',
↪ 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2]]
```

One can also get latex representations of the crystallographic Cartan types and their corresponding Dynkin diagrams:

```
sage: [latex(ct) for ct in CartanType.samples(crystallographic=True)]
[A_{1}, A_{5}, B_{1}, B_{5}, C_{1}, C_{5}, D_{2}, D_{3}, D_{5},
 E_6, E_7, E_8, F_4, G_2,
 A_{1}^{(1)}, A_{5}^{(1)}, B_{1}^{(1)}, B_{5}^{(1)},
 C_{1}^{(1)}, C_{5}^{(1)}, D_{3}^{(1)}, D_{5}^{(1)},
 E_6^{(1)}, E_7^{(1)}, E_8^{(1)}, F_4^{(1)}, G_2^{(1)},
 BC_{1}^{(2)}, BC_{5}^{(2)},
 B_{5}^{(1)\vee}, C_{4}^{(1)\vee}, F_4^{(1)\vee},
 G_2^{(1)\vee}, BC_{1}^{(2)\vee}, BC_{5}^{(2)\vee}]
sage: view([DynkinDiagram(ct) for ct in CartanType.samples(crystallographic=True)]) #␣
↪not tested
```

```
>>> from sage.all import *
>>> [latex(ct) for ct in CartanType.samples(crystallographic=True)]
[A_{1}, A_{5}, B_{1}, B_{5}, C_{1}, C_{5}, D_{2}, D_{3}, D_{5},
 E_6, E_7, E_8, F_4, G_2,
 A_{1}^{(1)}, A_{5}^{(1)}, B_{1}^{(1)}, B_{5}^{(1)},
 C_{1}^{(1)}, C_{5}^{(1)}, D_{3}^{(1)}, D_{5}^{(1)},
 E_6^{(1)}, E_7^{(1)}, E_8^{(1)}, F_4^{(1)}, G_2^{(1)},
 BC_{1}^{(2)}, BC_{5}^{(2)},
 B_{5}^{(1)\vee}, C_{4}^{(1)\vee}, F_4^{(1)\vee},
 G_2^{(1)\vee}, BC_{1}^{(2)\vee}, BC_{5}^{(2)\vee}]
>>> view([DynkinDiagram(ct) for ct in CartanType.samples(crystallographic=True)]) #␣
↪not tested
```

Non-crystallographic Cartan types are also partially supported:

```
sage: CartanType.samples(finite=True, crystallographic=False)
[['I', 5], ['H', 3], ['H', 4]]
```

```
>>> from sage.all import *
>>> CartanType.samples(finite=True, crystallographic=False)
[['I', 5], ['H', 3], ['H', 4]]
```

In Sage, a Cartan type is used as a database of type-specific information and algorithms (see e.g. `sage.combinat.root_system.type_A`). This database includes how to construct the Dynkin diagram, the ambient space for the root system (see Wikipedia article Root_system), and further mathematical properties:

```
sage: T.is_finite(), T.is_simply_laced(), T.is_affine(), T.is_crystallographic()
(True, True, False, True)
```

```
>>> from sage.all import *
>>> T.is_finite(), T.is_simply_laced(), T.is_affine(), T.is_crystallographic()
(True, True, False, True)
```

In particular, a Sage Cartan type is endowed with a fixed choice of labels for the nodes of the Dynkin diagram. This choice follows the conventions of Nicolas Bourbaki, Lie Groups and Lie Algebras: Chapter 4-6, Elements of Mathematics, Springer (2002). ISBN 978-3540426509. For example:

```
sage: T = CartanType(['D', 4])
sage: DynkinDiagram(T)                                                         #␣
↪needs sage.graphs
    O 4
    |
    |
O---O---O
1   2   3
D4

sage: E6 = CartanType(['E',6])
sage: DynkinDiagram(E6)                                                        #␣
↪needs sage.graphs
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
```

```
>>> from sage.all import *
>>> T = CartanType(['D', Integer(4)])
>>> DynkinDiagram(T)                                                           #␣
↪needs sage.graphs
    O 4
    |
    |
O---O---O
1   2   3
D4
```

(continues on next page)

(continued from previous page)

```
>>> E6 = CartanType(['E',Integer(6)])
>>> DynkinDiagram(E6)                                                    #␣
↪needs sage.graphs
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
```

Note

The direction of the arrows is the **opposite** (i.e. the transpose) of Bourbaki's convention, but agrees with Kac's.

For example, in type C_2, we have:

```
sage: C2 = DynkinDiagram(['C',2]); C2                                    #␣
↪needs sage.graphs
O=<=O
1   2
C2
sage: C2.cartan_matrix()                                                 #␣
↪needs sage.graphs
[ 2 -2]
[-1  2]
```

```
>>> from sage.all import *
>>> C2 = DynkinDiagram(['C',Integer(2)]); C2                              ␣
↪    # needs sage.graphs
O=<=O
1   2
C2
>>> C2.cartan_matrix()                                                  #␣
↪needs sage.graphs
[ 2 -2]
[-1  2]
```

However Bourbaki would have the Cartan matrix as:

$$\begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}.$$

If desired, other node labelling conventions can be achieved. For example the Kac labelling for type E_6 can be obtained via:

```
sage: E6.relabel({1:1,2:6,3:2,4:3,5:4,6:5}).dynkin_diagram()            #␣
↪needs sage.graphs
        O 6
        |
        |
O---O---O---O---O
```

(continues on next page)

(continued from previous page)

```
1   2   3   4   5
E6 relabelled by {1: 1, 2: 6, 3: 2, 4: 3, 5: 4, 6: 5}
```

```
>>> from sage.all import *
>>> E6.relabel({Integer(1):Integer(1),Integer(2):Integer(6),Integer(3):Integer(2),
↪Integer(4):Integer(3),Integer(5):Integer(4),Integer(6):Integer(5)}).dynkin_
↪diagram()                              # needs sage.graphs
        O 6
        |
        |
O---O---O---O---O
1   2   3   4   5
E6 relabelled by {1: 1, 2: 6, 3: 2, 4: 3, 5: 4, 6: 5}
```

Contributions implementing other conventions are very welcome.

Another option is to build from scratch a new Dynkin diagram. The architecture has been designed to make it fairly easy to add other labelling conventions. In particular, we strived at choosing type free algorithms whenever possible, so in principle most features should remain available even with custom Cartan types. This has not been used much yet, so some rough corners certainly remain.

Here, we construct the hyperbolic example of Exercise 4.9 p. 57 of Kac, Infinite Dimensional Lie Algebras. We start with an empty Dynkin diagram, and add a couple nodes:

```
sage: g = DynkinDiagram()                                                    #␣
↪needs sage.graphs
sage: g.add_vertices([1,2,3])                                                #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> g = DynkinDiagram()                                                    #␣
↪needs sage.graphs
>>> g.add_vertices([Integer(1),Integer(2),Integer(3)])                        ␣
↪                    # needs sage.graphs
```

Note that the diagonal of the Cartan matrix is already initialized:

```
sage: g.cartan_matrix()                                                      #␣
↪needs sage.graphs
[2 0 0]
[0 2 0]
[0 0 2]
```

```
>>> from sage.all import *
>>> g.cartan_matrix()                                                      #␣
↪needs sage.graphs
[2 0 0]
[0 2 0]
[0 0 2]
```

Then we add a couple edges:

```
sage: g.add_edge(1,2,2)                                                   #␣
↪needs sage.graphs
sage: g.add_edge(1,3)                                                     #␣
↪needs sage.graphs
sage: g.add_edge(2,3)                                                     #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> g.add_edge(Integer(1),Integer(2),Integer(2))                           ␣
↪                     # needs sage.graphs
>>> g.add_edge(Integer(1),Integer(3))                                      ␣
↪              # needs sage.graphs
>>> g.add_edge(Integer(2),Integer(3))                                      ␣
↪              # needs sage.graphs
```

and we get the desired Cartan matrix:

```
sage: g.cartan_matrix()                                                   #␣
↪needs sage.graphs
[2 0 0]
[0 2 0]
[0 0 2]
```

```
>>> from sage.all import *
>>> g.cartan_matrix()                                                   #␣
↪needs sage.graphs
[2 0 0]
[0 2 0]
[0 0 2]
```

Oops, the Cartan matrix did not change! This is because it is cached for efficiency (see `cached_method`). In general, a Dynkin diagram should not be modified after having been used.

Warning: this is not checked currently

Todo: add a method `set_mutable()` as, say, for matrices

Here, we can work around this by clearing the cache:

```
sage: delattr(g, 'cartan_matrix')                                         #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> delattr(g, 'cartan_matrix')                                         #␣
↪needs sage.graphs
```

Now we get the desired Cartan matrix:

```
sage: g.cartan_matrix()                                                  #␣
↪needs sage.graphs
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

```
>>> from sage.all import *
>>> g.cartan_matrix()                                                  #␣
↪needs sage.graphs
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

Note that backward edges have been automatically added:

```
sage: g.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(1, 2, 2), (1, 3, 1), (2, 1, 1), (2, 3, 1), (3, 1, 1), (3, 2, 1)]
```

```
>>> from sage.all import *
>>> g.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(1, 2, 2), (1, 3, 1), (2, 1, 1), (2, 3, 1), (3, 1, 1), (3, 2, 1)]
```

Reducible Cartan types

Reducible Cartan types can be specified by passing a sequence or list of irreducible Cartan types:

```
sage: CartanType(['A',2],['B',2])
A2xB2
sage: CartanType([['A',2],['B',2]])
A2xB2
sage: CartanType(['A',2],['B',2]).is_reducible()
True
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(2)],['B',Integer(2)])
A2xB2
>>> CartanType([['A',Integer(2)],['B',Integer(2)]])
A2xB2
>>> CartanType(['A',Integer(2)],['B',Integer(2)]).is_reducible()
True
```

or using the following short hand notation:

```
sage: CartanType("A2xB2")
A2xB2
sage: CartanType("A2","B2") == CartanType("A2xB2")
True
```

```
>>> from sage.all import *
>>> CartanType("A2xB2")
A2xB2
>>> CartanType("A2","B2") == CartanType("A2xB2")
True
```

Degenerate cases

When possible, type I_n is automatically converted to the isomorphic crystallographic Cartan types (any reason not to do so?):

```
sage: CartanType(["I",1])
A1xA1
sage: CartanType(["I",3])
['A', 2]
sage: CartanType(["I",4])
['C', 2]
sage: CartanType(["I",6])
['G', 2]
```

```
>>> from sage.all import *
>>> CartanType(["I",Integer(1)])
A1xA1
>>> CartanType(["I",Integer(3)])
['A', 2]
>>> CartanType(["I",Integer(4)])
['C', 2]
>>> CartanType(["I",Integer(6)])
['G', 2]
```

The Dynkin diagrams for types B_1, C_1, D_2, and D_3 are isomorphic to that for A_1, A_1, $A_1 \times A_1$, and A_3, respectively. However their natural ambient space realizations (stemming from the corresponding infinite families of Lie groups) are different. Therefore, the Cartan types are considered as distinct:

```
sage: CartanType(['B',1])
['B', 1]
sage: CartanType(['C',1])
['C', 1]
sage: CartanType(['D',2])
['D', 2]
sage: CartanType(['D',3])
['D', 3]
```

```
>>> from sage.all import *
>>> CartanType(['B',Integer(1)])
['B', 1]
>>> CartanType(['C',Integer(1)])
['C', 1]
>>> CartanType(['D',Integer(2)])
['D', 2]
>>> CartanType(['D',Integer(3)])
['D', 3]
```

Affine Cartan types

For affine types, we use the usual conventions for affine Coxeter groups: each affine type is either untwisted (that is arise from the natural affinisation of a finite Cartan type):

```
sage: CartanType(["A", 4, 1]).dynkin_diagram()                              #␣
↪needs sage.graphs
0
O-----------+
|           |
|           |
O---O---O---O
1   2   3   4
A4~
sage: CartanType(["B", 4, 1]).dynkin_diagram()                              #␣
↪needs sage.graphs
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
B4~
```

```
>>> from sage.all import *
>>> CartanType(["A", Integer(4), Integer(1)]).dynkin_diagram()               ␣
↪               # needs sage.graphs
0
O-----------+
|           |
|           |
O---O---O---O
1   2   3   4
A4~
>>> CartanType(["B", Integer(4), Integer(1)]).dynkin_diagram()               ␣
↪               # needs sage.graphs
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
B4~
```

or dual thereof:

```
sage: CartanType(["B", 4, 1]).dual().dynkin_diagram()                       #␣
↪needs sage.graphs
    O 0
    |
    |
O---O---O=<=O
1   2   3   4
B4~*
```

```
>>> from sage.all import *
>>> CartanType(["B", Integer(4), Integer(1)]).dual().dynkin_diagram()
↪                # needs sage.graphs
    O 0
    |
    |
O---O---O=<=O
1   2   3   4
B4~*
```

or is of type $\widetilde{BC}_n$ (which yields an irreducible, but nonreduced root system):

```
sage: CartanType(["BC", 4, 2]).dynkin_diagram()                          #␣
↪needs sage.graphs
O=<=O---O---O=<=O
0   1   2   3   4
BC4~
```

```
>>> from sage.all import *
>>> CartanType(["BC", Integer(4), Integer(2)]).dynkin_diagram()
↪                # needs sage.graphs
O=<=O---O---O=<=O
0   1   2   3   4
BC4~
```

This includes the two degenerate cases:

```
sage: CartanType(["A", 1, 1]).dynkin_diagram()                           #␣
↪needs sage.graphs
O<=>O
0   1
A1~
sage: CartanType(["BC", 1, 2]).dynkin_diagram()                          #␣
↪needs sage.graphs
  4
O=<=O
0   1
BC1~
```

```
>>> from sage.all import *
>>> CartanType(["A", Integer(1), Integer(1)]).dynkin_diagram()
↪                # needs sage.graphs
O<=>O
0   1
A1~
>>> CartanType(["BC", Integer(1), Integer(2)]).dynkin_diagram()
↪                # needs sage.graphs
  4
O=<=O
0   1
BC1~
```

For the user convenience, Kac's notations for twisted affine types are automatically translated into the previous ones:

```
sage: # needs sage.graphs
sage: CartanType(["A", 9, 2])
['B', 5, 1]^*
sage: CartanType(["A", 9, 2]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
B5~*
sage: CartanType(["A", 10, 2]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
BC5~
sage: CartanType(["D", 5, 2]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~*
sage: CartanType(["D", 4, 3]).dynkin_diagram()
  3
O=>=O---O
2   1   0
G2~* relabelled by {0: 0, 1: 2, 2: 1}
sage: CartanType(["E", 6, 2]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
F4~*
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CartanType(["A", Integer(9), Integer(2)])
['B', 5, 1]^*
>>> CartanType(["A", Integer(9), Integer(2)]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
B5~*
>>> CartanType(["A", Integer(10), Integer(2)]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
BC5~
>>> CartanType(["D", Integer(5), Integer(2)]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~*
>>> CartanType(["D", Integer(4), Integer(3)]).dynkin_diagram()
  3
O=>=O---O
2   1   0
G2~* relabelled by {0: 0, 1: 2, 2: 1}
```

(continues on next page)

(continued from previous page)

```
>>> CartanType(["E", Integer(6), Integer(2)]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
F4~*
```

Additionally one can set the notation option to use Kac's notation:

```
sage: # needs sage.graphs
sage: CartanType.options['notation'] = 'Kac'
sage: CartanType(["A", 9, 2])
['A', 9, 2]
sage: CartanType(["A", 9, 2]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
A9^2
sage: CartanType(["A", 10, 2]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
A10^2
sage: CartanType(["D", 5, 2]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
D5^2
sage: CartanType(["D", 4, 3]).dynkin_diagram()
  3
O=>=O---O
2   1   0
D4^3
sage: CartanType(["E", 6, 2]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
E6^2
sage: CartanType.options['notation'] = 'BC'
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CartanType.options['notation'] = 'Kac'
>>> CartanType(["A", Integer(9), Integer(2)])
['A', 9, 2]
>>> CartanType(["A", Integer(9), Integer(2)]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
A9^2
>>> CartanType(["A", Integer(10), Integer(2)]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
```

(continues on next page)

(continued from previous page)

```
A10^2
>>> CartanType(["D", Integer(5), Integer(2)]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
D5^2
>>> CartanType(["D", Integer(4), Integer(3)]).dynkin_diagram()
  3
O=>=O---O
2   1   0
D4^3
>>> CartanType(["E", Integer(6), Integer(2)]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
E6^2
>>> CartanType.options['notation'] = 'BC'
```

Infinite Cartan types

There are minimal implementations of the Cartan types A_∞ and $A_{+\infty}$. In sage oo is the same as $+Infinity$, so NN and ZZ are used to differentiate between the $A_{+\infty}$ and A_∞ root systems:

```
sage: CartanType(['A', NN])
['A', NN]
sage: print(CartanType(['A', NN]).ascii_art())
O---O---O---O---O---O---O---..
0   1   2   3   4   5   6
sage: CartanType(['A', ZZ])
['A', ZZ]
sage: print(CartanType(['A', ZZ]).ascii_art())
..---O---O---O---O---O---O---O---..
    -3  -2  -1   0   1   2   3
```

```
>>> from sage.all import *
>>> CartanType(['A', NN])
['A', NN]
>>> print(CartanType(['A', NN]).ascii_art())
O---O---O---O---O---O---O---..
0   1   2   3   4   5   6
>>> CartanType(['A', ZZ])
['A', ZZ]
>>> print(CartanType(['A', ZZ]).ascii_art())
..---O---O---O---O---O---O---O---..
    -3  -2  -1   0   1   2   3
```

There are also the following shorthands:

```
sage: CartanType("Aoo")
['A', ZZ]
sage: CartanType("A+oo")
['A', NN]
```

```
>>> from sage.all import *
>>> CartanType("Aoo")
['A', ZZ]
>>> CartanType("A+oo")
['A', NN]
```

Abstract classes for Cartan types

- *CartanType_abstract*
- *CartanType_crystallographic*
- *CartanType_simply_laced*
- *CartanType_simple*
- *CartanType_finite*
- *CartanType_affine* (see also *Root system data for affine Cartan types*)
- *sage.combinat.root_system.cartan_type.CartanType*
- *Root system data for dual Cartan types*
- *Root system data for reducible Cartan types*
- *Root system data for relabelled Cartan types*

Concrete classes for Cartan types

- *CartanType_standard*
- *CartanType_standard_finite*
- *CartanType_standard_affine*
- *CartanType_standard_untwisted_affine*

Type specific data

The data essentially consists of a description of the Dynkin/Coxeter diagram and, when relevant, of the natural embedding of the root system in a Euclidean space. Everything else is reconstructed from this data.

- *Root system data for type A*
- *Root system data for type B*
- *Root system data for type C*
- *Root system data for type D*
- *Root system data for type E*
- *Root system data for type F*
- *Root system data for type G*
- *Root system data for type H*
- *Root system data for type I*
- *Root system data for super type A*
- *Root system data for type Q*
- *Root system data for (untwisted) type A affine*
- *Root system data for (untwisted) type B affine*

- *Root system data for (untwisted) type C affine*
- *Root system data for (untwisted) type D affine*
- *Root system data for (untwisted) type E affine*
- *Root system data for (untwisted) type F affine*
- *Root system data for (untwisted) type G affine*
- *Root system data for type BC affine*
- *Root system data for type A infinity*

Todo

Should those indexes come before the introduction?

`sage.combinat.root_system.cartan_type.`**`CartanType`**`(*args)`

Cartan types

Todo

Why does sphinx complain if I use sections here?

Introduction

Loosely speaking, Dynkin diagrams (or equivalently Cartan matrices) are graphs which are used to classify root systems, Coxeter and Weyl groups, Lie algebras, Lie groups, crystals, etc. up to an isomorphism. *Cartan types* are a standard set of names for those Dynkin diagrams (see Wikipedia article Dynkin_diagram).

Let us consider, for example, the Cartan type A_4:

```
sage: T = CartanType(['A', 4]); T
['A', 4]
```

```
>>> from sage.all import *
>>> T = CartanType(['A', Integer(4)]); T
['A', 4]
```

It is the name of the following Dynkin diagram:

```
sage: DynkinDiagram(T)                                                    
↪  # needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

```
>>> from sage.all import *
>>> DynkinDiagram(T)
↪# needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

Note

For convenience, the following shortcuts are available:

```
sage: DynkinDiagram(['A',4])
↪ # needs sage.graphs
O---O---O---O
1   2   3   4
A4
sage: DynkinDiagram('A4')
↪ # needs sage.graphs
O---O---O---O
1   2   3   4
A4
sage: T.dynkin_diagram()
↪ # needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

```
>>> from sage.all import *
>>> DynkinDiagram(['A',Integer(4)])
↪         # needs sage.graphs
O---O---O---O
1   2   3   4
A4
>>> DynkinDiagram('A4')
↪# needs sage.graphs
O---O---O---O
1   2   3   4
A4
>>> T.dynkin_diagram()
↪# needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

See *DynkinDiagram* for how to further manipulate Dynkin diagrams.

From this data (the *Cartan datum*), one can construct the associated root system:

```
sage: RootSystem(T)
Root system of type ['A', 4]
```

```
>>> from sage.all import *
>>> RootSystem(T)
Root system of type ['A', 4]
```

The associated Weyl group of A_n is the symmetric group S_{n+1}:

```
sage: W = WeylGroup(T); W
↪  # needs sage.libs.gap
Weyl Group of type ['A', 4] (as a matrix group acting on the ambient space)
```

(continues on next page)

(continued from previous page)

```
sage: W.cardinality()                                                          ␣
↪   # needs sage.libs.gap
120
```

```
>>> from sage.all import *
>>> W = WeylGroup(T); W
↪# needs sage.libs.gap
Weyl Group of type ['A', 4] (as a matrix group acting on the ambient space)
>>> W.cardinality()
↪# needs sage.libs.gap
120
```

while the Lie algebra is sl_{n+1}, and the Lie group SL_{n+1} (TODO: illustrate this once this is implemented).

One may also construct crystals associated to various Dynkin diagrams. For example:

```
sage: C = crystals.Letters(T); C                                               ␣
↪   # needs sage.combinat
The crystal of letters for type ['A', 4]
sage: C.list()                                                                 ␣
↪   # needs sage.combinat
[1, 2, 3, 4, 5]

sage: C = crystals.Tableaux(T, shape=[2]); C                                   ␣
↪   # needs sage.combinat
The crystal of tableaux of type ['A', 4] and shape(s) [[2]]
sage: C.cardinality()                                                          ␣
↪   # needs sage.combinat
15
```

```
>>> from sage.all import *
>>> C = crystals.Letters(T); C
↪# needs sage.combinat
The crystal of letters for type ['A', 4]
>>> C.list()
↪# needs sage.combinat
[1, 2, 3, 4, 5]

>>> C = crystals.Tableaux(T, shape=[Integer(2)]); C                            ␣
↪          # needs sage.combinat
The crystal of tableaux of type ['A', 4] and shape(s) [[2]]
>>> C.cardinality()
↪# needs sage.combinat
15
```

Here is a sample of all the finite irreducible crystallographic Cartan types:

```
sage: CartanType.samples(finite=True, crystallographic=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], [
↪'D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2]]
```

```
>>> from sage.all import *
>>> CartanType.samples(finite=True, crystallographic=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], [
↪'D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2]]
```

One can also get latex representations of the crystallographic Cartan types and their corresponding Dynkin diagrams:

```
sage: [latex(ct) for ct in CartanType.samples(crystallographic=True)]
[A_{1}, A_{5}, B_{1}, B_{5}, C_{1}, C_{5}, D_{2}, D_{3}, D_{5},
 E_6, E_7, E_8, F_4, G_2,
 A_{1}^{(1)}, A_{5}^{(1)}, B_{1}^{(1)}, B_{5}^{(1)},
 C_{1}^{(1)}, C_{5}^{(1)}, D_{3}^{(1)}, D_{5}^{(1)},
 E_6^{(1)}, E_7^{(1)}, E_8^{(1)}, F_4^{(1)}, G_2^{(1)},
 BC_{1}^{(2)}, BC_{5}^{(2)},
 B_{5}^{(1)\vee}, C_{4}^{(1)\vee}, F_4^{(1)\vee},
 G_2^{(1)\vee}, BC_{1}^{(2)\vee}, BC_{5}^{(2)\vee}]
sage: view([DynkinDiagram(ct) for ct in CartanType.
↪samples(crystallographic=True)]) # not tested
```

```
>>> from sage.all import *
>>> [latex(ct) for ct in CartanType.samples(crystallographic=True)]
[A_{1}, A_{5}, B_{1}, B_{5}, C_{1}, C_{5}, D_{2}, D_{3}, D_{5},
 E_6, E_7, E_8, F_4, G_2,
 A_{1}^{(1)}, A_{5}^{(1)}, B_{1}^{(1)}, B_{5}^{(1)},
 C_{1}^{(1)}, C_{5}^{(1)}, D_{3}^{(1)}, D_{5}^{(1)},
 E_6^{(1)}, E_7^{(1)}, E_8^{(1)}, F_4^{(1)}, G_2^{(1)},
 BC_{1}^{(2)}, BC_{5}^{(2)},
 B_{5}^{(1)\vee}, C_{4}^{(1)\vee}, F_4^{(1)\vee},
 G_2^{(1)\vee}, BC_{1}^{(2)\vee}, BC_{5}^{(2)\vee}]
>>> view([DynkinDiagram(ct) for ct in CartanType.samples(crystallographic=True)])
↪# not tested
```

Non-crystallographic Cartan types are also partially supported:

```
sage: CartanType.samples(finite=True, crystallographic=False)
[['I', 5], ['H', 3], ['H', 4]]
```

```
>>> from sage.all import *
>>> CartanType.samples(finite=True, crystallographic=False)
[['I', 5], ['H', 3], ['H', 4]]
```

In Sage, a Cartan type is used as a database of type-specific information and algorithms (see e.g. *sage.combinat.root_system.type_A*). This database includes how to construct the Dynkin diagram, the ambient space for the root system (see Wikipedia article Root_system), and further mathematical properties:

```
sage: T.is_finite(), T.is_simply_laced(), T.is_affine(), T.is_crystallographic()
(True, True, False, True)
```

```
>>> from sage.all import *
>>> T.is_finite(), T.is_simply_laced(), T.is_affine(), T.is_crystallographic()
(True, True, False, True)
```

In particular, a Sage Cartan type is endowed with a fixed choice of labels for the nodes of the Dynkin diagram. This choice follows the conventions of Nicolas Bourbaki, Lie Groups and Lie Algebras: Chapter 4-6, Elements of Mathematics, Springer (2002). ISBN 978-3540426509. For example:

```
sage: T = CartanType(['D', 4])
sage: DynkinDiagram(T)
↪ # needs sage.graphs
    O 4
    |
    |
O---O---O
1   2   3
D4

sage: E6 = CartanType(['E',6])
sage: DynkinDiagram(E6)
↪ # needs sage.graphs
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
```

```
>>> from sage.all import *
>>> T = CartanType(['D', Integer(4)])
>>> DynkinDiagram(T)
↪# needs sage.graphs
    O 4
    |
    |
O---O---O
1   2   3
D4

>>> E6 = CartanType(['E',Integer(6)])
>>> DynkinDiagram(E6)
↪# needs sage.graphs
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
```

Note

The direction of the arrows is the **opposite** (i.e. the transpose) of Bourbaki's convention, but agrees with Kac's.

For example, in type C_2, we have:

```
sage: C2 = DynkinDiagram(['C',2]); C2
↪ # needs sage.graphs
O=<=O
```

```
1   2
C2
sage: C2.cartan_matrix()                                                    ␣
↪ # needs sage.graphs
[ 2 -2]
[-1  2]
```

```
>>> from sage.all import *
>>> C2 = DynkinDiagram(['C',Integer(2)]); C2                                ␣
↪         # needs sage.graphs
O=<=O
1   2
C2
>>> C2.cartan_matrix()
↪# needs sage.graphs
[ 2 -2]
[-1  2]
```

However Bourbaki would have the Cartan matrix as:

$$\begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}.$$

If desired, other node labelling conventions can be achieved. For example the Kac labelling for type E_6 can be obtained via:

```
sage: E6.relabel({1:1,2:6,3:2,4:3,5:4,6:5}).dynkin_diagram()                ␣
↪ # needs sage.graphs
        O 6
        |
        |
O---O---O---O---O
1   2   3   4   5
E6 relabelled by {1: 1, 2: 6, 3: 2, 4: 3, 5: 4, 6: 5}
```

```
>>> from sage.all import *
>>> E6.relabel({Integer(1):Integer(1),Integer(2):Integer(6),Integer(3):Integer(2),
↪Integer(4):Integer(3),Integer(5):Integer(4),Integer(6):Integer(5)}).dynkin_
↪diagram()                        # needs sage.graphs
        O 6
        |
        |
O---O---O---O---O
1   2   3   4   5
E6 relabelled by {1: 1, 2: 6, 3: 2, 4: 3, 5: 4, 6: 5}
```

Contributions implementing other conventions are very welcome.

Another option is to build from scratch a new Dynkin diagram. The architecture has been designed to make it fairly easy to add other labelling conventions. In particular, we strived at choosing type free algorithms whenever possible, so in principle most features should remain available even with custom Cartan types. This has not been used much yet, so some rough corners certainly remain.

Here, we construct the hyperbolic example of Exercise 4.9 p. 57 of Kac, Infinite Dimensional Lie Algebras. We start with an empty Dynkin diagram, and add a couple nodes:

```
sage: g = DynkinDiagram()
↪ # needs sage.graphs
sage: g.add_vertices([1,2,3])
↪ # needs sage.graphs
```

```
>>> from sage.all import *
>>> g = DynkinDiagram()
↪# needs sage.graphs
>>> g.add_vertices([Integer(1),Integer(2),Integer(3)])
↪                         # needs sage.graphs
```

Note that the diagonal of the Cartan matrix is already initialized:

```
sage: g.cartan_matrix()
↪ # needs sage.graphs
[2 0 0]
[0 2 0]
[0 0 2]
```

```
>>> from sage.all import *
>>> g.cartan_matrix()
↪# needs sage.graphs
[2 0 0]
[0 2 0]
[0 0 2]
```

Then we add a couple edges:

```
sage: g.add_edge(1,2,2)
↪ # needs sage.graphs
sage: g.add_edge(1,3)
↪ # needs sage.graphs
sage: g.add_edge(2,3)
↪ # needs sage.graphs
```

```
>>> from sage.all import *
>>> g.add_edge(Integer(1),Integer(2),Integer(2))
↪                         # needs sage.graphs
>>> g.add_edge(Integer(1),Integer(3))
↪                    # needs sage.graphs
>>> g.add_edge(Integer(2),Integer(3))
↪                    # needs sage.graphs
```

and we get the desired Cartan matrix:

```
sage: g.cartan_matrix()
↪ # needs sage.graphs
[2 0 0]
[0 2 0]
[0 0 2]
```

```
>>> from sage.all import *
>>> g.cartan_matrix()
↪# needs sage.graphs
[2 0 0]
[0 2 0]
[0 0 2]
```

Oops, the Cartan matrix did not change! This is because it is cached for efficiency (see `cached_method`). In general, a Dynkin diagram should not be modified after having been used.

Warning: this is not checked currently

Todo: add a method `set_mutable()` as, say, for matrices

Here, we can work around this by clearing the cache:

```
sage: delattr(g, 'cartan_matrix')
↪    # needs sage.graphs
```

```
>>> from sage.all import *
>>> delattr(g, 'cartan_matrix')
↪# needs sage.graphs
```

Now we get the desired Cartan matrix:

```
sage: g.cartan_matrix()
↪    # needs sage.graphs
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

```
>>> from sage.all import *
>>> g.cartan_matrix()
↪# needs sage.graphs
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

Note that backward edges have been automatically added:

```
sage: g.edges(sort=True)
↪    # needs sage.graphs
[(1, 2, 2), (1, 3, 1), (2, 1, 1), (2, 3, 1), (3, 1, 1), (3, 2, 1)]
```

```
>>> from sage.all import *
>>> g.edges(sort=True)
```

(continues on next page)

(continued from previous page)

```
↪# needs sage.graphs
[(1, 2, 2), (1, 3, 1), (2, 1, 1), (2, 3, 1), (3, 1, 1), (3, 2, 1)]
```

Reducible Cartan types

Reducible Cartan types can be specified by passing a sequence or list of irreducible Cartan types:

```
sage: CartanType(['A',2],['B',2])
A2xB2
sage: CartanType([['A',2],['B',2]])
A2xB2
sage: CartanType(['A',2],['B',2]).is_reducible()
True
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(2)],['B',Integer(2)])
A2xB2
>>> CartanType([['A',Integer(2)],['B',Integer(2)]])
A2xB2
>>> CartanType(['A',Integer(2)],['B',Integer(2)]).is_reducible()
True
```

or using the following short hand notation:

```
sage: CartanType("A2xB2")
A2xB2
sage: CartanType("A2","B2") == CartanType("A2xB2")
True
```

```
>>> from sage.all import *
>>> CartanType("A2xB2")
A2xB2
>>> CartanType("A2","B2") == CartanType("A2xB2")
True
```

Degenerate cases

When possible, type I_n is automatically converted to the isomorphic crystallographic Cartan types (any reason not to do so?):

```
sage: CartanType(["I",1])
A1xA1
sage: CartanType(["I",3])
['A', 2]
sage: CartanType(["I",4])
['C', 2]
sage: CartanType(["I",6])
['G', 2]
```

```
>>> from sage.all import *
>>> CartanType(["I",Integer(1)])
```

(continues on next page)

(continued from previous page)

```
A1xA1
>>> CartanType(["I",Integer(3)])
['A', 2]
>>> CartanType(["I",Integer(4)])
['C', 2]
>>> CartanType(["I",Integer(6)])
['G', 2]
```

The Dynkin diagrams for types B_1, C_1, D_2, and D_3 are isomorphic to that for A_1, A_1, $A_1 \times A_1$, and A_3, respectively. However their natural ambient space realizations (stemming from the corresponding infinite families of Lie groups) are different. Therefore, the Cartan types are considered as distinct:

```
sage: CartanType(['B',1])
['B', 1]
sage: CartanType(['C',1])
['C', 1]
sage: CartanType(['D',2])
['D', 2]
sage: CartanType(['D',3])
['D', 3]
```

```
>>> from sage.all import *
>>> CartanType(['B',Integer(1)])
['B', 1]
>>> CartanType(['C',Integer(1)])
['C', 1]
>>> CartanType(['D',Integer(2)])
['D', 2]
>>> CartanType(['D',Integer(3)])
['D', 3]
```

Affine Cartan types

For affine types, we use the usual conventions for affine Coxeter groups: each affine type is either untwisted (that is arise from the natural affinisation of a finite Cartan type):

```
sage: CartanType(["A", 4, 1]).dynkin_diagram()
↪ # needs sage.graphs
0
O-----------+
|           |
|           |
O---O---O---O
1   2   3   4
A4~
sage: CartanType(["B", 4, 1]).dynkin_diagram()
↪ # needs sage.graphs
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
```

(continues on next page)

(continued from previous page)

```
B4~
```

```
>>> from sage.all import *
>>> CartanType(["A", Integer(4), Integer(1)]).dynkin_diagram()                    # needs sage.graphs
0
O-----------+
|           |
|           |
O---O---O---O
1   2   3   4
A4~
>>> CartanType(["B", Integer(4), Integer(1)]).dynkin_diagram()                    # needs sage.graphs
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
B4~
```

or dual thereof:

```
sage: CartanType(["B", 4, 1]).dual().dynkin_diagram()                             # needs sage.graphs
    O 0
    |
    |
O---O---O=<=O
1   2   3   4
B4~*
```

```
>>> from sage.all import *
>>> CartanType(["B", Integer(4), Integer(1)]).dual().dynkin_diagram()             # needs sage.graphs
    O 0
    |
    |
O---O---O=<=O
1   2   3   4
B4~*
```

or is of type $\widetilde{BC}_n$ (which yields an irreducible, but nonreduced root system):

```
sage: CartanType(["BC", 4, 2]).dynkin_diagram()                                   # needs sage.graphs
O=<=O---O---O=<=O
0   1   2   3   4
BC4~
```

```
>>> from sage.all import *
>>> CartanType(["BC", Integer(4), Integer(2)]).dynkin_diagram()
```

(continues on next page)

(continued from previous page)

```
↪                   # needs sage.graphs
O=<=O---O---O=<=O
0   1   2   3   4
BC4~
```

This includes the two degenerate cases:

```
sage: CartanType(["A", 1, 1]).dynkin_diagram()                              ␣
↪ # needs sage.graphs
O<=>O
0   1
A1~
sage: CartanType(["BC", 1, 2]).dynkin_diagram()                             ␣
↪ # needs sage.graphs
  4
O=<=O
0   1
BC1~
```

```
>>> from sage.all import *
>>> CartanType(["A", Integer(1), Integer(1)]).dynkin_diagram()              ␣
↪                   # needs sage.graphs
O<=>O
0   1
A1~
>>> CartanType(["BC", Integer(1), Integer(2)]).dynkin_diagram()             ␣
↪                   # needs sage.graphs
  4
O=<=O
0   1
BC1~
```

For the user convenience, Kac's notations for twisted affine types are automatically translated into the previous ones:

```
sage: # needs sage.graphs
sage: CartanType(["A", 9, 2])
['B', 5, 1]^*
sage: CartanType(["A", 9, 2]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
B5~*
sage: CartanType(["A", 10, 2]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
BC5~
sage: CartanType(["D", 5, 2]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
```

(continues on next page)

(continued from previous page)

```
C4~*
sage: CartanType(["D", 4, 3]).dynkin_diagram()
  3
O=>=O---O
2   1   0
G2~* relabelled by {0: 0, 1: 2, 2: 1}
sage: CartanType(["E", 6, 2]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
F4~*
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CartanType(["A", Integer(9), Integer(2)])
['B', 5, 1]^*
>>> CartanType(["A", Integer(9), Integer(2)]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
B5~*
>>> CartanType(["A", Integer(10), Integer(2)]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
BC5~
>>> CartanType(["D", Integer(5), Integer(2)]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~*
>>> CartanType(["D", Integer(4), Integer(3)]).dynkin_diagram()
  3
O=>=O---O
2   1   0
G2~* relabelled by {0: 0, 1: 2, 2: 1}
>>> CartanType(["E", Integer(6), Integer(2)]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
F4~*
```

Additionally one can set the notation option to use Kac's notation:

```
sage: # needs sage.graphs
sage: CartanType.options['notation'] = 'Kac'
sage: CartanType(["A", 9, 2])
['A', 9, 2]
sage: CartanType(["A", 9, 2]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
```

(continues on next page)

(continued from previous page)

```
A9^2
sage: CartanType(["A", 10, 2]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
A10^2
sage: CartanType(["D", 5, 2]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
D5^2
sage: CartanType(["D", 4, 3]).dynkin_diagram()
  3
O=>=O---O
2   1   0
D4^3
sage: CartanType(["E", 6, 2]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
E6^2
sage: CartanType.options['notation'] = 'BC'
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CartanType.options['notation'] = 'Kac'
>>> CartanType(["A", Integer(9), Integer(2)])
['A', 9, 2]
>>> CartanType(["A", Integer(9), Integer(2)]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
A9^2
>>> CartanType(["A", Integer(10), Integer(2)]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
A10^2
>>> CartanType(["D", Integer(5), Integer(2)]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
D5^2
>>> CartanType(["D", Integer(4), Integer(3)]).dynkin_diagram()
  3
O=>=O---O
2   1   0
D4^3
>>> CartanType(["E", Integer(6), Integer(2)]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
E6^2
>>> CartanType.options['notation'] = 'BC'
```

Infinite Cartan types

There are minimal implementations of the Cartan types A_∞ and $A_{+\infty}$. In sage oo is the same as $+Infinity$, so NN and ZZ are used to differentiate between the $A_{+\infty}$ and A_∞ root systems:

```
sage: CartanType(['A', NN])
['A', NN]
sage: print(CartanType(['A', NN]).ascii_art())
O---O---O---O---O---O---O---..
0   1   2   3   4   5   6
sage: CartanType(['A', ZZ])
['A', ZZ]
sage: print(CartanType(['A', ZZ]).ascii_art())
..---O---O---O---O---O---O---O---..
    -3  -2  -1   0   1   2   3
```

```
>>> from sage.all import *
>>> CartanType(['A', NN])
['A', NN]
>>> print(CartanType(['A', NN]).ascii_art())
O---O---O---O---O---O---O---..
0   1   2   3   4   5   6
>>> CartanType(['A', ZZ])
['A', ZZ]
>>> print(CartanType(['A', ZZ]).ascii_art())
..---O---O---O---O---O---O---O---..
    -3  -2  -1   0   1   2   3
```

There are also the following shorthands:

```
sage: CartanType("Aoo")
['A', ZZ]
sage: CartanType("A+oo")
['A', NN]
```

```
>>> from sage.all import *
>>> CartanType("Aoo")
['A', ZZ]
>>> CartanType("A+oo")
['A', NN]
```

Abstract classes for Cartan types

- *CartanType_abstract*
- *CartanType_crystallographic*
- *CartanType_simply_laced*
- *CartanType_simple*
- *CartanType_finite*
- *CartanType_affine* (see also *Root system data for affine Cartan types*)
- *sage.combinat.root_system.cartan_type.CartanType*
- *Root system data for dual Cartan types*

- *Root system data for reducible Cartan types*
- *Root system data for relabelled Cartan types*

Concrete classes for Cartan types

- *CartanType_standard*
- *CartanType_standard_finite*
- *CartanType_standard_affine*
- *CartanType_standard_untwisted_affine*

Type specific data

The data essentially consists of a description of the Dynkin/Coxeter diagram and, when relevant, of the natural embedding of the root system in a Euclidean space. Everything else is reconstructed from this data.

- *Root system data for type A*
- *Root system data for type B*
- *Root system data for type C*
- *Root system data for type D*
- *Root system data for type E*
- *Root system data for type F*
- *Root system data for type G*
- *Root system data for type H*
- *Root system data for type I*
- *Root system data for super type A*
- *Root system data for type Q*
- *Root system data for (untwisted) type A affine*
- *Root system data for (untwisted) type B affine*
- *Root system data for (untwisted) type C affine*
- *Root system data for (untwisted) type D affine*
- *Root system data for (untwisted) type E affine*
- *Root system data for (untwisted) type F affine*
- *Root system data for (untwisted) type G affine*
- *Root system data for type BC affine*
- *Root system data for type A infinity*

Todo

Should those indexes come before the introduction?

class sage.combinat.root_system.cartan_type.**CartanTypeFactory**

Bases: SageObject

classmethod color(*i*)

Default color scheme for the vertices of a Dynkin diagram (and associated objects).

EXAMPLES:

```
sage: CartanType.color(1)
'blue'
sage: CartanType.color(2)
'red'
sage: CartanType.color(3)
'green'
```

```
>>> from sage.all import *
>>> CartanType.color(Integer(1))
'blue'
>>> CartanType.color(Integer(2))
'red'
>>> CartanType.color(Integer(3))
'green'
```

The default color is black:

```
sage: CartanType.color(0)
'black'
```

```
>>> from sage.all import *
>>> CartanType.color(Integer(0))
'black'
```

Negative indices get the same color as their positive counterparts:

```
sage: CartanType.color(-1)
'blue'
sage: CartanType.color(-2)
'red'
sage: CartanType.color(-3)
'green'
```

```
>>> from sage.all import *
>>> CartanType.color(-Integer(1))
'blue'
>>> CartanType.color(-Integer(2))
'red'
>>> CartanType.color(-Integer(3))
'green'
```

options = Current options for CartanType - dual_latex: \vee - dual_str: * - latex_marked: True - latex_relabel: True - mark_special_node: none - marked_node_str: X - notation: Stembridge - special_node_str: @

samples(*finite=None*, *affine=None*, *crystallographic=None*)

Return a sample of the available Cartan types.

INPUT:

- `finite` – boolean or `None` (default: `None`)
- `affine` – boolean or `None` (default: `None`)
- `crystallographic` – boolean or `None` (default: `None`)

The sample contains all the exceptional finite and affine Cartan types, as well as typical representatives of the infinite families.

EXAMPLES:

```
sage: CartanType.samples()
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D',␣
→3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2], ['I', 5], ['H', 3], ['H',␣
→4],
 ['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1,␣
→2], ['BC', 5, 2],
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, [
→'BC', 5, 2]^*]
```

```
>>> from sage.all import *
>>> CartanType.samples()
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D',␣
→3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2], ['I', 5], ['H', 3], ['H',␣
→4],
 ['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1,␣
→2], ['BC', 5, 2],
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, [
→'BC', 5, 2]^*]
```

The finite, affine and crystallographic options allow respectively for restricting to (non) finite, (non) affine, and (non) crystallographic Cartan types:

```
sage: CartanType.samples(finite=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D',␣
→3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2], ['I', 5], ['H', 3], ['H',␣
→4]]

sage: CartanType.samples(affine=True)
[['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1,␣
→2], ['BC', 5, 2],
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, [
→'BC', 5, 2]^*]

sage: CartanType.samples(crystallographic=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D',␣
```

(continues on next page)

(continued from previous page)

```
↪3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2],
 ['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1,␣
↪2], ['BC', 5, 2],
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, [
↪'BC', 5, 2]^*]

sage: CartanType.samples(crystallographic=False)
[['I', 5], ['H', 3], ['H', 4]]
```

```
>>> from sage.all import *
>>> CartanType.samples(finite=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D',␣
↪3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2], ['I', 5], ['H', 3], ['H',␣
↪4]]

>>> CartanType.samples(affine=True)
[['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1,␣
↪2], ['BC', 5, 2],
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, [
↪'BC', 5, 2]^*]

>>> CartanType.samples(crystallographic=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D',␣
↪3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2],
 ['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1,␣
↪2], ['BC', 5, 2],
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, [
↪'BC', 5, 2]^*]

>>> CartanType.samples(crystallographic=False)
[['I', 5], ['H', 3], ['H', 4]]
```

Todo

add some reducible Cartan types (suggestions?)

class `sage.combinat.root_system.cartan_type.`**`CartanType_abstract`**

Bases: `object`

Abstract class for Cartan types

Subclasses should implement:

- dynkin_diagram()
- cartan_matrix()
- *is_finite()*
- *is_affine()*
- *is_irreducible()*

as_folding(*folding_of=None*, *sigma=None*)

Return self realized as a folded Cartan type.

For finite and affine types, this is realized by the Dynkin diagram foldings:

$$\begin{array}{ccl} C_n^{(1)}, A_{2n}^{(2)}, A_{2n}^{(2)\dagger}, D_{n+1}^{(2)} & \hookrightarrow & A_{2n-1}^{(1)}, \\ A_{2n-1}^{(2)}, B_n^{(1)} & \hookrightarrow & D_{n+1}^{(1)}, \\ E_6^{(2)}, F_4^{(1)} & \hookrightarrow & E_6^{(1)}, \\ D_4^{(3)}, G_2^{(1)} & \hookrightarrow & D_4^{(1)}, \\ C_n & \hookrightarrow & A_{2n-1}, \\ B_n & \hookrightarrow & D_{n+1}, \\ F_4 & \hookrightarrow & E_6, \\ G_2 & \hookrightarrow & D_4. \end{array}$$

For general types, this returns self as a folded type of self with σ as the identity map.

For more information on these foldings and folded Cartan types, see *sage.combinat.root_system.type_folded.CartanTypeFolded*.

If the optional inputs folding_of and sigma are specified, then this returns the folded Cartan type of self in folding_of given by the automorphism sigma.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).as_folding()
['B', 3, 1] as a folding of  ['D', 4, 1]
sage: CartanType(['F', 4]).as_folding()
['F', 4] as a folding of  ['E', 6]
sage: CartanType(['BC', 3, 2]).as_folding()
['BC', 3, 2] as a folding of  ['A', 5, 1]
sage: CartanType(['D', 4, 3]).as_folding()
['G', 2, 1]^* relabelled by {0: 0, 1: 2, 2: 1} as a folding of ['D', 4, 1]
```

```
>>> from sage.all import *
>>> CartanType(['B', Integer(3), Integer(1)]).as_folding()
['B', 3, 1] as a folding of  ['D', 4, 1]
>>> CartanType(['F', Integer(4)]).as_folding()
['F', 4] as a folding of  ['E', 6]
>>> CartanType(['BC', Integer(3), Integer(2)]).as_folding()
['BC', 3, 2] as a folding of  ['A', 5, 1]
>>> CartanType(['D', Integer(4), Integer(3)]).as_folding()
['G', 2, 1]^* relabelled by {0: 0, 1: 2, 2: 1} as a folding of ['D', 4, 1]
```

coxeter_diagram()

Return the Coxeter diagram for self.

EXAMPLES:

```
sage: # needs sage.graphs
sage: CartanType(['B',3]).coxeter_diagram()
Graph on 3 vertices
sage: CartanType(['A',3]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 3)]
sage: CartanType(['B',3]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 4)]
sage: CartanType(['G',2]).coxeter_diagram().edges(sort=True)
[(1, 2, 6)]
sage: CartanType(['F',4]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 4), (3, 4, 3)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CartanType(['B',Integer(3)]).coxeter_diagram()
Graph on 3 vertices
>>> CartanType(['A',Integer(3)]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 3)]
>>> CartanType(['B',Integer(3)]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 4)]
>>> CartanType(['G',Integer(2)]).coxeter_diagram().edges(sort=True)
[(1, 2, 6)]
>>> CartanType(['F',Integer(4)]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 4), (3, 4, 3)]
```

coxeter_matrix()

Return the Coxeter matrix for `self`.

EXAMPLES:

```
sage: CartanType(['A', 4]).coxeter_matrix()                                    #␣
↪needs sage.graphs
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(4)]).coxeter_matrix()                            ␣
↪      # needs sage.graphs
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
```

coxeter_type()

Return the Coxeter type for `self`.

EXAMPLES:

```
sage: CartanType(['A', 4]).coxeter_type()
Coxeter type of ['A', 4]
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(4)]).coxeter_type()
Coxeter type of ['A', 4]
```

dual()

Return the dual Cartan type, possibly just as a formal dual.

EXAMPLES:

```
sage: CartanType(['A',3]).dual()
['A', 3]
sage: CartanType(["B", 3]).dual()
['C', 3]
sage: CartanType(['C',2]).dual()
['B', 2]
sage: CartanType(['D',4]).dual()
['D', 4]
sage: CartanType(['E',8]).dual()
['E', 8]
sage: CartanType(['F',4]).dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(3)]).dual()
['A', 3]
>>> CartanType(["B", Integer(3)]).dual()
['C', 3]
>>> CartanType(['C',Integer(2)]).dual()
['B', 2]
>>> CartanType(['D',Integer(4)]).dual()
['D', 4]
>>> CartanType(['E',Integer(8)]).dual()
['E', 8]
>>> CartanType(['F',Integer(4)]).dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

index_set()

Return the index set for `self`.

This is the list of the nodes of the associated Coxeter or Dynkin diagram.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).index_set()
(0, 1, 2, 3)
sage: CartanType(['D', 4]).index_set()
(1, 2, 3, 4)
sage: CartanType(['A', 7, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CartanType(['A', 7, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CartanType(['A', 6, 2]).index_set()
(0, 1, 2, 3)
sage: CartanType(['D', 6, 2]).index_set()
```

(continues on next page)

(continued from previous page)

```
(0, 1, 2, 3, 4, 5)
sage: CartanType(['E', 6, 1]).index_set()
(0, 1, 2, 3, 4, 5, 6)
sage: CartanType(['E', 6, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CartanType(['A', 2, 2]).index_set()
(0, 1)
sage: CartanType(['G', 2, 1]).index_set()
(0, 1, 2)
sage: CartanType(['F', 4, 1]).index_set()
(0, 1, 2, 3, 4)
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).index_set()
(0, 1, 2, 3)
>>> CartanType(['D', Integer(4)]).index_set()
(1, 2, 3, 4)
>>> CartanType(['A', Integer(7), Integer(2)]).index_set()
(0, 1, 2, 3, 4)
>>> CartanType(['A', Integer(7), Integer(2)]).index_set()
(0, 1, 2, 3, 4)
>>> CartanType(['A', Integer(6), Integer(2)]).index_set()
(0, 1, 2, 3)
>>> CartanType(['D', Integer(6), Integer(2)]).index_set()
(0, 1, 2, 3, 4, 5)
>>> CartanType(['E', Integer(6), Integer(1)]).index_set()
(0, 1, 2, 3, 4, 5, 6)
>>> CartanType(['E', Integer(6), Integer(2)]).index_set()
(0, 1, 2, 3, 4)
>>> CartanType(['A', Integer(2), Integer(2)]).index_set()
(0, 1)
>>> CartanType(['G', Integer(2), Integer(1)]).index_set()
(0, 1, 2)
>>> CartanType(['F', Integer(4), Integer(1)]).index_set()
(0, 1, 2, 3, 4)
```

is_affine()

Return whether `self` is affine.

EXAMPLES:

```
sage: CartanType(['A', 3]).is_affine()
False
sage: CartanType(['A', 3, 1]).is_affine()
True
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3)]).is_affine()
False
>>> CartanType(['A', Integer(3), Integer(1)]).is_affine()
True
```

is_atomic()

This method is usually equivalent to *is_reducible()*, except for the Cartan type D_2.

D_2 is not a standard Cartan type. It is equivalent to type $A_1 \times A_1$ which is reducible; however the isomorphism from its ambient space (for the orthogonal group of degree 4) to that of $A_1 \times A_1$ is non trivial, and it is useful to have it.

From a programming point of view its implementation is more similar to the irreducible types, and so the method *is_atomic()* is supplied.

EXAMPLES:

```
sage: CartanType("D2").is_atomic()
True
sage: CartanType("D2").is_irreducible()
False
```

```
>>> from sage.all import *
>>> CartanType("D2").is_atomic()
True
>>> CartanType("D2").is_irreducible()
False
```

is_compound()

A short hand for not *is_atomic()*.

is_crystallographic()

Return whether this Cartan type is crystallographic.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: [ [t, t.is_crystallographic() ] for t in CartanType.
↪samples(finite=True) ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], True],
 [['C', 1], True], [['C', 5], True],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], True], [['G', 2], True],
 [['I', 5], False], [['H', 3], False], [['H', 4], False]]
```

```
>>> from sage.all import *
>>> [ [t, t.is_crystallographic() ] for t in CartanType.samples(finite=True) ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], True],
 [['C', 1], True], [['C', 5], True],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], True], [['G', 2], True],
 [['I', 5], False], [['H', 3], False], [['H', 4], False]]
```

is_finite()

Return whether this Cartan type is finite.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
sage: C.is_finite()
Traceback (most recent call last):
...
NotImplementedError: <abstract method is_finite at ...>
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.cartan_type import CartanType_abstract
>>> C = CartanType_abstract()
>>> C.is_finite()
Traceback (most recent call last):
...
NotImplementedError: <abstract method is_finite at ...>
```

```
sage: CartanType(['A',4]).is_finite()
True
sage: CartanType(['A',4, 1]).is_finite()
False
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).is_finite()
True
>>> CartanType(['A',Integer(4), Integer(1)]).is_finite()
False
```

is_implemented()

Check whether the Cartan datum for `self` is actually implemented.

EXAMPLES:

```
sage: CartanType(["A",4,1]).is_implemented()                                   #␣
↪needs sage.graphs
True
sage: CartanType(['H',3]).is_implemented()
True
```

```
>>> from sage.all import *
>>> CartanType(["A",Integer(4),Integer(1)]).is_implemented()                   ␣
↪               # needs sage.graphs
True
>>> CartanType(['H',Integer(3)]).is_implemented()
True
```

is_irreducible()

Report whether this Cartan type is irreducible (i.e. simple). This should be overridden in any subclass.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
```

(continues on next page)

(continued from previous page)

```
sage: C.is_irreducible()
False
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.cartan_type import CartanType_abstract
>>> C = CartanType_abstract()
>>> C.is_irreducible()
False
```

is_reducible()

Report whether the root system is reducible (i.e. not simple), that is whether it can be factored as a product of root systems.

EXAMPLES:

```
sage: CartanType("A2xB3").is_reducible()
True
sage: CartanType(['A',2]).is_reducible()
False
```

```
>>> from sage.all import *
>>> CartanType("A2xB3").is_reducible()
True
>>> CartanType(['A',Integer(2)]).is_reducible()
False
```

is_simply_laced()

Return whether this Cartan type is simply laced.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: [ [t, t.is_simply_laced() ] for t in CartanType.samples() ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], False],
 [['C', 1], True], [['C', 5], False],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], False], [['G', 2], False], [['I', 5], False],
 [['H', 3], False], [['H', 4], False],
 [['A', 1, 1], False], [['A', 5, 1], True],
 [['B', 1, 1], False], [['B', 5, 1], False],
 [['C', 1, 1], False], [['C', 5, 1], False],
 [['D', 3, 1], True], [['D', 5, 1], True],
 [['E', 6, 1], True], [['E', 7, 1], True], [['E', 8, 1], True],
 [['F', 4, 1], False], [['G', 2, 1], False],
 [['BC', 1, 2], False], [['BC', 5, 2], False],
 [['B', 5, 1]^*, False], [['C', 4, 1]^*, False],
 [['F', 4, 1]^*, False], [['G', 2, 1]^*, False],
 [['BC', 1, 2]^*, False], [['BC', 5, 2]^*, False]]
```

```
>>> from sage.all import *
>>> [ [t, t.is_simply_laced() ] for t in CartanType.samples() ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], False],
 [['C', 1], True], [['C', 5], False],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], False], [['G', 2], False], [['I', 5], False],
 [['H', 3], False], [['H', 4], False],
 [['A', 1, 1], False], [['A', 5, 1], True],
 [['B', 1, 1], False], [['B', 5, 1], False],
 [['C', 1, 1], False], [['C', 5, 1], False],
 [['D', 3, 1], True], [['D', 5, 1], True],
 [['E', 6, 1], True], [['E', 7, 1], True], [['E', 8, 1], True],
 [['F', 4, 1], False], [['G', 2, 1], False],
 [['BC', 1, 2], False], [['BC', 5, 2], False],
 [['B', 5, 1]^*, False], [['C', 4, 1]^*, False],
 [['F', 4, 1]^*, False], [['G', 2, 1]^*, False],
 [['BC', 1, 2]^*, False], [['BC', 5, 2]^*, False]]
```

marked_nodes(*marked_nodes*)

Return a Cartan type with the nodes `marked_nodes` marked.

INPUT:

- `marked_nodes` – list of nodes to mark

EXAMPLES:

```
sage: CartanType(['F',4]).marked_nodes([1, 3]).dynkin_diagram()                 #␣
↪needs sage.graphs
X---O=>=X---O
1   2   3   4
F4 with nodes (1, 3) marked
```

```
>>> from sage.all import *
>>> CartanType(['F',Integer(4)]).marked_nodes([Integer(1), Integer(3)]).
↪dynkin_diagram()                # needs sage.graphs
X---O=>=X---O
1   2   3   4
F4 with nodes (1, 3) marked
```

options = Current options for CartanType - dual_latex: \vee - dual_str: * - latex_marked: True - latex_relabel: True - mark_special_node: none - marked_node_str: X - notation: Stembridge - special_node_str: @

rank()

Return the rank of `self`.

This is the number of nodes of the associated Coxeter or Dynkin diagram.

EXAMPLES:

```
sage: CartanType(['A', 4]).rank()
4
```

(continues on next page)

(continued from previous page)

```
sage: CartanType(['A', 7, 2]).rank()
5
sage: CartanType(['I', 8]).rank()
2
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(4)]).rank()
4
>>> CartanType(['A', Integer(7), Integer(2)]).rank()
5
>>> CartanType(['I', Integer(8)]).rank()
2
```

relabel(*relabelling*)

Return a relabelled copy of this Cartan type.

INPUT:

- `relabelling` – a function (or a list or dictionary)

OUTPUT:

an isomorphic Cartan type obtained by relabelling the nodes of the Dynkin diagram. Namely, the node with label `i` is relabelled `f(i)` (or, by `f[i]` if `f` is a list or dictionary).

EXAMPLES:

```
sage: CartanType(['F',4]).relabel({ 1:4, 2:3, 3:2, 4:1 }).dynkin_diagram()
↪# needs sage.graphs
O---O=>=O---O
4   3   2   1
F4 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> CartanType(['F',Integer(4)]).relabel({ Integer(1):Integer(4),␣
↪Integer(2):Integer(3), Integer(3):Integer(2), Integer(4):Integer(1) }).
↪dynkin_diagram()    # needs sage.graphs
O---O=>=O---O
4   3   2   1
F4 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

root_system()

Return the root system associated to `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).root_system()
Root system of type ['A', 4]
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).root_system()
Root system of type ['A', 4]
```

subtype(*index_set*)

Return a subtype of `self` given by `index_set`.

A subtype can be considered the Dynkin diagram induced from the Dynkin diagram of `self` by `index_set`.

EXAMPLES:

```
sage: ct = CartanType(['A',6,2])
sage: ct.dynkin_diagram()                                               #␣
↪needs sage.graphs
O=<=O---O=<=O
0   1   2   3
BC3~
sage: ct.subtype([1,2,3])                                               #␣
↪needs sage.graphs
['C', 3]
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(6),Integer(2)])
>>> ct.dynkin_diagram()                                               #␣
↪needs sage.graphs
O=<=O---O=<=O
0   1   2   3
BC3~
>>> ct.subtype([Integer(1),Integer(2),Integer(3)])                      ␣
↪                      # needs sage.graphs
['C', 3]
```

type()

Return the type of `self`, or `None` if unknown.

This method should be overridden in any subclass.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
sage: C.type() is None
True
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.cartan_type import CartanType_abstract
>>> C = CartanType_abstract()
>>> C.type() is None
True
```

class sage.combinat.root_system.cartan_type.**CartanType_affine**

Bases: *CartanType_simple*, *CartanType_crystallographic*

An abstract class for simple affine Cartan types

AmbientSpace

alias of *AmbientSpace*

a()

Return the unique minimal non trivial annihilating linear combination of $\alpha_0^\vee, \alpha^\vee, \ldots, \alpha^\vee$ with nonnegative coefficients (or alternatively, the unique minimal non trivial annihilating linear combination of the columns of the Cartan matrix with nonnegative coefficients).

Throw an error if the existence or uniqueness does not hold

FIXME: the current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.

EXAMPLES:

```
sage: # needs sage.graphs
sage: RootSystem(['C',2,1]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().a()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 3, 3: 4, 4: 2}
sage: RootSystem(['BC',4,2]).cartan_type().a()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> RootSystem(['C',Integer(2),Integer(1)]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 1}
>>> RootSystem(['D',Integer(4),Integer(1)]).cartan_type().a()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
>>> RootSystem(['F',Integer(4),Integer(1)]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 3, 3: 4, 4: 2}
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().a()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

`a` is a shortcut for col_annihilator:

```
sage: RootSystem(['BC',4,2]).cartan_type().col_annihilator()                    #␣
↪needs sage.graphs
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().col_annihilator() ␣
↪               # needs sage.graphs
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

acheck(*m=None*)

Return the unique minimal non trivial annihilating linear combination of $\alpha_0, \alpha_1, \ldots, \alpha_n$ with nonnegative coefficients (or alternatively, the unique minimal non trivial annihilating linear combination of the rows of the Cartan matrix with nonnegative coefficients).

Throw an error if the existence of uniqueness does not hold

The optional argument `m` is for internal use only.

EXAMPLES:

```
sage: # needs sage.graphs
sage: RootSystem(['C',2,1]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 1}
```

(continues on next page)

(continued from previous page)

```
sage: RootSystem(['D',4,1]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 3, 3: 2, 4: 1}
sage: RootSystem(['BC',4,2]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> RootSystem(['C',Integer(2),Integer(1)]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 1}
>>> RootSystem(['D',Integer(4),Integer(1)]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
>>> RootSystem(['F',Integer(4),Integer(1)]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 3, 3: 2, 4: 1}
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

`acheck` is a shortcut for row_annihilator:

```
sage: RootSystem(['BC',4,2]).cartan_type().row_annihilator()                  #␣
↪needs sage.graphs
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

```
>>> from sage.all import *
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().row_annihilator() ␣
↪              # needs sage.graphs
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

FIXME:

- The current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.
- This really should be a method of *CartanMatrix*.

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: CartanType(['A', 1, 1]).basic_untwisted()
['A', 1]
sage: CartanType(['A', 3, 1]).basic_untwisted()
['A', 3]
sage: CartanType(['B', 3, 1]).basic_untwisted()
['B', 3]
sage: CartanType(['E', 6, 1]).basic_untwisted()
['E', 6]
sage: CartanType(['G', 2, 1]).basic_untwisted()
['G', 2]
```

(continues on next page)

(continued from previous page)

```
sage: CartanType(['A', 2, 2]).basic_untwisted()
['A', 2]
sage: CartanType(['A', 4, 2]).basic_untwisted()
['A', 4]
sage: CartanType(['A', 11, 2]).basic_untwisted()
['A', 11]
sage: CartanType(['D', 5, 2]).basic_untwisted()
['D', 5]
sage: CartanType(['E', 6, 2]).basic_untwisted()
['E', 6]
sage: CartanType(['D', 4, 3]).basic_untwisted()
['D', 4]
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(1), Integer(1)]).basic_untwisted()
['A', 1]
>>> CartanType(['A', Integer(3), Integer(1)]).basic_untwisted()
['A', 3]
>>> CartanType(['B', Integer(3), Integer(1)]).basic_untwisted()
['B', 3]
>>> CartanType(['E', Integer(6), Integer(1)]).basic_untwisted()
['E', 6]
>>> CartanType(['G', Integer(2), Integer(1)]).basic_untwisted()
['G', 2]

>>> CartanType(['A', Integer(2), Integer(2)]).basic_untwisted()
['A', 2]
>>> CartanType(['A', Integer(4), Integer(2)]).basic_untwisted()
['A', 4]
>>> CartanType(['A', Integer(11), Integer(2)]).basic_untwisted()
['A', 11]
>>> CartanType(['D', Integer(5), Integer(2)]).basic_untwisted()
['D', 5]
>>> CartanType(['E', Integer(6), Integer(2)]).basic_untwisted()
['E', 6]
>>> CartanType(['D', Integer(4), Integer(3)]).basic_untwisted()
['D', 4]
```

c()

Return the family (c_i)_i of integer coefficients defined by $c_i = max(1, a_i/a^vee_i)$ (see e.g. [FSS07] p. 3)

FIXME: the current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.

EXAMPLES:

```
sage: # needs sage.graphs
sage: RootSystem(['C',2,1]).cartan_type().c()
Finite family {0: 1, 1: 2, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().c()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().c()
```

(continues on next page)

(continued from previous page)

```
Finite family {0: 1, 1: 1, 2: 1, 3: 2, 4: 2}
sage: RootSystem(['BC',4,2]).cartan_type().c()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> RootSystem(['C',Integer(2),Integer(1)]).cartan_type().c()
Finite family {0: 1, 1: 2, 2: 1}
>>> RootSystem(['D',Integer(4),Integer(1)]).cartan_type().c()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
>>> RootSystem(['F',Integer(4),Integer(1)]).cartan_type().c()
Finite family {0: 1, 1: 1, 2: 1, 3: 2, 4: 2}
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().c()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
```

REFERENCES:

classical()

Return the classical Cartan type associated with this affine Cartan type.

EXAMPLES:

```
sage: CartanType(['A', 1, 1]).classical()
['A', 1]
sage: CartanType(['A', 3, 1]).classical()
['A', 3]
sage: CartanType(['B', 3, 1]).classical()
['B', 3]

sage: CartanType(['A', 2, 2]).classical()
['C', 1]
sage: CartanType(['BC', 1, 2]).classical()
['C', 1]
sage: CartanType(['A', 4, 2]).classical()
['C', 2]
sage: CartanType(['BC', 2, 2]).classical()
['C', 2]
sage: CartanType(['A', 10, 2]).classical()
['C', 5]
sage: CartanType(['BC', 5, 2]).classical()
['C', 5]

sage: CartanType(['D', 5, 2]).classical()
['B', 4]
sage: CartanType(['E', 6, 1]).classical()
['E', 6]
sage: CartanType(['G', 2, 1]).classical()
['G', 2]
sage: CartanType(['E', 6, 2]).classical()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: CartanType(['D', 4, 3]).classical()
['G', 2]
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(1), Integer(1)]).classical()
['A', 1]
>>> CartanType(['A', Integer(3), Integer(1)]).classical()
['A', 3]
>>> CartanType(['B', Integer(3), Integer(1)]).classical()
['B', 3]

>>> CartanType(['A', Integer(2), Integer(2)]).classical()
['C', 1]
>>> CartanType(['BC', Integer(1), Integer(2)]).classical()
['C', 1]
>>> CartanType(['A', Integer(4), Integer(2)]).classical()
['C', 2]
>>> CartanType(['BC', Integer(2), Integer(2)]).classical()
['C', 2]
>>> CartanType(['A', Integer(10), Integer(2)]).classical()
['C', 5]
>>> CartanType(['BC', Integer(5), Integer(2)]).classical()
['C', 5]

>>> CartanType(['D', Integer(5), Integer(2)]).classical()
['B', 4]
>>> CartanType(['E', Integer(6), Integer(1)]).classical()
['E', 6]
>>> CartanType(['G', Integer(2), Integer(1)]).classical()
['G', 2]
>>> CartanType(['E', Integer(6), Integer(2)]).classical()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
>>> CartanType(['D', Integer(4), Integer(3)]).classical()
['G', 2]
```

We check that `classical()`, `sage.combinat.root_system.cartan_type.CartanType_crystallographic.dynkin_diagram()`, and `special_node()` are consistent:

```
sage: for ct in CartanType.samples(affine=True):                            #␣
↪needs sage.graphs
....:     g1 = ct.classical().dynkin_diagram()
....:     g2 = ct.dynkin_diagram()
....:     g2.delete_vertex(ct.special_node())
....:     assert g1.vertices(sort=True) == g2.vertices(sort=True)
....:     assert g1.edges(sort=True) == g2.edges(sort=True)
```

```
>>> from sage.all import *
>>> for ct in CartanType.samples(affine=True):                              #␣
↪needs sage.graphs
...     g1 = ct.classical().dynkin_diagram()
...     g2 = ct.dynkin_diagram()
...     g2.delete_vertex(ct.special_node())
...     assert g1.vertices(sort=True) == g2.vertices(sort=True)
...     assert g1.edges(sort=True) == g2.edges(sort=True)
```

col_annihilator()

Return the unique minimal non trivial annihilating linear combination of $\alpha_0^\vee, \alpha^\vee, \ldots, \alpha^\vee$ with nonnegative coefficients (or alternatively, the unique minimal non trivial annihilating linear combination of the columns of the Cartan matrix with nonnegative coefficients).

Throw an error if the existence or uniqueness does not hold

FIXME: the current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.

EXAMPLES:

```
sage: # needs sage.graphs
sage: RootSystem(['C',2,1]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().a()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 3, 3: 4, 4: 2}
sage: RootSystem(['BC',4,2]).cartan_type().a()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> RootSystem(['C',Integer(2),Integer(1)]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 1}
>>> RootSystem(['D',Integer(4),Integer(1)]).cartan_type().a()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
>>> RootSystem(['F',Integer(4),Integer(1)]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 3, 3: 4, 4: 2}
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().a()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

`a` is a shortcut for col_annihilator:

```
sage: RootSystem(['BC',4,2]).cartan_type().col_annihilator()                    #␣
↪needs sage.graphs
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().col_annihilator() ␣
↪              # needs sage.graphs
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

is_affine()

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_affine()
True
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).is_affine()
True
```

is_finite()

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_finite()
False
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).is_finite()
False
```

is_untwisted_affine()

Return whether `self` is untwisted affine.

A Cartan type is untwisted affine if it is the canonical affine extension of some finite type. Every affine type is either untwisted affine, dual thereof, or of type `BC`.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_untwisted_affine()
True
sage: CartanType(['A', 3, 1]).dual().is_untwisted_affine()   # this one is␣
↪self dual!
True
sage: CartanType(['B', 3, 1]).dual().is_untwisted_affine()
False
sage: CartanType(['BC', 3, 2]).is_untwisted_affine()
False
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).is_untwisted_affine()
True
>>> CartanType(['A', Integer(3), Integer(1)]).dual().is_untwisted_affine()   #␣
↪this one is self dual!
True
>>> CartanType(['B', Integer(3), Integer(1)]).dual().is_untwisted_affine()
False
>>> CartanType(['BC', Integer(3), Integer(2)]).is_untwisted_affine()
False
```

other_affinization()

Return the other affinization of the same classical type.

EXAMPLES:

```
sage: CartanType(["A", 3, 1]).other_affinization()
['A', 3, 1]
sage: CartanType(["B", 3, 1]).other_affinization()
['C', 3, 1]^*
sage: CartanType(["C", 3, 1]).dual().other_affinization()
['B', 3, 1]
```

```
>>> from sage.all import *
>>> CartanType(["A", Integer(3), Integer(1)]).other_affinization()
['A', 3, 1]
>>> CartanType(["B", Integer(3), Integer(1)]).other_affinization()
['C', 3, 1]^*
```

(continues on next page)

(continued from previous page)

```
>>> CartanType(["C", Integer(3), Integer(1)]).dual().other_affinization()
['B', 3, 1]
```

Is this what we want?:

```
sage: CartanType(["BC", 3, 2]).dual().other_affinization()
['B', 3, 1]
```

```
>>> from sage.all import *
>>> CartanType(["BC", Integer(3), Integer(2)]).dual().other_affinization()
['B', 3, 1]
```

row_annihilator(*m=None*)

Return the unique minimal non trivial annihilating linear combination of $\alpha_0, \alpha_1, \ldots, \alpha_n$ with nonnegative coefficients (or alternatively, the unique minimal non trivial annihilating linear combination of the rows of the Cartan matrix with nonnegative coefficients).

Throw an error if the existence of uniqueness does not hold

The optional argument `m` is for internal use only.

EXAMPLES:

```
sage: # needs sage.graphs
sage: RootSystem(['C',2,1]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 3, 3: 2, 4: 1}
sage: RootSystem(['BC',4,2]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> RootSystem(['C',Integer(2),Integer(1)]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 1}
>>> RootSystem(['D',Integer(4),Integer(1)]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
>>> RootSystem(['F',Integer(4),Integer(1)]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 3, 3: 2, 4: 1}
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

`acheck` is a shortcut for row_annihilator:

```
sage: RootSystem(['BC',4,2]).cartan_type().row_annihilator()                #
↪needs sage.graphs
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

```
>>> from sage.all import *
>>> RootSystem(['BC',Integer(4),Integer(2)]).cartan_type().row_annihilator()
```

(continues on next page)

(continued from previous page)

```
↪                # needs sage.graphs
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

FIXME:

- The current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.
- This really should be a method of `CartanMatrix`.

special_node()

Return a special node of the Dynkin diagram.

A *special* node is a node of the Dynkin diagram such that pruning it yields a Dynkin diagram for the associated classical type (see `classical()`).

This method returns the label of some special node. This is usually 0 in the standard conventions.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).special_node()
0
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).special_node()
0
```

The choice is guaranteed to be consistent with the indexing of the nodes of the classical Dynkin diagram:

```
sage: CartanType(['A', 3, 1]).index_set()
(0, 1, 2, 3)
sage: CartanType(['A', 3, 1]).classical().index_set()
(1, 2, 3)
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).index_set()
(0, 1, 2, 3)
>>> CartanType(['A', Integer(3), Integer(1)]).classical().index_set()
(1, 2, 3)
```

special_nodes()

Return the set of special nodes of the affine Dynkin diagram.

EXAMPLES:

```
sage: # needs sage.graphs sage.groups
sage: CartanType(['A',3,1]).special_nodes()
(0, 1, 2, 3)
sage: CartanType(['C',2,1]).special_nodes()
(0, 2)
sage: CartanType(['D',4,1]).special_nodes()
(0, 1, 3, 4)
sage: CartanType(['E',6,1]).special_nodes()
(0, 1, 6)
sage: CartanType(['D',3,2]).special_nodes()
```

(continues on next page)

(continued from previous page)

```
(0, 2)
sage: CartanType(['A',4,2]).special_nodes()
(0,)
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> CartanType(['A',Integer(3),Integer(1)]).special_nodes()
(0, 1, 2, 3)
>>> CartanType(['C',Integer(2),Integer(1)]).special_nodes()
(0, 2)
>>> CartanType(['D',Integer(4),Integer(1)]).special_nodes()
(0, 1, 3, 4)
>>> CartanType(['E',Integer(6),Integer(1)]).special_nodes()
(0, 1, 6)
>>> CartanType(['D',Integer(3),Integer(2)]).special_nodes()
(0, 2)
>>> CartanType(['A',Integer(4),Integer(2)]).special_nodes()
(0,)
```

translation_factors()

Return the translation factors for `self`.

Those are the smallest factors t_i such that the translation by $t_i\alpha_i$ maps the fundamental polygon to another polygon in the alcove picture.

OUTPUT:

a dictionary from `self.index_set()` to $\mathbf{Z}$ (or $\mathbf{Q}$ for affine type BC)

Those coefficients are all 1 for dual untwisted, and in particular for simply laced. They coincide with the usual c_i coefficients (see `c()`) for untwisted and dual thereof. See the discussion below for affine type BC.

Note

One usually realizes the alcove picture in the coweight lattice, with translations by coroots; in that case, one will use the translation factors for the dual Cartan type.

FIXME: the current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.

EXAMPLES:

```
sage: # needs sage.graphs
sage: CartanType(['C',2,1]).translation_factors()
Finite family {0: 1, 1: 2, 2: 1}
sage: CartanType(['C',2,1]).dual().translation_factors()
Finite family {0: 1, 1: 1, 2: 1}
sage: CartanType(['D',4,1]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
sage: CartanType(['F',4,1]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 2, 4: 2}
sage: CartanType(['BC',4,2]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1/2}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> CartanType(['C',Integer(2),Integer(1)]).translation_factors()
Finite family {0: 1, 1: 2, 2: 1}
>>> CartanType(['C',Integer(2),Integer(1)]).dual().translation_factors()
Finite family {0: 1, 1: 1, 2: 1}
>>> CartanType(['D',Integer(4),Integer(1)]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
>>> CartanType(['F',Integer(4),Integer(1)]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 2, 4: 2}
>>> CartanType(['BC',Integer(4),Integer(2)]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1/2}
```

We proceed with systematic tests taken from MuPAD-Combinat's testsuite:

```
sage: # needs sage.graphs
sage: list(CartanType(["A", 1, 1]).translation_factors())
[1, 1]
sage: list(CartanType(["A", 5, 1]).translation_factors())
[1, 1, 1, 1, 1, 1]
sage: list(CartanType(["B", 5, 1]).translation_factors())
[1, 1, 1, 1, 1, 2]
sage: list(CartanType(["C", 5, 1]).translation_factors())
[1, 2, 2, 2, 2, 1]
sage: list(CartanType(["D", 5, 1]).translation_factors())
[1, 1, 1, 1, 1, 1]
sage: list(CartanType(["E", 6, 1]).translation_factors())
[1, 1, 1, 1, 1, 1, 1]
sage: list(CartanType(["E", 7, 1]).translation_factors())
[1, 1, 1, 1, 1, 1, 1, 1]
sage: list(CartanType(["E", 8, 1]).translation_factors())
[1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: list(CartanType(["F", 4, 1]).translation_factors())
[1, 1, 1, 2, 2]
sage: list(CartanType(["G", 2, 1]).translation_factors())
[1, 3, 1]
sage: list(CartanType(["A", 2, 2]).translation_factors())
[1, 1/2]
sage: list(CartanType(["A", 2, 2]).dual().translation_factors())
[1/2, 1]
sage: list(CartanType(["A", 10, 2]).translation_factors())
[1, 1, 1, 1, 1, 1/2]
sage: list(CartanType(["A", 10, 2]).dual().translation_factors())
[1/2, 1, 1, 1, 1, 1]
sage: list(CartanType(["A", 9, 2]).translation_factors())
[1, 1, 1, 1, 1, 1]
sage: list(CartanType(["D", 5, 2]).translation_factors())
[1, 1, 1, 1, 1]
sage: list(CartanType(["D", 4, 3]).translation_factors())
[1, 1, 1]
sage: list(CartanType(["E", 6, 2]).translation_factors())
[1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> list(CartanType(["A", Integer(1), Integer(1)]).translation_factors())
[1, 1]
>>> list(CartanType(["A", Integer(5), Integer(1)]).translation_factors())
[1, 1, 1, 1, 1, 1]
>>> list(CartanType(["B", Integer(5), Integer(1)]).translation_factors())
[1, 1, 1, 1, 1, 2]
>>> list(CartanType(["C", Integer(5), Integer(1)]).translation_factors())
[1, 2, 2, 2, 2, 1]
>>> list(CartanType(["D", Integer(5), Integer(1)]).translation_factors())
[1, 1, 1, 1, 1, 1]
>>> list(CartanType(["E", Integer(6), Integer(1)]).translation_factors())
[1, 1, 1, 1, 1, 1, 1]
>>> list(CartanType(["E", Integer(7), Integer(1)]).translation_factors())
[1, 1, 1, 1, 1, 1, 1, 1]
>>> list(CartanType(["E", Integer(8), Integer(1)]).translation_factors())
[1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> list(CartanType(["F", Integer(4), Integer(1)]).translation_factors())
[1, 1, 1, 2, 2]
>>> list(CartanType(["G", Integer(2), Integer(1)]).translation_factors())
[1, 3, 1]
>>> list(CartanType(["A", Integer(2), Integer(2)]).translation_factors())
[1, 1/2]
>>> list(CartanType(["A", Integer(2), Integer(2)]).dual().translation_
↪factors())
[1/2, 1]
>>> list(CartanType(["A", Integer(10), Integer(2)]).translation_factors())
[1, 1, 1, 1, 1, 1/2]
>>> list(CartanType(["A", Integer(10), Integer(2)]).dual().translation_
↪factors())
[1/2, 1, 1, 1, 1, 1]
>>> list(CartanType(["A", Integer(9), Integer(2)]).translation_factors())
[1, 1, 1, 1, 1, 1]
>>> list(CartanType(["D", Integer(5), Integer(2)]).translation_factors())
[1, 1, 1, 1, 1]
>>> list(CartanType(["D", Integer(4), Integer(3)]).translation_factors())
[1, 1, 1]
>>> list(CartanType(["E", Integer(6), Integer(2)]).translation_factors())
[1, 1, 1, 1, 1]
```

We conclude with a discussion of the appropriate value for affine type BC. Let us consider the alcove picture realized in the weight lattice. It is obtained by taking the level-1 affine hyperplane in the weight lattice, and projecting it along Λ_0:

```
sage: R = RootSystem(["BC",2,2])
sage: alpha = R.weight_space().simple_roots()                              #␣
↪needs sage.graphs
sage: alphacheck = R.coroot_space().simple_roots()
sage: Lambda = R.weight_space().fundamental_weights()
```

```
>>> from sage.all import *
>>> R = RootSystem(["BC",Integer(2),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> alpha = R.weight_space().simple_roots()                                    #␣
↪needs sage.graphs
>>> alphacheck = R.coroot_space().simple_roots()
>>> Lambda = R.weight_space().fundamental_weights()
```

Here are the levels of the fundamental weights:

```
sage: Lambda[0].level(), Lambda[1].level(), Lambda[2].level()                  #␣
↪needs sage.graphs
(1, 2, 2)
```

```
>>> from sage.all import *
>>> Lambda[Integer(0)].level(), Lambda[Integer(1)].level(),␣
↪Lambda[Integer(2)].level()                         # needs sage.graphs
(1, 2, 2)
```

So the "center" of the fundamental polygon at level 1 is:

```
sage: O = Lambda[0]
sage: O.level()                                                                #␣
↪needs sage.graphs
1
```

```
>>> from sage.all import *
>>> O = Lambda[Integer(0)]
>>> O.level()                                                                  #␣
↪needs sage.graphs
1
```

We take the projection ω_1 at level 0 of Λ_1 as unit vector on the x-axis, and the projection ω_2 at level 0 of Λ_2 as unit vector of the y-axis:

```
sage: omega1 = Lambda[1] - 2*Lambda[0]
sage: omega2 = Lambda[2] - 2*Lambda[0]
sage: omega1.level(), omega2.level()                                           #␣
↪needs sage.graphs
(0, 0)
```

```
>>> from sage.all import *
>>> omega1 = Lambda[Integer(1)] - Integer(2)*Lambda[Integer(0)]
>>> omega2 = Lambda[Integer(2)] - Integer(2)*Lambda[Integer(0)]
>>> omega1.level(), omega2.level()                                             #␣
↪needs sage.graphs
(0, 0)
```

The projections of the simple roots can be read off:

```
sage: alpha[0]                                                                 #␣
↪needs sage.graphs
2*Lambda[0] - Lambda[1]
sage: alpha[1]                                                                 #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
-2*Lambda[0] + 2*Lambda[1] - Lambda[2]
sage: alpha[2]                                                                    #␣
↪needs sage.graphs
-2*Lambda[1] + 2*Lambda[2]
```

```
>>> from sage.all import *
>>> alpha[Integer(0)]                                                             ␣
↪     # needs sage.graphs
2*Lambda[0] - Lambda[1]
>>> alpha[Integer(1)]                                                             ␣
↪     # needs sage.graphs
-2*Lambda[0] + 2*Lambda[1] - Lambda[2]
>>> alpha[Integer(2)]                                                             ␣
↪     # needs sage.graphs
-2*Lambda[1] + 2*Lambda[2]
```

Namely $\alpha_0 = -\omega_1$, $\alpha_1 = 2\omega_1 - \omega_2$ and $\alpha_2 = -2\omega_1 + 2\omega_2$.

The reflection hyperplane defined by $\alpha_0^\vee$ goes through the points $O + 1/2\omega_1$ and $O + 1/2\omega_2$:

```
sage: (O+(1/2)*omega1).scalar(alphacheck[0])
0
sage: (O+(1/2)*omega2).scalar(alphacheck[0])
0
```

```
>>> from sage.all import *
>>> (O+(Integer(1)/Integer(2))*omega1).scalar(alphacheck[Integer(0)])
0
>>> (O+(Integer(1)/Integer(2))*omega2).scalar(alphacheck[Integer(0)])
0
```

Hence, the fundamental alcove is the triangle $(O, O + 1/2\omega_1, O + 1/2\omega_2)$. By successive reflections, one can tile the full plane. This induces a tiling of the full plane by translates of the fundamental polygon.

Todo

Add the picture here, once root system plots in the weight lattice will be implemented. In the mean time, the reader may look up the dual picture on Figure 2 of [HST09] which was produced by MuPAD-Combinat.

From this picture, one can read that translations by α_0, α_1, and $1/2\alpha_2$ map the fundamental polygon to translates of it in the alcove picture, and are smallest with this property. Hence, the translation factors for affine type BC are $t_0 = 1, t_1 = 1, t_2 = 1/2$:

```
sage: CartanType(['BC',2,2]).translation_factors()                                #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 1/2}
```

```
>>> from sage.all import *
>>> CartanType(['BC',Integer(2),Integer(2)]).translation_factors()                ␣
↪             # needs sage.graphs
Finite family {0: 1, 1: 1, 2: 1/2}
```

REFERENCES:

class sage.combinat.root_system.cartan_type.**CartanType_crystallographic**

Bases: *CartanType_abstract*

An abstract class for crystallographic Cartan types.

ascii_art(*label='lambda x: x'*, *node=None*)

Return an ascii art representation of the Dynkin diagram.

INPUT:

- label – (default: the identity) a relabeling function for the nodes
- node – (optional) a function which returns the character for a node

EXAMPLES:

```
sage: cartan_type = CartanType(['B',5,1])
sage: print(cartan_type.ascii_art())
    O 0
    |
    |
O---O---O---O=>=O
1   2   3   4   5
```

```
>>> from sage.all import *
>>> cartan_type = CartanType(['B',Integer(5),Integer(1)])
>>> print(cartan_type.ascii_art())
    O 0
    |
    |
O---O---O---O=>=O
1   2   3   4   5
```

The label option is useful to visualize various statistics on the nodes of the Dynkin diagram:

```
sage: a = cartan_type.col_annihilator(); a                                    #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 2, 3: 2, 4: 2, 5: 2}
sage: print(CartanType(['B',5,1]).ascii_art(label=a.__getitem__))             #␣
↪needs sage.graphs
    O 1
    |
    |
O---O---O---O=>=O
1   2   2   2   2
```

```
>>> from sage.all import *
>>> a = cartan_type.col_annihilator(); a                                    #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 2, 3: 2, 4: 2, 5: 2}
>>> print(CartanType(['B',Integer(5),Integer(1)]).ascii_art(label=a.__getitem_
↪_))           # needs sage.graphs
    O 1
    |
```

(continues on next page)

(continued from previous page)

```
    |
O---O---O---O=>=O
1   2   2   2   2
```

cartan_matrix()

Return the Cartan matrix associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).cartan_matrix()                                    #␣
↪needs sage.graphs
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).cartan_matrix()                            ␣
↪     # needs sage.graphs
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

coxeter_diagram()

Return the Coxeter diagram for `self`.

This implementation constructs it from the Dynkin diagram.

> **See also**
>
> *CartanType_abstract.coxeter_diagram()*

EXAMPLES:

```
sage: # needs sage.graphs
sage: CartanType(['A',3]).coxeter_diagram()
Graph on 3 vertices
sage: CartanType(['A',3]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 3)]
sage: CartanType(['B',3]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 4)]
sage: CartanType(['G',2]).coxeter_diagram().edges(sort=True)
[(1, 2, 6)]
sage: CartanType(['F',4]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 4), (3, 4, 3)]
sage: CartanType(['A',2,2]).coxeter_diagram().edges(sort=True)
[(0, 1, +Infinity)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
>>> CartanType(['A',Integer(3)]).coxeter_diagram()
Graph on 3 vertices
>>> CartanType(['A',Integer(3)]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 3)]
>>> CartanType(['B',Integer(3)]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 4)]
>>> CartanType(['G',Integer(2)]).coxeter_diagram().edges(sort=True)
[(1, 2, 6)]
>>> CartanType(['F',Integer(4)]).coxeter_diagram().edges(sort=True)
[(1, 2, 3), (2, 3, 4), (3, 4, 3)]
>>> CartanType(['A',Integer(2),Integer(2)]).coxeter_diagram().edges(sort=True)
[(0, 1, +Infinity)]
```

dynkin_diagram()

Return the Dynkin diagram associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).dynkin_diagram()                              #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).dynkin_diagram()                       ␣
↪     # needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

> **Note**
>
> Derived subclasses should typically implement this as a cached method.

index_set_bipartition()

Return a bipartition $\{L, R\}$ of the vertices of the Dynkin diagram.

For i and j both in L (or both in R), the simple reflections s_i and s_j commute.

Of course, the Dynkin diagram should be bipartite. This is always the case for all finite types.

EXAMPLES:

```
sage: CartanType(['A',5]).index_set_bipartition()                       #␣
↪needs sage.graphs
({1, 3, 5}, {2, 4})

sage: CartanType(['A',2,1]).index_set_bipartition()                     #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: the Dynkin diagram must be bipartite
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(5)]).index_set_bipartition()                    ␣
↪     # needs sage.graphs
({1, 3, 5}, {2, 4})

>>> CartanType(['A',Integer(2),Integer(1)]).index_set_bipartition()         ␣
↪               # needs sage.graphs
Traceback (most recent call last):
...
ValueError: the Dynkin diagram must be bipartite
```

is_crystallographic()

Implement *CartanType_abstract.is_crystallographic()* by returning `True`.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_crystallographic()
True
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).is_crystallographic()
True
```

symmetrizer()

Return the symmetrizer of the Cartan matrix of `self`.

A Cartan matrix M is symmetrizable if there exists a non trivial diagonal matrix D such that DM is a symmetric matrix, that is $DM = M^t D$. In that case, D is unique, up to a scalar factor for each connected component of the Dynkin diagram.

This method computes the unique minimal such D with positive integral coefficients. If D exists, it is returned as a family. Otherwise `None` is returned.

The coefficients are coerced to `base_ring`.

EXAMPLES:

```
sage: CartanType(["B",5]).symmetrizer()                                    #␣
↪needs sage.graphs
Finite family {1: 2, 2: 2, 3: 2, 4: 2, 5: 1}
```

```
>>> from sage.all import *
>>> CartanType(["B",Integer(5)]).symmetrizer()                              ␣
↪     # needs sage.graphs
Finite family {1: 2, 2: 2, 3: 2, 4: 2, 5: 1}
```

Here is a neat trick to visualize it better:

```
sage: T = CartanType(["B",5])
sage: print(T.ascii_art(T.symmetrizer().__getitem__))                      #␣
↪needs sage.graphs
O---O---O---O=>=O
2   2   2   2   1

sage: T = CartanType(["BC",5, 2])
```

(continues on next page)

(continued from previous page)

```
sage: print(T.ascii_art(T.symmetrizer().__getitem__))                       #␣
↪needs sage.graphs
O=<=O---O---O---O=<=O
1   2   2   2   2   4
```

```
>>> from sage.all import *
>>> T = CartanType(["B",Integer(5)])
>>> print(T.ascii_art(T.symmetrizer().__getitem__))                       #␣
↪needs sage.graphs
O---O---O---O=>=O
2   2   2   2   1

>>> T = CartanType(["BC",Integer(5), Integer(2)])
>>> print(T.ascii_art(T.symmetrizer().__getitem__))                       #␣
↪needs sage.graphs
O=<=O---O---O---O=<=O
1   2   2   2   2   4
```

Here is the symmetrizer of some reducible Cartan types:

```
sage: T = CartanType(["D", 2])
sage: print(T.ascii_art(T.symmetrizer().__getitem__))                       #␣
↪needs sage.graphs
O   O
1   1

sage: T = CartanType(["B",5],["BC",5, 2])
sage: print(T.ascii_art(T.symmetrizer().__getitem__))                       #␣
↪needs sage.graphs
O---O---O---O=>=O
2   2   2   2   1
O=<=O---O---O---O=<=O
1   2   2   2   2   4
```

```
>>> from sage.all import *
>>> T = CartanType(["D", Integer(2)])
>>> print(T.ascii_art(T.symmetrizer().__getitem__))                       #␣
↪needs sage.graphs
O   O
1   1

>>> T = CartanType(["B",Integer(5)],["BC",Integer(5), Integer(2)])
>>> print(T.ascii_art(T.symmetrizer().__getitem__))                       #␣
↪needs sage.graphs
O---O---O---O=>=O
2   2   2   2   1
O=<=O---O---O---O=<=O
1   2   2   2   2   4
```

Property: up to an overall scalar factor, this gives the norm of the simple roots in the ambient space:

```
sage: T = CartanType(["C",5])
sage: print(T.ascii_art(T.symmetrizer().__getitem__))                          #␣
↪needs sage.graphs
O---O---O---O=<=O
1   1   1   1   2

sage: alpha = RootSystem(T).ambient_space().simple_roots()
sage: print(T.ascii_art(lambda i: alpha[i].scalar(alpha[i])))
O---O---O---O=<=O
2   2   2   2   4
```

```
>>> from sage.all import *
>>> T = CartanType(["C",Integer(5)])
>>> print(T.ascii_art(T.symmetrizer().__getitem__))                          #␣
↪needs sage.graphs
O---O---O---O=<=O
1   1   1   1   2

>>> alpha = RootSystem(T).ambient_space().simple_roots()
>>> print(T.ascii_art(lambda i: alpha[i].scalar(alpha[i])))
O---O---O---O=<=O
2   2   2   2   4
```

class sage.combinat.root_system.cartan_type.**CartanType_decorator**(*ct*)

Bases: UniqueRepresentation, SageObject, *CartanType_abstract*

Concrete base class for Cartan types that decorate another Cartan type.

index_set()

EXAMPLES:

```
sage: ct = CartanType(['F', 4, 1]).dual()
sage: ct.index_set()
(0, 1, 2, 3, 4)
```

```
>>> from sage.all import *
>>> ct = CartanType(['F', Integer(4), Integer(1)]).dual()
>>> ct.index_set()
(0, 1, 2, 3, 4)
```

is_affine()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.is_affine()
False
```

```
>>> from sage.all import *
>>> ct = CartanType(['G', Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)})
>>> ct.is_affine()
False
```

is_crystallographic()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.is_crystallographic()
True
```

```
>>> from sage.all import *
>>> ct = CartanType(['G', Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)})
>>> ct.is_crystallographic()
True
```

is_finite()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.is_finite()
True
```

```
>>> from sage.all import *
>>> ct = CartanType(['G', Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)})
>>> ct.is_finite()
True
```

is_irreducible()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.is_irreducible()
True
```

```
>>> from sage.all import *
>>> ct = CartanType(['G', Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)})
>>> ct.is_irreducible()
True
```

rank()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.rank()
2
```

```
>>> from sage.all import *
>>> ct = CartanType(['G', Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)})
>>> ct.rank()
2
```

class sage.combinat.root_system.cartan_type.**CartanType_finite**

Bases: *CartanType_abstract*

An abstract class for simple affine Cartan types.

is_affine()

EXAMPLES:

```
sage: CartanType(["A", 3]).is_affine()
False
```

```
>>> from sage.all import *
>>> CartanType(["A", Integer(3)]).is_affine()
False
```

is_finite()

EXAMPLES:

```
sage: CartanType(["A", 3]).is_finite()
True
```

```
>>> from sage.all import *
>>> CartanType(["A", Integer(3)]).is_finite()
True
```

class sage.combinat.root_system.cartan_type.**CartanType_simple**

Bases: *CartanType_abstract*

An abstract class for simple Cartan types.

is_irreducible()

Return whether `self` is irreducible, which is `True`.

EXAMPLES:

```
sage: CartanType(['A', 3]).is_irreducible()
True
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3)]).is_irreducible()
True
```

class sage.combinat.root_system.cartan_type.**CartanType_simply_laced**

Bases: *CartanType_crystallographic*

An abstract class for simply laced Cartan types.

dual()

Simply laced Cartan types are self-dual, so return `self`.

EXAMPLES:

```
sage: CartanType(["A", 3]).dual()
['A', 3]
sage: CartanType(["A", 3, 1]).dual()
```

(continues on next page)

(continued from previous page)

```
['A', 3, 1]
sage: CartanType(["D", 3]).dual()
['D', 3]
sage: CartanType(["D", 4, 1]).dual()
['D', 4, 1]
sage: CartanType(["E", 6]).dual()
['E', 6]
sage: CartanType(["E", 6, 1]).dual()
['E', 6, 1]
```

```
>>> from sage.all import *
>>> CartanType(["A", Integer(3)]).dual()
['A', 3]
>>> CartanType(["A", Integer(3), Integer(1)]).dual()
['A', 3, 1]
>>> CartanType(["D", Integer(3)]).dual()
['D', 3]
>>> CartanType(["D", Integer(4), Integer(1)]).dual()
['D', 4, 1]
>>> CartanType(["E", Integer(6)]).dual()
['E', 6]
>>> CartanType(["E", Integer(6), Integer(1)]).dual()
['E', 6, 1]
```

is_simply_laced()

Return whether `self` is simply laced, which is `True`.

EXAMPLES:

```
sage: CartanType(['A',3,1]).is_simply_laced()
True
sage: CartanType(['A',2]).is_simply_laced()
True
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(3),Integer(1)]).is_simply_laced()
True
>>> CartanType(['A',Integer(2)]).is_simply_laced()
True
```

class sage.combinat.root_system.cartan_type.**CartanType_standard**

Bases: `UniqueRepresentation`, `SageObject`

class sage.combinat.root_system.cartan_type.**CartanType_standard_affine**(*letter*, *n*, *affine=1*)

Bases: *CartanType_standard*, *CartanType_affine*

A concrete class for affine simple Cartan types.

index_set()

Implement *CartanType_abstract.index_set()*.

The index set for all standard affine Cartan types is of the form $\{0, \ldots, n\}$.

EXAMPLES:

```
sage: CartanType(['A', 5, 1]).index_set()
(0, 1, 2, 3, 4, 5)
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(5), Integer(1)]).index_set()
(0, 1, 2, 3, 4, 5)
```

rank()

Return the rank of `self` which for type $X_n^{(1)}$ is $n + 1$.

EXAMPLES:

```
sage: CartanType(['A', 4, 1]).rank()
5
sage: CartanType(['B', 4, 1]).rank()
5
sage: CartanType(['C', 3, 1]).rank()
4
sage: CartanType(['D', 4, 1]).rank()
5
sage: CartanType(['E', 6, 1]).rank()
7
sage: CartanType(['E', 7, 1]).rank()
8
sage: CartanType(['F', 4, 1]).rank()
5
sage: CartanType(['G', 2, 1]).rank()
3
sage: CartanType(['A', 2, 2]).rank()
2
sage: CartanType(['A', 6, 2]).rank()
4
sage: CartanType(['A', 7, 2]).rank()
5
sage: CartanType(['D', 5, 2]).rank()
5
sage: CartanType(['E', 6, 2]).rank()
5
sage: CartanType(['D', 4, 3]).rank()
3
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(4), Integer(1)]).rank()
5
>>> CartanType(['B', Integer(4), Integer(1)]).rank()
5
>>> CartanType(['C', Integer(3), Integer(1)]).rank()
4
>>> CartanType(['D', Integer(4), Integer(1)]).rank()
5
>>> CartanType(['E', Integer(6), Integer(1)]).rank()
7
```

(continues on next page)

(continued from previous page)

```
>>> CartanType(['E', Integer(7), Integer(1)]).rank()
8
>>> CartanType(['F', Integer(4), Integer(1)]).rank()
5
>>> CartanType(['G', Integer(2), Integer(1)]).rank()
3
>>> CartanType(['A', Integer(2), Integer(2)]).rank()
2
>>> CartanType(['A', Integer(6), Integer(2)]).rank()
4
>>> CartanType(['A', Integer(7), Integer(2)]).rank()
5
>>> CartanType(['D', Integer(5), Integer(2)]).rank()
5
>>> CartanType(['E', Integer(6), Integer(2)]).rank()
5
>>> CartanType(['D', Integer(4), Integer(3)]).rank()
3
```

special_node()

Implement `CartanType_abstract.special_node()`.

With the standard labelling conventions, 0 is always a special node.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).special_node()
0
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).special_node()
0
```

type()

Return the type of `self`.

EXAMPLES:

```
sage: CartanType(['A', 4, 1]).type()
'A'
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(4), Integer(1)]).type()
'A'
```

class sage.combinat.root_system.cartan_type.**CartanType_standard_finite**(*letter*, *n*)

Bases: *CartanType_standard*, *CartanType_finite*

A concrete base class for the finite standard Cartan types.

This includes for example A_3, D_4, or E_8.

affine()

Return the corresponding untwisted affine Cartan type.

EXAMPLES:

```
sage: CartanType(['A',3]).affine()
['A', 3, 1]
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(3)]).affine()
['A', 3, 1]
```

coxeter_number()

Return the Coxeter number associated with `self`.

The Coxeter number is the order of a Coxeter element of the corresponding Weyl group.

See Bourbaki, Lie Groups and Lie Algebras V.6.1 or Wikipedia article Coxeter_element for more information.

EXAMPLES:

```
sage: CartanType(['A',4]).coxeter_number()
5
sage: CartanType(['B',4]).coxeter_number()
8
sage: CartanType(['C',4]).coxeter_number()
8
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).coxeter_number()
5
>>> CartanType(['B',Integer(4)]).coxeter_number()
8
>>> CartanType(['C',Integer(4)]).coxeter_number()
8
```

dual_coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).dual_coxeter_number()
5
sage: CartanType(['B',4]).dual_coxeter_number()
7
sage: CartanType(['C',4]).dual_coxeter_number()
5
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).dual_coxeter_number()
5
>>> CartanType(['B',Integer(4)]).dual_coxeter_number()
7
>>> CartanType(['C',Integer(4)]).dual_coxeter_number()
5
```

index_set()

Implement *CartanType_abstract.index_set()*.

The index set for all standard finite Cartan types is of the form $\{1, \ldots, n\}$. (See *type_I* for a slight abuse of this).

EXAMPLES:

```
sage: CartanType(['A', 5]).index_set()
(1, 2, 3, 4, 5)
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(5)]).index_set()
(1, 2, 3, 4, 5)
```

opposition_automorphism()

Return the opposition automorphism.

The *opposition automorphism* is the automorphism $i \mapsto i^*$ of the vertices Dynkin diagram such that, for w_0 the longest element of the Weyl group, and any simple root α_i, one has $\alpha_{i^*} = -w_0(\alpha_i)$.

The automorphism is returned as a Family.

EXAMPLES:

```
sage: ct = CartanType(['A', 5])
sage: ct.opposition_automorphism()                                          #␣
↪needs sage.libs.gap
Finite family {1: 5, 2: 4, 3: 3, 4: 2, 5: 1}

sage: ct = CartanType(['D', 4])
sage: ct.opposition_automorphism()                                          #␣
↪needs sage.libs.gap
Finite family {1: 1, 2: 2, 3: 3, 4: 4}

sage: ct = CartanType(['D', 5])
sage: ct.opposition_automorphism()                                          #␣
↪needs sage.libs.gap
Finite family {1: 1, 2: 2, 3: 3, 4: 5, 5: 4}

sage: ct = CartanType(['C', 4])
sage: ct.opposition_automorphism()                                          #␣
↪needs sage.libs.gap
Finite family {1: 1, 2: 2, 3: 3, 4: 4}
```

```
>>> from sage.all import *
>>> ct = CartanType(['A', Integer(5)])
>>> ct.opposition_automorphism()                                          #␣
↪needs sage.libs.gap
Finite family {1: 5, 2: 4, 3: 3, 4: 2, 5: 1}

>>> ct = CartanType(['D', Integer(4)])
>>> ct.opposition_automorphism()                                          #␣
↪needs sage.libs.gap
Finite family {1: 1, 2: 2, 3: 3, 4: 4}
```

(continues on next page)

(continued from previous page)

```
>>> ct = CartanType(['D', Integer(5)])
>>> ct.opposition_automorphism()                                    #␣
↪needs sage.libs.gap
Finite family {1: 1, 2: 2, 3: 3, 4: 5, 5: 4}

>>> ct = CartanType(['C', Integer(4)])
>>> ct.opposition_automorphism()                                    #␣
↪needs sage.libs.gap
Finite family {1: 1, 2: 2, 3: 3, 4: 4}
```

rank()

Return the rank of `self` which for type X_n is n.

EXAMPLES:

```
sage: CartanType(['A', 3]).rank()
3
sage: CartanType(['B', 3]).rank()
3
sage: CartanType(['C', 3]).rank()
3
sage: CartanType(['D', 4]).rank()
4
sage: CartanType(['E', 6]).rank()
6
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3)]).rank()
3
>>> CartanType(['B', Integer(3)]).rank()
3
>>> CartanType(['C', Integer(3)]).rank()
3
>>> CartanType(['D', Integer(4)]).rank()
4
>>> CartanType(['E', Integer(6)]).rank()
6
```

type()

Return the type of `self`.

EXAMPLES:

```
sage: CartanType(['A', 4]).type()
'A'
sage: CartanType(['A', 4, 1]).type()
'A'
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(4)]).type()
'A'
```

(continues on next page)

(continued from previous page)

```
>>> CartanType(['A', Integer(4), Integer(1)]).type()
'A'
```

class sage.combinat.root_system.cartan_type.**CartanType_standard_untwisted_affine**(*letter*, *n*, *affine=1*)

Bases: *CartanType_standard_affine*

A concrete class for the standard untwisted affine Cartan types.

basic_untwisted()

Return the basic_untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic_untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain self.

EXAMPLES:

```
sage: CartanType(['A', 1, 1]).basic_untwisted()
['A', 1]
sage: CartanType(['A', 3, 1]).basic_untwisted()
['A', 3]
sage: CartanType(['B', 3, 1]).basic_untwisted()
['B', 3]
sage: CartanType(['E', 6, 1]).basic_untwisted()
['E', 6]
sage: CartanType(['G', 2, 1]).basic_untwisted()
['G', 2]
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(1), Integer(1)]).basic_untwisted()
['A', 1]
>>> CartanType(['A', Integer(3), Integer(1)]).basic_untwisted()
['A', 3]
>>> CartanType(['B', Integer(3), Integer(1)]).basic_untwisted()
['B', 3]
>>> CartanType(['E', Integer(6), Integer(1)]).basic_untwisted()
['E', 6]
>>> CartanType(['G', Integer(2), Integer(1)]).basic_untwisted()
['G', 2]
```

classical()

Return the classical Cartan type associated with self.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).classical()
['A', 3]
sage: CartanType(['B', 3, 1]).classical()
['B', 3]
sage: CartanType(['C', 3, 1]).classical()
['C', 3]
sage: CartanType(['D', 4, 1]).classical()
['D', 4]
sage: CartanType(['E', 6, 1]).classical()
```

(continues on next page)

(continued from previous page)

```
['E', 6]
sage: CartanType(['F', 4, 1]).classical()
['F', 4]
sage: CartanType(['G', 2, 1]).classical()
['G', 2]
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(3), Integer(1)]).classical()
['A', 3]
>>> CartanType(['B', Integer(3), Integer(1)]).classical()
['B', 3]
>>> CartanType(['C', Integer(3), Integer(1)]).classical()
['C', 3]
>>> CartanType(['D', Integer(4), Integer(1)]).classical()
['D', 4]
>>> CartanType(['E', Integer(6), Integer(1)]).classical()
['E', 6]
>>> CartanType(['F', Integer(4), Integer(1)]).classical()
['F', 4]
>>> CartanType(['G', Integer(2), Integer(1)]).classical()
['G', 2]
```

is_untwisted_affine()

Implement `CartanType_affine.is_untwisted_affine()` by returning `True`.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).is_untwisted_affine()
True
```

```
>>> from sage.all import *
>>> CartanType(['B', Integer(3), Integer(1)]).is_untwisted_affine()
True
```

class sage.combinat.root_system.cartan_type.**SuperCartanType_standard**

Bases: `UniqueRepresentation`, `SageObject`

options = Current options for CartanType - dual_latex: \vee - dual_str: * - latex_marked: True - latex_relabel: True - mark_special_node: none - marked_node_str: X - notation: Stembridge - special_node_str: @

5.1.227 Coxeter Groups

sage.combinat.root_system.coxeter_group.**CoxeterGroup**(*data*, *implementation='reflection'*, *base_ring=None*, *index_set=None*)

Return an implementation of the Coxeter group given by `data`.

INPUT:

- `data` – a Cartan type (or coercible into; see `CartanType`) or a Coxeter matrix or graph
- `implementation` – (default: `'reflection'`) can be one of the following:
  - `'permutation'` – as a permutation representation

- 'matrix' – as a Weyl group (as a matrix group acting on the root space); if this is not implemented, this uses the "reflection" implementation
- 'coxeter3' – using the coxeter3 package
- 'reflection' – as elements in the reflection representation; see CoxeterMatrixGroup

* base_ring – (optional) the base ring for the 'reflection' implementation
* index_set – (optional) the index set for the 'reflection' implementation

EXAMPLES:

Now assume that data represents a Cartan type. If implementation is not specified, the reflection representation is returned:

```
sage: W = CoxeterGroup(["A",2]); W                                          #␣
↪needs sage.libs.gap
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3]
[3 1]

sage: W = CoxeterGroup(["A",3,1]); W                                        #␣
↪needs sage.libs.gap
Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2 3]
[3 1 3 2]
[2 3 1 3]
[3 2 3 1]

sage: W = CoxeterGroup(['H',3]); W                                          #␣
↪needs sage.libs.gap
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 5
 with a = 2.236067977499790? with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(["A",Integer(2)]); W                                    ␣
↪     # needs sage.libs.gap
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3]
[3 1]

>>> W = CoxeterGroup(["A",Integer(3),Integer(1)]); W                         ␣
↪              # needs sage.libs.gap
Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2 3]
[3 1 3 2]
[2 3 1 3]
[3 2 3 1]

>>> W = CoxeterGroup(['H',Integer(3)]); W                                    ␣
↪     # needs sage.libs.gap
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 5
```

(continues on next page)

(continued from previous page)

```
 with a = 2.236067977499790? with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]
```

We now use the `implementation` option:

```
sage: W = CoxeterGroup(["A",2], implementation='permutation'); W    # optional -␣
↪gap3
Permutation Group with generators [(1,4)(2,3)(5,6), (1,3)(2,5)(4,6)]
sage: W.category()                                                  # optional -␣
↪gap3
Join of Category of finite enumerated permutation groups
    and Category of finite Weyl groups
    and Category of well generated finite irreducible complex reflection groups

sage: W = CoxeterGroup(["A",2], implementation='matrix'); W                   #␣
↪needs sage.libs.gap
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)

sage: W = CoxeterGroup(["H",3], implementation='matrix'); W                   #␣
↪needs sage.libs.gap sage.rings.number_field
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 5
 with a = 2.236067977499790? with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]

sage: W = CoxeterGroup(["H",3], implementation='reflection'); W               #␣
↪needs sage.libs.gap sage.rings.number_field
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 5
 with a = 2.236067977499790? with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]

sage: W = CoxeterGroup(["A",4,1], implementation='permutation')               #␣
↪needs sage.libs.gap
Traceback (most recent call last):
...
ValueError: the type must be finite

sage: W = CoxeterGroup(["A",4], implementation='chevie'); W         # optional -␣
↪gap3
Irreducible real reflection group of rank 4 and type A4
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(["A",Integer(2)], implementation='permutation'); W    #␣
↪optional - gap3
Permutation Group with generators [(1,4)(2,3)(5,6), (1,3)(2,5)(4,6)]
>>> W.category()                                                  # optional -␣
↪gap3
```

(continues on next page)

(continued from previous page)

```
Join of Category of finite enumerated permutation groups
    and Category of finite Weyl groups
    and Category of well generated finite irreducible complex reflection groups

>>> W = CoxeterGroup(["A",Integer(2)], implementation='matrix'); W                  ␣
↪     # needs sage.libs.gap
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)

>>> W = CoxeterGroup(["H",Integer(3)], implementation='matrix'); W                  ␣
↪     # needs sage.libs.gap sage.rings.number_field
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 5
 with a = 2.236067977499790? with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]

>>> W = CoxeterGroup(["H",Integer(3)], implementation='reflection'); W              ␣
↪     # needs sage.libs.gap sage.rings.number_field
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 5
 with a = 2.236067977499790? with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]

>>> W = CoxeterGroup(["A",Integer(4),Integer(1)], implementation='permutation')  ␣
↪              # needs sage.libs.gap
Traceback (most recent call last):
...
ValueError: the type must be finite

>>> W = CoxeterGroup(["A",Integer(4)], implementation='chevie'); W          #␣
↪optional - gap3
Irreducible real reflection group of rank 4 and type A4
```

We use the different options for the “reflection” implementation:

```
sage: W = CoxeterGroup(["H",3], implementation='reflection', base_ring=RR); W    #␣
↪needs sage.libs.gap
Finite Coxeter group over Real Field with 53 bits of precision with Coxeter␣
↪matrix:
[1 3 2]
[3 1 5]
[2 5 1]
sage: W = CoxeterGroup([[1,10],[10,1]], implementation='reflection',            #␣
↪needs sage.symbolics
....:                  index_set=['a','b'], base_ring=SR); W
Finite Coxeter group over Symbolic Ring with Coxeter matrix:
[ 1 10]
[10  1]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(["H",Integer(3)], implementation='reflection', base_ring=RR);
```

(continues on next page)

(continued from previous page)

```
↪ W   # needs sage.libs.gap
Finite Coxeter group over Real Field with 53 bits of precision with Coxeter␣
↪matrix:
[1 3 2]
[3 1 5]
[2 5 1]
>>> W = CoxeterGroup([[Integer(1),Integer(10)],[Integer(10),Integer(1)]],␣
↪implementation='reflection',            # needs sage.symbolics
...                      index_set=['a','b'], base_ring=SR); W
Finite Coxeter group over Symbolic Ring with Coxeter matrix:
[ 1 10]
[10  1]
```

5.1.228 Coxeter Matrices

class sage.combinat.root_system.coxeter_matrix.**CoxeterMatrix**(*parent*, *data*, *coxeter_type*, *index_set*)

Bases: *CoxeterType*

A Coxeter matrix.

A Coxeter matrix $M = (m_{ij})_{i,j \in I}$ is a matrix encoding a Coxeter system (W, S), where the relations are given by $(s_i s_j)^{m_{ij}}$. Thus M is symmetric and has entries in $\{1, 2, 3, \ldots, \infty\}$ with $m_{ij} = 1$ if and only if $i = j$.

We represent $m_{ij} = \infty$ by any number $m_{ij} \leq -1$. In particular, we can construct a bilinear form $B = (b_{ij})_{i,j \in I}$ from M by

$$b_{ij} = \begin{cases} m_{ij} & m_{ij} < 0 \text{ (i.e., } m_{ij} = \infty\text{)}, \\ -\cos\left(\frac{\pi}{m_{ij}}\right) & \text{otherwise.} \end{cases}$$

EXAMPLES:

```
sage: CoxeterMatrix(['A', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
sage: CoxeterMatrix(['B', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
sage: CoxeterMatrix(['C', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
sage: CoxeterMatrix(['D', 4])
[1 3 2 2]
[3 1 3 3]
[2 3 1 2]
[2 3 2 1]
```

(continues on next page)

(continued from previous page)

```
sage: CoxeterMatrix(['E', 6])
[1 2 3 2 2 2]
[2 1 2 3 2 2]
[3 2 1 3 2 2]
[2 3 3 1 3 2]
[2 2 2 3 1 3]
[2 2 2 2 3 1]

sage: CoxeterMatrix(['F', 4])
[1 3 2 2]
[3 1 4 2]
[2 4 1 3]
[2 2 3 1]

sage: CoxeterMatrix(['G', 2])
[1 6]
[6 1]
```

```
>>> from sage.all import *
>>> CoxeterMatrix(['A', Integer(4)])
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
>>> CoxeterMatrix(['B', Integer(4)])
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
>>> CoxeterMatrix(['C', Integer(4)])
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
>>> CoxeterMatrix(['D', Integer(4)])
[1 3 2 2]
[3 1 3 3]
[2 3 1 2]
[2 3 2 1]

>>> CoxeterMatrix(['E', Integer(6)])
[1 2 3 2 2 2]
[2 1 2 3 2 2]
[3 2 1 3 2 2]
[2 3 3 1 3 2]
[2 2 2 3 1 3]
[2 2 2 2 3 1]

>>> CoxeterMatrix(['F', Integer(4)])
[1 3 2 2]
[3 1 4 2]
[2 4 1 3]
```

(continues on next page)

(continued from previous page)

```
[2 2 3 1]

>>> CoxeterMatrix(['G', Integer(2)])
[1 6]
[6 1]
```

By default, entries representing ∞ are given by -1 in the Coxeter matrix:

```
sage: G = Graph([(0,1,None), (1,2,4), (0,2,oo)])
sage: CoxeterMatrix(G)
[ 1  3 -1]
[ 3  1  4]
[-1  4  1]
```

```
>>> from sage.all import *
>>> G = Graph([(Integer(0),Integer(1),None), (Integer(1),Integer(2),Integer(4)),␣
↪(Integer(0),Integer(2),oo)])
>>> CoxeterMatrix(G)
[ 1  3 -1]
[ 3  1  4]
[-1  4  1]
```

It is possible to give a number ≤ -1 to represent an infinite label:

```
sage: CoxeterMatrix([[1,-1],[-1,1]])
[ 1 -1]
[-1  1]
sage: CoxeterMatrix([[1,-3/2],[-3/2,1]])
[   1 -3/2]
[-3/2    1]
```

```
>>> from sage.all import *
>>> CoxeterMatrix([[Integer(1),-Integer(1)],[-Integer(1),Integer(1)]])
[ 1 -1]
[-1  1]
>>> CoxeterMatrix([[Integer(1),-Integer(3)/Integer(2)],[-Integer(3)/Integer(2),
↪Integer(1)]])
[   1 -3/2]
[-3/2    1]
```

bilinear_form(*R=None*)

Return the bilinear form of `self`.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: CoxeterType(['A', 2, 1]).bilinear_form()
[   1 -1/2 -1/2]
[-1/2    1 -1/2]
[-1/2 -1/2    1]
sage: CoxeterType(['H', 3]).bilinear_form()
[                      1                    -1/2                       0]
```

(continues on next page)

(continued from previous page)

```
[                    -1/2                       1 1/2*E(5)^2 + 1/2*E(5)^3]
[                       0 1/2*E(5)^2 + 1/2*E(5)^3                       1]
sage: C = CoxeterMatrix([[1,-1,-1],[-1,1,-1],[-1,-1,1]])
sage: C.bilinear_form()
[ 1 -1 -1]
[-1  1 -1]
[-1 -1  1]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> CoxeterType(['A', Integer(2), Integer(1)]).bilinear_form()
[   1 -1/2 -1/2]
[-1/2    1 -1/2]
[-1/2 -1/2    1]
>>> CoxeterType(['H', Integer(3)]).bilinear_form()
[                       1                    -1/2                       0]
[                    -1/2                       1 1/2*E(5)^2 + 1/2*E(5)^3]
[                       0 1/2*E(5)^2 + 1/2*E(5)^3                       1]
>>> C = CoxeterMatrix([[Integer(1),-Integer(1),-Integer(1)],[-Integer(1),
↪Integer(1),-Integer(1)],[-Integer(1),-Integer(1),Integer(1)]])
>>> C.bilinear_form()
[ 1 -1 -1]
[-1  1 -1]
[-1 -1  1]
```

coxeter_graph()

Return the Coxeter graph of `self`.

EXAMPLES:

```
sage: C = CoxeterMatrix(['A',3])
sage: C.coxeter_graph()
Graph on 3 vertices

sage: C = CoxeterMatrix([['A',3],['A',1]])
sage: C.coxeter_graph()
Graph on 4 vertices
```

```
>>> from sage.all import *
>>> C = CoxeterMatrix(['A',Integer(3)])
>>> C.coxeter_graph()
Graph on 3 vertices

>>> C = CoxeterMatrix([['A',Integer(3)],['A',Integer(1)]])
>>> C.coxeter_graph()
Graph on 4 vertices
```

coxeter_matrix()

Return the Coxeter matrix of `self`.

EXAMPLES:

```
sage: CoxeterMatrix(['C',3]).coxeter_matrix()
[1 3 2]
[3 1 4]
[2 4 1]
```

```
>>> from sage.all import *
>>> CoxeterMatrix(['C',Integer(3)]).coxeter_matrix()
[1 3 2]
[3 1 4]
[2 4 1]
```

coxeter_type()

Return the Coxeter type of `self` or `self` if unknown.

EXAMPLES:

```
sage: C = CoxeterMatrix(['A',4,1])
sage: C.coxeter_type()
Coxeter type of ['A', 4, 1]
```

```
>>> from sage.all import *
>>> C = CoxeterMatrix(['A',Integer(4),Integer(1)])
>>> C.coxeter_type()
Coxeter type of ['A', 4, 1]
```

If the Coxeter type is unknown:

```
sage: C = CoxeterMatrix([[1,3,4], [3,1,-1], [4,-1,1]])
sage: C.coxeter_type()
[ 1  3  4]
[ 3  1 -1]
[ 4 -1  1]
```

```
>>> from sage.all import *
>>> C = CoxeterMatrix([[Integer(1),Integer(3),Integer(4)], [Integer(3),
↪Integer(1),-Integer(1)], [Integer(4),-Integer(1),Integer(1)]])
>>> C.coxeter_type()
[ 1  3  4]
[ 3  1 -1]
[ 4 -1  1]
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: C = CoxeterMatrix(['A',1,1])
sage: C.index_set()
(0, 1)
sage: C = CoxeterMatrix(['E',6])
sage: C.index_set()
(1, 2, 3, 4, 5, 6)
```

```
>>> from sage.all import *
>>> C = CoxeterMatrix(['A',Integer(1),Integer(1)])
>>> C.index_set()
(0, 1)
>>> C = CoxeterMatrix(['E',Integer(6)])
>>> C.index_set()
(1, 2, 3, 4, 5, 6)
```

is_affine()

Return if `self` is an affine type or `False` if unknown.

EXAMPLES:

```
sage: M = CoxeterMatrix(['C',4])
sage: M.is_affine()
False
sage: M = CoxeterMatrix(['D',4,1])
sage: M.is_affine()
True
sage: M = CoxeterMatrix([[1, 3],[3,1]])
sage: M.is_affine()
False
sage: M = CoxeterMatrix([[1, -1, 7], [-1, 1, 3], [7, 3, 1]])
sage: M.is_affine()
False
```

```
>>> from sage.all import *
>>> M = CoxeterMatrix(['C',Integer(4)])
>>> M.is_affine()
False
>>> M = CoxeterMatrix(['D',Integer(4),Integer(1)])
>>> M.is_affine()
True
>>> M = CoxeterMatrix([[Integer(1), Integer(3)],[Integer(3),Integer(1)]])
>>> M.is_affine()
False
>>> M = CoxeterMatrix([[Integer(1), -Integer(1), Integer(7)], [-Integer(1),␣
↪Integer(1), Integer(3)], [Integer(7), Integer(3), Integer(1)]])
>>> M.is_affine()
False
```

is_crystallographic()

Return whether `self` is crystallographic.

A Coxeter matrix is crystallographic if all non-diagonal entries are either 2, 3, 4, or 6.

EXAMPLES:

```
sage: CoxeterMatrix(['F',4]).is_crystallographic()
True
sage: CoxeterMatrix(['H',3]).is_crystallographic()
False
```

```
>>> from sage.all import *
>>> CoxeterMatrix(['F',Integer(4)]).is_crystallographic()
True
>>> CoxeterMatrix(['H',Integer(3)]).is_crystallographic()
False
```

is_finite()

Return if `self` is a finite type or `False` if unknown.

EXAMPLES:

```
sage: M = CoxeterMatrix(['C',4])
sage: M.is_finite()
True
sage: M = CoxeterMatrix(['D',4,1])
sage: M.is_finite()
False
sage: M = CoxeterMatrix([[1, -1], [-1, 1]])
sage: M.is_finite()
False
```

```
>>> from sage.all import *
>>> M = CoxeterMatrix(['C',Integer(4)])
>>> M.is_finite()
True
>>> M = CoxeterMatrix(['D',Integer(4),Integer(1)])
>>> M.is_finite()
False
>>> M = CoxeterMatrix([[Integer(1), -Integer(1)], [-Integer(1), Integer(1)]])
>>> M.is_finite()
False
```

is_irreducible()

Return whether `self` is irreducible.

A Coxeter matrix is irreducible if the Coxeter graph is connected.

EXAMPLES:

```
sage: CoxeterMatrix([['F',4],['A',1]]).is_irreducible()
False
sage: CoxeterMatrix(['H',3]).is_irreducible()
True
```

```
>>> from sage.all import *
>>> CoxeterMatrix([['F',Integer(4)],['A',Integer(1)]]).is_irreducible()
False
>>> CoxeterMatrix(['H',Integer(3)]).is_irreducible()
True
```

is_simply_laced()

Return if `self` is simply-laced.

A Coxeter matrix is simply-laced if all non-diagonal entries are either 2 or 3.

EXAMPLES:

```
sage: cm = CoxeterMatrix([[1,3,3,3], [3,1,3,3], [3,3,1,3], [3,3,3,1]])
sage: cm.is_simply_laced()
True
```

```
>>> from sage.all import *
>>> cm = CoxeterMatrix([[Integer(1),Integer(3),Integer(3),Integer(3)],␣
↪[Integer(3),Integer(1),Integer(3),Integer(3)], [Integer(3),Integer(3),
↪Integer(1),Integer(3)], [Integer(3),Integer(3),Integer(3),Integer(1)]])
>>> cm.is_simply_laced()
True
```

rank()

Return the rank of `self`.

EXAMPLES:

```
sage: CoxeterMatrix(['C',3]).rank()
3
sage: CoxeterMatrix(["A2","B2","F4"]).rank()
8
```

```
>>> from sage.all import *
>>> CoxeterMatrix(['C',Integer(3)]).rank()
3
>>> CoxeterMatrix(["A2","B2","F4"]).rank()
8
```

relabel(*relabelling*)

Return a relabelled copy of this Coxeter matrix.

INPUT:

- `relabelling` – a function (or dictionary)

OUTPUT:

an isomorphic Coxeter type obtained by relabelling the nodes of the Coxeter graph. Namely, the node with label `i` is relabelled `f(i)` (or, by `f[i]` if `f` is a dictionary).

EXAMPLES:

```
sage: CoxeterMatrix(['F',4]).relabel({ 1:2, 2:3, 3:4, 4:1})
[1 4 2 3]
[4 1 3 2]
[2 3 1 2]
[3 2 2 1]
sage: CoxeterMatrix(['F',4]).relabel(lambda x: x+1 if x<4 else 1)
[1 4 2 3]
[4 1 3 2]
[2 3 1 2]
[3 2 2 1]
```

```
>>> from sage.all import *
>>> CoxeterMatrix(['F',Integer(4)]).relabel({ Integer(1):Integer(2),␣
↪Integer(2):Integer(3), Integer(3):Integer(4), Integer(4):Integer(1)})
[1 4 2 3]
[4 1 3 2]
[2 3 1 2]
[3 2 2 1]
>>> CoxeterMatrix(['F',Integer(4)]).relabel(lambda x: x+Integer(1) if x
↪<Integer(4) else Integer(1))
[1 4 2 3]
[4 1 3 2]
[2 3 1 2]
[3 2 2 1]
```

classmethod samples(*finite=None*, *affine=None*, *crystallographic=None*, *higher_rank=None*)

Return a sample of the available Coxeter types.

INPUT:

- `finite` – (default: `None`) a boolean or `None`
- `affine` – (default: `None`) a boolean or `None`
- `crystallographic` – (default: `None`) a boolean or `None`
- `higher_rank` – (default: `None`) a boolean or `None`

The sample contains all the exceptional finite and affine Coxeter types, as well as typical representatives of the infinite families.

Here the `higher_rank` term denotes non-finite, non-affine, Coxeter groups (including hyperbolic types).

Todo

Implement the hyperbolic and compact hyperbolic in the samples.

EXAMPLES:

```
sage: [CM.coxeter_type() for CM in CoxeterMatrix.samples()]
[
Coxeter type of ['A', 1], Coxeter type of ['A', 5],

Coxeter type of ['B', 5], Coxeter type of ['D', 4],

Coxeter type of ['D', 5], Coxeter type of ['E', 6],

Coxeter type of ['E', 7], Coxeter type of ['E', 8],

Coxeter type of ['F', 4], Coxeter type of ['H', 3],

Coxeter type of ['H', 4], Coxeter type of ['I', 10],

Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],

Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
```

(continues on next page)

(continued from previous page)

```
Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],

Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],

                                                              [ 1 -1 -1]
                                                              [-1  1 -1]
Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1], [-1 -1  1],

          [ 1 -2  3  2]
[1 2 3]   [-2  1  2  3]
[2 1 7]   [ 3  2  1 -8]
[3 7 1], [ 2  3 -8  1]
]
```

```
>>> from sage.all import *
>>> [CM.coxeter_type() for CM in CoxeterMatrix.samples()]
[
Coxeter type of ['A', 1], Coxeter type of ['A', 5],
<BLANKLINE>
Coxeter type of ['B', 5], Coxeter type of ['D', 4],
<BLANKLINE>
Coxeter type of ['D', 5], Coxeter type of ['E', 6],
<BLANKLINE>
Coxeter type of ['E', 7], Coxeter type of ['E', 8],
<BLANKLINE>
Coxeter type of ['F', 4], Coxeter type of ['H', 3],
<BLANKLINE>
Coxeter type of ['H', 4], Coxeter type of ['I', 10],
<BLANKLINE>
Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
<BLANKLINE>
Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
<BLANKLINE>
Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
<BLANKLINE>
Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
<BLANKLINE>
                                                              [ 1 -1 -1]
                                                              [-1  1 -1]
Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1], [-1 -1  1],
<BLANKLINE>
          [ 1 -2  3  2]
[1 2 3]   [-2  1  2  3]
[2 1 7]   [ 3  2  1 -8]
[3 7 1], [ 2  3 -8  1]
]
```

The finite, affine and crystallographic options allow respectively for restricting to (non) finite, (non) affine, and (non) crystallographic Cartan types:

```
sage: [CM.coxeter_type() for CM in CoxeterMatrix.samples(finite=True)]
```

(continues on next page)

(continued from previous page)

```
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 5], Coxeter type of ['D', 4],
 Coxeter type of ['D', 5], Coxeter type of ['E', 6],
 Coxeter type of ['E', 7], Coxeter type of ['E', 8],
 Coxeter type of ['F', 4], Coxeter type of ['H', 3],
 Coxeter type of ['H', 4], Coxeter type of ['I', 10]]

sage: [CM.coxeter_type() for CM in CoxeterMatrix.samples(affine=True)]
[Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

sage: [CM.coxeter_type() for CM in CoxeterMatrix.
↪samples(crystallographic=True)]
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 5], Coxeter type of ['D', 4],
 Coxeter type of ['D', 5], Coxeter type of ['E', 6],
 Coxeter type of ['E', 7], Coxeter type of ['E', 8],
 Coxeter type of ['F', 4], Coxeter type of ['A', 2, 1],
 Coxeter type of ['B', 5, 1], Coxeter type of ['C', 5, 1],
 Coxeter type of ['D', 5, 1], Coxeter type of ['E', 6, 1],
 Coxeter type of ['E', 7, 1], Coxeter type of ['E', 8, 1],
 Coxeter type of ['F', 4, 1], Coxeter type of ['G', 2, 1]]

sage: CoxeterMatrix.samples(crystallographic=False)
[
         [1 3 2 2]
[1 3 2]  [3 1 3 2]                      [ 1 -1 -1]  [1 2 3]
[3 1 5]  [2 3 1 5]  [ 1 10]  [ 1 -1]  [-1  1 -1]  [2 1 7]
[2 5 1], [2 2 5 1], [10  1], [-1  1], [-1 -1  1], [3 7 1],

[ 1 -2  3  2]
[-2  1  2  3]
[ 3  2  1 -8]
[ 2  3 -8  1]
]
```

```
>>> from sage.all import *
>>> [CM.coxeter_type() for CM in CoxeterMatrix.samples(finite=True)]
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 5], Coxeter type of ['D', 4],
 Coxeter type of ['D', 5], Coxeter type of ['E', 6],
 Coxeter type of ['E', 7], Coxeter type of ['E', 8],
 Coxeter type of ['F', 4], Coxeter type of ['H', 3],
 Coxeter type of ['H', 4], Coxeter type of ['I', 10]]

>>> [CM.coxeter_type() for CM in CoxeterMatrix.samples(affine=True)]
[Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
```

(continues on next page)

(continued from previous page)

```
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

>>> [CM.coxeter_type() for CM in CoxeterMatrix.samples(crystallographic=True)]
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 5], Coxeter type of ['D', 4],
 Coxeter type of ['D', 5], Coxeter type of ['E', 6],
 Coxeter type of ['E', 7], Coxeter type of ['E', 8],
 Coxeter type of ['F', 4], Coxeter type of ['A', 2, 1],
 Coxeter type of ['B', 5, 1], Coxeter type of ['C', 5, 1],
 Coxeter type of ['D', 5, 1], Coxeter type of ['E', 6, 1],
 Coxeter type of ['E', 7, 1], Coxeter type of ['E', 8, 1],
 Coxeter type of ['F', 4, 1], Coxeter type of ['G', 2, 1]]

>>> CoxeterMatrix.samples(crystallographic=False)
[
          [1 3 2 2]
[1 3 2]   [3 1 3 2]                        [ 1 -1 -1]  [1 2 3]
[3 1 5]   [2 3 1 5]   [ 1 10]   [ 1 -1]   [-1  1 -1]  [2 1 7]
[2 5 1], [2 2 5 1], [10  1], [-1  1], [-1 -1  1], [3 7 1],
<BLANKLINE>
[ 1 -2  3  2]
[-2  1  2  3]
[ 3  2  1 -8]
[ 2  3 -8  1]
]
```

Todo

add some reducible Coxeter types (suggestions?)

sage.combinat.root_system.coxeter_matrix.**check_coxeter_matrix**(*m*)

Check if m represents a generalized Coxeter matrix and raise and error if not.

EXAMPLES:

```
sage: from sage.combinat.root_system.coxeter_matrix import check_coxeter_matrix
sage: m = matrix([[1,3,2],[3,1,-1],[2,-1,1]])
sage: check_coxeter_matrix(m)

sage: m = matrix([[1,3],[3,1],[2,-1]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: not a square matrix

sage: m = matrix([[1,3,2],[3,1,-1],[2,-1,2]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: the matrix diagonal is not all 1
```

(continues on next page)

(continued from previous page)

```
sage: m = matrix([[1,3,3],[3,1,-1],[2,-1,1]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: the matrix is not symmetric

sage: m = matrix([[1,3,1/2],[3,1,-1],[1/2,-1,1]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: invalid Coxeter label 1/2

sage: m = matrix([[1,3,1],[3,1,-1],[1,-1,1]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: invalid Coxeter label 1
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.coxeter_matrix import check_coxeter_matrix
>>> m = matrix([[Integer(1),Integer(3),Integer(2)],[Integer(3),Integer(1),-
↪Integer(1)],[Integer(2),-Integer(1),Integer(1)]])
>>> check_coxeter_matrix(m)

>>> m = matrix([[Integer(1),Integer(3)],[Integer(3),Integer(1)],[Integer(2),-
↪Integer(1)]])
>>> check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: not a square matrix

>>> m = matrix([[Integer(1),Integer(3),Integer(2)],[Integer(3),Integer(1),-
↪Integer(1)],[Integer(2),-Integer(1),Integer(2)]])
>>> check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: the matrix diagonal is not all 1

>>> m = matrix([[Integer(1),Integer(3),Integer(3)],[Integer(3),Integer(1),-
↪Integer(1)],[Integer(2),-Integer(1),Integer(1)]])
>>> check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: the matrix is not symmetric

>>> m = matrix([[Integer(1),Integer(3),Integer(1)/Integer(2)],[Integer(3),
↪Integer(1),-Integer(1)],[Integer(1)/Integer(2),-Integer(1),Integer(1)]])
>>> check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: invalid Coxeter label 1/2
```

(continues on next page)

(continued from previous page)

```
>>> m = matrix([[Integer(1),Integer(3),Integer(1)],[Integer(3),Integer(1),-
↪Integer(1)],[Integer(1),-Integer(1),Integer(1)]])
>>> check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: invalid Coxeter label 1
```

sage.combinat.root_system.coxeter_matrix.**coxeter_matrix_as_function**(*t*)

Return the Coxeter matrix, as a function.

INPUT:

- t – a Cartan type

EXAMPLES:

```
sage: from sage.combinat.root_system.coxeter_matrix import coxeter_matrix_as_
↪function
sage: f = coxeter_matrix_as_function(['A',4])
sage: matrix([[f(i,j) for j in range(1,5)] for i in range(1,5)])
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.coxeter_matrix import coxeter_matrix_as_
↪function
>>> f = coxeter_matrix_as_function(['A',Integer(4)])
>>> matrix([[f(i,j) for j in range(Integer(1),Integer(5))] for i in␣
↪range(Integer(1),Integer(5))])
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
```

sage.combinat.root_system.coxeter_matrix.**recognize_coxeter_type_from_matrix**(*coxeter_matrix*, *index_set*)

Return the Coxeter type of coxeter_matrix if known, otherwise return None.

EXAMPLES:

Some infinite ones:

```
sage: C = CoxeterMatrix([[1,3,2],[3,1,-1],[2,-1,1]])
sage: C.is_finite()  # indirect doctest
False
sage: C = CoxeterMatrix([[1,-1,-1],[-1,1,-1],[-1,-1,1]])
sage: C.is_finite()  # indirect doctest
False
```

```
>>> from sage.all import *
>>> C = CoxeterMatrix([[Integer(1),Integer(3),Integer(2)],[Integer(3),Integer(1),-
↪Integer(1)],[Integer(2),-Integer(1),Integer(1)]])
>>> C.is_finite()  # indirect doctest
False
>>> C = CoxeterMatrix([[Integer(1),-Integer(1),-Integer(1)],[-Integer(1),
↪Integer(1),-Integer(1)],[-Integer(1),-Integer(1),Integer(1)]])
>>> C.is_finite()  # indirect doctest
False
```

Some finite ones:

```
sage: m = matrix(CoxeterMatrix(['D', 4]))
sage: CoxeterMatrix(m).is_finite()  # indirect doctest
True
sage: m = matrix(CoxeterMatrix(['H', 4]))
sage: CoxeterMatrix(m).is_finite()  # indirect doctest
True

sage: CoxeterMatrix(CoxeterType(['A',10]).coxeter_graph()).coxeter_type()
Coxeter type of ['A', 10]
sage: CoxeterMatrix(CoxeterType(['B',10]).coxeter_graph()).coxeter_type()
Coxeter type of ['B', 10]
sage: CoxeterMatrix(CoxeterType(['C',10]).coxeter_graph()).coxeter_type()
Coxeter type of ['B', 10]
sage: CoxeterMatrix(CoxeterType(['D',10]).coxeter_graph()).coxeter_type()
Coxeter type of ['D', 10]
sage: CoxeterMatrix(CoxeterType(['E',6]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 6]
sage: CoxeterMatrix(CoxeterType(['E',7]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 7]
sage: CoxeterMatrix(CoxeterType(['E',8]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 8]
sage: CoxeterMatrix(CoxeterType(['F',4]).coxeter_graph()).coxeter_type()
Coxeter type of ['F', 4]
sage: CoxeterMatrix(CoxeterType(['G',2]).coxeter_graph()).coxeter_type()
Coxeter type of ['G', 2]
sage: CoxeterMatrix(CoxeterType(['H',3]).coxeter_graph()).coxeter_type()
Coxeter type of ['H', 3]
sage: CoxeterMatrix(CoxeterType(['H',4]).coxeter_graph()).coxeter_type()
Coxeter type of ['H', 4]
sage: CoxeterMatrix(CoxeterType(['I',100]).coxeter_graph()).coxeter_type()
Coxeter type of ['I', 100]
```

```
>>> from sage.all import *
>>> m = matrix(CoxeterMatrix(['D', Integer(4)]))
>>> CoxeterMatrix(m).is_finite()  # indirect doctest
True
>>> m = matrix(CoxeterMatrix(['H', Integer(4)]))
>>> CoxeterMatrix(m).is_finite()  # indirect doctest
True

>>> CoxeterMatrix(CoxeterType(['A',Integer(10)]).coxeter_graph()).coxeter_type()
```

(continues on next page)

(continued from previous page)

```
Coxeter type of ['A', 10]
>>> CoxeterMatrix(CoxeterType(['B',Integer(10)]).coxeter_graph()).coxeter_type()
Coxeter type of ['B', 10]
>>> CoxeterMatrix(CoxeterType(['C',Integer(10)]).coxeter_graph()).coxeter_type()
Coxeter type of ['B', 10]
>>> CoxeterMatrix(CoxeterType(['D',Integer(10)]).coxeter_graph()).coxeter_type()
Coxeter type of ['D', 10]
>>> CoxeterMatrix(CoxeterType(['E',Integer(6)]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 6]
>>> CoxeterMatrix(CoxeterType(['E',Integer(7)]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 7]
>>> CoxeterMatrix(CoxeterType(['E',Integer(8)]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 8]
>>> CoxeterMatrix(CoxeterType(['F',Integer(4)]).coxeter_graph()).coxeter_type()
Coxeter type of ['F', 4]
>>> CoxeterMatrix(CoxeterType(['G',Integer(2)]).coxeter_graph()).coxeter_type()
Coxeter type of ['G', 2]
>>> CoxeterMatrix(CoxeterType(['H',Integer(3)]).coxeter_graph()).coxeter_type()
Coxeter type of ['H', 3]
>>> CoxeterMatrix(CoxeterType(['H',Integer(4)]).coxeter_graph()).coxeter_type()
Coxeter type of ['H', 4]
>>> CoxeterMatrix(CoxeterType(['I',Integer(100)]).coxeter_graph()).coxeter_type()
Coxeter type of ['I', 100]
```

Some affine graphs:

```
sage: CoxeterMatrix(CoxeterType(['A',1,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['A', 1, 1]
sage: CoxeterMatrix(CoxeterType(['A',10,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['A', 10, 1]
sage: CoxeterMatrix(CoxeterType(['B',10,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['B', 10, 1]
sage: CoxeterMatrix(CoxeterType(['C',10,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['C', 10, 1]
sage: CoxeterMatrix(CoxeterType(['D',10,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['D', 10, 1]
sage: CoxeterMatrix(CoxeterType(['E',6,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 6, 1]
sage: CoxeterMatrix(CoxeterType(['E',7,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 7, 1]
sage: CoxeterMatrix(CoxeterType(['E',8,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 8, 1]
sage: CoxeterMatrix(CoxeterType(['F',4,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['F', 4, 1]
sage: CoxeterMatrix(CoxeterType(['G',2,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['G', 2, 1]
```

```
>>> from sage.all import *
>>> CoxeterMatrix(CoxeterType(['A',Integer(1),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['A', 1, 1]
>>> CoxeterMatrix(CoxeterType(['A',Integer(10),Integer(1)]).coxeter_graph()).
```

(continues on next page)

(continued from previous page)

```
↪coxeter_type()
Coxeter type of ['A', 10, 1]
>>> CoxeterMatrix(CoxeterType(['B',Integer(10),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['B', 10, 1]
>>> CoxeterMatrix(CoxeterType(['C',Integer(10),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['C', 10, 1]
>>> CoxeterMatrix(CoxeterType(['D',Integer(10),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['D', 10, 1]
>>> CoxeterMatrix(CoxeterType(['E',Integer(6),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['E', 6, 1]
>>> CoxeterMatrix(CoxeterType(['E',Integer(7),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['E', 7, 1]
>>> CoxeterMatrix(CoxeterType(['E',Integer(8),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['E', 8, 1]
>>> CoxeterMatrix(CoxeterType(['F',Integer(4),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['F', 4, 1]
>>> CoxeterMatrix(CoxeterType(['G',Integer(2),Integer(1)]).coxeter_graph()).
↪coxeter_type()
Coxeter type of ['G', 2, 1]
```

5.1.229 Coxeter Types

class sage.combinat.root_system.coxeter_type.**CoxeterType**

Bases: SageObject

Abstract class for Coxeter types.

bilinear_form(*R=None*)

Return the bilinear form over R associated to self.

INPUT:

- R – (default: universal cyclotomic field) a ring used to compute the bilinear form

EXAMPLES:

```
sage: # needs sage.graphs sage.libs.gap
sage: CoxeterType(['A', 2, 1]).bilinear_form()
[   1 -1/2 -1/2]
[-1/2    1 -1/2]
[-1/2 -1/2    1]
sage: CoxeterType(['H', 3]).bilinear_form()
[                      1                    -1/2                       0]
[                   -1/2                       1 1/2*E(5)^2 + 1/2*E(5)^3]
[                      0 1/2*E(5)^2 + 1/2*E(5)^3                       1]
sage: C = CoxeterMatrix([[1,-1,-1],[-1,1,-1],[-1,-1,1]])
sage: C.bilinear_form()
```

(continues on next page)

(continued from previous page)

```
[ 1 -1 -1]
[-1  1 -1]
[-1 -1  1]
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap
>>> CoxeterType(['A', Integer(2), Integer(1)]).bilinear_form()
[   1 -1/2 -1/2]
[-1/2    1 -1/2]
[-1/2 -1/2    1]
>>> CoxeterType(['H', Integer(3)]).bilinear_form()
[                      1                    -1/2                       0]
[                   -1/2                       1 1/2*E(5)^2 + 1/2*E(5)^3]
[                      0 1/2*E(5)^2 + 1/2*E(5)^3                       1]
>>> C = CoxeterMatrix([[Integer(1),-Integer(1),-Integer(1)],[-Integer(1),
↪Integer(1),-Integer(1)],[-Integer(1),-Integer(1),Integer(1)]])
>>> C.bilinear_form()
[ 1 -1 -1]
[-1  1 -1]
[-1 -1  1]
```

coxeter_graph()

Return the Coxeter graph associated to self.

EXAMPLES:

```
sage: CoxeterType(['A', 3]).coxeter_graph()                                 #␣
↪needs sage.graphs
Graph on 3 vertices
sage: CoxeterType(['A', 3, 1]).coxeter_graph()                              #␣
↪needs sage.graphs
Graph on 4 vertices
```

```
>>> from sage.all import *
>>> CoxeterType(['A', Integer(3)]).coxeter_graph()                         ␣
↪     # needs sage.graphs
Graph on 3 vertices
>>> CoxeterType(['A', Integer(3), Integer(1)]).coxeter_graph()             ␣
↪              # needs sage.graphs
Graph on 4 vertices
```

coxeter_matrix()

Return the Coxeter matrix associated to self.

EXAMPLES:

```
sage: CoxeterType(['A', 3]).coxeter_matrix()                                #␣
↪needs sage.graphs
[1 3 2]
[3 1 3]
[2 3 1]
sage: CoxeterType(['A', 3, 1]).coxeter_matrix()                             #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[1 3 2 3]
[3 1 3 2]
[2 3 1 3]
[3 2 3 1]
```

```
>>> from sage.all import *
>>> CoxeterType(['A', Integer(3)]).coxeter_matrix()                    ␣
↪      # needs sage.graphs
[1 3 2]
[3 1 3]
[2 3 1]
>>> CoxeterType(['A', Integer(3), Integer(1)]).coxeter_matrix()        ␣
↪               # needs sage.graphs
[1 3 2 3]
[3 1 3 2]
[2 3 1 3]
[3 2 3 1]
```

index_set()

Return the index set for `self`.

This is the list of the nodes of the associated Coxeter graph.

EXAMPLES:

```
sage: CoxeterType(['A', 3, 1]).index_set()
(0, 1, 2, 3)
sage: CoxeterType(['D', 4]).index_set()
(1, 2, 3, 4)
sage: CoxeterType(['A', 7, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CoxeterType(['A', 7, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CoxeterType(['A', 6, 2]).index_set()
(0, 1, 2, 3)
sage: CoxeterType(['D', 6, 2]).index_set()
(0, 1, 2, 3, 4, 5)
sage: CoxeterType(['E', 6, 1]).index_set()
(0, 1, 2, 3, 4, 5, 6)
sage: CoxeterType(['E', 6, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CoxeterType(['A', 2, 2]).index_set()
(0, 1)
sage: CoxeterType(['G', 2, 1]).index_set()
(0, 1, 2)
sage: CoxeterType(['F', 4, 1]).index_set()
(0, 1, 2, 3, 4)
```

```
>>> from sage.all import *
>>> CoxeterType(['A', Integer(3), Integer(1)]).index_set()
(0, 1, 2, 3)
```

(continues on next page)

(continued from previous page)

```
>>> CoxeterType(['D', Integer(4)]).index_set()
(1, 2, 3, 4)
>>> CoxeterType(['A', Integer(7), Integer(2)]).index_set()
(0, 1, 2, 3, 4)
>>> CoxeterType(['A', Integer(7), Integer(2)]).index_set()
(0, 1, 2, 3, 4)
>>> CoxeterType(['A', Integer(6), Integer(2)]).index_set()
(0, 1, 2, 3)
>>> CoxeterType(['D', Integer(6), Integer(2)]).index_set()
(0, 1, 2, 3, 4, 5)
>>> CoxeterType(['E', Integer(6), Integer(1)]).index_set()
(0, 1, 2, 3, 4, 5, 6)
>>> CoxeterType(['E', Integer(6), Integer(2)]).index_set()
(0, 1, 2, 3, 4)
>>> CoxeterType(['A', Integer(2), Integer(2)]).index_set()
(0, 1)
>>> CoxeterType(['G', Integer(2), Integer(1)]).index_set()
(0, 1, 2)
>>> CoxeterType(['F', Integer(4), Integer(1)]).index_set()
(0, 1, 2, 3, 4)
```

is_affine()

Return whether `self` is affine.

EXAMPLES:

```
sage: CoxeterType(['A', 3]).is_affine()
False
sage: CoxeterType(['A', 3, 1]).is_affine()
True
```

```
>>> from sage.all import *
>>> CoxeterType(['A', Integer(3)]).is_affine()
False
>>> CoxeterType(['A', Integer(3), Integer(1)]).is_affine()
True
```

is_crystallographic()

Return whether `self` is crystallographic.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: [ [t, t.is_crystallographic() ] for t in CartanType.
↪samples(finite=True) ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], True],
 [['C', 1], True], [['C', 5], True],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], True], [['G', 2], True],
 [['I', 5], False], [['H', 3], False], [['H', 4], False]]
```

```
>>> from sage.all import *
>>> [ [t, t.is_crystallographic() ] for t in CartanType.samples(finite=True) ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], True],
 [['C', 1], True], [['C', 5], True],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], True], [['G', 2], True],
 [['I', 5], False], [['H', 3], False], [['H', 4], False]]
```

is_finite()

Return whether `self` is finite.

EXAMPLES:

```
sage: CoxeterType(['A',4]).is_finite()
True
sage: CoxeterType(['A',4, 1]).is_finite()
False
```

```
>>> from sage.all import *
>>> CoxeterType(['A',Integer(4)]).is_finite()
True
>>> CoxeterType(['A',Integer(4), Integer(1)]).is_finite()
False
```

is_simply_laced()

Return whether `self` is simply laced.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: [ [t, t.is_simply_laced() ] for t in CartanType.samples() ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], False],
 [['C', 1], True], [['C', 5], False],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], False], [['G', 2], False],
 [['I', 5], False], [['H', 3], False], [['H', 4], False],
 [['A', 1, 1], False], [['A', 5, 1], True],
 [['B', 1, 1], False], [['B', 5, 1], False],
 [['C', 1, 1], False], [['C', 5, 1], False],
 [['D', 3, 1], True], [['D', 5, 1], True],
 [['E', 6, 1], True], [['E', 7, 1], True], [['E', 8, 1], True],
 [['F', 4, 1], False], [['G', 2, 1], False],
 [['BC', 1, 2], False], [['BC', 5, 2], False],
 [['B', 5, 1]^*, False], [['C', 4, 1]^*, False],
 [['F', 4, 1]^*, False], [['G', 2, 1]^*, False],
 [['BC', 1, 2]^*, False], [['BC', 5, 2]^*, False]]
```

```
>>> from sage.all import *
>>> [ [t, t.is_simply_laced() ] for t in CartanType.samples() ]
```

(continues on next page)

(continued from previous page)

```
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], False],
 [['C', 1], True], [['C', 5], False],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], False], [['G', 2], False],
 [['I', 5], False], [['H', 3], False], [['H', 4], False],
 [['A', 1, 1], False], [['A', 5, 1], True],
 [['B', 1, 1], False], [['B', 5, 1], False],
 [['C', 1, 1], False], [['C', 5, 1], False],
 [['D', 3, 1], True], [['D', 5, 1], True],
 [['E', 6, 1], True], [['E', 7, 1], True], [['E', 8, 1], True],
 [['F', 4, 1], False], [['G', 2, 1], False],
 [['BC', 1, 2], False], [['BC', 5, 2], False],
 [['B', 5, 1]^*, False], [['C', 4, 1]^*, False],
 [['F', 4, 1]^*, False], [['G', 2, 1]^*, False],
 [['BC', 1, 2]^*, False], [['BC', 5, 2]^*, False]]
```

rank()

Return the rank of `self`.

This is the number of nodes of the associated Coxeter graph.

EXAMPLES:

```
sage: CoxeterType(['A', 4]).rank()
4
sage: CoxeterType(['A', 7, 2]).rank()
5
sage: CoxeterType(['I', 8]).rank()
2
```

```
>>> from sage.all import *
>>> CoxeterType(['A', Integer(4)]).rank()
4
>>> CoxeterType(['A', Integer(7), Integer(2)]).rank()
5
>>> CoxeterType(['I', Integer(8)]).rank()
2
```

classmethod samples(*finite=None*, *affine=None*, *crystallographic=None*)

Return a sample of the available Coxeter types.

INPUT:

- `finite` – boolean or `None` (default: `None`)
- `affine` – boolean or `None` (default: `None`)
- `crystallographic` – boolean or `None` (default: `None`)

The sample contains all the exceptional finite and affine Coxeter types, as well as typical representatives of the infinite families.

EXAMPLES:

```
sage: CoxeterType.samples()
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['H', 3], Coxeter type of ['H', 4],
 Coxeter type of ['I', 10], Coxeter type of ['A', 2, 1],
 Coxeter type of ['B', 5, 1], Coxeter type of ['C', 5, 1],
 Coxeter type of ['D', 5, 1], Coxeter type of ['E', 6, 1],
 Coxeter type of ['E', 7, 1], Coxeter type of ['E', 8, 1],
 Coxeter type of ['F', 4, 1], Coxeter type of ['G', 2, 1],
 Coxeter type of ['A', 1, 1]]
```

```
>>> from sage.all import *
>>> CoxeterType.samples()
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['H', 3], Coxeter type of ['H', 4],
 Coxeter type of ['I', 10], Coxeter type of ['A', 2, 1],
 Coxeter type of ['B', 5, 1], Coxeter type of ['C', 5, 1],
 Coxeter type of ['D', 5, 1], Coxeter type of ['E', 6, 1],
 Coxeter type of ['E', 7, 1], Coxeter type of ['E', 8, 1],
 Coxeter type of ['F', 4, 1], Coxeter type of ['G', 2, 1],
 Coxeter type of ['A', 1, 1]]
```

The finite, affine and crystallographic options allow respectively for restricting to (non) finite, (non) affine, and (non) crystallographic Cartan types:

```
sage: CoxeterType.samples(finite=True)
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['H', 3], Coxeter type of ['H', 4],
 Coxeter type of ['I', 10]]

sage: CoxeterType.samples(affine=True)
[Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

sage: CoxeterType.samples(crystallographic=True)
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
```

(continues on next page)

(continued from previous page)

```
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

sage: CoxeterType.samples(crystallographic=False)
[Coxeter type of ['H', 3],
 Coxeter type of ['H', 4],
 Coxeter type of ['I', 10]]
```

```
>>> from sage.all import *
>>> CoxeterType.samples(finite=True)
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['H', 3], Coxeter type of ['H', 4],
 Coxeter type of ['I', 10]]

>>> CoxeterType.samples(affine=True)
[Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

>>> CoxeterType.samples(crystallographic=True)
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

>>> CoxeterType.samples(crystallographic=False)
[Coxeter type of ['H', 3],
 Coxeter type of ['H', 4],
 Coxeter type of ['I', 10]]
```

Todo

add some reducible Coxeter types (suggestions?)

class sage.combinat.root_system.coxeter_type.**CoxeterTypeFromCartanType**(*cartan_type*)

Bases: UniqueRepresentation, *CoxeterType*

A Coxeter type associated to a Cartan type.

cartan_type()

Return the Cartan type used to construct self.

EXAMPLES:

```
sage: C = CoxeterType(['C',3])
sage: C.cartan_type()
['C', 3]
```

```
>>> from sage.all import *
>>> C = CoxeterType(['C',Integer(3)])
>>> C.cartan_type()
['C', 3]
```

component_types()

A list of Coxeter types making up the reducible type.

EXAMPLES:

```
sage: CoxeterType(['A',2],['B',2]).component_types()
[Coxeter type of ['A', 2], Coxeter type of ['B', 2]]

sage: CoxeterType('A4xB3').component_types()
[Coxeter type of ['A', 4], Coxeter type of ['B', 3]]

sage: CoxeterType(['A', 2]).component_types()
Traceback (most recent call last):
...
ValueError: component types only defined for reducible types
```

```
>>> from sage.all import *
>>> CoxeterType(['A',Integer(2)],['B',Integer(2)]).component_types()
[Coxeter type of ['A', 2], Coxeter type of ['B', 2]]

>>> CoxeterType('A4xB3').component_types()
[Coxeter type of ['A', 4], Coxeter type of ['B', 3]]

>>> CoxeterType(['A', Integer(2)]).component_types()
Traceback (most recent call last):
...
ValueError: component types only defined for reducible types
```

coxeter_graph()

Return the Coxeter graph of self.

EXAMPLES:

```
sage: C = CoxeterType(['H',3])
sage: C.coxeter_graph().edges(sort=True)                                    #␣
↪needs sage.graphs
[(1, 2, 3), (2, 3, 5)]
```

```
>>> from sage.all import *
>>> C = CoxeterType(['H',Integer(3)])
>>> C.coxeter_graph().edges(sort=True)                                    #␣
↪needs sage.graphs
[(1, 2, 3), (2, 3, 5)]
```

coxeter_matrix()

Return the Coxeter matrix associated to `self`.

EXAMPLES:

```
sage: C = CoxeterType(['H',3])
sage: C.coxeter_matrix()                                                    #␣
↪needs sage.graphs
[1 3 2]
[3 1 5]
[2 5 1]
```

```
>>> from sage.all import *
>>> C = CoxeterType(['H',Integer(3)])
>>> C.coxeter_matrix()                                                    #␣
↪needs sage.graphs
[1 3 2]
[3 1 5]
[2 5 1]
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: C = CoxeterType(['A', 4])
sage: C.index_set()
(1, 2, 3, 4)
```

```
>>> from sage.all import *
>>> C = CoxeterType(['A', Integer(4)])
>>> C.index_set()
(1, 2, 3, 4)
```

is_affine()

Return if `self` is an affine type.

EXAMPLES:

```
sage: C = CoxeterType(['F', 4, 1])
sage: C.is_affine()
True
```

```
>>> from sage.all import *
>>> C = CoxeterType(['F', Integer(4), Integer(1)])
>>> C.is_affine()
True
```

is_crystallographic()

Return if `self` is crystallographic.

EXAMPLES:

```
sage: C = CoxeterType(['C', 3])
sage: C.is_crystallographic()
True

sage: C = CoxeterType(['H', 3])
sage: C.is_crystallographic()
False
```

```
>>> from sage.all import *
>>> C = CoxeterType(['C', Integer(3)])
>>> C.is_crystallographic()
True

>>> C = CoxeterType(['H', Integer(3)])
>>> C.is_crystallographic()
False
```

is_finite()

Return if `self` is a finite type.

EXAMPLES:

```
sage: C = CoxeterType(['E', 6])
sage: C.is_finite()
True
```

```
>>> from sage.all import *
>>> C = CoxeterType(['E', Integer(6)])
>>> C.is_finite()
True
```

is_irreducible()

Return if `self` is irreducible.

EXAMPLES:

```
sage: C = CoxeterType(['A', 5])
sage: C.is_irreducible()
True

sage: C = CoxeterType('B3xB3')
sage: C.is_irreducible()
False
```

```
>>> from sage.all import *
>>> C = CoxeterType(['A', Integer(5)])
>>> C.is_irreducible()
True

>>> C = CoxeterType('B3xB3')
>>> C.is_irreducible()
False
```

is_reducible()

Return if `self` is reducible.

EXAMPLES:

```
sage: C = CoxeterType(['A', 5])
sage: C.is_reducible()
False

sage: C = CoxeterType('A2xA2')
sage: C.is_reducible()
True
```

```
>>> from sage.all import *
>>> C = CoxeterType(['A', Integer(5)])
>>> C.is_reducible()
False

>>> C = CoxeterType('A2xA2')
>>> C.is_reducible()
True
```

is_simply_laced()

Return if `self` is simply-laced.

EXAMPLES:

```
sage: C = CoxeterType(['A', 5])
sage: C.is_simply_laced()
True

sage: C = CoxeterType(['B', 3])
sage: C.is_simply_laced()
False
```

```
>>> from sage.all import *
>>> C = CoxeterType(['A', Integer(5)])
>>> C.is_simply_laced()
True

>>> C = CoxeterType(['B', Integer(3)])
>>> C.is_simply_laced()
False
```

rank()

Return the rank of self.

EXAMPLES:

```
sage: C = CoxeterType(['I', 16])
sage: C.rank()
2
```

```
>>> from sage.all import *
>>> C = CoxeterType(['I', Integer(16)])
>>> C.rank()
2
```

relabel(*relabelling*)

Return a relabelled copy of self.

EXAMPLES:

```
sage: ct = CoxeterType(['A',2])
sage: ct.relabel({1:-1, 2:-2})
Coxeter type of ['A', 2] relabelled by {1: -1, 2: -2}
```

```
>>> from sage.all import *
>>> ct = CoxeterType(['A',Integer(2)])
>>> ct.relabel({Integer(1):-Integer(1), Integer(2):-Integer(2)})
Coxeter type of ['A', 2] relabelled by {1: -1, 2: -2}
```

type()

Return the type of self.

EXAMPLES:

```
sage: C = CoxeterType(['A', 4])
sage: C.type()
'A'
```

```
>>> from sage.all import *
>>> C = CoxeterType(['A', Integer(4)])
>>> C.type()
'A'
```

5.1.230 Dynkin diagrams

AUTHORS:

- Travis Scrimshaw (2012-04-22): Nicolas M. Thiery moved Cartan matrix creation to here and I cached results for speed.
- Travis Scrimshaw (2013-06-11): Changed inputs of Dynkin diagrams to handle other Dynkin diagrams and graphs. Implemented remaining Cartan type methods.
- Christian Stump, Travis Scrimshaw (2013-04-11): Added Cartan matrix as possible input for Dynkin diagrams.

sage.combinat.root_system.dynkin_diagram.**DynkinDiagram**(**args*, ***kwds*)

Return the Dynkin diagram corresponding to the input.

INPUT:

The input can be one of the following:

- empty to obtain an empty Dynkin diagram
- a Cartan type
- a Cartan matrix
- a Cartan matrix and an indexing set

One can also input an indexing set by passing a tuple using the optional argument `index_set`.

The edge multiplicities are encoded as edge labels. For the corresponding Cartan matrices, this uses the convention in Hong and Kang, Kac, Fulton and Harris, and crystals. This is the **opposite** convention in Bourbaki and Wikipedia's Dynkin diagram (Wikipedia article Dynkin_diagram). That is for $i \neq j$:

```
i <--k-- j <==> a_ij = -k
           <==> -scalar(coroot[i], root[j]) = k
           <==> multiple arrows point from the longer root
                to the shorter one
```

For example, in type C_2, we have:

```
sage: C2 = DynkinDiagram(['C',2]); C2
O=<=O
1   2
C2
sage: C2.cartan_matrix()
[ 2 -2]
[-1  2]
```

```
>>> from sage.all import *
>>> C2 = DynkinDiagram(['C',Integer(2)]); C2
O=<=O
1   2
C2
>>> C2.cartan_matrix()
[ 2 -2]
[-1  2]
```

However Bourbaki would have the Cartan matrix as:

$$\begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}.$$

EXAMPLES:

```
sage: DynkinDiagram(['A', 4])
O---O---O---O
1   2   3   4
A4

sage: DynkinDiagram(['A',1],['A',1])
```

(continues on next page)

(continued from previous page)

```
O
1
O
2
A1xA1

sage: R = RootSystem("A2xB2xF4")
sage: DynkinDiagram(R)
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4

sage: R = RootSystem("A2xB2xF4")
sage: CM = R.cartan_matrix(); CM
[ 2 -1| 0  0| 0  0  0  0]
[-1  2| 0  0| 0  0  0  0]
[-----+-----+-----------]
[ 0  0| 2 -1| 0  0  0  0]
[ 0  0|-2  2| 0  0  0  0]
[-----+-----+-----------]
[ 0  0| 0  0| 2 -1  0  0]
[ 0  0| 0  0|-1  2 -1  0]
[ 0  0| 0  0| 0 -2  2 -1]
[ 0  0| 0  0| 0  0 -1  2]
sage: DD = DynkinDiagram(CM); DD
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4
sage: DD.cartan_matrix()
[ 2 -1  0  0  0  0  0  0]
[-1  2  0  0  0  0  0  0]
[ 0  0  2 -1  0  0  0  0]
[ 0  0 -2  2  0  0  0  0]
[ 0  0  0  0  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -2  2 -1]
[ 0  0  0  0  0  0 -1  2]
```

```
>>> from sage.all import *
>>> DynkinDiagram(['A', Integer(4)])
O---O---O---O
1   2   3   4
A4
```

(continues on next page)

(continued from previous page)

```
>>> DynkinDiagram(['A',Integer(1)],['A',Integer(1)])
O
1
O
2
A1xA1

>>> R = RootSystem("A2xB2xF4")
>>> DynkinDiagram(R)
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4

>>> R = RootSystem("A2xB2xF4")
>>> CM = R.cartan_matrix(); CM
[ 2 -1| 0  0| 0  0  0  0]
[-1  2| 0  0| 0  0  0  0]
[-----+-----+-----------]
[ 0  0| 2 -1| 0  0  0  0]
[ 0  0|-2  2| 0  0  0  0]
[-----+-----+-----------]
[ 0  0| 0  0| 2 -1  0  0]
[ 0  0| 0  0|-1  2 -1  0]
[ 0  0| 0  0| 0 -2  2 -1]
[ 0  0| 0  0| 0  0 -1  2]
>>> DD = DynkinDiagram(CM); DD
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4
>>> DD.cartan_matrix()
[ 2 -1  0  0  0  0  0  0]
[-1  2  0  0  0  0  0  0]
[ 0  0  2 -1  0  0  0  0]
[ 0  0 -2  2  0  0  0  0]
[ 0  0  0  0  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -2  2 -1]
[ 0  0  0  0  0  0 -1  2]
```

We can also create Dynkin diagrams from arbitrary Cartan matrices:

```
sage: C = CartanMatrix([[2, -3], [-4, 2]])
sage: DynkinDiagram(C)
Dynkin diagram of rank 2
```

(continues on next page)

(continued from previous page)

```
sage: C.index_set()
(0, 1)
sage: CI = CartanMatrix([[2, -3], [-4, 2]], [3, 5])
sage: DI = DynkinDiagram(CI)
sage: DI.index_set()
(3, 5)
sage: CII = CartanMatrix([[2, -3], [-4, 2]])
sage: DII = DynkinDiagram(CII, ('y', 'x'))
sage: DII.index_set()
('x', 'y')
```

```
>>> from sage.all import *
>>> C = CartanMatrix([[Integer(2), -Integer(3)], [-Integer(4), Integer(2)]])
>>> DynkinDiagram(C)
Dynkin diagram of rank 2
>>> C.index_set()
(0, 1)
>>> CI = CartanMatrix([[Integer(2), -Integer(3)], [-Integer(4), Integer(2)]],
↪[Integer(3), Integer(5)])
>>> DI = DynkinDiagram(CI)
>>> DI.index_set()
(3, 5)
>>> CII = CartanMatrix([[Integer(2), -Integer(3)], [-Integer(4), Integer(2)]])
>>> DII = DynkinDiagram(CII, ('y', 'x'))
>>> DII.index_set()
('x', 'y')
```

See also:

`CartanType()` for a general discussion on Cartan types and in particular node labeling conventions.

class `sage.combinat.root_system.dynkin_diagram.`**`DynkinDiagram_class`**(*t=None*, *index_set=None*, *odd_isotropic_roots=[]*, ***options*)

Bases: `DiGraph`, `CartanType_abstract`

A Dynkin diagram.

See also:

`DynkinDiagram()`

INPUT:

- `t` – a Cartan type, Cartan matrix, or `None`

EXAMPLES:

```
sage: DynkinDiagram(['A', 3])
O---O---O
1   2   3
```

(continues on next page)

(continued from previous page)

```
A3
sage: C = CartanMatrix([[2, -3], [-4, 2]])
sage: DynkinDiagram(C)
Dynkin diagram of rank 2
sage: C.dynkin_diagram().cartan_matrix() == C
True
```

```
>>> from sage.all import *
>>> DynkinDiagram(['A', Integer(3)])
O---O---O
1   2   3
A3
>>> C = CartanMatrix([[Integer(2), -Integer(3)], [-Integer(4), Integer(2)]])
>>> DynkinDiagram(C)
Dynkin diagram of rank 2
>>> C.dynkin_diagram().cartan_matrix() == C
True
```

add_edge(*i*, *j*, *label=1*)

EXAMPLES:

```
sage: from sage.combinat.root_system.dynkin_diagram import DynkinDiagram_class
sage: d = DynkinDiagram_class(CartanType(['A',3]))
sage: sorted(d.edges(sort=True))
[]
sage: d.add_edge(2, 3)
sage: sorted(d.edges(sort=True))
[(2, 3, 1), (3, 2, 1)]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.dynkin_diagram import DynkinDiagram_class
>>> d = DynkinDiagram_class(CartanType(['A',Integer(3)]))
>>> sorted(d.edges(sort=True))
[]
>>> d.add_edge(Integer(2), Integer(3))
>>> sorted(d.edges(sort=True))
[(2, 3, 1), (3, 2, 1)]
```

static an_instance()

Return an example of Dynkin diagram.

EXAMPLES:

```
sage: from sage.combinat.root_system.dynkin_diagram import DynkinDiagram_class
sage: g = DynkinDiagram_class.an_instance()
sage: g
Dynkin diagram of rank 3
sage: g.cartan_matrix()
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.dynkin_diagram import DynkinDiagram_class
>>> g = DynkinDiagram_class.an_instance()
>>> g
Dynkin diagram of rank 3
>>> g.cartan_matrix()
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

cartan_matrix()

Return the Cartan matrix for this Dynkin diagram.

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).cartan_matrix()
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

```
>>> from sage.all import *
>>> DynkinDiagram(['C',Integer(3)]).cartan_matrix()
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

cartan_type()

EXAMPLES:

```
sage: DynkinDiagram("A2","B2","F4").cartan_type()
A2xB2xF4
```

```
>>> from sage.all import *
>>> DynkinDiagram("A2","B2","F4").cartan_type()
A2xB2xF4
```

column(j)

Return the j-th column $(a_{i,j})_i$ of the Cartan matrix corresponding to this Dynkin diagram, as a container (or iterator) of tuples $(i, a_{i,j})$.

EXAMPLES:

```
sage: g = DynkinDiagram(["B",4])
sage: [ (i,a) for (i,a) in g.column(3) ]
[(3, 2), (2, -1), (4, -2)]
```

```
>>> from sage.all import *
>>> g = DynkinDiagram(["B",Integer(4)])
>>> [ (i,a) for (i,a) in g.column(Integer(3)) ]
[(3, 2), (2, -1), (4, -2)]
```

coxeter_diagram()

Construct the Coxeter diagram of `self`.

See also

CartanType_abstract.coxeter_diagram()

EXAMPLES:

```
sage: cm = CartanMatrix([[2,-5,0],[-2,2,-1],[0,-1,2]])
sage: D = cm.dynkin_diagram()
sage: G = D.coxeter_diagram(); G
Graph on 3 vertices
sage: G.edges(sort=True)
[(0, 1, +Infinity), (1, 2, 3)]

sage: ct = CartanType([['A',2,2], ['B',3]])
sage: ct.coxeter_diagram()
Graph on 5 vertices
sage: ct.dynkin_diagram().coxeter_diagram() == ct.coxeter_diagram()
True
```

```
>>> from sage.all import *
>>> cm = CartanMatrix([[Integer(2),-Integer(5),Integer(0)],[-Integer(2),
→Integer(2),-Integer(1)],[Integer(0),-Integer(1),Integer(2)]])
>>> D = cm.dynkin_diagram()
>>> G = D.coxeter_diagram(); G
Graph on 3 vertices
>>> G.edges(sort=True)
[(0, 1, +Infinity), (1, 2, 3)]

>>> ct = CartanType([['A',Integer(2),Integer(2)], ['B',Integer(3)]])
>>> ct.coxeter_diagram()
Graph on 5 vertices
>>> ct.dynkin_diagram().coxeter_diagram() == ct.coxeter_diagram()
True
```

dual()

Return the dual Dynkin diagram, obtained by reversing all edges.

EXAMPLES:

```
sage: D = DynkinDiagram(['C',3])
sage: D.edges(sort=True)
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]
sage: D.dual()
O---O=>=O
1   2   3
B3
sage: D.dual().edges(sort=True)
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]
sage: D.dual() == DynkinDiagram(['B',3])
True
```

```
>>> from sage.all import *
>>> D = DynkinDiagram(['C',Integer(3)])
```

(continues on next page)

(continued from previous page)

```
>>> D.edges(sort=True)
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]
>>> D.dual()
O---O=>=O
1   2   3
B3
>>> D.dual().edges(sort=True)
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]
>>> D.dual() == DynkinDiagram(['B',Integer(3)])
True
```

dynkin_diagram()

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).dynkin_diagram()
O---O=<=O
1   2   3
C3
```

```
>>> from sage.all import *
>>> DynkinDiagram(['C',Integer(3)]).dynkin_diagram()
O---O=<=O
1   2   3
C3
```

index_set()

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).index_set()
(1, 2, 3)
sage: DynkinDiagram("A2","B2","F4").index_set()
(1, 2, 3, 4, 5, 6, 7, 8)
```

```
>>> from sage.all import *
>>> DynkinDiagram(['C',Integer(3)]).index_set()
(1, 2, 3)
>>> DynkinDiagram("A2","B2","F4").index_set()
(1, 2, 3, 4, 5, 6, 7, 8)
```

is_affine()

Check if `self` corresponds to an affine root system.

EXAMPLES:

```
sage: CartanType(['F',4]).dynkin_diagram().is_affine()
False
sage: D = DynkinDiagram(CartanMatrix([[2, -4], [-3, 2]]))
sage: D.is_affine()
False
```

```
>>> from sage.all import *
>>> CartanType(['F',Integer(4)]).dynkin_diagram().is_affine()
```

(continues on next page)

(continued from previous page)

```
False
>>> D = DynkinDiagram(CartanMatrix([[Integer(2), -Integer(4)], [-Integer(3),␣
↪Integer(2)]]))
>>> D.is_affine()
False
```

is_crystallographic()

Implement *CartanType_abstract.is_crystallographic()*.

A Dynkin diagram always corresponds to a crystallographic root system.

EXAMPLES:

```
sage: CartanType(['F',4]).dynkin_diagram().is_crystallographic()
True
```

```
>>> from sage.all import *
>>> CartanType(['F',Integer(4)]).dynkin_diagram().is_crystallographic()
True
```

is_finite()

Check if `self` corresponds to a finite root system.

EXAMPLES:

```
sage: CartanType(['F',4]).dynkin_diagram().is_finite()
True
sage: D = DynkinDiagram(CartanMatrix([[2, -4], [-3, 2]]))
sage: D.is_finite()
False
```

```
>>> from sage.all import *
>>> CartanType(['F',Integer(4)]).dynkin_diagram().is_finite()
True
>>> D = DynkinDiagram(CartanMatrix([[Integer(2), -Integer(4)], [-Integer(3),␣
↪Integer(2)]]))
>>> D.is_finite()
False
```

is_irreducible()

Check if `self` corresponds to an irreducible root system.

EXAMPLES:

```
sage: CartanType(['F',4]).dynkin_diagram().is_irreducible()
True
sage: CM = CartanMatrix([[2,-6],[-4,2]])
sage: CM.dynkin_diagram().is_irreducible()
True
sage: CartanType("A2xB3").dynkin_diagram().is_irreducible()
False
sage: CM = CartanMatrix([[2,-6,0],[-4,2,0],[0,0,2]])
sage: CM.dynkin_diagram().is_irreducible()
False
```

```
>>> from sage.all import *
>>> CartanType(['F',Integer(4)]).dynkin_diagram().is_irreducible()
True
>>> CM = CartanMatrix([[Integer(2),-Integer(6)],[-Integer(4),Integer(2)]])
>>> CM.dynkin_diagram().is_irreducible()
True
>>> CartanType("A2xB3").dynkin_diagram().is_irreducible()
False
>>> CM = CartanMatrix([[Integer(2),-Integer(6),Integer(0)],[-Integer(4),
↪Integer(2),Integer(0)],[Integer(0),Integer(0),Integer(2)]])
>>> CM.dynkin_diagram().is_irreducible()
False
```

odd_isotropic_roots()

Return the odd isotropic roots of `self`.

EXAMPLES:

```
sage: g = DynkinDiagram(['A',4])
sage: g.odd_isotropic_roots()
()
sage: g = DynkinDiagram(['A',[4,3]])
sage: g.odd_isotropic_roots()
(0,)
```

```
>>> from sage.all import *
>>> g = DynkinDiagram(['A',Integer(4)])
>>> g.odd_isotropic_roots()
()
>>> g = DynkinDiagram(['A',[Integer(4),Integer(3)]])
>>> g.odd_isotropic_roots()
(0,)
```

rank()

Return the index set for this Dynkin diagram.

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).rank()
3
sage: DynkinDiagram("A2","B2","F4").rank()
8
```

```
>>> from sage.all import *
>>> DynkinDiagram(['C',Integer(3)]).rank()
3
>>> DynkinDiagram("A2","B2","F4").rank()
8
```

relabel(**args*, ***kwds*)

Return the relabelled Dynkin diagram of `self`.

INPUT: see `relabel()`

There is one difference: the default value for `inplace` is `False` instead of `True`.

EXAMPLES:

```
sage: D = DynkinDiagram(['C',3])
sage: D.relabel({1:0, 2:4, 3:1})
O---O=<=O
0   4   1
C3 relabelled by {1: 0, 2: 4, 3: 1}
sage: D
O---O=<=O
1   2   3
C3

sage: _ = D.relabel({1:0, 2:4, 3:1}, inplace=True)
sage: D
O---O=<=O
0   4   1
C3 relabelled by {1: 0, 2: 4, 3: 1}

sage: D = DynkinDiagram(['A', [1,2]])
sage: Dp = D.relabel({-1:4, 0:-3, 1:3, 2:2})
sage: Dp
O---X---O---O
4   -3  3   2
A1|2 relabelled by {-1: 4, 0: -3, 1: 3, 2: 2}
sage: Dp.odd_isotropic_roots()
(-3,)

sage: D = DynkinDiagram(['D', 5])
sage: G, perm = D.relabel(range(5), return_map=True)
sage: G
        O 4
        |
        |
O---O---O---O
0   1   2   3
D5 relabelled by {1: 0, 2: 1, 3: 2, 4: 3, 5: 4}
sage: perm
{1: 0, 2: 1, 3: 2, 4: 3, 5: 4}

sage: perm = D.relabel(range(5), return_map=True, inplace=True)
sage: D
        O 4
        |
        |
O---O---O---O
0   1   2   3
D5 relabelled by {1: 0, 2: 1, 3: 2, 4: 3, 5: 4}
sage: perm
{1: 0, 2: 1, 3: 2, 4: 3, 5: 4}
```

```
>>> from sage.all import *
>>> D = DynkinDiagram(['C',Integer(3)])
>>> D.relabel({Integer(1):Integer(0), Integer(2):Integer(4),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(3):Integer(1)})
O---O=<=O
0   4   1
C3 relabelled by {1: 0, 2: 4, 3: 1}
>>> D
O---O=<=O
1   2   3
C3

>>> _ = D.relabel({Integer(1):Integer(0), Integer(2):Integer(4),␣
↪Integer(3):Integer(1)}, inplace=True)
>>> D
O---O=<=O
0   4   1
C3 relabelled by {1: 0, 2: 4, 3: 1}

>>> D = DynkinDiagram(['A', [Integer(1),Integer(2)]])
>>> Dp = D.relabel({-Integer(1):Integer(4), Integer(0):-Integer(3),␣
↪Integer(1):Integer(3), Integer(2):Integer(2)})
>>> Dp
O---X---O---O
4   -3  3   2
A1|2 relabelled by {-1: 4, 0: -3, 1: 3, 2: 2}
>>> Dp.odd_isotropic_roots()
(-3,)

>>> D = DynkinDiagram(['D', Integer(5)])
>>> G, perm = D.relabel(range(Integer(5)), return_map=True)
>>> G
        O 4
        |
        |
O---O---O---O
0   1   2   3
D5 relabelled by {1: 0, 2: 1, 3: 2, 4: 3, 5: 4}
>>> perm
{1: 0, 2: 1, 3: 2, 4: 3, 5: 4}

>>> perm = D.relabel(range(Integer(5)), return_map=True, inplace=True)
>>> D
        O 4
        |
        |
O---O---O---O
0   1   2   3
D5 relabelled by {1: 0, 2: 1, 3: 2, 4: 3, 5: 4}
>>> perm
{1: 0, 2: 1, 3: 2, 4: 3, 5: 4}
```

row(i)

Return the i-th row $(a_{i,j})_j$ of the Cartan matrix corresponding to this Dynkin diagram, as a container (or iterator) of tuples $(j, a_{i,j})$.

EXAMPLES:

```
sage: g = DynkinDiagram(["C",4])
sage: [ (i,a) for (i,a) in g.row(3) ]
[(3, 2), (2, -1), (4, -2)]
```

```
>>> from sage.all import *
>>> g = DynkinDiagram(["C",Integer(4)])
>>> [ (i,a) for (i,a) in g.row(Integer(3)) ]
[(3, 2), (2, -1), (4, -2)]
```

subtype(*index_set*)

Return a subtype of `self` given by `index_set`.

A subtype can be considered the Dynkin diagram induced from the Dynkin diagram of `self` by `index_set`.

EXAMPLES:

```
sage: D = DynkinDiagram(['A',6,2]); D
O=<=O---O=<=O
0   1   2   3
BC3~
sage: D.subtype([1,2,3])
Dynkin diagram of rank 3
```

```
>>> from sage.all import *
>>> D = DynkinDiagram(['A',Integer(6),Integer(2)]); D
O=<=O---O=<=O
0   1   2   3
BC3~
>>> D.subtype([Integer(1),Integer(2),Integer(3)])
Dynkin diagram of rank 3
```

symmetrizer()

Return the symmetrizer of the corresponding Cartan matrix.

EXAMPLES:

```
sage: d = DynkinDiagram()
sage: d.add_edge(1,2,3)
sage: d.add_edge(2,3)
sage: d.add_edge(3,4,3)
sage: d.symmetrizer()
Finite family {1: 9, 2: 3, 3: 3, 4: 1}
```

```
>>> from sage.all import *
>>> d = DynkinDiagram()
>>> d.add_edge(Integer(1),Integer(2),Integer(3))
>>> d.add_edge(Integer(2),Integer(3))
>>> d.add_edge(Integer(3),Integer(4),Integer(3))
>>> d.symmetrizer()
Finite family {1: 9, 2: 3, 3: 3, 4: 1}
```

`sage.combinat.root_system.dynkin_diagram.`**precheck**(*t*, *letter=None*, *length=None*, *affine=None*, *n_ge=None*, *n=None*)

EXAMPLES:

```
sage: from sage.combinat.root_system.dynkin_diagram import precheck
sage: ct = CartanType(['A',4])
sage: precheck(ct, letter='C')
Traceback (most recent call last):
...
ValueError: t[0] must be = 'C'
sage: precheck(ct, affine=1)
Traceback (most recent call last):
...
ValueError: t[2] must be = 1
sage: precheck(ct, length=3)
Traceback (most recent call last):
...
ValueError: len(t) must be = 3
sage: precheck(ct, n=3)
Traceback (most recent call last):
...
ValueError: t[1] must be = 3
sage: precheck(ct, n_ge=5)
Traceback (most recent call last):
...
ValueError: t[1] must be >= 5
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.dynkin_diagram import precheck
>>> ct = CartanType(['A',Integer(4)])
>>> precheck(ct, letter='C')
Traceback (most recent call last):
...
ValueError: t[0] must be = 'C'
>>> precheck(ct, affine=Integer(1))
Traceback (most recent call last):
...
ValueError: t[2] must be = 1
>>> precheck(ct, length=Integer(3))
Traceback (most recent call last):
...
ValueError: len(t) must be = 3
>>> precheck(ct, n=Integer(3))
Traceback (most recent call last):
...
ValueError: t[1] must be = 3
>>> precheck(ct, n_ge=Integer(5))
Traceback (most recent call last):
...
ValueError: t[1] must be >= 5
```

5.1.231 Hecke algebra representations

class sage.combinat.root_system.hecke_algebra_representation.**CherednikOperatorsEigenvectors**(*T*, *T_Y=None*, *normalized=True*)

Bases: UniqueRepresentation, SageObject

A class for the family of eigenvectors of the Y Cherednik operators for a module over a (Double) Affine Hecke algebra

INPUT:

- T – a family $(T_i)_{i\in I}$ implementing the action of the generators of an affine Hecke algebra on self. The intertwiner operators are built from these.
- T_Y – a family $(T_i^Y)_{i\in I}$ implementing the action of the generators of an affine Hecke algebra on self. By default, this is T. But this can be used to get the action of the full Double Affine Hecke Algebra. The Y operators are built from the T_Y.

This returns a function $\mu \mapsto E_\mu$ which uses intertwining operators to calculate recursively eigenvectors E_μ for the action of the torus of the affine Hecke algebra with eigenvalue given by f. Namely:

$$E_\mu.Y^{\lambda^\vee} = f(\lambda^\vee, \mu)E_\mu$$

Assumptions:

- seed(mu) initializes the recurrence by returning an appropriate eigenvector E_μ for μ trivial enough. For example, for nonsymmetric Macdonald polynomials seed(mu) returns the monomial X^μ for a minuscule weight μ.
- f is almost equivariant. Namely, $f(\lambda^\vee, \mu) = f(\lambda^\vee s_i, twist(\mu, i))$ whenever i is a descent of μ.
- $twist(\mu, i)$ maps μ closer to the dominant chamber whenever i is a descent of μ.

Todo

Add tests for the above assumptions, and also that the classical operators $T_1, \ldots, T_n$ from T and T_Y coincide.

Y()

Return the Cherednik operators.

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.Y()
Lazy family (...)_{i in Coroot lattice of the Root system of type ['B', 2, 1]}
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(2)])
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
```

(continues on next page)

(continued from previous page)

```
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> E.Y()
Lazy family (...)_{i in Coroot lattice of the Root system of type ['B', 2, 1]}
```

affine_lift(*mu*)

Lift the index `\mu` to a space admitting an action of the affine Weyl group.

INPUT:

- `mu` – an element μ of the indexing set

In this space, one should have `first_descent` and `apply_simple_reflection` act properly.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w = W.an_element(); w
123
sage: E.affine_lift(w)
121
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> w = W.an_element(); w
123
>>> E.affine_lift(w)
121
```

affine_retract(*mu*)

Retract μ from a space admitting an action of the affine Weyl group.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w = W.an_element(); w
123
```

(continues on next page)

(continued from previous page)

```
sage: E.affine_retract(E.affine_lift(w)) == w
True
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> w = W.an_element(); w
123
>>> E.affine_retract(E.affine_lift(w)) == w
True
```

cartan_type()

Return Cartan type of `self`.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.cartan_type()
['B', 3, 1]

sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).cartan_type()          #␣
↪needs sage.graphs
['B', 2, 1]
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(3)])
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> E.cartan_type()
['B', 3, 1]

>>> NonSymmetricMacdonaldPolynomials(["B", Integer(2), Integer(1)]).cartan_
↪type()           # needs sage.graphs
['B', 2, 1]
```

domain()

The module on which the affine Hecke algebra acts.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: K = QQ['q1,q2']
```

(continues on next page)

(continued from previous page)

```
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.domain()
Algebra of Weyl Group of type ['B', 3]
 (as a matrix group acting on the ambient space)
 over Multivariate Polynomial Ring in q1, q2 over Rational Field
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(3)])
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> E.domain()
Algebra of Weyl Group of type ['B', 3]
 (as a matrix group acting on the ambient space)
 over Multivariate Polynomial Ring in q1, q2 over Rational Field
```

eigenvalue(*mu*, *l*)

Return the eigenvalue of $Y_{\lambda^\vee}$ on E_μ computed by applying $Y_{\lambda^\vee}$ on E_μ.

INPUT:

- `mu` – the index μ of an eigenvector, or a tentative eigenvector
- `l` – the index $\lambda^\vee$ of a Cherednik operator in `self.Y_index_set()`

This default implementation applies explicitly Y_μ to E_λ.

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w0 = W.long_element()
sage: Y = E.Y()
sage: alphacheck = Y.keys().simple_roots()
sage: E.eigenvalue(w0, alphacheck[1])
q1/(-q2)
sage: E.eigenvalue(w0, alphacheck[2])
q1/(-q2)
sage: E.eigenvalue(w0, alphacheck[0])
q2^2/q1^2
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(2)])
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> w0 = W.long_element()
```

(continues on next page)

(continued from previous page)

```
>>> Y = E.Y()
>>> alphacheck = Y.keys().simple_roots()
>>> E.eigenvalue(w0, alphacheck[Integer(1)])
q1/(-q2)
>>> E.eigenvalue(w0, alphacheck[Integer(2)])
q1/(-q2)
>>> E.eigenvalue(w0, alphacheck[Integer(0)])
q2^2/q1^2
```

The following checks that all E_w are eigenvectors, with eigenvalue given by Lemma 7.5 of [HST2008] (checked for A_2, A_3):

```
sage: Pcheck = Y.keys()
sage: Wcheck = Pcheck.weyl_group()
sage: P0check = Pcheck.classical()
sage: def height(root):
....:     return sum(P0check(root).coefficients())
sage: def eigenvalue(w, mu):
....:     return (-q2/q1)^height(Wcheck.from_reduced_word(w.reduced_word()).
↪action(mu))
sage: all(E.eigenvalue(w, a) == eigenvalue(w, a) for w in E.keys() for a in Y.
↪keys().simple_roots()) # long time (2.5s)
True
```

```
>>> from sage.all import *
>>> Pcheck = Y.keys()
>>> Wcheck = Pcheck.weyl_group()
>>> P0check = Pcheck.classical()
>>> def height(root):
...     return sum(P0check(root).coefficients())
>>> def eigenvalue(w, mu):
...     return (-q2/q1)**height(Wcheck.from_reduced_word(w.reduced_word()).
↪action(mu))
>>> all(E.eigenvalue(w, a) == eigenvalue(w, a) for w in E.keys() for a in Y.
↪keys().simple_roots()) # long time (2.5s)
True
```

eigenvalues(*mu*)

Return the eigenvalues of $Y_{\alpha_0}, \dots, Y_{\alpha_n}$ on E_μ.

INPUT:

- `mu` – the index μ of an eigenvector or a tentative eigenvector

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w0 = W.long_element()
sage: E.eigenvalues(w0)
```

(continues on next page)

(continued from previous page)

```
[q2^2/q1^2, q1/(-q2), q1/(-q2)]
sage: w = W.an_element()
sage: E.eigenvalues(w)
[(-q2)/q1, (-q2^2)/(-q1^2), q1^3/(-q2^3)]
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(2)])
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> w0 = W.long_element()
>>> E.eigenvalues(w0)
[q2^2/q1^2, q1/(-q2), q1/(-q2)]
>>> w = W.an_element()
>>> E.eigenvalues(w)
[(-q2)/q1, (-q2^2)/(-q1^2), q1^3/(-q2^3)]
```

hecke_parameters(*i*)

Return the Hecke parameters for index `i`.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: T = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: E = T.Y_eigenvectors()
sage: E.hecke_parameters(1)
(q1, q2)
sage: E.hecke_parameters(2)
(q1, q2)
sage: E.hecke_parameters(0)
(q1, q2)
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(3)])
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> T = KW.demazure_lusztig_operators(q1, q2, affine=True)
>>> E = T.Y_eigenvectors()
>>> E.hecke_parameters(Integer(1))
(q1, q2)
>>> E.hecke_parameters(Integer(2))
(q1, q2)
>>> E.hecke_parameters(Integer(0))
(q1, q2)
```

keys()

The index set for the eigenvectors.

By default, this assumes that the eigenvectors span the full affine Hecke algebra module and that the eigenvectors have the same indexing as the basis of this module.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.keys()
Weyl Group of type ['A', 3] (as a matrix group acting on the ambient space)
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> E.keys()
Weyl Group of type ['A', 3] (as a matrix group acting on the ambient space)
```

recursion(*mu*)

Return the indices used in the recursion.

INPUT:

- mu – the index μ of an eigenvector

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w0 = W.long_element()
sage: E.recursion(w0)
[]
sage: w = W.an_element(); w
123
sage: E.recursion(w)
[1, 2, 1]
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> w0 = W.long_element()
>>> E.recursion(w0)
```

(continues on next page)

(continued from previous page)

```
[]
>>> w = W.an_element(); w
123
>>> E.recursion(w)
[1, 2, 1]
```

seed(*mu*)

Return the eigenvector for μ minuscule.

INPUT:

- `mu` – an element μ of the indexing set

OUTPUT: an element of `T.domain()`

This default implementation returns the monomial indexed by μ.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.seed(W.long_element())
123121
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1, q2)
>>> E.seed(W.long_element())
123121
```

twist(*mu*, *i*)

Act by s_i on μ.

By default, this calls the method `apply_simple_reflection`.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: T = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: E = T.Y_eigenvectors()
sage: w = W.an_element(); w
```

(continues on next page)

(continued from previous page)

```
123
sage: E.twist(w,1)
1231
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(3)])
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> T = KW.demazure_lusztig_operators(q1, q2, affine=True)
>>> E = T.Y_eigenvectors()
>>> w = W.an_element(); w
123
>>> E.twist(w,Integer(1))
1231
```

class sage.combinat.root_system.hecke_algebra_representation.**HeckeAlgebraRepresentation**(*domain*, *on_basis*, *cartan_type*, *q1*, *q2*, *q=1*, *side='right'*)

Bases: WithEqualityById, SageObject

A representation of an (affine) Hecke algebra given by the action of the T generators.

Let F_i be a family of operators implementing an action of the operators $(T_i)_{i\in I}$ of the Hecke algebra on some vector space domain, given by their action on the basis of domain. This constructs the family of operators $(F_w)_{w\in W}$ describing the action of all elements of the basis $(T_w)_{w\in W}$ of the Hecke algebra. This is achieved by linearity on the first argument, and applying recursively the F_i along a reduced word for $w = s_{i_1}\cdots s_{i_k}$:

$$F_w(x) = F_{i_k}\circ\cdots\circ F_{i_1}(x).$$

INPUT:

- domain – a vector space
- f – a function f(l,i) taking a basis element l of domain and an index i, and returning F_i
- cartan_type – the Cartan type of the Hecke algebra
- q1, q2 – the eigenvalues of the generators T of the Hecke algebra
- side – 'left' or 'right' (default: 'right') whether this is a left or right representation

EXAMPLES:

```
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = WeylGroup(["A",3]).algebra(QQ)
sage: H = KW.demazure_lusztig_operators(q1,q2); H
```

(continues on next page)

(continued from previous page)

```
A representation of the (q1, q2)-Hecke algebra of type ['A', 3, 1]
on Algebra of Weyl Group of type ['A', 3]
(as a matrix group acting on the ambient space)
over Rational Field
```

```
>>> from sage.all import *
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = WeylGroup(["A",Integer(3)]).algebra(QQ)
>>> H = KW.demazure_lusztig_operators(q1,q2); H
A representation of the (q1, q2)-Hecke algebra of type ['A', 3, 1]
on Algebra of Weyl Group of type ['A', 3]
(as a matrix group acting on the ambient space)
over Rational Field
```

Among other things, it implements the T_w operators, their inverses and compositions thereof:

```
sage: H.Tw((1,2))
Generic endomorphism of Algebra of Weyl Group of type ['A', 3]
(as a matrix group acting on the ambient space) over Rational Field
```

```
>>> from sage.all import *
>>> H.Tw((Integer(1),Integer(2)))
Generic endomorphism of Algebra of Weyl Group of type ['A', 3]
(as a matrix group acting on the ambient space) over Rational Field
```

and the Cherednik operators $Y^{\lambda^\vee}$:

```
sage: H.Y()
Lazy family (...)_{i in Coroot lattice of the Root system of type ['A', 3, 1]}
```

```
>>> from sage.all import *
>>> H.Y()
Lazy family (...)_{i in Coroot lattice of the Root system of type ['A', 3, 1]}
```

REFERENCES:

- [HST2008]

Ti_inverse_on_basis (x, i)

The T_i^{-1} operators, on basis elements.

INPUT:

- `x` – the index of a basis element
- `i` – the index of a generator

EXAMPLES:

```
sage: W = WeylGroup("A3")
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2'].fraction_field()
```

(continues on next page)

(continued from previous page)

```
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1,q2)
sage: w = W.an_element()
sage: rho.Ti_inverse_on_basis(w, 1)
-1/q2*1231 + ((q1+q2)/(q1*q2))*123
```

```
>>> from sage.all import *
>>> W = WeylGroup("A3")
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> rho = KW.demazure_lusztig_operators(q1,q2)
>>> w = W.an_element()
>>> rho.Ti_inverse_on_basis(w, Integer(1))
-1/q2*1231 + ((q1+q2)/(q1*q2))*123
```

Ti_on_basis(*x*, *i*)

The T_i operators, on basis elements.

INPUT:

- x – the index of a basis element
- i – the index of a generator

EXAMPLES:

```
sage: W = WeylGroup("A3")
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1,q2)
sage: w = W.an_element()
sage: rho.Ti_on_basis(w,1)
q1*1231
```

```
>>> from sage.all import *
>>> W = WeylGroup("A3")
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> rho = KW.demazure_lusztig_operators(q1,q2)
>>> w = W.an_element()
>>> rho.Ti_on_basis(w,Integer(1))
q1*1231
```

Tw(*word*, *signs=None*, *scalar=None*)

Return T_w.

INPUT:

- `word` – a word $i_1, \ldots, i_k$ for some element w of the Weyl group See `straighten_word()` for how this word can be specified
- `signs` – list $\epsilon_1, \ldots, \epsilon_k$ of the same length as `word` with $\epsilon_i = \pm 1$ or `None` for $1, \ldots, 1$ (default: `None`)
- `scalar` – an element c of the base ring or `None` for 1 (default: `None`)

OUTPUT: a module morphism implementing

$$T_w = T_{i_k} \circ \cdots \circ T_{i_1}$$

in left action notation; that is T_{i_1} is applied first, then T_{i_2}, etc.

More generally, if `scalar` or `signs` is specified, the morphism implements

$$cT_{i_k}^{\epsilon_k} \circ \cdots \circ T_{i_1}^{\epsilon_k}.$$

EXAMPLES:

```
sage: W = WeylGroup("A3")
sage: W.element_class._repr_=lambda x: ('e' if not x.reduced_word()
....:                   else "".join(str(i) for i in x.reduced_word()))
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: x = KW.an_element(); x
123 + 3*2312 + 2*31 + e

sage: T = KW.demazure_lusztig_operators(q1,q2)
sage: T12 = T.Tw( (1,2) )
sage: T12(KW.one())
q1^2*12
```

```
>>> from sage.all import *
>>> W = WeylGroup("A3")
>>> W.element_class._repr_=lambda x: ('e' if not x.reduced_word()
...                   else "".join(str(i) for i in x.reduced_word()))
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> x = KW.an_element(); x
123 + 3*2312 + 2*31 + e

>>> T = KW.demazure_lusztig_operators(q1,q2)
>>> T12 = T.Tw( (Integer(1),Integer(2)) )
>>> T12(KW.one())
q1^2*12
```

This is $T_2 \circ T_1$:

```
sage: T[2](T[1](KW.one()))
q1^2*12
sage: T[1](T[2](KW.one()))
q1^2*21
sage: T12(x) == T[2](T[1](x))
True
```

```
>>> from sage.all import *
>>> T[Integer(2)](T[Integer(1)](KW.one()))
q1^2*12
>>> T[Integer(1)](T[Integer(2)](KW.one()))
q1^2*21
>>> T12(x) == T[Integer(2)](T[Integer(1)](x))
True
```

Now with signs and scalar coefficient we construct $3T_2 \circ T_1^{-1}$:

```
sage: phi = T.Tw((1,2), (-1,1), 3)
sage: phi(KW.one())
-3*q1/q2*12 + ((3*q1+3*q2)/q2)*2
sage: phi(T[1](x)) == 3*T[2](x)
True
```

```
>>> from sage.all import *
>>> phi = T.Tw((Integer(1),Integer(2)), (-Integer(1),Integer(1)), Integer(3))
>>> phi(KW.one())
-3*q1/q2*12 + ((3*q1+3*q2)/q2)*2
>>> phi(T[Integer(1)](x)) == Integer(3)*T[Integer(2)](x)
True
```

For debugging purposes, one can recover the input data:

```
sage: phi.word
(1, 2)
sage: phi.signs
(-1, 1)
sage: phi.scalar
3
```

```
>>> from sage.all import *
>>> phi.word
(1, 2)
>>> phi.signs
(-1, 1)
>>> phi.scalar
3
```

Tw_inverse(*word*)

Return T_w^{-1}.

This is essentially a shorthand for *Tw()* with all minus signs.

Todo

Add an example where $T_i \neq T_i^{-1}$

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_ = lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: KW = W.algebra(QQ)
sage: rho = KW.demazure_lusztig_operators(1, -1)
sage: x = KW.monomial(W.an_element()); x
123
sage: word = [1,2]
sage: rho.Tw(word)(x)
12312
sage: rho.Tw_inverse(word)(x)
12321

sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1, q2)
sage: x = KW.monomial(W.an_element()); x
123
sage: rho.Tw_inverse(word)(x)
1/q2^2*12321 - ((q1+q2)/(q1*q2^2))*1231 - ((q1+q2)/(q1*q2^2))*1232
+ ((q1^2+2*q1*q2+q2^2)/(q1^2*q2^2))*123
sage: rho.Tw(word)(_)
123
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> W.element_class._repr_ = lambda x: "".join(str(i) for i in x.reduced_
↪word())
>>> KW = W.algebra(QQ)
>>> rho = KW.demazure_lusztig_operators(Integer(1), -Integer(1))
>>> x = KW.monomial(W.an_element()); x
123
>>> word = [Integer(1),Integer(2)]
>>> rho.Tw(word)(x)
12312
>>> rho.Tw_inverse(word)(x)
12321

>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> rho = KW.demazure_lusztig_operators(q1, q2)
>>> x = KW.monomial(W.an_element()); x
123
>>> rho.Tw_inverse(word)(x)
1/q2^2*12321 - ((q1+q2)/(q1*q2^2))*1231 - ((q1+q2)/(q1*q2^2))*1232
+ ((q1^2+2*q1*q2+q2^2)/(q1^2*q2^2))*123
>>> rho.Tw(word)(_)
123
```

Y(*base_ring=Integer Ring*)

Return the Cherednik operators Y for this representation of an affine Hecke algebra.

INPUT:

- `self` – a representation of an affine Hecke algebra
- `base_ring` – the base ring of the coroot lattice

This is a family of operators indexed by the coroot lattice for this Cartan type. In practice this is currently indexed instead by the affine coroot lattice, even if this indexing is not one to one, in order to allow for $Y[\alpha_0^\vee]$.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q2, q1)
sage: Y = rho.Y(); Y
Lazy family (...(i))_{i in Coroot lattice of the Root system of type ['A', 3,␣
↪1]}
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> rho = KW.demazure_lusztig_operators(q2, q1)
>>> Y = rho.Y(); Y
Lazy family (...(i))_{i in Coroot lattice of the Root system of type ['A', 3,␣
↪1]}
```

Y_eigenvectors()

Return the family of eigenvectors for the Cherednik operators Y of this representation of an affine Hecke algebra.

INPUT:

- `self` – a representation of an affine Hecke algebra
- `base_ring` – the base ring of the coroot lattice

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: E = rho.Y_eigenvectors()
sage: E.keys()
Weyl Group of type ['B', 2] (as a matrix group acting on the ambient space)
sage: w0 = W.long_element()
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(2)])
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
```

(continues on next page)

(continued from previous page)

```
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> rho = KW.demazure_lusztig_operators(q1, q2, affine=True)
>>> E = rho.Y_eigenvectors()
>>> E.keys()
Weyl Group of type ['B', 2] (as a matrix group acting on the ambient space)
>>> w0 = W.long_element()
```

To set the recurrence up properly, one often needs to customize the *CherednikOperatorsEigenvectors.affine_lift()* and *CherednikOperatorsEigenvectors.affine_retract()* methods. This would usually be done by subclassing *CherednikOperatorsEigenvectors*; here we just override the methods directly.

In this particular case, we multiply by w_0 to take into account that w_0 is the seed for the recursion:

```
sage: E.affine_lift = w0._mul_
sage: E.affine_retract = w0._mul_

sage: E[w0]
2121
sage: E.eigenvalues(E[w0])
[q2^2/q1^2, q1/(-q2), q1/(-q2)]
```

```
>>> from sage.all import *
>>> E.affine_lift = w0._mul_
>>> E.affine_retract = w0._mul_

>>> E[w0]
2121
>>> E.eigenvalues(E[w0])
[q2^2/q1^2, q1/(-q2), q1/(-q2)]
```

This step is taken care of automatically if one instead calls the specialization `sage.coxeter_groups.CoxeterGroups.Algebras.demazure_lusztig_eigenvectors()`.

Now we can compute all eigenvectors:

```
sage: [E[w] for w in W]
[2121 - 121 - 212 + 12 + 21 - 1 - 2 + ,
 -2121 + 212,
 (q2/(q1-q2))*2121 + (q2/(-q1+q2))*121 + (q2/(-q1+q2))*212 - 12 - (q2/(-
↪q1+q2))*21 + 2,
 -(q2^2/(-q1^2+q1*q2-q2^2))*2121 - 121 + (q2^2/(-q1^2+q1*q2-q2^2))*212 + 21,
 -((q1^2+q2^2)/(q1^2-q1*q2+q2^2))*2121 - ((q1^2+q2^2)/(-q1^2+q1*q2-q2^2))*121
 - (q2^2/(-q1^2+q1*q2-q2^2))*212 + (q2^2/(-q1^2+q1*q2-q2^2))*12 - 21 + 1,
 2121,
 (q2/(-q1+q2))*2121 - (q2/(-q1+q2))*121 - 212 + 12,
 -2121 + 121]
```

```
>>> from sage.all import *
>>> [E[w] for w in W]
[2121 - 121 - 212 + 12 + 21 - 1 - 2 + ,
```

(continues on next page)

(continued from previous page)

```
-2121 + 212,
(q2/(q1-q2))*2121 + (q2/(-q1+q2))*121 + (q2/(-q1+q2))*212 - 12 - (q2/(-
↪q1+q2))*21 + 2,
-(q2^2/(-q1^2+q1*q2-q2^2))*2121 - 121 + (q2^2/(-q1^2+q1*q2-q2^2))*212 + 21,
-((q1^2+q2^2)/(q1^2-q1*q2+q2^2))*2121 - ((q1^2+q2^2)/(-q1^2+q1*q2-q2^2))*121
- (q2^2/(-q1^2+q1*q2-q2^2))*212 + (q2^2/(-q1^2+q1*q2-q2^2))*12 - 21 + 1,
2121,
(q2/(-q1+q2))*2121 - (q2/(-q1+q2))*121 - 212 + 12,
-2121 + 121]
```

Y_lambdacheck(*lambdacheck*)

Return the Cherednik operators $Y^{\lambda^\vee}$ for this representation of an affine Hecke algebra.

INPUT:

- `lambdacheck` – an element of the coroot lattice for this Cartan type

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(2)])
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
```

We take q_2 and q_1 as eigenvalues to match with the notations of [HST2008]

```
sage: rho = KW.demazure_lusztig_operators(q2, q1)
sage: L = rho.Y().keys()
sage: alpha = L.simple_roots()
sage: Y0 = rho.Y_lambdacheck(alpha[0])
sage: Y1 = rho.Y_lambdacheck(alpha[1])
sage: Y2 = rho.Y_lambdacheck(alpha[2])

sage: x = KW.monomial(W.an_element()); x
12
sage: Y1(x)
-((q1^2+2*q1*q2+q2^2)/(-q2^2))*2121
+ ((q1^3+q1^2*q2+q1*q2^2+q2^3)/(-q1*q2^2))*121
+ ((q1^2+q1*q2)/(-q2^2))*212 - (q1^2/(-q2^2))*12
sage: Y2(x)
-((q1^4+q1^3*q2+q1*q2^3+q2^4)/(-q1^3*q2))*2121
+ ((q1^3+q1^2*q2+q1*q2^2+q2^3)/(-q1^2*q2))*121
+ (q2^3/(-q1^3))*12
sage: Y1(Y2(x))
```

(continues on next page)

(continued from previous page)

```
((q1*q2+q2^2)/q1^2)*212 - q2/q1*12
sage: Y2(Y1(x))
((q1*q2+q2^2)/q1^2)*212 - q2/q1*12
```

```
>>> from sage.all import *
>>> rho = KW.demazure_lusztig_operators(q2, q1)
>>> L = rho.Y().keys()
>>> alpha = L.simple_roots()
>>> Y0 = rho.Y_lambdacheck(alpha[Integer(0)])
>>> Y1 = rho.Y_lambdacheck(alpha[Integer(1)])
>>> Y2 = rho.Y_lambdacheck(alpha[Integer(2)])

>>> x = KW.monomial(W.an_element()); x
12
>>> Y1(x)
-((q1^2+2*q1*q2+q2^2)/(-q2^2))*2121
+ ((q1^3+q1^2*q2+q1*q2^2+q2^3)/(-q1*q2^2))*121
+ ((q1^2+q1*q2)/(-q2^2))*212 - (q1^2/(-q2^2))*12
>>> Y2(x)
-((q1^4+q1^3*q2+q1*q2^3+q2^4)/(-q1^3*q2))*2121
+ ((q1^3+q1^2*q2+q1*q2^2+q2^3)/(-q1^2*q2))*121
+ (q2^3/(-q1^3))*12
>>> Y1(Y2(x))
((q1*q2+q2^2)/q1^2)*212 - q2/q1*12
>>> Y2(Y1(x))
((q1*q2+q2^2)/q1^2)*212 - q2/q1*12
```

The Y operators commute:

```
sage: Y0(Y1(x)) - Y1(Y0(x))
0
sage: Y2(Y1(x)) - Y1(Y2(x))
0
```

```
>>> from sage.all import *
>>> Y0(Y1(x)) - Y1(Y0(x))
0
>>> Y2(Y1(x)) - Y1(Y2(x))
0
```

In the classical root lattice, $\alpha_0 + \alpha_1 + \alpha_2 = 0$:

```
sage: Y0(Y1(Y2(x)))
12
```

```
>>> from sage.all import *
>>> Y0(Y1(Y2(x)))
12
```

Lemma 7.2 of [HST2008]:

```
sage: w0 = KW.monomial(W.long_element())
sage: rho.Tw(0)(w0)
q2
sage: rho.Tw_inverse(1)(w0)
1/q2*212
sage: rho.Tw_inverse(2)(w0)
1/q2*121
```

```
>>> from sage.all import *
>>> w0 = KW.monomial(W.long_element())
>>> rho.Tw(Integer(0))(w0)
q2
>>> rho.Tw_inverse(Integer(1))(w0)
1/q2*212
>>> rho.Tw_inverse(Integer(2))(w0)
1/q2*121
```

Lemma 7.5 of [HST2008]:

```
sage: Y0(w0)
q1^2/q2^2*2121
sage: Y1(w0)
(q2/(-q1))*2121
sage: Y2(w0)
(q2/(-q1))*2121
```

```
>>> from sage.all import *
>>> Y0(w0)
q1^2/q2^2*2121
>>> Y1(w0)
(q2/(-q1))*2121
>>> Y2(w0)
(q2/(-q1))*2121
```

Todo

Add more tests

Add tests in type BC affine where the null coroot $\delta^\vee$ can have non trivial coefficient in term of α_0

See also:

- [HST2008] for the formula in terms of q_1, q_2

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.hecke_algebra_representation import␣
↪HeckeAlgebraRepresentation
sage: KW = SymmetricGroup(3).algebra(QQ)
sage: action = lambda x,i: KW.monomial(x.apply_simple_reflection(i, side=
↪'right'))
sage: H = HeckeAlgebraRepresentation(KW, action, CartanType(["A",2]), 1, -1)
sage: H.cartan_type()
['A', 2]

sage: H = WeylGroup(["A",3]).algebra(QQ).demazure_lusztig_operators(-1,1)
sage: H.cartan_type()
['A', 3, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.hecke_algebra_representation import␣
↪HeckeAlgebraRepresentation
>>> KW = SymmetricGroup(Integer(3)).algebra(QQ)
>>> action = lambda x,i: KW.monomial(x.apply_simple_reflection(i, side='right
↪'))
>>> H = HeckeAlgebraRepresentation(KW, action, CartanType(["A",Integer(2)]),␣
↪Integer(1), -Integer(1))
>>> H.cartan_type()
['A', 2]

>>> H = WeylGroup(["A",Integer(3)]).algebra(QQ).demazure_lusztig_operators(-
↪Integer(1),Integer(1))
>>> H.cartan_type()
['A', 3, 1]
```

domain()

Return the domain of `self`.

EXAMPLES:

```
sage: H = WeylGroup(["A",3]).algebra(QQ).demazure_lusztig_operators(-1,1)
sage: H.domain()
Algebra of Weyl Group of type ['A', 3] (as a matrix group
 acting on the ambient space) over Rational Field
```

```
>>> from sage.all import *
>>> H = WeylGroup(["A",Integer(3)]).algebra(QQ).demazure_lusztig_operators(-
↪Integer(1),Integer(1))
>>> H.domain()
Algebra of Weyl Group of type ['A', 3] (as a matrix group
 acting on the ambient space) over Rational Field
```

on_basis(*x*, *word*, *signs=None*, *scalar=None*)

Action of product of T_i and T_i^{-1} on x.

INPUT:

- `x` – the index of a basis element
- `word` – word of indices of generators

- `signs` – (default: `None`) sequence of signs of same length as `word`; determines which operators are supposed to be taken as inverses.
- `scalar` – (default: `None`) scalar to multiply the answer by

EXAMPLES:

```
sage: from sage.combinat.root_system.hecke_algebra_representation import␣
↪HeckeAlgebraRepresentation
sage: W = SymmetricGroup(3)
sage: domain = W.algebra(QQ)
sage: action = lambda x,i: domain.monomial(x.apply_simple_reflection(i, side=
↪'right'))
sage: rho = HeckeAlgebraRepresentation(domain, action, CartanType(["A",2]), 1,
↪ -1)

sage: rho.on_basis(W.one(), (1,2,1))
(1,3)

sage: word = (1,2)
sage: u = W.from_reduced_word(word)
sage: for w in W:  assert rho.on_basis(w, word) == domain.monomial(w*u)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.hecke_algebra_representation import␣
↪HeckeAlgebraRepresentation
>>> W = SymmetricGroup(Integer(3))
>>> domain = W.algebra(QQ)
>>> action = lambda x,i: domain.monomial(x.apply_simple_reflection(i, side=
↪'right'))
>>> rho = HeckeAlgebraRepresentation(domain, action, CartanType(["A",
↪Integer(2)]), Integer(1), -Integer(1))

>>> rho.on_basis(W.one(), (Integer(1),Integer(2),Integer(1)))
(1,3)

>>> word = (Integer(1),Integer(2))
>>> u = W.from_reduced_word(word)
>>> for w in W:  assert rho.on_basis(w, word) == domain.monomial(w*u)
```

The next example tests the signs:

```
sage: W = WeylGroup("A3")
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_
↪word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1,q2)
sage: w = W.an_element(); w
123
sage: rho.on_basis(w, (1,),  signs=(-1,))
-1/q2*1231 + ((q1+q2)/(q1*q2))*123
sage: rho.on_basis(w, (1,),  signs=( 1,))
```

(continues on next page)

(continued from previous page)

```
q1*1231
sage: rho.on_basis(w, (1,1), signs=(1,-1))
123
sage: rho.on_basis(w, (1,1), signs=(-1,1))
123
```

```
>>> from sage.all import *
>>> W = WeylGroup("A3")
>>> W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> rho = KW.demazure_lusztig_operators(q1,q2)
>>> w = W.an_element(); w
123
>>> rho.on_basis(w, (Integer(1),),  signs=(-Integer(1),))
-1/q2*1231 + ((q1+q2)/(q1*q2))*123
>>> rho.on_basis(w, (Integer(1),),  signs=( Integer(1),))
q1*1231
>>> rho.on_basis(w, (Integer(1),Integer(1)), signs=(Integer(1),-Integer(1)))
123
>>> rho.on_basis(w, (Integer(1),Integer(1)), signs=(-Integer(1),Integer(1)))
123
```

parameters(i)

Return q_1, q_2 such that $(T_i - q_1)(T_i - q_2) = 0$.

EXAMPLES:

```
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = WeylGroup(["A",3]).algebra(QQ)
sage: H = KW.demazure_lusztig_operators(q1,q2)
sage: H.parameters(1)
(q1, q2)

sage: H = KW.demazure_lusztig_operators(1,-1)
sage: H.parameters(1)
(1, -1)
```

```
>>> from sage.all import *
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = WeylGroup(["A",Integer(3)]).algebra(QQ)
>>> H = KW.demazure_lusztig_operators(q1,q2)
>>> H.parameters(Integer(1))
(q1, q2)

>>> H = KW.demazure_lusztig_operators(Integer(1),-Integer(1))
>>> H.parameters(Integer(1))
(1, -1)
```

Todo

At this point, this method is constant. It's meant as a starting point for implementing parameters depending on the node i of the Dynkin diagram.

straighten_word(*word*)

Return a tuple of indices of generators after some straightening.

INPUT:

- `word` – a list/tuple of indices of generators, the index of a generator, or an object with a reduced word method

OUTPUT: a tuple of indices of generators

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: H = W.algebra(QQ).demazure_lusztig_operators(-1,1)
sage: H.straighten_word(1)
(1,)
sage: H.straighten_word((2,1))
(2, 1)
sage: H.straighten_word([2,1])
(2, 1)
sage: H.straighten_word(W.an_element())
(1, 2, 3)
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> H = W.algebra(QQ).demazure_lusztig_operators(-Integer(1),Integer(1))
>>> H.straighten_word(Integer(1))
(1,)
>>> H.straighten_word((Integer(2),Integer(1)))
(2, 1)
>>> H.straighten_word([Integer(2),Integer(1)])
(2, 1)
>>> H.straighten_word(W.an_element())
(1, 2, 3)
```

5.1.232 Integrable Representations of Affine Lie Algebras

class sage.combinat.root_system.integrable_representations.**IntegrableRepresentation**(*Lam*)

Bases: `UniqueRepresentation`, `CategoryObject`

An irreducible integrable highest weight representation of an affine Lie algebra.

INPUT:

- `Lam` – a dominant weight in an extended weight lattice of affine type

REFERENCES:

- [Ka1990]

If Λ is a dominant integral weight for an affine root system, there exists a unique integrable representation $V = V_\Lambda$ of highest weight Λ. If μ is another weight, let $m(\mu)$ denote the multiplicity of the weight μ in this representation.

The set supp(V) of μ such that $m(\mu) > 0$ is contained in the paraboloid

$$(\Lambda + \rho|\Lambda + \rho) - (\mu + \rho|\mu + \rho) \geq 0$$

where $(|)$ is the invariant inner product on the weight lattice and ρ is the Weyl vector. Moreover if $m(\mu) > 0$ then $\mu \in$ supp(V) differs from Λ by an element of the root lattice ([Ka1990], Propositions 11.3 and 11.4).

Let δ be the nullroot, which is the lowest positive imaginary root. Then by [Ka1990], Proposition 11.3 or Corollary 11.9, for fixed μ the function $m(\mu - k\delta)$ is a monotone increasing function of k. It is useful to take μ to be such that this function is nonzero if and only if $k \geq 0$. Therefore we make the following definition. If μ is such that $m(\mu) \neq 0$ but $m(\mu + \delta) = 0$ then μ is called *maximal*.

Since δ is fixed under the action of the affine Weyl group, and since the weight multiplicities are Weyl group invariant, the function $k \mapsto m(\mu - k\delta)$ is unchanged if μ is replaced by an equivalent weight. Therefore in tabulating these functions, we may assume that μ is dominant. There are only a finite number of dominant maximal weights.

Since every nonzero weight multiplicity appears in the string $\mu - k\delta$ for one of the finite number of dominant maximal weights μ, it is important to be able to compute these. We may do this as follows.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',3,1]).weight_lattice(extended=true).fundamental_
↪weights()
sage: IntegrableRepresentation(Lambda[1]+Lambda[2]+Lambda[3]).print_strings()
2*Lambda[0] + Lambda[2]: 4 31 161 665 2380 7658 22721 63120 166085 417295 1007601␣
↪2349655
Lambda[0] + 2*Lambda[1]: 2 18 99 430 1593 5274 16005 45324 121200 308829 754884␣
↪1779570
Lambda[0] + 2*Lambda[3]: 2 18 99 430 1593 5274 16005 45324 121200 308829 754884␣
↪1779570
Lambda[1] + Lambda[2] + Lambda[3]: 1 10 60 274 1056 3601 11199 32354 88009 227555␣
↪563390 1343178
3*Lambda[2] - delta: 3 21 107 450 1638 5367 16194 45687 121876 310056 757056␣
↪1783324
sage: Lambda = RootSystem(['D',4,1]).weight_lattice(extended=true).fundamental_
↪weights()
sage: IntegrableRepresentation(Lambda[0]+Lambda[1]).print_strings()                ␣
↪         # long time
Lambda[0] + Lambda[1]: 1 10 62 293 1165 4097 13120 38997 109036 289575 735870␣
↪1799620
Lambda[3] + Lambda[4] - delta: 3 25 136 590 2205 7391 22780 65613 178660 463842␣
↪1155717 2777795
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(3),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> IntegrableRepresentation(Lambda[Integer(1)]+Lambda[Integer(2)]+Lambda[Inte-
ger(3)]).print_strings()
2*Lambda[0] + Lambda[2]: 4 31 161 665 2380 7658 22721 63120 166085 417295 1007601␣
↪2349655
Lambda[0] + 2*Lambda[1]: 2 18 99 430 1593 5274 16005 45324 121200 308829 754884␣
↪1779570
Lambda[0] + 2*Lambda[3]: 2 18 99 430 1593 5274 16005 45324 121200 308829 754884␣
↪1779570
```

(continues on next page)

(continued from previous page)

```
Lambda[1] + Lambda[2] + Lambda[3]: 1 10 60 274 1056 3601 11199 32354 88009 227555␣
↪563390 1343178
3*Lambda[2] - delta: 3 21 107 450 1638 5367 16194 45687 121876 310056 757056␣
↪1783324
>>> Lambda = RootSystem(['D',Integer(4),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> IntegrableRepresentation(Lambda[Integer(0)]+Lambda[Integer(1)]).print_
↪strings()                        # long time
Lambda[0] + Lambda[1]: 1 10 62 293 1165 4097 13120 38997 109036 289575 735870␣
↪1799620
Lambda[3] + Lambda[4] - delta: 3 25 136 590 2205 7391 22780 65613 178660 463842␣
↪1155717 2777795
```

In this example, we construct the extended weight lattice of Cartan type $A_3^{(1)}$, then define `Lambda` to be the fundamental weights $(\Lambda_i)_{i\in I}$. We find there are 5 maximal dominant weights in irreducible representation of highest weight $\Lambda_1 + \Lambda_2 + \Lambda_3$, and we determine their strings.

It was shown in [KacPeterson] that each string is the set of Fourier coefficients of a modular form.

Every weight μ such that the weight multiplicity $m(\mu)$ is nonzero has the form

$$\Lambda - n_0\alpha_0 - n_1\alpha_1 - \cdots,$$

where the n_i are nonnegative integers. This is represented internally as a tuple $(n_0, n_1, n_2, \ldots)$. If you want an individual multiplicity you use the method *m()* and supply it with this tuple:

```
sage: Lambda = RootSystem(['C',2,1]).weight_lattice(extended=true).fundamental_
↪weights()
sage: V = IntegrableRepresentation(2*Lambda[0]); V
Integrable representation of ['C', 2, 1] with highest weight 2*Lambda[0]
sage: V.m((3,5,3))
18
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['C',Integer(2),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(2)*Lambda[Integer(0)]); V
Integrable representation of ['C', 2, 1] with highest weight 2*Lambda[0]
>>> V.m((Integer(3),Integer(5),Integer(3)))
18
```

The *IntegrableRepresentation* class has methods *to_weight()* and *from_weight()* to convert between this internal representation and the weight lattice:

```
sage: delta = V.weight_lattice().null_root()
sage: V.to_weight((4,3,2))
-3*Lambda[0] + 6*Lambda[1] - Lambda[2] - 4*delta
sage: V.from_weight(-3*Lambda[0] + 6*Lambda[1] - Lambda[2] - 4*delta)
(4, 3, 2)
```

```
>>> from sage.all import *
>>> delta = V.weight_lattice().null_root()
>>> V.to_weight((Integer(4),Integer(3),Integer(2)))
-3*Lambda[0] + 6*Lambda[1] - Lambda[2] - 4*delta
>>> V.from_weight(-Integer(3)*Lambda[Integer(0)] + Integer(6)*Lambda[Integer(1)] -
↪ Lambda[Integer(2)] - Integer(4)*delta)
(4, 3, 2)
```

To get more values, use the depth parameter:

```
sage: L0 = RootSystem(["A",1,1]).weight_lattice(extended=true).fundamental_
↪weight(0); L0
Lambda[0]
sage: IntegrableRepresentation(4*L0).print_strings(depth=20)
4*Lambda[0]: 1 1 3 6 13 23 44 75 131 215 354 561 889 1368 2097 3153 4712 6936␣
↪10151 14677
2*Lambda[0] + 2*Lambda[1] - delta: 1 2 5 10 20 36 66 112 190 310 501 788 1230␣
↪1880 2850 4256 6303 9222 13396 19262
4*Lambda[1] - 2*delta: 1 2 6 11 23 41 75 126 215 347 561 878 1368 2082 3153 4690␣
↪6936 10121 14677 21055
```

```
>>> from sage.all import *
>>> L0 = RootSystem(["A",Integer(1),Integer(1)]).weight_lattice(extended=true).
↪fundamental_weight(Integer(0)); L0
Lambda[0]
>>> IntegrableRepresentation(Integer(4)*L0).print_strings(depth=Integer(20))
4*Lambda[0]: 1 1 3 6 13 23 44 75 131 215 354 561 889 1368 2097 3153 4712 6936␣
↪10151 14677
2*Lambda[0] + 2*Lambda[1] - delta: 1 2 5 10 20 36 66 112 190 310 501 788 1230␣
↪1880 2850 4256 6303 9222 13396 19262
4*Lambda[1] - 2*delta: 1 2 6 11 23 41 75 126 215 347 561 878 1368 2082 3153 4690␣
↪6936 10121 14677 21055
```

An example in type $C_2^{(1)}$:

```
sage: Lambda = RootSystem(['C',2,1]).weight_lattice(extended=true).fundamental_
↪weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: V.print_strings()     # long time
2*Lambda[0]: 1 2 9 26 77 194 477 1084 2387 5010 10227 20198
Lambda[0] + Lambda[2] - delta: 1 5 18 55 149 372 872 1941 4141 8523 17005 33019
2*Lambda[1] - delta: 1 4 15 44 122 304 721 1612 3469 7176 14414 28124
2*Lambda[2] - 2*delta: 2 7 26 72 194 467 1084 2367 5010 10191 20198 38907
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['C',Integer(2),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(2)*Lambda[Integer(0)])
>>> V.print_strings()     # long time
2*Lambda[0]: 1 2 9 26 77 194 477 1084 2387 5010 10227 20198
Lambda[0] + Lambda[2] - delta: 1 5 18 55 149 372 872 1941 4141 8523 17005 33019
2*Lambda[1] - delta: 1 4 15 44 122 304 721 1612 3469 7176 14414 28124
2*Lambda[2] - 2*delta: 2 7 26 72 194 467 1084 2367 5010 10191 20198 38907
```

Examples for twisted affine types:

```
sage: Lambda = RootSystem(["A",2,2]).weight_lattice(extended=True).fundamental_
→weights()
sage: IntegrableRepresentation(Lambda[0]).strings()
{Lambda[0]: [1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56]}
sage: Lambda = RootSystem(['G',2,1]).dual.weight_lattice(extended=true).
→fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0]+Lambda[1]+Lambda[2])
sage: V.print_strings() # long time
6*Lambdacheck[0]: 4 28 100 320 944 2460 6064 14300 31968 69020 144676 293916
3*Lambdacheck[0] + Lambdacheck[1]: 2 16 58 192 588 1568 3952 9520 21644 47456␣
→100906 207536
4*Lambdacheck[0] + Lambdacheck[2]: 4 22 84 276 800 2124 5288 12470 28116 61056␣
→128304 261972
2*Lambdacheck[1] - deltacheck: 2 8 32 120 354 980 2576 6244 14498 32480 69776␣
→145528
Lambdacheck[0] + Lambdacheck[1] + Lambdacheck[2]: 1 6 26 94 294 832 2184 5388␣
→12634 28390 61488 128976
2*Lambdacheck[0] + 2*Lambdacheck[2]: 2 12 48 164 492 1344 3428 8256 18960 41844␣
→89208 184512
3*Lambdacheck[2] - deltacheck: 4 16 60 208 592 1584 4032 9552 21728 47776 101068␣
→207888
sage: Lambda = RootSystem(['A',6,2]).weight_lattice(extended=true).fundamental_
→weights()
sage: V = IntegrableRepresentation(Lambda[0]+2*Lambda[1])
sage: V.print_strings() # long time
5*Lambda[0]: 3 42 378 2508 13707 64650 272211 1045470 3721815 12425064 39254163␣
→118191378
3*Lambda[0] + Lambda[2]: 1 23 234 1690 9689 47313 204247 800029 2893198 9786257␣
→31262198 95035357
Lambda[0] + 2*Lambda[1]: 1 14 154 1160 6920 34756 153523 612354 2248318 7702198␣
→24875351 76341630
Lambda[0] + Lambda[1] + Lambda[3] - 2*delta: 6 87 751 4779 25060 113971 464842␣
→1736620 6034717 19723537 61152367 181068152
Lambda[0] + 2*Lambda[2] - 2*delta: 3 54 499 3349 18166 84836 353092 1341250␣
→4725259 15625727 48938396 146190544
Lambda[0] + 2*Lambda[3] - 4*delta: 15 195 1539 9186 45804 200073 789201 2866560␣
→9723582 31120281 94724550 275919741
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(["A",Integer(2),Integer(2)]).weight_
→lattice(extended=True).fundamental_weights()
>>> IntegrableRepresentation(Lambda[Integer(0)]).strings()
{Lambda[0]: [1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56]}
>>> Lambda = RootSystem(['G',Integer(2),Integer(1)]).dual.weight_
→lattice(extended=true).fundamental_weights()
>>> V =␣
→IntegrableRepresentation(Lambda[Integer(0)]+Lambda[Integer(1)]+Lambda[Integer(2)])
>>> V.print_strings() # long time
6*Lambdacheck[0]: 4 28 100 320 944 2460 6064 14300 31968 69020 144676 293916
3*Lambdacheck[0] + Lambdacheck[1]: 2 16 58 192 588 1568 3952 9520 21644 47456␣
→100906 207536
```

(continues on next page)

(continued from previous page)

```
4*Lambdacheck[0] + Lambdacheck[2]: 4 22 84 276 800 2124 5288 12470 28116 61056␣
↪128304 261972
2*Lambdacheck[1] - deltacheck: 2 8 32 120 354 980 2576 6244 14498 32480 69776␣
↪145528
Lambdacheck[0] + Lambdacheck[1] + Lambdacheck[2]: 1 6 26 94 294 832 2184 5388␣
↪12634 28390 61488 128976
2*Lambdacheck[0] + 2*Lambdacheck[2]: 2 12 48 164 492 1344 3428 8256 18960 41844␣
↪89208 184512
3*Lambdacheck[2] - deltacheck: 4 16 60 208 592 1584 4032 9552 21728 47776 101068␣
↪207888
>>> Lambda = RootSystem(['A',Integer(6),Integer(2)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Lambda[Integer(0)]+Integer(2)*Lambda[Integer(1)])
>>> V.print_strings() # long time
5*Lambda[0]: 3 42 378 2508 13707 64650 272211 1045470 3721815 12425064 39254163␣
↪118191378
3*Lambda[0] + Lambda[2]: 1 23 234 1690 9689 47313 204247 800029 2893198 9786257␣
↪31262198 95035357
Lambda[0] + 2*Lambda[1]: 1 14 154 1160 6920 34756 153523 612354 2248318 7702198␣
↪24875351 76341630
Lambda[0] + Lambda[1] + Lambda[3] - 2*delta: 6 87 751 4779 25060 113971 464842␣
↪1736620 6034717 19723537 61152367 181068152
Lambda[0] + 2*Lambda[2] - 2*delta: 3 54 499 3349 18166 84836 353092 1341250␣
↪4725259 15625727 48938396 146190544
Lambda[0] + 2*Lambda[3] - 4*delta: 15 195 1539 9186 45804 200073 789201 2866560␣
↪9723582 31120281 94724550 275919741
```

branch(*i=None*, *weyl_character_ring=None*, *sequence=None*, *depth=5*)

Return the branching rule on `self`.

Removing any node from the extended Dynkin diagram of the affine Lie algebra results in the Dynkin diagram of a classical Lie algebra, which is therefore a Lie subalgebra. For example removing the 0 node from the Dynkin diagram of type `[X, r, 1]` produces the classical Dynkin diagram of `[X, r]`.

Thus for each i in the index set, we may restrict `self` to the corresponding classical subalgebra. Of course `self` is an infinite dimensional representation, but each weight μ is assigned a grading by the number of times the simple root α_i appears in $\Lambda - \mu$. Thus the branched representation is graded and we get sequence of finite-dimensional representations which this method is able to compute.

OPTIONAL:

- `i` – (default: 0) an element of the index set
- `weyl_character_ring` – a WeylCharacterRing
- `sequence` – dictionary
- `depth` – (default: 5) an upper bound for k determining how many terms to give

In the default case where $i = 0$, you do not need to specify anything else, though you may want to increase the depth if you need more terms.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).
↪fundamental_weights()
```

(continues on next page)

(continued from previous page)

```
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: b = V.branch(); b                                                 #␣
↪needs sage.libs.gap
[A2(0,0),
 A2(1,1),
 A2(0,0) + 2*A2(1,1) + A2(2,2),
 2*A2(0,0) + 2*A2(0,3) + 4*A2(1,1) + 2*A2(3,0) + 2*A2(2,2),
 4*A2(0,0) + 3*A2(0,3) + 10*A2(1,1) + 3*A2(3,0) + A2(1,4) + 6*A2(2,2) + A2(4,
↪1),
 6*A2(0,0) + 9*A2(0,3) + 20*A2(1,1) + 9*A2(3,0) + 3*A2(1,4) + 12*A2(2,2) +␣
↪3*A2(4,1) + A2(3,3)]
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(2)*Lambda[Integer(0)])
>>> b = V.branch(); b                                                   #␣
↪needs sage.libs.gap
[A2(0,0),
 A2(1,1),
 A2(0,0) + 2*A2(1,1) + A2(2,2),
 2*A2(0,0) + 2*A2(0,3) + 4*A2(1,1) + 2*A2(3,0) + 2*A2(2,2),
 4*A2(0,0) + 3*A2(0,3) + 10*A2(1,1) + 3*A2(3,0) + A2(1,4) + 6*A2(2,2) + A2(4,
↪1),
 6*A2(0,0) + 9*A2(0,3) + 20*A2(1,1) + 9*A2(3,0) + 3*A2(1,4) + 12*A2(2,2) +␣
↪3*A2(4,1) + A2(3,3)]
```

If the parameter `weyl_character_ring` is omitted, the ring may be recovered as the parent of one of the branched coefficients:

```
sage: A2 = b[0].parent(); A2                                            #␣
↪needs sage.libs.gap
The Weyl Character Ring of Type A2 with Integer Ring coefficients
```

```
>>> from sage.all import *
>>> A2 = b[Integer(0)].parent(); A2                                      ␣
↪     # needs sage.libs.gap
The Weyl Character Ring of Type A2 with Integer Ring coefficients
```

If i is not zero then you should specify the `WeylCharacterRing` that you are branching to. This is determined by the Dynkin diagram:

```
sage: Lambda = RootSystem(['B',3,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.cartan_type().dynkin_diagram()
    O 0
    |
    |
O---O=>=O
1   2   3
B3~
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['B',Integer(3),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Lambda[Integer(0)])
>>> V.cartan_type().dynkin_diagram()
    O 0
    |
    |
O---O=>=O
1   2   3
B3~
```

In this example, we observe that removing the $i = 2$ node from the Dynkin diagram produces a reducible diagram of type `A1xA1xA1`. Thus we have a branching to $\mathfrak{sl}(2) \times \mathfrak{sl}(2) \times \mathfrak{sl}(2)$:

```
sage: A1xA1xA1 = WeylCharacterRing("A1xA1xA1",style='coroots')            #␣
↪needs sage.libs.gap
sage: V.branch(i=2,weyl_character_ring=A1xA1xA1)                          #␣
↪needs sage.libs.gap
[A1xA1xA1(1,0,0),
 A1xA1xA1(0,1,2),
 A1xA1xA1(1,0,0) + A1xA1xA1(1,2,0) + A1xA1xA1(1,0,2),
 A1xA1xA1(2,1,2) + A1xA1xA1(0,1,0) + 2*A1xA1xA1(0,1,2),
 3*A1xA1xA1(1,0,0) + 2*A1xA1xA1(1,2,0) + A1xA1xA1(1,2,2) + 2*A1xA1xA1(1,0,2)␣
↪+ A1xA1xA1(1,0,4) + A1xA1xA1(3,0,0),
 A1xA1xA1(2,1,0) + 3*A1xA1xA1(2,1,2) + 2*A1xA1xA1(0,1,0) + 5*A1xA1xA1(0,1,2)␣
↪+ A1xA1xA1(0,1,4) + A1xA1xA1(0,3,2)]
```

```
>>> from sage.all import *
>>> A1xA1xA1 = WeylCharacterRing("A1xA1xA1",style='coroots')            #␣
↪needs sage.libs.gap
>>> V.branch(i=Integer(2),weyl_character_ring=A1xA1xA1)                 ␣
↪     # needs sage.libs.gap
[A1xA1xA1(1,0,0),
 A1xA1xA1(0,1,2),
 A1xA1xA1(1,0,0) + A1xA1xA1(1,2,0) + A1xA1xA1(1,0,2),
 A1xA1xA1(2,1,2) + A1xA1xA1(0,1,0) + 2*A1xA1xA1(0,1,2),
 3*A1xA1xA1(1,0,0) + 2*A1xA1xA1(1,2,0) + A1xA1xA1(1,2,2) + 2*A1xA1xA1(1,0,2)␣
↪+ A1xA1xA1(1,0,4) + A1xA1xA1(3,0,0),
 A1xA1xA1(2,1,0) + 3*A1xA1xA1(2,1,2) + 2*A1xA1xA1(0,1,0) + 5*A1xA1xA1(0,1,2)␣
↪+ A1xA1xA1(0,1,4) + A1xA1xA1(0,3,2)]
```

If the nodes of the two Dynkin diagrams are not in the same order, you must specify an additional parameter, `sequence` which gives a dictionary to the affine Dynkin diagram to the classical one.

EXAMPLES:

```
sage: Lambda = RootSystem(['F',4,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.cartan_type().dynkin_diagram()
O---O---O=>=O---O
0   1   2   3   4
```

(continues on next page)

(continued from previous page)

```
F4~
sage: A1xC3=WeylCharacterRing("A1xC3",style='coroots')
sage: A1xC3.dynkin_diagram()
O
1
O---O=<=O
2   3   4
A1xC3
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['F',Integer(4),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Lambda[Integer(0)])
>>> V.cartan_type().dynkin_diagram()
O---O---O=>=O---O
0   1   2   3   4
F4~
>>> A1xC3=WeylCharacterRing("A1xC3",style='coroots')
>>> A1xC3.dynkin_diagram()
O
1
O---O=<=O
2   3   4
A1xC3
```

Observe that removing the $i = 1$ node from the `F4~` Dynkin diagram gives the `A1xC3` diagram, but the roots are in a different order. The nodes $0, 2, 3, 4$ of `F4~` correspond to `1, 4, 3, 2` of `A1xC3` and so we encode this in a dictionary:

```
sage: V.branch(i=1, weyl_character_ring=A1xC3, sequence={0:1,2:4,3:3,4:2})   #␣
↪long time
[A1xC3(1,0,0,0),
 A1xC3(0,0,0,1),
 A1xC3(1,0,0,0) + A1xC3(1,2,0,0),
 A1xC3(2,0,0,1) + A1xC3(0,0,0,1) + A1xC3(0,1,1,0),
 2*A1xC3(1,0,0,0) + A1xC3(1,0,1,0) + 2*A1xC3(1,2,0,0) + A1xC3(1,0,2,0) +␣
↪A1xC3(3,0,0,0),
 2*A1xC3(2,0,0,1) + A1xC3(2,1,1,0) + A1xC3(0,1,0,0) + 3*A1xC3(0,0,0,1) +␣
↪2*A1xC3(0,1,1,0) + A1xC3(0,2,0,1)]
```

```
>>> from sage.all import *
>>> V.branch(i=Integer(1), weyl_character_ring=A1xC3, sequence=
↪{Integer(0):Integer(1),Integer(2):Integer(4),Integer(3):Integer(3),
↪Integer(4):Integer(2)})   # long time
[A1xC3(1,0,0,0),
 A1xC3(0,0,0,1),
 A1xC3(1,0,0,0) + A1xC3(1,2,0,0),
 A1xC3(2,0,0,1) + A1xC3(0,0,0,1) + A1xC3(0,1,1,0),
 2*A1xC3(1,0,0,0) + A1xC3(1,0,1,0) + 2*A1xC3(1,2,0,0) + A1xC3(1,0,2,0) +␣
↪A1xC3(3,0,0,0),
 2*A1xC3(2,0,0,1) + A1xC3(2,1,1,0) + A1xC3(0,1,0,0) + 3*A1xC3(0,0,0,1) +␣
↪2*A1xC3(0,1,1,0) + A1xC3(0,2,0,1)]
```

The branch method gives a way of computing the graded dimension of the integrable representation:

```
sage: Lambda = RootSystem("A1~").weight_lattice(extended=true).fundamental_
↪weights()
sage: V=IntegrableRepresentation(Lambda[0])
sage: r = [x.degree() for x in V.branch(depth=15)]; r
[1, 3, 4, 7, 13, 19, 29, 43, 62, 90, 126, 174, 239, 325, 435, 580]
sage: oeis(r)                                                             #␣
↪optional -- internet
0: A029552: Expansion of phi(x) / f(-x) in powers of x where phi(), f() are␣
↪Ramanujan theta functions.
```

```
>>> from sage.all import *
>>> Lambda = RootSystem("A1~").weight_lattice(extended=true).fundamental_
↪weights()
>>> V=IntegrableRepresentation(Lambda[Integer(0)])
>>> r = [x.degree() for x in V.branch(depth=Integer(15))]; r
[1, 3, 4, 7, 13, 19, 29, 43, 62, 90, 126, 174, 239, 325, 435, 580]
>>> oeis(r)                                                    # optional␣
↪-- internet
0: A029552: Expansion of phi(x) / f(-x) in powers of x where phi(), f() are␣
↪Ramanujan theta functions.
```

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: Lambda = RootSystem(['F',4,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.cartan_type()
['F', 4, 1]
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['F',Integer(4),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Lambda[Integer(0)])
>>> V.cartan_type()
['F', 4, 1]
```

coxeter_number()

Return the Coxeter number of the Cartan type of `self`.

The Coxeter number is defined in [Ka1990] Chapter 6, and commonly denoted h.

EXAMPLES:

```
sage: Lambda = RootSystem(['F',4,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.coxeter_number()
12
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['F',Integer(4),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Lambda[Integer(0)])
>>> V.coxeter_number()
12
```

dominant_maximal_weights()

Return the dominant maximal weights of `self`.

A weight μ is *maximal* if it has nonzero multiplicity but $\mu + \delta$` has multiplicity zero. There are a finite number of dominant maximal weights. Indeed, [Ka1990] Proposition 12.6 shows that the dominant maximal weights are in bijection with the classical weights in $k \cdot F$ where F is the fundamental alcove and k is the level. The construction used in this method is based on that Proposition.

EXAMPLES:

```
sage: Lambda = RootSystem(['C',3,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: IntegrableRepresentation(2*Lambda[0]).dominant_maximal_weights()
(2*Lambda[0],
 Lambda[0] + Lambda[2] - delta,
 2*Lambda[1] - delta,
 Lambda[1] + Lambda[3] - 2*delta,
 2*Lambda[2] - 2*delta,
 2*Lambda[3] - 3*delta)
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['C',Integer(3),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> IntegrableRepresentation(Integer(2)*Lambda[Integer(0)]).dominant_maximal_
↪weights()
(2*Lambda[0],
 Lambda[0] + Lambda[2] - delta,
 2*Lambda[1] - delta,
 Lambda[1] + Lambda[3] - 2*delta,
 2*Lambda[2] - 2*delta,
 2*Lambda[3] - 3*delta)
```

dual_coxeter_number()

Return the dual Coxeter number of the Cartan type of `self`.

The dual Coxeter number is defined in [Ka1990] Chapter 6, and commonly denoted $h^\vee$.

EXAMPLES:

```
sage: Lambda = RootSystem(['F',4,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.dual_coxeter_number()
9
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['F',Integer(4),Integer(1)]).weight_
```

(continues on next page)

(continued from previous page)

```
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Lambda[Integer(0)])
>>> V.dual_coxeter_number()
9
```

from_weight(*mu*)

Return the tuple $(n_0, n_1, ...)$` such that `mu` equals $\Lambda - \sum_{i\in I} n_i\alpha_i$ in `self`, where $\Lambda$ is the highest weight of `self`.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[2])
sage: V.to_weight((1,0,0))
-2*Lambda[0] + Lambda[1] + 3*Lambda[2] - delta
sage: delta = V.weight_lattice().null_root()
sage: V.from_weight(-2*Lambda[0] + Lambda[1] + 3*Lambda[2] - delta)
(1, 0, 0)
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(2)*Lambda[Integer(2)])
>>> V.to_weight((Integer(1),Integer(0),Integer(0)))
-2*Lambda[0] + Lambda[1] + 3*Lambda[2] - delta
>>> delta = V.weight_lattice().null_root()
>>> V.from_weight(-Integer(2)*Lambda[Integer(0)] + Lambda[Integer(1)] +␣
↪Integer(3)*Lambda[Integer(2)] - delta)
(1, 0, 0)
```

highest_weight()

Return the highest weight of `self`.

EXAMPLES:

```
sage: Lambda = RootSystem(['D',4,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: IntegrableRepresentation(Lambda[0]+2*Lambda[2]).highest_weight()
Lambda[0] + 2*Lambda[2]
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['D',Integer(4),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>>␣
↪IntegrableRepresentation(Lambda[Integer(0)]+Integer(2)*Lambda[Integer(2)]).
↪highest_weight()
Lambda[0] + 2*Lambda[2]
```

level()

Return the level of `self`.

The level of a highest weight representation V_Λ is defined as $(\Lambda|\delta)$ See [Ka1990] section 12.4.

EXAMPLES:

```
sage: Lambda = RootSystem(['G',2,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: [IntegrableRepresentation(Lambda[i]).level() for i in [0,1,2]]
[1, 1, 2]
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['G',Integer(2),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> [IntegrableRepresentation(Lambda[i]).level() for i in [Integer(0),
↪Integer(1),Integer(2)]]
[1, 1, 2]
```

m(*n*)

Return the multiplicity of the weight μ in `self`, where $\mu = \Lambda - \sum_i n_i \alpha_i$.

INPUT:

- `n` – tuple representing a weight μ

EXAMPLES:

```
sage: Lambda = RootSystem(['E',6,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: u = V.highest_weight() - V.weight_lattice().null_root()
sage: V.from_weight(u)
(1, 1, 2, 2, 3, 2, 1)
sage: V.m(V.from_weight(u))
6
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['E',Integer(6),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Lambda[Integer(0)])
>>> u = V.highest_weight() - V.weight_lattice().null_root()
>>> V.from_weight(u)
(1, 1, 2, 2, 3, 2, 1)
>>> V.m(V.from_weight(u))
6
```

modular_characteristic(*mu=None*)

Return the modular characteristic of `self`.

The modular characteristic is a rational number introduced by Kac and Peterson [KacPeterson], required to interpret the string functions as Fourier coefficients of modular forms. See [Ka1990] Section 12.7. Let k be the level, and let $h^\vee$ be the dual Coxeter number. Then

$$m_\Lambda = \frac{|\Lambda+\rho|^2}{2(k+h^\vee)} - \frac{|\rho|^2}{2h^\vee}$$

If μ is a weight, then

$$m_{\Lambda,\mu} = m_\Lambda - \frac{|\mu|^2}{2k}.$$

OPTIONAL:

- mu – a weight, or alternatively,
- n – tuple representing a weight μ

If no optional parameter is specified, this returns m_Λ. If mu is specified, it returns $m_{\Lambda,\mu}$. You may use the tuple n to specify μ. If you do this, μ is $\Lambda - \sum_i n_i\alpha_i$.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',1,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(3*Lambda[0]+2*Lambda[1])
sage: [V.modular_characteristic(x) for x in V.dominant_maximal_weights()]
[11/56, -1/280, 111/280]
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(1),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(3)*Lambda[Integer(0)]+Integer(2)*Lamb
↪da[Integer(1)])
>>> [V.modular_characteristic(x) for x in V.dominant_maximal_weights()]
[11/56, -1/280, 111/280]
```

mult(*mu*)

Return the weight multiplicity of mu.

INPUT:

- mu – an element of the weight lattice

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: L = RootSystem("B3~").weight_lattice(extended=True)
sage: Lambda = L.fundamental_weights()
sage: delta = L.null_root()
sage: W = L.weyl_group(prefix='s')
sage: [s0,s1,s2,s3] = W.simple_reflections()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.mult(Lambda[2] - 2*delta)
3
sage: V.mult(Lambda[2] - Lambda[1])
0
sage: weights = [w.action(Lambda[1] - 4*delta) for w in [s1,s2,s0*s1*s2*s3]]
sage: weights
[-Lambda[1] + Lambda[2] - 4*delta,
 Lambda[1] - 4*delta,
 -Lambda[1] + Lambda[2] - 4*delta]
sage: [V.mult(mu) for mu in weights]
[35, 35, 35]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> L = RootSystem("B3~").weight_lattice(extended=True)
>>> Lambda = L.fundamental_weights()
>>> delta = L.null_root()
```

(continues on next page)

(continued from previous page)

```
>>> W = L.weyl_group(prefix='s')
>>> [s0,s1,s2,s3] = W.simple_reflections()
>>> V = IntegrableRepresentation(Lambda[Integer(0)])
>>> V.mult(Lambda[Integer(2)] - Integer(2)*delta)
3
>>> V.mult(Lambda[Integer(2)] - Lambda[Integer(1)])
0
>>> weights = [w.action(Lambda[Integer(1)] - Integer(4)*delta) for w in [s1,
↪s2,s0*s1*s2*s3]]
>>> weights
[-Lambda[1] + Lambda[2] - 4*delta,
 Lambda[1] - 4*delta,
 -Lambda[1] + Lambda[2] - 4*delta]
>>> [V.mult(mu) for mu in weights]
[35, 35, 35]
```

print_strings(*depth=12*)

Print the strings of `self`.

> **See also**
>
> *strings()*

EXAMPLES:

```
sage: Lambda = RootSystem(['A',1,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: V.print_strings(depth=25)
2*Lambda[0]: 1 1 3 5 10 16 28 43 70 105 161 236 350 501 722 1016 1431 1981␣
↪2741 3740 5096 6868 9233 12306 16357
2*Lambda[1] - delta: 1 2 4 7 13 21 35 55 86 130 196 287 420 602 858 1206 1687␣
↪2331 3206 4368 5922 7967 10670 14193 18803
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(1),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(2)*Lambda[Integer(0)])
>>> V.print_strings(depth=Integer(25))
2*Lambda[0]: 1 1 3 5 10 16 28 43 70 105 161 236 350 501 722 1016 1431 1981␣
↪2741 3740 5096 6868 9233 12306 16357
2*Lambda[1] - delta: 1 2 4 7 13 21 35 55 86 130 196 287 420 602 858 1206 1687␣
↪2331 3206 4368 5922 7967 10670 14193 18803
```

root_lattice()

Return the root lattice associated to `self`.

EXAMPLES:

```
sage: V=IntegrableRepresentation(RootSystem(['F',4,1]).weight_
↪lattice(extended=true).fundamental_weight(0))
```

(continues on next page)

(continued from previous page)

```
sage: V.root_lattice()
Root lattice of the Root system of type ['F', 4, 1]
```

```
>>> from sage.all import *
>>> V=IntegrableRepresentation(RootSystem(['F',Integer(4),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weight(Integer(0)))
>>> V.root_lattice()
Root lattice of the Root system of type ['F', 4, 1]
```

s (*n*, *i*)

Return the action of the `i`-th simple reflection on the internal representation of weights by tuples `n` in `self`.

EXAMPLES:

```
sage: V = IntegrableRepresentation(RootSystem(['A',2,1]).weight_
↪lattice(extended=true).fundamental_weight(0))
sage: [V.s((0,0,0),i) for i in V._index_set]
[(1, 0, 0), (0, 0, 0), (0, 0, 0)]
```

```
>>> from sage.all import *
>>> V = IntegrableRepresentation(RootSystem(['A',Integer(2),Integer(1)]).
↪weight_lattice(extended=true).fundamental_weight(Integer(0)))
>>> [V.s((Integer(0),Integer(0),Integer(0)),i) for i in V._index_set]
[(1, 0, 0), (0, 0, 0), (0, 0, 0)]
```

string (*max_weight*, *depth=12*)

Return the list of multiplicities $m(\Lambda - k\delta)$ in `self`, where Λ is `max_weight` and k runs from 0 to `depth`.

INPUT:

- `max_weight` – a dominant maximal weight
- `depth` – (default: 12) the maximum value of k

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: V.string(2*Lambda[0])
[1, 2, 8, 20, 52, 116, 256, 522, 1045, 1996, 3736, 6780]
sage: V.string(Lambda[1] + Lambda[2])
[0, 1, 4, 12, 32, 77, 172, 365, 740, 1445, 2736, 5041]
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(2)*Lambda[Integer(0)])
>>> V.string(Integer(2)*Lambda[Integer(0)])
[1, 2, 8, 20, 52, 116, 256, 522, 1045, 1996, 3736, 6780]
>>> V.string(Lambda[Integer(1)] + Lambda[Integer(2)])
[0, 1, 4, 12, 32, 77, 172, 365, 740, 1445, 2736, 5041]
```

strings(*depth=12*)

Return the set of dominant maximal weights of `self`, together with the string coefficients for each.

OPTIONAL:

- `depth` – (default: 12) a parameter indicating how far to push computations

EXAMPLES:

```
sage: Lambda = RootSystem(['A',1,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: S = V.strings(depth=25)
sage: for k in S:
....:     print("{}: {}".format(k, ' '.join(str(x) for x in S[k])))
2*Lambda[0]: 1 1 3 5 10 16 28 43 70 105 161 236 350 501 722 1016 1431 1981␣
↪2741 3740 5096 6868 9233 12306 16357
2*Lambda[1] - delta: 1 2 4 7 13 21 35 55 86 130 196 287 420 602 858 1206 1687␣
↪2331 3206 4368 5922 7967 10670 14193 18803
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(1),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(2)*Lambda[Integer(0)])
>>> S = V.strings(depth=Integer(25))
>>> for k in S:
...     print("{}: {}".format(k, ' '.join(str(x) for x in S[k])))
2*Lambda[0]: 1 1 3 5 10 16 28 43 70 105 161 236 350 501 722 1016 1431 1981␣
↪2741 3740 5096 6868 9233 12306 16357
2*Lambda[1] - delta: 1 2 4 7 13 21 35 55 86 130 196 287 420 602 858 1206 1687␣
↪2331 3206 4368 5922 7967 10670 14193 18803
```

to_dominant(*n*)

Return the dominant weight in `self` equivalent to `n` under the affine Weyl group.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(3*Lambda[0])
sage: n = V.to_dominant((13,11,7)); n
(4, 3, 3)
sage: V.to_weight(n)
Lambda[0] + Lambda[1] + Lambda[2] - 4*delta
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(3)*Lambda[Integer(0)])
>>> n = V.to_dominant((Integer(13),Integer(11),Integer(7))); n
(4, 3, 3)
>>> V.to_weight(n)
Lambda[0] + Lambda[1] + Lambda[2] - 4*delta
```

to_weight(*n*)

Return the weight associated to the tuple `n` in `self`.

If `n` is the tuple $(n_1, n_2, \ldots)$, then the associated weight is $\Lambda - \sum_i n_i \alpha_i$, where Λ is the weight of the representation.

INPUT:

- `n` – tuple representing a weight

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).
↪fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[2])
sage: V.to_weight((1,0,0))
-2*Lambda[0] + Lambda[1] + 3*Lambda[2] - delta
```

```
>>> from sage.all import *
>>> Lambda = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weights()
>>> V = IntegrableRepresentation(Integer(2)*Lambda[Integer(2)])
>>> V.to_weight((Integer(1),Integer(0),Integer(0)))
-2*Lambda[0] + Lambda[1] + 3*Lambda[2] - delta
```

weight_lattice()

Return the weight lattice associated to `self`.

EXAMPLES:

```
sage: V=IntegrableRepresentation(RootSystem(['E',6,1]).weight_
↪lattice(extended=true).fundamental_weight(0))
sage: V.weight_lattice()
Extended weight lattice of the Root system of type ['E', 6, 1]
```

```
>>> from sage.all import *
>>> V=IntegrableRepresentation(RootSystem(['E',Integer(6),Integer(1)]).weight_
↪lattice(extended=true).fundamental_weight(Integer(0)))
>>> V.weight_lattice()
Extended weight lattice of the Root system of type ['E', 6, 1]
```

5.1.233 Nonsymmetric Macdonald polynomials

AUTHORS:

- Anne Schilling and Nicolas M. Thiéry (2013): initial version

ACKNOWLEDGEMENTS:

The initial version of this code (together with *root_lattice_realization_algebras.Algebras* and *hecke_algebra_representation.HeckeAlgebraRepresentation*) was written by Anne Schilling and Nicolas M. Thiery during the ICERM Semester Program on “Automorphic Forms, Combinatorial Representation Theory and Multiple Dirichlet Series” (January 28, 2013 - May 3, 2013) with the help of Dan Bump, Ben Brubaker, Bogdan Ion, Dan Orr, Arun Ram, Siddhartha Sahi, and Mark Shimozono. Special thanks go to Bogdan Ion and Mark Shimozono for their patient explanations and hand computations to check the code.

class sage.combinat.root_system.non_symmetric_macdonald_polynomials.**NonSymmetricMacdonaldPolynomials**

Bases: *CherednikOperatorsEigenvectors*

Nonsymmetric Macdonald polynomials.

INPUT:

- KL – an affine Cartan type or the group algebra of a realization of the affine weight lattice
- q, q1, q2 – parameters in the base ring of the group algebra (default: q, q1, q2)
- normalized – boolean (default: True); whether to normalize the result to have leading coefficient 1

This implementation covers all reduced affine root systems. The polynomials are constructed recursively by the application of intertwining operators.

Todo

- Non-reduced case (Koornwinder polynomials).
- Non-equal parameters for the affine Hecke algebra.
- Choice of convention (dominant/anti-dominant, …).
- More uniform implementation of the $T_0^\vee$ operator.
- Optimizations, in particular in the calculation of the eigenvalues for the recursion.

EXAMPLES:

We construct the family of nonsymmetric Macdonald polynomials in three variables in type A:

```
sage: E = NonSymmetricMacdonaldPolynomials(["A",2,1])
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials(["A",Integer(2),Integer(1)])
```

They are constructed as elements of the group algebra of the classical weight lattice L_0 (or one of its realizations, such as the ambient space, which is used here) and indexed by elements of L_0:

```
sage: L0 = E.keys(); L0
Ambient space of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> L0 = E.keys(); L0
Ambient space of the Root system of type ['A', 2]
```

Here is the nonsymmetric Macdonald polynomial with leading term $[2, 0, 1]$:

```
sage: E[L0([2,0,1])]                                                    #␣
↪needs sage.libs.gap
```

(continues on next page)

(continued from previous page)

```
-((q*q1+q*q2)/(-q*q1-q2))*B[(1, 1, 1)]
- ((q1+q2)/(-q*q1-q2))*B[(2, 1, 0)] + B[(2, 0, 1)]
```

```
>>> from sage.all import *
>>> E[L0([Integer(2),Integer(0),Integer(1)])]                              ␣
↪                        # needs sage.libs.gap
-((q*q1+q*q2)/(-q*q1-q2))*B[(1, 1, 1)]
- ((q1+q2)/(-q*q1-q2))*B[(2, 1, 0)] + B[(2, 0, 1)]
```

It can be seen as a polynomial (or in general a Laurent polynomial) by interpreting each term as an exponent vector. The parameter q is the exponential of the null (co)root, whereas q_1 and q_2 are the two eigenvalues of each generator T_i of the affine Hecke algebra (see the background section for details).

By setting $q_1 = t$, $q_2 = -1$ and using the *root_lattice_realization_algebras.Algebras.ElementMethods.expand()* method, we recover the nonsymmetric Macdonald polynomial as computed by [HHL06]'s combinatorial formula:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: E = NonSymmetricMacdonaldPolynomials(["A",2,1], q=q, q1=t, q2=-1)
sage: vars = K['x0,x1,x2'].gens()
sage: E[L0([2,0,1])].expand(vars)                                        #␣
↪needs sage.libs.gap
(t - 1)/(q*t - 1)*x0^2*x1 + x0^2*x2 + (q*t - q)/(q*t - 1)*x0*x1*x2

sage: from sage.combinat.sf.ns_macdonald import E                        #␣
↪needs sage.combinat
sage: E([2,0,1])                                                         #␣
↪needs sage.combinat sage.groups
(t - 1)/(q*t - 1)*x0^2*x1 + x0^2*x2 + (q*t - q)/(q*t - 1)*x0*x1*x2
```

```
>>> from sage.all import *
>>> K = QQ['q,t'].fraction_field()
>>> q,t = K.gens()
>>> E = NonSymmetricMacdonaldPolynomials(["A",Integer(2),Integer(1)], q=q, q1=t,␣
↪q2=-Integer(1))
>>> vars = K['x0,x1,x2'].gens()
>>> E[L0([Integer(2),Integer(0),Integer(1)])].expand(vars)                 ␣
↪                        # needs sage.libs.gap
(t - 1)/(q*t - 1)*x0^2*x1 + x0^2*x2 + (q*t - q)/(q*t - 1)*x0*x1*x2

>>> from sage.combinat.sf.ns_macdonald import E                          #␣
↪needs sage.combinat
>>> E([Integer(2),Integer(0),Integer(1)])                                  ␣
↪                        # needs sage.combinat sage.groups
(t - 1)/(q*t - 1)*x0^2*x1 + x0^2*x2 + (q*t - q)/(q*t - 1)*x0*x1*x2
```

Here is a type $G_2^{(1)}$ nonsymmetric Macdonald polynomial:

```
sage: E = NonSymmetricMacdonaldPolynomials(["G",2,1])
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
```

(continues on next page)

(continued from previous page)

```
sage: E[omega[2] - omega[1]]
-((q*q1^3*q2+q*q1^2*q2^2)/(q*q1^4-q2^4))*B[(0, 0, 0)] + B[(1, -1, 0)]
- ((q1*q2^3+q2^4)/(q*q1^4-q2^4))*B[(1, 0, -1)]
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials(["G",Integer(2),Integer(1)])
>>> L0 = E.keys()
>>> omega = L0.fundamental_weights()
>>> E[omega[Integer(2)] - omega[Integer(1)]]
-((q*q1^3*q2+q*q1^2*q2^2)/(q*q1^4-q2^4))*B[(0, 0, 0)] + B[(1, -1, 0)]
- ((q1*q2^3+q2^4)/(q*q1^4-q2^4))*B[(1, 0, -1)]
```

Many more examples are given after the background section.

See also

- *sage.combinat.sf.ns_macdonald.E()*
- *SymmetricFunctions.macdonald()*

Background

The polynomial module

The nonsymmetric Macdonald polynomials are a distinguished basis of the "polynomial" module of the affine Hecke algebra. Given:

```
- a ground ring `K`, which contains the input parameters `q, q_1, q_2`
- an affine root system, specified by a Cartan type `C`
- a realization `L` of the weight lattice of type `C`
```

the polynomial module is the group algebra $K[L_0]$ of the classical weight lattice L_0 with coefficients in K. It is isomorphic to the Laurent polynomial ring over K generated by the formal exponentials of any basis of L_0.

In our running example L is the ambient space of type $C_2^{(1)}$:

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q, q1, q2 = K.gens()
sage: C = CartanType(["C",2,1])
sage: L = RootSystem(C).ambient_space(); L
Ambient space of the Root system of type ['C', 2, 1]

sage: L.simple_roots()
Finite family {0: -2*e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
sage: omega = L.fundamental_weights(); omega
Finite family {0: e['deltacheck'],
               1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + e['deltacheck']}

sage: L0 = L.classical(); L0
Ambient space of the Root system of type ['C', 2]
sage: KL0 = L0.algebra(K); KL0
```

(continues on next page)

(continued from previous page)

```
Algebra of the Ambient space of the Root system of type ['C', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational
↪Field
```

```
>>> from sage.all import *
>>> K = QQ['q,q1,q2'].fraction_field()
>>> q, q1, q2 = K.gens()
>>> C = CartanType(["C",Integer(2),Integer(1)])
>>> L = RootSystem(C).ambient_space(); L
Ambient space of the Root system of type ['C', 2, 1]

>>> L.simple_roots()
Finite family {0: -2*e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
>>> omega = L.fundamental_weights(); omega
Finite family {0: e['deltacheck'],
               1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + e['deltacheck']}

>>> L0 = L.classical(); L0
Ambient space of the Root system of type ['C', 2]
>>> KL0 = L0.algebra(K); KL0
Algebra of the Ambient space of the Root system of type ['C', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational
↪Field
```

Affine Hecke algebra

The affine Hecke algebra is generated by elements T_i for `i` in the set of affine Dynkin nodes. They satisfy the same braid relations as the simple reflections s_i of the affine Weyl group. The T_i satisfy the quadratic relation

$$(T_i - q_1) \circ (T_i - q_2) = 0,$$

where q_1 and q_2 are the input parameters. Some of the representation theory requires that q_1 and q_2 satisfy additional relations; typically one uses the specializations $q_1 = u$ and $q_2 = -1/u$ or $q_1 = t$ and $q_2 = -1$). This can be achieved by constructing an appropriate field and passing q_1 and q_2 appropriately; see the examples. In principle, the parameter(s) could further depend on `i`; this is not yet implemented but the code has been designed in such a way that this feature is easy to add.

Demazure-Lusztig operators

The `i`-th Demazure-Lusztig operator is an operator on $K[L]$ which interpolates between the reflection s_i and the Demazure operator π_i (see `root_lattice_realization.RootLatticeRealization.Algebras.ParentMethods.demazure_lusztig_operators()`).:

```
sage: KL = L.algebra(K); KL
Algebra of the Ambient space of the Root system of type ['C', 2, 1]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational
↪Field
sage: T = KL.demazure_lusztig_operators(q1, q2)
sage: x = KL.monomial(omega[1]); x
B[e[0] + e['deltacheck']]
sage: T[2](x)
```

(continues on next page)

(continued from previous page)

```
q1*B[e[0] + e['deltacheck']]
sage: T[1](x)
(q1+q2)*B[e[0] + e['deltacheck']] + q1*B[e[1] + e['deltacheck']]
sage: T[0](x)
q1*B[e[0] + e['deltacheck']]
```

```
>>> from sage.all import *
>>> KL = L.algebra(K); KL
Algebra of the Ambient space of the Root system of type ['C', 2, 1]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational␣
↪Field
>>> T = KL.demazure_lusztig_operators(q1, q2)
>>> x = KL.monomial(omega[Integer(1)]); x
B[e[0] + e['deltacheck']]
>>> T[Integer(2)](x)
q1*B[e[0] + e['deltacheck']]
>>> T[Integer(1)](x)
(q1+q2)*B[e[0] + e['deltacheck']] + q1*B[e[1] + e['deltacheck']]
>>> T[Integer(0)](x)
q1*B[e[0] + e['deltacheck']]
```

The affine Hecke algebra acts on $K[L]$ by letting the generators T_i act by the Demazure-Lusztig operators. The class *sage.combinat.root_system.hecke_algebra_representation.HeckeAlgebraRepresentation* implements some simple generic features for representations of affine Hecke algebras defined by the action of their T-generators.:

```
sage: T
A representation of the (q1, q2)-Hecke algebra of type ['C', 2, 1] on
 Algebra of the Ambient space of the Root system of type ['C', 2, 1] over
 Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational Field
sage: type(T)
<class 'sage.combinat.root_system.hecke_algebra_representation.
↪HeckeAlgebraRepresentation'>
sage: T._test_relations()                    # long time (1.3s)
```

```
>>> from sage.all import *
>>> T
A representation of the (q1, q2)-Hecke algebra of type ['C', 2, 1] on
 Algebra of the Ambient space of the Root system of type ['C', 2, 1] over
 Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational Field
>>> type(T)
<class 'sage.combinat.root_system.hecke_algebra_representation.
↪HeckeAlgebraRepresentation'>
>>> T._test_relations()                    # long time (1.3s)
```

Here we construct the operator $q_1 T_2^{-1} \circ T_1^{-1} T_0$ from a signed reduced word:

```
sage: T.Tw([0,1,2],[1,-1,-1], q1^2)
Generic endomorphism of Algebra of the Ambient space of the Root system of type [
↪'C', 2, 1]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational␣
↪Field
```

```
>>> from sage.all import *
>>> T.Tw([Integer(0),Integer(1),Integer(2)],[Integer(1),-Integer(1),-Integer(1)],
↪q1**Integer(2))
Generic endomorphism of Algebra of the Ambient space of the Root system of type [
↪'C', 2, 1]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational
↪Field
```

(note the reversal of the word). Inverses are computed using the quadratic relation.

Cherednik operators

The affine Hecke algebra contains elements Y_λ indexed by the coroot lattice. Their action on $K[L]$ is implemented in Sage:

```
sage: Y = T.Y(); Y
Lazy family (...)_{i in Coroot lattice of the Root system of type ['C', 2, 1]}
sage: alphacheck = Y.keys().simple_roots()
sage: Y1 = Y[alphacheck[1]]
sage: Y1(x)
((q1^2+2*q1*q2+q2^2)/(-q1*q2))*B[e[0] + e['deltacheck']]
- ((q1^2+2*q1*q2+q2^2)/(-q2^2))*B[-e[1] + e['deltacheck']]
- ((q1^2+q1*q2)/(-q2^2))*B[2*e[0] - e[1] - e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - 2*e[1] - e['delta'] + e['deltacheck']]
+ ((q1+q2)/(-q2))*B[e[1] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[-e[1] - e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[2*e[0] - e[1] - 2*e['delta'] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[-e[0] - e['delta'] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[e[0] - 2*e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[3*e[0] - 3*e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[-e[0] - 2*e[1] - e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - 2*e[1] - 2*e['delta'] + e['deltacheck']]
+ (q1^3/(-q2^3))*B[3*e[0] - 2*e[1] - 3*e['delta'] + e['deltacheck']]
```

```
>>> from sage.all import *
>>> Y = T.Y(); Y
Lazy family (...)_{i in Coroot lattice of the Root system of type ['C', 2, 1]}
>>> alphacheck = Y.keys().simple_roots()
>>> Y1 = Y[alphacheck[Integer(1)]]
>>> Y1(x)
((q1^2+2*q1*q2+q2^2)/(-q1*q2))*B[e[0] + e['deltacheck']]
- ((q1^2+2*q1*q2+q2^2)/(-q2^2))*B[-e[1] + e['deltacheck']]
- ((q1^2+q1*q2)/(-q2^2))*B[2*e[0] - e[1] - e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - 2*e[1] - e['delta'] + e['deltacheck']]
+ ((q1+q2)/(-q2))*B[e[1] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[-e[1] - e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[2*e[0] - e[1] - 2*e['delta'] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[-e[0] - e['delta'] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[e[0] - 2*e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[3*e[0] - 3*e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[-e[0] - 2*e[1] - e['delta'] + e['deltacheck']]
```

(continues on next page)

(continued from previous page)

```
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - 2*e[1] - 2*e['delta'] + e['deltacheck']]
+ (q1^3/(-q2^3))*B[3*e[0] - 2*e[1] - 3*e['delta'] + e['deltacheck']]
```

The Cherednik operators span a Laurent polynomial ring inside the affine Hecke algebra; namely $\lambda \mapsto Y_\lambda$ is a group isomorphism from the classical root lattice (viewed additively) to the affine Hecke algebra (viewed multiplicatively). In practice, Y_λ is constructed by computing combinatorially its signed reduced word (and an overall scalar factor) using the periodic orientation of the alcove model in the coweight lattice (see `hecke_algebra_representation.HeckeAlgebraRepresentation.Y_lambdacheck()`):

```
sage: Lcheck = L.root_system.coweight_lattice()
sage: w = Lcheck.reduced_word_of_translation(Lcheck(alphacheck[1])); w
[0, 2, 1, 0, 2, 1]
sage: Lcheck.signs_of_alcovewalk(w)
[1, -1, 1, -1, 1, 1]
```

```
>>> from sage.all import *
>>> Lcheck = L.root_system.coweight_lattice()
>>> w = Lcheck.reduced_word_of_translation(Lcheck(alphacheck[Integer(1)])); w
[0, 2, 1, 0, 2, 1]
>>> Lcheck.signs_of_alcovewalk(w)
[1, -1, 1, -1, 1, 1]
```

Level zero representation of the affine Hecke algebra

The action of the affine Hecke algebra on $K[L]$ induces an action on $K[L_0]$: the action of T_i on X^λ for λ a classical weight in L_0 is obtained by embedding the weight at level zero in the affine weight lattice (see `weight_lattice_realizations.WeightLatticeRealizations.ParentMethods.embed_at_level()`) applying the Demazure-Lusztig operator there, and projecting from $K[L] \rightarrow K[L_0]$ mapping the exponential of δ to q (see `root_lattice_realization_algebras.Algebras.ParentMethods.q_project()`). This is implemented in `root_lattice_realization_algebras.Algebras.ParentMethods.demazure_lusztig_operators_on_classical()`:

```
sage: T = KL.demazure_lusztig_operators_on_classical(q, q1,q2)
sage: omega = L0.fundamental_weights()
sage: x = KL0.monomial(omega[1])
sage: T[0](x)
-q*q2*B[(-1, 0)]
```

```
>>> from sage.all import *
>>> T = KL.demazure_lusztig_operators_on_classical(q, q1,q2)
>>> omega = L0.fundamental_weights()
>>> x = KL0.monomial(omega[Integer(1)])
>>> T[Integer(0)](x)
-q*q2*B[(-1, 0)]
```

For classical nodes these are the usual Demazure-Lusztig operators:

```
sage: T[1](x)
(q1+q2)*B[(1, 0)] + q1*B[(0, 1)]
```

```
>>> from sage.all import *
>>> T[Integer(1)](x)
(q1+q2)*B[(1, 0)] + q1*B[(0, 1)]
```

Nonsymmetric Macdonald polynomials

We can now finally define the nonsymmetric Macdonald polynomials. Because the Cherednik operators commute (and there is no radical), they can be simultaneously diagonalized; namely, $K[L_0]$ admits a K-basis of joint eigenvectors for the Y_λ. For $\mu \in L_0$, the nonsymmetric Macdonald polynomial E_μ is the unique eigenvector of the family of Cherednik operators Y_λ having μ as leading term:

```
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, q1, q2); E
The family of the Macdonald polynomials of type ['C', 2, 1]
 with parameters q, q1, q2
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, q1, q2); E
The family of the Macdonald polynomials of type ['C', 2, 1]
 with parameters q, q1, q2
```

Or for short:

```
sage: E = NonSymmetricMacdonaldPolynomials(C)
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials(C)
```

Recursive construction of the nonsymmetric Macdonald polynomials

The generators T_i of the affine Hecke algebra almost skew commute with the Cherednik operators. More precisely, one can deform them into the so-called intertwining operators:

$$\tau_i = T_i - (q_1 + q_2)\frac{Y_i^{a-1}}{1 - Y_i^a} .$$

(where $a = 1$ except for $i = 0$ in type BC where $a = a_0 = 2$) which satisfy the following skew commutation relations:

$$\tau_i Y_\lambda = \tau_i Y_{s_i\lambda} .$$

If $s_i\mu \neq \mu$, applying τ_i on an eigenvector E_μ produces a new eigenvector (essentially $E_{s_i\mu}$) with a distinct eigenvalue. It follows that the eigenvectors indexed by an affine Weyl orbit of weights, may be recursively computed from a single weight in the orbit.

In the case at hand, there is a little complication: namely, the simple reflections s_i acting at level 0 do not act transitively on classical weights; in fact the orbits for the classical Weyl group and for the affine Weyl group are the same. Thus, one can construct the nonsymmetric Macdonald polynomials for all weights from those for the classical dominant weights, but one is lacking a creation operator to construct the nonsymmetric Macdonald polynomials for dominant weights.

Twisted Demazure-Lusztig operators

To compensate for this, one needs to consider another affinization of the action of the classical Demazure-Lusztig operators $T_1, \ldots, T_n$, which gives rise to the double affine Hecke algebra. Following Cherednik, one adds another operator $T_0^\vee$ implemented in: *root_lattice_realization_algebras.Algebras.ParentMethods.T0_check_on_basis()*. See also: *root_lattice_realization_algebras.Algebras.ParentMethods.twisted_demazure_lusztig_operators()*.

Depending on the type (untwisted or not), this is a representation of the affine Hecke algebra for another affinization of the classical Cartan type. The corresponding action of the affine Weyl group – which is used to compute the recursion on μ – occurs in the corresponding weight lattice realization:

```
sage: E.L()
Ambient space of the Root system of type ['C', 2, 1]
sage: E.L_prime()
Coambient space of the Root system of type ['B', 2, 1]
sage: E.L_prime().classical()
Ambient space of the Root system of type ['C', 2]
```

```
>>> from sage.all import *
>>> E.L()
Ambient space of the Root system of type ['C', 2, 1]
>>> E.L_prime()
Coambient space of the Root system of type ['B', 2, 1]
>>> E.L_prime().classical()
Ambient space of the Root system of type ['C', 2]
```

See *L_prime()* and *cartan_type.CartanType_affine.other_affinization()*.

REFERENCES:

More examples

We show how to create the nonsymmetric Macdonald polynomials in two different ways and check that they are the same:

```
sage: K = QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: E = NonSymmetricMacdonaldPolynomials(['D',3,1], q, u, -1/u)
sage: omega = E.keys().fundamental_weights()
sage: E[omega[1]+omega[3]]
-((q*u^2-q)/(-q*u^4+1))*B[(1/2, -1/2, 1/2)]
- ((q*u^2-q)/(-q*u^4+1))*B[(1/2, 1/2, -1/2)] + B[(3/2, 1/2, 1/2)]

sage: KL = RootSystem(["D",3,1]).ambient_space().algebra(K)
sage: P = NonSymmetricMacdonaldPolynomials(KL, q, u, -1/u)
sage: E[omega[1]+omega[3]] == P[omega[1]+omega[3]]
True
sage: E[E.keys()((0,1,-1))]
-((q*u^2-q)/(-q*u^2+1))*B[(0, 0, 0)] - ((u^2-1)/(-q*u^2+1))*B[(1, 1, 0)]
- ((u^2-1)/(-q*u^2+1))*B[(1, 0, -1)] + B[(0, 1, -1)]
```

```
>>> from sage.all import *
>>> K = QQ['q,u'].fraction_field()
>>> q, u = K.gens()
>>> E = NonSymmetricMacdonaldPolynomials(['D',Integer(3),Integer(1)], q, u, -
↪Integer(1)/u)
>>> omega = E.keys().fundamental_weights()
```

(continues on next page)

(continued from previous page)

```
>>> E[omega[Integer(1)]+omega[Integer(3)]]
-((q*u^2-q)/(-q*u^4+1))*B[(1/2, -1/2, 1/2)]
- ((q*u^2-q)/(-q*u^4+1))*B[(1/2, 1/2, -1/2)] + B[(3/2, 1/2, 1/2)]

>>> KL = RootSystem(["D",Integer(3),Integer(1)]).ambient_space().algebra(K)
>>> P = NonSymmetricMacdonaldPolynomials(KL, q, u, -Integer(1)/u)
>>> E[omega[Integer(1)]+omega[Integer(3)]] ==␣
↪P[omega[Integer(1)]+omega[Integer(3)]]
True
>>> E[E.keys()((Integer(0),Integer(1),-Integer(1)))]
-((q*u^2-q)/(-q*u^2+1))*B[(0, 0, 0)] - ((u^2-1)/(-q*u^2+1))*B[(1, 1, 0)]
- ((u^2-1)/(-q*u^2+1))*B[(1, 0, -1)] + B[(0, 1, -1)]
```

In type A, there is also a combinatorial implementation of the nonsymmetric Macdonald polynomials in terms of augmented diagram fillings as in [HHL06]. See *sage.combinat.sf.ns_macdonald.E()*. First we check that these polynomials are indeed eigenvectors of the Cherednik operators:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: q1 = t; q2 = -1
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: KL0 = KL.classical()
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: omega = E.keys().fundamental_weights()
sage: w = omega[1]

sage: # needs sage.combinat sage.groups
sage: import sage.combinat.sf.ns_macdonald as NS
sage: p = NS.E([1,0,0]); p
x0
sage: pp = KL0.from_polynomial(p)
sage: E.eigenvalues(KL0.from_polynomial(p))
[t, (-1)/(-q*t^2), t]

sage: def eig(l): return E.eigenvalues(KL0.from_polynomial(NS.E(l)))

sage: # needs sage.combinat sage.groups
sage: eig([1,0,0])
[t, (-1)/(-q*t^2), t]
sage: eig([2,0,0])
[q*t, (-1)/(-q^2*t^2), t]
sage: eig([3,0,0])
[q^2*t, (-1)/(-q^3*t^2), t]
sage: eig([2,0,4])
[(-1)/(-q^3*t), 1/(q^2*t), q^4*t^2]
```

```
>>> from sage.all import *
>>> K = QQ['q,t'].fraction_field()
>>> q,t = K.gens()
>>> q1 = t; q2 = -Integer(1)
>>> KL = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().algebra(K)
>>> KL0 = KL.classical()
```

(continues on next page)

(continued from previous page)

```
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
>>> omega = E.keys().fundamental_weights()
>>> w = omega[Integer(1)]

>>> # needs sage.combinat sage.groups
>>> import sage.combinat.sf.ns_macdonald as NS
>>> p = NS.E([Integer(1),Integer(0),Integer(0)]); p
x0
>>> pp = KL0.from_polynomial(p)
>>> E.eigenvalues(KL0.from_polynomial(p))
[t, (-1)/(-q*t^2), t]

>>> def eig(l): return E.eigenvalues(KL0.from_polynomial(NS.E(l)))

>>> # needs sage.combinat sage.groups
>>> eig([Integer(1),Integer(0),Integer(0)])
[t, (-1)/(-q*t^2), t]
>>> eig([Integer(2),Integer(0),Integer(0)])
[q*t, (-1)/(-q^2*t^2), t]
>>> eig([Integer(3),Integer(0),Integer(0)])
[q^2*t, (-1)/(-q^3*t^2), t]
>>> eig([Integer(2),Integer(0),Integer(4)])
[(-1)/(-q^3*t), 1/(q^2*t), q^4*t^2]
```

Next we check explicitly that they agree with the current implementation:

```
sage: K = QQ['q','t'].fraction_field()
sage: q,t = K.gens()
sage: KL = RootSystem(["A",1,1]).ambient_lattice().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: L0 = E.keys()
sage: KL0 = KL.classical()
sage: P = K['x0,x1']
sage: def EE(weight): return E[L0(weight)].expand(P.gens())

sage: # needs sage.combinat
sage: import sage.combinat.sf.ns_macdonald as NS
sage: EE([0,0])
1
sage: NS.E([0,0])                                                    #␣
↪needs sage.groups
1
sage: EE([1,0])
x0
sage: NS.E([1,0])                                                    #␣
↪needs sage.groups
x0
sage: EE([0,1])
(t - 1)/(q*t - 1)*x0 + x1
sage: NS.E([0,1])                                                    #␣
↪needs sage.groups
(t - 1)/(q*t - 1)*x0 + x1
```

(continues on next page)

(continued from previous page)

```
sage: EE([2,0])
x0^2 + (q*t - q)/(q*t - 1)*x0*x1
sage: NS.E([2,0])                                                    #␣
↪needs sage.groups
x0^2 + (q*t - q)/(q*t - 1)*x0*x1
```

```
>>> from sage.all import *
>>> K = QQ['q','t'].fraction_field()
>>> q,t = K.gens()
>>> KL = RootSystem(["A",Integer(1),Integer(1)]).ambient_lattice().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, t, -Integer(1))
>>> L0 = E.keys()
>>> KL0 = KL.classical()
>>> P = K['x0,x1']
>>> def EE(weight): return E[L0(weight)].expand(P.gens())

>>> # needs sage.combinat
>>> import sage.combinat.sf.ns_macdonald as NS
>>> EE([Integer(0),Integer(0)])
1
>>> NS.E([Integer(0),Integer(0)])                                     ␣
↪               # needs sage.groups
1
>>> EE([Integer(1),Integer(0)])
x0
>>> NS.E([Integer(1),Integer(0)])                                     ␣
↪               # needs sage.groups
x0
>>> EE([Integer(0),Integer(1)])
(t - 1)/(q*t - 1)*x0 + x1
>>> NS.E([Integer(0),Integer(1)])                                     ␣
↪               # needs sage.groups
(t - 1)/(q*t - 1)*x0 + x1
>>> EE([Integer(2),Integer(0)])
x0^2 + (q*t - q)/(q*t - 1)*x0*x1
>>> NS.E([Integer(2),Integer(0)])                                     ␣
↪               # needs sage.groups
x0^2 + (q*t - q)/(q*t - 1)*x0*x1
```

The same, directly in the ambient lattice with several shifts:

```
sage: E[L0([2,0])]                                                   #␣
↪needs sage.combinat
-((q*t-q)/(-q*t+1))*B[(1, 1)] + B[(2, 0)]
sage: E[L0([1,-1])]                                                  #␣
↪needs sage.combinat
-((q*t-q)/(-q*t+1))*B[(0, 0)] + B[(1, -1)]
sage: E[L0([0,-2])]                                                  #␣
↪needs sage.combinat
-((q*t-q)/(-q*t+1))*B[(-1, -1)] + B[(0, -2)]
```

```
>>> from sage.all import *
>>> E[L0([Integer(2),Integer(0)])]                                              ␣
↪               # needs sage.combinat
-((q*t-q)/(-q*t+1))*B[(1, 1)] + B[(2, 0)]
>>> E[L0([Integer(1),-Integer(1)])]                                             ␣
↪               # needs sage.combinat
-((q*t-q)/(-q*t+1))*B[(0, 0)] + B[(1, -1)]
>>> E[L0([Integer(0),-Integer(2)])]                                             ␣
↪               # needs sage.combinat
-((q*t-q)/(-q*t+1))*B[(-1, -1)] + B[(0, -2)]
```

Systematic checks with Sage's implementation of [HHL06]:

```
sage: assert all(EE([x,y]) == NS.E([x,y])                                       #␣
↪needs sage.combinat
....:            for d in range(5) for x,y in IntegerVectors(d,2))
```

```
>>> from sage.all import *
>>> assert all(EE([x,y]) == NS.E([x,y])                                         #␣
↪needs sage.combinat
...            for d in range(Integer(5)) for x,y in IntegerVectors(d,Integer(2)))
```

With the current implementation, we can compute nonsymmetric Macdonald polynomials for any type, for example for type $E_6^{(1)}$:

```
sage: K = QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: KL = RootSystem(["E",6,1]).weight_space(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q,u,-1/u)
sage: L0 = E.keys()

sage: E[L0.fundamental_weight(1).weyl_action([2,4,3,2,1])]
-((u^2-1)/(-q*u^16+1))*B[-Lambda[1] + Lambda[3]]
- ((u^2-1)/(-q*u^16+1))*B[Lambda[1]] + B[-Lambda[2] + Lambda[5]]
- ((u^2-1)/(-q*u^16+1))*B[Lambda[2]
- Lambda[4] + Lambda[5]] - ((u^2-1)/(-q*u^16+1))*B[-Lambda[3] + Lambda[4]]

sage: E[L0.fundamental_weight(2).weyl_action([2,5,3,4,2])]      # long time (6s)
-((q^2*u^20-q^2*u^18-q*u^2+q)/(-q^2*u^32+2*q*u^16-1))*B[0]
+ B[Lambda[1] - Lambda[3] + Lambda[4] - Lambda[5] + Lambda[6]]
- ((u^2-1)/(-q*u^16+1))*B[Lambda[1] - Lambda[3] + Lambda[5]]
- ((q*u^20-q*u^18-u^2+1)/(-q^2*u^32+2*q*u^16-1))*B[-Lambda[2] + Lambda[4]]
- ((q*u^20-q*u^18-u^2+1)/(-q^2*u^32+2*q*u^16-1))*B[Lambda[2]]
+ ((u^4-2*u^2+1)/(q^2*u^32-2*q*u^16+1))*B[Lambda[3] - Lambda[4] + Lambda[5]]
- ((u^2-1)/(-q*u^16+1))*B[Lambda[3] - Lambda[5] + Lambda[6]]
sage: E[L0.fundamental_weight(1)+L0.fundamental_weight(6)]      # long time (13s)
((q^2*u^10-q^2*u^8-q^2*u^2+q^2)/(q^2*u^26-q*u^16-q*u^10+1))*B[0]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[1] - Lambda[2] + Lambda[6]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[1] + Lambda[2] - Lambda[4] + Lambda[6]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[1] - Lambda[3] + Lambda[4] - Lambda[5] +␣
↪Lambda[6]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[1] - Lambda[3] + Lambda[5]] + B[Lambda[1] +␣
↪Lambda[6]]
```

(continues on next page)

(continued from previous page)

```
- ((q*u^2-q)/(-q*u^10+1))*B[-Lambda[2] + Lambda[4]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[2]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[3] - Lambda[4] + Lambda[5]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[3] - Lambda[5] + Lambda[6]]
```

```
>>> from sage.all import *
>>> K = QQ['q,u'].fraction_field()
>>> q, u = K.gens()
>>> KL = RootSystem(["E",Integer(6),Integer(1)]).weight_space(extended=True).
↪algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q,u,-Integer(1)/u)
>>> L0 = E.keys()

>>> E[L0.fundamental_weight(Integer(1)).weyl_action([Integer(2),Integer(4),
↪Integer(3),Integer(2),Integer(1)])]
-((u^2-1)/(-q*u^16+1))*B[-Lambda[1] + Lambda[3]]
- ((u^2-1)/(-q*u^16+1))*B[Lambda[1]] + B[-Lambda[2] + Lambda[5]]
- ((u^2-1)/(-q*u^16+1))*B[Lambda[2]
- Lambda[4] + Lambda[5]] - ((u^2-1)/(-q*u^16+1))*B[-Lambda[3] + Lambda[4]]

>>> E[L0.fundamental_weight(Integer(2)).weyl_action([Integer(2),Integer(5),
↪Integer(3),Integer(4),Integer(2)])]      # long time (6s)
-((q^2*u^20-q^2*u^18-q*u^2+q)/(-q^2*u^32+2*q*u^16-1))*B[0]
+ B[Lambda[1] - Lambda[3] + Lambda[4] - Lambda[5] + Lambda[6]]
- ((u^2-1)/(-q*u^16+1))*B[Lambda[1] - Lambda[3] + Lambda[5]]
- ((q*u^20-q*u^18-u^2+1)/(-q^2*u^32+2*q*u^16-1))*B[-Lambda[2] + Lambda[4]]
- ((q*u^20-q*u^18-u^2+1)/(-q^2*u^32+2*q*u^16-1))*B[Lambda[2]]
+ ((u^4-2*u^2+1)/(q^2*u^32-2*q*u^16+1))*B[Lambda[3] - Lambda[4] + Lambda[5]]
- ((u^2-1)/(-q*u^16+1))*B[Lambda[3] - Lambda[5] + Lambda[6]]
>>> E[L0.fundamental_weight(Integer(1))+L0.fundamental_weight(Integer(6))]      #␣
↪long time (13s)
((q^2*u^10-q^2*u^8-q^2*u^2+q^2)/(q^2*u^26-q*u^16-q*u^10+1))*B[0]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[1] - Lambda[2] + Lambda[6]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[1] + Lambda[2] - Lambda[4] + Lambda[6]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[1] - Lambda[3] + Lambda[4] - Lambda[5] +␣
↪Lambda[6]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[1] - Lambda[3] + Lambda[5]] + B[Lambda[1] +␣
↪Lambda[6]]
- ((q*u^2-q)/(-q*u^10+1))*B[-Lambda[2] + Lambda[4]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[2]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[3] - Lambda[4] + Lambda[5]]
- ((q*u^2-q)/(-q*u^10+1))*B[Lambda[3] - Lambda[5] + Lambda[6]]
```

We test various other types:

```
sage: K = QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: KL = RootSystem(["A",5,2]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, u, -1/u)
sage: L0 = E.keys()
sage: E[L0.fundamental_weight(2)]
-((q*u^2-q)/(-q*u^8+1))*B[(0, 0, 0)] + B[(1, 1, 0)]
```

(continues on next page)

(continued from previous page)

```
sage: E[L0((0,-1,1))]                          # long time (1.5s)
-((q^2*u^10-q^2*u^8+q*u^6-q*u^4-q*u^2-u^2+q+1)/(-q^3*u^12+q^2*u^8+q*u^4-1))*B[(0,␣
↪0, 0)]
- ((u^2-1)/(-q*u^4+1))*B[(1, -1, 0)]
+ ((u^6-u^4-u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 1, 0)]
+ ((u^4-2*u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 0, -1)]
+ ((q^2*u^12-q^2*u^10-u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 0, 1)]
+ B[(0, -1, 1)]
- ((u^2-1)/(-q^2*u^8+1))*B[(0, 1, -1)] - ((u^2-1)/(-q^2*u^8+1))*B[(0, 1, 1)]
sage: K = QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: KL = RootSystem(["E",6,2]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q,u,-1/u)
sage: L0 = E.keys()
sage: E[L0.fundamental_weight(4)]              # long time (5s)
-((q^3*u^20-q^3*u^18-q^2*u^2+q^2)/(-q^3*u^28+q^2*u^22+q*u^6-1))*B[(0, 0, 0, 0)]
- ((q*u^2-q)/(-q*u^6+1))*B[(1/2, 1/2, -1/2, -1/2)]
- ((q*u^2-q)/(-q*u^6+1))*B[(1/2, 1/2, -1/2, 1/2)]
- ((q*u^2-q)/(-q*u^6+1))*B[(1/2, 1/2, 1/2, -1/2)]
- ((q*u^2-q)/(-q*u^6+1))*B[(1/2, 1/2, 1/2, 1/2)]
+ ((q*u^2-q)/(q*u^6-1))*B[(1, 0, 0, 0)] + B[(1, 1, 0, 0)]
- ((q*u^2-q)/(-q*u^6+1))*B[(0, 1, 0, 0)]
sage: E[L0((1,-1,0,0))]                        # long time (23s)
((q^3*u^18-q^3*u^16+q*u^4-q^2*u^2-2*q*u^2+q^2+q)/(q^3*u^18-q^2*u^12-q*u^
↪6+1))*B[(0, 0, 0, 0)]
- ((q^3*u^18-q^3*u^16-q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, -1/2, -1/2, -
↪1/2)]
- ((q^3*u^18-q^3*u^16-q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, -1/2, -1/2,␣
↪1/2)]
+ ((q^3*u^18-q^3*u^16-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, -1/2, 1/2, -1/
↪2)]
+ ((q^3*u^18-q^3*u^16-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, -1/2, 1/2, 1/
↪2)]
+ ((q*u^8-q*u^6-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, 1/2, -1/2, -1/2)]
+ ((q*u^8-q*u^6-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, 1/2, -1/2, 1/2)]
- ((q*u^8-q*u^6-q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, 1/2, 1/2, -1/2)]
- ((q*u^8-q*u^6-q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, 1/2, 1/2, 1/2)]
- ((q^2*u^18-q^2*u^16+q*u^8-q*u^6-q*u^2-u^2+q+1)/(-q^3*u^18+q^2*u^12+q*u^6-
↪1))*B[(1, 0, 0, 0)]
+ B[(1, -1, 0, 0)] - ((u^2-1)/(-q^2*u^12+1))*B[(1, 1, 0, 0)]
- ((u^2-1)/(-q^2*u^12+1))*B[(1, 0, -1, 0)]
+ ((u^2-1)/(q^2*u^12-1))*B[(1, 0, 1, 0)]
- ((u^2-1)/(-q^2*u^12+1))*B[(1, 0, 0, -1)]
- ((u^2-1)/(-q^2*u^12+1))*B[(1, 0, 0, 1)]
- ((q*u^2-q)/(-q*u^6+1))*B[(0, -1, 0, 0)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 1, 0, 0)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, -1, 0)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 1, 0)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 0, -1)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 0, 1)]
```

```
>>> from sage.all import *
>>> K = QQ['q,u'].fraction_field()
>>> q, u = K.gens()
>>> KL = RootSystem(["A",Integer(5),Integer(2)]).ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, u, -Integer(1)/u)
>>> L0 = E.keys()
>>> E[L0.fundamental_weight(Integer(2))]
-((q*u^2-q)/(-q*u^8+1))*B[(0, 0, 0)] + B[(1, 1, 0)]
>>> E[L0((Integer(0),-Integer(1),Integer(1)))]                     # long time (1.
↪5s)
-((q^2*u^10-q^2*u^8+q*u^6-q*u^4-q*u^2-u^2+q+1)/(-q^3*u^12+q^2*u^8+q*u^4-1))*B[(0,␣
↪0, 0)]
- ((u^2-1)/(-q*u^4+1))*B[(1, -1, 0)]
+ ((u^6-u^4-u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 1, 0)]
+ ((u^4-2*u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 0, -1)]
+ ((q^2*u^12-q^2*u^10-u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 0, 1)]
+ B[(0, -1, 1)]
- ((u^2-1)/(-q^2*u^8+1))*B[(0, 1, -1)] - ((u^2-1)/(-q^2*u^8+1))*B[(0, 1, 1)]
>>> K = QQ['q,u'].fraction_field()
>>> q, u = K.gens()
>>> KL = RootSystem(["E",Integer(6),Integer(2)]).ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q,u,-Integer(1)/u)
>>> L0 = E.keys()
>>> E[L0.fundamental_weight(Integer(4))]       # long time (5s)
-((q^3*u^20-q^3*u^18-q^2*u^2+q^2)/(-q^3*u^28+q^2*u^22+q*u^6-1))*B[(0, 0, 0, 0)]
- ((q*u^2-q)/(-q*u^6+1))*B[(1/2, 1/2, -1/2, -1/2)]
- ((q*u^2-q)/(-q*u^6+1))*B[(1/2, 1/2, -1/2, 1/2)]
- ((q*u^2-q)/(-q*u^6+1))*B[(1/2, 1/2, 1/2, -1/2)]
- ((q*u^2-q)/(-q*u^6+1))*B[(1/2, 1/2, 1/2, 1/2)]
+ ((q*u^2-q)/(q*u^6-1))*B[(1, 0, 0, 0)] + B[(1, 1, 0, 0)]
- ((q*u^2-q)/(-q*u^6+1))*B[(0, 1, 0, 0)]
>>> E[L0((Integer(1),-Integer(1),Integer(0),Integer(0)))]                # long␣
↪time (23s)
((q^3*u^18-q^3*u^16+q*u^4-q^2*u^2-2*q*u^2+q^2+q)/(q^3*u^18-q^2*u^12-q*u^
↪6+1))*B[(0, 0, 0, 0)]
- ((q^3*u^18-q^3*u^16-q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, -1/2, -1/2, -
↪1/2)]
- ((q^3*u^18-q^3*u^16-q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, -1/2, -1/2,␣
↪1/2)]
+ ((q^3*u^18-q^3*u^16-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, -1/2, 1/2, -1/
↪2)]
+ ((q^3*u^18-q^3*u^16-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, -1/2, 1/2, 1/
↪2)]
+ ((q*u^8-q*u^6-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, 1/2, -1/2, -1/2)]
+ ((q*u^8-q*u^6-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, 1/2, -1/2, 1/2)]
- ((q*u^8-q*u^6-q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, 1/2, 1/2, -1/2)]
- ((q*u^8-q*u^6-q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, 1/2, 1/2, 1/2)]
- ((q^2*u^18-q^2*u^16+q*u^8-q*u^6-q*u^2-u^2+q+1)/(-q^3*u^18+q^2*u^12+q*u^6-
↪1))*B[(1, 0, 0, 0)]
+ B[(1, -1, 0, 0)] - ((u^2-1)/(-q^2*u^12+1))*B[(1, 1, 0, 0)]
- ((u^2-1)/(-q^2*u^12+1))*B[(1, 0, -1, 0)]
+ ((u^2-1)/(q^2*u^12-1))*B[(1, 0, 1, 0)]
- ((u^2-1)/(-q^2*u^12+1))*B[(1, 0, 0, -1)]
```

(continues on next page)

(continued from previous page)

```
- ((u^2-1)/(-q^2*u^12+1))*B[(1, 0, 0, 1)]
- ((q*u^2-q)/(-q*u^6+1))*B[(0, -1, 0, 0)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 1, 0, 0)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, -1, 0)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 1, 0)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 0, -1)]
- ((q*u^4-2*q*u^2+q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 0, 1)]
```

Next we test a twisted type (checked against Maple computation by Bogdan Ion for $q_1 = t^2$ and $q_2 = -1$):

```
sage: E = NonSymmetricMacdonaldPolynomials(["A",5,2])
sage: omega = E.keys()

sage: E[omega[1]]
B[(1, 0, 0)]

sage: E[-omega[1]]
B[(-1, 0, 0)]
+ ((q*q1^6+q*q1^5*q2+q1*q2^5+q2^6)/(q^3*q1^6+q^2*q1^5*q2+q*q1*q2^5+q2^6))*B[(1, 0,
↪ 0)]
+ ((q1+q2)/(q*q1+q2))*B[(0, -1, 0)] + ((q1+q2)/(q*q1+q2))*B[(0, 1, 0)]
+ ((q1+q2)/(q*q1+q2))*B[(0, 0, -1)] + ((q1+q2)/(q*q1+q2))*B[(0, 0, 1)]

sage: E[omega[2]]
-((q1*q2^3+q2^4)/(q*q1^4-q2^4))*B[(1, 0, 0)] + B[(0, 1, 0)]

sage: E[-omega[2]]
((q^2*q1^7+q^2*q1^6*q2-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^
↪7))*B[(1, 0, 0)]
+ B[(0, -1, 0)]
+ ((q*q1^5*q2^2+q*q1^4*q2^3-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^
↪7))*B[(0, 1, 0)]
- ((q1*q2+q2^2)/(q*q1^2-q2^2))*B[(0, 0, -1)]
+ ((q1*q2+q2^2)/(-q*q1^2+q2^2))*B[(0, 0, 1)]

sage: E[-omega[1]-omega[2]]
((q^3*q1^6+q^3*q1^5*q2+2*q^2*q1^6+3*q^2*q1^5*q2-q^2*q1^4*q2^2-2*q^2*q1^3*q2^3-
↪q*q1^5*q2-2*q*q1^4*q2^2+q*q1^3*q2^3+2*q*q1^2*q2^4-q*q1*q2^5-q*q2^6+q1^3*q2^3+q1^
↪2*q2^4-2*q1*q2^5-2*q2^6)/(q^4*q1^6+q^3*q1^5*q2-q^3*q1^4*q2^2+q*q1^2*q2^4-
↪q*q1*q2^5-q2^6))*B[(0, 0, 0)]
+ B[(-1, -1, 0)]
+ ((q*q1^4+q*q1^3*q2+q1*q2^3+q2^4)/(q^3*q1^4+q^2*q1^3*q2+q*q1*q2^3+q2^4))*B[(-1,␣
↪1, 0)]
+ ((q1+q2)/(q*q1+q2))*B[(-1, 0, -1)]
- ((q1+q2)/(-q*q1-q2))*B[(-1, 0, 1)]
+ ((q*q1^4+q*q1^3*q2+q1*q2^3+q2^4)/(q^3*q1^4+q^2*q1^3*q2+q*q1*q2^3+q2^4))*B[(1, -
↪1, 0)]
+ ((q^2*q1^6+q^2*q1^5*q2+q*q1^5*q2-q*q1^3*q2^3-q1^5*q2-q1^4*q2^2+q1^3*q2^3+q1^
↪2*q2^4-q1*q2^5-q2^6)/(q^4*q1^6+q^3*q1^5*q2-q^3*q1^4*q2^2+q*q1^2*q2^4-q*q1*q2^5-
↪q2^6))*B[(1, 1, 0)]
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^
↪2*q1^3*q2+q*q1*q2^3+q2^4))*B[(1, 0, -1)]
```

(continues on next page)

(continued from previous page)

```
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^
↪2*q1^3*q2+q*q1*q2^3+q2^4))*B[(1, 0, 1)]
+ ((q1+q2)/(q*q1+q2))*B[(0, -1, -1)]
+ ((q1+q2)/(q*q1+q2))*B[(0, -1, 1)]
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^
↪2*q1^3*q2+q*q1*q2^3+q2^4))*B[(0, 1, -1)]
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^
↪2*q1^3*q2+q*q1*q2^3+q2^4))*B[(0, 1, 1)]

sage: E[omega[1]-omega[2]]
((q^3*q1^7+q^3*q1^6*q2-q*q1*q2^6-q*q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^
↪7))*B[(0, 0, 0)]
+ B[(1, -1, 0)]
+ ((q*q1^5*q2^2+q*q1^4*q2^3-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^
↪7))*B[(1, 1, 0)]
- ((q1*q2+q2^2)/(q*q1^2-q2^2))*B[(1, 0, -1)]
+ ((q1*q2+q2^2)/(-q*q1^2+q2^2))*B[(1, 0, 1)]

sage: E[omega[3]]
-((q1*q2^2+q2^3)/(-q*q1^3-q2^3))*B[(1, 0, 0)]
- ((q1*q2^2+q2^3)/(-q*q1^3-q2^3))*B[(0, 1, 0)] + B[(0, 0, 1)]

sage: E[-omega[3]]
((q*q1^4*q2+q*q1^3*q2^2-q1*q2^4-q2^5)/(-q^2*q1^5-q2^5))*B[(1, 0, 0)]
+ ((q*q1^4*q2+q*q1^3*q2^2-q1*q2^4-q2^5)/(-q^2*q1^5-q2^5))*B[(0, 1, 0)]
+ B[(0, 0, -1)] - ((q1*q2^4+q2^5)/(-q^2*q1^5-q2^5))*B[(0, 0, 1)]
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials(["A",Integer(5),Integer(2)])
>>> omega = E.keys()

>>> E[omega[Integer(1)]]
B[(1, 0, 0)]

>>> E[-omega[Integer(1)]]
B[(-1, 0, 0)]
+ ((q*q1^6+q*q1^5*q2+q1*q2^5+q2^6)/(q^3*q1^6+q^2*q1^5*q2+q*q1*q2^5+q2^6))*B[(1, 0,
↪ 0)]
+ ((q1+q2)/(q*q1+q2))*B[(0, -1, 0)] + ((q1+q2)/(q*q1+q2))*B[(0, 1, 0)]
+ ((q1+q2)/(q*q1+q2))*B[(0, 0, -1)] + ((q1+q2)/(q*q1+q2))*B[(0, 0, 1)]

>>> E[omega[Integer(2)]]
-((q1*q2^3+q2^4)/(q*q1^4-q2^4))*B[(1, 0, 0)] + B[(0, 1, 0)]

>>> E[-omega[Integer(2)]]
((q^2*q1^7+q^2*q1^6*q2-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^
↪7))*B[(1, 0, 0)]
+ B[(0, -1, 0)]
+ ((q*q1^5*q2^2+q*q1^4*q2^3-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^
↪7))*B[(0, 1, 0)]
- ((q1*q2+q2^2)/(q*q1^2-q2^2))*B[(0, 0, -1)]
+ ((q1*q2+q2^2)/(-q*q1^2+q2^2))*B[(0, 0, 1)]
```

(continues on next page)

(continued from previous page)

```
>>> E[-omega[Integer(1)]-omega[Integer(2)]]
((q^3*q1^6+q^3*q1^5*q2+2*q^2*q1^6+3*q^2*q1^5*q2-q^2*q1^4*q2^2-2*q^2*q1^3*q2^3-
↪q*q1^5*q2-2*q*q1^4*q2^2+q*q1^3*q2^3+2*q*q1^2*q2^4-q*q1*q2^5-q*q2^6+q1^3*q2^3+q1^
↪2*q2^4-2*q1*q2^5-2*q2^6)/(q^4*q1^6+q^3*q1^5*q2-q^3*q1^4*q2^2+q*q1^2*q2^4-
↪q*q1*q2^5-q2^6))*B[(0, 0, 0)]
+ B[(-1, -1, 0)]
+ ((q*q1^4+q*q1^3*q2+q1*q2^3+q2^4)/(q^3*q1^4+q^2*q1^3*q2+q*q1*q2^3+q2^4))*B[(-1,␣
↪1, 0)]
+ ((q1+q2)/(q*q1+q2))*B[(-1, 0, -1)]
- ((q1+q2)/(-q*q1-q2))*B[(-1, 0, 1)]
+ ((q*q1^4+q*q1^3*q2+q1*q2^3+q2^4)/(q^3*q1^4+q^2*q1^3*q2+q*q1*q2^3+q2^4))*B[(1, -
↪1, 0)]
+ ((q^2*q1^6+q^2*q1^5*q2+q*q1^5*q2-q*q1^3*q2^3-q1^5*q2-q1^4*q2^2+q1^3*q2^3+q1^
↪2*q2^4-q1*q2^5-q2^6)/(q^4*q1^6+q^3*q1^5*q2-q^3*q1^4*q2^2+q*q1^2*q2^4-q*q1*q2^5-
↪q2^6))*B[(1, 1, 0)]
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^
↪2*q1^3*q2+q*q1*q2^3+q2^4))*B[(1, 0, -1)]
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^
↪2*q1^3*q2+q*q1*q2^3+q2^4))*B[(1, 0, 1)]
+ ((q1+q2)/(q*q1+q2))*B[(0, -1, -1)]
+ ((q1+q2)/(q*q1+q2))*B[(0, -1, 1)]
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^
↪2*q1^3*q2+q*q1*q2^3+q2^4))*B[(0, 1, -1)]
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^
↪2*q1^3*q2+q*q1*q2^3+q2^4))*B[(0, 1, 1)]

>>> E[omega[Integer(1)]-omega[Integer(2)]]
((q^3*q1^7+q^3*q1^6*q2-q*q1*q2^6-q*q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^
↪7))*B[(0, 0, 0)]
+ B[(1, -1, 0)]
+ ((q*q1^5*q2^2+q*q1^4*q2^3-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^
↪7))*B[(1, 1, 0)]
- ((q1*q2+q2^2)/(q*q1^2-q2^2))*B[(1, 0, -1)]
+ ((q1*q2+q2^2)/(-q*q1^2+q2^2))*B[(1, 0, 1)]

>>> E[omega[Integer(3)]]
-((q1*q2^2+q2^3)/(-q*q1^3-q2^3))*B[(1, 0, 0)]
- ((q1*q2^2+q2^3)/(-q*q1^3-q2^3))*B[(0, 1, 0)] + B[(0, 0, 1)]

>>> E[-omega[Integer(3)]]
((q*q1^4*q2+q*q1^3*q2^2-q1*q2^4-q2^5)/(-q^2*q1^5-q2^5))*B[(1, 0, 0)]
+ ((q*q1^4*q2+q*q1^3*q2^2-q1*q2^4-q2^5)/(-q^2*q1^5-q2^5))*B[(0, 1, 0)]
+ B[(0, 0, -1)] - ((q1*q2^4+q2^5)/(-q^2*q1^5-q2^5))*B[(0, 0, 1)]
```

Comparison with the energy function of crystals

Next we test that the nonsymmetric Macdonald polynomials at $t = 0$ match with the one-dimensional configuration sums involving Kirillov-Reshetikhin crystals for various types. See [LNSSS12]:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
```

(continues on next page)

(continued from previous page)

```
sage: KL = RootSystem(["A",5,2]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: E[-omega[1]].map_coefficients(lambda x: x.subs(t=0))
B[(-1, 0, 0)] + B[(1, 0, 0)] + B[(0, -1, 0)] + B[(0, 1, 0)]
 + B[(0, 0, -1)] + B[(0, 0, 1)]
sage: E[-omega[2]].map_coefficients(lambda x: x.subs(t=0))      # long time (3s)
(q+2)*B[(0, 0, 0)] + B[(-1, -1, 0)] + B[(-1, 1, 0)] + B[(-1, 0, -1)]
 + B[(-1, 0, 1)] + B[(1, -1, 0)] + B[(1, 1, 0)] + B[(1, 0, -1)] + B[(1, 0, 1)]
 + B[(0, -1, -1)] + B[(0, -1, 1)] + B[(0, 1, -1)] + B[(0, 1, 1)]
```

```
>>> from sage.all import *
>>> K = QQ['q,t'].fraction_field()
>>> q,t = K.gens()
>>> KL = RootSystem(["A",Integer(5),Integer(2)]).ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> E[-omega[Integer(1)]].map_coefficients(lambda x: x.subs(t=Integer(0)))
B[(-1, 0, 0)] + B[(1, 0, 0)] + B[(0, -1, 0)] + B[(0, 1, 0)]
 + B[(0, 0, -1)] + B[(0, 0, 1)]
>>> E[-omega[Integer(2)]].map_coefficients(lambda x: x.subs(t=Integer(0)))      #␣
↪long time (3s)
(q+2)*B[(0, 0, 0)] + B[(-1, -1, 0)] + B[(-1, 1, 0)] + B[(-1, 0, -1)]
 + B[(-1, 0, 1)] + B[(1, -1, 0)] + B[(1, 1, 0)] + B[(1, 0, -1)] + B[(1, 0, 1)]
 + B[(0, -1, -1)] + B[(0, -1, 1)] + B[(0, 1, -1)] + B[(0, 1, 1)]
```

```
sage: KL = RootSystem(["C",3,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: E[-omega[2]].map_coefficients(lambda x: x.subs(t=0))      # long time (5s)
2*B[(0, 0, 0)] + B[(-1, -1, 0)] + B[(-1, 1, 0)] + B[(-1, 0, -1)]
+ B[(-1, 0, 1)] + B[(1, -1, 0)] + B[(1, 1, 0)] + B[(1, 0, -1)] + B[(1, 0, 1)]
+ B[(0, -1, -1)] + B[(0, -1, 1)] + B[(0, 1, -1)] + B[(0, 1, 1)]
```

```
>>> from sage.all import *
>>> KL = RootSystem(["C",Integer(3),Integer(1)]).ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> E[-omega[Integer(2)]].map_coefficients(lambda x: x.subs(t=Integer(0)))      #␣
↪long time (5s)
2*B[(0, 0, 0)] + B[(-1, -1, 0)] + B[(-1, 1, 0)] + B[(-1, 0, -1)]
+ B[(-1, 0, 1)] + B[(1, -1, 0)] + B[(1, 1, 0)] + B[(1, 0, -1)] + B[(1, 0, 1)]
+ B[(0, -1, -1)] + B[(0, -1, 1)] + B[(0, 1, -1)] + B[(0, 1, 1)]
```

```
sage: R = RootSystem(['C',3,1])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: (E[-2*omega[1]].map_coefficients(lambda x: x.subs(t=0))   # long time (15s)
```

(continues on next page)

(continued from previous page)

```
....:    == LS.one_dimensional_configuration_sum(q))
True
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1] + La[2])
sage: (E[-omega[1] - omega[2]].map_coefficients(lambda x: x.subs(t=0))   # long␣
↪time (45s)
....:    == LS.one_dimensional_configuration_sum(q))
True
```

```
>>> from sage.all import *
>>> R = RootSystem(['C',Integer(3),Integer(1)])
>>> KL = R.weight_lattice(extended=True).algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])
>>> (E[-Integer(2)*omega[Integer(1)]].map_coefficients(lambda x: x.
↪subs(t=Integer(0)))    # long time (15s)
...    == LS.one_dimensional_configuration_sum(q))
True
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)] + La[Integer(2)])
>>> (E[-omega[Integer(1)] - omega[Integer(2)]].map_coefficients(lambda x: x.
↪subs(t=Integer(0)))  # long time (45s)
...    == LS.one_dimensional_configuration_sum(q))
True
```

```
sage: R = RootSystem(['C',2,1])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: for d in range(1,3):                # long time (45s)
....:     for x,y in IntegerVectors(d,2):
....:         weight = x*La[1]+y*La[2]
....:         weight0 = -x*omega[1]-y*omega[2]
....:         LS = crystals.ProjectedLevelZeroLSPaths(weight)
....:         assert (E[weight0].map_coefficients(lambda x:x.subs(t=0))
....:                  == LS.one_dimensional_configuration_sum(q))
```

```
>>> from sage.all import *
>>> R = RootSystem(['C',Integer(2),Integer(1)])
>>> KL = R.weight_lattice(extended=True).algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> La = R.weight_space().basis()
>>> for d in range(Integer(1),Integer(3)):                # long time (45s)
...     for x,y in IntegerVectors(d,Integer(2)):
...         weight = x*La[Integer(1)]+y*La[Integer(2)]
...         weight0 = -x*omega[Integer(1)]-y*omega[Integer(2)]
...         LS = crystals.ProjectedLevelZeroLSPaths(weight)
...         assert (E[weight0].map_coefficients(lambda x:x.subs(t=Integer(0)))
...                  == LS.one_dimensional_configuration_sum(q))
```

```
sage: R = RootSystem(['B',3,1])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()

sage: # needs sage.combinat
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: (E[-2*omega[1]].map_coefficients(lambda x: x.subs(t=0))    # long time (23s)
....:    == LS.one_dimensional_configuration_sum(q))
True
sage: B = crystals.KirillovReshetikhin(['B',3,1],1,1)
sage: T = crystals.TensorProduct(B,B)
sage: (T.one_dimensional_configuration_sum(q)    # long time (2s)
....:    == LS.one_dimensional_configuration_sum(q))
True
```

```
>>> from sage.all import *
>>> R = RootSystem(['B',Integer(3),Integer(1)])
>>> KL = R.weight_lattice(extended=True).algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> La = R.weight_space().basis()

>>> # needs sage.combinat
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])
>>> (E[-Integer(2)*omega[Integer(1)]].map_coefficients(lambda x: x.
↪subs(t=Integer(0)))    # long time (23s)
...    == LS.one_dimensional_configuration_sum(q))
True
>>> B = crystals.KirillovReshetikhin(['B',Integer(3),Integer(1)],Integer(1),
↪Integer(1))
>>> T = crystals.TensorProduct(B,B)
>>> (T.one_dimensional_configuration_sum(q)    # long time (2s)
...    == LS.one_dimensional_configuration_sum(q))
True
```

```
sage: R = RootSystem(['BC',3,2])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])                          #␣
↪needs sage.combinat
sage: (E[-2*omega[1]].map_coefficients(lambda x: x.subs(t=0))    # long time (21s),
↪ needs sage.combinat
....:    == LS.one_dimensional_configuration_sum(q))
True
```

```
>>> from sage.all import *
>>> R = RootSystem(['BC',Integer(3),Integer(2)])
>>> KL = R.weight_lattice(extended=True).algebra(K)
```

(continues on next page)

(continued from previous page)

```
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])                    ␣
↪              # needs sage.combinat
>>> (E[-Integer(2)*omega[Integer(1)]].map_coefficients(lambda x: x.
↪subs(t=Integer(0)))    # long time (21s), needs sage.combinat
...    == LS.one_dimensional_configuration_sum(q))
True
```

```
sage: R = RootSystem(CartanType(['BC',3,2]).dual())
sage: KL = R.weight_space(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()

sage: # long time, needs sage.combinat
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: g = E[-2*omega[1]].map_coefficients(lambda x: x.subs(t=0))     # 30s
sage: f = LS.one_dimensional_configuration_sum(q)                    # 1.5s
sage: P = g.support()[0].parent()
sage: B = P.algebra(q.parent())
sage: sum(p[1]*B(P(p[0])) for p in f) == g
True
```

```
>>> from sage.all import *
>>> R = RootSystem(CartanType(['BC',Integer(3),Integer(2)]).dual())
>>> KL = R.weight_space(extended=True).algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> La = R.weight_space().basis()

>>> # long time, needs sage.combinat
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])
>>> g = E[-Integer(2)*omega[Integer(1)]].map_coefficients(lambda x: x.
↪subs(t=Integer(0)))     # 30s
>>> f = LS.one_dimensional_configuration_sum(q)                    # 1.5s
>>> P = g.support()[Integer(0)].parent()
>>> B = P.algebra(q.parent())
>>> sum(p[Integer(1)]*B(P(p[Integer(0)])) for p in f) == g
True
```

```
sage: C = CartanType(['G',2,1])
sage: R = RootSystem(C.dual())
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.combinat
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: (E[-2*omega[1]].map_coefficients(lambda x: x.subs(t=0))           # long␣
↪time (20s), not tested
....:    == LS.one_dimensional_configuration_sum(q)
True
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1] + La[2])
sage: (E[-omega[1]-omega[2]].map_coefficients(lambda x: x.subs(t=0))     # long␣
↪time (23s), not tested
....:    == LS.one_dimensional_configuration_sum(q))
True
```

```
>>> from sage.all import *
>>> C = CartanType(['G',Integer(2),Integer(1)])
>>> R = RootSystem(C.dual())
>>> K = QQ['q,t'].fraction_field()
>>> q,t = K.gens()
>>> KL = R.weight_lattice(extended=True).algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> La = R.weight_space().basis()

>>> # needs sage.combinat
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])
>>> (E[-Integer(2)*omega[Integer(1)]].map_coefficients(lambda x: x.
↪subs(t=Integer(0)))            # long time (20s), not tested
...    == LS.one_dimensional_configuration_sum(q)
True
>>> LS = crystals.ProjectedLevelZeroLSPaths(La[Integer(1)] + La[Integer(2)])
>>> (E[-omega[Integer(1)]-omega[Integer(2)]].map_coefficients(lambda x: x.
↪subs(t=Integer(0)))     # long time (23s), not tested
...    == LS.one_dimensional_configuration_sum(q))
True
```

The next test breaks if the energy is not scaled by the translation factor for dual type $G_2^{(1)}$:

```
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])                    #␣
↪needs sage.combinat
sage: (E[-2*omega[1]-omega[2]].map_coefficients(lambda x: x.subs(t=0))   # long␣
↪time (100s), not tested, needs sage.combinat
....:    == LS.one_dimensional_configuration_sum(q)
True

sage: R = RootSystem(['D',4,1])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: for d in range(1,2):                # long time (41s)
....:     for a,b,c,d in IntegerVectors(d,4):
....:         weight = a*La[1] + b*La[2] + c*La[3] + d*La[4]
....:         weight0 = -a*omega[1] - b*omega[2] - c*omega[3] - d*omega[4]
```

(continues on next page)

(continued from previous page)

```
....:         LS = crystals.ProjectedLevelZeroLSPaths(weight)
....:         assert (E[weight0].map_coefficients(lambda x: x.subs(t=0))
....:                  == LS.one_dimensional_configuration_sum(q))
```

```
>>> from sage.all import *
>>> LS = crystals.
↪ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)]+La[Integer(2)])            ␣
↪       # needs sage.combinat
>>> (E[-Integer(2)*omega[Integer(1)]-omega[Integer(2)]].map_coefficients(lambda␣
↪x: x.subs(t=Integer(0)))  # long time (100s), not tested, needs sage.combinat
...   == LS.one_dimensional_configuration_sum(q)
True

>>> R = RootSystem(['D',Integer(4),Integer(1)])
>>> KL = R.weight_lattice(extended=True).algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> La = R.weight_space().basis()
>>> for d in range(Integer(1),Integer(2)):               # long time (41s)
...     for a,b,c,d in IntegerVectors(d,Integer(4)):
...         weight = a*La[Integer(1)] + b*La[Integer(2)] + c*La[Integer(3)] +␣
↪d*La[Integer(4)]
...         weight0 = -a*omega[Integer(1)] - b*omega[Integer(2)] -␣
↪c*omega[Integer(3)] - d*omega[Integer(4)]
...         LS = crystals.ProjectedLevelZeroLSPaths(weight)
...         assert (E[weight0].map_coefficients(lambda x: x.subs(t=Integer(0)))
...                  == LS.one_dimensional_configuration_sum(q))
```

Todo

add his notes in latex

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q,q1,q2=K.gens()
sage: L = RootSystem(["A",4,2]).ambient_space()
sage: L.cartan_type()
['BC', 2, 2]
sage: L.null_root()
2*e['delta']
sage: L.simple_roots()
Finite family {0: -e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
sage: KL = L.algebra(K)
sage: KL0 = KL.classical()
sage: L0 = L.classical()
sage: L0.cartan_type()
['C', 2]

sage: E = NonSymmetricMacdonaldPolynomials(KL, q=q,q1=q1,q2=q2)
sage: E.keys()
Ambient space of the Root system of type ['C', 2]
```

(continues on next page)

(continued from previous page)

```
sage: E.keys().simple_roots()
Finite family {1: (1, -1), 2: (0, 2)}
sage: omega = E.keys().fundamental_weights()

sage: E[0*omega[1]]
B[(0, 0)]
sage: E[omega[1]]
-((q*q1*q2^3+q*q2^4)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]

sage: E[2*omega[2]]  # not checked against Bogdan's notes, but a good self-
↪consistency test  # long time
-((q^12*q1^6+q^12*q1^5*q2-2*q^10*q1^5*q2-5*q^10*q1^4*q2^2-3*q^10*q1^3*q2^3-2*q^
↪8*q1^5*q2-4*q^8*q1^4*q2^2-q^8*q1^3*q2^3+q^8*q1^2*q2^4-q^8*q1*q2^5-q^8*q2^6+q^
↪6*q1^3*q2^3-q^6*q1^2*q2^4-4*q^6*q1*q2^5-2*q^6*q2^6-q^4*q1^3*q2^3-3*q^4*q1^2*q2^
↪4-4*q^4*q1*q2^5-2*q^4*q2^6)/(-q^12*q1^6-q^10*q1^5*q2-q^8*q1^3*q2^3+q^6*q1^4*q2^
↪2-q^6*q1^2*q2^4+q^4*q1^3*q2^3+q^2*q1*q2^5+q2^6))*B[(0, 0)]
+ ((q^7*q1^2*q2+2*q^7*q1*q2^2+q^7*q2^3+q^5*q1^2*q2+2*q^5*q1*q2^2+q^5*q2^3)/(-q^
↪8*q1^3-q^6*q1^2*q2+q^2*q1*q2^2+q2^3))*B[(-1, 0)]
- ((q^6*q1*q2+q^6*q2^2)/(q^6*q1^2-q2^2))*B[(-1, -1)]
+ ((q^6*q1^2*q2+2*q^6*q1*q2^2+q^6*q2^3+q^4*q1^2*q2+2*q^4*q1*q2^2+q^4*q2^3)/(-q^
↪8*q1^3-q^6*q1^2*q2+q^2*q1*q2^2+q2^3))*B[(-1, 1)]
- ((q^3*q1*q2+q^3*q2^2)/(q^6*q1^2-q2^2))*B[(-1, 2)]
+ ((q^7*q1^3+q^7*q1^2*q2-q^7*q1*q2^2-q^7*q2^3-2*q^5*q1^2*q2-4*q^5*q1*q2^2-2*q^
↪5*q2^3-2*q^3*q1^2*q2-4*q^3*q1*q2^2-2*q^3*q2^3)/(q^8*q1^3+q^6*q1^2*q2-q^2*q1*q2^
↪2-q2^3))*B[(1, 0)]
+ ((q^6*q1^2*q2+2*q^6*q1*q2^2+q^6*q2^3+q^4*q1^2*q2+2*q^4*q1*q2^2+q^4*q2^3)/(-q^
↪8*q1^3-q^6*q1^2*q2+q^2*q1*q2^2+q2^3))*B[(1, -1)]
+ ((q^8*q1^3+q^8*q1^2*q2+q^6*q1^3+q^6*q1^2*q2-q^6*q1*q2^2-q^6*q2^3-2*q^4*q1^2*q2-
↪4*q^4*q1*q2^2-2*q^4*q2^3-q^2*q1^2*q2-3*q^2*q1*q2^2-2*q^2*q2^3)/(q^8*q1^3+q^6*q1^
↪2*q2-q^2*q1*q2^2-q2^3))*B[(1, 1)]
+ ((q^5*q1^2+q^5*q1*q2-q^3*q1*q2-q^3*q2^2-q*q1*q2-q*q2^2)/(q^6*q1^2-q2^2))*B[(1,␣
↪2)]
- ((q^6*q1^2+q^6*q1*q2-q^4*q1*q2-q^4*q2^2-q^2*q1*q2-q^2*q2^2)/(-q^6*q1^2+q2^
↪2))*B[(2, 0)]
- ((q^3*q1*q2+q^3*q2^2)/(q^6*q1^2-q2^2))*B[(2, -1)]
- ((q^5*q1^2+q^5*q1*q2-q^3*q1*q2-q^3*q2^2-q*q1*q2-q*q2^2)/(-q^6*q1^2+q2^2))*B[(2,␣
↪1)]
+ B[(2, 2)]
+ ((q^7*q1^2*q2+2*q^7*q1*q2^2+q^7*q2^3+q^5*q1^2*q2+2*q^5*q1*q2^2+q^5*q2^3)/(-q^
↪8*q1^3-q^6*q1^2*q2+q^2*q1*q2^2+q2^3))*B[(0, -1)]
+ ((q^7*q1^3+q^7*q1^2*q2-q^7*q1*q2^2-q^7*q2^3-2*q^5*q1^2*q2-4*q^5*q1*q2^2-2*q^
↪5*q2^3-2*q^3*q1^2*q2-4*q^3*q1*q2^2-2*q^3*q2^3)/(q^8*q1^3+q^6*q1^2*q2-q^2*q1*q2^
↪2-q2^3))*B[(0, 1)]
+ ((q^6*q1^2+q^6*q1*q2-q^4*q1*q2-q^4*q2^2-q^2*q1*q2-q^2*q2^2)/(q^6*q1^2-q2^
↪2))*B[(0, 2)]
sage: E.recursion(2*omega[2])
[0, 1, 0, 2, 1, 0, 2, 1, 0]
```

```
>>> from sage.all import *
>>> K = QQ['q,q1,q2'].fraction_field()
>>> q,q1,q2=K.gens()
>>> L = RootSystem(["A",Integer(4),Integer(2)]).ambient_space()
```

(continues on next page)

(continued from previous page)

```
>>> L.cartan_type()
['BC', 2, 2]
>>> L.null_root()
2*e['delta']
>>> L.simple_roots()
Finite family {0: -e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
>>> KL = L.algebra(K)
>>> KL0 = KL.classical()
>>> L0 = L.classical()
>>> L0.cartan_type()
['C', 2]

>>> E = NonSymmetricMacdonaldPolynomials(KL, q=q,q1=q1,q2=q2)
>>> E.keys()
Ambient space of the Root system of type ['C', 2]
>>> E.keys().simple_roots()
Finite family {1: (1, -1), 2: (0, 2)}
>>> omega = E.keys().fundamental_weights()

>>> E[Integer(0)*omega[Integer(1)]]
B[(0, 0)]
>>> E[omega[Integer(1)]]
-((q*q1*q2^3+q*q2^4)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]

>>> E[Integer(2)*omega[Integer(2)]]  # not checked against Bogdan's notes, but a␣
↪good self-consistency test  # long time
-((q^12*q1^6+q^12*q1^5*q2-2*q^10*q1^5*q2-5*q^10*q1^4*q2^2-3*q^10*q1^3*q2^3-2*q^
↪8*q1^5*q2-4*q^8*q1^4*q2^2-q^8*q1^3*q2^3+q^8*q1^2*q2^4-q^8*q1*q2^5-q^8*q2^6+q^
↪6*q1^3*q2^3-q^6*q1^2*q2^4-4*q^6*q1*q2^5-2*q^6*q2^6-q^4*q1^3*q2^3-3*q^4*q1^2*q2^
↪4-4*q^4*q1*q2^5-2*q^4*q2^6)/(-q^12*q1^6-q^10*q1^5*q2-q^8*q1^3*q2^3+q^6*q1^4*q2^
↪2-q^6*q1^2*q2^4+q^4*q1^3*q2^3+q^2*q1*q2^5+q2^6))*B[(0, 0)]
+ ((q^7*q1^2*q2+2*q^7*q1*q2^2+q^7*q2^3+q^5*q1^2*q2+2*q^5*q1*q2^2+q^5*q2^3)/(-q^
↪8*q1^3-q^6*q1^2*q2+q^2*q1*q2^2+q2^3))*B[(-1, 0)]
- ((q^6*q1*q2+q^6*q2^2)/(q^6*q1^2-q2^2))*B[(-1, -1)]
+ ((q^6*q1^2*q2+2*q^6*q1*q2^2+q^6*q2^3+q^4*q1^2*q2+2*q^4*q1*q2^2+q^4*q2^3)/(-q^
↪8*q1^3-q^6*q1^2*q2+q^2*q1*q2^2+q2^3))*B[(-1, 1)]
- ((q^3*q1*q2+q^3*q2^2)/(q^6*q1^2-q2^2))*B[(-1, 2)]
+ ((q^7*q1^3+q^7*q1^2*q2-q^7*q1*q2^2-q^7*q2^3-2*q^5*q1^2*q2-4*q^5*q1*q2^2-2*q^
↪5*q2^3-2*q^3*q1^2*q2-4*q^3*q1*q2^2-2*q^3*q2^3)/(q^8*q1^3+q^6*q1^2*q2-q^2*q1*q2^
↪2-q2^3))*B[(1, 0)]
+ ((q^6*q1^2*q2+2*q^6*q1*q2^2+q^6*q2^3+q^4*q1^2*q2+2*q^4*q1*q2^2+q^4*q2^3)/(-q^
↪8*q1^3-q^6*q1^2*q2+q^2*q1*q2^2+q2^3))*B[(1, -1)]
+ ((q^8*q1^3+q^8*q1^2*q2+q^6*q1^3+q^6*q1^2*q2-q^6*q1*q2^2-q^6*q2^3-2*q^4*q1^2*q2-
↪4*q^4*q1*q2^2-2*q^4*q2^3-q^2*q1^2*q2-3*q^2*q1*q2^2-2*q^2*q2^3)/(q^8*q1^3+q^6*q1^
↪2*q2-q^2*q1*q2^2-q2^3))*B[(1, 1)]
+ ((q^5*q1^2+q^5*q1*q2-q^3*q1*q2-q^3*q2^2-q*q1*q2-q*q2^2)/(q^6*q1^2-q2^2))*B[(1,␣
↪2)]
- ((q^6*q1^2+q^6*q1*q2-q^4*q1*q2-q^4*q2^2-q^2*q1*q2-q^2*q2^2)/(-q^6*q1^2+q2^
↪2))*B[(2, 0)]
- ((q^3*q1*q2+q^3*q2^2)/(q^6*q1^2-q2^2))*B[(2, -1)]
- ((q^5*q1^2+q^5*q1*q2-q^3*q1*q2-q^3*q2^2-q*q1*q2-q*q2^2)/(-q^6*q1^2+q2^2))*B[(2,␣
↪1)]
```

(continues on next page)

(continued from previous page)

```
+ B[(2, 2)]
+ ((q^7*q1^2*q2+2*q^7*q1*q2^2+q^7*q2^3+q^5*q1^2*q2+2*q^5*q1*q2^2+q^5*q2^3)/(-q^
↪8*q1^3-q^6*q1^2*q2+q^2*q1*q2^2+q2^3))*B[(0, -1)]
+ ((q^7*q1^3+q^7*q1^2*q2-q^7*q1*q2^2-q^7*q2^3-2*q^5*q1^2*q2-4*q^5*q1*q2^2-2*q^
↪5*q2^3-2*q^3*q1^2*q2-4*q^3*q1*q2^2-2*q^3*q2^3)/(q^8*q1^3+q^6*q1^2*q2-q^2*q1*q2^
↪2-q2^3))*B[(0, 1)]
+ ((q^6*q1^2+q^6*q1*q2-q^4*q1*q2-q^4*q2^2-q^2*q1*q2-q^2*q2^2)/(q^6*q1^2-q2^
↪2))*B[(0, 2)]
>>> E.recursion(Integer(2)*omega[Integer(2)])
[0, 1, 0, 2, 1, 0, 2, 1, 0]
```

Some tests that the T s are implemented properly by hand defining the Y s in terms of them:

```
sage: T = E._T_Y
sage: Ye1     = T.Tw((1,2,1,0), scalar=(-1/(q1*q2))^2)
sage: Ye2     = T.Tw((2,1,0,1), signs=(1,1,1,-1), scalar=(-1/(q1*q2)))
sage: Yalpha0 = T.Tw((0,1,2,1), signs=(-1,-1,-1,-1), scalar=q^-1*(-q1*q2)^2)
sage: Yalpha1 = T.Tw((1,2,0,1,2,0), signs=(1,1,-1,1,-1,1), scalar=-1/(q1*q2))
sage: Yalpha2 = T.Tw((2,1,0,1,2,1,0,1), signs=(1,1,1,-1,1,1,1,-1),
....:                scalar=(1/(q1*q2))^2)

sage: Ye1(KL0.one())
q1^2/q2^2*B[(0, 0)]
sage: Ye2(KL0.one())
-q1/q2*B[(0, 0)]
sage: Yalpha0(KL0.one())
q2^2/(q*q1^2)*B[(0, 0)]
sage: Yalpha1(KL0.one())
-q1/q2*B[(0, 0)]
sage: Yalpha2(KL0.one())
q1^2/q2^2*B[(0, 0)]
```

```
>>> from sage.all import *
>>> T = E._T_Y
>>> Ye1     = T.Tw((Integer(1),Integer(2),Integer(1),Integer(0)), scalar=(-
↪Integer(1)/(q1*q2))**Integer(2))
>>> Ye2     = T.Tw((Integer(2),Integer(1),Integer(0),Integer(1)),␣
↪signs=(Integer(1),Integer(1),Integer(1),-Integer(1)), scalar=(-Integer(1)/
↪(q1*q2)))
>>> Yalpha0 = T.Tw((Integer(0),Integer(1),Integer(2),Integer(1)), signs=(-
↪Integer(1),-Integer(1),-Integer(1),-Integer(1)), scalar=q**-Integer(1)*(-
↪q1*q2)**Integer(2))
>>> Yalpha1 = T.Tw((Integer(1),Integer(2),Integer(0),Integer(1),Integer(2),
↪Integer(0)), signs=(Integer(1),Integer(1),-Integer(1),Integer(1),-Integer(1),
↪Integer(1)), scalar=-Integer(1)/(q1*q2))
>>> Yalpha2 = T.Tw((Integer(2),Integer(1),Integer(0),Integer(1),Integer(2),
↪Integer(1),Integer(0),Integer(1)), signs=(Integer(1),Integer(1),Integer(1),-
↪Integer(1),Integer(1),Integer(1),Integer(1),-Integer(1)),
...                scalar=(Integer(1)/(q1*q2))**Integer(2))

>>> Ye1(KL0.one())
q1^2/q2^2*B[(0, 0)]
```

(continues on next page)

(continued from previous page)

```
>>> Ye2(KL0.one())
-q1/q2*B[(0, 0)]
>>> Yalpha0(KL0.one())
q2^2/(q*q1^2)*B[(0, 0)]
>>> Yalpha1(KL0.one())
-q1/q2*B[(0, 0)]
>>> Yalpha2(KL0.one())
q1^2/q2^2*B[(0, 0)]
```

Testing the Y s directly:

```
sage: Y = E.Y()
sage: Y.keys()
Coroot lattice of the Root system of type ['BC', 2, 2]
sage: alpha = Y.keys().simple_roots()
sage: L(alpha[0])
-2*e[0] + e['deltacheck']
sage: L(alpha[1])
e[0] - e[1]
sage: L(alpha[2])
e[1]
sage: Y[alpha[0]].word
(0, 1, 2, 1)
sage: Y[alpha[0]].signs
(-1, -1, -1, -1)
sage: Y[alpha[0]].scalar   # mind that Sage's q is the usual q^{1/2}
q1^2*q2^2/q
sage: Y[alpha[0]](KL0.one())
q2^2/(q*q1^2)*B[(0, 0)]

sage: Y[alpha[1]].word
(1, 2, 0, 1, 2, 0)
sage: Y[alpha[1]].signs
(1, 1, -1, 1, -1, 1)
sage: Y[alpha[1]].scalar
1/(-q1*q2)

sage: Y[alpha[2]].word  # Bogdan says it should be the square of that; do we need␣
↪to take translation factors into account or not?
(2, 1, 0, 1)
sage: Y[alpha[2]].signs
(1, 1, 1, -1)
sage: Y[alpha[2]].scalar
1/(-q1*q2)
```

```
>>> from sage.all import *
>>> Y = E.Y()
>>> Y.keys()
Coroot lattice of the Root system of type ['BC', 2, 2]
>>> alpha = Y.keys().simple_roots()
>>> L(alpha[Integer(0)])
-2*e[0] + e['deltacheck']
```

(continues on next page)

(continued from previous page)

```
>>> L(alpha[Integer(1)])
e[0] - e[1]
>>> L(alpha[Integer(2)])
e[1]
>>> Y[alpha[Integer(0)]].word
(0, 1, 2, 1)
>>> Y[alpha[Integer(0)]].signs
(-1, -1, -1, -1)
>>> Y[alpha[Integer(0)]].scalar   # mind that Sage's q is the usual q^{1/2}
q1^2*q2^2/q
>>> Y[alpha[Integer(0)]](KL0.one())
q2^2/(q*q1^2)*B[(0, 0)]

>>> Y[alpha[Integer(1)]].word
(1, 2, 0, 1, 2, 0)
>>> Y[alpha[Integer(1)]].signs
(1, 1, -1, 1, -1, 1)
>>> Y[alpha[Integer(1)]].scalar
1/(-q1*q2)

>>> Y[alpha[Integer(2)]].word  # Bogdan says it should be the square of that; do␣
↪we need to take translation factors into account or not?
(2, 1, 0, 1)
>>> Y[alpha[Integer(2)]].signs
(1, 1, 1, -1)
>>> Y[alpha[Integer(2)]].scalar
1/(-q1*q2)
```

Checking the provided nonsymmetric Macdonald polynomial:

```
sage: E10 = KL0.monomial(L0((1,0))) + KL0( q*(1-(-q1/q2)) / (1-q^2*(-q1/q2)^4) )
sage: E10 == E[omega[1]]
True
sage: E.eigenvalues(E10)  # not checked
[q*q1^2/q2^2, q2^3/(-q^2*q1^3), q1/(-q2)]
```

```
>>> from sage.all import *
>>> E10 = KL0.monomial(L0((Integer(1),Integer(0)))) + KL0( q*(Integer(1)-(-q1/
↪q2)) / (Integer(1)-q**Integer(2)*(-q1/q2)**Integer(4)) )
>>> E10 == E[omega[Integer(1)]]
True
>>> E.eigenvalues(E10)  # not checked
[q*q1^2/q2^2, q2^3/(-q^2*q1^3), q1/(-q2)]
```

Checking T0check:

```
sage: T0check_on_basis = KL.T0_check_on_basis(q1,q2, convention='dominant')
sage: T0check_on_basis.phi  # note: this is in fact a0 phi
(2, 0)
sage: T0check_on_basis.v    # what to match it with?
(1,)
sage: T0check_on_basis.j    # what to match it with?
```

(continues on next page)

(continued from previous page)

```
2
sage: T0check_on_basis(KL0.basis().keys().zero())
-q1^2/q2*B[(1, 0)]

sage: T0check = E._T[0]
sage: T0check(KL0.one())
-q1^2/q2*B[(1, 0)]
```

```
>>> from sage.all import *
>>> T0check_on_basis = KL.T0_check_on_basis(q1,q2, convention='dominant')
>>> T0check_on_basis.phi  # note: this is in fact a0 phi
(2, 0)
>>> T0check_on_basis.v    # what to match it with?
(1,)
>>> T0check_on_basis.j    # what to match it with?
2
>>> T0check_on_basis(KL0.basis().keys().zero())
-q1^2/q2*B[(1, 0)]

>>> T0check = E._T[Integer(0)]
>>> T0check(KL0.one())
-q1^2/q2*B[(1, 0)]
```

Systematic tests of nonsymmetric Macdonald polynomials in type $A_1^{(1)}$, in the weight lattice. Each time, we specify the eigenvalues for the action of Y_{α_0}, and Y_{α_1}:

```
sage: K = QQ['q','t'].fraction_field()
sage: q,t = K.gens()
sage: KL = RootSystem(["A",1,1]).weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: omega = E.keys().fundamental_weights()

sage: x = E[0*omega[1]]; x
B[0]
sage: E.eigenvalues(x)
[1/(q*t), t]
sage: x.is_one()
True
sage: x.parent()
Algebra of the Weight lattice of the Root system of type ['A', 1]
over Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field
sage: E[omega[1]]
B[Lambda[1]]
sage: E.eigenvalues(_)
[t, 1/(q*t)]
sage: E[2*omega[1]]
-((q*t-q)/(-q*t+1))*B[0] + B[2*Lambda[1]]
sage: E.eigenvalues(_)
[q*t, 1/(q^2*t)]
sage: E[3*omega[1]]
-((q^2*t-q^2)/(-q^2*t+1))*B[-Lambda[1]]
- ((q^2*t-q^2+q*t-q)/(-q^2*t+1))*B[Lambda[1]] + B[3*Lambda[1]]
```

(continues on next page)

(continued from previous page)

```
sage: E.eigenvalues(_)
[q^2*t, 1/(q^3*t)]
sage: E[4*omega[1]]
((q^5*t^2-q^5*t+q^4*t^2-2*q^4*t+q^3*t^2+q^4-2*q^3*t+q^3-q^2*t+q^2)/(q^5*t^2-q^3*t-
↪q^2*t+1))*B[0]
- ((q^3*t-q^3)/(-q^3*t+1))*B[-2*Lambda[1]]
- ((q^3*t-q^3+q^2*t-q^2+q*t-q)/(-q^3*t+1))*B[2*Lambda[1]]
+ B[4*Lambda[1]]
sage: E.eigenvalues(_)
[q^3*t, 1/(q^4*t)]
sage: E[6*omega[1]]
-((q^12*t^3-q^12*t^2+q^11*t^3-2*q^11*t^2+2*q^10*t^3+q^11*t-4*q^10*t^2+2*q^9*t^
↪3+2*q^10*t-5*q^9*t^2+2*q^8*t^3+4*q^9*t-6*q^8*t^2+q^7*t^3-q^9+5*q^8*t-5*q^7*t^
↪2+q^6*t^3-q^8+6*q^7*t-4*q^6*t^2-2*q^7+5*q^6*t-2*q^5*t^2-2*q^6+4*q^5*t-q^4*t^2-
↪2*q^5+2*q^4*t-q^4+q^3*t-q^3)/(-q^12*t^3+q^9*t^2+q^8*t^2+q^7*t^2-q^5*t-q^4*t-q^
↪3*t+1))*B[0]
- ((q^5*t-q^5)/(-q^5*t+1))*B[-4*Lambda[1]]
+ ((q^9*t^2-q^9*t+q^8*t^2-2*q^8*t+q^7*t^2+q^8-2*q^7*t+q^6*t^2+q^7-2*q^6*t+q^5*t^
↪2+q^6-2*q^5*t+q^5-q^4*t+q^4)/(q^9*t^2-q^5*t-q^4*t+1))*B[-2*Lambda[1]]
+ ((q^9*t^2-q^9*t+q^8*t^2-2*q^8*t+2*q^7*t^2+q^8-3*q^7*t+2*q^6*t^2+q^7-4*q^6*t+2*q^
↪5*t^2+2*q^6-4*q^5*t+q^4*t^2+2*q^5-3*q^4*t+q^3*t^2+2*q^4-2*q^3*t+q^3-q^2*t+q^2)/
↪(q^9*t^2-q^5*t-q^4*t+1))*B[2*Lambda[1]]
+ ((q^5*t-q^5+q^4*t-q^4+q^3*t-q^3+q^2*t-q^2+q*t-q)/(q^5*t-1))*B[4*Lambda[1]]
+ B[6*Lambda[1]]
sage: E.eigenvalues(_)
[q^5*t, 1/(q^6*t)]
sage: E[-omega[1]]
B[-Lambda[1]] - ((t-1)/(-q*t+1))*B[Lambda[1]]
sage: E.eigenvalues(_)
[(-1)/(-q^2*t), q*t]
```

```
>>> from sage.all import *
>>> K = QQ['q','t'].fraction_field()
>>> q,t = K.gens()
>>> KL = RootSystem(["A",Integer(1),Integer(1)]).weight_lattice(extended=True).
↪algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()

>>> x = E[Integer(0)*omega[Integer(1)]]; x
B[0]
>>> E.eigenvalues(x)
[1/(q*t), t]
>>> x.is_one()
True
>>> x.parent()
Algebra of the Weight lattice of the Root system of type ['A', 1]
over Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field
>>> E[omega[Integer(1)]]
B[Lambda[1]]
>>> E.eigenvalues(_)
[t, 1/(q*t)]
```

(continues on next page)

(continued from previous page)

```
>>> E[Integer(2)*omega[Integer(1)]]
-((q*t-q)/(-q*t+1))*B[0] + B[2*Lambda[1]]
>>> E.eigenvalues(_)
[q*t, 1/(q^2*t)]
>>> E[Integer(3)*omega[Integer(1)]]
-((q^2*t-q^2)/(-q^2*t+1))*B[-Lambda[1]]
- ((q^2*t-q^2+q*t-q)/(-q^2*t+1))*B[Lambda[1]] + B[3*Lambda[1]]
>>> E.eigenvalues(_)
[q^2*t, 1/(q^3*t)]
>>> E[Integer(4)*omega[Integer(1)]]
((q^5*t^2-q^5*t+q^4*t^2-2*q^4*t+q^3*t^2+q^4-2*q^3*t+q^3-q^2*t+q^2)/(q^5*t^2-q^3*t-
↪q^2*t+1))*B[0]
- ((q^3*t-q^3)/(-q^3*t+1))*B[-2*Lambda[1]]
- ((q^3*t-q^3+q^2*t-q^2+q*t-q)/(-q^3*t+1))*B[2*Lambda[1]]
+ B[4*Lambda[1]]
>>> E.eigenvalues(_)
[q^3*t, 1/(q^4*t)]
>>> E[Integer(6)*omega[Integer(1)]]
-((q^12*t^3-q^12*t^2+q^11*t^3-2*q^11*t^2+2*q^10*t^3+q^11*t-4*q^10*t^2+2*q^9*t^
↪3+2*q^10*t-5*q^9*t^2+2*q^8*t^3+4*q^9*t-6*q^8*t^2+q^7*t^3-q^9+5*q^8*t-5*q^7*t^
↪2+q^6*t^3-q^8+6*q^7*t-4*q^6*t^2-2*q^7+5*q^6*t-2*q^5*t^2-2*q^6+4*q^5*t-q^4*t^2-
↪2*q^5+2*q^4*t-q^4+q^3*t-q^3)/(-q^12*t^3+q^9*t^2+q^8*t^2+q^7*t^2-q^5*t-q^4*t-q^
↪3*t+1))*B[0]
- ((q^5*t-q^5)/(-q^5*t+1))*B[-4*Lambda[1]]
+ ((q^9*t^2-q^9*t+q^8*t^2-2*q^8*t+q^7*t^2+q^8-2*q^7*t+q^6*t^2+q^7-2*q^6*t+q^5*t^
↪2+q^6-2*q^5*t+q^5-q^4*t+q^4)/(q^9*t^2-q^5*t-q^4*t+1))*B[-2*Lambda[1]]
+ ((q^9*t^2-q^9*t+q^8*t^2-2*q^8*t+2*q^7*t^2+q^8-3*q^7*t+2*q^6*t^2+q^7-4*q^6*t+2*q^
↪5*t^2+2*q^6-4*q^5*t+q^4*t^2+2*q^5-3*q^4*t+q^3*t^2+2*q^4-2*q^3*t+q^3-q^2*t+q^2)/
↪(q^9*t^2-q^5*t-q^4*t+1))*B[2*Lambda[1]]
+ ((q^5*t-q^5+q^4*t-q^4+q^3*t-q^3+q^2*t-q^2+q*t-q)/(q^5*t-1))*B[4*Lambda[1]]
+ B[6*Lambda[1]]
>>> E.eigenvalues(_)
[q^5*t, 1/(q^6*t)]
>>> E[-omega[Integer(1)]]
B[-Lambda[1]] - ((t-1)/(-q*t+1))*B[Lambda[1]]
>>> E.eigenvalues(_)
[(-1)/(-q^2*t), q*t]
```

As expected, $e^{-\omega}$ is not an eigenvector:

```
sage: E.eigenvalues(KL.classical().monomial(-omega[1]))
Traceback (most recent call last):
...
AssertionError
```

```
>>> from sage.all import *
>>> E.eigenvalues(KL.classical().monomial(-omega[Integer(1)]))
Traceback (most recent call last):
...
AssertionError
```

We proceed by comparing against the examples from the appendix of [HHL06] in type $A_2^{(1)}$:

```
sage: K = QQ['q','t'].fraction_field()
sage: q,t = K.gens()
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
sage: P = K['x0,x1,x2']
sage: def EE(weight): return E[L0(weight)].expand(P.gens())

sage: EE([0,0,0])
1
sage: EE([1,0,0])
x0
sage: EE([0,1,0])
(t - 1)/(q*t^2 - 1)*x0 + x1
sage: EE([0,0,1])
(t - 1)/(q*t - 1)*x0 + (t - 1)/(q*t - 1)*x1 + x2
sage: EE([1,1,0])
x0*x1
sage: EE([1,0,1])
(t - 1)/(q*t^2 - 1)*x0*x1 + x0*x2
sage: EE([0,1,1])
(t - 1)/(q*t - 1)*x0*x1 + (t - 1)/(q*t - 1)*x0*x2 + x1*x2
sage: EE([2,0,0])
x0^2 + (q*t - q)/(q*t - 1)*x0*x1 + (q*t - q)/(q*t - 1)*x0*x2

sage: EE([0,2,0])
(t - 1)/(q^2*t^2 - 1)*x0^2
+ (q^2*t^3 - q^2*t^2 + q*t^2 - 2*q*t + q - t + 1)/(q^3*t^3 - q^2*t^2 - q*t +
↪1)*x0*x1
+ x1^2
+ (q*t^2 - 2*q*t + q)/(q^3*t^3 - q^2*t^2 - q*t + 1)*x0*x2
+ (q*t - q)/(q*t - 1)*x1*x2
```

```
>>> from sage.all import *
>>> K = QQ['q','t'].fraction_field()
>>> q,t = K.gens()
>>> KL = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, t, -Integer(1))
>>> L0 = E.keys()
>>> omega = L0.fundamental_weights()
>>> P = K['x0,x1,x2']
>>> def EE(weight): return E[L0(weight)].expand(P.gens())

>>> EE([Integer(0),Integer(0),Integer(0)])
1
>>> EE([Integer(1),Integer(0),Integer(0)])
x0
>>> EE([Integer(0),Integer(1),Integer(0)])
(t - 1)/(q*t^2 - 1)*x0 + x1
>>> EE([Integer(0),Integer(0),Integer(1)])
(t - 1)/(q*t - 1)*x0 + (t - 1)/(q*t - 1)*x1 + x2
```

(continues on next page)

(continued from previous page)

```
>>> EE([Integer(1),Integer(1),Integer(0)])
x0*x1
>>> EE([Integer(1),Integer(0),Integer(1)])
(t - 1)/(q*t^2 - 1)*x0*x1 + x0*x2
>>> EE([Integer(0),Integer(1),Integer(1)])
(t - 1)/(q*t - 1)*x0*x1 + (t - 1)/(q*t - 1)*x0*x2 + x1*x2
>>> EE([Integer(2),Integer(0),Integer(0)])
x0^2 + (q*t - q)/(q*t - 1)*x0*x1 + (q*t - q)/(q*t - 1)*x0*x2

>>> EE([Integer(0),Integer(2),Integer(0)])
(t - 1)/(q^2*t^2 - 1)*x0^2
+ (q^2*t^3 - q^2*t^2 + q*t^2 - 2*q*t + q - t + 1)/(q^3*t^3 - q^2*t^2 - q*t +␣
↪1)*x0*x1
+ x1^2
+ (q*t^2 - 2*q*t + q)/(q^3*t^3 - q^2*t^2 - q*t + 1)*x0*x2
+ (q*t - q)/(q*t - 1)*x1*x2
```

Systematic checks with Sage's implementation of [HHL06]:

```
sage: import sage.combinat.sf.ns_macdonald as NS                              #␣
↪needs sage.combinat
sage: assert all(EE([x,y,z]) == NS.E([x,y,z]) for d in range(5)          # long␣
↪time (9s), needs sage.combinat
....:                for x,y,z in IntegerVectors(d,3))
```

```
>>> from sage.all import *
>>> import sage.combinat.sf.ns_macdonald as NS                              #␣
↪needs sage.combinat
>>> assert all(EE([x,y,z]) == NS.E([x,y,z]) for d in range(Integer(5))          #␣
↪long time (9s), needs sage.combinat
...                for x,y,z in IntegerVectors(d,Integer(3)))
```

We check that we get eigenvectors for generic q_1, q_2:

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q,q1,q2 = K.gens()
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, q1, q2)
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
sage: E[2*omega[2]]
-((q*q1+q*q2)/(-q*q1-q2))*B[(1, 2, 1)] - ((q*q1+q*q2)/(-q*q1-q2))*B[(2, 1, 1)] +␣
↪B[(2, 2, 0)]
sage: for d in range(4):                          # long time (9s)
....:     for weight in IntegerVectors(d,3).map(list).map(L0):
....:         eigenvalues = E.eigenvalues(E[L0(weight)])
```

```
>>> from sage.all import *
>>> K = QQ['q,q1,q2'].fraction_field()
>>> q,q1,q2 = K.gens()
>>> KL = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL, q, q1, q2)
```

(continues on next page)

(continued from previous page)

```
>>> L0 = E.keys()
>>> omega = L0.fundamental_weights()
>>> E[Integer(2)*omega[Integer(2)]]
-((q*q1+q*q2)/(-q*q1-q2))*B[(1, 2, 1)] - ((q*q1+q*q2)/(-q*q1-q2))*B[(2, 1, 1)] +
↪B[(2, 2, 0)]
>>> for d in range(Integer(4)):                     # long time (9s)
...     for weight in IntegerVectors(d,Integer(3)).map(list).map(L0):
...         eigenvalues = E.eigenvalues(E[L0(weight)])
```

Some type C calculations:

```
sage: K = QQ['q','t'].fraction_field()
sage: q, t = K.gens()
sage: KL = RootSystem(["C",2,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
sage: E[0*omega[1]]
B[(0, 0)]
sage: E.eigenvalues(_)  # checked for i=0 with previous calculation
[1/(q*t^3), t, t]
sage: E[omega[1]]
B[(1, 0)]
sage: E.eigenvalues(_)  # not checked
[t, 1/(q*t^3), t]

sage: E[-omega[1]]            # consistent with before refactoring
B[(-1, 0)] - ((t-1)/(-q*t+1))*B[(1, 0)] - ((t-1)/(-q*t+1))*B[(0, -1)]
+ ((t-1)/(q*t-1))*B[(0, 1)]
sage: E.eigenvalues(_)  # not checked
[(-1)/(-q^2*t^3), q*t, t]
sage: E[-omega[1]+omega[2]]   # consistent with before refactoring
-((t-1)/(-q*t^3+1))*B[(1, 0)] + B[(0, 1)]
sage: E.eigenvalues(_)  # not checked
[t, q*t^3, (-1)/(-q*t^2)]
sage: E[omega[1]-omega[2]]    # consistent with before refactoring
-((t-1)/(-q*t^2+1))*B[(1, 0)] + B[(0, -1)] - ((t-1)/(-q*t^2+1))*B[(0, 1)]
sage: E.eigenvalues(_)  # not checked
[1/(q^2*t^3), 1/(q*t), q*t^2]

sage: E[-omega[2]]
-((q^2*t^4-q^2*t^3+q*t^3-2*q*t^2+q*t-t+1)/(-q^3*t^4+q^2*t^3+q*t-1))*B[(0, 0)]
+ B[(-1, -1)] - ((t-1)/(-q*t+1))*B[(-1, 1)] + ((t-1)/(q*t-1))*B[(1, -1)]
- ((q*t^4-q*t^3-t+1)/(-q^3*t^4+q^2*t^3+q*t-1))*B[(1, 1)]
sage: E.eigenvalues(_)  # not checked   # long time (1s)
[1/(q^3*t^3), t, q*t]
sage: E[-omega[2]].map_coefficients(lambda c: c.subs(t=0))     # checking against
↪crystals
B[(0, 0)] + B[(-1, -1)] + B[(-1, 1)] + B[(1, -1)] + B[(1, 1)]

sage: E[2*omega[2]]
-((q^6*t^7-q^6*t^6+q^5*t^6-2*q^5*t^5+q^4*t^5+q^5*t^3-3*q^4*t^4+3*q^4*t^3-q^3*t^4-
```

(continues on next page)

(continued from previous page)

```
↪q^4*t^2+2*q^3*t^2-q^3*t+q^2*t-q^2)/(-q^6*t^7+q^5*t^6+q^4*t^4+q^3*t^4-q^3*t^3-q^
↪2*t^3-q*t+1))*B[(0, 0)]
- ((q^3*t^2-q^3*t)/(-q^3*t^3+1))*B[(-1, -1)]
- ((q^3*t^3-2*q^3*t^2+q^3*t)/(-q^4*t^4+q^3*t^3+q*t-1))*B[(-1, 1)]
- ((q^3*t^3-2*q^3*t^2+q^3*t)/(-q^4*t^4+q^3*t^3+q*t-1))*B[(1, -1)]
- ((q^4*t^4-q^4*t^3+q^3*t^3-2*q^3*t^2+q^2*t^3+q^3*t-2*q^2*t^2+q^2*t-q*t+q)/(-q^
↪4*t^4+q^3*t^3+q*t-1))*B[(1, 1)]
+ ((q*t-q)/(q*t-1))*B[(2, 0)] + B[(2, 2)] - ((q*t-q)/(-q*t+1))*B[(0, 2)]
sage: E.eigenvalues(_)  # not checked
[q^3*t^3, t, (-1)/(-q^2*t^2)]
```

```
>>> from sage.all import *
>>> K = QQ['q','t'].fraction_field()
>>> q, t = K.gens()
>>> KL = RootSystem(["C",Integer(2),Integer(1)]).ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, t, -Integer(1))
>>> L0 = E.keys()
>>> omega = L0.fundamental_weights()
>>> E[Integer(0)*omega[Integer(1)]]
B[(0, 0)]
>>> E.eigenvalues(_)  # checked for i=0 with previous calculation
[1/(q*t^3), t, t]
>>> E[omega[Integer(1)]]
B[(1, 0)]
>>> E.eigenvalues(_)  # not checked
[t, 1/(q*t^3), t]

>>> E[-omega[Integer(1)]]            # consistent with before refactoring
B[(-1, 0)] - ((t-1)/(-q*t+1))*B[(1, 0)] - ((t-1)/(-q*t+1))*B[(0, -1)]
+ ((t-1)/(q*t-1))*B[(0, 1)]
>>> E.eigenvalues(_)  # not checked
[(-1)/(-q^2*t^3), q*t, t]
>>> E[-omega[Integer(1)]+omega[Integer(2)]]  # consistent with before refactoring
-((t-1)/(-q*t^3+1))*B[(1, 0)] + B[(0, 1)]
>>> E.eigenvalues(_)  # not checked
[t, q*t^3, (-1)/(-q*t^2)]
>>> E[omega[Integer(1)]-omega[Integer(2)]]   # consistent with before refactoring
-((t-1)/(-q*t^2+1))*B[(1, 0)] + B[(0, -1)] - ((t-1)/(-q*t^2+1))*B[(0, 1)]
>>> E.eigenvalues(_)  # not checked
[1/(q^2*t^3), 1/(q*t), q*t^2]

>>> E[-omega[Integer(2)]]
-((q^2*t^4-q^2*t^3+q*t^3-2*q*t^2+q*t-t+1)/(-q^3*t^4+q^2*t^3+q*t-1))*B[(0, 0)]
+ B[(-1, -1)] - ((t-1)/(-q*t+1))*B[(-1, 1)] + ((t-1)/(q*t-1))*B[(1, -1)]
- ((q*t^4-q*t^3-t+1)/(-q^3*t^4+q^2*t^3+q*t-1))*B[(1, 1)]
>>> E.eigenvalues(_)  # not checked   # long time (1s)
[1/(q^3*t^3), t, q*t]
>>> E[-omega[Integer(2)]].map_coefficients(lambda c: c.subs(t=Integer(0)))     #␣
↪checking against crystals
B[(0, 0)] + B[(-1, -1)] + B[(-1, 1)] + B[(1, -1)] + B[(1, 1)]

>>> E[Integer(2)*omega[Integer(2)]]
```

(continues on next page)

(continued from previous page)

```
-((q^6*t^7-q^6*t^6+q^5*t^6-2*q^5*t^5+q^4*t^5+q^5*t^3-3*q^4*t^4+3*q^4*t^3-q^3*t^4-
↪q^4*t^2+2*q^3*t^2-q^3*t+q^2*t-q^2)/(-q^6*t^7+q^5*t^6+q^4*t^4+q^3*t^4-q^3*t^3-q^
↪2*t^3-q*t+1))*B[(0, 0)]
- ((q^3*t^2-q^3*t)/(-q^3*t^3+1))*B[(-1, -1)]
- ((q^3*t^3-2*q^3*t^2+q^3*t)/(-q^4*t^4+q^3*t^3+q*t-1))*B[(-1, 1)]
- ((q^3*t^3-2*q^3*t^2+q^3*t)/(-q^4*t^4+q^3*t^3+q*t-1))*B[(1, -1)]
- ((q^4*t^4-q^4*t^3+q^3*t^3-2*q^3*t^2+q^2*t^3+q^3*t-2*q^2*t^2+q^2*t-q*t+q)/(-q^
↪4*t^4+q^3*t^3+q*t-1))*B[(1, 1)]
+ ((q*t-q)/(q*t-1))*B[(2, 0)] + B[(2, 2)] - ((q*t-q)/(-q*t+1))*B[(0, 2)]
>>> E.eigenvalues(_)  # not checked
[q^3*t^3, t, (-1)/(-q^2*t^2)]
```

The following computations were calculated by hand:

```
sage: KL0 = KL.classical()
sage: E11 = KL0.sum_of_terms([[L0([1,1]), 1], [L0([0,0]), (-q*t^2 + q*t)/(1-q*t^
↪3)]])
sage: E11 == E[omega[2]]
True
sage: E.eigenvalues(E11)
[q*t^3, t, (-1)/(-q*t^2)]

sage: E1m1 = KL0.sum_of_terms([[L0([1,-1]), 1], [L0([1,1]), (1-t)/(1-q*t^2)],
....:                            [L0([0,0]), q*t*(1-t)/(1-q*t^2)]])
sage: E1m1 == E[2*omega[1]-omega[2]]
True
sage: E.eigenvalues(E1m1)
[1/(q*t), 1/(q^2*t^3), q*t^2]
```

```
>>> from sage.all import *
>>> KL0 = KL.classical()
>>> E11 = KL0.sum_of_terms([[L0([Integer(1),Integer(1)]), Integer(1)],␣
↪[L0([Integer(0),Integer(0)]), (-q*t**Integer(2) + q*t)/(Integer(1)-
↪q*t**Integer(3))]])
>>> E11 == E[omega[Integer(2)]]
True
>>> E.eigenvalues(E11)
[q*t^3, t, (-1)/(-q*t^2)]

>>> E1m1 = KL0.sum_of_terms([[L0([Integer(1),-Integer(1)]), Integer(1)],␣
↪[L0([Integer(1),Integer(1)]), (Integer(1)-t)/(Integer(1)-q*t**Integer(2))],
...                            [L0([Integer(0),Integer(0)]), q*t*(Integer(1)-t)/
↪(Integer(1)-q*t**Integer(2))]])
>>> E1m1 == E[Integer(2)*omega[Integer(1)]-omega[Integer(2)]]
True
>>> E.eigenvalues(E1m1)
[1/(q*t), 1/(q^2*t^3), q*t^2]
```

Now we present an example for a twisted affine root system. The results are eigenvectors:

```
sage: K = QQ['q','t'].fraction_field()
sage: q, t = K.gens()
```

(continues on next page)

(continued from previous page)

```
sage: KL = RootSystem("C2~*").ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: E[0*omega[1]]
B[(0, 0)]
sage: E.eigenvalues(_)
[1/(q*t^2), t, t]
sage: E[omega[1]]
-((q*t-q)/(-q*t^2+1))*B[(0, 0)] + B[(1, 0)]
sage: E.eigenvalues(_)
[q*t^2, 1/(q^2*t^3), t]

sage: E[-omega[1]]
-((q*t-q+t-1)/(-q^2*t+1))*B[(0, 0)] + B[(-1, 0)] - ((t-1)/(-q^2*t+1))*B[(1, 0)]
- ((t-1)/(-q^2*t+1))*B[(0, -1)] + ((t-1)/(q^2*t-1))*B[(0, 1)]
sage: E.eigenvalues(_)
[(-1)/(-q^3*t^2), q^2*t, t]
sage: E[-omega[1]+omega[2]]
B[(-1/2, 1/2)] - ((t-1)/(-q^2*t^3+1))*B[(1/2, -1/2)]
- ((q*t^3-q*t^2+t-1)/(-q^2*t^3+1))*B[(1/2, 1/2)]
sage: E.eigenvalues(_)
[(-1)/(-q^2*t^2), q^2*t^3, (-1)/(-q*t)]
sage: E[omega[1]-omega[2]]
B[(1/2, -1/2)] - ((t-1)/(-q*t^2+1))*B[(1/2, 1/2)]
sage: E.eigenvalues(_)
[t, 1/(q^2*t^3), q*t^2]
```

```
>>> from sage.all import *
>>> K = QQ['q','t'].fraction_field()
>>> q, t = K.gens()
>>> KL = RootSystem("C2~*").ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, t, -Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> E[Integer(0)*omega[Integer(1)]]
B[(0, 0)]
>>> E.eigenvalues(_)
[1/(q*t^2), t, t]
>>> E[omega[Integer(1)]]
-((q*t-q)/(-q*t^2+1))*B[(0, 0)] + B[(1, 0)]
>>> E.eigenvalues(_)
[q*t^2, 1/(q^2*t^3), t]

>>> E[-omega[Integer(1)]]
-((q*t-q+t-1)/(-q^2*t+1))*B[(0, 0)] + B[(-1, 0)] - ((t-1)/(-q^2*t+1))*B[(1, 0)]
- ((t-1)/(-q^2*t+1))*B[(0, -1)] + ((t-1)/(q^2*t-1))*B[(0, 1)]
>>> E.eigenvalues(_)
[(-1)/(-q^3*t^2), q^2*t, t]
>>> E[-omega[Integer(1)]+omega[Integer(2)]]
B[(-1/2, 1/2)] - ((t-1)/(-q^2*t^3+1))*B[(1/2, -1/2)]
- ((q*t^3-q*t^2+t-1)/(-q^2*t^3+1))*B[(1/2, 1/2)]
>>> E.eigenvalues(_)
[(-1)/(-q^2*t^2), q^2*t^3, (-1)/(-q*t)]
```

(continues on next page)

(continued from previous page)

```
>>> E[omega[Integer(1)]-omega[Integer(2)]]
B[(1/2, -1/2)] - ((t-1)/(-q*t^2+1))*B[(1/2, 1/2)]
>>> E.eigenvalues(_)
[t, 1/(q^2*t^3), q*t^2]
```

Type BC, comparison with calculations with Maple by Bogdan Ion:

```
sage: K = QQ['q','t'].fraction_field()
sage: q,t = K.gens()
sage: def to_SR(x):
....:     dim = x.parent().basis().keys().dimension()
....:     x_expanded = x.expand([SR.var('x%s'%i) for i in range(1, dim + 1)])
....:     return x_expanded.subs(q=SR.var('q'), t=SR.var('t'))
sage: var('x1,x2,x3')                                                    #␣
↪needs sage.symbolic
(x1, x2, x3)

sage: E = NonSymmetricMacdonaldPolynomials(["BC",2,2], q=q, q1=t^2, q2=-1)
sage: omega = E.keys().fundamental_weights()
sage: expected = (t-1)*(t+1)*(2+q^4+2*q^2-2*t^2-2*q^2*t^2-t^4*q^2-q^4*t^4+t^4-3*q^
↪6*t^6-2*q^4*t^6+2*q^6*t^8+2*q^4*t^8+t^10*q^8)*q^4/((q^2*t^3-1)*(q^2*t^3+1)*(t*q-
↪1)*(t*q+1)*(t^2*q^3+1)*(t^2*q^3-1))+(t-1)^2*(t+1)^2*(2*q^2+q^4+2+q^4*t^2)*q^
↪3*x1/((t^2*q^3+1)*(t^2*q^3-1)*(t*q-1)*(t*q+1))+(t-1)^2*(t+1)^2*(q^2+1)*q^5/((t^
↪2*q^3+1)*(t^2*q^3-1)*(t*q-1)*(t*q+1)*x1)+(t-1)^2*(t+1)^2*(q^2+1)*q^4*x2/((t^2*q^
↪3+1)*(t^2*q^3-1)*(t*q-1)*(t*q+1)*x1)+(t-1)^2*(t+1)^2*(2*q^2+q^4+2+q^4*t^2)*q^
↪3*x2/((t^2*q^3+1)*(t^2*q^3-1)*(t*q-1)*(t*q+1))+(t-1)^2*(t+1)^2*(q^2+1)*q^5/((t^
↪2*q^3+1)*(t^2*q^3-1)*(t*q-1)*(t*q+1)*x2)+x1^2*x2^2+(t-1)*(t+1)*(-2*q^2-q^4-
↪2+2*q^2*t^2+t^2+q^6*t^4+q^4*t^4)*q^2*x2*x1/((t^2*q^3+1)*(t^2*q^3-1)*(t*q-
↪1)*(t*q+1))+(t-1)*(t+1)*(q^2+1+q^4*t^2)*q*x2^2*x1/((t^2*q^3-1)*(t^2*q^3+1))+(t-
↪1)*(t+1)*q^3*x1^2/((t^2*q^3-1)*(t^2*q^3+1)*x2)+(t-1)*(t+1)*(q^2+1+q^4*t^
↪2)*q*x2*x1^2/((t^2*q^3-1)*(t^2*q^3+1))+(t-1)*(t+1)*q^6/((t^2*q^3+1)*(t^2*q^3-
↪1)*x1*x2)+(t-1)*(t+1)*(q^2+1+q^4*t^2)*q^2*x1^2/((t^2*q^3-1)*(t^2*q^3+1))+(t-
↪1)*(t+1)*(q^2+1+q^4*t^2)*q^2*x2^2/((t^2*q^3-1)*(t^2*q^3+1))+(t-1)*(t+1)*q^3*x2^
↪2/((t^2*q^3-1)*(t^2*q^3+1)*x1)+(t-1)^2*(t+1)^2*(q^2+1)*q^4*x1/((t^2*q^3+1)*(t^
↪2*q^3-1)*(t*q-1)*(t*q+1)*x2)   # needs sage.symbolic
sage: to_SR(E[2*omega[2]]) - expected   # long time (3.5s)              #␣
↪needs sage.symbolic
0

sage: E = NonSymmetricMacdonaldPolynomials(["BC",3,2], q=q, q1=t^2, q2=-1)
sage: omega=E.keys().fundamental_weights()
sage: mu = -3*omega[1] + 3*omega[2] - omega[3]; mu
(-1, 2, -1)
sage: expected = (t-1)^2*(t+1)^2*(3*q^2+q^4+1+t^2*q^4+q^2*t^2-3*t^4*q^2-5*t^6*q^
↪4+2*t^8*q^4-4*t^8*q^6-q^8*t^10+2*t^10*q^6-2*q^8*t^12+t^14*q^8-t^14*q^10+q^10*t^
↪16+q^8*t^16+q^10*t^18+t^18*q^12)*x2*x1/((q^3*t^5+1)*(q^3*t^5-1)*(t*q-
↪1)*(t*q+1)*(t^3*q^2+1)*(t^3*q^2-1)*(t^2*q-1)*(t^2*q+1))+(t-1)^2*(t+1)^2*(q^2*t^
↪6+2*t^6*q^4-q^4*t^4+t^4*q^2-q^2*t^2+t^2-2-q^2)*q^2*x1/((t^3*q^2-1)*(t^3*q^
↪2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x2)+(t-1)^2*(t+1)^2*(-q^2-1+t^4*q^
↪2-q^4*t^4+2*t^6*q^4)*x1^2/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-
↪1)*(q^3*t^5+1))+(t+1)*(t-1)*x2^2*x3/((t*q-1)*(t*q+1)*x1)+(t-1)^2*(t+1)^2*(3*q^
↪2+q^4+2+t^2*q^4+2*q^2*t^2-4*t^4*q^2+q^4*t^4-6*t^6*q^4+t^8*q^4-4*t^8*q^6-q^8*t^
```

(continues on next page)

(continued from previous page)

```
↪10+t^10*q^6-3*q^8*t^12-2*t^14*q^10+2*t^14*q^8+2*q^10*t^16+q^8*t^16+t^18*q^
↪12+2*q^10*t^18)*q*x2/((q^3*t^5+1)*(q^3*t^5-1)*(t*q-1)*(t*q+1)*(t^3*q^2+1)*(t^
↪3*q^2-1)*(t^2*q-1)*(t^2*q+1))+(t-1)^2*(t+1)^2*(1+q^4+2*q^2+t^2*q^4-3*t^4*q^2+q^
↪2*t^6-5*t^6*q^4+3*t^8*q^4-4*t^8*q^6+2*t^10*q^6-q^8*t^12-t^14*q^10+t^14*q^8+q^
↪10*t^16+t^18*q^12)*x3*x1/((q^3*t^5+1)*(q^3*t^5-1)*(t*q-1)*(t*q+1)*(t^3*q^
↪2+1)*(t^3*q^2-1)*(t^2*q-1)*(t^2*q+1))+(t-1)^2*(t+1)^2*(2*q^2+1+q^4+t^2*q^4-t^
↪2+q^2*t^2-4*t^4*q^2+q^4*t^4+q^2*t^6-5*t^6*q^4+3*t^8*q^4-4*t^8*q^6+2*t^10*q^6+q^
↪6*t^12-2*q^8*t^12-2*t^14*q^10+2*t^14*q^8+q^10*t^16+t^18*q^12)*q*x3/((q^3*t^
↪5+1)*(q^3*t^5-1)*(t*q-1)*(t*q+1)*(t^3*q^2+1)*(t^3*q^2-1)*(t^2*q-1)*(t^
↪2*q+1))+(t-1)^2*(t+1)^2*(1+t^2+t^4*q^2)*q*x3*x2^2/((t*q-1)*(t*q+1)*(t^3*q^
↪2+1)*(t^3*q^2-1))+(t-1)^2*(t+1)^2*(-q^2-2-q^2*t^2+t^4-q^4*t^4-t^4*q^2+3*q^2*t^6-
↪t^6*q^4-t^8*q^6+t^8*q^4+t^10*q^4+2*q^6*t^12-q^8*t^12+t^14*q^8)*q*x3*x2*x1/((t^
↪3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1))+(t-1)*(t+1)*x1^2/
↪((q^3*t^5-1)*(q^3*t^5+1)*x3*x2)+(t-1)*(t+1)*(-q^2-1+t^4*q^2-q^4*t^4+2*t^6*q^
↪4)*x2^2/((t*q-1)*(t*q+1)*(t^3*q^2+1)*(t^3*q^2-1))+(t-1)*(t+1)*(t^3*q-1)*(t^
↪3*q+1)*x3*x2^2*x1/((t*q-1)*(t*q+1)*(t^3*q^2+1)*(t^3*q^2-1))+(t-1)^2*(t+1)^2*(q^
↪2+1)*q*x1/((t*q+1)*(t*q-1)*(q^3*t^5+1)*(q^3*t^5-1)*x3*x2)+(t-1)^2*(t+1)^2*(t^
↪3*q-1)*(t^3*q+1)*x3*x2*x1^2/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-
↪1)*(q^3*t^5+1))+(t-1)^2*(t+1)^2*q^3*x3/((t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^
↪5+1)*x1*x2)+(t-1)*(t+1)*(-1-q^2+q^2*t^2+t^10*q^6)*q*x2/((t*q+1)*(t*q-1)*(q^3*t^
↪5+1)*(q^3*t^5-1)*x3*x1)+x2^2/(x1*x3)+(t-1)*(t+1)*q*x2^2/((t*q-1)*(t*q+1)*x3)+(t-
↪1)^3*(t+1)^3*(1+t^2+t^4*q^2)*q*x2*x1^2/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-
↪1)*(q^3*t^5-1)*(q^3*t^5+1))+(t-1)^2*(t+1)^2*q*x1^2/((t*q+1)*(t*q-1)*(q^3*t^5-
↪1)*(q^3*t^5+1)*x3)+(t-1)^2*(t+1)^2*(q^2*t^6+2*t^6*q^4-q^4*t^4+t^4*q^2-q^2*t^2+t^
↪2-2-q^2)*q^3/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^
↪5+1)*x2)+(t-1)*(t+1)*(q^2+2-t^2+q^4*t^4-t^4*q^2-3*t^6*q^4+t^8*q^4-2*t^10*q^6-q^
↪8*t^12+q^6*t^12+q^8*t^16+q^10*t^16)*q^2*x2/((t^3*q^2-1)*(t^3*q^
↪2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x1)+(t-1)^2*(t+1)^2*(q^2+1)*q^2/
↪((t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x3*x2)+(t-1)*(t+1)*(1+q^4+2*q^2-2*q^
↪2*t^2+t^4*q^6-q^4*t^4-3*q^6*t^6-t^6*q^4+2*t^8*q^6-t^10*q^6-q^8*t^10-t^14*q^10+t^
↪14*q^8+2*q^10*t^16)*x2/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^
↪3*t^5+1)*x3)+(t-1)^2*(t+1)^2*(-q^2-2-q^2*t^2-q^4*t^4+2*t^6*q^4+t^10*q^6+q^8*t^
↪12+t^14*q^8)*q^3/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^
↪5+1)*x1)+(t-1)^2*(t+1)^2*(-1-q^2-q^2*t^2+t^2+t^4*q^2-q^4*t^4+2*t^6*q^4)*q^2*x3/
↪((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x2)+(t-
↪1)*(t+1)*q*x2^2/((t*q-1)*(t*q+1)*x1)+(t-1)^2*(t+1)^2*(1+t^2+t^4*q^2)*q*x2^2*x1/
↪((t*q-1)*(t*q+1)*(t^3*q^2+1)*(t^3*q^2-1))+(t-1)^2*(t+1)^2*q*x1^2/((t*q+1)*(t*q-
↪1)*(q^3*t^5-1)*(q^3*t^5+1)*x2)+(t-1)^2*(t+1)^2*(-1-q^4-2*q^2-t^2*q^4-q^2*t^2+t^
↪4*q^2-t^4*q^6-2*q^4*t^4+3*t^6*q^4-q^6*t^6-t^8*q^8+t^8*q^6+2*t^10*q^6-q^10*t^
↪12+3*q^8*t^12+2*t^14*q^10)*x3*x2/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^
↪3*t^5-1)*(q^3*t^5+1))+(t-1)*(t+1)*(q^2+1-t^2+q^4*t^4-t^4*q^2+q^2*t^6-3*t^6*q^
↪4+t^8*q^4-t^10*q^6+q^6*t^12-q^8*t^12+q^10*t^16)*q^2*x3/((t^3*q^2-1)*(t^3*q^
↪2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x1)+(t-1)*(t+1)*(-1-q^2+q^2*t^2+t^
↪10*q^6)*q^2/((t*q-1)*(t*q+1)*(q^3*t^5+1)*(q^3*t^5-1)*x1*x3)+(t-1)*(t+1)*(1+q^
↪4+2*q^2-3*q^2*t^2+t^4*q^6-q^4*t^4-3*q^6*t^6-t^6*q^4+t^8*q^4+2*t^8*q^6-t^10*q^
↪6+t^14*q^8-t^14*q^10+q^10*t^16)*x1/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^
↪3*t^5-1)*(q^3*t^5+1)*x3)+(t-1)^2*(t+1)^2*(3*q^2+q^4+2+q^2*t^2-t^2+t^2*q^4-6*t^
↪4*q^2+q^4*t^4-7*t^6*q^4+q^2*t^6+3*t^8*q^4-4*t^8*q^6+t^10*q^4+3*t^10*q^6-q^8*t^
↪12-t^14*q^10+t^14*q^8+q^8*t^16+q^10*t^18)*q*x1/((q^3*t^5+1)*(q^3*t^5-1)*(t*q-
↪1)*(t*q+1)*(t^3*q^2+1)*(t^3*q^2-1)*(t^2*q-1)*(t^2*q+1))+(t-1)^2*(t+1)^2*(-q^2-2-
↪q^2*t^2-q^4*t^4+2*t^6*q^4+t^10*q^6+q^6*t^12+t^14*q^8)*q*x2*x1/((t^3*q^2-1)*(t^
↪3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x3)+(t+1)*(t-1)*x2^2*x1/((t*q-
```

(continues on next page)

(continued from previous page)

```
↪1)*(t*q+1)*x3)+(t-1)^3*(t+1)^3*(1+t^2+t^4*q^2)*q*x3*x1^2/((t^3*q^2-1)*(t^3*q^
↪2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1))+(t-1)*(t+1)*q^3/((q^3*t^5+1)*(q^
↪3*t^5-1)*x1*x2*x3)+(t-1)^2*(t+1)^2*(3+3*q^2+q^4+2*q^2*t^2-t^2+t^2*q^4-6*t^4*q^
↪2+q^4*t^4-8*t^6*q^4+q^2*t^6+2*t^8*q^4-4*t^8*q^6+t^10*q^4+2*t^10*q^6-2*q^8*t^12-
↪t^14*q^10+t^14*q^8+q^8*t^16+q^10*t^16+2*q^10*t^18)*q^2/((q^3*t^5+1)*(q^3*t^5-
↪1)*(t*q-1)*(t*q+1)*(t^3*q^2+1)*(t^3*q^2-1)*(t^2*q-1)*(t^2*q+1))+(t-1)^2*(t+1)^
↪2*(-q^4-2*q^2-1-t^2*q^4-t^4*q^6+2*q^6*t^6+t^6*q^4+t^10*q^6+q^8*t^12+t^14*q^
↪10)*q/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x3)+(t-
↪1)^2*(t+1)^2*(-1-q^2-q^2*t^2+t^2+t^4*q^2-q^4*t^4+2*t^6*q^4)*q*x3*x1/((t^3*q^2-
↪1)*(t^3*q^2+1)*(t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x2)+(t-1)^2*(t+1)^
↪2*x2*x1^2/((t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x3)+(t-1)^2*(t+1)^2*x3*x1^2/
↪((t*q+1)*(t*q-1)*(q^3*t^5-1)*(q^3*t^5+1)*x2)+(t-1)^2*(t+1)^2*q^4/((t*q+1)*(t*q-
↪1)*(q^3*t^5+1)*(q^3*t^5-1)*x1*x2)+(t-1)^2*(t+1)^2*(-q^2-1-q^2*t^2-q^4*t^4+t^6*q^
↪4+t^10*q^6+q^8*t^12+t^14*q^10)*q*x3*x2/((t^3*q^2-1)*(t^3*q^2+1)*(t*q+1)*(t*q-
↪1)*(q^3*t^5-1)*(q^3*t^5+1)*x1)  # needs sage.symbolic
sage: to_SR(E[mu]) - expected          # long time (20s)                    #␣
↪needs sage.symbolic
0

sage: E = NonSymmetricMacdonaldPolynomials(["BC",1,2], q=q, q1=t^2, q2=-1)
sage: omega = E.keys().fundamental_weights()
sage: mu = -4*omega[1]; mu
(-4)
sage: expected = (t-1)*(t+1)*(-1+q^2*t^2-q^2-3*q^10-7*q^26*t^8+5*t^2*q^6-q^16-3*q^
↪4+4*t^10*q^30-4*t^6*q^22-10*q^20*t^6+2*q^32*t^10-3*q^6-4*q^8+q^34*t^10-4*t^8*q^
↪24-2*q^12-q^14+2*q^22*t^10+4*q^26*t^10+4*q^28*t^10+t^6*q^30-2*q^32*t^8-2*t^8*q^
↪22+2*q^24*t^10-q^20*t^2-2*t^6*q^12+t^8*q^14+2*t^4*q^24-4*t^8*q^30+2*t^8*q^20-
↪9*t^6*q^16+3*q^26*t^6+q^28*t^6+3*t^2*q^4+2*q^18*t^8-6*t^6*q^14+4*t^4*q^22-2*q^
↪24*t^6+3*t^2*q^12+7*t^4*q^20-t^2*q^16+11*q^18*t^4-2*t^2*q^18+9*q^16*t^4-t^4*q^
↪6+6*q^8*t^2+5*q^10*t^2-6*q^28*t^8+q^12*t^4+8*t^4*q^14-10*t^6*q^18-q^4*t^4+q^
↪16*t^8-2*t^4*q^8)/((t*q^4-1)*(t*q^4+1)*(q^7*t^2-1)*(q^7*t^2+1)*(t*q^3-1)*(t*q^
↪3+1)*(q^5*t^2+1)*(q^5*t^2-1))+(q^2+1)*(q^4+1)*(t-1)*(t+1)*(-1+q^2*t^2-q^2+t^2*q^
↪6-q^4+t^6*q^22+3*q^10*t^4+t^2-q^8-2*t^8*q^24+q^22*t^10+q^26*t^10-2*t^8*q^22+q^
↪24*t^10-4*t^6*q^12-2*t^8*q^20-3*t^6*q^16+2*t^2*q^4-t^6*q^10-2*t^6*q^14+t^8*q^12-
↪t^2*q^12+2*q^16*t^4+q^8*t^2-q^10*t^2+3*q^12*t^4+2*t^4*q^14+t^6*q^18-2*q^4*t^4+q^
↪16*t^8+q^20*t^10)*q*x1/((t*q^4-1)*(t*q^4+1)*(q^7*t^2-1)*(q^7*t^2+1)*(t*q^3-
↪1)*(t*q^3+1)*(q^5*t^2+1)*(q^5*t^2-1))+(q^2+1)*(q^4+1)*(t-1)*(t+1)*(1+q^8+q^4+q^
↪2-q^8*t^2-2*t^2*q^4-t^2*q^6+t^2*q^12-t^2+t^4*q^6-2*q^16*t^4-t^4*q^14-2*q^12*t^
↪4+t^6*q^12+t^6*q^16+t^6*q^18+t^6*q^14)*q/((t*q^4-1)*(t*q^4+1)*(q^7*t^2-1)*(q^
↪7*t^2+1)*(t*q^3-1)*(t*q^3+1)*x1)+(t-1)*(t+1)*(-1-q^2-q^6-q^4-q^8+t^2*q^4-t^2*q^
↪14+t^2*q^6-q^10*t^2+q^8*t^2-t^2*q^12+q^12*t^4+q^10*t^4+q^16*t^4+2*t^4*q^14)*(q^
↪4+1)/((q^7*t^2+1)*(q^7*t^2-1)*(t*q^4-1)*(t*q^4+1)*x1^2)+(t-1)*(t+1)*(q^4+1)*(q^
↪2+1)*q/((t*q^4-1)*(t*q^4+1)*x1^3)+(q^4+1)*(t-1)*(t+1)*(1+q^6+q^8+q^2+q^4-q^2*t^
↪2-3*t^2*q^4+q^10*t^2+t^2*q^12-2*t^2*q^6-q^8*t^2-2*q^16*t^4+q^4*t^4+t^4*q^6-q^
↪10*t^4-2*q^12*t^4-2*t^4*q^14+t^6*q^12+t^6*q^18+2*t^6*q^16+t^6*q^14)*x1^2/((t*q^
↪4-1)*(t*q^4+1)*(q^7*t^2-1)*(q^7*t^2+1)*(t*q^3-1)*(t*q^3+1))+(t-1)*(t+1)*(-1-t^
↪2*q^6+t^2+t^4*q^8)*(q^4+1)*(q^2+1)*q*x1^3/((q^7*t^2+1)*(q^7*t^2-1)*(t*q^4-
↪1)*(t*q^4+1))+1/x1^4+(t-1)*(t+1)*x1^4/((t*q^4-1)*(t*q^4+1))  # needs sage.
↪symbolic
sage: to_SR(E[mu]) - expected                                               #␣
↪needs sage.symbolic
0
```

```
>>> from sage.all import *
>>> K = QQ['q','t'].fraction_field()
>>> q,t = K.gens()
>>> def to_SR(x):
...     dim = x.parent().basis().keys().dimension()
...     x_expanded = x.expand([SR.var('x%s'%i) for i in range(Integer(1), dim +␣
↪Integer(1))])
...     return x_expanded.subs(q=SR.var('q'), t=SR.var('t'))
>>> var('x1,x2,x3')                                                      #␣
↪needs sage.symbolic
(x1, x2, x3)

>>> E = NonSymmetricMacdonaldPolynomials(["BC",Integer(2),Integer(2)], q=q,␣
↪q1=t**Integer(2), q2=-Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> expected = (t-Integer(1))*(t+Integer(1))*(Integer(2)+q**Integer(4)+Integer(2)*
↪q**Integer(2)-Integer(2)*t**Integer(2)-Integer(2)*q**Integer(2)*t**Integer(2)-t*
↪*Integer(4)*q**Integer(2)-q**Integer(4)*t**Integer(4)+t**Integer(4)-Integer(3)*q
↪**Integer(6)*t**Integer(6)-Integer(2)*q**Integer(4)*t**Integer(6)+Integer(2)*q**
↪Integer(6)*t**Integer(8)+Integer(2)*q**Integer(4)*t**Integer(8)+t**Integer(10)*q
↪**Integer(8))*q**Integer(4)/((q**Integer(2)*t**Integer(3)-Integer(1))*(q**Intege
↪r(2)*t**Integer(3)+Integer(1))*(t*q-Integer(1))*(t*q+Integer(1))*(t**Integer(2)*
↪q**Integer(3)+Integer(1))*(t**Integer(2)*q**Integer(3)-Integer(1)))+(t-Integer(1
↪))**Integer(2)*(t+Integer(1))**Integer(2)*(Integer(2)*q**Integer(2)+q**Integer(4
↪)+Integer(2)+q**Integer(4)*t**Integer(2))*q**Integer(3)*x1/((t**Integer(2)*q**In
↪teger(3)+Integer(1))*(t**Integer(2)*q**Integer(3)-Integer(1))*(t*q-Integer(1))*(
↪t*q+Integer(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(q**Integ
↪er(2)+Integer(1))*q**Integer(5)/((t**Integer(2)*q**Integer(3)+Integer(1))*(t**In
↪teger(2)*q**Integer(3)-Integer(1))*(t*q-Integer(1))*(t*q+Integer(1))*x1)+(t-Inte
↪ger(1))**Integer(2)*(t+Integer(1))**Integer(2)*(q**Integer(2)+Integer(1))*q**Int
↪eger(4)*x2/((t**Integer(2)*q**Integer(3)+Integer(1))*(t**Integer(2)*q**Integer(3
↪)-Integer(1))*(t*q-Integer(1))*(t*q+Integer(1))*x1)+(t-Integer(1))**Integer(2)*(
↪t+Integer(1))**Integer(2)*(Integer(2)*q**Integer(2)+q**Integer(4)+Integer(2)+q**
↪Integer(4)*t**Integer(2))*q**Integer(3)*x2/((t**Integer(2)*q**Integer(3)+Integer
↪(1))*(t**Integer(2)*q**Integer(3)-Integer(1))*(t*q-Integer(1))*(t*q+Integer(1)))
↪+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(q**Integer(2)+Integer(1)
↪)*q**Integer(5)/((t**Integer(2)*q**Integer(3)+Integer(1))*(t**Integer(2)*q**Inte
↪ger(3)-Integer(1))*(t*q-Integer(1))*(t*q+Integer(1))*x2)+x1**Integer(2)*x2**Inte
↪ger(2)+(t-Integer(1))*(t+Integer(1))*(-Integer(2)*q**Integer(2)-q**Integer(4)-In
↪teger(2)+Integer(2)*q**Integer(2)*t**Integer(2)+t**Integer(2)+q**Integer(6)*t**I
↪nteger(4)+q**Integer(4)*t**Integer(4))*q**Integer(2)*x2*x1/((t**Integer(2)*q**In
↪teger(3)+Integer(1))*(t**Integer(2)*q**Integer(3)-Integer(1))*(t*q-Integer(1))*(
↪t*q+Integer(1)))+(t-Integer(1))*(t+Integer(1))*(q**Integer(2)+Integer(1)+q**Inte
↪ger(4)*t**Integer(2))*q*x2**Integer(2)*x1/((t**Integer(2)*q**Integer(3)-Integer(
↪1))*(t**Integer(2)*q**Integer(3)+Integer(1)))+(t-Integer(1))*(t+Integer(1))*q**I
↪nteger(3)*x1**Integer(2)/((t**Integer(2)*q**Integer(3)-Integer(1))*(t**Integer(2
↪)*q**Integer(3)+Integer(1))*x2)+(t-Integer(1))*(t+Integer(1))*(q**Integer(2)+Int
↪eger(1)+q**Integer(4)*t**Integer(2))*q*x2*x1**Integer(2)/((t**Integer(2)*q**Inte
↪ger(3)-Integer(1))*(t**Integer(2)*q**Integer(3)+Integer(1)))+(t-Integer(1))*(t+I
↪nteger(1))*q**Integer(6)/((t**Integer(2)*q**Integer(3)+Integer(1))*(t**Integer(2
↪)*q**Integer(3)-Integer(1))*x1*x2)+(t-Integer(1))*(t+Integer(1))*(q**Integer(2)+
↪Integer(1)+q**Integer(4)*t**Integer(2))*q**Integer(2)*x1**Integer(2)/((t**Intege
↪r(2)*q**Integer(3)-Integer(1))*(t**Integer(2)*q**Integer(3)+Integer(1)))+(t-Inte
```

(continues on next page)

(continued from previous page)

```
↪ger(1))*(t+Integer(1))*(q**Integer(2)+Integer(1)+q**Integer(4)*t**Integer(2))*q*
↪*Integer(2)*x2**Integer(2)/((t**Integer(2)*q**Integer(3)-Integer(1))*(t**Integer
↪(2)*q**Integer(3)+Integer(1)))+(t-Integer(1))*(t+Integer(1))*q**Integer(3)*x2**I
↪nteger(2)/((t**Integer(2)*q**Integer(3)-Integer(1))*(t**Integer(2)*q**Integer(3)
↪+Integer(1))*x1)+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(q**Integ
↪er(2)+Integer(1))*q**Integer(4)*x1/((t**Integer(2)*q**Integer(3)+Integer(1))*(t*
↪*Integer(2)*q**Integer(3)-Integer(1))*(t*q-Integer(1))*(t*q+Integer(1))*x2)   # n
↪eeds sage.symbolic
>>> to_SR(E[Integer(2)*omega[Integer(2)]]) - expected   # long time (3.5s)       ␣
↪               # needs sage.symbolic
0

>>> E = NonSymmetricMacdonaldPolynomials(["BC",Integer(3),Integer(2)], q=q,␣
↪q1=t**Integer(2), q2=-Integer(1))
>>> omega=E.keys().fundamental_weights()
>>> mu = -Integer(3)*omega[Integer(1)] + Integer(3)*omega[Integer(2)] -␣
↪omega[Integer(3)]; mu
(-1, 2, -1)
>>> expected = (t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(Integer(3)*q
↪**Integer(2)+q**Integer(4)+Integer(1)+t**Integer(2)*q**Integer(4)+q**Integer(2)*
↪t**Integer(2)-Integer(3)*t**Integer(4)*q**Integer(2)-Integer(5)*t**Integer(6)*q*
↪*Integer(4)+Integer(2)*t**Integer(8)*q**Integer(4)-Integer(4)*t**Integer(8)*q**I
↪nteger(6)-q**Integer(8)*t**Integer(10)+Integer(2)*t**Integer(10)*q**Integer(6)-I
↪nteger(2)*q**Integer(8)*t**Integer(12)+t**Integer(14)*q**Integer(8)-t**Integer(1
↪4)*q**Integer(10)+q**Integer(10)*t**Integer(16)+q**Integer(8)*t**Integer(16)+q**
↪Integer(10)*t**Integer(18)+t**Integer(18)*q**Integer(12))*x2*x1/((q**Integer(3)*
↪t**Integer(5)+Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(t*q-Integer(
↪1))*(t*q+Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t**Integer(3)*q**
↪Integer(2)-Integer(1))*(t**Integer(2)*q-Integer(1))*(t**Integer(2)*q+Integer(1))
↪)+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(q**Integer(2)*t**Intege
↪r(6)+Integer(2)*t**Integer(6)*q**Integer(4)-q**Integer(4)*t**Integer(4)+t**Integ
↪er(4)*q**Integer(2)-q**Integer(2)*t**Integer(2)+t**Integer(2)-Integer(2)-q**Inte
↪ger(2))*q**Integer(2)*x1/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3
↪)*q**Integer(2)+Integer(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**
↪Integer(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x2)+(t-Integer(1
↪))**Integer(2)*(t+Integer(1))**Integer(2)*(-q**Integer(2)-Integer(1)+t**Integer(
↪4)*q**Integer(2)-q**Integer(4)*t**Integer(4)+Integer(2)*t**Integer(6)*q**Integer
↪(4))*x1**Integer(2)/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**
↪Integer(2)+Integer(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integ
↪er(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1)))+(t+Integer(1))*(t-In
↪teger(1))*x2**Integer(2)*x3/((t*q-Integer(1))*(t*q+Integer(1))*x1)+(t-Integer(1)
↪)**Integer(2)*(t+Integer(1))**Integer(2)*(Integer(3)*q**Integer(2)+q**Integer(4)
↪+Integer(2)+t**Integer(2)*q**Integer(4)+Integer(2)*q**Integer(2)*t**Integer(2)-I
↪nteger(4)*t**Integer(4)*q**Integer(2)+q**Integer(4)*t**Integer(4)-Integer(6)*t**
↪Integer(6)*q**Integer(4)+t**Integer(8)*q**Integer(4)-Integer(4)*t**Integer(8)*q*
↪*Integer(6)-q**Integer(8)*t**Integer(10)+t**Integer(10)*q**Integer(6)-Integer(3)
↪*q**Integer(8)*t**Integer(12)-Integer(2)*t**Integer(14)*q**Integer(10)+Integer(2
↪)*t**Integer(14)*q**Integer(8)+Integer(2)*q**Integer(10)*t**Integer(16)+q**Integ
↪er(8)*t**Integer(16)+t**Integer(18)*q**Integer(12)+Integer(2)*q**Integer(10)*t**
↪Integer(18))*q*x2/((q**Integer(3)*t**Integer(5)+Integer(1))*(q**Integer(3)*t**In
↪teger(5)-Integer(1))*(t*q-Integer(1))*(t*q+Integer(1))*(t**Integer(3)*q**Integer
↪(2)+Integer(1))*(t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(2)*q-Intege
```

(continues on next page)

(continued from previous page)

```
↪r(1))*(t**Integer(2)*q+Integer(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**I
↪nteger(2)*(Integer(1)+q**Integer(4)+Integer(2)*q**Integer(2)+t**Integer(2)*q**In
↪teger(4)-Integer(3)*t**Integer(4)*q**Integer(2)+q**Integer(2)*t**Integer(6)-Inte
↪ger(5)*t**Integer(6)*q**Integer(4)+Integer(3)*t**Integer(8)*q**Integer(4)-Intege
↪r(4)*t**Integer(8)*q**Integer(6)+Integer(2)*t**Integer(10)*q**Integer(6)-q**Inte
↪ger(8)*t**Integer(12)-t**Integer(14)*q**Integer(10)+t**Integer(14)*q**Integer(8)
↪+q**Integer(10)*t**Integer(16)+t**Integer(18)*q**Integer(12))*x3*x1/((q**Integer
↪(3)*t**Integer(5)+Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(t*q-Inte
↪ger(1))*(t*q+Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t**Integer(3)
↪*q**Integer(2)-Integer(1))*(t**Integer(2)*q-Integer(1))*(t**Integer(2)*q+Integer
↪(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(Integer(2)*q**Integ
↪er(2)+Integer(1)+q**Integer(4)+t**Integer(2)*q**Integer(4)-t**Integer(2)+q**Inte
↪ger(2)*t**Integer(2)-Integer(4)*t**Integer(4)*q**Integer(2)+q**Integer(4)*t**Int
↪eger(4)+q**Integer(2)*t**Integer(6)-Integer(5)*t**Integer(6)*q**Integer(4)+Integ
↪er(3)*t**Integer(8)*q**Integer(4)-Integer(4)*t**Integer(8)*q**Integer(6)+Integer
↪(2)*t**Integer(10)*q**Integer(6)+q**Integer(6)*t**Integer(12)-Integer(2)*q**Inte
↪ger(8)*t**Integer(12)-Integer(2)*t**Integer(14)*q**Integer(10)+Integer(2)*t**Int
↪eger(14)*q**Integer(8)+q**Integer(10)*t**Integer(16)+t**Integer(18)*q**Integer(1
↪2))*q*x3/((q**Integer(3)*t**Integer(5)+Integer(1))*(q**Integer(3)*t**Integer(5)-
↪Integer(1))*(t*q-Integer(1))*(t*q+Integer(1))*(t**Integer(3)*q**Integer(2)+Integ
↪er(1))*(t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(2)*q-Integer(1))*(t*
↪*Integer(2)*q+Integer(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)
↪*(Integer(1)+t**Integer(2)+t**Integer(4)*q**Integer(2))*q*x3*x2**Integer(2)/((t*
↪q-Integer(1))*(t*q+Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t**Inte
↪ger(3)*q**Integer(2)-Integer(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**Int
↪eger(2)*(-q**Integer(2)-Integer(2)-q**Integer(2)*t**Integer(2)+t**Integer(4)-q**
↪Integer(4)*t**Integer(4)-t**Integer(4)*q**Integer(2)+Integer(3)*q**Integer(2)*t*
↪*Integer(6)-t**Integer(6)*q**Integer(4)-t**Integer(8)*q**Integer(6)+t**Integer(8
↪)*q**Integer(4)+t**Integer(10)*q**Integer(4)+Integer(2)*q**Integer(6)*t**Integer
↪(12)-q**Integer(8)*t**Integer(12)+t**Integer(14)*q**Integer(8))*q*x3*x2*x1/((t**
↪Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t
↪*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Int
↪eger(3)*t**Integer(5)+Integer(1)))+(t-Integer(1))*(t+Integer(1))*x1**Integer(2)/
↪((q**Integer(3)*t**Integer(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1
↪))*x3*x2)+(t-Integer(1))*(t+Integer(1))*(-q**Integer(2)-Integer(1)+t**Integer(4)
↪*q**Integer(2)-q**Integer(4)*t**Integer(4)+Integer(2)*t**Integer(6)*q**Integer(4
↪))*x2**Integer(2)/((t*q-Integer(1))*(t*q+Integer(1))*(t**Integer(3)*q**Integer(2
↪)+Integer(1))*(t**Integer(3)*q**Integer(2)-Integer(1)))+(t-Integer(1))*(t+Intege
↪r(1))*(t**Integer(3)*q-Integer(1))*(t**Integer(3)*q+Integer(1))*x3*x2**Integer(2
↪)*x1/((t*q-Integer(1))*(t*q+Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))
↪*(t**Integer(3)*q**Integer(2)-Integer(1)))+(t-Integer(1))**Integer(2)*(t+Integer
↪(1))**Integer(2)*(q**Integer(2)+Integer(1))*q*x1/((t*q+Integer(1))*(t*q-Integer(
↪1))*(q**Integer(3)*t**Integer(5)+Integer(1))*(q**Integer(3)*t**Integer(5)-Intege
↪r(1))*x3*x2)+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(t**Integer(3
↪)*q-Integer(1))*(t**Integer(3)*q+Integer(1))*x3*x2*x1**Integer(2)/((t**Integer(3
↪)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t*q+Intege
↪r(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Integer(3)*t
↪**Integer(5)+Integer(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*
↪q**Integer(3)*x3/((t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)
↪-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x1*x2)+(t-Integer(1))*(t+I
↪nteger(1))*(-Integer(1)-q**Integer(2)+q**Integer(2)*t**Integer(2)+t**Integer(10)
↪*q**Integer(6))*q*x2/((t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Intege
```

(continues on next page)

(continued from previous page)

```
↪r(5)+Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*x3*x1)+x2**Integer(2)/
↪(x1*x3)+(t-Integer(1))*(t+Integer(1))*q*x2**Integer(2)/((t*q-Integer(1))*(t*q+In
↪teger(1))*x3)+(t-Integer(1))**Integer(3)*(t+Integer(1))**Integer(3)*(Integer(1)+
↪t**Integer(2)+t**Integer(4)*q**Integer(2))*q*x2*x1**Integer(2)/((t**Integer(3)*q
↪**Integer(2)-Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t*q+Integer(1
↪))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Integer(3)*t**I
↪nteger(5)+Integer(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*q*x
↪1**Integer(2)/((t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-In
↪teger(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x3)+(t-Integer(1))**Integer(2
↪)*(t+Integer(1))**Integer(2)*(q**Integer(2)*t**Integer(6)+Integer(2)*t**Integer(
↪6)*q**Integer(4)-q**Integer(4)*t**Integer(4)+t**Integer(4)*q**Integer(2)-q**Inte
↪ger(2)*t**Integer(2)+t**Integer(2)-Integer(2)-q**Integer(2))*q**Integer(3)/((t**
↪Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t
↪*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Int
↪eger(3)*t**Integer(5)+Integer(1))*x2)+(t-Integer(1))*(t+Integer(1))*(q**Integer(
↪2)+Integer(2)-t**Integer(2)+q**Integer(4)*t**Integer(4)-t**Integer(4)*q**Integer
↪(2)-Integer(3)*t**Integer(6)*q**Integer(4)+t**Integer(8)*q**Integer(4)-Integer(2
↪)*t**Integer(10)*q**Integer(6)-q**Integer(8)*t**Integer(12)+q**Integer(6)*t**Int
↪eger(12)+q**Integer(8)*t**Integer(16)+q**Integer(10)*t**Integer(16))*q**Integer(
↪2)*x2/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**Integer(2)+Int
↪eger(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(
↪1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x1)+(t-Integer(1))**Integer(2)*(t+I
↪nteger(1))**Integer(2)*(q**Integer(2)+Integer(1))*q**Integer(2)/((t*q+Integer(1)
↪)*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Integer(3)*t**In
↪teger(5)+Integer(1))*x3*x2)+(t-Integer(1))*(t+Integer(1))*(Integer(1)+q**Integer
↪(4)+Integer(2)*q**Integer(2)-Integer(2)*q**Integer(2)*t**Integer(2)+t**Integer(4
↪)*q**Integer(6)-q**Integer(4)*t**Integer(4)-Integer(3)*q**Integer(6)*t**Integer(
↪6)-t**Integer(6)*q**Integer(4)+Integer(2)*t**Integer(8)*q**Integer(6)-t**Integer
↪(10)*q**Integer(6)-q**Integer(8)*t**Integer(10)-t**Integer(14)*q**Integer(10)+t*
↪*Integer(14)*q**Integer(8)+Integer(2)*q**Integer(10)*t**Integer(16))*x2/((t**Int
↪eger(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t*q+
↪Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Intege
↪r(3)*t**Integer(5)+Integer(1))*x3)+(t-Integer(1))**Integer(2)*(t+Integer(1))**In
↪teger(2)*(-q**Integer(2)-Integer(2)-q**Integer(2)*t**Integer(2)-q**Integer(4)*t*
↪*Integer(4)+Integer(2)*t**Integer(6)*q**Integer(4)+t**Integer(10)*q**Integer(6)+
↪q**Integer(8)*t**Integer(12)+t**Integer(14)*q**Integer(8))*q**Integer(3)/((t**In
↪teger(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t*q
↪+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Integ
↪er(3)*t**Integer(5)+Integer(1))*x1)+(t-Integer(1))**Integer(2)*(t+Integer(1))**I
↪nteger(2)*(-Integer(1)-q**Integer(2)-q**Integer(2)*t**Integer(2)+t**Integer(2)+t
↪**Integer(4)*q**Integer(2)-q**Integer(4)*t**Integer(4)+Integer(2)*t**Integer(6)*
↪q**Integer(4))*q**Integer(2)*x3/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**In
↪teger(3)*q**Integer(2)+Integer(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer
↪(3)*t**Integer(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x2)+(t-In
↪teger(1))*(t+Integer(1))*q*x2**Integer(2)/((t*q-Integer(1))*(t*q+Integer(1))*x1)
↪+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(Integer(1)+t**Integer(2)
↪+t**Integer(4)*q**Integer(2))*q*x2**Integer(2)*x1/((t*q-Integer(1))*(t*q+Integer
↪(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t**Integer(3)*q**Integer(2)-Integ
↪er(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*q*x1**Integer(2)/(
↪(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**I
↪nteger(3)*t**Integer(5)+Integer(1))*x2)+(t-Integer(1))**Integer(2)*(t+Integer(1)
↪)**Integer(2)*(-Integer(1)-q**Integer(4)-Integer(2)*q**Integer(2)-t**Integer(2)*
```

(continues on next page)

(continued from previous page)

```
↪q**Integer(4)-q**Integer(2)*t**Integer(2)+t**Integer(4)*q**Integer(2)-t**Integer
↪(4)*q**Integer(6)-Integer(2)*q**Integer(4)*t**Integer(4)+Integer(3)*t**Integer(6
↪)*q**Integer(4)-q**Integer(6)*t**Integer(6)-t**Integer(8)*q**Integer(8)+t**Integ
↪er(8)*q**Integer(6)+Integer(2)*t**Integer(10)*q**Integer(6)-q**Integer(10)*t**In
↪teger(12)+Integer(3)*q**Integer(8)*t**Integer(12)+Integer(2)*t**Integer(14)*q**I
↪nteger(10))*x3*x2/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**In
↪teger(2)+Integer(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer
↪(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1)))+(t-Integer(1))*(t+Inte
↪ger(1))*(q**Integer(2)+Integer(1)-t**Integer(2)+q**Integer(4)*t**Integer(4)-t**I
↪nteger(4)*q**Integer(2)+q**Integer(2)*t**Integer(6)-Integer(3)*t**Integer(6)*q**
↪Integer(4)+t**Integer(8)*q**Integer(4)-t**Integer(10)*q**Integer(6)+q**Integer(6
↪)*t**Integer(12)-q**Integer(8)*t**Integer(12)+q**Integer(10)*t**Integer(16))*q**
↪Integer(2)*x3/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**Intege
↪r(2)+Integer(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-
↪Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x1)+(t-Integer(1))*(t+Integ
↪er(1))*(-Integer(1)-q**Integer(2)+q**Integer(2)*t**Integer(2)+t**Integer(10)*q**
↪Integer(6))*q**Integer(2)/((t*q-Integer(1))*(t*q+Integer(1))*(q**Integer(3)*t**I
↪nteger(5)+Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*x1*x3)+(t-Integer
↪(1))*(t+Integer(1))*(Integer(1)+q**Integer(4)+Integer(2)*q**Integer(2)-Integer(3
↪)*q**Integer(2)*t**Integer(2)+t**Integer(4)*q**Integer(6)-q**Integer(4)*t**Integ
↪er(4)-Integer(3)*q**Integer(6)*t**Integer(6)-t**Integer(6)*q**Integer(4)+t**Inte
↪ger(8)*q**Integer(4)+Integer(2)*t**Integer(8)*q**Integer(6)-t**Integer(10)*q**In
↪teger(6)+t**Integer(14)*q**Integer(8)-t**Integer(14)*q**Integer(10)+q**Integer(1
↪0)*t**Integer(16))*x1/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q
↪**Integer(2)+Integer(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Int
↪eger(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x3)+(t-Integer(1))*
↪*Integer(2)*(t+Integer(1))**Integer(2)*(Integer(3)*q**Integer(2)+q**Integer(4)+I
↪nteger(2)+q**Integer(2)*t**Integer(2)-t**Integer(2)+t**Integer(2)*q**Integer(4)-
↪Integer(6)*t**Integer(4)*q**Integer(2)+q**Integer(4)*t**Integer(4)-Integer(7)*t*
↪*Integer(6)*q**Integer(4)+q**Integer(2)*t**Integer(6)+Integer(3)*t**Integer(8)*q
↪**Integer(4)-Integer(4)*t**Integer(8)*q**Integer(6)+t**Integer(10)*q**Integer(4)
↪+Integer(3)*t**Integer(10)*q**Integer(6)-q**Integer(8)*t**Integer(12)-t**Integer
↪(14)*q**Integer(10)+t**Integer(14)*q**Integer(8)+q**Integer(8)*t**Integer(16)+q*
↪*Integer(10)*t**Integer(18))*q*x1/((q**Integer(3)*t**Integer(5)+Integer(1))*(q**
↪Integer(3)*t**Integer(5)-Integer(1))*(t*q-Integer(1))*(t*q+Integer(1))*(t**Integ
↪er(3)*q**Integer(2)+Integer(1))*(t**Integer(3)*q**Integer(2)-Integer(1))*(t**Int
↪eger(2)*q-Integer(1))*(t**Integer(2)*q+Integer(1)))+(t-Integer(1))**Integer(2)*(
↪t+Integer(1))**Integer(2)*(-q**Integer(2)-Integer(2)-q**Integer(2)*t**Integer(2)
↪-q**Integer(4)*t**Integer(4)+Integer(2)*t**Integer(6)*q**Integer(4)+t**Integer(1
↪0)*q**Integer(6)+q**Integer(6)*t**Integer(12)+t**Integer(14)*q**Integer(8))*q*x2
↪*x1/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**Integer(2)+Integ
↪er(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1)
↪)*(q**Integer(3)*t**Integer(5)+Integer(1))*x3)+(t+Integer(1))*(t-Integer(1))*x2*
↪*Integer(2)*x1/((t*q-Integer(1))*(t*q+Integer(1))*x3)+(t-Integer(1))**Integer(3)
↪*(t+Integer(1))**Integer(3)*(Integer(1)+t**Integer(2)+t**Integer(4)*q**Integer(2
↪))*q*x3*x1**Integer(2)/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*
↪q**Integer(2)+Integer(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**In
↪teger(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1)))+(t-Integer(1))*(t
↪+Integer(1))*q**Integer(3)/((q**Integer(3)*t**Integer(5)+Integer(1))*(q**Integer
↪(3)*t**Integer(5)-Integer(1))*x1*x2*x3)+(t-Integer(1))**Integer(2)*(t+Integer(1)
↪)**Integer(2)*(Integer(3)+Integer(3)*q**Integer(2)+q**Integer(4)+Integer(2)*q**I
↪nteger(2)*t**Integer(2)-t**Integer(2)+t**Integer(2)*q**Integer(4)-Integer(6)*t**
```

(continues on next page)

(continued from previous page)

```
↪Integer(4)*q**Integer(2)+q**Integer(4)*t**Integer(4)-Integer(8)*t**Integer(6)*q*
↪*Integer(4)+q**Integer(2)*t**Integer(6)+Integer(2)*t**Integer(8)*q**Integer(4)-I
↪nteger(4)*t**Integer(8)*q**Integer(6)+t**Integer(10)*q**Integer(4)+Integer(2)*t*
↪*Integer(10)*q**Integer(6)-Integer(2)*q**Integer(8)*t**Integer(12)-t**Integer(14
↪)*q**Integer(10)+t**Integer(14)*q**Integer(8)+q**Integer(8)*t**Integer(16)+q**In
↪teger(10)*t**Integer(16)+Integer(2)*q**Integer(10)*t**Integer(18))*q**Integer(2)
↪/((q**Integer(3)*t**Integer(5)+Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(
↪1))*(t*q-Integer(1))*(t*q+Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(
↪t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(2)*q-Integer(1))*(t**Integer
↪(2)*q+Integer(1)))+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*(-q**In
↪teger(4)-Integer(2)*q**Integer(2)-Integer(1)-t**Integer(2)*q**Integer(4)-t**Inte
↪ger(4)*q**Integer(6)+Integer(2)*q**Integer(6)*t**Integer(6)+t**Integer(6)*q**Int
↪eger(4)+t**Integer(10)*q**Integer(6)+q**Integer(8)*t**Integer(12)+t**Integer(14)
↪*q**Integer(10))*q/((t**Integer(3)*q**Integer(2)-Integer(1))*(t**Integer(3)*q**I
↪nteger(2)+Integer(1))*(t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Intege
↪r(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x3)+(t-Integer(1))**In
↪teger(2)*(t+Integer(1))**Integer(2)*(-Integer(1)-q**Integer(2)-q**Integer(2)*t**
↪Integer(2)+t**Integer(2)+t**Integer(4)*q**Integer(2)-q**Integer(4)*t**Integer(4)
↪+Integer(2)*t**Integer(6)*q**Integer(4))*q*x3*x1/((t**Integer(3)*q**Integer(2)-I
↪nteger(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t*q+Integer(1))*(t*q-Intege
↪r(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Inte
↪ger(1))*x2)+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*x2*x1**Integer
↪(2)/((t*q+Integer(1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*
↪(q**Integer(3)*t**Integer(5)+Integer(1))*x3)+(t-Integer(1))**Integer(2)*(t+Integ
↪er(1))**Integer(2)*x3*x1**Integer(2)/((t*q+Integer(1))*(t*q-Integer(1))*(q**Inte
↪ger(3)*t**Integer(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*x2)+(t
↪-Integer(1))**Integer(2)*(t+Integer(1))**Integer(2)*q**Integer(4)/((t*q+Integer(
↪1))*(t*q-Integer(1))*(q**Integer(3)*t**Integer(5)+Integer(1))*(q**Integer(3)*t**
↪Integer(5)-Integer(1))*x1*x2)+(t-Integer(1))**Integer(2)*(t+Integer(1))**Integer
↪(2)*(-q**Integer(2)-Integer(1)-q**Integer(2)*t**Integer(2)-q**Integer(4)*t**Inte
↪ger(4)+t**Integer(6)*q**Integer(4)+t**Integer(10)*q**Integer(6)+q**Integer(8)*t*
↪*Integer(12)+t**Integer(14)*q**Integer(10))*q*x3*x2/((t**Integer(3)*q**Integer(2
↪)-Integer(1))*(t**Integer(3)*q**Integer(2)+Integer(1))*(t*q+Integer(1))*(t*q-Int
↪eger(1))*(q**Integer(3)*t**Integer(5)-Integer(1))*(q**Integer(3)*t**Integer(5)+I
↪nteger(1))*x1)  # needs sage.symbolic
>>> to_SR(E[mu]) - expected          # long time (20s)                  #␣
↪needs sage.symbolic
0

>>> E = NonSymmetricMacdonaldPolynomials(["BC",Integer(1),Integer(2)], q=q,␣
↪q1=t**Integer(2), q2=-Integer(1))
>>> omega = E.keys().fundamental_weights()
>>> mu = -Integer(4)*omega[Integer(1)]; mu
(-4)
>>> expected = (t-Integer(1))*(t+Integer(1))*(-Integer(1)+q**Integer(2)*t**Integer
↪(2)-q**Integer(2)-Integer(3)*q**Integer(10)-Integer(7)*q**Integer(26)*t**Integer
↪(8)+Integer(5)*t**Integer(2)*q**Integer(6)-q**Integer(16)-Integer(3)*q**Integer(
↪4)+Integer(4)*t**Integer(10)*q**Integer(30)-Integer(4)*t**Integer(6)*q**Integer(
↪22)-Integer(10)*q**Integer(20)*t**Integer(6)+Integer(2)*q**Integer(32)*t**Intege
↪r(10)-Integer(3)*q**Integer(6)-Integer(4)*q**Integer(8)+q**Integer(34)*t**Intege
↪r(10)-Integer(4)*t**Integer(8)*q**Integer(24)-Integer(2)*q**Integer(12)-q**Integ
↪er(14)+Integer(2)*q**Integer(22)*t**Integer(10)+Integer(4)*q**Integer(26)*t**Int
```

(continues on next page)

(continued from previous page)

```
↪eger(10)+Integer(4)*q**Integer(28)*t**Integer(10)+t**Integer(6)*q**Integer(30)-I
↪nteger(2)*q**Integer(32)*t**Integer(8)-Integer(2)*t**Integer(8)*q**Integer(22)+I
↪nteger(2)*q**Integer(24)*t**Integer(10)-q**Integer(20)*t**Integer(2)-Integer(2)*
↪t**Integer(6)*q**Integer(12)+t**Integer(8)*q**Integer(14)+Integer(2)*t**Integer(
↪4)*q**Integer(24)-Integer(4)*t**Integer(8)*q**Integer(30)+Integer(2)*t**Integer(
↪8)*q**Integer(20)-Integer(9)*t**Integer(6)*q**Integer(16)+Integer(3)*q**Integer(
↪26)*t**Integer(6)+q**Integer(28)*t**Integer(6)+Integer(3)*t**Integer(2)*q**Integ
↪er(4)+Integer(2)*q**Integer(18)*t**Integer(8)-Integer(6)*t**Integer(6)*q**Intege
↪r(14)+Integer(4)*t**Integer(4)*q**Integer(22)-Integer(2)*q**Integer(24)*t**Integ
↪er(6)+Integer(3)*t**Integer(2)*q**Integer(12)+Integer(7)*t**Integer(4)*q**Intege
↪r(20)-t**Integer(2)*q**Integer(16)+Integer(11)*q**Integer(18)*t**Integer(4)-Inte
↪ger(2)*t**Integer(2)*q**Integer(18)+Integer(9)*q**Integer(16)*t**Integer(4)-t**I
↪nteger(4)*q**Integer(6)+Integer(6)*q**Integer(8)*t**Integer(2)+Integer(5)*q**Int
↪eger(10)*t**Integer(2)-Integer(6)*q**Integer(28)*t**Integer(8)+q**Integer(12)*t*
↪*Integer(4)+Integer(8)*t**Integer(4)*q**Integer(14)-Integer(10)*t**Integer(6)*q*
↪*Integer(18)-q**Integer(4)*t**Integer(4)+q**Integer(16)*t**Integer(8)-Integer(2)
↪*t**Integer(4)*q**Integer(8))/((t*q**Integer(4)-Integer(1))*(t*q**Integer(4)+Int
↪eger(1))*(q**Integer(7)*t**Integer(2)-Integer(1))*(q**Integer(7)*t**Integer(2)+I
↪nteger(1))*(t*q**Integer(3)-Integer(1))*(t*q**Integer(3)+Integer(1))*(q**Integer
↪(5)*t**Integer(2)+Integer(1))*(q**Integer(5)*t**Integer(2)-Integer(1)))+(q**Inte
↪ger(2)+Integer(1))*(q**Integer(4)+Integer(1))*(t-Integer(1))*(t+Integer(1))*(-In
↪teger(1)+q**Integer(2)*t**Integer(2)-q**Integer(2)+t**Integer(2)*q**Integer(6)-q
↪**Integer(4)+t**Integer(6)*q**Integer(22)+Integer(3)*q**Integer(10)*t**Integer(4
↪)+t**Integer(2)-q**Integer(8)-Integer(2)*t**Integer(8)*q**Integer(24)+q**Integer
↪(22)*t**Integer(10)+q**Integer(26)*t**Integer(10)-Integer(2)*t**Integer(8)*q**In
↪teger(22)+q**Integer(24)*t**Integer(10)-Integer(4)*t**Integer(6)*q**Integer(12)-
↪Integer(2)*t**Integer(8)*q**Integer(20)-Integer(3)*t**Integer(6)*q**Integer(16)+
↪Integer(2)*t**Integer(2)*q**Integer(4)-t**Integer(6)*q**Integer(10)-Integer(2)*t
↪**Integer(6)*q**Integer(14)+t**Integer(8)*q**Integer(12)-t**Integer(2)*q**Intege
↪r(12)+Integer(2)*q**Integer(16)*t**Integer(4)+q**Integer(8)*t**Integer(2)-q**Int
↪eger(10)*t**Integer(2)+Integer(3)*q**Integer(12)*t**Integer(4)+Integer(2)*t**Int
↪eger(4)*q**Integer(14)+t**Integer(6)*q**Integer(18)-Integer(2)*q**Integer(4)*t**
↪Integer(4)+q**Integer(16)*t**Integer(8)+q**Integer(20)*t**Integer(10))*q*x1/((t*
↪q**Integer(4)-Integer(1))*(t*q**Integer(4)+Integer(1))*(q**Integer(7)*t**Integer
↪(2)-Integer(1))*(q**Integer(7)*t**Integer(2)+Integer(1))*(t*q**Integer(3)-Intege
↪r(1))*(t*q**Integer(3)+Integer(1))*(q**Integer(5)*t**Integer(2)+Integer(1))*(q**
↪Integer(5)*t**Integer(2)-Integer(1)))+(q**Integer(2)+Integer(1))*(q**Integer(4)+
↪Integer(1))*(t-Integer(1))*(t+Integer(1))*(Integer(1)+q**Integer(8)+q**Integer(4
↪)+q**Integer(2)-q**Integer(8)*t**Integer(2)-Integer(2)*t**Integer(2)*q**Integer(
↪4)-t**Integer(2)*q**Integer(6)+t**Integer(2)*q**Integer(12)-t**Integer(2)+t**Int
↪eger(4)*q**Integer(6)-Integer(2)*q**Integer(16)*t**Integer(4)-t**Integer(4)*q**I
↪nteger(14)-Integer(2)*q**Integer(12)*t**Integer(4)+t**Integer(6)*q**Integer(12)+
↪t**Integer(6)*q**Integer(16)+t**Integer(6)*q**Integer(18)+t**Integer(6)*q**Integ
↪er(14))*q/((t*q**Integer(4)-Integer(1))*(t*q**Integer(4)+Integer(1))*(q**Integer
↪(7)*t**Integer(2)-Integer(1))*(q**Integer(7)*t**Integer(2)+Integer(1))*(t*q**Int
↪eger(3)-Integer(1))*(t*q**Integer(3)+Integer(1))*x1)+(t-Integer(1))*(t+Integer(1
↪))*(-Integer(1)-q**Integer(2)-q**Integer(6)-q**Integer(4)-q**Integer(8)+t**Integ
↪er(2)*q**Integer(4)-t**Integer(2)*q**Integer(14)+t**Integer(2)*q**Integer(6)-q**
↪Integer(10)*t**Integer(2)+q**Integer(8)*t**Integer(2)-t**Integer(2)*q**Integer(1
↪2)+q**Integer(12)*t**Integer(4)+q**Integer(10)*t**Integer(4)+q**Integer(16)*t**I
↪nteger(4)+Integer(2)*t**Integer(4)*q**Integer(14))*(q**Integer(4)+Integer(1))/((
↪q**Integer(7)*t**Integer(2)+Integer(1))*(q**Integer(7)*t**Integer(2)-Integer(1))
```

(continues on next page)

(continued from previous page)

```
↪*(t*q**Integer(4)-Integer(1))*(t*q**Integer(4)+Integer(1))*x1**Integer(2))+(t-In
↪teger(1))*(t+Integer(1))*(q**Integer(4)+Integer(1))*(q**Integer(2)+Integer(1))*q
↪/((t*q**Integer(4)-Integer(1))*(t*q**Integer(4)+Integer(1))*x1**Integer(3))+(q**
↪Integer(4)+Integer(1))*(t-Integer(1))*(t+Integer(1))*(Integer(1)+q**Integer(6)+q
↪**Integer(8)+q**Integer(2)+q**Integer(4)-q**Integer(2)*t**Integer(2)-Integer(3)*
↪t**Integer(2)*q**Integer(4)+q**Integer(10)*t**Integer(2)+t**Integer(2)*q**Intege
↪r(12)-Integer(2)*t**Integer(2)*q**Integer(6)-q**Integer(8)*t**Integer(2)-Integer
↪(2)*q**Integer(16)*t**Integer(4)+q**Integer(4)*t**Integer(4)+t**Integer(4)*q**In
↪teger(6)-q**Integer(10)*t**Integer(4)-Integer(2)*q**Integer(12)*t**Integer(4)-In
↪teger(2)*t**Integer(4)*q**Integer(14)+t**Integer(6)*q**Integer(12)+t**Integer(6)
↪*q**Integer(18)+Integer(2)*t**Integer(6)*q**Integer(16)+t**Integer(6)*q**Integer
↪(14))*x1**Integer(2)/((t*q**Integer(4)-Integer(1))*(t*q**Integer(4)+Integer(1))*
↪(q**Integer(7)*t**Integer(2)-Integer(1))*(q**Integer(7)*t**Integer(2)+Integer(1)
↪)*(t*q**Integer(3)-Integer(1))*(t*q**Integer(3)+Integer(1)))+(t-Integer(1))*(t+I
↪nteger(1))*(-Integer(1)-t**Integer(2)*q**Integer(6)+t**Integer(2)+t**Integer(4)*
↪q**Integer(8))*(q**Integer(4)+Integer(1))*(q**Integer(2)+Integer(1))*q*x1**Integ
↪er(3)/((q**Integer(7)*t**Integer(2)+Integer(1))*(q**Integer(7)*t**Integer(2)-Int
↪eger(1))*(t*q**Integer(4)-Integer(1))*(t*q**Integer(4)+Integer(1)))+Integer(1)/x
↪1**Integer(4)+(t-Integer(1))*(t+Integer(1))*x1**Integer(4)/((t*q**Integer(4)-Int
↪eger(1))*(t*q**Integer(4)+Integer(1)))   # needs sage.symbolic
>>> to_SR(E[mu]) - expected                                              #␣
↪needs sage.symbolic
0
```

Type BC dual, comparison with hand calculations by Bogdan Ion:

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q,q1,q2 = K.gens()
sage: ct = CartanType(["BC",2,2]).dual()
sage: E = NonSymmetricMacdonaldPolynomials(ct, q=q, q1=q1, q2=q2)
sage: KL = E.domain(); KL
Algebra of the Ambient space of the Root system of type ['B', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational␣
↪Field
sage: alpha = E.keys().simple_roots(); alpha
Finite family {1: (1, -1), 2: (0, 1)}
sage: omega=E.keys().fundamental_weights(); omega
Finite family {1: (1, 0), 2: (1/2, 1/2)}
sage: epsilon = E.keys().basis(); epsilon
Finite family {0: (1, 0), 1: (0, 1)}
```

```
>>> from sage.all import *
>>> K = QQ['q,q1,q2'].fraction_field()
>>> q,q1,q2 = K.gens()
>>> ct = CartanType(["BC",Integer(2),Integer(2)]).dual()
>>> E = NonSymmetricMacdonaldPolynomials(ct, q=q, q1=q1, q2=q2)
>>> KL = E.domain(); KL
Algebra of the Ambient space of the Root system of type ['B', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational␣
↪Field
>>> alpha = E.keys().simple_roots(); alpha
Finite family {1: (1, -1), 2: (0, 1)}
```

(continues on next page)

(continued from previous page)

```
>>> omega=E.keys().fundamental_weights(); omega
Finite family {1: (1, 0), 2: (1/2, 1/2)}
>>> epsilon = E.keys().basis(); epsilon
Finite family {0: (1, 0), 1: (0, 1)}
```

Note: Sage's q is the usual q^2:

```
sage: E.L().null_root()
e['delta']
sage: E.L().null_coroot()
2*e['deltacheck']
```

```
>>> from sage.all import *
>>> E.L().null_root()
e['delta']
>>> E.L().null_coroot()
2*e['deltacheck']
```

Some eigenvectors:

```
sage: E[0*omega[1]]
B[(0, 0)]
sage: E[omega[1]]
-((q^2*q1^3*q2+q^2*q1^2*q2^2)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]
sage: Eomega1 = (KL.one() * (q^2*(-q1/q2)^2*(1-(-q1/q2))) / (1-q^2*(-q1/q2)^4)
....:            + KL.monomial(omega[1]))
sage: E[omega[1]] == Eomega1
True
```

```
>>> from sage.all import *
>>> E[Integer(0)*omega[Integer(1)]]
B[(0, 0)]
>>> E[omega[Integer(1)]]
-((q^2*q1^3*q2+q^2*q1^2*q2^2)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]
>>> Eomega1 = (KL.one() * (q**Integer(2)*(-q1/q2)**Integer(2)*(Integer(1)-(-q1/
↪q2))) / (Integer(1)-q**Integer(2)*(-q1/q2)**Integer(4))
...            + KL.monomial(omega[Integer(1)]))
>>> E[omega[Integer(1)]] == Eomega1
True
```

Checking the Y s:

```
sage: Y = E.Y()
sage: alphacheck = Y.keys().simple_roots()
sage: Y0 = Y[alphacheck[0]]
sage: Y1 = Y[alphacheck[1]]
sage: Y2 = Y[alphacheck[2]]

sage: Y0.word, Y0.signs, Y0.scalar
((0, 1, 2, 1, 0, 1, 2, 1), (-1, -1, -1, -1, -1, -1, -1, -1), q1^4*q2^4/q^2)
sage: Y1.word, Y1.signs, Y1.scalar
((1, 2, 0, 1, 2, 0), (1, 1, -1, 1, -1, 1), 1/(-q1*q2))
```

(continues on next page)

(continued from previous page)

```
sage: Y2.word, Y2.signs, Y2.scalar
((2, 1, 0, 1), (1, 1, 1, -1), 1/(-q1*q2))

sage: E.eigenvalues(0*omega[1])
[q2^4/(q^2*q1^4), q1/(-q2), q1/(-q2)]
```

```
>>> from sage.all import *
>>> Y = E.Y()
>>> alphacheck = Y.keys().simple_roots()
>>> Y0 = Y[alphacheck[Integer(0)]]
>>> Y1 = Y[alphacheck[Integer(1)]]
>>> Y2 = Y[alphacheck[Integer(2)]]

>>> Y0.word, Y0.signs, Y0.scalar
((0, 1, 2, 1, 0, 1, 2, 1), (-1, -1, -1, -1, -1, -1, -1, -1), q1^4*q2^4/q^2)
>>> Y1.word, Y1.signs, Y1.scalar
((1, 2, 0, 1, 2, 0), (1, 1, -1, 1, -1, 1), 1/(-q1*q2))
>>> Y2.word, Y2.signs, Y2.scalar
((2, 1, 0, 1), (1, 1, 1, -1), 1/(-q1*q2))

>>> E.eigenvalues(Integer(0)*omega[Integer(1)])
[q2^4/(q^2*q1^4), q1/(-q2), q1/(-q2)]
```

Checking the T and T^{-1} s:

```
sage: T = E._T_Y
sage: Tinv0 = T.Tw_inverse([0])
sage: Tinv1 = T.Tw_inverse([1])
sage: Tinv2 = T.Tw_inverse([2])

sage: for x in [0*epsilon[0], -epsilon[0], -epsilon[1], epsilon[0], epsilon[1]]:
....:     x = KL.monomial(x)
....:     assert Tinv0(T[0](x)) == x and T[0](Tinv0(x)) == x
....:     assert Tinv1(T[1](x)) == x and T[1](Tinv1(x)) == x
....:     assert Tinv2(T[2](x)) == x and T[2](Tinv2(x)) == x

sage: start = E[omega[1]]; start
-((q^2*q1^3*q2+q^2*q1^2*q2^2)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]
sage: Tinv1(Tinv2(Tinv1(Tinv0(Tinv1(Tinv2(Tinv1(Tinv0(start)))))))) * (q1*q2)^4/q^
↪2 == Y0(start)
True
sage: Y0(start) == q^2*q1^4/q2^4 * start
True
```

```
>>> from sage.all import *
>>> T = E._T_Y
>>> Tinv0 = T.Tw_inverse([Integer(0)])
>>> Tinv1 = T.Tw_inverse([Integer(1)])
>>> Tinv2 = T.Tw_inverse([Integer(2)])

>>> for x in [Integer(0)*epsilon[Integer(0)], -epsilon[Integer(0)], -
↪epsilon[Integer(1)], epsilon[Integer(0)], epsilon[Integer(1)]]:
```

(continues on next page)

(continued from previous page)

```
...     x = KL.monomial(x)
...     assert Tinv0(T[Integer(0)](x)) == x and T[Integer(0)](Tinv0(x)) == x
...     assert Tinv1(T[Integer(1)](x)) == x and T[Integer(1)](Tinv1(x)) == x
...     assert Tinv2(T[Integer(2)](x)) == x and T[Integer(2)](Tinv2(x)) == x

>>> start = E[omega[Integer(1)]]; start
-((q^2*q1^3*q2+q^2*q1^2*q2^2)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]
>>> Tinv1(Tinv2(Tinv1(Tinv0(Tinv1(Tinv2(Tinv1(Tinv0(start)))))))) *␣
↪(q1*q2)**Integer(4)/q**Integer(2) == Y0(start)
True
>>> Y0(start) == q**Integer(2)*q1**Integer(4)/q2**Integer(4) * start
True
```

Checking the relation between the Y s:

```
sage: q^2 * Y0(Y1(Y1(Y2(Y2(start))))) == start
True
sage: for x in [0*epsilon[0], -epsilon[0], -epsilon[1], epsilon[0], epsilon[1]]:
....:     x = KL.monomial(x)
....:     assert q^2 * Y0(Y1(Y1(Y2(Y2(start))))) == start
```

```
>>> from sage.all import *
>>> q**Integer(2) * Y0(Y1(Y1(Y2(Y2(start))))) == start
True
>>> for x in [Integer(0)*epsilon[Integer(0)], -epsilon[Integer(0)], -
↪epsilon[Integer(1)], epsilon[Integer(0)], epsilon[Integer(1)]]:
...     x = KL.monomial(x)
...     assert q**Integer(2) * Y0(Y1(Y1(Y2(Y2(start))))) == start
```

KL0()

Return the group algebra where the nonsymmetric Macdonald polynomials live.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials("B2~").KL0()
Algebra of the Ambient space of the Root system of type ['B', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over␣
↪Rational Field
sage: NonSymmetricMacdonaldPolynomials("B2~*").KL0()
Algebra of the Ambient space of the Root system of type ['C', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over␣
↪Rational Field
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials("B2~").KL0()
Algebra of the Ambient space of the Root system of type ['B', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over␣
↪Rational Field
>>> NonSymmetricMacdonaldPolynomials("B2~*").KL0()
Algebra of the Ambient space of the Root system of type ['C', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over␣
↪Rational Field
```

L()

Return the affinization of the classical weight space.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).L()
Ambient space of the Root system of type ['B', 2, 1]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials(["B", Integer(2), Integer(1)]).L()
Ambient space of the Root system of type ['B', 2, 1]
```

L0()

Return the space indexing the monomials of the nonsymmetric Macdonald polynomials.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials("B2~").L0()
Ambient space of the Root system of type ['B', 2]
sage: NonSymmetricMacdonaldPolynomials("B2~*").L0()
Ambient space of the Root system of type ['C', 2]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials("B2~").L0()
Ambient space of the Root system of type ['B', 2]
>>> NonSymmetricMacdonaldPolynomials("B2~*").L0()
Ambient space of the Root system of type ['C', 2]
```

L_check()

Return the other affinization of the classical weight space.

Todo

should this just return L in the simply laced case?

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).L_check()
Coambient space of the Root system of type ['C', 2, 1]
sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).L_check().classical()
Ambient space of the Root system of type ['B', 2]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials(["B", Integer(2), Integer(1)]).L_check()
Coambient space of the Root system of type ['C', 2, 1]
>>> NonSymmetricMacdonaldPolynomials(["B", Integer(2), Integer(1)]).L_check().
↪classical()
Ambient space of the Root system of type ['B', 2]
```

L_prime()

The affine space where classical weights are lifted for the recursion.

Also the parent of ρ'.

EXAMPLES:

In the twisted case, this is the affinization of the classical ambient space:

```
sage: NonSymmetricMacdonaldPolynomials("B2~*").L()
Ambient space of the Root system of type ['B', 2, 1]^*
sage: NonSymmetricMacdonaldPolynomials("B2~*").L().classical()
Ambient space of the Root system of type ['C', 2]

sage: NonSymmetricMacdonaldPolynomials("B2~*").L_prime()
Ambient space of the Root system of type ['B', 2, 1]^*
sage: NonSymmetricMacdonaldPolynomials("B2~*").L_prime().classical()
Ambient space of the Root system of type ['C', 2]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials("B2~*").L()
Ambient space of the Root system of type ['B', 2, 1]^*
>>> NonSymmetricMacdonaldPolynomials("B2~*").L().classical()
Ambient space of the Root system of type ['C', 2]

>>> NonSymmetricMacdonaldPolynomials("B2~*").L_prime()
Ambient space of the Root system of type ['B', 2, 1]^*
>>> NonSymmetricMacdonaldPolynomials("B2~*").L_prime().classical()
Ambient space of the Root system of type ['C', 2]
```

In the untwisted case, this is the other affinization of the classical ambient space:

```
sage: NonSymmetricMacdonaldPolynomials("B2~").L()
Ambient space of the Root system of type ['B', 2, 1]
sage: NonSymmetricMacdonaldPolynomials("B2~").L().classical()
Ambient space of the Root system of type ['B', 2]

sage: NonSymmetricMacdonaldPolynomials("B2~").L_prime()
Coambient space of the Root system of type ['C', 2, 1]
sage: NonSymmetricMacdonaldPolynomials("B2~").L_prime().classical()
Ambient space of the Root system of type ['B', 2]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials("B2~").L()
Ambient space of the Root system of type ['B', 2, 1]
>>> NonSymmetricMacdonaldPolynomials("B2~").L().classical()
Ambient space of the Root system of type ['B', 2]

>>> NonSymmetricMacdonaldPolynomials("B2~").L_prime()
Coambient space of the Root system of type ['C', 2, 1]
>>> NonSymmetricMacdonaldPolynomials("B2~").L_prime().classical()
Ambient space of the Root system of type ['B', 2]
```

For simply laced, the two affinizations coincide:

```
sage: NonSymmetricMacdonaldPolynomials("A2~").L()
Ambient space of the Root system of type ['A', 2, 1]
sage: NonSymmetricMacdonaldPolynomials("A2~").L().classical()
Ambient space of the Root system of type ['A', 2]
```

(continues on next page)

(continued from previous page)

```
sage: NonSymmetricMacdonaldPolynomials("A2~").L_prime()
Coambient space of the Root system of type ['A', 2, 1]
sage: NonSymmetricMacdonaldPolynomials("A2~").L_prime().classical()
Ambient space of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials("A2~").L()
Ambient space of the Root system of type ['A', 2, 1]
>>> NonSymmetricMacdonaldPolynomials("A2~").L().classical()
Ambient space of the Root system of type ['A', 2]

>>> NonSymmetricMacdonaldPolynomials("A2~").L_prime()
Coambient space of the Root system of type ['A', 2, 1]
>>> NonSymmetricMacdonaldPolynomials("A2~").L_prime().classical()
Ambient space of the Root system of type ['A', 2]
```

Note

do we want the coambient space of type $A_2^{(1)}$ instead?

For type BC:

```
sage: NonSymmetricMacdonaldPolynomials(["BC",3,2]).L_prime()
Ambient space of the Root system of type ['BC', 3, 2]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials(["BC",Integer(3),Integer(2)]).L_prime()
Ambient space of the Root system of type ['BC', 3, 2]
```

Q_to_Qcheck()

The reindexing of the index set of the Y's by the coroot lattice.

EXAMPLES:

```
sage: E = NonSymmetricMacdonaldPolynomials("C2~")
sage: alphacheck = E.Y().keys().simple_roots()
sage: E.Q_to_Qcheck(alphacheck[0])
alphacheck[0] - alphacheck[2]
sage: E.Q_to_Qcheck(alphacheck[1])
alphacheck[1]
sage: E.Q_to_Qcheck(alphacheck[2])
alphacheck[2]

sage: x = alphacheck[1] + 2*alphacheck[2]
sage: x.parent()
Root lattice of the Root system of type ['B', 2, 1]
sage: E.Q_to_Qcheck(x)
alphacheck[1] + 2*alphacheck[2]
sage: _.parent()
Coroot lattice of the Root system of type ['C', 2, 1]
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials("C2~")
>>> alphacheck = E.Y().keys().simple_roots()
>>> E.Q_to_Qcheck(alphacheck[Integer(0)])
alphacheck[0] - alphacheck[2]
>>> E.Q_to_Qcheck(alphacheck[Integer(1)])
alphacheck[1]
>>> E.Q_to_Qcheck(alphacheck[Integer(2)])
alphacheck[2]

>>> x = alphacheck[Integer(1)] + Integer(2)*alphacheck[Integer(2)]
>>> x.parent()
Root lattice of the Root system of type ['B', 2, 1]
>>> E.Q_to_Qcheck(x)
alphacheck[1] + 2*alphacheck[2]
>>> _.parent()
Coroot lattice of the Root system of type ['C', 2, 1]
```

Y()

Return the family of Y operators whose eigenvectors are the nonsymmetric Macdonald polynomials.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials("C2~").Y()
Lazy family (<lambda>(i))_{i in Root lattice of the Root system of type ['B',␣
↪2, 1]}
sage: _.keys().classical()
Root lattice of the Root system of type ['B', 2]
sage: NonSymmetricMacdonaldPolynomials("C2~*").Y()
Lazy family (<...Y_lambdacheck...>(i))_{i in Coroot lattice of the Root␣
↪system of type ['C', 2, 1]^*}
sage: _.keys().classical()
Root lattice of the Root system of type ['C', 2]
sage: NonSymmetricMacdonaldPolynomials(["BC", 3, 2]).Y()
Lazy family (<...Y_lambdacheck...>(i))_{i in Coroot lattice of the Root␣
↪system of type ['BC', 3, 2]}
sage: _.keys().classical()
Root lattice of the Root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials("C2~").Y()
Lazy family (<lambda>(i))_{i in Root lattice of the Root system of type ['B',␣
↪2, 1]}
>>> _.keys().classical()
Root lattice of the Root system of type ['B', 2]
>>> NonSymmetricMacdonaldPolynomials("C2~*").Y()
Lazy family (<...Y_lambdacheck...>(i))_{i in Coroot lattice of the Root␣
↪system of type ['C', 2, 1]^*}
>>> _.keys().classical()
Root lattice of the Root system of type ['C', 2]
>>> NonSymmetricMacdonaldPolynomials(["BC", Integer(3), Integer(2)]).Y()
Lazy family (<...Y_lambdacheck...>(i))_{i in Coroot lattice of the Root␣
↪system of type ['BC', 3, 2]}
```

(continues on next page)

(continued from previous page)

```
>>> _.keys().classical()
Root lattice of the Root system of type ['B', 3]
```

affine_lift(*mu*)

Return the affinization of μ in L'.

INPUT:

- `mu` – a classical weight μ

See also

- *hecke_algebra_representation.CherednikOperatorsEigenvectors.affine_lift()*
- *affine_retract()*
- *L_prime()*

EXAMPLES:

In the untwisted case, this is the other affinization at level 1:

```
sage: E = NonSymmetricMacdonaldPolynomials("B2~")
sage: L0 = E.keys(); L0
Ambient space of the Root system of type ['B', 2]
sage: omega = L0.fundamental_weights()
sage: E.affine_lift(omega[1])
e[0] + e['deltacheck']
sage: E.affine_lift(omega[1]).parent()
Coambient space of the Root system of type ['C', 2, 1]
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials("B2~")
>>> L0 = E.keys(); L0
Ambient space of the Root system of type ['B', 2]
>>> omega = L0.fundamental_weights()
>>> E.affine_lift(omega[Integer(1)])
e[0] + e['deltacheck']
>>> E.affine_lift(omega[Integer(1)]).parent()
Coambient space of the Root system of type ['C', 2, 1]
```

In the twisted case, this is the usual affinization at level 1:

```
sage: E = NonSymmetricMacdonaldPolynomials("B2~*")
sage: L0 = E.keys(); L0
Ambient space of the Root system of type ['C', 2]
sage: omega = L0.fundamental_weights()
sage: E.affine_lift(omega[1])
e[0] + e['deltacheck']
sage: E.affine_lift(omega[1]).parent()
Ambient space of the Root system of type ['B', 2, 1]^*
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials("B2~*")
>>> L0 = E.keys(); L0
Ambient space of the Root system of type ['C', 2]
>>> omega = L0.fundamental_weights()
>>> E.affine_lift(omega[Integer(1)])
e[0] + e['deltacheck']
>>> E.affine_lift(omega[Integer(1)]).parent()
Ambient space of the Root system of type ['B', 2, 1]^*
```

affine_retract(*mu*)

Retract the affine weight μ into a classical weight.

INPUT:

- mu – an affine weight μ in L'

See also

- hecke_algebra_representation.HeckeAlgebraRepresentation.affine_retract()
- *affine_lift()*
- *L_prime()*

EXAMPLES:

```
sage: E = NonSymmetricMacdonaldPolynomials("B2~")
sage: L0 = E.keys(); L0
Ambient space of the Root system of type ['B', 2]
sage: omega = L0.fundamental_weights()
sage: E.affine_lift(omega[1])
e[0] + e['deltacheck']
sage: E.affine_retract(E.affine_lift(omega[1]))
(1, 0)
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials("B2~")
>>> L0 = E.keys(); L0
Ambient space of the Root system of type ['B', 2]
>>> omega = L0.fundamental_weights()
>>> E.affine_lift(omega[Integer(1)])
e[0] + e['deltacheck']
>>> E.affine_retract(E.affine_lift(omega[Integer(1)]))
(1, 0)
```

cartan_type()

Return Cartan type of self.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).cartan_type()
['B', 2, 1]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials(["B", Integer(2), Integer(1)]).cartan_
→type()
['B', 2, 1]
```

eigenvalue_experimental(*mu*, *l*)

Return the eigenvalue of $Y^{\lambda^\vee}$ acting on the macdonald polynomial E_μ.

INPUT:

- `mu` – the index μ of an eigenvector
- `l` – an index $\lambda^\vee$ of some Y

Note

- This method is currently not used; most tests below even test the naive method. They are left here as a basis for a future implementation.
- This is actually equivariant, as long as s_i does not fix λ.
- This method is only really needed for $\lambda^\vee = \alpha_i^\vee$ with $i = 0, ..., n$.

See Corollary 6.11 of [Haiman06].

EXAMPLES:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: q1 = t
sage: q2 = -1
sage: KL = RootSystem(["A",1,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: L0 = E.keys()
sage: E.eigenvalues(L0([0,0]))   # Checked by hand by Mark and Arun           #␣
→needs sage.libs.gap
[1/(q*t), t]
sage: alpha = E.Y().keys().simple_roots()
sage: E.eigenvalue_experimental(L0([0,0]), alpha[0]) # todo: not implemented
1/(q*t)
sage: E.eigenvalue_experimental(L0([0,0]), alpha[1])
t
```

```
>>> from sage.all import *
>>> K = QQ['q,t'].fraction_field()
>>> q,t = K.gens()
>>> q1 = t
>>> q2 = -Integer(1)
>>> KL = RootSystem(["A",Integer(1),Integer(1)]).ambient_space().algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
>>> L0 = E.keys()
>>> E.eigenvalues(L0([Integer(0),Integer(0)]))   # Checked by hand by Mark and␣
→Arun          # needs sage.libs.gap
[1/(q*t), t]
```

(continues on next page)

(continued from previous page)

```
>>> alpha = E.Y().keys().simple_roots()
>>> E.eigenvalue_experimental(L0([Integer(0),Integer(0)]), alpha[Integer(0)])
↪# todo: not implemented
1/(q*t)
>>> E.eigenvalue_experimental(L0([Integer(0),Integer(0)]), alpha[Integer(1)])
t
```

Some examples of eigenvalues (not mathematically checked!!!):

```
sage: # needs sage.libs.gap
sage: E.eigenvalues(L0([1,0]))
[t, 1/(q*t)]
sage: E.eigenvalues(L0([0,1]))
[1/(q^2*t), q*t]
sage: E.eigenvalues(L0([1,1]))
[1/(q*t), t]
sage: E.eigenvalues(L0([2,1]))
[t, 1/(q*t)]
sage: E.eigenvalues(L0([-1,1]))
[(-1)/(-q^3*t), q^2*t]
sage: E.eigenvalues(L0([-2,1]))
[(-1)/(-q^4*t), q^3*t]
sage: E.eigenvalues(L0([-2,0]))
[(-1)/(-q^3*t), q^2*t]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> E.eigenvalues(L0([Integer(1),Integer(0)]))
[t, 1/(q*t)]
>>> E.eigenvalues(L0([Integer(0),Integer(1)]))
[1/(q^2*t), q*t]
>>> E.eigenvalues(L0([Integer(1),Integer(1)]))
[1/(q*t), t]
>>> E.eigenvalues(L0([Integer(2),Integer(1)]))
[t, 1/(q*t)]
>>> E.eigenvalues(L0([-Integer(1),Integer(1)]))
[(-1)/(-q^3*t), q^2*t]
>>> E.eigenvalues(L0([-Integer(2),Integer(1)]))
[(-1)/(-q^4*t), q^3*t]
>>> E.eigenvalues(L0([-Integer(2),Integer(0)]))
[(-1)/(-q^3*t), q^2*t]
```

Some type B examples:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: q1 = t
sage: q2 = -1
sage: L = RootSystem(["B",2,1]).ambient_space()
sage: KL = L.algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: L0 = E.keys()
```

(continues on next page)

(continued from previous page)

```
sage: alpha = L.simple_coroots()

sage: # not tested
sage: E.eigenvalue(L0((0,0)), alpha[0])  # not checked
q/t
sage: E.eigenvalue(L0((1,0)), alpha[1])  # What Mark got by hand
q
sage: E.eigenvalue(L0((1,0)), alpha[2])  # not checked
t
sage: E.eigenvalue(L0((1,0)), alpha[0])  # not checked
1

sage: L = RootSystem("B2~*").ambient_space()
sage: KL = L.algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: L0 = E.keys()
sage: alpha = L.simple_coroots()
sage: E.eigenvalue(L0((0,0)), alpha[0]) # assuming Mark's calculation is␣
↪correct, one should get # not tested
1/(q*t^2)
```

```
>>> from sage.all import *
>>> K = QQ['q,t'].fraction_field()
>>> q,t = K.gens()
>>> q1 = t
>>> q2 = -Integer(1)
>>> L = RootSystem(["B",Integer(2),Integer(1)]).ambient_space()
>>> KL = L.algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
>>> L0 = E.keys()
>>> alpha = L.simple_coroots()

>>> # not tested
>>> E.eigenvalue(L0((Integer(0),Integer(0))), alpha[Integer(0)])  # not␣
↪checked
q/t
>>> E.eigenvalue(L0((Integer(1),Integer(0))), alpha[Integer(1)])  # What Mark␣
↪got by hand
q
>>> E.eigenvalue(L0((Integer(1),Integer(0))), alpha[Integer(2)])  # not␣
↪checked
t
>>> E.eigenvalue(L0((Integer(1),Integer(0))), alpha[Integer(0)])  # not␣
↪checked
1

>>> L = RootSystem("B2~*").ambient_space()
>>> KL = L.algebra(K)
>>> E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
>>> L0 = E.keys()
>>> alpha = L.simple_coroots()
>>> E.eigenvalue(L0((Integer(0),Integer(0))), alpha[Integer(0)]) # assuming␣
```

(continues on next page)

(continued from previous page)

```
↪Mark's calculation is correct, one should get # not tested
1/(q*t^2)
```

The expected value can more or less be read off from equation (37), Corollary 6.15 of [Haiman06]

Todo

- Use proposition 6.9 of [Haiman06] to check the action of the Y s on monomials.
- Generalize to any q_1, q_2.
- Check claim by Mark: all scalar products should occur in the finite weight lattice, with alpha 0 being the appropriate projection of the affine alpha 0. Question: can this be emulated by being at level 0?

rho_prime()

Return the level 0 sum of the classical fundamental weights in L'.

See also

L_prime()

EXAMPLES:

Untwisted case:

```
sage: NonSymmetricMacdonaldPolynomials("B2~").rho_prime()   # CHECKME
3/2*e[0] + 1/2*e[1]
sage: NonSymmetricMacdonaldPolynomials("B2~").rho_prime().parent()
Coambient space of the Root system of type ['C', 2, 1]
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials("B2~").rho_prime()   # CHECKME
3/2*e[0] + 1/2*e[1]
>>> NonSymmetricMacdonaldPolynomials("B2~").rho_prime().parent()
Coambient space of the Root system of type ['C', 2, 1]
```

Twisted case:

```
sage: NonSymmetricMacdonaldPolynomials("B2~*").rho_prime()   # CHECKME
2*e[0] + e[1]
sage: NonSymmetricMacdonaldPolynomials("B2~*").rho_prime().parent()
Ambient space of the Root system of type ['B', 2, 1]^*
```

```
>>> from sage.all import *
>>> NonSymmetricMacdonaldPolynomials("B2~*").rho_prime()   # CHECKME
2*e[0] + e[1]
>>> NonSymmetricMacdonaldPolynomials("B2~*").rho_prime().parent()
Ambient space of the Root system of type ['B', 2, 1]^*
```

seed(*mu*)

Return E_μ for μ minuscule, i.e. in the fundamental alcove.

INPUT:

- mu – the index μ of an eigenvector

EXAMPLES:

```
sage: E = NonSymmetricMacdonaldPolynomials(["A",2,1])
sage: omega = E.keys().fundamental_weights()
sage: E.seed(omega[1])
B[(1, 0, 0)]
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials(["A",Integer(2),Integer(1)])
>>> omega = E.keys().fundamental_weights()
>>> E.seed(omega[Integer(1)])
B[(1, 0, 0)]
```

symmetric_macdonald_polynomial(*mu*)

Return the symmetric Macdonald polynomial indexed by μ.

INPUT:

- mu – a dominant weight μ

> **Warning**
>
> The result is Weyl-symmetric only for Hecke parameters of the form $q_1 = v$ and $q_2 = -1/v$. In general the value of v below, should be the square root of $-q_1/q_2$, but the use of $q_1 = t$ and $q_2 = -1$ results in nonintegral powers of t.

EXAMPLES:

```
sage: K = QQ['q,v,t'].fraction_field()
sage: q,v,t = K.gens()
sage: E = NonSymmetricMacdonaldPolynomials(['A',2,1], q, v, -1/v)
sage: om = E.L0().fundamental_weights()
sage: E.symmetric_macdonald_polynomial(om[2])                          #␣
↪needs sage.libs.gap
B[(1, 1, 0)] + B[(1, 0, 1)] + B[(0, 1, 1)]
sage: E.symmetric_macdonald_polynomial(2*om[1])                        #␣
↪needs sage.libs.gap
((q*v^2+v^2-q-1)/(q*v^2-1))*B[(1, 1, 0)]
+ ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(1, 0, 1)] + B[(2, 0, 0)]
+ ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(0, 1, 1)] + B[(0, 2, 0)] + B[(0, 0, 2)]
sage: f = E.symmetric_macdonald_polynomial(E.L0()((2,1,0))); f        #␣
↪needs sage.libs.gap
((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, 1, 1)] + B[(1, 2, 0)]
+ B[(1, 0, 2)] + B[(2, 1, 0)] + B[(2, 0, 1)] + B[(0, 1, 2)] + B[(0, 2, 1)]
```

```
>>> from sage.all import *
>>> K = QQ['q,v,t'].fraction_field()
>>> q,v,t = K.gens()
>>> E = NonSymmetricMacdonaldPolynomials(['A',Integer(2),Integer(1)], q, v, -
↪Integer(1)/v)
```

(continues on next page)

(continued from previous page)

```
>>> om = E.L0().fundamental_weights()
>>> E.symmetric_macdonald_polynomial(om[Integer(2)])                    ␣
↪     # needs sage.libs.gap
B[(1, 1, 0)] + B[(1, 0, 1)] + B[(0, 1, 1)]
>>> E.symmetric_macdonald_polynomial(Integer(2)*om[Integer(1)])         ␣
↪              # needs sage.libs.gap
((q*v^2+v^2-q-1)/(q*v^2-1))*B[(1, 1, 0)]
+ ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(1, 0, 1)] + B[(2, 0, 0)]
+ ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(0, 1, 1)] + B[(0, 2, 0)] + B[(0, 0, 2)]
>>> f = E.symmetric_macdonald_polynomial(E.L0()((Integer(2),Integer(1),
↪Integer(0)))); f              # needs sage.libs.gap
((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, 1, 1)] + B[(1, 2, 0)]
+ B[(1, 0, 2)] + B[(2, 1, 0)] + B[(2, 0, 1)] + B[(0, 1, 2)] + B[(0, 2, 1)]
```

We compare with the type A Macdonald polynomials coming from symmetric functions:

```
sage: # needs sage.combinat
sage: P = SymmetricFunctions(K).macdonald().P()
sage: g = P[2,1].expand(3); g
x0^2*x1 + x0*x1^2 + x0^2*x2
+ (2*q*t^2 - q*t - q  + t^2 + t - 2)/(q*t^2 - 1)*x0*x1*x2
+ x1^2*x2 + x0*x2^2 + x1*x2^2
sage: fe = f.expand(g.parent().gens()); fe                              #␣
↪needs sage.libs.gap
x0^2*x1 + x0*x1^2 + x0^2*x2
+ (2*q*v^4 - q*v^2 - q + v^4 + v^2 - 2)/(q*v^4 - 1)*x0*x1*x2
+ x1^2*x2 + x0*x2^2 + x1*x2^2
sage: g.map_coefficients(lambda x: x.subs(t=v*v)) == fe                 #␣
↪needs sage.libs.gap
True

sage: E = NonSymmetricMacdonaldPolynomials(['C',3,1], q, v, -1/v)
sage: om = E.L0().fundamental_weights()
sage: E.symmetric_macdonald_polynomial(om[1]+om[2])
B[(-2, -1, 0)] + B[(-2, 1, 0)] + B[(-2, 0, -1)] + B[(-2, 0, 1)]
+ ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(q^3*v^14-q^2*v^10-q*v^4+1))*B[(-
↪1, 0, 0)]
+ B[(-1, -2, 0)] + ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(-1, -1, -1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(-1, -1, 1)] + ((2*q*v^4+v^4-q*v^
↪2+v^2-q-2)/(q*v^4-1))*B[(-1, 1, -1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(-1, 1, 1)] + B[(-1, 2, 0)] + B[(-
↪1, 0, -2)] + B[(-1, 0, 2)]
+ ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(q^3*v^14-q^2*v^10-q*v^
↪4+1))*B[(1, 0, 0)] + B[(1, -2, 0)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, -1, -1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, -1, 1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, 1, -1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, 1, 1)] + B[(1, 2, 0)] + B[(1,␣
↪0, -2)] + B[(1, 0, 2)]
+ B[(2, -1, 0)] + B[(2, 1, 0)] + B[(2, 0, -1)] + B[(2, 0, 1)] + B[(0, -2, -
```

(continues on next page)

(continued from previous page)

```
↪1)] + B[(0, -2, 1)]
- ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(-q^3*v^14+q^2*v^10+q*v^4-
↪1))*B[(0, -1, 0)]
+ B[(0, -1, -2)]
+ B[(0, -1, 2)]
- ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(-q^3*v^14+q^2*v^10+q*v^4-
↪1))*B[(0, 1, 0)]
+ B[(0, 1, -2)] + B[(0, 1, 2)] + B[(0, 2, -1)] + B[(0, 2, 1)]
+ ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(q^3*v^14-q^2*v^10-q*v^
↪4+1))*B[(0, 0, -1)]
+ ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(q^3*v^14-q^2*v^10-q*v^
↪4+1))*B[(0, 0, 1)]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = SymmetricFunctions(K).macdonald().P()
>>> g = P[Integer(2),Integer(1)].expand(Integer(3)); g
x0^2*x1 + x0*x1^2 + x0^2*x2
+ (2*q*t^2 - q*t - q  + t^2 + t - 2)/(q*t^2 - 1)*x0*x1*x2
+ x1^2*x2 + x0*x2^2 + x1*x2^2
>>> fe = f.expand(g.parent().gens()); fe                                   #␣
↪needs sage.libs.gap
x0^2*x1 + x0*x1^2 + x0^2*x2
+ (2*q*v^4 - q*v^2 - q + v^4 + v^2 - 2)/(q*v^4 - 1)*x0*x1*x2
+ x1^2*x2 + x0*x2^2 + x1*x2^2
>>> g.map_coefficients(lambda x: x.subs(t=v*v)) == fe                     #␣
↪needs sage.libs.gap
True

>>> E = NonSymmetricMacdonaldPolynomials(['C',Integer(3),Integer(1)], q, v, -
↪Integer(1)/v)
>>> om = E.L0().fundamental_weights()
>>> E.symmetric_macdonald_polynomial(om[Integer(1)]+om[Integer(2)])
B[(-2, -1, 0)] + B[(-2, 1, 0)] + B[(-2, 0, -1)] + B[(-2, 0, 1)]
+ ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(q^3*v^14-q^2*v^10-q*v^4+1))*B[(-
↪1, 0, 0)]
+ B[(-1, -2, 0)] + ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(-1, -1, -1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(-1, -1, 1)] + ((2*q*v^4+v^4-q*v^
↪2+v^2-q-2)/(q*v^4-1))*B[(-1, 1, -1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(-1, 1, 1)] + B[(-1, 2, 0)] + B[(-
↪1, 0, -2)] + B[(-1, 0, 2)]
+ ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(q^3*v^14-q^2*v^10-q*v^
↪4+1))*B[(1, 0, 0)] + B[(1, -2, 0)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, -1, -1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, -1, 1)]
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, 1, -1)]
```

(continues on next page)

(continued from previous page)

```
+ ((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, 1, 1)] + B[(1, 2, 0)] + B[(1,␣
↪0, -2)] + B[(1, 0, 2)]
+ B[(2, -1, 0)] + B[(2, 1, 0)] + B[(2, 0, -1)] + B[(2, 0, 1)] + B[(0, -2, -
↪1)] + B[(0, -2, 1)]
- ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(-q^3*v^14+q^2*v^10+q*v^4-
↪1))*B[(0, -1, 0)]
+ B[(0, -1, -2)]
+ B[(0, -1, 2)]
- ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(-q^3*v^14+q^2*v^10+q*v^4-
↪1))*B[(0, 1, 0)]
+ B[(0, 1, -2)] + B[(0, 1, 2)] + B[(0, 2, -1)] + B[(0, 2, 1)]
+ ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(q^3*v^14-q^2*v^10-q*v^
↪4+1))*B[(0, 0, -1)]
+ ((4*q^3*v^14+2*q^2*v^14-2*q^3*v^12+2*q^2*v^12-2*q^3*v^10+q*v^12-5*q^2*v^10-
↪5*q*v^4+q^2*v^2-2*v^4+2*q*v^2-2*v^2+2*q+4)/(q^3*v^14-q^2*v^10-q*v^
↪4+1))*B[(0, 0, 1)]
```

An example for type G:

```
sage: E = NonSymmetricMacdonaldPolynomials(['G',2,1], q, v, -1/v)
sage: om = E.L0().fundamental_weights()
sage: E.symmetric_macdonald_polynomial(2*om[1])
((3*q^6*v^22+3*q^5*v^22-3*q^6*v^20+q^4*v^22-4*q^5*v^20+q^4*v^18-q^5*v^16+q^
↪3*v^18-2*q^4*v^16+q^5*v^14-q^3*v^16+q^4*v^14-4*q^4*v^12+q^2*v^14+q^5*v^10-
↪8*q^3*v^12+4*q^4*v^10-4*q^2*v^12+8*q^3*v^10-q*v^12-q^4*v^8+4*q^2*v^10-q^2*v^
↪8+q^3*v^6-q*v^8+2*q^2*v^6-q^3*v^4+q*v^6-q^2*v^4+4*q*v^2-q^2+3*v^2-3*q-3)/(q^
↪6*v^22-q^5*v^20-q^4*v^12-q^3*v^12+q^3*v^10+q^2*v^10+q*v^2-1))*B[(0, 0, 0)]
+ ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(-2, 1, 1)] + B[(-2, 2, 0)] + B[(-2, 0, 2)]
- ((q*v^2+v^2-q-1)/(-q*v^2+1))*B[(-1, -1, 2)]
+ ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(q^4*v^12-q^3*v^10-q*v^
↪2+1))*B[(-1, 1, 0)]
- ((q*v^2+v^2-q-1)/(-q*v^2+1))*B[(-1, 2, -1)]
+ ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(q^4*v^12-q^3*v^10-q*v^
↪2+1))*B[(-1, 0, 1)]
- ((q*v^2+v^2-q-1)/(-q*v^2+1))*B[(1, -2, 1)]
- ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(-q^4*v^12+q^3*v^10+q*v^2-
↪1))*B[(1, -1, 0)]
- ((q*v^2+v^2-q-1)/(-q*v^2+1))*B[(1, 1, -2)]
- ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(-q^4*v^12+q^3*v^10+q*v^2-
↪1))*B[(1, 0, -1)]
+ B[(2, -2, 0)] + ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(2, -1, -1)] + B[(2, 0, -2)]␣
↪+ B[(0, -2, 2)]
- ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(-q^4*v^12+q^3*v^10+q*v^2-
↪1))*B[(0, -1, 1)]
```

(continues on next page)

(continued from previous page)

```
+ ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(q^4*v^12-q^3*v^10-q*v^
↪2+1))*B[(0, 1, -1)]
+ B[(0, 2, -2)]
```

```
>>> from sage.all import *
>>> E = NonSymmetricMacdonaldPolynomials(['G',Integer(2),Integer(1)], q, v, -
↪Integer(1)/v)
>>> om = E.L0().fundamental_weights()
>>> E.symmetric_macdonald_polynomial(Integer(2)*om[Integer(1)])
((3*q^6*v^22+3*q^5*v^22-3*q^6*v^20+q^4*v^22-4*q^5*v^20+q^4*v^18-q^5*v^16+q^
↪3*v^18-2*q^4*v^16+q^5*v^14-q^3*v^16+q^4*v^14-4*q^4*v^12+q^2*v^14+q^5*v^10-
↪8*q^3*v^12+4*q^4*v^10-4*q^2*v^12+8*q^3*v^10-q*v^12-q^4*v^8+4*q^2*v^10-q^2*v^
↪8+q^3*v^6-q*v^8+2*q^2*v^6-q^3*v^4+q*v^6-q^2*v^4+4*q*v^2-q^2+3*v^2-3*q-3)/(q^
↪6*v^22-q^5*v^20-q^4*v^12-q^3*v^12+q^3*v^10+q^2*v^10+q*v^2-1))*B[(0, 0, 0)]
+ ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(-2, 1, 1)] + B[(-2, 2, 0)] + B[(-2, 0, 2)]
- ((q*v^2+v^2-q-1)/(-q*v^2+1))*B[(-1, -1, 2)]
+ ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(q^4*v^12-q^3*v^10-q*v^
↪2+1))*B[(-1, 1, 0)]
- ((q*v^2+v^2-q-1)/(-q*v^2+1))*B[(-1, 2, -1)]
+ ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(q^4*v^12-q^3*v^10-q*v^
↪2+1))*B[(-1, 0, 1)]
- ((q*v^2+v^2-q-1)/(-q*v^2+1))*B[(1, -2, 1)]
- ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(-q^4*v^12+q^3*v^10+q*v^2-
↪1))*B[(1, -1, 0)]
- ((q*v^2+v^2-q-1)/(-q*v^2+1))*B[(1, 1, -2)]
- ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(-q^4*v^12+q^3*v^10+q*v^2-
↪1))*B[(1, 0, -1)]
+ B[(2, -2, 0)] + ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(2, -1, -1)] + B[(2, 0, -2)]␣
↪+ B[(0, -2, 2)]
- ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(-q^4*v^12+q^3*v^10+q*v^2-
↪1))*B[(0, -1, 1)]
+ ((2*q^4*v^12+2*q^3*v^12-2*q^4*v^10-2*q^3*v^10+q^2*v^8-q^3*v^6+q*v^8-2*q^2*v^
↪6+q^3*v^4-q*v^6+q^2*v^4-2*q*v^2-2*v^2+2*q+2)/(q^4*v^12-q^3*v^10-q*v^
↪2+1))*B[(0, 1, -1)]
+ B[(0, 2, -2)]
```

twist (*mu*, *i*)

Act by s_i on the affine weight μ.

This calls `simple_reflection`; which is semantically the same as the default implementation.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: W = WeylGroup(["B",3])
sage: W.element_class._repr_ = lambda x: "".join(str(i)
....:                                            for i in x.reduced_word())
```

(continues on next page)

(continued from previous page)

```
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: T = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: E = T.Y_eigenvectors()
sage: w = W.an_element(); w
123
sage: E.twist(w,1)
1231
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> W = WeylGroup(["B",Integer(3)])
>>> W.element_class._repr_ = lambda x: "".join(str(i)
...                                            for i in x.reduced_word())
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> T = KW.demazure_lusztig_operators(q1, q2, affine=True)
>>> E = T.Y_eigenvectors()
>>> w = W.an_element(); w
123
>>> E.twist(w,Integer(1))
1231
```

5.1.234 Pieri Factors

class sage.combinat.root_system.pieri_factors.**PieriFactors**

Bases: UniqueRepresentation, Parent

An abstract class for sets of Pieri factors, used for constructing Stanley symmetric functions. The set of Pieri factors for a given type can be realized as an order ideal of the Bruhat order poset generated by a certain set of maximal elements.

See also

- WeylGroups.ParentMethods.pieri_factors()
- WeylGroups.ElementMethods.stanley_symmetric_function()

EXAMPLES:

```
sage: W = WeylGroup(['A',4])
sage: PF = W.pieri_factors()
sage: PF.an_element().reduced_word()
[4, 3, 2, 1]
sage: Waff = WeylGroup(['A',4,1])
sage: PFaff = Waff.pieri_factors()
sage: Waff.from_reduced_word(PF.an_element().reduced_word()) in PFaff
True
```

(continues on next page)

(continued from previous page)

```
sage: W = WeylGroup(['B',3,1])
sage: PF = W.pieri_factors()
sage: W.from_reduced_word([2,3,2]) in PF.elements()
True
sage: PF.cardinality()
47

sage: W = WeylGroup(['C',3,1])
sage: PF = W.pieri_factors()
sage: PF.generating_series()
6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1
sage: sorted(w.reduced_word() for w in PF if w.length() == 2)
[[0, 1], [1, 0], [1, 2], [2, 0], [2, 1],
[2, 3], [3, 0], [3, 1], [3, 2]]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)])
>>> PF = W.pieri_factors()
>>> PF.an_element().reduced_word()
[4, 3, 2, 1]
>>> Waff = WeylGroup(['A',Integer(4),Integer(1)])
>>> PFaff = Waff.pieri_factors()
>>> Waff.from_reduced_word(PF.an_element().reduced_word()) in PFaff
True

>>> W = WeylGroup(['B',Integer(3),Integer(1)])
>>> PF = W.pieri_factors()
>>> W.from_reduced_word([Integer(2),Integer(3),Integer(2)]) in PF.elements()
True
>>> PF.cardinality()
47

>>> W = WeylGroup(['C',Integer(3),Integer(1)])
>>> PF = W.pieri_factors()
>>> PF.generating_series()
6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1
>>> sorted(w.reduced_word() for w in PF if w.length() == Integer(2))
[[0, 1], [1, 0], [1, 2], [2, 0], [2, 1],
[2, 3], [3, 0], [3, 1], [3, 2]]
```

REFERENCES:

- [FoSta1994]
- [BH1994]
- [Lam1996]
- [Lam2008]
- [LSS2009]
- [Pon2010]

default_weight()

Return the function $i \mapsto z^i$, where z is the generator of `QQ['z']`.

EXAMPLES:

```
sage: W = WeylGroup(["A", 3, 1])
sage: weight = W.pieri_factors().default_weight()
sage: weight(1)
z
sage: weight(5)
z^5
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A", Integer(3), Integer(1)])
>>> weight = W.pieri_factors().default_weight()
>>> weight(Integer(1))
z
>>> weight(Integer(5))
z^5
```

elements()

Return the elements of `self`.

Those are constructed as the elements below the maximal elements of `self` in Bruhat order.

OUTPUT: a `RecursivelyEnumeratedSet_generic` object

EXAMPLES:

```
sage: PF = WeylGroup(['A',3]).pieri_factors()
sage: sorted(w.reduced_word() for w in PF.elements())
[[], [1], [2], [2, 1], [3], [3, 1], [3, 2], [3, 2, 1]]
```

```
>>> from sage.all import *
>>> PF = WeylGroup(['A',Integer(3)]).pieri_factors()
>>> sorted(w.reduced_word() for w in PF.elements())
[[], [1], [2], [2, 1], [3], [3, 1], [3, 2], [3, 2, 1]]
```

See also

`maximal_elements()`

Todo

Possibly remove this method and instead have this class inherit from `RecursivelyEnumeratedSet_generic`.

generating_series(*weight=None*)

Return a length generating series for the elements of `self`.

EXAMPLES:

```
sage: PF = WeylGroup(['C',3,1]).pieri_factors()
sage: PF.generating_series()
6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1
```

(continues on next page)

(continued from previous page)

```
sage: PF = WeylGroup(['B',4]).pieri_factors()
sage: PF.generating_series()
z^7 + 6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1
```

```
>>> from sage.all import *
>>> PF = WeylGroup(['C',Integer(3),Integer(1)]).pieri_factors()
>>> PF.generating_series()
6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1

>>> PF = WeylGroup(['B',Integer(4)]).pieri_factors()
>>> PF.generating_series()
z^7 + 6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1
```

max_length()

Return the maximal length of a Pieri factor.

EXAMPLES:

In type A and A affine, this is n:

```
sage: WeylGroup(['A',5]).pieri_factors().max_length()
5
sage: WeylGroup(['A',5,1]).pieri_factors().max_length()
5
```

```
>>> from sage.all import *
>>> WeylGroup(['A',Integer(5)]).pieri_factors().max_length()
5
>>> WeylGroup(['A',Integer(5),Integer(1)]).pieri_factors().max_length()
5
```

In type B and B affine, this is $2n - 1$:

```
sage: WeylGroup(['B',5,1]).pieri_factors().max_length()
9
sage: WeylGroup(['B',5]).pieri_factors().max_length()
9
```

```
>>> from sage.all import *
>>> WeylGroup(['B',Integer(5),Integer(1)]).pieri_factors().max_length()
9
>>> WeylGroup(['B',Integer(5)]).pieri_factors().max_length()
9
```

In type C affine this is $2n$:

```
sage: WeylGroup(['C',5,1]).pieri_factors().max_length()
10
```

```
>>> from sage.all import *
>>> WeylGroup(['C',Integer(5),Integer(1)]).pieri_factors().max_length()
10
```

In type D affine this is $2n - 2$:

```
sage: WeylGroup(['D',5,1]).pieri_factors().max_length()
8
```

```
>>> from sage.all import *
>>> WeylGroup(['D',Integer(5),Integer(1)]).pieri_factors().max_length()
8
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_affine_type**

Bases: *PieriFactors*

maximal_elements()

Return the maximal elements of `self` with respect to Bruhat order.

The current implementation is via a conjectural type-free formula. Use `maximal_elements_combinatorial()` for proven type-specific implementations. To compare type-free and type-specific (combinatorial) implementations, use method `_test_maximal_elements()`.

EXAMPLES:

```
sage: W = WeylGroup(['A',4,1])
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 4, 3, 2], [1, 0, 4, 3], [2, 1, 0, 4], [3, 2, 1, 0], [4, 3, 2, 1]]

sage: W = WeylGroup(RootSystem(["C",3,1]).weight_space())
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [1, 2, 3, 2, 1, 0],
 [2, 1, 0, 1, 2, 3], [2, 3, 2, 1, 0, 1], [3, 2, 1, 0, 1, 2]]

sage: W = WeylGroup(RootSystem(["B",3,1]).weight_space())
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 2, 3, 2, 0], [1, 0, 2, 3, 2], [1, 2, 3, 2, 1],
 [2, 1, 0, 2, 3], [2, 3, 2, 1, 0], [3, 2, 1, 0, 2]]

sage: W = WeylGroup(['D',4,1])
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 2, 4, 3, 2, 0], [1, 0, 2, 4, 3, 2], [1, 2, 4, 3, 2, 1],
 [2, 1, 0, 2, 4, 3], [2, 4, 3, 2, 1, 0], [3, 2, 1, 0, 2, 3],
 [4, 2, 1, 0, 2, 4], [4, 3, 2, 1, 0, 2]]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4),Integer(1)])
>>> PF = W.pieri_factors()
>>> sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 4, 3, 2], [1, 0, 4, 3], [2, 1, 0, 4], [3, 2, 1, 0], [4, 3, 2, 1]]

>>> W = WeylGroup(RootSystem(["C",Integer(3),Integer(1)]).weight_space())
>>> PF = W.pieri_factors()
>>> sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
```

(continues on next page)

(continued from previous page)

```
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [1, 2, 3, 2, 1, 0],
 [2, 1, 0, 1, 2, 3], [2, 3, 2, 1, 0, 1], [3, 2, 1, 0, 1, 2]]

>>> W = WeylGroup(RootSystem(["B",Integer(3),Integer(1)]).weight_space())
>>> PF = W.pieri_factors()
>>> sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 2, 3, 2, 0], [1, 0, 2, 3, 2], [1, 2, 3, 2, 1],
 [2, 1, 0, 2, 3], [2, 3, 2, 1, 0], [3, 2, 1, 0, 2]]

>>> W = WeylGroup(['D',Integer(4),Integer(1)])
>>> PF = W.pieri_factors()
>>> sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 2, 4, 3, 2, 0], [1, 0, 2, 4, 3, 2], [1, 2, 4, 3, 2, 1],
 [2, 1, 0, 2, 4, 3], [2, 4, 3, 2, 1, 0], [3, 2, 1, 0, 2, 3],
 [4, 2, 1, 0, 2, 4], [4, 3, 2, 1, 0, 2]]
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_finite_type**

Bases: *PieriFactors*

The Pieri factors of finite type A are the restriction of the Pieri factors of affine type A to finite permutations (under the canonical embedding of finite type A into the affine Weyl group), and the Pieri factors of finite type B are the restriction of the Pieri factors of affine type C. The finite type D Pieri factors are (weakly) conjectured to be the restriction of the Pieri factors of affine type D.

maximal_elements()

The current algorithm uses the fact that the maximal Pieri factors of affine type A,B,C, or D either contain a finite Weyl group element, or contain an affine Weyl group element whose reflection by s_0 gets a finite Weyl group element, and that either of these finite group elements will serve as a maximal element for finite Pieri factors. A better algorithm is desirable.

EXAMPLES:

```
sage: PF = WeylGroup(['A',5]).pieri_factors()
sage: [v.reduced_word() for v in PF.maximal_elements()]
[[5, 4, 3, 2, 1]]

sage: WeylGroup(['B',4]).pieri_factors().maximal_elements()
[
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
]
```

```
>>> from sage.all import *
>>> PF = WeylGroup(['A',Integer(5)]).pieri_factors()
>>> [v.reduced_word() for v in PF.maximal_elements()]
[[5, 4, 3, 2, 1]]

>>> WeylGroup(['B',Integer(4)]).pieri_factors().maximal_elements()
[
[-1  0  0  0]
[ 0  1  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  1  0]
[ 0  0  0  1]
]
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_A**(*W*)

Bases: *PieriFactors_finite_type*

The set of Pieri factors for finite type A.

This is the set of elements of the Weyl group that have a reduced word that is strictly decreasing. This may also be viewed as the restriction of affine type A Pieri factors to finite Weyl group elements.

maximal_elements_combinatorial()

Return the maximal Pieri factors, using the type A combinatorial description.

EXAMPLES:

```
sage: W = WeylGroup(['A',4])
sage: PF = W.pieri_factors()
sage: PF.maximal_elements_combinatorial()[0].reduced_word()
[4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)])
>>> PF = W.pieri_factors()
>>> PF.maximal_elements_combinatorial()[Integer(0)].reduced_word()
[4, 3, 2, 1]
```

stanley_symm_poly_weight(*w*)

EXAMPLES:

```
sage: W = WeylGroup(['A',4])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([3,1]))
0
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)])
>>> PF = W.pieri_factors()
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(3),Integer(1)]))
0
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_A_affine**(*W*, *min_length*, *max_length*, *min_support*, *max_support*)

Bases: *PieriFactors_affine_type*

The set of Pieri factors for type A affine, that is the set of elements of the Weyl Group which are cyclically decreasing.

Those are used for constructing (affine) Stanley symmetric functions.

The Pieri factors are in bijection with the proper subsets of the `index_set`. The bijection is given by the support. Namely, let f be a Pieri factor, and red a reduced word for f. No simple reflection appears twice in red, and the support S of red (that is the i such that s_i appears in red) does not depend on the reduced word).

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: WeylGroup(["A", 3, 1]).pieri_factors().cardinality()
15
```

```
>>> from sage.all import *
>>> WeylGroup(["A", Integer(3), Integer(1)]).pieri_factors().cardinality()
15
```

generating_series(*weight=None*)

Return a length generating series for the elements of `self`.

EXAMPLES:

```
sage: W = WeylGroup(["A", 3, 1])
sage: W.pieri_factors().cardinality()
15
sage: W.pieri_factors().generating_series()
4*z^3 + 6*z^2 + 4*z + 1
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A", Integer(3), Integer(1)])
>>> W.pieri_factors().cardinality()
15
>>> W.pieri_factors().generating_series()
4*z^3 + 6*z^2 + 4*z + 1
```

maximal_elements_combinatorial()

Return the maximal Pieri factors, using the affine type A combinatorial description.

EXAMPLES:

```
sage: W = WeylGroup(['A',4,1])
sage: PF = W.pieri_factors()
sage: [w.reduced_word() for w in PF.maximal_elements_combinatorial()]
[[3, 2, 1, 0], [2, 1, 0, 4], [1, 0, 4, 3], [0, 4, 3, 2], [4, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4),Integer(1)])
>>> PF = W.pieri_factors()
>>> [w.reduced_word() for w in PF.maximal_elements_combinatorial()]
[[3, 2, 1, 0], [2, 1, 0, 4], [1, 0, 4, 3], [0, 4, 3, 2], [4, 3, 2, 1]]
```

stanley_symm_poly_weight(*w*)

Weight used in computing (affine) Stanley symmetric polynomials for affine type A.

EXAMPLES:

```
sage: W = WeylGroup(['A',5,1])
sage: PF = W.pieri_factors()
```

(continues on next page)

(continued from previous page)

```
sage: PF.stanley_symm_poly_weight(W.one())
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,4,2,1,0]))
0
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(5),Integer(1)])
>>> PF = W.pieri_factors()
>>> PF.stanley_symm_poly_weight(W.one())
0
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(5),Integer(4),
↪Integer(2),Integer(1),Integer(0)]))
0
```

subset(*length*)

Return the subset of the elements of `self` of length `length`.

INPUT:

- `length` – nonnegative integer

EXAMPLES:

```
sage: PF = WeylGroup(["A", 3, 1]).pieri_factors(); PF
Pieri factors for Weyl Group of type ['A', 3, 1] (as a matrix group acting on␣
↪the root space)
sage: PF3 = PF.subset(length = 2)
sage: PF3.cardinality()
6
```

```
>>> from sage.all import *
>>> PF = WeylGroup(["A", Integer(3), Integer(1)]).pieri_factors(); PF
Pieri factors for Weyl Group of type ['A', 3, 1] (as a matrix group acting on␣
↪the root space)
>>> PF3 = PF.subset(length = Integer(2))
>>> PF3.cardinality()
6
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_B**(*W*)

Bases: *PieriFactors_finite_type*

The type B finite Pieri factors are realized as the set of elements that have a reduced word that is a subword of $12...(n-1)n(n-1)...21$. They are the restriction of the type C affine Pieri factors to the set of finite Weyl group elements under the usual embedding.

maximal_elements_combinatorial()

Return the maximal Pieri factors, using the type B combinatorial description.

EXAMPLES:

```
sage: PF = WeylGroup(['B',4]).pieri_factors()
sage: PF.maximal_elements_combinatorial()[0].reduced_word()
[1, 2, 3, 4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> PF = WeylGroup(['B',Integer(4)]).pieri_factors()
>>> PF.maximal_elements_combinatorial()[Integer(0)].reduced_word()
[1, 2, 3, 4, 3, 2, 1]
```

stanley_symm_poly_weight(*w*)

Weight used in computing Stanley symmetric polynomials of type B.

The weight for finite type B is the number of components of the support of an element minus the number of occurrences of n in a reduced word.

EXAMPLES:

```
sage: W = WeylGroup(['B',5])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([3,1,5]))
2
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([3,4,5]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([1,2,3,4,5,4]))
0
```

```
>>> from sage.all import *
>>> W = WeylGroup(['B',Integer(5)])
>>> PF = W.pieri_factors()
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(3),Integer(1),
↪Integer(5)]))
2
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(3),Integer(4),
↪Integer(5)]))
0
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(1),Integer(2),
↪Integer(3),Integer(4),Integer(5),Integer(4)]))
0
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_B_affine**(*W*)

Bases: *PieriFactors_affine_type*

The type B affine Pieri factors are realized as the order ideal (in Bruhat order) generated by the following elements:

- cyclic rotations of the element with reduced word $234...(n-1)n(n-1)...3210$, except for $123...n...320$ and $023...n...321$.
- $123...(n-1)n(n-1)...321$
- $023...(n-1)n(n-1)...320$

EXAMPLES:

```
sage: W = WeylGroup(['B',4,1])
sage: PF = W.pieri_factors()
sage: W.from_reduced_word([2,3,4,3,2,1,0]) in PF.maximal_elements()
True
sage: W.from_reduced_word([0,2,3,4,3,2,1]) in PF.maximal_elements()
False
sage: W.from_reduced_word([1,0,2,3,4,3,2]) in PF.maximal_elements()
```

(continues on next page)

(continued from previous page)

```
True
sage: W.from_reduced_word([0,2,3,4,3,2,0]) in PF.maximal_elements()
True
sage: W.from_reduced_word([0,2,0]) in PF
True
```

```
>>> from sage.all import *
>>> W = WeylGroup(['B',Integer(4),Integer(1)])
>>> PF = W.pieri_factors()
>>> W.from_reduced_word([Integer(2),Integer(3),Integer(4),Integer(3),Integer(2),
↪Integer(1),Integer(0)]) in PF.maximal_elements()
True
>>> W.from_reduced_word([Integer(0),Integer(2),Integer(3),Integer(4),Integer(3),
↪Integer(2),Integer(1)]) in PF.maximal_elements()
False
>>> W.from_reduced_word([Integer(1),Integer(0),Integer(2),Integer(3),Integer(4),
↪Integer(3),Integer(2)]) in PF.maximal_elements()
True
>>> W.from_reduced_word([Integer(0),Integer(2),Integer(3),Integer(4),Integer(3),
↪Integer(2),Integer(0)]) in PF.maximal_elements()
True
>>> W.from_reduced_word([Integer(0),Integer(2),Integer(0)]) in PF
True
```

maximal_elements_combinatorial()

Return the maximal Pieri factors, using the affine type B combinatorial description.

EXAMPLES:

```
sage: W = WeylGroup(['B',4,1])
sage: [u.reduced_word() for u in W.pieri_factors().maximal_elements_
↪combinatorial()]
[[1, 0, 2, 3, 4, 3, 2], [2, 1, 0, 2, 3, 4, 3], [3, 2, 1, 0, 2, 3, 4], [4, 3,␣
↪2, 1, 0, 2, 3], [3, 4, 3, 2, 1, 0, 2], [2, 3, 4, 3, 2, 1, 0], [1, 2, 3, 4,␣
↪3, 2, 1], [0, 2, 3, 4, 3, 2, 0]]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['B',Integer(4),Integer(1)])
>>> [u.reduced_word() for u in W.pieri_factors().maximal_elements_
↪combinatorial()]
[[1, 0, 2, 3, 4, 3, 2], [2, 1, 0, 2, 3, 4, 3], [3, 2, 1, 0, 2, 3, 4], [4, 3,␣
↪2, 1, 0, 2, 3], [3, 4, 3, 2, 1, 0, 2], [2, 3, 4, 3, 2, 1, 0], [1, 2, 3, 4,␣
↪3, 2, 1], [0, 2, 3, 4, 3, 2, 0]]
```

stanley_symm_poly_weight(*w*)

Return the weight of a Pieri factor to be used in the definition of Stanley symmetric functions.

For type B, this weight involves the number of components of the complement of the support of an element, where we consider 0 and 1 to be one node – if 1 is in the support, then we pretend 0 in the support, and vice versa. We also consider 0 and 1 to be one node for the purpose of counting components of the complement (as if the Dynkin diagram were that of type C). Let n be the rank of the affine Weyl group in question (if type `['B',k,1]` then we have n = k+1). Let `chi(v.length() < n-1)` be the indicator function that is 1 if the length of v is smaller than n-1, and 0 if the length of v is greater than or equal to n-1. If we call `c'(v)` the number of components of the complement of the support of v, then the type B weight is given by `weight = c'(v) - chi(v.length() < n-1)`.

EXAMPLES:

```
sage: W = WeylGroup(['B',5,1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([0,3]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([0,1,3]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([2,3]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([2,3,4,5]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([0,5]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([2,4,5,4,3,0]))
-1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([4,5,4,3,0]))
0
```

```
>>> from sage.all import *
>>> W = WeylGroup(['B',Integer(5),Integer(1)])
>>> PF = W.pieri_factors()
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(0),Integer(3)]))
1
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(0),Integer(1),
↪Integer(3)]))
1
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(2),Integer(3)]))
1
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(2),Integer(3),
↪Integer(4),Integer(5)]))
0
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(0),Integer(5)]))
0
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(2),Integer(4),
↪Integer(5),Integer(4),Integer(3),Integer(0)]))
-1
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(4),Integer(5),
↪Integer(4),Integer(3),Integer(0)]))
0
```

class `sage.combinat.root_system.pieri_factors.`**`PieriFactors_type_C_affine`**`(W)`

Bases: *PieriFactors_affine_type*

The type C affine Pieri factors are realized as the order ideal (in Bruhat order) generated by cyclic rotations of the element with unique reduced word $123...(n-1)n(n-1)...3210$.

EXAMPLES:

```
sage: W = WeylGroup(['C',3,1])
sage: PF = W.pieri_factors()
sage: sorted([u.reduced_word() for u in PF.maximal_elements()], key=str)
```

(continues on next page)

(continued from previous page)

```
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [1, 2, 3, 2, 1, 0],
 [2, 1, 0, 1, 2, 3], [2, 3, 2, 1, 0, 1], [3, 2, 1, 0, 1, 2]]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['C',Integer(3),Integer(1)])
>>> PF = W.pieri_factors()
>>> sorted([u.reduced_word() for u in PF.maximal_elements()], key=str)
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [1, 2, 3, 2, 1, 0],
 [2, 1, 0, 1, 2, 3], [2, 3, 2, 1, 0, 1], [3, 2, 1, 0, 1, 2]]
```

maximal_elements_combinatorial()

Return the maximal Pieri factors, using the affine type C combinatorial description.

EXAMPLES:

```
sage: PF = WeylGroup(['C',3,1]).pieri_factors()
sage: [w.reduced_word() for w in PF.maximal_elements_combinatorial()]
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [2, 1, 0, 1, 2, 3], [3, 2, 1, 0, 1,
↪2], [2, 3, 2, 1, 0, 1], [1, 2, 3, 2, 1, 0]]
```

```
>>> from sage.all import *
>>> PF = WeylGroup(['C',Integer(3),Integer(1)]).pieri_factors()
>>> [w.reduced_word() for w in PF.maximal_elements_combinatorial()]
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [2, 1, 0, 1, 2, 3], [3, 2, 1, 0, 1,
↪2], [2, 3, 2, 1, 0, 1], [1, 2, 3, 2, 1, 0]]
```

stanley_symm_poly_weight(w)

Return the weight of a Pieri factor to be used in the definition of Stanley symmetric functions.

For type C, this weight is the number of connected components of the support (the indices appearing in a reduced word) of an element.

EXAMPLES:

```
sage: W = WeylGroup(['C',5,1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([1,3]))
2
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([1,3,2,0]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,3,0]))
3
sage: PF.stanley_symm_poly_weight(W.one())
0
```

```
>>> from sage.all import *
>>> W = WeylGroup(['C',Integer(5),Integer(1)])
>>> PF = W.pieri_factors()
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(1),Integer(3)]))
2
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(1),Integer(3),
↪Integer(2),Integer(0)]))
1
```

(continues on next page)

(continued from previous page)

```
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(5),Integer(3),
↪Integer(0)]))
3
>>> PF.stanley_symm_poly_weight(W.one())
0
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_D_affine**(*W*)

Bases: *PieriFactors_affine_type*

The type D affine Pieri factors are realized as the order ideal (in Bruhat order) generated by the following elements:

- cyclic rotations of the element with reduced word $234...(n-2)n(n-1)(n-2)...3210$ such that 1 and 0 are always adjacent and (n-1) and n are always adjacent.
- $123...(n-2)n(n-1)(n-2)...321$
- $023...(n-2)n(n-1)(n-2)...320$
- $n(n-2)...2102...(n-2)n$
- $(n-1)(n-2)...2102...(n-2)(n-1)$

EXAMPLES:

```
sage: W = WeylGroup(['D',5,1])
sage: PF = W.pieri_factors()
sage: W.from_reduced_word([3,2,1,0]) in PF
True
sage: W.from_reduced_word([0,3,2,1]) in PF
False
sage: W.from_reduced_word([0,1,3,2]) in PF
True
sage: W.from_reduced_word([2,0,1,3]) in PF
True
sage: sorted([u.reduced_word() for u in PF.maximal_elements()], key=str)
[[0, 2, 3, 5, 4, 3, 2, 0], [1, 0, 2, 3, 5, 4, 3, 2], [1, 2, 3, 5, 4, 3, 2, 1],
 [2, 1, 0, 2, 3, 5, 4, 3], [2, 3, 5, 4, 3, 2, 1, 0], [3, 2, 1, 0, 2, 3, 5, 4],
 [3, 5, 4, 3, 2, 1, 0, 2], [4, 3, 2, 1, 0, 2, 3, 4], [5, 3, 2, 1, 0, 2, 3, 5],
 [5, 4, 3, 2, 1, 0, 2, 3]]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['D',Integer(5),Integer(1)])
>>> PF = W.pieri_factors()
>>> W.from_reduced_word([Integer(3),Integer(2),Integer(1),Integer(0)]) in PF
True
>>> W.from_reduced_word([Integer(0),Integer(3),Integer(2),Integer(1)]) in PF
False
>>> W.from_reduced_word([Integer(0),Integer(1),Integer(3),Integer(2)]) in PF
True
>>> W.from_reduced_word([Integer(2),Integer(0),Integer(1),Integer(3)]) in PF
True
>>> sorted([u.reduced_word() for u in PF.maximal_elements()], key=str)
[[0, 2, 3, 5, 4, 3, 2, 0], [1, 0, 2, 3, 5, 4, 3, 2], [1, 2, 3, 5, 4, 3, 2, 1],
 [2, 1, 0, 2, 3, 5, 4, 3], [2, 3, 5, 4, 3, 2, 1, 0], [3, 2, 1, 0, 2, 3, 5, 4],
```

(continues on next page)

(continued from previous page)

```
[3, 5, 4, 3, 2, 1, 0, 2], [4, 3, 2, 1, 0, 2, 3, 4], [5, 3, 2, 1, 0, 2, 3, 5],
[5, 4, 3, 2, 1, 0, 2, 3]]
```

maximal_elements_combinatorial()

Return the maximal Pieri factors, using the affine type D combinatorial description.

EXAMPLES:

```
sage: W = WeylGroup(['D',5,1])
sage: PF = W.pieri_factors()
sage: set(PF.maximal_elements_combinatorial()) == set(PF.maximal_elements())
True
```

```
>>> from sage.all import *
>>> W = WeylGroup(['D',Integer(5),Integer(1)])
>>> PF = W.pieri_factors()
>>> set(PF.maximal_elements_combinatorial()) == set(PF.maximal_elements())
True
```

stanley_symm_poly_weight(w)

Return the weight of w, to be used in the definition of Stanley symmetric functions.

INPUT:

- `w` – a Pieri factor for this type

For type D, this weight involves the number of components of the complement of the support of an element, where we consider 0 and 1 to be one node – if 1 is in the support, then we pretend 0 in the support, and vice versa. Similarly with $n-1$ and n. We also consider 0 and 1, $n-1$ and n to be one node for the purpose of counting components of the complement (as if the Dynkin diagram were that of type C).

Type D Stanley symmetric polynomial weights are still conjectural. The given weight comes from conditions on elements of the affine Fomin-Stanley subalgebra, but work is needed to show this weight is correct for affine Stanley symmetric functions – see [LSS2009, Pon2010]_ for details.

EXAMPLES:

```
sage: W = WeylGroup(['D', 5, 1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,2,1]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,2,1,0]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,2]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([]))
0

sage: W = WeylGroup(['D',7,1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([2,4,6]))
2
```

```
>>> from sage.all import *
>>> W = WeylGroup(['D', Integer(5), Integer(1)])
>>> PF = W.pieri_factors()
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(5),Integer(2),
↪Integer(1)]))
0
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(5),Integer(2),
↪Integer(1),Integer(0)]))
0
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(5),Integer(2)]))
1
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([]))
0

>>> W = WeylGroup(['D',Integer(7),Integer(1)])
>>> PF = W.pieri_factors()
>>> PF.stanley_symm_poly_weight(W.from_reduced_word([Integer(2),Integer(4),
↪Integer(6)]))
2
```

5.1.235 Tutorial: visualizing root systems

Root systems encode the positions of collections of hyperplanes in space, and form the fundamental combinatorial data underlying Coxeter and Weyl groups, Lie algebras and groups, etc. The theory can be a bit intimidating at first because of the many technical gadgets (roots, coroots, weights, …). Visualizing them goes a long way toward building a geometric intuition.

This tutorial starts from simple plots and guides you all the way to advanced plots with your own combinatorial data drawn on top of it.

See also

- *Root systems* – An overview of root systems in Sage
- `RootLatticeRealizations.ParentMethods.plot()` – the main plotting function, with pointers to all the subroutines

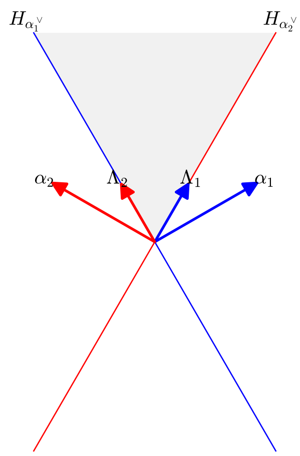
First plots

In this first plot, we draw the root system for type A_2 in the ambient space. It is generated from two hyperplanes at a 120 degree angle:

```
sage: L = RootSystem(["A",2]).ambient_space()
sage: L.plot()
Graphics object consisting of 13 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).ambient_space()
>>> L.plot()
Graphics object consisting of 13 graphics primitives
```

Each of those hyperplane $H_{\alpha_i^\vee}$ is described by a linear form $\alpha_i^\vee$ called simple coroot. To each such hyperplane corresponds a reflection along a vector called root. In this picture, the reflections are orthogonal and the two simple roots α_1 and α_2

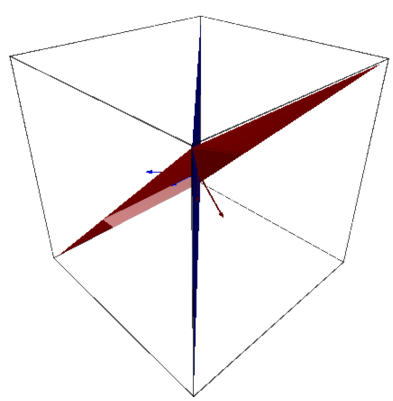


are vectors which are normal to the reflection hyperplanes. The same color code is used uniformly: blue for 1, red for 2, green for 3, … (see `CartanType.color()`). The fundamental weights, Λ_1 and Λ_2 form the dual basis of the coroots.

The two reflections generate a group of order six which is nothing but the usual symmetric group S_3, in its natural action by permutations of the coordinates of the ambient space. Wait, but the ambient space should be of dimension 3 then? That's perfectly right. Here is the full picture in 3D:

```
sage: L = RootSystem(["A",2]).ambient_space()
sage: L.plot(projection=False)
Graphics3d Object
```

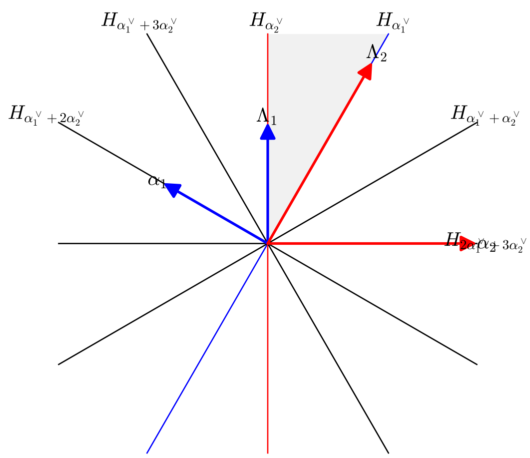
```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).ambient_space()
>>> L.plot(projection=False)
Graphics3d Object
```



However in this space, the line $(1, 1, 1)$ is fixed by the action of the group. Therefore, the so called barycentric projection orthogonal to $(1, 1, 1)$ gives a convenient 2D picture which contains all the essential information. The same projection is used by default in type G_2:

```
sage: L = RootSystem(["G",2]).ambient_space()
sage: L.plot(reflection_hyperplanes='all')
Graphics object consisting of 21 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["G",Integer(2)]).ambient_space()
>>> L.plot(reflection_hyperplanes='all')
Graphics object consisting of 21 graphics primitives
```



The group is now the dihedral group of order 12, generated by the two reflections s_1 and s_2. The picture displays the hyperplanes for all 12 reflections of the group. Those reflections delimit 12 chambers which are in one to one correspondence with the elements of the group. The fundamental chamber, which is grayed out, is associated with the identity of the group.

Warning

The fundamental chamber is currently plotted as the cone generated by the fundamental weights. As can be seen on the previous 3D picture this is not quite correct if the fundamental weights do not span the space.

Another caveat is that some plotting features may require manipulating elements with rational coordinates which will fail if one is working in, say, the weight lattice. It is therefore recommended to use the root, weight, or ambient spaces for plotting purposes rather than their lattice counterparts.

Coming back to the symmetric group, here is the picture in the weight space, with all roots and all reflection hyperplanes; remark that, unlike in the ambient space, a root is not necessarily orthogonal to its corresponding reflection hyperplane:

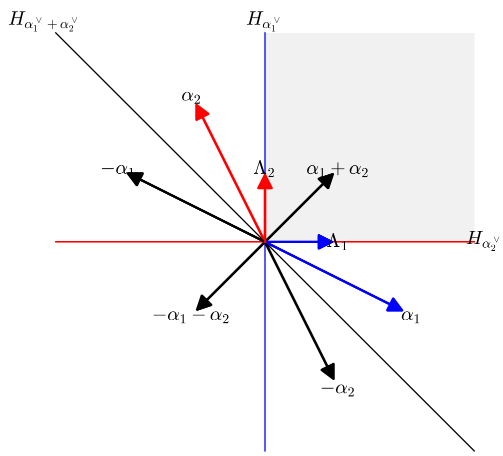
```
sage: L = RootSystem(["A",2]).weight_space()
sage: L.plot(roots='all', reflection_hyperplanes='all').show(figsize=15)
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).weight_space()
>>> L.plot(roots='all', reflection_hyperplanes='all').show(figsize=Integer(15))
```

Note

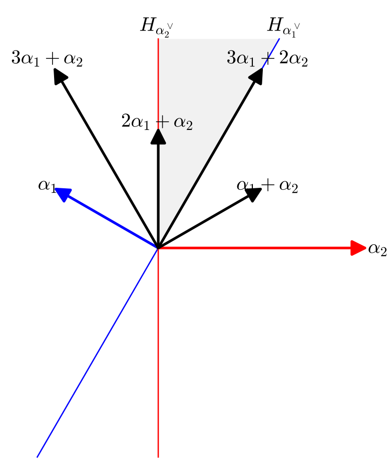
Setting a larger figure size as above can help reduce the overlap between the text labels when the figure gets crowded.

One can further customize which roots to display, as in the following example showing the positive roots in the weight space for type ['G',2], labelled by their coordinates in the root lattice:




```
sage: Q = RootSystem(["G",2]).root_space()
sage: L = RootSystem(["G",2]).ambient_space()
sage: L.plot(roots=list(Q.positive_roots()), fundamental_weights=False)
Graphics object consisting of 17 graphics primitives
```

```
>>> from sage.all import *
>>> Q = RootSystem(["G",Integer(2)]).root_space()
>>> L = RootSystem(["G",Integer(2)]).ambient_space()
>>> L.plot(roots=list(Q.positive_roots()), fundamental_weights=False)
Graphics object consisting of 17 graphics primitives
```



One can also customize the projection by specifying a function. Here, we display all the roots for type E_8 using the projection from its eight dimensional ambient space onto 3D described on Wikipedia%27s E8 3D picture:

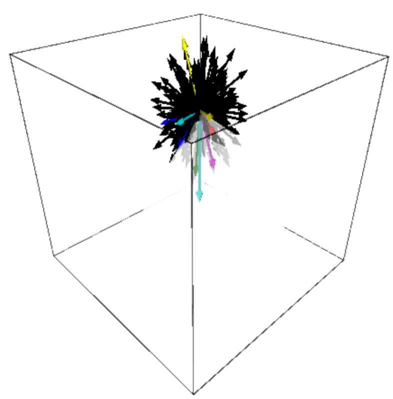
```
sage: M = matrix([[0., -0.556793440452, 0.19694925177, -0.19694925177, 0.
↪0805477263944, -0.385290876171, 0., 0.385290876171],
....:             [0.180913155536, 0., 0.160212955043, 0.160212955043, 0., 0.
↪0990170516545, 0.766360424875, 0.0990170516545],
....:             [0.338261212718, 0, 0, -0.338261212718, 0.672816364803, 0.
↪171502564281, 0, -0.171502564281]])
sage: L = RootSystem(["E",8]).ambient_space()
```

(continues on next page)

(continued from previous page)

```
sage: L.dimension()
8
sage: L.plot(roots='all', reflection_hyperplanes=False,            # long time
....:        projection=lambda v: M*vector(v), labels=False)
Graphics3d Object
```

```
>>> from sage.all import *
>>> M = matrix([[RealNumber('0.'), -RealNumber('0.556793440452'), RealNumber('0.
↪19694925177'), -RealNumber('0.19694925177'), RealNumber('0.0805477263944'), -
↪RealNumber('0.385290876171'), RealNumber('0.'), RealNumber('0.385290876171')],
...             [RealNumber('0.180913155536'), RealNumber('0.'), RealNumber('0.
↪160212955043'), RealNumber('0.160212955043'), RealNumber('0.'), RealNumber('0.
↪0990170516545'), RealNumber('0.766360424875'), RealNumber('0.0990170516545')],
...             [RealNumber('0.338261212718'), Integer(0), Integer(0), -RealNumber('0.
↪338261212718'), RealNumber('0.672816364803'), RealNumber('0.171502564281'),␣
↪Integer(0), -RealNumber('0.171502564281')]])
>>> L = RootSystem(["E",Integer(8)]).ambient_space()
>>> L.dimension()
8
>>> L.plot(roots='all', reflection_hyperplanes=False,            # long time
...        projection=lambda v: M*vector(v), labels=False)
Graphics3d Object
```



The projection function should be linear or affine, and return a vector with rational coordinates. The rationale for the later constraint is to allow for using the PPL exact library for manipulating polytopes. Indeed exact calculations give cleaner pictures (adjacent objects, intersection with the bounding box, …). Besides the interface to PPL is indeed currently faster than that for CDD, and it is likely to become even more so.

Exercise

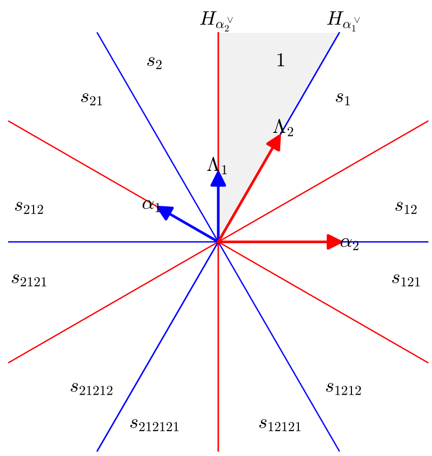
Draw all finite root systems in 2D, using the canonical projection onto their Coxeter plane. See Stembridge's page.

Alcoves and chambers

We now draw the root system for type G_2, with its alcoves (in finite type, those really are the chambers) and the corresponding elements of the Weyl group. We enlarge a bit the bounding box to make sure everything fits in the picture:

```
sage: RootSystem(["G",2]).ambient_space().plot(alcoves=True,
....:                                          alcove_labels=True, bounding_box=5)
Graphics object consisting of 37 graphics primitives
```

```
>>> from sage.all import *
>>> RootSystem(["G",Integer(2)]).ambient_space().plot(alcoves=True,
...                                              alcove_labels=True, bounding_
↪box=Integer(5))
Graphics object consisting of 37 graphics primitives
```



The same picture in 3D, for type B_3:

```
sage: RootSystem(["B",3]).ambient_space().plot(alcoves=True, alcove_labels=True)
Graphics3d Object
```

```
>>> from sage.all import *
>>> RootSystem(["B",Integer(3)]).ambient_space().plot(alcoves=True, alcove_
↪labels=True)
Graphics3d Object
```

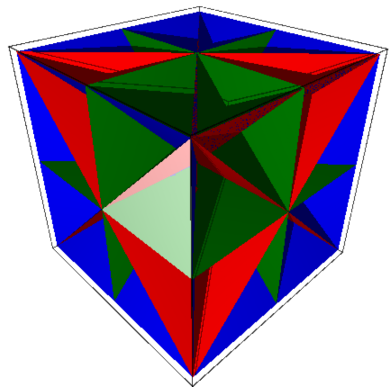
Exercise

Can you spot the fundamental chamber? The fundamental weights? The simple roots? The longest element of the Weyl group?

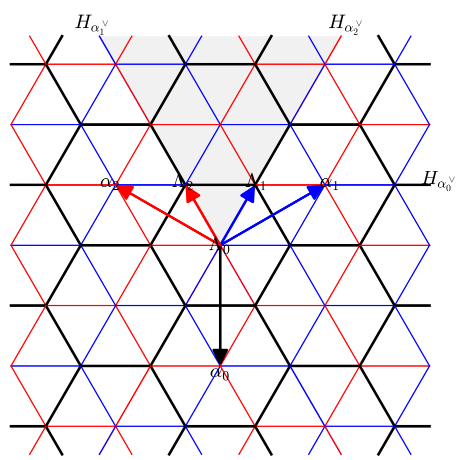
Alcove pictures for affine types

We now draw the usual alcove picture for affine type $A_2^{(1)}$:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: L.plot()                              # long time
Graphics object consisting of 160 graphics primitives
```



```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> L.plot()                                   # long time
Graphics object consisting of 160 graphics primitives
```



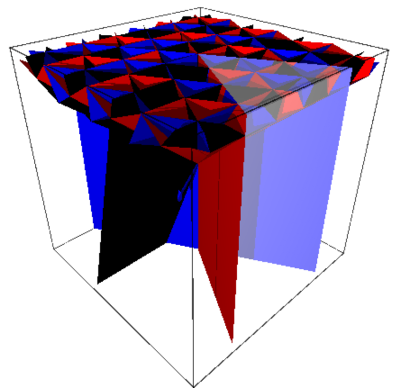
This picture is convenient because it is low dimensional and contains most of the relevant information. Beside, by choosing the ambient space, the elements of the Weyl group act as orthogonal affine maps. In particular, reflections are usual (affine) orthogonal reflections. However this is in fact only a slice of the real picture: the Weyl group actually acts by linear maps on the full ambient space. Those maps stabilize the so-called level l hyperplanes, and we are visualizing here what's happening at level 1. Here is the full picture in 3D:

```
sage: L.plot(bounding_box=[[-3,3],[-3,3],[-1,1]], affine=False)      # long time
Graphics3d Object
```

```
>>> from sage.all import *
>>> L.plot(bounding_box=[[-Integer(3),Integer(3)],[-Integer(3),Integer(3)],[-
↪Integer(1),Integer(1)]], affine=False)      # long time
Graphics3d Object
```

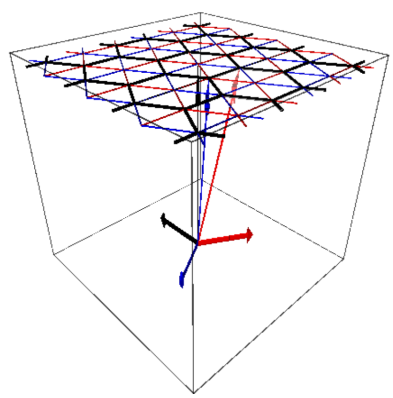
In fact, in type A, this really is a picture in 4D, but as usual the barycentric projection kills the boring extra dimension for us.

It's usually more readable to only draw the intersection of the reflection hyperplanes with the level 1 hyperplane:



```
sage: L.plot(affine=False, level=1)                          # long time
Graphics3d Object
```

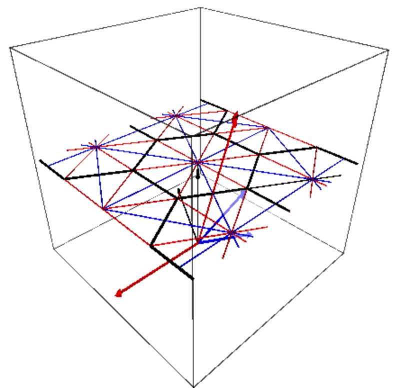
```
>>> from sage.all import *
>>> L.plot(affine=False, level=Integer(1))                          # long time
Graphics3d Object
```



Such 3D pictures are useful to better understand technicalities, like the fact that the fundamental weights do not necessarily all live at level 1:

```
sage: L = RootSystem(["G",2,1]).ambient_space()
sage: L.plot(affine=False, level=1)
Graphics3d Object
```

```
>>> from sage.all import *
>>> L = RootSystem(["G",Integer(2),Integer(1)]).ambient_space()
>>> L.plot(affine=False, level=Integer(1))
Graphics3d Object
```



Note

Such pictures may tend to be a bit flat, and it may be helpful to play with the aspect_ratio and more generally with the various options of the `show()` method:

```
sage: p = L.plot(affine=False, level=1)
sage: p.show(aspect_ratio=[1,1,2], frame=False)
```

```
>>> from sage.all import *
>>> p = L.plot(affine=False, level=Integer(1))
>>> p.show(aspect_ratio=[Integer(1),Integer(1),Integer(2)], frame=False)
```

Exercise

Draw the alcove picture at level 1, and compare the position of the fundamental weights and the vertices of the fundamental alcove.

As for finite root systems, the alcoves are indexed by the elements of the Weyl group W. Two alcoves indexed by u and v respectively share a wall if u and v are neighbors in the right Cayley graph: $u = vs_i$; the color of that wall is given by i:

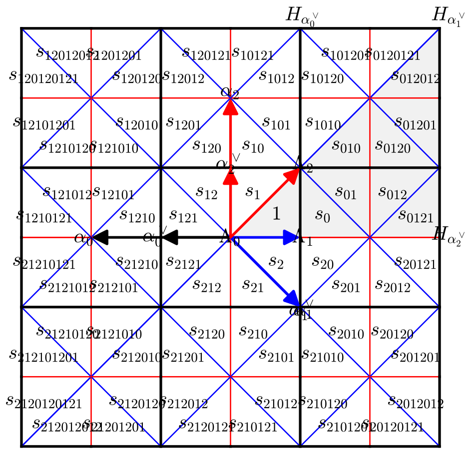
```
sage: L = RootSystem(["C",2,1]).ambient_space()
sage: L.plot(coroots='simple', alcove_labels=True)                  # long time
Graphics object consisting of 216 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["C",Integer(2),Integer(1)]).ambient_space()
>>> L.plot(coroots='simple', alcove_labels=True)                  # long time
Graphics object consisting of 216 graphics primitives
```

Even 2D pictures of the rank $1 + 1$ cases can give some food for thought. Here, we draw the root lattice, with the positive roots of small height in the root poset:

```
sage: L = RootSystem(["A",1,1]).root_lattice()
sage: seed = L.simple_roots()
sage: succ = attrcall("pred")
sage: positive_roots = RecursivelyEnumeratedSet(seed, succ, structure='graded')
```

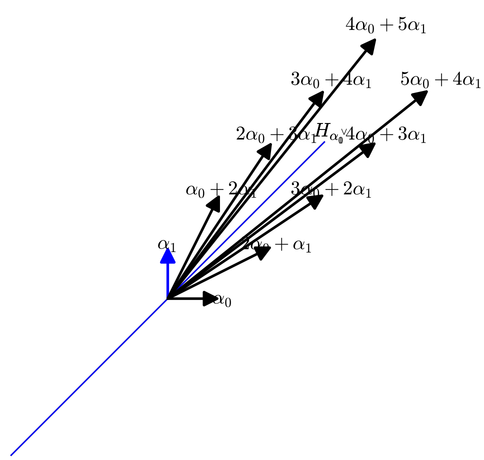
(continues on next page)



(continued from previous page)

```
sage: it = iter(positive_roots)
sage: first_positive_roots = [next(it) for i in range(10)]
sage: L.plot(roots=first_positive_roots, affine=False, alcoves=False)
Graphics object consisting of 24 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1),Integer(1)]).root_lattice()
>>> seed = L.simple_roots()
>>> succ = attrcall("pred")
>>> positive_roots = RecursivelyEnumeratedSet(seed, succ, structure='graded')
>>> it = iter(positive_roots)
>>> first_positive_roots = [next(it) for i in range(Integer(10))]
>>> L.plot(roots=first_positive_roots, affine=False, alcoves=False)
Graphics object consisting of 24 graphics primitives
```



Exercises

1. Use the same trick to draw the reflection hyperplanes in the weight lattice for the coroots of small height. Add the indexing of the alcoves by elements of the Weyl group. See below for a solution.

2. Draw the positive roots in the weight lattice and in the extended weight lattice.
3. Draw the reflection hyperplanes in the root lattice
4. Recreate John Stembridge's "Sandwich" arrangement pictures by choosing appropriate coroots for the reflection hyperplanes.

Here is a polished solution for the first exercise:

```
sage: L = RootSystem(["A",1,1]).weight_space()
sage: seed = L.simple_coroots()
sage: succ = attrcall("pred")
sage: positive_coroots = RecursivelyEnumeratedSet(seed, succ, structure='graded')
sage: it = iter(positive_coroots)
sage: first_positive_coroots = [next(it) for i in range(20)]
sage: p = L.plot(fundamental_chamber=True,
....:            reflection_hyperplanes=first_positive_coroots,
....:            affine=False, alcove_labels=1,
....:            bounding_box=[[-9,9],[-1,2]],
....:            projection=lambda x: matrix([[1,-1],[1,1]])*vector(x))
sage: p.show(figsize=20)                                        # long time
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1),Integer(1)]).weight_space()
>>> seed = L.simple_coroots()
>>> succ = attrcall("pred")
>>> positive_coroots = RecursivelyEnumeratedSet(seed, succ, structure='graded')
>>> it = iter(positive_coroots)
>>> first_positive_coroots = [next(it) for i in range(Integer(20))]
>>> p = L.plot(fundamental_chamber=True,
...            reflection_hyperplanes=first_positive_coroots,
...            affine=False, alcove_labels=Integer(1),
...            bounding_box=[[-Integer(9),Integer(9)],[-Integer(1),Integer(2)]],
...            projection=lambda x: matrix([[Integer(1),-Integer(1)],[Integer(1),
↪Integer(1)]])*vector(x))
>>> p.show(figsize=Integer(20))                                 # long time
```

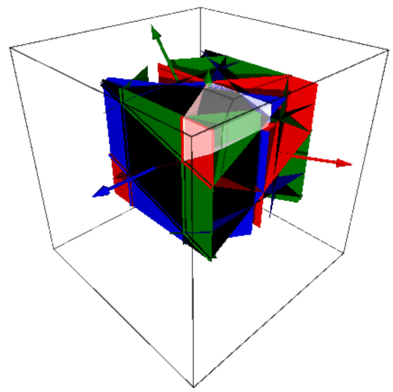
Higher dimension affine pictures

We now do some plots for rank 4 affine types, at level 1. The space is tiled by the alcoves, each of which is a 3D simplex:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: L.plot(reflection_hyperplanes=False, bounding_box=85/100)     # long time
Graphics3d Object
```

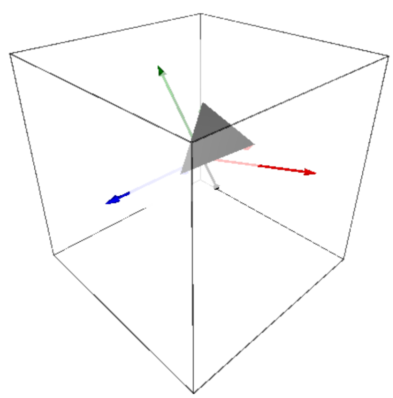
```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3),Integer(1)]).ambient_space()
>>> L.plot(reflection_hyperplanes=False, bounding_box=Integer(85)/Integer(100))     #␣
↪long time
Graphics3d Object
```

It is recommended to use a small bounding box here, for otherwise the number of simplices grows quicker than what Sage can handle smoothly. It can help to specify explicitly which alcoves to visualize. Here is the fundamental alcove, specified by an element of the Weyl group:



```
sage: W = L.weyl_group()
sage: L.plot(reflection_hyperplanes=False, alcoves=[W.one()], bounding_box=2)
Graphics3d Object
```

```
>>> from sage.all import *
>>> W = L.weyl_group()
>>> L.plot(reflection_hyperplanes=False, alcoves=[W.one()], bounding_box=Integer(2))
Graphics3d Object
```

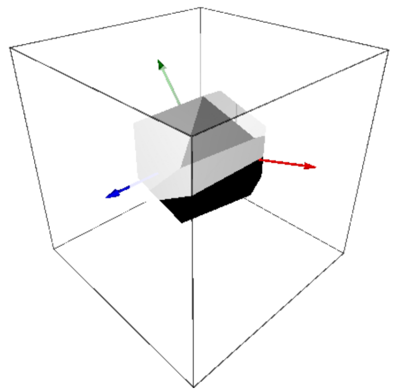


and the fundamental polygon, specified by the coordinates of its center in the root lattice:

```
sage: W = L.weyl_group()
sage: L.plot(reflection_hyperplanes=False, alcoves=[[0,0]], bounding_box=2)
Graphics3d Object
```

```
>>> from sage.all import *
>>> W = L.weyl_group()
>>> L.plot(reflection_hyperplanes=False, alcoves=[[Integer(0),Integer(0)]], bounding_
↪box=Integer(2))
Graphics3d Object
```

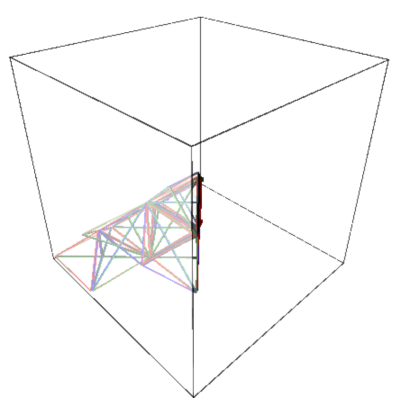
Finally, we draw the alcoves in the classical fundamental chambers, using that those are indexed by the elements of the



Weyl group having no other left descent than 0. In order to see the inner structure, we only draw the wireframe of the facets of the alcoves. Specifying the `wireframe` option requires a more flexible syntax for plots which will be explained later on in this tutorial:

```
sage: L = RootSystem(["B",3,1]).ambient_space()
sage: W = L.weyl_group()
sage: alcoves = [~w for d in range(12)
....:            for w in W.affine_grassmannian_elements_of_given_length(d)]
sage: p = L.plot_fundamental_chamber("classical")
sage: p += L.plot_alcoves(alcoves=alcoves, wireframe=True)
sage: p += L.plot_fundamental_weights()
sage: p.show(frame=False)
```

```
>>> from sage.all import *
>>> L = RootSystem(["B",Integer(3),Integer(1)]).ambient_space()
>>> W = L.weyl_group()
>>> alcoves = [~w for d in range(Integer(12))
...            for w in W.affine_grassmannian_elements_of_given_length(d)]
>>> p = L.plot_fundamental_chamber("classical")
>>> p += L.plot_alcoves(alcoves=alcoves, wireframe=True)
>>> p += L.plot_fundamental_weights()
>>> p.show(frame=False)
```



Exercises

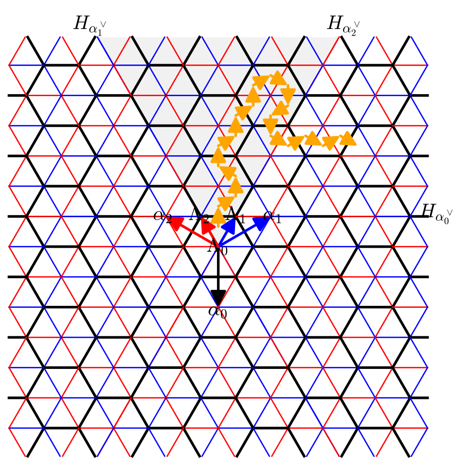
1. Draw the fundamental alcove in the ambient space, just by itself (no reflection hyperplane, root, ...). The automorphism group of the Dynkin diagram for $A_3^{(1)}$ (a cycle of length 4) is the dihedral group. Visualize the corresponding symmetries of the fundamental alcove.
2. Draw the fundamental alcoves for the other rank 4 affine types, and recover the automorphism groups of their Dynkin diagram from the pictures.

Drawing on top of a root system plot

The root system plots have been designed to be used as wallpaper on top of which to draw more information. In the following example, we draw an alcove walk, specified by a word of indices of simple reflections, on top of the weight lattice in affine type $A_{2,1}$:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: w1 = [0,2,1,2,0,2,1,0,2,1,2,1,2,0,2,0,1,2,0]
sage: L.plot(alcove_walk=w1, bounding_box=6)        # long time
Graphics object consisting of 535 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> w1 = [Integer(0),Integer(2),Integer(1),Integer(2),Integer(0),Integer(2),
↪Integer(1),Integer(0),Integer(2),Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(0),Integer(2),Integer(0),Integer(1),Integer(2),Integer(0)]
>>> L.plot(alcove_walk=w1, bounding_box=Integer(6))        # long time
Graphics object consisting of 535 graphics primitives
```



Now, what about drawing several alcove walks, and specifying some colors? A single do-it-all plot method would be cumbersome; so instead, it is actually built on top of many methods (see the list below) that can be called independently and combined at will:

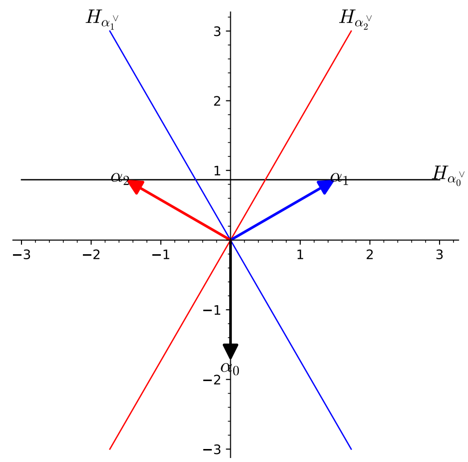
```
sage: L.plot_roots() + L.plot_reflection_hyperplanes()
Graphics object consisting of 12 graphics primitives
```

```
>>> from sage.all import *
>>> L.plot_roots() + L.plot_reflection_hyperplanes()
```

(continues on next page)

(continued from previous page)

```
Graphics object consisting of 12 graphics primitives
```

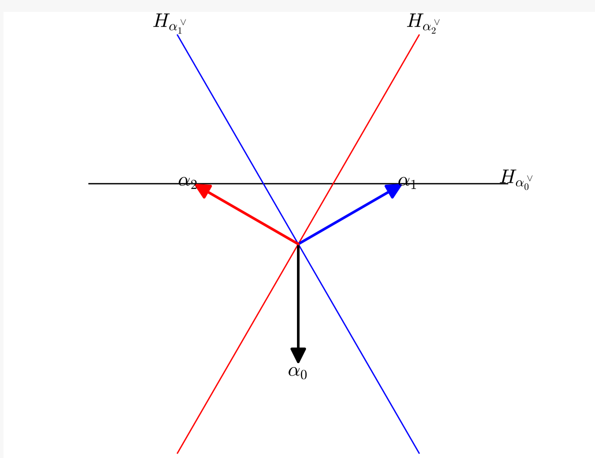


Note

By default the axes are disabled in root system plots since they tend to pollute the picture. Annoyingly they come back when combining them. Here is a workaround:

```
sage: p = L.plot_roots() + L.plot_reflection_hyperplanes()
sage: p.axes(False)
sage: p
Graphics object consisting of 12 graphics primitives
```

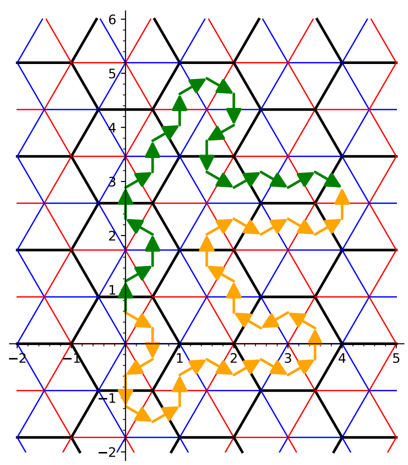
```
>>> from sage.all import *
>>> p = L.plot_roots() + L.plot_reflection_hyperplanes()
>>> p.axes(False)
>>> p
Graphics object consisting of 12 graphics primitives
```



In order to specify common information for all the pieces of a root system plot (choice of projection, bounding box, color code for the index set, ...), the easiest is to create an option object using `plot_parse_options()`, and pass it down to each piece. We use this to plot our two walks:

```
sage: # long time
sage: plot_options = L.plot_parse_options(bounding_box=[[-2,5],[-2,6]])
sage: w2 = [2,1,2,0,2,0,2,1,2,0,1,2,1,2,1,0,1,2,0,2,0,1,2,0,2]
sage: p = L.plot_alcoves(plot_options=plot_options)
sage: p += L.plot_alcove_walk(w1, color='green',
....:                         plot_options=plot_options)
sage: p += L.plot_alcove_walk(w2, color='orange',
....:                         plot_options=plot_options)
sage: p
Graphics object consisting of ... graphics primitives
```

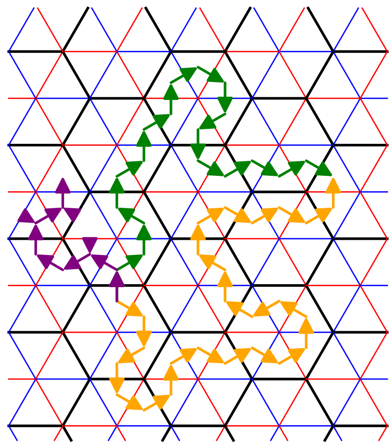
```
>>> from sage.all import *
>>> # long time
>>> plot_options = L.plot_parse_options(bounding_box=[[-Integer(2),Integer(5)],[-
↪Integer(2),Integer(6)]])
>>> w2 = [Integer(2),Integer(1),Integer(2),Integer(0),Integer(2),Integer(0),
↪Integer(2),Integer(1),Integer(2),Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(1),Integer(0),Integer(1),Integer(2),Integer(0),Integer(2),
↪Integer(0),Integer(1),Integer(2),Integer(0),Integer(2)]
>>> p = L.plot_alcoves(plot_options=plot_options)
>>> p += L.plot_alcove_walk(w1, color='green',
...                         plot_options=plot_options)
>>> p += L.plot_alcove_walk(w2, color='orange',
...                         plot_options=plot_options)
>>> p
Graphics object consisting of ... graphics primitives
```



And another with some foldings:

```
sage: p += L.plot_alcove_walk([0,1,2,0,2,0,1,2,0,1],
....:                         foldings=[False, False, True, False, False,
....:                                   False, True, False, True, False],
....:                         color='purple')
sage: p.axes(False)
sage: p.show(figsize=20)
```

```
>>> from sage.all import *
>>> p += L.plot_alcove_walk([Integer(0),Integer(1),Integer(2),Integer(0),Integer(2),
↪Integer(0),Integer(1),Integer(2),Integer(0),Integer(1)],
...                          foldings=[False, False, True, False, False,
...                                    False, True, False, True, False],
...                          color='purple')
>>> p.axes(False)
>>> p.show(figsize=Integer(20))
```



Here we show a weight at level 0 and the reduced word implementing the translation by this weight:

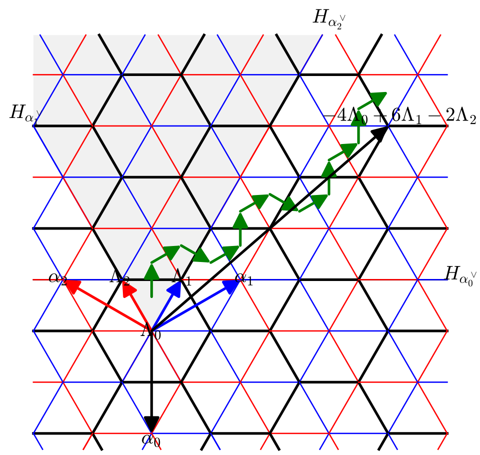
```
sage: # long time
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: P = RootSystem(["A",2,1]).weight_space(extended=True)
sage: Lambda = P.fundamental_weights()
sage: t = 6*Lambda[1] - 2*Lambda[2] - 4*Lambda[0]
sage: walk = L.reduced_word_of_translation(L(t))
sage: plot_options = L.plot_parse_options(bounding_box=[[-2,5],[-2,5]])
sage: p = L.plot(plot_options=plot_options)
sage: p += L.plot_alcove_walk(walk, color='green',
....:                         plot_options=plot_options)
sage: p += plot_options.family_of_vectors({t: L(t)})
sage: plot_options.finalize(p)
Graphics object consisting of ... graphics primitives
sage: p
Graphics object consisting of ... graphics primitives
```

```
>>> from sage.all import *
>>> # long time
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> P = RootSystem(["A",Integer(2),Integer(1)]).weight_space(extended=True)
>>> Lambda = P.fundamental_weights()
>>> t = Integer(6)*Lambda[Integer(1)] - Integer(2)*Lambda[Integer(2)] -␣
↪Integer(4)*Lambda[Integer(0)]
>>> walk = L.reduced_word_of_translation(L(t))
>>> plot_options = L.plot_parse_options(bounding_box=[[-Integer(2),Integer(5)],[-
↪Integer(2),Integer(5)]])
```

(continues on next page)

(continued from previous page)

```
>>> p = L.plot(plot_options=plot_options)
>>> p += L.plot_alcove_walk(walk, color='green',
...                         plot_options=plot_options)
>>> p += plot_options.family_of_vectors({t: L(t)})
>>> plot_options.finalize(p)
Graphics object consisting of ... graphics primitives
>>> p
Graphics object consisting of ... graphics primitives
```



Note that the coloring of the translated alcove does not match with that of the fundamental alcove: the translation actually lives in the extended Weyl group and is the composition of the simple reflections indexed by the alcove walk together with a rotation implementing an automorphism of the Dynkin diagram.

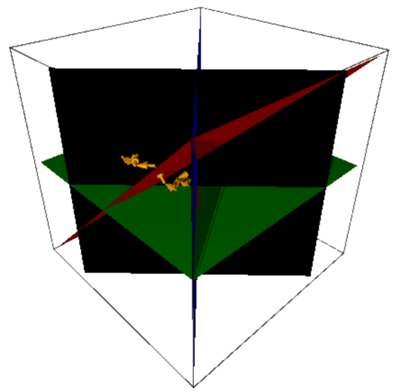
We conclude with a rank $3 + 1$ alcove walk:

```
sage: L = RootSystem(["B",3,1]).ambient_space()
sage: w3 = [0,2,1,3,2,0,2,1,0,2,3,1,2,1,3,2,0,2,0,1,2,0]
sage: (L.plot_fundamental_weights()
....:   + L.plot_reflection_hyperplanes(bounding_box=2) + L.plot_alcove_walk(w3))
Graphics3d Object
```

```
>>> from sage.all import *
>>> L = RootSystem(["B",Integer(3),Integer(1)]).ambient_space()
>>> w3 = [Integer(0),Integer(2),Integer(1),Integer(3),Integer(2),Integer(0),
↪Integer(2),Integer(1),Integer(0),Integer(2),Integer(3),Integer(1),Integer(2),
↪Integer(1),Integer(3),Integer(2),Integer(0),Integer(2),Integer(0),Integer(1),
↪Integer(2),Integer(0)]
>>> (L.plot_fundamental_weights()
...   + L.plot_reflection_hyperplanes(bounding_box=Integer(2)) + L.plot_alcove_
↪walk(w3))
Graphics3d Object
```

Exercise

1. Draw the tiling of 3D space by the fundamental polygons for types A,B,C,D. Hints: use the `wireframe` option of *RootLatticeRealizations.ParentMethods.plot_alcoves()* and the `color` option of `plot()`

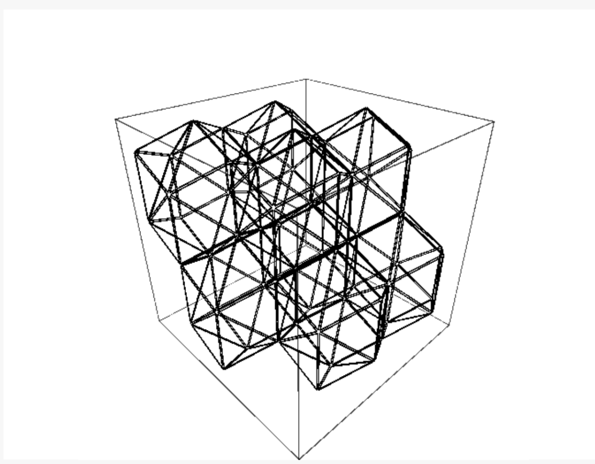


to only draw the alcove facets indexed by 0.

Solution

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: alcoves = cartesian_product([[0,1],[0,1],[0,1]])
sage: color = lambda i: "black" if i==0 else None
sage: L.plot_alcoves(alcoves=alcoves, color=color,              # long time
....:                bounding_box=10, wireframe=True).show(frame=False)
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3),Integer(1)]).ambient_space()
>>> alcoves = cartesian_product([[Integer(0),Integer(1)],[Integer(0),Integer(1)],
↪[Integer(0),Integer(1)]])
>>> color = lambda i: "black" if i==Integer(0) else None
>>> L.plot_alcoves(alcoves=alcoves, color=color,              # long time
...                bounding_box=Integer(10), wireframe=True).show(frame=False)
```



Hand drawing on top of a root system plot (aka Coxeter graph paper)

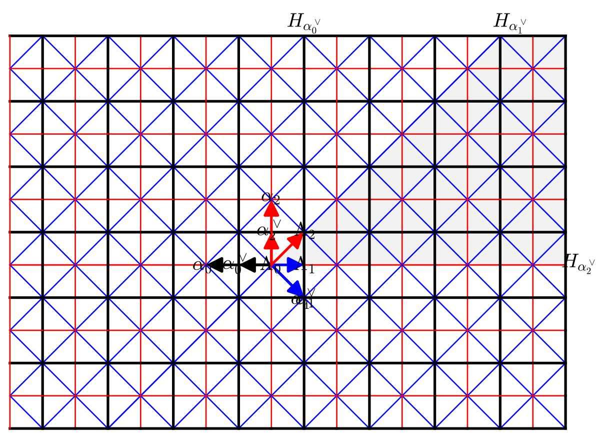
Taken from John Stembridge's excellent data archive:

"If you've ever worked with affine reflection groups, you've probably wasted lots of time drawing the reflecting hyperplanes of the rank 2 groups on scraps of paper. You may also have wished you had pads of graph paper with these lines drawn in for you. If so, you've come to the right place. Behold! Coxeter graph paper!".

Now you can create your own customized color Coxeter graph paper:

```
sage: L = RootSystem(["C",2,1]).ambient_space()
sage: p = L.plot(bounding_box=[[-8,9],[-5,7]],       # long time (10 s)
....:            coroots='simple')
sage: p                                              # long time
Graphics object consisting of ... graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["C",Integer(2),Integer(1)]).ambient_space()
>>> p = L.plot(bounding_box=[[-Integer(8),Integer(9)],[-Integer(5),Integer(7)]],
↪# long time (10 s)
...            coroots='simple')
>>> p                                              # long time
Graphics object consisting of ... graphics primitives
```



By default Sage's plot are bitmap pictures which would come out ugly if printed on paper. Instead, we recommend saving the picture in postscript or svg before printing it:

```
sage: p.save("C21paper.eps")                  # not tested
```

```
>>> from sage.all import *
>>> p.save("C21paper.eps")                  # not tested
```

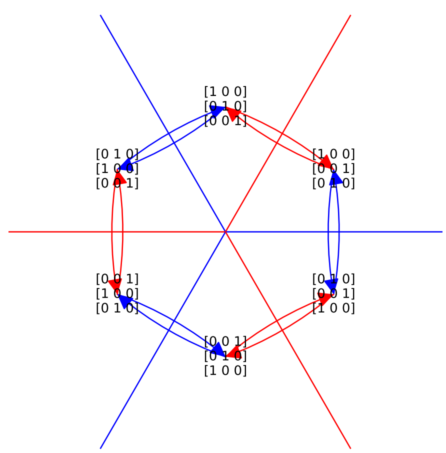
> **Note:** Drawing pictures with a large number of alcoves is currently somewhat ridiculously slow. This is due to the use of generic code that works uniformly in all dimension rather than taylor-made code for 2D. Things should improve with the fast interface to the PPL library (see e.g. Issue #12553).

Drawing custom objects on top of a root system plot

So far so good. Now, what if one wants to draw, on top of a root system plot, some object for which there is no preexisting plot method? Again, the `plot_options` object come in handy, as it can be used to compute appropriate coordinates. Here we draw the permutohedron, that is the Cayley graph of the symmetric group W, by positioning each element w at $w(\rho)$, where ρ is in the fundamental alcove:

```
sage: L = RootSystem(["A",2]).ambient_space()
sage: rho = L.rho()
sage: plot_options = L.plot_parse_options()
sage: W = L.weyl_group()
sage: g = W.cayley_graph(side='right')
sage: positions = {w: plot_options.projection(w.action(rho)) for w in W}
sage: p = L.plot_alcoves()
sage: p += g.plot(pos=positions, vertex_size=0,
....:             color_by_label=plot_options.color)
sage: p.axes(False)
sage: p
Graphics object consisting of 30 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).ambient_space()
>>> rho = L.rho()
>>> plot_options = L.plot_parse_options()
>>> W = L.weyl_group()
>>> g = W.cayley_graph(side='right')
>>> positions = {w: plot_options.projection(w.action(rho)) for w in W}
>>> p = L.plot_alcoves()
>>> p += g.plot(pos=positions, vertex_size=Integer(0),
...             color_by_label=plot_options.color)
>>> p.axes(False)
>>> p
Graphics object consisting of 30 graphics primitives
```



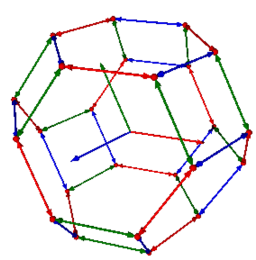
Todo

Could we have nice LaTeX labels in this graph?

The same picture for A_3 gives a nice 3D permutohedron:

```
sage: L = RootSystem(["A",3]).ambient_space()
sage: rho = L.rho()
sage: plot_options = L.plot_parse_options()
sage: W = L.weyl_group()
sage: g = W.cayley_graph(side='right')
sage: positions = {w: plot_options.projection(w.action(rho)) for w in W}
sage: p = L.plot_roots()
sage: p += g.plot3d(pos3d=positions, color_by_label=plot_options.color)
sage: p
Graphics3d Object
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3)]).ambient_space()
>>> rho = L.rho()
>>> plot_options = L.plot_parse_options()
>>> W = L.weyl_group()
>>> g = W.cayley_graph(side='right')
>>> positions = {w: plot_options.projection(w.action(rho)) for w in W}
>>> p = L.plot_roots()
>>> p += g.plot3d(pos3d=positions, color_by_label=plot_options.color)
>>> p
Graphics3d Object
```



Exercises

1. Locate the identity element of W in the previous picture
2. Rotate the picture appropriately to highlight the various symmetries of the permutohedron.
3. Make a function out of the previous example, and explore the Cayley graphs of all rank 2 and 3 Weyl groups.
4. Draw the root poset for type B_2 and B_3
5. Draw the root poset for type E_8 to recover the picture from Wikipedia article File:E8_3D.png

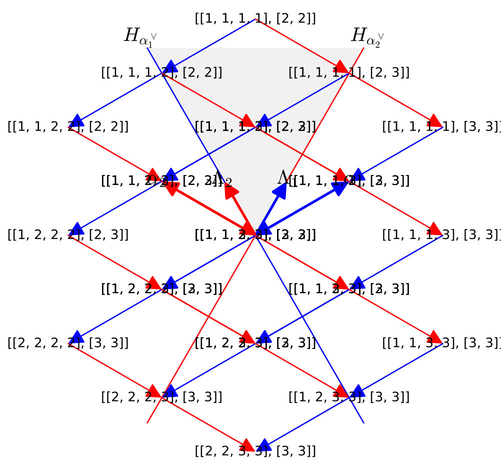
Similarly, we display a crystal graph by positioning each element according to its weight:

```
sage: C = crystals.Tableaux(["A",2], shape=[4,2])
sage: L = C.weight_lattice_realization()
sage: plot_options = L.plot_parse_options()

sage: g = C.digraph()
sage: positions = {x: plot_options.projection(x.weight()) for x in C}
sage: p = L.plot()
sage: p += g.plot(pos=positions,
....:             color_by_label=plot_options.color, vertex_size=0)
sage: p.axes(False)
sage: p.show(figsize=15)
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(["A",Integer(2)], shape=[Integer(4),Integer(2)])
>>> L = C.weight_lattice_realization()
>>> plot_options = L.plot_parse_options()

>>> g = C.digraph()
>>> positions = {x: plot_options.projection(x.weight()) for x in C}
>>> p = L.plot()
>>> p += g.plot(pos=positions,
...             color_by_label=plot_options.color, vertex_size=Integer(0))
>>> p.axes(False)
>>> p.show(figsize=Integer(15))
```



Note

In the above picture, many pairs of tableaux have the same weight and are thus superposed (look for example near the center). Some more layout logic would be needed to separate those nodes properly, but the foundations are laid firmly and uniformly across all types of root systems for writing such extensions.

Here is an analogue picture in 3D:

```
sage: C = crystals.Tableaux(["A",3], shape=[3,2,1])
sage: L = C.weight_lattice_realization()
sage: plot_options = L.plot_parse_options()
```

(continues on next page)

(continued from previous page)

```
sage: g = C.digraph()
sage: positions = {x: plot_options.projection(x.weight()) for x in C}
sage: p = L.plot(reflection_hyperplanes=False, fundamental_weights=False)
sage: p += g.plot3d(pos3d=positions, vertex_labels=True,
....:               color_by_label=plot_options.color, edge_labels=True)
sage: p
Graphics3d Object
```

```
>>> from sage.all import *
>>> C = crystals.Tableaux(["A",Integer(3)], shape=[Integer(3),Integer(2),Integer(1)])
>>> L = C.weight_lattice_realization()
>>> plot_options = L.plot_parse_options()
>>> g = C.digraph()
>>> positions = {x: plot_options.projection(x.weight()) for x in C}
>>> p = L.plot(reflection_hyperplanes=False, fundamental_weights=False)
>>> p += g.plot3d(pos3d=positions, vertex_labels=True,
...               color_by_label=plot_options.color, edge_labels=True)
>>> p
Graphics3d Object
```

Exercise

Explore the previous picture and notice how the edges of the crystal graph are parallel to the simple roots.

Enjoy and please post your best pictures on the Sage-Combinat wiki.

class sage.combinat.root_system.plot.**PlotOptions**(*space*, *projection=True*, *bounding_box=3*, *color=<bound method CartanTypeFactory.color of <class 'sage.combinat.root_system.cartan_type.CartanTypeFactory'>>*, *labels=True*, *level=None*, *affine=None*, *arrowsize=5*)

Bases: object

A class for plotting options for root lattice realizations.

See also

- *RootLatticeRealizations.ParentMethods.plot()* for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

color(i)

Return the color to be used for objects indexed by i.

INPUT:

- i – an index

See also

index_of_object()

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options(labels=False)
sage: alpha = L.simple_roots()
sage: options.color(1)
'blue'
sage: options.color(2)
'red'
sage: for alpha in L.roots():
....:     print("{} {}".format(alpha, options.color(alpha)))
alpha[1]             blue
alpha[2]             red
alpha[1] + alpha[2]  black
-alpha[1]            black
-alpha[2]            black
-alpha[1] - alpha[2] black
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).root_lattice()
>>> options = L.plot_parse_options(labels=False)
>>> alpha = L.simple_roots()
>>> options.color(Integer(1))
'blue'
>>> options.color(Integer(2))
'red'
>>> for alpha in L.roots():
...     print("{} {}".format(alpha, options.color(alpha)))
alpha[1]             blue
alpha[2]             red
alpha[1] + alpha[2]  black
-alpha[1]            black
-alpha[2]            black
-alpha[1] - alpha[2] black
```

cone (*rays=[]*, *lines=[]*, *color='black'*, *thickness=1*, *alpha=1*, *wireframe=False*, *label=None*, *draw_degenerate=True*, *as_polyhedron=False*)

Return the cone generated by the given rays and lines.

INPUT:

- `rays`, `lines` – lists of elements of the root lattice realization (default: `[]`)
- `color` – a color (default: `'black'`)
- `alpha` – a number in the interval $[0, 1]$ (default: 1) the desired transparency
- `label` – an object to be used as the label for this cone The label itself will be constructed by calling `latex()` or `repr()` on the object depending on the graphics backend.
- `draw_degenerate` – boolean (default: `True`) whether to draw cones with a degenerate intersection with the bounding box
- `as_polyhedron` – boolean (default: `False`) whether to return the result as a polyhedron, without clipping it to the bounding box, and without making a plot out of it (for testing purposes)

OUTPUT: a graphic object, a polyhedron, or `0`

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options()
sage: alpha = L.simple_roots()
sage: p = options.cone(rays=[alpha[1]], lines=[alpha[2]],
....:                  color='green', label=2); p
Graphics object consisting of 2 graphics primitives
sage: list(p)
[Polygon defined by 4 points,
 Text '$2$' at the point (3.15...,3.15...)]
sage: options.cone(rays=[alpha[1]], lines=[alpha[2]],
....:              color='green', label=2, as_polyhedron=True)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex, 1␣
↪ray, 1 line
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).root_lattice()
>>> options = L.plot_parse_options()
>>> alpha = L.simple_roots()
>>> p = options.cone(rays=[alpha[Integer(1)]], lines=[alpha[Integer(2)]],
...                  color='green', label=Integer(2)); p
Graphics object consisting of 2 graphics primitives
>>> list(p)
[Polygon defined by 4 points,
 Text '$2$' at the point (3.15...,3.15...)]
>>> options.cone(rays=[alpha[Integer(1)]], lines=[alpha[Integer(2)]],
...              color='green', label=Integer(2), as_polyhedron=True)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex, 1␣
↪ray, 1 line
```

An empty result, being outside of the bounding box:

```
sage: options = L.plot_parse_options(labels=True,
....:                                bounding_box=[[-10,-9]]*2)
sage: options.cone(rays=[alpha[1]], lines=[alpha[2]],
....:              color='green', label=2)
0
```

```
>>> from sage.all import *
>>> options = L.plot_parse_options(labels=True,
...                                bounding_box=[[-Integer(10),-
↪Integer(9)]]*Integer(2))
>>> options.cone(rays=[alpha[Integer(1)]], lines=[alpha[Integer(2)]],
...              color='green', label=Integer(2))
0
```

Test that the options are properly passed down:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options()
sage: p = options.cone(rays=[alpha[1] + alpha[2]],
....:                  color='green', label=2, thickness=4, alpha=.5)
```

(continues on next page)

(continued from previous page)

```
sage: list(p)
[Line defined by 2 points, Text '$2$' at the point (3.15...,3.15...)]
sage: sorted(p[0].options().items())
[('alpha', 0.500000000000000), ('legend_color', None),
 ('legend_label', None), ('rgbcolor', 'green'), ('thickness', 4),
 ('zorder', 1)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).root_lattice()
>>> options = L.plot_parse_options()
>>> p = options.cone(rays=[alpha[Integer(1)] + alpha[Integer(2)]],
...                  color='green', label=Integer(2), thickness=Integer(4),␣
↪alpha=RealNumber('.5'))
>>> list(p)
[Line defined by 2 points, Text '$2$' at the point (3.15...,3.15...)]
>>> sorted(p[Integer(0)].options().items())
[('alpha', 0.500000000000000), ('legend_color', None),
 ('legend_label', None), ('rgbcolor', 'green'), ('thickness', 4),
 ('zorder', 1)]
```

This method is tested indirectly but extensively by the various plot methods of root lattice realizations.

empty(**args*)

Return an empty plot.

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options(labels=True)
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).root_lattice()
>>> options = L.plot_parse_options(labels=True)
```

This currently returns `int(0)`:

```
sage: options.empty()
0
```

```
>>> from sage.all import *
>>> options.empty()
0
```

This is not a plot, so may cause some corner cases. On the other hand, 0 behaves as a fast neutral element, which is important given the typical idioms used in the plotting code:

```
sage: p = point([0,0])
sage: p + options.empty() is p
True
```

```
>>> from sage.all import *
>>> p = point([Integer(0),Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> p + options.empty() is p
True
```

family_of_vectors(*vectors*)

Return a plot of a family of vectors.

INPUT:

- `vectors` – family or vectors in `self`

The vectors are labelled by their index.

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options()
sage: alpha = L.simple_roots()
sage: p = options.family_of_vectors(alpha); p
Graphics object consisting of 4 graphics primitives
sage: list(p)
[Arrow from (0.0,0.0) to (1.0,0.0),
 Text '$1$' at the point (1.05,0.0),
 Arrow from (0.0,0.0) to (0.0,1.0),
 Text '$2$' at the point (0.0,1.05)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).root_lattice()
>>> options = L.plot_parse_options()
>>> alpha = L.simple_roots()
>>> p = options.family_of_vectors(alpha); p
Graphics object consisting of 4 graphics primitives
>>> list(p)
[Arrow from (0.0,0.0) to (1.0,0.0),
 Text '$1$' at the point (1.05,0.0),
 Arrow from (0.0,0.0) to (0.0,1.0),
 Text '$2$' at the point (0.0,1.05)]
```

Handling of colors and labels:

```
sage: def color(i):
....:     return "purple" if i==1 else None
sage: options = L.plot_parse_options(labels=False, color=color)
sage: p = options.family_of_vectors(alpha)
sage: list(p)
[Arrow from (0.0,0.0) to (1.0,0.0)]
sage: p[0].options()['rgbcolor']
'purple'
```

```
>>> from sage.all import *
>>> def color(i):
...     return "purple" if i==Integer(1) else None
>>> options = L.plot_parse_options(labels=False, color=color)
>>> p = options.family_of_vectors(alpha)
>>> list(p)
```

(continues on next page)

(continued from previous page)

```
[Arrow from (0.0,0.0) to (1.0,0.0)]
>>> p[Integer(0)].options()['rgbcolor']
'purple'
```

Matplotlib emits a warning for arrows of length 0 and draws nothing anyway. So we do not draw them at all:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: Lambda = L.fundamental_weights()
sage: p = options.family_of_vectors(Lambda); p
Graphics object consisting of 5 graphics primitives
sage: list(p)
[Text '$0$' at the point (0.0,0.0),
 Arrow from (0.0,0.0) to (0.5,0.86602451838...),
 Text '$1$' at the point (0.525,0.909325744308...),
 Arrow from (0.0,0.0) to (-0.5,0.86602451838...),
 Text '$2$' at the point (-0.525,0.909325744308...)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> options = L.plot_parse_options()
>>> Lambda = L.fundamental_weights()
>>> p = options.family_of_vectors(Lambda); p
Graphics object consisting of 5 graphics primitives
>>> list(p)
[Text '$0$' at the point (0.0,0.0),
 Arrow from (0.0,0.0) to (0.5,0.86602451838...),
 Text '$1$' at the point (0.525,0.909325744308...),
 Arrow from (0.0,0.0) to (-0.5,0.86602451838...),
 Text '$2$' at the point (-0.525,0.909325744308...)]
```

finalize(G)

Finalize a root system plot.

INPUT:

- `G` – a root system plot or `0`

This sets the aspect ratio to 1 and remove the axes. This should be called by all the user-level plotting methods of root systems. This will become mostly obsolete when customization options won't be lost anymore upon addition of graphics objects and there will be a proper empty object for 2D and 3D plots.

EXAMPLES:

```
sage: L = RootSystem(["B",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: p = L.plot_roots(plot_options=options)
sage: p += L.plot_coroots(plot_options=options)
sage: p.axes()
True
sage: p = options.finalize(p)
sage: p.axes()
False
sage: p.aspect_ratio()
```

(continues on next page)

(continued from previous page)

```
1.0

sage: options = L.plot_parse_options(affine=False)
sage: p = L.plot_roots(plot_options=options)
sage: p += point([[1,1,0]])
sage: p = options.finalize(p)
sage: p.aspect_ratio()
[1.0, 1.0, 1.0]
```

```
>>> from sage.all import *
>>> L = RootSystem(["B",Integer(2),Integer(1)]).ambient_space()
>>> options = L.plot_parse_options()
>>> p = L.plot_roots(plot_options=options)
>>> p += L.plot_coroots(plot_options=options)
>>> p.axes()
True
>>> p = options.finalize(p)
>>> p.axes()
False
>>> p.aspect_ratio()
1.0

>>> options = L.plot_parse_options(affine=False)
>>> p = L.plot_roots(plot_options=options)
>>> p += point([[Integer(1),Integer(1),Integer(0)]])
>>> p = options.finalize(p)
>>> p.aspect_ratio()
[1.0, 1.0, 1.0]
```

If the input is `0`, this returns an empty graphics object:

```
sage: type(options.finalize(0))
<class 'sage.plot.plot3d.base.Graphics3dGroup'>

sage: options = L.plot_parse_options()
sage: type(options.finalize(0))
<class 'sage.plot.graphics.Graphics'>
sage: list(options.finalize(0))
[]
```

```
>>> from sage.all import *
>>> type(options.finalize(Integer(0)))
<class 'sage.plot.plot3d.base.Graphics3dGroup'>

>>> options = L.plot_parse_options()
>>> type(options.finalize(Integer(0)))
<class 'sage.plot.graphics.Graphics'>
>>> list(options.finalize(Integer(0)))
[]
```

in_bounding_box(*x*)

Return whether `x` is in the bounding box.

INPUT:

- `x` – an element of the root lattice realization

This method is currently one of the bottlenecks, and therefore cached.

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: alpha = L.simple_roots()
sage: options.in_bounding_box(alpha[1])
True
sage: options.in_bounding_box(3*alpha[1])
False
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> options = L.plot_parse_options()
>>> alpha = L.simple_roots()
>>> options.in_bounding_box(alpha[Integer(1)])
True
>>> options.in_bounding_box(Integer(3)*alpha[Integer(1)])
False
```

index_of_object(*i*)

Try to return the node of the Dynkin diagram indexing the object i.

OUTPUT: a node of the Dynkin diagram or `None`

EXAMPLES:

```
sage: L = RootSystem(["A",3]).root_lattice()
sage: alpha = L.simple_roots()
sage: omega = RootSystem(["A",3]).weight_lattice().fundamental_weights()
sage: options = L.plot_parse_options(labels=False)
sage: options.index_of_object(3)
3
sage: options.index_of_object(alpha[1])
1
sage: options.index_of_object(omega[2])
2
sage: options.index_of_object(omega[2]+omega[3])
sage: options.index_of_object(30)
sage: options.index_of_object("bla")
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3)]).root_lattice()
>>> alpha = L.simple_roots()
>>> omega = RootSystem(["A",Integer(3)]).weight_lattice().fundamental_
→weights()
>>> options = L.plot_parse_options(labels=False)
>>> options.index_of_object(Integer(3))
3
>>> options.index_of_object(alpha[Integer(1)])
```

(continues on next page)

(continued from previous page)

```
1
>>> options.index_of_object(omega[Integer(2)])
2
>>> options.index_of_object(omega[Integer(2)]+omega[Integer(3)])
>>> options.index_of_object(Integer(30))
>>> options.index_of_object("bla")
```

intersection_at_level_1(*x*)

Return x scaled at the appropriate level, if level is set; otherwise return x.

INPUT:

- x – an element of the root lattice realization

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).weight_space()
sage: options = L.plot_parse_options()
sage: options.intersection_at_level_1(L.rho())
1/3*Lambda[0] + 1/3*Lambda[1] + 1/3*Lambda[2]

sage: options = L.plot_parse_options(affine=False, level=2)
sage: options.intersection_at_level_1(L.rho())
2/3*Lambda[0] + 2/3*Lambda[1] + 2/3*Lambda[2]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).weight_space()
>>> options = L.plot_parse_options()
>>> options.intersection_at_level_1(L.rho())
1/3*Lambda[0] + 1/3*Lambda[1] + 1/3*Lambda[2]

>>> options = L.plot_parse_options(affine=False, level=Integer(2))
>>> options.intersection_at_level_1(L.rho())
2/3*Lambda[0] + 2/3*Lambda[1] + 2/3*Lambda[2]
```

When level is not set, x is returned:

```
sage: options = L.plot_parse_options(affine=False)
sage: options.intersection_at_level_1(L.rho())
Lambda[0] + Lambda[1] + Lambda[2]
```

```
>>> from sage.all import *
>>> options = L.plot_parse_options(affine=False)
>>> options.intersection_at_level_1(L.rho())
Lambda[0] + Lambda[1] + Lambda[2]
```

projection(*v*)

Return the projection of v.

INPUT:

- x – an element of the root lattice realization

OUTPUT: an immutable vector with integer or rational coefficients

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: options.projection(L.rho())
(0, 989/571)

sage: options = L.plot_parse_options(projection=False)
sage: options.projection(L.rho())
(2, 1, 0)
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> options = L.plot_parse_options()
>>> options.projection(L.rho())
(0, 989/571)

>>> options = L.plot_parse_options(projection=False)
>>> options.projection(L.rho())
(2, 1, 0)
```

reflection_hyperplane(*coroot*, *as_polyhedron=False*)

Return a plot of the reflection hyperplane indexed by this coroot.

- `coroot` – a coroot

EXAMPLES:

```
sage: L = RootSystem(["B",2]).weight_space()
sage: alphacheck = L.simple_coroots()
sage: options = L.plot_parse_options()
sage: H = options.reflection_hyperplane(alphacheck[1]); H
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["B",Integer(2)]).weight_space()
>>> alphacheck = L.simple_coroots()
>>> options = L.plot_parse_options()
>>> H = options.reflection_hyperplane(alphacheck[Integer(1)]); H
Graphics object consisting of 2 graphics primitives
```

Todo

Display the periodic orientation by adding a $+$ and a $-$ sign close to the label. Typically by using the associated root to shift a bit from the vertex upon which the hyperplane label is attached.

text(*label*, *position*, *rgbcolor=(0, 0, 0)*)

Return text widget with label `label` at position `position`.

INPUT:

- `label` – string or Sage object upon which latex will be called
- `position` – a position
- `rgbcolor` – the color as an RGB tuple

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options()
sage: list(options.text("coucou", [0,1]))
[Text 'coucou' at the point (0.0,1.0)]
sage: list(options.text(L.simple_root(1), [0,1]))
[Text '$\alpha_{1}$' at the point (0.0,1.0)]
sage: list(options.text(L.simple_root(2), [1,0], rgbcolor=(1,0.5,0)))
[Text '$\alpha_{2}$' at the point (1.0,0.0)]

sage: options = L.plot_parse_options(labels=False)
sage: options.text("coucou", [0,1])
0

sage: options = RootSystem(["B",3]).root_lattice().plot_parse_options()
sage: print(options.text("coucou", [0,1,2]).x3d_str())
<Transform translation='0 1 2'>
<Shape><Text string='coucou' solid='true'/><Appearance><Material diffuseColor=
↪'0.0 0.0 0.0' shininess='1.0' specularColor='0.0 0.0 0.0'/></Appearance></
↪Shape>

</Transform>
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).root_lattice()
>>> options = L.plot_parse_options()
>>> list(options.text("coucou", [Integer(0),Integer(1)]))
[Text 'coucou' at the point (0.0,1.0)]
>>> list(options.text(L.simple_root(Integer(1)), [Integer(0),Integer(1)]))
[Text '$\alpha_{1}$' at the point (0.0,1.0)]
>>> list(options.text(L.simple_root(Integer(2)), [Integer(1),Integer(0)],␣
↪rgbcolor=(Integer(1),RealNumber('0.5'),Integer(0))))
[Text '$\alpha_{2}$' at the point (1.0,0.0)]

>>> options = L.plot_parse_options(labels=False)
>>> options.text("coucou", [Integer(0),Integer(1)])
0

>>> options = RootSystem(["B",Integer(3)]).root_lattice().plot_parse_options()
>>> print(options.text("coucou", [Integer(0),Integer(1),Integer(2)]).x3d_
↪str())
<Transform translation='0 1 2'>
<Shape><Text string='coucou' solid='true'/><Appearance><Material diffuseColor=
↪'0.0 0.0 0.0' shininess='1.0' specularColor='0.0 0.0 0.0'/></Appearance></
↪Shape>
<BLANKLINE>
</Transform>
```

thickness(*i*)

Return the thickness to be used for lines indexed by i.

INPUT:

- `i` – an index

See also

index_of_object()

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).root_lattice()
sage: options = L.plot_parse_options(labels=False)
sage: alpha = L.simple_roots()
sage: options.thickness(0)
2
sage: options.thickness(1)
1
sage: options.thickness(2)
1
sage: for alpha in L.simple_roots():
....:     print("{} {}".format(alpha, options.thickness(alpha)))
alpha[0] 2
alpha[1] 1
alpha[2] 1
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).root_lattice()
>>> options = L.plot_parse_options(labels=False)
>>> alpha = L.simple_roots()
>>> options.thickness(Integer(0))
2
>>> options.thickness(Integer(1))
1
>>> options.thickness(Integer(2))
1
>>> for alpha in L.simple_roots():
...     print("{} {}".format(alpha, options.thickness(alpha)))
alpha[0] 2
alpha[1] 1
alpha[2] 1
```

sage.combinat.root_system.plot.**barycentric_projection_matrix**(*angle=0*)

Return a family of $n+1$ vectors evenly spaced in a real vector space of dimension n.

Those vectors are of norm 1, the scalar product between any two vector is $1/n$, thus the distance between two tips is constant.

The family is built recursively and uniquely determined by the following property: the last vector is $(0,\dots,0,-1)$, and the projection of the first n vectors in dimension $n-1$, after appropriate rescaling to norm 1, retrieves the family for $n-1$.

OUTPUT:

A matrix with $n+1$ columns of height n with rational or symbolic coefficients.

EXAMPLES:

One vector in dimension 0:

```
sage: from sage.combinat.root_system.root_lattice_realizations import barycentric_
↪projection_matrix
sage: m = barycentric_projection_matrix(0); m
[]
sage: matrix(QQ,0,1).nrows()
0
sage: matrix(QQ,0,1).ncols()
1
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.root_lattice_realizations import barycentric_
↪projection_matrix
>>> m = barycentric_projection_matrix(Integer(0)); m
[]
>>> matrix(QQ,Integer(0),Integer(1)).nrows()
0
>>> matrix(QQ,Integer(0),Integer(1)).ncols()
1
```

Two vectors in dimension 1:

```
sage: barycentric_projection_matrix(1)
[ 1 -1]
```

```
>>> from sage.all import *
>>> barycentric_projection_matrix(Integer(1))
[ 1 -1]
```

Three vectors in dimension 2:

```
sage: barycentric_projection_matrix(2)
[ 1/2*sqrt(3) -1/2*sqrt(3)            0]
[         1/2          1/2           -1]
```

```
>>> from sage.all import *
>>> barycentric_projection_matrix(Integer(2))
[ 1/2*sqrt(3) -1/2*sqrt(3)            0]
[         1/2          1/2           -1]
```

Four vectors in dimension 3:

```
sage: m = barycentric_projection_matrix(3); m
[ 1/3*sqrt(3)*sqrt(2) -1/3*sqrt(3)*sqrt(2)                    0                    ␣
↪ 0]
[         1/3*sqrt(2)          1/3*sqrt(2)         -2/3*sqrt(2)                    ␣
↪ 0]
[                 1/3                  1/3                  1/3                    ␣
↪-1]
```

```
>>> from sage.all import *
>>> m = barycentric_projection_matrix(Integer(3)); m
[ 1/3*sqrt(3)*sqrt(2) -1/3*sqrt(3)*sqrt(2)                    0                    ␣
```

(continues on next page)

(continued from previous page)

```
↪ 0]
[           1/3*sqrt(2)            1/3*sqrt(2)          -2/3*sqrt(2)                 ␣
↪ 0]
[                   1/3                    1/3                    1/3                ␣
↪-1]
```

The columns give four vectors that sum up to zero:

```
sage: sum(m.columns())
(0, 0, 0)
```

```
>>> from sage.all import *
>>> sum(m.columns())
(0, 0, 0)
```

and have regular mutual angles:

```
sage: m.transpose()*m
[   1 -1/3 -1/3 -1/3]
[-1/3    1 -1/3 -1/3]
[-1/3 -1/3    1 -1/3]
[-1/3 -1/3 -1/3    1]
```

```
>>> from sage.all import *
>>> m.transpose()*m
[   1 -1/3 -1/3 -1/3]
[-1/3    1 -1/3 -1/3]
[-1/3 -1/3    1 -1/3]
[-1/3 -1/3 -1/3    1]
```

Here is a plot of them:

```
sage: sum(arrow((0,0,0),x) for x in m.columns())
Graphics3d Object
```

```
>>> from sage.all import *
>>> sum(arrow((Integer(0),Integer(0),Integer(0)),x) for x in m.columns())
Graphics3d Object
```

For 2D drawings of root systems, it is desirable to rotate the result to match with the usual conventions:

```
sage: barycentric_projection_matrix(2, angle=2*pi/3)
[         1/2           -1          1/2]
[ 1/2*sqrt(3)            0 -1/2*sqrt(3)]
```

```
>>> from sage.all import *
>>> barycentric_projection_matrix(Integer(2), angle=Integer(2)*pi/Integer(3))
[         1/2           -1          1/2]
[ 1/2*sqrt(3)            0 -1/2*sqrt(3)]
```

5.1.236 Finite complex reflection groups

Let V be a finite-dimensional complex vector space. A reflection of V is an operator $r \in \mathrm{GL}(V)$ that has finite order and fixes pointwise a hyperplane in V.

For more definitions and classification types of finite complex reflection groups, see Wikipedia article Complex_reflection_group.

The point of entry to work with reflection groups is `ReflectionGroup()` which can be used with finite Cartan-Killing types:

```
sage: ReflectionGroup(['A',2])
Irreducible real reflection group of rank 2 and type A2
sage: ReflectionGroup(['F',4])
Irreducible real reflection group of rank 4 and type F4
sage: ReflectionGroup(['H',3])
Irreducible real reflection group of rank 3 and type H3
```

```
>>> from sage.all import *
>>> ReflectionGroup(['A',Integer(2)])
Irreducible real reflection group of rank 2 and type A2
>>> ReflectionGroup(['F',Integer(4)])
Irreducible real reflection group of rank 4 and type F4
>>> ReflectionGroup(['H',Integer(3)])
Irreducible real reflection group of rank 3 and type H3
```

or with Shephard-Todd types:

```
sage: ReflectionGroup((1,1,3))
Irreducible real reflection group of rank 2 and type A2
sage: ReflectionGroup((2,1,3))
Irreducible real reflection group of rank 3 and type B3
sage: ReflectionGroup((3,1,3))
Irreducible complex reflection group of rank 3 and type G(3,1,3)
sage: ReflectionGroup((4,2,3))
Irreducible complex reflection group of rank 3 and type G(4,2,3)
sage: ReflectionGroup(4)
Irreducible complex reflection group of rank 2 and type ST4
sage: ReflectionGroup(31)
Irreducible complex reflection group of rank 4 and type ST31
```

```
>>> from sage.all import *
>>> ReflectionGroup((Integer(1),Integer(1),Integer(3)))
Irreducible real reflection group of rank 2 and type A2
>>> ReflectionGroup((Integer(2),Integer(1),Integer(3)))
Irreducible real reflection group of rank 3 and type B3
>>> ReflectionGroup((Integer(3),Integer(1),Integer(3)))
Irreducible complex reflection group of rank 3 and type G(3,1,3)
>>> ReflectionGroup((Integer(4),Integer(2),Integer(3)))
Irreducible complex reflection group of rank 3 and type G(4,2,3)
>>> ReflectionGroup(Integer(4))
Irreducible complex reflection group of rank 2 and type ST4
```

(continues on next page)

(continued from previous page)

```
>>> ReflectionGroup(Integer(31))
Irreducible complex reflection group of rank 4 and type ST31
```

Also reducible types are allowed using concatenation:

```
sage: ReflectionGroup(['A',3],(4,2,3))
Reducible complex reflection group of rank 6 and type A3 x G(4,2,3)
```

```
>>> from sage.all import *
>>> ReflectionGroup(['A',Integer(3)],(Integer(4),Integer(2),Integer(3)))
Reducible complex reflection group of rank 6 and type A3 x G(4,2,3)
```

Some special cases also occur, among them are:

```
sage: W = ReflectionGroup((2,2,2)); W
Reducible real reflection group of rank 2 and type A1 x A1
sage: W = ReflectionGroup((2,2,3)); W
Irreducible real reflection group of rank 3 and type A3
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(2),Integer(2),Integer(2))); W
Reducible real reflection group of rank 2 and type A1 x A1
>>> W = ReflectionGroup((Integer(2),Integer(2),Integer(3))); W
Irreducible real reflection group of rank 3 and type A3
```

Warning

Uses the GAP3 package *Chevie* which is available as an experimental package (installed by `sage -i gap3`) or to download by hand from Jean Michel's website.

A guided tour

We start with the example type B_2:

```
sage: W = ReflectionGroup(['B',2]); W
Irreducible real reflection group of rank 2 and type B2
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['B',Integer(2)]); W
Irreducible real reflection group of rank 2 and type B2
```

Most importantly, observe that the group elements are usually represented by permutations of the roots:

```
sage: for w in W: print(w)
()
(1,3)(2,6)(5,7)
(1,5)(2,4)(6,8)
(1,7,5,3)(2,4,6,8)
(1,3,5,7)(2,8,6,4)
(2,8)(3,7)(4,6)
```

(continues on next page)

(continued from previous page)

```
(1,7)(3,5)(4,8)
(1,5)(2,6)(3,7)(4,8)
```

```
>>> from sage.all import *
>>> for w in W: print(w)
()
(1,3)(2,6)(5,7)
(1,5)(2,4)(6,8)
(1,7,5,3)(2,4,6,8)
(1,3,5,7)(2,8,6,4)
(2,8)(3,7)(4,6)
(1,7)(3,5)(4,8)
(1,5)(2,6)(3,7)(4,8)
```

This has the drawback that one can hardly see anything. Usually, one would look at elements with either of the following methods:

```
sage: for w in W: w.reduced_word()
[]
[2]
[1]
[1, 2]
[2, 1]
[2, 1, 2]
[1, 2, 1]
[1, 2, 1, 2]

sage: for w in W: w.reduced_word_in_reflections()
[]
[2]
[1]
[1, 2]
[1, 4]
[3]
[4]
[1, 3]

sage: for w in W: w.reduced_word(); w.to_matrix(); print("")
[]
[1 0]
[0 1]

[2]
[ 1  1]
[ 0 -1]

[1]
[-1  0]
[ 2  1]

[1, 2]
[-1 -1]
```

(continues on next page)

(continued from previous page)

```
[ 2  1]

[2, 1]
[ 1  1]
[-2 -1]

[2, 1, 2]
[ 1  0]
[-2 -1]

[1, 2, 1]
[-1 -1]
[ 0  1]

[1, 2, 1, 2]
[-1  0]
[ 0 -1]
```

```
>>> from sage.all import *
>>> for w in W: w.reduced_word()
[]
[2]
[1]
[1, 2]
[2, 1]
[2, 1, 2]
[1, 2, 1]
[1, 2, 1, 2]

>>> for w in W: w.reduced_word_in_reflections()
[]
[2]
[1]
[1, 2]
[1, 4]
[3]
[4]
[1, 3]

>>> for w in W: w.reduced_word(); w.to_matrix(); print("")
[]
[1 0]
[0 1]
<BLANKLINE>
[2]
[ 1  1]
[ 0 -1]
<BLANKLINE>
[1]
[-1  0]
[ 2  1]
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
[1, 2]
[-1 -1]
[ 2  1]
<BLANKLINE>
[2, 1]
[ 1  1]
[-2 -1]
<BLANKLINE>
[2, 1, 2]
[ 1  0]
[-2 -1]
<BLANKLINE>
[1, 2, 1]
[-1 -1]
[ 0  1]
<BLANKLINE>
[1, 2, 1, 2]
[-1  0]
[ 0 -1]
<BLANKLINE>
```

The standard references for actions of complex reflection groups have the matrices acting on the right, so:

```
sage: W.simple_reflection(1).to_matrix()
[-1  0]
[ 2  1]
```

```
>>> from sage.all import *
>>> W.simple_reflection(Integer(1)).to_matrix()
[-1  0]
[ 2  1]
```

sends the simple root α_0, or `(1,0)` in vector notation, to its negative, while sending α_1 to $2\alpha_0 + \alpha_1$.

Todo

- properly provide root systems for real reflection groups
- element class should be unique to be able to work with large groups without creating elements multiple times
- `is_shephard_group`, `is_generalized_coxeter_group`
- exponents and coexponents
- coinvariant ring:
  - fake degrees from Torsten Hoge
  - operation of linear characters on all characters
  - harmonic polynomials
- linear forms for hyperplanes
- field of definition
- intersection lattice and characteristic polynomial:

```
X = [ alpha(t) for t in W.distinguished_reflections() ]
X = Matrix(CF,X).transpose()
Y = Matroid(X)
```

- linear characters
- permutation pi on irreducibles
- hyperplane orbits (76.13 in Gap Manual)
- improve invariant_form with a code similar to the one in `reflection_group_real.py`
- add a method `reflection_to_root` or `distinguished_reflection_to_positive_root`
- diagrams in ASCII-art (76.15)
- standard (BMR) presentations
- character table directly from Chevie
- `GenericOrder` (76.20), `TorusOrder` (76.21)
- correct fundamental invariants for G_34, check the others
- copy hardcoded data (degrees, invariants, braid relations…) into sage
- add other hardcoded data from the tables in chevie (location is SAGEDIR/local/gap3/gap-jm5-2015-02-01/gap3/pkg/chevie/tbl): basic derivations, discriminant, …
- transfer code for `reduced_word_in_reflections` into Gap4 or Sage
- list of reduced words for an element
- list of reduced words in reflections for an element
- Hurwitz action?
- `is_crystallographic()` should be hardcoded

AUTHORS:

- Christian Stump (2015): initial version

class sage.combinat.root_system.reflection_group_complex.**ComplexReflectionGroup**(*W_types*, *index_set=None*, *hyperplane_index_set=None*, *reflection_index_set=None*)

Bases: `UniqueRepresentation`, `PermutationGroup_generic`

A complex reflection group given as a permutation group.

See also

ReflectionGroup()

class Element

Bases: `ComplexReflectionGroupElement`

conjugacy_class()

Return the conjugacy class of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for w in W: sorted(w.conjugacy_class())
[()]
[(1,3)(2,5)(4,6), (1,4)(2,3)(5,6), (1,5)(2,4)(3,6)]
[(1,3)(2,5)(4,6), (1,4)(2,3)(5,6), (1,5)(2,4)(3,6)]
[(1,2,6)(3,4,5), (1,6,2)(3,5,4)]
[(1,2,6)(3,4,5), (1,6,2)(3,5,4)]
[(1,3)(2,5)(4,6), (1,4)(2,3)(5,6), (1,5)(2,4)(3,6)]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for w in W: sorted(w.conjugacy_class())
[()]
[(1,3)(2,5)(4,6), (1,4)(2,3)(5,6), (1,5)(2,4)(3,6)]
[(1,3)(2,5)(4,6), (1,4)(2,3)(5,6), (1,5)(2,4)(3,6)]
[(1,2,6)(3,4,5), (1,6,2)(3,5,4)]
[(1,2,6)(3,4,5), (1,6,2)(3,5,4)]
[(1,3)(2,5)(4,6), (1,4)(2,3)(5,6), (1,5)(2,4)(3,6)]
```

conjugacy_class_representative()

Return a representative of the conjugacy class of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for w in W:
....:     print('%s %s'%(w.reduced_word(), w.conjugacy_class_
↪representative().reduced_word()))
[] []
[2] [1]
[1] [1]
[1, 2] [1, 2]
[2, 1] [1, 2]
[1, 2, 1] [1]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for w in W:
...     print('%s %s'%(w.reduced_word(), w.conjugacy_class_
↪representative().reduced_word()))
[] []
[2] [1]
[1] [1]
[1, 2] [1, 2]
[2, 1] [1, 2]
[1, 2, 1] [1]
```

reflection_length(*in_unitary_group=False*)

Return the reflection length of `self`.

This is the minimal numbers of reflections needed to obtain `self`.

INPUT:

- `in_unitary_group` – boolean (default: `False`); if `True`, the reflection length is computed in the unitary group which is the dimension of the move space of `self`

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 2, 2]

sage: W = ReflectionGroup((2,1,2))
sage: sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 1, 2, 2, 2]

sage: W = ReflectionGroup((2,2,2))
sage: sorted([t.reflection_length() for t in W])
[0, 1, 1, 2]

sage: W = ReflectionGroup((3,1,2))
sage: sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 2, 2]

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(2)))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 1, 2, 2, 2]

>>> W = ReflectionGroup((Integer(2),Integer(2),Integer(2)))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 2]

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

apply_vector_field(*f*, *vf=None*)

Return a rational function obtained by applying the vector field `vf` to the rational function `f`.

If `vf` is not given, the primitive vector field is used.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: for x in W.primitive_vector_field()[0].parent().gens():
....:     print(W.apply_vector_field(x))
3*x1/(6*x0^2 - 6*x0*x1 - 12*x1^2)
1/(6*x0^2 - 6*x0*x1 - 12*x1^2)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> for x in W.primitive_vector_field()[Integer(0)].parent().gens():
...     print(W.apply_vector_field(x))
3*x1/(6*x0^2 - 6*x0*x1 - 12*x1^2)
1/(6*x0^2 - 6*x0*x1 - 12*x1^2)
```

braid_relations()

Return the braid relations of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.braid_relations()
[[[1, 2, 1], [2, 1, 2]]]

sage: W = ReflectionGroup((2,1,3))
sage: W.braid_relations()
[[[1, 2, 1, 2], [2, 1, 2, 1]], [[1, 3], [3, 1]], [[2, 3, 2], [3, 2, 3]]]

sage: W = ReflectionGroup((2,2,3))
sage: W.braid_relations()
[[[1, 2, 1], [2, 1, 2]], [[1, 3], [3, 1]], [[2, 3, 2], [3, 2, 3]]]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.braid_relations()
[[[1, 2, 1], [2, 1, 2]]]

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(3)))
>>> W.braid_relations()
[[[1, 2, 1, 2], [2, 1, 2, 1]], [[1, 3], [3, 1]], [[2, 3, 2], [3, 2, 3]]]

>>> W = ReflectionGroup((Integer(2),Integer(2),Integer(3)))
>>> W.braid_relations()
[[[1, 2, 1], [2, 1, 2]], [[1, 3], [3, 1]], [[2, 3, 2], [3, 2, 3]]]
```

cartan_matrix()

Return the Cartan matrix associated with `self`.

If `self` is crystallographic, the returned Cartan matrix is an instance of *CartanMatrix*, and a normal matrix otherwise.

Let $s_1, \ldots, s_n$ be a set of reflections which generate `self` with associated simple roots $s_1, \ldots, s_n$ and simple coroots $s_i^\vee$. Then the Cartan matrix $C = (c_{ij})$ is given by $s_i^\vee(s_j)$. The Cartan matrix completely determines the reflection representation if the s_i are linearly independent.

EXAMPLES:

```
sage: ReflectionGroup(['A',4]).cartan_matrix()
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

(continues on next page)

(continued from previous page)

```
sage: ReflectionGroup(['H',4]).cartan_matrix()
[              2 E(5)^2 + E(5)^3               0               0]
[E(5)^2 + E(5)^3               2              -1               0]
[              0              -1               2              -1]
[              0               0              -1               2]

sage: ReflectionGroup(4).cartan_matrix()
[-2*E(3) - E(3)^2           E(3)^2]
[         -E(3)^2 -2*E(3) - E(3)^2]

sage: ReflectionGroup((4,2,2)).cartan_matrix()
[       2  -2*E(4)       -2]
[    E(4)        2 1 - E(4)]
[      -1 1 + E(4)        2]
```

```
>>> from sage.all import *
>>> ReflectionGroup(['A',Integer(4)]).cartan_matrix()
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]

>>> ReflectionGroup(['H',Integer(4)]).cartan_matrix()
[              2 E(5)^2 + E(5)^3               0               0]
[E(5)^2 + E(5)^3               2              -1               0]
[              0              -1               2              -1]
[              0               0              -1               2]

>>> ReflectionGroup(Integer(4)).cartan_matrix()
[-2*E(3) - E(3)^2           E(3)^2]
[         -E(3)^2 -2*E(3) - E(3)^2]

>>> ReflectionGroup((Integer(4),Integer(2),Integer(2))).cartan_matrix()
[       2  -2*E(4)       -2]
[    E(4)        2 1 - E(4)]
[      -1 1 + E(4)        2]
```

codegrees()

Return the codegrees of `self` ordered within each irreducible component of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,4))
sage: W.codegrees()
(2, 1, 0)

sage: W = ReflectionGroup((2,1,4))
sage: W.codegrees()
(6, 4, 2, 0)

sage: W = ReflectionGroup((4,1,4))
```

(continues on next page)

(continued from previous page)

```
sage: W.codegrees()
(12, 8, 4, 0)

sage: W = ReflectionGroup((4,2,4))
sage: W.codegrees()
(12, 8, 4, 0)

sage: W = ReflectionGroup((4,4,4))
sage: W.codegrees()
(8, 8, 4, 0)

sage: W = ReflectionGroup((1,1,4), (3,1,2))
sage: W.codegrees()
(2, 1, 0, 3, 0)

sage: W = ReflectionGroup((1,1,4), (6,1,12), 23)
sage: W.codegrees()
(2, 1, 0, 66, 60, 54, 48, 42, 36, 30, 24, 18, 12, 6, 0, 8, 4, 0)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
>>> W.codegrees()
(2, 1, 0)

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(4)))
>>> W.codegrees()
(6, 4, 2, 0)

>>> W = ReflectionGroup((Integer(4),Integer(1),Integer(4)))
>>> W.codegrees()
(12, 8, 4, 0)

>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(4)))
>>> W.codegrees()
(12, 8, 4, 0)

>>> W = ReflectionGroup((Integer(4),Integer(4),Integer(4)))
>>> W.codegrees()
(8, 8, 4, 0)

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(3),
↪Integer(1),Integer(2)))
>>> W.codegrees()
(2, 1, 0, 3, 0)

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(6),
↪Integer(1),Integer(12)), Integer(23))
>>> W.codegrees()
(2, 1, 0, 66, 60, 54, 48, 42, 36, 30, 24, 18, 12, 6, 0, 8, 4, 0)
```

conjugacy_classes()

Return the conjugacy classes of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for C in W.conjugacy_classes(): sorted(C)
[()]
[(1,3)(2,5)(4,6), (1,4)(2,3)(5,6), (1,5)(2,4)(3,6)]
[(1,2,6)(3,4,5), (1,6,2)(3,5,4)]

sage: W = ReflectionGroup((1,1,4))
sage: sum(len(C) for C in W.conjugacy_classes()) == W.cardinality()
True

sage: W = ReflectionGroup((3,1,2))
sage: sum(len(C) for C in W.conjugacy_classes()) == W.cardinality()
True

sage: W = ReflectionGroup(23)
sage: sum(len(C) for C in W.conjugacy_classes()) == W.cardinality()
True
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for C in W.conjugacy_classes(): sorted(C)
[()]
[(1,3)(2,5)(4,6), (1,4)(2,3)(5,6), (1,5)(2,4)(3,6)]
[(1,2,6)(3,4,5), (1,6,2)(3,5,4)]

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
>>> sum(len(C) for C in W.conjugacy_classes()) == W.cardinality()
True

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> sum(len(C) for C in W.conjugacy_classes()) == W.cardinality()
True

>>> W = ReflectionGroup(Integer(23))
>>> sum(len(C) for C in W.conjugacy_classes()) == W.cardinality()
True
```

conjugacy_classes_representatives()

Return the shortest representatives of the conjugacy classes of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: [w.reduced_word() for w in W.conjugacy_classes_representatives()]
[[], [1], [1, 2]]

sage: W = ReflectionGroup((1,1,4))
sage: [w.reduced_word() for w in W.conjugacy_classes_representatives()]
[[], [1], [1, 3], [1, 2], [1, 3, 2]]

sage: W = ReflectionGroup((3,1,2))
sage: [w.reduced_word() for w in W.conjugacy_classes_representatives()]
```

(continues on next page)

(continued from previous page)

```
[[], [1], [1, 1], [2, 1, 2, 1], [2, 1, 2, 1, 1],
 [2, 1, 1, 2, 1, 1], [2], [1, 2], [1, 1, 2]]

sage: W = ReflectionGroup(23)
sage: [w.reduced_word() for w in W.conjugacy_classes_representatives()]
    [[],
     [1],
     [1, 2],
     [1, 3],
     [2, 3],
     [1, 2, 3],
     [1, 2, 1, 2],
     [1, 2, 1, 2, 3],
     [1, 2, 1, 2, 3, 2, 1, 2, 3],
     [1, 2, 1, 2, 1, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> [w.reduced_word() for w in W.conjugacy_classes_representatives()]
[[], [1], [1, 2]]

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
>>> [w.reduced_word() for w in W.conjugacy_classes_representatives()]
[[], [1], [1, 3], [1, 2], [1, 3, 2]]

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> [w.reduced_word() for w in W.conjugacy_classes_representatives()]
[[], [1], [1, 1], [2, 1, 2, 1], [2, 1, 2, 1, 1],
 [2, 1, 1, 2, 1, 1], [2], [1, 2], [1, 1, 2]]

>>> W = ReflectionGroup(Integer(23))
>>> [w.reduced_word() for w in W.conjugacy_classes_representatives()]
    [[],
     [1],
     [1, 2],
     [1, 3],
     [2, 3],
     [1, 2, 3],
     [1, 2, 1, 2],
     [1, 2, 1, 2, 3],
     [1, 2, 1, 2, 3, 2, 1, 2, 3],
     [1, 2, 1, 2, 1, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]]
```

coxeter_number(*chi=None*)

Return the Coxeter number associated to the irreducible character chi of the reflection group `self`.

The *Coxeter number* of a complex reflection group W is the trace in a character χ of $\sum_t (Id - t)$, where t runs over all reflections. The result is always an integer.

When χ is the reflection representation, the Coxeter number is equal to $\frac{N+N^*}{n}$ where N is the number of reflections, N^* is the number of reflection hyperplanes, and n is the rank of W. If W is further well-generated, the Coxeter number is equal to the highest degree d_n and to the order of a Coxeter element c of W.

EXAMPLES:

```
sage: W = ReflectionGroup(["H",4])
sage: W.coxeter_number()
30
sage: all(W.coxeter_number(chi).is_integer()
....:     for chi in W.irreducible_characters())
True
sage: W = ReflectionGroup(14)
sage: W.coxeter_number()
24
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(["H",Integer(4)])
>>> W.coxeter_number()
30
>>> all(W.coxeter_number(chi).is_integer()
...     for chi in W.irreducible_characters())
True
>>> W = ReflectionGroup(Integer(14))
>>> W.coxeter_number()
24
```

degrees()

Return the degrees of `self` ordered within each irreducible component of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,4))
sage: W.degrees()
(2, 3, 4)

sage: W = ReflectionGroup((2,1,4))
sage: W.degrees()
(2, 4, 6, 8)

sage: W = ReflectionGroup((4,1,4))
sage: W.degrees()
(4, 8, 12, 16)

sage: W = ReflectionGroup((4,2,4))
sage: W.degrees()
(4, 8, 8, 12)

sage: W = ReflectionGroup((4,4,4))
sage: W.degrees()
(4, 4, 8, 12)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
>>> W.degrees()
(2, 3, 4)

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(4)))
>>> W.degrees()
```

(continues on next page)

(continued from previous page)

```
(2, 4, 6, 8)

>>> W = ReflectionGroup((Integer(4),Integer(1),Integer(4)))
>>> W.degrees()
(4, 8, 12, 16)

>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(4)))
>>> W.degrees()
(4, 8, 8, 12)

>>> W = ReflectionGroup((Integer(4),Integer(4),Integer(4)))
>>> W.degrees()
(4, 4, 8, 12)
```

Examples of reducible types:

```
sage: W = ReflectionGroup((1,1,4), (3,1,2)); W
Reducible complex reflection group of rank 5 and type A3 x G(3,1,2)
sage: W.degrees()
(2, 3, 4, 3, 6)

sage: W = ReflectionGroup((1,1,4), (6,1,12), 23)
sage: W.degrees()
(2, 3, 4, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 2, 6, 10)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(3),
↪Integer(1),Integer(2))); W
Reducible complex reflection group of rank 5 and type A3 x G(3,1,2)
>>> W.degrees()
(2, 3, 4, 3, 6)

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(6),
↪Integer(1),Integer(12)), Integer(23))
>>> W.degrees()
(2, 3, 4, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 2, 6, 10)
```

discriminant()

Return the discriminant of `self` in the polynomial ring on which the group acts.

This is the product

$$\prod_H \alpha_H^{e_H},$$

where α_H is the linear form of the hyperplane H and e_H is its stabilizer order.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: W.discriminant()
x0^6 - 3*x0^5*x1 - 3/4*x0^4*x1^2 + 13/2*x0^3*x1^3
 - 3/4*x0^2*x1^4 - 3*x0*x1^5 + x1^6
```

(continues on next page)

(continued from previous page)

```
sage: W = ReflectionGroup(['B',2])
sage: W.discriminant()
x0^6*x1^2 - 6*x0^5*x1^3 + 13*x0^4*x1^4 - 12*x0^3*x1^5 + 4*x0^2*x1^6
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> W.discriminant()
x0^6 - 3*x0^5*x1 - 3/4*x0^4*x1^2 + 13/2*x0^3*x1^3
 - 3/4*x0^2*x1^4 - 3*x0*x1^5 + x1^6

>>> W = ReflectionGroup(['B',Integer(2)])
>>> W.discriminant()
x0^6*x1^2 - 6*x0^5*x1^3 + 13*x0^4*x1^4 - 12*x0^3*x1^5 + 4*x0^2*x1^6
```

discriminant_in_invariant_ring(*invariants=None*)

Return the discriminant of `self` in the invariant ring.

This is the function f in the invariants such that $f(F_1(x), \ldots, F_n(x))$ is the discriminant.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3])
sage: W.discriminant_in_invariant_ring()
6*t0^3*t1^2 - 18*t0^4*t2 + 9*t1^4 - 36*t0*t1^2*t2 + 24*t0^2*t2^2 - 8*t2^3

sage: W = ReflectionGroup(['B',3])
sage: W.discriminant_in_invariant_ring()
-t0^2*t1^2*t2 + 16*t0^3*t2^2 + 2*t1^3*t2 - 36*t0*t1*t2^2 + 108*t2^3

sage: W = ReflectionGroup(['H',3])
sage: W.discriminant_in_invariant_ring()     # long time
(-829*E(5) - 1658*E(5)^2 - 1658*E(5)^3 - 829*E(5)^4)*t0^15
 + (213700*E(5) + 427400*E(5)^2 + 427400*E(5)^3 + 213700*E(5)^4)*t0^12*t1
 + (-22233750*E(5) - 44467500*E(5)^2 - 44467500*E(5)^3 - 22233750*E(5)^4)*t0^
↪9*t1^2
 + (438750*E(5) + 877500*E(5)^2 + 877500*E(5)^3 + 438750*E(5)^4)*t0^10*t2
 + (1162187500*E(5) + 2324375000*E(5)^2 + 2324375000*E(5)^3 + 1162187500*E(5)^
↪4)*t0^6*t1^3
 + (-74250000*E(5) - 148500000*E(5)^2 - 148500000*E(5)^3 - 74250000*E(5)^
↪4)*t0^7*t1*t2
 + (-28369140625*E(5) - 56738281250*E(5)^2 - 56738281250*E(5)^3 -␣
↪28369140625*E(5)^4)*t0^3*t1^4
 + (1371093750*E(5) + 2742187500*E(5)^2 + 2742187500*E(5)^3 + 1371093750*E(5)^
↪4)*t0^4*t1^2*t2
 + (1191796875*E(5) + 2383593750*E(5)^2 + 2383593750*E(5)^3 + 1191796875*E(5)^
↪4)*t0^5*t2^2
 + (175781250000*E(5) + 351562500000*E(5)^2 + 351562500000*E(5)^3 +␣
↪175781250000*E(5)^4)*t1^5
 + (131835937500*E(5) + 263671875000*E(5)^2 + 263671875000*E(5)^3 +␣
↪131835937500*E(5)^4)*t0*t1^3*t2
 + (-100195312500*E(5) - 200390625000*E(5)^2 - 200390625000*E(5)^3 -␣
↪100195312500*E(5)^4)*t0^2*t1*t2^2
 + (395507812500*E(5) + 791015625000*E(5)^2 + 791015625000*E(5)^3 +␣
```

(continues on next page)

(continued from previous page)

```
↪395507812500*E(5)^4)*t2^3
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)])
>>> W.discriminant_in_invariant_ring()
6*t0^3*t1^2 - 18*t0^4*t2 + 9*t1^4 - 36*t0*t1^2*t2 + 24*t0^2*t2^2 - 8*t2^3

>>> W = ReflectionGroup(['B',Integer(3)])
>>> W.discriminant_in_invariant_ring()
-t0^2*t1^2*t2 + 16*t0^3*t2^2 + 2*t1^3*t2 - 36*t0*t1*t2^2 + 108*t2^3

>>> W = ReflectionGroup(['H',Integer(3)])
>>> W.discriminant_in_invariant_ring()     # long time
(-829*E(5) - 1658*E(5)^2 - 1658*E(5)^3 - 829*E(5)^4)*t0^15
 + (213700*E(5) + 427400*E(5)^2 + 427400*E(5)^3 + 213700*E(5)^4)*t0^12*t1
 + (-22233750*E(5) - 44467500*E(5)^2 - 44467500*E(5)^3 - 22233750*E(5)^4)*t0^
↪9*t1^2
 + (438750*E(5) + 877500*E(5)^2 + 877500*E(5)^3 + 438750*E(5)^4)*t0^10*t2
 + (1162187500*E(5) + 2324375000*E(5)^2 + 2324375000*E(5)^3 + 1162187500*E(5)^
↪4)*t0^6*t1^3
 + (-74250000*E(5) - 148500000*E(5)^2 - 148500000*E(5)^3 - 74250000*E(5)^
↪4)*t0^7*t1*t2
 + (-28369140625*E(5) - 56738281250*E(5)^2 - 56738281250*E(5)^3 -␣
↪28369140625*E(5)^4)*t0^3*t1^4
 + (1371093750*E(5) + 2742187500*E(5)^2 + 2742187500*E(5)^3 + 1371093750*E(5)^
↪4)*t0^4*t1^2*t2
 + (1191796875*E(5) + 2383593750*E(5)^2 + 2383593750*E(5)^3 + 1191796875*E(5)^
↪4)*t0^5*t2^2
 + (175781250000*E(5) + 351562500000*E(5)^2 + 351562500000*E(5)^3 +␣
↪175781250000*E(5)^4)*t1^5
 + (131835937500*E(5) + 263671875000*E(5)^2 + 263671875000*E(5)^3 +␣
↪131835937500*E(5)^4)*t0*t1^3*t2
 + (-100195312500*E(5) - 200390625000*E(5)^2 - 200390625000*E(5)^3 -␣
↪100195312500*E(5)^4)*t0^2*t1*t2^2
 + (395507812500*E(5) + 791015625000*E(5)^2 + 791015625000*E(5)^3 +␣
↪395507812500*E(5)^4)*t2^3
```

distinguished_reflection(*i*)

Return the `i`-th distinguished reflection of `self`.

These are the reflections in `self` acting on the complement of the fixed hyperplane H as $\exp(2\pi i/n)$, where n is the order of the reflection subgroup fixing H.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.distinguished_reflection(1)
(1,4)(2,3)(5,6)
sage: W.distinguished_reflection(2)
(1,3)(2,5)(4,6)
sage: W.distinguished_reflection(3)
(1,5)(2,4)(3,6)
```

(continues on next page)

(continued from previous page)

```
sage: W = ReflectionGroup((3,1,1),hyperplane_index_set=['a'])
sage: W.distinguished_reflection('a')
(1,2,3)

sage: W = ReflectionGroup((1,1,3),(3,1,2))
sage: for i in range(W.number_of_reflection_hyperplanes()):
....:     W.distinguished_reflection(i+1)
(1,6)(2,5)(7,8)
(1,5)(2,7)(6,8)
(3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,26)(27,28,30)
(3,11)(4,12)(9,13)(10,14)(15,19)(16,20)(17,21)(18,22)(23,27)(24,28)(25,26)(29,
→30)
(1,7)(2,6)(5,8)
(3,19)(4,25)(9,11)(10,17)(12,28)(13,15)(14,30)(16,18)(20,27)(21,29)(22,23)(24,
→26)
(4,21,27)(10,22,28)(11,13,19)(12,14,20)(16,26,30)(17,18,25)(23,24,29)
(3,13)(4,24)(9,19)(10,29)(11,15)(12,26)(14,21)(16,23)(17,30)(18,27)(20,22)(25,
→28)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.distinguished_reflection(Integer(1))
(1,4)(2,3)(5,6)
>>> W.distinguished_reflection(Integer(2))
(1,3)(2,5)(4,6)
>>> W.distinguished_reflection(Integer(3))
(1,5)(2,4)(3,6)

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(1)),hyperplane_index_
→set=['a'])
>>> W.distinguished_reflection('a')
(1,2,3)

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)),(Integer(3),
→Integer(1),Integer(2)))
>>> for i in range(W.number_of_reflection_hyperplanes()):
...     W.distinguished_reflection(i+Integer(1))
(1,6)(2,5)(7,8)
(1,5)(2,7)(6,8)
(3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,26)(27,28,30)
(3,11)(4,12)(9,13)(10,14)(15,19)(16,20)(17,21)(18,22)(23,27)(24,28)(25,26)(29,
→30)
(1,7)(2,6)(5,8)
(3,19)(4,25)(9,11)(10,17)(12,28)(13,15)(14,30)(16,18)(20,27)(21,29)(22,23)(24,
→26)
(4,21,27)(10,22,28)(11,13,19)(12,14,20)(16,26,30)(17,18,25)(23,24,29)
(3,13)(4,24)(9,19)(10,29)(11,15)(12,26)(14,21)(16,23)(17,30)(18,27)(20,22)(25,
→28)
```

distinguished_reflections()

Return a finite family containing the distinguished reflections of `self` indexed by *hyperplane_index_set()*.

These are the reflections in `self` acting on the complement of the fixed hyperplane H as $\exp(2\pi i/n)$, where n is the order of the reflection subgroup fixing H.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.distinguished_reflections()
Finite family {1: (1,4)(2,3)(5,6), 2: (1,3)(2,5)(4,6), 3: (1,5)(2,4)(3,6)}

sage: W = ReflectionGroup((1,1,3),hyperplane_index_set=['a','b','c'])
sage: W.distinguished_reflections()
Finite family {'a': (1,4)(2,3)(5,6), 'b': (1,3)(2,5)(4,6), 'c': (1,5)(2,4)(3,
→6)}

sage: W = ReflectionGroup((3,1,1))
sage: W.distinguished_reflections()
Finite family {1: (1,2,3)}

sage: W = ReflectionGroup((1,1,3),(3,1,2))
sage: W.distinguished_reflections()
Finite family {1: (1,6)(2,5)(7,8), 2: (1,5)(2,7)(6,8),
 3: (3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,26)(27,28,30),
 4: (3,11)(4,12)(9,13)(10,14)(15,19)(16,20)(17,21)(18,22)(23,27)(24,28)(25,
→26)(29,30),
 5: (1,7)(2,6)(5,8),
 6: (3,19)(4,25)(9,11)(10,17)(12,28)(13,15)(14,30)(16,18)(20,27)(21,29)(22,
→23)(24,26),
 7: (4,21,27)(10,22,28)(11,13,19)(12,14,20)(16,26,30)(17,18,25)(23,24,29),
 8: (3,13)(4,24)(9,19)(10,29)(11,15)(12,26)(14,21)(16,23)(17,30)(18,27)(20,
→22)(25,28)}
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.distinguished_reflections()
Finite family {1: (1,4)(2,3)(5,6), 2: (1,3)(2,5)(4,6), 3: (1,5)(2,4)(3,6)}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)),hyperplane_index_
→set=['a','b','c'])
>>> W.distinguished_reflections()
Finite family {'a': (1,4)(2,3)(5,6), 'b': (1,3)(2,5)(4,6), 'c': (1,5)(2,4)(3,
→6)}

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(1)))
>>> W.distinguished_reflections()
Finite family {1: (1,2,3)}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)),(Integer(3),
→Integer(1),Integer(2)))
>>> W.distinguished_reflections()
Finite family {1: (1,6)(2,5)(7,8), 2: (1,5)(2,7)(6,8),
 3: (3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,26)(27,28,30),
 4: (3,11)(4,12)(9,13)(10,14)(15,19)(16,20)(17,21)(18,22)(23,27)(24,28)(25,
→26)(29,30),
 5: (1,7)(2,6)(5,8),
```

(continues on next page)

(continued from previous page)

```
 6: (3,19)(4,25)(9,11)(10,17)(12,28)(13,15)(14,30)(16,18)(20,27)(21,29)(22,
↪23)(24,26),
 7: (4,21,27)(10,22,28)(11,13,19)(12,14,20)(16,26,30)(17,18,25)(23,24,29),
 8: (3,13)(4,24)(9,19)(10,29)(11,15)(12,26)(14,21)(16,23)(17,30)(18,27)(20,
↪22)(25,28)}
```

fake_degrees()

Return the list of the fake degrees associated to `self`.

The fake degrees are q-versions of the degree of the character. In particular, they sum to Hilbert series of the coinvariant algebra of `self`.

Note

The ordering follows the one in Chevie and is not compatible with the current implementation of `irredubile_characters()`.

EXAMPLES:

```
sage: W = ReflectionGroup(12)
sage: W.fake_degrees()
[1, q^12, q^11 + q, q^8 + q^4, q^7 + q^5, q^6 + q^4 + q^2,
 q^10 + q^8 + q^6, q^9 + q^7 + q^5 + q^3]

sage: W = ReflectionGroup(["H",4])
sage: W.cardinality()
14400
sage: sum(fdeg.subs(q=1)**2 for fdeg in W.fake_degrees())
14400
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(Integer(12))
>>> W.fake_degrees()
[1, q^12, q^11 + q, q^8 + q^4, q^7 + q^5, q^6 + q^4 + q^2,
 q^10 + q^8 + q^6, q^9 + q^7 + q^5 + q^3]

>>> W = ReflectionGroup(["H",Integer(4)])
>>> W.cardinality()
14400
>>> sum(fdeg.subs(q=Integer(1))**Integer(2) for fdeg in W.fake_degrees())
14400
```

fundamental_invariants()

Return the fundamental invariants of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.fundamental_invariants()
(-2*x0^2 + 2*x0*x1 - 2*x1^2, 6*x0^2*x1 - 6*x0*x1^2)

sage: W = ReflectionGroup((3,1,2))
```

(continues on next page)

(continued from previous page)

```
sage: W.fundamental_invariants()
(x0^3 + x1^3, x0^3*x1^3)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.fundamental_invariants()
(-2*x0^2 + 2*x0*x1 - 2*x1^2, 6*x0^2*x1 - 6*x0*x1^2)

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> W.fundamental_invariants()
(x0^3 + x1^3, x0^3*x1^3)
```

hyperplane_index_set()

Return the index set of the hyperplanes of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,4))
sage: W.hyperplane_index_set()
(1, 2, 3, 4, 5, 6)
sage: W = ReflectionGroup((1,1,4), hyperplane_index_set=[1,3,'asdf',7,9,11])
sage: W.hyperplane_index_set()
(1, 3, 'asdf', 7, 9, 11)
sage: W = ReflectionGroup((1,1,4),hyperplane_index_set=('a','b','c','d','e','f
→'))
sage: W.hyperplane_index_set()
('a', 'b', 'c', 'd', 'e', 'f')
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
>>> W.hyperplane_index_set()
(1, 2, 3, 4, 5, 6)
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), hyperplane_index_
→set=[Integer(1),Integer(3),'asdf',Integer(7),Integer(9),Integer(11)])
>>> W.hyperplane_index_set()
(1, 3, 'asdf', 7, 9, 11)
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)),hyperplane_index_
→set=('a','b','c','d','e','f'))
>>> W.hyperplane_index_set()
('a', 'b', 'c', 'd', 'e', 'f')
```

independent_roots()

Return a collection of simple roots generating the underlying vector space of `self`.

For well-generated groups, these are all simple roots. Otherwise, a linearly independent subset of the simple roots is chosen.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.independent_roots()
Finite family {1: (1, 0), 2: (0, 1)}
```

(continues on next page)

(continued from previous page)

```
sage: W = ReflectionGroup((4,2,3))
sage: W.simple_roots()
Finite family {1: (1, 0, 0), 2: (-E(4), 1, 0), 3: (-1, 1, 0), 4: (0, -1, 1)}
sage: W.independent_roots()
Finite family {1: (1, 0, 0), 2: (-E(4), 1, 0), 4: (0, -1, 1)}
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.independent_roots()
Finite family {1: (1, 0), 2: (0, 1)}

>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(3)))
>>> W.simple_roots()
Finite family {1: (1, 0, 0), 2: (-E(4), 1, 0), 3: (-1, 1, 0), 4: (0, -1, 1)}
>>> W.independent_roots()
Finite family {1: (1, 0, 0), 2: (-E(4), 1, 0), 4: (0, -1, 1)}
```

index_set()

Return the index set of the simple reflections of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,4))
sage: W.index_set()
(1, 2, 3)
sage: W = ReflectionGroup((1,1,4), index_set=[1,3,'asdf'])
sage: W.index_set()
(1, 3, 'asdf')
sage: W = ReflectionGroup((1,1,4), index_set=('a', 'b', 'c'))
sage: W.index_set()
('a', 'b', 'c')
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
>>> W.index_set()
(1, 2, 3)
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), index_
↪set=[Integer(1),Integer(3),'asdf'])
>>> W.index_set()
(1, 3, 'asdf')
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), index_set=('a', 'b
↪', 'c'))
>>> W.index_set()
('a', 'b', 'c')
```

invariant_form(*brute_force=False*)

Return the form that is invariant under the action of `self`.

This is unique only up to a global scalar on the irreducible components.

INPUT:

- `brute_force` – if `True`, the computation is done by applying the Reynolds operator; this is, the invariant form of e_i and e_j is computed as the sum $\langle w(e_i), w(e_j)\rangle$, where $\langle\cdot,\cdot\rangle$ is the standard scalar product

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3])
sage: F = W.invariant_form(); F
[   1 -1/2    0]
[-1/2    1 -1/2]
[   0 -1/2    1]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)])
>>> F = W.invariant_form(); F
[   1 -1/2    0]
[-1/2    1 -1/2]
[   0 -1/2    1]
```

To check that this is indeed the invariant form, see:

```
sage: S = W.simple_reflections()
sage: all( F == S[i].matrix()*F*S[i].matrix().transpose() for i in W.index_
↪set() )
True

sage: W = ReflectionGroup(['B',3])
sage: F = W.invariant_form(); F
[ 1 -1  0]
[-1  2 -1]
[ 0 -1  2]
sage: w = W.an_element().to_matrix()
sage: w * F * w.transpose().conjugate() == F
True

sage: S = W.simple_reflections()
sage: all( F == S[i].matrix()*F*S[i].matrix().transpose() for i in W.index_
↪set() )
True

sage: W = ReflectionGroup((3,1,2))
sage: F = W.invariant_form(); F
[1 0]
[0 1]

sage: S = W.simple_reflections()
sage: all( F == S[i].matrix()*F*S[i].matrix().transpose().conjugate() for i␣
↪in W.index_set() )
True
```

```
>>> from sage.all import *
>>> S = W.simple_reflections()
>>> all( F == S[i].matrix()*F*S[i].matrix().transpose() for i in W.index_
↪set() )
True
```

(continues on next page)

(continued from previous page)

```
>>> W = ReflectionGroup(['B',Integer(3)])
>>> F = W.invariant_form(); F
[ 1 -1  0]
[-1  2 -1]
[ 0 -1  2]
>>> w = W.an_element().to_matrix()
>>> w * F * w.transpose().conjugate() == F
True

>>> S = W.simple_reflections()
>>> all( F == S[i].matrix()*F*S[i].matrix().transpose() for i in W.index_
↪set() )
True

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> F = W.invariant_form(); F
[1 0]
[0 1]

>>> S = W.simple_reflections()
>>> all( F == S[i].matrix()*F*S[i].matrix().transpose().conjugate() for i in␣
↪W.index_set() )
True
```

It also worked for badly generated groups:

```
sage: W = ReflectionGroup(7)
sage: W.is_well_generated()
False

sage: F = W.invariant_form(); F
[1 0]
[0 1]
sage: S = W.simple_reflections()
sage: all( F == S[i].matrix()*F*S[i].matrix().transpose().conjugate() for i␣
↪in W.index_set() )
True
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(Integer(7))
>>> W.is_well_generated()
False

>>> F = W.invariant_form(); F
[1 0]
[0 1]
>>> S = W.simple_reflections()
>>> all( F == S[i].matrix()*F*S[i].matrix().transpose().conjugate() for i in␣
↪W.index_set() )
True
```

And also for reducible types:

```
sage: W = ReflectionGroup(['B',3],(4,2,3),4,7); W
Reducible complex reflection group of rank 10 and type B3 x G(4,2,3) x ST4 x␣
↪ST7
sage: F = W.invariant_form(); S = W.simple_reflections()
sage: all( F == S[i].matrix()*F*S[i].matrix().transpose().conjugate() for i␣
↪in W.index_set() )
True
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['B',Integer(3)],(Integer(4),Integer(2),Integer(3)),
↪Integer(4),Integer(7)); W
Reducible complex reflection group of rank 10 and type B3 x G(4,2,3) x ST4 x␣
↪ST7
>>> F = W.invariant_form(); S = W.simple_reflections()
>>> all( F == S[i].matrix()*F*S[i].matrix().transpose().conjugate() for i in␣
↪W.index_set() )
True
```

invariant_form_standardization()

Return the transformation of the space that turns the invariant form of `self` into the standard scalar product.

Let I be the invariant form of a complex reflection group, and let A be the Hermitian matrix such that $A^2 = I$. The matrix A defines a change of basis such that the identity matrix is the invariant form. Indeed, we have

$$(A^{-1}xA)\mathcal{I}(A^{-1}yA)^* = A^{-1}xIy^*A^{-1} = A^{-1}IA^{-1} = \mathcal{I},$$

where $\mathcal{I}$ is the identity matrix.

EXAMPLES:

```
sage: W = ReflectionGroup((4,2,5))
sage: I = W.invariant_form()
sage: A = W.invariant_form_standardization()
sage: A^2 == I
True
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(5)))
>>> I = W.invariant_form()
>>> A = W.invariant_form_standardization()
>>> A**Integer(2) == I
True
```

irreducible_components()

Return a list containing the irreducible components of `self` as finite reflection groups.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.irreducible_components()
[Irreducible real reflection group of rank 2 and type A2]

sage: W = ReflectionGroup((1,1,3),(2,1,3))
sage: W.irreducible_components()
```

(continues on next page)

(continued from previous page)

```
[Irreducible real reflection group of rank 2 and type A2,
Irreducible real reflection group of rank 3 and type B3]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.irreducible_components()
[Irreducible real reflection group of rank 2 and type A2]

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)),(Integer(2),
↪Integer(1),Integer(3)))
>>> W.irreducible_components()
[Irreducible real reflection group of rank 2 and type A2,
Irreducible real reflection group of rank 3 and type B3]
```

is_crystallographic()

Return `True` if `self` is crystallographic.

This is, if the field of definition is the rational field.

Todo

Make this more robust and do not use the matrix representation of the simple reflections.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3)); W
Irreducible real reflection group of rank 2 and type A2
sage: W.is_crystallographic()
True

sage: W = ReflectionGroup((2,1,3)); W
Irreducible real reflection group of rank 3 and type B3
sage: W.is_crystallographic()
True

sage: W = ReflectionGroup(23); W
Irreducible real reflection group of rank 3 and type H3
sage: W.is_crystallographic()
False

sage: W = ReflectionGroup((3,1,3)); W
Irreducible complex reflection group of rank 3 and type G(3,1,3)
sage: W.is_crystallographic()
False

sage: W = ReflectionGroup((4,2,2)); W
Irreducible complex reflection group of rank 2 and type G(4,2,2)
sage: W.is_crystallographic()
False
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3))); W
Irreducible real reflection group of rank 2 and type A2
>>> W.is_crystallographic()
True

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(3))); W
Irreducible real reflection group of rank 3 and type B3
>>> W.is_crystallographic()
True

>>> W = ReflectionGroup(Integer(23)); W
Irreducible real reflection group of rank 3 and type H3
>>> W.is_crystallographic()
False

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(3))); W
Irreducible complex reflection group of rank 3 and type G(3,1,3)
>>> W.is_crystallographic()
False

>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(2))); W
Irreducible complex reflection group of rank 2 and type G(4,2,2)
>>> W.is_crystallographic()
False
```

iteration_tracking_words()

Return an iterator going through all elements in `self` that tracks the reduced expressions.

This can be much slower than using the iteration as a permutation group with strong generating set.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for w in W.iteration_tracking_words(): w
()
(1,4)(2,3)(5,6)
(1,3)(2,5)(4,6)
(1,6,2)(3,5,4)
(1,2,6)(3,4,5)
(1,5)(2,4)(3,6)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for w in W.iteration_tracking_words(): w
()
(1,4)(2,3)(5,6)
(1,3)(2,5)(4,6)
(1,6,2)(3,5,4)
(1,2,6)(3,4,5)
(1,5)(2,4)(3,6)
```

jacobian_of_fundamental_invariants(*invs=None*)

Return the matrix $[\partial_{x_i} F_j]$, where `invs` are are any polynomials $F_1, \ldots, F_n$ in $x_1, \ldots, x_n$.

INPUT:

- `invs` – (default: the fundamental invariants) the polynomials $F_1, \ldots, F_n$

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: W.fundamental_invariants()
(-2*x0^2 + 2*x0*x1 - 2*x1^2, 6*x0^2*x1 - 6*x0*x1^2)

sage: W.jacobian_of_fundamental_invariants()
[     -4*x0 + 2*x1       2*x0 - 4*x1]
[12*x0*x1 - 6*x1^2 6*x0^2 - 12*x0*x1]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> W.fundamental_invariants()
(-2*x0^2 + 2*x0*x1 - 2*x1^2, 6*x0^2*x1 - 6*x0*x1^2)

>>> W.jacobian_of_fundamental_invariants()
[     -4*x0 + 2*x1       2*x0 - 4*x1]
[12*x0*x1 - 6*x1^2 6*x0^2 - 12*x0*x1]
```

number_of_irreducible_components()

Return the number of irreducible components of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.number_of_irreducible_components()
1

sage: W = ReflectionGroup((1,1,3),(2,1,3))
sage: W.number_of_irreducible_components()
2
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.number_of_irreducible_components()
1

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)),(Integer(2),
↪Integer(1),Integer(3)))
>>> W.number_of_irreducible_components()
2
```

primitive_vector_field(*invs=None*)

Return the primitive vector field of `self` is irreducible and well-generated.

The primitive vector field is given as the coefficients (being rational functions) in the basis $\partial_{x_1}, \ldots, \partial_{x_n}$.

This is the partial derivation along the unique invariant of degree given by the Coxeter number. It can be computed as the row of the inverse of the Jacobian given by the highest degree.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: W.primitive_vector_field()
(3*x1/(6*x0^2 - 6*x0*x1 - 12*x1^2), 1/(6*x0^2 - 6*x0*x1 - 12*x1^2))
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> W.primitive_vector_field()
(3*x1/(6*x0^2 - 6*x0*x1 - 12*x1^2), 1/(6*x0^2 - 6*x0*x1 - 12*x1^2))
```

rank()

Return the rank of `self`.

This is the dimension of the underlying vector space.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.rank()
2
sage: W = ReflectionGroup((2,1,3))
sage: W.rank()
3
sage: W = ReflectionGroup((4,1,3))
sage: W.rank()
3
sage: W = ReflectionGroup((4,2,3))
sage: W.rank()
3
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.rank()
2
>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(3)))
>>> W.rank()
3
>>> W = ReflectionGroup((Integer(4),Integer(1),Integer(3)))
>>> W.rank()
3
>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(3)))
>>> W.rank()
3
```

reflection(*i*)

Return the `i`-th reflection of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.reflection(1)
(1,4)(2,3)(5,6)
sage: W.reflection(2)
(1,3)(2,5)(4,6)
sage: W.reflection(3)
```

(continues on next page)

(continued from previous page)

```
(1,5)(2,4)(3,6)

sage: W = ReflectionGroup((3,1,1),reflection_index_set=['a','b'])
sage: W.reflection('a')
(1,2,3)
sage: W.reflection('b')
(1,3,2)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.reflection(Integer(1))
(1,4)(2,3)(5,6)
>>> W.reflection(Integer(2))
(1,3)(2,5)(4,6)
>>> W.reflection(Integer(3))
(1,5)(2,4)(3,6)

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(1)),reflection_index_
↪set=['a','b'])
>>> W.reflection('a')
(1,2,3)
>>> W.reflection('b')
(1,3,2)
```

reflection_character()

Return the reflection characters of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.reflection_character()
[2, 0, -1]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.reflection_character()
[2, 0, -1]
```

reflection_eigenvalues(*w*, *is_class_representative=False*)

Return the reflection eigenvalue of `w` in `self`.

INPUT:

- `is_class_representative` – boolean (default: `True`) whether to compute instead on the conjugacy class representative

See also

reflection_eigenvalues_family()

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for w in W:
....:     print('%s %s'%(w.reduced_word(), W.reflection_eigenvalues(w)))
[] [0, 0]
[2] [1/2, 0]
[1] [1/2, 0]
[1, 2] [1/3, 2/3]
[2, 1] [1/3, 2/3]
[1, 2, 1] [1/2, 0]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for w in W:
...     print('%s %s'%(w.reduced_word(), W.reflection_eigenvalues(w)))
[] [0, 0]
[2] [1/2, 0]
[1] [1/2, 0]
[1, 2] [1/3, 2/3]
[2, 1] [1/3, 2/3]
[1, 2, 1] [1/2, 0]
```

reflection_eigenvalues_family()

Return the reflection eigenvalues of `self` as a finite family indexed by the class representatives of `self`.

OUTPUT: list with entries k/n representing the eigenvalue ζ_n^k

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.reflection_eigenvalues_family()
Finite family {(): [0, 0], (1,4)(2,3)(5,6): [1/2, 0], (1,6,2)(3,5,4): [1/3, 2/
↪3]}

sage: W = ReflectionGroup((3,1,2))
sage: reflection_eigenvalues = W.reflection_eigenvalues_family()
sage: for elt in sorted(reflection_eigenvalues.keys()):
....:     print('%s %s'%(elt, reflection_eigenvalues[elt]))
() [0, 0]
(1,3,9)(2,4,10)(6,11,17)(8,12,18)(14,19,23)(15,16,20)(21,22,24) [1/3, 0]
(1,3,9)(2,16,24)(4,20,21)(5,7,13)(6,12,23)(8,19,17)(10,15,22)(11,18,14) [1/3,␣
↪1/3]
(1,5)(2,6)(3,7)(4,8)(9,13)(10,14)(11,15)(12,16)(17,21)(18,22)(19,20)(23,24)␣
↪[1/2, 0]
(1,7,3,13,9,5)(2,8,16,19,24,17)(4,14,20,11,21,18)(6,15,12,22,23,10) [1/6, 2/3]
(1,9,3)(2,10,4)(6,17,11)(8,18,12)(14,23,19)(15,20,16)(21,24,22) [2/3, 0]
(1,9,3)(2,20,22)(4,15,24)(5,7,13)(6,18,19)(8,23,11)(10,16,21)(12,14,17) [1/3,␣
↪2/3]
(1,9,3)(2,24,16)(4,21,20)(5,13,7)(6,23,12)(8,17,19)(10,22,15)(11,14,18) [2/3,␣
↪2/3]
(1,13,9,7,3,5)(2,14,24,18,16,11)(4,6,21,23,20,12)(8,22,17,15,19,10) [1/3, 5/6]

sage: W = ReflectionGroup(23)
sage: reflection_eigenvalues = W.reflection_eigenvalues_family()
```

(continues on next page)

(continued from previous page)

```
sage: for elt in sorted(reflection_eigenvalues.keys()):
....:     print('%s %s'%(elt, reflection_eigenvalues[elt]))
() [0, 0, 0]
(1,8,4)(2,21,3)(5,10,11)(6,18,17)(7,9,12)(13,14,15)(16,23,19)(20,25,26)(22,24,
→27)(28,29,30) [1/3, 2/3, 0]
(1,16)(2,5)(4,7)(6,9)(8,10)(11,13)(12,14)(17,20)(19,22)(21,24)(23,25)(26,
→28)(27,29) [1/2, 0, 0]
(1,16)(2,9)(3,18)(4,10)(5,6)(7,8)(11,14)(12,13)(17,24)(19,25)(20,21)(22,
→23)(26,29)(27,28) [1/2, 1/2, 0]
(1,16)(2,17)(3,18)(4,19)(5,20)(6,21)(7,22)(8,23)(9,24)(10,25)(11,26)(12,
→27)(13,28)(14,29)(15,30) [1/2, 1/2, 1/2]
(1,19,20,2,7)(3,6,11,13,9)(4,5,17,22,16)(8,12,15,14,10)(18,21,26,28,24)(23,27,
→30,29,25) [1/5, 4/5, 0]
(1,20,7,19,2)(3,11,9,6,13)(4,17,16,5,22)(8,15,10,12,14)(18,26,24,21,28)(23,30,
→25,27,29) [2/5, 3/5, 0]
(1,23,26,29,22,16,8,11,14,7)(2,10,4,9,18,17,25,19,24,3)(5,21,27,30,28,20,6,12,
→15,13) [1/10, 1/2, 9/10]
(1,24,17,16,9,2)(3,12,13,18,27,28)(4,21,29,19,6,14)(5,25,26,20,10,11)(7,23,30,
→22,8,15) [1/6, 1/2, 5/6]
(1,29,8,7,26,16,14,23,22,11)(2,9,25,3,4,17,24,10,18,19)(5,30,6,13,27,20,15,21,
→28,12) [3/10, 1/2, 7/10]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.reflection_eigenvalues_family()
Finite family {(): [0, 0], (1,4)(2,3)(5,6): [1/2, 0], (1,6,2)(3,5,4): [1/3, 2/
→3]}

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> reflection_eigenvalues = W.reflection_eigenvalues_family()
>>> for elt in sorted(reflection_eigenvalues.keys()):
...     print('%s %s'%(elt, reflection_eigenvalues[elt]))
() [0, 0]
(1,3,9)(2,4,10)(6,11,17)(8,12,18)(14,19,23)(15,16,20)(21,22,24) [1/3, 0]
(1,3,9)(2,16,24)(4,20,21)(5,7,13)(6,12,23)(8,19,17)(10,15,22)(11,18,14) [1/3,␣
→1/3]
(1,5)(2,6)(3,7)(4,8)(9,13)(10,14)(11,15)(12,16)(17,21)(18,22)(19,20)(23,24)␣
→[1/2, 0]
(1,7,3,13,9,5)(2,8,16,19,24,17)(4,14,20,11,21,18)(6,15,12,22,23,10) [1/6, 2/3]
(1,9,3)(2,10,4)(6,17,11)(8,18,12)(14,23,19)(15,20,16)(21,24,22) [2/3, 0]
(1,9,3)(2,20,22)(4,15,24)(5,7,13)(6,18,19)(8,23,11)(10,16,21)(12,14,17) [1/3,␣
→2/3]
(1,9,3)(2,24,16)(4,21,20)(5,13,7)(6,23,12)(8,17,19)(10,22,15)(11,14,18) [2/3,␣
→2/3]
(1,13,9,7,3,5)(2,14,24,18,16,11)(4,6,21,23,20,12)(8,22,17,15,19,10) [1/3, 5/6]

>>> W = ReflectionGroup(Integer(23))
>>> reflection_eigenvalues = W.reflection_eigenvalues_family()
>>> for elt in sorted(reflection_eigenvalues.keys()):
...     print('%s %s'%(elt, reflection_eigenvalues[elt]))
() [0, 0, 0]
(1,8,4)(2,21,3)(5,10,11)(6,18,17)(7,9,12)(13,14,15)(16,23,19)(20,25,26)(22,24,
```

(continues on next page)

(continued from previous page)

```
↪27)(28,29,30) [1/3, 2/3, 0]
(1,16)(2,5)(4,7)(6,9)(8,10)(11,13)(12,14)(17,20)(19,22)(21,24)(23,25)(26,
↪28)(27,29) [1/2, 0, 0]
(1,16)(2,9)(3,18)(4,10)(5,6)(7,8)(11,14)(12,13)(17,24)(19,25)(20,21)(22,
↪23)(26,29)(27,28) [1/2, 1/2, 0]
(1,16)(2,17)(3,18)(4,19)(5,20)(6,21)(7,22)(8,23)(9,24)(10,25)(11,26)(12,
↪27)(13,28)(14,29)(15,30) [1/2, 1/2, 1/2]
(1,19,20,2,7)(3,6,11,13,9)(4,5,17,22,16)(8,12,15,14,10)(18,21,26,28,24)(23,27,
↪30,29,25) [1/5, 4/5, 0]
(1,20,7,19,2)(3,11,9,6,13)(4,17,16,5,22)(8,15,10,12,14)(18,26,24,21,28)(23,30,
↪25,27,29) [2/5, 3/5, 0]
(1,23,26,29,22,16,8,11,14,7)(2,10,4,9,18,17,25,19,24,3)(5,21,27,30,28,20,6,12,
↪15,13) [1/10, 1/2, 9/10]
(1,24,17,16,9,2)(3,12,13,18,27,28)(4,21,29,19,6,14)(5,25,26,20,10,11)(7,23,30,
↪22,8,15) [1/6, 1/2, 5/6]
(1,29,8,7,26,16,14,23,22,11)(2,9,25,3,4,17,24,10,18,19)(5,30,6,13,27,20,15,21,
↪28,12) [3/10, 1/2, 7/10]
```

reflection_hyperplane(*i*, *as_linear_functional=False*, *with_order=False*)

Return the `i`-th reflection hyperplane of `self`.

The `i`-th reflection hyperplane corresponds to the `i` distinguished reflection.

INPUT:

- `i` – an index in the index set
- `as_linear_functionals` – boolean (default: `False`); whether to return the hyperplane or its linear functional in the basis dual to the given root basis

EXAMPLES:

```
sage: W = ReflectionGroup((2,1,2))
sage: W.reflection_hyperplane(3)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(2)))
>>> W.reflection_hyperplane(Integer(3))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
```

One can ask for the result as a linear form:

```
sage: W.reflection_hyperplane(3, True)
(0, 1)
```

```
>>> from sage.all import *
>>> W.reflection_hyperplane(Integer(3), True)
(0, 1)
```

reflection_hyperplanes(*as_linear_functionals=False*, *with_order=False*)

Return the list of all reflection hyperplanes of `self`, either as a codimension 1 space, or as its linear functional.

INPUT:

- `as_linear_functionals` – boolean (default: `False`); whether to return the hyperplane or its linear functional in the basis dual to the given root basis

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for H in W.reflection_hyperplanes(): H
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 2]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[  1 1/2]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1]

sage: for H in W.reflection_hyperplanes(as_linear_functionals=True): H
(1, -1/2)
(1, -2)
(1, 1)


sage: W = ReflectionGroup((2,1,2))
sage: for H in W.reflection_hyperplanes(): H
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 1]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[  1 1/2]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]

sage: for H in W.reflection_hyperplanes(as_linear_functionals=True): H
(1, -1)
(1, -2)
(0, 1)
(1, 0)

sage: for H in W.reflection_hyperplanes(as_linear_functionals=True, with_
↪order=True): H
((1, -1), 2)
((1, -2), 2)
((0, 1), 2)
((1, 0), 2)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for H in W.reflection_hyperplanes(): H
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 2]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[  1 1/2]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1]

>>> for H in W.reflection_hyperplanes(as_linear_functionals=True): H
(1, -1/2)
(1, -2)
(1, 1)


>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(2)))
>>> for H in W.reflection_hyperplanes(): H
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 1]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[  1 1/2]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]

>>> for H in W.reflection_hyperplanes(as_linear_functionals=True): H
(1, -1)
(1, -2)
(0, 1)
(1, 0)

>>> for H in W.reflection_hyperplanes(as_linear_functionals=True, with_
→order=True): H
((1, -1), 2)
((1, -2), 2)
((0, 1), 2)
((1, 0), 2)
```

reflection_index_set()

Return the index set of the reflections of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,4))
```

(continues on next page)

(continued from previous page)

```
sage: W.reflection_index_set()
(1, 2, 3, 4, 5, 6)
sage: W = ReflectionGroup((1,1,4), reflection_index_set=[1,3,'asdf',7,9,11])
sage: W.reflection_index_set()
(1, 3, 'asdf', 7, 9, 11)
sage: W = ReflectionGroup((1,1,4), reflection_index_set=('a','b','c','d','e',
↪'f'))
sage: W.reflection_index_set()
('a', 'b', 'c', 'd', 'e', 'f')
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
>>> W.reflection_index_set()
(1, 2, 3, 4, 5, 6)
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), reflection_index_
↪set=[Integer(1),Integer(3),'asdf',Integer(7),Integer(9),Integer(11)])
>>> W.reflection_index_set()
(1, 3, 'asdf', 7, 9, 11)
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), reflection_index_
↪set=('a','b','c','d','e','f'))
>>> W.reflection_index_set()
('a', 'b', 'c', 'd', 'e', 'f')
```

reflections()

Return a finite family containing the reflections of `self`, indexed by `self.reflection_index_set()`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.reflections()
Finite family {1: (1,4)(2,3)(5,6), 2: (1,3)(2,5)(4,6), 3: (1,5)(2,4)(3,6)}

sage: W = ReflectionGroup((1,1,3),reflection_index_set=['a','b','c'])
sage: W.reflections()
Finite family {'a': (1,4)(2,3)(5,6), 'b': (1,3)(2,5)(4,6), 'c': (1,5)(2,4)(3,
↪6)}

sage: W = ReflectionGroup((3,1,1))
sage: W.reflections()
Finite family {1: (1,2,3), 2: (1,3,2)}

sage: W = ReflectionGroup((1,1,3),(3,1,2))
sage: W.reflections()
Finite family {1: (1,6)(2,5)(7,8), 2: (1,5)(2,7)(6,8),
               3: (3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,
↪26)(27,28,30),
               4: (3,11)(4,12)(9,13)(10,14)(15,19)(16,20)(17,21)(18,22)(23,
↪27)(24,28)(25,26)(29,30),
               5: (1,7)(2,6)(5,8),
               6: (3,19)(4,25)(9,11)(10,17)(12,28)(13,15)(14,30)(16,18)(20,
↪27)(21,29)(22,23)(24,26),
               7: (4,21,27)(10,22,28)(11,13,19)(12,14,20)(16,26,30)(17,18,
```

(continues on next page)

(continued from previous page)

```
↪25)(23,24,29),
                 8: (3,13)(4,24)(9,19)(10,29)(11,15)(12,26)(14,21)(16,23)(17,
↪30)(18,27)(20,22)(25,28),
                 9: (3,15,9)(4,16,10)(12,23,17)(14,24,18)(20,29,25)(21,26,
↪22)(27,30,28),
                 10: (4,27,21)(10,28,22)(11,19,13)(12,20,14)(16,30,26)(17,25,
↪18)(23,29,24)}
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.reflections()
Finite family {1: (1,4)(2,3)(5,6), 2: (1,3)(2,5)(4,6), 3: (1,5)(2,4)(3,6)}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)),reflection_index_
↪set=['a','b','c'])
>>> W.reflections()
Finite family {'a': (1,4)(2,3)(5,6), 'b': (1,3)(2,5)(4,6), 'c': (1,5)(2,4)(3,
↪6)}

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(1)))
>>> W.reflections()
Finite family {1: (1,2,3), 2: (1,3,2)}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)),(Integer(3),
↪Integer(1),Integer(2)))
>>> W.reflections()
Finite family {1: (1,6)(2,5)(7,8), 2: (1,5)(2,7)(6,8),
                 3: (3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,
↪26)(27,28,30),
                 4: (3,11)(4,12)(9,13)(10,14)(15,19)(16,20)(17,21)(18,22)(23,
↪27)(24,28)(25,26)(29,30),
                 5: (1,7)(2,6)(5,8),
                 6: (3,19)(4,25)(9,11)(10,17)(12,28)(13,15)(14,30)(16,18)(20,
↪27)(21,29)(22,23)(24,26),
                 7: (4,21,27)(10,22,28)(11,13,19)(12,14,20)(16,26,30)(17,18,
↪25)(23,24,29),
                 8: (3,13)(4,24)(9,19)(10,29)(11,15)(12,26)(14,21)(16,23)(17,
↪30)(18,27)(20,22)(25,28),
                 9: (3,15,9)(4,16,10)(12,23,17)(14,24,18)(20,29,25)(21,26,
↪22)(27,30,28),
                 10: (4,27,21)(10,28,22)(11,19,13)(12,20,14)(16,30,26)(17,25,
↪18)(23,29,24)}
```

roots()

Return all roots corresponding to all reflections of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.roots()
[(1, 0), (0, 1), (1, 1), (-1, 0), (0, -1), (-1, -1)]
```

(continues on next page)

(continued from previous page)

```
sage: W = ReflectionGroup((3,1,2))
sage: W.roots()
[(1, 0), (-1, 1), (E(3), 0), (-E(3), 1), (0, 1), (1, -1),
 (0, E(3)), (1, -E(3)), (E(3)^2, 0), (-E(3)^2, 1),
 (E(3), -1), (E(3), -E(3)), (0, E(3)^2), (1, -E(3)^2),
 (-1, E(3)), (-E(3), E(3)), (E(3)^2, -1), (E(3)^2, -E(3)),
 (E(3), -E(3)^2), (-E(3)^2, E(3)), (-1, E(3)^2),
 (-E(3), E(3)^2), (E(3)^2, -E(3)^2), (-E(3)^2, E(3)^2)]

sage: W = ReflectionGroup((4,2,2))
sage: W.roots()
[(1, 0), (-E(4), 1), (-1, 1), (-1, 0), (E(4), 1), (1, 1),
 (0, -E(4)), (E(4), -1), (E(4), E(4)), (0, E(4)),
 (E(4), -E(4)), (0, 1), (1, -E(4)), (1, -1), (0, -1),
 (1, E(4)), (-E(4), 0), (-1, E(4)), (E(4), 0), (-E(4), E(4)),
 (-E(4), -1), (-E(4), -E(4)), (-1, -E(4)), (-1, -1)]

sage: W = ReflectionGroup((1,1,4), (3,1,2))
sage: W.roots()
[(1, 0, 0, 0, 0), (0, 1, 0, 0, 0), (0, 0, 1, 0, 0),
 (0, 0, 0, 1, 0), (0, 0, 0, -1, 1), (1, 1, 0, 0, 0),
 (0, 1, 1, 0, 0), (1, 1, 1, 0, 0), (-1, 0, 0, 0, 0),
 (0, -1, 0, 0, 0), (0, 0, -1, 0, 0), (-1, -1, 0, 0, 0),
 (0, -1, -1, 0, 0), (-1, -1, -1, 0, 0), (0, 0, 0, E(3), 0),
 (0, 0, 0, -E(3), 1), (0, 0, 0, 0, 1), (0, 0, 0, 1, -1),
 (0, 0, 0, 0, E(3)), (0, 0, 0, 1, -E(3)), (0, 0, 0, E(3)^2, 0),
 (0, 0, 0, -E(3)^2, 1), (0, 0, 0, E(3), -1), (0, 0, 0, E(3), -E(3)),
 (0, 0, 0, 0, E(3)^2), (0, 0, 0, 1, -E(3)^2), (0, 0, 0, -1, E(3)),
 (0, 0, 0, -E(3), E(3)), (0, 0, 0, E(3)^2, -1),
 (0, 0, 0, E(3)^2, -E(3)), (0, 0, 0, E(3), -E(3)^2),
 (0, 0, 0, -E(3)^2, E(3)), (0, 0, 0, -1, E(3)^2),
 (0, 0, 0, -E(3), E(3)^2), (0, 0, 0, E(3)^2, -E(3)^2),
 (0, 0, 0, -E(3)^2, E(3)^2)]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.roots()
[(1, 0), (0, 1), (1, 1), (-1, 0), (0, -1), (-1, -1)]

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> W.roots()
[(1, 0), (-1, 1), (E(3), 0), (-E(3), 1), (0, 1), (1, -1),
 (0, E(3)), (1, -E(3)), (E(3)^2, 0), (-E(3)^2, 1),
 (E(3), -1), (E(3), -E(3)), (0, E(3)^2), (1, -E(3)^2),
 (-1, E(3)), (-E(3), E(3)), (E(3)^2, -1), (E(3)^2, -E(3)),
 (E(3), -E(3)^2), (-E(3)^2, E(3)), (-1, E(3)^2),
 (-E(3), E(3)^2), (E(3)^2, -E(3)^2), (-E(3)^2, E(3)^2)]

>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(2)))
>>> W.roots()
[(1, 0), (-E(4), 1), (-1, 1), (-1, 0), (E(4), 1), (1, 1),
 (0, -E(4)), (E(4), -1), (E(4), E(4)), (0, E(4)),
```

(continues on next page)

(continued from previous page)

```
 (E(4), -E(4)), (0, 1), (1, -E(4)), (1, -1), (0, -1),
 (1, E(4)), (-E(4), 0), (-1, E(4)), (E(4), 0), (-E(4), E(4)),
 (-E(4), -1), (-E(4), -E(4)), (-1, -E(4)), (-1, -1)]

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(3),
↪Integer(1),Integer(2)))
>>> W.roots()
[(1, 0, 0, 0, 0), (0, 1, 0, 0, 0), (0, 0, 1, 0, 0),
 (0, 0, 0, 1, 0), (0, 0, 0, -1, 1), (1, 1, 0, 0, 0),
 (0, 1, 1, 0, 0), (1, 1, 1, 0, 0), (-1, 0, 0, 0, 0),
 (0, -1, 0, 0, 0), (0, 0, -1, 0, 0), (-1, -1, 0, 0, 0),
 (0, -1, -1, 0, 0), (-1, -1, -1, 0, 0), (0, 0, 0, E(3), 0),
 (0, 0, 0, -E(3), 1), (0, 0, 0, 0, 1), (0, 0, 0, 1, -1),
 (0, 0, 0, 0, E(3)), (0, 0, 0, 1, -E(3)), (0, 0, 0, E(3)^2, 0),
 (0, 0, 0, -E(3)^2, 1), (0, 0, 0, E(3), -1), (0, 0, 0, E(3), -E(3)),
 (0, 0, 0, 0, E(3)^2), (0, 0, 0, 1, -E(3)^2), (0, 0, 0, -1, E(3)),
 (0, 0, 0, -E(3), E(3)), (0, 0, 0, E(3)^2, -1),
 (0, 0, 0, E(3)^2, -E(3)), (0, 0, 0, E(3), -E(3)^2),
 (0, 0, 0, -E(3)^2, E(3)), (0, 0, 0, -1, E(3)^2),
 (0, 0, 0, -E(3), E(3)^2), (0, 0, 0, E(3)^2, -E(3)^2),
 (0, 0, 0, -E(3)^2, E(3)^2)]
```

series()

Return the series of the classification type to which `self` belongs.

For real reflection groups, these are the Cartan-Killing classification types "A","B","C","D","E","F","G","H","I", and for complx non-real reflection groups these are the Shephard-Todd classification type "ST".

EXAMPLES:

```
sage: ReflectionGroup((1,1,3)).series()
['A']
sage: ReflectionGroup((3,1,3)).series()
['ST']
```

```
>>> from sage.all import *
>>> ReflectionGroup((Integer(1),Integer(1),Integer(3))).series()
['A']
>>> ReflectionGroup((Integer(3),Integer(1),Integer(3))).series()
['ST']
```

set_reflection_representation(*refl_repr=None*)

Set the reflection representation of `self`.

INPUT:

- `refl_repr` – dictionary representing the matrices of the generators of `self` with keys given by the index set, or `None` to reset to the default reflection representation

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for w in W: w.to_matrix(); print("-----")
```

(continues on next page)

(continued from previous page)

```
[1 0]
[0 1]
-----
[ 1  1]
[ 0 -1]
-----
[-1  0]
[ 1  1]
-----
[-1 -1]
[ 1  0]
-----
[ 0  1]
[-1 -1]
-----
[ 0 -1]
[-1  0]
-----

sage: W.set_reflection_representation({1: matrix([[0,1,0],[1,0,0],[0,0,1]]),␣
↪2: matrix([[1,0,0],[0,0,1],[0,1,0]])})
sage: for w in W: w.to_matrix(); print("-----")
[1 0 0]
[0 1 0]
[0 0 1]
-----
[1 0 0]
[0 0 1]
[0 1 0]
-----
[0 1 0]
[1 0 0]
[0 0 1]
-----
[0 0 1]
[1 0 0]
[0 1 0]
-----
[0 1 0]
[0 0 1]
[1 0 0]
-----
[0 0 1]
[0 1 0]
[1 0 0]
-----
sage: W.set_reflection_representation()
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for w in W: w.to_matrix(); print("-----")
[1 0]
```

(continues on next page)

(continued from previous page)

```
[0 1]
-----
[ 1  1]
[ 0 -1]
-----
[-1  0]
[ 1  1]
-----
[-1 -1]
[ 1  0]
-----
[ 0  1]
[-1 -1]
-----
[ 0 -1]
[-1  0]
-----

>>> W.set_reflection_representation({Integer(1): matrix([[Integer(0),
↪Integer(1),Integer(0)],[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)]]), Integer(2): matrix([[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(1),
↪Integer(0)]])})
>>> for w in W: w.to_matrix(); print("-----")
[1 0 0]
[0 1 0]
[0 0 1]
-----
[1 0 0]
[0 0 1]
[0 1 0]
-----
[0 1 0]
[1 0 0]
[0 0 1]
-----
[0 0 1]
[1 0 0]
[0 1 0]
-----
[0 1 0]
[0 0 1]
[1 0 0]
-----
[0 0 1]
[0 1 0]
[1 0 0]
-----
>>> W.set_reflection_representation()
```

simple_coroot(*i*)

Return the simple root with index `i`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3])
sage: W.simple_coroot(1)
(2, -1, 0)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)])
>>> W.simple_coroot(Integer(1))
(2, -1, 0)
```

simple_coroots()

Return the simple coroots of `self`.

These are the coroots corresponding to the simple reflections.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.simple_coroots()
Finite family {1: (2, -1), 2: (-1, 2)}

sage: W = ReflectionGroup((1,1,4), (2,1,2))
sage: W.simple_coroots()
Finite family {1: (2, -1, 0, 0, 0), 2: (-1, 2, -1, 0, 0), 3: (0, -1, 2, 0, 0),
↪ 4: (0, 0, 0, 2, -2), 5: (0, 0, 0, -1, 2)}

sage: W = ReflectionGroup((3,1,2))
sage: W.simple_coroots()
Finite family {1: (-2*E(3) - E(3)^2, 0), 2: (-1, 1)}

sage: W = ReflectionGroup((1,1,4), (3,1,2))
sage: W.simple_coroots()
Finite family {1: (2, -1, 0, 0, 0), 2: (-1, 2, -1, 0, 0), 3: (0, -1, 2, 0, 0),
↪ 4: (0, 0, 0, -2*E(3) - E(3)^2, 0), 5: (0, 0, 0, -1, 1)}
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.simple_coroots()
Finite family {1: (2, -1), 2: (-1, 2)}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(2),
↪Integer(1),Integer(2)))
>>> W.simple_coroots()
Finite family {1: (2, -1, 0, 0, 0), 2: (-1, 2, -1, 0, 0), 3: (0, -1, 2, 0, 0),
↪ 4: (0, 0, 0, 2, -2), 5: (0, 0, 0, -1, 2)}

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> W.simple_coroots()
Finite family {1: (-2*E(3) - E(3)^2, 0), 2: (-1, 1)}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(3),
↪Integer(1),Integer(2)))
>>> W.simple_coroots()
```

(continues on next page)

(continued from previous page)

```
Finite family {1: (2, -1, 0, 0, 0), 2: (-1, 2, -1, 0, 0), 3: (0, -1, 2, 0, 0),
↪ 4: (0, 0, 0, -2*E(3) - E(3)^2, 0), 5: (0, 0, 0, -1, 1)}
```

simple_reflection(*i*)

Return the `i`-th simple reflection of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.simple_reflection(1)
(1,4)(2,3)(5,6)
sage: W.simple_reflections()
Finite family {1: (1,4)(2,3)(5,6), 2: (1,3)(2,5)(4,6)}
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.simple_reflection(Integer(1))
(1,4)(2,3)(5,6)
>>> W.simple_reflections()
Finite family {1: (1,4)(2,3)(5,6), 2: (1,3)(2,5)(4,6)}
```

simple_root(*i*)

Return the simple root with index `i`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3])
sage: W.simple_root(1)
(1, 0, 0)
sage: W.simple_root(2)
(0, 1, 0)
sage: W.simple_root(3)
(0, 0, 1)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)])
>>> W.simple_root(Integer(1))
(1, 0, 0)
>>> W.simple_root(Integer(2))
(0, 1, 0)
>>> W.simple_root(Integer(3))
(0, 0, 1)
```

simple_roots()

Return the simple roots of `self`.

These are the roots corresponding to the simple reflections.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: W.simple_roots()
Finite family {1: (1, 0), 2: (0, 1)}
```

(continues on next page)

(continued from previous page)

```
sage: W = ReflectionGroup((1,1,4), (2,1,2))
sage: W.simple_roots()
Finite family {1: (1, 0, 0, 0, 0), 2: (0, 1, 0, 0, 0), 3: (0, 0, 1, 0, 0), 4:␣
↪(0, 0, 0, 1, 0), 5: (0, 0, 0, 0, 1)}

sage: W = ReflectionGroup((3,1,2))
sage: W.simple_roots()
Finite family {1: (1, 0), 2: (-1, 1)}

sage: W = ReflectionGroup((1,1,4), (3,1,2))
sage: W.simple_roots()
Finite family {1: (1, 0, 0, 0, 0), 2: (0, 1, 0, 0, 0), 3: (0, 0, 1, 0, 0), 4:␣
↪(0, 0, 0, 1, 0), 5: (0, 0, 0, -1, 1)}
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> W.simple_roots()
Finite family {1: (1, 0), 2: (0, 1)}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(2),
↪Integer(1),Integer(2)))
>>> W.simple_roots()
Finite family {1: (1, 0, 0, 0, 0), 2: (0, 1, 0, 0, 0), 3: (0, 0, 1, 0, 0), 4:␣
↪(0, 0, 0, 1, 0), 5: (0, 0, 0, 0, 1)}

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> W.simple_roots()
Finite family {1: (1, 0), 2: (-1, 1)}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), (Integer(3),
↪Integer(1),Integer(2)))
>>> W.simple_roots()
Finite family {1: (1, 0, 0, 0, 0), 2: (0, 1, 0, 0, 0), 3: (0, 0, 1, 0, 0), 4:␣
↪(0, 0, 0, 1, 0), 5: (0, 0, 0, -1, 1)}
```

class sage.combinat.root_system.reflection_group_complex.**IrreducibleComplexReflectionGroup**(*W_types*, *index_set=None*, *hyperplane_index_set=None*, *reflection_index_set=None*)

Bases: `ComplexReflectionGroup`

class Element

Bases: `Element`

is_coxeter_element(*which_primitive=1*, *is_class_representative=False*)

Return `True` if `self` is a Coxeter element.

This is, whether `self` has an eigenvalue that is a primitive h-th root of unity.

INPUT:

- `which_primitive` – (default: `1`) for which power of the first primitive `h`-th root of unity to look as a reflection eigenvalue for a regular element
- `is_class_representative` – boolean (default: `True`); whether to compute instead on the conjugacy class representative

See also

`coxeter_element()` `coxeter_elements()`

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for w in W:
....:     print('%s %s'%(w.reduced_word(), w.is_coxeter_element()))
[] False
[2] False
[1] False
[1, 2] True
[2, 1] True
[1, 2, 1] False
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for w in W:
...     print('%s %s'%(w.reduced_word(), w.is_coxeter_element()))
[] False
[2] False
[1] False
[1, 2] True
[2, 1] True
[1, 2, 1] False
```

is_h_regular(*is_class_representative=False*)

Return whether `self` is regular.

This is if `self` has an eigenvector with eigenvalue h and which does not lie in any reflection hyperplane. Here, h denotes the Coxeter number.

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: for w in W:
....:     print('%s %s'%(w.reduced_word(), w.is_h_regular()))
[] False
[2] False
[1] False
[1, 2] True
[2, 1] True
[1, 2, 1] False
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> for w in W:
...     print('%s %s'%(w.reduced_word(), w.is_h_regular()))
[] False
[2] False
[1] False
[1, 2] True
[2, 1] True
[1, 2, 1] False
```

is_regular(*h*, *is_class_representative=False*)

Return whether `self` is regular.

This is, if `self` has an eigenvector with eigenvalue of order `h` and which does not lie in any reflection hyperplane.

INPUT:

- `h` – the order of the eigenvalue
- `is_class_representative` – boolean (default: `True`); whether to compute instead on the conjugacy class representative

EXAMPLES:

```
sage: W = ReflectionGroup((1,1,3))
sage: h = W.coxeter_number()
sage: for w in W:
....:     print("{} {}".format(w.reduced_word(), w.is_regular(h)))
[] False
[2] False
[1] False
[1, 2] True
[2, 1] True
[1, 2, 1] False

sage: W = ReflectionGroup(23); h = W.coxeter_number()
sage: for w in W:
....:     if w.is_regular(h):
....:         w.reduced_word()
[1, 2, 3]
[2, 1, 3]
[1, 3, 2]
[3, 2, 1]
[2, 1, 2, 3, 2]
[2, 3, 2, 1, 2]
[1, 2, 1, 2, 3, 2, 1]
[1, 2, 3, 2, 1, 2, 1]
[1, 2, 1, 2, 3, 2, 1, 2, 3]
[2, 1, 2, 1, 3, 2, 1, 2, 3]
[2, 1, 2, 3, 2, 1, 2, 1, 3]
[1, 2, 3, 2, 1, 2, 1, 3, 2]
[3, 2, 1, 2, 1, 3, 2, 1, 2]
[1, 2, 1, 2, 1, 3, 2, 1, 2]
[2, 3, 2, 1, 2, 1, 3, 2, 1]
[2, 1, 2, 1, 3, 2, 1, 2, 1]
```

(continues on next page)

(continued from previous page)

```
[2, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]
[1, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]
[1, 2, 1, 2, 1, 3, 2, 1, 2, 1, 3]
[1, 2, 1, 2, 3, 2, 1, 2, 1, 3, 2]
[1, 2, 3, 2, 1, 2, 1, 3, 2, 1, 2]
[2, 1, 2, 3, 2, 1, 2, 1, 3, 2, 1]
[2, 1, 2, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]
[1, 2, 1, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))
>>> h = W.coxeter_number()
>>> for w in W:
...     print("{} {}".format(w.reduced_word(), w.is_regular(h)))
[] False
[2] False
[1] False
[1, 2] True
[2, 1] True
[1, 2, 1] False

>>> W = ReflectionGroup(Integer(23)); h = W.coxeter_number()
>>> for w in W:
...     if w.is_regular(h):
...         w.reduced_word()
[1, 2, 3]
[2, 1, 3]
[1, 3, 2]
[3, 2, 1]
[2, 1, 2, 3, 2]
[2, 3, 2, 1, 2]
[1, 2, 1, 2, 3, 2, 1]
[1, 2, 3, 2, 1, 2, 1]
[1, 2, 1, 2, 3, 2, 1, 2, 3]
[2, 1, 2, 1, 3, 2, 1, 2, 3]
[2, 1, 2, 3, 2, 1, 2, 1, 3]
[1, 2, 3, 2, 1, 2, 1, 3, 2]
[3, 2, 1, 2, 1, 3, 2, 1, 2]
[1, 2, 1, 2, 1, 3, 2, 1, 2]
[2, 3, 2, 1, 2, 1, 3, 2, 1]
[2, 1, 2, 1, 3, 2, 1, 2, 1]
[2, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]
[1, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]
[1, 2, 1, 2, 1, 3, 2, 1, 2, 1, 3]
[1, 2, 1, 2, 3, 2, 1, 2, 1, 3, 2]
[1, 2, 3, 2, 1, 2, 1, 3, 2, 1, 2]
[2, 1, 2, 3, 2, 1, 2, 1, 3, 2, 1]
[2, 1, 2, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]
[1, 2, 1, 3, 2, 1, 2, 1, 3, 2, 1, 2, 3]
```

Check that Issue #25478 is fixed:

```
sage: W = ReflectionGroup(["A",5])
sage: w = W.from_reduced_word([1,2,3,5])
sage: w.is_regular(4)
False
sage: W = ReflectionGroup(["A",3])
sage: len([w for w in W if w.is_regular(w.order())])
18
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(["A",Integer(5)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(3),Integer(5)])
>>> w.is_regular(Integer(4))
False
>>> W = ReflectionGroup(["A",Integer(3)])
>>> len([w for w in W if w.is_regular(w.order())])
18
```

sage.combinat.root_system.reflection_group_complex.**multi_partitions**(*n*, *S*, *i=None*)

Return all vectors as lists of the same length as S whose standard inner product with S equals n.

EXAMPLES:

```
sage: from sage.combinat.root_system.reflection_group_complex import multi_
↪partitions
sage: multi_partitions(10, [2,3,3,4])
[[5, 0, 0, 0],
 [3, 0, 0, 1],
 [2, 2, 0, 0],
 [2, 1, 1, 0],
 [2, 0, 2, 0],
 [1, 0, 0, 2],
 [0, 2, 0, 1],
 [0, 1, 1, 1],
 [0, 0, 2, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.reflection_group_complex import multi_
↪partitions
>>> multi_partitions(Integer(10), [Integer(2),Integer(3),Integer(3),Integer(4)])
[[5, 0, 0, 0],
 [3, 0, 0, 1],
 [2, 2, 0, 0],
 [2, 1, 1, 0],
 [2, 0, 2, 0],
 [1, 0, 0, 2],
 [0, 2, 0, 1],
 [0, 1, 1, 1],
 [0, 0, 2, 1]]
```

sage.combinat.root_system.reflection_group_complex.**power**(*k*)

Return f^k and caching all intermediate results.

Speeds the computation if one has to compute f^k's for many values of k.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: f = -2*x^2 + 2*x*y - 2*y^2 + 2*y*z - 2*z^2
sage: all( f^k == power(f,k) for k in range(20) )
True
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = P._first_ngens(3)
>>> f = -Integer(2)*x**Integer(2) + Integer(2)*x*y - Integer(2)*y**Integer(2) +␣
↪Integer(2)*y*z - Integer(2)*z**Integer(2)
>>> all( f**k == power(f,k) for k in range(Integer(20)) )
True
```

5.1.237 Finite real reflection groups

Let V be a finite-dimensional real vector space. A reflection of V is an operator $r \in \mathrm{GL}(V)$ that has order 2 and fixes pointwise a hyperplane in V. In the present implementation, finite real reflection groups are tied with a root system.

Finite real reflection groups with root systems have been classified according to finite Cartan-Killing types. For more definitions and classification types of finite complex reflection groups, see Wikipedia article Complex_reflection_group.

The point of entry to work with reflection groups is `ReflectionGroup()` which can be used with finite Cartan-Killing types:

```
sage: ReflectionGroup(['A',2])
Irreducible real reflection group of rank 2 and type A2
sage: ReflectionGroup(['F',4])
Irreducible real reflection group of rank 4 and type F4
sage: ReflectionGroup(['H',3])
Irreducible real reflection group of rank 3 and type H3
```

```
>>> from sage.all import *
>>> ReflectionGroup(['A',Integer(2)])
Irreducible real reflection group of rank 2 and type A2
>>> ReflectionGroup(['F',Integer(4)])
Irreducible real reflection group of rank 4 and type F4
>>> ReflectionGroup(['H',Integer(3)])
Irreducible real reflection group of rank 3 and type H3
```

AUTHORS:

- Christian Stump (initial version 2011–2015)

Warning

Uses the GAP3 package *Chevie* which is available as an experimental package (installed by `sage -i gap3`) or to download by hand from Jean Michel's website.

class sage.combinat.root_system.reflection_group_real.**IrreducibleRealReflectionGroup**(*W_types, index_set=None, hyperplane_index_set=None, reflection_index_set=None*)

Bases: *RealReflectionGroup*, *IrreducibleComplexReflectionGroup*

class Element

Bases: *Element*, *Element*

class sage.combinat.root_system.reflection_group_real.**RealReflectionGroup**(*W_types, index_set=None, hyperplane_index_set=None, reflection_index_set=None*)

Bases: *ComplexReflectionGroup*

A real reflection group given as a permutation group.

See also

ReflectionGroup()

class Element

Bases: RealReflectionGroupElement, *Element*

left_coset_representatives()

Return the left coset representatives of self.

See also

right_coset_representatives()

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: for w in W:
....:     lcr = w.left_coset_representatives()
....:     print("%s %s"%(w.reduced_word(),
....:                    [v.reduced_word() for v in lcr]))
[] [[], [2], [1], [1, 2], [2, 1], [1, 2, 1]]
[2] [[], [2], [1]]
[1] [[], [1], [2, 1]]
[1, 2] [[]]
[2, 1] [[]]
[1, 2, 1] [[], [2], [1, 2]]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> for w in W:
...     lcr = w.left_coset_representatives()
...     print("%s %s"%(w.reduced_word(),
...                    [v.reduced_word() for v in lcr]))
[] [[], [2], [1], [1, 2], [2, 1], [1, 2, 1]]
[2] [[], [2], [1]]
[1] [[], [1], [2, 1]]
[1, 2] [[]]
[2, 1] [[]]
[1, 2, 1] [[], [2], [1, 2]]
```

right_coset_representatives()

Return the right coset representatives of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: for w in W:
....:     rcr = w.right_coset_representatives()
....:     print("%s %s"%(w.reduced_word(),
....:                    [v.reduced_word() for v in rcr]))
[] [[], [2], [1], [2, 1], [1, 2], [1, 2, 1]]
[2] [[], [2], [1]]
[1] [[], [1], [1, 2]]
[1, 2] [[]]
[2, 1] [[]]
[1, 2, 1] [[], [2], [2, 1]]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> for w in W:
...     rcr = w.right_coset_representatives()
...     print("%s %s"%(w.reduced_word(),
...                    [v.reduced_word() for v in rcr]))
[] [[], [2], [1], [2, 1], [1, 2], [1, 2, 1]]
[2] [[], [2], [1]]
[1] [[], [1], [1, 2]]
[1, 2] [[]]
[2, 1] [[]]
[1, 2, 1] [[], [2], [2, 1]]
```

almost_positive_roots()

Return the almost positive roots of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3], ['B',2])
sage: W.almost_positive_roots()
[(-1, 0, 0, 0, 0),
 (0, -1, 0, 0, 0),
 (0, 0, -1, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 0, 0, -1, 0),
 (0, 0, 0, 0, -1),
 (1, 0, 0, 0, 0),
 (0, 1, 0, 0, 0),
 (0, 0, 1, 0, 0),
 (0, 0, 0, 1, 0),
 (0, 0, 0, 0, 1),
 (1, 1, 0, 0, 0),
 (0, 1, 1, 0, 0),
 (0, 0, 0, 1, 1),
 (1, 1, 1, 0, 0),
 (0, 0, 0, 2, 1)]

sage: W = ReflectionGroup(['A',3])
sage: W.almost_positive_roots()
[(-1, 0, 0),
 (0, -1, 0),
 (0, 0, -1),
 (1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
 (1, 1, 0),
 (0, 1, 1),
 (1, 1, 1)]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)], ['B',Integer(2)])
>>> W.almost_positive_roots()
[(-1, 0, 0, 0, 0),
 (0, -1, 0, 0, 0),
 (0, 0, -1, 0, 0),
 (0, 0, 0, -1, 0),
 (0, 0, 0, 0, -1),
 (1, 0, 0, 0, 0),
 (0, 1, 0, 0, 0),
 (0, 0, 1, 0, 0),
 (0, 0, 0, 1, 0),
 (0, 0, 0, 0, 1),
 (1, 1, 0, 0, 0),
 (0, 1, 1, 0, 0),
 (0, 0, 0, 1, 1),
 (1, 1, 1, 0, 0),
 (0, 0, 0, 2, 1)]

>>> W = ReflectionGroup(['A',Integer(3)])
>>> W.almost_positive_roots()
[(-1, 0, 0),
 (0, -1, 0),
 (0, 0, -1),
 (1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
```

(continues on next page)

(continued from previous page)

```
 (1, 1, 0),
 (0, 1, 1),
 (1, 1, 1)]
```

bipartite_index_set()

Return the bipartite index set of a real reflection group.

EXAMPLES:

```
sage: W = ReflectionGroup(["A",5])
sage: W.bipartite_index_set()
[[1, 3, 5], [2, 4]]

sage: W = ReflectionGroup(["A",5],index_set=['a','b','c','d','e'])
sage: W.bipartite_index_set()
[['a', 'c', 'e'], ['b', 'd']]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(["A",Integer(5)])
>>> W.bipartite_index_set()
[[1, 3, 5], [2, 4]]

>>> W = ReflectionGroup(["A",Integer(5)],index_set=['a','b','c','d','e'])
>>> W.bipartite_index_set()
[['a', 'c', 'e'], ['b', 'd']]
```

bruhat_cone(*x*, *y*, *side='upper'*, *backend='cdd'*)

Return the (upper or lower) Bruhat cone associated to the interval `[x,y]`.

To a cover relation $v \prec w$ in strong Bruhat order you can assign a positive root β given by the unique reflection s_β such that $s_\beta v = w$.

The upper Bruhat cone of the interval $[x, y]$ is the non-empty, polyhedral cone generated by the roots corresponding to $x \prec a$ for all atoms a in the interval. The lower Bruhat cone of the interval $[x, y]$ is the non-empty, polyhedral cone generated by the roots corresponding to $c \prec y$ for all coatoms c in the interval.

INPUT:

- `x` – an element in the group W
- `y` – an element in the group W
- `side` – (default: `'upper'`) must be one of the following:
  - `'upper'` – return the upper Bruhat cone of the interval [`x`, `y`]
  - `'lower'` – return the lower Bruhat cone of the interval [`x`, `y`]
- `backend` – string (default: `'cdd'`); the backend to use to create the polyhedron

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: x = W.from_reduced_word([1])
sage: y = W.w0
sage: W.bruhat_cone(x, y)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and
```

(continues on next page)

(continued from previous page)

```
↪2 rays

sage: W = ReflectionGroup(['E',6])
sage: x = W.one()
sage: y = W.w0
sage: W.bruhat_cone(x, y, side='lower')
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex and␣
↪6 rays
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> x = W.from_reduced_word([Integer(1)])
>>> y = W.w0
>>> W.bruhat_cone(x, y)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and␣
↪2 rays

>>> W = ReflectionGroup(['E',Integer(6)])
>>> x = W.one()
>>> y = W.w0
>>> W.bruhat_cone(x, y, side='lower')
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex and␣
↪6 rays
```

REFERENCES:

- [Dy1994]
- [JS2021]

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3])
sage: W.cartan_type()
['A', 3]

sage: W = ReflectionGroup(['A',3], ['B',3])
sage: W.cartan_type()
A3xB3 relabelled by {1: 3, 2: 2, 3: 1}
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)])
>>> W.cartan_type()
['A', 3]

>>> W = ReflectionGroup(['A',Integer(3)], ['B',Integer(3)])
>>> W.cartan_type()
A3xB3 relabelled by {1: 3, 2: 2, 3: 1}
```

coxeter_diagram()

Return the Coxeter diagram associated to `self`.

EXAMPLES:

```
sage: G = ReflectionGroup(['B',3])
sage: G.coxeter_diagram().edges(labels=True, sort=True)
[(1, 2, 4), (2, 3, 3)]
```

```
>>> from sage.all import *
>>> G = ReflectionGroup(['B',Integer(3)])
>>> G.coxeter_diagram().edges(labels=True, sort=True)
[(1, 2, 4), (2, 3, 3)]
```

coxeter_matrix()

Return the Coxeter matrix associated to `self`.

EXAMPLES:

```
sage: G = ReflectionGroup(['A',3])
sage: G.coxeter_matrix()
[1 3 2]
[3 1 3]
[2 3 1]
```

```
>>> from sage.all import *
>>> G = ReflectionGroup(['A',Integer(3)])
>>> G.coxeter_matrix()
[1 3 2]
[3 1 3]
[2 3 1]
```

fundamental_weight(*i*)

Return the fundamental weight with index `i`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3])
sage: [ W.fundamental_weight(i) for i in W.index_set() ]
[(3/4, 1/2, 1/4), (1/2, 1, 1/2), (1/4, 1/2, 3/4)]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)])
>>> [ W.fundamental_weight(i) for i in W.index_set() ]
[(3/4, 1/2, 1/4), (1/2, 1, 1/2), (1/4, 1/2, 3/4)]
```

fundamental_weights()

Return the fundamental weights of `self` in terms of the simple roots.

The fundamental weights are defined by $s_j(\omega_i) = \omega_i - \delta_{i=j}\alpha_j$ for the simple reflection s_j with corresponding simple roots α_j.

In other words, the transpose Cartan matrix sends the weight basis to the root basis. Observe again that the action here is defined as a right action, see the example below.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3], ['B',2])
sage: W.fundamental_weights()
Finite family {1: (3/4, 1/2, 1/4, 0, 0), 2: (1/2, 1, 1/2, 0, 0), 3: (1/4, 1/2,
↪ 3/4, 0, 0), 4: (0, 0, 0, 1, 1/2), 5: (0, 0, 0, 1, 1)}

sage: W = ReflectionGroup(['A',3])
sage: W.fundamental_weights()
Finite family {1: (3/4, 1/2, 1/4), 2: (1/2, 1, 1/2), 3: (1/4, 1/2, 3/4)}

sage: W = ReflectionGroup(['A',3])
sage: S = W.simple_reflections()
sage: N = W.fundamental_weights()
sage: for i in W.index_set():
....:     for j in W.index_set():
....:         print("{} {} {} {}".format(i, j, N[i], N[i]*S[j].to_matrix()))
1 1 (3/4, 1/2, 1/4) (-1/4, 1/2, 1/4)
1 2 (3/4, 1/2, 1/4) (3/4, 1/2, 1/4)
1 3 (3/4, 1/2, 1/4) (3/4, 1/2, 1/4)
2 1 (1/2, 1, 1/2) (1/2, 1, 1/2)
2 2 (1/2, 1, 1/2) (1/2, 0, 1/2)
2 3 (1/2, 1, 1/2) (1/2, 1, 1/2)
3 1 (1/4, 1/2, 3/4) (1/4, 1/2, 3/4)
3 2 (1/4, 1/2, 3/4) (1/4, 1/2, 3/4)
3 3 (1/4, 1/2, 3/4) (1/4, 1/2, -1/4)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)], ['B',Integer(2)])
>>> W.fundamental_weights()
Finite family {1: (3/4, 1/2, 1/4, 0, 0), 2: (1/2, 1, 1/2, 0, 0), 3: (1/4, 1/2,
↪ 3/4, 0, 0), 4: (0, 0, 0, 1, 1/2), 5: (0, 0, 0, 1, 1)}

>>> W = ReflectionGroup(['A',Integer(3)])
>>> W.fundamental_weights()
Finite family {1: (3/4, 1/2, 1/4), 2: (1/2, 1, 1/2), 3: (1/4, 1/2, 3/4)}

>>> W = ReflectionGroup(['A',Integer(3)])
>>> S = W.simple_reflections()
>>> N = W.fundamental_weights()
>>> for i in W.index_set():
...     for j in W.index_set():
...         print("{} {} {} {}".format(i, j, N[i], N[i]*S[j].to_matrix()))
1 1 (3/4, 1/2, 1/4) (-1/4, 1/2, 1/4)
1 2 (3/4, 1/2, 1/4) (3/4, 1/2, 1/4)
1 3 (3/4, 1/2, 1/4) (3/4, 1/2, 1/4)
2 1 (1/2, 1, 1/2) (1/2, 1, 1/2)
2 2 (1/2, 1, 1/2) (1/2, 0, 1/2)
2 3 (1/2, 1, 1/2) (1/2, 1, 1/2)
3 1 (1/4, 1/2, 3/4) (1/4, 1/2, 3/4)
3 2 (1/4, 1/2, 3/4) (1/4, 1/2, 3/4)
3 3 (1/4, 1/2, 3/4) (1/4, 1/2, -1/4)
```

iteration(*algorithm='breadth'*, *tracking_words=True*)

Return an iterator going through all elements in `self`.

INPUT:

- `algorithm` – (default: `'breadth'`) must be one of the following:
  - `'breadth'` – iterate over in a linear extension of the weak order
  - `'depth'` – iterate by a depth-first-search
  - `'parabolic'` – iterate by using parabolic subgroups
- `tracking_words` – boolean (default: `True`); whether or not to keep track of the reduced words and store them in `_reduced_word`

Note

The fastest iteration is the parabolic iteration and the depth first algorithm without tracking words is second. In particular, `'depth'` is ~1.5x faster than `'breadth'`.

Note

The `'parabolic'` iteration does not track words and requires keeping the subgroup corresponding to $I \setminus \{i\}$ in memory (for each i individually).

EXAMPLES:

```
sage: W = ReflectionGroup(["B",2])

sage: for w in W.iteration("breadth",True):
....:     print("%s %s"%(w, w._reduced_word))
() []
(1,3)(2,6)(5,7) [1]
(1,5)(2,4)(6,8) [0]
(1,7,5,3)(2,4,6,8) [0, 1]
(1,3,5,7)(2,8,6,4) [1, 0]
(2,8)(3,7)(4,6) [1, 0, 1]
(1,7)(3,5)(4,8) [0, 1, 0]
(1,5)(2,6)(3,7)(4,8) [0, 1, 0, 1]

sage: for w in W.iteration("depth", False): w
()
(1,3)(2,6)(5,7)
(1,5)(2,4)(6,8)
(1,3,5,7)(2,8,6,4)
(1,7)(3,5)(4,8)
(1,7,5,3)(2,4,6,8)
(2,8)(3,7)(4,6)
(1,5)(2,6)(3,7)(4,8)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(["B",Integer(2)])

>>> for w in W.iteration("breadth",True):
...     print("%s %s"%(w, w._reduced_word))
```

(continues on next page)

(continued from previous page)

```
() []
(1,3)(2,6)(5,7) [1]
(1,5)(2,4)(6,8) [0]
(1,7,5,3)(2,4,6,8) [0, 1]
(1,3,5,7)(2,8,6,4) [1, 0]
(2,8)(3,7)(4,6) [1, 0, 1]
(1,7)(3,5)(4,8) [0, 1, 0]
(1,5)(2,6)(3,7)(4,8) [0, 1, 0, 1]

>>> for w in W.iteration("depth", False): w
()
(1,3)(2,6)(5,7)
(1,5)(2,4)(6,8)
(1,3,5,7)(2,8,6,4)
(1,7)(3,5)(4,8)
(1,7,5,3)(2,4,6,8)
(2,8)(3,7)(4,6)
(1,5)(2,6)(3,7)(4,8)
```

positive_roots()

Return the positive roots of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3], ['B',2])
sage: W.positive_roots()
[(1, 0, 0, 0, 0),
 (0, 1, 0, 0, 0),
 (0, 0, 1, 0, 0),
 (0, 0, 0, 1, 0),
 (0, 0, 0, 0, 1),
 (1, 1, 0, 0, 0),
 (0, 1, 1, 0, 0),
 (0, 0, 0, 1, 1),
 (1, 1, 1, 0, 0),
 (0, 0, 0, 2, 1)]

sage: W = ReflectionGroup(['A',3])
sage: W.positive_roots()
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 0), (0, 1, 1), (1, 1, 1)]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)], ['B',Integer(2)])
>>> W.positive_roots()
[(1, 0, 0, 0, 0),
 (0, 1, 0, 0, 0),
 (0, 0, 1, 0, 0),
 (0, 0, 0, 1, 0),
 (0, 0, 0, 0, 1),
 (1, 1, 0, 0, 0),
 (0, 1, 1, 0, 0),
 (0, 0, 0, 1, 1),
```

(continues on next page)

(continued from previous page)

```
 (1, 1, 1, 0, 0),
 (0, 0, 0, 2, 1)]

>>> W = ReflectionGroup(['A',Integer(3)])
>>> W.positive_roots()
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 0), (0, 1, 1), (1, 1, 1)]
```

reflection_to_positive_root(*r*)

Return the positive root orthogonal to the given reflection.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: for r in W.reflections():
....:     print(W.reflection_to_positive_root(r))
(1, 0)
(0, 1)
(1, 1)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> for r in W.reflections():
...     print(W.reflection_to_positive_root(r))
(1, 0)
(0, 1)
(1, 1)
```

right_coset_representatives(*J*)

Return the right coset representatives of `self` for the parabolic subgroup generated by the simple reflections in `J`.

EXAMPLES:

```
sage: W = ReflectionGroup(["A",3])
sage: for J in Subsets([1,2,3]): W.right_coset_representatives(J)
[(), (2,5)(3,9)(4,6)(8,11)(10,12), (1,4)(2,8)(3,5)(7,10)(9,11),
 (1,7)(2,4)(5,6)(8,10)(11,12), (1,2,10)(3,6,5)(4,7,8)(9,12,11),
 (1,4,6)(2,3,11)(5,8,9)(7,10,12), (1,6,4)(2,11,3)(5,9,8)(7,12,10),
 (1,7)(2,6)(3,9)(4,5)(8,12)(10,11),
 (1,10,2)(3,5,6)(4,8,7)(9,11,12), (1,2,3,12)(4,5,10,11)(6,7,8,9),
 (1,5,9,10)(2,12,8,6)(3,4,7,11), (1,6)(2,9)(3,8)(5,11)(7,12),
 (1,8)(2,7)(3,6)(4,10)(9,12), (1,10,9,5)(2,6,8,12)(3,11,7,4),
 (1,12,3,2)(4,11,10,5)(6,9,8,7), (1,3)(2,12)(4,10)(5,11)(6,8)(7,9),
 (1,5,12)(2,9,4)(3,10,8)(6,7,11), (1,8,11)(2,5,7)(3,12,4)(6,10,9),
 (1,11,8)(2,7,5)(3,4,12)(6,9,10), (1,12,5)(2,4,9)(3,8,10)(6,11,7),
 (1,3,7,9)(2,11,6,10)(4,8,5,12), (1,9,7,3)(2,10,6,11)(4,12,5,8),
 (1,11)(3,10)(4,9)(5,7)(6,12), (1,9)(2,8)(3,7)(4,11)(5,10)(6,12)]
[(), (2,5)(3,9)(4,6)(8,11)(10,12), (1,4)(2,8)(3,5)(7,10)(9,11),
 (1,2,10)(3,6,5)(4,7,8)(9,12,11), (1,4,6)(2,3,11)(5,8,9)(7,10,12),
 (1,6,4)(2,11,3)(5,9,8)(7,12,10), (1,2,3,12)(4,5,10,11)(6,7,8,9),
 (1,5,9,10)(2,12,8,6)(3,4,7,11), (1,6)(2,9)(3,8)(5,11)(7,12),
 (1,3)(2,12)(4,10)(5,11)(6,8)(7,9),
 (1,5,12)(2,9,4)(3,10,8)(6,7,11), (1,3,7,9)(2,11,6,10)(4,8,5,12)]
```

(continues on next page)

(continued from previous page)

```
[(), (2,5)(3,9)(4,6)(8,11)(10,12), (1,7)(2,4)(5,6)(8,10)(11,12),
 (1,4,6)(2,3,11)(5,8,9)(7,10,12),
 (1,7)(2,6)(3,9)(4,5)(8,12)(10,11),
 (1,10,2)(3,5,6)(4,8,7)(9,11,12), (1,2,3,12)(4,5,10,11)(6,7,8,9),
 (1,10,9,5)(2,6,8,12)(3,11,7,4), (1,12,3,2)(4,11,10,5)(6,9,8,7),
 (1,8,11)(2,5,7)(3,12,4)(6,10,9), (1,12,5)(2,4,9)(3,8,10)(6,11,7),
 (1,11)(3,10)(4,9)(5,7)(6,12)]
[(), (1,4)(2,8)(3,5)(7,10)(9,11), (1,7)(2,4)(5,6)(8,10)(11,12),
 (1,2,10)(3,6,5)(4,7,8)(9,12,11), (1,6,4)(2,11,3)(5,9,8)(7,12,10),
 (1,10,2)(3,5,6)(4,8,7)(9,11,12), (1,5,9,10)(2,12,8,6)(3,4,7,11),
 (1,8)(2,7)(3,6)(4,10)(9,12), (1,12,3,2)(4,11,10,5)(6,9,8,7),
 (1,3)(2,12)(4,10)(5,11)(6,8)(7,9),
 (1,11,8)(2,7,5)(3,4,12)(6,9,10), (1,9,7,3)(2,10,6,11)(4,12,5,8)]
[(), (2,5)(3,9)(4,6)(8,11)(10,12), (1,4,6)(2,3,11)(5,8,9)(7,10,12),
 (1,2,3,12)(4,5,10,11)(6,7,8,9)]
[(), (1,4)(2,8)(3,5)(7,10)(9,11), (1,2,10)(3,6,5)(4,7,8)(9,12,11),
 (1,6,4)(2,11,3)(5,9,8)(7,12,10), (1,5,9,10)(2,12,8,6)(3,4,7,11),
 (1,3)(2,12)(4,10)(5,11)(6,8)(7,9)]
[(), (1,7)(2,4)(5,6)(8,10)(11,12), (1,10,2)(3,5,6)(4,8,7)(9,11,12),
 (1,12,3,2)(4,11,10,5)(6,9,8,7)]
[()]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(["A",Integer(3)])
>>> for J in Subsets([Integer(1),Integer(2),Integer(3)]): W.right_coset_
↪representatives(J)
[(), (2,5)(3,9)(4,6)(8,11)(10,12), (1,4)(2,8)(3,5)(7,10)(9,11),
 (1,7)(2,4)(5,6)(8,10)(11,12), (1,2,10)(3,6,5)(4,7,8)(9,12,11),
 (1,4,6)(2,3,11)(5,8,9)(7,10,12), (1,6,4)(2,11,3)(5,9,8)(7,12,10),
 (1,7)(2,6)(3,9)(4,5)(8,12)(10,11),
 (1,10,2)(3,5,6)(4,8,7)(9,11,12), (1,2,3,12)(4,5,10,11)(6,7,8,9),
 (1,5,9,10)(2,12,8,6)(3,4,7,11), (1,6)(2,9)(3,8)(5,11)(7,12),
 (1,8)(2,7)(3,6)(4,10)(9,12), (1,10,9,5)(2,6,8,12)(3,11,7,4),
 (1,12,3,2)(4,11,10,5)(6,9,8,7), (1,3)(2,12)(4,10)(5,11)(6,8)(7,9),
 (1,5,12)(2,9,4)(3,10,8)(6,7,11), (1,8,11)(2,5,7)(3,12,4)(6,10,9),
 (1,11,8)(2,7,5)(3,4,12)(6,9,10), (1,12,5)(2,4,9)(3,8,10)(6,11,7),
 (1,3,7,9)(2,11,6,10)(4,8,5,12), (1,9,7,3)(2,10,6,11)(4,12,5,8),
 (1,11)(3,10)(4,9)(5,7)(6,12), (1,9)(2,8)(3,7)(4,11)(5,10)(6,12)]
[(), (2,5)(3,9)(4,6)(8,11)(10,12), (1,4)(2,8)(3,5)(7,10)(9,11),
 (1,2,10)(3,6,5)(4,7,8)(9,12,11), (1,4,6)(2,3,11)(5,8,9)(7,10,12),
 (1,6,4)(2,11,3)(5,9,8)(7,12,10), (1,2,3,12)(4,5,10,11)(6,7,8,9),
 (1,5,9,10)(2,12,8,6)(3,4,7,11), (1,6)(2,9)(3,8)(5,11)(7,12),
 (1,3)(2,12)(4,10)(5,11)(6,8)(7,9),
 (1,5,12)(2,9,4)(3,10,8)(6,7,11), (1,3,7,9)(2,11,6,10)(4,8,5,12)]
[(), (2,5)(3,9)(4,6)(8,11)(10,12), (1,7)(2,4)(5,6)(8,10)(11,12),
 (1,4,6)(2,3,11)(5,8,9)(7,10,12),
 (1,7)(2,6)(3,9)(4,5)(8,12)(10,11),
 (1,10,2)(3,5,6)(4,8,7)(9,11,12), (1,2,3,12)(4,5,10,11)(6,7,8,9),
 (1,10,9,5)(2,6,8,12)(3,11,7,4), (1,12,3,2)(4,11,10,5)(6,9,8,7),
 (1,8,11)(2,5,7)(3,12,4)(6,10,9), (1,12,5)(2,4,9)(3,8,10)(6,11,7),
 (1,11)(3,10)(4,9)(5,7)(6,12)]
[(), (1,4)(2,8)(3,5)(7,10)(9,11), (1,7)(2,4)(5,6)(8,10)(11,12),
```

(continues on next page)

(continued from previous page)

```
 (1,2,10)(3,6,5)(4,7,8)(9,12,11), (1,6,4)(2,11,3)(5,9,8)(7,12,10),
 (1,10,2)(3,5,6)(4,8,7)(9,11,12), (1,5,9,10)(2,12,8,6)(3,4,7,11),
 (1,8)(2,7)(3,6)(4,10)(9,12), (1,12,3,2)(4,11,10,5)(6,9,8,7),
 (1,3)(2,12)(4,10)(5,11)(6,8)(7,9),
 (1,11,8)(2,7,5)(3,4,12)(6,9,10), (1,9,7,3)(2,10,6,11)(4,12,5,8)]
[(), (2,5)(3,9)(4,6)(8,11)(10,12), (1,4,6)(2,3,11)(5,8,9)(7,10,12),
 (1,2,3,12)(4,5,10,11)(6,7,8,9)]
[(), (1,4)(2,8)(3,5)(7,10)(9,11), (1,2,10)(3,6,5)(4,7,8)(9,12,11),
 (1,6,4)(2,11,3)(5,9,8)(7,12,10), (1,5,9,10)(2,12,8,6)(3,4,7,11),
 (1,3)(2,12)(4,10)(5,11)(6,8)(7,9)]
[(), (1,7)(2,4)(5,6)(8,10)(11,12), (1,10,2)(3,5,6)(4,8,7)(9,11,12),
 (1,12,3,2)(4,11,10,5)(6,9,8,7)]
[()]
```

root_to_reflection(*root*)

Return the reflection along the given `root`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])
sage: for beta in W.roots(): W.root_to_reflection(beta)
(1,4)(2,3)(5,6)
(1,3)(2,5)(4,6)
(1,5)(2,4)(3,6)
(1,4)(2,3)(5,6)
(1,3)(2,5)(4,6)
(1,5)(2,4)(3,6)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])
>>> for beta in W.roots(): W.root_to_reflection(beta)
(1,4)(2,3)(5,6)
(1,3)(2,5)(4,6)
(1,5)(2,4)(3,6)
(1,4)(2,3)(5,6)
(1,3)(2,5)(4,6)
(1,5)(2,4)(3,6)
```

simple_root_index(*i*)

Return the index of the simple root α_i.

This is the position of α_i in the list of simple roots.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3])
sage: [W.simple_root_index(i) for i in W.index_set()]
[0, 1, 2]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)])
>>> [W.simple_root_index(i) for i in W.index_set()]
[0, 1, 2]
```

sage.combinat.root_system.reflection_group_real.**ReflectionGroup**(**args*, ***kwds*)

Construct a finite (complex or real) reflection group as a Sage permutation group by fetching the permutation representation of the generators from chevie's database.

INPUT:

Can be one or multiple of the following:

- a triple (r, p, n) with p divides r, which denotes the group $G(r, p, n)$
- an integer between 4 and 37, which denotes an exceptional irreducible complex reflection group
- a finite Cartan-Killing type

EXAMPLES:

Finite reflection groups can be constructed from

Cartan-Killing classification types:

```
sage: W = ReflectionGroup(['A',3]); W
 Irreducible real reflection group of rank 3 and type A3

sage: W = ReflectionGroup(['H',4]); W
 Irreducible real reflection group of rank 4 and type H4

sage: W = ReflectionGroup(['I',5]); W
 Irreducible real reflection group of rank 2 and type I2(5)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)]); W
 Irreducible real reflection group of rank 3 and type A3

>>> W = ReflectionGroup(['H',Integer(4)]); W
 Irreducible real reflection group of rank 4 and type H4

>>> W = ReflectionGroup(['I',Integer(5)]); W
 Irreducible real reflection group of rank 2 and type I2(5)
```

the complex infinite family $G(r, p, n)$ with p divides r:

```
sage: W = ReflectionGroup((1,1,4)); W
Irreducible real reflection group of rank 3 and type A3

sage: W = ReflectionGroup((2,1,3)); W
Irreducible real reflection group of rank 3 and type B3
```

```
>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4))); W
Irreducible real reflection group of rank 3 and type A3

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(3))); W
Irreducible real reflection group of rank 3 and type B3
```

Chevalley-Shepard-Todd exceptional classification types:

```
sage: W = ReflectionGroup(23); W
Irreducible real reflection group of rank 3 and type H3
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(Integer(23)); W
Irreducible real reflection group of rank 3 and type H3
```

Cartan types and matrices:

```
sage: ReflectionGroup(CartanType(['A',2]))
Irreducible real reflection group of rank 2 and type A2

sage: ReflectionGroup(CartanType((['A',2],['A',2])))
Reducible real reflection group of rank 4 and type A2 x A2

sage: C = CartanMatrix(['A',2])
sage: ReflectionGroup(C)
Irreducible real reflection group of rank 2 and type A2
```

```
>>> from sage.all import *
>>> ReflectionGroup(CartanType(['A',Integer(2)]))
Irreducible real reflection group of rank 2 and type A2

>>> ReflectionGroup(CartanType((['A',Integer(2)],['A',Integer(2)])))
Reducible real reflection group of rank 4 and type A2 x A2

>>> C = CartanMatrix(['A',Integer(2)])
>>> ReflectionGroup(C)
Irreducible real reflection group of rank 2 and type A2
```

multiples of the above:

```
sage: W = ReflectionGroup(['A',2],['B',2]); W
Reducible real reflection group of rank 4 and type A2 x B2

sage: W = ReflectionGroup(['A',2],4); W
Reducible complex reflection group of rank 4 and type A2 x ST4

sage: W = ReflectionGroup((4,2,2),4); W
Reducible complex reflection group of rank 4 and type G(4,2,2) x ST4
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)],['B',Integer(2)]); W
Reducible real reflection group of rank 4 and type A2 x B2

>>> W = ReflectionGroup(['A',Integer(2)],Integer(4)); W
Reducible complex reflection group of rank 4 and type A2 x ST4

>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(2)),Integer(4)); W
Reducible complex reflection group of rank 4 and type G(4,2,2) x ST4
```

sage.combinat.root_system.reflection_group_real.**is_chevie_available**()

Test whether the GAP3 Chevie package is available.

EXAMPLES:

```
sage: # needs sage.groups
sage: from sage.combinat.root_system.reflection_group_real import is_chevie_
↪available
sage: is_chevie_available() # random
False
sage: is_chevie_available() in [True, False]
True
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> from sage.combinat.root_system.reflection_group_real import is_chevie_
↪available
>>> is_chevie_available() # random
False
>>> is_chevie_available() in [True, False]
True
```

5.1.238 Group algebras of root lattice realizations

class sage.combinat.root_system.root_lattice_realization_algebras.**Algebras**(*category*, **args*)

Bases: AlgebrasCategory

The category of group algebras of root lattice realizations.

This includes typically weight rings (group algebras of weight lattices).

class ElementMethods

Bases: object

acted_upon(*w*)

Implement the action of w on self.

INPUT:

- w – an element of the Weyl group acting on the underlying weight lattice realization

EXAMPLES:

```
sage: L = RootSystem(["A",3]).ambient_space()
sage: W = L.weyl_group()                                                    #␣
↪needs sage.libs.gap
sage: M = L.algebra(QQ['q','t'])
sage: m = M.an_element(); m  # TODO: investigate why we don't get␣
↪something more interesting
B[(2, 2, 3, 0)]
sage: m = (m+1)^2; m
B[(0, 0, 0, 0)] + 2*B[(2, 2, 3, 0)] + B[(4, 4, 6, 0)]
sage: w = W.an_element(); w.reduced_word()                                  #␣
↪needs sage.libs.gap
[1, 2, 3]
sage: m.acted_upon(w)                                                       #␣
↪needs sage.libs.gap
B[(0, 0, 0, 0)] + 2*B[(0, 2, 2, 3)] + B[(0, 4, 4, 6)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3)]).ambient_space()
>>> W = L.weyl_group()                                                  #␣
↪needs sage.libs.gap
>>> M = L.algebra(QQ['q','t'])
>>> m = M.an_element(); m  # TODO: investigate why we don't get something␣
↪more interesting
B[(2, 2, 3, 0)]
>>> m = (m+Integer(1))**Integer(2); m
B[(0, 0, 0, 0)] + 2*B[(2, 2, 3, 0)] + B[(4, 4, 6, 0)]
>>> w = W.an_element(); w.reduced_word()                                #␣
↪needs sage.libs.gap
[1, 2, 3]
>>> m.acted_upon(w)                                                     #␣
↪needs sage.libs.gap
B[(0, 0, 0, 0)] + 2*B[(0, 2, 2, 3)] + B[(0, 4, 4, 6)]
```

expand(*alphabet*)

Expand `self` into variables in the `alphabet`.

INPUT:

- `alphabet` – a non empty list/tuple of (invertible) variables in a ring to expand in

EXAMPLES:

```
sage: L = RootSystem(["A",2]).ambient_lattice()
sage: KL = L.algebra(QQ)
sage: p = KL.an_element() + KL.sum_of_monomials(L.some_elements()); p
B[(1, 0, 0)] + B[(1, -1, 0)] + B[(1, 1, 0)] + 2*B[(2, 2, 3)] + B[(0, 1, -
↪1)]
sage: F = LaurentPolynomialRing(QQ, 'x,y,z')
sage: p.expand(F.gens())
2*x^2*y^2*z^3 + x*y + x + y*z^-1 + x*y^-1
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).ambient_lattice()
>>> KL = L.algebra(QQ)
>>> p = KL.an_element() + KL.sum_of_monomials(L.some_elements()); p
B[(1, 0, 0)] + B[(1, -1, 0)] + B[(1, 1, 0)] + 2*B[(2, 2, 3)] + B[(0, 1, -
↪1)]
>>> F = LaurentPolynomialRing(QQ, 'x,y,z')
>>> p.expand(F.gens())
2*x^2*y^2*z^3 + x*y + x + y*z^-1 + x*y^-1
```

class ParentMethods

Bases: `object`

T0_check_on_basis(*q1*, *q2*, *convention='antidominant'*)

Return the $T_0^\vee$ operator acting on the basis.

This implements the formula for $T_{0'}$ in Section 6.12 of [Haiman06].

REFERENCES:

> **Warning**
>
> The current implementation probably returns just nonsense, if the convention is not "dominant".

EXAMPLES:

```
sage: K = QQ['q1,q2'].fraction_field()
sage: q1,q2 = K.gens()

sage: L = RootSystem(["A",1,1]).ambient_space()
sage: L0 = L.classical()
sage: KL = L.algebra(K)
sage: some_weights = L.fundamental_weights()                           #␣
↪needs sage.graphs
sage: f = KL.T0_check_on_basis(q1,q2, convention='dominant')           #␣
↪needs sage.graphs
sage: f(L0.zero())                                                     #␣
↪needs sage.graphs
(q1+q2)*B[(0, 0)] + q1*B[(1, -1)]

sage: L = RootSystem(["A",3,1]).ambient_space()
sage: L0 = L.classical()
sage: KL = L.algebra(K)
sage: some_weights = L0.fundamental_weights()
sage: f = KL.T0_check_on_basis(q1,q2, convention='dominant')           #␣
↪needs sage.graphs
sage: f(L0.zero())        # not checked                                #␣
↪needs sage.graphs
(q1+q2)*B[(0, 0, 0, 0)] + q1^3/q2^2*B[(1, 0, 0, -1)]
```

```
>>> from sage.all import *
>>> K = QQ['q1,q2'].fraction_field()
>>> q1,q2 = K.gens()

>>> L = RootSystem(["A",Integer(1),Integer(1)]).ambient_space()
>>> L0 = L.classical()
>>> KL = L.algebra(K)
>>> some_weights = L.fundamental_weights()                             #␣
↪needs sage.graphs
>>> f = KL.T0_check_on_basis(q1,q2, convention='dominant')             #␣
↪needs sage.graphs
>>> f(L0.zero())                                                       #␣
↪needs sage.graphs
(q1+q2)*B[(0, 0)] + q1*B[(1, -1)]

>>> L = RootSystem(["A",Integer(3),Integer(1)]).ambient_space()
>>> L0 = L.classical()
>>> KL = L.algebra(K)
>>> some_weights = L0.fundamental_weights()
>>> f = KL.T0_check_on_basis(q1,q2, convention='dominant')             #␣
↪needs sage.graphs
>>> f(L0.zero())        # not checked                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
(q1+q2)*B[(0, 0, 0, 0)] + q1^3/q2^2*B[(1, 0, 0, -1)]
```

The following results have not been checked:

```
sage: for x in some_weights:                                                    #␣
↪needs sage.graphs
....:     print("{} : {}".format(x, f(x)))
(1, 0, 0, 0) : q1*B[(1, 0, 0, 0)]
(1, 1, 0, 0) : q1*B[(1, 1, 0, 0)]
(1, 1, 1, 0) : q1*B[(1, 1, 1, 0)]
```

```
>>> from sage.all import *
>>> for x in some_weights:                                                    #␣
↪needs sage.graphs
...     print("{} : {}".format(x, f(x)))
(1, 0, 0, 0) : q1*B[(1, 0, 0, 0)]
(1, 1, 0, 0) : q1*B[(1, 1, 0, 0)]
(1, 1, 1, 0) : q1*B[(1, 1, 1, 0)]
```

Some examples for type $B_2^{(1)}$ dual:

```
sage: L = RootSystem("B2~*").ambient_space()
sage: L0 = L.classical()
sage: e = L.basis()
sage: K = QQ['q,u'].fraction_field()
sage: q,u = K.gens()
sage: q1 = u
sage: q2 = -1/u
sage: KL = L.algebra(K)
sage: KL0 = KL.classical()
sage: f = KL.T0_check_on_basis(q1,q2, convention='dominant')                   #␣
↪needs sage.graphs
sage: T = KL.twisted_demazure_lusztig_operators(q1,q2, convention=
↪'dominant')
```

```
>>> from sage.all import *
>>> L = RootSystem("B2~*").ambient_space()
>>> L0 = L.classical()
>>> e = L.basis()
>>> K = QQ['q,u'].fraction_field()
>>> q,u = K.gens()
>>> q1 = u
>>> q2 = -Integer(1)/u
>>> KL = L.algebra(K)
>>> KL0 = KL.classical()
>>> f = KL.T0_check_on_basis(q1,q2, convention='dominant')                   #␣
↪needs sage.graphs
>>> T = KL.twisted_demazure_lusztig_operators(q1,q2, convention='dominant
↪')
```

Direct calculation:

```
sage: T.Tw(0)(KL0.monomial(L0([0,0])))                                          #␣
↪needs sage.graphs
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
sage: KL.T0_check_on_basis(q1,q2, convention='dominant')(L0([0,0]))            #␣
↪needs sage.graphs
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
```

```
>>> from sage.all import *
>>> T.Tw(Integer(0))(KL0.monomial(L0([Integer(0),Integer(0)])))                ␣
↪                        # needs sage.graphs
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
>>> KL.T0_check_on_basis(q1,q2, convention='dominant')(L0([Integer(0),
↪Integer(0)]))      # needs sage.graphs
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
```

Step by step calculation, comparing by hand with Mark Shimozono:

```
sage: res = T.Tw(2)(KL0.monomial(L0([0,0]))); res
u*B[(0, 0)]
sage: res = res * KL0.monomial(L0([-1,1])); res
u*B[(-1, 1)]
sage: res = T.Tw_inverse(1)(res); res
(u^2-1)*B[(0, 0)] + u^2*B[(1, -1)]
sage: res = T.Tw_inverse(2)(res); res
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
```

```
>>> from sage.all import *
>>> res = T.Tw(Integer(2))(KL0.monomial(L0([Integer(0),Integer(0)]))); res
u*B[(0, 0)]
>>> res = res * KL0.monomial(L0([-Integer(1),Integer(1)])); res
u*B[(-1, 1)]
>>> res = T.Tw_inverse(Integer(1))(res); res
(u^2-1)*B[(0, 0)] + u^2*B[(1, -1)]
>>> res = T.Tw_inverse(Integer(2))(res); res
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
```

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: A = RootSystem(["A",2,1]).ambient_space().algebra(QQ)
sage: A.cartan_type()
['A', 2, 1]
sage: A = RootSystem(["B",2]).weight_space().algebra(QQ)
sage: A.cartan_type()
['B', 2]
```

```
>>> from sage.all import *
>>> A = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().
↪algebra(QQ)
>>> A.cartan_type()
```

(continues on next page)

(continued from previous page)

```
['A', 2, 1]
>>> A = RootSystem(["B",Integer(2)]).weight_space().algebra(QQ)
>>> A.cartan_type()
['B', 2]
```

classical()

Return the group algebra of the corresponding classical lattice.

EXAMPLES:

```
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(QQ)
sage: KL.classical()
Algebra of the Ambient space of the Root system of type ['A', 2] over
↪Rational Field
```

```
>>> from sage.all import *
>>> KL = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().
↪algebra(QQ)
>>> KL.classical()
Algebra of the Ambient space of the Root system of type ['A', 2] over
↪Rational Field
```

demazure_lusztig_operator_on_basis(*weight*, *i*, *q1*, *q2*, *convention='antidominant'*)

Return the result of applying the i-th Demazure-Lusztig operator on `weight`.

INPUT:

- `weight` – an element λ of the weight lattice
- `i` – an element of the index set
- `q1`, `q2` – two elements of the ground ring
- `convention` – `'antidominant'`, `'bar'`, or `'dominant'` (default: `'antidominant'`)

See *demazure_lusztig_operators()* for the details.

EXAMPLES:

```
sage: L = RootSystem(["A",1]).ambient_space()
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: KL.demazure_lusztig_operator_on_basis(L((2,2)), 1, q1, q2)
q1*B[(2, 2)]
sage: KL.demazure_lusztig_operator_on_basis(L((3,0)), 1, q1, q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
sage: KL.demazure_lusztig_operator_on_basis(L((0,3)), 1, q1, q2)
-(q1+q2)*B[(1, 2)] - (q1+q2)*B[(2, 1)] - q2*B[(3, 0)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1)]).ambient_space()
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KL = L.algebra(K)
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(2),Integer(2))),
↪Integer(1), q1, q2)
q1*B[(2, 2)]
```

(continues on next page)

(continued from previous page)

```
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(3),Integer(0))),␣
↪Integer(1), q1, q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(0),Integer(3))),␣
↪Integer(1), q1, q2)
-(q1+q2)*B[(1, 2)] - (q1+q2)*B[(2, 1)] - q2*B[(3, 0)]
```

At $q_1 = 1$ and $q_2 = 0$ we recover the action of the isobaric divided differences π_i:

```
sage: KL.demazure_lusztig_operator_on_basis(L((2,2)), 1, 1, 0)
B[(2, 2)]
sage: KL.demazure_lusztig_operator_on_basis(L((3,0)), 1, 1, 0)
B[(1, 2)] + B[(2, 1)] + B[(3, 0)] + B[(0, 3)]
sage: KL.demazure_lusztig_operator_on_basis(L((0,3)), 1, 1, 0)
-B[(1, 2)] - B[(2, 1)]
```

```
>>> from sage.all import *
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(2),Integer(2))),␣
↪Integer(1), Integer(1), Integer(0))
B[(2, 2)]
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(3),Integer(0))),␣
↪Integer(1), Integer(1), Integer(0))
B[(1, 2)] + B[(2, 1)] + B[(3, 0)] + B[(0, 3)]
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(0),Integer(3))),␣
↪Integer(1), Integer(1), Integer(0))
-B[(1, 2)] - B[(2, 1)]
```

Or $1 - \pi_i$ for `bar=True`:

```
sage: KL.demazure_lusztig_operator_on_basis(L((2,2)), 1, 1, 0, convention=
↪'bar')
0
sage: KL.demazure_lusztig_operator_on_basis(L((3,0)), 1, 1, 0, convention=
↪'bar')
-B[(1, 2)] - B[(2, 1)] - B[(0, 3)]
sage: KL.demazure_lusztig_operator_on_basis(L((0,3)), 1, 1, 0, convention=
↪'bar')
B[(1, 2)] + B[(2, 1)] + B[(0, 3)]
```

```
>>> from sage.all import *
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(2),Integer(2))),␣
↪Integer(1), Integer(1), Integer(0), convention='bar')
0
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(3),Integer(0))),␣
↪Integer(1), Integer(1), Integer(0), convention='bar')
-B[(1, 2)] - B[(2, 1)] - B[(0, 3)]
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(0),Integer(3))),␣
↪Integer(1), Integer(1), Integer(0), convention='bar')
B[(1, 2)] + B[(2, 1)] + B[(0, 3)]
```

At $q_1 = 1$ and $q_2 = -1$ we recover the action of the simple reflection s_i:

```
sage: KL.demazure_lusztig_operator_on_basis(L((2,2)), 1, 1, -1)
B[(2, 2)]
sage: KL.demazure_lusztig_operator_on_basis(L((3,0)), 1, 1, -1)
B[(0, 3)]
sage: KL.demazure_lusztig_operator_on_basis(L((0,3)), 1, 1, -1)
B[(3, 0)]
```

```
>>> from sage.all import *
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(2),Integer(2))),␣
↪Integer(1), Integer(1), -Integer(1))
B[(2, 2)]
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(3),Integer(0))),␣
↪Integer(1), Integer(1), -Integer(1))
B[(0, 3)]
>>> KL.demazure_lusztig_operator_on_basis(L((Integer(0),Integer(3))),␣
↪Integer(1), Integer(1), -Integer(1))
B[(3, 0)]
```

demazure_lusztig_operator_on_classical_on_basis(*weight*, *i*, *q*, *q1*, *q2*, *convention='antidominant'*)

Return the result of applying the i-th Demazure-Lusztig operator on the classical weight `weight` embedded at level 0.

INPUT:

- `weight` – a classical weight λ
- `i` – an element of the index set
- `q1`, `q2` – two elements of the ground ring
- `convention` – `'antidominant'`, `'bar'`, or `'dominant'` (default: `'antidominant'`)

See *demazure_lusztig_operators()* for the details.

Todo

- Optimize the code to only do the embedding/projection for T_0
- Add an option to specify at which level one wants to work. Currently this is level 0.

EXAMPLES:

```
sage: L = RootSystem(["A",1,1]).ambient_space()
sage: L0 = L.classical()
sage: K = QQ['q,q1,q2']
sage: q, q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: KL0 = L0.algebra(K)
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1),Integer(1)]).ambient_space()
>>> L0 = L.classical()
>>> K = QQ['q,q1,q2']
>>> q, q1, q2 = K.gens()
>>> KL = L.algebra(K)
>>> KL0 = L0.algebra(K)
```

These operators coincide with the usual Demazure-Lusztig operators:

```
sage: KL.demazure_lusztig_operator_on_classical_on_basis(L0((2,2)),      #␣
↪needs sage.graphs
....:                                                    1, q, q1, q2)
q1*B[(2, 2)]
sage: KL0.demazure_lusztig_operator_on_basis(L0((2,2)), 1, q1, q2)
q1*B[(2, 2)]

sage: KL.demazure_lusztig_operator_on_classical_on_basis(L0((3,0)),      #␣
↪needs sage.graphs
....:                                                    1, q, q1, q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
sage: KL0.demazure_lusztig_operator_on_basis(L0((3,0)), 1, q1, q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
```

```
>>> from sage.all import *
>>> KL.demazure_lusztig_operator_on_classical_on_basis(L0((Integer(2),
↪Integer(2))),      # needs sage.graphs
...                                                    Integer(1), q, q1,␣
↪q2)
q1*B[(2, 2)]
>>> KL0.demazure_lusztig_operator_on_basis(L0((Integer(2),Integer(2))),␣
↪Integer(1), q1, q2)
q1*B[(2, 2)]

>>> KL.demazure_lusztig_operator_on_classical_on_basis(L0((Integer(3),
↪Integer(0))),      # needs sage.graphs
...                                                    Integer(1), q, q1,␣
↪q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
>>> KL0.demazure_lusztig_operator_on_basis(L0((Integer(3),Integer(0))),␣
↪Integer(1), q1, q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
```

except that we now have an action of T_0, which introduces some q s:

```
sage: KL.demazure_lusztig_operator_on_classical_on_basis(L0((2,2)),      #␣
↪needs sage.graphs
....:                                                    0, q, q1, q2)
q1*B[(2, 2)]
sage: KL.demazure_lusztig_operator_on_classical_on_basis(L0((3,0)),      #␣
↪needs sage.graphs
....:                                                    0, q, q1, q2)
-(q^2*q1+q^2*q2)*B[(1, 2)] - (q*q1+q*q2)*B[(2, 1)] - q^3*q2*B[(0, 3)]
```

```
>>> from sage.all import *
>>> KL.demazure_lusztig_operator_on_classical_on_basis(L0((Integer(2),
↪Integer(2))),      # needs sage.graphs
...                                                    Integer(0), q, q1,␣
↪q2)
q1*B[(2, 2)]
>>> KL.demazure_lusztig_operator_on_classical_on_basis(L0((Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0))),      # needs sage.graphs
...                                                    Integer(0), q, q1,␣
↪q2)
-(q^2*q1+q^2*q2)*B[(1, 2)] - (q*q1+q*q2)*B[(2, 1)] - q^3*q2*B[(0, 3)]
```

demazure_lusztig_operators(*q1*, *q2*, *convention='antidominant'*)

Return the Demazure-Lusztig operators acting on `self`.

INPUT:

- `q1`, `q2` – two elements of the ground ring
- `convention` – "antidominant", "bar", or "dominant" (default: `'antidominant'`)

If R is the parent weight ring, the Demazure-Lusztig operator T_i is the linear map $R \to R$ obtained by interpolating between the isobaric divided difference operator π_i (see `isobaric_divided_difference_on_basis()`) and the simple reflection s_i.

$$(q_1 + q_2)\pi_i - q_2 s_i$$

The Demazure-Lusztig operators give the usual representation of the operator T_i of the (affine) Hecke algebra with eigenvalues q_1 and q_2 associated to the Weyl group.

Several variants are available to match with various conventions used in the literature:

- "bar" replaces π_i in the formula above by $\overline{\pi}_i = (1 - \pi_i)$.
- "dominant" conjugates the operator by $x^\lambda \mapsto x^-\lambda$.

The names dominant and antidominant for the conventions were chosen with regards to the nonsymmetric Macdonald polynomials. The Y operators for the Macdonald polynomials in the "dominant" convention satisfy $Y_\lambda = T_{t_\lambda}$ for λ dominant. This is also the convention used in [Haiman06]. For the "antidominant" convention, $Y_\lambda = T_{t_\lambda}$ with λ antidominant.

See also

- `demazure_lusztig_operator_on_basis()`.
- `NonSymmetricMacdonaldPolynomials`.

REFERENCES:

EXAMPLES:

```
sage: L = RootSystem(["A",1]).ambient_space()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: T = KL.demazure_lusztig_operators(q1, q2)
sage: Tbar = KL.demazure_lusztig_operators(q1, q2, convention='bar')
sage: Tdominant = KL.demazure_lusztig_operators(q1, q2,
....:                                           convention='dominant')
sage: x = KL.monomial(L((3,0)))
sage: T[1](x)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
sage: Tbar[1](x)
-(q1+q2)*B[(1, 2)] - (q1+q2)*B[(2, 1)] - (q1+2*q2)*B[(0, 3)]
sage: Tbar[1](x) + T[1](x)
(q1+q2)*B[(3, 0)] - 2*q2*B[(0, 3)]
```

(continues on next page)

(continued from previous page)

```
sage: Tdominant[1](x)
-(q1+q2)*B[(1, 2)] - (q1+q2)*B[(2, 1)] - q2*B[(0, 3)]

sage: Tdominant.Tw_inverse(1)(KL.monomial(-L.simple_root(1)))
-((q1+q2)/(q1*q2))*B[(0, 0)] - 1/q2*B[(1, -1)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1)]).ambient_space()
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KL = L.algebra(K)
>>> T = KL.demazure_lusztig_operators(q1, q2)
>>> Tbar = KL.demazure_lusztig_operators(q1, q2, convention='bar')
>>> Tdominant = KL.demazure_lusztig_operators(q1, q2,
...                                           convention='dominant')
>>> x = KL.monomial(L((Integer(3),Integer(0))))
>>> T[Integer(1)](x)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
>>> Tbar[Integer(1)](x)
-(q1+q2)*B[(1, 2)] - (q1+q2)*B[(2, 1)] - (q1+2*q2)*B[(0, 3)]
>>> Tbar[Integer(1)](x) + T[Integer(1)](x)
(q1+q2)*B[(3, 0)] - 2*q2*B[(0, 3)]
>>> Tdominant[Integer(1)](x)
-(q1+q2)*B[(1, 2)] - (q1+q2)*B[(2, 1)] - q2*B[(0, 3)]

>>> Tdominant.Tw_inverse(Integer(1))(KL.monomial(-L.simple_
↪root(Integer(1))))
-((q1+q2)/(q1*q2))*B[(0, 0)] - 1/q2*B[(1, -1)]
```

We repeat similar computation in the affine setting:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: T = KL.demazure_lusztig_operators(q1, q2)
sage: Tbar = KL.demazure_lusztig_operators(q1, q2, convention='bar')
sage: Tdominant = KL.demazure_lusztig_operators(q1, q2,
....:                                           convention='dominant')
sage: e = L.basis()
sage: x = KL.monomial(3*e[0])
sage: T[1](x)
(q1+q2)*B[e[0] + 2*e[1]] + (q1+q2)*B[2*e[0] + e[1]]
 + (q1+q2)*B[3*e[0]] + q1*B[3*e[1]]
sage: Tbar[1](x)
-(q1+q2)*B[e[0] + 2*e[1]] - (q1+q2)*B[2*e[0] + e[1]] - (q1+2*q2)*B[3*e[1]]
sage: Tbar[1](x) + T[1](x)
(q1+q2)*B[3*e[0]] - 2*q2*B[3*e[1]]
sage: Tdominant[1](x)
-(q1+q2)*B[e[0] + 2*e[1]] - (q1+q2)*B[2*e[0] + e[1]] - q2*B[3*e[1]]
sage: Tdominant.Tw_inverse(1)(KL.monomial(-L.simple_root(1)))
-((q1+q2)/(q1*q2))*B[0] - 1/q2*B[e[0] - e[1]]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KL = L.algebra(K)
>>> T = KL.demazure_lusztig_operators(q1, q2)
>>> Tbar = KL.demazure_lusztig_operators(q1, q2, convention='bar')
>>> Tdominant = KL.demazure_lusztig_operators(q1, q2,
...                                           convention='dominant')
>>> e = L.basis()
>>> x = KL.monomial(Integer(3)*e[Integer(0)])
>>> T[Integer(1)](x)
(q1+q2)*B[e[0] + 2*e[1]] + (q1+q2)*B[2*e[0] + e[1]]
 + (q1+q2)*B[3*e[0]] + q1*B[3*e[1]]
>>> Tbar[Integer(1)](x)
-(q1+q2)*B[e[0] + 2*e[1]] - (q1+q2)*B[2*e[0] + e[1]] - (q1+2*q2)*B[3*e[1]]
>>> Tbar[Integer(1)](x) + T[Integer(1)](x)
(q1+q2)*B[3*e[0]] - 2*q2*B[3*e[1]]
>>> Tdominant[Integer(1)](x)
-(q1+q2)*B[e[0] + 2*e[1]] - (q1+q2)*B[2*e[0] + e[1]] - q2*B[3*e[1]]
>>> Tdominant.Tw_inverse(Integer(1))(KL.monomial(-L.simple_
↪root(Integer(1))))
-((q1+q2)/(q1*q2))*B[0] - 1/q2*B[e[0] - e[1]]
```

One can obtain iterated operators by passing a reduced word or an element of the Weyl group:

```
sage: T[1,2](x)
(q1^2+2*q1*q2+q2^2)*B[e[0] + e[1] + e[2]] +
(q1^2+2*q1*q2+q2^2)*B[e[0] + 2*e[1]] +
(q1^2+q1*q2)*B[e[0] + 2*e[2]] + (q1^2+2*q1*q2+q2^2)*B[2*e[0] + e[1]] +
(q1^2+q1*q2)*B[2*e[0] + e[2]] + (q1^2+q1*q2)*B[3*e[0]] +
(q1^2+q1*q2)*B[e[1] + 2*e[2]] + (q1^2+q1*q2)*B[2*e[1] + e[2]] +
(q1^2+q1*q2)*B[3*e[1]] + q1^2*B[3*e[2]]
```

```
>>> from sage.all import *
>>> T[Integer(1),Integer(2)](x)
(q1^2+2*q1*q2+q2^2)*B[e[0] + e[1] + e[2]] +
(q1^2+2*q1*q2+q2^2)*B[e[0] + 2*e[1]] +
(q1^2+q1*q2)*B[e[0] + 2*e[2]] + (q1^2+2*q1*q2+q2^2)*B[2*e[0] + e[1]] +
(q1^2+q1*q2)*B[2*e[0] + e[2]] + (q1^2+q1*q2)*B[3*e[0]] +
(q1^2+q1*q2)*B[e[1] + 2*e[2]] + (q1^2+q1*q2)*B[2*e[1] + e[2]] +
(q1^2+q1*q2)*B[3*e[1]] + q1^2*B[3*e[2]]
```

and use that to check, for example, the braid relations:

```
sage: T[1,2,1](x) - T[2,1,2](x)
0
```

```
>>> from sage.all import *
>>> T[Integer(1),Integer(2),Integer(1)](x) - T[Integer(2),Integer(1),
↪Integer(2)](x)
0
```

The operators satisfy the relations of the affine Hecke algebra with parameters q_1, q_2:

```
sage: T._test_relations()
sage: Tdominant._test_relations()
sage: Tbar._test_relations() #-q2,q1+2*q2   # todo: not implemented: set␣
↪the appropriate eigenvalues!
```

```
>>> from sage.all import *
>>> T._test_relations()
>>> Tdominant._test_relations()
>>> Tbar._test_relations() #-q2,q1+2*q2   # todo: not implemented: set␣
↪the appropriate eigenvalues!
```

And the $\bar{T}$ are basically the inverses of the T s:

```
sage: Tinv = KL.demazure_lusztig_operators(2/q1 + 1/q2, -1/q1,
....:                                       convention='bar')
sage: [Tinv[1](T[1](x)) - x for x in KL.some_elements()]                #␣
↪needs sage.graphs
[0, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> Tinv = KL.demazure_lusztig_operators(Integer(2)/q1 + Integer(1)/q2, -
↪Integer(1)/q1,
...                                       convention='bar')
>>> [Tinv[Integer(1)](T[Integer(1)](x)) - x for x in KL.some_elements()] ␣
↪              # needs sage.graphs
[0, 0, 0, 0, 0, 0, 0]
```

We check that $\Lambda_1 - \Lambda_0$ is an eigenvector for the Y s in affine type:

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q,q1,q2 = K.gens()
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: L0 = L.classical()
sage: Lambda = L.fundamental_weights()                                   #␣
↪needs sage.graphs
sage: alphacheck = L0.simple_coroots()
sage: KL = L.algebra(K)
sage: T = KL.demazure_lusztig_operators(q1, q2, convention='dominant')
sage: Y = T.Y()
sage: alphacheck = Y.keys().alpha()  # alpha of coroot lattice is␣
↪alphacheck
sage: alphacheck
Finite family {0: alphacheck[0], 1: alphacheck[1], 2: alphacheck[2]}
sage: x = KL.monomial(Lambda[1] - Lambda[0]); x                          #␣
↪needs sage.graphs
B[e[0]]
```

```
>>> from sage.all import *
>>> K = QQ['q,q1,q2'].fraction_field()
>>> q,q1,q2 = K.gens()
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
```

(continues on next page)

(continued from previous page)

```
>>> L0 = L.classical()
>>> Lambda = L.fundamental_weights()                                    #␣
↪needs sage.graphs
>>> alphacheck = L0.simple_coroots()
>>> KL = L.algebra(K)
>>> T = KL.demazure_lusztig_operators(q1, q2, convention='dominant')
>>> Y = T.Y()
>>> alphacheck = Y.keys().alpha()  # alpha of coroot lattice is alphacheck
>>> alphacheck
Finite family {0: alphacheck[0], 1: alphacheck[1], 2: alphacheck[2]}
>>> x = KL.monomial(Lambda[Integer(1)] - Lambda[Integer(0)]); x         ␣
↪              # needs sage.graphs
B[e[0]]
```

In fact it is not exactly an eigenvector, but the extra 'delta` term is to be interpreted as a q parameter:

```
sage: # needs sage.graphs
sage: Y[alphacheck[0]](KL.one())
q2^2/q1^2*B[0]
sage: Y[alphacheck[1]](x)
-(q2^2/(-q1^2))*B[e[0] - e['delta']]
sage: Y[alphacheck[2]](x)
(q1/(-q2))*B[e[0]]
sage: KL.q_project(Y[alphacheck[1]](x),q)
-(q2^2/(-q*q1^2))*B[(1, 0, 0)]

sage: # needs sage.graphs
sage: KL.q_project(x, q)
B[(1, 0, 0)]
sage: KL.q_project(Y[alphacheck[0]](x),q)
-q*q1/q2*B[(1, 0, 0)]
sage: KL.q_project(Y[alphacheck[1]](x),q)
-(q2^2/(-q*q1^2))*B[(1, 0, 0)]
sage: KL.q_project(Y[alphacheck[2]](x),q)
(q1/(-q2))*B[(1, 0, 0)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> Y[alphacheck[Integer(0)]](KL.one())
q2^2/q1^2*B[0]
>>> Y[alphacheck[Integer(1)]](x)
-(q2^2/(-q1^2))*B[e[0] - e['delta']]
>>> Y[alphacheck[Integer(2)]](x)
(q1/(-q2))*B[e[0]]
>>> KL.q_project(Y[alphacheck[Integer(1)]](x),q)
-(q2^2/(-q*q1^2))*B[(1, 0, 0)]

>>> # needs sage.graphs
>>> KL.q_project(x, q)
B[(1, 0, 0)]
>>> KL.q_project(Y[alphacheck[Integer(0)]](x),q)
-q*q1/q2*B[(1, 0, 0)]
```

(continues on next page)

(continued from previous page)

```
>>> KL.q_project(Y[alphacheck[Integer(1)]](x),q)
-(q2^2/(-q*q1^2))*B[(1, 0, 0)]
>>> KL.q_project(Y[alphacheck[Integer(2)]](x),q)
(q1/(-q2))*B[(1, 0, 0)]
```

We now check systematically that the Demazure-Lusztig operators satisfy the relations of the Iwahori-Hecke algebra:

```
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: for cartan_type in CartanType.samples(crystallographic=True):   #␣
↪long time (12s)
....:     L = RootSystem(cartan_type).root_lattice()
....:     KL = L.algebra(K)
....:     T = KL.demazure_lusztig_operators(q1,q2)
....:     T._test_relations()

sage: for cartan_type in CartanType.samples(crystallographic=True):   #␣
↪long time (12s)
....:     L = RootSystem(cartan_type).weight_lattice()
....:     KL = L.algebra(K)
....:     T = KL.demazure_lusztig_operators(q1,q2)
....:     T._test_relations()
```

```
>>> from sage.all import *
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> for cartan_type in CartanType.samples(crystallographic=True):   # long␣
↪time (12s)
...     L = RootSystem(cartan_type).root_lattice()
...     KL = L.algebra(K)
...     T = KL.demazure_lusztig_operators(q1,q2)
...     T._test_relations()

>>> for cartan_type in CartanType.samples(crystallographic=True):   # long␣
↪time (12s)
...     L = RootSystem(cartan_type).weight_lattice()
...     KL = L.algebra(K)
...     T = KL.demazure_lusztig_operators(q1,q2)
...     T._test_relations()
```

Recall that the Demazure-Lusztig operators are only defined when all monomials belong to the weight lattice. Thus, in the group algebra of the ambient space, we need to specify explicitly the elements on which to run the tests:

```
sage: for cartan_type in CartanType.samples(crystallographic=True):   #␣
↪long time (12s)
....:     L = RootSystem(cartan_type).ambient_space()
....:     KL = L.algebra(K)
....:     weight_lattice = RootSystem(cartan_type).weight_
↪lattice(extended=L.is_extended())
....:     elements = [ KL.monomial(L(weight)) for weight in weight_lattice.
```

(continues on next page)

(continued from previous page)

```
↪some_elements() ]
....:     T = KL.demazure_lusztig_operators(q1,q2)
....:     T._test_relations(elements=elements)
```

```
>>> from sage.all import *
>>> for cartan_type in CartanType.samples(crystallographic=True):  # long␣
↪time (12s)
...     L = RootSystem(cartan_type).ambient_space()
...     KL = L.algebra(K)
...     weight_lattice = RootSystem(cartan_type).weight_lattice(extended=L.
↪is_extended())
...     elements = [ KL.monomial(L(weight)) for weight in weight_lattice.
↪some_elements() ]
...     T = KL.demazure_lusztig_operators(q1,q2)
...     T._test_relations(elements=elements)
```

demazure_lusztig_operators_on_classical(*q*, *q1*, *q2*, *convention='antidominant'*)

Return the Demazure-Lusztig operators acting at level 1 on `self.classical()`.

INPUT:

- `q`, `q1`, `q2` – three elements of the ground ring
- `convention` – `'antidominant'`, `'bar'`, or `'dominant'` (default: `'antidominant'`)

Let KL be the group algebra of an affine weight lattice realization L. The Demazure-Lusztig operators for KL act on the group algebra of the corresponding classical weight lattice by embedding it at level 1, and projecting back.

See also

- *demazure_lusztig_operators()*.
- *demazure_lusztig_operator_on_classical_on_basis()*.
- *q_project()*

EXAMPLES:

```
sage: L = RootSystem(["A",1,1]).ambient_space()
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q, q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: KL0 = KL.classical()
sage: L0 = KL0.basis().keys()
sage: T = KL.demazure_lusztig_operators_on_classical(q, q1, q2)          #␣
↪needs sage.graphs

sage: x = KL0.monomial(L0((3,0))); x
B[(3, 0)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1),Integer(1)]).ambient_space()
>>> K = QQ['q,q1,q2'].fraction_field()
>>> q, q1, q2 = K.gens()
```

(continues on next page)

(continued from previous page)

```
>>> KL = L.algebra(K)
>>> KL0 = KL.classical()
>>> L0 = KL0.basis().keys()
>>> T = KL.demazure_lusztig_operators_on_classical(q, q1, q2)          #␣
↪needs sage.graphs

>>> x = KL0.monomial(L0((Integer(3),Integer(0)))); x
B[(3, 0)]
```

For $T_1, \ldots$ we recover the usual Demazure-Lusztig operators:

```
sage: T[1](x)                                                           #␣
↪needs sage.graphs
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
```

```
>>> from sage.all import *
>>> T[Integer(1)](x)                                                    ␣
↪     # needs sage.graphs
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
```

For T_0, we can note that, in the projection, δ is mapped to q:

```
sage: T[0](x)                                                           #␣
↪needs sage.graphs
-(q^2*q1+q^2*q2)*B[(1, 2)] - (q*q1+q*q2)*B[(2, 1)] - q^3*q2*B[(0, 3)]
```

```
>>> from sage.all import *
>>> T[Integer(0)](x)                                                    ␣
↪     # needs sage.graphs
-(q^2*q1+q^2*q2)*B[(1, 2)] - (q*q1+q*q2)*B[(2, 1)] - q^3*q2*B[(0, 3)]
```

Note that there is no translation part, and in particular 1 is an eigenvector for all T_i's:

```
sage: # needs sage.graphs
sage: T[0](KL0.one())
q1*B[(0, 0)]
sage: T[1](KL0.one())
q1*B[(0, 0)]
sage: Y = T.Y()
sage: alphacheck = Y.keys().simple_roots()
sage: Y[alphacheck[0]](KL0.one())
-q2/(q*q1)*B[(0, 0)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> T[Integer(0)](KL0.one())
q1*B[(0, 0)]
>>> T[Integer(1)](KL0.one())
q1*B[(0, 0)]
>>> Y = T.Y()
>>> alphacheck = Y.keys().simple_roots()
```

(continues on next page)

(continued from previous page)

```
>>> Y[alphacheck[Integer(0)]](KL0.one())
-q2/(q*q1)*B[(0, 0)]
```

Matching with Ion Bogdan's hand calculations from 3/15/2013:

```
sage: L = RootSystem(["A",1,1]).weight_space(extended=True)
sage: K = QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: KL = L.algebra(K)
sage: KL0 = KL.classical()
sage: L0 = KL0.basis().keys()
sage: omega = L0.fundamental_weights()

sage: # needs sage.graphs
sage: T = KL.demazure_lusztig_operators_on_classical(q, u, -1/u,
....:                                                  convention='dominant
↪')
sage: Y = T.Y()
sage: alphacheck = Y.keys().simple_roots()
sage: Ydelta = Y[Y.keys().null_root()]
sage: Ydelta.word, Ydelta.signs, Ydelta.scalar
((), (), 1/q)
sage: Y1 = Y[alphacheck[1]]
sage: Y1.word, Y1.signs, Y1.scalar  # This is T_0 T_1 (T_1 acts first,␣
↪then T_0); Ion gets T_1 T_0
((1, 0), (1, 1), 1)
sage: Y0 = Y[alphacheck[0]]
sage: Y0.word, Y0.signs, Y0.scalar  # This is 1/q T_1^-1 T_0^-1
((0, 1), (-1, -1), 1/q)
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1),Integer(1)]).weight_
↪space(extended=True)
>>> K = QQ['q,u'].fraction_field()
>>> q, u = K.gens()
>>> KL = L.algebra(K)
>>> KL0 = KL.classical()
>>> L0 = KL0.basis().keys()
>>> omega = L0.fundamental_weights()

>>> # needs sage.graphs
>>> T = KL.demazure_lusztig_operators_on_classical(q, u, -Integer(1)/u,
...                                                  convention='dominant')
>>> Y = T.Y()
>>> alphacheck = Y.keys().simple_roots()
>>> Ydelta = Y[Y.keys().null_root()]
>>> Ydelta.word, Ydelta.signs, Ydelta.scalar
((), (), 1/q)
>>> Y1 = Y[alphacheck[Integer(1)]]
>>> Y1.word, Y1.signs, Y1.scalar  # This is T_0 T_1 (T_1 acts first, then␣
↪T_0); Ion gets T_1 T_0
((1, 0), (1, 1), 1)
```

(continues on next page)

(continued from previous page)

```
>>> Y0 = Y[alphacheck[Integer(0)]]
>>> Y0.word, Y0.signs, Y0.scalar  # This is 1/q T_1^-1 T_0^-1
((0, 1), (-1, -1), 1/q)
```

Note that the following computations use the “dominant” convention:

```
sage: # needs sage.graphs
sage: T0 = T.Tw(0)
sage: T0(KL0.monomial(omega[1]))
q*u*B[-Lambda[1]] + ((u^2-1)/u)*B[Lambda[1]]
sage: T0(KL0.monomial(2*omega[1]))
((q*u^2-q)/u)*B[0] + q^2*u*B[-2*Lambda[1]] + ((u^2-1)/u)*B[2*Lambda[1]]
sage: T0(KL0.monomial(-omega[1]))
1/(q*u)*B[Lambda[1]]
sage: T0(KL0.monomial(-2*omega[1]))
-((u^2-1)/(q*u))*B[0] + 1/(q^2*u)*B[2*Lambda[1]]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> T0 = T.Tw(Integer(0))
>>> T0(KL0.monomial(omega[Integer(1)]))
q*u*B[-Lambda[1]] + ((u^2-1)/u)*B[Lambda[1]]
>>> T0(KL0.monomial(Integer(2)*omega[Integer(1)]))
((q*u^2-q)/u)*B[0] + q^2*u*B[-2*Lambda[1]] + ((u^2-1)/u)*B[2*Lambda[1]]
>>> T0(KL0.monomial(-omega[Integer(1)]))
1/(q*u)*B[Lambda[1]]
>>> T0(KL0.monomial(-Integer(2)*omega[Integer(1)]))
-((u^2-1)/(q*u))*B[0] + 1/(q^2*u)*B[2*Lambda[1]]
```

demazure_operators()

Return the Demazure operators acting on `self`.

The i-th Demazure operator is defined by:

$$\pi_i = \frac{1 - e^{-\alpha_i}s_i}{1 - e^{-\alpha_i}}$$

It acts on e^λ, for λ a weight, by:

$$\pi_i e^\lambda = \frac{e^\lambda - e^{-\alpha_i + s_i\lambda}}{1 - e^{-\alpha_i}}$$

This matches with Lascoux’ definition [Lascoux2003] of π_i, and with the i-th Demazure operator of [Kumar1987], which also works for general Kac-Moody types.

REFERENCES:

EXAMPLES:

We compute some Schur functions, as images of dominant monomials under the action of the maximal isobaric divided difference Δ_{w_0}:

```
sage: L = RootSystem(["A",2]).ambient_lattice()
sage: KL = L.algebra(QQ)
sage: w0 = tuple(L.weyl_group().long_element().reduced_word())             #
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.gap
sage: pi = KL.demazure_operators()
sage: pi0 = pi[w0]                                                             #␣
↪needs sage.libs.gap
sage: pi0(KL.monomial(L((2,1))))                                               #␣
↪needs sage.libs.gap
2*B[(1, 1, 1)] + B[(1, 2, 0)] + B[(1, 0, 2)] + B[(2, 1, 0)]
 + B[(2, 0, 1)] + B[(0, 1, 2)] + B[(0, 2, 1)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).ambient_lattice()
>>> KL = L.algebra(QQ)
>>> w0 = tuple(L.weyl_group().long_element().reduced_word())                 #␣
↪needs sage.libs.gap
>>> pi = KL.demazure_operators()
>>> pi0 = pi[w0]                                                             #␣
↪needs sage.libs.gap
>>> pi0(KL.monomial(L((Integer(2),Integer(1)))))                              ␣
↪              # needs sage.libs.gap
2*B[(1, 1, 1)] + B[(1, 2, 0)] + B[(1, 0, 2)] + B[(2, 1, 0)]
 + B[(2, 0, 1)] + B[(0, 1, 2)] + B[(0, 2, 1)]
```

Let us make the result into an actual polynomial:

```
sage: P = QQ['x,y,z']
sage: pi0(KL.monomial(L((2,1)))).expand(P.gens())                              #␣
↪needs sage.libs.gap
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

```
>>> from sage.all import *
>>> P = QQ['x,y,z']
>>> pi0(KL.monomial(L((Integer(2),Integer(1))))).expand(P.gens())             ␣
↪              # needs sage.libs.gap
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

This is indeed a Schur function:

```
sage: s = SymmetricFunctions(QQ).s()                                           #␣
↪needs sage.combinat
sage: s[2,1].expand(3, P.variable_names())                                     #␣
↪needs sage.combinat
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()                                           #␣
↪needs sage.combinat
>>> s[Integer(2),Integer(1)].expand(Integer(3), P.variable_names())           ␣
↪                       # needs sage.combinat
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

Let us check this systematically on Schur functions of degree 6:

```
sage: for p in Partitions(6, max_length=3).list():                          #␣
↪needs sage.combinat sage.libs.gap
....:     assert (s.monomial(p).expand(3, P.variable_names())
....:              == pi0(KL.monomial(L(tuple(p)))).expand(P.gens()))
```

```
>>> from sage.all import *
>>> for p in Partitions(Integer(6), max_length=Integer(3)).list():              ␣
↪             # needs sage.combinat sage.libs.gap
...     assert (s.monomial(p).expand(Integer(3), P.variable_names())
...              == pi0(KL.monomial(L(tuple(p)))).expand(P.gens()))
```

We check systematically that these operators satisfy the Iwahori-Hecke algebra relations:

```
sage: for cartan_type in CartanType.samples(crystallographic=True):  #␣
↪long time (12s)
....:     L = RootSystem(cartan_type).weight_lattice()
....:     KL = L.algebra(QQ)
....:     T = KL.demazure_operators()
....:     T._test_relations()

sage: L = RootSystem(['A',1,1]).weight_lattice()
sage: KL = L.algebra(QQ)
sage: T = KL.demazure_operators()
sage: T._test_relations()
```

```
>>> from sage.all import *
>>> for cartan_type in CartanType.samples(crystallographic=True):  # long␣
↪time (12s)
...     L = RootSystem(cartan_type).weight_lattice()
...     KL = L.algebra(QQ)
...     T = KL.demazure_operators()
...     T._test_relations()

>>> L = RootSystem(['A',Integer(1),Integer(1)]).weight_lattice()
>>> KL = L.algebra(QQ)
>>> T = KL.demazure_operators()
>>> T._test_relations()
```

Warning

The Demazure operators are only defined if all the elements in the support have integral scalar products with the coroots (basically, they are in the weight lattice). Otherwise an error is raised:

```
sage: L = RootSystem(CartanType(["G",2]).dual()).ambient_space()
sage: KL = L.algebra(QQ)
sage: pi = KL.demazure_operators()
sage: pi[1](KL.monomial(L([0,0,1])))
Traceback (most recent call last):
...
ValueError: the weight does not have an integral scalar product with␣
↪the coroot
```

```
>>> from sage.all import *
>>> L = RootSystem(CartanType(["G",Integer(2)]).dual()).ambient_space()
>>> KL = L.algebra(QQ)
>>> pi = KL.demazure_operators()
>>> pi[Integer(1)](KL.monomial(L([Integer(0),Integer(0),Integer(1)])))
Traceback (most recent call last):
...
ValueError: the weight does not have an integral scalar product with␣
↪the coroot
```

divided_difference_on_basis(*weight*, *i*)

Return the result of applying the i-th divided difference on `weight`.

INPUT:

- `weight` – a weight
- `i` – an element of the index set

Todo

type free definition (Viviane's definition uses that we are in the ambient space)

EXAMPLES:

```
sage: L = RootSystem(["A",1]).ambient_space()
sage: KL = L.algebra(QQ)
sage: KL.divided_difference_on_basis(L((2,2)), 1) # todo: not implemented
0
sage: KL.divided_difference_on_basis(L((3,0)), 1) # todo: not implemented
B[(2, 0)] + B[(1, 1)] + B[(0, 2)]
sage: KL.divided_difference_on_basis(L((0,3)), 1) # todo: not implemented
-B[(2, 0)] - B[(1, 1)] - B[(0, 2)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1)]).ambient_space()
>>> KL = L.algebra(QQ)
>>> KL.divided_difference_on_basis(L((Integer(2),Integer(2))),␣
↪Integer(1)) # todo: not implemented
0
>>> KL.divided_difference_on_basis(L((Integer(3),Integer(0))),␣
↪Integer(1)) # todo: not implemented
B[(2, 0)] + B[(1, 1)] + B[(0, 2)]
>>> KL.divided_difference_on_basis(L((Integer(0),Integer(3))),␣
↪Integer(1)) # todo: not implemented
-B[(2, 0)] - B[(1, 1)] - B[(0, 2)]
```

In type A and in the ambient lattice, we recover the usual action of divided differences polynomials:

```
sage: x,y = QQ['x,y'].gens()
sage: d = lambda p: (p - p(y,x)) / (x-y)
sage: d(x^2*y^2)
0
```

(continues on next page)

(continued from previous page)

```
sage: d(x^3)
x^2 + x*y + y^2
sage: d(y^3)
-x^2 - x*y - y^2
```

```
>>> from sage.all import *
>>> x,y = QQ['x,y'].gens()
>>> d = lambda p: (p - p(y,x)) / (x-y)
>>> d(x**Integer(2)*y**Integer(2))
0
>>> d(x**Integer(3))
x^2 + x*y + y^2
>>> d(y**Integer(3))
-x^2 - x*y - y^2
```

from_polynomial(*p*)

Construct an element of `self` from a polynomial p.

INPUT:

- `p` – a polynomial

EXAMPLES:

```
sage: L = RootSystem(["A",2]).ambient_lattice()
sage: KL = L.algebra(QQ)
sage: x,y,z = QQ['x,y,z'].gens()
sage: KL.from_polynomial(x)
B[(1, 0, 0)]
sage: KL.from_polynomial(x^2*y + 2*y - z)
B[(2, 1, 0)] + 2*B[(0, 1, 0)] - B[(0, 0, 1)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).ambient_lattice()
>>> KL = L.algebra(QQ)
>>> x,y,z = QQ['x,y,z'].gens()
>>> KL.from_polynomial(x)
B[(1, 0, 0)]
>>> KL.from_polynomial(x**Integer(2)*y + Integer(2)*y - z)
B[(2, 1, 0)] + 2*B[(0, 1, 0)] - B[(0, 0, 1)]
```

Todo

make this work for Laurent polynomials too

isobaric_divided_difference_on_basis(*weight*, *i*)

Return the result of applying the i-th isobaric divided difference on `weight`.

INPUT:

- `weight` – a weight
- `i` – an element of the index set

See also:

demazure_operators()

EXAMPLES:

```
sage: L = RootSystem(["A",1]).ambient_space()
sage: KL = L.algebra(QQ)
sage: KL.isobaric_divided_difference_on_basis(L((2,2)), 1)
B[(2, 2)]
sage: KL.isobaric_divided_difference_on_basis(L((3,0)), 1)
B[(1, 2)] + B[(2, 1)] + B[(3, 0)] + B[(0, 3)]
sage: KL.isobaric_divided_difference_on_basis(L((0,3)), 1)
-B[(1, 2)] - B[(2, 1)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(1)]).ambient_space()
>>> KL = L.algebra(QQ)
>>> KL.isobaric_divided_difference_on_basis(L((Integer(2),Integer(2))),␣
↪Integer(1))
B[(2, 2)]
>>> KL.isobaric_divided_difference_on_basis(L((Integer(3),Integer(0))),␣
↪Integer(1))
B[(1, 2)] + B[(2, 1)] + B[(3, 0)] + B[(0, 3)]
>>> KL.isobaric_divided_difference_on_basis(L((Integer(0),Integer(3))),␣
↪Integer(1))
-B[(1, 2)] - B[(2, 1)]
```

In type A and in the ambient lattice, we recover the usual action of divided differences on polynomials:

```
sage: x,y = QQ['x,y'].gens()
sage: d = lambda p: (x*p - (x*p)(y,x)) / (x-y)
sage: d(x^2*y^2)
x^2*y^2
sage: d(x^3)
x^3 + x^2*y + x*y^2 + y^3
sage: d(y^3)
-x^2*y - x*y^2
```

```
>>> from sage.all import *
>>> x,y = QQ['x,y'].gens()
>>> d = lambda p: (x*p - (x*p)(y,x)) / (x-y)
>>> d(x**Integer(2)*y**Integer(2))
x^2*y^2
>>> d(x**Integer(3))
x^3 + x^2*y + x*y^2 + y^3
>>> d(y**Integer(3))
-x^2*y - x*y^2
```

REFERENCES:

q_project(x, q)

Implement the q-projection morphism from `self` to the group algebra of the classical space.

INPUT:

- `x` – an element of the group algebra of `self`
- `q` – an element of the ground ring

This is an algebra morphism mapping δ to q and X^b to its classical counterpart for the other elements b of the basis of the realization.

EXAMPLES:

```
sage: K = QQ['q'].fraction_field()
sage: q = K.gen()
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: L = KL.basis().keys()
sage: e = L.basis()
sage: x = (KL.an_element()
....:       + KL.monomial(4*e[1] + 3*e[2]
....:                     + e['deltacheck'] - 2*e['delta'])); x
B[2*e[0] + 2*e[1] + 3*e[2]]
 + B[4*e[1] + 3*e[2] - 2*e['delta'] + e['deltacheck']]
sage: KL.q_project(x, q)
B[(2, 2, 3)] + 1/q^2*B[(0, 4, 3)]

sage: KL = RootSystem(["BC",3,2]).ambient_space().algebra(K)
sage: L = KL.basis().keys()
sage: e = L.basis()
sage: x = (KL.an_element()
....:       + KL.monomial(4*e[1] + 3*e[2]
....:                     + e['deltacheck'] - 2*e['delta'])); x
B[2*e[0] + 2*e[1] + 3*e[2]]
 + B[4*e[1] + 3*e[2] - 2*e['delta'] + e['deltacheck']]
sage: KL.q_project(x, q)
B[(2, 2, 3)] + 1/q^2*B[(0, 4, 3)]
```

```
>>> from sage.all import *
>>> K = QQ['q'].fraction_field()
>>> q = K.gen()
>>> KL = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().
↪algebra(K)
>>> L = KL.basis().keys()
>>> e = L.basis()
>>> x = (KL.an_element()
...       + KL.monomial(Integer(4)*e[Integer(1)] +␣
↪Integer(3)*e[Integer(2)]
...                     + e['deltacheck'] - Integer(2)*e['delta'])); x
B[2*e[0] + 2*e[1] + 3*e[2]]
 + B[4*e[1] + 3*e[2] - 2*e['delta'] + e['deltacheck']]
>>> KL.q_project(x, q)
B[(2, 2, 3)] + 1/q^2*B[(0, 4, 3)]

>>> KL = RootSystem(["BC",Integer(3),Integer(2)]).ambient_space().
↪algebra(K)
>>> L = KL.basis().keys()
>>> e = L.basis()
>>> x = (KL.an_element()
...       + KL.monomial(Integer(4)*e[Integer(1)] +␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)*e[Integer(2)]
...                          + e['deltacheck'] - Integer(2)*e['delta'])); x
B[2*e[0] + 2*e[1] + 3*e[2]]
 + B[4*e[1] + 3*e[2] - 2*e['delta'] + e['deltacheck']]
>>> KL.q_project(x, q)
B[(2, 2, 3)] + 1/q^2*B[(0, 4, 3)]
```

Warning

Recall that the null root, usually denoted δ, is in fact `a[0]\delta` in Sage's notation, in order to avoid half integer coefficients (this only makes a difference in type BC). Similarly, what's usually denoted q is in fact `q^a[0]` in Sage's notations, to avoid manipulating square roots:

```
sage: KL.q_project(KL.monomial(L.null_root()),q)                           #␣
↪needs sage.graphs
q^2*B[(0, 0, 0)]
```

```
>>> from sage.all import *
>>> KL.q_project(KL.monomial(L.null_root()),q)                           #␣
↪needs sage.graphs
q^2*B[(0, 0, 0)]
```

q_project_on_basis(l, q)

Return the monomial $c * cl(l)$ in the group algebra of the classical lattice.

INPUT:

- `l` – an element of the root lattice realization
- `q` – an element of the ground ring

Here, $cl(l)$ is the projection of l in the classical lattice, and c is the coefficient of l in δ.

See also

q_project_on_basis()

EXAMPLES:

```
sage: K = QQ['q'].fraction_field()
sage: q = K.gen()
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: L = KL.basis().keys()
sage: e = L.basis()
sage: KL.q_project_on_basis(4*e[1] + 3*e[2]
....:                          + e['deltacheck'] - 2*e['delta'], q)
1/q^2*B[(0, 4, 3)]
```

```
>>> from sage.all import *
>>> K = QQ['q'].fraction_field()
>>> q = K.gen()
>>> KL = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().
↪algebra(K)
```

(continues on next page)

(continued from previous page)

```
>>> L = KL.basis().keys()
>>> e = L.basis()
>>> KL.q_project_on_basis(Integer(4)*e[Integer(1)] +␣
↪Integer(3)*e[Integer(2)]
...                          + e['deltacheck'] - Integer(2)*e['delta'], q)
1/q^2*B[(0, 4, 3)]
```

some_elements()

Return some elements of the algebra self.

EXAMPLES:

```
sage: A = RootSystem(["A",2,1]).ambient_space().algebra(QQ)
sage: A.some_elements()
[B[2*e[0] + 2*e[1] + 3*e[2]]...]
sage: A.some_elements()                                                 #␣
↪needs sage.graphs
[B[2*e[0] + 2*e[1] + 3*e[2]],
 B[-e[0] + e[2] + e['delta']],
 B[e[0] - e[1]],
 B[e[1] - e[2]],
 B[e['deltacheck']],
 B[e[0] + e['deltacheck']],
 B[e[0] + e[1] + e['deltacheck']]]

sage: A = RootSystem(["B",2]).weight_space().algebra(QQ)
sage: A.some_elements()
[B[2*Lambda[1] + 2*Lambda[2]], ... B[Lambda[1]], B[Lambda[2]]]
sage: A.some_elements()                                                 #␣
↪needs sage.graphs
[B[2*Lambda[1] + 2*Lambda[2]],
 B[2*Lambda[1] - 2*Lambda[2]],
 B[-Lambda[1] + 2*Lambda[2]],
 B[Lambda[1]],
 B[Lambda[2]]]
```

```
>>> from sage.all import *
>>> A = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().
↪algebra(QQ)
>>> A.some_elements()
[B[2*e[0] + 2*e[1] + 3*e[2]]...]
>>> A.some_elements()                                                 #␣
↪needs sage.graphs
[B[2*e[0] + 2*e[1] + 3*e[2]],
 B[-e[0] + e[2] + e['delta']],
 B[e[0] - e[1]],
 B[e[1] - e[2]],
 B[e['deltacheck']],
 B[e[0] + e['deltacheck']],
 B[e[0] + e[1] + e['deltacheck']]]

>>> A = RootSystem(["B",Integer(2)]).weight_space().algebra(QQ)
```

(continues on next page)

(continued from previous page)

```
>>> A.some_elements()
[B[2*Lambda[1] + 2*Lambda[2]], ... B[Lambda[1]], B[Lambda[2]]]
>>> A.some_elements()                                                    #␣
↪needs sage.graphs
[B[2*Lambda[1] + 2*Lambda[2]],
 B[2*Lambda[1] - 2*Lambda[2]],
 B[-Lambda[1] + 2*Lambda[2]],
 B[Lambda[1]],
 B[Lambda[2]]]
```

twisted_demazure_lusztig_operator_on_basis(*weight*, *i*, *q1*, *q2*, *convention='antidominant'*)

Return the twisted Demazure-Lusztig operator acting on the basis.

INPUT:

- `weight` – an element λ of the weight lattice
- `i` – an element of the index set
- `q1`, `q2` – two elements of the ground ring
- `convention` – `'antidominant'`, `'bar'`, or `'dominant'` (default: `'antidominant'`)

See also

twisted_demazure_lusztig_operators()

EXAMPLES:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: e = L.basis()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: Lambda = L.classical().fundamental_weights()
sage: KL.twisted_demazure_lusztig_operator_on_basis(
....:     Lambda[1] + 2*Lambda[2], 1, q1, q2, convention='dominant')
-q2*B[(2, 3, 0, 0)]
sage: KL.twisted_demazure_lusztig_operator_on_basis(
....:     Lambda[1] + 2*Lambda[2], 2, q1, q2, convention='dominant')
-(q1+q2)*B[(3, 1, 1, 0)] - q2*B[(3, 0, 2, 0)]
sage: KL.twisted_demazure_lusztig_operator_on_basis(
....:     Lambda[1] + 2*Lambda[2], 3, q1, q2, convention='dominant')
q1*B[(3, 2, 0, 0)]
sage: KL.twisted_demazure_lusztig_operator_on_basis(                      #␣
↪needs sage.graphs
....:     Lambda[1]+2*Lambda[2], 0, q1, q2, convention='dominant')
((q1*q2+q2^2)/q1)*B[(1, 2, 1, 1)] + ((q1*q2+q2^2)/q1)*B[(1, 2, 2, 0)]
 + q2^2/q1*B[(1, 2, 0, 2)] + ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 1, 1, 1)]
 + ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 1, 2, 0)]
 + ((q1*q2+q2^2)/q1)*B[(2, 1, 0, 2)]
 + ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 2, 1, 0)]
 + ((q1*q2+q2^2)/q1)*B[(2, 2, 0, 1)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3),Integer(1)]).ambient_space()
```

(continues on next page)

(continued from previous page)

```
>>> e = L.basis()
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KL = L.algebra(K)
>>> Lambda = L.classical().fundamental_weights()
>>> KL.twisted_demazure_lusztig_operator_on_basis(
...     Lambda[Integer(1)] + Integer(2)*Lambda[Integer(2)], Integer(1),␣
↪q1, q2, convention='dominant')
-q2*B[(2, 3, 0, 0)]
>>> KL.twisted_demazure_lusztig_operator_on_basis(
...     Lambda[Integer(1)] + Integer(2)*Lambda[Integer(2)], Integer(2),␣
↪q1, q2, convention='dominant')
-(q1+q2)*B[(3, 1, 1, 0)] - q2*B[(3, 0, 2, 0)]
>>> KL.twisted_demazure_lusztig_operator_on_basis(
...     Lambda[Integer(1)] + Integer(2)*Lambda[Integer(2)], Integer(3),␣
↪q1, q2, convention='dominant')
q1*B[(3, 2, 0, 0)]
>>> KL.twisted_demazure_lusztig_operator_on_basis(                   #␣
↪needs sage.graphs
...     Lambda[Integer(1)]+Integer(2)*Lambda[Integer(2)], Integer(0), q1,␣
↪q2, convention='dominant')
((q1*q2+q2^2)/q1)*B[(1, 2, 1, 1)] + ((q1*q2+q2^2)/q1)*B[(1, 2, 2, 0)]
 + q2^2/q1*B[(1, 2, 0, 2)] + ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 1, 1, 1)]
 + ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 1, 2, 0)]
 + ((q1*q2+q2^2)/q1)*B[(2, 1, 0, 2)]
 + ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 2, 1, 0)]
 + ((q1*q2+q2^2)/q1)*B[(2, 2, 0, 1)]
```

twisted_demazure_lusztig_operators(*q1*, *q2*, *convention='antidominant'*)

Return the twisted Demazure-Lusztig operators acting on `self`.

INPUT:

- `q1`, `q2` – two elements of the ground ring
- `convention` – `'antidominant'`, `'bar'`, or `'dominant'` (default: `'antidominant'`)

Warning

- the code is currently only tested for $q_1 q_2 = -1$
- only the `'dominant'` convention is functional for $i = 0$

For $T_1, \ldots, T_n$, these operators are the usual Demazure-Lusztig operators. On the other hand, the operator T_0 is twisted:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: e = L.basis()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: T = KL.twisted_demazure_lusztig_operators(q1, q2, convention=
↪'dominant')
sage: T._test_relations()
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3),Integer(1)]).ambient_space()
>>> e = L.basis()
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KL = L.algebra(K)
>>> T = KL.twisted_demazure_lusztig_operators(q1, q2, convention='dominant
↪')
>>> T._test_relations()
```

Todo

Choose a good set of Cartan Type to run on. Rank >4 is too big. But C_1 and B_1 are boring.

We now check systematically that those operators satisfy the relations of the Iwahori-Hecke algebra:

```
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: for cartan_type in CartanType.samples(affine=True,␣
↪crystallographic=True):  # long time (12s)
....:     if cartan_type.rank() > 4: continue
....:     if cartan_type.type() == 'BC': continue
....:     KL = RootSystem(cartan_type).weight_lattice().algebra(K)
....:     T = KL.twisted_demazure_lusztig_operators(q1, q2, convention=
↪'dominant')
....:     T._test_relations()
```

```
>>> from sage.all import *
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> for cartan_type in CartanType.samples(affine=True,␣
↪crystallographic=True):  # long time (12s)
...     if cartan_type.rank() > Integer(4): continue
...     if cartan_type.type() == 'BC': continue
...     KL = RootSystem(cartan_type).weight_lattice().algebra(K)
...     T = KL.twisted_demazure_lusztig_operators(q1, q2, convention=
↪'dominant')
...     T._test_relations()
```

Todo

Investigate why $T_0^\vee$ currently does not satisfy the quadratic relation in type BC. This should hopefully be fixed when $T_0^\vee$ will have a more uniform implementation:

```
sage: cartan_type = CartanType(["BC",1,2])
sage: KL = RootSystem(cartan_type).weight_lattice().algebra(K)
sage: T = KL.twisted_demazure_lusztig_operators(q1,q2, convention=
↪'dominant')
sage: T._test_relations()                                          #␣
↪needs sage.graphs
Traceback (most recent call last):
... tester.assertTrue(Ti(Ti(x,i,-q2),i,-q1).is_zero()) ...
AssertionError: False is not true
```

```
>>> from sage.all import *
>>> cartan_type = CartanType(["BC",Integer(1),Integer(2)])
>>> KL = RootSystem(cartan_type).weight_lattice().algebra(K)
>>> T = KL.twisted_demazure_lusztig_operators(q1,q2, convention=
↪'dominant')
>>> T._test_relations()                                           #␣
↪needs sage.graphs
Traceback (most recent call last):
... tester.assertTrue(Ti(Ti(x,i,-q2),i,-q1).is_zero()) ...
AssertionError: False is not true
```

Comparison with T0:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: e = L.basis()
sage: K = QQ['t,q'].fraction_field()
sage: t,q = K.gens()
sage: q1 = t
sage: q2 = -1
sage: KL = L.algebra(K)
sage: L0 = L.classical()
sage: T = KL.demazure_lusztig_operators(q1,q2, convention='dominant')
sage: def T0(*l0): return KL.q_project(T[0].on_basis()(L.embed_at_
↪level(L0(l0), 1)), q)
sage: T0_check_on_basis = KL.T0_check_on_basis(q1, q2,                #␣
↪needs sage.graphs
....:                                          convention='dominant')
sage: def T0c(*l0): return T0_check_on_basis(L0(l0))

sage: T0(0,0,1)                                   # not double checked    #␣
↪needs sage.graphs
-((t-1)/q)*B[(1, 0, 0)] + 1/q^2*B[(2, 0, -1)]
sage: T0c(0,0,1)                                                      #␣
↪needs sage.graphs
(t^2-t)*B[(1, 0, 0)] + (t^2-t)*B[(1, 1, -1)] + t^2*B[(2, 0, -1)] + (t-
↪1)*B[(0, 0, 1)]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> e = L.basis()
>>> K = QQ['t,q'].fraction_field()
>>> t,q = K.gens()
>>> q1 = t
>>> q2 = -Integer(1)
>>> KL = L.algebra(K)
>>> L0 = L.classical()
>>> T = KL.demazure_lusztig_operators(q1,q2, convention='dominant')
>>> def T0(*l0): return KL.q_project(T[Integer(0)].on_basis()(L.embed_at_
↪level(L0(l0), Integer(1))), q)
>>> T0_check_on_basis = KL.T0_check_on_basis(q1, q2,                  #␣
↪needs sage.graphs
...                                          convention='dominant')
```

(continues on next page)

(continued from previous page)

```
>>> def T0c(*l0): return T0_check_on_basis(L0(l0))

>>> T0(Integer(0),Integer(0),Integer(1))
↪# not double checked     # needs sage.graphs
-((t-1)/q)*B[(1, 0, 0)] + 1/q^2*B[(2, 0, -1)]
>>> T0c(Integer(0),Integer(0),Integer(1))                                                ␣
↪                              # needs sage.graphs
(t^2-t)*B[(1, 0, 0)] + (t^2-t)*B[(1, 1, -1)] + t^2*B[(2, 0, -1)] + (t-
↪1)*B[(0, 0, 1)]
```

5.1.239 Root lattice realizations

class sage.combinat.root_system.root_lattice_realizations.**RootLatticeRealizations**(*base*, *name=None*)

Bases: `Category_over_base_ring`

The category of root lattice realizations over a given base ring.

A *root lattice realization* L over a base ring R is a free module (or vector space if R is a field) endowed with an embedding of the root lattice of some root system.

Typical root lattice realizations over $\mathbf{Z}$ include the root lattice, weight lattice, and ambient lattice. Typical root lattice realizations over $\mathbf{Q}$ include the root space, weight space, and ambient space.

To describe the embedding, a root lattice realization must implement a method `simple_root()` returning for each i in the index set the image of the simple root α_i under the embedding.

A root lattice realization must further implement a method on elements `scalar()`, computing the scalar product with elements of the coroot lattice or coroot space.

Using those, this category provides tools for reflections, roots, the Weyl group and its action, …

See also

- *RootSystem*
- *WeightLatticeRealizations*
- *RootSpace*
- *WeightSpace*
- *AmbientSpace*

EXAMPLES:

Here, we consider the root system of type A_7, and embed the root lattice element $x = \alpha_2 + 2\alpha_6$ in several root lattice realizations:

```
sage: R = RootSystem(["A",7])
sage: alpha = R.root_lattice().simple_roots()
sage: x = alpha[2] + 2 * alpha[5]

sage: L = R.root_space()
sage: L(x)
```

(continues on next page)

(continued from previous page)

```
alpha[2] + 2*alpha[5]

sage: L = R.weight_lattice()
sage: L(x)                                                                    #␣
↪needs sage.graphs
-Lambda[1] + 2*Lambda[2] - Lambda[3] - 2*Lambda[4] + 4*Lambda[5] - 2*Lambda[6]

sage: L = R.ambient_space()
sage: L(x)
(0, 1, -1, 0, 2, -2, 0, 0)
```

```
>>> from sage.all import *
>>> R = RootSystem(["A",Integer(7)])
>>> alpha = R.root_lattice().simple_roots()
>>> x = alpha[Integer(2)] + Integer(2) * alpha[Integer(5)]

>>> L = R.root_space()
>>> L(x)
alpha[2] + 2*alpha[5]

>>> L = R.weight_lattice()
>>> L(x)                                                                    #␣
↪needs sage.graphs
-Lambda[1] + 2*Lambda[2] - Lambda[3] - 2*Lambda[4] + 4*Lambda[5] - 2*Lambda[6]

>>> L = R.ambient_space()
>>> L(x)
(0, 1, -1, 0, 2, -2, 0, 0)
```

We embed the root space element $x = \alpha_2 + 1/2\alpha_6$ in several root lattice realizations:

```
sage: alpha = R.root_space().simple_roots()
sage: x = alpha[2] + 1/2 * alpha[5]

sage: L = R.weight_space()
sage: L(x)                                                                    #␣
↪needs sage.graphs
-Lambda[1] + 2*Lambda[2] - Lambda[3] - 1/2*Lambda[4] + Lambda[5] - 1/2*Lambda[6]

sage: L = R.ambient_space()
sage: L(x)
(0, 1, -1, 0, 1/2, -1/2, 0, 0)
```

```
>>> from sage.all import *
>>> alpha = R.root_space().simple_roots()
>>> x = alpha[Integer(2)] + Integer(1)/Integer(2) * alpha[Integer(5)]

>>> L = R.weight_space()
>>> L(x)                                                                    #␣
↪needs sage.graphs
-Lambda[1] + 2*Lambda[2] - Lambda[3] - 1/2*Lambda[4] + Lambda[5] - 1/2*Lambda[6]
```

(continues on next page)

(continued from previous page)

```
>>> L = R.ambient_space()
>>> L(x)
(0, 1, -1, 0, 1/2, -1/2, 0, 0)
```

Of course, one can't embed the root space in the weight lattice:

```
sage: L = R.weight_lattice()
sage: L(x)
Traceback (most recent call last):
...
TypeError: do not know how to make x (= alpha[2] + 1/2*alpha[5])
an element of self (=Weight lattice of the Root system of type ['A', 7])
```

```
>>> from sage.all import *
>>> L = R.weight_lattice()
>>> L(x)
Traceback (most recent call last):
...
TypeError: do not know how to make x (= alpha[2] + 1/2*alpha[5])
an element of self (=Weight lattice of the Root system of type ['A', 7])
```

If K_1 is a subring of K_2, then one could in theory have an embedding from the root space over K_1 to any root lattice realization over K_2; this is not implemented:

```
sage: K1 = QQ
sage: K2 = QQ['q']
sage: L = R.weight_space(K2)

sage: alpha = R.root_space(K2).simple_roots()
sage: L(alpha[1])                                                    #␣
↪needs sage.graphs
2*Lambda[1] - Lambda[2]

sage: alpha = R.root_space(K1).simple_roots()
sage: L(alpha[1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= alpha[1]) an element of self
(=Weight space over the Univariate Polynomial Ring in q
over Rational Field of the Root system of type ['A', 7])
```

```
>>> from sage.all import *
>>> K1 = QQ
>>> K2 = QQ['q']
>>> L = R.weight_space(K2)

>>> alpha = R.root_space(K2).simple_roots()
>>> L(alpha[Integer(1)])                                             ␣
↪     # needs sage.graphs
2*Lambda[1] - Lambda[2]

>>> alpha = R.root_space(K1).simple_roots()
```

(continues on next page)

(continued from previous page)

```
>>> L(alpha[Integer(1)])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= alpha[1]) an element of self
(=Weight space over the Univariate Polynomial Ring in q
over Rational Field of the Root system of type ['A', 7])
```

By a slight abuse, the embedding of the root lattice is not actually required to be faithful. Typically for an affine root system, the null root of the root lattice is killed in the non extended weight lattice:

```
sage: R = RootSystem(["A", 3, 1])
sage: delta = R.root_lattice().null_root()                                    #␣
↪needs sage.graphs
sage: L = R.weight_lattice()
sage: L(delta)                                                                #␣
↪needs sage.graphs
0
```

```
>>> from sage.all import *
>>> R = RootSystem(["A", Integer(3), Integer(1)])
>>> delta = R.root_lattice().null_root()                                    #␣
↪needs sage.graphs
>>> L = R.weight_lattice()
>>> L(delta)                                                                #␣
↪needs sage.graphs
0
```

Algebras

alias of *Algebras*

class ElementMethods

Bases: `object`

affine_orbit()

The orbit of `self` under the dot or affine action of the Weyl group.

EXAMPLES:

```
sage: L = RootSystem(['A', 2]).ambient_lattice()
sage: sorted(L.rho().dot_orbit())                   # the output order␣
↪is not specified                # needs sage.graphs
[(-2, 1, 4), (-2, 3, 2), (2, -1, 2),
 (2, 1, 0), (0, -1, 4), (0, 3, 0)]

sage: L = RootSystem(['B',2]).weight_lattice()
sage: sorted(L.fundamental_weights()[1].dot_orbit())    # the output order␣
↪is not specified                # needs sage.graphs
[-4*Lambda[1], -4*Lambda[1] + 4*Lambda[2],
 -3*Lambda[1] - 2*Lambda[2], -3*Lambda[1] + 4*Lambda[2],
 Lambda[1], Lambda[1] - 6*Lambda[2],
 2*Lambda[1] - 6*Lambda[2], 2*Lambda[1] - 2*Lambda[2]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', Integer(2)]).ambient_lattice()
>>> sorted(L.rho().dot_orbit())                          # the output order␣
↪is not specified                # needs sage.graphs
[(-2, 1, 4), (-2, 3, 2), (2, -1, 2),
 (2, 1, 0), (0, -1, 4), (0, 3, 0)]

>>> L = RootSystem(['B',Integer(2)]).weight_lattice()
>>> sorted(L.fundamental_weights()[Integer(1)].dot_orbit())    # the␣
↪output order is not specified             # needs sage.graphs
[-4*Lambda[1], -4*Lambda[1] + 4*Lambda[2],
 -3*Lambda[1] - 2*Lambda[2], -3*Lambda[1] + 4*Lambda[2],
 Lambda[1], Lambda[1] - 6*Lambda[2],
 2*Lambda[1] - 6*Lambda[2], 2*Lambda[1] - 2*Lambda[2]]
```

We compare the dot action orbit to the regular orbit:

```
sage: # needs sage.graphs
sage: L = RootSystem(['A', 3]).weight_lattice()
sage: len(L.rho().dot_orbit())
24
sage: len((-L.rho()).dot_orbit())
1
sage: La = L.fundamental_weights()
sage: len(La[1].dot_orbit())
24
sage: len(La[1].orbit())
4
sage: len((-L.rho() + La[1]).dot_orbit())
4
sage: len(La[2].dot_orbit())
24
sage: len(La[2].orbit())
6
sage: len((-L.rho() + La[2]).dot_orbit())
6
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> L = RootSystem(['A', Integer(3)]).weight_lattice()
>>> len(L.rho().dot_orbit())
24
>>> len((-L.rho()).dot_orbit())
1
>>> La = L.fundamental_weights()
>>> len(La[Integer(1)].dot_orbit())
24
>>> len(La[Integer(1)].orbit())
4
>>> len((-L.rho() + La[Integer(1)]).dot_orbit())
4
>>> len(La[Integer(2)].dot_orbit())
24
```

(continues on next page)

(continued from previous page)

```
>>> len(La[Integer(2)].orbit())
6
>>> len((-L.rho() + La[Integer(2)]).dot_orbit())
6
```

associated_coroot()

Return the coroot associated to this root.

EXAMPLES:

```
sage: alpha = RootSystem(["A", 3]).root_space().simple_roots()
sage: alpha[1].associated_coroot()                                          #␣
↪needs sage.graphs
alphacheck[1]
```

```
>>> from sage.all import *
>>> alpha = RootSystem(["A", Integer(3)]).root_space().simple_roots()
>>> alpha[Integer(1)].associated_coroot()                                  ␣
↪     # needs sage.graphs
alphacheck[1]
```

associated_reflection()

Given a positive root `self`, return a reduced word for the reflection orthogonal to `self`.

Since the answer is cached, it is a tuple instead of a list.

EXAMPLES:

```
sage: C3_rl = RootSystem(['C',3]).root_lattice()                           #␣
↪needs sage.graphs
sage: C3_rl.simple_root(3).weyl_action([1,2]).associated_reflection()     #␣
↪needs sage.graphs
(1, 2, 3, 2, 1)
sage: C3_rl.simple_root(2).associated_reflection()                         #␣
↪needs sage.graphs
(2,)
```

```
>>> from sage.all import *
>>> C3_rl = RootSystem(['C',Integer(3)]).root_lattice()                    ␣
↪     # needs sage.graphs
>>> C3_rl.simple_root(Integer(3)).weyl_action([Integer(1),Integer(2)]).
↪associated_reflection()    # needs sage.graphs
(1, 2, 3, 2, 1)
>>> C3_rl.simple_root(Integer(2)).associated_reflection()                  ␣
↪     # needs sage.graphs
(2,)
```

descents(*index_set=None*, *positive=False*)

Return the descents of pt.

EXAMPLES:

```
sage: space=RootSystem(['A',5]).weight_space()
sage: alpha = space.simple_roots()                                      #␣
↪needs sage.graphs
sage: (alpha[1]+alpha[2]+alpha[4]).descents()                          #␣
↪needs sage.graphs
[3, 5]
```

```
>>> from sage.all import *
>>> space=RootSystem(['A',Integer(5)]).weight_space()
>>> alpha = space.simple_roots()                                     #␣
↪needs sage.graphs
>>> (alpha[Integer(1)]+alpha[Integer(2)]+alpha[Integer(4)]).descents()   ␣
↪                          # needs sage.graphs
[3, 5]
```

dot_action(*w*, *inverse=False*)

Act on `self` by `w` using the dot or affine action.

Let w be an element of the Weyl group. The *dot action* or *affine action* is given by:

$$w \bullet \lambda = w(\lambda + \rho) - \rho,$$

where ρ is the sum of the fundamental weights.

INPUT:

- `w` – an element of a Coxeter or Weyl group of the same Cartan type, or a tuple or a list (such as a reduced word) of elements from the index set
- `inverse` – boolean (default: `False`); whether to act by the inverse element

EXAMPLES:

```
sage: P = RootSystem(['B',3]).weight_lattice()
sage: La = P.fundamental_weights()
sage: mu = La[1] + 2*La[2] - 3*La[3]
sage: mu.dot_action([1])                                                #␣
↪needs sage.graphs
-3*Lambda[1] + 4*Lambda[2] - 3*Lambda[3]
sage: mu.dot_action([3])                                                #␣
↪needs sage.graphs
Lambda[1] + Lambda[3]
sage: mu.dot_action([1,2,3])                                            #␣
↪needs sage.graphs
-4*Lambda[1] + Lambda[2] + 3*Lambda[3]
```

```
>>> from sage.all import *
>>> P = RootSystem(['B',Integer(3)]).weight_lattice()
>>> La = P.fundamental_weights()
>>> mu = La[Integer(1)] + Integer(2)*La[Integer(2)] -␣
↪Integer(3)*La[Integer(3)]
>>> mu.dot_action([Integer(1)])                                          ␣
↪      # needs sage.graphs
-3*Lambda[1] + 4*Lambda[2] - 3*Lambda[3]
>>> mu.dot_action([Integer(3)])                                          ␣
↪      # needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
Lambda[1] + Lambda[3]
>>> mu.dot_action([Integer(1),Integer(2),Integer(3)])                          ␣
↪                       # needs sage.graphs
-4*Lambda[1] + Lambda[2] + 3*Lambda[3]
```

We check that the origin of this action is at $-\rho$:

```
sage: all((-P.rho()).dot_action([i]) == -P.rho()                            #␣
↪needs sage.graphs
....:     for i in P.index_set())
True
```

```
>>> from sage.all import *
>>> all((-P.rho()).dot_action([i]) == -P.rho()                            #␣
↪needs sage.graphs
...     for i in P.index_set())
True
```

REFERENCES:

- Wikipedia article Affine_action

dot_orbit()

The orbit of `self` under the dot or affine action of the Weyl group.

EXAMPLES:

```
sage: L = RootSystem(['A', 2]).ambient_lattice()
sage: sorted(L.rho().dot_orbit())                        # the output order␣
↪is not specified              # needs sage.graphs
[(-2, 1, 4), (-2, 3, 2), (2, -1, 2),
 (2, 1, 0), (0, -1, 4), (0, 3, 0)]

sage: L = RootSystem(['B',2]).weight_lattice()
sage: sorted(L.fundamental_weights()[1].dot_orbit())    # the output order␣
↪is not specified              # needs sage.graphs
[-4*Lambda[1], -4*Lambda[1] + 4*Lambda[2],
 -3*Lambda[1] - 2*Lambda[2], -3*Lambda[1] + 4*Lambda[2],
 Lambda[1], Lambda[1] - 6*Lambda[2],
 2*Lambda[1] - 6*Lambda[2], 2*Lambda[1] - 2*Lambda[2]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', Integer(2)]).ambient_lattice()
>>> sorted(L.rho().dot_orbit())                        # the output order␣
↪is not specified              # needs sage.graphs
[(-2, 1, 4), (-2, 3, 2), (2, -1, 2),
 (2, 1, 0), (0, -1, 4), (0, 3, 0)]

>>> L = RootSystem(['B',Integer(2)]).weight_lattice()
>>> sorted(L.fundamental_weights()[Integer(1)].dot_orbit())    # the␣
↪output order is not specified              # needs sage.graphs
[-4*Lambda[1], -4*Lambda[1] + 4*Lambda[2],
 -3*Lambda[1] - 2*Lambda[2], -3*Lambda[1] + 4*Lambda[2],
```

(continues on next page)

(continued from previous page)

```
Lambda[1], Lambda[1] - 6*Lambda[2],
2*Lambda[1] - 6*Lambda[2], 2*Lambda[1] - 2*Lambda[2]]
```

We compare the dot action orbit to the regular orbit:

```
sage: # needs sage.graphs
sage: L = RootSystem(['A', 3]).weight_lattice()
sage: len(L.rho().dot_orbit())
24
sage: len((-L.rho()).dot_orbit())
1
sage: La = L.fundamental_weights()
sage: len(La[1].dot_orbit())
24
sage: len(La[1].orbit())
4
sage: len((-L.rho() + La[1]).dot_orbit())
4
sage: len(La[2].dot_orbit())
24
sage: len(La[2].orbit())
6
sage: len((-L.rho() + La[2]).dot_orbit())
6
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> L = RootSystem(['A', Integer(3)]).weight_lattice()
>>> len(L.rho().dot_orbit())
24
>>> len((-L.rho()).dot_orbit())
1
>>> La = L.fundamental_weights()
>>> len(La[Integer(1)].dot_orbit())
24
>>> len(La[Integer(1)].orbit())
4
>>> len((-L.rho() + La[Integer(1)]).dot_orbit())
4
>>> len(La[Integer(2)].dot_orbit())
24
>>> len(La[Integer(2)].orbit())
6
>>> len((-L.rho() + La[Integer(2)]).dot_orbit())
6
```

extraspecial_pair()

Return the extraspecial pair of `self` under the ordering defined by *positive_roots_by_height()*.

The *extraspecial pair* of a positive root γ with some total ordering $<$ of the root lattice that respects height is the pair of positive roots (α, β) such that $\gamma = \alpha + \beta$ and α is as small as possible.

EXAMPLES:

```
sage: Q = RootSystem(['G', 2]).root_lattice()
sage: Q.highest_root().extraspecial_pair()                                  #␣
↪needs sage.graphs
(alpha[2], 3*alpha[1] + alpha[2])
```

```
>>> from sage.all import *
>>> Q = RootSystem(['G', Integer(2)]).root_lattice()
>>> Q.highest_root().extraspecial_pair()                                    #␣
↪needs sage.graphs
(alpha[2], 3*alpha[1] + alpha[2])
```

first_descent(*index_set=None*, *positive=False*)

Return the first descent of pt.

One can use the `index_set` option to restrict to the parabolic subgroup indexed by `index_set`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: space = RootSystem(['A',5]).weight_space()
sage: alpha = space.simple_roots()
sage: (alpha[1]+alpha[2]+alpha[4]).first_descent()
3
sage: (alpha[1]+alpha[2]+alpha[4]).first_descent([1,2,5])
5
sage: (alpha[1]+alpha[2]+alpha[4]).first_descent([1,2,5,3,4])
5
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> space = RootSystem(['A',Integer(5)]).weight_space()
>>> alpha = space.simple_roots()
>>> (alpha[Integer(1)]+alpha[Integer(2)]+alpha[Integer(4)]).first_
↪descent()
3
>>> (alpha[Integer(1)]+alpha[Integer(2)]+alpha[Integer(4)]).first_
↪descent([Integer(1),Integer(2),Integer(5)])
5
>>> (alpha[Integer(1)]+alpha[Integer(2)]+alpha[Integer(4)]).first_
↪descent([Integer(1),Integer(2),Integer(5),Integer(3),Integer(4)])
5
```

greater()

Return the elements in the orbit of `self` which are greater than `self` in the weak order.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).ambient_lattice()
sage: e = L.basis()
sage: e[2].greater()
[(0, 0, 1, 0), (0, 0, 0, 1)]
sage: len(L.rho().greater())
24
sage: len((-L.rho()).greater())
```

(continues on next page)

(continued from previous page)

```
1
sage: sorted([len(x.greater()) for x in L.rho().orbit()])
[1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6, 6, 6, 8, 8, 8, 8, 12, 12, 12,␣
↪24]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(3)]).ambient_lattice()
>>> e = L.basis()
>>> e[Integer(2)].greater()
[(0, 0, 1, 0), (0, 0, 0, 1)]
>>> len(L.rho().greater())
24
>>> len((-L.rho()).greater())
1
>>> sorted([len(x.greater()) for x in L.rho().orbit()])
[1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6, 6, 6, 8, 8, 8, 8, 12, 12, 12,␣
↪24]
```

has_descent(*i*, *positive=False*)

Test if `self` has a descent at position i, that is, if `self` is on the strict negative side of the i-th simple reflection hyperplane.

If positive is `True`, tests if it is on the strict positive side instead.

EXAMPLES:

```
sage: # needs sage.graphs
sage: space = RootSystem(['A',5]).weight_space()
sage: alpha = RootSystem(['A',5]).weight_space().simple_roots()
sage: [alpha[i].has_descent(1) for i in space.index_set()]
[False, True, False, False, False]
sage: [(-alpha[i]).has_descent(1) for i in space.index_set()]
[True, False, False, False, False]
sage: [alpha[i].has_descent(1, True) for i in space.index_set()]
[True, False, False, False, False]
sage: [(-alpha[i]).has_descent(1, True) for i in space.index_set()]
[False, True, False, False, False]
sage: (alpha[1]+alpha[2]+alpha[4]).has_descent(3)
True
sage: (alpha[1]+alpha[2]+alpha[4]).has_descent(1)
False
sage: (alpha[1]+alpha[2]+alpha[4]).has_descent(1, True)
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> space = RootSystem(['A',Integer(5)]).weight_space()
>>> alpha = RootSystem(['A',Integer(5)]).weight_space().simple_roots()
>>> [alpha[i].has_descent(Integer(1)) for i in space.index_set()]
[False, True, False, False, False]
>>> [(-alpha[i]).has_descent(Integer(1)) for i in space.index_set()]
[True, False, False, False, False]
```

(continues on next page)

(continued from previous page)

```
>>> [alpha[i].has_descent(Integer(1), True) for i in space.index_set()]
[True, False, False, False, False]
>>> [(-alpha[i]).has_descent(Integer(1), True) for i in space.index_set()]
[False, True, False, False, False]
>>> (alpha[Integer(1)]+alpha[Integer(2)]+alpha[Integer(4)]).has_
↪descent(Integer(3))
True
>>> (alpha[Integer(1)]+alpha[Integer(2)]+alpha[Integer(4)]).has_
↪descent(Integer(1))
False
>>> (alpha[Integer(1)]+alpha[Integer(2)]+alpha[Integer(4)]).has_
↪descent(Integer(1), True)
True
```

height()

Return the height of `self`.

The height of a root $\alpha = \sum_i a_i \alpha_i$ is defined to be $h(\alpha) := \sum_i a_i$.

EXAMPLES:

```
sage: Q = RootSystem(['G', 2]).root_lattice()
sage: Q.highest_root().height()                                          #␣
↪needs sage.graphs
5
```

```
>>> from sage.all import *
>>> Q = RootSystem(['G', Integer(2)]).root_lattice()
>>> Q.highest_root().height()                                          #␣
↪needs sage.graphs
5
```

is_dominant(*index_set=None*, *positive=True*)

Return whether `self` is dominant.

This is done with respect to the sub–root system indicated by the subset of Dynkin nodes `index_set`. If `index_set` is `None`, then the entire Dynkin node set is used. If positive is `False`, then the dominance condition is replaced by antidominance.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).ambient_lattice()
sage: Lambda = L.fundamental_weights()
sage: [x.is_dominant() for x in Lambda]
[True, True]
sage: [x.is_dominant(positive=False) for x in Lambda]
[False, False]
sage: (Lambda[1]-Lambda[2]).is_dominant()
False
sage: (-Lambda[1]+Lambda[2]).is_dominant()
False
sage: (Lambda[1]-Lambda[2]).is_dominant([1])
True
sage: (Lambda[1]-Lambda[2]).is_dominant([2])
```

(continues on next page)

(continued from previous page)

```
False
sage: [x.is_dominant() for x in L.roots()]
[False, True, False, False, False, False]
sage: [x.is_dominant(positive=False) for x in L.roots()]
[False, False, False, False, True, False]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2)]).ambient_lattice()
>>> Lambda = L.fundamental_weights()
>>> [x.is_dominant() for x in Lambda]
[True, True]
>>> [x.is_dominant(positive=False) for x in Lambda]
[False, False]
>>> (Lambda[Integer(1)]-Lambda[Integer(2)]).is_dominant()
False
>>> (-Lambda[Integer(1)]+Lambda[Integer(2)]).is_dominant()
False
>>> (Lambda[Integer(1)]-Lambda[Integer(2)]).is_dominant([Integer(1)])
True
>>> (Lambda[Integer(1)]-Lambda[Integer(2)]).is_dominant([Integer(2)])
False
>>> [x.is_dominant() for x in L.roots()]
[False, True, False, False, False, False]
>>> [x.is_dominant(positive=False) for x in L.roots()]
[False, False, False, False, True, False]
```

is_dominant_weight()

Test whether `self` is a dominant element of the weight lattice.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).ambient_lattice()
sage: Lambda = L.fundamental_weights()
sage: [x.is_dominant() for x in Lambda]
[True, True]
sage: (3*Lambda[1]+Lambda[2]).is_dominant()
True
sage: (Lambda[1]-Lambda[2]).is_dominant()
False
sage: (-Lambda[1]+Lambda[2]).is_dominant()
False
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2)]).ambient_lattice()
>>> Lambda = L.fundamental_weights()
>>> [x.is_dominant() for x in Lambda]
[True, True]
>>> (Integer(3)*Lambda[Integer(1)]+Lambda[Integer(2)]).is_dominant()
True
>>> (Lambda[Integer(1)]-Lambda[Integer(2)]).is_dominant()
False
>>> (-Lambda[Integer(1)]+Lambda[Integer(2)]).is_dominant()
```

(continues on next page)

(continued from previous page)

```
False
```

Tests that the scalar products with the coroots are all nonnegative integers. For example, if x is the sum of a dominant element of the weight lattice plus some other element orthogonal to all coroots, then the implementation correctly reports x to be a dominant weight:

```
sage: x = Lambda[1] + L([-1,-1,-1])
sage: x.is_dominant_weight()
True
```

```
>>> from sage.all import *
>>> x = Lambda[Integer(1)] + L([-Integer(1),-Integer(1),-Integer(1)])
>>> x.is_dominant_weight()
True
```

is_imaginary_root()

Return `True` if `self` is an imaginary root.

A root α is imaginary if it is not W-conjugate to a simple root where W is the corresponding Weyl group.

EXAMPLES:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[0].is_imaginary_root()                                          #␣
↪needs sage.graphs
False
sage: elt = alpha[0] + alpha[1] + 2*alpha[2]
sage: elt.is_imaginary_root()                                               #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> Q = RootSystem(['B',Integer(2),Integer(1)]).root_lattice()
>>> alpha = Q.simple_roots()
>>> alpha[Integer(0)].is_imaginary_root()                                    ␣
↪     # needs sage.graphs
False
>>> elt = alpha[Integer(0)] + alpha[Integer(1)] +␣
↪Integer(2)*alpha[Integer(2)]
>>> elt.is_imaginary_root()                                                #␣
↪needs sage.graphs
True
```

is_long_root()

Return `True` if `self` is a long (real) root.

EXAMPLES:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[0].is_long_root()                                               #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
True
sage: alpha[1].is_long_root()                                             #␣
↪needs sage.graphs
True
sage: alpha[2].is_long_root()                                             #␣
↪needs sage.graphs
False
```

```
>>> from sage.all import *
>>> Q = RootSystem(['B',Integer(2),Integer(1)]).root_lattice()
>>> alpha = Q.simple_roots()
>>> alpha[Integer(0)].is_long_root()                                      ␣
↪      # needs sage.graphs
True
>>> alpha[Integer(1)].is_long_root()                                      ␣
↪      # needs sage.graphs
True
>>> alpha[Integer(2)].is_long_root()                                      ␣
↪      # needs sage.graphs
False
```

is_parabolic_root(*index_set*)

Return whether `root` is in the parabolic subsystem with Dynkin nodes `index_set`.

This assumes that `self` is a root.

INPUT:

- `index_set` – the Dynkin node set of the parabolic subsystem

Todo

This implementation is only valid in the root or weight lattice

EXAMPLES:

```
sage: alpha = RootSystem(['A',3]).root_lattice().from_vector(vector([1,1,
↪0]))
sage: alpha.is_parabolic_root([1,3])
False
sage: alpha.is_parabolic_root([1,2])
True
sage: alpha.is_parabolic_root([2])
False
```

```
>>> from sage.all import *
>>> alpha = RootSystem(['A',Integer(3)]).root_lattice().from_
↪vector(vector([Integer(1),Integer(1),Integer(0)]))
>>> alpha.is_parabolic_root([Integer(1),Integer(3)])
False
>>> alpha.is_parabolic_root([Integer(1),Integer(2)])
True
```

(continues on next page)

(continued from previous page)

```
>>> alpha.is_parabolic_root([Integer(2)])
False
```

is_real_root()

Return `True` if `self` is a real root.

A root α is real if it is W-conjugate to a simple root where W is the corresponding Weyl group.

EXAMPLES:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[0].is_real_root()                                          #␣
↪needs sage.graphs
True
sage: elt = alpha[0] + alpha[1] + 2*alpha[2]
sage: elt.is_real_root()                                               #␣
↪needs sage.graphs
False
```

```
>>> from sage.all import *
>>> Q = RootSystem(['B',Integer(2),Integer(1)]).root_lattice()
>>> alpha = Q.simple_roots()
>>> alpha[Integer(0)].is_real_root()                                   ␣
↪     # needs sage.graphs
True
>>> elt = alpha[Integer(0)] + alpha[Integer(1)] +␣
↪Integer(2)*alpha[Integer(2)]
>>> elt.is_real_root()                                               #␣
↪needs sage.graphs
False
```

is_short_root()

Return `True` if `self` is a short (real) root.

Returns `False` unless the parent is an irreducible root system of finite type having two root lengths and `self` is of the shorter length. There is no check of whether `self` is actually a root.

EXAMPLES:

```
sage: # needs sage.graphs
sage: Q = RootSystem(['C',2]).root_lattice()
sage: al = Q.simple_root(1).weyl_action([1,2]); al
alpha[1] + alpha[2]
sage: al.is_short_root()
True
sage: bt = Q.simple_root(2).weyl_action([2,1,2]); bt
-2*alpha[1] - alpha[2]
sage: bt.is_short_root()
False
sage: RootSystem(['A',2]).root_lattice().simple_root(1).is_short_root()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> Q = RootSystem(['C',Integer(2)]).root_lattice()
>>> al = Q.simple_root(Integer(1)).weyl_action([Integer(1),Integer(2)]);␣
↪al
alpha[1] + alpha[2]
>>> al.is_short_root()
True
>>> bt = Q.simple_root(Integer(2)).weyl_action([Integer(2),Integer(1),
↪Integer(2)]); bt
-2*alpha[1] - alpha[2]
>>> bt.is_short_root()
False
>>> RootSystem(['A',Integer(2)]).root_lattice().simple_root(Integer(1)).
↪is_short_root()
False
```

An example in affine type:

```
sage: # needs sage.graphs
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[0].is_short_root()
False
sage: alpha[1].is_short_root()
False
sage: alpha[2].is_short_root()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> Q = RootSystem(['B',Integer(2),Integer(1)]).root_lattice()
>>> alpha = Q.simple_roots()
>>> alpha[Integer(0)].is_short_root()
False
>>> alpha[Integer(1)].is_short_root()
False
>>> alpha[Integer(2)].is_short_root()
True
```

is_verma_dominant(*positive=True*)

Return if `self` is Verma dominant.

A weight λ is *Verma dominant* if

$$\langle \lambda + \rho, \alpha^\vee \rangle \notin \mathbf{Z}_{<0}$$

for all positive roots α. Note that begin Verma dominant does *not* imply that $\langle \lambda + \rho, \alpha^\vee \rangle \geq 0$ for any positive root α. This is used to determine if a Verma module is simple or projective.

INPUT:

- `positive` – boolean (default: `True`); if `False`, then this checks if the weight is Verma anti-dominant, where $\mathbf{Z}_{<0}$ is replaced with $\mathbf{Z}_{>0}$ in the definition.

EXAMPLES:

```
sage: P = RootSystem(['A', 3]).weight_space()
sage: La = P.fundamental_weights()
sage: alphacheck = P.coroot_lattice().positive_roots()
sage: rho = P.rho()
sage: (La[1] + 2*La[2]).is_verma_dominant()
True
sage: la = La[1] - 3/2*La[3] - rho
sage: la.is_verma_dominant()
True
sage: la.is_verma_dominant(positive=False)
False
sage: [(la+rho).scalar(coroot) for coroot in alphacheck]
[1, 0, -3/2, 1, -3/2, -1/2]
sage: mu = 1/2*La[1] - 3/2*La[3] - rho
sage: mu.is_verma_dominant()
False
sage: mu.is_verma_dominant(positive=False)
True
sage: [(mu+rho).scalar(coroot) for coroot in alphacheck]
[1/2, 0, -3/2, 1/2, -3/2, -1]
```

```
>>> from sage.all import *
>>> P = RootSystem(['A', Integer(3)]).weight_space()
>>> La = P.fundamental_weights()
>>> alphacheck = P.coroot_lattice().positive_roots()
>>> rho = P.rho()
>>> (La[Integer(1)] + Integer(2)*La[Integer(2)]).is_verma_dominant()
True
>>> la = La[Integer(1)] - Integer(3)/Integer(2)*La[Integer(3)] - rho
>>> la.is_verma_dominant()
True
>>> la.is_verma_dominant(positive=False)
False
>>> [(la+rho).scalar(coroot) for coroot in alphacheck]
[1, 0, -3/2, 1, -3/2, -1/2]
>>> mu = Integer(1)/Integer(2)*La[Integer(1)] - Integer(3)/
↪Integer(2)*La[Integer(3)] - rho
>>> mu.is_verma_dominant()
False
>>> mu.is_verma_dominant(positive=False)
True
>>> [(mu+rho).scalar(coroot) for coroot in alphacheck]
[1/2, 0, -3/2, 1/2, -3/2, -1]
```

level()

EXAMPLES:

```
sage: L = RootSystem(['A',2,1]).weight_lattice()
sage: L.rho().level()                                                   #␣
↪needs sage.graphs
3
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice()
>>> L.rho().level()                                                         #␣
↪needs sage.graphs
3
```

norm_squared()

Return the norm squared of `self` with respect to the symmetric form.

EXAMPLES:

```
sage: # needs sage.graphs
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[1].norm_squared()
4
sage: alpha[2].norm_squared()
2
sage: elt = alpha[0] - 3*alpha[1] + alpha[2]
sage: elt.norm_squared()
50
sage: elt = alpha[0] + alpha[1] + 2*alpha[2]
sage: elt.norm_squared()
0

sage: Q = RootSystem(CartanType(['A',4,2]).dual()).root_lattice()
sage: Qc = RootSystem(['A',4,2]).coroot_lattice()
sage: alpha = Q.simple_roots()
sage: alphac = Qc.simple_roots()
sage: elt = alpha[0] + 2*alpha[1] + 2*alpha[2]
sage: eltc = alphac[0] + 2*alphac[1] + 2*alphac[2]
sage: elt.norm_squared()                                                    #␣
↪needs sage.graphs
0
sage: eltc.norm_squared()                                                   #␣
↪needs sage.graphs
0
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> Q = RootSystem(['B',Integer(2),Integer(1)]).root_lattice()
>>> alpha = Q.simple_roots()
>>> alpha[Integer(1)].norm_squared()
4
>>> alpha[Integer(2)].norm_squared()
2
>>> elt = alpha[Integer(0)] - Integer(3)*alpha[Integer(1)] +␣
↪alpha[Integer(2)]
>>> elt.norm_squared()
50
>>> elt = alpha[Integer(0)] + alpha[Integer(1)] +␣
↪Integer(2)*alpha[Integer(2)]
>>> elt.norm_squared()
```

(continues on next page)

(continued from previous page)

```
0

>>> Q = RootSystem(CartanType(['A',Integer(4),Integer(2)]).dual()).root_
↪lattice()
>>> Qc = RootSystem(['A',Integer(4),Integer(2)]).coroot_lattice()
>>> alpha = Q.simple_roots()
>>> alphac = Qc.simple_roots()
>>> elt = alpha[Integer(0)] + Integer(2)*alpha[Integer(1)] +␣
↪Integer(2)*alpha[Integer(2)]
>>> eltc = alphac[Integer(0)] + Integer(2)*alphac[Integer(1)] +␣
↪Integer(2)*alphac[Integer(2)]
>>> elt.norm_squared()                                                  #␣
↪needs sage.graphs
0
>>> eltc.norm_squared()                                                 #␣
↪needs sage.graphs
0
```

orbit()

The orbit of `self` under the action of the Weyl group.

EXAMPLES:

ρ is a regular element whose orbit is in bijection with the Weyl group. In particular, it has 6 elements for the symmetric group S_3:

```
sage: L = RootSystem(["A", 2]).ambient_lattice()
sage: sorted(L.rho().orbit())               # the output order is not␣
↪specified
[(1, 2, 0), (1, 0, 2), (2, 1, 0),
 (2, 0, 1), (0, 1, 2), (0, 2, 1)]

sage: L = RootSystem(["A", 3]).weight_lattice()
sage: len(L.rho().orbit())                                                #␣
↪needs sage.graphs
24
sage: len(L.fundamental_weights()[1].orbit())                             #␣
↪needs sage.graphs
4
sage: len(L.fundamental_weights()[2].orbit())                             #␣
↪needs sage.graphs
6
```

```
>>> from sage.all import *
>>> L = RootSystem(["A", Integer(2)]).ambient_lattice()
>>> sorted(L.rho().orbit())               # the output order is not␣
↪specified
[(1, 2, 0), (1, 0, 2), (2, 1, 0),
 (2, 0, 1), (0, 1, 2), (0, 2, 1)]

>>> L = RootSystem(["A", Integer(3)]).weight_lattice()
>>> len(L.rho().orbit())                                                #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
24
>>> len(L.fundamental_weights()[Integer(1)].orbit())                    ␣
↪     # needs sage.graphs
4
>>> len(L.fundamental_weights()[Integer(2)].orbit())                    ␣
↪     # needs sage.graphs
6
```

pred(*index_set=None*)

Return the immediate predecessors of `self` for the weak order.

INPUT:

- `index_set` – a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: `None` for all of them)

If `index_set` is specified, the successors for the corresponding parabolic subsystem are returned.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).weight_lattice()
sage: Lambda = L.fundamental_weights()
sage: Lambda[1].pred()
[]
sage: L.rho().pred()
[]
sage: (-L.rho()).pred()                                                 #␣
↪needs sage.graphs
[Lambda[1] - 2*Lambda[2] - Lambda[3],
 -2*Lambda[1] + Lambda[2] - 2*Lambda[3],
 -Lambda[1] - 2*Lambda[2] + Lambda[3]]
sage: (-L.rho()).pred(index_set=[1])                                    #␣
↪needs sage.graphs
[Lambda[1] - 2*Lambda[2] - Lambda[3]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(3)]).weight_lattice()
>>> Lambda = L.fundamental_weights()
>>> Lambda[Integer(1)].pred()
[]
>>> L.rho().pred()
[]
>>> (-L.rho()).pred()                                                 #␣
↪needs sage.graphs
[Lambda[1] - 2*Lambda[2] - Lambda[3],
 -2*Lambda[1] + Lambda[2] - 2*Lambda[3],
 -Lambda[1] - 2*Lambda[2] + Lambda[3]]
>>> (-L.rho()).pred(index_set=[Integer(1)])                             ␣
↪     # needs sage.graphs
[Lambda[1] - 2*Lambda[2] - Lambda[3]]
```

reduced_word(*index_set=None*, *positive=True*)

Return a reduced word for the inverse of the shortest Weyl group element that sends the vector `self` into the dominant chamber.

With the `index_set` optional parameter, this is done with respect to the corresponding parabolic subgroup.

If `positive` is False, use the antidominant chamber instead.

EXAMPLES:

```
sage: space = RootSystem(['A',5]).weight_space()
sage: alpha = RootSystem(['A',5]).weight_space().simple_roots()          #␣
↪needs sage.graphs
sage: alpha[1].reduced_word()                                            #␣
↪needs sage.graphs
[2, 3, 4, 5]
sage: alpha[1].reduced_word([1,2])                                       #␣
↪needs sage.graphs
[2]
```

```
>>> from sage.all import *
>>> space = RootSystem(['A',Integer(5)]).weight_space()
>>> alpha = RootSystem(['A',Integer(5)]).weight_space().simple_roots()     ␣
↪     # needs sage.graphs
>>> alpha[Integer(1)].reduced_word()                                      ␣
↪     # needs sage.graphs
[2, 3, 4, 5]
>>> alpha[Integer(1)].reduced_word([Integer(1),Integer(2)])               ␣
↪                         # needs sage.graphs
[2]
```

reflection(*root*, *use_coroot=False*)

Reflect `self` across the hyperplane orthogonal to `root`.

If `use_coroot` is `True`, `root` is interpreted as a coroot.

EXAMPLES:

```
sage: R = RootSystem(['C',4])
sage: weight_lattice = R.weight_lattice()
sage: mu = weight_lattice.from_vector(vector([0,0,1,2]))
sage: coroot_lattice = R.coroot_lattice()
sage: alphavee = coroot_lattice.from_vector(vector([0,0,1,1]))
sage: mu.reflection(alphavee, use_coroot=True)                           #␣
↪needs sage.graphs
6*Lambda[2] - 5*Lambda[3] + 2*Lambda[4]
sage: root_lattice = R.root_lattice()
sage: beta = root_lattice.from_vector(vector([0,1,1,0]))
sage: mu.reflection(beta)                                                #␣
↪needs sage.graphs
Lambda[1] - Lambda[2] + 3*Lambda[4]
```

```
>>> from sage.all import *
>>> R = RootSystem(['C',Integer(4)])
>>> weight_lattice = R.weight_lattice()
>>> mu = weight_lattice.from_vector(vector([Integer(0),Integer(0),
↪Integer(1),Integer(2)]))
>>> coroot_lattice = R.coroot_lattice()
```

(continues on next page)

(continued from previous page)

```
>>> alphavee = coroot_lattice.from_vector(vector([Integer(0),Integer(0),
↪Integer(1),Integer(1)]))
>>> mu.reflection(alphavee, use_coroot=True)                            #␣
↪needs sage.graphs
6*Lambda[2] - 5*Lambda[3] + 2*Lambda[4]
>>> root_lattice = R.root_lattice()
>>> beta = root_lattice.from_vector(vector([Integer(0),Integer(1),
↪Integer(1),Integer(0)]))
>>> mu.reflection(beta)                                                 #␣
↪needs sage.graphs
Lambda[1] - Lambda[2] + 3*Lambda[4]
```

scalar(*lambdacheck*)

Implement the natural pairing with the coroot lattice.

INPUT:

- `self` – an element of a root lattice realization
- `lambdacheck` – an element of the coroot lattice or coroot space

OUTPUT: the scalar product of `self` and `lambdacheck`

EXAMPLES:

```
sage: L = RootSystem(['A',4]).root_lattice()
sage: alpha      = L.simple_roots()
sage: alphacheck = L.simple_coroots()
sage: alpha[1].scalar(alphacheck[1])                                     #␣
↪needs sage.graphs
2
sage: alpha[1].scalar(alphacheck[2])                                     #␣
↪needs sage.graphs
-1
sage: matrix([ [ alpha[i].scalar(alphacheck[j])                          #␣
↪needs sage.graphs
....:           for i in L.index_set() ]
....:         for j in L.index_set() ])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(4)]).root_lattice()
>>> alpha      = L.simple_roots()
>>> alphacheck = L.simple_coroots()
>>> alpha[Integer(1)].scalar(alphacheck[Integer(1)])                      ␣
↪               # needs sage.graphs
2
>>> alpha[Integer(1)].scalar(alphacheck[Integer(2)])                      ␣
↪               # needs sage.graphs
-1
>>> matrix([ [ alpha[i].scalar(alphacheck[j])                           #␣
↪needs sage.graphs
...           for i in L.index_set() ]
```

(continues on next page)

(continued from previous page)

```
...             for j in L.index_set() ])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

simple_reflection(*i*)

Return the image of `self` by the i-th simple reflection.

EXAMPLES:

```
sage: alpha = RootSystem(["A", 3]).root_lattice().alpha()
sage: alpha[1].simple_reflection(2)                                    #␣
↪needs sage.graphs
alpha[1] + alpha[2]

sage: Q = RootSystem(['A', 3, 1]).weight_lattice(extended=True)
sage: Lambda = Q.fundamental_weights()
sage: L = Lambda[0] + Q.null_root()                                    #␣
↪needs sage.graphs
sage: L.simple_reflection(0)                                           #␣
↪needs sage.graphs
-Lambda[0] + Lambda[1] + Lambda[3]
```

```
>>> from sage.all import *
>>> alpha = RootSystem(["A", Integer(3)]).root_lattice().alpha()
>>> alpha[Integer(1)].simple_reflection(Integer(2))                    ␣
↪               # needs sage.graphs
alpha[1] + alpha[2]

>>> Q = RootSystem(['A', Integer(3), Integer(1)]).weight_
↪lattice(extended=True)
>>> Lambda = Q.fundamental_weights()
>>> L = Lambda[Integer(0)] + Q.null_root()                             ␣
↪      # needs sage.graphs
>>> L.simple_reflection(Integer(0))                                    ␣
↪      # needs sage.graphs
-Lambda[0] + Lambda[1] + Lambda[3]
```

simple_reflections()

The images of `self` by all the simple reflections.

EXAMPLES:

```
sage: alpha = RootSystem(["A", 3]).root_lattice().alpha()
sage: alpha[1].simple_reflections()                                    #␣
↪needs sage.graphs
[-alpha[1], alpha[1] + alpha[2], alpha[1]]
```

```
>>> from sage.all import *
>>> alpha = RootSystem(["A", Integer(3)]).root_lattice().alpha()
>>> alpha[Integer(1)].simple_reflections()                             ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.graphs
[-alpha[1], alpha[1] + alpha[2], alpha[1]]
```

smaller()

Return the elements in the orbit of `self` which are smaller than `self` in the weak order.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).ambient_lattice()
sage: e = L.basis()
sage: e[2].smaller()
[(0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)]
sage: len(L.rho().smaller())
1
sage: len((-L.rho()).smaller())
24
sage: sorted([len(x.smaller()) for x in L.rho().orbit()])
[1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6, 6, 6, 8, 8, 8, 8, 12, 12, 12,␣
↪24]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(3)]).ambient_lattice()
>>> e = L.basis()
>>> e[Integer(2)].smaller()
[(0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)]
>>> len(L.rho().smaller())
1
>>> len((-L.rho()).smaller())
24
>>> sorted([len(x.smaller()) for x in L.rho().orbit()])
[1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6, 6, 6, 8, 8, 8, 8, 12, 12, 12,␣
↪24]
```

succ(*index_set=None*)

Return the immediate successors of `self` for the weak order.

INPUT:

- `index_set` – a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: `None` for all of them)

If `index_set` is specified, the successors for the corresponding parabolic subsystem are returned.

EXAMPLES:

```
sage: # needs sage.graphs
sage: L = RootSystem(['A',3]).weight_lattice()
sage: Lambda = L.fundamental_weights()
sage: Lambda[1].succ()
[-Lambda[1] + Lambda[2]]
sage: L.rho().succ()
[-Lambda[1] + 2*Lambda[2] + Lambda[3],
 2*Lambda[1] - Lambda[2] + 2*Lambda[3],
 Lambda[1] + 2*Lambda[2] - Lambda[3]]
sage: (-L.rho()).succ()
```

(continues on next page)

(continued from previous page)

```
[]
sage: L.rho().succ(index_set=[1])
[-Lambda[1] + 2*Lambda[2] + Lambda[3]]
sage: L.rho().succ(index_set=[2])
[2*Lambda[1] - Lambda[2] + 2*Lambda[3]]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> L = RootSystem(['A',Integer(3)]).weight_lattice()
>>> Lambda = L.fundamental_weights()
>>> Lambda[Integer(1)].succ()
[-Lambda[1] + Lambda[2]]
>>> L.rho().succ()
[-Lambda[1] + 2*Lambda[2] + Lambda[3],
 2*Lambda[1] - Lambda[2] + 2*Lambda[3],
 Lambda[1] + 2*Lambda[2] - Lambda[3]]
>>> (-L.rho()).succ()
[]
>>> L.rho().succ(index_set=[Integer(1)])
[-Lambda[1] + 2*Lambda[2] + Lambda[3]]
>>> L.rho().succ(index_set=[Integer(2)])
[2*Lambda[1] - Lambda[2] + 2*Lambda[3]]
```

symmetric_form(*alpha*)

Return the symmetric form of `self` with `alpha`.

Consider the simple roots α_i and let $(b_{ij})_{ij}$ denote the symmetrized Cartan matrix $(a_{ij})_{ij}$, we have

$$(\alpha_i|\alpha_j) = b_{ij}$$

and extended bilinearly. See Chapter 6 in Kac, Infinite Dimensional Lie Algebras for more details.

EXAMPLES:

```
sage: # needs sage.graphs
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[1].symmetric_form(alpha[0])
0
sage: alpha[1].symmetric_form(alpha[1])
4
sage: elt = alpha[0] - 3*alpha[1] + alpha[2]
sage: elt.symmetric_form(alpha[1])
-14
sage: elt.symmetric_form(alpha[0]+2*alpha[2])
14

sage: Q = RootSystem(CartanType(['A',4,2]).dual()).root_lattice()
sage: Qc = RootSystem(['A',4,2]).coroot_lattice()
sage: alpha = Q.simple_roots()
sage: alphac = Qc.simple_roots()
sage: elt = alpha[0] + 2*alpha[1] + 2*alpha[2]
sage: eltc = alphac[0] + 2*alphac[1] + 2*alphac[2]
```

(continues on next page)

(continued from previous page)

```
sage: elt.symmetric_form(alpha[1])                                        #␣
↪needs sage.graphs
0
sage: eltc.symmetric_form(alphac[1])                                      #␣
↪needs sage.graphs
0
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> Q = RootSystem(['B',Integer(2),Integer(1)]).root_lattice()
>>> alpha = Q.simple_roots()
>>> alpha[Integer(1)].symmetric_form(alpha[Integer(0)])
0
>>> alpha[Integer(1)].symmetric_form(alpha[Integer(1)])
4
>>> elt = alpha[Integer(0)] - Integer(3)*alpha[Integer(1)] +␣
↪alpha[Integer(2)]
>>> elt.symmetric_form(alpha[Integer(1)])
-14
>>> elt.symmetric_form(alpha[Integer(0)]+Integer(2)*alpha[Integer(2)])
14

>>> Q = RootSystem(CartanType(['A',Integer(4),Integer(2)]).dual()).root_
↪lattice()
>>> Qc = RootSystem(['A',Integer(4),Integer(2)]).coroot_lattice()
>>> alpha = Q.simple_roots()
>>> alphac = Qc.simple_roots()
>>> elt = alpha[Integer(0)] + Integer(2)*alpha[Integer(1)] +␣
↪Integer(2)*alpha[Integer(2)]
>>> eltc = alphac[Integer(0)] + Integer(2)*alphac[Integer(1)] +␣
↪Integer(2)*alphac[Integer(2)]
>>> elt.symmetric_form(alpha[Integer(1)])                                  ␣
↪     # needs sage.graphs
0
>>> eltc.symmetric_form(alphac[Integer(1)])                                ␣
↪     # needs sage.graphs
0
```

to_ambient()

Map `self` to the ambient space.

EXAMPLES:

```
sage: B4_rs = CartanType(['B',4]).root_system()
sage: alpha = B4_rs.root_lattice().an_element(); alpha
2*alpha[1] + 2*alpha[2] + 3*alpha[3]
sage: alpha.to_ambient()
(2, 0, 1, -3)
sage: mu = B4_rs.weight_lattice().an_element(); mu
2*Lambda[1] + 2*Lambda[2] + 3*Lambda[3]
sage: mu.to_ambient()
(7, 5, 3, 0)
```

(continues on next page)

(continued from previous page)

```
sage: v = B4_rs.ambient_space().an_element(); v
(2, 2, 3, 0)
sage: v.to_ambient()
(2, 2, 3, 0)
sage: alphavee = B4_rs.coroot_lattice().an_element(); alphavee
2*alphacheck[1] + 2*alphacheck[2] + 3*alphacheck[3]
sage: alphavee.to_ambient()
(2, 0, 1, -3)
```

```
>>> from sage.all import *
>>> B4_rs = CartanType(['B',Integer(4)]).root_system()
>>> alpha = B4_rs.root_lattice().an_element(); alpha
2*alpha[1] + 2*alpha[2] + 3*alpha[3]
>>> alpha.to_ambient()
(2, 0, 1, -3)
>>> mu = B4_rs.weight_lattice().an_element(); mu
2*Lambda[1] + 2*Lambda[2] + 3*Lambda[3]
>>> mu.to_ambient()
(7, 5, 3, 0)
>>> v = B4_rs.ambient_space().an_element(); v
(2, 2, 3, 0)
>>> v.to_ambient()
(2, 2, 3, 0)
>>> alphavee = B4_rs.coroot_lattice().an_element(); alphavee
2*alphacheck[1] + 2*alphacheck[2] + 3*alphacheck[3]
>>> alphavee.to_ambient()
(2, 0, 1, -3)
```

to_classical()

Map `self` to the classical lattice/space.

Only makes sense for affine type.

EXAMPLES:

```
sage: R = CartanType(['A',3,1]).root_system()
sage: alpha = R.root_lattice().an_element(); alpha
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
sage: alb = alpha.to_classical(); alb                                  #␣
↪needs sage.graphs
alpha[2] - 2*alpha[3]
sage: alb.parent()                                                     #␣
↪needs sage.graphs
Root lattice of the Root system of type ['A', 3]
sage: v = R.ambient_space().an_element(); v
2*e[0] + 2*e[1] + 3*e[2]
sage: v.to_classical()                                                 #␣
↪needs sage.graphs
(2, 2, 3, 0)
```

```
>>> from sage.all import *
>>> R = CartanType(['A',Integer(3),Integer(1)]).root_system()
```

(continues on next page)

(continued from previous page)

```
>>> alpha = R.root_lattice().an_element(); alpha
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
>>> alb = alpha.to_classical(); alb                                   #␣
↪needs sage.graphs
alpha[2] - 2*alpha[3]
>>> alb.parent()                                                      #␣
↪needs sage.graphs
Root lattice of the Root system of type ['A', 3]
>>> v = R.ambient_space().an_element(); v
2*e[0] + 2*e[1] + 3*e[2]
>>> v.to_classical()                                                  #␣
↪needs sage.graphs
(2, 2, 3, 0)
```

to_dominant_chamber(*index_set=None*, *positive=True*, *reduced_word=False*)

Return the unique dominant element in the Weyl group orbit of the vector `self`.

If `positive` is `False`, returns the antidominant orbit element.

With the `index_set` optional parameter, this is done with respect to the corresponding parabolic subgroup.

If `reduced_word` is `True`, returns the 2-tuple (`weight`, `direction`) where `weight` is the (anti)dominant orbit element and `direction` is a reduced word for the Weyl group element sending `weight` to `self`.

Warning

In infinite type, an orbit may not contain a dominant element. In this case the function may go into an infinite loop.

For affine root systems, errors are generated if the orbit does not contain the requested kind of representative. If the input vector is of positive (resp. negative) level, then there is a dominant (resp. antidominant) element in its orbit but not an antidominant (resp. dominant) one. If the vector is of level zero, then there are neither dominant nor antidominant orbit representatives, except for multiples of the null root, which are themselves both dominant and antidominant orbit representatives.

EXAMPLES:

```
sage: # needs sage.graphs
sage: space = RootSystem(['A',5]).weight_space()
sage: alpha = RootSystem(['A',5]).weight_space().simple_roots()
sage: alpha[1].to_dominant_chamber()
Lambda[1] + Lambda[5]
sage: alpha[1].to_dominant_chamber([1,2])
Lambda[1] + Lambda[2] - Lambda[3]
sage: wl = RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: mu = wl.from_vector(vector([1,-3,0]))
sage: mu.to_dominant_chamber(positive=False, reduced_word=True)
(-Lambda[1] - Lambda[2] - delta, [0, 2])

sage: # needs sage.graphs
sage: R = RootSystem(['A',1,1])
```

(continues on next page)

(continued from previous page)

```
sage: rl = R.root_lattice()
sage: nu = rl.zero()
sage: nu.to_dominant_chamber()
0
sage: nu.to_dominant_chamber(positive=False)
0
sage: mu = rl.from_vector(vector([0,1]))
sage: mu.to_dominant_chamber()
Traceback (most recent call last):
...
ValueError: alpha[1] is not in the orbit of the fundamental chamber
sage: mu.to_dominant_chamber(positive=False)
Traceback (most recent call last):
...
ValueError: alpha[1] is not in the orbit of the negative of the␣
↪fundamental chamber
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> space = RootSystem(['A',Integer(5)]).weight_space()
>>> alpha = RootSystem(['A',Integer(5)]).weight_space().simple_roots()
>>> alpha[Integer(1)].to_dominant_chamber()
Lambda[1] + Lambda[5]
>>> alpha[Integer(1)].to_dominant_chamber([Integer(1),Integer(2)])
Lambda[1] + Lambda[2] - Lambda[3]
>>> wl = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True)
>>> mu = wl.from_vector(vector([Integer(1),-Integer(3),Integer(0)]))
>>> mu.to_dominant_chamber(positive=False, reduced_word=True)
(-Lambda[1] - Lambda[2] - delta, [0, 2])

>>> # needs sage.graphs
>>> R = RootSystem(['A',Integer(1),Integer(1)])
>>> rl = R.root_lattice()
>>> nu = rl.zero()
>>> nu.to_dominant_chamber()
0
>>> nu.to_dominant_chamber(positive=False)
0
>>> mu = rl.from_vector(vector([Integer(0),Integer(1)]))
>>> mu.to_dominant_chamber()
Traceback (most recent call last):
...
ValueError: alpha[1] is not in the orbit of the fundamental chamber
>>> mu.to_dominant_chamber(positive=False)
Traceback (most recent call last):
...
ValueError: alpha[1] is not in the orbit of the negative of the␣
↪fundamental chamber
```

to_dual_type_cospace()

Map `self` to the dual type cospace.

For example, if `self` is in the root lattice of type $['B', 2]$, send it to the coroot lattice of type $['C', 2]$.

EXAMPLES:

```
sage: v = CartanType(['C',3]).root_system().weight_lattice().an_element();
↪ v
2*Lambda[1] + 2*Lambda[2] + 3*Lambda[3]
sage: w = v.to_dual_type_cospace(); w
2*Lambdacheck[1] + 2*Lambdacheck[2] + 3*Lambdacheck[3]
sage: w.parent()
Coweight lattice of the Root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> v = CartanType(['C',Integer(3)]).root_system().weight_lattice().an_
↪element(); v
2*Lambda[1] + 2*Lambda[2] + 3*Lambda[3]
>>> w = v.to_dual_type_cospace(); w
2*Lambdacheck[1] + 2*Lambdacheck[2] + 3*Lambdacheck[3]
>>> w.parent()
Coweight lattice of the Root system of type ['B', 3]
```

to_simple_root(*reduced_word=False*)

Return (the index of) a simple root in the orbit of the positive root `self`.

INPUT:

- `self` – a positive root
- `reduced_word` – boolean (default: `False`)

OUTPUT:

- The index i of a simple root α_i. If `reduced_word` is `True`, this returns instead a pair `(i, word)`, where word is a sequence of reflections mapping α_i up the root poset to `self`.

EXAMPLES:

```
sage: L = RootSystem(["A",3]).root_lattice()
sage: positive_roots = L.positive_roots()
sage: for alpha in sorted(positive_roots):                              #␣
↪needs sage.graphs
....:     print("{} {}".format(alpha, alpha.to_simple_root()))
alpha[1] 1
alpha[1] + alpha[2] 2
alpha[1] + alpha[2] + alpha[3] 3
alpha[2] 2
alpha[2] + alpha[3] 3
alpha[3] 3
sage: for alpha in sorted(positive_roots):                              #␣
↪needs sage.graphs
....:     print("{} {}".format(alpha, alpha.to_simple_root(reduced_
↪word=True)))
alpha[1] (1, ())
alpha[1] + alpha[2] (2, (1,))
alpha[1] + alpha[2] + alpha[3] (3, (1, 2))
alpha[2] (2, ())
alpha[2] + alpha[3] (3, (2,))
alpha[3] (3, ())
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3)]).root_lattice()
>>> positive_roots = L.positive_roots()
>>> for alpha in sorted(positive_roots):                                    #␣
↪needs sage.graphs
...     print("{} {}".format(alpha, alpha.to_simple_root()))
alpha[1] 1
alpha[1] + alpha[2] 2
alpha[1] + alpha[2] + alpha[3] 3
alpha[2] 2
alpha[2] + alpha[3] 3
alpha[3] 3
>>> for alpha in sorted(positive_roots):                                    #␣
↪needs sage.graphs
...     print("{} {}".format(alpha, alpha.to_simple_root(reduced_
↪word=True)))
alpha[1] (1, ())
alpha[1] + alpha[2] (2, (1,))
alpha[1] + alpha[2] + alpha[3] (3, (1, 2))
alpha[2] (2, ())
alpha[2] + alpha[3] (3, (2,))
alpha[3] (3, ())
```

ALGORITHM:

This method walks from `self` down to the antidominant chamber by applying successively the simple reflection given by the first descent. Since `self` is a positive root, each step goes down the root poset, and one must eventually cross a simple root α_i.

See also

- *first_descent()*
- *to_dominant_chamber()*

Warning

The behavior is not specified if the input is not a positive root. For a finite root system, this is currently caught (albeit with a not perfect message):

```
sage: alpha = L.simple_roots()                                             #␣
↪needs sage.graphs
sage: (2*alpha[1]).to_simple_root()                                        #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: -2*alpha[1] - 2*alpha[2] - 2*alpha[3] is not a positive root
```

```
>>> from sage.all import *
>>> alpha = L.simple_roots()                                             #␣
↪needs sage.graphs
>>> (Integer(2)*alpha[Integer(1)]).to_simple_root()                              ␣
↪             # needs sage.graphs
Traceback (most recent call last):
...
ValueError: -2*alpha[1] - 2*alpha[2] - 2*alpha[3] is not a positive root
```

For an infinite root system, this method may run into an infinite recursion if the input is not a positive root.

translation(x)

Return x translated by t, that is, $x + level(x)t$.

INPUT:

- `self` – an element t at level 0
- `x` – an element of the same space

EXAMPLES:

```
sage: L = RootSystem(['A',2,1]).weight_lattice()
sage: alpha = L.simple_roots()                                          #␣
↪needs sage.graphs
sage: Lambda = L.fundamental_weights()                                  #␣
↪needs sage.graphs
sage: t = alpha[2]                                                      #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice()
>>> alpha = L.simple_roots()                                          #␣
↪needs sage.graphs
>>> Lambda = L.fundamental_weights()                                  #␣
↪needs sage.graphs
>>> t = alpha[Integer(2)]                                             ␣
↪      # needs sage.graphs
```

Let us look at the translation of an element of level 1:

```
sage: Lambda[1].level()                                                 #␣
↪needs sage.graphs
1
sage: t.translation(Lambda[1])                                          #␣
↪needs sage.graphs
-Lambda[0] + 2*Lambda[2]
sage: Lambda[1] + t                                                     #␣
↪needs sage.graphs
-Lambda[0] + 2*Lambda[2]
```

```
>>> from sage.all import *
>>> Lambda[Integer(1)].level()                                          ␣
↪      # needs sage.graphs
1
>>> t.translation(Lambda[Integer(1)])                                   ␣
↪      # needs sage.graphs
-Lambda[0] + 2*Lambda[2]
>>> Lambda[Integer(1)] + t                                              ␣
↪      # needs sage.graphs
-Lambda[0] + 2*Lambda[2]
```

and of an element of level 0:

```
sage: alpha[1].level()                                                  #␣
↪needs sage.graphs
0
sage: t.translation(alpha [1])                                          #␣
↪needs sage.graphs
-Lambda[0] + 2*Lambda[1] - Lambda[2]
sage: alpha[1] + 0*t                                                    #␣
↪needs sage.graphs
-Lambda[0] + 2*Lambda[1] - Lambda[2]
```

```
>>> from sage.all import *
>>> alpha[Integer(1)].level()                                           ␣
↪     # needs sage.graphs
0
>>> t.translation(alpha [Integer(1)])                                   ␣
↪     # needs sage.graphs
-Lambda[0] + 2*Lambda[1] - Lambda[2]
>>> alpha[Integer(1)] + Integer(0)*t                                    ␣
↪              # needs sage.graphs
-Lambda[0] + 2*Lambda[1] - Lambda[2]
```

The arguments are given in this seemingly unnatural order to make it easy to construct the translation function:

```
sage: f = t.translation                                                 #␣
↪needs sage.graphs
sage: f(Lambda[1])                                                      #␣
↪needs sage.graphs
-Lambda[0] + 2*Lambda[2]
```

```
>>> from sage.all import *
>>> f = t.translation                                                 #␣
↪needs sage.graphs
>>> f(Lambda[Integer(1)])                                               ␣
↪     # needs sage.graphs
-Lambda[0] + 2*Lambda[2]
```

weyl_action(*element*, *inverse=False*)

Act on `self` by an element of the Coxeter or Weyl group.

INPUT:

- `element` – an element of a Coxeter or Weyl group of the same Cartan type, or a tuple or a list (such as a reduced word) of elements from the index set
- `inverse` – boolean (default: `False`); whether to act by the inverse element

EXAMPLES:

```
sage: wl = RootSystem(['A',3]).weight_lattice()
sage: mu = wl.from_vector(vector([1,0,-2]))
sage: mu
Lambda[1] - 2*Lambda[3]
sage: mudom, rw = mu.to_dominant_chamber(positive=False,                #␣
↪needs sage.graphs
....:                                        reduced_word=True)
```

(continues on next page)

(continued from previous page)

```
sage: mudom, rw                                                          #␣
↪needs sage.graphs
(-Lambda[2] - Lambda[3], [1, 2])
```

```
>>> from sage.all import *
>>> wl = RootSystem(['A',Integer(3)]).weight_lattice()
>>> mu = wl.from_vector(vector([Integer(1),Integer(0),-Integer(2)]))
>>> mu
Lambda[1] - 2*Lambda[3]
>>> mudom, rw = mu.to_dominant_chamber(positive=False,                 #␣
↪needs sage.graphs
...                                    reduced_word=True)
>>> mudom, rw                                                          #␣
↪needs sage.graphs
(-Lambda[2] - Lambda[3], [1, 2])
```

Acting by a (reduced) word:

```
sage: mudom.weyl_action(rw)                                              #␣
↪needs sage.graphs
Lambda[1] - 2*Lambda[3]
sage: mu.weyl_action(rw, inverse=True)                                   #␣
↪needs sage.graphs
-Lambda[2] - Lambda[3]
```

```
>>> from sage.all import *
>>> mudom.weyl_action(rw)                                              #␣
↪needs sage.graphs
Lambda[1] - 2*Lambda[3]
>>> mu.weyl_action(rw, inverse=True)                                   #␣
↪needs sage.graphs
-Lambda[2] - Lambda[3]
```

Acting by an element of the Coxeter or Weyl group on a vector in its own lattice of definition (implemented by matrix multiplication on a vector):

```
sage: w = wl.weyl_group().from_reduced_word([1, 2])                      #␣
↪needs sage.graphs sage.libs.gap
sage: mudom.weyl_action(w)                                               #␣
↪needs sage.graphs sage.libs.gap
Lambda[1] - 2*Lambda[3]
```

```
>>> from sage.all import *
>>> w = wl.weyl_group().from_reduced_word([Integer(1), Integer(2)])     ␣
↪              # needs sage.graphs sage.libs.gap
>>> mudom.weyl_action(w)                                               #␣
↪needs sage.graphs sage.libs.gap
Lambda[1] - 2*Lambda[3]
```

Acting by an element of an isomorphic Coxeter or Weyl group (implemented by the action of a corresponding reduced word):

```
sage: # needs sage.libs.gap
sage: W = WeylGroup(['A',3], prefix='s')
sage: w = W.from_reduced_word([1, 2])
sage: wl.weyl_group() == W
False
sage: mudom.weyl_action(w)                                              #␣
↪needs sage.graphs
Lambda[1] - 2*Lambda[3]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> W = WeylGroup(['A',Integer(3)], prefix='s')
>>> w = W.from_reduced_word([Integer(1), Integer(2)])
>>> wl.weyl_group() == W
False
>>> mudom.weyl_action(w)                                                #␣
↪needs sage.graphs
Lambda[1] - 2*Lambda[3]
```

weyl_stabilizer(*index_set=None*)

Return the subset of Dynkin nodes whose reflections fix `self`.

If `index_set` is not `None`, only consider nodes in this set. Note that if `self` is dominant or antidominant, then its stabilizer is the parabolic subgroup defined by the returned node set.

EXAMPLES:

```
sage: wl = RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: al = wl.null_root()                                               #␣
↪needs sage.graphs
sage: al.weyl_stabilizer()                                              #␣
↪needs sage.graphs
[0, 1, 2]
sage: wl = RootSystem(['A',4]).weight_lattice()
sage: mu = wl.from_vector(vector([1,1,0,0]))
sage: mu.weyl_stabilizer()
[3, 4]
sage: mu.weyl_stabilizer(index_set = [1,2,3])
[3]
```

```
>>> from sage.all import *
>>> wl = RootSystem(['A',Integer(2),Integer(1)]).weight_
↪lattice(extended=True)
>>> al = wl.null_root()                                                 #␣
↪needs sage.graphs
>>> al.weyl_stabilizer()                                                #␣
↪needs sage.graphs
[0, 1, 2]
>>> wl = RootSystem(['A',Integer(4)]).weight_lattice()
>>> mu = wl.from_vector(vector([Integer(1),Integer(1),Integer(0),
↪Integer(0)]))
>>> mu.weyl_stabilizer()
[3, 4]
```

(continues on next page)

(continued from previous page)

```
>>> mu.weyl_stabilizer(index_set = [Integer(1),Integer(2),Integer(3)])
[3]
```

class ParentMethods

Bases: `object`

a_long_simple_root()

Return a long simple root, corresponding to the highest outgoing edge in the Dynkin diagram.

Warning

This may be broken in affine type $A_{2n}^{(2)}$

Is it meaningful/broken for non irreducible?

Todo

implement CartanType.nodes_by_length as in MuPAD-Combinat (using CartanType.symmetrizer), and use it here.

almost_positive_roots()

Return the almost positive roots of `self`.

These are the positive roots together with the simple negative roots.

See also

`almost_positive_root_decomposition()`, *`tau_plus_minus()`*

EXAMPLES:

```
sage: L = RootSystem(['A',2]).root_lattice()
sage: L.almost_positive_roots()                                         #␣
↪needs sage.graphs
[-alpha[1], alpha[1], alpha[1] + alpha[2], -alpha[2], alpha[2]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2)]).root_lattice()
>>> L.almost_positive_roots()                                         #␣
↪needs sage.graphs
[-alpha[1], alpha[1], alpha[1] + alpha[2], -alpha[2], alpha[2]]
```

almost_positive_roots_decomposition()

Return the decomposition of the almost positive roots of `self`.

This is the list of the orbits of the almost positive roots under the action of the dihedral group generated by the operators τ_+ and τ_-.

See also

- *almost_positive_roots()*
- *tau_plus_minus()*

EXAMPLES:

```
sage: RootSystem(['A',2]).root_lattice().almost_positive_roots_
↪decomposition()      # needs sage.graphs sage.libs.gap
[[-alpha[1], alpha[1], alpha[1] + alpha[2], alpha[2], -alpha[2]]]

sage: RootSystem(['B',2]).root_lattice().almost_positive_roots_
↪decomposition()      # needs sage.graphs sage.libs.gap
[[-alpha[1], alpha[1], alpha[1] + 2*alpha[2]],
 [-alpha[2], alpha[2], alpha[1] + alpha[2]]]

sage: RootSystem(['D',4]).root_lattice().almost_positive_roots_
↪decomposition()      # needs sage.graphs sage.libs.gap
[[-alpha[1], alpha[1], alpha[1] + alpha[2], alpha[2] + alpha[3] +␣
↪alpha[4]],
 [-alpha[2], alpha[2], alpha[1] + alpha[2] + alpha[3] + alpha[4],
     alpha[1] + 2*alpha[2] + alpha[3] + alpha[4]],
 [-alpha[3], alpha[3], alpha[2] + alpha[3], alpha[1] + alpha[2] +␣
↪alpha[4]],
 [-alpha[4], alpha[4], alpha[2] + alpha[4], alpha[1] + alpha[2] +␣
↪alpha[3]]]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(2)]).root_lattice().almost_positive_roots_
↪decomposition()      # needs sage.graphs sage.libs.gap
[[-alpha[1], alpha[1], alpha[1] + alpha[2], alpha[2], -alpha[2]]]

>>> RootSystem(['B',Integer(2)]).root_lattice().almost_positive_roots_
↪decomposition()      # needs sage.graphs sage.libs.gap
[[-alpha[1], alpha[1], alpha[1] + 2*alpha[2]],
 [-alpha[2], alpha[2], alpha[1] + alpha[2]]]

>>> RootSystem(['D',Integer(4)]).root_lattice().almost_positive_roots_
↪decomposition()      # needs sage.graphs sage.libs.gap
[[-alpha[1], alpha[1], alpha[1] + alpha[2], alpha[2] + alpha[3] +␣
↪alpha[4]],
 [-alpha[2], alpha[2], alpha[1] + alpha[2] + alpha[3] + alpha[4],
     alpha[1] + 2*alpha[2] + alpha[3] + alpha[4]],
 [-alpha[3], alpha[3], alpha[2] + alpha[3], alpha[1] + alpha[2] +␣
↪alpha[4]],
 [-alpha[4], alpha[4], alpha[2] + alpha[4], alpha[1] + alpha[2] +␣
↪alpha[3]]]
```

alpha()

Return the family $(\alpha_i)_{i \in I}$ of the simple roots, with the extra feature that, for simple irreducible root systems, α_0 yields the opposite of the highest root.

EXAMPLES:

```
sage: alpha = RootSystem(["A",2]).root_lattice().alpha()
sage: alpha[1]
alpha[1]
sage: alpha[0]                                                              #␣
↪needs sage.graphs
-alpha[1] - alpha[2]
```

```
>>> from sage.all import *
>>> alpha = RootSystem(["A",Integer(2)]).root_lattice().alpha()
>>> alpha[Integer(1)]
alpha[1]
>>> alpha[Integer(0)]                                                        ␣
↪     # needs sage.graphs
-alpha[1] - alpha[2]
```

alphacheck()

Return the family $(\alpha_i^\vee)_{i\in I}$ of the simple coroots, with the extra feature that, for simple irreducible root systems, $\alpha_0^\vee$ yields the coroot associated to the opposite of the highest root (caveat: for non-simply-laced root systems, this is not the opposite of the highest coroot!).

EXAMPLES:

```
sage: alphacheck = RootSystem(["A",2]).ambient_space().alphacheck()
sage: alphacheck
Finite family {1: (1, -1, 0), 2: (0, 1, -1)}
```

```
>>> from sage.all import *
>>> alphacheck = RootSystem(["A",Integer(2)]).ambient_space().alphacheck()
>>> alphacheck
Finite family {1: (1, -1, 0), 2: (0, 1, -1)}
```

Here is now $\alpha_0^\vee$:

(-1, 0, 1)

Todo

add a non simply laced example

Finally, here is an affine example:

```
sage: RootSystem(["A",2,1]).weight_space().alphacheck()
Finite family {0: alphacheck[0], 1: alphacheck[1], 2: alphacheck[2]}

sage: RootSystem(["A",3]).ambient_space().alphacheck()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(2),Integer(1)]).weight_space().alphacheck()
Finite family {0: alphacheck[0], 1: alphacheck[1], 2: alphacheck[2]}
```

(continues on next page)

(continued from previous page)

```
>>> RootSystem(["A",Integer(3)]).ambient_space().alphacheck()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

basic_imaginary_roots()

Return the basic imaginary roots of `self`.

The basic imaginary roots δ are the set of imaginary roots in $-C^\vee$ where C is the dominant chamber (i.e., $\langle \beta, \alpha_i^\vee \rangle \leq 0$ for all $i \in I$). All imaginary roots are W-conjugate to a simple imaginary root.

EXAMPLES:

```
sage: RootSystem(['A', 2]).root_lattice().basic_imaginary_roots()
()
sage: Q = RootSystem(['A', 2, 1]).root_lattice()
sage: Q.basic_imaginary_roots()                                        #␣
↪needs sage.graphs
(alpha[0] + alpha[1] + alpha[2],)
sage: delta = Q.basic_imaginary_roots()[0]                             #␣
↪needs sage.graphs
sage: all(delta.scalar(Q.simple_coroot(i)) <= 0                        #␣
↪needs sage.graphs
....:     for i in Q.index_set())
True
```

```
>>> from sage.all import *
>>> RootSystem(['A', Integer(2)]).root_lattice().basic_imaginary_roots()
()
>>> Q = RootSystem(['A', Integer(2), Integer(1)]).root_lattice()
>>> Q.basic_imaginary_roots()                                        #␣
↪needs sage.graphs
(alpha[0] + alpha[1] + alpha[2],)
>>> delta = Q.basic_imaginary_roots()[Integer(0)]                       ␣
↪     # needs sage.graphs
>>> all(delta.scalar(Q.simple_coroot(i)) <= Integer(0)                  ␣
↪     # needs sage.graphs
...     for i in Q.index_set())
True
```

cartan_type()

EXAMPLES:

```
sage: r = RootSystem(['A',4]).root_space()
sage: r.cartan_type()
['A', 4]
```

```
>>> from sage.all import *
>>> r = RootSystem(['A',Integer(4)]).root_space()
>>> r.cartan_type()
['A', 4]
```

classical()

Return the corresponding root/weight/ambient lattice/space.

EXAMPLES:

```
sage: RootSystem(["A",4,1]).root_lattice().classical()
Root lattice of the Root system of type ['A', 4]
sage: RootSystem(["A",4,1]).weight_lattice().classical()
Weight lattice of the Root system of type ['A', 4]
sage: RootSystem(["A",4,1]).ambient_space().classical()
Ambient space of the Root system of type ['A', 4]
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(4),Integer(1)]).root_lattice().classical()
Root lattice of the Root system of type ['A', 4]
>>> RootSystem(["A",Integer(4),Integer(1)]).weight_lattice().classical()
Weight lattice of the Root system of type ['A', 4]
>>> RootSystem(["A",Integer(4),Integer(1)]).ambient_space().classical()
Ambient space of the Root system of type ['A', 4]
```

cohighest_root()

Return the associated coroot of the highest root.

> **Note**
>
> this is usually not the highest coroot.

EXAMPLES:

```
sage: RootSystem(['A', 3]).ambient_space().cohighest_root()
(1, 0, 0, -1)
```

```
>>> from sage.all import *
>>> RootSystem(['A', Integer(3)]).ambient_space().cohighest_root()
(1, 0, 0, -1)
```

coroot_lattice()

Return the coroot lattice.

EXAMPLES:

```
sage: RootSystem(['A',2]).root_lattice().coroot_lattice()
Coroot lattice of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(2)]).root_lattice().coroot_lattice()
Coroot lattice of the Root system of type ['A', 2]
```

coroot_space(*base_ring=Rational Field*)

Return the coroot space over `base_ring`.

INPUT:

- `base_ring` – a ring (default: $\mathbf{Q}$)

EXAMPLES:

```
sage: RootSystem(['A',2]).root_lattice().coroot_space()
Coroot space over the Rational Field of the Root system of type ['A', 2]
```

(continues on next page)

(continued from previous page)

```
sage: RootSystem(['A',2]).root_lattice().coroot_space(QQ['q'])
Coroot space over the Univariate Polynomial Ring in q over Rational Field
 of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(2)]).root_lattice().coroot_space()
Coroot space over the Rational Field of the Root system of type ['A', 2]

>>> RootSystem(['A',Integer(2)]).root_lattice().coroot_space(QQ['q'])
Coroot space over the Univariate Polynomial Ring in q over Rational Field
 of the Root system of type ['A', 2]
```

dual_type_cospace()

Return the cospace of dual type.

For example, if invoked on the root lattice of type $['B', 2]$, returns the coroot lattice of type $['C', 2]$.

> **Warning**
>
> Not implemented for ambient spaces.

EXAMPLES:

```
sage: CartanType(['B',2]).root_system().root_lattice().dual_type_cospace()
Coroot lattice of the Root system of type ['C', 2]
sage: CartanType(['F',4]).root_system().coweight_lattice().dual_type_
↪cospace()
Weight lattice of the Root system of type ['F', 4]
 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> CartanType(['B',Integer(2)]).root_system().root_lattice().dual_type_
↪cospace()
Coroot lattice of the Root system of type ['C', 2]
>>> CartanType(['F',Integer(4)]).root_system().coweight_lattice().dual_
↪type_cospace()
Weight lattice of the Root system of type ['F', 4]
 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

dynkin_diagram()

EXAMPLES:

```
sage: r = RootSystem(['A',4]).root_space()
sage: r.dynkin_diagram()                                                #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

```
>>> from sage.all import *
>>> r = RootSystem(['A',Integer(4)]).root_space()
>>> r.dynkin_diagram()                                                  #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
A4
```

fundamental_weights_from_simple_roots()

Return the fundamental weights.

This is computed from the simple roots by using the inverse of the Cartan matrix. This method is therefore only valid for finite types and if this realization of the root lattice is large enough to contain them.

EXAMPLES:

In the root space, we retrieve the inverse of the Cartan matrix:

```
sage: L = RootSystem(["B",3]).root_space()
sage: L.fundamental_weights_from_simple_roots()                           #␣
↪needs sage.graphs
Finite family {1:     alpha[1] +   alpha[2] +     alpha[3],
               2:     alpha[1] + 2*alpha[2] +   2*alpha[3],
               3: 1/2*alpha[1] +   alpha[2] + 3/2*alpha[3]}
sage: ~L.cartan_type().cartan_matrix()                                    #␣
↪needs sage.graphs
[  1   1 1/2]
[  1   2   1]
[  1   2 3/2]
```

```
>>> from sage.all import *
>>> L = RootSystem(["B",Integer(3)]).root_space()
>>> L.fundamental_weights_from_simple_roots()                           #␣
↪needs sage.graphs
Finite family {1:     alpha[1] +   alpha[2] +     alpha[3],
               2:     alpha[1] + 2*alpha[2] +   2*alpha[3],
               3: 1/2*alpha[1] +   alpha[2] + 3/2*alpha[3]}
>>> ~L.cartan_type().cartan_matrix()                                    #␣
↪needs sage.graphs
[  1   1 1/2]
[  1   2   1]
[  1   2 3/2]
```

In the weight lattice and the ambient space, we retrieve the fundamental weights:

```
sage: L = RootSystem(["B",3]).weight_lattice()
sage: L.fundamental_weights_from_simple_roots()                           #␣
↪needs sage.graphs
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}

sage: L = RootSystem(["B",3]).ambient_space()
sage: L.fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
```

(continues on next page)

(continued from previous page)

```
sage: L.fundamental_weights_from_simple_roots()                              #␣
↪needs sage.graphs
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
```

```
>>> from sage.all import *
>>> L = RootSystem(["B",Integer(3)]).weight_lattice()
>>> L.fundamental_weights_from_simple_roots()                              #␣
↪needs sage.graphs
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}

>>> L = RootSystem(["B",Integer(3)]).ambient_space()
>>> L.fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
>>> L.fundamental_weights_from_simple_roots()                              #␣
↪needs sage.graphs
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
```

However the fundamental weights do not belong to the root lattice:

```
sage: L = RootSystem(["B",3]).root_lattice()
sage: L.fundamental_weights_from_simple_roots()                              #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: The fundamental weights do not live in this realization
of the root lattice
```

```
>>> from sage.all import *
>>> L = RootSystem(["B",Integer(3)]).root_lattice()
>>> L.fundamental_weights_from_simple_roots()                              #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: The fundamental weights do not live in this realization
of the root lattice
```

Beware of the usual GL_n vs SL_n catch in type A:

```
sage: L = RootSystem(["A",3]).ambient_space()
sage: L.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
sage: L.fundamental_weights_from_simple_roots()                              #␣
↪needs sage.graphs
Finite family {1: (3/4, -1/4, -1/4, -1/4),
               2: (1/2, 1/2, -1/2, -1/2),
               3: (1/4, 1/4, 1/4, -3/4)}

sage: L = RootSystem(["A",3]).ambient_lattice()
sage: L.fundamental_weights_from_simple_roots()                              #␣
↪needs sage.graphs
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: The fundamental weights do not live in this realization
of the root lattice
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3)]).ambient_space()
>>> L.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
>>> L.fundamental_weights_from_simple_roots()                          #␣
↪needs sage.graphs
Finite family {1: (3/4, -1/4, -1/4, -1/4),
               2: (1/2, 1/2, -1/2, -1/2),
               3: (1/4, 1/4, 1/4, -3/4)}

>>> L = RootSystem(["A",Integer(3)]).ambient_lattice()
>>> L.fundamental_weights_from_simple_roots()                          #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: The fundamental weights do not live in this realization
of the root lattice
```

generalized_nonnesting_partition_lattice(*m*, *facade=False*)

Return the lattice of m-nonnesting partitions.

This has been defined by Athanasiadis, see chapter 5 of [Arm06].

INPUT:

- m – integer

See also

nonnesting_partition_lattice()

EXAMPLES:

```
sage: R = RootSystem(['A', 2])
sage: RS = R.root_lattice()
sage: P = RS.generalized_nonnesting_partition_lattice(2); P              #␣
↪needs sage.graphs
Finite lattice containing 12 elements
sage: P.coxeter_transformation()**20 == 1                               #␣
↪needs sage.graphs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> R = RootSystem(['A', Integer(2)])
>>> RS = R.root_lattice()
>>> P = RS.generalized_nonnesting_partition_lattice(Integer(2)); P      ␣
↪     # needs sage.graphs
Finite lattice containing 12 elements
>>> P.coxeter_transformation()**Integer(20) == Integer(1)               ␣
↪              # needs sage.graphs sage.libs.flint
```

(continues on next page)

(continued from previous page)

```
True
```

highest_root()

Return the highest root (for an irreducible finite root system).

EXAMPLES:

```
sage: RootSystem(['A',4]).ambient_space().highest_root()
(1, 0, 0, 0, -1)

sage: RootSystem(['E',6]).weight_space().highest_root()                    #␣
↪needs sage.graphs
Lambda[2]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(4)]).ambient_space().highest_root()
(1, 0, 0, 0, -1)

>>> RootSystem(['E',Integer(6)]).weight_space().highest_root()              ␣
↪     # needs sage.graphs
Lambda[2]
```

index_set()

EXAMPLES:

```
sage: r = RootSystem(['A',4]).root_space()
sage: r.index_set()
(1, 2, 3, 4)
```

```
>>> from sage.all import *
>>> r = RootSystem(['A',Integer(4)]).root_space()
>>> r.index_set()
(1, 2, 3, 4)
```

long_roots()

Return a list of the long roots of `self`.

EXAMPLES:

```
sage: L = RootSystem(['B',3]).root_lattice()
sage: sorted(L.long_roots())                                                #␣
↪needs sage.graphs
[-alpha[1], -alpha[1] - 2*alpha[2] - 2*alpha[3],
 -alpha[1] - alpha[2], -alpha[1] - alpha[2] - 2*alpha[3],
 alpha[1], alpha[1] + alpha[2],
 alpha[1] + alpha[2] + 2*alpha[3],
 alpha[1] + 2*alpha[2] + 2*alpha[3], -alpha[2],
 -alpha[2] - 2*alpha[3], alpha[2], alpha[2] + 2*alpha[3]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['B',Integer(3)]).root_lattice()
>>> sorted(L.long_roots())                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[-alpha[1], -alpha[1] - 2*alpha[2] - 2*alpha[3],
 -alpha[1] - alpha[2], -alpha[1] - alpha[2] - 2*alpha[3],
 alpha[1], alpha[1] + alpha[2],
 alpha[1] + alpha[2] + 2*alpha[3],
 alpha[1] + 2*alpha[2] + 2*alpha[3], -alpha[2],
 -alpha[2] - 2*alpha[3], alpha[2], alpha[2] + 2*alpha[3]]
```

negative_roots()

Return the negative roots of `self`.

EXAMPLES:

```
sage: L = RootSystem(['A', 2]).weight_lattice()
sage: sorted(L.negative_roots())                                          #␣
↪needs sage.graphs
[-2*Lambda[1] + Lambda[2], -Lambda[1] - Lambda[2], Lambda[1] -␣
↪2*Lambda[2]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', Integer(2)]).weight_lattice()
>>> sorted(L.negative_roots())                                            #␣
↪needs sage.graphs
[-2*Lambda[1] + Lambda[2], -Lambda[1] - Lambda[2], Lambda[1] -␣
↪2*Lambda[2]]
```

Algorithm: negate the positive roots

nonnesting_partition_lattice(*facade=False*)

Return the lattice of nonnesting partitions.

This is the lattice of order ideals of the root poset.

This has been defined by Postnikov, see Remark 2 in [Reiner97].

See also

generalized_nonnesting_partition_lattice(), *root_poset()*

EXAMPLES:

```
sage: R = RootSystem(['A', 3])
sage: RS = R.root_lattice()
sage: P = RS.nonnesting_partition_lattice(); P                            #␣
↪needs sage.graphs
Finite lattice containing 14 elements
sage: P.coxeter_transformation()**10 == 1                                 #␣
↪needs sage.graphs sage.libs.flint
True

sage: # needs sage.graphs
sage: R = RootSystem(['B', 3])
sage: RS = R.root_lattice()
```

(continues on next page)

(continued from previous page)

```
sage: P = RS.nonnesting_partition_lattice(); P
Finite lattice containing 20 elements
sage: P.coxeter_transformation()**7 == 1                                    #␣
↪needs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> R = RootSystem(['A', Integer(3)])
>>> RS = R.root_lattice()
>>> P = RS.nonnesting_partition_lattice(); P                              #␣
↪needs sage.graphs
Finite lattice containing 14 elements
>>> P.coxeter_transformation()**Integer(10) == Integer(1)                     ␣
↪              # needs sage.graphs sage.libs.flint
True

>>> # needs sage.graphs
>>> R = RootSystem(['B', Integer(3)])
>>> RS = R.root_lattice()
>>> P = RS.nonnesting_partition_lattice(); P
Finite lattice containing 20 elements
>>> P.coxeter_transformation()**Integer(7) == Integer(1)                      ␣
↪              # needs sage.libs.flint
True
```

REFERENCES:

nonparabolic_positive_root_sum(*index_set=None*)

Return the sum of positive roots not in a parabolic subsystem.

The conventions for `index_set` are as in *nonparabolic_positive_roots()*.

EXAMPLES:

```
sage: Q = RootSystem(['A',3]).root_lattice()
sage: Q.nonparabolic_positive_root_sum((1,2))                                #␣
↪needs sage.graphs
alpha[1] + 2*alpha[2] + 3*alpha[3]
sage: Q.nonparabolic_positive_root_sum()                                     #␣
↪needs sage.graphs
0
sage: Q.nonparabolic_positive_root_sum(())                                   #␣
↪needs sage.graphs
3*alpha[1] + 4*alpha[2] + 3*alpha[3]
```

```
>>> from sage.all import *
>>> Q = RootSystem(['A',Integer(3)]).root_lattice()
>>> Q.nonparabolic_positive_root_sum((Integer(1),Integer(2)))                 ␣
↪              # needs sage.graphs
alpha[1] + 2*alpha[2] + 3*alpha[3]
>>> Q.nonparabolic_positive_root_sum()                                      #␣
↪needs sage.graphs
0
```

(continues on next page)

(continued from previous page)

```
>>> Q.nonparabolic_positive_root_sum(())                                  #␣
↪needs sage.graphs
3*alpha[1] + 4*alpha[2] + 3*alpha[3]
```

nonparabolic_positive_roots(*index_set=None*)

Return the positive roots of `self` that are not in the parabolic subsystem indicated by `index_set`.

If `index_set` is `None`, as in *positive_roots()* it is assumed to be the entire Dynkin node set. Then the parabolic subsystem consists of all positive roots and the empty list is returned.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).root_lattice()
sage: L.nonparabolic_positive_roots()                                      #␣
↪needs sage.graphs
[]
sage: sorted(L.nonparabolic_positive_roots((1,2)))                         #␣
↪needs sage.graphs
[alpha[1] + alpha[2] + alpha[3], alpha[2] + alpha[3], alpha[3]]
sage: sorted(L.nonparabolic_positive_roots(()))                            #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3], alpha[3]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(3)]).root_lattice()
>>> L.nonparabolic_positive_roots()                                       #␣
↪needs sage.graphs
[]
>>> sorted(L.nonparabolic_positive_roots((Integer(1),Integer(2))))        ␣
↪              # needs sage.graphs
[alpha[1] + alpha[2] + alpha[3], alpha[2] + alpha[3], alpha[3]]
>>> sorted(L.nonparabolic_positive_roots(()))                             #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3], alpha[3]]
```

null_coroot()

Return the null coroot of `self`.

The null coroot is the smallest non trivial positive coroot which is orthogonal to all simple roots. It exists for any affine root system.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).root_lattice().null_coroot()                   #␣
↪needs sage.graphs
alphacheck[0] + alphacheck[1] + alphacheck[2]
sage: RootSystem(['D',4,1]).root_lattice().null_coroot()                   #␣
↪needs sage.graphs
alphacheck[0] + alphacheck[1] + 2*alphacheck[2]
 + alphacheck[3] + alphacheck[4]
sage: RootSystem(['F',4,1]).root_lattice().null_coroot()                   #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
alphacheck[0] + 2*alphacheck[1] + 3*alphacheck[2]
 + 2*alphacheck[3] + alphacheck[4]
```

```
>>> from sage.all import *
>>> RootSystem(['C',Integer(2),Integer(1)]).root_lattice().null_coroot()
↪                # needs sage.graphs
alphacheck[0] + alphacheck[1] + alphacheck[2]
>>> RootSystem(['D',Integer(4),Integer(1)]).root_lattice().null_coroot()
↪                # needs sage.graphs
alphacheck[0] + alphacheck[1] + 2*alphacheck[2]
 + alphacheck[3] + alphacheck[4]
>>> RootSystem(['F',Integer(4),Integer(1)]).root_lattice().null_coroot()
↪                # needs sage.graphs
alphacheck[0] + 2*alphacheck[1] + 3*alphacheck[2]
 + 2*alphacheck[3] + alphacheck[4]
```

null_root()

Return the null root of `self`.

The null root is the smallest non trivial positive root which is orthogonal to all simple coroots. It exists for any affine root system.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).root_lattice().null_root()                    #
↪needs sage.graphs
alpha[0] + 2*alpha[1] + alpha[2]
sage: RootSystem(['D',4,1]).root_lattice().null_root()                    #
↪needs sage.graphs
alpha[0] + alpha[1] + 2*alpha[2] + alpha[3] + alpha[4]
sage: RootSystem(['F',4,1]).root_lattice().null_root()                    #
↪needs sage.graphs
alpha[0] + 2*alpha[1] + 3*alpha[2] + 4*alpha[3] + 2*alpha[4]
```

```
>>> from sage.all import *
>>> RootSystem(['C',Integer(2),Integer(1)]).root_lattice().null_root()
↪                # needs sage.graphs
alpha[0] + 2*alpha[1] + alpha[2]
>>> RootSystem(['D',Integer(4),Integer(1)]).root_lattice().null_root()
↪                # needs sage.graphs
alpha[0] + alpha[1] + 2*alpha[2] + alpha[3] + alpha[4]
>>> RootSystem(['F',Integer(4),Integer(1)]).root_lattice().null_root()
↪                # needs sage.graphs
alpha[0] + 2*alpha[1] + 3*alpha[2] + 4*alpha[3] + 2*alpha[4]
```

plot(*roots='simple'*, *coroots=False*, *reflection_hyperplanes='simple'*, *fundamental_weights=None*, *fundamental_chamber=None*, *alcoves=None*, *alcove_labels=False*, *alcove_walk=None*, ***options*)

Return a picture of this root lattice realization.

INPUT:

- `roots` – which roots to display, if any Can be one of the following:
  - `'simple'` – the simple roots (the default)
  - `'classical'` – not yet implemented

  - 'all' – only works in the finite case
  - A list or tuple of roots
  - False
- coroots – which coroots to display, if any Can be one of the following:
  - 'simple' – the simple coroots (the default)
  - 'classical' – not yet implemented
  - 'all' – only works in the finite case
  - A list or tuple of coroots
  - False
- fundamental_weights – boolean or None (default: None) whether to display the fundamental weights. If None, the fundamental weights are drawn if available.
- reflection_hyperplanes – which reflection hyperplanes to display, if any. Can be one of the following:
  - 'simple' – the simple roots
  - 'classical' – not yet implemented
  - 'all' – only works in the finite case
  - A list or tuple of roots
  - False (the default)
- fundamental_chamber – whether and how to draw the fundamental chamber. Can be one of the following:
  - A boolean – Set to True to draw the fundamental chamber
  - 'classical' – draw the classical fundamental chamber
  - None – (the default) The fundamental chamber is drawn except in the root lattice where this is not yet implemented. For affine types the classical fundamental chamber is drawn instead.
- alcoves – one of the following (default: True):
  - A boolean – whether to display the alcoves
  - A list of alcoves – The alcoves to be drawn. Each alcove is specified by the coordinates of its center in the root lattice (affine type only). Otherwise the alcoves that intersect the bounding box are drawn.
- alcove_labels – one of the following (default: False):
  - A boolean – whether to display the elements of the Weyl group indexing the alcoves. This currently requires to also set the alcoves option.
  - A number l – the label is drawn at level l (affine type only), which only makes sense if affine is False.
- bounding_box – a rational number or a list of pairs thereof (default: 3)

  Specifies a bounding box, in the coordinate system for this plot, in which to plot alcoves and other infinite objects. If the bounding box is a number a, then the bounding box is of the form $[-a, a]$ in all directions. Beware that there can be some border effects and the returned graphic is not necessarily strictly contained in the bounding box.
- alcove_walk – an alcove walk or None (default: None)

  The alcove walk is described by a list (or iterable) of vertices of the Dynkin diagram which specifies which wall is crossed at each step, starting from the fundamental alcove.
- projection – one of the following (default: True):
  - True – the default projection for the root lattice realization is used.
  - False – no projection is used.
  - barycentric – a barycentric projection is used.
  - A function – If a function is specified, it should implement a linear (or affine) map taking as input an element of this root lattice realization and returning its desired coordinates in the plot, as a vector with rational coordinates.
- color – a function mapping vertices of the Dynkin diagram to colors (default: 'black' for 0, 'blue' for 1, 'red' for 2, 'green' for 3)

  This is used to set the color for the simple roots, fundamental weights, reflection hyperplanes, alcove

facets, etc. If the color is None, the object is not drawn.
- labels – boolean (default: True) whether to display labels on the simple roots, fundamental weights, etc.

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space().plot()   # long time,␣
↪needs sage.plot sage.symbolic
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space().plot()   #␣
↪long time, needs sage.plot sage.symbolic
```

See also

- *plot_parse_options()*
- *plot_roots()*, *plot_coroots()*
- *plot_fundamental_weights()*
- *plot_fundamental_chamber()*
- *plot_reflection_hyperplanes()*
- *plot_alcoves()*
- *plot_alcove_walk()*
- *plot_ls_paths()*
- *plot_mv_polytope()*
- *plot_crystal()*

plot_alcove_walk(*word*, *start=None*, *foldings=None*, *color='orange'*, ***options*)

Plot an alcove walk.

INPUT:
- word – list of elements of the index set
- foldings – list of booleans or None (default: None)
- start – an element of this space (default: None for ρ)
- **options – plotting options

See also

- *plot()* for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

An alcove walk of type $A_2^{(1)}$:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: w1 = [0,2,1,2,0,2,1,0,2,1,2,1,2,0,2,0,1,2,0]
sage: p = L.plot_alcoves(bounding_box=5)         # long time (5s)         #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot sage.symbolic
sage: p += L.plot_alcove_walk(w1)       # long time                          #␣
↪needs sage.plot sage.symbolic
sage: p                             # long time                              #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 375 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> w1 = [Integer(0),Integer(2),Integer(1),Integer(2),Integer(0),
↪Integer(2),Integer(1),Integer(0),Integer(2),Integer(1),Integer(2),
↪Integer(1),Integer(2),Integer(0),Integer(2),Integer(0),Integer(1),
↪Integer(2),Integer(0)]
>>> p = L.plot_alcoves(bounding_box=Integer(5))        # long time (5s)  ␣
↪     # needs sage.plot sage.symbolic
>>> p += L.plot_alcove_walk(w1)       # long time                          #␣
↪needs sage.plot sage.symbolic
>>> p                             # long time                              #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 375 graphics primitives
```

The same plot with another alcove walk:

```
sage: w2 = [2,1,2,0,2,0,2,1,2,0,1,2,1,2,1,0,1,2,0,2,0,1,2,0,2]
sage: p += L.plot_alcove_walk(w2, color='orange')       # long time,␣
↪needs sage.plot sage.symbolic
```

```
>>> from sage.all import *
>>> w2 = [Integer(2),Integer(1),Integer(2),Integer(0),Integer(2),
↪Integer(0),Integer(2),Integer(1),Integer(2),Integer(0),Integer(1),
↪Integer(2),Integer(1),Integer(2),Integer(1),Integer(0),Integer(1),
↪Integer(2),Integer(0),Integer(2),Integer(0),Integer(1),Integer(2),
↪Integer(0),Integer(2)]
>>> p += L.plot_alcove_walk(w2, color='orange')       # long time, needs␣
↪sage.plot sage.symbolic
```

And another with some foldings:

```
sage: pic = L.plot_alcoves(bounding_box=3)            # long time,␣
↪needs sage.plot sage.symbolic
sage: pic += L.plot_alcove_walk([0,1,2,0,2,0,1,2,0,1],  # long time (3s),␣
↪needs sage.plot sage.symbolic
....:                           foldings=[False, False, True, False, False,
....:                                     False, True, False, True, False],
....:                           color='green'); pic
Graphics object consisting of 155 graphics primitives
```

```
>>> from sage.all import *
>>> pic = L.plot_alcoves(bounding_box=Integer(3))            # long␣
↪time, needs sage.plot sage.symbolic
>>> pic += L.plot_alcove_walk([Integer(0),Integer(1),Integer(2),
↪Integer(0),Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)],  # long time (3s), needs sage.plot sage.symbolic
...                         foldings=[False, False, True, False, False,
...                                   False, True, False, True, False],
...                         color='green'); pic
Graphics object consisting of 155 graphics primitives
```

plot_alcoves(*alcoves=True*, *alcove_labels=False*, *wireframe=False*, ***options*)

Plot the alcoves and optionally their labels.

INPUT:

- `alcoves` – list of alcoves or `True` (default: `True`)
- `alcove_labels` – boolean or a number specifying at which level to put the label (default: `False`)
- `**options` – plotting options

See also

- *plot()* for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting, and in particular how the alcoves can be specified.

EXAMPLES:

2D plots:

```
sage: RootSystem(["B",2,1]).ambient_space().plot_alcoves()      # long␣
↪time (3s), needs sage.plot sage.symbolic
Graphics object consisting of 228 graphics primitives
```

```
>>> from sage.all import *
>>> RootSystem(["B",Integer(2),Integer(1)]).ambient_space().plot_
↪alcoves()      # long time (3s), needs sage.plot sage.symbolic
Graphics object consisting of 228 graphics primitives
```

3D plots:

```
sage: RootSystem(["A",2,1]).weight_space() .plot_alcoves(affine=False)  #␣
↪long time (3s), needs sage.plot sage.symbolic
Graphics3d Object
sage: RootSystem(["G",2,1]).ambient_space().plot_alcoves(affine=False,␣
↪level=1)  # long time (3s), needs sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(2),Integer(1)]).weight_space() .plot_
↪alcoves(affine=False)  # long time (3s), needs sage.plot sage.symbolic
Graphics3d Object
>>> RootSystem(["G",Integer(2),Integer(1)]).ambient_space().plot_
↪alcoves(affine=False, level=Integer(1))  # long time (3s), needs sage.
↪plot sage.symbolic
Graphics3d Object
```

Here we plot a single alcove:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: W = L.weyl_group()                                                   #␣
↪needs sage.libs.gap
sage: L.plot(alcoves=[W.one()], reflection_hyperplanes=False, bounding_
↪box=2)       # needs sage.libs.gap sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3),Integer(1)]).ambient_space()
>>> W = L.weyl_group()                                                  #␣
↪needs sage.libs.gap
>>> L.plot(alcoves=[W.one()], reflection_hyperplanes=False, bounding_
↪box=Integer(2))       # needs sage.libs.gap sage.plot sage.symbolic
Graphics3d Object
```

plot_bounding_box(***options*)

Plot the bounding box.

INPUT:

- `**options` – plotting options

This is mostly for testing purposes.

See also

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: L.plot_bounding_box()                                                #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> L.plot_bounding_box()                                               #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

plot_coroots(*collection='simple'*, ***options*)

Plot the (simple/classical) coroots of this root lattice.

INPUT:

- `collection` – which coroots to display Can be one of the following:
  - `'simple'` (the default)
  - `'classical'`
  - `'all'`
- `**options` – plotting options

See also

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: RootSystem(["B",3]).ambient_space().plot_coroots()                    #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> RootSystem(["B",Integer(3)]).ambient_space().plot_coroots()              ␣
↪     # needs sage.plot sage.symbolic
Graphics3d Object
```

plot_crystal(*crystal*, *plot_labels=True*, *label_color='black'*, *edge_labels=False*, *circle_size=0.06*, *circle_thickness=1.6*, ***options*)

Plot a finite crystal.

INPUT:

- `crystal` – the finite crystal to plot
- `plot_labels` – boolean (default: `True`); can be one of the following:
  - `True` – use the latex labels
  - `'circles'` – use circles for multiplicity up to 4; if the multiplicity is larger, then it uses the multiplicity
  - `'multiplicities'` – use the multiplicities
- `label_color` – (default: `'black'`) the color of the labels
- `edge_labels` – boolean (default: `False`); if `True`, then draw in the edge label
- `circle_size` – (default: 0.06) the size of the circles
- `circle_thickness` – (default: 1.6) the thinkness of the extra rings of circles
- `**options` – plotting options

See also

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: # needs sage.combinat sage.plot sage.symbolic
sage: L = RootSystem(['A',2]).ambient_space()
sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: L.plot_crystal(C, plot_labels='multiplicities')
Graphics object consisting of 15 graphics primitives
sage: C = crystals.Tableaux(['A',2], shape=[8,4])
sage: p = L.plot_crystal(C, plot_labels='circles')
sage: p.show(figsize=15)
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.plot sage.symbolic
>>> L = RootSystem(['A',Integer(2)]).ambient_space()
>>> C = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2),Integer(1)])
>>> L.plot_crystal(C, plot_labels='multiplicities')
Graphics object consisting of 15 graphics primitives
>>> C = crystals.Tableaux(['A',Integer(2)], shape=[Integer(8),Integer(4)])
>>> p = L.plot_crystal(C, plot_labels='circles')
>>> p.show(figsize=Integer(15))
```

A 3-dimensional example:

```
sage: L = RootSystem(['B',3]).ambient_space()
sage: C = crystals.Tableaux(['B',3], shape=[2,1])                        #␣
↪needs sage.combinat
sage: L.plot_crystal(C, plot_labels='circles',  # long time              #␣
↪needs sage.combinat sage.plot sage.symbolic
....:                edge_labels=True)
Graphics3d Object
```

```
>>> from sage.all import *
>>> L = RootSystem(['B',Integer(3)]).ambient_space()
>>> C = crystals.Tableaux(['B',Integer(3)], shape=[Integer(2),
↪Integer(1)])                               # needs sage.combinat
>>> L.plot_crystal(C, plot_labels='circles',  # long time              #␣
↪needs sage.combinat sage.plot sage.symbolic
...                edge_labels=True)
Graphics3d Object
```

plot_fundamental_chamber(*style='normal'*, ***options*)

Plot the (classical) fundamental chamber.

INPUT:

- `style` – `'normal'` or `'classical'` (default: `'normal'`)
- `**options` – plotting options

See also

- *plot()* for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

2D plots:

```
sage: RootSystem(["B",2]).ambient_space().plot_fundamental_chamber()     #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
sage: RootSystem(["B",2,1]).ambient_space().plot_fundamental_chamber()   #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
sage: RootSystem(["B",2,1]).ambient_space().plot_fundamental_chamber(
```

(continues on next page)

(continued from previous page)

```
↪"classical")    # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> RootSystem(["B",Integer(2)]).ambient_space().plot_fundamental_
↪chamber()     # needs sage.plot
Graphics object consisting of 1 graphics primitive
>>> RootSystem(["B",Integer(2),Integer(1)]).ambient_space().plot_
↪fundamental_chamber()   # needs sage.plot
Graphics object consisting of 1 graphics primitive
>>> RootSystem(["B",Integer(2),Integer(1)]).ambient_space().plot_
↪fundamental_chamber("classical")    # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

3D plots:

```
sage: RootSystem(["A",3,1]).weight_space() .plot_fundamental_chamber()   #␣
↪needs sage.plot
Graphics3d Object
sage: RootSystem(["B",3,1]).ambient_space().plot_fundamental_chamber()   #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3),Integer(1)]).weight_space() .plot_
↪fundamental_chamber()   # needs sage.plot
Graphics3d Object
>>> RootSystem(["B",Integer(3),Integer(1)]).ambient_space().plot_
↪fundamental_chamber()   # needs sage.plot
Graphics3d Object
```

This feature is currently not available in the root lattice/space:

```
sage: list(RootSystem(["A",2]).root_lattice().plot_fundamental_chamber())␣
↪          # needs sage.plot
Traceback (most recent call last):
...
TypeError: classical fundamental chamber not yet available in the root␣
↪lattice
```

```
>>> from sage.all import *
>>> list(RootSystem(["A",Integer(2)]).root_lattice().plot_fundamental_
↪chamber())           # needs sage.plot
Traceback (most recent call last):
...
TypeError: classical fundamental chamber not yet available in the root␣
↪lattice
```

plot_fundamental_weights(***options*)

Plot the fundamental weights of this root lattice.

INPUT:

- `**options` – plotting options

See also

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: RootSystem(["B",3]).ambient_space().plot_fundamental_weights()     #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> RootSystem(["B",Integer(3)]).ambient_space().plot_fundamental_
↪weights()     # needs sage.plot
Graphics3d Object
```

plot_hedron(***options*)

Plot the polyhedron whose vertices are given by the orbit of ρ.

In type A, this is the usual permutohedron.

See also

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: # needs sage.plot sage.symbolic
sage: RootSystem(["A",2]).ambient_space().plot_hedron()
Graphics object consisting of 8 graphics primitives
sage: RootSystem(["A",3]).ambient_space().plot_hedron()
Graphics3d Object
sage: RootSystem(["B",3]).ambient_space().plot_hedron()
Graphics3d Object
sage: RootSystem(["C",3]).ambient_space().plot_hedron()
Graphics3d Object
sage: RootSystem(["D",3]).ambient_space().plot_hedron()
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.plot sage.symbolic
>>> RootSystem(["A",Integer(2)]).ambient_space().plot_hedron()
Graphics object consisting of 8 graphics primitives
>>> RootSystem(["A",Integer(3)]).ambient_space().plot_hedron()
Graphics3d Object
>>> RootSystem(["B",Integer(3)]).ambient_space().plot_hedron()
Graphics3d Object
>>> RootSystem(["C",Integer(3)]).ambient_space().plot_hedron()
```

(continues on next page)

(continued from previous page)

```
Graphics3d Object
>>> RootSystem(["D",Integer(3)]).ambient_space().plot_hedron()
Graphics3d Object
```

Surprise: polyhedra of large dimension know how to project themselves nicely:

```
sage: RootSystem(["F",4]).ambient_space().plot_hedron()        # long␣
↪time, needs sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> RootSystem(["F",Integer(4)]).ambient_space().plot_hedron()        #␣
↪long time, needs sage.plot sage.symbolic
Graphics3d Object
```

plot_ls_paths(*paths*, *plot_labels=None*, *colored_labels=True*, ***options*)

Plot LS paths.

INPUT:

- `paths` – a finite crystal or list of LS paths
- `plot_labels` – (default: `None`) the distance to plot the LS labels from the endpoint of the path; set to `None` to not display the labels
- `colored_labels` – boolean (default: `True`); if `True`, then color the labels the same color as the LS path
- `**options` – plotting options

See also

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: B = crystals.LSPaths(['A',2], [1,1])                               #␣
↪needs sage.combinat
sage: L = RootSystem(['A',2]).ambient_space()
sage: L.plot_fundamental_weights() + L.plot_ls_paths(B)                  #␣
↪needs sage.combinat sage.plot sage.symbolic
Graphics object consisting of 14 graphics primitives
```

```
>>> from sage.all import *
>>> B = crystals.LSPaths(['A',Integer(2)], [Integer(1),Integer(1)])       ␣
↪                       # needs sage.combinat
>>> L = RootSystem(['A',Integer(2)]).ambient_space()
>>> L.plot_fundamental_weights() + L.plot_ls_paths(B)                  #␣
↪needs sage.combinat sage.plot sage.symbolic
Graphics object consisting of 14 graphics primitives
```

This also works in 3 dimensions:

```
sage: B = crystals.LSPaths(['B',3], [2,0,0])                                 #␣
↪needs sage.combinat
sage: L = RootSystem(['B',3]).ambient_space()
sage: L.plot_ls_paths(B)                                                     #␣
↪needs sage.combinat sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> B = crystals.LSPaths(['B',Integer(3)], [Integer(2),Integer(0),
↪Integer(0)])                            # needs sage.combinat
>>> L = RootSystem(['B',Integer(3)]).ambient_space()
>>> L.plot_ls_paths(B)                                                      #␣
↪needs sage.combinat sage.plot sage.symbolic
Graphics3d Object
```

plot_mv_polytope(*mv_polytope*, *mark_endpoints=True*, *circle_size=0.06*, *circle_thickness=1.6*, *wireframe='blue'*, *fill='green'*, *alpha=1*, ***options*)

Plot an MV polytope.

INPUT:

- `mv_polytope` – an MV polytope
- `mark_endpoints` – boolean (default: `True`); mark the endpoints of the MV polytope
- `circle_size` – (default: 0.06) the size of the circles
- `circle_thickness` – (default: 1.6) the thinkness of the extra rings of circles
- `wireframe` – (default: `'blue'`) color to draw the wireframe of the polytope with
- `fill` – (default: `'green'`) color to fill the polytope with
- `alpha` – (default: 1) the alpha value (opacity) of the fill
- `**options` – plotting options

See also

- *plot()* for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: B = crystals.infinity.MVPolytopes(['C',2])                             #␣
↪needs sage.combinat
sage: L = RootSystem(['C',2]).ambient_space()
sage: p = B.highest_weight_vector().f_string([1,2,1,2])                      #␣
↪needs sage.combinat
sage: L.plot_fundamental_weights() + L.plot_mv_polytope(p)                   #␣
↪needs sage.combinat sage.geometry.polyhedron sage.plot sage.symbolic
Graphics object consisting of 14 graphics primitives
```

```
>>> from sage.all import *
>>> B = crystals.infinity.MVPolytopes(['C',Integer(2)])                      ␣
↪     # needs sage.combinat
>>> L = RootSystem(['C',Integer(2)]).ambient_space()
>>> p = B.highest_weight_vector().f_string([Integer(1),Integer(2),
↪Integer(1),Integer(2)])                 # needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
>>> L.plot_fundamental_weights() + L.plot_mv_polytope(p)              #␣
↪needs sage.combinat sage.geometry.polyhedron sage.plot sage.symbolic
Graphics object consisting of 14 graphics primitives
```

This also works in 3 dimensions:

```
sage: B = crystals.infinity.MVPolytopes(['A',3])                       #␣
↪needs sage.combinat
sage: L = RootSystem(['A',3]).ambient_space()
sage: p = B.highest_weight_vector().f_string([2,1,3,2])                #␣
↪needs sage.combinat
sage: L.plot_mv_polytope(p)                                            #␣
↪needs sage.combinat sage.geometry.polyhedron sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> B = crystals.infinity.MVPolytopes(['A',Integer(3)])                 ␣
↪     # needs sage.combinat
>>> L = RootSystem(['A',Integer(3)]).ambient_space()
>>> p = B.highest_weight_vector().f_string([Integer(2),Integer(1),
↪Integer(3),Integer(2)])                  # needs sage.combinat
>>> L.plot_mv_polytope(p)                                              #␣
↪needs sage.combinat sage.geometry.polyhedron sage.plot sage.symbolic
Graphics3d Object
```

plot_parse_options(***args*)

Return an option object to be used for root system plotting.

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: options = L.plot_parse_options(); options                        #␣
↪needs sage.symbolic
<sage.combinat.root_system.plot.PlotOptions object at ...>
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space()
>>> options = L.plot_parse_options(); options                          #␣
↪needs sage.symbolic
<sage.combinat.root_system.plot.PlotOptions object at ...>
```

See also

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

plot_reflection_hyperplanes(*collection='simple'*, ***options*)

Plot the simple reflection hyperplanes.

INPUT:

- `collection` – which reflection hyperplanes to display Can be one of the following:

  - 'simple' (the default)
  - 'classical'
  - 'all'
- `**options` – plotting options

See also

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: # needs sage.plot sage.symbolic
sage: RootSystem(["A",2,1]).ambient_space().plot_reflection_hyperplanes()
Graphics object consisting of 6 graphics primitives
sage: RootSystem(["G",2,1]).ambient_space().plot_reflection_hyperplanes()
Graphics object consisting of 6 graphics primitives
sage: RootSystem(["A",3]).weight_space().plot_reflection_hyperplanes()
Graphics3d Object
sage: RootSystem(["B",3]).ambient_space().plot_reflection_hyperplanes()
Graphics3d Object
sage: RootSystem(["A",3,1]).weight_space().plot_reflection_hyperplanes()
Graphics3d Object
sage: RootSystem(["B",3,1]).ambient_space().plot_reflection_hyperplanes()
Graphics3d Object
sage: RootSystem(["A",2,1]).weight_space().plot_reflection_
↪hyperplanes(affine=False, level=1)
Graphics3d Object
sage: RootSystem(["A",2]).root_lattice().plot_reflection_hyperplanes()
Graphics object consisting of 4 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot sage.symbolic
>>> RootSystem(["A",Integer(2),Integer(1)]).ambient_space().plot_
↪reflection_hyperplanes()
Graphics object consisting of 6 graphics primitives
>>> RootSystem(["G",Integer(2),Integer(1)]).ambient_space().plot_
↪reflection_hyperplanes()
Graphics object consisting of 6 graphics primitives
>>> RootSystem(["A",Integer(3)]).weight_space().plot_reflection_
↪hyperplanes()
Graphics3d Object
>>> RootSystem(["B",Integer(3)]).ambient_space().plot_reflection_
↪hyperplanes()
Graphics3d Object
>>> RootSystem(["A",Integer(3),Integer(1)]).weight_space().plot_
↪reflection_hyperplanes()
Graphics3d Object
>>> RootSystem(["B",Integer(3),Integer(1)]).ambient_space().plot_
↪reflection_hyperplanes()
Graphics3d Object
>>> RootSystem(["A",Integer(2),Integer(1)]).weight_space().plot_
```

(continues on next page)

(continued from previous page)

```
↪reflection_hyperplanes(affine=False, level=Integer(1))
Graphics3d Object
>>> RootSystem(["A",Integer(2)]).root_lattice().plot_reflection_
↪hyperplanes()
Graphics object consisting of 4 graphics primitives
```

Todo

Provide an option for transparency?

plot_roots(*collection='simple'*, ***options*)

Plot the (simple/classical) roots of this root lattice.

INPUT:

- `collection` – which roots to display can be one of the following:
  - `'simple'` (the default)
  - `'classical'`
  - `'all'`
- `**options` – plotting options

See also

- *plot()* for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: RootSystem(["B",3]).ambient_space().plot_roots()                      #␣
↪needs sage.plot
Graphics3d Object
sage: RootSystem(["B",3]).ambient_space().plot_roots("all")                 #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> RootSystem(["B",Integer(3)]).ambient_space().plot_roots()              ␣
↪      # needs sage.plot
Graphics3d Object
>>> RootSystem(["B",Integer(3)]).ambient_space().plot_roots("all")         ␣
↪      # needs sage.plot
Graphics3d Object
```

positive_imaginary_roots()

Return the positive imaginary roots of `self`.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).root_lattice()
sage: L.positive_imaginary_roots()
()
```

(continues on next page)

(continued from previous page)

```
sage: L = RootSystem(['A',3,1]).root_lattice()
sage: PIR = L.positive_imaginary_roots(); PIR                               #␣
↪needs sage.graphs
Positive imaginary roots of type ['A', 3, 1]
sage: [PIR.unrank(i) for i in range(5)]                                     #␣
↪needs sage.graphs
[alpha[0] + alpha[1] + alpha[2] + alpha[3],
 2*alpha[0] + 2*alpha[1] + 2*alpha[2] + 2*alpha[3],
 3*alpha[0] + 3*alpha[1] + 3*alpha[2] + 3*alpha[3],
 4*alpha[0] + 4*alpha[1] + 4*alpha[2] + 4*alpha[3],
 5*alpha[0] + 5*alpha[1] + 5*alpha[2] + 5*alpha[3]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(3)]).root_lattice()
>>> L.positive_imaginary_roots()
()

>>> L = RootSystem(['A',Integer(3),Integer(1)]).root_lattice()
>>> PIR = L.positive_imaginary_roots(); PIR                               #␣
↪needs sage.graphs
Positive imaginary roots of type ['A', 3, 1]
>>> [PIR.unrank(i) for i in range(Integer(5))]                              ␣
↪       # needs sage.graphs
[alpha[0] + alpha[1] + alpha[2] + alpha[3],
 2*alpha[0] + 2*alpha[1] + 2*alpha[2] + 2*alpha[3],
 3*alpha[0] + 3*alpha[1] + 3*alpha[2] + 3*alpha[3],
 4*alpha[0] + 4*alpha[1] + 4*alpha[2] + 4*alpha[3],
 5*alpha[0] + 5*alpha[1] + 5*alpha[2] + 5*alpha[3]]
```

positive_real_roots()

Return the positive real roots of `self`.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).root_lattice()
sage: sorted(L.positive_real_roots())                                       #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3], alpha[3]]

sage: L = RootSystem(['A',3,1]).root_lattice()
sage: PRR = L.positive_real_roots(); PRR                                    #␣
↪needs sage.graphs
Positive real roots of type ['A', 3, 1]
sage: [PRR.unrank(i) for i in range(10)]                                    #␣
↪needs sage.graphs
[alpha[1],
 alpha[2],
 alpha[3],
 alpha[1] + alpha[2],
 alpha[2] + alpha[3],
```

(continues on next page)

(continued from previous page)

```
 alpha[1] + alpha[2] + alpha[3],
 alpha[0] + 2*alpha[1] + alpha[2] + alpha[3],
 alpha[0] + alpha[1] + 2*alpha[2] + alpha[3],
 alpha[0] + alpha[1] + alpha[2] + 2*alpha[3],
 alpha[0] + 2*alpha[1] + 2*alpha[2] + alpha[3]]

sage: Q = RootSystem(['A',4,2]).root_lattice()
sage: PR = Q.positive_roots()                                               #␣
↪needs sage.graphs
sage: [PR.unrank(i) for i in range(5)]                                      #␣
↪needs sage.graphs
[alpha[1],
 alpha[2],
 alpha[1] + alpha[2],
 2*alpha[1] + alpha[2],
 alpha[0] + alpha[1] + alpha[2]]

sage: Q = RootSystem(['D',3,2]).root_lattice()
sage: PR = Q.positive_roots()                                               #␣
↪needs sage.graphs
sage: [PR.unrank(i) for i in range(5)]                                      #␣
↪needs sage.graphs
[alpha[1],
 alpha[2],
 alpha[1] + 2*alpha[2],
 alpha[1] + alpha[2],
 alpha[0] + alpha[1] + 2*alpha[2]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(3)]).root_lattice()
>>> sorted(L.positive_real_roots())                                         #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3], alpha[3]]

>>> L = RootSystem(['A',Integer(3),Integer(1)]).root_lattice()
>>> PRR = L.positive_real_roots(); PRR                                      #␣
↪needs sage.graphs
Positive real roots of type ['A', 3, 1]
>>> [PRR.unrank(i) for i in range(Integer(10))]                             ␣
↪     # needs sage.graphs
[alpha[1],
 alpha[2],
 alpha[3],
 alpha[1] + alpha[2],
 alpha[2] + alpha[3],
 alpha[1] + alpha[2] + alpha[3],
 alpha[0] + 2*alpha[1] + alpha[2] + alpha[3],
 alpha[0] + alpha[1] + 2*alpha[2] + alpha[3],
 alpha[0] + alpha[1] + alpha[2] + 2*alpha[3],
 alpha[0] + 2*alpha[1] + 2*alpha[2] + alpha[3]]
```

(continues on next page)

(continued from previous page)

```
>>> Q = RootSystem(['A',Integer(4),Integer(2)]).root_lattice()
>>> PR = Q.positive_roots()                                              #␣
↪needs sage.graphs
>>> [PR.unrank(i) for i in range(Integer(5))]                            ␣
↪     # needs sage.graphs
[alpha[1],
 alpha[2],
 alpha[1] + alpha[2],
 2*alpha[1] + alpha[2],
 alpha[0] + alpha[1] + alpha[2]]

>>> Q = RootSystem(['D',Integer(3),Integer(2)]).root_lattice()
>>> PR = Q.positive_roots()                                              #␣
↪needs sage.graphs
>>> [PR.unrank(i) for i in range(Integer(5))]                            ␣
↪     # needs sage.graphs
[alpha[1],
 alpha[2],
 alpha[1] + 2*alpha[2],
 alpha[1] + alpha[2],
 alpha[0] + alpha[1] + 2*alpha[2]]
```

positive_roots(*index_set=None*)

Return the positive roots of `self`.

If `index_set` is not `None`, returns the positive roots of the parabolic subsystem with simple roots in `index_set`.

Algorithm for finite type: generate them from the simple roots by applying successive reflections toward the positive chamber.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).root_lattice()
sage: sorted(L.positive_roots())                                         #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2],
 alpha[1] + alpha[2] + alpha[3], alpha[2],
 alpha[2] + alpha[3], alpha[3]]
sage: sorted(L.positive_roots((1,2)))                                    #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2], alpha[2]]
sage: sorted(L.positive_roots(()))                                       #␣
↪needs sage.graphs
[]

sage: L = RootSystem(['A',3,1]).root_lattice()
sage: PR = L.positive_roots(); PR                                        #␣
↪needs sage.graphs
Disjoint union of Family (Positive real roots of type ['A', 3, 1],
    Positive imaginary roots of type ['A', 3, 1])
sage: [PR.unrank(i) for i in range(10)]                                  #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
[alpha[1],
 alpha[2],
 alpha[3],
 alpha[1] + alpha[2],
 alpha[2] + alpha[3],
 alpha[1] + alpha[2] + alpha[3],
 alpha[0] + 2*alpha[1] + alpha[2] + alpha[3],
 alpha[0] + alpha[1] + 2*alpha[2] + alpha[3],
 alpha[0] + alpha[1] + alpha[2] + 2*alpha[3],
 alpha[0] + 2*alpha[1] + 2*alpha[2] + alpha[3]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(3)]).root_lattice()
>>> sorted(L.positive_roots())                                              #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2],
 alpha[1] + alpha[2] + alpha[3], alpha[2],
 alpha[2] + alpha[3], alpha[3]]
>>> sorted(L.positive_roots((Integer(1),Integer(2))))                         ␣
↪              # needs sage.graphs
[alpha[1], alpha[1] + alpha[2], alpha[2]]
>>> sorted(L.positive_roots(()))                                            #␣
↪needs sage.graphs
[]

>>> L = RootSystem(['A',Integer(3),Integer(1)]).root_lattice()
>>> PR = L.positive_roots(); PR                                             #␣
↪needs sage.graphs
Disjoint union of Family (Positive real roots of type ['A', 3, 1],
    Positive imaginary roots of type ['A', 3, 1])
>>> [PR.unrank(i) for i in range(Integer(10))]                               ␣
↪     # needs sage.graphs
[alpha[1],
 alpha[2],
 alpha[3],
 alpha[1] + alpha[2],
 alpha[2] + alpha[3],
 alpha[1] + alpha[2] + alpha[3],
 alpha[0] + 2*alpha[1] + alpha[2] + alpha[3],
 alpha[0] + alpha[1] + 2*alpha[2] + alpha[3],
 alpha[0] + alpha[1] + alpha[2] + 2*alpha[3],
 alpha[0] + 2*alpha[1] + 2*alpha[2] + alpha[3]]
```

positive_roots_by_height(*increasing=True*)

Return a list of positive roots in increasing order by height.

If `increasing` is False, returns them in decreasing order.

> **Warning**
>
> Raise an error if the Cartan type is not finite.

EXAMPLES:

```
sage: L = RootSystem(['C',2]).root_lattice()
sage: L.positive_roots_by_height()                                         #␣
↪needs sage.graphs
[alpha[2], alpha[1], alpha[1] + alpha[2], 2*alpha[1] + alpha[2]]
sage: L.positive_roots_by_height(increasing=False)                         #␣
↪needs sage.graphs
[2*alpha[1] + alpha[2], alpha[1] + alpha[2], alpha[2], alpha[1]]

sage: L = RootSystem(['A',2,1]).root_lattice()
sage: L.positive_roots_by_height()                                         #␣
↪needs sage.graphs
Traceback (most recent call last):
...
NotImplementedError: Only implemented for finite Cartan type
```

```
>>> from sage.all import *
>>> L = RootSystem(['C',Integer(2)]).root_lattice()
>>> L.positive_roots_by_height()                                         #␣
↪needs sage.graphs
[alpha[2], alpha[1], alpha[1] + alpha[2], 2*alpha[1] + alpha[2]]
>>> L.positive_roots_by_height(increasing=False)                         #␣
↪needs sage.graphs
[2*alpha[1] + alpha[2], alpha[1] + alpha[2], alpha[2], alpha[1]]

>>> L = RootSystem(['A',Integer(2),Integer(1)]).root_lattice()
>>> L.positive_roots_by_height()                                         #␣
↪needs sage.graphs
Traceback (most recent call last):
...
NotImplementedError: Only implemented for finite Cartan type
```

positive_roots_nonparabolic(*index_set=None*)

Return the set of positive roots outside the parabolic subsystem with Dynkin node set `index_set`.

INPUT:

- `index_set` – (default: `None`) the Dynkin node set of the parabolic subsystem. It should be a tuple. The default value implies the entire Dynkin node set

EXAMPLES:

```
sage: # needs sage.graphs
sage: lattice = RootSystem(['A',3]).root_lattice()
sage: sorted(lattice.positive_roots_nonparabolic((1,3)), key=str)
[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3]]
sage: sorted(lattice.positive_roots_nonparabolic((2,3)), key=str)
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]]
sage: lattice.positive_roots_nonparabolic()
[]
sage: lattice.positive_roots_nonparabolic((1,2,3))
[]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> lattice = RootSystem(['A',Integer(3)]).root_lattice()
>>> sorted(lattice.positive_roots_nonparabolic((Integer(1),Integer(3))),␣
↪key=str)
[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3]]
>>> sorted(lattice.positive_roots_nonparabolic((Integer(2),Integer(3))),␣
↪key=str)
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]]
>>> lattice.positive_roots_nonparabolic()
[]
>>> lattice.positive_roots_nonparabolic((Integer(1),Integer(2),
↪Integer(3)))
[]
```

Warning

This returns an error if the Cartan type is not finite.

positive_roots_nonparabolic_sum(*index_set=None*)

Return the sum of positive roots outside the parabolic subsystem with Dynkin node set `index_set`.

INPUT:

- `index_set` – (default: `None`) the Dynkin node set of the parabolic subsystem. It should be a tuple. The default value implies the entire Dynkin node set

EXAMPLES:

```
sage: # needs sage.graphs
sage: lattice = RootSystem(['A',3]).root_lattice()
sage: lattice.positive_roots_nonparabolic_sum((1,3))
2*alpha[1] + 4*alpha[2] + 2*alpha[3]
sage: lattice.positive_roots_nonparabolic_sum((2,3))
3*alpha[1] + 2*alpha[2] + alpha[3]
sage: lattice.positive_roots_nonparabolic_sum(())
3*alpha[1] + 4*alpha[2] + 3*alpha[3]
sage: lattice.positive_roots_nonparabolic_sum()
0
sage: lattice.positive_roots_nonparabolic_sum((1,2,3))
0
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> lattice = RootSystem(['A',Integer(3)]).root_lattice()
>>> lattice.positive_roots_nonparabolic_sum((Integer(1),Integer(3)))
2*alpha[1] + 4*alpha[2] + 2*alpha[3]
>>> lattice.positive_roots_nonparabolic_sum((Integer(2),Integer(3)))
3*alpha[1] + 2*alpha[2] + alpha[3]
>>> lattice.positive_roots_nonparabolic_sum(())
3*alpha[1] + 4*alpha[2] + 3*alpha[3]
>>> lattice.positive_roots_nonparabolic_sum()
```

(continues on next page)

(continued from previous page)

```
0
>>> lattice.positive_roots_nonparabolic_sum((Integer(1),Integer(2),
↪Integer(3)))
0
```

> **Warning**
>
> This returns an error if the Cartan type is not finite.

positive_roots_parabolic(*index_set=None*)

Return the set of positive roots for the parabolic subsystem with Dynkin node set `index_set`.

INPUT:

- `index_set` – (default: `None`) the Dynkin node set of the parabolic subsystem. It should be a tuple. The default value implies the entire Dynkin node set

EXAMPLES:

```
sage: lattice =  RootSystem(['A',3]).root_lattice()
sage: sorted(lattice.positive_roots_parabolic((1,3)), key=str)            #␣
↪needs sage.graphs
[alpha[1], alpha[3]]
sage: sorted(lattice.positive_roots_parabolic((2,3)), key=str)            #␣
↪needs sage.graphs
[alpha[2], alpha[2] + alpha[3], alpha[3]]
sage: sorted(lattice.positive_roots_parabolic(), key=str)                 #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3], alpha[3]]
```

```
>>> from sage.all import *
>>> lattice =  RootSystem(['A',Integer(3)]).root_lattice()
>>> sorted(lattice.positive_roots_parabolic((Integer(1),Integer(3))),␣
↪key=str)           # needs sage.graphs
[alpha[1], alpha[3]]
>>> sorted(lattice.positive_roots_parabolic((Integer(2),Integer(3))),␣
↪key=str)           # needs sage.graphs
[alpha[2], alpha[2] + alpha[3], alpha[3]]
>>> sorted(lattice.positive_roots_parabolic(), key=str)                 #␣
↪needs sage.graphs
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3], alpha[3]]
```

> **Warning**
>
> This returns an error if the Cartan type is not finite.

projection(*root*, *coroot=None*, *to_negative=True*)

Return the projection along the `root`, and across the hyperplane defined by `coroot`, as a function π from `self` to `self`.

π is a half-linear map which stabilizes the negative half space and acts by reflection on the positive half space.

If `to_negative` is `False`, then project onto the positive half space instead.

EXAMPLES:

```
sage: # needs sage.graphs
sage: space = RootSystem(['A',2]).weight_lattice()
sage: x = space.simple_roots()[1]
sage: y = space.simple_coroots()[1]
sage: pi = space.projection(x,y)
sage: x
2*Lambda[1] - Lambda[2]
sage: pi(x)
-2*Lambda[1] + Lambda[2]
sage: pi(-x)
-2*Lambda[1] + Lambda[2]
sage: pi = space.projection(x,y,False)
sage: pi(-x)
2*Lambda[1] - Lambda[2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> space = RootSystem(['A',Integer(2)]).weight_lattice()
>>> x = space.simple_roots()[Integer(1)]
>>> y = space.simple_coroots()[Integer(1)]
>>> pi = space.projection(x,y)
>>> x
2*Lambda[1] - Lambda[2]
>>> pi(x)
-2*Lambda[1] + Lambda[2]
>>> pi(-x)
-2*Lambda[1] + Lambda[2]
>>> pi = space.projection(x,y,False)
>>> pi(-x)
2*Lambda[1] - Lambda[2]
```

reflection(*root*, *coroot=None*)

Return the reflection along the `root`, and across the hyperplane defined by `coroot`, as a function from `self` to `self`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: space = RootSystem(['A',2]).weight_lattice()
sage: x = space.simple_roots()[1]
sage: y = space.simple_coroots()[1]
sage: s = space.reflection(x,y)
sage: x
2*Lambda[1] - Lambda[2]
sage: s(x)
-2*Lambda[1] + Lambda[2]
sage: s(-x)
2*Lambda[1] - Lambda[2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> space = RootSystem(['A',Integer(2)]).weight_lattice()
>>> x = space.simple_roots()[Integer(1)]
>>> y = space.simple_coroots()[Integer(1)]
>>> s = space.reflection(x,y)
>>> x
2*Lambda[1] - Lambda[2]
>>> s(x)
-2*Lambda[1] + Lambda[2]
>>> s(-x)
2*Lambda[1] - Lambda[2]
```

root_poset(*restricted=False*, *facade=False*)

Return the (restricted) root poset associated to `self`.

The elements are given by the positive roots (resp. non-simple, positive roots), and $\alpha \leq \beta$ iff $\beta - \alpha$ is a nonnegative linear combination of simple roots.

INPUT:

- `restricted` – boolean (default: `False`); if `True`, only non-simple roots are considered.
- `facade` – boolean (default: `False`); passes facade option to the poset generator

EXAMPLES:

```
sage: # needs sage.graphs
sage: Phi = RootSystem(['A',1]).root_poset(); Phi
Finite poset containing 1 elements
sage: Phi.cover_relations()
[]
sage: Phi = RootSystem(['A',2]).root_poset(); Phi
Finite poset containing 3 elements
sage: sorted(Phi.cover_relations(), key=str)
[[alpha[1], alpha[1] + alpha[2]], [alpha[2], alpha[1] + alpha[2]]]
sage: Phi = RootSystem(['A',3]).root_poset(restricted=True); Phi
Finite poset containing 3 elements
sage: sorted(Phi.cover_relations(), key=str)
[[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]],
 [alpha[2] + alpha[3], alpha[1] + alpha[2] + alpha[3]]]
sage: Phi = RootSystem(['B',2]).root_poset(); Phi
Finite poset containing 4 elements
sage: sorted(Phi.cover_relations(), key=str)
[[alpha[1] + alpha[2], alpha[1] + 2*alpha[2]],
 [alpha[1], alpha[1] + alpha[2]],
 [alpha[2], alpha[1] + alpha[2]]]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> Phi = RootSystem(['A',Integer(1)]).root_poset(); Phi
Finite poset containing 1 elements
>>> Phi.cover_relations()
[]
>>> Phi = RootSystem(['A',Integer(2)]).root_poset(); Phi
Finite poset containing 3 elements
```

(continues on next page)

(continued from previous page)

```
>>> sorted(Phi.cover_relations(), key=str)
[[alpha[1], alpha[1] + alpha[2]], [alpha[2], alpha[1] + alpha[2]]]
>>> Phi = RootSystem(['A',Integer(3)]).root_poset(restricted=True); Phi
Finite poset containing 3 elements
>>> sorted(Phi.cover_relations(), key=str)
[[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]],
 [alpha[2] + alpha[3], alpha[1] + alpha[2] + alpha[3]]]
>>> Phi = RootSystem(['B',Integer(2)]).root_poset(); Phi
Finite poset containing 4 elements
>>> sorted(Phi.cover_relations(), key=str)
[[alpha[1] + alpha[2], alpha[1] + 2*alpha[2]],
 [alpha[1], alpha[1] + alpha[2]],
 [alpha[2], alpha[1] + alpha[2]]]
```

roots()

Return the roots of `self`.

EXAMPLES:

```
sage: RootSystem(['A',2]).ambient_lattice().roots()
[(1, -1, 0), (1, 0, -1), (0, 1, -1), (-1, 1, 0), (-1, 0, 1), (0, -1, 1)]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(2)]).ambient_lattice().roots()
[(1, -1, 0), (1, 0, -1), (0, 1, -1), (-1, 1, 0), (-1, 0, 1), (0, -1, 1)]
```

This matches with Wikipedia article Root_systems:

```
sage: for T in CartanType.samples(finite=True, crystallographic=True):   #␣
↪needs sage.graphs
....:     print("%s %3s %3s"%(T, len(RootSystem(T).root_lattice().
↪roots()),
....:                            len(RootSystem(T).weight_lattice().
↪roots())))
['A', 1]   2   2
['A', 5]  30  30
['B', 1]   2   2
['B', 5]  50  50
['C', 1]   2   2
['C', 5]  50  50
['D', 2]   4   4
['D', 3]  12  12
['D', 5]  40  40
['E', 6]  72  72
['E', 7] 126 126
['E', 8] 240 240
['F', 4]  48  48
['G', 2]  12  12
```

```
>>> from sage.all import *
>>> for T in CartanType.samples(finite=True, crystallographic=True):   #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
...     print("%s %3s %3s"%(T, len(RootSystem(T).root_lattice().roots()),
...                              len(RootSystem(T).weight_lattice().
↪roots())))
['A', 1]   2   2
['A', 5]  30  30
['B', 1]   2   2
['B', 5]  50  50
['C', 1]   2   2
['C', 5]  50  50
['D', 2]   4   4
['D', 3]  12  12
['D', 5]  40  40
['E', 6]  72  72
['E', 7] 126 126
['E', 8] 240 240
['F', 4]  48  48
['G', 2]  12  12
```

Todo

The result should be an enumerated set, and handle infinite root systems.

s()

Return the family $(s_i)_{i\in I}$ of the simple reflections of this root system.

EXAMPLES:

```
sage: r = RootSystem(["A", 2]).root_lattice()
sage: s = r.simple_reflections()
sage: s[1]( r.simple_root(1) )                                          #␣
↪needs sage.graphs
-alpha[1]
```

```
>>> from sage.all import *
>>> r = RootSystem(["A", Integer(2)]).root_lattice()
>>> s = r.simple_reflections()
>>> s[Integer(1)]( r.simple_root(Integer(1)) )                          ␣
↪              # needs sage.graphs
-alpha[1]
```

short_roots()

Return a list of the short roots of `self`.

EXAMPLES:

```
sage: L = RootSystem(['B',3]).root_lattice()
sage: sorted(L.short_roots())                                           #␣
↪needs sage.graphs
[-alpha[1] - alpha[2] - alpha[3],
 alpha[1] + alpha[2] + alpha[3],
 -alpha[2] - alpha[3],
```

(continues on next page)

(continued from previous page)

```
 alpha[2] + alpha[3],
 -alpha[3],
 alpha[3]]
```

```
>>> from sage.all import *
>>> L = RootSystem(['B',Integer(3)]).root_lattice()
>>> sorted(L.short_roots())                                              #␣
↪needs sage.graphs
[-alpha[1] - alpha[2] - alpha[3],
 alpha[1] + alpha[2] + alpha[3],
 -alpha[2] - alpha[3],
 alpha[2] + alpha[3],
 -alpha[3],
 alpha[3]]
```

simple_coroot(*i*)

Return the i-th simple coroot.

EXAMPLES:

```
sage: RootSystem(['A',2]).root_lattice().simple_coroot(1)
alphacheck[1]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(2)]).root_lattice().simple_coroot(Integer(1))
alphacheck[1]
```

simple_coroots()

Return the family $(\alpha_i^\vee)_{i\in I}$ of the simple coroots.

EXAMPLES:

```
sage: alphacheck = RootSystem(['A',3]).root_lattice().simple_coroots()
sage: [alphacheck[i] for i in [1, 2, 3]]
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

```
>>> from sage.all import *
>>> alphacheck = RootSystem(['A',Integer(3)]).root_lattice().simple_
↪coroots()
>>> [alphacheck[i] for i in [Integer(1), Integer(2), Integer(3)]]
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

simple_projection(*i*, *to_negative=True*)

Return the projection along the i-th simple root, and across the hyperplane define by the i-th simple coroot, as a function from `self` to `self`.

INPUT:

- `i` – an element of the index set of `self`

EXAMPLES:

```
sage: # needs sage.graphs
sage: space = RootSystem(['A',2]).weight_lattice()
```

(continues on next page)

(continued from previous page)

```
sage: x = space.simple_roots()[1]
sage: pi = space.simple_projection(1)
sage: x
2*Lambda[1] - Lambda[2]
sage: pi(x)
-2*Lambda[1] + Lambda[2]
sage: pi(-x)
-2*Lambda[1] + Lambda[2]
sage: pi = space.simple_projection(1,False)
sage: pi(-x)
2*Lambda[1] - Lambda[2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> space = RootSystem(['A',Integer(2)]).weight_lattice()
>>> x = space.simple_roots()[Integer(1)]
>>> pi = space.simple_projection(Integer(1))
>>> x
2*Lambda[1] - Lambda[2]
>>> pi(x)
-2*Lambda[1] + Lambda[2]
>>> pi(-x)
-2*Lambda[1] + Lambda[2]
>>> pi = space.simple_projection(Integer(1),False)
>>> pi(-x)
2*Lambda[1] - Lambda[2]
```

simple_projections(*to_negative=True*)

Return the family $(s_i)_{i\in I}$ of the simple projections of this root system.

EXAMPLES:

```
sage: space = RootSystem(['A',2]).weight_lattice()
sage: pi = space.simple_projections()                                    #␣
↪needs sage.graphs
sage: x = space.simple_roots()                                           #␣
↪needs sage.graphs
sage: pi[1](x[2])                                                        #␣
↪needs sage.graphs
-Lambda[1] + 2*Lambda[2]
```

```
>>> from sage.all import *
>>> space = RootSystem(['A',Integer(2)]).weight_lattice()
>>> pi = space.simple_projections()                                    #␣
↪needs sage.graphs
>>> x = space.simple_roots()                                           #␣
↪needs sage.graphs
>>> pi[Integer(1)](x[Integer(2)])                                       ␣
↪              # needs sage.graphs
-Lambda[1] + 2*Lambda[2]
```

simple_reflection(*i*)

Return the i-th simple reflection, as a function from `self` to `self`.

INPUT:

- `i` – an element of the index set of `self`

EXAMPLES:

```
sage: space = RootSystem(['A',2]).ambient_lattice()
sage: s = space.simple_reflection(1)
sage: x = space.simple_roots()[1]; x
(1, -1, 0)
sage: s(x)
(-1, 1, 0)
```

```
>>> from sage.all import *
>>> space = RootSystem(['A',Integer(2)]).ambient_lattice()
>>> s = space.simple_reflection(Integer(1))
>>> x = space.simple_roots()[Integer(1)]; x
(1, -1, 0)
>>> s(x)
(-1, 1, 0)
```

simple_reflections()

Return the family $(s_i)_{i\in I}$ of the simple reflections of this root system.

EXAMPLES:

```
sage: r = RootSystem(["A", 2]).root_lattice()
sage: s = r.simple_reflections()
sage: s[1]( r.simple_root(1) )                                              #␣
↪needs sage.graphs
-alpha[1]
```

```
>>> from sage.all import *
>>> r = RootSystem(["A", Integer(2)]).root_lattice()
>>> s = r.simple_reflections()
>>> s[Integer(1)]( r.simple_root(Integer(1)) )                              ␣
↪              # needs sage.graphs
-alpha[1]
```

simple_root(i)

Return the i-th simple root.

This should be overridden by any subclass, and typically implemented as a cached method for efficiency.

EXAMPLES:

```
sage: r = RootSystem(["A",3]).root_lattice()
sage: r.simple_root(1)
alpha[1]
```

```
>>> from sage.all import *
>>> r = RootSystem(["A",Integer(3)]).root_lattice()
>>> r.simple_root(Integer(1))
alpha[1]
```

simple_roots()

Return the family $(\alpha_i)_{i\in I}$ of the simple roots.

EXAMPLES:

```
sage: alpha = RootSystem(["A",3]).root_lattice().simple_roots()
sage: [alpha[i] for i in [1,2,3]]
[alpha[1], alpha[2], alpha[3]]
```

```
>>> from sage.all import *
>>> alpha = RootSystem(["A",Integer(3)]).root_lattice().simple_roots()
>>> [alpha[i] for i in [Integer(1),Integer(2),Integer(3)]]
[alpha[1], alpha[2], alpha[3]]
```

simple_roots_tilde()

Return the family $(\tilde{\alpha}_i)_{i\in I}$ of the simple roots.

INPUT:

- self – an affine root lattice realization

The $\tilde{\alpha}_i$ give the embedding of the root lattice of the other affinization of the same classical root lattice into this root lattice (space?).

This uses the fact that $\alpha_i = \tilde{\alpha}_i$ for i not a special node, and that

$$\delta = \sum a_i\alpha_i = \sum b_i\tilde{\alpha}_i$$

EXAMPLES:

In simply laced cases, this is boring:

```
sage: RootSystem(["A",3, 1]).root_lattice().simple_roots_tilde()          #␣
↪needs sage.graphs
Finite family {0: alpha[0], 1: alpha[1], 2: alpha[2], 3: alpha[3]}
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3), Integer(1)]).root_lattice().simple_roots_
↪tilde()         # needs sage.graphs
Finite family {0: alpha[0], 1: alpha[1], 2: alpha[2], 3: alpha[3]}
```

This was checked by hand:

```
sage: RootSystem(["C",2,1]).coroot_lattice().simple_roots_tilde()         #␣
↪needs sage.graphs
Finite family {0: alphacheck[0] - alphacheck[2],
               1: alphacheck[1],
               2: alphacheck[2]}
sage: RootSystem(["B",2,1]).coroot_lattice().simple_roots_tilde()         #␣
↪needs sage.graphs
Finite family {0: alphacheck[0] - alphacheck[1],
               1: alphacheck[1],
               2: alphacheck[2]}
```

```
>>> from sage.all import *
>>> RootSystem(["C",Integer(2),Integer(1)]).coroot_lattice().simple_roots_
↪tilde()        # needs sage.graphs
Finite family {0: alphacheck[0] - alphacheck[2],
               1: alphacheck[1],
```

(continues on next page)

(continued from previous page)

```
                 2: alphacheck[2]}
>>> RootSystem(["B",Integer(2),Integer(1)]).coroot_lattice().simple_roots_
↪tilde()        # needs sage.graphs
Finite family {0: alphacheck[0] - alphacheck[1],
               1: alphacheck[1],
               2: alphacheck[2]}
```

What about type BC?

some_elements()

Return some elements of this root lattice realization.

EXAMPLES:

```
sage: L = RootSystem(["A",2]).weight_lattice()
sage: L.some_elements()                                                  #␣
↪needs sage.graphs
[2*Lambda[1] + 2*Lambda[2], 2*Lambda[1] - Lambda[2],
 -Lambda[1] + 2*Lambda[2], Lambda[1], Lambda[2]]
sage: L = RootSystem(["A",2]).root_lattice()
sage: L.some_elements()
[2*alpha[1] + 2*alpha[2], alpha[1], alpha[2]]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2)]).weight_lattice()
>>> L.some_elements()                                                  #␣
↪needs sage.graphs
[2*Lambda[1] + 2*Lambda[2], 2*Lambda[1] - Lambda[2],
 -Lambda[1] + 2*Lambda[2], Lambda[1], Lambda[2]]
>>> L = RootSystem(["A",Integer(2)]).root_lattice()
>>> L.some_elements()
[2*alpha[1] + 2*alpha[2], alpha[1], alpha[2]]
```

tau_epsilon_operator_on_almost_positive_roots(J)

The τ_ϵ operator on almost positive roots.

Given a subset J of non adjacent vertices of the Dynkin diagram, this constructs the operator on the almost positive roots which fixes the negative simple roots α_i for i not in J, and acts otherwise by:

$$\tau_+(\beta) = (\prod_{i \in J} s_i)(\beta)$$

See Equation (1.2) of [CFZ2002].

EXAMPLES:

```
sage: L = RootSystem(['A',4]).root_lattice()
sage: tau = L.tau_epsilon_operator_on_almost_positive_roots([1,3])      #␣
↪needs sage.libs.gap
sage: alpha = L.simple_roots()                                           #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(4)]).root_lattice()
```

(continues on next page)

(continued from previous page)

```
>>> tau = L.tau_epsilon_operator_on_almost_positive_roots([Integer(1),
↪Integer(3)])       # needs sage.libs.gap
>>> alpha = L.simple_roots()                                           #␣
↪needs sage.graphs
```

The action on a negative simple root not in J:

```
sage: tau(-alpha[2])                                                   #␣
↪needs sage.graphs sage.libs.gap
-alpha[2]
```

```
>>> from sage.all import *
>>> tau(-alpha[Integer(2)])                                            ␣
↪       # needs sage.graphs sage.libs.gap
-alpha[2]
```

The action on a negative simple root in J:

```
sage: tau(-alpha[1])                                                   #␣
↪needs sage.graphs sage.libs.gap
alpha[1]
```

```
>>> from sage.all import *
>>> tau(-alpha[Integer(1)])                                            ␣
↪       # needs sage.graphs sage.libs.gap
alpha[1]
```

The action on all almost positive roots:

```
sage: for root in L.almost_positive_roots():                           #␣
↪needs sage.graphs sage.libs.gap
....:     print('tau({:<41}) = {}'.format(str(root), tau(root)))
tau(-alpha[1]                                ) = alpha[1]
tau(alpha[1]                                 ) = -alpha[1]
tau(alpha[1] + alpha[2]                      ) = alpha[2] + alpha[3]
tau(alpha[1] + alpha[2] + alpha[3]           ) = alpha[2]
tau(alpha[1] + alpha[2] + alpha[3] + alpha[4]) = alpha[2] + alpha[3] +␣
↪alpha[4]
tau(-alpha[2]                                ) = -alpha[2]
tau(alpha[2]                                 ) = alpha[1] + alpha[2] +␣
↪alpha[3]
tau(alpha[2] + alpha[3]                      ) = alpha[1] + alpha[2]
tau(alpha[2] + alpha[3] + alpha[4]           ) = alpha[1] + alpha[2] +␣
↪alpha[3] + alpha[4]
tau(-alpha[3]                                ) = alpha[3]
tau(alpha[3]                                 ) = -alpha[3]
tau(alpha[3] + alpha[4]                      ) = alpha[4]
tau(-alpha[4]                                ) = -alpha[4]
tau(alpha[4]                                 ) = alpha[3] + alpha[4]
```

```
>>> from sage.all import *
>>> for root in L.almost_positive_roots():                             #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs sage.libs.gap
...     print('tau({:<41}) = {}'.format(str(root), tau(root)))
tau(-alpha[1]                                 ) = alpha[1]
tau(alpha[1]                                  ) = -alpha[1]
tau(alpha[1] + alpha[2]                       ) = alpha[2] + alpha[3]
tau(alpha[1] + alpha[2] + alpha[3]            ) = alpha[2]
tau(alpha[1] + alpha[2] + alpha[3] + alpha[4]) = alpha[2] + alpha[3] +␣
↪alpha[4]
tau(-alpha[2]                                 ) = -alpha[2]
tau(alpha[2]                                  ) = alpha[1] + alpha[2] +␣
↪alpha[3]
tau(alpha[2] + alpha[3]                       ) = alpha[1] + alpha[2]
tau(alpha[2] + alpha[3] + alpha[4]            ) = alpha[1] + alpha[2] +␣
↪alpha[3] + alpha[4]
tau(-alpha[3]                                 ) = alpha[3]
tau(alpha[3]                                  ) = -alpha[3]
tau(alpha[3] + alpha[4]                       ) = alpha[4]
tau(-alpha[4]                                 ) = -alpha[4]
tau(alpha[4]                                  ) = alpha[3] + alpha[4]
```

This method works on any root lattice realization:

```
sage: L = RootSystem(['B',3]).ambient_space()
sage: tau = L.tau_epsilon_operator_on_almost_positive_roots([1,3])      #␣
↪needs sage.libs.gap
sage: for root in L.almost_positive_roots():                            #␣
↪needs sage.graphs sage.libs.gap
....:     print('tau({:<41}) = {}'.format(str(root), tau(root)))
tau((-1, 1, 0)                                ) = (1, -1, 0)
tau((1, 0, 0)                                 ) = (0, 1, 0)
tau((1, -1, 0)                                ) = (-1, 1, 0)
tau((1, 1, 0)                                 ) = (1, 1, 0)
tau((1, 0, -1)                                ) = (0, 1, 1)
tau((1, 0, 1)                                 ) = (0, 1, -1)
tau((0, -1, 1)                                ) = (0, -1, 1)
tau((0, 1, 0)                                 ) = (1, 0, 0)
tau((0, 1, -1)                                ) = (1, 0, 1)
tau((0, 1, 1)                                 ) = (1, 0, -1)
tau((0, 0, -1)                                ) = (0, 0, 1)
tau((0, 0, 1)                                 ) = (0, 0, -1)
```

```
>>> from sage.all import *
>>> L = RootSystem(['B',Integer(3)]).ambient_space()
>>> tau = L.tau_epsilon_operator_on_almost_positive_roots([Integer(1),
↪Integer(3)])      # needs sage.libs.gap
>>> for root in L.almost_positive_roots():                            #␣
↪needs sage.graphs sage.libs.gap
...     print('tau({:<41}) = {}'.format(str(root), tau(root)))
tau((-1, 1, 0)                                ) = (1, -1, 0)
tau((1, 0, 0)                                 ) = (0, 1, 0)
tau((1, -1, 0)                                ) = (-1, 1, 0)
tau((1, 1, 0)                                 ) = (1, 1, 0)
```

(continues on next page)

(continued from previous page)

```
tau((1, 0, -1)                                    ) = (0, 1, 1)
tau((1, 0, 1)                                     ) = (0, 1, -1)
tau((0, -1, 1)                                    ) = (0, -1, 1)
tau((0, 1, 0)                                     ) = (1, 0, 0)
tau((0, 1, -1)                                    ) = (1, 0, 1)
tau((0, 1, 1)                                     ) = (1, 0, -1)
tau((0, 0, -1)                                    ) = (0, 0, 1)
tau((0, 0, 1)                                     ) = (0, 0, -1)
```

See also

tau_plus_minus()

tau_plus_minus()

Return the τ^+ and τ^- piecewise linear operators on `self`.

Those operators are induced by the bipartition $\{L, R\}$ of the simple roots of `self`, and stabilize the almost positive roots. Namely, τ_+ fixes the negative simple roots α_i for i in R, and acts otherwise by:

$$\tau_+(\beta) = (\prod_{i \in L} s_i)(\beta)$$

τ_- acts analogously, with L and R interchanged.

Those operators are used to construct the associahedron, a polytopal realization of the cluster complex (see `Associahedron`).

See also

tau_epsilon_operator_on_almost_positive_roots()

EXAMPLES:

We explore the example of [CFZ2002] Eq.(1.3):

```
sage: S = RootSystem(['A',2]).root_lattice()
sage: taup, taum = S.tau_plus_minus()                                     #␣
↪needs sage.graphs sage.libs.gap
sage: for beta in S.almost_positive_roots():                              #␣
↪needs sage.graphs sage.libs.gap
....:     print("{} , {} , {}".format(beta, taup(beta), taum(beta)))
-alpha[1] , alpha[1] , -alpha[1]
alpha[1] , -alpha[1] , alpha[1] + alpha[2]
alpha[1] + alpha[2] , alpha[2] , alpha[1]
-alpha[2] , -alpha[2] , alpha[2]
alpha[2] , alpha[1] + alpha[2] , -alpha[2]
```

```
>>> from sage.all import *
>>> S = RootSystem(['A',Integer(2)]).root_lattice()
>>> taup, taum = S.tau_plus_minus()                                     #␣
↪needs sage.graphs sage.libs.gap
>>> for beta in S.almost_positive_roots():                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs sage.libs.gap
...     print("{} , {} , {}".format(beta, taup(beta), taum(beta)))
-alpha[1] , alpha[1] , -alpha[1]
alpha[1] , -alpha[1] , alpha[1] + alpha[2]
alpha[1] + alpha[2] , alpha[2] , alpha[1]
-alpha[2] , -alpha[2] , alpha[2]
alpha[2] , alpha[1] + alpha[2] , -alpha[2]
```

to_ambient_space_morphism()

Return the morphism to the ambient space.

EXAMPLES:

```
sage: B2rs = CartanType(['B',2]).root_system()
sage: B2rs.root_lattice().to_ambient_space_morphism()
Generic morphism:
  From: Root lattice of the Root system of type ['B', 2]
  To:   Ambient space of the Root system of type ['B', 2]
sage: B2rs.coroot_lattice().to_ambient_space_morphism()
Generic morphism:
  From: Coroot lattice of the Root system of type ['B', 2]
  To:   Ambient space of the Root system of type ['B', 2]
sage: B2rs.weight_lattice().to_ambient_space_morphism()
Generic morphism:
  From: Weight lattice of the Root system of type ['B', 2]
  To:   Ambient space of the Root system of type ['B', 2]
```

```
>>> from sage.all import *
>>> B2rs = CartanType(['B',Integer(2)]).root_system()
>>> B2rs.root_lattice().to_ambient_space_morphism()
Generic morphism:
  From: Root lattice of the Root system of type ['B', 2]
  To:   Ambient space of the Root system of type ['B', 2]
>>> B2rs.coroot_lattice().to_ambient_space_morphism()
Generic morphism:
  From: Coroot lattice of the Root system of type ['B', 2]
  To:   Ambient space of the Root system of type ['B', 2]
>>> B2rs.weight_lattice().to_ambient_space_morphism()
Generic morphism:
  From: Weight lattice of the Root system of type ['B', 2]
  To:   Ambient space of the Root system of type ['B', 2]
```

weyl_group(*prefix=None*)

Return the Weyl group associated to `self`.

EXAMPLES:

```
sage: RootSystem(['F',4]).ambient_space().weyl_group()                  #␣
↪needs sage.libs.gap
Weyl Group of type ['F', 4] (as a matrix group acting on the ambient␣
↪space)
sage: RootSystem(['F',4]).root_space().weyl_group()                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.gap
Weyl Group of type ['F', 4] (as a matrix group acting on the root space)
```

```
>>> from sage.all import *
>>> RootSystem(['F',Integer(4)]).ambient_space().weyl_group()                  ␣
↪     # needs sage.libs.gap
Weyl Group of type ['F', 4] (as a matrix group acting on the ambient␣
↪space)
>>> RootSystem(['F',Integer(4)]).root_space().weyl_group()                     ␣
↪     # needs sage.libs.gap
Weyl Group of type ['F', 4] (as a matrix group acting on the root space)
```

super_categories()

EXAMPLES:

```
sage: from sage.combinat.root_system.root_lattice_realizations import␣
↪RootLatticeRealizations
sage: RootLatticeRealizations(QQ).super_categories()
[Category of vector spaces with basis over Rational Field]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.root_lattice_realizations import␣
↪RootLatticeRealizations
>>> RootLatticeRealizations(QQ).super_categories()
[Category of vector spaces with basis over Rational Field]
```

5.1.240 Root lattices and root spaces

class sage.combinat.root_system.root_space.**RootSpace**(*root_system*, *base_ring*)

Bases: `CombinatorialFreeModule`

The root space of a root system over a given base ring.

INPUT:

- `root_system` – a root system
- `base_ring` – a ring R

The *root space* (or lattice if `base_ring` is $\mathbf{Z}$) of a root system is the formal free module $\bigoplus_i R\alpha_i$ generated by the simple roots $(\alpha_i)_{i\in I}$ of the root system.

This class is also used for coroot spaces (or lattices).

See also

- `RootSystem()`
- `RootSystem.root_lattice()` and `RootSystem.root_space()`
- `RootLatticeRealizations()`

Todo: standardize the variable used for the root space in the examples (P?)

Element

alias of *RootSpaceElement*

simple_root()

Return the basis element indexed by `i`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.monomial('a')
B['a']
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> F.monomial('a')
B['a']
```

`F.monomial` is in fact (almost) a map:

```
sage: F.monomial
Term map from {'a', 'b', 'c'} to Free module generated by {'a', 'b', 'c'}␣
↪over Rational Field
```

```
>>> from sage.all import *
>>> F.monomial
Term map from {'a', 'b', 'c'} to Free module generated by {'a', 'b', 'c'}␣
↪over Rational Field
```

to_ambient_space_morphism()

The morphism from `self` to its associated ambient space.

EXAMPLES:

```
sage: CartanType(['A',2]).root_system().root_lattice().to_ambient_space_
↪morphism()
Generic morphism:
From: Root lattice of the Root system of type ['A', 2]
To:   Ambient space of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(2)]).root_system().root_lattice().to_ambient_
↪space_morphism()
Generic morphism:
From: Root lattice of the Root system of type ['A', 2]
To:   Ambient space of the Root system of type ['A', 2]
```

to_coroot_space_morphism()

Return the `nu` map to the coroot space over the same base ring, using the symmetrizer of the Cartan matrix.

It does not map the root lattice to the coroot lattice, but has the property that any root is mapped to some scalar multiple of its associated coroot.

EXAMPLES:

```
sage: R = RootSystem(['A',3]).root_space()
sage: alpha = R.simple_roots()
sage: f = R.to_coroot_space_morphism()                                          #␣
↪needs sage.graphs
sage: f(alpha[1])                                                               #␣
↪needs sage.graphs
alphacheck[1]
sage: f(alpha[1] + alpha[2])                                                    #␣
↪needs sage.graphs
alphacheck[1] + alphacheck[2]

sage: R = RootSystem(['A',3]).root_lattice()
sage: alpha = R.simple_roots()
sage: f = R.to_coroot_space_morphism()                                          #␣
↪needs sage.graphs
sage: f(alpha[1])                                                               #␣
↪needs sage.graphs
alphacheck[1]
sage: f(alpha[1] + alpha[2])                                                    #␣
↪needs sage.graphs
alphacheck[1] + alphacheck[2]

sage: S = RootSystem(['G',2]).root_space()
sage: alpha = S.simple_roots()
sage: f = S.to_coroot_space_morphism()                                          #␣
↪needs sage.graphs
sage: f(alpha[1])                                                               #␣
↪needs sage.graphs
alphacheck[1]
sage: f(alpha[1] + alpha[2])                                                    #␣
↪needs sage.graphs
alphacheck[1] + 3*alphacheck[2]
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(3)]).root_space()
>>> alpha = R.simple_roots()
>>> f = R.to_coroot_space_morphism()                                          #␣
↪needs sage.graphs
>>> f(alpha[Integer(1)])                                                        ␣
↪     # needs sage.graphs
alphacheck[1]
>>> f(alpha[Integer(1)] + alpha[Integer(2)])                                    ␣
↪                # needs sage.graphs
alphacheck[1] + alphacheck[2]

>>> R = RootSystem(['A',Integer(3)]).root_lattice()
>>> alpha = R.simple_roots()
>>> f = R.to_coroot_space_morphism()                                          #␣
↪needs sage.graphs
>>> f(alpha[Integer(1)])                                                        ␣
↪     # needs sage.graphs
alphacheck[1]
```

(continues on next page)

(continued from previous page)

```
>>> f(alpha[Integer(1)] + alpha[Integer(2)])
↪                # needs sage.graphs
alphacheck[1] + alphacheck[2]

>>> S = RootSystem(['G',Integer(2)]).root_space()
>>> alpha = S.simple_roots()
>>> f = S.to_coroot_space_morphism()                                    #␣
↪needs sage.graphs
>>> f(alpha[Integer(1)])
↪      # needs sage.graphs
alphacheck[1]
>>> f(alpha[Integer(1)] + alpha[Integer(2)])
↪                # needs sage.graphs
alphacheck[1] + 3*alphacheck[2]
```

class sage.combinat.root_system.root_space.**RootSpaceElement**

Bases: IndexedFreeModuleElement

associated_coroot()

Return the coroot associated to this root.

OUTPUT:

An element of the coroot space over the same base ring; in particular the result is in the coroot lattice whenever `self` is in the root lattice.

EXAMPLES:

```
sage: L = RootSystem(["B", 3]).root_space()
sage: alpha = L.simple_roots()
sage: alpha[1].associated_coroot()                                      #␣
↪needs sage.graphs
alphacheck[1]
sage: alpha[1].associated_coroot().parent()                             #␣
↪needs sage.graphs
Coroot space over the Rational Field of the Root system of type ['B', 3]

sage: L.highest_root()                                                  #␣
↪needs sage.graphs
alpha[1] + 2*alpha[2] + 2*alpha[3]
sage: L.highest_root().associated_coroot()                              #␣
↪needs sage.graphs
alphacheck[1] + 2*alphacheck[2] + alphacheck[3]

sage: alpha = RootSystem(["B", 3]).root_lattice().simple_roots()
sage: alpha[1].associated_coroot()                                      #␣
↪needs sage.graphs
alphacheck[1]
sage: alpha[1].associated_coroot().parent()                             #␣
↪needs sage.graphs
Coroot lattice of the Root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> L = RootSystem(["B", Integer(3)]).root_space()
```

(continues on next page)

(continued from previous page)

```
>>> alpha = L.simple_roots()
>>> alpha[Integer(1)].associated_coroot()                                      ␣
↪     # needs sage.graphs
alphacheck[1]
>>> alpha[Integer(1)].associated_coroot().parent()                             ␣
↪     # needs sage.graphs
Coroot space over the Rational Field of the Root system of type ['B', 3]

>>> L.highest_root()                                                          #␣
↪needs sage.graphs
alpha[1] + 2*alpha[2] + 2*alpha[3]
>>> L.highest_root().associated_coroot()                                      #␣
↪needs sage.graphs
alphacheck[1] + 2*alphacheck[2] + alphacheck[3]

>>> alpha = RootSystem(["B", Integer(3)]).root_lattice().simple_roots()
>>> alpha[Integer(1)].associated_coroot()                                      ␣
↪     # needs sage.graphs
alphacheck[1]
>>> alpha[Integer(1)].associated_coroot().parent()                             ␣
↪     # needs sage.graphs
Coroot lattice of the Root system of type ['B', 3]
```

is_positive_root()

Check whether an element in the root space lies in the nonnegative cone spanned by the simple roots.

EXAMPLES:

```
sage: R = RootSystem(['A',3,1]).root_space()
sage: B = R.basis()
sage: w = B[0] + B[3]
sage: w.is_positive_root()
True
sage: w = B[1] - B[2]
sage: w.is_positive_root()
False
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(3),Integer(1)]).root_space()
>>> B = R.basis()
>>> w = B[Integer(0)] + B[Integer(3)]
>>> w.is_positive_root()
True
>>> w = B[Integer(1)] - B[Integer(2)]
>>> w.is_positive_root()
False
```

max_coroot_le()

Return the highest positive coroot whose associated root is less than or equal to `self`.

INPUT:

- `self` – an element of the nonnegative integer span of simple roots

Returns None for the zero element.

Really `self` is an element of a coroot lattice and this method returns the highest root whose associated coroot is <= `self`.

> **Warning**
>
> This implementation only handles finite Cartan types

EXAMPLES:

```
sage: # needs sage.graphs
sage: root_lattice = RootSystem(['C',2]).root_lattice()
sage: root_lattice.from_vector(vector([1,1])).max_coroot_le()
alphacheck[1] + 2*alphacheck[2]
sage: root_lattice.from_vector(vector([2,1])).max_coroot_le()
alphacheck[1] + 2*alphacheck[2]
sage: root_lattice = RootSystem(['B',2]).root_lattice()
sage: root_lattice.from_vector(vector([1,1])).max_coroot_le()
2*alphacheck[1] + alphacheck[2]
sage: root_lattice.from_vector(vector([1,2])).max_coroot_le()
2*alphacheck[1] + alphacheck[2]

sage: root_lattice.zero().max_coroot_le() is None                          #␣
↪needs sage.graphs
True
sage: root_lattice.from_vector(vector([-1,0])).max_coroot_le()             #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: -alpha[1] is not in the positive cone of roots
sage: root_lattice = RootSystem(['A',2,1]).root_lattice()
sage: root_lattice.simple_root(1).max_coroot_le()                          #␣
↪needs sage.graphs
Traceback (most recent call last):
...
NotImplementedError: Only implemented for finite Cartan type
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> root_lattice = RootSystem(['C',Integer(2)]).root_lattice()
>>> root_lattice.from_vector(vector([Integer(1),Integer(1)])).max_coroot_le()
alphacheck[1] + 2*alphacheck[2]
>>> root_lattice.from_vector(vector([Integer(2),Integer(1)])).max_coroot_le()
alphacheck[1] + 2*alphacheck[2]
>>> root_lattice = RootSystem(['B',Integer(2)]).root_lattice()
>>> root_lattice.from_vector(vector([Integer(1),Integer(1)])).max_coroot_le()
2*alphacheck[1] + alphacheck[2]
>>> root_lattice.from_vector(vector([Integer(1),Integer(2)])).max_coroot_le()
2*alphacheck[1] + alphacheck[2]

>>> root_lattice.zero().max_coroot_le() is None                            #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
True
>>> root_lattice.from_vector(vector([-Integer(1),Integer(0)])).max_coroot_
↪le()                  # needs sage.graphs
Traceback (most recent call last):
...
ValueError: -alpha[1] is not in the positive cone of roots
>>> root_lattice = RootSystem(['A',Integer(2),Integer(1)]).root_lattice()
>>> root_lattice.simple_root(Integer(1)).max_coroot_le()                      ␣
↪      # needs sage.graphs
Traceback (most recent call last):
...
NotImplementedError: Only implemented for finite Cartan type
```

max_quantum_element()

Return a reduced word for the longest element of the Weyl group whose shortest path in the quantum Bruhat graph to the identity has sum of quantum coroots at most `self`.

INPUT:

- `self` – an element of the nonnegative integer span of simple roots

Really `self` is an element of a coroot lattice.

> **Warning**
>
> This implementation only handles finite Cartan types

EXAMPLES:

```
sage: # needs sage.graphs sage.libs.gap
sage: Qvee = RootSystem(['C',2]).coroot_lattice()
sage: Qvee.from_vector(vector([1,2])).max_quantum_element()
[2, 1, 2, 1]
sage: Qvee.from_vector(vector([1,1])).max_quantum_element()
[1, 2, 1]
sage: Qvee.from_vector(vector([0,2])).max_quantum_element()
[2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap
>>> Qvee = RootSystem(['C',Integer(2)]).coroot_lattice()
>>> Qvee.from_vector(vector([Integer(1),Integer(2)])).max_quantum_element()
[2, 1, 2, 1]
>>> Qvee.from_vector(vector([Integer(1),Integer(1)])).max_quantum_element()
[1, 2, 1]
>>> Qvee.from_vector(vector([Integer(0),Integer(2)])).max_quantum_element()
[2]
```

quantum_root()

Check whether `self` is a quantum root.

INPUT:

- `self` – an element of the nonnegative integer span of simple roots

A root α is a quantum root if $\ell(s_\alpha) = \langle 2\rho, \alpha^\vee \rangle - 1$ where ℓ is the length function, s_α is the reflection across the hyperplane orthogonal to α, and 2ρ is the sum of positive roots.

Warning

This implementation only handles finite Cartan types and assumes that `self` is a root.

Todo

Rename to is_quantum_root

EXAMPLES:

```
sage: Q = RootSystem(['C',2]).root_lattice()
sage: positive_roots = Q.positive_roots()
sage: for x in sorted(positive_roots):                                          #␣
↪needs sage.graphs
....:     print("{} {}".format(x, x.quantum_root()))
alpha[1] True
alpha[1] + alpha[2] False
2*alpha[1] + alpha[2] True
alpha[2] True
```

```
>>> from sage.all import *
>>> Q = RootSystem(['C',Integer(2)]).root_lattice()
>>> positive_roots = Q.positive_roots()
>>> for x in sorted(positive_roots):                                            #␣
↪needs sage.graphs
...     print("{} {}".format(x, x.quantum_root()))
alpha[1] True
alpha[1] + alpha[2] False
2*alpha[1] + alpha[2] True
alpha[2] True
```

scalar(*lambdacheck*)

The scalar product between the root lattice and the coroot lattice.

EXAMPLES:

```
sage: L = RootSystem(['B',4]).root_lattice()
sage: alpha      = L.simple_roots()
sage: alphacheck = L.simple_coroots()
sage: alpha[1].scalar(alphacheck[1])                                            #␣
↪needs sage.graphs
2
sage: alpha[1].scalar(alphacheck[2])                                            #␣
↪needs sage.graphs
-1
```

```
>>> from sage.all import *
>>> L = RootSystem(['B',Integer(4)]).root_lattice()
>>> alpha      = L.simple_roots()
>>> alphacheck = L.simple_coroots()
>>> alpha[Integer(1)].scalar(alphacheck[Integer(1)])                         ␣
↪               # needs sage.graphs
2
>>> alpha[Integer(1)].scalar(alphacheck[Integer(2)])                         ␣
↪               # needs sage.graphs
-1
```

The scalar products between the roots and coroots are given by the Cartan matrix:

```
sage: matrix([ [ alpha[i].scalar(alphacheck[j])                              #␣
↪needs sage.graphs
....:              for i in L.index_set() ]
....:            for j in L.index_set() ])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -2  2]

sage: L.cartan_type().cartan_matrix()                                        #␣
↪needs sage.graphs
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -2  2]
```

```
>>> from sage.all import *
>>> matrix([ [ alpha[i].scalar(alphacheck[j])                              #␣
↪needs sage.graphs
...              for i in L.index_set() ]
...            for j in L.index_set() ])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -2  2]

>>> L.cartan_type().cartan_matrix()                                        #␣
↪needs sage.graphs
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -2  2]
```

to_ambient()

Map `self` to the ambient space.

EXAMPLES:

```
sage: alpha = CartanType(['B',2]).root_system().root_lattice().an_element();␣
↪alpha
```

(continues on next page)

(continued from previous page)

```
2*alpha[1] + 2*alpha[2]
sage: alpha.to_ambient()
(2, 0)
sage: alphavee = CartanType(['B',2]).root_system().coroot_lattice().an_
↪element(); alphavee
2*alphacheck[1] + 2*alphacheck[2]
sage: alphavee.to_ambient()
(2, 2)
```

```
>>> from sage.all import *
>>> alpha = CartanType(['B',Integer(2)]).root_system().root_lattice().an_
↪element(); alpha
2*alpha[1] + 2*alpha[2]
>>> alpha.to_ambient()
(2, 0)
>>> alphavee = CartanType(['B',Integer(2)]).root_system().coroot_lattice().an_
↪element(); alphavee
2*alphacheck[1] + 2*alphacheck[2]
>>> alphavee.to_ambient()
(2, 2)
```

5.1.241 Root systems

See *Root Systems* for an overview.

class sage.combinat.root_system.root_system.**RootSystem**(*cartan_type*, *as_dual_of=None*)

Bases: UniqueRepresentation, SageObject

A class for root systems.

EXAMPLES:

We construct the root system for type B_3:

```
sage: R = RootSystem(['B',3]); R
Root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> R = RootSystem(['B',Integer(3)]); R
Root system of type ['B', 3]
```

R models the root system abstractly. It comes equipped with various realizations of the root and weight lattices, where all computations take place. Let us play first with the root lattice:

```
sage: space = R.root_lattice(); space
Root lattice of the Root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> space = R.root_lattice(); space
Root lattice of the Root system of type ['B', 3]
```

This is the free $\mathbf{Z}$-module $\bigoplus_i \mathbf{Z}.\alpha_i$ spanned by the simple roots:

```
sage: space.base_ring()
Integer Ring
sage: list(space.basis())
[alpha[1], alpha[2], alpha[3]]
```

```
>>> from sage.all import *
>>> space.base_ring()
Integer Ring
>>> list(space.basis())
[alpha[1], alpha[2], alpha[3]]
```

Let us do some computations with the simple roots:

```
sage: alpha = space.simple_roots()
sage: alpha[1] + alpha[2]
alpha[1] + alpha[2]
```

```
>>> from sage.all import *
>>> alpha = space.simple_roots()
>>> alpha[Integer(1)] + alpha[Integer(2)]
alpha[1] + alpha[2]
```

There is a canonical pairing between the root lattice and the coroot lattice:

```
sage: R.coroot_lattice()
Coroot lattice of the Root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> R.coroot_lattice()
Coroot lattice of the Root system of type ['B', 3]
```

We construct the simple coroots, and do some computations (see comments about duality below for some caveat):

```
sage: alphacheck = space.simple_coroots()
sage: list(alphacheck)
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

```
>>> from sage.all import *
>>> alphacheck = space.simple_coroots()
>>> list(alphacheck)
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

We can carry over the same computations in any of the other realizations of the root lattice, like the root space $\bigoplus_i \mathbf{Q}.\alpha_i$, the weight lattice $\bigoplus_i \mathbf{Z}.\Lambda_i$, the weight space $\bigoplus_i \mathbf{Q}.\Lambda_i$. For example:

```
sage: space = R.weight_space(); space
Weight space over the Rational Field of the Root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> space = R.weight_space(); space
Weight space over the Rational Field of the Root system of type ['B', 3]
```

```
sage: space.base_ring()
Rational Field
sage: list(space.basis())
[Lambda[1], Lambda[2], Lambda[3]]
```

```
>>> from sage.all import *
>>> space.base_ring()
Rational Field
>>> list(space.basis())
[Lambda[1], Lambda[2], Lambda[3]]
```

```
sage: alpha = space.simple_roots()                                          #␣
↪needs sage.graphs
sage: alpha[1] + alpha[2]                                                   #␣
↪needs sage.graphs
Lambda[1] + Lambda[2] - 2*Lambda[3]
```

```
>>> from sage.all import *
>>> alpha = space.simple_roots()                                          #␣
↪needs sage.graphs
>>> alpha[Integer(1)] + alpha[Integer(2)]                                   ␣
↪                # needs sage.graphs
Lambda[1] + Lambda[2] - 2*Lambda[3]
```

The fundamental weights are the dual basis of the coroots:

```
sage: Lambda = space.fundamental_weights()
sage: Lambda[1]
Lambda[1]
```

```
>>> from sage.all import *
>>> Lambda = space.fundamental_weights()
>>> Lambda[Integer(1)]
Lambda[1]
```

```
sage: alphacheck = space.simple_coroots()                                   #␣
↪needs sage.graphs
sage: list(alphacheck)                                                      #␣
↪needs sage.graphs
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

```
>>> from sage.all import *
>>> alphacheck = space.simple_coroots()                                   #␣
↪needs sage.graphs
>>> list(alphacheck)                                                      #␣
↪needs sage.graphs
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

```
sage: [Lambda[i].scalar(alphacheck[1]) for i in space.index_set()]
[1, 0, 0]
sage: [Lambda[i].scalar(alphacheck[2]) for i in space.index_set()]
```

(continues on next page)

(continued from previous page)

```
[0, 1, 0]
sage: [Lambda[i].scalar(alphacheck[3]) for i in space.index_set()]
[0, 0, 1]
```

```
>>> from sage.all import *
>>> [Lambda[i].scalar(alphacheck[Integer(1)]) for i in space.index_set()]
[1, 0, 0]
>>> [Lambda[i].scalar(alphacheck[Integer(2)]) for i in space.index_set()]
[0, 1, 0]
>>> [Lambda[i].scalar(alphacheck[Integer(3)]) for i in space.index_set()]
[0, 0, 1]
```

Let us use the simple reflections. In the weight space, they work as in the *number game*: firing the node i on an element x adds c times the simple root α_i, where c is the coefficient of i in x:

```
sage: # needs sage.graphs
sage: Lambda[1].simple_reflection(1)
-Lambda[1] + Lambda[2]
sage: Lambda[2].simple_reflection(1)
Lambda[2]
sage: Lambda[3].simple_reflection(1)
Lambda[3]
sage: (-2*Lambda[1] + Lambda[2] + Lambda[3]).simple_reflection(1)
2*Lambda[1] - Lambda[2] + Lambda[3]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> Lambda[Integer(1)].simple_reflection(Integer(1))
-Lambda[1] + Lambda[2]
>>> Lambda[Integer(2)].simple_reflection(Integer(1))
Lambda[2]
>>> Lambda[Integer(3)].simple_reflection(Integer(1))
Lambda[3]
>>> (-Integer(2)*Lambda[Integer(1)] + Lambda[Integer(2)] + Lambda[Integer(3)]).
↪simple_reflection(Integer(1))
2*Lambda[1] - Lambda[2] + Lambda[3]
```

It can be convenient to manipulate the simple reflections themselves:

```
sage: # needs sage.graphs
sage: s = space.simple_reflections()
sage: s[1](Lambda[1])
-Lambda[1] + Lambda[2]
sage: s[1](Lambda[2])
Lambda[2]
sage: s[1](Lambda[3])
Lambda[3]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> s = space.simple_reflections()
>>> s[Integer(1)](Lambda[Integer(1)])
```

(continues on next page)

(continued from previous page)

```
-Lambda[1] + Lambda[2]
>>> s[Integer(1)](Lambda[Integer(2)])
Lambda[2]
>>> s[Integer(1)](Lambda[Integer(3)])
Lambda[3]
```

Ambient spaces

The root system may also come equipped with an ambient space. This is a **Q**-module, endowed with its canonical Euclidean scalar product, which admits simultaneous embeddings of the (extended) weight and the (extended) coweight lattice, and therefore the root and the coroot lattice. This is implemented on a type by type basis for the finite crystallographic root systems following Bourbaki's conventions and is extended to the affine cases. Coefficients permitting, this is also available as an ambient lattice.

See also:

ambient_space() and *ambient_lattice()* for details

In finite type A, we recover the natural representation of the symmetric group as group of permutation matrices:

```
sage: RootSystem(["A",2]).ambient_space().weyl_group().simple_reflections()      #␣
↪needs sage.libs.gap sage.libs.pari
Finite family {1: [0 1 0]
                  [1 0 0]
                  [0 0 1],
               2: [1 0 0]
                  [0 0 1]
                  [0 1 0]}
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(2)]).ambient_space().weyl_group().simple_
↪reflections()      # needs sage.libs.gap sage.libs.pari
Finite family {1: [0 1 0]
                  [1 0 0]
                  [0 0 1],
               2: [1 0 0]
                  [0 0 1]
                  [0 1 0]}
```

In type B, C, and D, we recover the natural representation of the Weyl group as groups of signed permutation matrices:

```
sage: RootSystem(["B",3]).ambient_space().weyl_group().simple_reflections()      #␣
↪needs sage.libs.gap sage.libs.pari
Finite family {1: [0 1 0]
                  [1 0 0]
                  [0 0 1],
               2: [1 0 0]
                  [0 0 1]
                  [0 1 0],
               3: [ 1  0  0]
```

(continues on next page)

(continued from previous page)

```
                  [ 0  1  0]
                  [ 0  0 -1]}
```

```
>>> from sage.all import *
>>> RootSystem(["B",Integer(3)]).ambient_space().weyl_group().simple_
↪reflections()      # needs sage.libs.gap sage.libs.pari
Finite family {1: [0 1 0]
                  [1 0 0]
                  [0 0 1],
               2: [1 0 0]
                  [0 0 1]
                  [0 1 0],
               3: [ 1  0  0]
                  [ 0  1  0]
                  [ 0  0 -1]}
```

In (untwisted) affine types A, …, D, one can recover from the ambient space the affine permutation representation, in window notation. Let us consider the ambient space for affine type A:

```
sage: L = RootSystem(["A",2,1]).ambient_space(); L
Ambient space of the Root system of type ['A', 2, 1]
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(2),Integer(1)]).ambient_space(); L
Ambient space of the Root system of type ['A', 2, 1]
```

Define the "identity" by an appropriate vector at level -3:

```
sage: e = L.basis(); Lambda = L.fundamental_weights()                          #␣
↪needs sage.graphs
sage: id = e[0] + 2*e[1] + 3*e[2]  - 3*Lambda[0]                               #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> e = L.basis(); Lambda = L.fundamental_weights()                          #␣
↪needs sage.graphs
>>> id = e[Integer(0)] + Integer(2)*e[Integer(1)] + Integer(3)*e[Integer(2)]  -␣
↪Integer(3)*Lambda[Integer(0)]                                 # needs sage.graphs
```

The corresponding permutation is obtained by projecting it onto the classical ambient space:

```
sage: L.classical()
Ambient space of the Root system of type ['A', 2]
sage: L.classical()(id)                                                        #␣
↪needs sage.graphs
(1, 2, 3)
```

```
>>> from sage.all import *
>>> L.classical()
Ambient space of the Root system of type ['A', 2]
>>> L.classical()(id)                                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
(1, 2, 3)
```

Here is the orbit of the identity under the action of the finite group:

```
sage: # needs sage.graphs sage.libs.gap sage.libs.pari
sage: W = L.weyl_group()
sage: S3 = [ w.action(id) for w in W.classical() ]
sage: [L.classical()(x) for x in S3]
[(1, 2, 3), (3, 1, 2), (2, 3, 1), (2, 1, 3), (1, 3, 2), (3, 2, 1)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap sage.libs.pari
>>> W = L.weyl_group()
>>> S3 = [ w.action(id) for w in W.classical() ]
>>> [L.classical()(x) for x in S3]
[(1, 2, 3), (3, 1, 2), (2, 3, 1), (2, 1, 3), (1, 3, 2), (3, 2, 1)]
```

And the action of s_0 on these yields:

```
sage: # needs sage.graphs sage.libs.gap sage.libs.pari
sage: s = W.simple_reflections()
sage: [L.classical()(s[0].action(x)) for x in S3]
[(0, 2, 4), (-1, 1, 6), (-2, 3, 5), (0, 1, 5), (-1, 3, 4), (-2, 2, 6)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap sage.libs.pari
>>> s = W.simple_reflections()
>>> [L.classical()(s[Integer(0)].action(x)) for x in S3]
[(0, 2, 4), (-1, 1, 6), (-2, 3, 5), (0, 1, 5), (-1, 3, 4), (-2, 2, 6)]
```

We can also plot various components of the ambient spaces:

```
sage: L = RootSystem(['A',2]).ambient_space()
sage: L.plot()                                                              #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 13 graphics primitives
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2)]).ambient_space()
>>> L.plot()                                                              #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 13 graphics primitives
```

For more on plotting, see *Tutorial: visualizing root systems*.

Dual root systems

The root system is aware of its dual root system:

```
sage: R.dual
Dual of root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> R.dual
Dual of root system of type ['B', 3]
```

R.dual is really the root system of type C_3:

```
sage: R.dual.cartan_type()
['C', 3]
```

```
>>> from sage.all import *
>>> R.dual.cartan_type()
['C', 3]
```

And the coroot lattice that we have been manipulating before is really implemented as the root lattice of the dual root system:

```
sage: R.dual.root_lattice()
Coroot lattice of the Root system of type ['B', 3]
```

```
>>> from sage.all import *
>>> R.dual.root_lattice()
Coroot lattice of the Root system of type ['B', 3]
```

In particular, the coroots for the root lattice are in fact the roots of the coroot lattice:

```
sage: list(R.root_lattice().simple_coroots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
sage: list(R.coroot_lattice().simple_roots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
sage: list(R.dual.root_lattice().simple_roots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

```
>>> from sage.all import *
>>> list(R.root_lattice().simple_coroots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
>>> list(R.coroot_lattice().simple_roots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
>>> list(R.dual.root_lattice().simple_roots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

The coweight lattice and space are defined similarly. Note that, to limit confusion, all the output have been tweaked appropriately.

See also

- sage.combinat.root_system
- *RootSpace*
- *WeightSpace*
- AmbientSpace
- *RootLatticeRealizations*

- *WeightLatticeRealizations*

ambient_lattice()

Return the ambient lattice for this root_system.

This is the ambient space, over **Z**.

See also

- *ambient_space()*
- *root_lattice()*
- *weight_lattice()*

EXAMPLES:

```
sage: RootSystem(['A',4]).ambient_lattice()
Ambient lattice of the Root system of type ['A', 4]
sage: RootSystem(['A',4,1]).ambient_lattice()
Ambient lattice of the Root system of type ['A', 4, 1]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(4)]).ambient_lattice()
Ambient lattice of the Root system of type ['A', 4]
>>> RootSystem(['A',Integer(4),Integer(1)]).ambient_lattice()
Ambient lattice of the Root system of type ['A', 4, 1]
```

Except in type A, only an ambient space can be realized:

```
sage: RootSystem(['B',4]).ambient_lattice()
sage: RootSystem(['C',4]).ambient_lattice()
sage: RootSystem(['D',4]).ambient_lattice()
sage: RootSystem(['E',6]).ambient_lattice()
sage: RootSystem(['F',4]).ambient_lattice()
sage: RootSystem(['G',2]).ambient_lattice()
```

```
>>> from sage.all import *
>>> RootSystem(['B',Integer(4)]).ambient_lattice()
>>> RootSystem(['C',Integer(4)]).ambient_lattice()
>>> RootSystem(['D',Integer(4)]).ambient_lattice()
>>> RootSystem(['E',Integer(6)]).ambient_lattice()
>>> RootSystem(['F',Integer(4)]).ambient_lattice()
>>> RootSystem(['G',Integer(2)]).ambient_lattice()
```

ambient_space(*base_ring=Rational Field*)

Return the usual ambient space for this root_system.

INPUT:

- `base_ring` – a base ring (default: **Q**)

This is a `base_ring`-module, endowed with its canonical Euclidean scalar product, which admits simultaneous embeddings into the weight and the coweight lattice, and therefore the root and the coroot lattice, and preserves scalar products between elements of the coroot lattice and elements of the root or weight lattice (and dually).

There is no mechanical way to define the ambient space just from the Cartan matrix. Instead it is constructed from hard coded type by type data, according to the usual Bourbaki conventions. Such data is provided for all the finite (crystallographic) types. From this data, ambient spaces can be built as well for dual types, reducible types and affine types. When no data is available, or if the base ring is not large enough, None is returned.

Warning

for affine types

See also

- The section on ambient spaces in *RootSystem*
- *ambient_lattice()*
- *AmbientSpace*
- AmbientSpace
- *root_space()*
- weight:space()

EXAMPLES:

```
sage: RootSystem(['A',4]).ambient_space()
Ambient space of the Root system of type ['A', 4]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(4)]).ambient_space()
Ambient space of the Root system of type ['A', 4]
```

```
sage: RootSystem(['B',4]).ambient_space()
Ambient space of the Root system of type ['B', 4]
```

```
>>> from sage.all import *
>>> RootSystem(['B',Integer(4)]).ambient_space()
Ambient space of the Root system of type ['B', 4]
```

```
sage: RootSystem(['C',4]).ambient_space()
Ambient space of the Root system of type ['C', 4]
```

```
>>> from sage.all import *
>>> RootSystem(['C',Integer(4)]).ambient_space()
Ambient space of the Root system of type ['C', 4]
```

```
sage: RootSystem(['D',4]).ambient_space()
Ambient space of the Root system of type ['D', 4]
```

```
>>> from sage.all import *
>>> RootSystem(['D',Integer(4)]).ambient_space()
Ambient space of the Root system of type ['D', 4]
```

```
sage: RootSystem(['E',6]).ambient_space()
Ambient space of the Root system of type ['E', 6]
```

```
>>> from sage.all import *
>>> RootSystem(['E',Integer(6)]).ambient_space()
Ambient space of the Root system of type ['E', 6]
```

```
sage: RootSystem(['F',4]).ambient_space()
Ambient space of the Root system of type ['F', 4]
```

```
>>> from sage.all import *
>>> RootSystem(['F',Integer(4)]).ambient_space()
Ambient space of the Root system of type ['F', 4]
```

```
sage: RootSystem(['G',2]).ambient_space()
Ambient space of the Root system of type ['G', 2]
```

```
>>> from sage.all import *
>>> RootSystem(['G',Integer(2)]).ambient_space()
Ambient space of the Root system of type ['G', 2]
```

An alternative base ring can be provided as an option:

```
sage: e = RootSystem(['B',3]).ambient_space(RR)
sage: TestSuite(e).run()                                                    #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> e = RootSystem(['B',Integer(3)]).ambient_space(RR)
>>> TestSuite(e).run()                                                    #␣
↪needs sage.graphs
```

It should contain the smallest ring over which the ambient space can be defined (**Z** in type A or **Q** otherwise). Otherwise None is returned:

```
sage: RootSystem(['B',2]).ambient_space(ZZ)
```

```
>>> from sage.all import *
>>> RootSystem(['B',Integer(2)]).ambient_space(ZZ)
```

The base ring should also be totally ordered. In practice, only **Z** and **Q** are really supported at this point, but you are welcome to experiment:

```
sage: e = RootSystem(['G',2]).ambient_space(RR)
sage: TestSuite(e).run()                                                    #␣
↪needs sage.graphs
Failure in _test_root_lattice_realization:
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
AssertionError: 2.00000000000000 != 2.00000000000000
------------------------------------------------------------
The following tests failed: _test_root_lattice_realization
```

```
>>> from sage.all import *
>>> e = RootSystem(['G',Integer(2)]).ambient_space(RR)
>>> TestSuite(e).run()                                                      #␣
↪needs sage.graphs
Failure in _test_root_lattice_realization:
Traceback (most recent call last):
...
AssertionError: 2.00000000000000 != 2.00000000000000
------------------------------------------------------------
The following tests failed: _test_root_lattice_realization
```

cartan_matrix()

EXAMPLES:

```
sage: RootSystem(['A',3]).cartan_matrix()                                     #␣
↪needs sage.graphs
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).cartan_matrix()                              ␣
↪      # needs sage.graphs
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
```

cartan_type()

Return the Cartan type of the root system.

EXAMPLES:

```
sage: R = RootSystem(['A',3])
sage: R.cartan_type()
['A', 3]
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(3)])
>>> R.cartan_type()
['A', 3]
```

coambient_space(*base_ring=Rational Field*)

Return the coambient space for this root system.

This is the ambient space of the dual root system.

See also

- *ambient_space()*

EXAMPLES:

```
sage: L = RootSystem(["B",2]).ambient_space(); L
Ambient space of the Root system of type ['B', 2]
sage: coL = RootSystem(["B",2]).coambient_space(); coL
Coambient space of the Root system of type ['B', 2]
```

```
>>> from sage.all import *
>>> L = RootSystem(["B",Integer(2)]).ambient_space(); L
Ambient space of the Root system of type ['B', 2]
>>> coL = RootSystem(["B",Integer(2)]).coambient_space(); coL
Coambient space of the Root system of type ['B', 2]
```

The roots and coroots are interchanged:

```
sage: coL.simple_roots()
Finite family {1: (1, -1), 2: (0, 2)}
sage: L.simple_coroots()
Finite family {1: (1, -1), 2: (0, 2)}

sage: coL.simple_coroots()
Finite family {1: (1, -1), 2: (0, 1)}
sage: L.simple_roots()
Finite family {1: (1, -1), 2: (0, 1)}
```

```
>>> from sage.all import *
>>> coL.simple_roots()
Finite family {1: (1, -1), 2: (0, 2)}
>>> L.simple_coroots()
Finite family {1: (1, -1), 2: (0, 2)}

>>> coL.simple_coroots()
Finite family {1: (1, -1), 2: (0, 1)}
>>> L.simple_roots()
Finite family {1: (1, -1), 2: (0, 1)}
```

coroot_lattice()

Return the coroot lattice associated to `self`.

EXAMPLES:

```
sage: RootSystem(['A',3]).coroot_lattice()
Coroot lattice of the Root system of type ['A', 3]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).coroot_lattice()
Coroot lattice of the Root system of type ['A', 3]
```

coroot_space(*base_ring=Rational Field*)

Return the coroot space associated to `self`.

EXAMPLES:

```
sage: RootSystem(['A',3]).coroot_space()
Coroot space over the Rational Field of the Root system of type ['A', 3]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).coroot_space()
Coroot space over the Rational Field of the Root system of type ['A', 3]
```

coweight_lattice(*extended=False*)

Return the coweight lattice associated to `self`.

This is the weight lattice of the dual root system.

See also

- *coweight_space()*
- *weight_space()*, *weight_lattice()*
- `WeightSpace`

EXAMPLES:

```
sage: RootSystem(['A',3]).coweight_lattice()
Coweight lattice of the Root system of type ['A', 3]

sage: RootSystem(['A',3,1]).coweight_lattice(extended=True)
Extended coweight lattice of the Root system of type ['A', 3, 1]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).coweight_lattice()
Coweight lattice of the Root system of type ['A', 3]

>>> RootSystem(['A',Integer(3),Integer(1)]).coweight_lattice(extended=True)
Extended coweight lattice of the Root system of type ['A', 3, 1]
```

coweight_space(*base_ring=Rational Field*, *extended=False*)

Return the coweight space associated to `self`.

This is the weight space of the dual root system.

See also

- *coweight_lattice()*
- *weight_space()*, *weight_lattice()*
- `WeightSpace`

EXAMPLES:

```
sage: RootSystem(['A',3]).coweight_space()
Coweight space over the Rational Field of the Root system of type ['A', 3]

sage: RootSystem(['A',3,1]).coweight_space(extended=True)
Extended coweight space over the Rational Field
 of the Root system of type ['A', 3, 1]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).coweight_space()
Coweight space over the Rational Field of the Root system of type ['A', 3]

>>> RootSystem(['A',Integer(3),Integer(1)]).coweight_space(extended=True)
Extended coweight space over the Rational Field
 of the Root system of type ['A', 3, 1]
```

coxeter_number()

Return the Coxeter number of an irreducible finite root system.

See also

coxeter_number().

EXAMPLES:

```
sage: rt = RootSystem(['C', 5])
sage: rt.coxeter_number()
10
```

```
>>> from sage.all import *
>>> rt = RootSystem(['C', Integer(5)])
>>> rt.coxeter_number()
10
```

dual_coxeter_number()

Return the dual Coxeter number of a irreducible finite root system.

The dual Coxeter number is equal to 1 plus the sum of the coefficients of simple roots in the highest short root of the dual root system.

See also

dual_coxeter_number()

EXAMPLES:

```
sage: rt = RootSystem(['C', 5])
sage: rt.dual_coxeter_number()
6
```

```
>>> from sage.all import *
>>> rt = RootSystem(['C', Integer(5)])
```

(continues on next page)

(continued from previous page)

```
>>> rt.dual_coxeter_number()
6
```

The dual Coxeter number is not the same concept as the Coxeter number of the dual root system:

```
sage: rt.dual
Dual of root system of type ['C', 5]
sage: rt.dual.coxeter_number()
10
```

```
>>> from sage.all import *
>>> rt.dual
Dual of root system of type ['C', 5]
>>> rt.dual.coxeter_number()
10
```

dynkin_diagram()

Return the Dynkin diagram of the root system.

EXAMPLES:

```
sage: R = RootSystem(['A',3])
sage: R.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O
1   2   3
A3
```

```
>>> from sage.all import *
>>> R = RootSystem(['A',Integer(3)])
>>> R.dynkin_diagram()                                                  #␣
↪needs sage.graphs
O---O---O
1   2   3
A3
```

index_set()

EXAMPLES:

```
sage: RootSystem(['A',3]).index_set()
(1, 2, 3)
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).index_set()
(1, 2, 3)
```

is_finite()

Return `True` if `self` is a finite root system.

EXAMPLES:

```
sage: RootSystem(["A",3]).is_finite()
True
sage: RootSystem(["A",3,1]).is_finite()
False
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3)]).is_finite()
True
>>> RootSystem(["A",Integer(3),Integer(1)]).is_finite()
False
```

is_irreducible()

Return `True` if `self` is an irreducible root system.

EXAMPLES:

```
sage: RootSystem(['A', 3]).is_irreducible()
True
sage: RootSystem("A2xB2").is_irreducible()
False
```

```
>>> from sage.all import *
>>> RootSystem(['A', Integer(3)]).is_irreducible()
True
>>> RootSystem("A2xB2").is_irreducible()
False
```

root_lattice()

Return the root lattice associated to `self`.

EXAMPLES:

```
sage: RootSystem(['A',3]).root_lattice()
Root lattice of the Root system of type ['A', 3]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).root_lattice()
Root lattice of the Root system of type ['A', 3]
```

root_poset(*restricted=False*, *facade=False*)

Return the (restricted) root poset associated to `self`.

The elements are given by the positive roots (resp. non-simple, positive roots), and $\alpha \leq \beta$ iff $\beta - \alpha$ is a nonnegative linear combination of simple roots.

INPUT:

- `restricted` – boolean (default: `False`); if `True`, only non-simple roots are considered
- `facade` – boolean (default: `False`); passes facade option to the poset generator

EXAMPLES:

```
sage: Phi = RootSystem(['A',2]).root_poset(); Phi                          #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
Finite poset containing 3 elements
sage: sorted(Phi.cover_relations(), key=str)                               #␣
↪needs sage.graphs
[[alpha[1], alpha[1] + alpha[2]], [alpha[2], alpha[1] + alpha[2]]]

sage: Phi = RootSystem(['A',3]).root_poset(restricted=True); Phi           #␣
↪needs sage.graphs
Finite poset containing 3 elements
sage: sorted(Phi.cover_relations(), key=str)                               #␣
↪needs sage.graphs
[[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]],
 [alpha[2] + alpha[3], alpha[1] + alpha[2] + alpha[3]]]

sage: Phi = RootSystem(['B',2]).root_poset(); Phi                          #␣
↪needs sage.graphs
Finite poset containing 4 elements
sage: Phi.cover_relations()                                                #␣
↪needs sage.graphs
[[alpha[2], alpha[1] + alpha[2]], [alpha[1], alpha[1] + alpha[2]],
 [alpha[1] + alpha[2], alpha[1] + 2*alpha[2]]]
```

```
>>> from sage.all import *
>>> Phi = RootSystem(['A',Integer(2)]).root_poset(); Phi                  ␣
↪     # needs sage.graphs
Finite poset containing 3 elements
>>> sorted(Phi.cover_relations(), key=str)                               #␣
↪needs sage.graphs
[[alpha[1], alpha[1] + alpha[2]], [alpha[2], alpha[1] + alpha[2]]]

>>> Phi = RootSystem(['A',Integer(3)]).root_poset(restricted=True); Phi   ␣
↪     # needs sage.graphs
Finite poset containing 3 elements
>>> sorted(Phi.cover_relations(), key=str)                               #␣
↪needs sage.graphs
[[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]],
 [alpha[2] + alpha[3], alpha[1] + alpha[2] + alpha[3]]]

>>> Phi = RootSystem(['B',Integer(2)]).root_poset(); Phi                  ␣
↪     # needs sage.graphs
Finite poset containing 4 elements
>>> Phi.cover_relations()                                                #␣
↪needs sage.graphs
[[alpha[2], alpha[1] + alpha[2]], [alpha[1], alpha[1] + alpha[2]],
 [alpha[1] + alpha[2], alpha[1] + 2*alpha[2]]]
```

root_space(*base_ring=Rational Field*)

Return the root space associated to `self`.

EXAMPLES:

```
sage: RootSystem(['A',3]).root_space()
Root space over the Rational Field of the Root system of type ['A', 3]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).root_space()
Root space over the Rational Field of the Root system of type ['A', 3]
```

weight_lattice(*extended=False*)

Return the weight lattice associated to `self`.

> **See also**
>
> - *weight_space()*
> - *coweight_space()*, *coweight_lattice()*
> - WeightSpace

EXAMPLES:

```
sage: RootSystem(['A',3]).weight_lattice()
Weight lattice of the Root system of type ['A', 3]

sage: RootSystem(['A',3,1]).weight_space(extended=True)
Extended weight space over the Rational Field
 of the Root system of type ['A', 3, 1]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).weight_lattice()
Weight lattice of the Root system of type ['A', 3]

>>> RootSystem(['A',Integer(3),Integer(1)]).weight_space(extended=True)
Extended weight space over the Rational Field
 of the Root system of type ['A', 3, 1]
```

weight_space(*base_ring=Rational Field*, *extended=False*)

Return the weight space associated to `self`.

> **See also**
>
> - *weight_lattice()*
> - *coweight_space()*, *coweight_lattice()*
> - WeightSpace

EXAMPLES:

```
sage: RootSystem(['A',3]).weight_space()
Weight space over the Rational Field of the Root system of type ['A', 3]

sage: RootSystem(['A',3,1]).weight_space(extended=True)
Extended weight space over the Rational Field
 of the Root system of type ['A', 3, 1]
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).weight_space()
Weight space over the Rational Field of the Root system of type ['A', 3]

>>> RootSystem(['A',Integer(3),Integer(1)]).weight_space(extended=True)
Extended weight space over the Rational Field
 of the Root system of type ['A', 3, 1]
```

sage.combinat.root_system.root_system.**WeylDim**(*ct*, *coeffs*)

The Weyl Dimension Formula.

INPUT:

- ct – a Cartan type
- coeffs – list of nonnegative integers

The length of the list must equal the rank type[1]. A dominant weight hwv is constructed by summing the fundamental weights with coefficients from this list. The dimension of the irreducible representation of the semisimple complex Lie algebra with highest weight vector hwv is returned.

EXAMPLES:

For $SO(7)$, the Cartan type is B_3, so:

```
sage: WeylDim(['B',3],[1,0,0])  # standard representation of SO(7)
7
sage: WeylDim(['B',3],[0,1,0])  # exterior square
21
sage: WeylDim(['B',3],[0,0,1])  # spin representation of spin(7)
8
sage: WeylDim(['B',3],[1,0,1])  # sum of the first and third fundamental weights
48
sage: [WeylDim(['F',4],x) for x in ([1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1])]
[52, 1274, 273, 26]
sage: [WeylDim(['E', 6], x)
....:  for x in ([0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1],
....:            [0, 0, 0, 0, 0, 2], [0, 0, 0, 0, 1, 0], [0, 0, 1, 0, 0, 0],
....:            [1, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 1], [2, 0, 0, 0, 0, 0])]
[1, 78, 27, 351, 351, 351, 27, 650, 351]
```

```
>>> from sage.all import *
>>> WeylDim(['B',Integer(3)],[Integer(1),Integer(0),Integer(0)])  # standard␣
↪representation of SO(7)
7
>>> WeylDim(['B',Integer(3)],[Integer(0),Integer(1),Integer(0)])  # exterior␣
↪square
21
>>> WeylDim(['B',Integer(3)],[Integer(0),Integer(0),Integer(1)])  # spin␣
↪representation of spin(7)
8
>>> WeylDim(['B',Integer(3)],[Integer(1),Integer(0),Integer(1)])  # sum of the␣
↪first and third fundamental weights
48
>>> [WeylDim(['F',Integer(4)],x) for x in ([Integer(1),Integer(0),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(0),
↪Integer(1)]])]
[52, 1274, 273, 26]
>>> [WeylDim(['E', Integer(6)], x)
...  for x in ([Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0)], [Integer(0), Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(0)], [Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(1)],
...            [Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(2)], [Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)], [Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0)],
...            [Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0)], [Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(1)], [Integer(2), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0)])]
[1, 78, 27, 351, 351, 351, 27, 650, 351]
```

5.1.242 Root system data for super type A

class sage.combinat.root_system.type_super_A.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

The ambient space for (super) type $A(m|n)$.

EXAMPLES:

```
sage: R = RootSystem(['A', [2,1]])
sage: AL = R.ambient_space(); AL
Ambient space of the Root system of type ['A', [2, 1]]
sage: AL.basis()
Finite family {-3: (1, 0, 0, 0, 0),
 -2: (0, 1, 0, 0, 0),
 -1: (0, 0, 1, 0, 0),
 1: (0, 0, 0, 1, 0),
 2: (0, 0, 0, 0, 1)}
```

```
>>> from sage.all import *
>>> R = RootSystem(['A', [Integer(2),Integer(1)]])
>>> AL = R.ambient_space(); AL
Ambient space of the Root system of type ['A', [2, 1]]
>>> AL.basis()
Finite family {-3: (1, 0, 0, 0, 0),
 -2: (0, 1, 0, 0, 0),
 -1: (0, 0, 1, 0, 0),
 1: (0, 0, 0, 1, 0),
 2: (0, 0, 0, 0, 1)}
```

class Element

Bases: *AmbientSpaceElement*

associated_coroot()

Return the coroot associated to `self`.

EXAMPLES:

```
sage: L = RootSystem(['A', [3,2]]).ambient_space()
sage: al = L.simple_roots()
sage: al[-1].associated_coroot()
(0, 0, 1, -1, 0, 0, 0)
sage: al[0].associated_coroot()
(0, 0, 0, 1, -1, 0, 0)
sage: al[1].associated_coroot()
(0, 0, 0, 0, -1, 1, 0)

sage: a = al[-1] + al[0] + al[1]; a
(0, 0, 1, 0, 0, -1, 0)
sage: a.associated_coroot()
(0, 0, 1, 0, -2, 1, 0)
sage: h = L.simple_coroots()
sage: h[-1] + h[0] + h[1]
(0, 0, 1, 0, -2, 1, 0)

sage: (al[-1] + al[0] + al[2]).associated_coroot()
(0, 0, 1, 0, -1, -1, 1)
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', [Integer(3),Integer(2)]]).ambient_space()
>>> al = L.simple_roots()
>>> al[-Integer(1)].associated_coroot()
(0, 0, 1, -1, 0, 0, 0)
>>> al[Integer(0)].associated_coroot()
(0, 0, 0, 1, -1, 0, 0)
>>> al[Integer(1)].associated_coroot()
(0, 0, 0, 0, -1, 1, 0)

>>> a = al[-Integer(1)] + al[Integer(0)] + al[Integer(1)]; a
(0, 0, 1, 0, 0, -1, 0)
>>> a.associated_coroot()
(0, 0, 1, 0, -2, 1, 0)
>>> h = L.simple_coroots()
>>> h[-Integer(1)] + h[Integer(0)] + h[Integer(1)]
(0, 0, 1, 0, -2, 1, 0)

>>> (al[-Integer(1)] + al[Integer(0)] + al[Integer(2)]).associated_
↪coroot()
(0, 0, 1, 0, -1, -1, 1)
```

dot_product(*lambdacheck*)

The scalar product with elements of the coroot lattice embedded in the ambient space.

EXAMPLES:

```
sage: L = RootSystem(['A', [2,1]]).ambient_space()
sage: a = L.simple_roots()
```

(continues on next page)

(continued from previous page)

```
sage: matrix([[a[i].inner_product(a[j]) for j in L.index_set()] for i in␣
↪L.index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  0  1]
[ 0  0  1 -2]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_space()
>>> a = L.simple_roots()
>>> matrix([[a[i].inner_product(a[j]) for j in L.index_set()] for i in L.
↪index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  0  1]
[ 0  0  1 -2]
```

has_descent(*i*, *positive=False*)

Test if `self` has a descent at position i, that is if `self` is on the strict negative side of the i-th simple reflection hyperplane.

If `positive` is `True`, tests if it is on the strict positive side instead.

EXAMPLES:

```
sage: L = RootSystem(['A', [2,1]]).ambient_space()
sage: al = L.simple_roots()
sage: [al[i].has_descent(1) for i in L.index_set()]
[False, False, True, False]
sage: [(-al[i]).has_descent(1) for i in L.index_set()]
[False, False, False, True]
sage: [al[i].has_descent(1, True) for i in L.index_set()]
[False, False, False, True]
sage: [(-al[i]).has_descent(1, True) for i in L.index_set()]
[False, False, True, False]
sage: (al[-2] + al[0] + al[1]).has_descent(-1)
True
sage: (al[-2] + al[0] + al[1]).has_descent(1)
False
sage: (al[-2] + al[0] + al[1]).has_descent(1, positive=True)
True
sage: all(all(not la.has_descent(i) for i in L.index_set())
....:     for la in L.fundamental_weights())
True
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_space()
>>> al = L.simple_roots()
>>> [al[i].has_descent(Integer(1)) for i in L.index_set()]
[False, False, True, False]
>>> [(-al[i]).has_descent(Integer(1)) for i in L.index_set()]
[False, False, False, True]
```

(continues on next page)

(continued from previous page)

```
>>> [al[i].has_descent(Integer(1), True) for i in L.index_set()]
[False, False, False, True]
>>> [(-al[i]).has_descent(Integer(1), True) for i in L.index_set()]
[False, False, True, False]
>>> (al[-Integer(2)] + al[Integer(0)] + al[Integer(1)]).has_descent(-
↪Integer(1))
True
>>> (al[-Integer(2)] + al[Integer(0)] + al[Integer(1)]).has_
↪descent(Integer(1))
False
>>> (al[-Integer(2)] + al[Integer(0)] + al[Integer(1)]).has_
↪descent(Integer(1), positive=True)
True
>>> all(all(not la.has_descent(i) for i in L.index_set())
...     for la in L.fundamental_weights())
True
```

inner_product(*lambdacheck*)

The scalar product with elements of the coroot lattice embedded in the ambient space.

EXAMPLES:

```
sage: L = RootSystem(['A', [2,1]]).ambient_space()
sage: a = L.simple_roots()
sage: matrix([[a[i].inner_product(a[j]) for j in L.index_set()] for i in␣
↪L.index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  0  1]
[ 0  0  1 -2]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_space()
>>> a = L.simple_roots()
>>> matrix([[a[i].inner_product(a[j]) for j in L.index_set()] for i in L.
↪index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  0  1]
[ 0  0  1 -2]
```

is_dominant_weight()

Test whether `self` is a dominant element of the weight lattice.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).ambient_lattice()
sage: Lambda = L.fundamental_weights()
sage: [x.is_dominant() for x in Lambda]
[True, True]
sage: (3*Lambda[1]+Lambda[2]).is_dominant()
True
sage: (Lambda[1]-Lambda[2]).is_dominant()
```

(continues on next page)

(continued from previous page)

```
False
sage: (-Lambda[1]+Lambda[2]).is_dominant()
False
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2)]).ambient_lattice()
>>> Lambda = L.fundamental_weights()
>>> [x.is_dominant() for x in Lambda]
[True, True]
>>> (Integer(3)*Lambda[Integer(1)]+Lambda[Integer(2)]).is_dominant()
True
>>> (Lambda[Integer(1)]-Lambda[Integer(2)]).is_dominant()
False
>>> (-Lambda[Integer(1)]+Lambda[Integer(2)]).is_dominant()
False
```

Tests that the scalar products with the coroots are all nonnegative integers. For example, if x is the sum of a dominant element of the weight lattice plus some other element orthogonal to all coroots, then the implementation correctly reports x to be a dominant weight:

```
sage: x = Lambda[1] + L([-1,-1,-1])
sage: x.is_dominant_weight()
True
```

```
>>> from sage.all import *
>>> x = Lambda[Integer(1)] + L([-Integer(1),-Integer(1),-Integer(1)])
>>> x.is_dominant_weight()
True
```

scalar(*lambdacheck*)

The scalar product with elements of the coroot lattice embedded in the ambient space.

EXAMPLES:

```
sage: L = RootSystem(['A', [2,1]]).ambient_space()
sage: a = L.simple_roots()
sage: matrix([[a[i].inner_product(a[j]) for j in L.index_set()] for i in␣
↪L.index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  0  1]
[ 0  0  1 -2]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_space()
>>> a = L.simple_roots()
>>> matrix([[a[i].inner_product(a[j]) for j in L.index_set()] for i in L.
↪index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  0  1]
[ 0  0  1 -2]
```

dimension()

Return the dimension of this ambient space.

EXAMPLES:

```
sage: e = RootSystem(['A', [4,2]]).ambient_space()
sage: e.dimension()
8
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(4),Integer(2)]]).ambient_space()
>>> e.dimension()
8
```

fundamental_weight(i)

Return the fundamental weight Λ_i of `self`.

EXAMPLES:

```
sage: L = RootSystem(['A', [3,2]]).ambient_space()
sage: L.fundamental_weight(-1)
(1, 1, 1, 0, 0, 0, 0)
sage: L.fundamental_weight(0)
(1, 1, 1, 1, 0, 0, 0)
sage: L.fundamental_weight(2)
(1, 1, 1, 1, -1, -1, -2)
sage: list(L.fundamental_weights())
[(1, 0, 0, 0, 0, 0, 0),
 (1, 1, 0, 0, 0, 0, 0),
 (1, 1, 1, 0, 0, 0, 0),
 (1, 1, 1, 1, 0, 0, 0),
 (1, 1, 1, 1, -1, -2, -2),
 (1, 1, 1, 1, -1, -1, -2)]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', [Integer(3),Integer(2)]]).ambient_space()
>>> L.fundamental_weight(-Integer(1))
(1, 1, 1, 0, 0, 0, 0)
>>> L.fundamental_weight(Integer(0))
(1, 1, 1, 1, 0, 0, 0)
>>> L.fundamental_weight(Integer(2))
(1, 1, 1, 1, -1, -1, -2)
>>> list(L.fundamental_weights())
[(1, 0, 0, 0, 0, 0, 0),
 (1, 1, 0, 0, 0, 0, 0),
 (1, 1, 1, 0, 0, 0, 0),
 (1, 1, 1, 1, 0, 0, 0),
 (1, 1, 1, 1, -1, -2, -2),
 (1, 1, 1, 1, -1, -1, -2)]
```

```
sage: L = RootSystem(['A', [2,3]]).ambient_space()
sage: La = L.fundamental_weights()
sage: al = L.simple_roots()
```

(continues on next page)

(continued from previous page)

```
sage: I = L.index_set()
sage: matrix([[al[i].scalar(La[j]) for i in I] for j in I])
[ 1  0  0  0  0  0]
[ 0  1  0  0  0  0]
[ 0  0  1  0  0  0]
[ 0  0  0 -1  0  0]
[ 0  0  0  0 -1  0]
[ 0  0  0  0  0 -1]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', [Integer(2),Integer(3)]]).ambient_space()
>>> La = L.fundamental_weights()
>>> al = L.simple_roots()
>>> I = L.index_set()
>>> matrix([[al[i].scalar(La[j]) for i in I] for j in I])
[ 1  0  0  0  0  0]
[ 0  1  0  0  0  0]
[ 0  0  1  0  0  0]
[ 0  0  0 -1  0  0]
[ 0  0  0  0 -1  0]
[ 0  0  0  0  0 -1]
```

highest_root()

Return the highest root of `self`.

EXAMPLES:

```
sage: e = RootSystem(['A', [4,2]]).ambient_lattice()
sage: e.highest_root()
(1, 0, 0, 0, 0, 0, 0, -1)
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(4),Integer(2)]]).ambient_lattice()
>>> e.highest_root()
(1, 0, 0, 0, 0, 0, 0, -1)
```

negative_even_roots()

Return the negative even roots of `self`.

EXAMPLES:

```
sage: e = RootSystem(['A', [2,1]]).ambient_lattice()
sage: e.negative_even_roots()
[(0, -1, 1, 0, 0), (-1, 0, 1, 0, 0),
 (-1, 1, 0, 0, 0), (0, 0, 0, -1, 1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_lattice()
>>> e.negative_even_roots()
[(0, -1, 1, 0, 0), (-1, 0, 1, 0, 0),
 (-1, 1, 0, 0, 0), (0, 0, 0, -1, 1)]
```

negative_odd_roots()

Return the negative odd roots of `self`.

EXAMPLES:

```
sage: e = RootSystem(['A', [2,1]]).ambient_lattice()
sage: e.negative_odd_roots()
[(0, 0, -1, 1, 0),
 (0, 0, -1, 0, 1),
 (0, -1, 0, 1, 0),
 (0, -1, 0, 0, 1),
 (-1, 0, 0, 1, 0),
 (-1, 0, 0, 0, 1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_lattice()
>>> e.negative_odd_roots()
[(0, 0, -1, 1, 0),
 (0, 0, -1, 0, 1),
 (0, -1, 0, 1, 0),
 (0, -1, 0, 0, 1),
 (-1, 0, 0, 1, 0),
 (-1, 0, 0, 0, 1)]
```

negative_roots()

Return the negative roots of `self`.

EXAMPLES:

```
sage: e = RootSystem(['A', [2,1]]).ambient_lattice()
sage: e.negative_roots()
[(0, -1, 1, 0, 0),
 (-1, 0, 1, 0, 0),
 (-1, 1, 0, 0, 0),
 (0, 0, 0, -1, 1),
 (0, 0, -1, 1, 0),
 (0, 0, -1, 0, 1),
 (0, -1, 0, 1, 0),
 (0, -1, 0, 0, 1),
 (-1, 0, 0, 1, 0),
 (-1, 0, 0, 0, 1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_lattice()
>>> e.negative_roots()
[(0, -1, 1, 0, 0),
 (-1, 0, 1, 0, 0),
 (-1, 1, 0, 0, 0),
 (0, 0, 0, -1, 1),
 (0, 0, -1, 1, 0),
 (0, 0, -1, 0, 1),
 (0, -1, 0, 1, 0),
 (0, -1, 0, 0, 1),
```

(continues on next page)

(continued from previous page)

```
 (-1, 0, 0, 1, 0),
 (-1, 0, 0, 0, 1)]
```

positive_even_roots()

Return the positive even roots of `self`.

EXAMPLES:

```
sage: e = RootSystem(['A', [2,1]]).ambient_lattice()
sage: e.positive_even_roots()
[(0, 1, -1, 0, 0), (1, 0, -1, 0, 0),
 (1, -1, 0, 0, 0), (0, 0, 0, 1, -1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_lattice()
>>> e.positive_even_roots()
[(0, 1, -1, 0, 0), (1, 0, -1, 0, 0),
 (1, -1, 0, 0, 0), (0, 0, 0, 1, -1)]
```

positive_odd_roots()

Return the positive odd roots of `self`.

EXAMPLES:

```
sage: e = RootSystem(['A', [2,1]]).ambient_lattice()
sage: e.positive_odd_roots()
[(0, 0, 1, -1, 0),
 (0, 0, 1, 0, -1),
 (0, 1, 0, -1, 0),
 (0, 1, 0, 0, -1),
 (1, 0, 0, -1, 0),
 (1, 0, 0, 0, -1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_lattice()
>>> e.positive_odd_roots()
[(0, 0, 1, -1, 0),
 (0, 0, 1, 0, -1),
 (0, 1, 0, -1, 0),
 (0, 1, 0, 0, -1),
 (1, 0, 0, -1, 0),
 (1, 0, 0, 0, -1)]
```

positive_roots()

Return the positive roots of `self`.

EXAMPLES:

```
sage: e = RootSystem(['A', [2,1]]).ambient_lattice()
sage: e.positive_roots()
[(0, 1, -1, 0, 0),
 (1, 0, -1, 0, 0),
 (1, -1, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 0, 0, 1, -1),
 (0, 0, 1, -1, 0),
 (0, 0, 1, 0, -1),
 (0, 1, 0, -1, 0),
 (0, 1, 0, 0, -1),
 (1, 0, 0, -1, 0),
 (1, 0, 0, 0, -1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_lattice()
>>> e.positive_roots()
[(0, 1, -1, 0, 0),
 (1, 0, -1, 0, 0),
 (1, -1, 0, 0, 0),
 (0, 0, 0, 1, -1),
 (0, 0, 1, -1, 0),
 (0, 0, 1, 0, -1),
 (0, 1, 0, -1, 0),
 (0, 1, 0, 0, -1),
 (1, 0, 0, -1, 0),
 (1, 0, 0, 0, -1)]
```

simple_coroot(i)

Return the simple coroot h_i of `self`.

EXAMPLES:

```
sage: L = RootSystem(['A', [3,2]]).ambient_space()
sage: L.simple_coroot(-2)
(0, 1, -1, 0, 0, 0, 0)
sage: L.simple_coroot(0)
(0, 0, 0, 1, -1, 0, 0)
sage: L.simple_coroot(2)
(0, 0, 0, 0, 0, -1, 1)
sage: list(L.simple_coroots())
[(1, -1, 0, 0, 0, 0, 0),
 (0, 1, -1, 0, 0, 0, 0),
 (0, 0, 1, -1, 0, 0, 0),
 (0, 0, 0, 1, -1, 0, 0),
 (0, 0, 0, 0, -1, 1, 0),
 (0, 0, 0, 0, 0, -1, 1)]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A', [Integer(3),Integer(2)]]).ambient_space()
>>> L.simple_coroot(-Integer(2))
(0, 1, -1, 0, 0, 0, 0)
>>> L.simple_coroot(Integer(0))
(0, 0, 0, 1, -1, 0, 0)
>>> L.simple_coroot(Integer(2))
(0, 0, 0, 0, 0, -1, 1)
>>> list(L.simple_coroots())
[(1, -1, 0, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 1, -1, 0, 0, 0, 0),
 (0, 0, 1, -1, 0, 0, 0),
 (0, 0, 0, 1, -1, 0, 0),
 (0, 0, 0, 0, -1, 1, 0),
 (0, 0, 0, 0, 0, -1, 1)]
```

simple_root(*i*)

Return the i-th simple root of `self`.

EXAMPLES:

```
sage: e = RootSystem(['A', [2,1]]).ambient_lattice()
sage: list(e.simple_roots())
[(1, -1, 0, 0, 0), (0, 1, -1, 0, 0),
 (0, 0, 1, -1, 0), (0, 0, 0, 1, -1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(2),Integer(1)]]).ambient_lattice()
>>> list(e.simple_roots())
[(1, -1, 0, 0, 0), (0, 1, -1, 0, 0),
 (0, 0, 1, -1, 0), (0, 0, 0, 1, -1)]
```

classmethod smallest_base_ring(*cartan_type=None*)

Return the smallest base ring the ambient space can be defined upon.

See also

smallest_base_ring()

EXAMPLES:

```
sage: e = RootSystem(['A', [3,1]]).ambient_space()
sage: e.smallest_base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> e = RootSystem(['A', [Integer(3),Integer(1)]]).ambient_space()
>>> e.smallest_base_ring()
Integer Ring
```

class sage.combinat.root_system.type_super_A.**CartanType**(*m*, *n*)

Bases: *SuperCartanType_standard*

Cartan Type $A(m|n)$.

See also

`CartanType()`

AmbientSpace

alias of *AmbientSpace*

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the Dynkin diagram.

EXAMPLES:

```
sage: t = CartanType(['A', [3,2]])
sage: print(t.ascii_art())
O---O---O---X---O---O
-3  -2  -1  0   1   2
sage: t = CartanType(['A', [3,7]])
sage: print(t.ascii_art())
O---O---O---X---O---O---O---O---O---O---O
-3  -2  -1  0   1   2   3   4   5   6   7

sage: t = CartanType(['A', [0,7]])
sage: print(t.ascii_art())
X---O---O---O---O---O---O---O
0   1   2   3   4   5   6   7
sage: t = CartanType(['A', [0,0]])
sage: print(t.ascii_art())
X
0
sage: t = CartanType(['A', [5,0]])
sage: print(t.ascii_art())
O---O---O---O---O---X
-5  -4  -3  -2  -1  0
```

```
>>> from sage.all import *
>>> t = CartanType(['A', [Integer(3),Integer(2)]])
>>> print(t.ascii_art())
O---O---O---X---O---O
-3  -2  -1  0   1   2
>>> t = CartanType(['A', [Integer(3),Integer(7)]])
>>> print(t.ascii_art())
O---O---O---X---O---O---O---O---O---O---O
-3  -2  -1  0   1   2   3   4   5   6   7

>>> t = CartanType(['A', [Integer(0),Integer(7)]])
>>> print(t.ascii_art())
X---O---O---O---O---O---O---O
0   1   2   3   4   5   6   7
>>> t = CartanType(['A', [Integer(0),Integer(0)]])
>>> print(t.ascii_art())
X
0
>>> t = CartanType(['A', [Integer(5),Integer(0)]])
>>> print(t.ascii_art())
O---O---O---O---O---X
-5  -4  -3  -2  -1  0
```

cartan_matrix()

Return the Cartan matrix associated to `self`.

EXAMPLES:

```
sage: ct = CartanType(['A', [2,3]])
sage: ct.cartan_matrix()                                                    #␣
↪needs sage.graphs
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  0  1  0  0]
[ 0  0 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -1  2]
```

```
>>> from sage.all import *
>>> ct = CartanType(['A', [Integer(2),Integer(3)]])
>>> ct.cartan_matrix()                                                    #␣
↪needs sage.graphs
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  0  1  0  0]
[ 0  0 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -1  2]
```

dual()

Return dual of `self`.

EXAMPLES:

```
sage: CartanType(['A', [2,3]]).dual()
['A', [2, 3]]
```

```
>>> from sage.all import *
>>> CartanType(['A', [Integer(2),Integer(3)]]).dual()
['A', [2, 3]]
```

dynkin_diagram()

Return the Dynkin diagram of super type A.

EXAMPLES:

```
sage: a = CartanType(['A', [4,2]]).dynkin_diagram(); a                     #␣
↪needs sage.graphs
O---O---O---O---X---O---O
-4  -3  -2  -1  0   1   2
A4|2
sage: a.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(-4, -3, 1), (-3, -4, 1), (-3, -2, 1), (-2, -3, 1),
 (-2, -1, 1), (-1, -2, 1), (-1, 0, 1), (0, -1, 1),
 (0, 1, 1), (1, 0, -1), (1, 2, 1), (2, 1, 1)]
```

```
>>> from sage.all import *
>>> a = CartanType(['A', [Integer(4),Integer(2)]]).dynkin_diagram(); a     ␣
↪               # needs sage.graphs
O---O---O---O---X---O---O
```

(continues on next page)

(continued from previous page)

```
-4  -3  -2  -1  0   1   2
A4|2
>>> a.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(-4, -3, 1), (-3, -4, 1), (-3, -2, 1), (-2, -3, 1),
 (-2, -1, 1), (-1, -2, 1), (-1, 0, 1), (0, -1, 1),
 (0, 1, 1), (1, 0, -1), (1, 2, 1), (2, 1, 1)]
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: CartanType(['A', [2,3]]).index_set()
(-2, -1, 0, 1, 2, 3)
```

```
>>> from sage.all import *
>>> CartanType(['A', [Integer(2),Integer(3)]]).index_set()
(-2, -1, 0, 1, 2, 3)
```

is_affine()

Return whether `self` is affine or not.

EXAMPLES:

```
sage: CartanType(['A', [2,3]]).is_affine()
False
```

```
>>> from sage.all import *
>>> CartanType(['A', [Integer(2),Integer(3)]]).is_affine()
False
```

is_finite()

Return whether `self` is finite or not.

EXAMPLES:

```
sage: CartanType(['A', [2,3]]).is_finite()
True
```

```
>>> from sage.all import *
>>> CartanType(['A', [Integer(2),Integer(3)]]).is_finite()
True
```

is_irreducible()

Return whether `self` is irreducible, which is `True`.

EXAMPLES:

```
sage: CartanType(['A', [3,4]]).is_irreducible()
True
```

```
>>> from sage.all import *
>>> CartanType(['A', [Integer(3),Integer(4)]]).is_irreducible()
True
```

relabel(*relabelling*)

Return a relabelled copy of this Cartan type.

INPUT:

- `relabelling` – a function (or a list or dictionary)

OUTPUT:

an isomorphic Cartan type obtained by relabelling the nodes of the Dynkin diagram. Namely, the node with label `i` is relabelled `f(i)` (or, by `f[i]` if `f` is a list or dictionary).

EXAMPLES:

```
sage: ct = CartanType(['A', [1,2]])
sage: ct.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---X---O---O
-1   0   1   2
A1|2
sage: f = {1:2, 2:1, 0:0, -1:-1}
sage: ct.relabel(f)
['A', [1, 2]] relabelled by {-1: -1, 0: 0, 1: 2, 2: 1}
sage: ct.relabel(f).dynkin_diagram()                                         #␣
↪needs sage.graphs
O---X---O---O
-1   0   2   1
A1|2 relabelled by {-1: -1, 0: 0, 1: 2, 2: 1}
```

```
>>> from sage.all import *
>>> ct = CartanType(['A', [Integer(1),Integer(2)]])
>>> ct.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---X---O---O
-1   0   1   2
A1|2
>>> f = {Integer(1):Integer(2), Integer(2):Integer(1), Integer(0):Integer(0),␣
↪-Integer(1):-Integer(1)}
>>> ct.relabel(f)
['A', [1, 2]] relabelled by {-1: -1, 0: 0, 1: 2, 2: 1}
>>> ct.relabel(f).dynkin_diagram()                                         #␣
↪needs sage.graphs
O---X---O---O
-1   0   2   1
A1|2 relabelled by {-1: -1, 0: 0, 1: 2, 2: 1}
```

root_system()

Return root system of `self`.

EXAMPLES:

```
sage: CartanType(['A', [2,3]]).root_system()
Root system of type ['A', [2, 3]]
```

```
>>> from sage.all import *
>>> CartanType(['A', [Integer(2),Integer(3)]]).root_system()
Root system of type ['A', [2, 3]]
```

symmetrizer()

Return symmetrizing matrix for `self`.

EXAMPLES:

```
sage: CartanType(['A', [2,3]]).symmetrizer()
Finite family {-2: 1, -1: 1, 0: 1, 1: -1, 2: -1, 3: -1}
```

```
>>> from sage.all import *
>>> CartanType(['A', [Integer(2),Integer(3)]]).symmetrizer()
Finite family {-2: 1, -1: 1, 0: 1, 1: -1, 2: -1, 3: -1}
```

type()

Return type of `self`.

EXAMPLES:

```
sage: CartanType(['A', [2,3]]).type()
'A'
```

```
>>> from sage.all import *
>>> CartanType(['A', [Integer(2),Integer(3)]]).type()
'A'
```

5.1.243 Root system data for type A

class sage.combinat.root_system.type_A.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

EXAMPLES:

```
sage: R = RootSystem(["A",3])
sage: e = R.ambient_space(); e
Ambient space of the Root system of type ['A', 3]
sage: TestSuite(e).run()                                                    #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> R = RootSystem(["A",Integer(3)])
>>> e = R.ambient_space(); e
Ambient space of the Root system of type ['A', 3]
>>> TestSuite(e).run()                                                      #␣
↪needs sage.graphs
```

By default, this ambient space uses the barycentric projection for plotting:

```
sage: # needs sage.symbolic
sage: L = RootSystem(["A",2]).ambient_space()
sage: e = L.basis()
sage: L._plot_projection(e[0])
(1/2, 989/1142)
sage: L._plot_projection(e[1])
(-1, 0)
sage: L._plot_projection(e[2])
(1/2, -989/1142)
sage: L = RootSystem(["A",3]).ambient_space()
sage: l = L.an_element(); l
(2, 2, 3, 0)
sage: L._plot_projection(l)
(0, -1121/1189, 7/3)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> L = RootSystem(["A",Integer(2)]).ambient_space()
>>> e = L.basis()
>>> L._plot_projection(e[Integer(0)])
(1/2, 989/1142)
>>> L._plot_projection(e[Integer(1)])
(-1, 0)
>>> L._plot_projection(e[Integer(2)])
(1/2, -989/1142)
>>> L = RootSystem(["A",Integer(3)]).ambient_space()
>>> l = L.an_element(); l
(2, 2, 3, 0)
>>> L._plot_projection(l)
(0, -1121/1189, 7/3)
```

See also

- `sage.combinat.root_system.root_lattice_realizations.RootLatticeRealizations.ParentMethods._plot_projection()`

det(*k=1*)

Return the vector (1, … ,1) which in the ['A',r] weight lattice, interpreted as a weight of GL(r+1,CC) is the determinant. If the optional parameter k is given, returns (k, … ,k), the k-th power of the determinant.

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_space()
sage: e.det(1/2)
(1/2, 1/2, 1/2, 1/2)
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(3)]).ambient_space()
>>> e.det(Integer(1)/Integer(2))
(1/2, 1/2, 1/2, 1/2)
```

dimension()

EXAMPLES:

```
sage: e = RootSystem(["A",3]).ambient_space()
sage: e.dimension()
4
```

```
>>> from sage.all import *
>>> e = RootSystem(["A",Integer(3)]).ambient_space()
>>> e.dimension()
4
```

fundamental_weight(*i*)

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(3)]).ambient_lattice()
>>> e.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
```

highest_root()

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.highest_root()
(1, 0, 0, -1)
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(3)]).ambient_lattice()
>>> e.highest_root()
(1, 0, 0, -1)
```

negative_roots()

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.negative_roots()
[(-1, 1, 0, 0),
 (-1, 0, 1, 0),
 (-1, 0, 0, 1),
 (0, -1, 1, 0),
 (0, -1, 0, 1),
 (0, 0, -1, 1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(3)]).ambient_lattice()
>>> e.negative_roots()
[(-1, 1, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (-1, 0, 1, 0),
 (-1, 0, 0, 1),
 (0, -1, 1, 0),
 (0, -1, 0, 1),
 (0, 0, -1, 1)]
```

positive_roots()

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.positive_roots()
[(1, -1, 0, 0),
 (1, 0, -1, 0),
 (0, 1, -1, 0),
 (1, 0, 0, -1),
 (0, 1, 0, -1),
 (0, 0, 1, -1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(3)]).ambient_lattice()
>>> e.positive_roots()
[(1, -1, 0, 0),
 (1, 0, -1, 0),
 (0, 1, -1, 0),
 (1, 0, 0, -1),
 (0, 1, 0, -1),
 (0, 0, 1, -1)]
```

root(i, j)

Note that indexing starts at 0.

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.root(0,1)
(1, -1, 0, 0)
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(3)]).ambient_lattice()
>>> e.root(Integer(0),Integer(1))
(1, -1, 0, 0)
```

simple_root(i)

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(3)]).ambient_lattice()
```

(continues on next page)

(continued from previous page)

```
>>> e.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

classmethod smallest_base_ring(*cartan_type=None*)

Return the smallest base ring the ambient space can be defined upon.

See also:

smallest_base_ring()

EXAMPLES:

```
sage: e = RootSystem(["A",3]).ambient_space()
sage: e.smallest_base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> e = RootSystem(["A",Integer(3)]).ambient_space()
>>> e.smallest_base_ring()
Integer Ring
```

class sage.combinat.root_system.type_A.**CartanType**(*n*)

Bases: *CartanType_standard_finite*, *CartanType_simply_laced*, *CartanType_simple*

Cartan Type A_n.

See also:

CartanType()

AmbientSpace

alias of *AmbientSpace*

PieriFactors

alias of *PieriFactors_type_A*

ascii_art(*label=None, node=None*)

Return an ascii art representation of the Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['A',0]).ascii_art())
sage: print(CartanType(['A',1]).ascii_art())
O
1
sage: print(CartanType(['A',3]).ascii_art())
O---O---O
1   2   3
sage: print(CartanType(['A',12]).ascii_art())
O---O---O---O---O---O---O---O---O---O---O---O
1   2   3   4   5   6   7   8   9   10  11  12
```

(continues on next page)

(continued from previous page)

```
sage: print(CartanType(['A',5]).ascii_art(label = lambda x: x+2))
O---O---O---O---O
3   4   5   6   7
sage: print(CartanType(['A',5]).ascii_art(label = lambda x: x-2))
O---O---O---O---O
-1  0   1   2   3
```

```
>>> from sage.all import *
>>> print(CartanType(['A',Integer(0)]).ascii_art())
>>> print(CartanType(['A',Integer(1)]).ascii_art())
O
1
>>> print(CartanType(['A',Integer(3)]).ascii_art())
O---O---O
1   2   3
>>> print(CartanType(['A',Integer(12)]).ascii_art())
O---O---O---O---O---O---O---O---O---O---O---O
1   2   3   4   5   6   7   8   9   10  11  12
>>> print(CartanType(['A',Integer(5)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
O---O---O---O---O
3   4   5   6   7
>>> print(CartanType(['A',Integer(5)]).ascii_art(label = lambda x: x-
↪Integer(2)))
O---O---O---O---O
-1  0   1   2   3
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).coxeter_number()
5
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).coxeter_number()
5
```

dual_coxeter_number()

Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).dual_coxeter_number()
5
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).dual_coxeter_number()
5
```

dynkin_diagram()

Return the Dynkin diagram of type A.

EXAMPLES:

```
sage: a = CartanType(['A',3]).dynkin_diagram(); a                              #␣
↪needs sage.graphs
O---O---O
1   2   3
A3
sage: a.edges(sort=True)                                                       #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 1)]
```

```
>>> from sage.all import *
>>> a = CartanType(['A',Integer(3)]).dynkin_diagram(); a                        ␣
↪      # needs sage.graphs
O---O---O
1   2   3
A3
>>> a.edges(sort=True)                                                        #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 1)]
```

5.1.244 Root system data for (untwisted) type A affine

class sage.combinat.root_system.type_A_affine.**CartanType**(*n*)

Bases: *CartanType_standard_untwisted_affine*

EXAMPLES:

```
sage: ct = CartanType(['A',4,1])
sage: ct
['A', 4, 1]
sage: ct._repr_(compact = True)
'A4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.classical()
['A', 4]
sage: ct.dual()
['A', 4, 1]

sage: ct = CartanType(['A', 1, 1])
sage: ct.is_simply_laced()
```

(continues on next page)

(continued from previous page)

```
False
sage: ct.dual()
['A', 1, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(4),Integer(1)])
>>> ct
['A', 4, 1]
>>> ct._repr_(compact = True)
'A4~'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
False
>>> ct.is_affine()
True
>>> ct.is_untwisted_affine()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
True
>>> ct.classical()
['A', 4]
>>> ct.dual()
['A', 4, 1]

>>> ct = CartanType(['A', Integer(1), Integer(1)])
>>> ct.is_simply_laced()
False
>>> ct.dual()
['A', 1, 1]
```

PieriFactors

alias of *PieriFactors_type_A_affine*

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['A',3,1]).ascii_art())
0
O-------+
|       |
|       |
O---O---O
1   2   3

sage: print(CartanType(['A',5,1]).ascii_art(label = lambda x: x+2))
2
```

(continues on next page)

(continued from previous page)

```
O--------------+
|              |
|              |
O---O---O---O---O
3   4   5   6   7

sage: print(CartanType(['A',1,1]).ascii_art())
O<=>O
0   1

sage: print(CartanType(['A',1,1]).ascii_art(label = lambda x: x+2))
O<=>O
2   3
```

```
>>> from sage.all import *
>>> print(CartanType(['A',Integer(3),Integer(1)]).ascii_art())
0
O-------+
|       |
|       |
O---O---O
1   2   3

>>> print(CartanType(['A',Integer(5),Integer(1)]).ascii_art(label = lambda x:␣
→x+Integer(2)))
2
O--------------+
|              |
|              |
O---O---O---O---O
3   4   5   6   7

>>> print(CartanType(['A',Integer(1),Integer(1)]).ascii_art())
O<=>O
0   1

>>> print(CartanType(['A',Integer(1),Integer(1)]).ascii_art(label = lambda x:␣
→x+Integer(2)))
O<=>O
2   3
```

dual()

Type A_1^1 is self dual despite not being simply laced.

EXAMPLES:

```
sage: CartanType(['A',1,1]).dual()
['A', 1, 1]
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(1),Integer(1)]).dual()
['A', 1, 1]
```

dynkin_diagram()

Return the extended Dynkin diagram for affine type A.

EXAMPLES:

```
sage: a = CartanType(['A',3,1]).dynkin_diagram(); a                          #␣
↪needs sage.graphs
 0
 O-------+
 |       |
 |       |
 O---O---O
 1   2   3
 A3~
sage: a.edges(sort=True)                                                     #␣
↪needs sage.graphs
[(0, 1, 1),
 (0, 3, 1),
 (1, 0, 1),
 (1, 2, 1),
 (2, 1, 1),
 (2, 3, 1),
 (3, 0, 1),
 (3, 2, 1)]

sage: a = DynkinDiagram(['A',1,1]); a                                        #␣
↪needs sage.graphs
O<=>O
0   1
A1~
sage: a.edges(sort=True)                                                     #␣
↪needs sage.graphs
[(0, 1, 2), (1, 0, 2)]
```

```
>>> from sage.all import *
>>> a = CartanType(['A',Integer(3),Integer(1)]).dynkin_diagram(); a          ␣
↪                # needs sage.graphs
 0
 O-------+
 |       |
 |       |
 O---O---O
 1   2   3
 A3~
>>> a.edges(sort=True)                                                     #␣
↪needs sage.graphs
[(0, 1, 1),
 (0, 3, 1),
 (1, 0, 1),
 (1, 2, 1),
 (2, 1, 1),
 (2, 3, 1),
 (3, 0, 1),
```

(continues on next page)

(continued from previous page)

```
 (3, 2, 1)]

>>> a = DynkinDiagram(['A',Integer(1),Integer(1)]); a                    ␣
↪                # needs sage.graphs
O<=>O
0   1
A1~
>>> a.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(0, 1, 2), (1, 0, 2)]
```

5.1.245 Root system data for type A infinity

class sage.combinat.root_system.type_A_infinity.**CartanType**(*index_set*)

Bases: *CartanType_standard*, *CartanType_simple*

The Cartan type A_∞.

We use NN and ZZ to explicitly differentiate between the $A_{+\infty}$ and A_∞ root systems, respectively. While oo is the same as +Infinity in Sage, it is used as an alias for ZZ.

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['A', ZZ]).ascii_art())
..---O---O---O---O---O---O---O---..
    -3  -2  -1   0   1   2   3
sage: print(CartanType(['A', NN]).ascii_art())
O---O---O---O---O---O---O---..
0   1   2   3   4   5   6
```

```
>>> from sage.all import *
>>> print(CartanType(['A', ZZ]).ascii_art())
..---O---O---O---O---O---O---O---..
    -3  -2  -1   0   1   2   3
>>> print(CartanType(['A', NN]).ascii_art())
O---O---O---O---O---O---O---..
0   1   2   3   4   5   6
```

dual()

Simply laced Cartan types are self-dual, so return self.

EXAMPLES:

```
sage: CartanType(["A", NN]).dual()
['A', NN]
sage: CartanType(["A", ZZ]).dual()
['A', ZZ]
```

```
>>> from sage.all import *
>>> CartanType(["A", NN]).dual()
```

(continues on next page)

(continued from previous page)

```
['A', NN]
>>> CartanType(["A", ZZ]).dual()
['A', ZZ]
```

index_set()

Return the index set for the Cartan type `self`.

The index set for all standard finite Cartan types is of the form $\{1, \ldots, n\}$. (See `type_I` for a slight abuse of this).

EXAMPLES:

```
sage: CartanType(['A', NN]).index_set()
Non negative integer semiring
sage: CartanType(['A', ZZ]).index_set()
Integer Ring
```

```
>>> from sage.all import *
>>> CartanType(['A', NN]).index_set()
Non negative integer semiring
>>> CartanType(['A', ZZ]).index_set()
Integer Ring
```

is_affine()

Return `False` because `self` is not (untwisted) affine.

EXAMPLES:

```
sage: CartanType(['A', NN]).is_affine()
False
sage: CartanType(['A', ZZ]).is_affine()
False
```

```
>>> from sage.all import *
>>> CartanType(['A', NN]).is_affine()
False
>>> CartanType(['A', ZZ]).is_affine()
False
```

is_crystallographic()

Return `False` because `self` is not crystallographic.

EXAMPLES:

```
sage: CartanType(['A', NN]).is_crystallographic()
True
sage: CartanType(['A', ZZ]).is_crystallographic()
True
```

```
>>> from sage.all import *
>>> CartanType(['A', NN]).is_crystallographic()
True
>>> CartanType(['A', ZZ]).is_crystallographic()
True
```

is_finite()

Return True because self is not finite.

EXAMPLES:

```
sage: CartanType(['A', NN]).is_finite()
False
sage: CartanType(['A', ZZ]).is_finite()
False
```

```
>>> from sage.all import *
>>> CartanType(['A', NN]).is_finite()
False
>>> CartanType(['A', ZZ]).is_finite()
False
```

is_simply_laced()

Return True because self is simply laced.

EXAMPLES:

```
sage: CartanType(['A', NN]).is_simply_laced()
True
sage: CartanType(['A', ZZ]).is_simply_laced()
True
```

```
>>> from sage.all import *
>>> CartanType(['A', NN]).is_simply_laced()
True
>>> CartanType(['A', ZZ]).is_simply_laced()
True
```

is_untwisted_affine()

Return False because self is not (untwisted) affine.

EXAMPLES:

```
sage: CartanType(['A', NN]).is_untwisted_affine()
False
sage: CartanType(['A', ZZ]).is_untwisted_affine()
False
```

```
>>> from sage.all import *
>>> CartanType(['A', NN]).is_untwisted_affine()
False
>>> CartanType(['A', ZZ]).is_untwisted_affine()
False
```

rank()

Return the rank of self which for type X_n is n.

EXAMPLES:

```
sage: CartanType(['A', NN]).rank()
+Infinity
sage: CartanType(['A', ZZ]).rank()
+Infinity
```

```
>>> from sage.all import *
>>> CartanType(['A', NN]).rank()
+Infinity
>>> CartanType(['A', ZZ]).rank()
+Infinity
```

As this example shows, the rank is slightly ambiguous because the root systems of type $['A', NN]$ and type $['A', ZZ]$ have the same rank. Instead, it is better ot use `index_set()` to differentiate between these two root systems.

type()

Return the type of `self`.

EXAMPLES:

```
sage: CartanType(['A', NN]).type()
'A'
sage: CartanType(['A', ZZ]).type()
'A'
```

```
>>> from sage.all import *
>>> CartanType(['A', NN]).type()
'A'
>>> CartanType(['A', ZZ]).type()
'A'
```

5.1.246 Root system data for type B

class sage.combinat.root_system.type_B.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: `AmbientSpace`

dimension()

EXAMPLES:

```
sage: e = RootSystem(['B',3]).ambient_space()
sage: e.dimension()
3
```

```
>>> from sage.all import *
>>> e = RootSystem(['B',Integer(3)]).ambient_space()
>>> e.dimension()
3
```

fundamental_weight(*i*)

EXAMPLES:

```
sage: RootSystem(['B',3]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
```

```
>>> from sage.all import *
>>> RootSystem(['B',Integer(3)]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
```

negative_roots()

EXAMPLES:

```
sage: RootSystem(['B',3]).ambient_space().negative_roots()
[(-1, 1, 0),
 (-1, -1, 0),
 (-1, 0, 1),
 (-1, 0, -1),
 (0, -1, 1),
 (0, -1, -1),
 (-1, 0, 0),
 (0, -1, 0),
 (0, 0, -1)]
```

```
>>> from sage.all import *
>>> RootSystem(['B',Integer(3)]).ambient_space().negative_roots()
[(-1, 1, 0),
 (-1, -1, 0),
 (-1, 0, 1),
 (-1, 0, -1),
 (0, -1, 1),
 (0, -1, -1),
 (-1, 0, 0),
 (0, -1, 0),
 (0, 0, -1)]
```

positive_roots()

EXAMPLES:

```
sage: RootSystem(['B',3]).ambient_space().positive_roots()
[(1, -1, 0),
 (1, 1, 0),
 (1, 0, -1),
 (1, 0, 1),
 (0, 1, -1),
 (0, 1, 1),
 (1, 0, 0),
 (0, 1, 0),
 (0, 0, 1)]
```

```
>>> from sage.all import *
>>> RootSystem(['B',Integer(3)]).ambient_space().positive_roots()
[(1, -1, 0),
 (1, 1, 0),
 (1, 0, -1),
 (1, 0, 1),
 (0, 1, -1),
 (0, 1, 1),
```

(continues on next page)

(continued from previous page)

```
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)]
```

root (i, j)

Note that indexing starts at 0.

EXAMPLES:

```
sage: e = RootSystem(['B',3]).ambient_space()
sage: e.root(0,1)
(1, -1, 0)
```

```
>>> from sage.all import *
>>> e = RootSystem(['B',Integer(3)]).ambient_space()
>>> e.root(Integer(0),Integer(1))
(1, -1, 0)
```

simple_root (i)

EXAMPLES:

```
sage: e = RootSystem(['B',4]).ambient_space()
sage: e.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0,
↪ 0, 1)}
sage: e.positive_roots()
[(1, -1, 0, 0),
(1, 1, 0, 0),
(1, 0, -1, 0),
(1, 0, 1, 0),
(1, 0, 0, -1),
(1, 0, 0, 1),
(0, 1, -1, 0),
(0, 1, 1, 0),
(0, 1, 0, -1),
(0, 1, 0, 1),
(0, 0, 1, -1),
(0, 0, 1, 1),
(1, 0, 0, 0),
(0, 1, 0, 0),
(0, 0, 1, 0),
(0, 0, 0, 1)]
sage: e.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0), 4: (1/2, 1/
↪2, 1/2, 1/2)}
```

```
>>> from sage.all import *
>>> e = RootSystem(['B',Integer(4)]).ambient_space()
>>> e.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0,
↪ 0, 1)}
>>> e.positive_roots()
```

(continues on next page)

(continued from previous page)

```
[(1, -1, 0, 0),
(1, 1, 0, 0),
(1, 0, -1, 0),
(1, 0, 1, 0),
(1, 0, 0, -1),
(1, 0, 0, 1),
(0, 1, -1, 0),
(0, 1, 1, 0),
(0, 1, 0, -1),
(0, 1, 0, 1),
(0, 0, 1, -1),
(0, 0, 1, 1),
(1, 0, 0, 0),
(0, 1, 0, 0),
(0, 0, 1, 0),
(0, 0, 0, 1)]
>>> e.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0), 4: (1/2, 1/
↪2, 1/2, 1/2)}
```

class sage.combinat.root_system.type_B.**CartanType**(*n*)

Bases: *CartanType_standard_finite*, *CartanType_simple*, *CartanType_crystallographic*

EXAMPLES:

```
sage: ct = CartanType(['B',4])
sage: ct
['B', 4]
sage: ct._repr_(compact = True)
'B4'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_affine()
False
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.affine()
['B', 4, 1]
sage: ct.dual()
['C', 4]

sage: ct = CartanType(['B',1])
sage: ct.is_simply_laced()
True
sage: ct.affine()
['B', 1, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['B',Integer(4)])
>>> ct
['B', 4]
>>> ct._repr_(compact = True)
'B4'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
True
>>> ct.is_affine()
False
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.affine()
['B', 4, 1]
>>> ct.dual()
['C', 4]

>>> ct = CartanType(['B',Integer(1)])
>>> ct.is_simply_laced()
True
>>> ct.affine()
['B', 1, 1]
```

AmbientSpace

alias of *AmbientSpace*

PieriFactors

alias of *PieriFactors_type_B*

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['B',1]).ascii_art())
O
1
sage: print(CartanType(['B',2]).ascii_art())
O=>=O
1   2
sage: print(CartanType(['B',5]).ascii_art(label = lambda x: x+2))
O---O---O---O=>=O
3   4   5   6   7
```

```
>>> from sage.all import *
>>> print(CartanType(['B',Integer(1)]).ascii_art())
O
1
```

(continues on next page)

(continued from previous page)

```
>>> print(CartanType(['B',Integer(2)]).ascii_art())
O=>=O
1   2
>>> print(CartanType(['B',Integer(5)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
O---O---O---O=>=O
3   4   5   6   7
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['B',4]).coxeter_number()
8
```

```
>>> from sage.all import *
>>> CartanType(['B',Integer(4)]).coxeter_number()
8
```

dual()

Types B and C are in duality:

EXAMPLES:

```
sage: CartanType(["C", 3]).dual()
['B', 3]
```

```
>>> from sage.all import *
>>> CartanType(["C", Integer(3)]).dual()
['B', 3]
```

dual_coxeter_number()

Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['B',4]).dual_coxeter_number()
7
```

```
>>> from sage.all import *
>>> CartanType(['B',Integer(4)]).dual_coxeter_number()
7
```

dynkin_diagram()

Return a Dynkin diagram for type B.

EXAMPLES:

```
sage: b = CartanType(['B',3]).dynkin_diagram(); b                    #␣
↪needs sage.graphs
O---O=>=O
1   2   3
```

(continues on next page)

(continued from previous page)

```
B3
sage: b.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]

sage: b = CartanType(['B',1]).dynkin_diagram(); b                           #␣
↪needs sage.graphs
O
1
B1
sage: b.edges(sort=True)                                                    #␣
↪needs sage.graphs
[]
```

```
>>> from sage.all import *
>>> b = CartanType(['B',Integer(3)]).dynkin_diagram(); b                      ␣
↪    # needs sage.graphs
O---O=>=O
1   2   3
B3
>>> b.edges(sort=True)                                                     #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]

>>> b = CartanType(['B',Integer(1)]).dynkin_diagram(); b                      ␣
↪    # needs sage.graphs
O
1
B1
>>> b.edges(sort=True)                                                     #␣
↪needs sage.graphs
[]
```

5.1.247 Root system data for type BC affine

class sage.combinat.root_system.type_BC_affine.**CartanType**(*n*)

Bases: *CartanType_standard_affine*

EXAMPLES:

```
sage: ct = CartanType(['BC',4,2])
sage: ct
['BC', 4, 2]
sage: ct._repr_(compact=True)
'BC4~'
sage: ct.dynkin_diagram()                                                  #␣
↪needs sage.graphs
O=<=O---O---O=<=O
0   1   2   3   4
BC4~

sage: ct.is_irreducible()
```

(continues on next page)

(continued from previous page)

```
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['C', 4]

sage: dual = ct.dual()
sage: dual.dynkin_diagram()                                                 #␣
↪needs sage.graphs
O=>=O---O---O=>=O
0   1   2   3   4
BC4~*

sage: dual.special_node()
0
sage: dual.classical().dynkin_diagram()                                     #␣
↪needs sage.graphs
O---O---O=>=O
1   2   3   4
B4

sage: CartanType(['BC',1,2]).dynkin_diagram()                               #␣
↪needs sage.graphs
  4
O=<=O
0   1
BC1~
```

```
>>> from sage.all import *
>>> ct = CartanType(['BC',Integer(4),Integer(2)])
>>> ct
['BC', 4, 2]
>>> ct._repr_(compact=True)
'BC4~'
>>> ct.dynkin_diagram()                                                 # needs␣
↪sage.graphs
O=<=O---O---O=<=O
0   1   2   3   4
BC4~

>>> ct.is_irreducible()
True
>>> ct.is_finite()
False
>>> ct.is_affine()
True
```

(continues on next page)

(continued from previous page)

```
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.classical()
['C', 4]

>>> dual = ct.dual()
>>> dual.dynkin_diagram()                                              # needs␣
↪sage.graphs
O=>=O---O---O=>=O
0   1   2   3   4
BC4~*

>>> dual.special_node()
0
>>> dual.classical().dynkin_diagram()                                  # needs␣
↪sage.graphs
O---O---O=>=O
1   2   3   4
B4

>>> CartanType(['BC',Integer(1),Integer(2)]).dynkin_diagram()              ␣
↪          # needs sage.graphs
  4
O=<=O
0   1
BC1~
```

ascii_art(*label=None*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['BC',2,2]).ascii_art())
O=<=O=<=O
0   1   2
sage: print(CartanType(['BC',3,2]).ascii_art())
O=<=O---O=<=O
0   1   2   3
sage: print(CartanType(['BC',5,2]).ascii_art(label = lambda x: x+2))
O=<=O---O---O---O=<=O
2   3   4   5   6   7

sage: print(CartanType(['BC',1,2]).ascii_art(label = lambda x: x+2))
  4
O=<=O
2   3
```

```
>>> from sage.all import *
>>> print(CartanType(['BC',Integer(2),Integer(2)]).ascii_art())
O=<=O=<=O
```

(continues on next page)

(continued from previous page)

```
0   1   2
>>> print(CartanType(['BC',Integer(3),Integer(2)]).ascii_art())
O=<=O---O=<=O
0   1   2   3
>>> print(CartanType(['BC',Integer(5),Integer(2)]).ascii_art(label = lambda␣
↪x: x+Integer(2)))
O=<=O---O---O---O=<=O
2   3   4   5   6   7

>>> print(CartanType(['BC',Integer(1),Integer(2)]).ascii_art(label = lambda␣
↪x: x+Integer(2)))
  4
O=<=O
2   3
```

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: CartanType(['A', 2, 2]).basic_untwisted()
['A', 2]
sage: CartanType(['A', 4, 2]).basic_untwisted()
['A', 4]
sage: CartanType(['BC', 4, 2]).basic_untwisted()
['A', 8]
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(2), Integer(2)]).basic_untwisted()
['A', 2]
>>> CartanType(['A', Integer(4), Integer(2)]).basic_untwisted()
['A', 4]
>>> CartanType(['BC', Integer(4), Integer(2)]).basic_untwisted()
['A', 8]
```

classical()

Return the classical Cartan type associated with `self`.

sage: CartanType([“BC”, 3, 2]).classical() [‘C’, 3]

dynkin_diagram()

Return the extended Dynkin diagram for affine type BC.

EXAMPLES:

```
sage: c = CartanType(['BC',3,2]).dynkin_diagram(); c                    #␣
↪needs sage.graphs
O=<=O---O=<=O
0   1   2   3
BC3~
sage: c.edges(sort=True)                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

sage: c = CartanType(["A", 6, 2]).dynkin_diagram()  # should be the same as␣
↪above; did fail at some point!  # needs sage.graphs
sage: c                                                                   #␣
↪needs sage.graphs
O=<=O---O=<=O
0   1   2   3
BC3~
sage: c.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

sage: c = CartanType(['BC',2,2]).dynkin_diagram(); c                      #␣
↪needs sage.graphs
O=<=O=<=O
0   1   2
BC2~
sage: c.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 2)]

sage: c = CartanType(['BC',1,2]).dynkin_diagram(); c                      #␣
↪needs sage.graphs
  4
O=<=O
0   1
BC1~
sage: c.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(0, 1, 1), (1, 0, 4)]
```

```
>>> from sage.all import *
>>> c = CartanType(['BC',Integer(3),Integer(2)]).dynkin_diagram(); c      ␣
↪                # needs sage.graphs
O=<=O---O=<=O
0   1   2   3
BC3~
>>> c.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

>>> c = CartanType(["A", Integer(6), Integer(2)]).dynkin_diagram()  # should␣
↪be the same as above; did fail at some point!  # needs sage.graphs
>>> c                                                                   #␣
↪needs sage.graphs
O=<=O---O=<=O
0   1   2   3
BC3~
>>> c.edges(sort=True)                                                  #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

>>> c = CartanType(['BC',Integer(2),Integer(2)]).dynkin_diagram(); c
↪                # needs sage.graphs
O=<=O=<=O
0   1   2
BC2~
>>> c.edges(sort=True)                                                  #
↪needs sage.graphs
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 2)]

>>> c = CartanType(['BC',Integer(1),Integer(2)]).dynkin_diagram(); c
↪                # needs sage.graphs
  4
O=<=O
0   1
BC1~
>>> c.edges(sort=True)                                                  #
↪needs sage.graphs
[(0, 1, 1), (1, 0, 4)]
```

5.1.248 Root system data for (untwisted) type B affine

class sage.combinat.root_system.type_B_affine.**CartanType**(*n*)

Bases: *CartanType_standard_untwisted_affine*

EXAMPLES:

```
sage: ct = CartanType(['B',4,1])
sage: ct
['B', 4, 1]
sage: ct._repr_(compact = True)
'B4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['B', 4]
sage: ct.dual()
['B', 4, 1]^*
sage: ct.dual().is_untwisted_affine()
False
```

```
>>> from sage.all import *
>>> ct = CartanType(['B',Integer(4),Integer(1)])
>>> ct
['B', 4, 1]
>>> ct._repr_(compact = True)
'B4~'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
False
>>> ct.is_affine()
True
>>> ct.is_untwisted_affine()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.classical()
['B', 4]
>>> ct.dual()
['B', 4, 1]^*
>>> ct.dual().is_untwisted_affine()
False
```

PieriFactors

alias of *PieriFactors_type_B_affine*

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['B',3,1]).ascii_art())
    O 0
    |
    |
O---O=>=O
1   2   3

sage: print(CartanType(['B',5,1]).ascii_art(label = lambda x: x+2))
    O 2
    |
    |
O---O---O---O=>=O
3   4   5   6   7

sage: print(CartanType(['B',2,1]).ascii_art(label = lambda x: x+2))
O=>=O=<=O
2   4   3
sage: print(CartanType(['B',1,1]).ascii_art(label = lambda x: x+2))
O<=>O
2   3
```

```
>>> from sage.all import *
>>> print(CartanType(['B',Integer(3),Integer(1)]).ascii_art())
    O 0
    |
    |
O---O=>=O
1   2   3

>>> print(CartanType(['B',Integer(5),Integer(1)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
    O 2
    |
    |
O---O---O---O=>=O
3   4   5   6   7

>>> print(CartanType(['B',Integer(2),Integer(1)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
O=>=O=<=O
2   4   3
>>> print(CartanType(['B',Integer(1),Integer(1)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
O<=>O
2   3
```

dynkin_diagram()

Return the extended Dynkin diagram for affine type B.

EXAMPLES:

```
sage: # needs sage.graphs
sage: b = CartanType(['B',3,1]).dynkin_diagram(); b
    O 0
    |
    |
O---O=>=O
1   2   3
B3~
sage: b.edges(sort=True)
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]
sage: b = CartanType(['B',2,1]).dynkin_diagram(); b
O=>=O=<=O
0   2   1
B2~
sage: b.edges(sort=True)
[(0, 2, 2), (1, 2, 2), (2, 0, 1), (2, 1, 1)]
sage: b = CartanType(['B',1,1]).dynkin_diagram(); b
O<=>O
0   1
B1~
sage: b.edges(sort=True)
[(0, 1, 2), (1, 0, 2)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> b = CartanType(['B',Integer(3),Integer(1)]).dynkin_diagram(); b
    O 0
    |
    |
O---O=>=O
1   2   3
B3~
>>> b.edges(sort=True)
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]
>>> b = CartanType(['B',Integer(2),Integer(1)]).dynkin_diagram(); b
O=>=O=<=O
0   2   1
B2~
>>> b.edges(sort=True)
[(0, 2, 2), (1, 2, 2), (2, 0, 1), (2, 1, 1)]
>>> b = CartanType(['B',Integer(1),Integer(1)]).dynkin_diagram(); b
O<=>O
0   1
B1~
>>> b.edges(sort=True)
[(0, 1, 2), (1, 0, 2)]
```

5.1.249 Root system data for type C

class sage.combinat.root_system.type_C.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

EXAMPLES:

```
sage: e = RootSystem(['C',2]).ambient_space(); e
Ambient space of the Root system of type ['C', 2]
```

```
>>> from sage.all import *
>>> e = RootSystem(['C',Integer(2)]).ambient_space(); e
Ambient space of the Root system of type ['C', 2]
```

One cannot construct the ambient lattice because the fundamental coweights have rational coefficients:

```
sage: e.smallest_base_ring()
Rational Field

sage: RootSystem(['B',2]).ambient_space().fundamental_weights()
Finite family {1: (1, 0), 2: (1/2, 1/2)}
```

```
>>> from sage.all import *
>>> e.smallest_base_ring()
Rational Field

>>> RootSystem(['B',Integer(2)]).ambient_space().fundamental_weights()
Finite family {1: (1, 0), 2: (1/2, 1/2)}
```

dimension()

EXAMPLES:

```
sage: e = RootSystem(['C',3]).ambient_space()
sage: e.dimension()
3
```

```
>>> from sage.all import *
>>> e = RootSystem(['C',Integer(3)]).ambient_space()
>>> e.dimension()
3
```

fundamental_weight(*i*)

EXAMPLES:

```
sage: RootSystem(['C',3]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1, 1, 1)}
```

```
>>> from sage.all import *
>>> RootSystem(['C',Integer(3)]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1, 1, 1)}
```

negative_roots()

EXAMPLES:

```
sage: RootSystem(['C',3]).ambient_space().negative_roots()
[(-1, 1, 0),
 (-1, 0, 1),
 (0, -1, 1),
 (-1, -1, 0),
 (-1, 0, -1),
 (0, -1, -1),
 (-2, 0, 0),
 (0, -2, 0),
 (0, 0, -2)]
```

```
>>> from sage.all import *
>>> RootSystem(['C',Integer(3)]).ambient_space().negative_roots()
[(-1, 1, 0),
 (-1, 0, 1),
 (0, -1, 1),
 (-1, -1, 0),
 (-1, 0, -1),
 (0, -1, -1),
 (-2, 0, 0),
 (0, -2, 0),
 (0, 0, -2)]
```

positive_roots()

EXAMPLES:

```
sage: RootSystem(['C',3]).ambient_space().positive_roots()
[(1, 1, 0),
 (1, 0, 1),
 (0, 1, 1),
 (1, -1, 0),
 (1, 0, -1),
 (0, 1, -1),
 (2, 0, 0),
 (0, 2, 0),
 (0, 0, 2)]
```

```
>>> from sage.all import *
>>> RootSystem(['C',Integer(3)]).ambient_space().positive_roots()
[(1, 1, 0),
 (1, 0, 1),
 (0, 1, 1),
 (1, -1, 0),
 (1, 0, -1),
 (0, 1, -1),
 (2, 0, 0),
 (0, 2, 0),
 (0, 0, 2)]
```

root (*i*, *j*, *p1*, *p2*)

Note that indexing starts at 0.

EXAMPLES:

```
sage: e = RootSystem(['C',3]).ambient_space()
sage: e.root(0, 1, 1, 1)
(-1, -1, 0)
```

```
>>> from sage.all import *
>>> e = RootSystem(['C',Integer(3)]).ambient_space()
>>> e.root(Integer(0), Integer(1), Integer(1), Integer(1))
(-1, -1, 0)
```

simple_root (*i*)

EXAMPLES:

```
sage: RootSystem(['C',3]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 2)}
```

```
>>> from sage.all import *
>>> RootSystem(['C',Integer(3)]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 2)}
```

class sage.combinat.root_system.type_C.**CartanType** (*n*)

Bases: *CartanType_standard_finite*, *CartanType_simple*, *CartanType_crystallographic*

EXAMPLES:

```
sage: ct = CartanType(['C',4])
sage: ct
['C', 4]
sage: ct._repr_(compact = True)
'C4'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.affine()
['C', 4, 1]
sage: ct.dual()
['B', 4]

sage: ct = CartanType(['C',1])
sage: ct.is_simply_laced()
True
sage: ct.affine()
['C', 1, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['C',Integer(4)])
>>> ct
['C', 4]
>>> ct._repr_(compact = True)
'C4'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.affine()
['C', 4, 1]
>>> ct.dual()
['B', 4]

>>> ct = CartanType(['C',Integer(1)])
>>> ct.is_simply_laced()
True
>>> ct.affine()
['C', 1, 1]
```

AmbientSpace

alias of *AmbientSpace*

ascii_art(*label=None*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['C',1]).ascii_art())
O
1
sage: print(CartanType(['C',2]).ascii_art())
O=<=O
1   2
sage: print(CartanType(['C',3]).ascii_art())
O---O=<=O
1   2   3
sage: print(CartanType(['C',5]).ascii_art(label = lambda x: x+2))
O---O---O---O=<=O
3   4   5   6   7
```

```
>>> from sage.all import *
>>> print(CartanType(['C',Integer(1)]).ascii_art())
O
1
>>> print(CartanType(['C',Integer(2)]).ascii_art())
O=<=O
1   2
>>> print(CartanType(['C',Integer(3)]).ascii_art())
O---O=<=O
1   2   3
>>> print(CartanType(['C',Integer(5)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
O---O---O---O=<=O
3   4   5   6   7
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['C',4]).coxeter_number()
8
```

```
>>> from sage.all import *
>>> CartanType(['C',Integer(4)]).coxeter_number()
8
```

dual()

Types B and C are in duality:

EXAMPLES:

```
sage: CartanType(["C", 3]).dual()
['B', 3]
```

```
>>> from sage.all import *
>>> CartanType(["C", Integer(3)]).dual()
['B', 3]
```

dual_coxeter_number()

Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['C',4]).dual_coxeter_number()
5
```

```
>>> from sage.all import *
>>> CartanType(['C',Integer(4)]).dual_coxeter_number()
5
```

dynkin_diagram()

Return a Dynkin diagram for type C.

EXAMPLES:

```
sage: c = CartanType(['C',3]).dynkin_diagram(); c                          #␣
↪needs sage.graphs
O---O=<=O
1   2   3
C3
sage: c.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

 sage: b = CartanType(['C',1]).dynkin_diagram(); b                          #␣
↪needs sage.graphs
 O
 1
 C1
 sage: b.edges(sort=True)                                                   #␣
↪needs sage.graphs
 []
```

```
>>> from sage.all import *
>>> c = CartanType(['C',Integer(3)]).dynkin_diagram(); c                    ␣
↪     # needs sage.graphs
O---O=<=O
1   2   3
C3
>>> c.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

>>> b = CartanType(['C',Integer(1)]).dynkin_diagram(); b                    ␣
↪    # needs sage.graphs
 O
 1
```

(continues on next page)

(continued from previous page)

```
C1
>>> b.edges(sort=True)                                                    #␣
↪needs sage.graphs
[]
```

5.1.250 Root system data for (untwisted) type C affine

class sage.combinat.root_system.type_C_affine.**CartanType**(*n*)

Bases: *CartanType_standard_untwisted_affine*

EXAMPLES:

```
sage: ct = CartanType(['C',4,1])
sage: ct
['C', 4, 1]
sage: ct._repr_(compact = True)
'C4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['C', 4]
sage: ct.dual()
['C', 4, 1]^*
sage: ct.dual().is_untwisted_affine()
False
```

```
>>> from sage.all import *
>>> ct = CartanType(['C',Integer(4),Integer(1)])
>>> ct
['C', 4, 1]
>>> ct._repr_(compact = True)
'C4~'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
False
>>> ct.is_affine()
True
>>> ct.is_untwisted_affine()
True
```

(continues on next page)

(continued from previous page)

```
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.classical()
['C', 4]
>>> ct.dual()
['C', 4, 1]^*
>>> ct.dual().is_untwisted_affine()
False
```

PieriFactors

alias of *PieriFactors_type_C_affine*

ascii_art (*label=None*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['C',5,1]).ascii_art(label = lambda x: x+2))
O=>=O---O---O---O=<=O
2   3   4   5   6   7

sage: print(CartanType(['C',3,1]).ascii_art())
O=>=O---O=<=O
0   1   2   3

sage: print(CartanType(['C',2,1]).ascii_art())
O=>=O=<=O
0   1   2

sage: print(CartanType(['C',1,1]).ascii_art())
O<=>O
0   1
```

```
>>> from sage.all import *
>>> print(CartanType(['C',Integer(5),Integer(1)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
O=>=O---O---O---O=<=O
2   3   4   5   6   7

>>> print(CartanType(['C',Integer(3),Integer(1)]).ascii_art())
O=>=O---O=<=O
0   1   2   3

>>> print(CartanType(['C',Integer(2),Integer(1)]).ascii_art())
O=>=O=<=O
0   1   2

>>> print(CartanType(['C',Integer(1),Integer(1)]).ascii_art())
O<=>O
0   1
```

dynkin_diagram()

Return the extended Dynkin diagram for affine type C.

EXAMPLES:

```
sage: c = CartanType(['C',3,1]).dynkin_diagram(); c                          #␣
↪needs sage.graphs
O=>=O---O=<=O
0   1   2   3
C3~
sage: c.edges(sort=True)                                                     #␣
↪needs sage.graphs
[(0, 1, 2), (1, 0, 1), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]
```

```
>>> from sage.all import *
>>> c = CartanType(['C',Integer(3),Integer(1)]).dynkin_diagram(); c             ␣
↪              # needs sage.graphs
O=>=O---O=<=O
0   1   2   3
C3~
>>> c.edges(sort=True)                                                     #␣
↪needs sage.graphs
[(0, 1, 2), (1, 0, 1), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]
```

5.1.251 Root system data for type D

class sage.combinat.root_system.type_D.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

dimension()

EXAMPLES:

```
sage: e = RootSystem(['D',3]).ambient_space()
sage: e.dimension()
3
```

```
>>> from sage.all import *
>>> e = RootSystem(['D',Integer(3)]).ambient_space()
>>> e.dimension()
3
```

fundamental_weight(*i*)

EXAMPLES:

```
sage: RootSystem(['D',4]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1/2, 1/2, 1/2, -1/2), 4:␣
↪(1/2, 1/2, 1/2, 1/2)}
```

```
>>> from sage.all import *
>>> RootSystem(['D',Integer(4)]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1/2, 1/2, 1/2, -1/2), 4:␣
↪(1/2, 1/2, 1/2, 1/2)}
```

negative_roots()

EXAMPLES:

```
sage: RootSystem(['D',4]).ambient_space().negative_roots()
[(-1, 1, 0, 0),
 (-1, 0, 1, 0),
 (0, -1, 1, 0),
 (-1, 0, 0, 1),
 (0, -1, 0, 1),
 (0, 0, -1, 1),
 (-1, -1, 0, 0),
 (-1, 0, -1, 0),
 (0, -1, -1, 0),
 (-1, 0, 0, -1),
 (0, -1, 0, -1),
 (0, 0, -1, -1)]
```

```
>>> from sage.all import *
>>> RootSystem(['D',Integer(4)]).ambient_space().negative_roots()
[(-1, 1, 0, 0),
 (-1, 0, 1, 0),
 (0, -1, 1, 0),
 (-1, 0, 0, 1),
 (0, -1, 0, 1),
 (0, 0, -1, 1),
 (-1, -1, 0, 0),
 (-1, 0, -1, 0),
 (0, -1, -1, 0),
 (-1, 0, 0, -1),
 (0, -1, 0, -1),
 (0, 0, -1, -1)]
```

positive_roots()

EXAMPLES:

```
sage: RootSystem(['D',4]).ambient_space().positive_roots()
[(1, 1, 0, 0),
 (1, 0, 1, 0),
 (0, 1, 1, 0),
 (1, 0, 0, 1),
 (0, 1, 0, 1),
 (0, 0, 1, 1),
 (1, -1, 0, 0),
 (1, 0, -1, 0),
 (0, 1, -1, 0),
 (1, 0, 0, -1),
 (0, 1, 0, -1),
 (0, 0, 1, -1)]
```

```
>>> from sage.all import *
>>> RootSystem(['D',Integer(4)]).ambient_space().positive_roots()
[(1, 1, 0, 0),
 (1, 0, 1, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 1, 1, 0),
 (1, 0, 0, 1),
 (0, 1, 0, 1),
 (0, 0, 1, 1),
 (1, -1, 0, 0),
 (1, 0, -1, 0),
 (0, 1, -1, 0),
 (1, 0, 0, -1),
 (0, 1, 0, -1),
 (0, 0, 1, -1)]
```

root (*i*, *j*, *p1*, *p2*)

Note that indexing starts at 0.

EXAMPLES:

```
sage: e = RootSystem(['D',3]).ambient_space()
sage: e.root(0, 1, 1, 1)
(-1, -1, 0)
sage: e.root(0, 0, 1, 1)
(-1, 0, 0)
```

```
>>> from sage.all import *
>>> e = RootSystem(['D',Integer(3)]).ambient_space()
>>> e.root(Integer(0), Integer(1), Integer(1), Integer(1))
(-1, -1, 0)
>>> e.root(Integer(0), Integer(0), Integer(1), Integer(1))
(-1, 0, 0)
```

simple_root (*i*)

EXAMPLES:

```
sage: RootSystem(['D',4]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0,
↪ 1, 1)}
```

```
>>> from sage.all import *
>>> RootSystem(['D',Integer(4)]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0,
↪ 1, 1)}
```

class sage.combinat.root_system.type_D.**CartanType** (*n*)

Bases: *CartanType_standard_finite*, *CartanType_simply_laced*

EXAMPLES:

```
sage: ct = CartanType(['D',4])
sage: ct
['D', 4]
sage: ct._repr_(compact = True)
'D4'
```

(continues on next page)

(continued from previous page)

```
sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.dual()
['D', 4]
sage: ct.affine()
['D', 4, 1]

sage: ct = CartanType(['D',2])
sage: ct.is_irreducible()
False
sage: ct.dual()
['D', 2]
sage: ct.affine()
Traceback (most recent call last):
...
ValueError: ['D', 2, 1] is not a valid Cartan type
```

```
>>> from sage.all import *
>>> ct = CartanType(['D',Integer(4)])
>>> ct
['D', 4]
>>> ct._repr_(compact = True)
'D4'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
True
>>> ct.dual()
['D', 4]
>>> ct.affine()
['D', 4, 1]

>>> ct = CartanType(['D',Integer(2)])
>>> ct.is_irreducible()
False
>>> ct.dual()
['D', 2]
>>> ct.affine()
Traceback (most recent call last):
...
ValueError: ['D', 2, 1] is not a valid Cartan type
```

AmbientSpace

alias of *AmbientSpace*

ascii_art(*label=None*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['D',3]).ascii_art())
O 3
|
|
O---O
1   2
sage: print(CartanType(['D',4]).ascii_art())
    O 4
    |
    |
O---O---O
1   2   3
sage: print(CartanType(['D',4]).ascii_art(label = lambda x: x+2))
    O 6
    |
    |
O---O---O
3   4   5
sage: print(CartanType(['D',6]).ascii_art(label = lambda x: x+2))
            O 8
            |
            |
O---O---O---O---O
3   4   5   6   7
```

```
>>> from sage.all import *
>>> print(CartanType(['D',Integer(3)]).ascii_art())
O 3
|
|
O---O
1   2
>>> print(CartanType(['D',Integer(4)]).ascii_art())
    O 4
    |
    |
O---O---O
1   2   3
>>> print(CartanType(['D',Integer(4)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
    O 6
    |
    |
O---O---O
3   4   5
>>> print(CartanType(['D',Integer(6)]).ascii_art(label = lambda x:␣
```

(continues on next page)

(continued from previous page)

```
↪x+Integer(2)))
            O 8
            |
            |
O---O---O---O---O
3   4   5   6   7
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['D',4]).coxeter_number()
6
```

```
>>> from sage.all import *
>>> CartanType(['D',Integer(4)]).coxeter_number()
6
```

dual_coxeter_number()

Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['D',4]).dual_coxeter_number()
6
```

```
>>> from sage.all import *
>>> CartanType(['D',Integer(4)]).dual_coxeter_number()
6
```

dynkin_diagram()

Return a Dynkin diagram for type D.

EXAMPLES:

```
sage: d = CartanType(['D',5]).dynkin_diagram(); d                          #␣
↪needs sage.graphs
        O 5
        |
        |
O---O---O---O
1   2   3   4
D5
sage: d.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 1),
 (3, 4, 1), (3, 5, 1), (4, 3, 1), (5, 3, 1)]

sage: d = CartanType(['D',4]).dynkin_diagram(); d                          #␣
↪needs sage.graphs
    O 4
    |
```

(continues on next page)

(continued from previous page)

```
    |
O---O---O
1   2   3
D4
sage: d.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (2, 4, 1), (3, 2, 1), (4, 2, 1)]

sage: d = CartanType(['D',3]).dynkin_diagram(); d                           #␣
↪needs sage.graphs
O 3
|
|
O---O
1   2
D3
sage: d.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (1, 3, 1), (2, 1, 1), (3, 1, 1)]


sage: d = CartanType(['D',2]).dynkin_diagram(); d                           #␣
↪needs sage.graphs
O   O
1   2
D2
sage: d.edges(sort=True)                                                    #␣
↪needs sage.graphs
[]
```

```
>>> from sage.all import *
>>> d = CartanType(['D',Integer(5)]).dynkin_diagram(); d                    ␣
↪      # needs sage.graphs
        O 5
        |
        |
O---O---O---O
1   2   3   4
D5
>>> d.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 1),
 (3, 4, 1), (3, 5, 1), (4, 3, 1), (5, 3, 1)]

>>> d = CartanType(['D',Integer(4)]).dynkin_diagram(); d                    ␣
↪      # needs sage.graphs
    O 4
    |
    |
O---O---O
1   2   3
D4
```

(continues on next page)

(continued from previous page)

```
>>> d.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (2, 4, 1), (3, 2, 1), (4, 2, 1)]

>>> d = CartanType(['D',Integer(3)]).dynkin_diagram(); d                  ␣
↪     # needs sage.graphs
O 3
|
|
O---O
1   2
D3
>>> d.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (1, 3, 1), (2, 1, 1), (3, 1, 1)]


>>> d = CartanType(['D',Integer(2)]).dynkin_diagram(); d                  ␣
↪     # needs sage.graphs
O   O
1   2
D2
>>> d.edges(sort=True)                                                    #␣
↪needs sage.graphs
[]
```

is_atomic()

Implement `CartanType_abstract.is_atomic()`.

D_2 is atomic, like all D_n, despite being non irreducible.

EXAMPLES:

```
sage: CartanType(["D",2]).is_atomic()
True
sage: CartanType(["D",2]).is_irreducible()
False
```

```
>>> from sage.all import *
>>> CartanType(["D",Integer(2)]).is_atomic()
True
>>> CartanType(["D",Integer(2)]).is_irreducible()
False
```

5.1.252 Root system data for (untwisted) type D affine

class sage.combinat.root_system.type_D_affine.**CartanType**(*n*)

Bases: *CartanType_standard_untwisted_affine*, *CartanType_simply_laced*

EXAMPLES:

```
sage: ct = CartanType(['D',4,1])
sage: ct
```

(continues on next page)

(continued from previous page)

```
['D', 4, 1]
sage: ct._repr_(compact = True)
'D4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.classical()
['D', 4]
sage: ct.dual()
['D', 4, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['D',Integer(4),Integer(1)])
>>> ct
['D', 4, 1]
>>> ct._repr_(compact = True)
'D4~'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
False
>>> ct.is_affine()
True
>>> ct.is_untwisted_affine()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
True
>>> ct.classical()
['D', 4]
>>> ct.dual()
['D', 4, 1]
```

PieriFactors

alias of `PieriFactors_type_D_affine`

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

dynkin_diagram()

Return the extended Dynkin diagram for affine type D.

EXAMPLES:

```
sage: d = CartanType(['D', 6, 1]).dynkin_diagram(); d
↪# needs sage.graphs
   0 O       O 6
     |       |
     |       |
 O---O---O---O---O
 1   2   3   4   5
 D6~
sage: d.edges(sort=True)
↪# needs sage.graphs
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 1), (2, 3, 1),
 (3, 2, 1), (3, 4, 1), (4, 3, 1), (4, 5, 1), (4, 6, 1), (5, 4, 1), (6, 4, 1)]

sage: d = CartanType(['D', 4, 1]).dynkin_diagram(); d
↪# needs sage.graphs
    O 4
    |
    |
O---O---O
1   |2  3
    |
    O 0
D4~
sage: d.edges(sort=True)
↪# needs sage.graphs
[(0, 2, 1),
 (1, 2, 1),
 (2, 0, 1),
 (2, 1, 1),
 (2, 3, 1),
 (2, 4, 1),
 (3, 2, 1),
 (4, 2, 1)]

sage: d = CartanType(['D', 3, 1]).dynkin_diagram(); d
↪# needs sage.graphs
0
O-------+
|       |
|       |
O---O---O
3   1   2
D3~
sage: d.edges(sort=True)
↪# needs sage.graphs
[(0, 2, 1), (0, 3, 1), (1, 2, 1), (1, 3, 1),
 (2, 0, 1), (2, 1, 1), (3, 0, 1), (3, 1, 1)]
```

```
>>> from sage.all import *
>>> d = CartanType(['D', Integer(6), Integer(1)]).dynkin_diagram(); d
↪               # needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
  O O       O 6
    |       |
    |       |
 O---O---O---O---O
 1   2   3   4   5
 D6~
>>> d.edges(sort=True)                                                      #␣
↪needs sage.graphs
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 1), (2, 3, 1),
 (3, 2, 1), (3, 4, 1), (4, 3, 1), (4, 5, 1), (4, 6, 1), (5, 4, 1), (6, 4, 1)]

>>> d = CartanType(['D', Integer(4), Integer(1)]).dynkin_diagram(); d        ␣
↪               # needs sage.graphs
    O 4
    |
    |
O---O---O
1   |2  3
    |
    O 0
D4~
>>> d.edges(sort=True)                                                      #␣
↪needs sage.graphs
[(0, 2, 1),
 (1, 2, 1),
 (2, 0, 1),
 (2, 1, 1),
 (2, 3, 1),
 (2, 4, 1),
 (3, 2, 1),
 (4, 2, 1)]

>>> d = CartanType(['D', Integer(3), Integer(1)]).dynkin_diagram(); d        ␣
↪               # needs sage.graphs
0
O-------+
|       |
|       |
O---O---O
3   1   2
D3~
>>> d.edges(sort=True)                                                      #␣
↪needs sage.graphs
[(0, 2, 1), (0, 3, 1), (1, 2, 1), (1, 3, 1),
 (2, 0, 1), (2, 1, 1), (3, 0, 1), (3, 1, 1)]
```

5.1.253 Root system data for type E

class sage.combinat.root_system.type_E.**AmbientSpace**(*root_system*, *baseRing*)

Bases: *AmbientSpace*

The lattice behind E6, E7, or E8. The computations are based on Bourbaki, Groupes et Algèbres de Lie, Ch. 4,5,6 (planche V-VII).

dimension()

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: e.dimension()
8
```

```
>>> from sage.all import *
>>> e = RootSystem(['E',Integer(6)]).ambient_space()
>>> e.dimension()
8
```

fundamental_weights()

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: e.fundamental_weights()
Finite family {1: (0, 0, 0, 0, 0, -2/3, -2/3, 2/3), 2: (1/2, 1/2, 1/2, 1/2, 1/
↪2, -1/2, -1/2, 1/2), 3: (-1/2, 1/2, 1/2, 1/2, 1/2, -5/6, -5/6, 5/6), 4: (0,␣
↪0, 1, 1, 1, -1, -1, 1), 5: (0, 0, 0, 1, 1, -2/3, -2/3, 2/3), 6: (0, 0, 0, 0,
↪ 1, -1/3, -1/3, 1/3)}
```

```
>>> from sage.all import *
>>> e = RootSystem(['E',Integer(6)]).ambient_space()
>>> e.fundamental_weights()
Finite family {1: (0, 0, 0, 0, 0, -2/3, -2/3, 2/3), 2: (1/2, 1/2, 1/2, 1/2, 1/
↪2, -1/2, -1/2, 1/2), 3: (-1/2, 1/2, 1/2, 1/2, 1/2, -5/6, -5/6, 5/6), 4: (0,␣
↪0, 1, 1, 1, -1, -1, 1), 5: (0, 0, 0, 1, 1, -2/3, -2/3, 2/3), 6: (0, 0, 0, 0,
↪ 1, -1/3, -1/3, 1/3)}
```

negative_roots()

The negative roots.

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: e.negative_roots()
[(-1, -1, 0, 0, 0, 0, 0, 0),
 (-1, 0, -1, 0, 0, 0, 0, 0),
 (-1, 0, 0, -1, 0, 0, 0, 0),
 (-1, 0, 0, 0, -1, 0, 0, 0),
 (0, -1, -1, 0, 0, 0, 0, 0),
 (0, -1, 0, -1, 0, 0, 0, 0),
 (0, -1, 0, 0, -1, 0, 0, 0),
 (0, 0, -1, -1, 0, 0, 0, 0),
 (0, 0, -1, 0, -1, 0, 0, 0),
 (0, 0, 0, -1, -1, 0, 0, 0),
 (1, -1, 0, 0, 0, 0, 0, 0),
 (1, 0, -1, 0, 0, 0, 0, 0),
 (1, 0, 0, -1, 0, 0, 0, 0),
 (1, 0, 0, 0, -1, 0, 0, 0),
 (0, 1, -1, 0, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 1, 0, -1, 0, 0, 0, 0),
 (0, 1, 0, 0, -1, 0, 0, 0),
 (0, 0, 1, -1, 0, 0, 0, 0),
 (0, 0, 1, 0, -1, 0, 0, 0),
 (0, 0, 0, 1, -1, 0, 0, 0),
 (-1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['E',Integer(6)]).ambient_space()
>>> e.negative_roots()
[(-1, -1, 0, 0, 0, 0, 0, 0),
 (-1, 0, -1, 0, 0, 0, 0, 0),
 (-1, 0, 0, -1, 0, 0, 0, 0),
 (-1, 0, 0, 0, -1, 0, 0, 0),
 (0, -1, -1, 0, 0, 0, 0, 0),
 (0, -1, 0, -1, 0, 0, 0, 0),
 (0, -1, 0, 0, -1, 0, 0, 0),
 (0, 0, -1, -1, 0, 0, 0, 0),
 (0, 0, -1, 0, -1, 0, 0, 0),
 (0, 0, 0, -1, -1, 0, 0, 0),
 (1, -1, 0, 0, 0, 0, 0, 0),
 (1, 0, -1, 0, 0, 0, 0, 0),
 (1, 0, 0, -1, 0, 0, 0, 0),
 (1, 0, 0, 0, -1, 0, 0, 0),
 (0, 1, -1, 0, 0, 0, 0, 0),
 (0, 1, 0, -1, 0, 0, 0, 0),
 (0, 1, 0, 0, -1, 0, 0, 0),
 (0, 0, 1, -1, 0, 0, 0, 0),
 (0, 0, 1, 0, -1, 0, 0, 0),
 (0, 0, 0, 1, -1, 0, 0, 0),
 (-1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
```

(continues on next page)

(continued from previous page)

```
(-1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
(1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
(1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
(1/2, 1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
(1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
(1/2, 1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
(1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2)]
```

positive_roots()

These are the roots positive w.r.t. to lexicographic ordering of the basis elements (e1<…<e4).

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: e.positive_roots()
[(1, 1, 0, 0, 0, 0, 0, 0),
 (1, 0, 1, 0, 0, 0, 0, 0),
 (1, 0, 0, 1, 0, 0, 0, 0),
 (1, 0, 0, 0, 1, 0, 0, 0),
 (0, 1, 1, 0, 0, 0, 0, 0),
 (0, 1, 0, 1, 0, 0, 0, 0),
 (0, 1, 0, 0, 1, 0, 0, 0),
 (0, 0, 1, 1, 0, 0, 0, 0),
 (0, 0, 1, 0, 1, 0, 0, 0),
 (0, 0, 0, 1, 1, 0, 0, 0),
 (-1, 1, 0, 0, 0, 0, 0, 0),
 (-1, 0, 1, 0, 0, 0, 0, 0),
 (-1, 0, 0, 1, 0, 0, 0, 0),
 (-1, 0, 0, 0, 1, 0, 0, 0),
 (0, -1, 1, 0, 0, 0, 0, 0),
 (0, -1, 0, 1, 0, 0, 0, 0),
 (0, -1, 0, 0, 1, 0, 0, 0),
 (0, 0, -1, 1, 0, 0, 0, 0),
 (0, 0, -1, 0, 1, 0, 0, 0),
 (0, 0, 0, -1, 1, 0, 0, 0),
 (1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
```

(continues on next page)

(continued from previous page)

```
 (-1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2)]
sage: e.rho()
(0, 1, 2, 3, 4, -4, -4, 4)
sage: E8 = RootSystem(['E',8])
sage: e = E8.ambient_space()
sage: e.negative_roots()
[(-1, -1, 0, 0, 0, 0, 0, 0),
 (-1, 0, -1, 0, 0, 0, 0, 0),
 (-1, 0, 0, -1, 0, 0, 0, 0),
 (-1, 0, 0, 0, -1, 0, 0, 0),
 (-1, 0, 0, 0, 0, -1, 0, 0),
 (-1, 0, 0, 0, 0, 0, -1, 0),
 (-1, 0, 0, 0, 0, 0, 0, -1),
 (0, -1, -1, 0, 0, 0, 0, 0),
 (0, -1, 0, -1, 0, 0, 0, 0),
 (0, -1, 0, 0, -1, 0, 0, 0),
 (0, -1, 0, 0, 0, -1, 0, 0),
 (0, -1, 0, 0, 0, 0, -1, 0),
 (0, -1, 0, 0, 0, 0, 0, -1),
 (0, 0, -1, -1, 0, 0, 0, 0),
 (0, 0, -1, 0, -1, 0, 0, 0),
 (0, 0, -1, 0, 0, -1, 0, 0),
 (0, 0, -1, 0, 0, 0, -1, 0),
 (0, 0, -1, 0, 0, 0, 0, -1),
 (0, 0, 0, -1, -1, 0, 0, 0),
 (0, 0, 0, -1, 0, -1, 0, 0),
 (0, 0, 0, -1, 0, 0, -1, 0),
 (0, 0, 0, -1, 0, 0, 0, -1),
 (0, 0, 0, 0, -1, -1, 0, 0),
 (0, 0, 0, 0, -1, 0, -1, 0),
 (0, 0, 0, 0, -1, 0, 0, -1),
 (0, 0, 0, 0, 0, -1, -1, 0),
 (0, 0, 0, 0, 0, -1, 0, -1),
 (0, 0, 0, 0, 0, 0, -1, -1),
 (1, -1, 0, 0, 0, 0, 0, 0),
 (1, 0, -1, 0, 0, 0, 0, 0),
 (1, 0, 0, -1, 0, 0, 0, 0),
 (1, 0, 0, 0, -1, 0, 0, 0),
 (1, 0, 0, 0, 0, -1, 0, 0),
 (1, 0, 0, 0, 0, 0, -1, 0),
 (1, 0, 0, 0, 0, 0, 0, -1),
 (0, 1, -1, 0, 0, 0, 0, 0),
 (0, 1, 0, -1, 0, 0, 0, 0),
 (0, 1, 0, 0, -1, 0, 0, 0),
 (0, 1, 0, 0, 0, -1, 0, 0),
 (0, 1, 0, 0, 0, 0, -1, 0),
 (0, 1, 0, 0, 0, 0, 0, -1),
 (0, 0, 1, -1, 0, 0, 0, 0),
 (0, 0, 1, 0, -1, 0, 0, 0),
 (0, 0, 1, 0, 0, -1, 0, 0),
 (0, 0, 1, 0, 0, 0, -1, 0),
 (0, 0, 1, 0, 0, 0, 0, -1),
```

(continues on next page)

(continued from previous page)

```
(0, 0, 0, 1, -1, 0, 0, 0),
(0, 0, 0, 1, 0, -1, 0, 0),
(0, 0, 0, 1, 0, 0, -1, 0),
(0, 0, 0, 1, 0, 0, 0, -1),
(0, 0, 0, 0, 1, -1, 0, 0),
(0, 0, 0, 0, 1, 0, -1, 0),
(0, 0, 0, 0, 1, 0, 0, -1),
(0, 0, 0, 0, 0, 1, -1, 0),
(0, 0, 0, 0, 0, 1, 0, -1),
(0, 0, 0, 0, 0, 0, 1, -1),
(-1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
(1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
(1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
(1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
(1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
(1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
```

(continues on next page)

(continued from previous page)

```
 (1/2, -1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2)]
sage: e.rho()
(0, 1, 2, 3, 4, 5, 6, 23)
```

```
>>> from sage.all import *
>>> e = RootSystem(['E',Integer(6)]).ambient_space()
>>> e.positive_roots()
[(1, 1, 0, 0, 0, 0, 0, 0),
 (1, 0, 1, 0, 0, 0, 0, 0),
 (1, 0, 0, 1, 0, 0, 0, 0),
 (1, 0, 0, 0, 1, 0, 0, 0),
 (0, 1, 1, 0, 0, 0, 0, 0),
 (0, 1, 0, 1, 0, 0, 0, 0),
 (0, 1, 0, 0, 1, 0, 0, 0),
 (0, 0, 1, 1, 0, 0, 0, 0),
 (0, 0, 1, 0, 1, 0, 0, 0),
 (0, 0, 0, 1, 1, 0, 0, 0),
 (-1, 1, 0, 0, 0, 0, 0, 0),
 (-1, 0, 1, 0, 0, 0, 0, 0),
 (-1, 0, 0, 1, 0, 0, 0, 0),
 (-1, 0, 0, 0, 1, 0, 0, 0),
 (0, -1, 1, 0, 0, 0, 0, 0),
 (0, -1, 0, 1, 0, 0, 0, 0),
 (0, -1, 0, 0, 1, 0, 0, 0),
 (0, 0, -1, 1, 0, 0, 0, 0),
 (0, 0, -1, 0, 1, 0, 0, 0),
 (0, 0, 0, -1, 1, 0, 0, 0),
 (1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
```

(continues on next page)

(continued from previous page)

```
 (1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2)]
>>> e.rho()
(0, 1, 2, 3, 4, -4, -4, 4)
>>> E8 = RootSystem(['E',Integer(8)])
>>> e = E8.ambient_space()
>>> e.negative_roots()
[(-1, -1, 0, 0, 0, 0, 0, 0),
 (-1, 0, -1, 0, 0, 0, 0, 0),
 (-1, 0, 0, -1, 0, 0, 0, 0),
 (-1, 0, 0, 0, -1, 0, 0, 0),
 (-1, 0, 0, 0, 0, -1, 0, 0),
 (-1, 0, 0, 0, 0, 0, -1, 0),
 (-1, 0, 0, 0, 0, 0, 0, -1),
 (0, -1, -1, 0, 0, 0, 0, 0),
 (0, -1, 0, -1, 0, 0, 0, 0),
 (0, -1, 0, 0, -1, 0, 0, 0),
 (0, -1, 0, 0, 0, -1, 0, 0),
 (0, -1, 0, 0, 0, 0, -1, 0),
 (0, -1, 0, 0, 0, 0, 0, -1),
 (0, 0, -1, -1, 0, 0, 0, 0),
 (0, 0, -1, 0, -1, 0, 0, 0),
 (0, 0, -1, 0, 0, -1, 0, 0),
 (0, 0, -1, 0, 0, 0, -1, 0),
 (0, 0, -1, 0, 0, 0, 0, -1),
 (0, 0, 0, -1, -1, 0, 0, 0),
 (0, 0, 0, -1, 0, -1, 0, 0),
 (0, 0, 0, -1, 0, 0, -1, 0),
 (0, 0, 0, -1, 0, 0, 0, -1),
 (0, 0, 0, 0, -1, -1, 0, 0),
 (0, 0, 0, 0, -1, 0, -1, 0),
 (0, 0, 0, 0, -1, 0, 0, -1),
 (0, 0, 0, 0, 0, -1, -1, 0),
 (0, 0, 0, 0, 0, -1, 0, -1),
 (0, 0, 0, 0, 0, 0, -1, -1),
 (1, -1, 0, 0, 0, 0, 0, 0),
 (1, 0, -1, 0, 0, 0, 0, 0),
 (1, 0, 0, -1, 0, 0, 0, 0),
 (1, 0, 0, 0, -1, 0, 0, 0),
 (1, 0, 0, 0, 0, -1, 0, 0),
 (1, 0, 0, 0, 0, 0, -1, 0),
 (1, 0, 0, 0, 0, 0, 0, -1),
```

(continues on next page)

(continued from previous page)

```
(0, 1, -1, 0, 0, 0, 0, 0),
(0, 1, 0, -1, 0, 0, 0, 0),
(0, 1, 0, 0, -1, 0, 0, 0),
(0, 1, 0, 0, 0, -1, 0, 0),
(0, 1, 0, 0, 0, 0, -1, 0),
(0, 1, 0, 0, 0, 0, 0, -1),
(0, 0, 1, -1, 0, 0, 0, 0),
(0, 0, 1, 0, -1, 0, 0, 0),
(0, 0, 1, 0, 0, -1, 0, 0),
(0, 0, 1, 0, 0, 0, -1, 0),
(0, 0, 1, 0, 0, 0, 0, -1),
(0, 0, 0, 1, -1, 0, 0, 0),
(0, 0, 0, 1, 0, -1, 0, 0),
(0, 0, 0, 1, 0, 0, -1, 0),
(0, 0, 0, 1, 0, 0, 0, -1),
(0, 0, 0, 0, 1, -1, 0, 0),
(0, 0, 0, 0, 1, 0, -1, 0),
(0, 0, 0, 0, 1, 0, 0, -1),
(0, 0, 0, 0, 0, 1, -1, 0),
(0, 0, 0, 0, 0, 1, 0, -1),
(0, 0, 0, 0, 0, 0, 1, -1),
(-1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
```

(continues on next page)

(continued from previous page)

```
 (-1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2)]
>>> e.rho()
(0, 1, 2, 3, 4, 5, 6, 23)
```

root (*i1*, *i2=None*, *i3=None*, *i4=None*, *i5=None*, *i6=None*, *i7=None*, *i8=None*, *p1=0*, *p2=0*, *p3=0*, *p4=0*, *p5=0*, *p6=0*, *p7=0*, *p8=0*)

Compute an element of the underlying lattice, using the specified elements of the standard basis, with signs dictated by the corresponding 'pi' arguments. We rely on the caller to provide the correct arguments. This is typically used to generate roots, although the generated elements need not be roots themselves. We assume that if one of the indices is not given, the rest are not as well. This should work for E6, E7, E8.

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: [ e.root(i, j, p3=1) for i in range(e.n) for j in range(i+1, e.n) ]
[(1, 1, 0, 0, 0, 0, 0, 0),
 (1, 0, 1, 0, 0, 0, 0, 0),
 (1, 0, 0, 1, 0, 0, 0, 0),
 (1, 0, 0, 0, 1, 0, 0, 0),
 (1, 0, 0, 0, 0, 1, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 (1, 0, 0, 0, 0, 0, 1, 0),
 (1, 0, 0, 0, 0, 0, 0, 1),
 (0, 1, 1, 0, 0, 0, 0, 0),
 (0, 1, 0, 1, 0, 0, 0, 0),
 (0, 1, 0, 0, 1, 0, 0, 0),
 (0, 1, 0, 0, 0, 1, 0, 0),
 (0, 1, 0, 0, 0, 0, 1, 0),
 (0, 1, 0, 0, 0, 0, 0, 1),
 (0, 0, 1, 1, 0, 0, 0, 0),
 (0, 0, 1, 0, 1, 0, 0, 0),
 (0, 0, 1, 0, 0, 1, 0, 0),
 (0, 0, 1, 0, 0, 0, 1, 0),
 (0, 0, 1, 0, 0, 0, 0, 1),
 (0, 0, 0, 1, 1, 0, 0, 0),
 (0, 0, 0, 1, 0, 1, 0, 0),
 (0, 0, 0, 1, 0, 0, 1, 0),
 (0, 0, 0, 1, 0, 0, 0, 1),
 (0, 0, 0, 0, 1, 1, 0, 0),
 (0, 0, 0, 0, 1, 0, 1, 0),
 (0, 0, 0, 0, 1, 0, 0, 1),
 (0, 0, 0, 0, 0, 1, 1, 0),
 (0, 0, 0, 0, 0, 1, 0, 1),
 (0, 0, 0, 0, 0, 0, 1, 1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['E',Integer(6)]).ambient_space()
>>> [ e.root(i, j, p3=Integer(1)) for i in range(e.n) for j in␣
↪range(i+Integer(1), e.n) ]
[(1, 1, 0, 0, 0, 0, 0, 0),
 (1, 0, 1, 0, 0, 0, 0, 0),
 (1, 0, 0, 1, 0, 0, 0, 0),
 (1, 0, 0, 0, 1, 0, 0, 0),
 (1, 0, 0, 0, 0, 1, 0, 0),
 (1, 0, 0, 0, 0, 0, 1, 0),
 (1, 0, 0, 0, 0, 0, 0, 1),
 (0, 1, 1, 0, 0, 0, 0, 0),
 (0, 1, 0, 1, 0, 0, 0, 0),
 (0, 1, 0, 0, 1, 0, 0, 0),
 (0, 1, 0, 0, 0, 1, 0, 0),
 (0, 1, 0, 0, 0, 0, 1, 0),
 (0, 1, 0, 0, 0, 0, 0, 1),
 (0, 0, 1, 1, 0, 0, 0, 0),
 (0, 0, 1, 0, 1, 0, 0, 0),
 (0, 0, 1, 0, 0, 1, 0, 0),
 (0, 0, 1, 0, 0, 0, 1, 0),
 (0, 0, 1, 0, 0, 0, 0, 1),
 (0, 0, 0, 1, 1, 0, 0, 0),
 (0, 0, 0, 1, 0, 1, 0, 0),
 (0, 0, 0, 1, 0, 0, 1, 0),
 (0, 0, 0, 1, 0, 0, 0, 1),
 (0, 0, 0, 0, 1, 1, 0, 0),
 (0, 0, 0, 0, 1, 0, 1, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 0, 0, 0, 1, 0, 0, 1),
 (0, 0, 0, 0, 0, 1, 1, 0),
 (0, 0, 0, 0, 0, 1, 0, 1),
 (0, 0, 0, 0, 0, 0, 1, 1)]
```

simple_root(*i*)

There are computed as what Bourbaki calls the Base:
a1 = e2-e3, a2 = e3-e4, a3 = e4, a4 = 1/2*(e1-e2-e3-e4)

EXAMPLES:

```
sage: LE6 = RootSystem(['E',6]).ambient_space()
sage: LE6.simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2), 2: (1, 1, 0,
↪ 0, 0, 0, 0, 0), 3: (-1, 1, 0, 0, 0, 0, 0, 0), 4: (0, -1, 1, 0, 0, 0, 0, 0),
↪ 5: (0, 0, -1, 1, 0, 0, 0, 0), 6: (0, 0, 0, -1, 1, 0, 0, 0)}
```

```
>>> from sage.all import *
>>> LE6 = RootSystem(['E',Integer(6)]).ambient_space()
>>> LE6.simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2), 2: (1, 1, 0,
↪ 0, 0, 0, 0, 0), 3: (-1, 1, 0, 0, 0, 0, 0, 0), 4: (0, -1, 1, 0, 0, 0, 0, 0),
↪ 5: (0, 0, -1, 1, 0, 0, 0, 0), 6: (0, 0, 0, -1, 1, 0, 0, 0)}
```

class sage.combinat.root_system.type_E.**CartanType**(*n*)

Bases: *CartanType_standard_finite*, *CartanType_simple*, *CartanType_simply_laced*

EXAMPLES:

```
sage: ct = CartanType(['E',6])
sage: ct
['E', 6]
sage: ct._repr_(compact = True)
'E6'
sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_affine()
False
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.affine()
['E', 6, 1]
sage: ct.dual()
['E', 6]
```

```
>>> from sage.all import *
>>> ct = CartanType(['E',Integer(6)])
>>> ct
```

(continues on next page)

(continued from previous page)

```
['E', 6]
>>> ct._repr_(compact = True)
'E6'
>>> ct.is_irreducible()
True
>>> ct.is_finite()
True
>>> ct.is_affine()
False
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
True
>>> ct.affine()
['E', 6, 1]
>>> ct.dual()
['E', 6]
```

AmbientSpace

alias of *AmbientSpace*

ascii_art(*label=None*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['E',6]).ascii_art(label = lambda x: x+2))
        O 4
        |
        |
O---O---O---O---O
3   5   6   7   8
sage: print(CartanType(['E',7]).ascii_art(label = lambda x: x+2))
        O 4
        |
        |
O---O---O---O---O---O
3   5   6   7   8   9
sage: print(CartanType(['E',8]).ascii_art(label = lambda x: x+1))
        O 3
        |
        |
O---O---O---O---O---O---O
2   4   5   6   7   8   9
```

```
>>> from sage.all import *
>>> print(CartanType(['E',Integer(6)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
        O 4
        |
        |
O---O---O---O---O
```

(continues on next page)

(continued from previous page)

```
3   5   6   7   8
>>> print(CartanType(['E',Integer(7)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
        O 4
        |
        |
O---O---O---O---O---O
3   5   6   7   8   9
>>> print(CartanType(['E',Integer(8)]).ascii_art(label = lambda x:␣
↪x+Integer(1)))
        O 3
        |
        |
O---O---O---O---O---O---O
2   4   5   6   7   8   9
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['E',6]).coxeter_number()
12
sage: CartanType(['E',7]).coxeter_number()
18
sage: CartanType(['E',8]).coxeter_number()
30
```

```
>>> from sage.all import *
>>> CartanType(['E',Integer(6)]).coxeter_number()
12
>>> CartanType(['E',Integer(7)]).coxeter_number()
18
>>> CartanType(['E',Integer(8)]).coxeter_number()
30
```

dual_coxeter_number()

Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['E',6]).dual_coxeter_number()
12
sage: CartanType(['E',7]).dual_coxeter_number()
18
sage: CartanType(['E',8]).dual_coxeter_number()
30
```

```
>>> from sage.all import *
>>> CartanType(['E',Integer(6)]).dual_coxeter_number()
12
>>> CartanType(['E',Integer(7)]).dual_coxeter_number()
```

(continues on next page)

(continued from previous page)

```
18
>>> CartanType(['E',Integer(8)]).dual_coxeter_number()
30
```

dynkin_diagram()

Return a Dynkin diagram for type E.

EXAMPLES:

```
sage: # needs sage.graphs
sage: e = CartanType(['E',6]).dynkin_diagram(); e
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
sage: e.edges(sort=True)
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1),
 (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1), (6, 5, 1)]
sage: e = CartanType(['E',7]).dynkin_diagram(); e
        O 2
        |
        |
O---O---O---O---O---O
1   3   4   5   6   7
E7
sage: e.edges(sort=True)
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1),
 (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1), (6, 5, 1),
 (6, 7, 1), (7, 6, 1)]
sage: e = CartanType(['E',8]).dynkin_diagram(); e
        O 2
        |
        |
O---O---O---O---O---O---O
1   3   4   5   6   7   8
E8
sage: e.edges(sort=True)
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1),
 (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1), (6, 5, 1),
 (6, 7, 1), (7, 6, 1), (7, 8, 1), (8, 7, 1)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> e = CartanType(['E',Integer(6)]).dynkin_diagram(); e
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
```

(continues on next page)

(continued from previous page)

```
>>> e.edges(sort=True)
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1),
 (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1), (6, 5, 1)]
>>> e = CartanType(['E',Integer(7)]).dynkin_diagram(); e
        O 2
        |
        |
O---O---O---O---O---O
1   3   4   5   6   7
E7
>>> e.edges(sort=True)
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1),
 (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1), (6, 5, 1),
 (6, 7, 1), (7, 6, 1)]
>>> e = CartanType(['E',Integer(8)]).dynkin_diagram(); e
        O 2
        |
        |
O---O---O---O---O---O---O
1   3   4   5   6   7   8
E8
>>> e.edges(sort=True)
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1),
 (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1), (6, 5, 1),
 (6, 7, 1), (7, 6, 1), (7, 8, 1), (8, 7, 1)]
```

5.1.254 Root system data for (untwisted) type E affine

class sage.combinat.root_system.type_E_affine.**CartanType**(*n*)

Bases: *CartanType_standard_untwisted_affine*, *CartanType_simply_laced*

EXAMPLES:

```
sage: ct = CartanType(['E',6,1])
sage: ct
['E', 6, 1]
sage: ct._repr_(compact = True)
'E6~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.classical()
```

(continues on next page)

(continued from previous page)

```
['E', 6]
sage: ct.dual()
['E', 6, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['E',Integer(6),Integer(1)])
>>> ct
['E', 6, 1]
>>> ct._repr_(compact = True)
'E6~'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
False
>>> ct.is_affine()
True
>>> ct.is_untwisted_affine()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
True
>>> ct.classical()
['E', 6]
>>> ct.dual()
['E', 6, 1]
```

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['E',6,1]).ascii_art(label = lambda x: x+2))
        O 2
        |
        |
        O 4
        |
        |
O---O---O---O---O
3   5   6   7   8
sage: print(CartanType(['E',7,1]).ascii_art(label = lambda x: x+2))
            O 4
            |
            |
O---O---O---O---O---O---O
2   3   5   6   7   8   9
sage: print(CartanType(['E',8,1]).ascii_art(label = lambda x: x-3))
        O -1
        |
        |
```

(continues on next page)

(continued from previous page)

```
O---O---O---O---O---O---O---O
-2  0   1   2   3   4   5   -3
```

```
>>> from sage.all import *
>>> print(CartanType(['E',Integer(6),Integer(1)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
        O 2
        |
        |
        O 4
        |
        |
O---O---O---O---O
3   5   6   7   8
>>> print(CartanType(['E',Integer(7),Integer(1)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
            O 4
            |
            |
O---O---O---O---O---O---O
2   3   5   6   7   8   9
>>> print(CartanType(['E',Integer(8),Integer(1)]).ascii_art(label = lambda x:␣
↪x-Integer(3)))
        O -1
        |
        |
O---O---O---O---O---O---O---O
-2  0   1   2   3   4   5   -3
```

dynkin_diagram()

Return the extended Dynkin diagram for affine type E.

EXAMPLES:

```
sage: e = CartanType(['E', 6, 1]).dynkin_diagram(); e                    #␣
↪needs sage.graphs
        O 0
        |
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
sage: e.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(0, 2, 1),
 (1, 3, 1),
 (2, 0, 1),
 (2, 4, 1),
 (3, 1, 1),
```

(continues on next page)

(continued from previous page)

```
 (3, 4, 1),
 (4, 2, 1),
 (4, 3, 1),
 (4, 5, 1),
 (5, 4, 1),
 (5, 6, 1),
 (6, 5, 1)]

sage: # needs sage.graphs
sage: e = CartanType(['E', 7, 1]).dynkin_diagram(); e
            O 2
            |
            |
O---O---O---O---O---O---O
0   1   3   4   5   6   7
E7~
sage: e.edges(sort=True)
[(0, 1, 1), (1, 0, 1), (1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1),
 (4, 2, 1), (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1),
 (6, 5, 1), (6, 7, 1), (7, 6, 1)]
sage: e = CartanType(['E', 8, 1]).dynkin_diagram(); e
        O 2
        |
        |
O---O---O---O---O---O---O---O
1   3   4   5   6   7   8   0
E8~
sage: e.edges(sort=True)
[(0, 8, 1), (1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1),
 (4, 2, 1), (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1),
 (6, 5, 1), (6, 7, 1), (7, 6, 1), (7, 8, 1), (8, 0, 1), (8, 7, 1)]
```

```
>>> from sage.all import *
>>> e = CartanType(['E', Integer(6), Integer(1)]).dynkin_diagram(); e         ␣
↪              # needs sage.graphs
        O 0
        |
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
>>> e.edges(sort=True)                                                     #␣
↪needs sage.graphs
[(0, 2, 1),
 (1, 3, 1),
 (2, 0, 1),
 (2, 4, 1),
 (3, 1, 1),
 (3, 4, 1),
```

(continues on next page)

(continued from previous page)

```
 (4, 2, 1),
 (4, 3, 1),
 (4, 5, 1),
 (5, 4, 1),
 (5, 6, 1),
 (6, 5, 1)]

>>> # needs sage.graphs
>>> e = CartanType(['E', Integer(7), Integer(1)]).dynkin_diagram(); e
            O 2
            |
            |
O---O---O---O---O---O---O
0   1   3   4   5   6   7
E7~
>>> e.edges(sort=True)
[(0, 1, 1), (1, 0, 1), (1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1),
 (4, 2, 1), (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1),
 (6, 5, 1), (6, 7, 1), (7, 6, 1)]
>>> e = CartanType(['E', Integer(8), Integer(1)]).dynkin_diagram(); e
        O 2
        |
        |
O---O---O---O---O---O---O---O
1   3   4   5   6   7   8   0
E8~
>>> e.edges(sort=True)
[(0, 8, 1), (1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1),
 (4, 2, 1), (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1),
 (6, 5, 1), (6, 7, 1), (7, 6, 1), (7, 8, 1), (8, 0, 1), (8, 7, 1)]
```

5.1.255 Root system data for type F

class sage.combinat.root_system.type_F.**AmbientSpace**(*root_system*, *base_ring*)

Bases: *AmbientSpace*

The lattice behind F_4. The computations are based on Bourbaki, Groupes et Algèbres de Lie, Ch. 4,5,6 (planche VIII).

dimension()

Return the dimension of self.

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.dimension()
4
```

```
>>> from sage.all import *
>>> e = RootSystem(['F',Integer(4)]).ambient_space()
>>> e.dimension()
4
```

fundamental_weights()

Return the fundamental weights of `self`.

EXAMPLES:

```
sage: e =  RootSystem(['F',4]).ambient_space()
sage: e.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4:␣
↪(1, 0, 0, 0)}
```

```
>>> from sage.all import *
>>> e =  RootSystem(['F',Integer(4)]).ambient_space()
>>> e.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4:␣
↪(1, 0, 0, 0)}
```

negative_roots()

Return the negative roots.

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.negative_roots()
[(-1, 0, 0, 0),
(0, -1, 0, 0),
(0, 0, -1, 0),
(0, 0, 0, -1),
(-1, -1, 0, 0),
(-1, 0, -1, 0),
(-1, 0, 0, -1),
(0, -1, -1, 0),
(0, -1, 0, -1),
(0, 0, -1, -1),
(-1, 1, 0, 0),
(-1, 0, 1, 0),
(-1, 0, 0, 1),
(0, -1, 1, 0),
(0, -1, 0, 1),
(0, 0, -1, 1),
(-1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2),
(-1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2),
(-1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2),
(-1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['F',Integer(4)]).ambient_space()
>>> e.negative_roots()
[(-1, 0, 0, 0),
(0, -1, 0, 0),
(0, 0, -1, 0),
```

(continues on next page)

(continued from previous page)

```
(0, 0, 0, -1),
(-1, -1, 0, 0),
(-1, 0, -1, 0),
(-1, 0, 0, -1),
(0, -1, -1, 0),
(0, -1, 0, -1),
(0, 0, -1, -1),
(-1, 1, 0, 0),
(-1, 0, 1, 0),
(-1, 0, 0, 1),
(0, -1, 1, 0),
(0, -1, 0, 1),
(0, 0, -1, 1),
(-1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2),
(-1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2),
(-1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2),
(-1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2)]
```

positive_roots()

Return the positive roots.

These are the roots which are positive with respect to the lexicographic ordering of the basis elements ($\epsilon_1 < \epsilon_2 < \epsilon_3 < \epsilon_4$).

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.positive_roots()
[(1, 0, 0, 0),
(0, 1, 0, 0),
(0, 0, 1, 0),
(0, 0, 0, 1),
(1, 1, 0, 0),
(1, 0, 1, 0),
(1, 0, 0, 1),
(0, 1, 1, 0),
(0, 1, 0, 1),
(0, 0, 1, 1),
(1, -1, 0, 0),
(1, 0, -1, 0),
(1, 0, 0, -1),
(0, 1, -1, 0),
(0, 1, 0, -1),
(0, 0, 1, -1),
(1/2, 1/2, 1/2, 1/2),
(1/2, 1/2, 1/2, -1/2),
(1/2, 1/2, -1/2, 1/2),
(1/2, 1/2, -1/2, -1/2),
(1/2, -1/2, 1/2, 1/2),
```

(continues on next page)

(continued from previous page)

```
(1/2, -1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, 1/2),
(1/2, -1/2, -1/2, -1/2)]
sage: e.rho()
(11/2, 5/2, 3/2, 1/2)
```

```
>>> from sage.all import *
>>> e = RootSystem(['F',Integer(4)]).ambient_space()
>>> e.positive_roots()
[(1, 0, 0, 0),
(0, 1, 0, 0),
(0, 0, 1, 0),
(0, 0, 0, 1),
(1, 1, 0, 0),
(1, 0, 1, 0),
(1, 0, 0, 1),
(0, 1, 1, 0),
(0, 1, 0, 1),
(0, 0, 1, 1),
(1, -1, 0, 0),
(1, 0, -1, 0),
(1, 0, 0, -1),
(0, 1, -1, 0),
(0, 1, 0, -1),
(0, 0, 1, -1),
(1/2, 1/2, 1/2, 1/2),
(1/2, 1/2, 1/2, -1/2),
(1/2, 1/2, -1/2, 1/2),
(1/2, 1/2, -1/2, -1/2),
(1/2, -1/2, 1/2, 1/2),
(1/2, -1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, 1/2),
(1/2, -1/2, -1/2, -1/2)]
>>> e.rho()
(11/2, 5/2, 3/2, 1/2)
```

root(*i*, *j=None*, *k=None*, *l=None*, *p1=0*, *p2=0*, *p3=0*, *p4=0*)

Compute a root from base elements of the underlying lattice. The arguments specify the basis elements and the signs. Sadly, the base elements are indexed zero-based. We assume that if one of the indices is not given, the rest are not as well.

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: [ e.root(i,j,p2=1) for i in range(e.n) for j in range(i+1,e.n) ]
[(1, -1, 0, 0), (1, 0, -1, 0), (1, 0, 0, -1), (0, 1, -1, 0), (0, 1, 0, -1),
↪(0, 0, 1, -1)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['F',Integer(4)]).ambient_space()
>>> [ e.root(i,j,p2=Integer(1)) for i in range(e.n) for j in
↪range(i+Integer(1),e.n) ]
```

(continues on next page)

(continued from previous page)

```
[(1, -1, 0, 0), (1, 0, -1, 0), (1, 0, 0, -1), (0, 1, -1, 0), (0, 1, 0, -1),␣
↪(0, 0, 1, -1)]
```

simple_root (i)

Return the i-th simple root.

It is computed according to what Bourbaki calls the Base:

$$\alpha_1 = \epsilon_2 - \epsilon_3, \alpha_2 = \epsilon_3 - \epsilon_4, \alpha_3 = \epsilon_4, \alpha_4 = \frac{1}{2}\left(\epsilon_1 - \epsilon_2 - \epsilon_3 - \epsilon_4\right).$$

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1), 4: (1/2, -
↪1/2, -1/2, -1/2)}
```

```
>>> from sage.all import *
>>> e = RootSystem(['F',Integer(4)]).ambient_space()
>>> e.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1), 4: (1/2, -
↪1/2, -1/2, -1/2)}
```

class sage.combinat.root_system.type_F.**CartanType**

Bases: *CartanType_standard_finite*, *CartanType_simple*, *CartanType_crystallographic*

EXAMPLES:

```
sage: ct = CartanType(['F',4])
sage: ct
['F', 4]
sage: ct._repr_(compact = True)
'F4'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: ct.affine()
['F', 4, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['F',Integer(4)])
>>> ct
['F', 4]
>>> ct._repr_(compact = True)
```

(continues on next page)

(continued from previous page)

```
'F4'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
>>> ct.affine()
['F', 4, 1]
```

AmbientSpace

alias of *AmbientSpace*

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['F',4]).ascii_art(label = lambda x: x+2))
O---O=>=O---O
3   4   5   6
sage: print(CartanType(['F',4]).ascii_art(label = lambda x: x-2))
O---O=>=O---O
-1  0   1   2
```

```
>>> from sage.all import *
>>> print(CartanType(['F',Integer(4)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
O---O=>=O---O
3   4   5   6
>>> print(CartanType(['F',Integer(4)]).ascii_art(label = lambda x: x-
↪Integer(2)))
O---O=>=O---O
-1  0   1   2
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['F',4]).coxeter_number()
12
```

```
>>> from sage.all import *
>>> CartanType(['F',Integer(4)]).coxeter_number()
12
```

dual()

Return the dual Cartan type.

This uses that F_4 is self-dual up to relabelling.

EXAMPLES:

```
sage: F4 = CartanType(['F',4])
sage: F4.dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}

sage: F4.dynkin_diagram()                                            #␣
↪needs sage.graphs
O---O=>=O---O
1   2   3   4
F4
sage: F4.dual().dynkin_diagram()                                     #␣
↪needs sage.graphs
O---O=>=O---O
4   3   2   1
F4 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> F4 = CartanType(['F',Integer(4)])
>>> F4.dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}

>>> F4.dynkin_diagram()                                              #␣
↪needs sage.graphs
O---O=>=O---O
1   2   3   4
F4
>>> F4.dual().dynkin_diagram()                                       #␣
↪needs sage.graphs
O---O=>=O---O
4   3   2   1
F4 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

dual_coxeter_number()

Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['F',4]).dual_coxeter_number()
9
```

```
>>> from sage.all import *
>>> CartanType(['F',Integer(4)]).dual_coxeter_number()
9
```

dynkin_diagram()

Return a Dynkin diagram for type F.

EXAMPLES:

```
sage: f = CartanType(['F',4]).dynkin_diagram(); f                          #␣
↪needs sage.graphs
O---O=>=O---O
1   2   3   4
F4
sage: f.edges(sort=True)                                                   #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1), (3, 4, 1), (4, 3, 1)]
```

```
>>> from sage.all import *
>>> f = CartanType(['F',Integer(4)]).dynkin_diagram(); f                    ␣
↪     # needs sage.graphs
O---O=>=O---O
1   2   3   4
F4
>>> f.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1), (3, 4, 1), (4, 3, 1)]
```

5.1.256 Root system data for (untwisted) type F affine

class sage.combinat.root_system.type_F_affine.**CartanType**

Bases: *CartanType_standard_untwisted_affine*

EXAMPLES:

```
sage: ct = CartanType(['F',4,1])
sage: ct
['F', 4, 1]
sage: ct._repr_(compact = True)
'F4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['F', 4]
sage: ct.dual()
['F', 4, 1]^*
sage: ct.dual().is_untwisted_affine()
False
```

```
>>> from sage.all import *
>>> ct = CartanType(['F',Integer(4),Integer(1)])
>>> ct
['F', 4, 1]
>>> ct._repr_(compact = True)
'F4~'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
False
>>> ct.is_affine()
True
>>> ct.is_untwisted_affine()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.classical()
['F', 4]
>>> ct.dual()
['F', 4, 1]^*
>>> ct.dual().is_untwisted_affine()
False
```

ascii_art(*label=None*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['F',4,1]).ascii_art(label = lambda x: x+2))
O---O---O=>=O---O
2   3   4   5   6
```

```
>>> from sage.all import *
>>> print(CartanType(['F',Integer(4),Integer(1)]).ascii_art(label = lambda x:␣
→x+Integer(2)))
O---O---O=>=O---O
2   3   4   5   6
```

dynkin_diagram()

Return the extended Dynkin diagram for affine type F.

EXAMPLES:

```
sage: f = CartanType(['F', 4, 1]).dynkin_diagram(); f                  #␣
→needs sage.graphs
O---O---O=>=O---O
0   1   2   3   4
F4~
sage: f.edges(sort=True)                                                #␣
→needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
[(0, 1, 1), (1, 0, 1), (1, 2, 1), (2, 1, 1),
 (2, 3, 2), (3, 2, 1), (3, 4, 1), (4, 3, 1)]
```

```
>>> from sage.all import *
>>> f = CartanType(['F', Integer(4), Integer(1)]).dynkin_diagram(); f      ␣
↪              # needs sage.graphs
O---O---O=>=O---O
0   1   2   3   4
F4~
>>> f.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(0, 1, 1), (1, 0, 1), (1, 2, 1), (2, 1, 1),
 (2, 3, 2), (3, 2, 1), (3, 4, 1), (4, 3, 1)]
```

5.1.257 Root system data for type G

class sage.combinat.root_system.type_G.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

EXAMPLES:

```
sage: e = RootSystem(['G',2]).ambient_space(); e
Ambient space of the Root system of type ['G', 2]
```

```
>>> from sage.all import *
>>> e = RootSystem(['G',Integer(2)]).ambient_space(); e
Ambient space of the Root system of type ['G', 2]
```

One can not construct the ambient lattice because the simple coroots have rational coefficients:

```
sage: e.simple_coroots()
Finite family {1: (0, 1, -1), 2: (1/3, -2/3, 1/3)}
sage: e.smallest_base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> e.simple_coroots()
Finite family {1: (0, 1, -1), 2: (1/3, -2/3, 1/3)}
>>> e.smallest_base_ring()
Rational Field
```

By default, this ambient space uses the barycentric projection for plotting:

```
sage: # needs sage.symbolic
sage: L = RootSystem(["G",2]).ambient_space()
sage: e = L.basis()
sage: L._plot_projection(e[0])
(1/2, 989/1142)
sage: L._plot_projection(e[1])
(-1, 0)
sage: L._plot_projection(e[2])
```

(continues on next page)

(continued from previous page)

```
(1/2, -989/1142)
sage: L = RootSystem(["A",3]).ambient_space()
sage: l = L.an_element(); l
(2, 2, 3, 0)
sage: L._plot_projection(l)
(0, -1121/1189, 7/3)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> L = RootSystem(["G",Integer(2)]).ambient_space()
>>> e = L.basis()
>>> L._plot_projection(e[Integer(0)])
(1/2, 989/1142)
>>> L._plot_projection(e[Integer(1)])
(-1, 0)
>>> L._plot_projection(e[Integer(2)])
(1/2, -989/1142)
>>> L = RootSystem(["A",Integer(3)]).ambient_space()
>>> l = L.an_element(); l
(2, 2, 3, 0)
>>> L._plot_projection(l)
(0, -1121/1189, 7/3)
```

See also

- `sage.combinat.root_system.root_lattice_realizations.RootLatticeRealizations.ParentMethods._plot_projection()`

dimension()

EXAMPLES:

```
sage: e = RootSystem(['G',2]).ambient_space()
sage: e.dimension()
3
```

```
>>> from sage.all import *
>>> e = RootSystem(['G',Integer(2)]).ambient_space()
>>> e.dimension()
3
```

fundamental_weights()

EXAMPLES:

```
sage: CartanType(['G',2]).root_system().ambient_space().fundamental_weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

```
>>> from sage.all import *
>>> CartanType(['G',Integer(2)]).root_system().ambient_space().fundamental_
↪weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

negative_roots()

EXAMPLES:

```
sage: CartanType(['G',2]).root_system().ambient_space().negative_roots()
[(0, -1, 1), (-1, 2, -1), (-1, 1, 0), (-1, 0, 1), (-1, -1, 2), (-2, 1, 1)]
```

```
>>> from sage.all import *
>>> CartanType(['G',Integer(2)]).root_system().ambient_space().negative_
→roots()
[(0, -1, 1), (-1, 2, -1), (-1, 1, 0), (-1, 0, 1), (-1, -1, 2), (-2, 1, 1)]
```

positive_roots()

EXAMPLES:

```
sage: CartanType(['G',2]).root_system().ambient_space().positive_roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

```
>>> from sage.all import *
>>> CartanType(['G',Integer(2)]).root_system().ambient_space().positive_
→roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

simple_root(*i*)

EXAMPLES:

```
sage: CartanType(['G',2]).root_system().ambient_space().simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

```
>>> from sage.all import *
>>> CartanType(['G',Integer(2)]).root_system().ambient_space().simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

class sage.combinat.root_system.type_G.**CartanType**

Bases: *CartanType_standard_finite*, *CartanType_simple*, *CartanType_crystallographic*

EXAMPLES:

```
sage: ct = CartanType(['G',2])
sage: ct
['G', 2]
sage: ct._repr_(compact = True)
'G2'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.dual()
['G', 2] relabelled by {1: 2, 2: 1}
```

(continues on next page)

(continued from previous page)

```
sage: ct.affine()
['G', 2, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['G',Integer(2)])
>>> ct
['G', 2]
>>> ct._repr_(compact = True)
'G2'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.dual()
['G', 2] relabelled by {1: 2, 2: 1}
>>> ct.affine()
['G', 2, 1]
```

AmbientSpace

alias of *AmbientSpace*

ascii_art (*label=None*, *node=None*)

Return an ascii art representation of the Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['G',2]).ascii_art(label=lambda x: x+2))
  3
O=<=O
3   4
```

```
>>> from sage.all import *
>>> print(CartanType(['G',Integer(2)]).ascii_art(label=lambda x:␣
↪x+Integer(2)))
  3
O=<=O
3   4
```

coxeter_number ()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['G',2]).coxeter_number()
6
```

```
>>> from sage.all import *
>>> CartanType(['G',Integer(2)]).coxeter_number()
6
```

dual()

Return the dual Cartan type.

This uses that G_2 is self-dual up to relabelling.

EXAMPLES:

```
sage: G2 = CartanType(['G',2])
sage: G2.dual()
['G', 2] relabelled by {1: 2, 2: 1}

sage: G2.dynkin_diagram()                                                    #␣
↪needs sage.graphs
  3
O=<=O
1   2
G2
sage: G2.dual().dynkin_diagram()                                             #␣
↪needs sage.graphs
  3
O=<=O
2   1
G2 relabelled by {1: 2, 2: 1}
```

```
>>> from sage.all import *
>>> G2 = CartanType(['G',Integer(2)])
>>> G2.dual()
['G', 2] relabelled by {1: 2, 2: 1}

>>> G2.dynkin_diagram()                                                    #␣
↪needs sage.graphs
  3
O=<=O
1   2
G2
>>> G2.dual().dynkin_diagram()                                             #␣
↪needs sage.graphs
  3
O=<=O
2   1
G2 relabelled by {1: 2, 2: 1}
```

dual_coxeter_number()

Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['G',2]).dual_coxeter_number()
4
```

```
>>> from sage.all import *
>>> CartanType(['G',Integer(2)]).dual_coxeter_number()
4
```

dynkin_diagram()

Return a Dynkin diagram for type G.

EXAMPLES:

```
sage: g = CartanType(['G',2]).dynkin_diagram(); g                        #␣
↪needs sage.graphs
  3
O=<=O
1   2
G2
sage: g.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 3)]
```

```
>>> from sage.all import *
>>> g = CartanType(['G',Integer(2)]).dynkin_diagram(); g                 ␣
↪       # needs sage.graphs
  3
O=<=O
1   2
G2
>>> g.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 3)]
```

5.1.258 Root system data for (untwisted) type G affine

class sage.combinat.root_system.type_G_affine.**CartanType**

Bases: *CartanType_standard_untwisted_affine*

EXAMPLES:

```
sage: ct = CartanType(['G',2,1])
sage: ct
['G', 2, 1]
sage: ct._repr_(compact = True)
'G2~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
```

(continues on next page)

(continued from previous page)

```
sage: ct.is_simply_laced()
False
sage: ct.classical()
['G', 2]
sage: ct.dual()
['G', 2, 1]^*
sage: ct.dual().is_untwisted_affine()
False
```

```
>>> from sage.all import *
>>> ct = CartanType(['G',Integer(2),Integer(1)])
>>> ct
['G', 2, 1]
>>> ct._repr_(compact = True)
'G2~'

>>> ct.is_irreducible()
True
>>> ct.is_finite()
False
>>> ct.is_affine()
True
>>> ct.is_untwisted_affine()
True
>>> ct.is_crystallographic()
True
>>> ct.is_simply_laced()
False
>>> ct.classical()
['G', 2]
>>> ct.dual()
['G', 2, 1]^*
>>> ct.dual().is_untwisted_affine()
False
```

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of the Dynkin diagram.

EXAMPLES:

```
sage: print(CartanType(['G',2,1]).ascii_art(label = lambda x: x+2))
  3
O=<=O---O
3   4   2
```

```
>>> from sage.all import *
>>> print(CartanType(['G',Integer(2),Integer(1)]).ascii_art(label = lambda x:␣
↪x+Integer(2)))
  3
O=<=O---O
3   4   2
```

dynkin_diagram()

Return the extended Dynkin diagram for type G.

EXAMPLES:

```
sage: g = CartanType(['G',2,1]).dynkin_diagram(); g                     #␣
↪needs sage.graphs
  3
O=<=O---O
1   2   0
G2~
sage: g.edges(sort=True)                                                #␣
↪needs sage.graphs
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 3)]
```

```
>>> from sage.all import *
>>> g = CartanType(['G',Integer(2),Integer(1)]).dynkin_diagram(); g    ␣
↪              # needs sage.graphs
  3
O=<=O---O
1   2   0
G2~
>>> g.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 3)]
```

5.1.259 Root system data for type H

class sage.combinat.root_system.type_H.**CartanType**(*n*)

Bases: *CartanType_standard_finite*, *CartanType_simple*

EXAMPLES:

```
sage: ct = CartanType(['H',3])
sage: ct
['H', 3]
sage: ct._repr_(compact = True)
'H3'
sage: ct.rank()
3

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_affine()
False
sage: ct.is_crystallographic()
False
sage: ct.is_simply_laced()
False
```

```
>>> from sage.all import *
>>> ct = CartanType(['H',Integer(3)])
>>> ct
['H', 3]
>>> ct._repr_(compact = True)
'H3'
>>> ct.rank()
3

>>> ct.is_irreducible()
True
>>> ct.is_finite()
True
>>> ct.is_affine()
False
>>> ct.is_crystallographic()
False
>>> ct.is_simply_laced()
False
```

coxeter_diagram()

Return a Coxeter diagram for type H.

EXAMPLES:

```
sage: ct = CartanType(['H',3])
sage: ct.coxeter_diagram()                                              #␣
↪needs sage.graphs
Graph on 3 vertices
sage: ct.coxeter_diagram().edges(sort=True)                            #␣
↪needs sage.graphs
[(1, 2, 3), (2, 3, 5)]
sage: ct.coxeter_matrix()                                               #␣
↪needs sage.graphs
[1 3 2]
[3 1 5]
[2 5 1]

sage: ct = CartanType(['H',4])
sage: ct.coxeter_diagram()                                              #␣
↪needs sage.graphs
Graph on 4 vertices
sage: ct.coxeter_diagram().edges(sort=True)                            #␣
↪needs sage.graphs
[(1, 2, 3), (2, 3, 3), (3, 4, 5)]
sage: ct.coxeter_matrix()                                               #␣
↪needs sage.graphs
[1 3 2 2]
[3 1 3 2]
[2 3 1 5]
[2 2 5 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['H',Integer(3)])
>>> ct.coxeter_diagram()                                                    #␣
↪needs sage.graphs
Graph on 3 vertices
>>> ct.coxeter_diagram().edges(sort=True)                                   #␣
↪needs sage.graphs
[(1, 2, 3), (2, 3, 5)]
>>> ct.coxeter_matrix()                                                     #␣
↪needs sage.graphs
[1 3 2]
[3 1 5]
[2 5 1]

>>> ct = CartanType(['H',Integer(4)])
>>> ct.coxeter_diagram()                                                    #␣
↪needs sage.graphs
Graph on 4 vertices
>>> ct.coxeter_diagram().edges(sort=True)                                   #␣
↪needs sage.graphs
[(1, 2, 3), (2, 3, 3), (3, 4, 5)]
>>> ct.coxeter_matrix()                                                     #␣
↪needs sage.graphs
[1 3 2 2]
[3 1 3 2]
[2 3 1 5]
[2 2 5 1]
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['H',3]).coxeter_number()
10
sage: CartanType(['H',4]).coxeter_number()
30
```

```
>>> from sage.all import *
>>> CartanType(['H',Integer(3)]).coxeter_number()
10
>>> CartanType(['H',Integer(4)]).coxeter_number()
30
```

5.1.260 Root system data for type I

class sage.combinat.root_system.type_I.**CartanType**(*n*)

Bases: *CartanType_standard_finite*, *CartanType_simple*

EXAMPLES:

```
sage: ct = CartanType(['I',5])
sage: ct
```

(continues on next page)

(continued from previous page)

```
['I', 5]
sage: ct._repr_(compact = True)
'I5'
sage: ct.rank()
2
sage: ct.index_set()
(1, 2)

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_affine()
False
sage: ct.is_crystallographic()
False
sage: ct.is_simply_laced()
False
```

```
>>> from sage.all import *
>>> ct = CartanType(['I',Integer(5)])
>>> ct
['I', 5]
>>> ct._repr_(compact = True)
'I5'
>>> ct.rank()
2
>>> ct.index_set()
(1, 2)

>>> ct.is_irreducible()
True
>>> ct.is_finite()
True
>>> ct.is_affine()
False
>>> ct.is_crystallographic()
False
>>> ct.is_simply_laced()
False
```

coxeter_diagram()

Return the Coxeter matrix for this type.

EXAMPLES:

```
sage: ct = CartanType(['I', 4])
sage: ct.coxeter_diagram()                                              #␣
↪needs sage.graphs
Graph on 2 vertices
sage: ct.coxeter_diagram().edges(sort=True)                             #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
[(1, 2, 4)]
sage: ct.coxeter_matrix()                                                    #␣
↪needs sage.graphs
[1 4]
[4 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['I', Integer(4)])
>>> ct.coxeter_diagram()                                                     #␣
↪needs sage.graphs
Graph on 2 vertices
>>> ct.coxeter_diagram().edges(sort=True)                                    #␣
↪needs sage.graphs
[(1, 2, 4)]
>>> ct.coxeter_matrix()                                                      #␣
↪needs sage.graphs
[1 4]
[4 1]
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['I',3]).coxeter_number()
3
sage: CartanType(['I',12]).coxeter_number()
12
```

```
>>> from sage.all import *
>>> CartanType(['I',Integer(3)]).coxeter_number()
3
>>> CartanType(['I',Integer(12)]).coxeter_number()
12
```

index_set()

Type $I_2(p)$ is indexed by $\{1, 2\}$.

EXAMPLES:

```
sage: CartanType(['I', 5]).index_set()
(1, 2)
```

```
>>> from sage.all import *
>>> CartanType(['I', Integer(5)]).index_set()
(1, 2)
```

rank()

Type $I_2(p)$ is of rank 2.

EXAMPLES:

```
sage: CartanType(['I', 5]).rank()
2
```

```
>>> from sage.all import *
>>> CartanType(['I', Integer(5)]).rank()
2
```

5.1.261 Root system data for type Q

class sage.combinat.root_system.type_Q.**CartanType**(*m*)

Bases: *CartanType_standard_finite*

Cartan Type Q_n.

See also

CartanType()

dual()

Return dual of self.

EXAMPLES:

```
sage: Q = CartanType(['Q',3])
sage: Q.dual()
['Q', 3]
```

```
>>> from sage.all import *
>>> Q = CartanType(['Q',Integer(3)])
>>> Q.dual()
['Q', 3]
```

index_set()

Return the index set for Cartan type Q.

The index set for type Q is of the form $\{-n, \ldots, -1, 1, \ldots, n\}$.

EXAMPLES:

```
sage: CartanType(['Q', 3]).index_set()
(1, 2, -2, -1)
```

```
>>> from sage.all import *
>>> CartanType(['Q', Integer(3)]).index_set()
(1, 2, -2, -1)
```

is_irreducible()

Return whether this Cartan type is irreducible.

EXAMPLES:

```
sage: Q = CartanType(['Q',3])
sage: Q.is_irreducible()
True
```

```
>>> from sage.all import *
>>> Q = CartanType(['Q',Integer(3)])
>>> Q.is_irreducible()
True
```

is_simply_laced()

Return whether this Cartan type is simply-laced.

EXAMPLES:

```
sage: Q = CartanType(['Q',3])
sage: Q.is_simply_laced()
True
```

```
>>> from sage.all import *
>>> Q = CartanType(['Q',Integer(3)])
>>> Q.is_simply_laced()
True
```

root_system()

Return the root system of `self`.

EXAMPLES:

```
sage: Q = CartanType(['Q',3])
sage: Q.root_system()
Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> Q = CartanType(['Q',Integer(3)])
>>> Q.root_system()
Root system of type ['A', 2]
```

5.1.262 Root system data for affine Cartan types

class sage.combinat.root_system.type_affine.**AmbientSpace**(*root_system*, *base_ring*)

Bases: *CombinatorialFreeModule*

Ambient space for affine types.

This is constructed from the data in the corresponding classical ambient space. Namely, this space is obtained by adding two elements δ and $\delta^\vee$ to the basis of the classical ambient space, and by endowing it with the canonical scalar product.

The coefficient of an element in $\delta^\vee$, thus its scalar product with $\delta^\vee$ gives its level, and dually for the colevel. The canonical projection onto the classical ambient space (by killing δ and $\delta^\vee$) maps the simple roots (except α_0) onto the corresponding classical simple roots, and similarly for the coroots, fundamental weights, … Altogether, this uniquely determines the embedding of the root, coroot, weight, and coweight lattices. See *simple_root()* and *fundamental_weight()* for the details.

⚠ **Warning**

In type BC, the null root is in fact:

```
sage: R = RootSystem(["BC",3,2]).ambient_space()
sage: R.null_root()                                                    #␣
↪needs sage.graphs
2*e['delta']
```

```
>>> from sage.all import *
>>> R = RootSystem(["BC",Integer(3),Integer(2)]).ambient_space()
>>> R.null_root()                                                    #␣
↪needs sage.graphs
2*e['delta']
```

⚠ **Warning**

In the literature one often considers a larger affine ambient space obtained from the classical ambient space by adding four dimensions, namely for the fundamental weight Λ_0 the fundamental coweight $\Lambda_0^\vee$, the null root δ, and the null coroot c (aka central element). In this larger ambient space, the scalar product is degenerate: $\langle\delta, \delta\rangle = 0$ and similarly for the null coroot.

In the current implementation, Λ_0 and the null coroot are identified:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: Lambda = L.fundamental_weights()                                 #␣
↪needs sage.graphs
sage: Lambda[0]                                                        #␣
↪needs sage.graphs
e['deltacheck']
sage: L.null_coroot()                                                  #␣
↪needs sage.graphs
e['deltacheck']
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3),Integer(1)]).ambient_space()
>>> Lambda = L.fundamental_weights()                                 #␣
↪needs sage.graphs
>>> Lambda[Integer(0)]                                                  ␣
↪    # needs sage.graphs
e['deltacheck']
>>> L.null_coroot()                                                  #␣
↪needs sage.graphs
e['deltacheck']
```

Therefore the scalar product of the null coroot with itself differs from the larger ambient space:

```
sage: L.null_coroot().scalar(L.null_coroot())                          #␣
↪needs sage.graphs
1
```

```
>>> from sage.all import *
>>> L.null_coroot().scalar(L.null_coroot())                          #␣
↪needs sage.graphs
1
```

In general, scalar products between two elements that do not live on "opposite sides" won't necessarily match.

EXAMPLES:

```
sage: R = RootSystem(["A",3,1])
sage: e = R.ambient_space(); e
Ambient space of the Root system of type ['A', 3, 1]
sage: TestSuite(e).run()
```

```
>>> from sage.all import *
>>> R = RootSystem(["A",Integer(3),Integer(1)])
>>> e = R.ambient_space(); e
Ambient space of the Root system of type ['A', 3, 1]
>>> TestSuite(e).run()
```

Systematic checks on all affine types:

```
sage: for ct in CartanType.samples(affine=True, crystallographic=True):
....:     if ct.classical().root_system().ambient_space() is not None:
....:         print(ct)
....:         L = ct.root_system().ambient_space()
....:         assert L
....:         TestSuite(L).run()
['A', 1, 1]
['A', 5, 1]
['B', 1, 1]
['B', 5, 1]
['C', 1, 1]
['C', 5, 1]
['D', 3, 1]
['D', 5, 1]
['E', 6, 1]
['E', 7, 1]
['E', 8, 1]
['F', 4, 1]
['G', 2, 1]
['BC', 1, 2]
['BC', 5, 2]
['B', 5, 1]^*
['C', 4, 1]^*
['F', 4, 1]^*
['G', 2, 1]^*
['BC', 1, 2]^*
['BC', 5, 2]^*
```

```
>>> from sage.all import *
>>> for ct in CartanType.samples(affine=True, crystallographic=True):
...     if ct.classical().root_system().ambient_space() is not None:
...         print(ct)
...         L = ct.root_system().ambient_space()
...         assert L
...         TestSuite(L).run()
```

(continues on next page)

(continued from previous page)

```
['A', 1, 1]
['A', 5, 1]
['B', 1, 1]
['B', 5, 1]
['C', 1, 1]
['C', 5, 1]
['D', 3, 1]
['D', 5, 1]
['E', 6, 1]
['E', 7, 1]
['E', 8, 1]
['F', 4, 1]
['G', 2, 1]
['BC', 1, 2]
['BC', 5, 2]
['B', 5, 1]^*
['C', 4, 1]^*
['F', 4, 1]^*
['G', 2, 1]^*
['BC', 1, 2]^*
['BC', 5, 2]^*
```

class Element

Bases: `IndexedFreeModuleElement`

associated_coroot()

Return the coroot associated to `self`.

INPUT:

- `self` – a root

EXAMPLES:

```
sage: # needs sage.graphs
sage: alpha = RootSystem(['C',2,1]).ambient_space().simple_roots()
sage: alpha
Finite family {0: -2*e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
sage: alpha[0].associated_coroot()
-e[0] + e['deltacheck']
sage: alpha[1].associated_coroot()
e[0] - e[1]
sage: alpha[2].associated_coroot()
e[1]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> alpha = RootSystem(['C',Integer(2),Integer(1)]).ambient_space().
↪simple_roots()
>>> alpha
Finite family {0: -2*e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
>>> alpha[Integer(0)].associated_coroot()
-e[0] + e['deltacheck']
>>> alpha[Integer(1)].associated_coroot()
```

(continues on next page)

(continued from previous page)

```
e[0] - e[1]
>>> alpha[Integer(2)].associated_coroot()
e[1]
```

inner_product(*other*)

Implement the canonical inner product of `self` with `other`.

EXAMPLES:

```
sage: e = RootSystem(['B',3,1]).ambient_space()
sage: B = e.basis()
sage: matrix([[x.inner_product(y) for x in B] for y in B])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: x = e.an_element(); x
2*e[0] + 2*e[1] + 3*e[2]
sage: x.inner_product(x)
17
```

```
>>> from sage.all import *
>>> e = RootSystem(['B',Integer(3),Integer(1)]).ambient_space()
>>> B = e.basis()
>>> matrix([[x.inner_product(y) for x in B] for y in B])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
>>> x = e.an_element(); x
2*e[0] + 2*e[1] + 3*e[2]
>>> x.inner_product(x)
17
```

`scalar()` is an alias for this method:

```
sage: x.scalar(x)
17
```

```
>>> from sage.all import *
>>> x.scalar(x)
17
```

Todo

Lift to CombinatorialFreeModule.Element as canonical_inner_product

scalar(*other*)

Implement the canonical inner product of `self` with `other`.

EXAMPLES:

```
sage: e = RootSystem(['B',3,1]).ambient_space()
sage: B = e.basis()
sage: matrix([[x.inner_product(y) for x in B] for y in B])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: x = e.an_element(); x
2*e[0] + 2*e[1] + 3*e[2]
sage: x.inner_product(x)
17
```

```
>>> from sage.all import *
>>> e = RootSystem(['B',Integer(3),Integer(1)]).ambient_space()
>>> B = e.basis()
>>> matrix([[x.inner_product(y) for x in B] for y in B])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
>>> x = e.an_element(); x
2*e[0] + 2*e[1] + 3*e[2]
>>> x.inner_product(x)
17
```

`scalar()` is an alias for this method:

```
sage: x.scalar(x)
17
```

```
>>> from sage.all import *
>>> x.scalar(x)
17
```

Todo

Lift to CombinatorialFreeModule.Element as canonical_inner_product

coroot_lattice()

EXAMPLES:

```
sage: RootSystem(["A",3,1]).ambient_lattice().coroot_lattice()
Ambient lattice of the Root system of type ['A', 3, 1]
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3),Integer(1)]).ambient_lattice().coroot_lattice()
Ambient lattice of the Root system of type ['A', 3, 1]
```

Todo

Factor out this code with the classical ambient space.

fundamental_weight(*i*)

Return the fundamental weight Λ_i in this ambient space.

It is constructed by taking the corresponding fundamental weight of the classical ambient space (or 0 for Λ_0) and raising it to the appropriate level by adding a suitable multiple of $\delta^\vee$.

EXAMPLES:

```
sage: RootSystem(['A',3,1]).ambient_space().fundamental_weight(2)              #␣
↪needs sage.graphs
e[0] + e[1] + e['deltacheck']
sage: RootSystem(['A',3,1]).ambient_space().fundamental_weights()              #␣
↪needs sage.graphs
Finite family {0: e['deltacheck'],
               1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + e['deltacheck'],
               3: e[0] + e[1] + e[2] + e['deltacheck']}
sage: RootSystem(['A',3]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
sage: A31wl = RootSystem(['A',3,1]).weight_lattice()
sage: A31wl.fundamental_weights().map(attrcall("level"))                        #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 1, 3: 1}

sage: RootSystem(['B',3,1]).ambient_space().fundamental_weights()              #␣
↪needs sage.graphs
Finite family {0: e['deltacheck'],
               1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + 2*e['deltacheck'],
               3: 1/2*e[0] + 1/2*e[1] + 1/2*e[2] + e['deltacheck']}
sage: RootSystem(['B',3]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
sage: B31wl = RootSystem(['B',3,1]).weight_lattice()
sage: B31wl.fundamental_weights().map(attrcall("level"))                        #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 2, 3: 1}
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3),Integer(1)]).ambient_space().fundamental_
↪weight(Integer(2))            # needs sage.graphs
e[0] + e[1] + e['deltacheck']
>>> RootSystem(['A',Integer(3),Integer(1)]).ambient_space().fundamental_
↪weights()            # needs sage.graphs
Finite family {0: e['deltacheck'],
               1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + e['deltacheck'],
               3: e[0] + e[1] + e[2] + e['deltacheck']}
>>> RootSystem(['A',Integer(3)]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
```

(continues on next page)

(continued from previous page)

```
>>> A31wl = RootSystem(['A',Integer(3),Integer(1)]).weight_lattice()
>>> A31wl.fundamental_weights().map(attrcall("level"))                      #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 1, 3: 1}

>>> RootSystem(['B',Integer(3),Integer(1)]).ambient_space().fundamental_
↪weights()            # needs sage.graphs
Finite family {0: e['deltacheck'],
               1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + 2*e['deltacheck'],
               3: 1/2*e[0] + 1/2*e[1] + 1/2*e[2] + e['deltacheck']}
>>> RootSystem(['B',Integer(3)]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
>>> B31wl = RootSystem(['B',Integer(3),Integer(1)]).weight_lattice()
>>> B31wl.fundamental_weights().map(attrcall("level"))                      #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 2, 3: 1}
```

In type BC dual, the coefficient of 'delta^vee' is the level divided by 2 to take into account that the null coroot is $2\delta^\vee$:

```
sage: R = CartanType(['BC',3,2]).dual().root_system()
sage: R.ambient_space().fundamental_weights()                                 #␣
↪needs sage.graphs
Finite family {0: e['deltacheck'],
               1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + e['deltacheck'],
               3: 1/2*e[0] + 1/2*e[1] + 1/2*e[2] + 1/2*e['deltacheck']}
sage: R.weight_lattice().fundamental_weights().map(attrcall("level"))         #␣
↪needs sage.graphs
Finite family {0: 2, 1: 2, 2: 2, 3: 1}
sage: R.ambient_space().null_coroot()                                         #␣
↪needs sage.graphs
2*e['deltacheck']
```

```
>>> from sage.all import *
>>> R = CartanType(['BC',Integer(3),Integer(2)]).dual().root_system()
>>> R.ambient_space().fundamental_weights()                                 #␣
↪needs sage.graphs
Finite family {0: e['deltacheck'],
               1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + e['deltacheck'],
               3: 1/2*e[0] + 1/2*e[1] + 1/2*e[2] + 1/2*e['deltacheck']}
>>> R.weight_lattice().fundamental_weights().map(attrcall("level"))         #␣
↪needs sage.graphs
Finite family {0: 2, 1: 2, 2: 2, 3: 1}
>>> R.ambient_space().null_coroot()                                         #␣
↪needs sage.graphs
2*e['deltacheck']
```

By a slight naming abuse this function also accepts "delta" as input so that it can be used to implement the embedding from the extended weight lattice:

```
sage: RootSystem(['A',3,1]).ambient_space().fundamental_weight("delta")
e['delta']
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3),Integer(1)]).ambient_space().fundamental_
↪weight("delta")
e['delta']
```

is_extended()

Return whether this is a realization of the extended weight lattice: yes!

See also

- *sage.combinat.root_system.weight_space.WeightSpace*
- *sage.combinat.root_system.weight_lattice_realizations.WeightLatticeRealizations.ParentMethods.is_extended()*

EXAMPLES:

```
sage: RootSystem(['A',3,1]).ambient_space().is_extended()
True
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3),Integer(1)]).ambient_space().is_extended()
True
```

simple_coroot(i)

Return the i-th simple coroot $\alpha_i^\vee$ of this affine ambient space.

EXAMPLES:

```
sage: RootSystem(["A",3,1]).ambient_space().simple_coroot(1)
e[0] - e[1]
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3),Integer(1)]).ambient_space().simple_
↪coroot(Integer(1))
e[0] - e[1]
```

It is built as the coroot associated to the simple root α_i:

```
sage: RootSystem(["B",3,1]).ambient_space().simple_roots()                  #␣
↪needs sage.graphs
Finite family {0: -e[0] - e[1] + e['delta'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: e[2]}
sage: RootSystem(["B",3,1]).ambient_space().simple_coroots()                #␣
↪needs sage.graphs
Finite family {0: -e[0] - e[1] + e['deltacheck'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: 2*e[2]}
```

```
>>> from sage.all import *
>>> RootSystem(["B",Integer(3),Integer(1)]).ambient_space().simple_roots()      ␣
↪               # needs sage.graphs
Finite family {0: -e[0] - e[1] + e['delta'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: e[2]}
>>> RootSystem(["B",Integer(3),Integer(1)]).ambient_space().simple_coroots()  ␣
↪               # needs sage.graphs
Finite family {0: -e[0] - e[1] + e['deltacheck'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: 2*e[2]}
```

Todo

Factor out this code with the classical ambient space.

simple_root(*i*)

Return the i-th simple root of this affine ambient space.

EXAMPLES:

It is built straightforwardly from the corresponding simple root α_i in the classical ambient space:

```
sage: RootSystem(["A",3,1]).ambient_space().simple_root(1)
e[0] - e[1]
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3),Integer(1)]).ambient_space().simple_
↪root(Integer(1))
e[0] - e[1]
```

For the special node (typically $i = 0$), α_0 is built from the other simple roots using the column annihilator of the Cartan matrix and adding δ, where δ is the null root:

```
sage: RootSystem(["A",3]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
sage: RootSystem(["A",3,1]).ambient_space().simple_roots()                  #␣
↪needs sage.graphs
Finite family {0: -e[0] + e[3] + e['delta'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: e[2] - e[3]}
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3)]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
>>> RootSystem(["A",Integer(3),Integer(1)]).ambient_space().simple_roots()      ␣
↪               # needs sage.graphs
Finite family {0: -e[0] + e[3] + e['delta'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: e[2] - e[3]}
```

Here is a twisted affine example:

```
sage: B31v = RootSystem(CartanType(["B",3,1]).dual())
sage: B31v.ambient_space().simple_roots()                                   #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
Finite family {0: -e[0] - e[1] + e['delta'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: 2*e[2]}
```

```
>>> from sage.all import *
>>> B31v = RootSystem(CartanType(["B",Integer(3),Integer(1)]).dual())
>>> B31v.ambient_space().simple_roots()                                     #␣
↪needs sage.graphs
Finite family {0: -e[0] - e[1] + e['delta'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: 2*e[2]}
```

In fact δ is really $1/a_0$ times the null root (see the discussion in `WeightSpace`) but this only makes a difference in type BC:

```
sage: L = RootSystem(CartanType(["BC",3,2])).ambient_space()
sage: L.simple_roots()                                                      #␣
↪needs sage.graphs
Finite family {0: -e[0] + e['delta'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: 2*e[2]}
sage: L.null_root()                                                         #␣
↪needs sage.graphs
2*e['delta']
```

```
>>> from sage.all import *
>>> L = RootSystem(CartanType(["BC",Integer(3),Integer(2)])).ambient_space()
>>> L.simple_roots()                                                        #␣
↪needs sage.graphs
Finite family {0: -e[0] + e['delta'], 1: e[0] - e[1],
               2: e[1] - e[2], 3: 2*e[2]}
>>> L.null_root()                                                           #␣
↪needs sage.graphs
2*e['delta']
```

Note

An alternative would have been to use the default implementation of the simple roots as linear combinations of the fundamental weights. However, as in type A_n it is preferable to take a slight variant to avoid rational coefficient (the usual GL_n vs SL_n issue).

See also

- `simple_root()`
- `WeightSpace`
- `CartanType.col_annihilator()`
- `null_root()`

classmethod smallest_base_ring(*cartan_type*)

Return the smallest base ring the ambient space can be defined on.

This is the smallest base ring for the associated classical ambient space.

See also

smallest_base_ring()

EXAMPLES:

```
sage: cartan_type = CartanType(["A",3,1])
sage: cartan_type.AmbientSpace.smallest_base_ring(cartan_type)
Integer Ring
sage: cartan_type = CartanType(["B",3,1])
sage: cartan_type.AmbientSpace.smallest_base_ring(cartan_type)
Rational Field
```

```
>>> from sage.all import *
>>> cartan_type = CartanType(["A",Integer(3),Integer(1)])
>>> cartan_type.AmbientSpace.smallest_base_ring(cartan_type)
Integer Ring
>>> cartan_type = CartanType(["B",Integer(3),Integer(1)])
>>> cartan_type.AmbientSpace.smallest_base_ring(cartan_type)
Rational Field
```

5.1.263 Root system data for dual Cartan types

class sage.combinat.root_system.type_dual.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

Ambient space for a dual finite Cartan type.

It is constructed in the canonical way from the ambient space of the original Cartan type by switching the roles of simple roots, fundamental weights, etc.

Note

Recall that, for any finite Cartan type, and in particular the a simply laced one, the dual Cartan type is constructed as another preexisting Cartan type. Furthermore the ambient space for an affine type is constructed from the ambient space for its classical type. Thus this code is not actually currently used.

It is kept for cross-checking and for reference in case it could become useful, e.g., for dual of general Kac-Moody types.

For the doctests, we need to explicitly create a dual type. Subsequently, since reconstruction of the dual of type F_4 is the relabelled Cartan type, pickling fails on the `TestSuite` run.

EXAMPLES:

```
sage: ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',4]))
sage: L = ct.root_system().ambient_space(); L
Ambient space of the Root system of type ['F', 4]^*
sage: TestSuite(L).run(skip=["_test_elements","_test_pickling"])            #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',
↪Integer(4)]))
>>> L = ct.root_system().ambient_space(); L
Ambient space of the Root system of type ['F', 4]^*
>>> TestSuite(L).run(skip=["_test_elements","_test_pickling"])          #␣
↪needs sage.graphs
```

dimension()

Return the dimension of this ambient space.

See also

sage.combinat.root_system.ambient_space.AmbientSpace.dimension()

EXAMPLES:

```
sage: ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',4]))
sage: L = ct.root_system().ambient_space()
sage: L.dimension()
4
```

```
>>> from sage.all import *
>>> ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',
↪Integer(4)]))
>>> L = ct.root_system().ambient_space()
>>> L.dimension()
4
```

fundamental_weights()

Return the fundamental weights.

They are computed from the simple roots by inverting the Cartan matrix. This is acceptable since this is only about ambient spaces for finite Cartan types. Also, we do not have to worry about the usual GL_n vs SL_n catch because type A is self dual.

An alternative would have been to start from the fundamental coweights in the dual ambient space, but those are not yet implemented.

EXAMPLES:

```
sage: ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',4]))
sage: L = ct.root_system().ambient_space()
sage: L.fundamental_weights()                                            #␣
↪needs sage.graphs
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3, 1, 1, 1), 4: (2, 0, 0,
↪ 0)}
```

```
>>> from sage.all import *
>>> ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',
↪Integer(4)]))
>>> L = ct.root_system().ambient_space()
>>> L.fundamental_weights()                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3, 1, 1, 1), 4: (2, 0, 0,
↪ 0)}
```

Note that this ambient space is isomorphic, but not equal, to that obtained by constructing F_4 dual by relabelling:

```
sage: ct = CartanType(['F',4]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: ct.root_system().ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (3/2, 1/2, 1/2, 1/2), 3: (2, 1, 1, 0), 4:␣
↪(1, 1, 0, 0)}
```

```
>>> from sage.all import *
>>> ct = CartanType(['F',Integer(4)]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
>>> ct.root_system().ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (3/2, 1/2, 1/2, 1/2), 3: (2, 1, 1, 0), 4:␣
↪(1, 1, 0, 0)}
```

simple_root(*i*)

Return the i-th simple root.

It is constructed by looking up the corresponding simple coroot in the ambient space for the dual Cartan type.

EXAMPLES:

```
sage: ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',4]))
sage: ct.root_system().ambient_space().simple_root(1)
(0, 1, -1, 0)

sage: ct.root_system().ambient_space().simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1,
↪ -1, -1)}

sage: ct.dual().root_system().ambient_space().simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1,
↪ -1, -1)}
```

```
>>> from sage.all import *
>>> ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',
↪Integer(4)]))
>>> ct.root_system().ambient_space().simple_root(Integer(1))
(0, 1, -1, 0)

>>> ct.root_system().ambient_space().simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1,
↪ -1, -1)}

>>> ct.dual().root_system().ambient_space().simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1,
↪ -1, -1)}
```

Note that this ambient space is isomorphic, but not equal, to that obtained by constructing F_4 dual by relabelling:

```
sage: ct = CartanType(['F',4]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: ct.root_system().ambient_space().simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2), 2: (0, 0, 0, 1), 3: (0, 0, 1, -1),␣
↪4: (0, 1, -1, 0)}
```

```
>>> from sage.all import *
>>> ct = CartanType(['F',Integer(4)]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
>>> ct.root_system().ambient_space().simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2), 2: (0, 0, 0, 1), 3: (0, 0, 1, -1),␣
↪4: (0, 1, -1, 0)}
```

class sage.combinat.root_system.type_dual.**CartanType**(*type*)

Bases: *CartanType_decorator*, *CartanType_crystallographic*

A class for dual Cartan types.

The dual of a (crystallographic) Cartan type is a Cartan type with the same index set, but all arrows reversed in the Dynkin diagram (otherwise said, the Cartan matrix is transposed). It shares a lot of properties in common with its dual. In particular, the Weyl group is isomorphic to that of the dual as a Coxeter group.

EXAMPLES:

For all finite Cartan types, and in particular the simply laced ones, the dual Cartan type is given by another preexisting Cartan type:

```
sage: CartanType(['A',4]).dual()
['A', 4]
sage: CartanType(['B',4]).dual()
['C', 4]
sage: CartanType(['C',4]).dual()
['B', 4]
sage: CartanType(['F',4]).dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(4)]).dual()
['A', 4]
>>> CartanType(['B',Integer(4)]).dual()
['C', 4]
>>> CartanType(['C',Integer(4)]).dual()
['B', 4]
>>> CartanType(['F',Integer(4)]).dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

So to exercise this class we consider some non simply laced affine Cartan types and also create explicitly F_4^* as a dual cartan type:

```
sage: from sage.combinat.root_system.type_dual import CartanType as CartanTypeDual
sage: F4d = CartanTypeDual(CartanType(['F',4])); F4d
['F', 4]^*
```

(continues on next page)

(continued from previous page)

```
sage: G21d = CartanType(['G',2,1]).dual(); G21d
['G', 2, 1]^*
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.type_dual import CartanType as CartanTypeDual
>>> F4d = CartanTypeDual(CartanType(['F',Integer(4)])); F4d
['F', 4]^*
>>> G21d = CartanType(['G',Integer(2),Integer(1)]).dual(); G21d
['G', 2, 1]^*
```

They share many properties with their original Cartan types:

```
sage: F4d.is_irreducible()
True
sage: F4d.is_crystallographic()
True
sage: F4d.is_simply_laced()
False
sage: F4d.is_finite()
True
sage: G21d.is_finite()
False
sage: F4d.is_affine()
False
sage: G21d.is_affine()
True
```

```
>>> from sage.all import *
>>> F4d.is_irreducible()
True
>>> F4d.is_crystallographic()
True
>>> F4d.is_simply_laced()
False
>>> F4d.is_finite()
True
>>> G21d.is_finite()
False
>>> F4d.is_affine()
False
>>> G21d.is_affine()
True
```

Note

F4d is pickled by construction as F4.dual() hence the above failure.

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of this Cartan type.

(by hacking the ascii art representation of the dual Cartan type)

EXAMPLES:

```
sage: print(CartanType(["B", 3, 1]).dual().ascii_art())
    O 0
    |
    |
O---O=<=O
1   2   3
sage: print(CartanType(["C", 4, 1]).dual().ascii_art())
O=<=O---O---O=>=O
0   1   2   3   4
sage: print(CartanType(["G", 2, 1]).dual().ascii_art())
  3
O=>=O---O
1   2   0
sage: print(CartanType(["F", 4, 1]).dual().ascii_art())
O---O---O=<=O---O
0   1   2   3   4
sage: print(CartanType(["BC", 4, 2]).dual().ascii_art())
O=>=O---O---O=>=O
0   1   2   3   4
```

```
>>> from sage.all import *
>>> print(CartanType(["B", Integer(3), Integer(1)]).dual().ascii_art())
    O 0
    |
    |
O---O=<=O
1   2   3
>>> print(CartanType(["C", Integer(4), Integer(1)]).dual().ascii_art())
O=<=O---O---O=>=O
0   1   2   3   4
>>> print(CartanType(["G", Integer(2), Integer(1)]).dual().ascii_art())
  3
O=>=O---O
1   2   0
>>> print(CartanType(["F", Integer(4), Integer(1)]).dual().ascii_art())
O---O---O=<=O---O
0   1   2   3   4
>>> print(CartanType(["BC", Integer(4), Integer(2)]).dual().ascii_art())
O=>=O---O---O=>=O
0   1   2   3   4
```

dual()

EXAMPLES:

```
sage: ct = CartanType(['F', 4, 1]).dual()
sage: ct.dual()
['F', 4, 1]
```

```
>>> from sage.all import *
>>> ct = CartanType(['F', Integer(4), Integer(1)]).dual()
>>> ct.dual()
```

(continues on next page)

(continued from previous page)

```
['F', 4, 1]
```

dynkin_diagram()

EXAMPLES:

```
sage: ct = CartanType(['F', 4, 1]).dual()
sage: ct.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O=<=O---O
0   1   2   3   4
F4~*
```

```
>>> from sage.all import *
>>> ct = CartanType(['F', Integer(4), Integer(1)]).dual()
>>> ct.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O=<=O---O
0   1   2   3   4
F4~*
```

class sage.combinat.root_system.type_dual.**CartanType_affine**(*type*)

Bases: *CartanType*, *CartanType_affine*

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: CartanType(['A', 7, 2]).basic_untwisted()
['A', 7]
sage: CartanType(['E', 6, 2]).basic_untwisted()
['E', 6]
sage: CartanType(['D', 4, 3]).basic_untwisted()
['D', 4]
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(7), Integer(2)]).basic_untwisted()
['A', 7]
>>> CartanType(['E', Integer(6), Integer(2)]).basic_untwisted()
['E', 6]
>>> CartanType(['D', Integer(4), Integer(3)]).basic_untwisted()
['D', 4]
```

classical()

Return the classical Cartan type associated with `self` (which should be affine).

EXAMPLES:

```
sage: CartanType(['A',3,1]).dual().classical()
['A', 3]
```

(continues on next page)

(continued from previous page)

```
sage: CartanType(['B',3,1]).dual().classical()
['C', 3]
sage: CartanType(['F',4,1]).dual().classical()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: CartanType(['BC',4,2]).dual().classical()
['B', 4]
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(3),Integer(1)]).dual().classical()
['A', 3]
>>> CartanType(['B',Integer(3),Integer(1)]).dual().classical()
['C', 3]
>>> CartanType(['F',Integer(4),Integer(1)]).dual().classical()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
>>> CartanType(['BC',Integer(4),Integer(2)]).dual().classical()
['B', 4]
```

special_node()

Implement *CartanType_affine.special_node()*.

The special node of the dual of an affine type T is the special node of T.

EXAMPLES:

```
sage: CartanType(['A',3,1]).dual().special_node()
0
sage: CartanType(['B',3,1]).dual().special_node()
0
sage: CartanType(['F',4,1]).dual().special_node()
0
sage: CartanType(['BC',4,2]).dual().special_node()
0
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(3),Integer(1)]).dual().special_node()
0
>>> CartanType(['B',Integer(3),Integer(1)]).dual().special_node()
0
>>> CartanType(['F',Integer(4),Integer(1)]).dual().special_node()
0
>>> CartanType(['BC',Integer(4),Integer(2)]).dual().special_node()
0
```

class sage.combinat.root_system.type_dual.**CartanType_finite**(*type*)

Bases: *CartanType*, *CartanType_finite*

AmbientSpace

alias of *AmbientSpace*

5.1.264 Extended Affine Weyl Groups

AUTHORS:

- Daniel Bump (2012): initial version
- Daniel Orr (2012): initial version
- Anne Schilling (2012): initial version
- Mark Shimozono (2012): initial version
- Nicolas M. Thiery (2012): initial version
- Mark Shimozono (2013): twisted affine root systems, multiple realizations, GL_n

`sage.combinat.root_system.extended_affine_weyl_group.`**`ExtendedAffineWeylGroup`**`(`*cartan_type*, *general_linear=None*, ***print_options*`)`

The extended affine Weyl group.

INPUT:

- `cartan_type` – an affine or finite Cartan type (a finite Cartan type is an abbreviation for its untwisted affinization)
- `general_linear` – (default: `None`) if `True` and `cartan_type` indicates untwisted type A, returns the universal central extension
- `print_options` – special instructions for printing elements (see below)

Mnemonics

- "P" – subgroup of translations
- "Pv" – subgroup of translations in a dual form
- "W0" – classical Weyl group
- "W" – affine Weyl group
- "F" – fundamental group of length zero elements

There are currently six realizations: "PW0", "W0P, "WF", "FW", "PvW0", and "W0Pv".

"PW0" means the semidirect product of "P" with "W0" acting from the right. "W0P" is similar but with "W0" acting from the left. "WF" is the semidirect product of "W" with "F" acting from the right, etc.

Recognized arguments for `print_options` are:

- `print_tuple` – boolean (default: `False`); if `True`, elements are printed (a, b), otherwise as $a * b$
- `affine` – prefix for simple reflections in the affine Weyl group
- `classical` – prefix for simple reflections in the classical Weyl group
- `translation` – prefix for the translation elements
- `fundamental` – prefix for the elements of the fundamental group

These options are not mutable.

The *extended affine Weyl group* was introduced in the following references.

REFERENCES:

- [Ka1990]

Notation

- R – an irreducible affine root system
- I – set of nodes of the Dynkin diagram of R
- R_0 – the classical subsystem of R
- I_0 – set of nodes of the Dynkin diagram of R_0
- E – extended affine Weyl group of type R
- W – affine Weyl group of type R
- W_0 – finite (classical) Weyl group (of type R_0)
- M – translation lattice for W
- L – translation lattice for E
- F – fundamental subgroup of E (the length zero elements)
- P – finite weight lattice
- Q – finite root lattice
- $P^\vee$ – finite coweight lattice
- $Q^\vee$ – finite coroot lattice

Translation lattices

The styles "PW0" and "W0P" use the following lattices:

- Untwisted affine: $L = P^\vee$, $M = Q^\vee$
- Dual of untwisted affine: $L = P$, $M = Q$
- BC_n ($A_{2n}^{(2)}$): $L = M = P$
- Dual of BC_n ($A_{2n}^{(2)\dagger}$): $L = M = P^\vee$

The styles "PvW0" and "W0Pv" use the following lattices:

- Untwisted affine: The weight lattice of the dual finite Cartan type.
- Dual untwisted affine: The same as for "PW0" and "W0P".

For mixed affine type ($A_{2n}^{(2)}$, aka $\tilde{BC}_n$, and their affine duals) the styles "PvW0" and "W0Pv" are not implemented.

Finite and affine Weyl groups W_0 and W

The finite Weyl group W_0 is generated by the simple reflections s_i for $i \in I_0$ where s_i is the reflection across a suitable hyperplane H_i through the origin in the real span V of the lattice M.

R specifies another (affine) hyperplane H_0. The affine Weyl group W is generated by W_0 and the reflection S_0 across H_0.

Extended affine Weyl group E

The complement in V of the set H of hyperplanes obtained from the H_i by the action of W, has connected components called alcoves. W acts freely and transitively on the set of alcoves. After the choice of a certain alcove (the fundamental alcove), there is an induced bijection from W to the set of alcoves under which the identity in W maps to the fundamental alcove.

Then L is the largest sublattice of V, whose translations stabilize the set of alcoves.

There are isomorphisms

$$W \cong M \rtimes W_0 \cong W_0 \ltimes M$$
$$E \cong L \rtimes W_0 \cong W_0 \ltimes L$$

Fundamental group of affine Dynkin automorphisms

Since L acts on the set of alcoves, the group $F = L/M$ may be viewed as a subgroup of the symmetries of the fundamental alcove or equivalently the symmetries of the affine Dynkin diagram. F acts on the set of alcoves and hence on W. Conjugation by an element of F acts on W by permuting the indices of simple reflections.

There are isomorphisms

$$E \cong F \ltimes W \cong W \rtimes F$$

An affine Dynkin node is *special* if it is conjugate to the zero node under some affine Dynkin automorphism.

There is a bijection $i \mapsto \pi_i$ from the set of special nodes to the group F, where π_i is the unique element of F that sends 0 to i. When $L = P$ (resp. $L = P^\vee$) the element π_i is induced (under the isomorphism $F \cong L/M$) by addition of the coset of the i-th fundamental weight (resp. coweight).

The length function of the Coxeter group W may be extended to E by $\ell(w\pi) = \ell(w)$ where $w \in W$ and $\pi \in F$. This is the number of hyperplanes in H separating the fundamental alcove from its image by $w\pi$ (or equivalently w).

It is known that if G is the compact Lie group of adjoint type with root system R_0 then F is isomorphic to the fundamental group of G, or to the center of its simply-connected covering group. That is why we call F the *fundamental group*.

In the future we may want to build an element of the group from an appropriate linear map f on some of the root lattice realizations for this Cartan type: W.from_endomorphism(f).

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(["A",2,1]); E
Extended affine Weyl group of type ['A', 2, 1]
sage: type(E)
<class 'sage.combinat.root_system.extended_affine_weyl_group.
↪ExtendedAffineWeylGroup_Class_with_category'>

sage: PW0 = E.PW0(); PW0
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2]
 (as a matrix group acting on the coweight lattice)

sage: W0P = E.W0P(); W0P
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Weyl Group of type ['A', 2] (as a matrix group acting on the coweight lattice)
```

(continues on next page)

(continued from previous page)

```
 acting on Multiplicative form of Coweight lattice of the Root system of type ['A
↪', 2]

sage: PvW0 = E.PvW0(); PvW0
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Multiplicative form of Weight lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2]
 (as a matrix group acting on the weight lattice)

sage: W0Pv = E.W0Pv(); W0Pv
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
 acting on Multiplicative form of Weight lattice of the Root system of type ['A',␣
↪2]

sage: WF = E.WF(); WF
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
 acted upon by Fundamental group of type ['A', 2, 1]

sage: FW = E.FW(); FW
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Fundamental group of type ['A', 2, 1] acting on Weyl Group of type ['A', 2, 1]
 (as a matrix group acting on the root lattice)
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(["A",Integer(2),Integer(1)]); E
Extended affine Weyl group of type ['A', 2, 1]
>>> type(E)
<class 'sage.combinat.root_system.extended_affine_weyl_group.
↪ExtendedAffineWeylGroup_Class_with_category'>

>>> PW0 = E.PW0(); PW0
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2]
 (as a matrix group acting on the coweight lattice)

>>> W0P = E.W0P(); W0P
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Weyl Group of type ['A', 2] (as a matrix group acting on the coweight lattice)
 acting on Multiplicative form of Coweight lattice of the Root system of type ['A
↪', 2]

>>> PvW0 = E.PvW0(); PvW0
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Multiplicative form of Weight lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2]
 (as a matrix group acting on the weight lattice)

>>> W0Pv = E.W0Pv(); W0Pv
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
```

(continues on next page)

(continued from previous page)

```
 Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
 acting on Multiplicative form of Weight lattice of the Root system of type ['A',␣
↪2]

>>> WF = E.WF(); WF
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
 acted upon by Fundamental group of type ['A', 2, 1]

>>> FW = E.FW(); FW
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of
 Fundamental group of type ['A', 2, 1] acting on Weyl Group of type ['A', 2, 1]
 (as a matrix group acting on the root lattice)
```

When the realizations are constructed from each other as above, there are built-in coercions between them.

```
sage: F = E.fundamental_group()
sage: x = WF.from_reduced_word([0,1,2]) * WF(F(2)); x
S0*S1*S2 * pi[2]
sage: FW(x)
pi[2] * S1*S2*S0
sage: W0P(x)
s1*s2*s1 * t[-2*Lambdacheck[1] - Lambdacheck[2]]
sage: PW0(x)
t[Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2*s1
sage: PvW0(x)
t[Lambda[1] + 2*Lambda[2]] * s1*s2*s1
```

```
>>> from sage.all import *
>>> F = E.fundamental_group()
>>> x = WF.from_reduced_word([Integer(0),Integer(1),Integer(2)]) *␣
↪WF(F(Integer(2))); x
S0*S1*S2 * pi[2]
>>> FW(x)
pi[2] * S1*S2*S0
>>> W0P(x)
s1*s2*s1 * t[-2*Lambdacheck[1] - Lambdacheck[2]]
>>> PW0(x)
t[Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2*s1
>>> PvW0(x)
t[Lambda[1] + 2*Lambda[2]] * s1*s2*s1
```

The translation lattice and its distinguished basis are obtained from `E`:

```
sage: L = E.lattice(); L
Coweight lattice of the Root system of type ['A', 2]
sage: b = E.lattice_basis(); b
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
```

```
>>> from sage.all import *
>>> L = E.lattice(); L
Coweight lattice of the Root system of type ['A', 2]
```

(continues on next page)

(continued from previous page)

```
>>> b = E.lattice_basis(); b
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
```

Translation lattice elements can be coerced into any realization:

```
sage: PW0(b[1]-b[2])
t[Lambdacheck[1] - Lambdacheck[2]]
sage: FW(b[1]-b[2])
pi[2] * S0*S1
```

```
>>> from sage.all import *
>>> PW0(b[Integer(1)]-b[Integer(2)])
t[Lambdacheck[1] - Lambdacheck[2]]
>>> FW(b[Integer(1)]-b[Integer(2)])
pi[2] * S0*S1
```

The dual form of the translation lattice and its basis are similarly obtained:

```
sage: Lv = E.dual_lattice(); Lv
Weight lattice of the Root system of type ['A', 2]
sage: bv = E.dual_lattice_basis(); bv
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: FW(bv[1]-bv[2])
pi[2] * S0*S1
```

```
>>> from sage.all import *
>>> Lv = E.dual_lattice(); Lv
Weight lattice of the Root system of type ['A', 2]
>>> bv = E.dual_lattice_basis(); bv
Finite family {1: Lambda[1], 2: Lambda[2]}
>>> FW(bv[Integer(1)]-bv[Integer(2)])
pi[2] * S0*S1
```

The abstract fundamental group is accessed from `E`:

```
sage: F = E.fundamental_group(); F
Fundamental group of type ['A', 2, 1]
```

```
>>> from sage.all import *
>>> F = E.fundamental_group(); F
Fundamental group of type ['A', 2, 1]
```

Its elements are indexed by the set of special nodes of the affine Dynkin diagram:

```
sage: E.cartan_type().special_nodes()
(0, 1, 2)
sage: F.special_nodes()
(0, 1, 2)
sage: [F(i) for i in F.special_nodes()]
[pi[0], pi[1], pi[2]]
```

```
>>> from sage.all import *
>>> E.cartan_type().special_nodes()
(0, 1, 2)
>>> F.special_nodes()
(0, 1, 2)
>>> [F(i) for i in F.special_nodes()]
[pi[0], pi[1], pi[2]]
```

There is a coercion from the fundamental group into each realization:

```
sage: F(2)
pi[2]
sage: WF(F(2))
pi[2]
sage: W0P(F(2))
s2*s1 * t[-Lambdacheck[1]]
sage: W0Pv(F(2))
s2*s1 * t[-Lambda[1]]
```

```
>>> from sage.all import *
>>> F(Integer(2))
pi[2]
>>> WF(F(Integer(2)))
pi[2]
>>> W0P(F(Integer(2)))
s2*s1 * t[-Lambdacheck[1]]
>>> W0Pv(F(Integer(2)))
s2*s1 * t[-Lambda[1]]
```

Using E one may access the classical and affine Weyl groups and their morphisms into each realization:

```
sage: W0 = E.classical_weyl(); W0
Weyl Group of type ['A', 2] (as a matrix group acting on the coweight lattice)
sage: v = W0.from_reduced_word([1,2,1]); v
s1*s2*s1
sage: PW0(v)
s1*s2*s1
sage: WF(v)
S1*S2*S1
sage: W = E.affine_weyl(); W
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
sage: w = W.from_reduced_word([2,1,0]); w
S2*S1*S0
sage: WF(w)
S2*S1*S0
sage: PW0(w)
t[Lambdacheck[1] - 2*Lambdacheck[2]] * s1
```

```
>>> from sage.all import *
>>> W0 = E.classical_weyl(); W0
Weyl Group of type ['A', 2] (as a matrix group acting on the coweight lattice)
>>> v = W0.from_reduced_word([Integer(1),Integer(2),Integer(1)]); v
s1*s2*s1
```

(continues on next page)

(continued from previous page)

```
>>> PW0(v)
s1*s2*s1
>>> WF(v)
S1*S2*S1
>>> W = E.affine_weyl(); W
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
>>> w = W.from_reduced_word([Integer(2),Integer(1),Integer(0)]); w
S2*S1*S0
>>> WF(w)
S2*S1*S0
>>> PW0(w)
t[Lambdacheck[1] - 2*Lambdacheck[2]] * s1
```

Note that for untwisted affine type, the dual form of the classical Weyl group is isomorphic to the usual one, but acts on a different lattice and is therefore different to sage:

```
sage: W0v = E.dual_classical_weyl(); W0v
Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
sage: v = W0v.from_reduced_word([1,2])
sage: x = PvW0(v); x
s1*s2
sage: y = PW0(v); y
s1*s2
sage: x.parent() == y.parent()
False
```

```
>>> from sage.all import *
>>> W0v = E.dual_classical_weyl(); W0v
Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
>>> v = W0v.from_reduced_word([Integer(1),Integer(2)])
>>> x = PvW0(v); x
s1*s2
>>> y = PW0(v); y
s1*s2
>>> x.parent() == y.parent()
False
```

However, because there is a coercion from `PvW0` to `PW0`, the elements `x` and `y` compare as equal:

```
sage: x == y
True
```

```
>>> from sage.all import *
>>> x == y
True
```

An element can be created directly from a reduced word:

```
sage: PW0.from_reduced_word([2,1,0])
t[Lambdacheck[1] - 2*Lambdacheck[2]] * s1
```

```
>>> from sage.all import *
>>> PW0.from_reduced_word([Integer(2),Integer(1),Integer(0)])
t[Lambdacheck[1] - 2*Lambdacheck[2]] * s1
```

Here is a demonstration of the printing options:

```
sage: E = ExtendedAffineWeylGroup(["A",2,1], affine='sx', classical='Sx',
....:                               translation='x', fundamental='pix')
sage: PW0 = E.PW0()
sage: y = PW0(E.lattice_basis()[1]); y
x[Lambdacheck[1]]
sage: FW = E.FW()
sage: FW(y)
pix[1] * sx2*sx1
sage: PW0.an_element()
x[2*Lambdacheck[1] + 2*Lambdacheck[2]] * Sx1*Sx2
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(["A",Integer(2),Integer(1)], affine='sx',␣
↪classical='Sx',
...                               translation='x', fundamental='pix')
>>> PW0 = E.PW0()
>>> y = PW0(E.lattice_basis()[Integer(1)]); y
x[Lambdacheck[1]]
>>> FW = E.FW()
>>> FW(y)
pix[1] * sx2*sx1
>>> PW0.an_element()
x[2*Lambdacheck[1] + 2*Lambdacheck[2]] * Sx1*Sx2
```

Todo

- Implement a “slow” action of E on any affine root or weight lattice realization.
- Implement the level m actions of E and W on the lattices of finite type.
- Implement the relevant methods from the usual affine Weyl group
- Implementation by matrices: style “M”.
- Use case: implement the Hecke algebra on top of this

The semidirect product construction in sage currently only admits multiplicative groups. Therefore for the styles involving “P” and “Pv”, one must convert the additive group of translations L into a multiplicative group by applying the `sage.groups.group_exp.GroupExp` functor.

The general linear case

The general linear group is not semisimple. Sage can build its extended affine Weyl group:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], general_linear=True); E
Extended affine Weyl group of GL(3)
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], general_linear=True);
↪ E
Extended affine Weyl group of GL(3)
```

If the Cartan type is `['A', n-1, 1]` and the parameter `general_linear` is not True, the extended affine Weyl group that is built will be for SL_n, not GL_n. But if `general_linear` is True, let W_a and W_e be the affine and extended affine Weyl groups. We make the following nonstandard definition: the extended affine Weyl group $W_e(GL_n)$ is defined by

$$W_e(GL_n) = P(GL_n) \rtimes W$$

where W is the finite Weyl group (the symmetric group S_n) and $P(GL_n)$ is the weight lattice of GL_n, which is usually identified with the lattice $\mathbf{Z}^n$ of n-tuples of integers:

```
sage: PW0 = E.PW0(); PW0
Extended affine Weyl group of GL(3) realized by Semidirect product of
 Multiplicative form of Ambient space of the Root system of type ['A', 2] acted␣
↪upon
  by Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)
sage: PW0.an_element()
t[(2, 2, 3)] * s1*s2
```

```
>>> from sage.all import *
>>> PW0 = E.PW0(); PW0
Extended affine Weyl group of GL(3) realized by Semidirect product of
 Multiplicative form of Ambient space of the Root system of type ['A', 2] acted␣
↪upon
  by Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)
>>> PW0.an_element()
t[(2, 2, 3)] * s1*s2
```

There is an isomorphism

$$W_e(GL_n) = \mathbf{Z} \ltimes W_a$$

where the group of integers $\mathbf{Z}$ (with generator π) acts on W_a by

$$\pi\, s_i\, \pi^{-1} = s_{i+1}$$

and the indices of the simple reflections are taken modulo n:

```
sage: FW = E.FW(); FW
Extended affine Weyl group of GL(3) realized by
 Semidirect product of Fundamental group of GL(3) acting on
  Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
sage: FW.an_element()
pi[5] * S0*S1*S2
```

```
>>> from sage.all import *
>>> FW = E.FW(); FW
Extended affine Weyl group of GL(3) realized by
 Semidirect product of Fundamental group of GL(3) acting on
```

(continues on next page)

(continued from previous page)

```
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
>>> FW.an_element()
pi[5] * S0*S1*S2
```

We regard **Z** as the fundamental group of affine type GL_n:

```
sage: F = E.fundamental_group(); F
Fundamental group of GL(3)
sage: F.special_nodes()
Integer Ring

sage: x = FW.from_fundamental(F(10)); x
pi[10]
sage: x*x
pi[20]
sage: E.PvW0()(x*x)
t[(7, 7, 6)] * s2*s1
```

```
>>> from sage.all import *
>>> F = E.fundamental_group(); F
Fundamental group of GL(3)
>>> F.special_nodes()
Integer Ring

>>> x = FW.from_fundamental(F(Integer(10))); x
pi[10]
>>> x*x
pi[20]
>>> E.PvW0()(x*x)
t[(7, 7, 6)] * s2*s1
```

class sage.combinat.root_system.extended_affine_weyl_group.**ExtendedAffineWeylGroup_Class**(*cartan_type*, *general_linear*, ***print_options*)

Bases: UniqueRepresentation, Parent

The parent-with-realization class of an extended affine Weyl group.

class ExtendedAffineWeylGroupFW(E)

Bases: GroupSemidirectProduct, BindableClass

Extended affine Weyl group, realized as the semidirect product of the affine Weyl group by the fundamental group.

INPUT:

- E – a parent with realization in *ExtendedAffineWeylGroup_Class*

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).FW()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Fundamental group of type ['A', 2, 1] acting on Weyl Group of type ['A', 2,␣
↪1]
 (as a matrix group acting on the root lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).FW()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Fundamental group of type ['A', 2, 1] acting on Weyl Group of type ['A', 2,␣
↪1]
 (as a matrix group acting on the root lattice)
```

Element

alias of *ExtendedAffineWeylGroupFWElement*

from_affine_weyl(*w*)

Return the image of w under the map of the affine Weyl group into the right (affine Weyl group) factor in the "FW" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], print_tuple=True)
sage: E.FW().from_affine_weyl(E.affine_weyl().from_reduced_word([0,2,1]))
(pi[0], S0*S2*S1)
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], print_
↪tuple=True)
>>> E.FW().from_affine_weyl(E.affine_weyl().from_reduced_word([Integer(0),
↪Integer(2),Integer(1)]))
(pi[0], S0*S2*S1)
```

from_fundamental(*f*)

Return the image of the fundamental group element f into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], print_tuple=True)
sage: E.FW().from_fundamental(E.fundamental_group()(2))
(pi[2], 1)
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], print_
↪tuple=True)
>>> E.FW().from_fundamental(E.fundamental_group()(Integer(2)))
(pi[2], 1)
```

simple_reflections()

Return the family of simple reflections of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1],print_tuple=True).FW().simple_
↪reflections()
Finite family {0: (pi[0], S0), 1: (pi[0], S1), 2: (pi[0], S2)}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],print_tuple=True).
↪FW().simple_reflections()
Finite family {0: (pi[0], S0), 1: (pi[0], S1), 2: (pi[0], S2)}
```

class ExtendedAffineWeylGroupFWElement

Bases: `GroupSemidirectProductElement`

The element class for the "FW" realization.

action_on_affine_roots(*beta*)

Act by `self` on the affine root lattice element `beta`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], affine='s')
sage: x = E.FW().an_element(); x
pi[2] * s0*s1*s2
sage: v = RootSystem(['A',2,1]).root_lattice().an_element(); v
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
sage: x.action_on_affine_roots(v)
alpha[0] + alpha[1]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], affine='s')
>>> x = E.FW().an_element(); x
pi[2] * s0*s1*s2
>>> v = RootSystem(['A',Integer(2),Integer(1)]).root_lattice().an_
↪element(); v
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
>>> x.action_on_affine_roots(v)
alpha[0] + alpha[1]
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has descent at i.

INPUT:

- `i` – an affine Dynkin index

OPTIONAL:

- `side` – `'left'` or `'right'` (default: `'right'`)
- `positive` – boolean (default: `False`)

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.FW().an_element(); x
pi[2] * S0*S1*S2
sage: [(i, x.has_descent(i)) for i in E.cartan_type().index_set()]
[(0, False), (1, False), (2, True)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> x = E.FW().an_element(); x
pi[2] * S0*S1*S2
>>> [(i, x.has_descent(i)) for i in E.cartan_type().index_set()]
[(0, False), (1, False), (2, True)]
```

to_affine_weyl_right()

Project `self` to the right (affine Weyl group) factor in the "FW" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.FW().from_translation(E.lattice_basis()[1]); x
pi[1] * S2*S1
sage: x.to_affine_weyl_right()
S2*S1
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> x = E.FW().from_translation(E.lattice_basis()[Integer(1)]); x
pi[1] * S2*S1
>>> x.to_affine_weyl_right()
S2*S1
```

to_fundamental_group()

Return the projection of `self` to the fundamental group in the "FW" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.FW().from_translation(E.lattice_basis()[2]); x
pi[2] * S1*S2
sage: x.to_fundamental_group()
pi[2]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> x = E.FW().from_translation(E.lattice_basis()[Integer(2)]); x
pi[2] * S1*S2
>>> x.to_fundamental_group()
pi[2]
```

class ExtendedAffineWeylGroupPW0(*E*)

Bases: `GroupSemidirectProduct`, `BindableClass`

Extended affine Weyl group, realized as the semidirect product of the translation lattice by the finite Weyl group.

INPUT:

- `E` – a parent with realization in *ExtendedAffineWeylGroup_Class*

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
  acted upon by Weyl Group of type ['A', 2]
   (as a matrix group acting on the coweight lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
  acted upon by Weyl Group of type ['A', 2]
   (as a matrix group acting on the coweight lattice)
```

Element

alias of *ExtendedAffineWeylGroupPW0Element*

S0()

Return the affine simple reflection.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['B',2]).PW0().S0()
t[Lambdacheck[2]] * s2*s1*s2
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['B',Integer(2)]).PW0().S0()
t[Lambdacheck[2]] * s2*s1*s2
```

from_classical_weyl(*w*)

Return the image of w under the homomorphism of the classical Weyl group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup("A3",print_tuple=True)
sage: E.PW0().from_classical_weyl(E.classical_weyl().from_reduced_word([1,
↪2]))
(t[0], s1*s2)
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup("A3",print_tuple=True)
>>> E.PW0().from_classical_weyl(E.classical_weyl().from_reduced_
↪word([Integer(1),Integer(2)]))
(t[0], s1*s2)
```

from_translation(*la*)

Map the translation lattice element `la` into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], translation='tau',
....:                             print_tuple=True)
```

(continues on next page)

(continued from previous page)

```
sage: la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
sage: E.PW0().from_translation(la)
(tau[2*Lambdacheck[1] + 2*Lambdacheck[2]], 1)
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], translation=
↪'tau',
...                             print_tuple=True)
>>> la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
>>> E.PW0().from_translation(la)
(tau[2*Lambdacheck[1] + 2*Lambdacheck[2]], 1)
```

simple_reflection(*i*)

Return the i-th simple reflection in `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup("G2")
sage: [(i, E.PW0().simple_reflection(i)) for i in E.cartan_type().index_
↪set()]
[(0, t[Lambdacheck[2]] * s2*s1*s2*s1*s2), (1, s1), (2, s2)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup("G2")
>>> [(i, E.PW0().simple_reflection(i)) for i in E.cartan_type().index_
↪set()]
[(0, t[Lambdacheck[2]] * s2*s1*s2*s1*s2), (1, s1), (2, s2)]
```

simple_reflections()

Return a family for the simple reflections of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup("A3").PW0().simple_reflections()
Finite family {0: t[Lambdacheck[1] + Lambdacheck[3]] * s1*s2*s3*s2*s1,
               1: s1, 2: s2, 3: s3}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup("A3").PW0().simple_reflections()
Finite family {0: t[Lambdacheck[1] + Lambdacheck[3]] * s1*s2*s3*s2*s1,
               1: s1, 2: s2, 3: s3}
```

class ExtendedAffineWeylGroupPW0Element

Bases: `GroupSemidirectProductElement`

The element class for the "PW0" realization.

action(*la*)

Return the action of `self` on an element `la` of the translation lattice.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); PW0 = E.PW0()
sage: x = PW0.an_element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
sage: la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
sage: x.action(la)
-2*Lambdacheck[1] + 4*Lambdacheck[2]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]); PW0 = E.
→PW0()
>>> x = PW0.an_element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
>>> la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
>>> x.action(la)
-2*Lambdacheck[1] + 4*Lambdacheck[2]
```

has_descent (*i*, *side='right'*, *positive=False*)

Return whether `self` has i as a descent.

INPUT:

- `i` – an affine Dynkin node

OPTIONAL:

- `side` – `'left'` or `'right'` (default: `'right'`)
- `positive` – boolean (default: `False`)

EXAMPLES:

```
sage: w = ExtendedAffineWeylGroup(['A',4,2]).PW0().from_reduced_word([0,
→1]); w
t[Lambda[1]] * s1*s2
sage: w.has_descent(0, side='left')
True
```

```
>>> from sage.all import *
>>> w = ExtendedAffineWeylGroup(['A',Integer(4),Integer(2)]).PW0().from_
→reduced_word([Integer(0),Integer(1)]); w
t[Lambda[1]] * s1*s2
>>> w.has_descent(Integer(0), side='left')
True
```

to_classical_weyl ()

Return the image of `self` under the homomorphism that projects to the classical Weyl group factor after rewriting it in either style "PW0" or "W0P".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).PW0().S0(); s
t[Lambdacheck[1] + Lambdacheck[2]] * s1*s2*s1
sage: s.to_classical_weyl()
s1*s2*s1
```

```
>>> from sage.all import *
>>> s = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PW0().S0(); s
t[Lambdacheck[1] + Lambdacheck[2]] * s1*s2*s1
>>> s.to_classical_weyl()
s1*s2*s1
```

to_translation_left()

The image of `self` under the map that projects to the translation lattice factor after factoring it to the left as in style "PW0".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).PW0().S0(); s
t[Lambdacheck[1] + Lambdacheck[2]] * s1*s2*s1
sage: s.to_translation_left()
Lambdacheck[1] + Lambdacheck[2]
```

```
>>> from sage.all import *
>>> s = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PW0().S0(); s
t[Lambdacheck[1] + Lambdacheck[2]] * s1*s2*s1
>>> s.to_translation_left()
Lambdacheck[1] + Lambdacheck[2]
```

class ExtendedAffineWeylGroupPvW0(E)

Bases: `GroupSemidirectProduct`, `BindableClass`

Extended affine Weyl group, realized as the semidirect product of the dual form of the translation lattice by the finite Weyl group.

INPUT:

- `E` – a parent with realization in *ExtendedAffineWeylGroup_Class*

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PvW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Multiplicative form of Weight lattice of the Root system of type ['A', 2]␣
↪acted
  upon by Weyl Group of type ['A', 2] (as a matrix group acting on the weight␣
↪lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PvW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Multiplicative form of Weight lattice of the Root system of type ['A', 2]␣
↪acted
  upon by Weyl Group of type ['A', 2] (as a matrix group acting on the weight␣
↪lattice)
```

Element

alias of *ExtendedAffineWeylGroupPvW0Element*

from_dual_classical_weyl(*w*)

Return the image of w under the homomorphism of the dual form of the classical Weyl group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1], print_tuple=True)
sage: E.PvW0().from_dual_classical_weyl(
....:     E.dual_classical_weyl().from_reduced_word([1,2]))
(t[0], s1*s2)
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)], print_
↪tuple=True)
>>> E.PvW0().from_dual_classical_weyl(
...     E.dual_classical_weyl().from_reduced_word([Integer(1),
↪Integer(2)]))
(t[0], s1*s2)
```

from_dual_translation(*la*)

Map the dual translation lattice element `la` into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], translation='tau',
....:                             print_tuple=True)
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: E.PvW0().from_dual_translation(la)
(tau[2*Lambda[1] + 2*Lambda[2]], 1)
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], translation=
↪'tau',
...                             print_tuple=True)
>>> la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
>>> E.PvW0().from_dual_translation(la)
(tau[2*Lambda[1] + 2*Lambda[2]], 1)
```

simple_reflections()

Return a family for the simple reflections of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PvW0().simple_reflections()
Finite family {0: t[Lambda[1] + Lambda[3]] * s1*s2*s3*s2*s1,
               1: s1, 2: s2, 3: s3}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).PvW0().simple_
↪reflections()
Finite family {0: t[Lambda[1] + Lambda[3]] * s1*s2*s3*s2*s1,
               1: s1, 2: s2, 3: s3}
```

class ExtendedAffineWeylGroupPvW0Element

Bases: `GroupSemidirectProductElement`

The element class for the "PvW0" realization.

dual_action(*la*)

Return the action of `self` on an element `la` of the dual version of the translation lattice.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.PvW0().an_element(); x
t[2*Lambda[1] + 2*Lambda[2]] * s1*s2
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: x.dual_action(la)
-2*Lambda[1] + 4*Lambda[2]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> x = E.PvW0().an_element(); x
t[2*Lambda[1] + 2*Lambda[2]] * s1*s2
>>> la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
>>> x.dual_action(la)
-2*Lambda[1] + 4*Lambda[2]
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has i as a descent.

INPUT:

- `i` – an affine Dynkin index

OPTIONAL:

- `side` – `'left'` or `'right'` (default: `'right'`)
- `positive` – boolean (default: `False`)

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',4,2])
sage: w = E.PvW0().from_reduced_word([0,1]); w
t[Lambda[1]] * s1*s2
sage: [(i, w.has_descent(i, side='left')) for i in E.cartan_type().index_
↪set()]
[(0, True), (1, False), (2, False)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(4),Integer(2)])
>>> w = E.PvW0().from_reduced_word([Integer(0),Integer(1)]); w
t[Lambda[1]] * s1*s2
>>> [(i, w.has_descent(i, side='left')) for i in E.cartan_type().index_
↪set()]
[(0, True), (1, False), (2, False)]
```

to_dual_classical_weyl()

Return the image of `self` under the homomorphism that projects to the dual classical Weyl group factor after rewriting it in either style "PvW0" or "W0Pv".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).PvW0().simple_reflection(0);␣
↪s
t[Lambda[1] + Lambda[2]] * s1*s2*s1
sage: s.to_dual_classical_weyl()
s1*s2*s1
```

```
>>> from sage.all import *
>>> s = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PvW0().
↪simple_reflection(Integer(0)); s
t[Lambda[1] + Lambda[2]] * s1*s2*s1
>>> s.to_dual_classical_weyl()
s1*s2*s1
```

to_dual_translation_left()

The image of `self` under the map that projects to the dual translation lattice factor after factoring it to the left as in style "PvW0".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).PvW0().simple_reflection(0);␣
↪s
t[Lambda[1] + Lambda[2]] * s1*s2*s1
sage: s.to_dual_translation_left()
Lambda[1] + Lambda[2]
```

```
>>> from sage.all import *
>>> s = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PvW0().
↪simple_reflection(Integer(0)); s
t[Lambda[1] + Lambda[2]] * s1*s2*s1
>>> s.to_dual_translation_left()
Lambda[1] + Lambda[2]
```

class ExtendedAffineWeylGroupW0P(E)

Bases: `GroupSemidirectProduct`, `BindableClass`

Extended affine Weyl group, realized as the semidirect product of the finite Weyl group by the translation lattice.

INPUT:

- `E` – a parent with realization in *ExtendedAffineWeylGroup_Class*

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).W0P()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2] (as a matrix group acting on the coweight␣
↪lattice)
 acting on Multiplicative form of Coweight lattice of the Root system of type␣
↪['A', 2]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).W0P()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2] (as a matrix group acting on the coweight␣
↪lattice)
 acting on Multiplicative form of Coweight lattice of the Root system of type␣
↪['A', 2]
```

Element

alias of *ExtendedAffineWeylGroupW0PElement*

S0()

Return the zero-th simple reflection in style "W0P".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(["A",3,1]).W0P().S0()
s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(["A",Integer(3),Integer(1)]).W0P().S0()
s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]]
```

from_classical_weyl(*w*)

Return the image of the classical Weyl group element w in `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],print_tuple=True)
sage: E.W0P().from_classical_weyl(E.classical_weyl().from_reduced_word([2,
↪1]))
(s2*s1, t[0])
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],print_
↪tuple=True)
>>> E.W0P().from_classical_weyl(E.classical_weyl().from_reduced_
↪word([Integer(2),Integer(1)]))
(s2*s1, t[0])
```

from_translation(*la*)

Return the image of the lattice element `la` in `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],print_tuple=True)
sage: E.W0P().from_translation(E.lattice().an_element())
(1, t[2*Lambdacheck[1] + 2*Lambdacheck[2]])
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],print_
↪tuple=True)
```

(continues on next page)

(continued from previous page)

```
>>> E.W0P().from_translation(E.lattice().an_element())
(1, t[2*Lambdacheck[1] + 2*Lambdacheck[2]])
```

simple_reflection(*i*)

Return the i-th simple reflection in `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); W0P = E.W0P()
sage: [(i, W0P.simple_reflection(i)) for i in E.cartan_type().index_set()]
[(0, s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]]),
 (1, s1), (2, s2), (3, s3)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); W0P = E.
↪W0P()
>>> [(i, W0P.simple_reflection(i)) for i in E.cartan_type().index_set()]
[(0, s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]]),
 (1, s1), (2, s2), (3, s3)]
```

simple_reflections()

Return the family of simple reflections.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(["A",3,1]).W0P().simple_reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]],
               1: s1, 2: s2, 3: s3}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(["A",Integer(3),Integer(1)]).W0P().simple_
↪reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]],
               1: s1, 2: s2, 3: s3}
```

class ExtendedAffineWeylGroupW0PElement

Bases: `GroupSemidirectProductElement`

The element class for the W0P realization.

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has i as a descent.

INPUT:

- `i` – an index

OPTIONAL:

- `side` – `'left'` or `'right'` (default: `'right'`)
- `positive` – boolean (default: `False`)

EXAMPLES:

```
sage: W0P = ExtendedAffineWeylGroup(['A',4,2]).W0P()
sage: w = W0P.from_reduced_word([0,1]); w
s1*s2 * t[Lambda[1] - Lambda[2]]
```

(continues on next page)

(continued from previous page)

```
sage: w.has_descent(0, side='left')
True
```

```
>>> from sage.all import *
>>> W0P = ExtendedAffineWeylGroup(['A',Integer(4),Integer(2)]).W0P()
>>> w = W0P.from_reduced_word([Integer(0),Integer(1)]); w
s1*s2 * t[Lambda[1] - Lambda[2]]
>>> w.has_descent(Integer(0), side='left')
True
```

to_classical_weyl()

Project self into the classical Weyl group.

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',2,1]).W0P().simple_reflection(0); x
s1*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[2]]
sage: x.to_classical_weyl()
s1*s2*s1
```

```
>>> from sage.all import *
>>> x = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).W0P().simple_
↪reflection(Integer(0)); x
s1*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[2]]
>>> x.to_classical_weyl()
s1*s2*s1
```

to_translation_right()

Project onto the right (translation) factor in the "W0P" style.

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',2,1]).W0P().simple_reflection(0); x
s1*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[2]]
sage: x.to_translation_right()
-Lambdacheck[1] - Lambdacheck[2]
```

```
>>> from sage.all import *
>>> x = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).W0P().simple_
↪reflection(Integer(0)); x
s1*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[2]]
>>> x.to_translation_right()
-Lambdacheck[1] - Lambdacheck[2]
```

class ExtendedAffineWeylGroupW0Pv(E)

Bases: GroupSemidirectProduct, BindableClass

Extended affine Weyl group, realized as the semidirect product of the finite Weyl group, acting on the dual form of the translation lattice.

INPUT:

- E – a parent with realization in *ExtendedAffineWeylGroup_Class*

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).W0Pv()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
 acting on Multiplicative form of Weight lattice of the Root system of type [
↪'A', 2]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).W0Pv()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
 acting on Multiplicative form of Weight lattice of the Root system of type [
↪'A', 2]
```

Element

alias of *ExtendedAffineWeylGroupW0PvElement*

from_dual_classical_weyl(*w*)

Return the image of w under the homomorphism of the dual form of the classical Weyl group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1],print_tuple=True)
sage: E.W0Pv().from_dual_classical_weyl(E.dual_classical_weyl().from_
↪reduced_word([1,2]))
(s1*s2, t[0])
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)],print_
↪tuple=True)
>>> E.W0Pv().from_dual_classical_weyl(E.dual_classical_weyl().from_
↪reduced_word([Integer(1),Integer(2)]))
(s1*s2, t[0])
```

from_dual_translation(*la*)

Map the dual translation lattice element `la` into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], translation='tau',
....:                             print_tuple=True)
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: E.W0Pv().from_dual_translation(la)
(1, tau[2*Lambda[1] + 2*Lambda[2]])
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], translation=
↪'tau',
...                             print_tuple=True)
>>> la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
```

(continues on next page)

(continued from previous page)

```
>>> E.W0Pv().from_dual_translation(la)
(1, tau[2*Lambda[1] + 2*Lambda[2]])
```

simple_reflections()

Return a family for the simple reflections of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).W0Pv().simple_reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambda[1] - Lambda[3]],
               1: s1, 2: s2, 3: s3}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).W0Pv().simple_
↪reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambda[1] - Lambda[3]],
               1: s1, 2: s2, 3: s3}
```

class ExtendedAffineWeylGroupW0PvElement

Bases: `GroupSemidirectProductElement`

The element class for the "W0Pv" realization.

dual_action(*la*)

Return the action of `self` on an element `la` of the dual version of the translation lattice.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.W0Pv().an_element(); x
s1*s2 * t[2*Lambda[1] + 2*Lambda[2]]
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: x.dual_action(la)
-8*Lambda[1] + 4*Lambda[2]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> x = E.W0Pv().an_element(); x
s1*s2 * t[2*Lambda[1] + 2*Lambda[2]]
>>> la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
>>> x.dual_action(la)
-8*Lambda[1] + 4*Lambda[2]
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has i as a descent.

INPUT:

- `i` – an affine Dynkin index

OPTIONAL:

- `side` – `'left'` or `'right'` (default: `'right'`)
- `positive` – boolean (default: `False`)

EXAMPLES:

```
sage: w = ExtendedAffineWeylGroup(['A',4,2]).W0Pv().from_reduced_word([0,
↪1]); w
s1*s2 * t[Lambda[1] - Lambda[2]]
sage: w.has_descent(0, side='left')
True
```

```
>>> from sage.all import *
>>> w = ExtendedAffineWeylGroup(['A',Integer(4),Integer(2)]).W0Pv().from_
↪reduced_word([Integer(0),Integer(1)]); w
s1*s2 * t[Lambda[1] - Lambda[2]]
>>> w.has_descent(Integer(0), side='left')
True
```

to_dual_classical_weyl()

Return the image of `self` under the homomorphism that projects to the dual classical Weyl group factor after rewriting it in either style "PvW0" or "W0Pv".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).W0Pv().simple_reflection(0);
↪s
s1*s2*s1 * t[-Lambda[1] - Lambda[2]]
sage: s.to_dual_classical_weyl()
s1*s2*s1
```

```
>>> from sage.all import *
>>> s = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).W0Pv().
↪simple_reflection(Integer(0)); s
s1*s2*s1 * t[-Lambda[1] - Lambda[2]]
>>> s.to_dual_classical_weyl()
s1*s2*s1
```

to_dual_translation_right()

The image of `self` under the map that projects to the dual translation lattice factor after factoring it to the right as in style "W0Pv".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).W0Pv().simple_reflection(0);
↪s
s1*s2*s1 * t[-Lambda[1] - Lambda[2]]
sage: s.to_dual_translation_right()
-Lambda[1] - Lambda[2]
```

```
>>> from sage.all import *
>>> s = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).W0Pv().
↪simple_reflection(Integer(0)); s
s1*s2*s1 * t[-Lambda[1] - Lambda[2]]
>>> s.to_dual_translation_right()
-Lambda[1] - Lambda[2]
```

class ExtendedAffineWeylGroupWF(E)

Bases: `GroupSemidirectProduct`, `BindableClass`

Extended affine Weyl group, realized as the semidirect product of the affine Weyl group by the fundamental group.

INPUT:

- E – a parent with realization in *ExtendedAffineWeylGroup_Class*

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).WF()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
  acted upon by Fundamental group of type ['A', 2, 1]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).WF()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
  acted upon by Fundamental group of type ['A', 2, 1]
```

Element

alias of *ExtendedAffineWeylGroupWFElement*

from_affine_weyl(w)

Return the image of the affine Weyl group element w in `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['C',2,1], print_tuple=True)
sage: E.WF().from_affine_weyl(E.affine_weyl().from_reduced_word([1,2,1,
↪0]))
(S1*S2*S1*S0, pi[0])
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['C',Integer(2),Integer(1)], print_
↪tuple=True)
>>> E.WF().from_affine_weyl(E.affine_weyl().from_reduced_word([Integer(1),
↪Integer(2),Integer(1),Integer(0)]))
(S1*S2*S1*S0, pi[0])
```

from_fundamental(f)

Return the image of f under the homomorphism from the fundamental group into the right (fundamental group) factor in "WF" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['E',6,1], print_tuple=True); WF = E.
↪WF()
sage: F = E.fundamental_group()
sage: [(x, WF.from_fundamental(x)) for x in F]
[(pi[0], (1, pi[0])), (pi[1], (1, pi[1])), (pi[6], (1, pi[6]))]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['E',Integer(6),Integer(1)], print_
↪tuple=True); WF = E.WF()
>>> F = E.fundamental_group()
>>> [(x, WF.from_fundamental(x)) for x in F]
[(pi[0], (1, pi[0])), (pi[1], (1, pi[1])), (pi[6], (1, pi[6]))]
```

simple_reflections()

Return the family of simple reflections.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(["A",3,1], affine='r').WF().simple_
↪reflections()
Finite family {0: r0, 1: r1, 2: r2, 3: r3}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(["A",Integer(3),Integer(1)], affine='r').WF().
↪simple_reflections()
Finite family {0: r0, 1: r1, 2: r2, 3: r3}
```

class ExtendedAffineWeylGroupWFElement

Bases: `GroupSemidirectProductElement`

Element class for the "WF" realization.

bruhat_le(x)

Return whether `self` is less than or equal to x in the Bruhat order.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], affine='s',
....:                             print_tuple=True); WF = E.WF()
sage: r = E.affine_weyl().from_reduced_word
sage: v = r([1,0])
sage: w = r([1,2,0])
sage: v.bruhat_le(w)
True
sage: vv = WF.from_affine_weyl(v); vv
(s1*s0, pi[0])
sage: ww = WF.from_affine_weyl(w); ww
(s1*s2*s0, pi[0])
sage: vv.bruhat_le(ww)
True
sage: f = E.fundamental_group()(2); f
pi[2]
sage: ff = WF.from_fundamental(f); ff
(1, pi[2])
sage: vv.bruhat_le(ww*ff)
False
sage: (vv*ff).bruhat_le(ww*ff)
True
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], affine='s',
...                             print_tuple=True); WF = E.WF()
>>> r = E.affine_weyl().from_reduced_word
>>> v = r([Integer(1),Integer(0)])
>>> w = r([Integer(1),Integer(2),Integer(0)])
>>> v.bruhat_le(w)
True
>>> vv = WF.from_affine_weyl(v); vv
(s1*s0, pi[0])
>>> ww = WF.from_affine_weyl(w); ww
(s1*s2*s0, pi[0])
>>> vv.bruhat_le(ww)
True
>>> f = E.fundamental_group()(Integer(2)); f
pi[2]
>>> ff = WF.from_fundamental(f); ff
(1, pi[2])
>>> vv.bruhat_le(ww*ff)
False
>>> (vv*ff).bruhat_le(ww*ff)
True
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has i as a descent.

INPUT:

- `i` – an affine Dynkin index

OPTIONAL:

- `side` – `'left'` or `'right'` (default: `'right'`)
- `positive` – boolean (default: `False`)

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.WF().an_element(); x
S0*S1*S2 * pi[2]
sage: [(i, x.has_descent(i)) for i in E.cartan_type().index_set()]
[(0, True), (1, False), (2, False)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> x = E.WF().an_element(); x
S0*S1*S2 * pi[2]
>>> [(i, x.has_descent(i)) for i in E.cartan_type().index_set()]
[(0, True), (1, False), (2, False)]
```

to_affine_weyl_left()

Project `self` to the left (affine Weyl group) factor in the “WF” style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.WF().from_translation(E.lattice_basis()[1]); x
```

(continues on next page)

(continued from previous page)

```
S0*S2 * pi[1]
sage: x.to_affine_weyl_left()
S0*S2
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> x = E.WF().from_translation(E.lattice_basis()[Integer(1)]); x
S0*S2 * pi[1]
>>> x.to_affine_weyl_left()
S0*S2
```

to_fundamental_group()

Project `self` to the right (fundamental group) factor in the "WF" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.WF().from_translation(E.lattice_basis()[1]); x
S0*S2 * pi[1]
sage: x.to_fundamental_group()
pi[1]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> x = E.WF().from_translation(E.lattice_basis()[Integer(1)]); x
S0*S2 * pi[1]
>>> x.to_fundamental_group()
pi[1]
```

FW()

Realizes `self` in "FW"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).FW()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Fundamental group of type ['A', 2, 1] acting on
  Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root␣
↪lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).FW()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Fundamental group of type ['A', 2, 1] acting on
  Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root␣
↪lattice)
```

PW0()

Realizes `self` in "PW0"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PW0()
Extended affine Weyl group of type ['A', 2, 1] realized by
 Semidirect product of Multiplicative form of
  Coweight lattice of the Root system of type ['A', 2] acted upon by
   Weyl Group of type ['A', 2] (as a matrix group acting on the coweight␣
↪lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PW0()
Extended affine Weyl group of type ['A', 2, 1] realized by
 Semidirect product of Multiplicative form of
  Coweight lattice of the Root system of type ['A', 2] acted upon by
   Weyl Group of type ['A', 2] (as a matrix group acting on the coweight␣
↪lattice)
```

PW0_to_WF_func(*x*)

Implement coercion from style "PW0" to "WF".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(["A", 2, 1])
sage: x = E.PW0().an_element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
sage: E.PW0_to_WF_func(x)
S0*S1*S2*S0*S1*S0
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(["A", Integer(2), Integer(1)])
>>> x = E.PW0().an_element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
>>> E.PW0_to_WF_func(x)
S0*S1*S2*S0*S1*S0
```

Warning

This function cannot use coercion, because it is used to define the coercion maps.

PvW0()

Realizes `self` in "PvW0"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PvW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Multiplicative form of Weight lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2] (as a matrix group acting on the␣
↪weight lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PvW0()
```

(continues on next page)

(continued from previous page)

```
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Multiplicative form of Weight lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2] (as a matrix group acting on the␣
↪weight lattice)
```

class Realizations(*parent_with_realization*)

Bases: `Category_realization_of_parent`

The category of the realizations of an extended affine Weyl group

class ElementMethods

Bases: `object`

action(*la*)

Action of `self` on a lattice element `la`.

INPUT:

- `self` – an element of the extended affine Weyl group
- `la` – an element of the translation lattice of the extended affine Weyl group, the lattice denoted by the mnemonic "P" in the documentation for `ExtendedAffineWeylGroup()`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], affine='s')
sage: x = E.FW().an_element(); x
pi[2] * s0*s1*s2
sage: la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
sage: x.action(la)
5*Lambdacheck[1] - 3*Lambdacheck[2]
sage: E = ExtendedAffineWeylGroup(['C',2,1], affine='s')
sage: x = E.PW0().from_translation(E.lattice_basis()[1])
sage: x.action(E.lattice_basis()[2])
Lambdacheck[1] + Lambdacheck[2]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], affine='s
↪')
>>> x = E.FW().an_element(); x
pi[2] * s0*s1*s2
>>> la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
>>> x.action(la)
5*Lambdacheck[1] - 3*Lambdacheck[2]
>>> E = ExtendedAffineWeylGroup(['C',Integer(2),Integer(1)], affine='s
↪')
>>> x = E.PW0().from_translation(E.lattice_basis()[Integer(1)])
>>> x.action(E.lattice_basis()[Integer(2)])
Lambdacheck[1] + Lambdacheck[2]
```

Warning

Must be implemented by style "PW0".

action_on_affine_roots(*beta*)

Act by `self` on the affine root lattice element `beta`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: beta = E.cartan_type().root_system().root_lattice().an_element();
↪ beta
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
sage: x = E.FW().an_element(); x
pi[2] * S0*S1*S2
sage: x.action_on_affine_roots(beta)
alpha[0] + alpha[1]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> beta = E.cartan_type().root_system().root_lattice().an_element();
↪beta
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
>>> x = E.FW().an_element(); x
pi[2] * S0*S1*S2
>>> x.action_on_affine_roots(beta)
alpha[0] + alpha[1]
```

Warning

Must be implemented by style "FW".

alcove_walk_signs()

Return a signed alcove walk for `self`.

INPUT:

- An element `self` of the extended affine Weyl group.

OUTPUT:

- A 3-tuple (g, `rw`, `signs`).

ALGORITHM:

The element `self` can be uniquely written `self` = $g * w$ where g has length zero and w is an element of the nonextended affine Weyl group. Let w have reduced word `rw`. Starting with g and applying simple reflections from `rw`, one obtains a sequence of extended affine Weyl group elements (that is, alcoves) and simple roots. The signs give the sequence of sides on which the alcoves lie, relative to the face indicated by the simple roots.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); FW=E.FW()
sage: w = FW.from_reduced_word([0,2,1,3,0])*FW.from_fundamental(1); w
pi[1] * S3*S1*S2*S0*S3
sage: w.alcove_walk_signs()
(pi[1], [3, 1, 2, 0, 3], [-1, 1, -1, -1, 1])
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); FW=E.FW()
>>> w = FW.from_reduced_word([Integer(0),Integer(2),Integer(1),
↪Integer(3),Integer(0)])*FW.from_fundamental(Integer(1)); w
pi[1] * S3*S1*S2*S0*S3
>>> w.alcove_walk_signs()
(pi[1], [3, 1, 2, 0, 3], [-1, 1, -1, -1, 1])
```

apply_simple_projection (*i*, *side='right'*, *length_increasing=True*)

Return the product of `self` by the simple reflection s_i if that product is of greater length than `self` and otherwise return `self`.

INPUT:

- `self` – an element of the extended affine Weyl group
- `i` – a Dynkin node (index of a simple reflection s_i)
- `side` – `'right'` or `'left'` (default: `'right'`) according to which side of `self` the reflection s_i should be multiplied
- `length_increasing` – boolean (default: `True`); if `False`, do the above with the word "greater" replaced by "less"

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',3,1]).WF().an_element(); x
S0*S1*S2*S3 * pi[3]
sage: x.apply_simple_projection(1)
S0*S1*S2*S3*S0 * pi[3]
sage: x.apply_simple_projection(1, length_increasing=False)
S0*S1*S2*S3 * pi[3]
```

```
>>> from sage.all import *
>>> x = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).WF().an_
↪element(); x
S0*S1*S2*S3 * pi[3]
>>> x.apply_simple_projection(Integer(1))
S0*S1*S2*S3*S0 * pi[3]
>>> x.apply_simple_projection(Integer(1), length_increasing=False)
S0*S1*S2*S3 * pi[3]
```

apply_simple_reflection (*i*, *side='right'*)

Apply the i-th simple reflection to `self`.

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',3,1]).WF().an_element(); x
S0*S1*S2*S3 * pi[3]
sage: x.apply_simple_reflection(1)
S0*S1*S2*S3*S0 * pi[3]
sage: x.apply_simple_reflection(0, side='left')
S1*S2*S3 * pi[3]
```

```
>>> from sage.all import *
>>> x = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).WF().an_
↪element(); x
S0*S1*S2*S3 * pi[3]
```

(continues on next page)

(continued from previous page)

```
>>> x.apply_simple_reflection(Integer(1))
S0*S1*S2*S3*S0 * pi[3]
>>> x.apply_simple_reflection(Integer(0), side='left')
S1*S2*S3 * pi[3]
```

bruhat_le(x)

Return whether `self` <= x in Bruhat order.

INPUT:

- `self` – an element of the extended affine Weyl group
- `x` – another element with the same parent as `self`

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], print_tuple=True); WF = E.
↪WF()
sage: W = E.affine_weyl()
sage: v = W.from_reduced_word([2,1,0])
sage: w = W.from_reduced_word([2,0,1,0])
sage: v.bruhat_le(w)
True
sage: vx = WF.from_affine_weyl(v); vx
(S2*S1*S0, pi[0])
sage: wx = WF.from_affine_weyl(w); wx
(S2*S0*S1*S0, pi[0])
sage: vx.bruhat_le(wx)
True
sage: F = E.fundamental_group()
sage: f = WF.from_fundamental(F(2))
sage: vx.bruhat_le(wx*f)
False
sage: (vx*f).bruhat_le(wx*f)
True
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], print_
↪tuple=True); WF = E.WF()
>>> W = E.affine_weyl()
>>> v = W.from_reduced_word([Integer(2),Integer(1),Integer(0)])
>>> w = W.from_reduced_word([Integer(2),Integer(0),Integer(1),
↪Integer(0)])
>>> v.bruhat_le(w)
True
>>> vx = WF.from_affine_weyl(v); vx
(S2*S1*S0, pi[0])
>>> wx = WF.from_affine_weyl(w); wx
(S2*S0*S1*S0, pi[0])
>>> vx.bruhat_le(wx)
True
>>> F = E.fundamental_group()
>>> f = WF.from_fundamental(F(Integer(2)))
>>> vx.bruhat_le(wx*f)
False
```

(continues on next page)

(continued from previous page)

```
>>> (vx*f).bruhat_le(wx*f)
True
```

> **Warning**
>
> Must be implemented by "WF".

coset_representative(*index_set*, *side='right'*)

Return the minimum length representative in the coset of `self` with respect to the subgroup generated by the reflections given by `index_set`.

INPUT:

- `self` – an element of the extended affine Weyl group
- `index_set` – a subset of the set of Dynkin nodes
- `side` – `'right'` or `'left'` (default: `'right'`) the side on which the subgroup acts

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); WF = E.WF()
sage: b = E.lattice_basis()
sage: I0 = E.cartan_type().classical().index_set()
sage: [WF.from_translation(x).coset_representative(index_set=I0) for x␣
↪in b]
[pi[1], pi[2], pi[3]]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); WF = E.
↪WF()
>>> b = E.lattice_basis()
>>> I0 = E.cartan_type().classical().index_set()
>>> [WF.from_translation(x).coset_representative(index_set=I0) for x␣
↪in b]
[pi[1], pi[2], pi[3]]
```

dual_action(*la*)

Action of `self` on a dual lattice element `la`.

INPUT:

- `self` – an element of the extended affine Weyl group
- `la` – an element of the dual translation lattice of the extended affine Weyl group, the lattice denoted by the mnemonic "Pv" in the documentation for `ExtendedAffineWeylGroup()`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], affine='s')
sage: x = E.FW().an_element(); x
pi[2] * s0*s1*s2
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: x.dual_action(la)
5*Lambda[1] - 3*Lambda[2]
sage: E = ExtendedAffineWeylGroup(['C',2,1], affine='s')
sage: x = E.PvW0().from_dual_translation(E.dual_lattice_basis()[1])
```

(continues on next page)

(continued from previous page)

```
sage: x.dual_action(E.dual_lattice_basis()[2])
Lambda[1] + Lambda[2]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)], affine='s
↪')
>>> x = E.FW().an_element(); x
pi[2] * s0*s1*s2
>>> la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
>>> x.dual_action(la)
5*Lambda[1] - 3*Lambda[2]
>>> E = ExtendedAffineWeylGroup(['C',Integer(2),Integer(1)], affine='s
↪')
>>> x = E.PvW0().from_dual_translation(E.dual_lattice_
↪basis()[Integer(1)])
>>> x.dual_action(E.dual_lattice_basis()[Integer(2)])
Lambda[1] + Lambda[2]
```

> **Warning**
>
> Must be implemented by style “PvW0”.

face_data(*i*)

Return a description of one of the bounding hyperplanes of the alcove of an extended affine Weyl group element.

INPUT:

- `self` – an element of the extended affine Weyl group
- `i` – an affine Dynkin node

OUTPUT:

- A 2-tuple (m, β) defined as follows.

ALGORITHM:

Each element of the extended affine Weyl group corresponds to an alcove, and each alcove has a face for each affine Dynkin node. Given the data of `self` and i, let the extended affine Weyl group element `self` act on the affine simple root α_i, yielding a real affine root, which can be expressed uniquely as

$$``self`` \cdot \alpha_i = m\delta + \beta$$

where m is an integer (the height of the i-th bounding hyperplane of the alcove of `self`) and β is a classical root (the normal vector for the hyperplane which points towards the alcove).

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',2,1]).PW0().an_element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
sage: x.face_data(0)
(-1, alpha[1])
```

```
>>> from sage.all import *
>>> x = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PW0().an_
↪element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
>>> x.face_data(Integer(0))
(-1, alpha[1])
```

first_descent (*side='right'*, *positive=False*, *index_set=None*)

Return the first descent of `self`.

INPUT:

- `side` – `'left'` or `'right'` (default: `'right'`)
- `positive` – boolean (default: `False`)
- `index_set` – an optional subset of Dynkin nodes

If `index_set` is not `None`, then the descent must be in the `index_set`.

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',3,1]).WF().an_element(); x
S0*S1*S2*S3 * pi[3]
sage: x.first_descent()
0
sage: x.first_descent(side='left')
0
sage: x.first_descent(positive=True)
1
sage: x.first_descent(side='left',positive=True)
1
```

```
>>> from sage.all import *
>>> x = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).WF().an_
↪element(); x
S0*S1*S2*S3 * pi[3]
>>> x.first_descent()
0
>>> x.first_descent(side='left')
0
>>> x.first_descent(positive=True)
1
>>> x.first_descent(side='left',positive=True)
1
```

has_descent (*i*, *side='right'*, *positive=False*)

Return whether `self` * s_i < `self` where s_i is the i-th simple reflection in the realized group.

INPUT:

- `i` – an affine Dynkin index

OPTIONAL:

- `side` – `'right'` or `'left'` (default: `'right'`)
- `positive` – boolean (default: `False`)

If `side='left'`, then the reflection acts on the left. If `positive=True`, then the inequality is reversed.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); WF = E.WF()
sage: F = E.fundamental_group()
sage: x = WF.an_element(); x
S0*S1*S2*S3 * pi[3]
sage: I = E.cartan_type().index_set()
sage: [(i, x.has_descent(i)) for i in I]
[(0, True), (1, False), (2, False), (3, False)]
sage: [(i, x.has_descent(i,side='left')) for i in I]
[(0, True), (1, False), (2, False), (3, False)]
sage: [(i, x.has_descent(i,positive=True)) for i in I]
[(0, False), (1, True), (2, True), (3, True)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); WF = E.
↪WF()
>>> F = E.fundamental_group()
>>> x = WF.an_element(); x
S0*S1*S2*S3 * pi[3]
>>> I = E.cartan_type().index_set()
>>> [(i, x.has_descent(i)) for i in I]
[(0, True), (1, False), (2, False), (3, False)]
>>> [(i, x.has_descent(i,side='left')) for i in I]
[(0, True), (1, False), (2, False), (3, False)]
>>> [(i, x.has_descent(i,positive=True)) for i in I]
[(0, False), (1, True), (2, True), (3, True)]
```

Warning

This method is abstract because it is used in the recursive coercions between "PW0" and "WF" and other methods use this coercion.

is_affine_grassmannian()

Return whether self is affine Grassmannian.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); PW0 = E.PW0()
sage: F = E.fundamental_group()
sage: [(x,PW0.from_fundamental(x).is_affine_grassmannian()) for x in F]
[(pi[0], True), (pi[1], True), (pi[2], True)]
sage: b = E.lattice_basis()
sage: [(-x,PW0.from_translation(-x).is_affine_grassmannian()) for x in␣
↪b]
[(-Lambdacheck[1], True), (-Lambdacheck[2], True)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]); PW0 = E.
↪PW0()
>>> F = E.fundamental_group()
>>> [(x,PW0.from_fundamental(x).is_affine_grassmannian()) for x in F]
[(pi[0], True), (pi[1], True), (pi[2], True)]
```

(continues on next page)

(continued from previous page)

```
>>> b = E.lattice_basis()
>>> [(-x,PW0.from_translation(-x).is_affine_grassmannian()) for x in b]
[(-Lambdacheck[1], True), (-Lambdacheck[2], True)]
```

is_grassmannian(*index_set*, *side='right'*)

Return whether `self` is of minimum length in its coset with respect to the subgroup generated by the reflections of `index_set`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0 = E.PW0()
sage: x = PW0.from_translation(E.lattice_basis()[1]); x
t[Lambdacheck[1]]
sage: I = E.cartan_type().index_set()
sage: [(i, x.is_grassmannian(index_set=[i])) for i in I]
[(0, True), (1, False), (2, True), (3, True)]
sage: [(i, x.is_grassmannian(index_set=[i], side='left')) for i in I]
[(0, False), (1, True), (2, True), (3, True)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); PW0 = E.
↪PW0()
>>> x = PW0.from_translation(E.lattice_basis()[Integer(1)]); x
t[Lambdacheck[1]]
>>> I = E.cartan_type().index_set()
>>> [(i, x.is_grassmannian(index_set=[i])) for i in I]
[(0, True), (1, False), (2, True), (3, True)]
>>> [(i, x.is_grassmannian(index_set=[i], side='left')) for i in I]
[(0, False), (1, True), (2, True), (3, True)]
```

is_translation()

Return whether `self` is a translation element or not.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); FW = E.FW()
sage: F = E.fundamental_group()
sage: FW.from_affine_weyl(E.affine_weyl().from_reduced_word([1,2,1,
↪0])).is_translation()
True
sage: FW.from_translation(E.lattice_basis()[1]).is_translation()
True
sage: FW.simple_reflection(0).is_translation()
False
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]); FW = E.
↪FW()
>>> F = E.fundamental_group()
>>> FW.from_affine_weyl(E.affine_weyl().from_reduced_word([Integer(1),
↪Integer(2),Integer(1),Integer(0)])).is_translation()
True
```

(continues on next page)

(continued from previous page)

```
>>> FW.from_translation(E.lattice_basis()[Integer(1)]).is_translation()
True
>>> FW.simple_reflection(Integer(0)).is_translation()
False
```

length()

Return the length of `self` in the Coxeter group sense.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0 = E.PW0()
sage: I0 = E.cartan_type().classical().index_set()
sage: [PW0.from_translation(E.lattice_basis()[i]).length() for i in I0]
[3, 4, 3]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); PW0 = E.
↪PW0()
>>> I0 = E.cartan_type().classical().index_set()
>>> [PW0.from_translation(E.lattice_basis()[i]).length() for i in I0]
[3, 4, 3]
```

to_affine_grassmannian()

Return the unique affine Grassmannian element in the same coset of `self` with respect to the finite Weyl group acting on the right.

EXAMPLES:

```
sage: elts = ExtendedAffineWeylGroup(['A',2,1]).PW0().some_elements()
sage: [(x, x.to_affine_grassmannian()) for x in elts]
[(t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2,
  t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2*s1)]
```

```
>>> from sage.all import *
>>> elts = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PW0().
↪some_elements()
>>> [(x, x.to_affine_grassmannian()) for x in elts]
[(t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2,
  t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2*s1)]
```

to_affine_weyl_left()

Return the projection of `self` to the affine Weyl group on the left, after factorizing using the style "WF".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0 = E.PW0()
sage: b = E.lattice_basis()
sage: [(x, PW0.from_translation(x).to_affine_weyl_left()) for x in b]
[(Lambdacheck[1], S0*S3*S2),
 (Lambdacheck[2], S0*S3*S1*S0),
 (Lambdacheck[3], S0*S1*S2)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); PW0 = E.
↪PW0()
>>> b = E.lattice_basis()
>>> [(x, PW0.from_translation(x).to_affine_weyl_left()) for x in b]
[(Lambdacheck[1], S0*S3*S2),
 (Lambdacheck[2], S0*S3*S1*S0),
 (Lambdacheck[3], S0*S1*S2)]
```

Warning: Must be implemented in style “WF”.

to_affine_weyl_right()

Return the projection of `self` to the affine Weyl group on the right, after factorizing using the style “FW”.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0 = E.PW0()
sage: b = E.lattice_basis()
sage: [(x, PW0.from_translation(x).to_affine_weyl_right()) for x in b]
[(Lambdacheck[1], S3*S2*S1),
 (Lambdacheck[2], S2*S3*S1*S2),
 (Lambdacheck[3], S1*S2*S3)]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); PW0 = E.
↪PW0()
>>> b = E.lattice_basis()
>>> [(x, PW0.from_translation(x).to_affine_weyl_right()) for x in b]
[(Lambdacheck[1], S3*S2*S1),
 (Lambdacheck[2], S2*S3*S1*S2),
 (Lambdacheck[3], S1*S2*S3)]
```

Warning: Must be implemented in style “FW”.

to_classical_weyl()

Return the image of `self` under the homomorphism to the classical Weyl group.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).WF().simple_reflection(0).to_
↪classical_weyl()
s1*s2*s3*s2*s1
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).WF().simple_
```

(continues on next page)

(continued from previous page)

```
→reflection(Integer(0)).to_classical_weyl()
s1*s2*s3*s2*s1
```

Warning: Must be implemented in style "PW0".

to_dual_classical_weyl()

Return the image of `self` under the homomorphism to the dual form of the classical Weyl group.

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',3,1]).WF().simple_reflection(0).
→to_dual_classical_weyl(); x
s1*s2*s3*s2*s1
sage: x.parent()
Weyl Group of type ['A', 3] (as a matrix group acting on the weight␣
→lattice)
```

```
>>> from sage.all import *
>>> x = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).WF().
→simple_reflection(Integer(0)).to_dual_classical_weyl(); x
s1*s2*s3*s2*s1
>>> x.parent()
Weyl Group of type ['A', 3] (as a matrix group acting on the weight␣
→lattice)
```

Warning: Must be implemented in style "PvW0".

to_dual_translation_left()

Return the projection of `self` to the dual translation lattice after factorizing it to the left using the style "PvW0".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PvW0().simple_reflection(0).
→to_dual_translation_left()
Lambda[1] + Lambda[3]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).PvW0().simple_
→reflection(Integer(0)).to_dual_translation_left()
Lambda[1] + Lambda[3]
```

Warning:

Must be implemented in style "PvW0".

to_dual_translation_right()

Return the projection of `self` to the dual translation lattice after factorizing it to the right using the style "W0Pv".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PW0().simple_reflection(0).to_
↪dual_translation_right()
-Lambda[1] - Lambda[3]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).PW0().simple_
↪reflection(Integer(0)).to_dual_translation_right()
-Lambda[1] - Lambda[3]
```

Warning

Must be implemented in style "W0Pv".

to_fundamental_group()

Return the image of `self` under the homomorphism to the fundamental group.

EXAMPLES:

```
sage: PW0 = ExtendedAffineWeylGroup(['A',3,1]).PW0()
sage: b = PW0.realization_of().lattice_basis()
sage: [(x, PW0.from_translation(x).to_fundamental_group()) for x in b]
[(Lambdacheck[1], pi[1]), (Lambdacheck[2], pi[2]), (Lambdacheck[3],␣
↪pi[3])]
```

```
>>> from sage.all import *
>>> PW0 = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).PW0()
>>> b = PW0.realization_of().lattice_basis()
>>> [(x, PW0.from_translation(x).to_fundamental_group()) for x in b]
[(Lambdacheck[1], pi[1]), (Lambdacheck[2], pi[2]), (Lambdacheck[3],␣
↪pi[3])]
```

Warning

Must be implemented in style "WF".

to_translation_left()

Return the projection of `self` to the translation lattice after factorizing it to the left using the style "PW0".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PW0().simple_reflection(0).to_
↪translation_left()
Lambdacheck[1] + Lambdacheck[3]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).PW0().simple_
↪reflection(Integer(0)).to_translation_left()
Lambdacheck[1] + Lambdacheck[3]
```

> **Warning**
>
> Must be implemented in style "PW0".

to_translation_right()

Return the projection of `self` to the translation lattice after factorizing it to the right using the style "W0P".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PW0().simple_reflection(0).to_
↪translation_right()
-Lambdacheck[1] - Lambdacheck[3]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).PW0().simple_
↪reflection(Integer(0)).to_translation_right()
-Lambdacheck[1] - Lambdacheck[3]
```

> **Warning**
>
> Must be implemented in style "W0P".

class ParentMethods

Bases: `object`

from_affine_weyl(w)

Return the image of w under the homomorphism from the affine Weyl group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0 = E.PW0()
sage: W = E.affine_weyl()
sage: w = W.from_reduced_word([2,1,3,0])
sage: x = PW0.from_affine_weyl(w); x
t[Lambdacheck[1] - 2*Lambdacheck[2] + Lambdacheck[3]] * s3*s1
sage: FW = E.FW()
sage: y = FW.from_affine_weyl(w); y
S2*S3*S1*S0
sage: FW(x) == y
True
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); PW0 = E.
↪PW0()
>>> W = E.affine_weyl()
>>> w = W.from_reduced_word([Integer(2),Integer(1),Integer(3),
↪Integer(0)])
>>> x = PW0.from_affine_weyl(w); x
t[Lambdacheck[1] - 2*Lambdacheck[2] + Lambdacheck[3]] * s3*s1
>>> FW = E.FW()
>>> y = FW.from_affine_weyl(w); y
S2*S3*S1*S0
>>> FW(x) == y
True
```

Warning

Must be implemented in style "WF" and "FW".

from_classical_weyl(*w*)

Return the image of w from the finite Weyl group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0=E.PW0()
sage: W0 = E.classical_weyl()
sage: w = W0.from_reduced_word([2,1,3])
sage: y = PW0.from_classical_weyl(w); y
s2*s3*s1
sage: y.parent() == PW0
True
sage: y.to_classical_weyl() == w
True
sage: W0P = E.W0P()
sage: z = W0P.from_classical_weyl(w); z
s2*s3*s1
sage: z.parent() == W0P
True
sage: W0P(y) == z
True
sage: FW = E.FW()
sage: x = FW.from_classical_weyl(w); x
S2*S3*S1
sage: x.parent() == FW
True
sage: FW(y) == x
True
sage: FW(z) == x
True
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); PW0=E.
```

(continues on next page)

(continued from previous page)

```
↪PW0()
>>> W0 = E.classical_weyl()
>>> w = W0.from_reduced_word([Integer(2),Integer(1),Integer(3)])
>>> y = PW0.from_classical_weyl(w); y
s2*s3*s1
>>> y.parent() == PW0
True
>>> y.to_classical_weyl() == w
True
>>> W0P = E.W0P()
>>> z = W0P.from_classical_weyl(w); z
s2*s3*s1
>>> z.parent() == W0P
True
>>> W0P(y) == z
True
>>> FW = E.FW()
>>> x = FW.from_classical_weyl(w); x
S2*S3*S1
>>> x.parent() == FW
True
>>> FW(y) == x
True
>>> FW(z) == x
True
```

> **Warning**
>
> Must be implemented in style "PW0" and "W0P".

from_dual_classical_weyl(*w*)

Return the image of w from the finite Weyl group of dual form into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PvW0 = E.PvW0()
sage: W0v = E.dual_classical_weyl()
sage: w = W0v.from_reduced_word([2,1,3])
sage: y = PvW0.from_dual_classical_weyl(w); y
s2*s3*s1
sage: y.parent() == PvW0
True
sage: y.to_dual_classical_weyl() == w
True
sage: x = E.FW().from_dual_classical_weyl(w); x
S2*S3*S1
sage: PvW0(x) == y
True
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); PvW0 = E.
```

(continues on next page)

(continued from previous page)

```
↪PvW0()
>>> W0v = E.dual_classical_weyl()
>>> w = W0v.from_reduced_word([Integer(2),Integer(1),Integer(3)])
>>> y = PvW0.from_dual_classical_weyl(w); y
s2*s3*s1
>>> y.parent() == PvW0
True
>>> y.to_dual_classical_weyl() == w
True
>>> x = E.FW().from_dual_classical_weyl(w); x
S2*S3*S1
>>> PvW0(x) == y
True
```

Warning

Must be implemented in style "PvW0" and "W0Pv".

from_dual_translation(*la*)

Return the image of `la` under the homomorphism of the dual version of the translation lattice into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); PvW0 = E.PvW0()
sage: bv = E.dual_lattice_basis(); bv
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: x = PvW0.from_dual_translation(2*bv[1] - bv[2]); x
t[2*Lambda[1] - Lambda[2]]
sage: FW = E.FW()
sage: y = FW.from_dual_translation(2*bv[1] - bv[2]); y
S0*S2*S0*S1
sage: FW(x) == y
True
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]); PvW0 = E.
↪PvW0()
>>> bv = E.dual_lattice_basis(); bv
Finite family {1: Lambda[1], 2: Lambda[2]}
>>> x = PvW0.from_dual_translation(Integer(2)*bv[Integer(1)] -␣
↪bv[Integer(2)]); x
t[2*Lambda[1] - Lambda[2]]
>>> FW = E.FW()
>>> y = FW.from_dual_translation(Integer(2)*bv[Integer(1)] -␣
↪bv[Integer(2)]); y
S0*S2*S0*S1
>>> FW(x) == y
True
```

from_fundamental(*x*)

Return the image of x under the homomorphism from the fundamental group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1])
sage: PW0=E.PW0()
sage: F = E.fundamental_group()
sage: Is = F.special_nodes()
sage: [(i, PW0.from_fundamental(F(i))) for i in Is]
[(0, 1),
 (1, t[Lambdacheck[1]] * s1*s2*s3),
 (2, t[Lambdacheck[2]] * s2*s3*s1*s2),
 (3, t[Lambdacheck[3]] * s3*s2*s1)]
sage: [(i, E.W0P().from_fundamental((F(i)))) for i in Is]
[(0, 1),
 (1, s1*s2*s3 * t[-Lambdacheck[3]]),
 (2, s2*s3*s1*s2 * t[-Lambdacheck[2]]),
 (3, s3*s2*s1 * t[-Lambdacheck[1]])]
sage: [(i, E.WF().from_fundamental(F(i))) for i in Is]
[(0, 1), (1, pi[1]), (2, pi[2]), (3, pi[3])]
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)])
>>> PW0=E.PW0()
>>> F = E.fundamental_group()
>>> Is = F.special_nodes()
>>> [(i, PW0.from_fundamental(F(i))) for i in Is]
[(0, 1),
 (1, t[Lambdacheck[1]] * s1*s2*s3),
 (2, t[Lambdacheck[2]] * s2*s3*s1*s2),
 (3, t[Lambdacheck[3]] * s3*s2*s1)]
>>> [(i, E.W0P().from_fundamental((F(i)))) for i in Is]
[(0, 1),
 (1, s1*s2*s3 * t[-Lambdacheck[3]]),
 (2, s2*s3*s1*s2 * t[-Lambdacheck[2]]),
 (3, s3*s2*s1 * t[-Lambdacheck[1]])]
>>> [(i, E.WF().from_fundamental(F(i))) for i in Is]
[(0, 1), (1, pi[1]), (2, pi[2]), (3, pi[3])]
```

Warning

This method must be implemented by the "WF" and "FW" realizations.

from_reduced_word(*word*)

Convert an affine or finite reduced word into a group element.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PW0().from_reduced_word([1,0,
↪1,2])
t[-Lambdacheck[1] + 2*Lambdacheck[2]]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).PW0().from_
↪reduced_word([Integer(1),Integer(0),Integer(1),Integer(2)])
t[-Lambdacheck[1] + 2*Lambdacheck[2]]
```

from_translation(*la*)

Return the element of translation by `la` in `self`.

INPUT:

- `self` – a realization of the extended affine Weyl group
- `la` – an element of the translation lattice

In the notation of the documentation for *ExtendedAffineWeylGroup()*, `la` must be an element of "P".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); PW0 = E.PW0()
sage: b = E.lattice_basis(); b
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
sage: x = PW0.from_translation(2*b[1] - b[2]); x
t[2*Lambdacheck[1] - Lambdacheck[2]]
sage: FW = E.FW()
sage: y = FW.from_translation(2*b[1] - b[2]); y
S0*S2*S0*S1
sage: FW(x) == y
True
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]); PW0 = E.
↪PW0()
>>> b = E.lattice_basis(); b
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
>>> x = PW0.from_translation(Integer(2)*b[Integer(1)] - b[Integer(2)]);
↪ x
t[2*Lambdacheck[1] - Lambdacheck[2]]
>>> FW = E.FW()
>>> y = FW.from_translation(Integer(2)*b[Integer(1)] - b[Integer(2)]);␣
↪y
S0*S2*S0*S1
>>> FW(x) == y
True
```

Since the implementation as a semidirect product requires wrapping the lattice group to make it multiplicative, we cannot declare that this map is a morphism for sage `Groups()`.

> **Warning**
>
> This method must be implemented by the "PW0" and "W0P" realizations.

simple_reflection(*i*)

Return the i-th simple reflection in `self`.

INPUT:

- `self` – a realization of the extended affine Weyl group
- `i` – an affine Dynkin node

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PW0().simple_reflection(0)
t[Lambdacheck[1] + Lambdacheck[3]] * s1*s2*s3*s2*s1
sage: ExtendedAffineWeylGroup(['C',2,1]).WF().simple_reflection(0)
S0
sage: ExtendedAffineWeylGroup(['D',3,2]).PvW0().simple_reflection(1)
s1
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).PW0().simple_
→reflection(Integer(0))
t[Lambdacheck[1] + Lambdacheck[3]] * s1*s2*s3*s2*s1
>>> ExtendedAffineWeylGroup(['C',Integer(2),Integer(1)]).WF().simple_
→reflection(Integer(0))
S0
>>> ExtendedAffineWeylGroup(['D',Integer(3),Integer(2)]).PvW0().simple_
→reflection(Integer(1))
s1
```

simple_reflections()

Return a family from the set of affine Dynkin nodes to the simple reflections in the realization of the extended affine Weyl group.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).W0P().simple_reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]],
               1: s1, 2: s2, 3: s3}
sage: ExtendedAffineWeylGroup(['A',3,1]).WF().simple_reflections()
Finite family {0: S0, 1: S1, 2: S2, 3: S3}
sage: ExtendedAffineWeylGroup(['A',3,1],
....:                         print_tuple=True).FW().simple_
→reflections()
Finite family {0: (pi[0], S0), 1: (pi[0], S1),
               2: (pi[0], S2), 3: (pi[0], S3)}
sage: ExtendedAffineWeylGroup(['A',3,1],
....:                         fundamental='f',
....:                         print_tuple=True).FW().simple_
→reflections()
Finite family {0: (f[0], S0), 1: (f[0], S1),
               2: (f[0], S2), 3: (f[0], S3)}
sage: ExtendedAffineWeylGroup(['A',3,1]).PvW0().simple_reflections()
Finite family {0: t[Lambda[1] + Lambda[3]] * s1*s2*s3*s2*s1,
               1: s1, 2: s2, 3: s3}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).W0P().simple_
→reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]],
               1: s1, 2: s2, 3: s3}
```

(continues on next page)

(continued from previous page)

```
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).WF().simple_
↪reflections()
Finite family {0: S0, 1: S1, 2: S2, 3: S3}
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)],
...                          print_tuple=True).FW().simple_reflections()
Finite family {0: (pi[0], S0), 1: (pi[0], S1),
               2: (pi[0], S2), 3: (pi[0], S3)}
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)],
...                          fundamental='f',
...                          print_tuple=True).FW().simple_reflections()
Finite family {0: (f[0], S0), 1: (f[0], S1),
               2: (f[0], S2), 3: (f[0], S3)}
>>> ExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).PvW0().simple_
↪reflections()
Finite family {0: t[Lambda[1] + Lambda[3]] * s1*s2*s3*s2*s1,
               1: s1, 2: s2, 3: s3}
```

super_categories()

EXAMPLES:

```
sage: R = ExtendedAffineWeylGroup(['A',2,1]).Realizations(); R
Category of realizations of Extended affine Weyl group of type ['A', 2, 1]
sage: R.super_categories()
[Category of associative inverse realizations of unital magmas]
```

```
>>> from sage.all import *
>>> R = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).
↪Realizations(); R
Category of realizations of Extended affine Weyl group of type ['A', 2, 1]
>>> R.super_categories()
[Category of associative inverse realizations of unital magmas]
```

W0P()

Realizes `self` in "W0P"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).W0P()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2] (as a matrix group acting on the coweight␣
↪lattice)
 acting on Multiplicative form of Coweight lattice of the Root system of type␣
↪['A', 2]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).W0P()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2] (as a matrix group acting on the coweight␣
↪lattice)
```

(continues on next page)

(continued from previous page)

```
acting on Multiplicative form of Coweight lattice of the Root system of type␣
↪['A', 2]
```

W0Pv()

Realizes `self` in "W0Pv"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).W0Pv()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
  acting on Multiplicative form of Weight lattice of the Root system of type [
↪'A', 2]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).W0Pv()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
  acting on Multiplicative form of Weight lattice of the Root system of type [
↪'A', 2]
```

WF()

Realizes `self` in "WF"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).WF()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
 acted upon by Fundamental group of type ['A', 2, 1]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).WF()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
 acted upon by Fundamental group of type ['A', 2, 1]
```

WF_to_PW0_func(*x*)

Coercion from style "WF" to "PW0".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(["A", 2, 1])
sage: x = E.WF().an_element(); x
S0*S1*S2 * pi[2]
sage: E.WF_to_PW0_func(x)
t[Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2*s1
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(["A", Integer(2), Integer(1)])
>>> x = E.WF().an_element(); x
S0*S1*S2 * pi[2]
>>> E.WF_to_PW0_func(x)
t[Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2*s1
```

Warning

Since this is used to define some coercion maps it cannot itself use coercion.

a_realization()

Return the default realization of an extended affine Weyl group.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).a_realization()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2] (as a matrix group acting on the␣
↪coweight lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).a_realization()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product␣
↪of
 Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2] (as a matrix group acting on the␣
↪coweight lattice)
```

affine_weyl()

Return the affine Weyl group of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).affine_weyl()
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
sage: ExtendedAffineWeylGroup(['A',5,2]).affine_weyl()
Weyl Group of type ['B', 3, 1]^* (as a matrix group acting on the root␣
↪lattice)
sage: ExtendedAffineWeylGroup(['A',4,2]).affine_weyl()
Weyl Group of type ['BC', 2, 2] (as a matrix group acting on the root lattice)
sage: ExtendedAffineWeylGroup(CartanType(['A',4,2]).dual()).affine_weyl()
Weyl Group of type ['BC', 2, 2]^* (as a matrix group acting on the root␣
↪lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).affine_weyl()
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
>>> ExtendedAffineWeylGroup(['A',Integer(5),Integer(2)]).affine_weyl()
```

(continues on next page)

(continued from previous page)

```
Weyl Group of type ['B', 3, 1]^* (as a matrix group acting on the root␣
↪lattice)
>>> ExtendedAffineWeylGroup(['A',Integer(4),Integer(2)]).affine_weyl()
Weyl Group of type ['BC', 2, 2] (as a matrix group acting on the root lattice)
>>> ExtendedAffineWeylGroup(CartanType(['A',Integer(4),Integer(2)]).dual()).
↪affine_weyl()
Weyl Group of type ['BC', 2, 2]^* (as a matrix group acting on the root␣
↪lattice)
```

cartan_type()

The Cartan type of self.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(["D",3,2]).cartan_type()
['C', 2, 1]^*
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(["D",Integer(3),Integer(2)]).cartan_type()
['C', 2, 1]^*
```

classical_weyl()

Return the classical Weyl group of self.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).classical_weyl()
Weyl Group of type ['A', 2] (as a matrix group acting on the coweight lattice)
sage: ExtendedAffineWeylGroup(['A',5,2]).classical_weyl()
Weyl Group of type ['C', 3] (as a matrix group acting on the weight lattice)
sage: ExtendedAffineWeylGroup(['A',4,2]).classical_weyl()
Weyl Group of type ['C', 2] (as a matrix group acting on the weight lattice)
sage: ExtendedAffineWeylGroup(CartanType(['A',4,2]).dual()).classical_weyl()
Weyl Group of type ['C', 2] (as a matrix group acting on the coweight lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).classical_weyl()
Weyl Group of type ['A', 2] (as a matrix group acting on the coweight lattice)
>>> ExtendedAffineWeylGroup(['A',Integer(5),Integer(2)]).classical_weyl()
Weyl Group of type ['C', 3] (as a matrix group acting on the weight lattice)
>>> ExtendedAffineWeylGroup(['A',Integer(4),Integer(2)]).classical_weyl()
Weyl Group of type ['C', 2] (as a matrix group acting on the weight lattice)
>>> ExtendedAffineWeylGroup(CartanType(['A',Integer(4),Integer(2)]).dual()).
↪classical_weyl()
Weyl Group of type ['C', 2] (as a matrix group acting on the coweight lattice)
```

classical_weyl_to_affine(*w*)

The image of w under the homomorphism from the classical Weyl group into the affine Weyl group.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: W0 = E.classical_weyl()
```

(continues on next page)

(continued from previous page)

```
sage: w = W0.from_reduced_word([1,2]); w
s1*s2
sage: v = E.classical_weyl_to_affine(w); v
S1*S2
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> W0 = E.classical_weyl()
>>> w = W0.from_reduced_word([Integer(1),Integer(2)]); w
s1*s2
>>> v = E.classical_weyl_to_affine(w); v
S1*S2
```

dual_classical_weyl()

Return the dual version of the classical Weyl group of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).dual_classical_weyl()
Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
sage: ExtendedAffineWeylGroup(['A',5,2]).dual_classical_weyl()
Weyl Group of type ['C', 3] (as a matrix group acting on the weight lattice)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).dual_classical_weyl()
Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
>>> ExtendedAffineWeylGroup(['A',Integer(5),Integer(2)]).dual_classical_weyl()
Weyl Group of type ['C', 3] (as a matrix group acting on the weight lattice)
```

dual_classical_weyl_to_affine(w)

The image of w under the homomorphism from the dual version of the classical Weyl group into the affine Weyl group.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: W0v = E.dual_classical_weyl()
sage: w = W0v.from_reduced_word([1,2]); w
s1*s2
sage: v = E.dual_classical_weyl_to_affine(w); v
S1*S2
```

```
>>> from sage.all import *
>>> E = ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> W0v = E.dual_classical_weyl()
>>> w = W0v.from_reduced_word([Integer(1),Integer(2)]); w
s1*s2
>>> v = E.dual_classical_weyl_to_affine(w); v
S1*S2
```

dual_lattice()

Return the dual version of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).dual_lattice()
Weight lattice of the Root system of type ['A', 2]
sage: ExtendedAffineWeylGroup(['A',5,2]).dual_lattice()
Weight lattice of the Root system of type ['C', 3]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).dual_lattice()
Weight lattice of the Root system of type ['A', 2]
>>> ExtendedAffineWeylGroup(['A',Integer(5),Integer(2)]).dual_lattice()
Weight lattice of the Root system of type ['C', 3]
```

dual_lattice_basis()

Return the distinguished basis of the dual version of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).dual_lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: ExtendedAffineWeylGroup(['A',5,2]).dual_lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).dual_lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2]}
>>> ExtendedAffineWeylGroup(['A',Integer(5),Integer(2)]).dual_lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}
```

exp_dual_lattice()

Return the multiplicative version of the dual version of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).exp_dual_lattice()
Multiplicative form of Weight lattice of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).exp_dual_lattice()
Multiplicative form of Weight lattice of the Root system of type ['A', 2]
```

exp_lattice()

Return the multiplicative version of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).exp_lattice()
Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).exp_lattice()
Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
```

fundamental_group()

Return the abstract fundamental group.

EXAMPLES:

```
sage: F = ExtendedAffineWeylGroup(['D',5,1]).fundamental_group(); F
Fundamental group of type ['D', 5, 1]
sage: [a for a in F]
[pi[0], pi[1], pi[4], pi[5]]
```

```
>>> from sage.all import *
>>> F = ExtendedAffineWeylGroup(['D',Integer(5),Integer(1)]).fundamental_
↪group(); F
Fundamental group of type ['D', 5, 1]
>>> [a for a in F]
[pi[0], pi[1], pi[4], pi[5]]
```

group_generators()

Return a set of generators for the default realization of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).group_generators()
(t[Lambdacheck[1]], t[Lambdacheck[2]], s1, s2)
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).group_generators()
(t[Lambdacheck[1]], t[Lambdacheck[2]], s1, s2)
```

lattice()

Return the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).lattice()
Coweight lattice of the Root system of type ['A', 2]
sage: ExtendedAffineWeylGroup(['A',5,2]).lattice()
Weight lattice of the Root system of type ['C', 3]
sage: ExtendedAffineWeylGroup(['A',4,2]).lattice()
Weight lattice of the Root system of type ['C', 2]
sage: ExtendedAffineWeylGroup(CartanType(['A',4,2]).dual()).lattice()
Coweight lattice of the Root system of type ['B', 2]
sage: ExtendedAffineWeylGroup(CartanType(['A',2,1]),
....:                         general_linear=True).lattice()
Ambient space of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).lattice()
Coweight lattice of the Root system of type ['A', 2]
>>> ExtendedAffineWeylGroup(['A',Integer(5),Integer(2)]).lattice()
Weight lattice of the Root system of type ['C', 3]
>>> ExtendedAffineWeylGroup(['A',Integer(4),Integer(2)]).lattice()
Weight lattice of the Root system of type ['C', 2]
>>> ExtendedAffineWeylGroup(CartanType(['A',Integer(4),Integer(2)]).dual()).
```

(continues on next page)

(continued from previous page)

```
↪lattice()
Coweight lattice of the Root system of type ['B', 2]
>>> ExtendedAffineWeylGroup(CartanType(['A',Integer(2),Integer(1)]),
...                          general_linear=True).lattice()
Ambient space of the Root system of type ['A', 2]
```

lattice_basis()

Return the distinguished basis of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).lattice_basis()
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
sage: ExtendedAffineWeylGroup(['A',5,2]).lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}
sage: ExtendedAffineWeylGroup(['A',4,2]).lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: ExtendedAffineWeylGroup(CartanType(['A',4,2]).dual()).lattice_basis()
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
```

```
>>> from sage.all import *
>>> ExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).lattice_basis()
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
>>> ExtendedAffineWeylGroup(['A',Integer(5),Integer(2)]).lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}
>>> ExtendedAffineWeylGroup(['A',Integer(4),Integer(2)]).lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2]}
>>> ExtendedAffineWeylGroup(CartanType(['A',Integer(4),Integer(2)]).dual()).
↪lattice_basis()
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
```

5.1.265 Fundamental Group of an Extended Affine Weyl Group

AUTHORS:

- Mark Shimozono (2013) initial version

class sage.combinat.root_system.fundamental_group.**FundamentalGroupElement**(*parent*, *x*)

Bases: `MultiplicativeGroupElement`

This should not be called directly.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: x = FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1], prefix='f').an_
↪element()
sage: TestSuite(x).run()
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
```

(continues on next page)

(continued from previous page)

```
>>> x = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(4),Integer(1)],␣
↪prefix='f').an_element()
>>> TestSuite(x).run()
```

act_on_affine_lattice(*wt*)

Act by `self` on the element `wt` of an affine root/weight lattice realization.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: wt = RootSystem(F.cartan_type()).weight_lattice().an_element(); wt
2*Lambda[0] + 2*Lambda[1] + 3*Lambda[2]
sage: F(3).act_on_affine_lattice(wt)
2*Lambda[0] + 3*Lambda[1] + 2*Lambda[3]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(3),Integer(1)])
>>> wt = RootSystem(F.cartan_type()).weight_lattice().an_element(); wt
2*Lambda[0] + 2*Lambda[1] + 3*Lambda[2]
>>> F(Integer(3)).act_on_affine_lattice(wt)
2*Lambda[0] + 3*Lambda[1] + 2*Lambda[3]
```

Warning

Doesn't work on ambient spaces.

act_on_affine_weyl(*w*)

Act by `self` on the element w of the affine Weyl group.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: W = WeylGroup(F.cartan_type(),prefix='s')
sage: w = W.from_reduced_word([2,3,0])
sage: F(1).act_on_affine_weyl(w).reduced_word()
[3, 0, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(3),Integer(1)])
>>> W = WeylGroup(F.cartan_type(),prefix='s')
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(0)])
>>> F(Integer(1)).act_on_affine_weyl(w).reduced_word()
[3, 0, 1]
```

value()

Return the special node which indexes the special automorphism `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1], prefix='f')
sage: F.special_nodes()
(0, 1, 2, 3, 4)
sage: x = F(4); x
f[4]
sage: x.value()
4
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(4),Integer(1)],
↪ prefix='f')
>>> F.special_nodes()
(0, 1, 2, 3, 4)
>>> x = F(Integer(4)); x
f[4]
>>> x.value()
4
```

class sage.combinat.root_system.fundamental_group.**FundamentalGroupGL**(*cartan_type*, *prefix='pi'*)

Bases: *FundamentalGroupOfExtendedAffineWeylGroup_Class*

Fundamental group of GL_n. It is just the integers with extra privileges.

Element

alias of *FundamentalGroupGLElement*

action(i)

The action of the i-th automorphism on the affine Dynkin node set.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True)
sage: F.action(4)(2)
0
sage: F.action(-4)(2)
1
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],
↪ general_linear=True)
>>> F.action(Integer(4))(Integer(2))
```

(continues on next page)

(continued from previous page)

```
0
>>> F.action(-Integer(4))(Integer(2))
1
```

an_element()

An element of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True).an_element()
pi[5]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],␣
↪general_linear=True).an_element()
pi[5]
```

dual_node(i)

The node whose special automorphism is inverse to that of i.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True)
sage: F.dual_node(2)
-2
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],
↪ general_linear=True)
>>> F.dual_node(Integer(2))
-2
```

family()

The family associated with the set of special nodes.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: fam = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True).family() # indirect doctest
sage: fam
Lazy family (<lambda>(i))_{i in Integer Ring}
```

(continues on next page)

(continued from previous page)

```
sage: fam[-3]
-3
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> fam = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),
↪Integer(1)], general_linear=True).family() # indirect doctest
>>> fam
Lazy family (<lambda>(i))_{i in Integer Ring}
>>> fam[-Integer(3)]
-3
```

group_generators()

Return group generators for `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True).group_generators()
(pi[1],)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],␣
↪general_linear=True).group_generators()
(pi[1],)
```

one()

Return the identity element of the fundamental group.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True).one()
pi[0]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],␣
↪general_linear=True).one()
pi[0]
```

product(x, y)

Return the product of x and y.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True)
sage: F.special_nodes()
Integer Ring
sage: F(2)*F(3)
pi[5]
sage: F(1)*F(3)^(-1)
pi[-2]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],
↪ general_linear=True)
>>> F.special_nodes()
Integer Ring
>>> F(Integer(2))*F(Integer(3))
pi[5]
>>> F(Integer(1))*F(Integer(3))**(-Integer(1))
pi[-2]
```

reduced_word(i)

A reduced word for the finite permutation part of the special automorphism indexed by i.

More precisely, return a reduced word for the finite Weyl group element u where i-th automorphism (expressed in the extended affine Weyl group) has the form tu where t is a translation element.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True)
sage: F.reduced_word(10)
(1, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],
↪ general_linear=True)
>>> F.reduced_word(Integer(10))
(1, 2)
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
```

(continues on next page)

(continued from previous page)

```
↪linear=True).some_elements()
[pi[-2], pi[2], pi[5]]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],␣
↪general_linear=True).some_elements()
[pi[-2], pi[2], pi[5]]
```

class sage.combinat.root_system.fundamental_group.**FundamentalGroupGLElement**(*parent*, *x*)

Bases: *FundamentalGroupElement*

act_on_classical_ambient(*wt*)

Act by `self` on the classical ambient weight vector `wt`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True)
sage: f = F.an_element(); f
pi[5]
sage: la = F.cartan_type().classical().root_system().ambient_space().an_
↪element(); la
(2, 2, 3)
sage: f.act_on_classical_ambient(la)
(2, 3, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],
↪ general_linear=True)
>>> f = F.an_element(); f
pi[5]
>>> la = F.cartan_type().classical().root_system().ambient_space().an_
↪element(); la
(2, 2, 3)
>>> f.act_on_classical_ambient(la)
(2, 3, 2)
```

sage.combinat.root_system.fundamental_group.**FundamentalGroupOfExtendedAffineWeylGroup**(*cartan_type*, *prefix='pi'*, *general_linear=None*)

Factory for the fundamental group of an extended affine Weyl group.

INPUT:

- `cartan_type` – a Cartan type that is either affine or finite, with the latter being a shorthand for the untwisted affinization
- `prefix` – (default: `'pi'`) string that labels the elements of the group
- `general_linear` – (default: `None`, meaning `False`) In untwisted type A, if `True`, use the universal central extension

Fundamental group

Associated to each affine Cartan type $\tilde{X}$ is an extended affine Weyl group E. Its subgroup of length-zero elements is called the fundamental group F. The group F can be identified with a subgroup of the group of automorphisms of the affine Dynkin diagram. As such, every element of F can be viewed as a permutation of the set I of affine Dynkin nodes.

Let $0 \in I$ be the distinguished affine node; it is the one whose removal produces the associated finite Cartan type (call it X). A node $i \in I$ is called *special* if some automorphism of the affine Dynkin diagram, sends 0 to i. The node 0 is always special due to the identity automorphism. There is a bijection of the set of special nodes with the fundamental group. We denote the image of i by π_i. The structure of F is determined as follows.

- $\tilde{X}$ is untwisted – F is isomorphic to $P^\vee/Q^\vee$ where $P^\vee$ and $Q^\vee$ are the coweight and coroot lattices of type X. The group $P^\vee/Q^\vee$ consists of the cosets $\omega_i^\vee + Q^\vee$ for special nodes i, where $\omega_0^\vee = 0$ by convention. In this case the special nodes i are the *cominuscule* nodes, the ones such that $\omega_i^\vee(\alpha_j)$ is 0 or 1 for all $j \in I_0 = I \setminus \{0\}$. For i special, addition by $\omega_i^\vee + Q^\vee$ permutes $P^\vee/Q^\vee$ and therefore permutes the set of special nodes. This permutation extends uniquely to an automorphism of the affine Dynkin diagram.
- $\tilde{X}$ is dual untwisted – (that is, the dual of $\tilde{X}$ is untwisted) F is isomorphic to P/Q where P and Q are the weight and root lattices of type X. The group P/Q consists of the cosets $\omega_i + Q$ for special nodes i, where $\omega_0 = 0$ by convention. In this case the special nodes i are the *minuscule* nodes, the ones such that $\alpha_j^\vee(\omega_i)$ is 0 or 1 for all $j \in I_0$. For i special, addition by $\omega_i + Q$ permutes P/Q and therefore permutes the set of special nodes. This permutation extends uniquely to an automorphism of the affine Dynkin diagram.
- $\tilde{X}$ is mixed – (that is, not of the above two types) F is the trivial group.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1]); F
Fundamental group of type ['A', 3, 1]
sage: F.cartan_type().dynkin_diagram()
0
O-------+
|       |
|       |
O---O---O
1   2   3
A3~
sage: F.special_nodes()
(0, 1, 2, 3)
sage: F(1)^2
pi[2]
sage: F(1)*F(2)
pi[3]
sage: F(3)^(-1)
pi[1]
```

(continues on next page)

(continued from previous page)

```
sage: F = FundamentalGroupOfExtendedAffineWeylGroup("B3"); F
Fundamental group of type ['B', 3, 1]
sage: F.cartan_type().dynkin_diagram()
    O 0
    |
    |
O---O=>=O
1   2   3
B3~
sage: F.special_nodes()
(0, 1)

sage: F = FundamentalGroupOfExtendedAffineWeylGroup("C2"); F
Fundamental group of type ['C', 2, 1]
sage: F.cartan_type().dynkin_diagram()
O=>=O=<=O
0   1   2
C2~
sage: F.special_nodes()
(0, 2)

sage: F = FundamentalGroupOfExtendedAffineWeylGroup("D4"); F
Fundamental group of type ['D', 4, 1]
sage: F.cartan_type().dynkin_diagram()
    O 4
    |
    |
O---O---O
1   |2  3
    |
    O 0
D4~
sage: F.special_nodes()
(0, 1, 3, 4)
sage: (F(4), F(4)^2)
(pi[4], pi[0])

sage: F = FundamentalGroupOfExtendedAffineWeylGroup("D5"); F
Fundamental group of type ['D', 5, 1]
sage: F.cartan_type().dynkin_diagram()
  0 O   O 5
    |   |
    |   |
O---O---O---O
1   2   3   4
D5~
sage: F.special_nodes()
(0, 1, 4, 5)
sage: (F(5), F(5)^2, F(5)^3, F(5)^4)
(pi[5], pi[1], pi[4], pi[0])
sage: F = FundamentalGroupOfExtendedAffineWeylGroup("E6"); F
Fundamental group of type ['E', 6, 1]
```

(continues on next page)

(continued from previous page)

```
sage: F.cartan_type().dynkin_diagram()
        O 0
        |
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
sage: F.special_nodes()
(0, 1, 6)
sage: F(1)^2
pi[6]

sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['D',4,2]); F
Fundamental group of type ['C', 3, 1]^*
sage: F.cartan_type().dynkin_diagram()
O=<=O---O=>=O
0   1   2   3
C3~*
sage: F.special_nodes()
(0, 3)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]); F
Fundamental group of type ['A', 3, 1]
>>> F.cartan_type().dynkin_diagram()
0
O-------+
|       |
|       |
O---O---O
1   2   3
A3~
>>> F.special_nodes()
(0, 1, 2, 3)
>>> F(Integer(1))**Integer(2)
pi[2]
>>> F(Integer(1))*F(Integer(2))
pi[3]
>>> F(Integer(3))**(-Integer(1))
pi[1]

>>> F = FundamentalGroupOfExtendedAffineWeylGroup("B3"); F
Fundamental group of type ['B', 3, 1]
>>> F.cartan_type().dynkin_diagram()
    O 0
    |
    |
```

(continues on next page)

(continued from previous page)

```
O---O=>=O
1   2   3
B3~
>>> F.special_nodes()
(0, 1)

>>> F = FundamentalGroupOfExtendedAffineWeylGroup("C2"); F
Fundamental group of type ['C', 2, 1]
>>> F.cartan_type().dynkin_diagram()
O=>=O=<=O
0   1   2
C2~
>>> F.special_nodes()
(0, 2)

>>> F = FundamentalGroupOfExtendedAffineWeylGroup("D4"); F
Fundamental group of type ['D', 4, 1]
>>> F.cartan_type().dynkin_diagram()
    O 4
    |
    |
O---O---O
1   |2  3
    |
    O 0
D4~
>>> F.special_nodes()
(0, 1, 3, 4)
>>> (F(Integer(4)), F(Integer(4))**Integer(2))
(pi[4], pi[0])

>>> F = FundamentalGroupOfExtendedAffineWeylGroup("D5"); F
Fundamental group of type ['D', 5, 1]
>>> F.cartan_type().dynkin_diagram()
  0 O   O 5
    |   |
    |   |
O---O---O---O
1   2   3   4
D5~
>>> F.special_nodes()
(0, 1, 4, 5)
>>> (F(Integer(5)), F(Integer(5))**Integer(2), F(Integer(5))**Integer(3),␣
↪F(Integer(5))**Integer(4))
(pi[5], pi[1], pi[4], pi[0])
>>> F = FundamentalGroupOfExtendedAffineWeylGroup("E6"); F
Fundamental group of type ['E', 6, 1]
>>> F.cartan_type().dynkin_diagram()
        O 0
        |
        |
        O 2
```

(continues on next page)

(continued from previous page)

```
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
>>> F.special_nodes()
(0, 1, 6)
>>> F(Integer(1))**Integer(2)
pi[6]

>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['D',Integer(4),Integer(2)]); F
Fundamental group of type ['C', 3, 1]^*
>>> F.cartan_type().dynkin_diagram()
O=<=O---O=>=O
0   1   2   3
C3~*
>>> F.special_nodes()
(0, 3)
```

We also implement a fundamental group for GL_n. It is defined to be the group of integers, which is the covering group of the fundamental group Z/nZ for affine SL_n:

```
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True); F
Fundamental group of GL(3)
sage: x = F.an_element(); x
pi[5]
sage: x*x
pi[10]
sage: x.inverse()
pi[-5]
sage: wt = F.cartan_type().classical().root_system().ambient_space().an_element();
↪ wt
(2, 2, 3)
sage: x.act_on_classical_ambient(wt)
(2, 3, 2)
sage: w = WeylGroup(F.cartan_type(),prefix='s').an_element(); w
s0*s1*s2
sage: x.act_on_affine_weyl(w)
s2*s0*s1
```

```
>>> from sage.all import *
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],␣
↪general_linear=True); F
Fundamental group of GL(3)
>>> x = F.an_element(); x
pi[5]
>>> x*x
pi[10]
>>> x.inverse()
pi[-5]
>>> wt = F.cartan_type().classical().root_system().ambient_space().an_element();␣
```

(continues on next page)

(continued from previous page)

```
↪wt
(2, 2, 3)
>>> x.act_on_classical_ambient(wt)
(2, 3, 2)
>>> w = WeylGroup(F.cartan_type(),prefix='s').an_element(); w
s0*s1*s2
>>> x.act_on_affine_weyl(w)
s2*s0*s1
```

class sage.combinat.root_system.fundamental_group.**FundamentalGroupOfExtendedAffineWeylGroup_Class**(*car tan, pre fix, fi- nite*

Bases: `UniqueRepresentation`, `Parent`

The group of length zero elements in the extended affine Weyl group.

Element

alias of *FundamentalGroupElement*

action(*i*)

Return a function which permutes the affine Dynkin node set by the i-th special automorphism.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1])
sage: [[(i, j, F.action(i)(j)) for j in F.index_set()] for i in F.special_
↪nodes()]
[[(0, 0, 0), (0, 1, 1), (0, 2, 2)], [(1, 0, 1), (1, 1, 2), (1, 2, 0)], [(2, 0,
↪ 2), (2, 1, 0), (2, 2, 1)]]
sage: G = FundamentalGroupOfExtendedAffineWeylGroup(['D',4,1])
sage: [[(i, j, G.action(i)(j)) for j in G.index_set()] for i in G.special_
↪nodes()]
[[(0, 0, 0), (0, 1, 1), (0, 2, 2), (0, 3, 3), (0, 4, 4)], [(1, 0, 1), (1, 1,␣
↪0), (1, 2, 2), (1, 3, 4), (1, 4, 3)], [(3, 0, 3), (3, 1, 4), (3, 2, 2), (3,␣
↪3, 0), (3, 4, 1)], [(4, 0, 4), (4, 1, 3), (4, 2, 2), (4, 3, 1), (4, 4, 0)]]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)])
>>> [[(i, j, F.action(i)(j)) for j in F.index_set()] for i in F.special_
↪nodes()]
[[(0, 0, 0), (0, 1, 1), (0, 2, 2)], [(1, 0, 1), (1, 1, 2), (1, 2, 0)], [(2, 0,
↪ 2), (2, 1, 0), (2, 2, 1)]]
>>> G = FundamentalGroupOfExtendedAffineWeylGroup(['D',Integer(4),Integer(1)])
>>> [[(i, j, G.action(i)(j)) for j in G.index_set()] for i in G.special_
↪nodes()]
[[(0, 0, 0), (0, 1, 1), (0, 2, 2), (0, 3, 3), (0, 4, 4)], [(1, 0, 1), (1, 1,␣
```

(continues on next page)

(continued from previous page)

```
↪0), (1, 2, 2), (1, 3, 4), (1, 4, 3)], [(3, 0, 3), (3, 1, 4), (3, 2, 2), (3,␣
↪3, 0), (3, 4, 1)], [(4, 0, 4), (4, 1, 3), (4, 2, 2), (4, 3, 1), (4, 4, 0)]]
```

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1],prefix='f').an_
↪element()
f[4]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(4),Integer(1)],
↪prefix='f').an_element()
f[4]
```

cartan_type()

The Cartan type of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1]).cartan_type()
['A', 3, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(3),Integer(1)]).
↪cartan_type()
['A', 3, 1]
```

dual_node(i)

Return the node that indexes the inverse of the i-th element.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1])
sage: [(i, F.dual_node(i)) for i in F.special_nodes()]
[(0, 0), (1, 4), (2, 3), (3, 2), (4, 1)]
sage: G = FundamentalGroupOfExtendedAffineWeylGroup(['E',6,1])
sage: [(i, G.dual_node(i)) for i in G.special_nodes()]
[(0, 0), (1, 6), (6, 1)]
sage: H = FundamentalGroupOfExtendedAffineWeylGroup(['D',5,1])
```

(continues on next page)

(continued from previous page)

```
sage: [(i, H.dual_node(i)) for i in H.special_nodes()]
[(0, 0), (1, 1), (4, 5), (5, 4)]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(4),Integer(1)])
>>> [(i, F.dual_node(i)) for i in F.special_nodes()]
[(0, 0), (1, 4), (2, 3), (3, 2), (4, 1)]
>>> G = FundamentalGroupOfExtendedAffineWeylGroup(['E',Integer(6),Integer(1)])
>>> [(i, G.dual_node(i)) for i in G.special_nodes()]
[(0, 0), (1, 6), (6, 1)]
>>> H = FundamentalGroupOfExtendedAffineWeylGroup(['D',Integer(5),Integer(1)])
>>> [(i, H.dual_node(i)) for i in H.special_nodes()]
[(0, 0), (1, 1), (4, 5), (5, 4)]
```

group_generators()

Return a tuple of generators of the fundamental group.

> **Warning**
>
> This returns the entire group, a necessary behavior because it is used in `__iter__()`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['E',6,1],prefix='f').group_
↪generators()
Finite family {0: f[0], 1: f[1], 6: f[6]}
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['E',Integer(6),Integer(1)],
↪prefix='f').group_generators()
Finite family {0: f[0], 1: f[1], 6: f[6]}
```

index_set()

The node set of the affine Cartan type of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1]).index_set()
(0, 1, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
```

(continues on next page)

(continued from previous page)

```
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).
↪index_set()
(0, 1, 2)
```

one()

Return the identity element of the fundamental group.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: F.one()
pi[0]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(3),Integer(1)])
>>> F.one()
pi[0]
```

product(x, y)

Return the product of x and y.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: F.special_nodes()
(0, 1, 2, 3)
sage: F(2)*F(3)
pi[1]
sage: F(1)*F(3)^(-1)
pi[2]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(3),Integer(1)])
>>> F.special_nodes()
(0, 1, 2, 3)
>>> F(Integer(2))*F(Integer(3))
pi[1]
>>> F(Integer(1))*F(Integer(3))**(-Integer(1))
pi[2]
```

reduced_word(i)

Return a reduced word for the finite Weyl group element associated with the i-th special automorphism.

More precisely, for each special node i, `self.reduced_word(i)` is a reduced word for the element v in the finite Weyl group such that in the extended affine Weyl group, the i-th special automorphism is equal to tv where t is a translation element.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: [(i, F.reduced_word(i)) for i in F.special_nodes()]
[(0, ()), (1, (1, 2, 3)), (2, (2, 1, 3, 2)), (3, (3, 2, 1))]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> F = FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(3),Integer(1)])
>>> [(i, F.reduced_word(i)) for i in F.special_nodes()]
[(0, ()), (1, (1, 2, 3)), (2, (2, 1, 3, 2)), (3, (3, 2, 1))]
```

special_nodes()

Return the special nodes of `self`.

See `sage.combinat.root_system.cartan_type.special_nodes()`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
sage: FundamentalGroupOfExtendedAffineWeylGroup(['D',4,1]).special_nodes()
(0, 1, 3, 4)
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1]).special_nodes()
(0, 1, 2)
sage: FundamentalGroupOfExtendedAffineWeylGroup(['C',3,1]).special_nodes()
(0, 3)
sage: FundamentalGroupOfExtendedAffineWeylGroup(['D',4,2]).special_nodes()
(0, 3)
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_
↪linear=True).special_nodes()
Integer Ring
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.fundamental_group import␣
↪FundamentalGroupOfExtendedAffineWeylGroup
>>> FundamentalGroupOfExtendedAffineWeylGroup(['D',Integer(4),Integer(1)]).
↪special_nodes()
(0, 1, 3, 4)
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)]).
↪special_nodes()
(0, 1, 2)
>>> FundamentalGroupOfExtendedAffineWeylGroup(['C',Integer(3),Integer(1)]).
↪special_nodes()
(0, 3)
>>> FundamentalGroupOfExtendedAffineWeylGroup(['D',Integer(4),Integer(2)]).
↪special_nodes()
(0, 3)
```

(continues on next page)

(continued from previous page)

```
>>> FundamentalGroupOfExtendedAffineWeylGroup(['A',Integer(2),Integer(1)],␣
↪general_linear=True).special_nodes()
Integer Ring
```

5.1.266 Root system data for folded Cartan types

AUTHORS:

- Travis Scrimshaw (2013-01-12) - Initial version

class sage.combinat.root_system.type_folded.**CartanTypeFolded**(*cartan_type*, *folding_of*, *orbit*)

Bases: UniqueRepresentation, SageObject

A Cartan type realized from a (Dynkin) diagram folding.

Given a Cartan type X, we say $\hat{X}$ is a folded Cartan type of X if there exists a diagram folding of the Dynkin diagram of $\hat{X}$ onto X.

A folding of a simply-laced Dynkin diagram D with index set I is an automorphism σ of D where all nodes any orbit of σ are not connected. The resulting Dynkin diagram $\hat{D}$ is induced by I/σ where we identify edges in $\hat{D}$ which are not incident and add a k-edge if we identify k incident edges and the arrow is pointing towards the indicent note. We denote the index set of $\hat{D}$ by $\hat{I}$, and by abuse of notation, we denote the folding by σ.

We also have scaling factors γ_i for $i \in \hat{I}$ and defined as the unique numbers such that the map $\Lambda_j \mapsto \gamma_j \sum_{i\in\sigma^{-1}(j)} \Lambda_i$ is the smallest proper embedding of the weight lattice of X to $\hat{X}$.

If the Cartan type is simply laced, the default folding is the one induced from the identity map on D.

If X is affine type, the default embeddings we consider here are:

$$\begin{array}{rcl} C_n^{(1)}, A_{2n}^{(2)}, A_{2n}^{(2)\dagger}, D_{n+1}^{(2)} & \hookrightarrow & A_{2n-1}^{(1)}, \\ A_{2n-1}^{(2)}, B_n^{(1)} & \hookrightarrow & D_{n+1}^{(1)}, \\ E_6^{(2)}, F_4^{(1)} & \hookrightarrow & E_6^{(1)}, \\ D_4^{(3)}, G_2^{(1)} & \hookrightarrow & D_4^{(1)}, \end{array}$$

and were chosen based on virtual crystals. In particular, the diagram foldings extend to crystal morphisms and gives a realization of Kirillov-Reshetikhin crystals for non-simply-laced types as simply-laced types. See [OSShimo03] and [FOS2009] for more details. Here we can compute $\gamma_i = \max(c)/c_i$ where $(c_i)_i$ are the translation factors of the root system. In a more type-dependent way, we can define γ_i as follows:

1. There exists a unique arrow (multiple bond) in X.
   a. Suppose the arrow points towards 0. Then $\gamma_i = 1$ for all $i \in I$.
   b. Otherwise γ_i is the order of σ for all i in the connected component of 0 after removing the arrow, else $\gamma_i = 1$.
2. There is not a unique arrow. Thus $\hat{X} = A_{2n-1}^{(1)}$ and $\gamma_i = 1$ for all $1 \leq i \leq n-1$. If $i \in \{0, n\}$, then $\gamma_i = 2$ if the arrow incident to i points away and is 1 otherwise.

We note that γ_i only depends upon X.

If the Cartan type is finite, then we consider the classical foldings/embeddings induced by the above affine foldings/embeddings:

$$\begin{aligned} C_n &\hookrightarrow A_{2n-1}, \\ B_n &\hookrightarrow D_{n+1}, \\ F_4 &\hookrightarrow E_6, \\ G_2 &\hookrightarrow D_4. \end{aligned}$$

For more information on Cartan types, see *sage.combinat.root_system.cartan_type*.

Other foldings may be constructed by passing in an optional `folding_of` second argument. See below.

INPUT:

- `cartan_type` – the Cartan type X to create the folded type
- `folding_of` – the Cartan type $\hat{X}$ which X is a folding of
- `orbit` – the orbit of the Dynkin diagram automorphism σ given as a list of lists where the a-th list corresponds to the a-th entry in I or a dictionary with keys in I and values as lists

Note

If X is an affine type, we assume the special node is fixed under σ.

EXAMPLES:

```
sage: fct = CartanType(['C',4,1]).as_folding(); fct
['C', 4, 1] as a folding of ['A', 7, 1]
sage: fct.scaling_factors()                                                 #␣
↪needs sage.graphs
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 2}
sage: fct.folding_orbit()
Finite family {0: (0,), 1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}
```

```
>>> from sage.all import *
>>> fct = CartanType(['C',Integer(4),Integer(1)]).as_folding(); fct
['C', 4, 1] as a folding of ['A', 7, 1]
>>> fct.scaling_factors()                                                 #␣
↪needs sage.graphs
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 2}
>>> fct.folding_orbit()
Finite family {0: (0,), 1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}
```

A simply laced Cartan type can be considered as a virtual type of itself:

```
sage: fct = CartanType(['A',4,1]).as_folding(); fct
['A', 4, 1] as a folding of ['A', 4, 1]
sage: fct.scaling_factors()                                                 #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
sage: fct.folding_orbit()
Finite family {0: (0,), 1: (1,), 2: (2,), 3: (3,), 4: (4,)}
```

```
>>> from sage.all import *
>>> fct = CartanType(['A',Integer(4),Integer(1)]).as_folding(); fct
['A', 4, 1] as a folding of ['A', 4, 1]
>>> fct.scaling_factors()                                                 #␣
↪needs sage.graphs
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
>>> fct.folding_orbit()
Finite family {0: (0,), 1: (1,), 2: (2,), 3: (3,), 4: (4,)}
```

Finite types:

```
sage: fct = CartanType(['C',4]).as_folding(); fct
['C', 4] as a folding of ['A', 7]
sage: fct.scaling_factors()
Finite family {1: 1, 2: 1, 3: 1, 4: 2}
sage: fct.folding_orbit()
Finite family {1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}

sage: fct = CartanType(['F',4]).dual().as_folding(); fct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1} as a folding of ['E', 6]
sage: fct.scaling_factors()
Finite family {1: 1, 2: 1, 3: 2, 4: 2}
sage: fct.folding_orbit()
Finite family {1: (1, 6), 2: (3, 5), 3: (4,), 4: (2,)}
```

```
>>> from sage.all import *
>>> fct = CartanType(['C',Integer(4)]).as_folding(); fct
['C', 4] as a folding of ['A', 7]
>>> fct.scaling_factors()
Finite family {1: 1, 2: 1, 3: 1, 4: 2}
>>> fct.folding_orbit()
Finite family {1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}

>>> fct = CartanType(['F',Integer(4)]).dual().as_folding(); fct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1} as a folding of ['E', 6]
>>> fct.scaling_factors()
Finite family {1: 1, 2: 1, 3: 2, 4: 2}
>>> fct.folding_orbit()
Finite family {1: (1, 6), 2: (3, 5), 3: (4,), 4: (2,)}
```

REFERENCES:

- Wikipedia article Dynkin_diagram#Folding

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: fct = CartanType(['C', 4, 1]).as_folding()
sage: fct.cartan_type()
['C', 4, 1]
```

```
>>> from sage.all import *
>>> fct = CartanType(['C', Integer(4), Integer(1)]).as_folding()
>>> fct.cartan_type()
['C', 4, 1]
```

folding_of()

Return the Cartan type of the virtual space.

EXAMPLES:

```
sage: fct = CartanType(['C', 4, 1]).as_folding()
sage: fct.folding_of()
['A', 7, 1]
```

```
>>> from sage.all import *
>>> fct = CartanType(['C', Integer(4), Integer(1)]).as_folding()
>>> fct.folding_of()
['A', 7, 1]
```

folding_orbit()

Return the orbits under the automorphism σ as a dictionary (of tuples).

EXAMPLES:

```
sage: fct = CartanType(['C', 4, 1]).as_folding()
sage: fct.folding_orbit()
Finite family {0: (0,), 1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}
```

```
>>> from sage.all import *
>>> fct = CartanType(['C', Integer(4), Integer(1)]).as_folding()
>>> fct.folding_orbit()
Finite family {0: (0,), 1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}
```

scaling_factors()

Return the scaling factors of `self`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: fct = CartanType(['C', 4, 1]).as_folding()
sage: fct.scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 2}
sage: fct = CartanType(['BC', 4, 2]).as_folding()
sage: fct.scaling_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 2}
sage: fct = CartanType(['BC', 4, 2]).dual().as_folding()
sage: fct.scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
sage: CartanType(['BC', 4, 2]).relabel({0:4, 1:3, 2:2, 3:1, 4:0}).as_
↪folding().scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> fct = CartanType(['C', Integer(4), Integer(1)]).as_folding()
>>> fct.scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 2}
>>> fct = CartanType(['BC', Integer(4), Integer(2)]).as_folding()
>>> fct.scaling_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 2}
>>> fct = CartanType(['BC', Integer(4), Integer(2)]).dual().as_folding()
>>> fct.scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
```

(continues on next page)

(continued from previous page)

```
>>> CartanType(['BC', Integer(4), Integer(2)]).relabel({Integer(0):Integer(4),
↪ Integer(1):Integer(3), Integer(2):Integer(2), Integer(3):Integer(1),␣
↪Integer(4):Integer(0)}).as_folding().scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
```

5.1.267 Root system data for Cartan types with marked nodes

class sage.combinat.root_system.type_marked.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

Ambient space for a marked finite Cartan type.

It is constructed in the canonical way from the ambient space of the original Cartan type.

EXAMPLES:

```
sage: L = CartanType(["F",4]).marked_nodes([1,3]).root_system().ambient_space(); L
Ambient space of the Root system of type ['F', 4] with nodes (1, 3) marked
sage: TestSuite(L).run()                                                        #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> L = CartanType(["F",Integer(4)]).marked_nodes([Integer(1),Integer(3)]).root_
↪system().ambient_space(); L
Ambient space of the Root system of type ['F', 4] with nodes (1, 3) marked
>>> TestSuite(L).run()                                                        #␣
↪needs sage.graphs
```

dimension()

Return the dimension of this ambient space.

See also

sage.combinat.root_system.ambient_space.AmbientSpace.dimension()

EXAMPLES:

```
sage: L = CartanType(["F",4]).marked_nodes([1,3]).root_system().ambient_
↪space()
sage: L.dimension()
4
```

```
>>> from sage.all import *
>>> L = CartanType(["F",Integer(4)]).marked_nodes([Integer(1),Integer(3)]).
↪root_system().ambient_space()
>>> L.dimension()
4
```

fundamental_weight(*i*)

Return the i-th fundamental weight.

It is constructed by looking up the corresponding simple coroot in the ambient space for the original Cartan type.

EXAMPLES:

```
sage: L = CartanType(["F",4]).marked_nodes([1,3]).root_system().ambient_
→space()
sage: L.fundamental_weight(1)
(1, 1, 0, 0)
sage: L.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0),
               3: (3/2, 1/2, 1/2, 1/2), 4: (1, 0, 0, 0)}
```

```
>>> from sage.all import *
>>> L = CartanType(["F",Integer(4)]).marked_nodes([Integer(1),Integer(3)]).
→root_system().ambient_space()
>>> L.fundamental_weight(Integer(1))
(1, 1, 0, 0)
>>> L.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0),
               3: (3/2, 1/2, 1/2, 1/2), 4: (1, 0, 0, 0)}
```

simple_root(*i*)

Return the `i`-th simple root.

It is constructed by looking up the corresponding simple coroot in the ambient space for the original Cartan type.

EXAMPLES:

```
sage: L = CartanType(["F",4]).marked_nodes([1,3]).root_system().ambient_
→space()
sage: L.simple_root(1)
(0, 1, -1, 0)
sage: L.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1),
               3: (0, 0, 0, 1), 4: (1/2, -1/2, -1/2, -1/2)}
sage: L.simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1),
               3: (0, 0, 0, 2), 4: (1, -1, -1, -1)}
```

```
>>> from sage.all import *
>>> L = CartanType(["F",Integer(4)]).marked_nodes([Integer(1),Integer(3)]).
→root_system().ambient_space()
>>> L.simple_root(Integer(1))
(0, 1, -1, 0)
>>> L.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1),
               3: (0, 0, 0, 1), 4: (1/2, -1/2, -1/2, -1/2)}
>>> L.simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1),
               3: (0, 0, 0, 2), 4: (1, -1, -1, -1)}
```

class sage.combinat.root_system.type_marked.**CartanType**(*ct*, *marked_nodes*)

Bases: *CartanType_decorator*

A class for Cartan types with marked nodes.

INPUT:

- `ct` – a Cartan type
- `marked_nodes` – list of marked nodes

EXAMPLES:

We take the Cartan type B_4:

```
sage: T = CartanType(['B',4])
sage: T.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O=>=O
1   2   3   4
B4
```

```
>>> from sage.all import *
>>> T = CartanType(['B',Integer(4)])
>>> T.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O=>=O
1   2   3   4
B4
```

And mark some of its nodes:

```
sage: T = T.marked_nodes([2,3])
sage: T.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---X---X=>=O
1   2   3   4
B4 with nodes (2, 3) marked
```

```
>>> from sage.all import *
>>> T = T.marked_nodes([Integer(2),Integer(3)])
>>> T.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---X---X=>=O
1   2   3   4
B4 with nodes (2, 3) marked
```

Markings are not additive:

```
sage: T.marked_nodes([1,4]).dynkin_diagram()                                #␣
↪needs sage.graphs
X---O---O=>=X
1   2   3   4
B4 with nodes (1, 4) marked
```

```
>>> from sage.all import *
>>> T.marked_nodes([Integer(1),Integer(4)]).dynkin_diagram()                ␣
↪              # needs sage.graphs
X---O---O=>=X
1   2   3   4
B4 with nodes (1, 4) marked
```

And trivial relabelling are honoured nicely:

```
sage: T = T.marked_nodes([])
sage: T.dynkin_diagram()                                                   #␣
↪needs sage.graphs
O---O---O=>=O
1   2   3   4
B4
```

```
>>> from sage.all import *
>>> T = T.marked_nodes([])
>>> T.dynkin_diagram()                                                   #␣
↪needs sage.graphs
O---O---O=>=O
1   2   3   4
B4
```

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of this Cartan type.

EXAMPLES:

```
sage: print(CartanType(["G", 2]).marked_nodes([2]).ascii_art())
  3
O=<=X
1   2
sage: print(CartanType(["B", 3, 1]).marked_nodes([0, 3]).ascii_art())
    X 0
    |
    |
O---O=>=X
1   2   3
sage: print(CartanType(["F", 4, 1]).marked_nodes([0, 2]).ascii_art())
X---O---X=>=O---O
0   1   2   3   4
```

```
>>> from sage.all import *
>>> print(CartanType(["G", Integer(2)]).marked_nodes([Integer(2)]).ascii_
↪art())
  3
O=<=X
1   2
>>> print(CartanType(["B", Integer(3), Integer(1)]).marked_nodes([Integer(0),␣
↪Integer(3)]).ascii_art())
    X 0
    |
    |
O---O=>=X
1   2   3
>>> print(CartanType(["F", Integer(4), Integer(1)]).marked_nodes([Integer(0),␣
↪Integer(2)]).ascii_art())
X---O---X=>=O---O
0   1   2   3   4
```

dual()

Implement *sage.combinat.root_system.cartan_type.CartanType_abstract.dual()*, using that taking the dual and marking nodes are commuting operations.

EXAMPLES:

```
sage: T = CartanType(["BC",3, 2])
sage: T.marked_nodes([1,3]).dual().dynkin_diagram()                          #␣
↪needs sage.graphs
O=>=X---O=>=X
0   1   2   3
BC3~* with nodes (1, 3) marked
sage: T.dual().marked_nodes([1,3]).dynkin_diagram()                          #␣
↪needs sage.graphs
O=>=X---O=>=X
0   1   2   3
BC3~* with nodes (1, 3) marked
```

```
>>> from sage.all import *
>>> T = CartanType(["BC",Integer(3), Integer(2)])
>>> T.marked_nodes([Integer(1),Integer(3)]).dual().dynkin_diagram()          ␣
↪                # needs sage.graphs
O=>=X---O=>=X
0   1   2   3
BC3~* with nodes (1, 3) marked
>>> T.dual().marked_nodes([Integer(1),Integer(3)]).dynkin_diagram()          ␣
↪                # needs sage.graphs
O=>=X---O=>=X
0   1   2   3
BC3~* with nodes (1, 3) marked
```

dynkin_diagram()

Return the Dynkin diagram for this Cartan type.

EXAMPLES:

```
sage: CartanType(["G", 2]).marked_nodes([2]).dynkin_diagram()                #␣
↪needs sage.graphs
  3
O=<=X
1   2
G2 with node 2 marked
```

```
>>> from sage.all import *
>>> CartanType(["G", Integer(2)]).marked_nodes([Integer(2)]).dynkin_diagram()␣
↪                # needs sage.graphs
  3
O=<=X
1   2
G2 with node 2 marked
```

marked_nodes(*marked_nodes*)

Return `self` with nodes `marked_nodes` marked.

EXAMPLES:

```
sage: ct = CartanType(['A',12])
sage: m = ct.marked_nodes([1,4,6,7,8,12]); m
['A', 12] with nodes (1, 4, 6, 7, 8, 12) marked
sage: m.marked_nodes([2])
['A', 12] with node 2 marked
sage: m.marked_nodes([]) is ct
True
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(12)])
>>> m = ct.marked_nodes([Integer(1),Integer(4),Integer(6),Integer(7),
↪Integer(8),Integer(12)]); m
['A', 12] with nodes (1, 4, 6, 7, 8, 12) marked
>>> m.marked_nodes([Integer(2)])
['A', 12] with node 2 marked
>>> m.marked_nodes([]) is ct
True
```

relabel(*relabelling*)

Return the relabelling of `self`.

EXAMPLES:

```
sage: T = CartanType(["BC",3, 2])
sage: T.marked_nodes([1,3]).relabel(lambda x: x+2).dynkin_diagram()          #␣
↪needs sage.graphs
O=<=X---O=<=X
2   3   4   5
BC3~ relabelled by {0: 2, 1: 3, 2: 4, 3: 5} with nodes (3, 5) marked
sage: T.relabel(lambda x: x+2).marked_nodes([3,5]).dynkin_diagram()          #␣
↪needs sage.graphs
O=<=X---O=<=X
2   3   4   5
BC3~ relabelled by {0: 2, 1: 3, 2: 4, 3: 5} with nodes (3, 5) marked
```

```
>>> from sage.all import *
>>> T = CartanType(["BC",Integer(3), Integer(2)])
>>> T.marked_nodes([Integer(1),Integer(3)]).relabel(lambda x: x+Integer(2)).
↪dynkin_diagram()          # needs sage.graphs
O=<=X---O=<=X
2   3   4   5
BC3~ relabelled by {0: 2, 1: 3, 2: 4, 3: 5} with nodes (3, 5) marked
>>> T.relabel(lambda x: x+Integer(2)).marked_nodes([Integer(3),Integer(5)]).
↪dynkin_diagram()          # needs sage.graphs
O=<=X---O=<=X
2   3   4   5
BC3~ relabelled by {0: 2, 1: 3, 2: 4, 3: 5} with nodes (3, 5) marked
```

type()

Return the type of `self` or `None` if unknown.

EXAMPLES:

```
sage: ct = CartanType(['F', 4]).marked_nodes([1,3])
sage: ct.type()
'F'
```

```
>>> from sage.all import *
>>> ct = CartanType(['F', Integer(4)]).marked_nodes([Integer(1),Integer(3)])
>>> ct.type()
'F'
```

class sage.combinat.root_system.type_marked.**CartanType_affine**(*ct*, *marked_nodes*)

Bases: *CartanType*, *CartanType_affine*

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain self.

EXAMPLES:

```
sage: CartanType(['A', 7, 2]).marked_nodes([1,3]).basic_untwisted()
['A', 7] with nodes (1, 3) marked
sage: CartanType(['D', 4, 3]).marked_nodes([0,2]).basic_untwisted()
['D', 4] with node 2 marked
```

```
>>> from sage.all import *
>>> CartanType(['A', Integer(7), Integer(2)]).marked_nodes([Integer(1),
↪Integer(3)]).basic_untwisted()
['A', 7] with nodes (1, 3) marked
>>> CartanType(['D', Integer(4), Integer(3)]).marked_nodes([Integer(0),
↪Integer(2)]).basic_untwisted()
['D', 4] with node 2 marked
```

classical()

Return the classical Cartan type associated with self.

EXAMPLES:

```
sage: T = CartanType(['A',4,1]).marked_nodes([0,2,4])
sage: T.dynkin_diagram()                                                  #␣
↪needs sage.graphs
0
X-----------+
|           |
|           |
O---X---O---X
1   2   3   4
A4~ with nodes (0, 2, 4) marked

sage: T0 = T.classical(); T0
['A', 4] with nodes (2, 4) marked
sage: T0.dynkin_diagram()                                                 #␣
↪needs sage.graphs
O---X---O---X
```

(continues on next page)

(continued from previous page)

```
1   2   3   4
A4 with nodes (2, 4) marked
```

```
>>> from sage.all import *
>>> T = CartanType(['A',Integer(4),Integer(1)]).marked_nodes([Integer(0),
↪Integer(2),Integer(4)])
>>> T.dynkin_diagram()                                                   #␣
↪needs sage.graphs
0
X-----------+
|           |
|           |
O---X---O---X
1   2   3   4
A4~ with nodes (0, 2, 4) marked

>>> T0 = T.classical(); T0
['A', 4] with nodes (2, 4) marked
>>> T0.dynkin_diagram()                                                  #␣
↪needs sage.graphs
O---X---O---X
1   2   3   4
A4 with nodes (2, 4) marked
```

is_untwisted_affine()

Implement *CartanType_affine.is_untwisted_affine()*.

A marked Cartan type is untwisted affine if the original is.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).marked_nodes([1,3]).is_untwisted_affine()
True
```

```
>>> from sage.all import *
>>> CartanType(['B', Integer(3), Integer(1)]).marked_nodes([Integer(1),
↪Integer(3)]).is_untwisted_affine()
True
```

special_node()

Return the special node of the Cartan type.

See also

special_node()

It is the special node of the non-marked Cartan type..

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).marked_nodes([1,3]).special_node()
0
```

```
>>> from sage.all import *
>>> CartanType(['B', Integer(3), Integer(1)]).marked_nodes([Integer(1),
↪Integer(3)]).special_node()
0
```

class sage.combinat.root_system.type_marked.**CartanType_finite**(*ct*, *marked_nodes*)

Bases: *CartanType*, *CartanType_finite*

AmbientSpace

alias of *AmbientSpace*

affine()

Return the affine Cartan type associated with `self`.

EXAMPLES:

```
sage: B4 = CartanType(['B',4]).marked_nodes([1,3])
sage: B4.dynkin_diagram()                                              #␣
↪needs sage.graphs
X---O---X=>=O
1   2   3   4
B4 with nodes (1, 3) marked
sage: B4.affine().dynkin_diagram()                                     #␣
↪needs sage.graphs
    O 0
    |
    |
X---O---X=>=O
1   2   3   4
B4~ with nodes (1, 3) marked
```

```
>>> from sage.all import *
>>> B4 = CartanType(['B',Integer(4)]).marked_nodes([Integer(1),Integer(3)])
>>> B4.dynkin_diagram()                                              #␣
↪needs sage.graphs
X---O---X=>=O
1   2   3   4
B4 with nodes (1, 3) marked
>>> B4.affine().dynkin_diagram()                                     #␣
↪needs sage.graphs
    O 0
    |
    |
X---O---X=>=O
1   2   3   4
B4~ with nodes (1, 3) marked
```

5.1.268 Root system data for reducible Cartan types

class sage.combinat.root_system.type_reducible.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space()
Ambient space of the Root system of type A2xB2
```

```
>>> from sage.all import *
>>> RootSystem("A2xB2").ambient_space()
Ambient space of the Root system of type A2xB2
```

ambient_spaces()

Return a list of the irreducible Cartan types of which the given reducible Cartan type is a product.

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().ambient_spaces()
[Ambient space of the Root system of type ['A', 2],
 Ambient space of the Root system of type ['B', 2]]
```

```
>>> from sage.all import *
>>> RootSystem("A2xB2").ambient_space().ambient_spaces()
[Ambient space of the Root system of type ['A', 2],
 Ambient space of the Root system of type ['B', 2]]
```

cartan_type()

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().cartan_type()
A2xB2
```

```
>>> from sage.all import *
>>> RootSystem("A2xB2").ambient_space().cartan_type()
A2xB2
```

component_types()

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().component_types()
[['A', 2], ['B', 2]]
```

```
>>> from sage.all import *
>>> RootSystem("A2xB2").ambient_space().component_types()
[['A', 2], ['B', 2]]
```

dimension()

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().dimension()
5
```

```
>>> from sage.all import *
>>> RootSystem("A2xB2").ambient_space().dimension()
5
```

fundamental_weights()

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0, 0), 2: (1, 1, 0, 0, 0), 3: (0, 0, 0, 1, 0), 4:␣
↪(0, 0, 0, 1/2, 1/2)}
```

```
>>> from sage.all import *
>>> RootSystem("A2xB2").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0, 0), 2: (1, 1, 0, 0, 0), 3: (0, 0, 0, 1, 0), 4:␣
↪(0, 0, 0, 1/2, 1/2)}
```

inject_weights(i, v)

Produces the corresponding element of the lattice.

INPUT:

- `i` – integer in `range(self.components)`
- `v` – a vector in the i-th component weight lattice

EXAMPLES:

```
sage: V = RootSystem("A2xB2").ambient_space()
sage: [V.inject_weights(i,V.ambient_spaces()[i].fundamental_weights()[1]) for␣
↪i in range(2)]
[(1, 0, 0, 0, 0), (0, 0, 0, 1, 0)]
sage: [V.inject_weights(i,V.ambient_spaces()[i].fundamental_weights()[2]) for␣
↪i in range(2)]
[(1, 1, 0, 0, 0), (0, 0, 0, 1/2, 1/2)]
```

```
>>> from sage.all import *
>>> V = RootSystem("A2xB2").ambient_space()
>>> [V.inject_weights(i,V.ambient_spaces()[i].fundamental_
↪weights()[Integer(1)]) for i in range(Integer(2))]
[(1, 0, 0, 0, 0), (0, 0, 0, 1, 0)]
>>> [V.inject_weights(i,V.ambient_spaces()[i].fundamental_
↪weights()[Integer(2)]) for i in range(Integer(2))]
[(1, 1, 0, 0, 0), (0, 0, 0, 1/2, 1/2)]
```

negative_roots()

EXAMPLES:

```
sage: RootSystem("A1xA2").ambient_space().negative_roots()
[(-1, 1, 0, 0, 0), (0, 0, -1, 1, 0), (0, 0, -1, 0, 1), (0, 0, 0, -1, 1)]
```

```
>>> from sage.all import *
>>> RootSystem("A1xA2").ambient_space().negative_roots()
[(-1, 1, 0, 0, 0), (0, 0, -1, 1, 0), (0, 0, -1, 0, 1), (0, 0, 0, -1, 1)]
```

positive_roots()

EXAMPLES:

```
sage: RootSystem("A1xA2").ambient_space().positive_roots()
[(1, -1, 0, 0, 0), (0, 0, 1, -1, 0), (0, 0, 1, 0, -1), (0, 0, 0, 1, -1)]
```

```
>>> from sage.all import *
>>> RootSystem("A1xA2").ambient_space().positive_roots()
[(1, -1, 0, 0, 0), (0, 0, 1, -1, 0), (0, 0, 1, 0, -1), (0, 0, 0, 1, -1)]
```

simple_coroot(*i*)

EXAMPLES:

```
sage: A = RootSystem("A1xB2").ambient_space()
sage: A.simple_coroot(2)
(0, 0, 1, -1)
sage: A.simple_coroots()
Finite family {1: (1, -1, 0, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2)}
```

```
>>> from sage.all import *
>>> A = RootSystem("A1xB2").ambient_space()
>>> A.simple_coroot(Integer(2))
(0, 0, 1, -1)
>>> A.simple_coroots()
Finite family {1: (1, -1, 0, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2)}
```

simple_root(*i*)

EXAMPLES:

```
sage: A = RootSystem("A1xB2").ambient_space()
sage: A.simple_root(2)
(0, 0, 1, -1)
sage: A.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1)}
```

```
>>> from sage.all import *
>>> A = RootSystem("A1xB2").ambient_space()
>>> A.simple_root(Integer(2))
(0, 0, 1, -1)
>>> A.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1)}
```

class sage.combinat.root_system.type_reducible.**CartanType**(*types*)

Bases: `SageObject`, `CartanType_abstract`

A class for reducible Cartan types.

Reducible root systems are ones that can be factored as direct products. Strictly speaking type D_2 (corresponding to orthogonal groups of degree 4) is reducible since it is isomorphic to $A_1 \times A_1$. However type D_2 is not built using this class for our purposes.

INPUT:

- `types` – list of simple Cartan types

EXAMPLES:

```
sage: t1, t2 = [CartanType(x) for x in (['A',1], ['B',2])]
sage: CartanType([t1, t2])
A1xB2
sage: t = CartanType("A2xB2")
```

```
>>> from sage.all import *
>>> t1, t2 = [CartanType(x) for x in (['A',Integer(1)], ['B',Integer(2)])]
>>> CartanType([t1, t2])
A1xB2
>>> t = CartanType("A2xB2")
```

A reducible Cartan type is finite (resp. crystallographic, simply laced) if all its components are:

```
sage: t.is_finite()
True
sage: t.is_crystallographic()
True
sage: t.is_simply_laced()
False
```

```
>>> from sage.all import *
>>> t.is_finite()
True
>>> t.is_crystallographic()
True
>>> t.is_simply_laced()
False
```

This is implemented by inserting the appropriate abstract super classes (see `_add_abstract_superclass()`):

```
sage: t.__class__.mro()
[<class 'sage.combinat.root_system.type_reducible.CartanType_with_superclass'>,
↪<class 'sage.combinat.root_system.type_reducible.CartanType'>, <class 'sage.
↪structure.sage_object.SageObject'>, <class 'sage.combinat.root_system.cartan_
↪type.CartanType_finite'>, <class 'sage.combinat.root_system.cartan_type.
↪CartanType_crystallographic'>, <class 'sage.combinat.root_system.cartan_type.
↪CartanType_abstract'>, <class 'object'>]
```

```
>>> from sage.all import *
>>> t.__class__.mro()
[<class 'sage.combinat.root_system.type_reducible.CartanType_with_superclass'>,
↪<class 'sage.combinat.root_system.type_reducible.CartanType'>, <class 'sage.
↪structure.sage_object.SageObject'>, <class 'sage.combinat.root_system.cartan_
↪type.CartanType_finite'>, <class 'sage.combinat.root_system.cartan_type.
↪CartanType_crystallographic'>, <class 'sage.combinat.root_system.cartan_type.
↪CartanType_abstract'>, <class 'object'>]
```

The index set of the reducible Cartan type is obtained by relabelling successively the nodes of the Dynkin diagrams of the components by 1,2,…:

```
sage: t = CartanType(["A",4], ["BC",5,2], ["C",3])
sage: t.index_set()
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)

sage: t.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
```

(continues on next page)

(continued from previous page)

```
O=<=O---O---O---O=<=O
5   6   7   8   9   10
O---O=<=O
11   12   13
A4xBC5~xC3
```

```
>>> from sage.all import *
>>> t = CartanType(["A",Integer(4)], ["BC",Integer(5),Integer(2)], ["C",
↪Integer(3)])
>>> t.index_set()
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)

>>> t.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O---O
1   2   3   4
O=<=O---O---O---O=<=O
5   6   7   8   9   10
O---O=<=O
11   12   13
A4xBC5~xC3
```

AmbientSpace

alias of *AmbientSpace*

ascii_art (*label=None*, *node=None*)

Return an ascii art representation of this reducible Cartan type.

EXAMPLES:

```
sage: print(CartanType("F4xA2").ascii_art(label = lambda x: x+2))
O---O=>=O---O
3   4   5   6
O---O
7   8

sage: print(CartanType(["BC",5,2], ["A",4]).ascii_art())
O=<=O---O---O---O=<=O
1   2   3   4   5   6
O---O---O---O
7   8   9   10

sage: print(CartanType(["A",4], ["BC",5,2], ["C",3]).ascii_art())
O---O---O---O
1   2   3   4
O=<=O---O---O---O=<=O
5   6   7   8   9   10
O---O=<=O
11   12   13
```

```
>>> from sage.all import *
>>> print(CartanType("F4xA2").ascii_art(label = lambda x: x+Integer(2)))
```

(continues on next page)

(continued from previous page)

```
O---O=>=O---O
3   4   5   6
O---O
7   8

>>> print(CartanType(["BC",Integer(5),Integer(2)], ["A",Integer(4)]).ascii_
↪art())
O=<=O---O---O---O=<=O
1   2   3   4   5   6
O---O---O---O
7   8   9   10

>>> print(CartanType(["A",Integer(4)], ["BC",Integer(5),Integer(2)], ["C",
↪Integer(3)]).ascii_art())
O---O---O---O
1   2   3   4
O=<=O---O---O---O=<=O
5   6   7   8   9   10
O---O=<=O
11   12   13
```

cartan_matrix(*subdivide=True*)

Return the Cartan matrix associated with `self`. By default the Cartan matrix is a subdivided block matrix showing the reducibility but the subdivision can be suppressed with the option `subdivide = False`.

EXAMPLES:

```
sage: ct = CartanType("A2","B2")
sage: ct.cartan_matrix()                                                    #␣
↪needs sage.graphs
[ 2 -1| 0  0]
[-1  2| 0  0]
[-----+-----]
[ 0  0| 2 -1]
[ 0  0|-2  2]
sage: ct.cartan_matrix(subdivide=False)                                     #␣
↪needs sage.graphs
[ 2 -1  0  0]
[-1  2  0  0]
[ 0  0  2 -1]
[ 0  0 -2  2]
sage: ct.index_set() == ct.cartan_matrix().index_set()                      #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> ct = CartanType("A2","B2")
>>> ct.cartan_matrix()                                                    #␣
↪needs sage.graphs
[ 2 -1| 0  0]
[-1  2| 0  0]
[-----+-----]
```

(continues on next page)

(continued from previous page)

```
[ 0  0| 2 -1]
[ 0  0|-2  2]
>>> ct.cartan_matrix(subdivide=False)                                    #␣
↪needs sage.graphs
[ 2 -1  0  0]
[-1  2  0  0]
[ 0  0  2 -1]
[ 0  0 -2  2]
>>> ct.index_set() == ct.cartan_matrix().index_set()                     #␣
↪needs sage.graphs
True
```

component_types()

A list of Cartan types making up the reducible type.

EXAMPLES:

```
sage: CartanType(['A',2],['B',2]).component_types()
[['A', 2], ['B', 2]]
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(2)],['B',Integer(2)]).component_types()
[['A', 2], ['B', 2]]
```

coxeter_diagram()

Return the Coxeter diagram for `self`.

EXAMPLES:

```
sage: cd = CartanType("A2xB2xF4").coxeter_diagram(); cd                    #␣
↪needs sage.graphs
Graph on 8 vertices
sage: cd.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(1, 2, 3), (3, 4, 4), (5, 6, 3), (6, 7, 4), (7, 8, 3)]

sage: CartanType("F4xA2").coxeter_diagram().edges(sort=True)               #␣
↪needs sage.graphs
[(1, 2, 3), (2, 3, 4), (3, 4, 3), (5, 6, 3)]

sage: cd = CartanType("A1xH3").coxeter_diagram(); cd                       #␣
↪needs sage.graphs
Graph on 4 vertices
sage: cd.edges(sort=True)                                                  #␣
↪needs sage.graphs
[(2, 3, 3), (3, 4, 5)]
```

```
>>> from sage.all import *
>>> cd = CartanType("A2xB2xF4").coxeter_diagram(); cd                     #␣
↪needs sage.graphs
Graph on 8 vertices
>>> cd.edges(sort=True)                                                   #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[(1, 2, 3), (3, 4, 4), (5, 6, 3), (6, 7, 4), (7, 8, 3)]

>>> CartanType("F4xA2").coxeter_diagram().edges(sort=True)                     #␣
↪needs sage.graphs
[(1, 2, 3), (2, 3, 4), (3, 4, 3), (5, 6, 3)]

>>> cd = CartanType("A1xH3").coxeter_diagram(); cd                             #␣
↪needs sage.graphs
Graph on 4 vertices
>>> cd.edges(sort=True)                                                        #␣
↪needs sage.graphs
[(2, 3, 3), (3, 4, 5)]
```

dual()

EXAMPLES:

```
sage: CartanType("A2xB2").dual()
A2xC2
```

```
>>> from sage.all import *
>>> CartanType("A2xB2").dual()
A2xC2
```

dynkin_diagram()

Return a Dynkin diagram for type reducible.

EXAMPLES:

```
sage: dd = CartanType("A2xB2xF4").dynkin_diagram(); dd                          #␣
↪needs sage.graphs
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4
sage: dd.edges(sort=True)                                                       #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (3, 4, 2), (4, 3, 1), (5, 6, 1),
 (6, 5, 1), (6, 7, 2), (7, 6, 1), (7, 8, 1), (8, 7, 1)]

sage: CartanType("F4xA2").dynkin_diagram()                                      #␣
↪needs sage.graphs
O---O=>=O---O
1   2   3   4
O---O
5   6
F4xA2
```

```
>>> from sage.all import *
>>> dd = CartanType("A2xB2xF4").dynkin_diagram(); dd                    #␣
↪needs sage.graphs
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4
>>> dd.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(1, 2, 1), (2, 1, 1), (3, 4, 2), (4, 3, 1), (5, 6, 1),
 (6, 5, 1), (6, 7, 2), (7, 6, 1), (7, 8, 1), (8, 7, 1)]

>>> CartanType("F4xA2").dynkin_diagram()                                #␣
↪needs sage.graphs
O---O=>=O---O
1   2   3   4
O---O
5   6
F4xA2
```

index_set()

Implement *CartanType_abstract.index_set()*.

For the moment, the index set is always of the form $\{1, \ldots, n\}$.

EXAMPLES:

```
sage: CartanType("A2","A1").index_set()
(1, 2, 3)
```

```
>>> from sage.all import *
>>> CartanType("A2","A1").index_set()
(1, 2, 3)
```

is_affine()

Report that this reducible Cartan type is not affine.

EXAMPLES:

```
sage: CartanType(['A',2],['B',2]).is_affine()
False
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(2)],['B',Integer(2)]).is_affine()
False
```

is_finite()

EXAMPLES:

```
sage: ct1 = CartanType(['A',2],['B',2])
sage: ct1.is_finite()
```

(continues on next page)

(continued from previous page)

```
True
sage: ct2 = CartanType(['A',2],['B',2,1])
sage: ct2.is_finite()
False
```

```
>>> from sage.all import *
>>> ct1 = CartanType(['A',Integer(2)],['B',Integer(2)])
>>> ct1.is_finite()
True
>>> ct2 = CartanType(['A',Integer(2)],['B',Integer(2),Integer(1)])
>>> ct2.is_finite()
False
```

is_irreducible()

Report that this Cartan type is not irreducible.

EXAMPLES:

```
sage: ct = CartanType(['A',2],['B',2])
sage: ct.is_irreducible()
False
```

```
>>> from sage.all import *
>>> ct = CartanType(['A',Integer(2)],['B',Integer(2)])
>>> ct.is_irreducible()
False
```

rank()

Return the rank of `self`.

EXAMPLES:

```
sage: CartanType("A2","A1").rank()
3
```

```
>>> from sage.all import *
>>> CartanType("A2","A1").rank()
3
```

type()

Return `"reducible"` since the type is reducible.

EXAMPLES:

```
sage: CartanType(['A',2],['B',2]).type()
'reducible'
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(2)],['B',Integer(2)]).type()
'reducible'
```

5.1.269 Root system data for relabelled Cartan types

class sage.combinat.root_system.type_relabel.**AmbientSpace**(*root_system*, *base_ring*, *index_set=None*)

Bases: *AmbientSpace*

Ambient space for a relabelled finite Cartan type.

It is constructed in the canonical way from the ambient space of the original Cartan type, by relabelling the simple roots, fundamental weights, etc.

EXAMPLES:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}
sage: L = CartanType(["F",4]).relabel(cycle).root_system().ambient_space(); L
Ambient space of the Root system of type ['F', 4] relabelled by {1: 2, 2: 3, 3: 4,
↪ 4: 1}
sage: TestSuite(L).run()                                                    #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> cycle = {Integer(1):Integer(2), Integer(2):Integer(3), Integer(3):Integer(4),␣
↪Integer(4):Integer(1)}
>>> L = CartanType(["F",Integer(4)]).relabel(cycle).root_system().ambient_space();
↪ L
Ambient space of the Root system of type ['F', 4] relabelled by {1: 2, 2: 3, 3: 4,
↪ 4: 1}
>>> TestSuite(L).run()                                                    #␣
↪needs sage.graphs
```

dimension()

Return the dimension of this ambient space.

See also

sage.combinat.root_system.ambient_space.AmbientSpace.dimension()

EXAMPLES:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}
sage: L = CartanType(["F",4]).relabel(cycle).root_system().ambient_space()
sage: L.dimension()
4
```

```
>>> from sage.all import *
>>> cycle = {Integer(1):Integer(2), Integer(2):Integer(3),␣
↪Integer(3):Integer(4), Integer(4):Integer(1)}
>>> L = CartanType(["F",Integer(4)]).relabel(cycle).root_system().ambient_
↪space()
>>> L.dimension()
4
```

fundamental_weight(*i*)

Return the `i`-th fundamental weight.

It is constructed by looking up the corresponding simple coroot in the ambient space for the original Cartan type.

EXAMPLES:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}
sage: L = CartanType(["F",4]).relabel(cycle).root_system().ambient_space()
sage: K = CartanType(["F",4]).root_system().ambient_space()
sage: K.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4:␣
↪(1, 0, 0, 0)}
sage: L.fundamental_weight(1)
(1, 0, 0, 0)
sage: L.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (2, 1, 1, 0), 4: (3/2, 1/
↪2, 1/2, 1/2)}
```

```
>>> from sage.all import *
>>> cycle = {Integer(1):Integer(2), Integer(2):Integer(3),␣
↪Integer(3):Integer(4), Integer(4):Integer(1)}
>>> L = CartanType(["F",Integer(4)]).relabel(cycle).root_system().ambient_
↪space()
>>> K = CartanType(["F",Integer(4)]).root_system().ambient_space()
>>> K.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4:␣
↪(1, 0, 0, 0)}
>>> L.fundamental_weight(Integer(1))
(1, 0, 0, 0)
>>> L.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (2, 1, 1, 0), 4: (3/2, 1/
↪2, 1/2, 1/2)}
```

simple_root(*i*)

Return the i-th simple root.

It is constructed by looking up the corresponding simple coroot in the ambient space for the original Cartan type.

EXAMPLES:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}
sage: L = CartanType(["F",4]).relabel(cycle).root_system().ambient_space()
sage: K = CartanType(["F",4]).root_system().ambient_space()
sage: K.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1), 4: (1/2, -
↪1/2, -1/2, -1/2)}
sage: K.simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1,
↪ -1, -1)}
sage: L.simple_root(1)
(1/2, -1/2, -1/2, -1/2)

sage: L.simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1),
↪ 4: (0, 0, 0, 1)}
```

(continues on next page)

(continued from previous page)

```
sage: L.simple_coroots()
Finite family {1: (1, -1, -1, -1), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0,␣
↪0, 0, 2)}
```

```
>>> from sage.all import *
>>> cycle = {Integer(1):Integer(2), Integer(2):Integer(3),␣
↪Integer(3):Integer(4), Integer(4):Integer(1)}
>>> L = CartanType(["F",Integer(4)]).relabel(cycle).root_system().ambient_
↪space()
>>> K = CartanType(["F",Integer(4)]).root_system().ambient_space()
>>> K.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1), 4: (1/2, -
↪1/2, -1/2, -1/2)}
>>> K.simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1,
↪ -1, -1)}
>>> L.simple_root(Integer(1))
(1/2, -1/2, -1/2, -1/2)

>>> L.simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1),
↪ 4: (0, 0, 0, 1)}

>>> L.simple_coroots()
Finite family {1: (1, -1, -1, -1), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0,␣
↪0, 0, 2)}
```

class sage.combinat.root_system.type_relabel.**CartanType**(*type*, *relabelling*)

Bases: *CartanType_decorator*

A class for relabelled Cartan types.

ascii_art(*label=None*, *node=None*)

Return an ascii art representation of this Cartan type.

EXAMPLES:

```
sage: print(CartanType(["G", 2]).relabel({1:2,2:1}).ascii_art())
  3
O=<=O
2   1
sage: print(CartanType(["B", 3, 1]).relabel([1,3,2,0]).ascii_art())
    O 1
    |
    |
O---O=>=O
3   2   0
sage: print(CartanType(["F", 4, 1]).relabel(lambda n: 4-n).ascii_art())
O---O---O=>=O---O
4   3   2   1   0
```

```
>>> from sage.all import *
>>> print(CartanType(["G", Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)}).ascii_art())
  3
O=<=O
2   1
>>> print(CartanType(["B", Integer(3), Integer(1)]).relabel([Integer(1),
↪Integer(3),Integer(2),Integer(0)]).ascii_art())
    O 1
    |
    |
O---O=>=O
3   2   0
>>> print(CartanType(["F", Integer(4), Integer(1)]).relabel(lambda n:␣
↪Integer(4)-n).ascii_art())
O---O---O=>=O---O
4   3   2   1   0
```

coxeter_diagram()

Return the Coxeter diagram for `self`.

EXAMPLES:

```
sage: ct = CartanType(['H', 3]).relabel({1:3,2:2,3:1})
sage: G = ct.coxeter_diagram(); G                                        #␣
↪needs sage.graphs
Graph on 3 vertices
sage: G.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(1, 2, 5), (2, 3, 3)]
```

```
>>> from sage.all import *
>>> ct = CartanType(['H', Integer(3)]).relabel({Integer(1):Integer(3),
↪Integer(2):Integer(2),Integer(3):Integer(1)})
>>> G = ct.coxeter_diagram(); G                                        #␣
↪needs sage.graphs
Graph on 3 vertices
>>> G.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(1, 2, 5), (2, 3, 3)]
```

dual()

Implement *sage.combinat.root_system.cartan_type.CartanType_abstract.dual()*, using that taking the dual and relabelling are commuting operations.

EXAMPLES:

```
sage: T = CartanType(["BC",3, 2])
sage: cycle = {1:2, 2:3, 3:0, 0:1}
sage: T.relabel(cycle).dual().dynkin_diagram()                           #␣
↪needs sage.graphs
O=>=O---O=>=O
1   2   3   0
```

(continues on next page)

(continued from previous page)

```
BC3~* relabelled by {0: 1, 1: 2, 2: 3, 3: 0}
sage: T.dual().relabel(cycle).dynkin_diagram()                          #␣
↪needs sage.graphs
O=>=O---O=>=O
1   2   3   0
BC3~* relabelled by {0: 1, 1: 2, 2: 3, 3: 0}
```

```
>>> from sage.all import *
>>> T = CartanType(["BC",Integer(3), Integer(2)])
>>> cycle = {Integer(1):Integer(2), Integer(2):Integer(3),␣
↪Integer(3):Integer(0), Integer(0):Integer(1)}
>>> T.relabel(cycle).dual().dynkin_diagram()                          #␣
↪needs sage.graphs
O=>=O---O=>=O
1   2   3   0
BC3~* relabelled by {0: 1, 1: 2, 2: 3, 3: 0}
>>> T.dual().relabel(cycle).dynkin_diagram()                          #␣
↪needs sage.graphs
O=>=O---O=>=O
1   2   3   0
BC3~* relabelled by {0: 1, 1: 2, 2: 3, 3: 0}
```

dynkin_diagram()

Return the Dynkin diagram for this Cartan type.

EXAMPLES:

```
sage: CartanType(["G", 2]).relabel({1:2,2:1}).dynkin_diagram()          #␣
↪needs sage.graphs
  3
O=<=O
2   1
G2 relabelled by {1: 2, 2: 1}
```

```
>>> from sage.all import *
>>> CartanType(["G", Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)}).dynkin_diagram()          # needs sage.graphs
  3
O=<=O
2   1
G2 relabelled by {1: 2, 2: 1}
```

index_set()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.index_set()
(1, 2)
```

```
>>> from sage.all import *
>>> ct = CartanType(['G', Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)})
```

(continues on next page)

(continued from previous page)

```
>>> ct.index_set()
(1, 2)
```

type()

Return the type of `self` or `None` if unknown.

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.type()
'G'
```

```
>>> from sage.all import *
>>> ct = CartanType(['G', Integer(2)]).relabel({Integer(1):Integer(2),
↪Integer(2):Integer(1)})
>>> ct.type()
'G'
```

class sage.combinat.root_system.type_relabel.**CartanType_affine**(*type*, *relabelling*)

Bases: *CartanType*, *CartanType_affine*

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: ct = CartanType(['A', 5, 2]).relabel({0:1, 1:0, 2:2, 3:3})
sage: ct.basic_untwisted()
['A', 5]
```

```
>>> from sage.all import *
>>> ct = CartanType(['A', Integer(5), Integer(2)]).relabel(
↪{Integer(0):Integer(1), Integer(1):Integer(0), Integer(2):Integer(2),␣
↪Integer(3):Integer(3)})
>>> ct.basic_untwisted()
['A', 5]
```

classical()

Return the classical Cartan type associated with `self`.

EXAMPLES:

```
sage: A41 = CartanType(['A',4,1])
sage: A41.dynkin_diagram()                                              #␣
↪needs sage.graphs
0
O-----------+
|           |
|           |
O---O---O---O
```

(continues on next page)

(continued from previous page)

```
1   2   3   4
A4~

sage: T = A41.relabel({0:1, 1:2, 2:3, 3:4, 4:0})
sage: T
['A', 4, 1] relabelled by {0: 1, 1: 2, 2: 3, 3: 4, 4: 0}
sage: T.dynkin_diagram()                                                    #␣
↪needs sage.graphs
1
O-----------+
|           |
|           |
O---O---O---O
2   3   4   0
A4~ relabelled by {0: 1, 1: 2, 2: 3, 3: 4, 4: 0}

sage: T0 = T.classical()
sage: T0
['A', 4] relabelled by {1: 2, 2: 3, 3: 4, 4: 0}
sage: T0.dynkin_diagram()                                                   #␣
↪needs sage.graphs
O---O---O---O
2   3   4   0
A4 relabelled by {1: 2, 2: 3, 3: 4, 4: 0}
```

```
>>> from sage.all import *
>>> A41 = CartanType(['A',Integer(4),Integer(1)])
>>> A41.dynkin_diagram()                                                  #␣
↪needs sage.graphs
0
O-----------+
|           |
|           |
O---O---O---O
1   2   3   4
A4~

>>> T = A41.relabel({Integer(0):Integer(1), Integer(1):Integer(2),␣
↪Integer(2):Integer(3), Integer(3):Integer(4), Integer(4):Integer(0)})
>>> T
['A', 4, 1] relabelled by {0: 1, 1: 2, 2: 3, 3: 4, 4: 0}
>>> T.dynkin_diagram()                                                    #␣
↪needs sage.graphs
1
O-----------+
|           |
|           |
O---O---O---O
2   3   4   0
A4~ relabelled by {0: 1, 1: 2, 2: 3, 3: 4, 4: 0}

>>> T0 = T.classical()
```

(continues on next page)

(continued from previous page)

```
>>> T0
['A', 4] relabelled by {1: 2, 2: 3, 3: 4, 4: 0}
>>> T0.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O---O
2   3   4   0
A4 relabelled by {1: 2, 2: 3, 3: 4, 4: 0}
```

is_untwisted_affine()

Implement *CartanType_affine.is_untwisted_affine()*.

A relabelled Cartan type is untwisted affine if the original is.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).relabel({1:2, 2:3, 3:0, 0:1}).is_untwisted_
↪affine()
True
```

```
>>> from sage.all import *
>>> CartanType(['B', Integer(3), Integer(1)]).relabel({Integer(1):Integer(2),␣
↪Integer(2):Integer(3), Integer(3):Integer(0), Integer(0):Integer(1)}).is_
↪untwisted_affine()
True
```

special_node()

Return a special node of the Dynkin diagram.

See also

special_node()

It is obtained by relabelling of the special node of the non relabelled Dynkin diagram.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).special_node()
0
sage: CartanType(['B', 3, 1]).relabel({1:2, 2:3, 3:0, 0:1}).special_node()
1
```

```
>>> from sage.all import *
>>> CartanType(['B', Integer(3), Integer(1)]).special_node()
0
>>> CartanType(['B', Integer(3), Integer(1)]).relabel({Integer(1):Integer(2),␣
↪Integer(2):Integer(3), Integer(3):Integer(0), Integer(0):Integer(1)}).
↪special_node()
1
```

class sage.combinat.root_system.type_relabel.**CartanType_finite**(*type*, *relabelling*)

Bases: *CartanType*, *CartanType_finite*

AmbientSpace
alias of *AmbientSpace*

affine()
Return the affine Cartan type associated with `self`.

EXAMPLES:

```
sage: B4 = CartanType(['B',4])
sage: B4.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O=>=O
1   2   3   4
B4
sage: B4.affine().dynkin_diagram()                                           #␣
↪needs sage.graphs
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
B4~
```

```
>>> from sage.all import *
>>> B4 = CartanType(['B',Integer(4)])
>>> B4.dynkin_diagram()                                                    #␣
↪needs sage.graphs
O---O---O=>=O
1   2   3   4
B4
>>> B4.affine().dynkin_diagram()                                           #␣
↪needs sage.graphs
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
B4~
```

If possible, this reuses the original label for the special node:

```
sage: T = B4.relabel({1:2, 2:3, 3:4, 4:1}); T.dynkin_diagram()               #␣
↪needs sage.graphs
O---O---O=>=O
2   3   4   1
B4 relabelled by {1: 2, 2: 3, 3: 4, 4: 1}
sage: T.affine().dynkin_diagram()                                            #␣
↪needs sage.graphs
    O 0
    |
    |
O---O---O=>=O
2   3   4   1
B4~ relabelled by {0: 0, 1: 2, 2: 3, 3: 4, 4: 1}
```

```
>>> from sage.all import *
>>> T = B4.relabel({Integer(1):Integer(2), Integer(2):Integer(3),␣
↪Integer(3):Integer(4), Integer(4):Integer(1)}); T.dynkin_diagram()                ␣
↪     # needs sage.graphs
O---O---O=>=O
2   3   4   1
B4 relabelled by {1: 2, 2: 3, 3: 4, 4: 1}
>>> T.affine().dynkin_diagram()                                                   #␣
↪needs sage.graphs
    O 0
    |
    |
O---O---O=>=O
2   3   4   1
B4~ relabelled by {0: 0, 1: 2, 2: 3, 3: 4, 4: 1}
```

Otherwise, it chooses a label for the special_node in $0, 1, \ldots$:

```
sage: T = B4.relabel({1:0, 2:1, 3:2, 4:3}); T.dynkin_diagram()                      #␣
↪needs sage.graphs
O---O---O=>=O
0   1   2   3
B4 relabelled by {1: 0, 2: 1, 3: 2, 4: 3}
sage: T.affine().dynkin_diagram()                                                   #␣
↪needs sage.graphs
    O 4
    |
    |
O---O---O=>=O
0   1   2   3
B4~ relabelled by {0: 4, 1: 0, 2: 1, 3: 2, 4: 3}
```

```
>>> from sage.all import *
>>> T = B4.relabel({Integer(1):Integer(0), Integer(2):Integer(1),␣
↪Integer(3):Integer(2), Integer(4):Integer(3)}); T.dynkin_diagram()                ␣
↪     # needs sage.graphs
O---O---O=>=O
0   1   2   3
B4 relabelled by {1: 0, 2: 1, 3: 2, 4: 3}
>>> T.affine().dynkin_diagram()                                                   #␣
↪needs sage.graphs
    O 4
    |
    |
O---O---O=>=O
0   1   2   3
B4~ relabelled by {0: 4, 1: 0, 2: 1, 3: 2, 4: 3}
```

This failed before Issue #13724:

```
sage: ct = CartanType(["G",2]).dual(); ct
['G', 2] relabelled by {1: 2, 2: 1}
sage: ct.affine()
```

(continues on next page)

(continued from previous page)

```
['G', 2, 1] relabelled by {0: 0, 1: 2, 2: 1}

sage: ct = CartanType(["F",4]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: ct.affine()
['F', 4, 1] relabelled by {0: 0, 1: 4, 2: 3, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> ct = CartanType(["G",Integer(2)]).dual(); ct
['G', 2] relabelled by {1: 2, 2: 1}
>>> ct.affine()
['G', 2, 1] relabelled by {0: 0, 1: 2, 2: 1}

>>> ct = CartanType(["F",Integer(4)]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
>>> ct.affine()
['F', 4, 1] relabelled by {0: 0, 1: 4, 2: 3, 3: 2, 4: 1}
```

Check that we don't inadvertently change the internal relabelling of `ct`:

```
sage: ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

```
>>> from sage.all import *
>>> ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

5.1.270 Weight lattice realizations

class `sage.combinat.root_system.weight_lattice_realizations.`**`WeightLatticeRealizations`**(*base*, *name=None*)

Bases: `Category_over_base_ring`

The category of weight lattice realizations over a given base ring.

A *weight lattice realization* L over a base ring R is a free module (or vector space if R is a field) endowed with an embedding of the root lattice of some root system. By restriction, this embedding defines an embedding of the root lattice of this root system, which makes L a root lattice realization.

Typical weight lattice realizations over **Z** include the weight lattice, and ambient lattice. Typical weight lattice realizations over **Q** include the weight space, and ambient space.

To describe the embedding, a weight lattice realization must implement a method `fundamental_weight`(i) returning for each `i()` in the index set the image of the fundamental weight Λ_i under the embedding.

In order to be a proper root lattice realization, a weight lattice realization should also implement the scalar product with the coroot lattice; on the other hand, the embedding of the simple roots is given for free.

See also

- *RootSystem*
- *RootLatticeRealizations*

- *WeightSpace*
- *AmbientSpace*

EXAMPLES:

Here, we consider the root system of type A_7, and embed the weight lattice element $x = \Lambda_1 + 2\Lambda_3$ in several root lattice realizations:

```
sage: R = RootSystem(["A",7])
sage: Lambda = R.weight_lattice().fundamental_weights()
sage: x = Lambda[2] + 2 * Lambda[5]

sage: L = R.weight_space()
sage: L(x)
Lambda[2] + 2*Lambda[5]

sage: L = R.ambient_lattice()
sage: L(x)
(3, 3, 2, 2, 2, 0, 0, 0)
```

```
>>> from sage.all import *
>>> R = RootSystem(["A",Integer(7)])
>>> Lambda = R.weight_lattice().fundamental_weights()
>>> x = Lambda[Integer(2)] + Integer(2) * Lambda[Integer(5)]

>>> L = R.weight_space()
>>> L(x)
Lambda[2] + 2*Lambda[5]

>>> L = R.ambient_lattice()
>>> L(x)
(3, 3, 2, 2, 2, 0, 0, 0)
```

We embed the weight space element $x = \Lambda_1 + 1/2\Lambda_3$ in the ambient space:

```
sage: Lambda = R.weight_space().fundamental_weights()
sage: x = Lambda[2] + 1/2 * Lambda[5]

sage: L = R.ambient_space()
sage: L(x)
(3/2, 3/2, 1/2, 1/2, 1/2, 0, 0, 0)
```

```
>>> from sage.all import *
>>> Lambda = R.weight_space().fundamental_weights()
>>> x = Lambda[Integer(2)] + Integer(1)/Integer(2) * Lambda[Integer(5)]

>>> L = R.ambient_space()
>>> L(x)
(3/2, 3/2, 1/2, 1/2, 1/2, 0, 0, 0)
```

Of course, one can't embed the weight space in the ambient lattice:

```
sage: L = R.ambient_lattice()
sage: L(x)
Traceback (most recent call last):
...
TypeError: do not know how to make x (= Lambda[2] + 1/2*Lambda[5])
an element of self (=Ambient lattice of the Root system of type ['A', 7])
```

```
>>> from sage.all import *
>>> L = R.ambient_lattice()
>>> L(x)
Traceback (most recent call last):
...
TypeError: do not know how to make x (= Lambda[2] + 1/2*Lambda[5])
an element of self (=Ambient lattice of the Root system of type ['A', 7])
```

If K_1 is a subring of K_2, then one could in theory have an embedding from the weight space over K_1 to any weight lattice realization over K_2; this is not implemented:

```
sage: K1 = QQ
sage: K2 = QQ['q']
sage: L = R.ambient_space(K2)

sage: Lambda = R.weight_space(K2).fundamental_weights()
sage: L(Lambda[1])
(1, 0, 0, 0, 0, 0, 0, 0)

sage: Lambda = R.weight_space(K1).fundamental_weights()
sage: L(Lambda[1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= Lambda[1]) an element
of self (=Ambient space of the Root system of type ['A', 7])
```

```
>>> from sage.all import *
>>> K1 = QQ
>>> K2 = QQ['q']
>>> L = R.ambient_space(K2)

>>> Lambda = R.weight_space(K2).fundamental_weights()
>>> L(Lambda[Integer(1)])
(1, 0, 0, 0, 0, 0, 0, 0)

>>> Lambda = R.weight_space(K1).fundamental_weights()
>>> L(Lambda[Integer(1)])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= Lambda[1]) an element
of self (=Ambient space of the Root system of type ['A', 7])
```

class ElementMethods

Bases: `object`

symmetric_form(*la*)

Return the symmetric form of `self` with `la`.

Return the pairing $(|)$ on the weight lattice. See Chapter 6 in Kac, Infinite Dimensional Lie Algebras for more details.

> **Warning**
>
> For affine root systems, if you are not working in the extended weight lattice/space, this may return incorrect results.

EXAMPLES:

```
sage: # needs sage.graphs
sage: P = RootSystem(['C',2]).weight_lattice()
sage: al = P.simple_roots()
sage: al[1].symmetric_form(al[1])
2
sage: al[1].symmetric_form(al[2])
-2
sage: al[2].symmetric_form(al[1])
-2
sage: Q = RootSystem(['C',2]).root_lattice()
sage: alQ = Q.simple_roots()
sage: all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])
....:     for i in P.index_set() for j in P.index_set())
True

sage: # needs sage.graphs
sage: P = RootSystem(['C',2,1]).weight_lattice(extended=True)
sage: al = P.simple_roots()
sage: al[1].symmetric_form(al[1])
2
sage: al[1].symmetric_form(al[2])
-2
sage: al[1].symmetric_form(al[0])
-2
sage: al[0].symmetric_form(al[1])
-2
sage: Q = RootSystem(['C',2,1]).root_lattice()
sage: alQ = Q.simple_roots()
sage: all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])
....:     for i in P.index_set() for j in P.index_set())
True
sage: La = P.basis()
sage: [La['delta'].symmetric_form(al) for al in P.simple_roots()]
[0, 0, 0]
sage: [La[0].symmetric_form(al) for al in P.simple_roots()]
[1, 0, 0]

sage: P = RootSystem(['C',2,1]).weight_lattice()
sage: Q = RootSystem(['C',2,1]).root_lattice()
sage: al = P.simple_roots()                                                     #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
sage: alQ = Q.simple_roots()                                                 #␣
↪needs sage.graphs
sage: all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])   #␣
↪needs sage.graphs
....:     for i in P.index_set() for j in P.index_set())
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> P = RootSystem(['C',Integer(2)]).weight_lattice()
>>> al = P.simple_roots()
>>> al[Integer(1)].symmetric_form(al[Integer(1)])
2
>>> al[Integer(1)].symmetric_form(al[Integer(2)])
-2
>>> al[Integer(2)].symmetric_form(al[Integer(1)])
-2
>>> Q = RootSystem(['C',Integer(2)]).root_lattice()
>>> alQ = Q.simple_roots()
>>> all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])
...     for i in P.index_set() for j in P.index_set())
True

>>> # needs sage.graphs
>>> P = RootSystem(['C',Integer(2),Integer(1)]).weight_
↪lattice(extended=True)
>>> al = P.simple_roots()
>>> al[Integer(1)].symmetric_form(al[Integer(1)])
2
>>> al[Integer(1)].symmetric_form(al[Integer(2)])
-2
>>> al[Integer(1)].symmetric_form(al[Integer(0)])
-2
>>> al[Integer(0)].symmetric_form(al[Integer(1)])
-2
>>> Q = RootSystem(['C',Integer(2),Integer(1)]).root_lattice()
>>> alQ = Q.simple_roots()
>>> all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])
...     for i in P.index_set() for j in P.index_set())
True
>>> La = P.basis()
>>> [La['delta'].symmetric_form(al) for al in P.simple_roots()]
[0, 0, 0]
>>> [La[Integer(0)].symmetric_form(al) for al in P.simple_roots()]
[1, 0, 0]

>>> P = RootSystem(['C',Integer(2),Integer(1)]).weight_lattice()
>>> Q = RootSystem(['C',Integer(2),Integer(1)]).root_lattice()
>>> al = P.simple_roots()                                                    #␣
↪needs sage.graphs
>>> alQ = Q.simple_roots()                                                   #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
>>> all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])   #␣
↪needs sage.graphs
...     for i in P.index_set() for j in P.index_set())
True
```

The result of $(\Lambda_0|\alpha_0)$ should be 1, however we get 0 because we are not working in the extended weight lattice:

```
sage: La = P.basis()
sage: [La[0].symmetric_form(al) for al in P.simple_roots()]           #␣
↪needs sage.graphs
[0, 0, 0]
```

```
>>> from sage.all import *
>>> La = P.basis()
>>> [La[Integer(0)].symmetric_form(al) for al in P.simple_roots()]          ␣
↪     # needs sage.graphs
[0, 0, 0]
```

to_weight_space(*base_ring=None*)

Map `self` to the weight space.

> **Warning**
>
> Implemented for finite Cartan type.

EXAMPLES:

```
sage: b = CartanType(['B',2]).root_system().ambient_space().from_
↪vector(vector([1,-2])); b
(1, -2)
sage: b.to_weight_space()
3*Lambda[1] - 4*Lambda[2]
sage: b = CartanType(['B',2]).root_system().ambient_space().from_
↪vector(vector([1/2,0])); b
(1/2, 0)
sage: b.to_weight_space()
1/2*Lambda[1]
sage: b.to_weight_space(ZZ)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
sage: b = CartanType(['G',2]).root_system().ambient_space().from_
↪vector(vector([4,-5,1])); b
(4, -5, 1)
sage: b.to_weight_space()
-6*Lambda[1] + 5*Lambda[2]
```

```
>>> from sage.all import *
>>> b = CartanType(['B',Integer(2)]).root_system().ambient_space().from_
```

(continues on next page)

(continued from previous page)

```
↪vector(vector([Integer(1),-Integer(2)])); b
(1, -2)
>>> b.to_weight_space()
3*Lambda[1] - 4*Lambda[2]
>>> b = CartanType(['B',Integer(2)]).root_system().ambient_space().from_
↪vector(vector([Integer(1)/Integer(2),Integer(0)])); b
(1/2, 0)
>>> b.to_weight_space()
1/2*Lambda[1]
>>> b.to_weight_space(ZZ)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
>>> b = CartanType(['G',Integer(2)]).root_system().ambient_space().from_
↪vector(vector([Integer(4),-Integer(5),Integer(1)])); b
(4, -5, 1)
>>> b.to_weight_space()
-6*Lambda[1] + 5*Lambda[2]
```

class ParentMethods

Bases: `object`

dynkin_diagram_automorphism_of_alcove_morphism(*f*)

Return the Dynkin diagram automorphism induced by an alcove morphism.

INPUT:

- `f` – a linear map from `self` to `self` which preserves alcoves

This method returns the Dynkin diagram automorphism for the decomposition $f = dw$ (see `reduced_word_of_alcove_morphism()`), as a dictionary mapping elements of the index set to itself.

EXAMPLES:

```
sage: R = RootSystem(["A",2,1]).weight_lattice()
sage: alpha = R.simple_roots()                                          #␣
↪needs sage.graphs
sage: Lambda = R.fundamental_weights()
```

```
>>> from sage.all import *
>>> R = RootSystem(["A",Integer(2),Integer(1)]).weight_lattice()
>>> alpha = R.simple_roots()                                        #␣
↪needs sage.graphs
>>> Lambda = R.fundamental_weights()
```

Translations by elements of the root lattice induce a trivial Dynkin diagram automorphism:

```
sage: # needs sage.graphs sage.libs.gap
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[0].
↪translation)
{0: 0, 1: 1, 2: 2}
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[1].
↪translation)
{0: 0, 1: 1, 2: 2}
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[2].
```

(continues on next page)

(continued from previous page)

```
↪translation)
{0: 0, 1: 1, 2: 2}
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap
>>> R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[Integer(0)].
↪translation)
{0: 0, 1: 1, 2: 2}
>>> R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[Integer(1)].
↪translation)
{0: 0, 1: 1, 2: 2}
>>> R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[Integer(2)].
↪translation)
{0: 0, 1: 1, 2: 2}
```

This is no more the case for translations by general elements of the (classical) weight lattice at level 0:

```
sage: omega1 = Lambda[1] - Lambda[0]
sage: omega2 = Lambda[2] - Lambda[0]

sage: # needs sage.graphs sage.libs.gap
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(omega1.translation)
{0: 1, 1: 2, 2: 0}
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(omega2.translation)
{0: 2, 1: 0, 2: 1}

sage: # needs sage.graphs sage.libs.gap
sage: R = RootSystem(['C',2,1]).weight_lattice()
sage: alpha = R.simple_roots()
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[1].
↪translation)
{0: 2, 1: 1, 2: 0}

sage: # needs sage.graphs sage.libs.gap
sage: R = RootSystem(['D',5,1]).weight_lattice()
sage: Lambda = R.fundamental_weights()
sage: omega1 = Lambda[1] - Lambda[0]
sage: omega2 = Lambda[2] - 2*Lambda[0]
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(omega1.translation)
{0: 1, 1: 0, 2: 2, 3: 3, 4: 5, 5: 4}
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(omega2.translation)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}
```

```
>>> from sage.all import *
>>> omega1 = Lambda[Integer(1)] - Lambda[Integer(0)]
>>> omega2 = Lambda[Integer(2)] - Lambda[Integer(0)]

>>> # needs sage.graphs sage.libs.gap
>>> R.dynkin_diagram_automorphism_of_alcove_morphism(omega1.translation)
{0: 1, 1: 2, 2: 0}
>>> R.dynkin_diagram_automorphism_of_alcove_morphism(omega2.translation)
{0: 2, 1: 0, 2: 1}
```

(continues on next page)

(continued from previous page)

```
>>> # needs sage.graphs sage.libs.gap
>>> R = RootSystem(['C',Integer(2),Integer(1)]).weight_lattice()
>>> alpha = R.simple_roots()
>>> R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[Integer(1)].
↪translation)
{0: 2, 1: 1, 2: 0}

>>> # needs sage.graphs sage.libs.gap
>>> R = RootSystem(['D',Integer(5),Integer(1)]).weight_lattice()
>>> Lambda = R.fundamental_weights()
>>> omega1 = Lambda[Integer(1)] - Lambda[Integer(0)]
>>> omega2 = Lambda[Integer(2)] - Integer(2)*Lambda[Integer(0)]
>>> R.dynkin_diagram_automorphism_of_alcove_morphism(omega1.translation)
{0: 1, 1: 0, 2: 2, 3: 3, 4: 5, 5: 4}
>>> R.dynkin_diagram_automorphism_of_alcove_morphism(omega2.translation)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}
```

Algorithm: computes w of the decomposition, and see how $f \circ w^{-1}$ permutes the simple roots.

embed_at_level(*x*, *level=1*)

Embed the classical weight x in the level `level` hyperplane.

This is achieved by translating the straightforward embedding of x by $c\Lambda_0$ for c some appropriate scalar.

INPUT:

- `x` – an element of the corresponding classical weight/ambient lattice
- `level` – integer or element of the base ring (default: 1)

EXAMPLES:

```
sage: # needs sage.graphs
sage: L = RootSystem(["B",3,1]).weight_space()
sage: L0 = L.classical()
sage: alpha = L0.simple_roots()
sage: omega = L0.fundamental_weights()
sage: L.embed_at_level(omega[1], 1)
Lambda[1]
sage: L.embed_at_level(omega[2], 1)
-Lambda[0] + Lambda[2]
sage: L.embed_at_level(omega[3], 1)
Lambda[3]
sage: L.embed_at_level(alpha[1], 1)
Lambda[0] + 2*Lambda[1] - Lambda[2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> L = RootSystem(["B",Integer(3),Integer(1)]).weight_space()
>>> L0 = L.classical()
>>> alpha = L0.simple_roots()
>>> omega = L0.fundamental_weights()
>>> L.embed_at_level(omega[Integer(1)], Integer(1))
Lambda[1]
>>> L.embed_at_level(omega[Integer(2)], Integer(1))
```

(continues on next page)

(continued from previous page)

```
-Lambda[0] + Lambda[2]
>>> L.embed_at_level(omega[Integer(3)], Integer(1))
Lambda[3]
>>> L.embed_at_level(alpha[Integer(1)], Integer(1))
Lambda[0] + 2*Lambda[1] - Lambda[2]
```

fundamental_weight(*i*)

Return the i-th fundamental weight.

INPUT:

- i – an element of the index set

By a slight notational abuse, for an affine type this method should also accept 'delta' as input, and return the image of δ of the extended weight lattice in this realization.

This should be overridden by any subclass, and typically be implemented as a cached method for efficiency.

EXAMPLES:

```
sage: L = RootSystem(["A",3]).ambient_lattice()
sage: L.fundamental_weight(1)
(1, 0, 0, 0)

sage: L = RootSystem(["A",3,1]).weight_lattice(extended=True)
sage: L.fundamental_weight(1)
Lambda[1]
sage: L.fundamental_weight("delta")
delta
```

```
>>> from sage.all import *
>>> L = RootSystem(["A",Integer(3)]).ambient_lattice()
>>> L.fundamental_weight(Integer(1))
(1, 0, 0, 0)

>>> L = RootSystem(["A",Integer(3),Integer(1)]).weight_
↪lattice(extended=True)
>>> L.fundamental_weight(Integer(1))
Lambda[1]
>>> L.fundamental_weight("delta")
delta
```

fundamental_weights()

Return the family $(\Lambda_i)_{i\in I}$ of the fundamental weights.

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: f = e.fundamental_weights()
sage: [f[i] for i in [1,2,3]]
[(1, 0, 0, 0), (1, 1, 0, 0), (1, 1, 1, 0)]
```

```
>>> from sage.all import *
>>> e = RootSystem(['A',Integer(3)]).ambient_lattice()
```

(continues on next page)

(continued from previous page)

```
>>> f = e.fundamental_weights()
>>> [f[i] for i in [Integer(1),Integer(2),Integer(3)]]
[(1, 0, 0, 0), (1, 1, 0, 0), (1, 1, 1, 0)]
```

is_extended()

Return whether this is a realization of the extended weight lattice.

See also

sage.combinat.root_system.weight_space.WeightSpace

EXAMPLES:

```
sage: RootSystem(["A",3,1]).weight_lattice().is_extended()
False
sage: RootSystem(["A",3,1]).weight_lattice(extended=True).is_extended()
True
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3),Integer(1)]).weight_lattice().is_extended()
False
>>> RootSystem(["A",Integer(3),Integer(1)]).weight_lattice(extended=True).
↪is_extended()
True
```

This method is irrelevant for finite root systems, since the weight lattice need not be extended to ensure that the root lattice embeds faithfully:

```
sage: RootSystem(["A",3]).weight_lattice().is_extended()
False
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3)]).weight_lattice().is_extended()
False
```

reduced_word_of_alcove_morphism(f)

Return the reduced word of an alcove morphism.

INPUT:

- `f` – a linear map from `self` to `self` which preserves alcoves

Let A be the fundamental alcove. This returns a reduced word $i_1, \dots, i_k$ such that the affine Weyl group element $w = s_{i_1} \circ \cdots \circ s_{i_k}$ maps the alcove $f(A)$ back to A. In other words, the alcove walk $i_1, \dots, i_k$ brings the fundamental alcove to the corresponding translated alcove.

Let us throw in a bit of context to explain the main use case. It is customary to realize the alcove picture in the coroot or coweight lattice $R^\vee$. The extended affine Weyl group is then the group of linear maps on $R^\vee$ which preserve the alcoves. By [Kac "Infinite-dimensional Lie algebra", Proposition 6.5] the affine Weyl group is the semidirect product of the associated finite Weyl group and the group of translations in the coroot lattice (the extended affine Weyl group uses the coweight lattice instead). In other words, an element of the extended affine Weyl group admits a unique decomposition of the form:

$$f = dw,$$

where w is in the Weyl group, and d is a function which maps the fundamental alcove to itself. As d permutes the walls of the fundamental alcove, it permutes accordingly the corresponding simple roots, which induces an automorphism of the Dynkin diagram.

This method returns a reduced word for w, whereas the method *dynkin_diagram_automorphism_of_alcove_morphism()* returns d as a permutation of the nodes of the Dynkin diagram.

Nota bene: recall that the coroot (resp. coweight) lattice is implemented as the root (resp weight) lattice of the dual root system. Hence, this method is implemented for weight lattice realizations, but in practice is most of the time used on the dual side.

EXAMPLES:

We start with type A which is simply laced; hence we do not have to worry about the distinction between the weight and coweight lattice:

```
sage: R = RootSystem(["A",2,1]).weight_lattice()
sage: alpha = R.simple_roots()                                              #␣
↪needs sage.graphs
sage: Lambda = R.fundamental_weights()
```

```
>>> from sage.all import *
>>> R = RootSystem(["A",Integer(2),Integer(1)]).weight_lattice()
>>> alpha = R.simple_roots()                                            #␣
↪needs sage.graphs
>>> Lambda = R.fundamental_weights()
```

We consider first translations by elements of the root lattice:

```
sage: R.reduced_word_of_alcove_morphism(alpha[0].translation)              #␣
↪needs sage.graphs
[1, 2, 1, 0]
sage: R.reduced_word_of_alcove_morphism(alpha[1].translation)              #␣
↪needs sage.graphs
[0, 2, 0, 1]
sage: R.reduced_word_of_alcove_morphism(alpha[2].translation)              #␣
↪needs sage.graphs
[0, 1, 0, 2]
```

```
>>> from sage.all import *
>>> R.reduced_word_of_alcove_morphism(alpha[Integer(0)].translation)      ␣
↪     # needs sage.graphs
[1, 2, 1, 0]
>>> R.reduced_word_of_alcove_morphism(alpha[Integer(1)].translation)      ␣
↪     # needs sage.graphs
[0, 2, 0, 1]
>>> R.reduced_word_of_alcove_morphism(alpha[Integer(2)].translation)      ␣
↪     # needs sage.graphs
[0, 1, 0, 2]
```

We continue with translations by elements of the classical weight lattice, embedded at level 0:

sage: omega1 = Lambda[1] - Lambda[0] sage: omega2 = Lambda[2] - Lambda[0]

sage: R.reduced_word_of_alcove_morphism(omega1.translation) # needs sage.graphs [0, 2] sage: R.reduced_word_of_alcove_morphism(omega2.translation) # needs sage.graphs [0, 1]

The following tests ensure that the code agrees with the tables in Kashiwara's private notes on affine quantum algebras (2008).

reduced_word_of_translation(*t*)

Given an element of the root lattice, this returns a reduced word $i_1, \ldots, i_k$ such that the Weyl group element $s_{i_1} \circ \cdots \circ s_{i_k}$ implements the "translation" where x maps to $x + level(x) * t$. In other words, the alcove walk $i_1, \ldots, i_k$ brings the fundamental alcove to the corresponding translated alcove.

Note

There are some technical conditions for t to actually be a translation; those are not tested (TODO: detail).

EXAMPLES:

```
sage: # needs sage.graphs
sage: R = RootSystem(["A",2,1]).weight_lattice()
sage: alpha = R.simple_roots()
sage: R.reduced_word_of_translation(alpha[1])
[0, 2, 0, 1]
sage: R.reduced_word_of_translation(alpha[2])
[0, 1, 0, 2]
sage: R.reduced_word_of_translation(alpha[0])
[1, 2, 1, 0]

sage: R = RootSystem(['D',5,1]).weight_lattice()
sage: Lambda = R.fundamental_weights()
sage: omega1 = Lambda[1] - Lambda[0]
sage: omega2 = Lambda[2] - 2*Lambda[0]
sage: R.reduced_word_of_translation(omega1)                              #␣
↪needs sage.graphs
[0, 2, 3, 4, 5, 3, 2, 0]
sage: R.reduced_word_of_translation(omega2)                              #␣
↪needs sage.graphs
[0, 2, 1, 3, 2, 4, 3, 5, 3, 2, 1, 4, 3, 2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> R = RootSystem(["A",Integer(2),Integer(1)]).weight_lattice()
>>> alpha = R.simple_roots()
>>> R.reduced_word_of_translation(alpha[Integer(1)])
[0, 2, 0, 1]
>>> R.reduced_word_of_translation(alpha[Integer(2)])
[0, 1, 0, 2]
>>> R.reduced_word_of_translation(alpha[Integer(0)])
[1, 2, 1, 0]

>>> R = RootSystem(['D',Integer(5),Integer(1)]).weight_lattice()
>>> Lambda = R.fundamental_weights()
>>> omega1 = Lambda[Integer(1)] - Lambda[Integer(0)]
>>> omega2 = Lambda[Integer(2)] - Integer(2)*Lambda[Integer(0)]
>>> R.reduced_word_of_translation(omega1)                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[0, 2, 3, 4, 5, 3, 2, 0]
>>> R.reduced_word_of_translation(omega2)                                    #␣
↪needs sage.graphs
[0, 2, 1, 3, 2, 4, 3, 5, 3, 2, 1, 4, 3, 2]
```

A non simply laced case:

```
sage: R = RootSystem(["C",2,1]).weight_lattice()
sage: Lambda = R.fundamental_weights()
sage: c = R.cartan_type().translation_factors(); c                             #␣
↪needs sage.graphs
Finite family {0: 1, 1: 2, 2: 1}
sage: R.reduced_word_of_translation((Lambda[1]-Lambda[0]) * c[1])             #␣
↪needs sage.graphs
[0, 1, 2, 1]
sage: R.reduced_word_of_translation((Lambda[2]-Lambda[0]) * c[2])             #␣
↪needs sage.graphs
[0, 1, 0]
```

```
>>> from sage.all import *
>>> R = RootSystem(["C",Integer(2),Integer(1)]).weight_lattice()
>>> Lambda = R.fundamental_weights()
>>> c = R.cartan_type().translation_factors(); c                             #␣
↪needs sage.graphs
Finite family {0: 1, 1: 2, 2: 1}
>>> R.reduced_word_of_translation((Lambda[Integer(1)]-Lambda[Integer(0)])␣
↪* c[Integer(1)])        # needs sage.graphs
[0, 1, 2, 1]
>>> R.reduced_word_of_translation((Lambda[Integer(2)]-Lambda[Integer(0)])␣
↪* c[Integer(2)])        # needs sage.graphs
[0, 1, 0]
```

See also `_test_reduced_word_of_translation()`.

Todo

- Add a picture in the doc
- Add a method which, given an element of the classical weight lattice, constructs the appropriate value for t

rho()

EXAMPLES:

```
sage: RootSystem(['A',3]).ambient_lattice().rho()
(3, 2, 1, 0)
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).ambient_lattice().rho()
(3, 2, 1, 0)
```

rho_classical()

Return the embedding at level 0 of ρ of the classical lattice.

EXAMPLES:

```
sage: RootSystem(['C',4,1]).weight_lattice().rho_classical()                    #␣
↪needs sage.graphs
-4*Lambda[0] + Lambda[1] + Lambda[2] + Lambda[3] + Lambda[4]
sage: L = RootSystem(['D',4,1]).weight_lattice()
sage: L.rho_classical().scalar(L.null_coroot())                                 #␣
↪needs sage.graphs
0
```

```
>>> from sage.all import *
>>> RootSystem(['C',Integer(4),Integer(1)]).weight_lattice().rho_
↪classical()            # needs sage.graphs
-4*Lambda[0] + Lambda[1] + Lambda[2] + Lambda[3] + Lambda[4]
>>> L = RootSystem(['D',Integer(4),Integer(1)]).weight_lattice()
>>> L.rho_classical().scalar(L.null_coroot())                                 #␣
↪needs sage.graphs
0
```

Warning: In affine type BC dual, this does not live in the weight lattice:

```
sage: L = CartanType(["BC",2,2]).dual().root_system().weight_space()
sage: L.rho_classical()                                                         #␣
↪needs sage.graphs
-3/2*Lambda[0] + Lambda[1] + Lambda[2]
sage: L = CartanType(["BC",2,2]).dual().root_system().weight_lattice()
sage: L.rho_classical()                                                         #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: 5 is not divisible by 2
```

```
>>> from sage.all import *
>>> L = CartanType(["BC",Integer(2),Integer(2)]).dual().root_system().
↪weight_space()
>>> L.rho_classical()                                                         #␣
↪needs sage.graphs
-3/2*Lambda[0] + Lambda[1] + Lambda[2]
>>> L = CartanType(["BC",Integer(2),Integer(2)]).dual().root_system().
↪weight_lattice()
>>> L.rho_classical()                                                         #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: 5 is not divisible by 2
```

signs_of_alcovewalk(*walk*)

Let walk = $[i_1, \ldots, i_n]$ denote an alcove walk starting from the fundamental alcove y_0, crossing at step

1 the wall i_1, and so on.

For each k, set $w_k = s_{i_1} \circ s_{i_k}$, and denote by $y_k = w_k(y_0)$ the alcove reached after k steps. Then, y_k is obtained recursively from y_{k-1} by applying the following reflection:

$$y_k = s_{w_{k-1}\alpha_{i_k}} y_{k-1}.$$

The step is said positive if $w_{k-1}\alpha_{i_k}$ is a negative root (considering w_{k-1} as element of the classical Weyl group and α_{i_k} as a classical root) and negative otherwise. The algorithm implemented here use the equivalent property:

```
.. MATH:: \langle w_{k-1}^{-1} \rho_0, \alpha^\vee_{i_k}\rangle > 0
```

Where ρ_0 is the sum of the classical fundamental weights embedded at level 0 in this space (see `rho_classical()`), and $\alpha^\vee_{i_k}$ is the simple coroot associated to α_{i_k}.

This function returns a list of the form $[+1, +1, -1, ...]$, where the k-th entry denotes whether the k-th step was positive or negative.

See equation 3.4, of Ram: Alcove walks …, arXiv math/0601343v1

EXAMPLES:

```
sage: L = RootSystem(['C',2,1]).weight_lattice()
sage: L.signs_of_alcovewalk([1,2,0,1,2,1,2,0,1,2])                           #␣
↪needs sage.libs.gap
[-1, -1, 1, -1, 1, 1, 1, 1, 1, 1]

sage: L = RootSystem(['A',2,1]).weight_lattice()
sage: L.signs_of_alcovewalk([0,1,2,1,2,0,1,2,0,1,2,0])                       #␣
↪needs sage.libs.gap
[1, 1, 1, 1, -1, 1, -1, 1, -1, 1, -1, 1]

sage: L = RootSystem(['B',2,1]).coweight_lattice()
sage: L.signs_of_alcovewalk([0,1,2,0,1,2])                                   #␣
↪needs sage.libs.gap
[1, -1, 1, -1, 1, 1]
```

```
>>> from sage.all import *
>>> L = RootSystem(['C',Integer(2),Integer(1)]).weight_lattice()
>>> L.signs_of_alcovewalk([Integer(1),Integer(2),Integer(0),Integer(1),
↪Integer(2),Integer(1),Integer(2),Integer(0),Integer(1),Integer(2)])    ␣
↪                 # needs sage.libs.gap
[-1, -1, 1, -1, 1, 1, 1, 1, 1, 1]

>>> L = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice()
>>> L.signs_of_alcovewalk([Integer(0),Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),Integer(1),
↪Integer(2),Integer(0)])                  # needs sage.libs.gap
[1, 1, 1, 1, -1, 1, -1, 1, -1, 1, -1, 1]

>>> L = RootSystem(['B',Integer(2),Integer(1)]).coweight_lattice()
>>> L.signs_of_alcovewalk([Integer(0),Integer(1),Integer(2),Integer(0),
↪Integer(1),Integer(2)])                              # needs sage.libs.
↪gap
[1, -1, 1, -1, 1, 1]
```

> **Warning**
>
> This method currently does not work in the weight lattice for type BC dual because ρ_0 does not live in this lattice (but an integral multiple of it would do the job as well).

simple_root(*i*)

Return the i-th simple root.

This default implementation takes the i-th simple root in the weight lattice and embeds it in `self`.

EXAMPLES:

Since all the weight lattice realizations in Sage currently implement a `simple_root` method, we have to call this one by hand:

```
sage: from sage.combinat.root_system.weight_lattice_realizations import␣
↪WeightLatticeRealizations
sage: simple_root = WeightLatticeRealizations(QQ).parent_class.simple_
↪root.f
sage: L = RootSystem("A3").ambient_space()
sage: simple_root(L, 1)                                                    #␣
↪needs sage.graphs
(1, -1, 0, 0)
sage: simple_root(L, 2)                                                    #␣
↪needs sage.graphs
(0, 1, -1, 0)
sage: simple_root(L, 3)                                                    #␣
↪needs sage.graphs
(1, 1, 2, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.weight_lattice_realizations import␣
↪WeightLatticeRealizations
>>> simple_root = WeightLatticeRealizations(QQ).parent_class.simple_root.f
>>> L = RootSystem("A3").ambient_space()
>>> simple_root(L, Integer(1))                                              ␣
↪     # needs sage.graphs
(1, -1, 0, 0)
>>> simple_root(L, Integer(2))                                              ␣
↪     # needs sage.graphs
(0, 1, -1, 0)
>>> simple_root(L, Integer(3))                                              ␣
↪     # needs sage.graphs
(1, 1, 2, 0)
```

Note that this last root differs from the one implemented in `L` by a multiple of the vector `(1,1,1,1)`:

```
sage: L.simple_roots()                                                     #␣
↪needs sage.graphs
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

```
>>> from sage.all import *
>>> L.simple_roots()                                                       #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

This is a harmless artefact of the SL versus GL interpretation of type A; see the thematic tutorial on Lie Methods and Related Combinatorics in Sage for details.

weyl_dimension(*highest_weight*)

Return the dimension of the highest weight representation of highest weight `highest_weight`.

EXAMPLES:

```
sage: RootSystem(['A',3]).ambient_lattice().weyl_dimension([2,1,0,0])
20
sage: P = RootSystem(['C',2]).weight_lattice()
sage: La = P.basis()
sage: P.weyl_dimension(La[1]+La[2])                                         #␣
↪needs sage.graphs
16

sage: type(RootSystem(['A',3]).ambient_lattice().weyl_dimension([2,1,0,
↪0]))
<class 'sage.rings.integer.Integer'>
```

```
>>> from sage.all import *
>>> RootSystem(['A',Integer(3)]).ambient_lattice().weyl_
↪dimension([Integer(2),Integer(1),Integer(0),Integer(0)])
20
>>> P = RootSystem(['C',Integer(2)]).weight_lattice()
>>> La = P.basis()
>>> P.weyl_dimension(La[Integer(1)]+La[Integer(2)])                         ␣
↪              # needs sage.graphs
16

>>> type(RootSystem(['A',Integer(3)]).ambient_lattice().weyl_
↪dimension([Integer(2),Integer(1),Integer(0),Integer(0)]))
<class 'sage.rings.integer.Integer'>
```

super_categories()

EXAMPLES:

```
sage: from sage.combinat.root_system.weight_lattice_realizations import␣
↪WeightLatticeRealizations
sage: WeightLatticeRealizations(QQ).super_categories()
[Category of root lattice realizations over Rational Field]
```

```
>>> from sage.all import *
>>> from sage.combinat.root_system.weight_lattice_realizations import␣
↪WeightLatticeRealizations
>>> WeightLatticeRealizations(QQ).super_categories()
[Category of root lattice realizations over Rational Field]
```

5.1.271 Weight lattices and weight spaces

class sage.combinat.root_system.weight_space.**WeightSpace**(*root_system*, *base_ring*, *extended*)

Bases: *CombinatorialFreeModule*

INPUT:

- root_system – a root system
- base_ring – a ring R
- extended – boolean (default: False)

The weight space (or lattice if base_ring is $\mathbf{Z}$) of a root system is the formal free module $\bigoplus_i R\Lambda_i$ generated by the fundamental weights $(\Lambda_i)_{i\in I}$ of the root system.

This class is also used for coweight spaces (or lattices).

See also

- *RootSystem()*
- *RootSystem.weight_lattice()* and *RootSystem.weight_space()*
- *WeightLatticeRealizations()*

EXAMPLES:

```
sage: Q = RootSystem(['A', 3]).weight_lattice(); Q
Weight lattice of the Root system of type ['A', 3]
sage: Q.simple_roots()                                                          #␣
↪needs sage.graphs
Finite family {1: 2*Lambda[1] -   Lambda[2],
               2:  -Lambda[1] + 2*Lambda[2] -   Lambda[3],
               3:               -Lambda[2] + 2*Lambda[3]}

sage: Q = RootSystem(['A', 3, 1]).weight_lattice(); Q
Weight lattice of the Root system of type ['A', 3, 1]
sage: Q.simple_roots()                                                          #␣
↪needs sage.graphs
Finite family {0: 2*Lambda[0] -   Lambda[1]               -   Lambda[3],
               1:  -Lambda[0] + 2*Lambda[1] -   Lambda[2],
               2:               -Lambda[1] + 2*Lambda[2] -   Lambda[3],
               3:  -Lambda[0]               -   Lambda[2] + 2*Lambda[3]}
```

```
>>> from sage.all import *
>>> Q = RootSystem(['A', Integer(3)]).weight_lattice(); Q
Weight lattice of the Root system of type ['A', 3]
>>> Q.simple_roots()                                                          #␣
↪needs sage.graphs
Finite family {1: 2*Lambda[1] -   Lambda[2],
               2:  -Lambda[1] + 2*Lambda[2] -   Lambda[3],
               3:               -Lambda[2] + 2*Lambda[3]}

>>> Q = RootSystem(['A', Integer(3), Integer(1)]).weight_lattice(); Q
Weight lattice of the Root system of type ['A', 3, 1]
```

(continues on next page)

(continued from previous page)

```
>>> Q.simple_roots()                                                    #␣
↪needs sage.graphs
Finite family {0: 2*Lambda[0] -   Lambda[1]               -   Lambda[3],
               1:  -Lambda[0] + 2*Lambda[1] -   Lambda[2],
               2:               -Lambda[1] + 2*Lambda[2] -   Lambda[3],
               3:  -Lambda[0]               -   Lambda[2] + 2*Lambda[3]}
```

For infinite types, the Cartan matrix is singular, and therefore the embedding of the root lattice is not faithful:

```
sage: sum(Q.simple_roots())                                             #␣
↪needs sage.graphs
0
```

```
>>> from sage.all import *
>>> sum(Q.simple_roots())                                               #␣
↪needs sage.graphs
0
```

In particular, the null root is zero:

```
sage: Q.null_root()                                                     #␣
↪needs sage.graphs
0
```

```
>>> from sage.all import *
>>> Q.null_root()                                                       #␣
↪needs sage.graphs
0
```

This can be compensated by extending the basis of the weight space and slightly deforming the simple roots to make them linearly independent, without affecting the scalar product with the coroots. This feature is currently only implemented for affine types. In that case, if `extended` is set, then the basis of the weight space is extended by an element δ:

```
sage: Q = RootSystem(['A', 3, 1]).weight_lattice(extended=True); Q
Extended weight lattice of the Root system of type ['A', 3, 1]
sage: Q.basis().keys()
{0, 1, 2, 3, 'delta'}
```

```
>>> from sage.all import *
>>> Q = RootSystem(['A', Integer(3), Integer(1)]).weight_lattice(extended=True); Q
Extended weight lattice of the Root system of type ['A', 3, 1]
>>> Q.basis().keys()
{0, 1, 2, 3, 'delta'}
```

And the simple root α_0 associated to the special node is deformed as follows:

```
sage: Q.simple_roots()                                                  #␣
↪needs sage.graphs
Finite family {0: 2*Lambda[0] -   Lambda[1]               -   Lambda[3] + delta,
               1:  -Lambda[0] + 2*Lambda[1] -   Lambda[2],
```

(continues on next page)

(continued from previous page)

```
              2:                -Lambda[1] + 2*Lambda[2] -   Lambda[3],
              3:  -Lambda[0]               -   Lambda[2] + 2*Lambda[3]}
```

```
>>> from sage.all import *
>>> Q.simple_roots()                                                    #␣
↪needs sage.graphs
Finite family {0: 2*Lambda[0] -   Lambda[1]              -   Lambda[3] + delta,
              1:  -Lambda[0] + 2*Lambda[1] -   Lambda[2],
              2:                -Lambda[1] + 2*Lambda[2] -   Lambda[3],
              3:  -Lambda[0]               -   Lambda[2] + 2*Lambda[3]}
```

Now, the null root is nonzero:

```
sage: Q.null_root()                                                      #␣
↪needs sage.graphs
delta
```

```
>>> from sage.all import *
>>> Q.null_root()                                                       #␣
↪needs sage.graphs
delta
```

Warning

By a slight notational abuse, the extra basis element used to extend the fundamental weights is called `\delta` in the current implementation. However, in the literature, `\delta` usually denotes instead the null root. Most of the time, those two objects coincide, but not for type BC (aka. $A_{2n}^{(2)}$). Therefore we currently have:

```
sage: Q = RootSystem(["A",4,2]).weight_lattice(extended=True)
sage: Q.simple_root(0)                                                  #␣
↪needs sage.graphs
2*Lambda[0] - Lambda[1] + delta
sage: Q.null_root()                                                     #␣
↪needs sage.graphs
2*delta
```

```
>>> from sage.all import *
>>> Q = RootSystem(["A",Integer(4),Integer(2)]).weight_lattice(extended=True)
>>> Q.simple_root(Integer(0))                                            ␣
↪    # needs sage.graphs
2*Lambda[0] - Lambda[1] + delta
>>> Q.null_root()                                                     #␣
↪needs sage.graphs
2*delta
```

whereas, with the standard notations from the literature, one would expect to get respectively $2\Lambda_0 - \Lambda_1 + 1/2\delta$ and δ.

Other than this notational glitch, the implementation remains correct for type BC.

The notations may get improved in a subsequent version, which might require changing the index of the extra basis element. To guarantee backward compatibility in code not included in Sage, it is recommended to use the following idiom to get that index:

```
sage: F = Q.basis_extension(); F
Finite family {'delta': delta}
sage: index = F.keys()[0]; index
'delta'
```

```
>>> from sage.all import *
>>> F = Q.basis_extension(); F
Finite family {'delta': delta}
>>> index = F.keys()[Integer(0)]; index
'delta'
```

Then, for example, the coefficient of an element of the extended weight lattice on that basis element can be recovered with:

```
sage: Q.null_root()[index]                                              #␣
↪needs sage.graphs
2
```

```
>>> from sage.all import *
>>> Q.null_root()[index]                                                #␣
↪needs sage.graphs
2
```

Element

alias of *WeightSpaceElement*

basis_extension()

Return the basis elements used to extend the fundamental weights.

EXAMPLES:

```
sage: Q = RootSystem(["A",3,1]).weight_lattice()
sage: Q.basis_extension()
Family ()

sage: Q = RootSystem(["A",3,1]).weight_lattice(extended=True)
sage: Q.basis_extension()
Finite family {'delta': delta}
```

```
>>> from sage.all import *
>>> Q = RootSystem(["A",Integer(3),Integer(1)]).weight_lattice()
>>> Q.basis_extension()
Family ()

>>> Q = RootSystem(["A",Integer(3),Integer(1)]).weight_lattice(extended=True)
>>> Q.basis_extension()
Finite family {'delta': delta}
```

This method is irrelevant for finite types:

```
sage: Q = RootSystem(["A",3]).weight_lattice()
sage: Q.basis_extension()
Family ()
```

```
>>> from sage.all import *
>>> Q = RootSystem(["A",Integer(3)]).weight_lattice()
>>> Q.basis_extension()
Family ()
```

fundamental_weight(*i*)

Return the i-th fundamental weight.

INPUT:

- `i` – an element of the index set or `'delta'`

By a slight notational abuse, for an affine type this method also accepts `'delta'` as input, and returns the image of δ of the extended weight lattice in this realization.

See also

`fundamental_weight()`

EXAMPLES:

```
sage: Q = RootSystem(["A",3]).weight_lattice()
sage: Q.fundamental_weight(1)
Lambda[1]

sage: Q = RootSystem(["A",3,1]).weight_lattice(extended=True)
sage: Q.fundamental_weight(1)
Lambda[1]
sage: Q.fundamental_weight("delta")
delta
```

```
>>> from sage.all import *
>>> Q = RootSystem(["A",Integer(3)]).weight_lattice()
>>> Q.fundamental_weight(Integer(1))
Lambda[1]

>>> Q = RootSystem(["A",Integer(3),Integer(1)]).weight_lattice(extended=True)
>>> Q.fundamental_weight(Integer(1))
Lambda[1]
>>> Q.fundamental_weight("delta")
delta
```

is_extended()

Return whether this is an extended weight lattice.

See also

`is_extended()`

EXAMPLES:

```
sage: RootSystem(["A",3,1]).weight_lattice().is_extended()
False
sage: RootSystem(["A",3,1]).weight_lattice(extended=True).is_extended()
True
```

```
>>> from sage.all import *
>>> RootSystem(["A",Integer(3),Integer(1)]).weight_lattice().is_extended()
False
>>> RootSystem(["A",Integer(3),Integer(1)]).weight_lattice(extended=True).is_
↪extended()
True
```

simple_root(*j*)

Return the j-th simple root.

EXAMPLES:

```
sage: L = RootSystem(["C",4]).weight_lattice()
sage: L.simple_root(3)                                                  #␣
↪needs sage.graphs
-Lambda[2] + 2*Lambda[3] - Lambda[4]
```

```
>>> from sage.all import *
>>> L = RootSystem(["C",Integer(4)]).weight_lattice()
>>> L.simple_root(Integer(3))                                            ␣
↪      # needs sage.graphs
-Lambda[2] + 2*Lambda[3] - Lambda[4]
```

Its coefficients are given by the corresponding column of the Cartan matrix:

```
sage: L.cartan_type().cartan_matrix()[:,2]                              #␣
↪needs sage.graphs
[ 0]
[-1]
[ 2]
[-1]
```

```
>>> from sage.all import *
>>> L.cartan_type().cartan_matrix()[:,Integer(2)]                        ␣
↪      # needs sage.graphs
[ 0]
[-1]
[ 2]
[-1]
```

Here are all simple roots:

```
sage: L.simple_roots()                                                  #␣
↪needs sage.graphs
Finite family {1:  2*Lambda[1]   - Lambda[2],
               2:   -Lambda[1] + 2*Lambda[2]   - Lambda[3],
               3:   -Lambda[2] + 2*Lambda[3]   - Lambda[4],
               4:              -2*Lambda[3] + 2*Lambda[4]}
```

```
>>> from sage.all import *
>>> L.simple_roots()                                                         #␣
↪needs sage.graphs
Finite family {1:  2*Lambda[1]    - Lambda[2],
               2:   -Lambda[1] + 2*Lambda[2]    - Lambda[3],
               3:   -Lambda[2] + 2*Lambda[3]    - Lambda[4],
               4:               -2*Lambda[3] + 2*Lambda[4]}
```

For the extended weight lattice of an affine type, the simple root associated to the special node is deformed by adding δ, where δ is the null root:

```
sage: L = RootSystem(["C",4,1]).weight_lattice(extended=True)
sage: L.simple_root(0)                                                         #␣
↪needs sage.graphs
2*Lambda[0] - 2*Lambda[1] + delta
```

```
>>> from sage.all import *
>>> L = RootSystem(["C",Integer(4),Integer(1)]).weight_lattice(extended=True)
>>> L.simple_root(Integer(0))                                                  ␣
↪     # needs sage.graphs
2*Lambda[0] - 2*Lambda[1] + delta
```

In fact δ is really $1/a_0$ times the null root (see the discussion in *WeightSpace*) but this only makes a difference in type BC:

```
sage: L = RootSystem(CartanType(["BC",4,2])).weight_lattice(extended=True)
sage: L.simple_root(0)                                                         #␣
↪needs sage.graphs
2*Lambda[0] - Lambda[1] + delta
sage: L.null_root()                                                            #␣
↪needs sage.graphs
2*delta
```

```
>>> from sage.all import *
>>> L = RootSystem(CartanType(["BC",Integer(4),Integer(2)])).weight_
↪lattice(extended=True)
>>> L.simple_root(Integer(0))                                                  ␣
↪     # needs sage.graphs
2*Lambda[0] - Lambda[1] + delta
>>> L.null_root()                                                            #␣
↪needs sage.graphs
2*delta
```

See also:

- *simple_root()*
- CartanType.col_annihilator()

to_ambient_space_morphism()

The morphism from self to its associated ambient space.

EXAMPLES:

```
sage: CartanType(['A',2]).root_system().weight_lattice().to_ambient_space_
↪morphism()
Generic morphism:
From: Weight lattice of the Root system of type ['A', 2]
To:   Ambient space of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> CartanType(['A',Integer(2)]).root_system().weight_lattice().to_ambient_
↪space_morphism()
Generic morphism:
From: Weight lattice of the Root system of type ['A', 2]
To:   Ambient space of the Root system of type ['A', 2]
```

Warning: Implemented only for finite Cartan type.

class sage.combinat.root_system.weight_space.**WeightSpaceElement**

Bases: IndexedFreeModuleElement

is_dominant()

Check whether an element in the weight space lies in the positive cone spanned by the basis elements (fundamental weights).

EXAMPLES:

```
sage: W = RootSystem(['A',3]).weight_space()
sage: Lambda = W.basis()
sage: w = Lambda[1] + Lambda[3]
sage: w.is_dominant()
True
sage: w = Lambda[1] - Lambda[2]
sage: w.is_dominant()
False
```

```
>>> from sage.all import *
>>> W = RootSystem(['A',Integer(3)]).weight_space()
>>> Lambda = W.basis()
>>> w = Lambda[Integer(1)] + Lambda[Integer(3)]
>>> w.is_dominant()
True
>>> w = Lambda[Integer(1)] - Lambda[Integer(2)]
>>> w.is_dominant()
False
```

In the extended affine weight lattice, `'delta'` is orthogonal to the positive coroots, so adding or subtracting it should not affect dominance:

```
sage: P = RootSystem(['A',2,1]).weight_lattice(extended=true)
sage: Lambda = P.fundamental_weights()
```

(continues on next page)

(continued from previous page)

```
sage: delta = P.null_root()                                             #␣
↪needs sage.graphs
sage: w = Lambda[1] - delta                                             #␣
↪needs sage.graphs
sage: w.is_dominant()                                                   #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> P = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice(extended=true)
>>> Lambda = P.fundamental_weights()
>>> delta = P.null_root()                                             #␣
↪needs sage.graphs
>>> w = Lambda[Integer(1)] - delta                                      ␣
↪     # needs sage.graphs
>>> w.is_dominant()                                                   #␣
↪needs sage.graphs
True
```

is_dominant_weight()

Check whether an element in the weight space lies in the positive $\mathbf{Z}$-lattice cone spanned by the basis elements (fundamental weights).

EXAMPLES:

```
sage: W = RootSystem(['A',3]).weight_space()
sage: Lambda = W.basis()
sage: w = Lambda[1] + Lambda[3]
sage: w.is_dominant_weight()
True
sage: w = Lambda[1] + 2/3*Lambda[3]
sage: w.is_dominant_weight()
False
sage: w = Lambda[1] - Lambda[2]
sage: w.is_dominant_weight()
False
```

```
>>> from sage.all import *
>>> W = RootSystem(['A',Integer(3)]).weight_space()
>>> Lambda = W.basis()
>>> w = Lambda[Integer(1)] + Lambda[Integer(3)]
>>> w.is_dominant_weight()
True
>>> w = Lambda[Integer(1)] + Integer(2)/Integer(3)*Lambda[Integer(3)]
>>> w.is_dominant_weight()
False
>>> w = Lambda[Integer(1)] - Lambda[Integer(2)]
>>> w.is_dominant_weight()
False
```

In the extended affine weight lattice, `'delta'` is orthogonal to the positive coroots, so adding or subtracting it should not affect dominance:

```
sage: P = RootSystem(['A',2,1]).weight_lattice(extended=true)
sage: Lambda = P.fundamental_weights()
sage: delta = P.null_root()                                          #␣
↪needs sage.graphs
sage: w = Lambda[1] - delta                                          #␣
↪needs sage.graphs
sage: w.is_dominant_weight()                                         #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> P = RootSystem(['A',Integer(2),Integer(1)]).weight_lattice(extended=true)
>>> Lambda = P.fundamental_weights()
>>> delta = P.null_root()                                          #␣
↪needs sage.graphs
>>> w = Lambda[Integer(1)] - delta                                   ␣
↪     # needs sage.graphs
>>> w.is_dominant_weight()                                         #␣
↪needs sage.graphs
True
```

scalar(*lambdacheck*)

The canonical scalar product between the weight lattice and the coroot lattice.

Todo

- merge with_apply_multi_module_morphism
- allow for any root space / lattice
- define properly the return type (depends on the base rings of the two spaces)
- make this robust for extended weight lattices (i might be "delta")

EXAMPLES:

```
sage: L = RootSystem(["C",4,1]).weight_lattice()
sage: Lambda     = L.fundamental_weights()
sage: alphacheck = L.simple_coroots()
sage: Lambda[1].scalar(alphacheck[1])
1
sage: Lambda[1].scalar(alphacheck[2])
0
```

```
>>> from sage.all import *
>>> L = RootSystem(["C",Integer(4),Integer(1)]).weight_lattice()
>>> Lambda     = L.fundamental_weights()
>>> alphacheck = L.simple_coroots()
>>> Lambda[Integer(1)].scalar(alphacheck[Integer(1)])
1
>>> Lambda[Integer(1)].scalar(alphacheck[Integer(2)])
0
```

The fundamental weights and the simple coroots are dual bases:

```
sage: matrix([ [ Lambda[i].scalar(alphacheck[j])
....:              for i in L.index_set() ]
....:            for j in L.index_set() ])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

```
>>> from sage.all import *
>>> matrix([ [ Lambda[i].scalar(alphacheck[j])
...              for i in L.index_set() ]
...            for j in L.index_set() ])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

Note that the scalar product is not yet implemented between the weight space and the coweight space; in any cases, that won't be the job of this method:

```
sage: R = RootSystem(["A",3])
sage: alpha = R.weight_space().roots()                                    #␣
↪needs sage.graphs
sage: alphacheck = R.coweight_space().roots()                             #␣
↪needs sage.graphs
sage: alpha[1].scalar(alphacheck[1])                                      #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: -Lambdacheck[1] + 2*Lambdacheck[2] - Lambdacheck[3]
is not in the coroot space
```

```
>>> from sage.all import *
>>> R = RootSystem(["A",Integer(3)])
>>> alpha = R.weight_space().roots()                                    #␣
↪needs sage.graphs
>>> alphacheck = R.coweight_space().roots()                             #␣
↪needs sage.graphs
>>> alpha[Integer(1)].scalar(alphacheck[Integer(1)])                      ␣
↪              # needs sage.graphs
Traceback (most recent call last):
...
ValueError: -Lambdacheck[1] + 2*Lambdacheck[2] - Lambdacheck[3]
is not in the coroot space
```

to_ambient()

Map `self` to the ambient space.

EXAMPLES:

```
sage: mu = CartanType(['B',2]).root_system().weight_lattice().an_element(); mu
2*Lambda[1] + 2*Lambda[2]
sage: mu.to_ambient()
(3, 1)
```

```
>>> from sage.all import *
>>> mu = CartanType(['B',Integer(2)]).root_system().weight_lattice().an_
↪element(); mu
2*Lambda[1] + 2*Lambda[2]
>>> mu.to_ambient()
(3, 1)
```

Warning

Only implemented in finite Cartan type. Does not work for coweight lattices because there is no implemented map from the coweight lattice to the ambient space.

to_weight_space()

Map `self` to the weight space.

Since $self.parent()$ is the weight space, this map just returns `self`. This overrides the generic method in $WeightSpaceRealizations$.

EXAMPLES:

```
sage: mu = CartanType(['A',2]).root_system().weight_lattice().an_element(); mu
2*Lambda[1] + 2*Lambda[2]
sage: mu.to_weight_space()
2*Lambda[1] + 2*Lambda[2]
```

```
>>> from sage.all import *
>>> mu = CartanType(['A',Integer(2)]).root_system().weight_lattice().an_
↪element(); mu
2*Lambda[1] + 2*Lambda[2]
>>> mu.to_weight_space()
2*Lambda[1] + 2*Lambda[2]
```

5.1.272 Weyl Character Rings

class sage.combinat.root_system.weyl_characters.**WeightRing**(*parent*, *prefix*)

Bases: `CombinatorialFreeModule`

The weight ring, which is the group algebra over a weight lattice.

A Weyl character may be regarded as an element of the weight ring. In fact, an element of the weight ring is an element of the `Weyl character ring` if and only if it is invariant under the action of the Weyl group.

The advantage of the weight ring over the Weyl character ring is that one may conduct calculations in the weight ring that involve sums of weights that are not Weyl group invariant.

EXAMPLES:

```
sage: A2 = WeylCharacterRing(['A',2])
sage: a2 = WeightRing(A2)
sage: wd = prod(a2(x/2)-a2(-x/2) for x in a2.space().positive_roots()); wd
a2(-1,1,0) - a2(-1,0,1) - a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) - a2(0,1,-1)
sage: chi = A2([5,3,0]); chi
A2(5,3,0)
sage: a2(chi)
a2(1,2,5) + 2*a2(1,3,4) + 2*a2(1,4,3) + a2(1,5,2) + a2(2,1,5)
+ 2*a2(2,2,4) + 3*a2(2,3,3) + 2*a2(2,4,2) + a2(2,5,1) + 2*a2(3,1,4)
+ 3*a2(3,2,3) + 3*a2(3,3,2) + 2*a2(3,4,1) + a2(3,5,0) + a2(3,0,5)
+ 2*a2(4,1,3) + 2*a2(4,2,2) + 2*a2(4,3,1) + a2(4,4,0) + a2(4,0,4)
+ a2(5,1,2) + a2(5,2,1) + a2(5,3,0) + a2(5,0,3) + a2(0,3,5)
+ a2(0,4,4) + a2(0,5,3)
sage: a2(chi)*wd
-a2(-1,3,6) + a2(-1,6,3) + a2(3,-1,6) - a2(3,6,-1) - a2(6,-1,3) + a2(6,3,-1)
sage: sum((-1)^w.length()*a2([6,3,-1]).weyl_group_action(w) for w in a2.space().
↪weyl_group())
-a2(-1,3,6) + a2(-1,6,3) + a2(3,-1,6) - a2(3,6,-1) - a2(6,-1,3) + a2(6,3,-1)
sage: a2(chi)*wd == sum((-1)^w.length()*a2([6,3,-1]).weyl_group_action(w) for w␣
↪in a2.space().weyl_group())
True
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing(['A',Integer(2)])
>>> a2 = WeightRing(A2)
>>> wd = prod(a2(x/Integer(2))-a2(-x/Integer(2)) for x in a2.space().positive_
↪roots()); wd
a2(-1,1,0) - a2(-1,0,1) - a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) - a2(0,1,-1)
>>> chi = A2([Integer(5),Integer(3),Integer(0)]); chi
A2(5,3,0)
>>> a2(chi)
a2(1,2,5) + 2*a2(1,3,4) + 2*a2(1,4,3) + a2(1,5,2) + a2(2,1,5)
+ 2*a2(2,2,4) + 3*a2(2,3,3) + 2*a2(2,4,2) + a2(2,5,1) + 2*a2(3,1,4)
+ 3*a2(3,2,3) + 3*a2(3,3,2) + 2*a2(3,4,1) + a2(3,5,0) + a2(3,0,5)
+ 2*a2(4,1,3) + 2*a2(4,2,2) + 2*a2(4,3,1) + a2(4,4,0) + a2(4,0,4)
+ a2(5,1,2) + a2(5,2,1) + a2(5,3,0) + a2(5,0,3) + a2(0,3,5)
+ a2(0,4,4) + a2(0,5,3)
>>> a2(chi)*wd
-a2(-1,3,6) + a2(-1,6,3) + a2(3,-1,6) - a2(3,6,-1) - a2(6,-1,3) + a2(6,3,-1)
>>> sum((-Integer(1))**w.length()*a2([Integer(6),Integer(3),-Integer(1)]).weyl_
↪group_action(w) for w in a2.space().weyl_group())
-a2(-1,3,6) + a2(-1,6,3) + a2(3,-1,6) - a2(3,6,-1) - a2(6,-1,3) + a2(6,3,-1)
>>> a2(chi)*wd == sum((-Integer(1))**w.length()*a2([Integer(6),Integer(3),-
↪Integer(1)]).weyl_group_action(w) for w in a2.space().weyl_group())
True
```

class Element

Bases: `IndexedFreeModuleElement`

A class for weight ring elements.

cartan_type()

Return the Cartan type.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: a2([0,1,0]).cartan_type()
['A', 2]
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> a2 = WeightRing(A2)
>>> a2([Integer(0),Integer(1),Integer(0)]).cartan_type()
['A', 2]
```

character()

Assuming that `self` is invariant under the Weyl group, this will express it as a linear combination of characters. If `self` is not Weyl group invariant, this method will not terminate.

EXAMPLES:

```
sage: A2 = WeylCharacterRing(['A',2])
sage: a2 = WeightRing(A2)
sage: W = a2.space().weyl_group()
sage: mu = a2(2,1,0)
sage: nu = sum(mu.weyl_group_action(w) for w in W) ; nu
a2(1,2,0) + a2(1,0,2) + a2(2,1,0) + a2(2,0,1) + a2(0,1,2) + a2(0,2,1)
sage: nu.character()
-2*A2(1,1,1) + A2(2,1,0)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing(['A',Integer(2)])
>>> a2 = WeightRing(A2)
>>> W = a2.space().weyl_group()
>>> mu = a2(Integer(2),Integer(1),Integer(0))
>>> nu = sum(mu.weyl_group_action(w) for w in W) ; nu
a2(1,2,0) + a2(1,0,2) + a2(2,1,0) + a2(2,0,1) + a2(0,1,2) + a2(0,2,1)
>>> nu.character()
-2*A2(1,1,1) + A2(2,1,0)
```

demazure(*w*, *debug=False*)

Return the result of applying the Demazure operator ∂_w to `self`.

INPUT:

- `w` – a Weyl group element, or its reduced word

If $w = s_i$ is a simple reflection, the operation ∂_w sends the weight λ to

$$\frac{\lambda - s_i \cdot \lambda + \alpha_i}{1 + \alpha_i},$$

where the numerator is divisible the denominator in the weight ring. This is extended by multiplicativity to all w in the Weyl group.

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2",style='coroots')
sage: b2 = WeightRing(B2)
```

(continues on next page)

(continued from previous page)

```
sage: b2(1,0).demazure([1])
b2(1,0) + b2(-1,2)
sage: b2(1,0).demazure([2])
b2(1,0)
sage: r = b2(1,0).demazure([1,2]); r
b2(1,0) + b2(-1,2)
sage: r.demazure([1])
b2(1,0) + b2(-1,2)
sage: r.demazure([2])
b2(0,0) + b2(1,0) + b2(1,-2) + b2(-1,2)
```

```
>>> from sage.all import *
>>> B2 = WeylCharacterRing("B2",style='coroots')
>>> b2 = WeightRing(B2)
>>> b2(Integer(1),Integer(0)).demazure([Integer(1)])
b2(1,0) + b2(-1,2)
>>> b2(Integer(1),Integer(0)).demazure([Integer(2)])
b2(1,0)
>>> r = b2(Integer(1),Integer(0)).demazure([Integer(1),Integer(2)]); r
b2(1,0) + b2(-1,2)
>>> r.demazure([Integer(1)])
b2(1,0) + b2(-1,2)
>>> r.demazure([Integer(2)])
b2(0,0) + b2(1,0) + b2(1,-2) + b2(-1,2)
```

demazure_lusztig(*i*, *v*)

Return the result of applying the Demazure-Lusztig operator T_i to `self`.

INPUT:

- `i` – an element of the index set (or a reduced word or Weyl group element)
- `v` – an element of the base ring

If R is the parent WeightRing, the Demazure-Lusztig operator T_i is the linear map $R \to R$ that sends (for a weight λ) $R(\lambda)$ to

$$(R(\alpha_i) - 1)^{-1}\big(R(\lambda) - R(s_i\lambda) - v(R(\lambda) - R(\alpha_i + s_i\lambda))\big)$$

where the numerator is divisible by the denominator in R. The Demazure-Lusztig operators give a representation of the Iwahori–Hecke algebra associated to the Weyl group. See

- Lusztig, Equivariant K-theory and representations of Hecke algebras, Proc. Amer. Math. Soc. 94 (1985), no. 2, 337-342.
- Cherednik, *Nonsymmetric Macdonald polynomials*. IMRN 10, 483-515 (1995).

In the examples, we confirm the braid and quadratic relations for type B_2.

EXAMPLES:

```
sage: P.<v> = PolynomialRing(QQ)
sage: B2 = WeylCharacterRing("B2",style='coroots',base_ring=P); b2 = B2.
↪ambient()
sage: def T1(f): return f.demazure_lusztig(1, v)
sage: def T2(f): return f.demazure_lusztig(2, v)
sage: T1(T2(T1(T2(b2(1,-1)))))
(v^2-v)*b2(0,-1) + v^2*b2(-1,1)
sage: [T1(T1(f))==(v-1)*T1(f)+v*f for f in [b2(0,0), b2(1,0), b2(2,3)]]
```

(continues on next page)

(continued from previous page)

```
[True, True, True]
sage: [T1(T2(T1(T2(b2(i,j))))) == T2(T1(T2(T1(b2(i,j))))) for i in [-2..
↪2] for j in [-1,1]]
[True, True, True, True, True, True, True, True, True, True]
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ, names=('v',)); (v,) = P._first_ngens(1)
>>> B2 = WeylCharacterRing("B2",style='coroots',base_ring=P); b2 = B2.
↪ambient()
>>> def T1(f): return f.demazure_lusztig(Integer(1), v)
>>> def T2(f): return f.demazure_lusztig(Integer(2), v)
>>> T1(T2(T1(T2(b2(Integer(1),-Integer(1))))))
(v^2-v)*b2(0,-1) + v^2*b2(-1,1)
>>> [T1(T1(f))==(v-Integer(1))*T1(f)+v*f for f in [b2(Integer(0),
↪Integer(0)), b2(Integer(1),Integer(0)), b2(Integer(2),Integer(3))]]
[True, True, True]
>>> [T1(T2(T1(T2(b2(i,j))))) == T2(T1(T2(T1(b2(i,j))))) for i in␣
↪(ellipsis_range(-Integer(2),Ellipsis,Integer(2))) for j in [-Integer(1),
↪Integer(1)]]
[True, True, True, True, True, True, True, True, True, True]
```

Instead of an index i one may use a reduced word or Weyl group element:

```
sage: b2(1,0).demazure_lusztig([2,1],v)==T2(T1(b2(1,0)))
True
sage: W = B2.space().weyl_group(prefix='s')
sage: [s1,s2]=W.simple_reflections()
sage: b2(1,0).demazure_lusztig(s2*s1,v)==T2(T1(b2(1,0)))
True
```

```
>>> from sage.all import *
>>> b2(Integer(1),Integer(0)).demazure_lusztig([Integer(2),Integer(1)],
↪v)==T2(T1(b2(Integer(1),Integer(0))))
True
>>> W = B2.space().weyl_group(prefix='s')
>>> [s1,s2]=W.simple_reflections()
>>> b2(Integer(1),Integer(0)).demazure_lusztig(s2*s1,
↪v)==T2(T1(b2(Integer(1),Integer(0))))
True
```

scale(k)

Multiply a weight by k.

The operation is extended by linearity to the weight ring.

INPUT:

- k – nonzero integer

EXAMPLES:

```
sage: g2 = WeylCharacterRing("G2",style='coroots').ambient()
sage: g2(2,3).scale(2)
g2(4,6)
```

```
>>> from sage.all import *
>>> g2 = WeylCharacterRing("G2",style='coroots').ambient()
>>> g2(Integer(2),Integer(3)).scale(Integer(2))
g2(4,6)
```

shift(*mu*)

Add μ to any weight.

Extended by linearity to the weight ring.

INPUT:

- mu – a weight

EXAMPLES:

```
sage: g2 = WeylCharacterRing("G2",style='coroots').ambient()
sage: [g2(1,2).shift(fw) for fw in g2.fundamental_weights()]
[g2(2,2), g2(1,3)]
```

```
>>> from sage.all import *
>>> g2 = WeylCharacterRing("G2",style='coroots').ambient()
>>> [g2(Integer(1),Integer(2)).shift(fw) for fw in g2.fundamental_
↪weights()]
[g2(2,2), g2(1,3)]
```

weyl_group_action(*w*)

Return the action of the Weyl group element w on self.

EXAMPLES:

```
sage: G2 = WeylCharacterRing(['G',2])
sage: g2 = WeightRing(G2)
sage: L = g2.space()
sage: [fw1, fw2] = L.fundamental_weights()
sage: sum(g2(fw2).weyl_group_action(w) for w in L.weyl_group())
2*g2(-2,1,1) + 2*g2(-1,-1,2) + 2*g2(-1,2,-1) + 2*g2(1,-2,1) + 2*g2(1,1,-
↪2) + 2*g2(2,-1,-1)
```

```
>>> from sage.all import *
>>> G2 = WeylCharacterRing(['G',Integer(2)])
>>> g2 = WeightRing(G2)
>>> L = g2.space()
>>> [fw1, fw2] = L.fundamental_weights()
>>> sum(g2(fw2).weyl_group_action(w) for w in L.weyl_group())
2*g2(-2,1,1) + 2*g2(-1,-1,2) + 2*g2(-1,2,-1) + 2*g2(1,-2,1) + 2*g2(1,1,-
↪2) + 2*g2(2,-1,-1)
```

cartan_type()

Return the Cartan type.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: WeightRing(A2).cartan_type()
['A', 2]
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> WeightRing(A2).cartan_type()
['A', 2]
```

fundamental_weights()

Return the fundamental weights.

EXAMPLES:

```
sage: WeightRing(WeylCharacterRing("G2")).fundamental_weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

```
>>> from sage.all import *
>>> WeightRing(WeylCharacterRing("G2")).fundamental_weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

one_basis()

Return the index of 1.

EXAMPLES:

```
sage: A3 = WeylCharacterRing("A3")
sage: WeightRing(A3).one_basis()
(0, 0, 0, 0)
sage: WeightRing(A3).one()
a3(0,0,0,0)
```

```
>>> from sage.all import *
>>> A3 = WeylCharacterRing("A3")
>>> WeightRing(A3).one_basis()
(0, 0, 0, 0)
>>> WeightRing(A3).one()
a3(0,0,0,0)
```

parent()

Return the parent Weyl character ring.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: a2.parent()
The Weyl Character Ring of Type A2 with Integer Ring coefficients
sage: a2.parent() == A2
True
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> a2 = WeightRing(A2)
>>> a2.parent()
The Weyl Character Ring of Type A2 with Integer Ring coefficients
>>> a2.parent() == A2
True
```

positive_roots()

Return the positive roots.

EXAMPLES:

```
sage: WeightRing(WeylCharacterRing("G2")).positive_roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

```
>>> from sage.all import *
>>> WeightRing(WeylCharacterRing("G2")).positive_roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

product_on_basis(*a*, *b*)

Return the product of basis elements indexed by `a` and `b`.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: a2(1,0,0) * a2(0,1,0) # indirect doctest
a2(1,1,0)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> a2 = WeightRing(A2)
>>> a2(Integer(1),Integer(0),Integer(0)) * a2(Integer(0),Integer(1),
↪Integer(0)) # indirect doctest
a2(1,1,0)
```

simple_roots()

Return the simple roots.

EXAMPLES:

```
sage: WeightRing(WeylCharacterRing("G2")).simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

```
>>> from sage.all import *
>>> WeightRing(WeylCharacterRing("G2")).simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: A3 = WeylCharacterRing("A3")
sage: a3 = WeightRing(A3)
sage: a3.some_elements()
[a3(1,0,0,0), a3(1,1,0,0), a3(1,1,1,0)]
```

```
>>> from sage.all import *
>>> A3 = WeylCharacterRing("A3")
>>> a3 = WeightRing(A3)
```

(continues on next page)

(continued from previous page)

```
>>> a3.some_elements()
[a3(1,0,0,0), a3(1,1,0,0), a3(1,1,1,0)]
```

space()

Return the weight space realization associated to `self`.

EXAMPLES:

```
sage: E8 = WeylCharacterRing(['E',8])
sage: e8 = WeightRing(E8)
sage: e8.space()
Ambient space of the Root system of type ['E', 8]
```

```
>>> from sage.all import *
>>> E8 = WeylCharacterRing(['E',Integer(8)])
>>> e8 = WeightRing(E8)
>>> e8.space()
Ambient space of the Root system of type ['E', 8]
```

weyl_character_ring()

Return the parent Weyl Character Ring.

A synonym for `self.parent()`.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: a2.weyl_character_ring()
The Weyl Character Ring of Type A2 with Integer Ring coefficients
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> a2 = WeightRing(A2)
>>> a2.weyl_character_ring()
The Weyl Character Ring of Type A2 with Integer Ring coefficients
```

wt_repr(*wt*)

Return a string representing the irreducible character with highest weight vector `wt`.

Uses coroot notation if the associated Weyl character ring is defined with `style="coroots"`.

EXAMPLES:

```
sage: G2 = WeylCharacterRing("G2")
sage: [G2.ambient().wt_repr(x) for x in G2.fundamental_weights()]
['g2(1,0,-1)', 'g2(2,-1,-1)']
sage: G2 = WeylCharacterRing("G2",style='coroots')
sage: [G2.ambient().wt_repr(x) for x in G2.fundamental_weights()]
['g2(1,0)', 'g2(0,1)']
```

```
>>> from sage.all import *
>>> G2 = WeylCharacterRing("G2")
```

(continues on next page)

(continued from previous page)

```
>>> [G2.ambient().wt_repr(x) for x in G2.fundamental_weights()]
['g2(1,0,-1)', 'g2(2,-1,-1)']
>>> G2 = WeylCharacterRing("G2",style='coroots')
>>> [G2.ambient().wt_repr(x) for x in G2.fundamental_weights()]
['g2(1,0)', 'g2(0,1)']
```

class sage.combinat.root_system.weyl_characters.**WeylCharacterRing**(*ct*, *base_ring=Integer Ring*, *prefix=None*, *style='lattice'*, *k=None*, *conjugate=False*, *cyclotomic_order=None*, *fusion_labels=None*, *inject_variables=False*)

Bases: *CombinatorialFreeModule*

A class for rings of Weyl characters.

Let K be a compact Lie group, which we assume is semisimple and simply-connected. Its complexified Lie algebra L is the Lie algebra of a complex analytic Lie group G. The following three categories are equivalent: finite-dimensional representations of K; finite-dimensional representations of L; and finite-dimensional analytic representations of G. In every case, there is a parametrization of the irreducible representations by their highest weight vectors. For this theory of Weyl, see (for example):

- Adams, *Lectures on Lie groups*
- Broecker and Tom Dieck, *Representations of Compact Lie groups*
- Bump, *Lie Groups*
- Fulton and Harris, *Representation Theory*
- Goodman and Wallach, *Representations and Invariants of the Classical Groups*
- Hall, *Lie Groups, Lie Algebras and Representations*
- Humphreys, *Introduction to Lie Algebras and their representations*
- Procesi, *Lie Groups*
- Samelson, *Notes on Lie Algebras*
- Varadarajan, *Lie groups, Lie algebras, and their representations*
- Zhelobenko, *Compact Lie Groups and their Representations*.

Computations that you can do with these include computing their weight multiplicities, products (thus decomposing the tensor product of a representation into irreducibles) and branching rules (restriction to a smaller group).

There is associated with K, L or G as above a lattice, the weight lattice, whose elements (called weights) are characters of a Cartan subgroup or subalgebra. There is an action of the Weyl group W on the lattice, and elements of a fixed fundamental domain for W, the positive Weyl chamber, are called dominant. There is for each representation a unique highest dominant weight that occurs with nonzero multiplicity with respect to a certain partial order, and it is called the highest weight vector.

EXAMPLES:

```
sage: L = RootSystem("A2").ambient_space()
sage: [fw1,fw2] = L.fundamental_weights()
sage: R = WeylCharacterRing(['A',2], prefix='R')
sage: [R(1),R(fw1),R(fw2)]
[R(0,0,0), R(1,0,0), R(1,1,0)]
```

```
>>> from sage.all import *
>>> L = RootSystem("A2").ambient_space()
>>> [fw1,fw2] = L.fundamental_weights()
>>> R = WeylCharacterRing(['A',Integer(2)], prefix='R')
>>> [R(Integer(1)),R(fw1),R(fw2)]
[R(0,0,0), R(1,0,0), R(1,1,0)]
```

Here `R(1)`, `R(fw1)`, and `R(fw2)` are irreducible representations with highest weight vectors 0, Λ_1, and Λ_2 respectively (the first two fundamental weights).

For type A (also G_2, F_4, E_6 and E_7) we will take as the weight lattice not the weight lattice of the semisimple group, but for a larger one. For type A, this means we are concerned with the representation theory of $K = U(n)$ or $G = GL(n, \mathbf{C})$ rather than $SU(n)$ or $SU(n, \mathbf{C})$. This is useful since the representation theory of $GL(n)$ is ubiquitous, and also since we may then represent the fundamental weights (in *sage.combinat.root_system.root_system*) by vectors with integer entries. If you are only interested in $SL(3)$, say, use `WeylCharacterRing(['A',2])` as above but be aware that `R([a,b,c])` and `R([a+1,b+1,c+1])` represent the same character of $SL(3)$ since `R([1,1,1])` is the determinant.

For more information, see the thematic tutorial *Lie Methods and Related Combinatorics in Sage*, available at:

https://doc.sagemath.org/html/en/thematic_tutorials/lie.html

class Element

Bases: `IndexedFreeModuleElement`

A class for Weyl characters.

adams_operation(*r*)

Return the r-th Adams operation of `self`.

INPUT:

- `r` – positive integer

This is a virtual character, whose weights are the weights of `self`, each multiplied by r.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2(1,1,0).adams_operator(3)
A2(2,2,2) - A2(3,2,1) + A2(3,3,0)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> A2(Integer(1),Integer(1),Integer(0)).adams_operator(Integer(3))
A2(2,2,2) - A2(3,2,1) + A2(3,3,0)
```

adams_operator(*r*)

Return the r-th Adams operation of `self`.

INPUT:

- `r` – positive integer

This is a virtual character, whose weights are the weights of `self`, each multiplied by r.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2(1,1,0).adams_operator(3)
A2(2,2,2) - A2(3,2,1) + A2(3,3,0)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> A2(Integer(1),Integer(1),Integer(0)).adams_operator(Integer(3))
A2(2,2,2) - A2(3,2,1) + A2(3,3,0)
```

branch(*S*, *rule='default'*)

Return the restriction of the character to the subalgebra.

If no rule is specified, we will try to specify one.

INPUT:

- `S` – a Weyl character ring for a Lie subgroup or subalgebra
- `rule` – a branching rule

See `branch_weyl_character()` for more information about branching rules.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: A2 = WeylCharacterRing(['A',2])
sage: [B3(w).branch(A2,rule='levi') for w in B3.fundamental_weights()]
[A2(0,0,0) + A2(1,0,0) + A2(0,0,-1),
A2(0,0,0) + A2(1,0,0) + A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1) + A2(0,0,-1),
A2(-1/2,-1/2,-1/2) + A2(1/2,-1/2,-1/2) + A2(1/2,1/2,-1/2) + A2(1/2,1/2,1/
↪2)]
```

```
>>> from sage.all import *
>>> B3 = WeylCharacterRing(['B',Integer(3)])
>>> A2 = WeylCharacterRing(['A',Integer(2)])
>>> [B3(w).branch(A2,rule='levi') for w in B3.fundamental_weights()]
[A2(0,0,0) + A2(1,0,0) + A2(0,0,-1),
A2(0,0,0) + A2(1,0,0) + A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1) + A2(0,0,-1),
A2(-1/2,-1/2,-1/2) + A2(1/2,-1/2,-1/2) + A2(1/2,1/2,-1/2) + A2(1/2,1/2,1/
↪2)]
```

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2([1,0,0]).cartan_type()
['A', 2]
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> A2([Integer(1),Integer(0),Integer(0)]).cartan_type()
['A', 2]
```

degree()

Return the degree of `self`.

This is the dimension of the associated module.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: [B3(x).degree() for x in B3.fundamental_weights()]
[7, 21, 8]
```

```
>>> from sage.all import *
>>> B3 = WeylCharacterRing(['B',Integer(3)])
>>> [B3(x).degree() for x in B3.fundamental_weights()]
[7, 21, 8]
```

dual()

The involution that replaces a representation with its contragredient. (For Fusion rings, this is the conjugation map.)

EXAMPLES:

```
sage: A3 = WeylCharacterRing("A3", style='coroots')
sage: A3(1,0,0)^2
A3(0,1,0) + A3(2,0,0)
sage: (A3(1,0,0)^2).dual()
A3(0,1,0) + A3(0,0,2)
```

```
>>> from sage.all import *
>>> A3 = WeylCharacterRing("A3", style='coroots')
>>> A3(Integer(1),Integer(0),Integer(0))**Integer(2)
A3(0,1,0) + A3(2,0,0)
>>> (A3(Integer(1),Integer(0),Integer(0))**Integer(2)).dual()
A3(0,1,0) + A3(0,0,2)
```

exterior_power(k)

Return the k-th exterior power of `self`.

INPUT:

- `k` – nonnegative integer

The algorithm is based on the identity $ke_k = \sum_{r=1}^{k}(-1)^{k-1}p_k e_{k-r}$ relating the power-sum and elementary symmetric polynomials. Applying this to the eigenvalues of an element of the parent Lie group in the representation `self`, the e_k become exterior powers and the p_k become Adams operations, giving an efficient recursive implementation.

EXAMPLES:

```
sage: B3 = WeylCharacterRing("B3",style='coroots')
sage: spin = B3(0,0,1)
sage: spin.exterior_power(6)
B3(1,0,0) + B3(0,1,0)
```

```
>>> from sage.all import *
>>> B3 = WeylCharacterRing("B3",style='coroots')
>>> spin = B3(Integer(0),Integer(0),Integer(1))
>>> spin.exterior_power(Integer(6))
B3(1,0,0) + B3(0,1,0)
```

exterior_square()

Return the exterior square of the character.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2",style='coroots')
sage: A2(1,0).exterior_square()
A2(0,1)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2",style='coroots')
>>> A2(Integer(1),Integer(0)).exterior_square()
A2(0,1)
```

frobenius_schur_indicator()

Return:

- 1 if the representation is real (orthogonal)
- -1 if the representation is quaternionic (symplectic)
- 0 if the representation is complex (not self dual)

The Frobenius-Schur indicator of a character χ of a compact group G is the Haar integral over the group of $\chi(g^2)$. Its value is 1, -1 or 0. This method computes it for irreducible characters of compact Lie groups by checking whether the symmetric and exterior square characters contain the trivial character.

Todo

Try to compute this directly without actually calculating the full symmetric and exterior squares.

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2",style='coroots')
sage: B2(1,0).frobenius_schur_indicator()
 1
sage: B2(0,1).frobenius_schur_indicator()
 -1
```

```
>>> from sage.all import *
>>> B2 = WeylCharacterRing("B2",style='coroots')
>>> B2(Integer(1),Integer(0)).frobenius_schur_indicator()
 1
>>> B2(Integer(0),Integer(1)).frobenius_schur_indicator()
 -1
```

highest_weight()

Return the parametrizing dominant weight of an irreducible character.

This method is only available for basis elements.

EXAMPLES:

```
sage: G2 = WeylCharacterRing("G2", style='coroots')
sage: [x.highest_weight() for x in [G2(1,0),G2(0,1)]]
[(1, 0, -1), (2, -1, -1)]
```

```
>>> from sage.all import *
>>> G2 = WeylCharacterRing("G2", style='coroots')
```

(continues on next page)

(continued from previous page)

```
>>> [x.highest_weight() for x in [G2(Integer(1),Integer(0)),G2(Integer(0),
↪Integer(1))]]
[(1, 0, -1), (2, -1, -1)]
```

inner_product(*other*)

Compute the inner product with another character.

The irreducible characters are an orthonormal basis with respect to the usual inner product of characters, interpreted as functions on a compact Lie group, by Schur orthogonality.

INPUT:

- `other` – another character

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: [f1,f2] = A2.fundamental_weights()
sage: r1 = A2(f1)*A2(f2); r1
A2(1,1,1) + A2(2,1,0)
sage: r2 = A2(f1)^3; r2
A2(1,1,1) + 2*A2(2,1,0) + A2(3,0,0)
sage: r1.inner_product(r2)
3
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> [f1,f2] = A2.fundamental_weights()
>>> r1 = A2(f1)*A2(f2); r1
A2(1,1,1) + A2(2,1,0)
>>> r2 = A2(f1)**Integer(3); r2
A2(1,1,1) + 2*A2(2,1,0) + A2(3,0,0)
>>> r1.inner_product(r2)
3
```

invariant_degree()

Return the multiplicity of the trivial representation in `self`.

Multiplicities of other irreducibles may be obtained using `multiplicity()`.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2",style='coroots')
sage: rep = A2(1,0)^2*A2(0,1)^2; rep
2*A2(0,0) + A2(0,3) + 4*A2(1,1) + A2(3,0) + A2(2,2)
sage: rep.invariant_degree()
2
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2",style='coroots')
>>> rep = A2(Integer(1),Integer(0))**Integer(2)*A2(Integer(0),
↪Integer(1))**Integer(2); rep
2*A2(0,0) + A2(0,3) + 4*A2(1,1) + A2(3,0) + A2(2,2)
>>> rep.invariant_degree()
2
```

is_irreducible()

Return whether `self` is an irreducible character.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: [B3(x).is_irreducible() for x in B3.fundamental_weights()]
[True, True, True]
sage: sum(B3(x) for x in B3.fundamental_weights()).is_irreducible()
False
```

```
>>> from sage.all import *
>>> B3 = WeylCharacterRing(['B',Integer(3)])
>>> [B3(x).is_irreducible() for x in B3.fundamental_weights()]
[True, True, True]
>>> sum(B3(x) for x in B3.fundamental_weights()).is_irreducible()
False
```

multiplicity(*other*)

Return the multiplicity of the irreducible `other` in `self`.

INPUT:

- `other` – an irreducible character

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2",style='coroots')
sage: rep = B2(1,1)^2; rep
B2(0,0) + B2(1,0) + 2*B2(0,2) + B2(2,0) + 2*B2(1,2) + B2(0,4) + B2(3,0) +
→B2(2,2)
sage: rep.multiplicity(B2(0,2))
2
```

```
>>> from sage.all import *
>>> B2 = WeylCharacterRing("B2",style='coroots')
>>> rep = B2(Integer(1),Integer(1))**Integer(2); rep
B2(0,0) + B2(1,0) + 2*B2(0,2) + B2(2,0) + 2*B2(1,2) + B2(0,4) + B2(3,0) +
→B2(2,2)
>>> rep.multiplicity(B2(Integer(0),Integer(2)))
2
```

symmetric_power(k)

Return the k-th symmetric power of `self`.

INPUT:

- `k` – nonnegative integer

The algorithm is based on the identity $kh_k = \sum_{r=1}^{k} p_k h_{k-r}$ relating the power-sum and complete symmetric polynomials. Applying this to the eigenvalues of an element of the parent Lie group in the representation `self`, the h_k become symmetric powers and the p_k become Adams operations, giving an efficient recursive implementation.

EXAMPLES:

```
sage: B3 = WeylCharacterRing("B3",style='coroots')
sage: spin = B3(0,0,1)
```

(continues on next page)

(continued from previous page)

```
sage: spin.symmetric_power(6)
B3(0,0,0) + B3(0,0,2) + B3(0,0,4) + B3(0,0,6)
```

```
>>> from sage.all import *
>>> B3 = WeylCharacterRing("B3",style='coroots')
>>> spin = B3(Integer(0),Integer(0),Integer(1))
>>> spin.symmetric_power(Integer(6))
B3(0,0,0) + B3(0,0,2) + B3(0,0,4) + B3(0,0,6)
```

symmetric_square()

Return the symmetric square of the character.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2",style='coroots')
sage: A2(1,0).symmetric_square()
A2(2,0)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2",style='coroots')
>>> A2(Integer(1),Integer(0)).symmetric_square()
A2(2,0)
```

weight_multiplicities()

Return the dictionary of weight multiplicities for the Weyl character `self`.

The character does not have to be irreducible.

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2",style='coroots')
sage: B2(0,1).weight_multiplicities()
{(-1/2, -1/2): 1, (-1/2, 1/2): 1, (1/2, -1/2): 1, (1/2, 1/2): 1}
```

```
>>> from sage.all import *
>>> B2 = WeylCharacterRing("B2",style='coroots')
>>> B2(Integer(0),Integer(1)).weight_multiplicities()
{(-1/2, -1/2): 1, (-1/2, 1/2): 1, (1/2, -1/2): 1, (1/2, 1/2): 1}
```

adjoint_representation()

Return the adjoint representation as an element of the WeylCharacterRing.

EXAMPLES:

```
sage: G2 = WeylCharacterRing("G2",style='coroots')
sage: G2.adjoint_representation()
G2(0,1)
```

```
>>> from sage.all import *
>>> G2 = WeylCharacterRing("G2",style='coroots')
>>> G2.adjoint_representation()
G2(0,1)
```

affine_reflect(*wt*, *k=0*)

Return the reflection of wt in the hyperplane θ.

Optionally, this also shifts by a multiple k of θ.

INPUT:

- wt – a weight
- k – (optional) a positive integer

EXAMPLES:

```
sage: B22 = FusionRing("B2",2)
sage: fw = B22.fundamental_weights(); fw
Finite family {1: (1, 0), 2: (1/2, 1/2)}
sage: [B22.affine_reflect(x,2) for x in fw]
[(2, 1), (3/2, 3/2)]
```

```
>>> from sage.all import *
>>> B22 = FusionRing("B2",Integer(2))
>>> fw = B22.fundamental_weights(); fw
Finite family {1: (1, 0), 2: (1/2, 1/2)}
>>> [B22.affine_reflect(x,Integer(2)) for x in fw]
[(2, 1), (3/2, 3/2)]
```

ambient()

Return the weight ring of self.

EXAMPLES:

```
sage: WeylCharacterRing("A2").ambient()
The Weight ring attached to The Weyl Character Ring of Type A2 with Integer␣
↪Ring coefficients
```

```
>>> from sage.all import *
>>> WeylCharacterRing("A2").ambient()
The Weight ring attached to The Weyl Character Ring of Type A2 with Integer␣
↪Ring coefficients
```

base_ring()

Return the base ring of self.

EXAMPLES:

```
sage: R = WeylCharacterRing(['A',3], base_ring = CC); R.base_ring()
Complex Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> R = WeylCharacterRing(['A',Integer(3)], base_ring = CC); R.base_ring()
Complex Field with 53 bits of precision
```

cartan_type()

Return the Cartan type of self.

EXAMPLES:

```
sage: WeylCharacterRing("A2").cartan_type()
['A', 2]
```

```
>>> from sage.all import *
>>> WeylCharacterRing("A2").cartan_type()
['A', 2]
```

char_from_weights(*mdict*)

Construct a Weyl character from an invariant linear combination of weights.

INPUT:

- `mdict` – dictionary mapping weights to coefficients, and representing a linear combination of weights which shall be invariant under the action of the Weyl group

OUTPUT: the corresponding Weyl character

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: v = A2._space([3,1,0]); v
(3, 1, 0)
sage: d = dict([(x,1) for x in v.orbit()]); d
{(1, 3, 0): 1,
 (1, 0, 3): 1,
 (3, 1, 0): 1,
 (3, 0, 1): 1,
 (0, 1, 3): 1,
 (0, 3, 1): 1}
sage: A2.char_from_weights(d)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> v = A2._space([Integer(3),Integer(1),Integer(0)]); v
(3, 1, 0)
>>> d = dict([(x,Integer(1)) for x in v.orbit()]); d
{(1, 3, 0): 1,
 (1, 0, 3): 1,
 (3, 1, 0): 1,
 (3, 0, 1): 1,
 (0, 1, 3): 1,
 (0, 3, 1): 1}
>>> A2.char_from_weights(d)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
```

demazure_character(*hwv*, *word*, *debug=False*)

Compute the Demazure character.

INPUT:

- `hwv` – a (usually dominant) weight
- `word` – a Weyl group word

Produces the Demazure character with highest weight `hwv` and `word` as an element of the weight ring. Only available if `style="coroots"`. The Demazure operators are also available as methods of `WeightRing` elements, and as methods of crystals. Given a *CrystalOfTableaux* with given highest weight vector, the Demazure method on the crystal will give the equivalent of this method, except that the Demazure character of the crystal is given as a sum of monomials instead of an element of the *WeightRing*.

See *WeightRing.Element.demazure()* and sage.categories.classical_crystals.ClassicalCrystals.ParentMethods.demazure_character()

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2",style='coroots')
sage: h = sum(A2.fundamental_weights()); h
(2, 1, 0)
sage: A2.demazure_character(h,word=[1,2])
a2(0,0) + a2(-2,1) + a2(2,-1) + a2(1,1) + a2(-1,2)
sage: A2.demazure_character((1,1),word=[1,2])
a2(0,0) + a2(-2,1) + a2(2,-1) + a2(1,1) + a2(-1,2)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2",style='coroots')
>>> h = sum(A2.fundamental_weights()); h
(2, 1, 0)
>>> A2.demazure_character(h,word=[Integer(1),Integer(2)])
a2(0,0) + a2(-2,1) + a2(2,-1) + a2(1,1) + a2(-1,2)
>>> A2.demazure_character((Integer(1),Integer(1)),word=[Integer(1),
↪Integer(2)])
a2(0,0) + a2(-2,1) + a2(2,-1) + a2(1,1) + a2(-1,2)
```

dot_reduce(*a*)

Auxiliary function for *product_on_basis()*.

Return a pair $[\epsilon, b]$ where b is a dominant weight and ϵ is 0, 1 or -1. To describe b, let w be an element of the Weyl group such that $w(a+\rho)$ is dominant. If $w(a+\rho)-\rho$ is dominant, then ϵ is the sign of w and b is $w(a+\rho)-\rho$. Otherwise, ϵ is zero.

INPUT:

- a – a weight

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: weights = sorted(A2(2,1,0).weight_multiplicities().keys(), key=str);␣
↪weights
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 1, 1), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
sage: [A2.dot_reduce(x) for x in weights]
[[0, (0, 0, 0)], [-1, (1, 1, 1)], [-1, (1, 1, 1)], [1, (1, 1, 1)], [0, (0, 0,␣
↪0)], [0, (0, 0, 0)], [1, (2, 1, 0)]]
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> weights = sorted(A2(Integer(2),Integer(1),Integer(0)).weight_
↪multiplicities().keys(), key=str); weights
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 1, 1), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
>>> [A2.dot_reduce(x) for x in weights]
[[0, (0, 0, 0)], [-1, (1, 1, 1)], [-1, (1, 1, 1)], [1, (1, 1, 1)], [0, (0, 0,␣
↪0)], [0, (0, 0, 0)], [1, (2, 1, 0)]]
```

dynkin_diagram()

Return the Dynkin diagram of `self`.

EXAMPLES:

```
sage: WeylCharacterRing("E7").dynkin_diagram()
        O 2
        |
        |
O---O---O---O---O---O
1   3   4   5   6   7
E7
```

```
>>> from sage.all import *
>>> WeylCharacterRing("E7").dynkin_diagram()
        O 2
        |
        |
O---O---O---O---O---O
1   3   4   5   6   7
E7
```

extended_dynkin_diagram()

Return the extended Dynkin diagram, which is the Dynkin diagram of the corresponding untwisted affine type.

EXAMPLES:

```
sage: WeylCharacterRing("E7").extended_dynkin_diagram()
            O 2
            |
            |
O---O---O---O---O---O---O
0   1   3   4   5   6   7
E7~
```

```
>>> from sage.all import *
>>> WeylCharacterRing("E7").extended_dynkin_diagram()
            O 2
            |
            |
O---O---O---O---O---O---O
0   1   3   4   5   6   7
E7~
```

fundamental_weights()

Return the fundamental weights.

EXAMPLES:

```
sage: WeylCharacterRing("G2").fundamental_weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

```
>>> from sage.all import *
>>> WeylCharacterRing("G2").fundamental_weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

highest_root()

Return the highest root.

EXAMPLES:

```
sage: WeylCharacterRing("G2").highest_root()
 (2, -1, -1)
```

```
>>> from sage.all import *
>>> WeylCharacterRing("G2").highest_root()
 (2, -1, -1)
```

irr_repr(*hwv*)

Return a string representing the irreducible character with highest weight vector `hwv`.

EXAMPLES:

```
sage: B3 = WeylCharacterRing("B3")
sage: [B3.irr_repr(v) for v in B3.fundamental_weights()]
['B3(1,0,0)', 'B3(1,1,0)', 'B3(1/2,1/2,1/2)']
sage: B3 = WeylCharacterRing("B3", style='coroots')
sage: [B3.irr_repr(v) for v in B3.fundamental_weights()]
['B3(1,0,0)', 'B3(0,1,0)', 'B3(0,0,1)']
```

```
>>> from sage.all import *
>>> B3 = WeylCharacterRing("B3")
>>> [B3.irr_repr(v) for v in B3.fundamental_weights()]
['B3(1,0,0)', 'B3(1,1,0)', 'B3(1/2,1/2,1/2)']
>>> B3 = WeylCharacterRing("B3", style='coroots')
>>> [B3.irr_repr(v) for v in B3.fundamental_weights()]
['B3(1,0,0)', 'B3(0,1,0)', 'B3(0,0,1)']
```

level(*wt*)

Return the level of the weight, defined to be the value of the weight on the coroot associated with the highest root.

EXAMPLES:

```
sage: R = FusionRing("F4",2); [R.level(x) for x in R.fundamental_weights()]
[2, 3, 2, 1]
sage: [CartanType("F4~").dual().a()[x] for x in [1..4]]
[2, 3, 2, 1]
```

```
>>> from sage.all import *
>>> R = FusionRing("F4",Integer(2)); [R.level(x) for x in R.fundamental_
↪weights()]
[2, 3, 2, 1]
>>> [CartanType("F4~").dual().a()[x] for x in (ellipsis_range(Integer(1),
↪Ellipsis,Integer(4)))]
[2, 3, 2, 1]
```

lift()

The embedding of `self` into its weight ring.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2.lift
Generic morphism:
  From: The Weyl Character Ring of Type A2 with Integer Ring coefficients
  To:   The Weight ring attached to The Weyl Character Ring of Type A2 with␣
↪Integer Ring coefficients
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> A2.lift
Generic morphism:
  From: The Weyl Character Ring of Type A2 with Integer Ring coefficients
  To:   The Weight ring attached to The Weyl Character Ring of Type A2 with␣
↪Integer Ring coefficients
```

```
sage: x = -A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
sage: A2.lift(x)
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

```
>>> from sage.all import *
>>> x = -A2(Integer(2),Integer(1),Integer(1)) - A2(Integer(2),Integer(2),
↪Integer(0)) + A2(Integer(3),Integer(1),Integer(0))
>>> A2.lift(x)
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

As a shortcut, you may also do:

```
sage: x.lift()
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

```
>>> from sage.all import *
>>> x.lift()
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

Or even:

```
sage: a2 = WeightRing(A2)
sage: a2(x)
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

```
>>> from sage.all import *
>>> a2 = WeightRing(A2)
>>> a2(x)
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

lift_on_basis(*irr*)

Expand the basis element indexed by the weight `irr` into the weight ring of `self`.

INPUT:

- `irr` – a dominant weight

This is used to implement *lift()*.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: v = A2._space([2,1,0]); v
(2, 1, 0)
sage: A2.lift_on_basis(v)
2*a2(1,1,1) + a2(1,2,0) + a2(1,0,2) + a2(2,1,0) + a2(2,0,1) + a2(0,1,2) +␣
↪a2(0,2,1)
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> v = A2._space([Integer(2),Integer(1),Integer(0)]); v
(2, 1, 0)
>>> A2.lift_on_basis(v)
2*a2(1,1,1) + a2(1,2,0) + a2(1,0,2) + a2(2,1,0) + a2(2,0,1) + a2(0,1,2) +␣
↪a2(0,2,1)
```

This is consistent with the analogous calculation with symmetric Schur functions:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s[2,1].expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s[Integer(2),Integer(1)].expand(Integer(3))
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
```

maximal_subgroup(*ct*)

Return a branching rule or a list of branching rules.

INPUT:

- `ct` – the Cartan type of a maximal subgroup of `self`

In rare cases where there is more than one maximal subgroup (up to outer automorphisms) with the given Cartan type, the function returns a list of branching rules.

EXAMPLES:

```
sage: WeylCharacterRing("E7").maximal_subgroup("A2")
miscellaneous branching rule E7 => A2
sage: WeylCharacterRing("E7").maximal_subgroup("A1")
[iii branching rule E7 => A1, iv branching rule E7 => A1]
```

```
>>> from sage.all import *
>>> WeylCharacterRing("E7").maximal_subgroup("A2")
miscellaneous branching rule E7 => A2
>>> WeylCharacterRing("E7").maximal_subgroup("A1")
[iii branching rule E7 => A1, iv branching rule E7 => A1]
```

For more information, see the related method *maximal_subgroups()*.

maximal_subgroups()

This method is only available if the Cartan type of `self` is irreducible and of rank no greater than 8. This method produces a list of the maximal subgroups of `self`, up to (possibly outer) automorphisms. Each line in the output gives the Cartan type of a maximal subgroup followed by a command that creates the branching rule.

EXAMPLES:

```
sage: WeylCharacterRing("E6").maximal_subgroups()
D5:branching_rule("E6","D5","levi")
C4:branching_rule("E6","C4","symmetric")
F4:branching_rule("E6","F4","symmetric")
A2:branching_rule("E6","A2","miscellaneous")
G2:branching_rule("E6","G2","miscellaneous")
A2xG2:branching_rule("E6","A2xG2","miscellaneous")
A1xA5:branching_rule("E6","A1xA5","extended")
A2xA2xA2:branching_rule("E6","A2xA2xA2","extended")
```

```
>>> from sage.all import *
>>> WeylCharacterRing("E6").maximal_subgroups()
D5:branching_rule("E6","D5","levi")
C4:branching_rule("E6","C4","symmetric")
F4:branching_rule("E6","F4","symmetric")
A2:branching_rule("E6","A2","miscellaneous")
G2:branching_rule("E6","G2","miscellaneous")
A2xG2:branching_rule("E6","A2xG2","miscellaneous")
A1xA5:branching_rule("E6","A1xA5","extended")
A2xA2xA2:branching_rule("E6","A2xA2xA2","extended")
```

Note that there are other embeddings of (for example A_2 into E_6 as nonmaximal subgroups. These embeddings may be constructed by composing branching rules through various subgroups.

Once you know which maximal subgroup you are interested in, to create the branching rule, you may either paste the command to the right of the colon from the above output onto the command line, or alternatively invoke the related method *maximal_subgroup()*:

```
sage: branching_rule("E6","G2","miscellaneous")
miscellaneous branching rule E6 => G2
sage: WeylCharacterRing("E6").maximal_subgroup("G2")
miscellaneous branching rule E6 => G2
```

```
>>> from sage.all import *
>>> branching_rule("E6","G2","miscellaneous")
miscellaneous branching rule E6 => G2
>>> WeylCharacterRing("E6").maximal_subgroup("G2")
miscellaneous branching rule E6 => G2
```

It is believed that the list of maximal subgroups is complete, except that some subgroups may be not be invariant under outer automorphisms. It is reasonable to want a list of maximal subgroups that is complete up to conjugation, but to obtain such a list you may have to apply outer automorphisms. The group of outer automorphisms modulo inner automorphisms is isomorphic to the group of symmetries of the Dynkin diagram, and these are available as branching rules. The following example shows that while a branching rule from D_4 to $A_1 \times C_2$ is supplied, another different one may be obtained by composing it with the triality automorphism of D_4:

```
sage: [D4,A1xC2]=[WeylCharacterRing(x,style='coroots') for x in ["D4","A1xC2
↪"]]
sage: fw = D4.fundamental_weights()
sage: b = D4.maximal_subgroup("A1xC2")
sage: [D4(fw).branch(A1xC2,rule=b) for fw in D4.fundamental_weights()]
[A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(2,0,1) + A1xC2(0,2,0),
A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(0,0,1)]
sage: b1 = branching_rule("D4","D4","triality")*b
sage: [D4(fw).branch(A1xC2,rule=b1) for fw in D4.fundamental_weights()]
[A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(2,0,1) + A1xC2(0,2,0),
A1xC2(2,0,0) + A1xC2(0,0,1),
A1xC2(1,1,0)]
```

```
>>> from sage.all import *
>>> [D4,A1xC2]=[WeylCharacterRing(x,style='coroots') for x in ["D4","A1xC2"]]
>>> fw = D4.fundamental_weights()
>>> b = D4.maximal_subgroup("A1xC2")
>>> [D4(fw).branch(A1xC2,rule=b) for fw in D4.fundamental_weights()]
[A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(2,0,1) + A1xC2(0,2,0),
A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(0,0,1)]
>>> b1 = branching_rule("D4","D4","triality")*b
>>> [D4(fw).branch(A1xC2,rule=b1) for fw in D4.fundamental_weights()]
[A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(2,0,1) + A1xC2(0,2,0),
A1xC2(2,0,0) + A1xC2(0,0,1),
A1xC2(1,1,0)]
```

one_basis()

Return the index of 1 in `self`.

EXAMPLES:

```
sage: WeylCharacterRing("A3").one_basis()
(0, 0, 0, 0)
sage: WeylCharacterRing("A3").one()
A3(0,0,0,0)
```

```
>>> from sage.all import *
>>> WeylCharacterRing("A3").one_basis()
(0, 0, 0, 0)
>>> WeylCharacterRing("A3").one()
A3(0,0,0,0)
```

positive_roots()

Return the positive roots.

EXAMPLES:

```
sage: WeylCharacterRing("G2").positive_roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

```
>>> from sage.all import *
>>> WeylCharacterRing("G2").positive_roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

product_on_basis(*a*, *b*)

Compute the tensor product of two irreducible representations `a` and `b`.

EXAMPLES:

```
sage: D4 = WeylCharacterRing(['D',4])
sage: spin_plus = D4(1/2,1/2,1/2,1/2)
sage: spin_minus = D4(1/2,1/2,1/2,-1/2)
sage: spin_plus * spin_minus # indirect doctest
D4(1,0,0,0) + D4(1,1,1,0)
sage: spin_minus * spin_plus
D4(1,0,0,0) + D4(1,1,1,0)
```

```
>>> from sage.all import *
>>> D4 = WeylCharacterRing(['D',Integer(4)])
>>> spin_plus = D4(Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(1)/
↪Integer(2),Integer(1)/Integer(2))
>>> spin_minus = D4(Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(1)/
↪Integer(2),-Integer(1)/Integer(2))
>>> spin_plus * spin_minus # indirect doctest
D4(1,0,0,0) + D4(1,1,1,0)
>>> spin_minus * spin_plus
D4(1,0,0,0) + D4(1,1,1,0)
```

Uses the Brauer-Klimyk method.

rank()

Return the rank.

EXAMPLES:

```
sage: WeylCharacterRing("G2").rank()
2
```

```
>>> from sage.all import *
>>> WeylCharacterRing("G2").rank()
2
```

retract()

The partial inverse map from the weight ring into `self`.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: A2.retract
Generic morphism:
```

(continues on next page)

(continued from previous page)

```
  From: The Weight ring attached to The Weyl Character Ring of Type A2 with␣
↪Integer Ring coefficients
  To:   The Weyl Character Ring of Type A2 with Integer Ring coefficients
```

```
>>> from sage.all import *
>>> A2 = WeylCharacterRing("A2")
>>> a2 = WeightRing(A2)
>>> A2.retract
Generic morphism:
  From: The Weight ring attached to The Weyl Character Ring of Type A2 with␣
↪Integer Ring coefficients
  To:   The Weyl Character Ring of Type A2 with Integer Ring coefficients
```

```
sage: v = A2._space([3,1,0]); v
(3, 1, 0)
sage: chi = a2.sum_of_monomials(v.orbit()); chi
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
sage: A2.retract(chi)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
```

```
>>> from sage.all import *
>>> v = A2._space([Integer(3),Integer(1),Integer(0)]); v
(3, 1, 0)
>>> chi = a2.sum_of_monomials(v.orbit()); chi
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
>>> A2.retract(chi)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
```

The input should be invariant:

```
sage: A2.retract(a2.monomial(v))
Traceback (most recent call last):
...
ValueError: multiplicity dictionary may not be Weyl group invariant
```

```
>>> from sage.all import *
>>> A2.retract(a2.monomial(v))
Traceback (most recent call last):
...
ValueError: multiplicity dictionary may not be Weyl group invariant
```

As a shortcut, you may use conversion:

```
sage: A2(chi)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
sage: A2(a2.monomial(v))
Traceback (most recent call last):
...
ValueError: multiplicity dictionary may not be Weyl group invariant
```

```
>>> from sage.all import *
>>> A2(chi)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
>>> A2(a2.monomial(v))
Traceback (most recent call last):
...
ValueError: multiplicity dictionary may not be Weyl group invariant
```

simple_coroots()

Return the simple coroots.

EXAMPLES:

```
sage: WeylCharacterRing("G2").simple_coroots()
Finite family {1: (0, 1, -1), 2: (1/3, -2/3, 1/3)}
```

```
>>> from sage.all import *
>>> WeylCharacterRing("G2").simple_coroots()
Finite family {1: (0, 1, -1), 2: (1/3, -2/3, 1/3)}
```

simple_roots()

Return the simple roots.

EXAMPLES:

```
sage: WeylCharacterRing("G2").simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

```
>>> from sage.all import *
>>> WeylCharacterRing("G2").simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: WeylCharacterRing("A3").some_elements()
[A3(1,0,0,0), A3(1,1,0,0), A3(1,1,1,0)]
```

```
>>> from sage.all import *
>>> WeylCharacterRing("A3").some_elements()
[A3(1,0,0,0), A3(1,1,0,0), A3(1,1,1,0)]
```

space()

Return the weight space associated to `self`.

EXAMPLES:

```
sage: WeylCharacterRing(['E',8]).space()
Ambient space of the Root system of type ['E', 8]
```

```
>>> from sage.all import *
>>> WeylCharacterRing(['E',Integer(8)]).space()
Ambient space of the Root system of type ['E', 8]
```

sage.combinat.root_system.weyl_characters.**irreducible_character_freudenthal**(*hwv*, *debug=False*)

Return the dictionary of multiplicities for the irreducible character with highest weight λ.

The weight multiplicities are computed by the Freudenthal multiplicity formula. The algorithm is based on recursion relation that is stated, for example, in Humphrey's book on Lie Algebras. The multiplicities are invariant under the Weyl group, so to compute them it would be sufficient to compute them for the weights in the positive Weyl chamber. However after some testing it was found to be faster to compute every weight using the recursion, since the use of the Weyl group is expensive in its current implementation.

INPUT:

- hwv – a dominant weight in a weight lattice
- L – the ambient space

EXAMPLES:

```
sage: WeylCharacterRing("A2")(2,1,0).weight_multiplicities() # indirect doctest
{(1, 1, 1): 2, (1, 2, 0): 1, (1, 0, 2): 1, (2, 1, 0): 1,
 (2, 0, 1): 1, (0, 1, 2): 1, (0, 2, 1): 1}
```

```
>>> from sage.all import *
>>> WeylCharacterRing("A2")(Integer(2),Integer(1),Integer(0)).weight_
↪multiplicities() # indirect doctest
{(1, 1, 1): 2, (1, 2, 0): 1, (1, 0, 2): 1, (2, 1, 0): 1,
 (2, 0, 1): 1, (0, 1, 2): 1, (0, 2, 1): 1}
```

5.1.273 Weyl Groups

AUTHORS:

- Daniel Bump (2008): initial version
- Mike Hansen (2008): initial version
- Anne Schilling (2008): initial version
- Nicolas Thiéry (2008): initial version
- Volker Braun (2013): LibGAP-based matrix groups

EXAMPLES:

The Cayley graph of the Weyl Group of type ['A', 3]:

```
sage: w = WeylGroup(['A',3])
sage: d = w.cayley_graph(); d
Digraph on 24 vertices
sage: d.show3d(color_by_label=True, edge_size=0.01, vertex_size=0.03)            #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> w = WeylGroup(['A',Integer(3)])
>>> d = w.cayley_graph(); d
Digraph on 24 vertices
>>> d.show3d(color_by_label=True, edge_size=RealNumber('0.01'), vertex_
↪size=RealNumber('0.03'))                # needs sage.plot
```

The Cayley graph of the Weyl Group of type ['D', 4]:

```
sage: w = WeylGroup(['D',4])
sage: d = w.cayley_graph(); d
Digraph on 192 vertices
sage: d.show3d(color_by_label=True, edge_size=0.01, vertex_size=0.03)        # long␣
↪time (less than one minute), needs sage.plot
```

```
>>> from sage.all import *
>>> w = WeylGroup(['D',Integer(4)])
>>> d = w.cayley_graph(); d
Digraph on 192 vertices
>>> d.show3d(color_by_label=True, edge_size=RealNumber('0.01'), vertex_
↪size=RealNumber('0.03'))        # long time (less than one minute), needs sage.plot
```

Todo

More examples on Weyl Groups should be added here.

class sage.combinat.root_system.weyl_group.**ClassicalWeylSubgroup**(*domain*, *prefix*)

Bases: *WeylGroup_gens*

A class for Classical Weyl Subgroup of an affine Weyl Group.

EXAMPLES:

```
sage: G = WeylGroup(["A",3,1]).classical()
sage: G
Parabolic Subgroup of the Weyl Group of type ['A', 3, 1] (as a matrix group␣
↪acting on the root space)
sage: G.category()
Category of finite irreducible Weyl groups
sage: G.cardinality()
24
sage: G.index_set()
(1, 2, 3)
sage: TestSuite(G).run()
```

```
>>> from sage.all import *
>>> G = WeylGroup(["A",Integer(3),Integer(1)]).classical()
>>> G
Parabolic Subgroup of the Weyl Group of type ['A', 3, 1] (as a matrix group␣
↪acting on the root space)
>>> G.category()
Category of finite irreducible Weyl groups
```

(continues on next page)

(continued from previous page)

```
>>> G.cardinality()
24
>>> G.index_set()
(1, 2, 3)
>>> TestSuite(G).run()
```

Todo

implement:

- Parabolic subrootsystems
- Parabolic subgroups with a set of nodes as argument

cartan_type()

EXAMPLES:

```
sage: WeylGroup(['A',3,1]).classical().cartan_type()
['A', 3]
sage: WeylGroup(['A',3,1]).classical().index_set()
(1, 2, 3)
```

```
>>> from sage.all import *
>>> WeylGroup(['A',Integer(3),Integer(1)]).classical().cartan_type()
['A', 3]
>>> WeylGroup(['A',Integer(3),Integer(1)]).classical().index_set()
(1, 2, 3)
```

Note: won't be needed, once the lattice will be a parabolic sub root system

simple_reflections()

EXAMPLES:

```
sage: WeylGroup(['A',2,1]).classical().simple_reflections()
Finite family {1: [ 1  0  0]
                  [ 1 -1  1]
                  [ 0  0  1],
               2: [ 1  0  0]
                  [ 0  1  0]
                  [ 1  1 -1]}
```

```
>>> from sage.all import *
>>> WeylGroup(['A',Integer(2),Integer(1)]).classical().simple_reflections()
Finite family {1: [ 1  0  0]
                  [ 1 -1  1]
                  [ 0  0  1],
               2: [ 1  0  0]
                  [ 0  1  0]
                  [ 1  1 -1]}
```

Note: won't be needed, once the lattice will be a parabolic sub root system

weyl_group(*prefix='hereditary'*)

Return the Weyl group associated to the parabolic subgroup.

EXAMPLES:

```
sage: WeylGroup(['A',4,1]).classical().weyl_group()
Weyl Group of type ['A', 4, 1] (as a matrix group acting on the root space)
sage: WeylGroup(['C',4,1]).classical().weyl_group()
Weyl Group of type ['C', 4, 1] (as a matrix group acting on the root space)
sage: WeylGroup(['E',8,1]).classical().weyl_group()
Weyl Group of type ['E', 8, 1] (as a matrix group acting on the root space)
```

```
>>> from sage.all import *
>>> WeylGroup(['A',Integer(4),Integer(1)]).classical().weyl_group()
Weyl Group of type ['A', 4, 1] (as a matrix group acting on the root space)
>>> WeylGroup(['C',Integer(4),Integer(1)]).classical().weyl_group()
Weyl Group of type ['C', 4, 1] (as a matrix group acting on the root space)
>>> WeylGroup(['E',Integer(8),Integer(1)]).classical().weyl_group()
Weyl Group of type ['E', 8, 1] (as a matrix group acting on the root space)
```

sage.combinat.root_system.weyl_group.**WeylGroup**(*x*, *prefix=None*, *implementation='matrix'*)

Return the Weyl group of the root system defined by the Cartan type (or matrix) `ct`.

INPUT:

- `x` – a root system or a Cartan type (or matrix)

OPTIONAL:

- `prefix` – changes the representation of elements from matrices to products of simple reflections
- `implementation` – one of the following:
  - `'matrix'` – as matrices acting on a root system
  - `'permutation'` – as a permutation group acting on the roots

EXAMPLES:

The following constructions yield the same result, namely a weight lattice and its corresponding Weyl group:

```
sage: G = WeylGroup(['F',4])
sage: L = G.domain()
```

```
>>> from sage.all import *
>>> G = WeylGroup(['F',Integer(4)])
>>> L = G.domain()
```

or alternatively and equivalently:

```
sage: L = RootSystem(['F',4]).ambient_space()
sage: G = L.weyl_group()
sage: W = WeylGroup(L)
```

```
>>> from sage.all import *
>>> L = RootSystem(['F',Integer(4)]).ambient_space()
>>> G = L.weyl_group()
>>> W = WeylGroup(L)
```

Either produces a weight lattice, with access to its roots and weights.

```
sage: G = WeylGroup(['F',4])
sage: G.order()
1152
sage: [s1,s2,s3,s4] = G.simple_reflections()
sage: w = s1*s2*s3*s4; w
[ 1/2  1/2  1/2  1/2]
[-1/2  1/2  1/2 -1/2]
[ 1/2  1/2 -1/2 -1/2]
[ 1/2 -1/2  1/2 -1/2]
sage: type(w) == G.element_class
True
sage: w.order()
12
sage: w.length() # length function on Weyl group
4
```

```
>>> from sage.all import *
>>> G = WeylGroup(['F',Integer(4)])
>>> G.order()
1152
>>> [s1,s2,s3,s4] = G.simple_reflections()
>>> w = s1*s2*s3*s4; w
[ 1/2  1/2  1/2  1/2]
[-1/2  1/2  1/2 -1/2]
[ 1/2  1/2 -1/2 -1/2]
[ 1/2 -1/2  1/2 -1/2]
>>> type(w) == G.element_class
True
>>> w.order()
12
>>> w.length() # length function on Weyl group
4
```

The default representation of Weyl group elements is as matrices. If you prefer, you may specify a prefix, in which case the elements are represented as products of simple reflections.

```
sage: W=WeylGroup("C3",prefix='s')
sage: [s1,s2,s3]=W.simple_reflections() # lets Sage parse its own output
sage: s2*s1*s2*s3
s1*s2*s3*s1
sage: s2*s1*s2*s3 == s1*s2*s3*s1
True
sage: (s2*s3)^2==(s3*s2)^2
True
sage: (s1*s2*s3*s1).matrix()
[ 0  0 -1]
[ 0  1  0]
[ 1  0  0]
```

```
>>> from sage.all import *
>>> W=WeylGroup("C3",prefix='s')
```

(continues on next page)

(continued from previous page)

```
>>> [s1,s2,s3]=W.simple_reflections() # lets Sage parse its own output
>>> s2*s1*s2*s3
s1*s2*s3*s1
>>> s2*s1*s2*s3 == s1*s2*s3*s1
True
>>> (s2*s3)**Integer(2)==(s3*s2)**Integer(2)
True
>>> (s1*s2*s3*s1).matrix()
[ 0  0 -1]
[ 0  1  0]
[ 1  0  0]
```

```
sage: L = G.domain()
sage: fw = L.fundamental_weights(); fw
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4: (1,␣
↪0, 0, 0)}
sage: rho = sum(fw); rho
(11/2, 5/2, 3/2, 1/2)
sage: w.action(rho) # action of G on weight lattice
(5, -1, 3, 2)
```

```
>>> from sage.all import *
>>> L = G.domain()
>>> fw = L.fundamental_weights(); fw
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4: (1,␣
↪0, 0, 0)}
>>> rho = sum(fw); rho
(11/2, 5/2, 3/2, 1/2)
>>> w.action(rho) # action of G on weight lattice
(5, -1, 3, 2)
```

We can also do the same for arbitrary Cartan matrices:

```
sage: cm = CartanMatrix([[2,-5,0],[-2,2,-1],[0,-1,2]])
sage: W = WeylGroup(cm)
sage: W.gens()
(
[-1  5  0]  [ 1  0  0]  [ 1  0  0]
[ 0  1  0]  [ 2 -1  1]  [ 0  1  0]
[ 0  0  1], [ 0  0  1], [ 0  1 -1]
)
sage: s0,s1,s2 = W.gens()
sage: s1*s2*s1
[ 1  0  0]
[ 2  0 -1]
[ 2 -1  0]
sage: s2*s1*s2
[ 1  0  0]
[ 2  0 -1]
[ 2 -1  0]
sage: s0*s1*s0*s2*s0
[ 9  0 -5]
```

(continues on next page)

(continued from previous page)

```
[ 2  0 -1]
[ 0  1 -1]
```

```
>>> from sage.all import *
>>> cm = CartanMatrix([[Integer(2),-Integer(5),Integer(0)],[-Integer(2),
↪Integer(2),-Integer(1)],[Integer(0),-Integer(1),Integer(2)]])
>>> W = WeylGroup(cm)
>>> W.gens()
(
[-1  5  0]  [ 1  0  0]  [ 1  0  0]
[ 0  1  0]  [ 2 -1  1]  [ 0  1  0]
[ 0  0  1], [ 0  0  1], [ 0  1 -1]
)
>>> s0,s1,s2 = W.gens()
>>> s1*s2*s1
[ 1  0  0]
[ 2  0 -1]
[ 2 -1  0]
>>> s2*s1*s2
[ 1  0  0]
[ 2  0 -1]
[ 2 -1  0]
>>> s0*s1*s0*s2*s0
[ 9  0 -5]
[ 2  0 -1]
[ 0  1 -1]
```

Same Cartan matrix, but with a prefix to display using simple reflections:

```
sage: W = WeylGroup(cm, prefix='s')
sage: s0,s1,s2 = W.gens()
sage: s0*s2*s1
s2*s0*s1
sage: (s1*s2)^3
1
sage: (s0*s1)^5
s0*s1*s0*s1*s0*s1*s0*s1*s0*s1
sage: s0*s1*s2*s1*s2
s2*s0*s1
sage: s0*s1*s2*s0*s2
s0*s1*s0
```

```
>>> from sage.all import *
>>> W = WeylGroup(cm, prefix='s')
>>> s0,s1,s2 = W.gens()
>>> s0*s2*s1
s2*s0*s1
>>> (s1*s2)**Integer(3)
1
>>> (s0*s1)**Integer(5)
s0*s1*s0*s1*s0*s1*s0*s1*s0*s1
>>> s0*s1*s2*s1*s2
```

(continues on next page)

(continued from previous page)

```
s2*s0*s1
>>> s0*s1*s2*s0*s2
s0*s1*s0
```

class sage.combinat.root_system.weyl_group.**WeylGroupElement**(*parent*, *g*, *check=False*)

Bases: MatrixGroupElement_gap

Class for a Weyl Group elements

action(*v*)

Return the action of self on the vector v.

EXAMPLES:

```
sage: W = WeylGroup(['A',2])
sage: s = W.simple_reflections()
sage: v = W.domain()([1,0,0])
sage: s[1].action(v)
(0, 1, 0)

sage: W = WeylGroup(RootSystem(['A',2]).root_lattice())
sage: s = W.simple_reflections()
sage: alpha = W.domain().simple_roots()
sage: s[1].action(alpha[1])
-alpha[1]

sage: W=WeylGroup(['A',2,1])
sage: alpha = W.domain().simple_roots()
sage: s = W.simple_reflections()
sage: s[1].action(alpha[1])
-alpha[1]
sage: s[1].action(alpha[0])
alpha[0] + alpha[1]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(2)])
>>> s = W.simple_reflections()
>>> v = W.domain()([Integer(1),Integer(0),Integer(0)])
>>> s[Integer(1)].action(v)
(0, 1, 0)

>>> W = WeylGroup(RootSystem(['A',Integer(2)]).root_lattice())
>>> s = W.simple_reflections()
>>> alpha = W.domain().simple_roots()
>>> s[Integer(1)].action(alpha[Integer(1)])
-alpha[1]

>>> W=WeylGroup(['A',Integer(2),Integer(1)])
>>> alpha = W.domain().simple_roots()
>>> s = W.simple_reflections()
>>> s[Integer(1)].action(alpha[Integer(1)])
-alpha[1]
>>> s[Integer(1)].action(alpha[Integer(0)])
```

(continues on next page)

(continued from previous page)

```
alpha[0] + alpha[1]
```

apply_simple_reflection(*i*, *side='right'*)

domain()

Return the ambient lattice associated with `self`.

EXAMPLES:

```
sage: W = WeylGroup(['A',2])
sage: s1 = W.simple_reflection(1)
sage: s1.domain()
Ambient space of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(2)])
>>> s1 = W.simple_reflection(Integer(1))
>>> s1.domain()
Ambient space of the Root system of type ['A', 2]
```

has_descent(*i*, *positive=False*, *side='right'*)

Test if `self` has a descent at position `i`.

An element w has a descent in position i if w is on the strict negative side of the i-th simple reflection hyperplane.

If `positive` is `True`, tests if it is on the strict positive side instead.

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: s = W.simple_reflections()
sage: [W.one().has_descent(i) for i in W.domain().index_set()]
[False, False, False]
sage: [s[1].has_descent(i) for i in W.domain().index_set()]
[True, False, False]
sage: [s[2].has_descent(i) for i in W.domain().index_set()]
[False, True, False]
sage: [s[3].has_descent(i) for i in W.domain().index_set()]
[False, False, True]
sage: [s[3].has_descent(i, True) for i in W.domain().index_set()]
[True, True, False]
sage: W = WeylGroup(['A',3,1])
sage: s = W.simple_reflections()
sage: [W.one().has_descent(i) for i in W.domain().index_set()]
[False, False, False, False]
sage: [s[0].has_descent(i) for i in W.domain().index_set()]
[True, False, False, False]
sage: w = s[0] * s[1]
sage: [w.has_descent(i) for i in W.domain().index_set()]
[False, True, False, False]
sage: [w.has_descent(i, side = "left") for i in W.domain().index_set()]
[True, False, False, False]
sage: w = s[0] * s[2]
```

(continues on next page)

(continued from previous page)

```
sage: [w.has_descent(i) for i in W.domain().index_set()]
[True, False, True, False]
sage: [w.has_descent(i, side = "left") for i in W.domain().index_set()]
[True, False, True, False]

sage: W = WeylGroup(['A',3])
sage: W.one().has_descent(0)
True
sage: W.w0.has_descent(0)
False
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3)])
>>> s = W.simple_reflections()
>>> [W.one().has_descent(i) for i in W.domain().index_set()]
[False, False, False]
>>> [s[Integer(1)].has_descent(i) for i in W.domain().index_set()]
[True, False, False]
>>> [s[Integer(2)].has_descent(i) for i in W.domain().index_set()]
[False, True, False]
>>> [s[Integer(3)].has_descent(i) for i in W.domain().index_set()]
[False, False, True]
>>> [s[Integer(3)].has_descent(i, True) for i in W.domain().index_set()]
[True, True, False]
>>> W = WeylGroup(['A',Integer(3),Integer(1)])
>>> s = W.simple_reflections()
>>> [W.one().has_descent(i) for i in W.domain().index_set()]
[False, False, False, False]
>>> [s[Integer(0)].has_descent(i) for i in W.domain().index_set()]
[True, False, False, False]
>>> w = s[Integer(0)] * s[Integer(1)]
>>> [w.has_descent(i) for i in W.domain().index_set()]
[False, True, False, False]
>>> [w.has_descent(i, side = "left") for i in W.domain().index_set()]
[True, False, False, False]
>>> w = s[Integer(0)] * s[Integer(2)]
>>> [w.has_descent(i) for i in W.domain().index_set()]
[True, False, True, False]
>>> [w.has_descent(i, side = "left") for i in W.domain().index_set()]
[True, False, True, False]

>>> W = WeylGroup(['A',Integer(3)])
>>> W.one().has_descent(Integer(0))
True
>>> W.w0.has_descent(Integer(0))
False
```

has_left_descent(*i*)

Test if `self` has a left descent at position `i`.

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: s = W.simple_reflections()
sage: [W.one().has_left_descent(i) for i in W.domain().index_set()]
[False, False, False]
sage: [s[1].has_left_descent(i) for i in W.domain().index_set()]
[True, False, False]
sage: [s[2].has_left_descent(i) for i in W.domain().index_set()]
[False, True, False]
sage: [s[3].has_left_descent(i) for i in W.domain().index_set()]
[False, False, True]
sage: [(s[3]*s[2]).has_left_descent(i) for i in W.domain().index_set()]
[False, False, True]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3)])
>>> s = W.simple_reflections()
>>> [W.one().has_left_descent(i) for i in W.domain().index_set()]
[False, False, False]
>>> [s[Integer(1)].has_left_descent(i) for i in W.domain().index_set()]
[True, False, False]
>>> [s[Integer(2)].has_left_descent(i) for i in W.domain().index_set()]
[False, True, False]
>>> [s[Integer(3)].has_left_descent(i) for i in W.domain().index_set()]
[False, False, True]
>>> [(s[Integer(3)]*s[Integer(2)]).has_left_descent(i) for i in W.domain().
↪index_set()]
[False, False, True]
```

has_right_descent(*i*)

Test if `self` has a right descent at position `i`.

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: s = W.simple_reflections()
sage: [W.one().has_right_descent(i) for i in W.domain().index_set()]
[False, False, False]
sage: [s[1].has_right_descent(i) for i in W.domain().index_set()]
[True, False, False]
sage: [s[2].has_right_descent(i) for i in W.domain().index_set()]
[False, True, False]
sage: [s[3].has_right_descent(i) for i in W.domain().index_set()]
[False, False, True]
sage: [(s[3]*s[2]).has_right_descent(i) for i in W.domain().index_set()]
[False, True, False]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3)])
>>> s = W.simple_reflections()
>>> [W.one().has_right_descent(i) for i in W.domain().index_set()]
[False, False, False]
>>> [s[Integer(1)].has_right_descent(i) for i in W.domain().index_set()]
[True, False, False]
```

(continues on next page)

(continued from previous page)

```
>>> [s[Integer(2)].has_right_descent(i) for i in W.domain().index_set()]
[False, True, False]
>>> [s[Integer(3)].has_right_descent(i) for i in W.domain().index_set()]
[False, False, True]
>>> [(s[Integer(3)]*s[Integer(2)]).has_right_descent(i) for i in W.domain().
↪index_set()]
[False, True, False]
```

to_matrix()

Return `self` as a matrix.

EXAMPLES:

```
sage: G = WeylGroup(['A',2])
sage: s1 = G.simple_reflection(1)
sage: s1.to_matrix() == s1.matrix()
True
```

```
>>> from sage.all import *
>>> G = WeylGroup(['A',Integer(2)])
>>> s1 = G.simple_reflection(Integer(1))
>>> s1.to_matrix() == s1.matrix()
True
```

to_permutation()

A first approximation of to_permutation …

This assumes types A,B,C,D on the ambient lattice

This further assume that the basis is indexed by 0,1,… and returns a permutation of (5,4,2,3,1) (beuargl), as a tuple

to_permutation_string()

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: s = W.simple_reflections()
sage: (s[1]*s[2]*s[3]).to_permutation_string()
'2341'
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(3)])
>>> s = W.simple_reflections()
>>> (s[Integer(1)]*s[Integer(2)]*s[Integer(3)]).to_permutation_string()
'2341'
```

class sage.combinat.root_system.weyl_group.**WeylGroup_gens**(*domain*, *prefix*)

Bases: `UniqueRepresentation`, `FinitelyGeneratedMatrixGroup_gap`

EXAMPLES:

```
sage: G = WeylGroup(['B',3])
sage: TestSuite(G).run()
sage: cm = CartanMatrix([[2,-5,0],[-2,2,-1],[0,-1,2]])
```

(continues on next page)

(continued from previous page)

```
sage: W = WeylGroup(cm)
sage: TestSuite(W).run() # long time
```

```
>>> from sage.all import *
>>> G = WeylGroup(['B',Integer(3)])
>>> TestSuite(G).run()
>>> cm = CartanMatrix([[Integer(2),-Integer(5),Integer(0)],[-Integer(2),
↪Integer(2),-Integer(1)],[Integer(0),-Integer(1),Integer(2)]])
>>> W = WeylGroup(cm)
>>> TestSuite(W).run() # long time
```

Element

alias of `WeylGroupElement`

cartan_type()

Return the `CartanType` associated to `self`.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.cartan_type()
['F', 4]
```

```
>>> from sage.all import *
>>> G = WeylGroup(['F',Integer(4)])
>>> G.cartan_type()
['F', 4]
```

character_table()

Return the character table as a matrix.

Each row is an irreducible character. For larger tables you may preface this with a command such as gap.eval("SizeScreen([120,40])") in order to widen the screen.

EXAMPLES:

```
sage: WeylGroup(['A',3]).character_table()
CT1

     2  3  2  2  .  3
     3  1  .  .  1  .

       1a 4a 2a 3a 2b

X.1     1 -1 -1  1  1
X.2     3  1 -1  . -1
X.3     2  .  . -1  2
X.4     3 -1  1  . -1
X.5     1  1  1  1  1
```

```
>>> from sage.all import *
>>> WeylGroup(['A',Integer(3)]).character_table()
CT1
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
     2  3  2  2  .  3
     3  1  .  .  1  .
<BLANKLINE>
       1a 4a 2a 3a 2b
<BLANKLINE>
X.1      1 -1 -1  1  1
X.2      3  1 -1  . -1
X.3      2  .  . -1  2
X.4      3 -1  1  . -1
X.5      1  1  1  1  1
```

classical()

If `self` is a Weyl group from an affine Cartan Type, this give the classical parabolic subgroup of `self`.

Caveat: we assume that 0 is a special node of the Dynkin diagram

Todo

extract parabolic subgroup method

EXAMPLES:

```
sage: G = WeylGroup(['A',3,1])
sage: G.classical()
Parabolic Subgroup of the Weyl Group of type ['A', 3, 1]
 (as a matrix group acting on the root space)
sage: WeylGroup(['A',3]).classical()
Traceback (most recent call last):
...
ValueError: classical subgroup only defined for affine types
```

```
>>> from sage.all import *
>>> G = WeylGroup(['A',Integer(3),Integer(1)])
>>> G.classical()
Parabolic Subgroup of the Weyl Group of type ['A', 3, 1]
 (as a matrix group acting on the root space)
>>> WeylGroup(['A',Integer(3)]).classical()
Traceback (most recent call last):
...
ValueError: classical subgroup only defined for affine types
```

domain()

Return the domain of the element of `self`, that is the root lattice realization on which they act.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.domain()
Ambient space of the Root system of type ['F', 4]
sage: G = WeylGroup(['A',3,1])
```

(continues on next page)

(continued from previous page)

```
sage: G.domain()
Root space over the Rational Field of the Root system of type ['A', 3, 1]
```

```
>>> from sage.all import *
>>> G = WeylGroup(['F',Integer(4)])
>>> G.domain()
Ambient space of the Root system of type ['F', 4]
>>> G = WeylGroup(['A',Integer(3),Integer(1)])
>>> G.domain()
Root space over the Rational Field of the Root system of type ['A', 3, 1]
```

from_morphism(*f*)

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.index_set()
(1, 2, 3, 4)
sage: G = WeylGroup(['A',3,1])
sage: G.index_set()
(0, 1, 2, 3)
```

```
>>> from sage.all import *
>>> G = WeylGroup(['F',Integer(4)])
>>> G.index_set()
(1, 2, 3, 4)
>>> G = WeylGroup(['A',Integer(3),Integer(1)])
>>> G.index_set()
(0, 1, 2, 3)
```

long_element_hardcoded()

Return the long Weyl group element (hardcoded data).

Do we really want to keep it? There is a generic implementation which works in all cases. The hardcoded should have a better complexity (for large classical types), but there is a cache, so does this really matter?

EXAMPLES:

```
sage: types = [ ['A',5],['B',3],['C',3],['D',4],['G',2],['F',4],['E',6] ]
sage: [WeylGroup(t).long_element().length() for t in types]
[15, 9, 9, 12, 6, 24, 36]
sage: all(WeylGroup(t).long_element() == WeylGroup(t).long_element_
↪hardcoded() for t in types)  # long time (17s on sage.math, 2011)
True
```

```
>>> from sage.all import *
>>> types = [ ['A',Integer(5)],['B',Integer(3)],['C',Integer(3)],['D',
↪Integer(4)],['G',Integer(2)],['F',Integer(4)],['E',Integer(6)] ]
>>> [WeylGroup(t).long_element().length() for t in types]
[15, 9, 9, 12, 6, 24, 36]
```

(continues on next page)

(continued from previous page)

```
>>> all(WeylGroup(t).long_element() == WeylGroup(t).long_element_hardcoded()␣
↪for t in types)   # long time (17s on sage.math, 2011)
True
```

morphism_matrix(*f*)

one()

Return the unit element of the Weyl group.

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: e = W.one(); e
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: type(e) == W.element_class
True
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3)])
>>> e = W.one(); e
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> type(e) == W.element_class
True
```

reflections()

Return the reflections of `self`.

The reflections of a Coxeter group W are the conjugates of the simple reflections. They are in bijection with the positive roots, for given a positive root, we may have the reflection in the hyperplane orthogonal to it. This method returns a family indexed by the positive roots taking values in the reflections. This requires `self` to be a finite Weyl group.

> **Note**
>
> Prior to Issue #20027, the reflections were the keys of the family and the values were the positive roots.

EXAMPLES:

```
sage: W = WeylGroup("B2", prefix='s')
sage: refdict = W.reflections(); refdict
Finite family {(1, -1): s1, (0, 1): s2, (1, 1): s2*s1*s2, (1, 0): s1*s2*s1}
sage: [r+refdict[r].action(r) for r in refdict.keys()]
[(0, 0), (0, 0), (0, 0), (0, 0)]

sage: W = WeylGroup(['A',2,1], prefix='s')
sage: W.reflections()
```

(continues on next page)

(continued from previous page)

```
Lazy family (real root to reflection(i))_{i in
             Positive real roots of type ['A', 2, 1]}
```

```
>>> from sage.all import *
>>> W = WeylGroup("B2", prefix='s')
>>> refdict = W.reflections(); refdict
Finite family {(1, -1): s1, (0, 1): s2, (1, 1): s2*s1*s2, (1, 0): s1*s2*s1}
>>> [r+refdict[r].action(r) for r in refdict.keys()]
[(0, 0), (0, 0), (0, 0), (0, 0)]

>>> W = WeylGroup(['A',Integer(2),Integer(1)], prefix='s')
>>> W.reflections()
Lazy family (real root to reflection(i))_{i in
             Positive real roots of type ['A', 2, 1]}
```

simple_reflection(*i*)

Return the i-th simple reflection.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.simple_reflection(1)
[1 0 0 0]
[0 0 1 0]
[0 1 0 0]
[0 0 0 1]
sage: W=WeylGroup(['A',2,1])
sage: W.simple_reflection(1)
[ 1  0  0]
[ 1 -1  1]
[ 0  0  1]
```

```
>>> from sage.all import *
>>> G = WeylGroup(['F',Integer(4)])
>>> G.simple_reflection(Integer(1))
[1 0 0 0]
[0 0 1 0]
[0 1 0 0]
[0 0 0 1]
>>> W=WeylGroup(['A',Integer(2),Integer(1)])
>>> W.simple_reflection(Integer(1))
[ 1  0  0]
[ 1 -1  1]
[ 0  0  1]
```

simple_reflections()

Return the simple reflections of `self`, as a family.

EXAMPLES:

There are the simple reflections for the symmetric group:

```
sage: W=WeylGroup(['A',2])
sage: s = W.simple_reflections(); s
Finite family {1: [0 1 0]
[1 0 0]
[0 0 1], 2: [1 0 0]
[0 0 1]
[0 1 0]}
```

```
>>> from sage.all import *
>>> W=WeylGroup(['A',Integer(2)])
>>> s = W.simple_reflections(); s
Finite family {1: [0 1 0]
[1 0 0]
[0 0 1], 2: [1 0 0]
[0 0 1]
[0 1 0]}
```

As a special feature, for finite irreducible root systems, s[0] gives the reflection along the highest root:

```
sage: s[0]
[0 0 1]
[0 1 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> s[Integer(0)]
[0 0 1]
[0 1 0]
[1 0 0]
```

We now look at some further examples:

```
sage: W=WeylGroup(['A',2,1])
sage: W.simple_reflections()
Finite family {0: [-1  1  1]
[ 0  1  0]
[ 0  0  1], 1: [ 1  0  0]
[ 1 -1  1]
[ 0  0  1], 2: [ 1  0  0]
[ 0  1  0]
[ 1  1 -1]}
sage: W = WeylGroup(['F',4])
sage: [s1,s2,s3,s4] = W.simple_reflections()
sage: w = s1*s2*s3*s4; w
[ 1/2  1/2  1/2  1/2]
[-1/2  1/2  1/2 -1/2]
[ 1/2  1/2 -1/2 -1/2]
[ 1/2 -1/2  1/2 -1/2]
sage: s4^2 == W.one()
True
sage: type(w) == W.element_class
True
```

```
>>> from sage.all import *
>>> W=WeylGroup(['A',Integer(2),Integer(1)])
>>> W.simple_reflections()
Finite family {0: [-1  1  1]
[ 0  1  0]
[ 0  0  1], 1: [ 1  0  0]
[ 1 -1  1]
[ 0  0  1], 2: [ 1  0  0]
[ 0  1  0]
[ 1  1 -1]}
>>> W = WeylGroup(['F',Integer(4)])
>>> [s1,s2,s3,s4] = W.simple_reflections()
>>> w = s1*s2*s3*s4; w
[ 1/2  1/2  1/2  1/2]
[-1/2  1/2  1/2 -1/2]
[ 1/2  1/2 -1/2 -1/2]
[ 1/2 -1/2  1/2 -1/2]
>>> s4**Integer(2) == W.one()
True
>>> type(w) == W.element_class
True
```

unit()

Return the unit element of the Weyl group.

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: e = W.one(); e
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: type(e) == W.element_class
True
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3)])
>>> e = W.one(); e
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> type(e) == W.element_class
True
```

class sage.combinat.root_system.weyl_group.**WeylGroup_permutation**(*cartan_type*, *prefix*)

Bases: UniqueRepresentation, PermutationGroup_generic

A Weyl group given as a permutation group.

class Element

Bases: RealReflectionGroupElement

cartan_type()

Return the Cartan type of self.

EXAMPLES:

```
sage: W = WeylGroup(['A',4], implementation='permutation')
sage: W.cartan_type()
['A', 4]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)], implementation='permutation')
>>> W.cartan_type()
['A', 4]
```

distinguished_reflections()

Return the reflections of self.

EXAMPLES:

```
sage: W = WeylGroup(['B',2], implementation='permutation')
sage: W.distinguished_reflections()
Finite family {1: (1,5)(2,4)(6,8), 2: (1,3)(2,6)(5,7),
               3: (2,8)(3,7)(4,6), 4: (1,7)(3,5)(4,8)}
```

```
>>> from sage.all import *
>>> W = WeylGroup(['B',Integer(2)], implementation='permutation')
>>> W.distinguished_reflections()
Finite family {1: (1,5)(2,4)(6,8), 2: (1,3)(2,6)(5,7),
               3: (2,8)(3,7)(4,6), 4: (1,7)(3,5)(4,8)}
```

independent_roots()

Return the simple roots of self.

EXAMPLES:

```
sage: W = WeylGroup(['A',4], implementation='permutation')
sage: W.simple_roots()
Finite family {1: (1, 0, 0, 0), 2: (0, 1, 0, 0),
               3: (0, 0, 1, 0), 4: (0, 0, 0, 1)}
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)], implementation='permutation')
>>> W.simple_roots()
Finite family {1: (1, 0, 0, 0), 2: (0, 1, 0, 0),
               3: (0, 0, 1, 0), 4: (0, 0, 0, 1)}
```

index_set()

Return the index set of self.

EXAMPLES:

```
sage: W = WeylGroup(['A',4], implementation='permutation')
sage: W.index_set()
(1, 2, 3, 4)
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)], implementation='permutation')
>>> W.index_set()
(1, 2, 3, 4)
```

iteration(*algorithm='breadth'*, *tracking_words=True*)

Return an iterator going through all elements in `self`.

INPUT:

- `algorithm` – (default: `'breadth'`) must be one of the following:
  - `'breadth'` – iterate over in a linear extension of the weak order
  - `'depth'` – iterate by a depth-first-search
- `tracking_words` – boolean (default: `True`); whether or not to keep track of the reduced words and store them in `_reduced_word`

Note

The fastest iteration is the depth first algorithm without tracking words. In particular, `'depth'` is ~1.5x faster.

EXAMPLES:

```
sage: W = WeylGroup(["B",2], implementation='permutation')

sage: for w in W.iteration("breadth",True):
....:     print("%s %s"%(w, w._reduced_word))
() []
(1,3)(2,6)(5,7) [1]
(1,5)(2,4)(6,8) [0]
(1,7,5,3)(2,4,6,8) [0, 1]
(1,3,5,7)(2,8,6,4) [1, 0]
(2,8)(3,7)(4,6) [1, 0, 1]
(1,7)(3,5)(4,8) [0, 1, 0]
(1,5)(2,6)(3,7)(4,8) [0, 1, 0, 1]

sage: for w in W.iteration("depth", False): w
()
(1,3)(2,6)(5,7)
(1,5)(2,4)(6,8)
(1,3,5,7)(2,8,6,4)
(1,7)(3,5)(4,8)
(1,7,5,3)(2,4,6,8)
(2,8)(3,7)(4,6)
(1,5)(2,6)(3,7)(4,8)
```

```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(2)], implementation='permutation')

>>> for w in W.iteration("breadth",True):
...     print("%s %s"%(w, w._reduced_word))
```

(continues on next page)

(continued from previous page)

```
() []
(1,3)(2,6)(5,7) [1]
(1,5)(2,4)(6,8) [0]
(1,7,5,3)(2,4,6,8) [0, 1]
(1,3,5,7)(2,8,6,4) [1, 0]
(2,8)(3,7)(4,6) [1, 0, 1]
(1,7)(3,5)(4,8) [0, 1, 0]
(1,5)(2,6)(3,7)(4,8) [0, 1, 0, 1]

>>> for w in W.iteration("depth", False): w
()
(1,3)(2,6)(5,7)
(1,5)(2,4)(6,8)
(1,3,5,7)(2,8,6,4)
(1,7)(3,5)(4,8)
(1,7,5,3)(2,4,6,8)
(2,8)(3,7)(4,6)
(1,5)(2,6)(3,7)(4,8)
```

number_of_reflections()

Return the number of reflections in `self`.

EXAMPLES:

```
sage: W = WeylGroup(['D',4], implementation='permutation')
sage: W.number_of_reflections()
12
```

```
>>> from sage.all import *
>>> W = WeylGroup(['D',Integer(4)], implementation='permutation')
>>> W.number_of_reflections()
12
```

positive_roots()

Return the positive roots of `self`.

EXAMPLES:

```
sage: W = WeylGroup(['C',3], implementation='permutation')
sage: W.positive_roots()
((1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
 (1, 1, 0),
 (0, 1, 1),
 (0, 2, 1),
 (1, 1, 1),
 (2, 2, 1),
 (1, 2, 1))
```

```
>>> from sage.all import *
>>> W = WeylGroup(['C',Integer(3)], implementation='permutation')
```

(continues on next page)

(continued from previous page)

```
>>> W.positive_roots()
((1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
 (1, 1, 0),
 (0, 1, 1),
 (0, 2, 1),
 (1, 1, 1),
 (2, 2, 1),
 (1, 2, 1))
```

rank()

Return the rank of `self`.

EXAMPLES:

```
sage: W = WeylGroup(['A',4], implementation='permutation')
sage: W.rank()
4
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)], implementation='permutation')
>>> W.rank()
4
```

reflection_index_set()

Return the index set of reflections of `self`.

EXAMPLES:

```
sage: W = WeylGroup(['A',3], implementation='permutation')
sage: W.reflection_index_set()
(1, 2, 3, 4, 5, 6)
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3)], implementation='permutation')
>>> W.reflection_index_set()
(1, 2, 3, 4, 5, 6)
```

reflections()

Return the reflections of `self`.

EXAMPLES:

```
sage: W = WeylGroup(['B',2], implementation='permutation')
sage: W.distinguished_reflections()
Finite family {1: (1,5)(2,4)(6,8), 2: (1,3)(2,6)(5,7),
               3: (2,8)(3,7)(4,6), 4: (1,7)(3,5)(4,8)}
```

```
>>> from sage.all import *
>>> W = WeylGroup(['B',Integer(2)], implementation='permutation')
>>> W.distinguished_reflections()
```

(continues on next page)

(continued from previous page)

```
Finite family {1: (1,5)(2,4)(6,8), 2: (1,3)(2,6)(5,7),
               3: (2,8)(3,7)(4,6), 4: (1,7)(3,5)(4,8)}
```

roots()

Return the roots of `self`.

EXAMPLES:

```
sage: W = WeylGroup(['G',2], implementation='permutation')
sage: W.roots()
((1, 0),
 (0, 1),
 (1, 1),
 (3, 1),
 (2, 1),
 (3, 2),
 (-1, 0),
 (0, -1),
 (-1, -1),
 (-3, -1),
 (-2, -1),
 (-3, -2))
```

```
>>> from sage.all import *
>>> W = WeylGroup(['G',Integer(2)], implementation='permutation')
>>> W.roots()
((1, 0),
 (0, 1),
 (1, 1),
 (3, 1),
 (2, 1),
 (3, 2),
 (-1, 0),
 (0, -1),
 (-1, -1),
 (-3, -1),
 (-2, -1),
 (-3, -2))
```

simple_reflection(i)

Return the `i`-th simple reflection of `self`.

EXAMPLES:

```
sage: W = WeylGroup(['A',4], implementation='permutation')
sage: W.simple_reflection(1)
(1,11)(2,5)(6,8)(9,10)(12,15)(16,18)(19,20)
sage: W.simple_reflections()
Finite family {1: (1,11)(2,5)(6,8)(9,10)(12,15)(16,18)(19,20),
               2: (1,5)(2,12)(3,6)(7,9)(11,15)(13,16)(17,19),
               3: (2,6)(3,13)(4,7)(5,8)(12,16)(14,17)(15,18),
               4: (3,7)(4,14)(6,9)(8,10)(13,17)(16,19)(18,20)}
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)], implementation='permutation')
>>> W.simple_reflection(Integer(1))
(1,11)(2,5)(6,8)(9,10)(12,15)(16,18)(19,20)
>>> W.simple_reflections()
Finite family {1: (1,11)(2,5)(6,8)(9,10)(12,15)(16,18)(19,20),
               2: (1,5)(2,12)(3,6)(7,9)(11,15)(13,16)(17,19),
               3: (2,6)(3,13)(4,7)(5,8)(12,16)(14,17)(15,18),
               4: (3,7)(4,14)(6,9)(8,10)(13,17)(16,19)(18,20)}
```

simple_root_index(i)

Return the index of the simple root α_i.

This is the position of α_i in the list of simple roots.

EXAMPLES:

```
sage: W = WeylGroup(['A',3], implementation='permutation')
sage: [W.simple_root_index(i) for i in W.index_set()]
[0, 1, 2]
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3)], implementation='permutation')
>>> [W.simple_root_index(i) for i in W.index_set()]
[0, 1, 2]
```

simple_roots()

Return the simple roots of `self`.

EXAMPLES:

```
sage: W = WeylGroup(['A',4], implementation='permutation')
sage: W.simple_roots()
Finite family {1: (1, 0, 0, 0), 2: (0, 1, 0, 0),
               3: (0, 0, 1, 0), 4: (0, 0, 0, 1)}
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)], implementation='permutation')
>>> W.simple_roots()
Finite family {1: (1, 0, 0, 0), 2: (0, 1, 0, 0),
               3: (0, 0, 1, 0), 4: (0, 0, 0, 1)}
```

5.1.274 Rooted (Unordered) Trees

AUTHORS:

- Florent Hivert (2011): initial version

class sage.combinat.rooted_tree.**LabelledRootedTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: *AbstractLabelledClonableTree*, *RootedTree*

Labelled rooted trees.

A labelled rooted tree is a rooted tree with a label attached at each node.

More formally: The *labelled rooted trees* are an inductive datatype defined as follows: A labelled rooted tree is a multiset of labelled rooted trees, endowed with a label (which can be any object, including `None`). The trees that belong to this multiset are said to be the *children* of the tree. (Notice that the labels of these children may and may not be of the same type as the label of the tree). A labelled rooted tree which has no children (so the only information it carries is its label) is said to be a *leaf*.

Every labelled rooted tree gives rise to an unlabelled rooted tree (`RootedTree`) by forgetting the labels. (This is implemented as a conversion.)

INPUT:

- `children` – list or tuple or more generally any iterable of trees or objects convertible to trees
- `label` – any hashable Sage object (default: `None`)

> **Note**
>
> It is required that all labels are comparable.

EXAMPLES:

```
sage: x = LabelledRootedTree([], label = 3); x
3[]
sage: LabelledRootedTree([x, x, x], label = 2)
2[3[], 3[], 3[]]
sage: LabelledRootedTree((x, x, x), label = 2)
2[3[], 3[], 3[]]
sage: LabelledRootedTree([[],[[], []]], label = 3)
3[None[], None[None[], None[]]]
```

```
>>> from sage.all import *
>>> x = LabelledRootedTree([], label = Integer(3)); x
3[]
>>> LabelledRootedTree([x, x, x], label = Integer(2))
2[3[], 3[], 3[]]
>>> LabelledRootedTree((x, x, x), label = Integer(2))
2[3[], 3[], 3[]]
>>> LabelledRootedTree([[],[[], []]], label = Integer(3))
3[None[], None[None[], None[]]]
```

Children are reordered using the value of the `sort_key()` method:

```
sage: y = LabelledRootedTree([], label = 5); y
5[]
sage: xyy2 = LabelledRootedTree((x, y, y), label = 2); xyy2
2[3[], 5[], 5[]]
sage: yxy2 = LabelledRootedTree((y, x, y), label = 2); yxy2
2[3[], 5[], 5[]]
sage: xyy2 == yxy2
True
```

```
>>> from sage.all import *
>>> y = LabelledRootedTree([], label = Integer(5)); y
5[]
```

(continues on next page)

(continued from previous page)

```
>>> xyy2 = LabelledRootedTree((x, y, y), label = Integer(2)); xyy2
2[3[], 5[], 5[]]
>>> yxy2 = LabelledRootedTree((y, x, y), label = Integer(2)); yxy2
2[3[], 5[], 5[]]
>>> xyy2 == yxy2
True
```

Converting labelled into unlabelled rooted trees by forgetting the labels, and back (the labels are initialized as `None`):

```
sage: yxy2crude = RootedTree(yxy2); yxy2crude
[[], [], []]
sage: LabelledRootedTree(yxy2crude)
None[None[], None[], None[]]
```

```
>>> from sage.all import *
>>> yxy2crude = RootedTree(yxy2); yxy2crude
[[], [], []]
>>> LabelledRootedTree(yxy2crude)
None[None[], None[], None[]]
```

sort_key()

Return a tuple of nonnegative integers encoding the labelled rooted tree `self`.

The first entry of the tuple is a pair consisting of the number of children of the root and the label of the root. Then the rest of the tuple is obtained as follows: List the tuples corresponding to all children (we are regarding the children themselves as trees). Order this list (not the tuples!) in lexicographically increasing order, and flatten it into a single tuple.

This tuple characterizes the labelled rooted tree uniquely, and can be used to sort the labelled rooted trees provided that the labels belong to a type which is totally ordered.

Note

The tree `self` must be normalized before calling this method (see `normalize()`). This does not matter unless you are inside the `clone()` context manager, because outside of it every rooted tree is already normalized.

Note

This method overrides *RootedTree.sort_key()* and returns a result different from what the latter would return, as it wants to encode the whole labelled tree including its labelling rather than just the unlabelled tree. Therefore, be careful with using this method on subclasses of `RootedOrderedTree`; under some circumstances they could inherit it from another superclass instead of from *RootedTree*, which would cause the method to forget the labelling. See the docstrings of *RootedTree.sort_key()* and *sage.combinat.ordered_tree.OrderedTree.sort_key()*.

EXAMPLES:

```
sage: LRT = LabelledRootedTrees(); LRT
Labelled rooted trees
```

(continues on next page)

(continued from previous page)

```
sage: x = LRT([], label = 3); x
3[]
sage: x.sort_key()
((0, 3),)
sage: y = LRT([x, x, x], label = 2); y
2[3[], 3[], 3[]]
sage: y.sort_key()
((3, 2), (0, 3), (0, 3), (0, 3))
sage: LRT.an_element().sort_key()
((3, 'alpha'), (0, 3), (1, 5), (0, None), (2, 42), (0, 3), (0, 3))
sage: lb = RootedTrees()([[],[[], []]]).canonical_labelling()
sage: lb.sort_key()
((2, 1), (0, 2), (2, 3), (0, 4), (0, 5))
```

```
>>> from sage.all import *
>>> LRT = LabelledRootedTrees(); LRT
Labelled rooted trees
>>> x = LRT([], label = Integer(3)); x
3[]
>>> x.sort_key()
((0, 3),)
>>> y = LRT([x, x, x], label = Integer(2)); y
2[3[], 3[], 3[]]
>>> y.sort_key()
((3, 2), (0, 3), (0, 3), (0, 3))
>>> LRT.an_element().sort_key()
((3, 'alpha'), (0, 3), (1, 5), (0, None), (2, 42), (0, 3), (0, 3))
>>> lb = RootedTrees()([[],[[], []]]).canonical_labelling()
>>> lb.sort_key()
((2, 1), (0, 2), (2, 3), (0, 4), (0, 5))
```

class sage.combinat.rooted_tree.**LabelledRootedTrees**

Bases: UniqueRepresentation, Parent

This is a parent stub to serve as a factory class for labelled rooted trees.

EXAMPLES:

```
sage: LRT = LabelledRootedTrees(); LRT
Labelled rooted trees
sage: x = LRT([], label = 3); x
3[]
sage: x.parent() is LRT
True
sage: y = LRT([x, x, x], label = 2); y
2[3[], 3[], 3[]]
sage: y.parent() is LRT
True
```

```
>>> from sage.all import *
>>> LRT = LabelledRootedTrees(); LRT
Labelled rooted trees
```

(continues on next page)

(continued from previous page)

```
>>> x = LRT([], label = Integer(3)); x
3[]
>>> x.parent() is LRT
True
>>> y = LRT([x, x, x], label = Integer(2)); y
2[3[], 3[], 3[]]
>>> y.parent() is LRT
True
```

Todo

Add the possibility to restrict the labels to a fixed set.

class sage.combinat.rooted_tree.**LabelledRootedTrees_all**(*category=None*)

Bases: *LabelledRootedTrees*

Class of all (unordered) labelled rooted trees.

See *LabelledRootedTree* for a definition.

Element

alias of *LabelledRootedTree*

labelled_trees()

Return the set of labelled trees associated to self.

EXAMPLES:

```
sage: LabelledRootedTrees().labelled_trees()
Labelled rooted trees
```

```
>>> from sage.all import *
>>> LabelledRootedTrees().labelled_trees()
Labelled rooted trees
```

unlabelled_trees()

Return the set of unlabelled trees associated to self.

EXAMPLES:

```
sage: LabelledRootedTrees().unlabelled_trees()
Rooted trees
```

```
>>> from sage.all import *
>>> LabelledRootedTrees().unlabelled_trees()
Rooted trees
```

class sage.combinat.rooted_tree.**RootedTree**(*parent=None*, *children=[]*, *check=True*)

Bases: *AbstractClonableTree*, NormalizedClonableList

The class for unordered rooted trees.

The *unordered rooted trees* are an inductive datatype defined as follows: An unordered rooted tree is a multiset of unordered rooted trees. The trees that belong to this multiset are said to be the *children* of the tree. The tree that has no children is called a *leaf*.

The *labelled rooted trees* (`LabelledRootedTree`) form a subclass of this class; they carry additional data.

One can create a tree from any list (or more generally iterable) of trees or objects convertible to a tree.

EXAMPLES:

```
sage: RootedTree([])
[]
sage: RootedTree([[], [[]]])
[[], [[]]]
sage: RootedTree([[[]], []])
[[], [[]]]
sage: O = OrderedTree([[[]], []]); O
[[[]], []]
sage: RootedTree(O)  # this is O with the ordering forgotten
[[], [[]]]
```

```
>>> from sage.all import *
>>> RootedTree([])
[]
>>> RootedTree([[], [[]]])
[[], [[]]]
>>> RootedTree([[[]], []])
[[], [[]]]
>>> O = OrderedTree([[[]], []]); O
[[[]], []]
>>> RootedTree(O)  # this is O with the ordering forgotten
[[], [[]]]
```

One can also enter any small rooted tree ("small" meaning that no vertex has more than 15 children) by using a simple numerical encoding of rooted trees, namely, the `from_hexacode()` function. (This function actually parametrizes ordered trees, and here we make it parametrize unordered trees by forgetting the ordering.)

```
sage: from sage.combinat.abstract_tree import from_hexacode
sage: RT = RootedTrees()
sage: from_hexacode('32001010', RT)
[[[]], [[]], [[], []]]
```

```
>>> from sage.all import *
>>> from sage.combinat.abstract_tree import from_hexacode
>>> RT = RootedTrees()
>>> from_hexacode('32001010', RT)
[[[]], [[]], [[], []]]
```

Note

Unlike an ordered tree, an (unordered) rooted tree is a multiset (rather than a list) of children. That is, two ordered trees which differ from each other by switching the order of children are equal to each other as (unordered) rooted trees. Internally, rooted trees are encoded as `sage.structure.list_clone.NormalizedClonableList` instances, and instead of storing their children as an actual multiset, they store their children as a list which is sorted according to their `sort_key()` value. This is as good as storing them as multisets, since the `sort_key()` values are sortable and distinguish different (unordered) trees. However, if you wish to define a subclass of `RootedTree` which implements rooted trees with extra structure (say, a class of edge-colored rooted trees, or a class of rooted trees with a cyclic order on the list of children), then the inherited `sort_key()` method will no longer distinguish different trees (and, as a consequence, equal trees will be regarded as distinct). Thus, you will have to override the method by one that does distinguish different trees.

graft_list(*other*)

Return the list of trees obtained by grafting `other` on `self`.

Here grafting means that one takes the disjoint union of `self` and `other`, chooses a node of `self`, and adds the root of `other` to the list of children of this node. The root of the resulting tree is the root of `self`. (This can be done for each node of `self`; this method returns the list of all results.)

This is useful for free pre-Lie algebras.

EXAMPLES:

```
sage: RT = RootedTree
sage: x = RT([])
sage: y = RT([x, x])
sage: x.graft_list(x)
[[[]]]
sage: l = y.graft_list(x); l
[[[], [], []], [[], [[]]], [[], [[]]]]
sage: [parent(i) for i in l]
[Rooted trees, Rooted trees, Rooted trees]
```

```
>>> from sage.all import *
>>> RT = RootedTree
>>> x = RT([])
>>> y = RT([x, x])
>>> x.graft_list(x)
[[[]]]
>>> l = y.graft_list(x); l
[[[], [], []], [[], [[]]], [[], [[]]]]
>>> [parent(i) for i in l]
[Rooted trees, Rooted trees, Rooted trees]
```

graft_on_root(*other*)

Return the tree obtained by grafting `other` on the root of `self`.

Here grafting means that one takes the disjoint union of `self` and `other`, and adds the root of `other` to the list of children of `self`. The root of the resulting tree is the root of `self`.

This is useful for free Nap algebras.

EXAMPLES:

```
sage: RT = RootedTree
sage: x = RT([])
sage: y = RT([x, x])
sage: x.graft_on_root(x)
[[]]
```

(continues on next page)

(continued from previous page)

```
sage: y.graft_on_root(x)
[[], [], []]
sage: x.graft_on_root(y)
[[[], []]]
```

```
>>> from sage.all import *
>>> RT = RootedTree
>>> x = RT([])
>>> y = RT([x, x])
>>> x.graft_on_root(x)
[[]]
>>> y.graft_on_root(x)
[[], [], []]
>>> x.graft_on_root(y)
[[[], []]]
```

is_empty()

Return if `self` is the empty tree.

For rooted trees, this always returns `False`.

Note

This is not the same as `bool(t)`, which returns whether `t` has some child or not.

EXAMPLES:

```
sage: t = RootedTrees(4)([[],[[]]])
sage: t.is_empty()
False
sage: bool(t)
True
sage: t = RootedTrees(1)([])
sage: t.is_empty()
False
sage: bool(t)
False
```

```
>>> from sage.all import *
>>> t = RootedTrees(Integer(4))([[],[[]]])
>>> t.is_empty()
False
>>> bool(t)
True
>>> t = RootedTrees(Integer(1))([])
>>> t.is_empty()
False
>>> bool(t)
False
```

normalize()

Normalize `self`.

This function is at the core of the implementation of rooted (unordered) trees. The underlying structure is provided by ordered rooted trees. Every rooted tree is represented by a normalized element in the set of its planar embeddings.

There should be no need to call `normalize` directly as it is called automatically upon creation and cloning or modification (by `NormalizedClonableList`).

The normalization has a recursive definition. It means first that every sub-tree is itself normalized, and also that sub-trees are sorted. Here the sort is performed according to the values of the `sort_key()` method.

EXAMPLES:

```
sage: RT = RootedTree
sage: RT([[],[[]]]) == RT([[[]],[]])  # indirect doctest
True
sage: rt1 = RT([[],[[]]])
sage: rt2 = RT([[[]],[]])
sage: rt1 is rt2
False
sage: rt1 == rt2
True
sage: rt1._get_list() == rt2._get_list()
True
```

```
>>> from sage.all import *
>>> RT = RootedTree
>>> RT([[],[[]]]) == RT([[[]],[]])  # indirect doctest
True
>>> rt1 = RT([[],[[]]])
>>> rt2 = RT([[[]],[]])
>>> rt1 is rt2
False
>>> rt1 == rt2
True
>>> rt1._get_list() == rt2._get_list()
True
```

single_graft(*x*, *grafting_function*, *path_prefix=()*)

Graft subtrees of x on `self` using the given function.

Let $x_1, x_2, \ldots, x_p$ be the children of the root of x. For each i, the subtree of x comprising all descendants of x_i is joined by a new edge to the vertex of `self` specified by the i-th path in the grafting function (i.e., by the path `grafting_function[i]`).

The number of vertices of the result is the sum of the numbers of vertices of `self` and x minus one, because the root of x is not used.

This is used to define the product of the Grossman-Larson algebras.

INPUT:

- `x` – a rooted tree
- `grafting_function` – list of paths in `self`
- `path_prefix` – tuple (default: `()`)

The `path_prefix` argument is only used for internal recursion.

EXAMPLES:

```
sage: LT = LabelledRootedTrees()
sage: y = LT([LT([],label='b')], label='a')
sage: x = LT([LT([],label='d')], label='c')
sage: y.single_graft(x,[(0,)])
a[b[d[]]]
sage: t = LT([LT([],label='b'),LT([],label='c')], label='a')
sage: s = LT([LT([],label='d'),LT([],label='e')], label='f')
sage: t.single_graft(s,[(0,),(1,)])
a[b[d[]], c[e[]]]
```

```
>>> from sage.all import *
>>> LT = LabelledRootedTrees()
>>> y = LT([LT([],label='b')], label='a')
>>> x = LT([LT([],label='d')], label='c')
>>> y.single_graft(x,[(Integer(0),)])
a[b[d[]]]
>>> t = LT([LT([],label='b'),LT([],label='c')], label='a')
>>> s = LT([LT([],label='d'),LT([],label='e')], label='f')
>>> t.single_graft(s,[(Integer(0),),(Integer(1),)])
a[b[d[]], c[e[]]]
```

sort_key()

Return a tuple of nonnegative integers encoding the rooted tree `self`.

The first entry of the tuple is the number of children of the root. Then the rest of the tuple is obtained as follows: List the tuples corresponding to all children (we are regarding the children themselves as trees). Order this list (not the tuples!) in lexicographically increasing order, and flatten it into a single tuple.

This tuple characterizes the rooted tree uniquely, and can be used to sort the rooted trees.

Note

The tree `self` must be normalized before calling this method (see *normalize()*). This does not matter unless you are inside the `clone()` context manager, because outside of it every rooted tree is already normalized.

Note

By default, this method does not encode any extra structure that `self` might have. If you have a subclass inheriting from *RootedTree* which allows for some extra structure, you need to override *sort_key()* in order to preserve this structure (for example, the *LabelledRootedTree* class does this in *LabelledRootedTree.sort_key()*). See the note in the docstring of *sage.combinat.ordered_tree.OrderedTree.sort_key()* for a pitfall.

EXAMPLES:

```
sage: RT = RootedTree
sage: RT([[],[[]]]).sort_key()
(2, 0, 1, 0)
sage: RT([[[]],[]]).sort_key()
(2, 0, 1, 0)
```

```
>>> from sage.all import *
>>> RT = RootedTree
>>> RT([[],[[]]]).sort_key()
(2, 0, 1, 0)
>>> RT([[[]],[]]).sort_key()
(2, 0, 1, 0)
```

class sage.combinat.rooted_tree.**RootedTrees**

Bases: UniqueRepresentation, Parent

Factory class for rooted trees.

INPUT:

- size – integer (optional)

OUTPUT:

the set of all rooted trees (of the given size size if specified)

EXAMPLES:

```
sage: RootedTrees()
Rooted trees

sage: RootedTrees(2)
Rooted trees with 2 nodes
```

```
>>> from sage.all import *
>>> RootedTrees()
Rooted trees

>>> RootedTrees(Integer(2))
Rooted trees with 2 nodes
```

class sage.combinat.rooted_tree.**RootedTrees_all**

Bases: DisjointUnionEnumeratedSets, *RootedTrees*

Class of all (unordered, unlabelled) rooted trees.

See *RootedTree* for a definition.

Element

alias of *RootedTree*

labelled_trees()

Return the set of labelled trees associated to self.

EXAMPLES:

```
sage: RootedTrees().labelled_trees()
Labelled rooted trees
```

```
>>> from sage.all import *
>>> RootedTrees().labelled_trees()
Labelled rooted trees
```

As a consequence:

```
sage: lb = RootedTrees()([[],[[], []]]).canonical_labelling()
sage: lb
1[2[], 3[4[], 5[]]]
sage: lb.__class__
<class 'sage.combinat.rooted_tree.LabelledRootedTrees_all_with_category.
↪element_class'>
sage: lb.parent()
Labelled rooted trees
```

```
>>> from sage.all import *
>>> lb = RootedTrees()([[],[[], []]]).canonical_labelling()
>>> lb
1[2[], 3[4[], 5[]]]
>>> lb.__class__
<class 'sage.combinat.rooted_tree.LabelledRootedTrees_all_with_category.
↪element_class'>
>>> lb.parent()
Labelled rooted trees
```

leaf()

Return a leaf tree with `self` as parent.

EXAMPLES:

```
sage: RootedTrees().leaf()
[]
```

```
>>> from sage.all import *
>>> RootedTrees().leaf()
[]
```

unlabelled_trees()

Return the set of unlabelled trees associated to `self`.

EXAMPLES:

```
sage: RootedTrees().unlabelled_trees()
Rooted trees
```

```
>>> from sage.all import *
>>> RootedTrees().unlabelled_trees()
Rooted trees
```

class sage.combinat.rooted_tree.**RootedTrees_size**(n)

Bases: *RootedTrees*

The enumerated set of rooted trees with a given number of nodes.

The number of nodes of a rooted tree is defined recursively: The number of nodes of a rooted tree with a children is a plus the sum of the number of nodes of each of these children.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: RootedTrees(1).cardinality()
1
sage: RootedTrees(3).cardinality()
2
```

```
>>> from sage.all import *
>>> RootedTrees(Integer(1)).cardinality()
1
>>> RootedTrees(Integer(3)).cardinality()
2
```

check_element(*el*, *check=True*)

Check that a given tree actually belongs to `self`.

This just checks the number of vertices.

EXAMPLES:

```
sage: RT3 = RootedTrees(3)
sage: RT3([[],[]])     # indirect doctest
[[], []]
sage: RT3([[],[],[]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: wrong number of nodes
```

```
>>> from sage.all import *
>>> RT3 = RootedTrees(Integer(3))
>>> RT3([[],[]])     # indirect doctest
[[], []]
>>> RT3([[],[],[]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: wrong number of nodes
```

element_class()

sage.combinat.rooted_tree.**number_of_rooted_trees**()

Return the number of rooted trees with n nodes.

Compute the number $a(n)$ of rooted trees with n nodes using the recursive formula ([SL000081]):

$$a(n+1) = \frac{1}{n}\sum_{k=1}^{n}\left(\sum_{d|k} da(d)\right) a(n-k+1)$$

EXAMPLES:

```
sage: from sage.combinat.rooted_tree import number_of_rooted_trees
sage: [number_of_rooted_trees(i) for i in range(10)]
[0, 1, 1, 2, 4, 9, 20, 48, 115, 286]
```

```
>>> from sage.all import *
>>> from sage.combinat.rooted_tree import number_of_rooted_trees
```

(continues on next page)

(continued from previous page)

```
>>> [number_of_rooted_trees(i) for i in range(Integer(10))]
[0, 1, 1, 2, 4, 9, 20, 48, 115, 286]
```

REFERENCES:

5.1.275 Robinson-Schensted-Knuth correspondence

AUTHORS:

- Travis Scrimshaw (2012-12-07): Initial version
- Chaman Agrawal (2019-06-24): Refactoring on the Rule class
- Matthew Lancellotti (2018): initial version of super RSK
- Jianping Pan, Wencin Poh, Anne Schilling (2020-08-31): initial version of RuleStar

Introduction

The Robinson-Schensted-Knuth (RSK) correspondence is most naturally stated as a bijection between generalized permutations (also known as two-line arrays, biwords, ...) and pairs of semi-standard Young tableaux (P, Q) of identical shape.

The basic operation in the RSK correspondence is a row insertion $P \leftarrow k$ (where P is a given semi-standard Young tableau, and k is an integer). Different insertion algorithms have been implemented for the RSK correspondence and can be specified as an argument in the function call.

EXAMPLES:

We can perform RSK and its inverse map on a variety of objects:

```
sage: p = Tableau([[1,2,2],[2]]); q = Tableau([[1,3,3],[2]])
sage: gp = RSK_inverse(p, q); gp
[[1, 2, 3, 3], [2, 1, 2, 2]]
sage: RSK(*gp) # RSK of a biword
[[[1, 2, 2], [2]], [[1, 3, 3], [2]]]
sage: RSK([2,3,2,1,2,3]) # Robinson-Schensted of a word
[[[1, 2, 2, 3], [2], [3]], [[1, 2, 5, 6], [3], [4]]]
sage: RSK([2,3,2,1,2,3], insertion=RSK.rules.EG) # Edelman-Greene
[[[1, 2, 3], [2, 3], [3]], [[1, 2, 6], [3, 5], [4]]]
sage: m = RSK_inverse(p, q, 'matrix'); m # output as matrix
[0 1]
[1 0]
[0 2]
sage: RSK(m) # RSK of a matrix
[[[1, 2, 2], [2]], [[1, 3, 3], [2]]]
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(2)]]); q =␣
↪Tableau([[Integer(1),Integer(3),Integer(3)],[Integer(2)]])
>>> gp = RSK_inverse(p, q); gp
[[1, 2, 3, 3], [2, 1, 2, 2]]
>>> RSK(*gp) # RSK of a biword
[[[1, 2, 2], [2]], [[1, 3, 3], [2]]]
>>> RSK([Integer(2),Integer(3),Integer(2),Integer(1),Integer(2),Integer(3)]) #␣
↪Robinson-Schensted of a word
```

(continues on next page)

(continued from previous page)

```
[[[1, 2, 2, 3], [2], [3]], [[1, 2, 5, 6], [3], [4]]]
>>> RSK([Integer(2),Integer(3),Integer(2),Integer(1),Integer(2),Integer(3)],␣
↪insertion=RSK.rules.EG) # Edelman-Greene
[[[1, 2, 3], [2, 3], [3]], [[1, 2, 6], [3, 5], [4]]]
>>> m = RSK_inverse(p, q, 'matrix'); m # output as matrix
[0 1]
[1 0]
[0 2]
>>> RSK(m) # RSK of a matrix
[[[1, 2, 2], [2]], [[1, 3, 3], [2]]]
```

Insertions currently available

The following insertion algorithms for RSK correspondence are currently available:

- RSK insertion (*RuleRSK*).
- Edelman-Greene insertion (*RuleEG*), an algorithm defined in [EG1987] Definition 6.20 (where it is referred to as Coxeter-Knuth insertion).
- Hecke RSK algorithm (*RuleHecke*) , defined using the Hecke insertion studied in [BKSTY06] (but using rows instead of columns).
- Dual RSK insertion (*RuleDualRSK*).
- CoRSK insertion (*RuleCoRSK*), defined in [GR2018v5sol].
- Super RSK insertion (*RuleSuperRSK*), a combination of row and column insertions defined in [Muth2019].
- Star insertion (*RuleStar*), defined in [MPPS2020].

Implementing your own insertion rule

The functions *RSK()* and *RSK_inverse()* are written so that it is easy to implement insertion algorithms you come across in your research.

To implement your own insertion algorithm, you first need to import the base class for a rule:

```
sage: from sage.combinat.rsk import Rule
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import Rule
```

Using the `Rule` class as parent class for your insertion rule, first implement the insertion and the reverse insertion algorithm for *RSK()* and *RSK_inverse()* respectively (as methods `forward_rule` and `backward_rule`). If your insertion algorithm uses the same forward and backward rules as `RuleRSK`, differing only in how an entry is inserted into a row, then this is not necessary, and it suffices to merely implement the `insertion` and `reverse_insertion` methods.

For more information, see *Rule*.

REFERENCES:

class sage.combinat.rsk.**InsertionRules**

Bases: `object`

Catalog of rules for RSK-like insertion algorithms.

EG

alias of *RuleEG*

Hecke

alias of *RuleHecke*

RSK

alias of *RuleRSK*

Star

alias of *RuleStar*

coRSK

alias of *RuleCoRSK*

dualRSK

alias of *RuleDualRSK*

superRSK

alias of *RuleSuperRSK*

sage.combinat.rsk.**RSK**(*obj1=None*, *obj2=None*, *insertion=<class 'sage.combinat.rsk.RuleRSK'>*, *check_standard=False*, ***options*)

Perform the Robinson-Schensted-Knuth (RSK) correspondence.

The Robinson-Schensted-Knuth (RSK) correspondence (also known as the RSK algorithm) is most naturally stated as a bijection between generalized permutations (also known as two-line arrays, biwords, ...) and pairs of semi-standard Young tableaux (P, Q) of identical shape. The tableau P is known as the insertion tableau, and Q is known as the recording tableau.

The basic operation is known as row insertion $P \leftarrow k$ (where P is a given semi-standard Young tableau, and k is an integer). Row insertion is a recursive algorithm which starts by setting $k_0 = k$, and in its i-th step inserts the number k_i into the i-th row of P (we start counting the rows at 0) by replacing the first integer greater than k_i in the row by k_i and defines k_{i+1} as the integer that has been replaced. If no integer greater than k_i exists in the i-th row, then k_i is simply appended to the row and the algorithm terminates at this point.

A *generalized permutation* (or *biword*) is a list $((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$ of pairs such that the letters $j_0, j_1, \ldots, j_{\ell-1}$ are weakly increasing (that is, $j_0 \leq j_1 \leq \cdots \leq j_{\ell-1}$), whereas the letters k_i satisfy $k_i \leq k_{i+1}$ whenever $j_i = j_{i+1}$. The ℓ-tuple $(j_0, j_1, \ldots, j_{\ell-1})$ is called the *top line* of this generalized permutation, whereas the ℓ-tuple $(k_0, k_1, \ldots, k_{\ell-1})$ is called its *bottom line*.

Now the RSK algorithm, applied to a generalized permutation $p = ((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$ (encoded as a lexicographically sorted list of pairs) starts by initializing two semi-standard tableaux P_0 and Q_0 as empty tableaux. For each nonnegative integer t starting at 0, take the pair (j_t, k_t) from p and set $P_{t+1} = P_t \leftarrow k_t$, and define Q_{t+1} by adding a new box filled with j_t to the tableau Q_t at the same location the row insertion on P_t ended (that is to say, adding a new box with entry j_t such that P_{t+1} and Q_{t+1} have the same shape). The iterative process stops when t reaches the size of p, and the pair (P_t, Q_t) at this point is the image of p under the Robinson-Schensted-Knuth correspondence.

This correspondence has been introduced in [Knu1970], where it has been referred to as "Construction A".

For more information, see Chapter 7 in [Sta-EC2].

We also note that integer matrices are in bijection with generalized permutations. Furthermore, we can convert any word w (and, in particular, any permutation) to a generalized permutation by considering the top row to be $(1, 2, \ldots, n)$ where n is the length of w.

The optional argument `insertion` allows to specify an alternative insertion procedure to be used instead of the standard Robinson-Schensted-Knuth insertion.

INPUT:

- `obj1`, `obj2` – can be one of the following:
  - a word in an ordered alphabet (in this case, `obj1` is said word, and `obj2` is `None`)
  - an integer matrix
  - two lists of equal length representing a generalized permutation (namely, the lists $(j_0, j_1, \ldots, j_{\ell-1})$ and $(k_0, k_1, \ldots, k_{\ell-1})$ represent the generalized permutation $((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$)
  - any object which has a method `_rsk_iter()` which returns an iterator over the object represented as generalized permutation or a pair of lists (in this case, `obj1` is said object, and `obj2` is `None`).
- `insertion` – (default: `RSK.rules.RSK`) the following types of insertion are currently supported:
  - `RSK.rules.RSK` (or `'RSK'`) – Robinson-Schensted-Knuth insertion (*RuleRSK*)
  - `RSK.rules.EG` (or `'EG'`) – Edelman-Greene insertion (only for reduced words of permutations/elements of a type A Coxeter group) (*RuleEG*)
  - `RSK.rules.Hecke` – (or `'hecke'`) Hecke insertion (only guaranteed for generalized permutations whose top row is strictly increasing) (*RuleHecke*)
  - `RSK.rules.dualRSK` (or `'dualRSK'`) – Dual RSK insertion (only for strict biwords) (*RuleDualRSK*)
  - `RSK.rules.coRSK` (or `'coRSK'`) – CoRSK insertion (only for strict cobiwords) (*RuleCoRSK*)
  - `RSK.rules.superRSK` (or `'super'`) – Super RSK insertion (only for restricted super biwords) (*RuleSuperRSK*)
  - `RSK.rules.Star` (or `'Star'`) – $\star$-insertion (only for fully commutative words in the 0-Hecke monoid) (*RuleStar*)
- `check_standard` – boolean (default: `False`); check if either of the resulting tableaux is a standard tableau, and if so, typecast it as such

For precise information about constraints on the input and output, as well as the definition of the algorithm (if it is not standard RSK), see the particular *Rule* class.

EXAMPLES:

If we only input one row, it is understood that the top row should be $(1, 2, \ldots, n)$:

```
sage: RSK([3,3,2,4,1])
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RSK(Word([3,3,2,4,1]))
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RSK(Word([2,3,3,2,1,3,2,3]))
[[[1, 2, 2, 3, 3], [2, 3], [3]], [[1, 2, 3, 6, 8], [4, 7], [5]]]
```

```
>>> from sage.all import *
>>> RSK([Integer(3),Integer(3),Integer(2),Integer(4),Integer(1)])
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
>>> RSK(Word([Integer(3),Integer(3),Integer(2),Integer(4),Integer(1)]))
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
>>> RSK(Word([Integer(2),Integer(3),Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3)]))
[[[1, 2, 2, 3, 3], [2, 3], [3]], [[1, 2, 3, 6, 8], [4, 7], [5]]]
```

We can provide a generalized permutation:

```
sage: RSK([1, 2, 2, 2], [2, 1, 1, 2])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
sage: RSK(Word([1,1,3,4,4]), [1,4,2,1,3])
[[[1, 1, 3], [2], [4]], [[1, 1, 4], [3], [4]]]
sage: RSK([1,3,3,4,4], Word([6,2,2,1,7]))
[[[1, 2, 7], [2], [6]], [[1, 3, 4], [3], [4]]]
```

```
>>> from sage.all import *
>>> RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2), Integer(1),
↪ Integer(1), Integer(2)])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
>>> RSK(Word([Integer(1),Integer(1),Integer(3),Integer(4),Integer(4)]),␣
↪[Integer(1),Integer(4),Integer(2),Integer(1),Integer(3)])
[[[1, 1, 3], [2], [4]], [[1, 1, 4], [3], [4]]]
>>> RSK([Integer(1),Integer(3),Integer(3),Integer(4),Integer(4)],␣
↪Word([Integer(6),Integer(2),Integer(2),Integer(1),Integer(7)]))
[[[1, 2, 7], [2], [6]], [[1, 3, 4], [3], [4]]]
```

We can provide a matrix:

```
sage: RSK(matrix([[0,1],[2,1]]))
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

```
>>> from sage.all import *
>>> RSK(matrix([[Integer(0),Integer(1)],[Integer(2),Integer(1)]]))
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

We can also provide something looking like a matrix:

```
sage: RSK([[0,1],[2,1]])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

```
>>> from sage.all import *
>>> RSK([[Integer(0),Integer(1)],[Integer(2),Integer(1)]])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

There is also `RSK_inverse()` which performs the inverse of the bijection on a pair of semistandard tableaux. We note that the inverse function takes 2 separate tableaux as inputs, so to compose with `RSK()`, we need to use the python `*` on the output:

```
sage: RSK_inverse(*RSK([1, 2, 2, 2], [2, 1, 1, 2]))
[[1, 2, 2, 2], [2, 1, 1, 2]]
sage: P,Q = RSK([1, 2, 2, 2], [2, 1, 1, 2])
sage: RSK_inverse(P, Q)
[[1, 2, 2, 2], [2, 1, 1, 2]]
```

```
>>> from sage.all import *
>>> RSK_inverse(*RSK([Integer(1), Integer(2), Integer(2), Integer(2)],␣
↪[Integer(2), Integer(1), Integer(1), Integer(2)]))
[[1, 2, 2, 2], [2, 1, 1, 2]]
>>> P,Q = RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2),␣
↪Integer(1), Integer(1), Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> RSK_inverse(P, Q)
[[1, 2, 2, 2], [2, 1, 1, 2]]
```

sage.combinat.rsk.**RSK_inverse**(*p*, *q*, *output='array'*, *insertion=<class 'sage.combinat.rsk.RuleRSK'>*)

Return the generalized permutation corresponding to the pair of tableaux (p, q) under the inverse of the Robinson-Schensted-Knuth correspondence.

For more information on the bijection, see *RSK()*.

INPUT:

- p, q – two semi-standard tableaux of the same shape, or (in the case when Hecke insertion is used) an increasing tableau and a set-valued tableau of the same shape (see the note below for the format of the set-valued tableau)
- output – (default: 'array') if q is semi-standard:
  - 'array' – as a two-line array (i.e. generalized permutation or biword)
  - 'matrix' – as an integer matrix

  and if q is standard, we can also have the output:
  - 'word' – as a word

  and additionally if p is standard, we can also have the output:
  - 'permutation' – as a permutation
- insertion – (default: RSK.rules.RSK) the insertion algorithm used in the bijection. Currently the following are supported:
  - RSK.rules.RSK (or 'RSK') – Robinson-Schensted-Knuth insertion (*RuleRSK*)
  - RSK.rules.EG (or 'EG') – Edelman-Greene insertion (only for reduced words of permutations/elements of a type A Coxeter group) (*RuleEG*)
  - RSK.rules.Hecke (or 'hecke') – Hecke insertion (only guaranteed for generalized permutations whose top row is strictly increasing) (*RuleHecke*)
  - RSK.rules.dualRSK (or 'dualRSK') – Dual RSK insertion (only for strict biwords) (*RuleDualRSK*)
  - RSK.rules.coRSK (or 'coRSK') – CoRSK insertion (only for strict cobiwords) (*RuleCoRSK*)
  - RSK.rules.superRSK (or 'super') – Super RSK insertion (only for restricted super biwords) (*RuleSuperRSK*)
  - RSK.rules.Star (or 'Star') – $\star$-insertion (only for fully commutative words in the 0-Hecke monoid) (*RuleStar*)

For precise information about constraints on the input and output, see the particular *Rule* class.

Note

In the case of Hecke insertion, the input variable q should be a set-valued tableau, encoded as a tableau whose entries are strictly increasing tuples of positive integers. Each such tuple encodes the set of its entries.

EXAMPLES:

If both p and q are standard:

```
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: RSK_inverse(t1, t2)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: RSK_inverse(t1, t2, 'word')
word: 14532
sage: RSK_inverse(t1, t2, 'matrix')
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
sage: RSK_inverse(t1, t2, 'permutation')
[1, 4, 5, 3, 2]
sage: RSK_inverse(t1, t1, 'permutation')
[1, 4, 3, 2, 5]
sage: RSK_inverse(t2, t2, 'permutation')
[1, 2, 5, 4, 3]
sage: RSK_inverse(t2, t1, 'permutation')
[1, 5, 4, 2, 3]
```

```
>>> from sage.all import *
>>> t1 = Tableau([[Integer(1), Integer(2), Integer(5)], [Integer(3)],␣
↪[Integer(4)]])
>>> t2 = Tableau([[Integer(1), Integer(2), Integer(3)], [Integer(4)],␣
↪[Integer(5)]])
>>> RSK_inverse(t1, t2)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
>>> RSK_inverse(t1, t2, 'word')
word: 14532
>>> RSK_inverse(t1, t2, 'matrix')
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
>>> RSK_inverse(t1, t2, 'permutation')
[1, 4, 5, 3, 2]
>>> RSK_inverse(t1, t1, 'permutation')
[1, 4, 3, 2, 5]
>>> RSK_inverse(t2, t2, 'permutation')
[1, 2, 5, 4, 3]
>>> RSK_inverse(t2, t1, 'permutation')
[1, 5, 4, 2, 3]
```

If the first tableau is semistandard:

```
sage: p = Tableau([[1,2,2],[3]]); q = Tableau([[1,2,4],[3]])
sage: ret = RSK_inverse(p, q); ret
[[1, 2, 3, 4], [1, 3, 2, 2]]
sage: RSK_inverse(p, q, 'word')
word: 1322
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(3)]]); q =␣
↪Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3)]])
>>> ret = RSK_inverse(p, q); ret
[[1, 2, 3, 4], [1, 3, 2, 2]]
>>> RSK_inverse(p, q, 'word')
word: 1322
```

In general:

```
sage: p = Tableau([[1,2,2],[2]]); q = Tableau([[1,3,3],[2]])
sage: RSK_inverse(p, q)
[[1, 2, 3, 3], [2, 1, 2, 2]]
sage: RSK_inverse(p, q, 'matrix')
[0 1]
[1 0]
[0 2]
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(2)]]); q =␣
↪Tableau([[Integer(1),Integer(3),Integer(3)],[Integer(2)]])
>>> RSK_inverse(p, q)
[[1, 2, 3, 3], [2, 1, 2, 2]]
>>> RSK_inverse(p, q, 'matrix')
[0 1]
[1 0]
[0 2]
```

Using Hecke insertion:

```
sage: w = [5, 4, 3, 1, 4, 2, 5, 5]
sage: pq = RSK(w, insertion=RSK.rules.Hecke)
sage: RSK_inverse(*pq, insertion=RSK.rules.Hecke, output='list')
[5, 4, 3, 1, 4, 2, 5, 5]
```

```
>>> from sage.all import *
>>> w = [Integer(5), Integer(4), Integer(3), Integer(1), Integer(4), Integer(2),␣
↪Integer(5), Integer(5)]
>>> pq = RSK(w, insertion=RSK.rules.Hecke)
>>> RSK_inverse(*pq, insertion=RSK.rules.Hecke, output='list')
[5, 4, 3, 1, 4, 2, 5, 5]
```

Note

The constructor of `Tableau` accepts not only semistandard tableaux, but also arbitrary lists that are fillings of a partition diagram. (And such lists are used, e.g., for the set-valued tableau `q` that is passed to `RSK_inverse(p, q, insertion='hecke')`.) The user is responsible for ensuring that the tableaux passed to `RSK_inverse` are of the right types (semistandard, standard, increasing, set-valued as needed).

class `sage.combinat.rsk.`**`Rule`**

Bases: `UniqueRepresentation`

Generic base class for an insertion rule for an RSK-type correspondence.

An instance of this class should implement a method `insertion()` (which can be applied to a letter `j` and a list `r`, and modifies `r` in place by "bumping" `j` into it appropriately; it then returns the bumped-out entry or `None` if no such entry exists) and a method `reverse_insertion()` (which does the same but for reverse bumping). It may also implement `_backward_format_output()` and `_forward_format_output()` if the RSK correspondence should return something other than (semi)standard tableaux (in the forward direction) and matrices or biwords (in the backward direction). The `to_pairs()` method should also be overridden if the input for the (forward) RSK correspondence is not the usual kind of biwords (i.e., pairs of two n-tuples $[a_1, a_2, \ldots, a_n]$ and $[b_1, b_2, \ldots, b_n]$ satisfying $(a_1, b_1) \leq (a_2, b_2) \leq \cdots \leq (a_n, b_n)$ in lexicographic order). Finally, it `forward_rule()` and `backward_rule()` have to be overridden if the overall structure of the RSK correspondence differs from that of classical RSK (see, e.g., the case of Hecke insertion, in which a letter bumped into a row may change a different row).

backward_rule(*p*, *q*, *output*)

Return the generalized permutation obtained by applying reverse insertion to a pair of tableaux `(p, q)`.

INPUT:

- `p`, `q` – two tableaux of the same shape
- `output` – (default: `'array'`) if `q` is semi-standard:
  - `'array'` – as a two-line array (i.e. generalized permutation or biword)
  - `'matrix'` – as an integer matrix

  and if `q` is standard, we can also have the output:
  - `'word'` – as a word

  and additionally if `p` is standard, we can also have the output:
  - `'permutation'` – as a permutation

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleRSK
sage: t1 = Tableau([[1, 3, 4], [2], [3]])
sage: t2 = Tableau([[1, 2, 4], [3], [5]])
sage: RuleRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [3, 3, 2, 4, 1]]
sage: t1 = Tableau([[1, 1, 1, 3, 7]])
sage: t2 = Tableau([[1, 2, 3, 4, 5]])
sage: RuleRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [1, 1, 1, 3, 7]]
sage: t1 = Tableau([[1, 3], [3], [6], [7]])
sage: t2 = Tableau([[1, 4], [2], [3], [5]])
sage: RuleRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [7, 6, 3, 3, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleRSK
>>> t1 = Tableau([[Integer(1), Integer(3), Integer(4)], [Integer(2)],
↪[Integer(3)]])
>>> t2 = Tableau([[Integer(1), Integer(2), Integer(4)], [Integer(3)],
↪[Integer(5)]])
>>> RuleRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [3, 3, 2, 4, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> t1 = Tableau([[Integer(1), Integer(1), Integer(1), Integer(3),␣
↪Integer(7)]])
>>> t2 = Tableau([[Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)]])
>>> RuleRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [1, 1, 1, 3, 7]]
>>> t1 = Tableau([[Integer(1), Integer(3)], [Integer(3)], [Integer(6)],␣
↪[Integer(7)]])
>>> t2 = Tableau([[Integer(1), Integer(4)], [Integer(2)], [Integer(3)],␣
↪[Integer(5)]])
>>> RuleRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [7, 6, 3, 3, 1]]
```

forward_rule(*obj1*, *obj2*, *check_standard=False*, *check=True*)

Return a pair of tableaux obtained by applying forward insertion to the generalized permutation `[obj1, obj2]`.

INPUT:

- `obj1`, `obj2` – can be one of the following ways to represent a generalized permutation (or, equivalently, biword):
  - two lists `obj1` and `obj2` of equal length, to be interpreted as the top row and the bottom row of the biword
  - a matrix `obj1` of nonnegative integers, to be interpreted as the generalized permutation in matrix form (in this case, `obj2` is `None`)
  - a word `obj1` in an ordered alphabet, to be interpreted as the bottom row of the biword (in this case, `obj2` is `None`; the top row of the biword is understood to be $(1, 2, \ldots, n)$ by default)
  - any object `obj1` which has a method `_rsk_iter()`, as long as this method returns an iterator yielding pairs of numbers, which then are interperted as top entries and bottom entries in the biword (in this case, `obj2` is `None`)
- `check_standard` – boolean (default: `False`); check if either of the resulting tableaux is a standard tableau, and if so, typecast it as such
- `check` – boolean (default: `True`); whether to check that `obj1` and `obj2` actually define a valid biword

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleRSK
sage: RuleRSK().forward_rule([3,3,2,4,1], None)
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RuleRSK().forward_rule([1, 1, 1, 3, 7], None)
[[[1, 1, 1, 3, 7]], [[1, 2, 3, 4, 5]]]
sage: RuleRSK().forward_rule([7, 6, 3, 3, 1], None)
[[[1, 3], [3], [6], [7]], [[1, 4], [2], [3], [5]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleRSK
>>> RuleRSK().forward_rule([Integer(3),Integer(3),Integer(2),Integer(4),
↪Integer(1)], None)
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
>>> RuleRSK().forward_rule([Integer(1), Integer(1), Integer(1), Integer(3),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(7)], None)
[[[1, 1, 1, 3, 7]], [[1, 2, 3, 4, 5]]]
>>> RuleRSK().forward_rule([Integer(7), Integer(6), Integer(3), Integer(3),␣
↪Integer(1)], None)
[[[1, 3], [3], [6], [7]], [[1, 4], [2], [3], [5]]]
```

to_pairs(*obj1=None*, *obj2=None*, *check=True*)

Given a valid input for the RSK algorithm, such as two n-tuples `obj1` $= [a_1, a_2, \ldots, a_n]$ and `obj2` $= [b_1, b_2, \ldots, b_n]$ forming a biword (i.e., satisfying $a_1 \leq a_2 \leq \cdots \leq a_n$, and if $a_i = a_{i+1}$, then $b_i \leq b_{i+1}$), or a matrix ("generalized permutation"), or a single word, return the array $[(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]$.

INPUT:

- `obj1`, `obj2` – anything representing a biword (see the doc of *forward_rule()* for the encodings accepted)
- `check` – boolean (default: `True`); whether to check that `obj1` and `obj2` actually define a valid biword

EXAMPLES:

```
sage: from sage.combinat.rsk import Rule
sage: list(Rule().to_pairs([1, 2, 2, 2], [2, 1, 1, 2]))
[(1, 2), (2, 1), (2, 1), (2, 2)]
sage: m = Matrix(ZZ, 3, 2, [0,1,1,0,0,2]) ; m
[0 1]
[1 0]
[0 2]
sage: list(Rule().to_pairs(m))
[(1, 2), (2, 1), (3, 2), (3, 2)]
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import Rule
>>> list(Rule().to_pairs([Integer(1), Integer(2), Integer(2), Integer(2)],␣
↪[Integer(2), Integer(1), Integer(1), Integer(2)]))
[(1, 2), (2, 1), (2, 1), (2, 2)]
>>> m = Matrix(ZZ, Integer(3), Integer(2), [Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(2)]) ; m
[0 1]
[1 0]
[0 2]
>>> list(Rule().to_pairs(m))
[(1, 2), (2, 1), (3, 2), (3, 2)]
```

class sage.combinat.rsk.**RuleCoRSK**

Bases: *RuleRSK*

Rule for coRSK insertion.

CoRSK insertion differs from classical RSK insertion in the following ways:

- The input (in terms of biwords) is no longer a biword, but rather a strict cobiword – i.e., a pair of two lists $[a_1, a_2, \ldots, a_n]$ and $[b_1, b_2, \ldots, b_n]$ that satisfy the strict inequalities $(a_1, b_1) \widetilde{<} (a_2, b_2) \widetilde{<} \cdots \widetilde{<} (a_n, b_n)$, where the binary relation $\widetilde{<}$ on pairs of integers is defined by having $(u_1, v_1) \widetilde{<} (u_2, v_2)$ if and only if either $u_1 < u_2$ or ($u_1 = u_2$ and $v_1 > v_2$). In terms of matrices, this means that the input is not an arbitrary matrix with nonnegative integer entries, but rather a $\{0, 1\}$-matrix (i.e., a matrix whose entries are 0's and 1's).

- The output still consists of two tableaux (P, Q) of equal shapes, but rather than both of them being semistandard, now Q is row-strict (i.e., its transpose is semistandard) while P is semistandard.

Bumping proceeds in the same way as for RSK insertion.

The RSK and coRSK algorithms agree for permutation matrices.

For more information, see Section A.4 in [Ful1997] (specifically, construction (1d)) or the second solution to Exercise 2.7.12(a) in [GR2018v5sol].

EXAMPLES:

```
sage: RSK([1,2,5,3,1], insertion = RSK.rules.coRSK)
[[[1, 1, 3], [2], [5]], [[1, 2, 3], [4], [5]]]
sage: RSK(Word([2,3,3,2,1,3,2,3]), insertion = RSK.rules.coRSK)
[[[1, 2, 2, 3, 3], [2, 3], [3]], [[1, 2, 3, 6, 8], [4, 7], [5]]]
sage: RSK(Word([3,3,2,4,1]), insertion = RSK.rules.coRSK)
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: from sage.combinat.rsk import to_matrix
sage: RSK(to_matrix([1, 1, 3, 3, 4], [3, 2, 2, 1, 3]), insertion = RSK.rules.
↪coRSK)
[[[1, 2, 3], [2], [3]], [[1, 3, 4], [1], [3]]]
```

```
>>> from sage.all import *
>>> RSK([Integer(1),Integer(2),Integer(5),Integer(3),Integer(1)], insertion = RSK.
↪rules.coRSK)
[[[1, 1, 3], [2], [5]], [[1, 2, 3], [4], [5]]]
>>> RSK(Word([Integer(2),Integer(3),Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3)]), insertion = RSK.rules.coRSK)
[[[1, 2, 2, 3, 3], [2, 3], [3]], [[1, 2, 3, 6, 8], [4, 7], [5]]]
>>> RSK(Word([Integer(3),Integer(3),Integer(2),Integer(4),Integer(1)]), insertion␣
↪= RSK.rules.coRSK)
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
>>> from sage.combinat.rsk import to_matrix
>>> RSK(to_matrix([Integer(1), Integer(1), Integer(3), Integer(3), Integer(4)],␣
↪[Integer(3), Integer(2), Integer(2), Integer(1), Integer(3)]), insertion = RSK.
↪rules.coRSK)
[[[1, 2, 3], [2], [3]], [[1, 3, 4], [1], [3]]]
```

Using coRSK insertion with a $\{0, 1\}$-matrix:

```
sage: RSK(matrix([[0,1],[1,0]]), insertion = RSK.rules.coRSK)
[[[1], [2]], [[1], [2]]]
```

```
>>> from sage.all import *
>>> RSK(matrix([[Integer(0),Integer(1)],[Integer(1),Integer(0)]]), insertion =␣
↪RSK.rules.coRSK)
[[[1], [2]], [[1], [2]]]
```

We can also give it something looking like a matrix:

```
sage: RSK([[0,1],[1,0]], insertion = RSK.rules.coRSK)
[[[1], [2]], [[1], [2]]]
```

```
>>> from sage.all import *
>>> RSK([[Integer(0),Integer(1)],[Integer(1),Integer(0)]], insertion = RSK.rules.
↪coRSK)
[[[1], [2]], [[1], [2]]]
```

We can also use the inverse correspondence:

```
sage: RSK_inverse(*RSK([1, 2, 2, 2], [2, 3, 2, 1],
....:          insertion=RSK.rules.coRSK),insertion=RSK.rules.coRSK)
[[1, 2, 2, 2], [2, 3, 2, 1]]
sage: P,Q = RSK([1, 2, 2, 2], [2, 3, 2, 1],insertion=RSK.rules.coRSK)
sage: RSK_inverse(P, Q, insertion=RSK.rules.coRSK)
[[1, 2, 2, 2], [2, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> RSK_inverse(*RSK([Integer(1), Integer(2), Integer(2), Integer(2)],
↪[Integer(2), Integer(3), Integer(2), Integer(1)],
...          insertion=RSK.rules.coRSK),insertion=RSK.rules.coRSK)
[[1, 2, 2, 2], [2, 3, 2, 1]]
>>> P,Q = RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2),
↪Integer(3), Integer(2), Integer(1)],insertion=RSK.rules.coRSK)
>>> RSK_inverse(P, Q, insertion=RSK.rules.coRSK)
[[1, 2, 2, 2], [2, 3, 2, 1]]
```

When applied to two standard tableaux, backwards coRSK insertion behaves identically to the usual backwards RSK insertion:

```
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: RSK_inverse(t1, t2, insertion=RSK.rules.coRSK)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: RSK_inverse(t1, t2, 'word', insertion=RSK.rules.coRSK)
word: 14532
sage: RSK_inverse(t1, t2, 'matrix', insertion=RSK.rules.coRSK)
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
sage: RSK_inverse(t1, t2, 'permutation', insertion=RSK.rules.coRSK)
[1, 4, 5, 3, 2]
sage: RSK_inverse(t1, t1, 'permutation', insertion=RSK.rules.coRSK)
[1, 4, 3, 2, 5]
sage: RSK_inverse(t2, t2, 'permutation', insertion=RSK.rules.coRSK)
[1, 2, 5, 4, 3]
sage: RSK_inverse(t2, t1, 'permutation', insertion=RSK.rules.coRSK)
[1, 5, 4, 2, 3]
```

```
>>> from sage.all import *
>>> t1 = Tableau([[Integer(1), Integer(2), Integer(5)], [Integer(3)],
↪[Integer(4)]])
>>> t2 = Tableau([[Integer(1), Integer(2), Integer(3)], [Integer(4)],
↪[Integer(5)]])
```

(continues on next page)

(continued from previous page)

```
>>> RSK_inverse(t1, t2, insertion=RSK.rules.coRSK)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
>>> RSK_inverse(t1, t2, 'word', insertion=RSK.rules.coRSK)
word: 14532
>>> RSK_inverse(t1, t2, 'matrix', insertion=RSK.rules.coRSK)
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
>>> RSK_inverse(t1, t2, 'permutation', insertion=RSK.rules.coRSK)
[1, 4, 5, 3, 2]
>>> RSK_inverse(t1, t1, 'permutation', insertion=RSK.rules.coRSK)
[1, 4, 3, 2, 5]
>>> RSK_inverse(t2, t2, 'permutation', insertion=RSK.rules.coRSK)
[1, 2, 5, 4, 3]
>>> RSK_inverse(t2, t1, 'permutation', insertion=RSK.rules.coRSK)
[1, 5, 4, 2, 3]
```

For coRSK, the first tableau is semistandard while the second tableau is transpose semistandard:

```
sage: p = Tableau([[1,2,2],[5]]); q = Tableau([[1,2,4],[3]])
sage: ret = RSK_inverse(p, q, insertion=RSK.rules.coRSK); ret
[[1, 2, 3, 4], [1, 5, 2, 2]]
sage: RSK_inverse(p, q, 'word', insertion=RSK.rules.coRSK)
word: 1522
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(5)]]); q =␣
↪Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3)]])
>>> ret = RSK_inverse(p, q, insertion=RSK.rules.coRSK); ret
[[1, 2, 3, 4], [1, 5, 2, 2]]
>>> RSK_inverse(p, q, 'word', insertion=RSK.rules.coRSK)
word: 1522
```

backward_rule(*p*, *q*, *output*)

Return the strict cobiword obtained by applying reverse coRSK insertion to a pair of tableaux `(p, q)`.

INPUT:

- `p`, `q` – two tableaux of the same shape
- `output` – (default: `'array'`) if `q` is row-strict:
  - `'array'` – as a two-line array (i.e. strict cobiword)
  - `'matrix'` – as a $\{0, 1\}$-matrix

  and if `q` is standard, we can have the output:
  - `'word'` – as a word

  and additionally if `p` is standard, we can also have the output:
  - `'permutation'` – as a permutation

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleCoRSK
sage: t1 = Tableau([[1, 1, 2], [2, 3], [4]])
sage: t2 = Tableau([[1, 4, 5], [1, 4], [2]])
sage: RuleCoRSK().backward_rule(t1, t2, 'array')
[[1, 1, 2, 4, 4, 5], [4, 2, 1, 3, 1, 2]]
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleCoRSK
>>> t1 = Tableau([[Integer(1), Integer(1), Integer(2)], [Integer(2),␣
↪Integer(3)], [Integer(4)]])
>>> t2 = Tableau([[Integer(1), Integer(4), Integer(5)], [Integer(1),␣
↪Integer(4)], [Integer(2)]])
>>> RuleCoRSK().backward_rule(t1, t2, 'array')
[[1, 1, 2, 4, 4, 5], [4, 2, 1, 3, 1, 2]]
```

to_pairs(*obj1=None*, *obj2=None*, *check=True*)

Given a valid input for the coRSK algorithm, such as two n-tuples `obj1` $= [a_1, a_2, \ldots, a_n]$ and `obj2` $= [b_1, b_2, \ldots, b_n]$ forming a strict cobiword (i.e., satisfying $a_1 \leq a_2 \leq \cdots \leq a_n$, and if $a_i = a_{i+1}$, then $b_i > b_{i+1}$), or a $\{0, 1\}$-matrix ("rook placement"), or a single word, return the array $[(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]$.

INPUT:

- `obj1`, `obj2` – anything representing a strict cobiword (see the doc of `forward_rule()` for the encodings accepted)
- `check` – boolean (default: `True`); whether to check that `obj1` and `obj2` actually define a valid strict cobiword

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleCoRSK
sage: list(RuleCoRSK().to_pairs([1, 2, 2, 2], [2, 3, 2, 1]))
[(1, 2), (2, 3), (2, 2), (2, 1)]
sage: RuleCoRSK().to_pairs([1, 2, 2, 2], [1, 2, 3, 3])
Traceback (most recent call last):
...
ValueError: invalid strict cobiword
sage: m = Matrix(ZZ, 3, 2, [0,1,1,1,0,1]) ; m
[0 1]
[1 1]
[0 1]
sage: list(RuleCoRSK().to_pairs(m))
[(1, 2), (2, 2), (2, 1), (3, 2)]
sage: m = Matrix(ZZ, 3, 2, [0,1,1,0,0,2]) ; m
[0 1]
[1 0]
[0 2]
sage: RuleCoRSK().to_pairs(m)
Traceback (most recent call last):
...
ValueError: coRSK requires a {0, 1}-matrix
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleCoRSK
>>> list(RuleCoRSK().to_pairs([Integer(1), Integer(2), Integer(2),␣
↪Integer(2)], [Integer(2), Integer(3), Integer(2), Integer(1)]))
[(1, 2), (2, 3), (2, 2), (2, 1)]
>>> RuleCoRSK().to_pairs([Integer(1), Integer(2), Integer(2), Integer(2)],␣
↪[Integer(1), Integer(2), Integer(3), Integer(3)])
Traceback (most recent call last):
...
ValueError: invalid strict cobiword
>>> m = Matrix(ZZ, Integer(3), Integer(2), [Integer(0),Integer(1),Integer(1),
↪Integer(1),Integer(0),Integer(1)]) ; m
[0 1]
[1 1]
[0 1]
>>> list(RuleCoRSK().to_pairs(m))
[(1, 2), (2, 2), (2, 1), (3, 2)]
>>> m = Matrix(ZZ, Integer(3), Integer(2), [Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(2)]) ; m
[0 1]
[1 0]
[0 2]
>>> RuleCoRSK().to_pairs(m)
Traceback (most recent call last):
...
ValueError: coRSK requires a {0, 1}-matrix
```

class sage.combinat.rsk.**RuleDualRSK**

Bases: *Rule*

Rule for dual RSK insertion.

Dual RSK insertion differs from classical RSK insertion in the following ways:

- The input (in terms of biwords) is no longer an arbitrary biword, but rather a strict biword (i.e., a pair of two lists $[a_1, a_2, \ldots, a_n]$ and $[b_1, b_2, \ldots, b_n]$ that satisfy the strict inequalities $(a_1, b_1) < (a_2, b_2) < \cdots < (a_n, b_n)$ in lexicographic order). In terms of matrices, this means that the input is not an arbitrary matrix with nonnegative integer entries, but rather a $\{0, 1\}$-matrix (i.e., a matrix whose entries are 0's and 1's).
- The output still consists of two tableaux (P, Q) of equal shapes, but rather than both of them being semistandard, now P is row-strict (i.e., its transpose is semistandard) while Q is semistandard.
- The main difference is in the way bumping works. Namely, when a number k_i is inserted into the i-th row of P, it bumps out the first integer greater **or equal to** k_i in this row (rather than greater than k_i).

The RSK and dual RSK algorithms agree for permutation matrices.

For more information, see Chapter 7, Section 14 in [Sta-EC2] (where dual RSK is called RSK*) or the third solution to Exercise 2.7.12(a) in [GR2018v5sol].

EXAMPLES:

```
sage: RSK([3,3,2,4,1], insertion=RSK.rules.dualRSK)
[[[1, 4], [2], [3], [3]], [[1, 4], [2], [3], [5]]]
sage: RSK(Word([3,3,2,4,1]), insertion=RSK.rules.dualRSK)
[[[1, 4], [2], [3], [3]], [[1, 4], [2], [3], [5]]]
```

(continues on next page)

(continued from previous page)

```
sage: RSK(Word([2,3,3,2,1,3,2,3]), insertion=RSK.rules.dualRSK)
[[[1, 2, 3], [2, 3], [2, 3], [3]], [[1, 2, 8], [3, 6], [4, 7], [5]]]
```

```
>>> from sage.all import *
>>> RSK([Integer(3),Integer(3),Integer(2),Integer(4),Integer(1)], insertion=RSK.
↪rules.dualRSK)
[[[1, 4], [2], [3], [3]], [[1, 4], [2], [3], [5]]]
>>> RSK(Word([Integer(3),Integer(3),Integer(2),Integer(4),Integer(1)]),␣
↪insertion=RSK.rules.dualRSK)
[[[1, 4], [2], [3], [3]], [[1, 4], [2], [3], [5]]]
>>> RSK(Word([Integer(2),Integer(3),Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3)]), insertion=RSK.rules.dualRSK)
[[[1, 2, 3], [2, 3], [2, 3], [3]], [[1, 2, 8], [3, 6], [4, 7], [5]]]
```

Using dual RSK insertion with a strict biword:

```
sage: RSK([1,1,2,4,4,5],[2,4,1,1,3,2], insertion=RSK.rules.dualRSK)
[[[1, 2], [1, 3], [2, 4]], [[1, 1], [2, 4], [4, 5]]]
sage: RSK([1,1,2,3,3,4,5],[1,3,2,1,3,3,2], insertion=RSK.rules.dualRSK)
[[[1, 2, 3], [1, 2], [3], [3]], [[1, 1, 3], [2, 4], [3], [5]]]
sage: RSK([1, 2, 2, 2], [2, 1, 2, 4], insertion=RSK.rules.dualRSK)
[[[1, 2, 4], [2]], [[1, 2, 2], [2]]]
sage: RSK(Word([1,1,3,4,4]), [1,4,2,1,3], insertion=RSK.rules.dualRSK)
[[[1, 2, 3], [1], [4]], [[1, 1, 4], [3], [4]]]
sage: RSK([1,3,3,4,4], Word([6,1,2,1,7]), insertion=RSK.rules.dualRSK)
[[[1, 2, 7], [1], [6]], [[1, 3, 4], [3], [4]]]
```

```
>>> from sage.all import *
>>> RSK([Integer(1),Integer(1),Integer(2),Integer(4),Integer(4),Integer(5)],
↪[Integer(2),Integer(4),Integer(1),Integer(1),Integer(3),Integer(2)],␣
↪insertion=RSK.rules.dualRSK)
[[[1, 2], [1, 3], [2, 4]], [[1, 1], [2, 4], [4, 5]]]
>>> RSK([Integer(1),Integer(1),Integer(2),Integer(3),Integer(3),Integer(4),
↪Integer(5)],[Integer(1),Integer(3),Integer(2),Integer(1),Integer(3),Integer(3),
↪Integer(2)], insertion=RSK.rules.dualRSK)
[[[1, 2, 3], [1, 2], [3], [3]], [[1, 1, 3], [2, 4], [3], [5]]]
>>> RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2), Integer(1),
↪ Integer(2), Integer(4)], insertion=RSK.rules.dualRSK)
[[[1, 2, 4], [2]], [[1, 2, 2], [2]]]
>>> RSK(Word([Integer(1),Integer(1),Integer(3),Integer(4),Integer(4)]),␣
↪[Integer(1),Integer(4),Integer(2),Integer(1),Integer(3)], insertion=RSK.rules.
↪dualRSK)
[[[1, 2, 3], [1], [4]], [[1, 1, 4], [3], [4]]]
>>> RSK([Integer(1),Integer(3),Integer(3),Integer(4),Integer(4)],␣
↪Word([Integer(6),Integer(1),Integer(2),Integer(1),Integer(7)]), insertion=RSK.
↪rules.dualRSK)
[[[1, 2, 7], [1], [6]], [[1, 3, 4], [3], [4]]]
```

Using dual RSK insertion with a $\{0, 1\}$-matrix:

```
sage: RSK(matrix([[0,1],[1,1]]), insertion=RSK.rules.dualRSK)
[[[1, 2], [2]], [[1, 2], [2]]]
```

```
>>> from sage.all import *
>>> RSK(matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]]), insertion=RSK.
↪rules.dualRSK)
[[[1, 2], [2]], [[1, 2], [2]]]
```

We can also give it something looking like a matrix:

```
sage: RSK([[0,1],[1,1]], insertion=RSK.rules.dualRSK)
[[[1, 2], [2]], [[1, 2], [2]]]
```

```
>>> from sage.all import *
>>> RSK([[Integer(0),Integer(1)],[Integer(1),Integer(1)]], insertion=RSK.rules.
↪dualRSK)
[[[1, 2], [2]], [[1, 2], [2]]]
```

Let us now call the inverse correspondence:

```
sage: RSK_inverse(*RSK([1, 2, 2, 2], [2, 1, 2, 3],
....:           insertion=RSK.rules.dualRSK),insertion=RSK.rules.dualRSK)
[[1, 2, 2, 2], [2, 1, 2, 3]]
sage: P,Q = RSK([1, 2, 2, 2], [2, 1, 2, 3],insertion=RSK.rules.dualRSK)
sage: RSK_inverse(P, Q, insertion=RSK.rules.dualRSK)
[[1, 2, 2, 2], [2, 1, 2, 3]]
```

```
>>> from sage.all import *
>>> RSK_inverse(*RSK([Integer(1), Integer(2), Integer(2), Integer(2)],␣
↪[Integer(2), Integer(1), Integer(2), Integer(3)],
...           insertion=RSK.rules.dualRSK),insertion=RSK.rules.dualRSK)
[[1, 2, 2, 2], [2, 1, 2, 3]]
>>> P,Q = RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2),␣
↪Integer(1), Integer(2), Integer(3)],insertion=RSK.rules.dualRSK)
>>> RSK_inverse(P, Q, insertion=RSK.rules.dualRSK)
[[1, 2, 2, 2], [2, 1, 2, 3]]
```

When applied to two standard tableaux, reverse dual RSK insertion behaves identically to the usual reverse RSK insertion:

```
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: RSK_inverse(t1, t2, insertion=RSK.rules.dualRSK)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: RSK_inverse(t1, t2, 'word', insertion=RSK.rules.dualRSK)
word: 14532
sage: RSK_inverse(t1, t2, 'matrix', insertion=RSK.rules.dualRSK)
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
sage: RSK_inverse(t1, t2, 'permutation', insertion=RSK.rules.dualRSK)
[1, 4, 5, 3, 2]
sage: RSK_inverse(t1, t1, 'permutation', insertion=RSK.rules.dualRSK)
[1, 4, 3, 2, 5]
```

(continues on next page)

(continued from previous page)

```
sage: RSK_inverse(t2, t2, 'permutation', insertion=RSK.rules.dualRSK)
[1, 2, 5, 4, 3]
sage: RSK_inverse(t2, t1, 'permutation', insertion=RSK.rules.dualRSK)
[1, 5, 4, 2, 3]
```

```
>>> from sage.all import *
>>> t1 = Tableau([[Integer(1), Integer(2), Integer(5)], [Integer(3)],␣
→[Integer(4)]])
>>> t2 = Tableau([[Integer(1), Integer(2), Integer(3)], [Integer(4)],␣
→[Integer(5)]])
>>> RSK_inverse(t1, t2, insertion=RSK.rules.dualRSK)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
>>> RSK_inverse(t1, t2, 'word', insertion=RSK.rules.dualRSK)
word: 14532
>>> RSK_inverse(t1, t2, 'matrix', insertion=RSK.rules.dualRSK)
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
>>> RSK_inverse(t1, t2, 'permutation', insertion=RSK.rules.dualRSK)
[1, 4, 5, 3, 2]
>>> RSK_inverse(t1, t1, 'permutation', insertion=RSK.rules.dualRSK)
[1, 4, 3, 2, 5]
>>> RSK_inverse(t2, t2, 'permutation', insertion=RSK.rules.dualRSK)
[1, 2, 5, 4, 3]
>>> RSK_inverse(t2, t1, 'permutation', insertion=RSK.rules.dualRSK)
[1, 5, 4, 2, 3]
```

Let us check that forward and backward dual RSK are mutually inverse when the first tableau is merely transpose semistandard:

```
sage: p = Tableau([[1,2,2],[1]]); q = Tableau([[1,2,4],[3]])
sage: ret = RSK_inverse(p, q, insertion=RSK.rules.dualRSK); ret
[[1, 2, 3, 4], [1, 2, 1, 2]]
sage: RSK_inverse(p, q, 'word', insertion=RSK.rules.dualRSK)
word: 1212
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(1)]]); q =␣
→Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3)]])
>>> ret = RSK_inverse(p, q, insertion=RSK.rules.dualRSK); ret
[[1, 2, 3, 4], [1, 2, 1, 2]]
>>> RSK_inverse(p, q, 'word', insertion=RSK.rules.dualRSK)
word: 1212
```

In general for dual RSK:

```
sage: p = Tableau([[1,1,2],[1]]); q = Tableau([[1,3,3],[2]])
sage: RSK_inverse(p, q, insertion=RSK.rules.dualRSK)
[[1, 2, 3, 3], [1, 1, 1, 2]]
sage: RSK_inverse(p, q, 'matrix', insertion=RSK.rules.dualRSK)
```

(continues on next page)

(continued from previous page)

```
[1 0]
[1 0]
[1 1]
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(1),Integer(2)],[Integer(1)]]); q =␣
↪Tableau([[Integer(1),Integer(3),Integer(3)],[Integer(2)]])
>>> RSK_inverse(p, q, insertion=RSK.rules.dualRSK)
[[1, 2, 3, 3], [1, 1, 1, 2]]
>>> RSK_inverse(p, q, 'matrix', insertion=RSK.rules.dualRSK)
[1 0]
[1 0]
[1 1]
```

insertion (j, r)

Insert the letter `j` from the second row of the biword into the row r using dual RSK insertion, if there is bumping to be done.

The row r is modified in place if bumping occurs. The bumped-out entry, if it exists, is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleDualRSK
sage: r = [1, 3, 4, 5]
sage: j = RuleDualRSK().insertion(4, r); j
4
sage: r
[1, 3, 4, 5]
sage: r = [1, 2, 3, 6, 7]
sage: j = RuleDualRSK().insertion(4, r); j
6
sage: r
[1, 2, 3, 4, 7]
sage: r = [1, 3]
sage: j = RuleDualRSK().insertion(4, r); j is None
True
sage: r
[1, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleDualRSK
>>> r = [Integer(1), Integer(3), Integer(4), Integer(5)]
>>> j = RuleDualRSK().insertion(Integer(4), r); j
4
>>> r
[1, 3, 4, 5]
>>> r = [Integer(1), Integer(2), Integer(3), Integer(6), Integer(7)]
>>> j = RuleDualRSK().insertion(Integer(4), r); j
6
>>> r
[1, 2, 3, 4, 7]
>>> r = [Integer(1), Integer(3)]
```

(continues on next page)

(continued from previous page)

```
>>> j = RuleDualRSK().insertion(Integer(4), r); j is None
True
>>> r
[1, 3]
```

reverse_insertion(*x*, *row*)

Reverse bump the row `row` of the current insertion tableau with the number `x` using dual RSK insertion.

The row `row` is modified in place. The bumped-out entry is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleDualRSK
sage: r = [1, 2, 4, 6, 7]
sage: x = RuleDualRSK().reverse_insertion(6, r); r
[1, 2, 4, 6, 7]
sage: x
6
sage: r = [1, 2, 4, 5, 7]
sage: x = RuleDualRSK().reverse_insertion(6, r); r
[1, 2, 4, 6, 7]
sage: x
5
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleDualRSK
>>> r = [Integer(1), Integer(2), Integer(4), Integer(6), Integer(7)]
>>> x = RuleDualRSK().reverse_insertion(Integer(6), r); r
[1, 2, 4, 6, 7]
>>> x
6
>>> r = [Integer(1), Integer(2), Integer(4), Integer(5), Integer(7)]
>>> x = RuleDualRSK().reverse_insertion(Integer(6), r); r
[1, 2, 4, 6, 7]
>>> x
5
```

to_pairs(*obj1=None*, *obj2=None*, *check=True*)

Given a valid input for the dual RSK algorithm, such as two n-tuples `obj1` $= [a_1, a_2, \ldots, a_n]$ and `obj2` $= [b_1, b_2, \ldots, b_n]$ forming a strict biword (i.e., satisfying $a_1 \leq a_2 \leq \cdots \leq a_n$, and if $a_i = a_{i+1}$, then $b_i < b_{i+1}$) or a $\{0, 1\}$-matrix ("rook placement"), or a single word, return the array $[(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]$.

INPUT:

- `obj1`, `obj2` – anything representing a strict biword (see the doc of `forward_rule()` for the encodings accepted)
- `check` – boolean (default: `True`); whether to check that `obj1` and `obj2` actually define a valid strict biword

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleDualRSK
sage: list(RuleDualRSK().to_pairs([1, 2, 2, 2], [2, 1, 2, 3]))
```

(continues on next page)

(continued from previous page)

```
[(1, 2), (2, 1), (2, 2), (2, 3)]
sage: RuleDualRSK().to_pairs([1, 2, 2, 2], [1, 2, 3, 3])
Traceback (most recent call last):
...
ValueError: invalid strict biword
sage: m = Matrix(ZZ, 3, 2, [0,1,1,1,0,1]) ; m
[0 1]
[1 1]
[0 1]
sage: list(RuleDualRSK().to_pairs(m))
[(1, 2), (2, 1), (2, 2), (3, 2)]
sage: m = Matrix(ZZ, 3, 2, [0,1,1,0,0,2]) ; m
[0 1]
[1 0]
[0 2]
sage: RuleDualRSK().to_pairs(m)
Traceback (most recent call last):
...
ValueError: dual RSK requires a {0, 1}-matrix
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleDualRSK
>>> list(RuleDualRSK().to_pairs([Integer(1), Integer(2), Integer(2),␣
→Integer(2)], [Integer(2), Integer(1), Integer(2), Integer(3)]))
[(1, 2), (2, 1), (2, 2), (2, 3)]
>>> RuleDualRSK().to_pairs([Integer(1), Integer(2), Integer(2), Integer(2)],␣
→[Integer(1), Integer(2), Integer(3), Integer(3)])
Traceback (most recent call last):
...
ValueError: invalid strict biword
>>> m = Matrix(ZZ, Integer(3), Integer(2), [Integer(0),Integer(1),Integer(1),
→Integer(1),Integer(0),Integer(1)]) ; m
[0 1]
[1 1]
[0 1]
>>> list(RuleDualRSK().to_pairs(m))
[(1, 2), (2, 1), (2, 2), (3, 2)]
>>> m = Matrix(ZZ, Integer(3), Integer(2), [Integer(0),Integer(1),Integer(1),
→Integer(0),Integer(0),Integer(2)]) ; m
[0 1]
[1 0]
[0 2]
>>> RuleDualRSK().to_pairs(m)
Traceback (most recent call last):
...
ValueError: dual RSK requires a {0, 1}-matrix
```

class sage.combinat.rsk.**RuleEG**

Bases: *Rule*

Rule for Edelman-Greene insertion.

For a reduced word of a permutation (i.e., an element of a type A Coxeter group), one can use Edelman-Greene insertion, an algorithm defined in [EG1987] Definition 6.20 (where it is referred to as Coxeter-Knuth insertion). The Edelman-Greene insertion is similar to the standard row insertion except that (using the notations in the documentation of `RSK()`) if k_i and $k_i + 1$ both exist in row i, we *only* set $k_{i+1} = k_i + 1$ and continue.

EXAMPLES:

Let us reproduce figure 6.4 in [EG1987]:

```
sage: RSK([2,3,2,1,2,3], insertion=RSK.rules.EG)
[[[1, 2, 3], [2, 3], [3]], [[1, 2, 6], [3, 5], [4]]]
```

```
>>> from sage.all import *
>>> RSK([Integer(2),Integer(3),Integer(2),Integer(1),Integer(2),Integer(3)],␣
↪insertion=RSK.rules.EG)
[[[1, 2, 3], [2, 3], [3]], [[1, 2, 6], [3, 5], [4]]]
```

Some more examples:

```
sage: a = [2, 1, 2, 3, 2]
sage: pq = RSK(a, insertion=RSK.rules.EG); pq
[[[1, 2, 3], [2, 3]], [[1, 3, 4], [2, 5]]]
sage: RSK(RSK_inverse(*pq, insertion=RSK.rules.EG, output='matrix'),
....:     insertion=RSK.rules.EG)
[[[1, 2, 3], [2, 3]], [[1, 3, 4], [2, 5]]]
sage: RSK_inverse(*pq, insertion=RSK.rules.EG)
[[1, 2, 3, 4, 5], [2, 1, 2, 3, 2]]
```

```
>>> from sage.all import *
>>> a = [Integer(2), Integer(1), Integer(2), Integer(3), Integer(2)]
>>> pq = RSK(a, insertion=RSK.rules.EG); pq
[[[1, 2, 3], [2, 3]], [[1, 3, 4], [2, 5]]]
>>> RSK(RSK_inverse(*pq, insertion=RSK.rules.EG, output='matrix'),
...     insertion=RSK.rules.EG)
[[[1, 2, 3], [2, 3]], [[1, 3, 4], [2, 5]]]
>>> RSK_inverse(*pq, insertion=RSK.rules.EG)
[[1, 2, 3, 4, 5], [2, 1, 2, 3, 2]]
```

The RSK algorithm (`RSK()`) built using the Edelman-Greene insertion rule `RuleEG` is a bijection from reduced words of permutations/elements of a type A Coxeter group to pairs consisting of an increasing tableau and a standard tableau of the same shape (see [EG1987] Theorem 6.25). The inverse of this bijection is obtained using `RSK_inverse()`. If the optional parameter `output = 'permutation'` is set in `RSK_inverse()`, then the function returns not the reduced word itself but the permutation (of smallest possible size) whose reduced word it is (although the order of the letters is reverse to the usual Sage convention):

```
sage: w = RSK_inverse(*pq, insertion=RSK.rules.EG, output='permutation'); w
[4, 3, 1, 2]
sage: list(reversed(a)) in w.reduced_words()
True
```

```
>>> from sage.all import *
>>> w = RSK_inverse(*pq, insertion=RSK.rules.EG, output='permutation'); w
[4, 3, 1, 2]
>>> list(reversed(a)) in w.reduced_words()
True
```

insertion(*j*, *r*)

Insert the letter j from the second row of the biword into the row r using Edelman-Greene insertion, if there is bumping to be done.

The row r is modified in place if bumping occurs. The bumped-out entry, if it exists, is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleEG
sage: qr, r =  [1,2,3,4,5], [3,3,2,4,8]
sage: j = RuleEG().insertion(9, r)
sage: j is None
True
sage: qr, r = [1,2,3,4,5], [2,3,4,5,8]
sage: j = RuleEG().insertion(3, r); r
[2, 3, 4, 5, 8]
sage: j
4
sage: qr, r = [1,2,3,4,5], [2,3,5,5,8]
sage: j = RuleEG().insertion(3, r); r
[2, 3, 3, 5, 8]
sage: j
5
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleEG
>>> qr, r =  [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],␣
↪[Integer(3),Integer(3),Integer(2),Integer(4),Integer(8)]
>>> j = RuleEG().insertion(Integer(9), r)
>>> j is None
True
>>> qr, r = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],␣
↪[Integer(2),Integer(3),Integer(4),Integer(5),Integer(8)]
>>> j = RuleEG().insertion(Integer(3), r); r
[2, 3, 4, 5, 8]
>>> j
4
>>> qr, r = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],␣
↪[Integer(2),Integer(3),Integer(5),Integer(5),Integer(8)]
>>> j = RuleEG().insertion(Integer(3), r); r
[2, 3, 3, 5, 8]
>>> j
5
```

reverse_insertion(*x*, *row*)

Reverse bump the row row of the current insertion tableau with the number x.

The row row is modified in place. The bumped-out entry is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleEG
sage: r =  [1,1,1,2,3,3]
sage: x = RuleEG().reverse_insertion(3, r); r
[1, 1, 1, 2, 3, 3]
```

(continues on next page)

(continued from previous page)

```
sage: x
2
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleEG
>>> r =  [Integer(1),Integer(1),Integer(1),Integer(2),Integer(3),Integer(3)]
>>> x = RuleEG().reverse_insertion(Integer(3), r); r
[1, 1, 1, 2, 3, 3]
>>> x
2
```

class sage.combinat.rsk.**RuleHecke**

Bases: *Rule*

Rule for Hecke insertion.

The Hecke RSK algorithm is similar to the classical RSK algorithm, but is defined using the Hecke insertion introduced in in [BKSTY06] (but using rows instead of columns). It is not clear in what generality it works; thus, following [BKSTY06], we shall assume that our biword p has top row $(1, 2, \ldots, n)$ (or, at least, has its top row strictly increasing).

The Hecke RSK algorithm returns a pair of an increasing tableau and a set-valued standard tableau. If $p = ((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$, then the algorithm recursively constructs pairs $(P_0, Q_0), (P_1, Q_1), \ldots, (P_\ell, Q_\ell)$ of tableaux. The construction of P_{t+1} and Q_{t+1} from P_t, Q_t, j_t and k_t proceeds as follows: Set $i = j_t$, $x = k_t$, $P = P_t$ and $Q = Q_t$. We are going to insert x into the increasing tableau P and update the set-valued "recording tableau" Q accordingly. As in the classical RSK algorithm, we first insert x into row 1 of P, then into row 2 of the resulting tableau, and so on, until the construction terminates. The details are different: Suppose we are inserting x into row R of P. If (Case 1) there exists an entry y in row R such that $x < y$, then let y be the minimal such entry. We replace this entry y with x if the result is still an increasing tableau; in either subcase, we then continue recursively, inserting y into the next row of P. If, on the other hand, (Case 2) no such y exists, then we append x to the end of R if the result is an increasing tableau (Subcase 2.1), and otherwise (Subcase 2.2) do nothing. Furthermore, in Subcase 2.1, we add the box that we have just filled with x in P to the shape of Q, and fill it with the one-element set $\{i\}$. In Subcase 2.2, we find the bottommost box of the column containing the rightmost box of row R, and add i to the entry of Q in this box (this entry is a set, since Q is set-valued). In either subcase, we terminate the recursion, and set $P_{t+1} = P$ and $Q_{t+1} = Q$.

Notice that set-valued tableaux are encoded as tableaux whose entries are tuples of positive integers; each such tuple is strictly increasing and encodes a set (namely, the set of its entries).

EXAMPLES:

As an example of Hecke insertion, we reproduce Example 2.1 in arXiv 0801.1319v2:

```
sage: w = [5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
sage: P,Q = RSK(w, insertion=RSK.rules.Hecke); [P,Q]
[[[1, 2, 4, 5], [2, 4, 5], [3, 5], [4], [5]],
 [[(1,), (4,), (5,), (7,)],
  [(2,), (9,), (11, 13)],
  [(3,), (12,)],
  [(6,)],
  [(8, 10)]]]
sage: wp = RSK_inverse(P, Q, insertion=RSK.rules.Hecke,
....:                     output='list'); wp
[5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
```

(continues on next page)

(continued from previous page)

```
sage: wp == w
True
```

```
>>> from sage.all import *
>>> w = [Integer(5), Integer(4), Integer(1), Integer(3), Integer(4), Integer(2),␣
↪Integer(5), Integer(1), Integer(2), Integer(1), Integer(4), Integer(2),␣
↪Integer(4)]
>>> P,Q = RSK(w, insertion=RSK.rules.Hecke); [P,Q]
[[[1, 2, 4, 5], [2, 4, 5], [3, 5], [4], [5]],
 [[(1,), (4,), (5,), (7,)],
  [(2,), (9,), (11, 13)],
  [(3,), (12,)],
  [(6,)],
  [(8, 10)]]]
>>> wp = RSK_inverse(P, Q, insertion=RSK.rules.Hecke,
...                     output='list'); wp
[5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
>>> wp == w
True
```

backward_rule(*p*, *q*, *output*)

Return the generalized permutation obtained by applying reverse Hecke insertion to a pair of tableaux `(p, q)`.

INPUT:

- `p`, `q` – two tableaux of the same shape
- `output` – (default: `'array'`) if `q` is semi-standard:
  - `'array'` – as a two-line array (i.e. generalized permutation or biword)

  and if `q` is standard set-valued, we can have the output:
  - `'word'` – as a word
  - `'list'` – as a list

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleHecke
sage: t1 = Tableau([[1, 4], [2], [3]])
sage: t2 = Tableau([[(1, 2), (4,)], [(3,)], [(5,)]])
sage: RuleHecke().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [3, 3, 2, 4, 1]]
sage: t1 = Tableau([[1, 4], [2, 3]])
sage: t2 = Tableau([[(1, 2), (4,)], [(3,)], [(5,)]])
sage: RuleHecke().backward_rule(t1, t2, 'array')
Traceback (most recent call last):
...
ValueError: p(=[[1, 4], [2, 3]]) and
 q(=[[(1, 2), (4,)], [(3,)], [(5,)]]) must have the same shape
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleHecke
```

(continues on next page)

(continued from previous page)

```
>>> t1 = Tableau([[Integer(1), Integer(4)], [Integer(2)], [Integer(3)]])
>>> t2 = Tableau([[(Integer(1), Integer(2)), (Integer(4),)], [(Integer(3),)],␣
↪[(Integer(5),)]])
>>> RuleHecke().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [3, 3, 2, 4, 1]]
>>> t1 = Tableau([[Integer(1), Integer(4)], [Integer(2), Integer(3)]])
>>> t2 = Tableau([[(Integer(1), Integer(2)), (Integer(4),)], [(Integer(3),)],␣
↪[(Integer(5),)]])
>>> RuleHecke().backward_rule(t1, t2, 'array')
Traceback (most recent call last):
...
ValueError: p(=[[1, 4], [2, 3]]) and
 q(=[[(1, 2), (4,)], [(3,)], [(5,)]]) must have the same shape
```

forward_rule(*obj1*, *obj2*, *check_standard=False*)

Return a pair of tableaux obtained by applying Hecke insertion to the generalized permutation `[obj1, obj2]`.

INPUT:

- `obj1`, `obj2` – can be one of the following ways to represent a generalized permutation (or, equivalently, biword):
  - two lists `obj1` and `obj2` of equal length, to be interpreted as the top row and the bottom row of the biword
  - a word `obj1` in an ordered alphabet, to be interpreted as the bottom row of the biword (in this case, `obj2` is `None`; the top row of the biword is understood to be $(1, 2, \ldots, n)$ by default)
- `check_standard` – boolean (default: `False`); check if either of the resulting tableaux is a standard tableau, and if so, typecast it as such

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleHecke
sage: p, q = RuleHecke().forward_rule([3,3,2,4,1], None);p
[[1, 4], [2], [3]]
sage: q
[[(1, 2), (4,)], [(3,)], [(5,)]]
sage: isinstance(p, SemistandardTableau)
True
sage: isinstance(q, Tableau)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleHecke
>>> p, q = RuleHecke().forward_rule([Integer(3),Integer(3),Integer(2),
↪Integer(4),Integer(1)], None);p
[[1, 4], [2], [3]]
>>> q
[[(1, 2), (4,)], [(3,)], [(5,)]]
>>> isinstance(p, SemistandardTableau)
True
>>> isinstance(q, Tableau)
True
```

insertion(*j*, *ir*, *r*, *p*)

Insert the letter `j` from the second row of the biword into the row r of the increasing tableau p using Hecke insertion, provided that r is the ir-th row of p, and provided that there is bumping to be done.

The row r is modified in place if bumping occurs. The bumped-out entry, if it exists, is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleHecke
sage: from bisect import bisect_right
sage: p, q, r =  [], [], [3,3,8,8,8,9]
sage: j, ir = 8, 1
sage: j1 = RuleHecke().insertion(j, ir, r, p)
sage: j1 == r[bisect_right(r, j)]
True
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleHecke
>>> from bisect import bisect_right
>>> p, q, r =  [], [], [Integer(3),Integer(3),Integer(8),Integer(8),
↪Integer(8),Integer(9)]
>>> j, ir = Integer(8), Integer(1)
>>> j1 = RuleHecke().insertion(j, ir, r, p)
>>> j1 == r[bisect_right(r, j)]
True
```

reverse_insertion(*i*, *x*, *row*, *p*)

Reverse bump the row `row` of the current insertion tableau `p` with the number `x`, provided that `row` is the i-th row of p.

The row `row` is modified in place. The bumped-out entry is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleHecke
sage: from bisect import bisect_left
sage: r =  [2,3,3,4,8,9]
sage: x, i, p = 9, 1, [1, 2]
sage: x1 = RuleHecke().reverse_insertion(i, x, r, p)
sage: x1 == r[bisect_left(r,x) - 1]
True
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleHecke
>>> from bisect import bisect_left
>>> r =  [Integer(2),Integer(3),Integer(3),Integer(4),Integer(8),Integer(9)]
>>> x, i, p = Integer(9), Integer(1), [Integer(1), Integer(2)]
>>> x1 = RuleHecke().reverse_insertion(i, x, r, p)
>>> x1 == r[bisect_left(r,x) - Integer(1)]
True
```

class sage.combinat.rsk.**RuleRSK**

Bases: *Rule*

Rule for the classical Robinson-Schensted-Knuth insertion.

See `RSK()` for the definition of this operation.

EXAMPLES:

```
sage: RSK([1, 2, 2, 2], [2, 1, 1, 2], insertion=RSK.rules.RSK)
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
sage: p = Tableau([[1,2,2],[2]]); q = Tableau([[1,3,3],[2]])
sage: RSK_inverse(p, q, insertion=RSK.rules.RSK)
[[1, 2, 3, 3], [2, 1, 2, 2]]
```

```
>>> from sage.all import *
>>> RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2), Integer(1),
↪ Integer(1), Integer(2)], insertion=RSK.rules.RSK)
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(2)]]); q =␣
↪Tableau([[Integer(1),Integer(3),Integer(3)],[Integer(2)]])
>>> RSK_inverse(p, q, insertion=RSK.rules.RSK)
[[1, 2, 3, 3], [2, 1, 2, 2]]
```

insertion(*j*, *r*)

Insert the letter `j` from the second row of the biword into the row r using classical Schensted insertion, if there is bumping to be done.

The row r is modified in place if bumping occurs. The bumped-out entry, if it exists, is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleRSK
sage: qr, r = [1,2,3,4,5], [3,3,2,4,8]
sage: j = RuleRSK().insertion(9, r)
sage: j is None
True
sage: qr, r = [1,2,3,4,5], [3,3,2,4,8]
sage: j = RuleRSK().insertion(3, r)
sage: j
4
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleRSK
>>> qr, r = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],␣
↪[Integer(3),Integer(3),Integer(2),Integer(4),Integer(8)]
>>> j = RuleRSK().insertion(Integer(9), r)
>>> j is None
True
>>> qr, r = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],␣
↪[Integer(3),Integer(3),Integer(2),Integer(4),Integer(8)]
>>> j = RuleRSK().insertion(Integer(3), r)
>>> j
4
```

reverse_insertion(*x*, *row*)

Reverse bump the row `row` of the current insertion tableau with the number `x`.

The row `row` is modified in place. The bumped-out entry is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleRSK
sage: r =  [2,3,3,4,8]
sage: x = RuleRSK().reverse_insertion(4, r); r
[2, 3, 4, 4, 8]
sage: x
3
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleRSK
>>> r =  [Integer(2),Integer(3),Integer(3),Integer(4),Integer(8)]
>>> x = RuleRSK().reverse_insertion(Integer(4), r); r
[2, 3, 4, 4, 8]
>>> x
3
```

class sage.combinat.rsk.**RuleStar**

Bases: *Rule*

Rule for $\star$-insertion.

The $\star$-insertion is similar to the classical RSK algorithm and is defined in [MPPS2020]. The bottom row of the increasing Hecke biword is a word in the 0-Hecke monoid that is fully commutative. When inserting a letter x into a row R, there are three cases:

- Case 1: If R is empty or $x > \max(R)$, append x to row R and terminate.
- Case 2: Otherwise if x is not in R, locate the smallest y in R with $y > x$. Bump y with x and insert y into the next row.
- Case 3: Otherwise, if x is in R, locate the smallest y in R with $y \leq x$ and interval $[y, x]$ contained in R. Row R remains unchanged and y is to be inserted into the next row.

The $\star$-insertion returns a pair consisting a conjugate of a semistandard tableau and a semistandard tableau. It is a bijection from the collection of all increasing Hecke biwords whose bottom row is a fully commutative word to pairs (P, Q) of tableaux of the same shape such that P is conjugate semistandard, Q is semistandard and the row reading word of P is fully commutative [MPPS2020].

EXAMPLES:

As an example of $\star$-insertion, we reproduce Example 28 in [MPPS2020]:

```
sage: from sage.combinat.rsk import RuleStar
sage: p,q = RuleStar().forward_rule([1,1,2,2,4,4], [1,3,2,4,2,4])
sage: ascii_art(p, q)
  1  2  4  1  1  2
  1  4     2  4
  3        4
sage: line1,line2 = RuleStar().backward_rule(p, q)
sage: line1,line2
([1, 1, 2, 2, 4, 4], [1, 3, 2, 4, 2, 4])
sage: RSK_inverse(p, q, output='DecreasingHeckeFactorization', insertion='Star')
(4, 2)()(4, 2)(3, 1)

sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
sage: h = DecreasingHeckeFactorization([[4, 2], [], [4, 2], [3, 1]])
```

(continues on next page)

(continued from previous page)

```
sage: RSK_inverse(*RSK(h,insertion='Star'),insertion='Star',
....:              output='DecreasingHeckeFactorization')
(4, 2)()(4, 2)(3, 1)
sage: p,q = RSK(h, insertion='Star')
sage: ascii_art(p, q)
1  2  4  1  1  2
1  4     2  4
3        4
sage: RSK_inverse(p, q, insertion='Star')
[[1, 1, 2, 2, 4, 4], [1, 3, 2, 4, 2, 4]]
sage: f = RSK_inverse(p, q, output='DecreasingHeckeFactorization', insertion='Star
↪')
sage: f == h
True
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleStar
>>> p,q = RuleStar().forward_rule([Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(4),Integer(4)], [Integer(1),Integer(3),Integer(2),Integer(4),Integer(2),
↪Integer(4)])
>>> ascii_art(p, q)
  1  2  4  1  1  2
  1  4     2  4
  3        4
>>> line1,line2 = RuleStar().backward_rule(p, q)
>>> line1,line2
([1, 1, 2, 2, 4, 4], [1, 3, 2, 4, 2, 4])
>>> RSK_inverse(p, q, output='DecreasingHeckeFactorization', insertion='Star')
(4, 2)()(4, 2)(3, 1)

>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
>>> h = DecreasingHeckeFactorization([[Integer(4), Integer(2)], [], [Integer(4),␣
↪Integer(2)], [Integer(3), Integer(1)]])
>>> RSK_inverse(*RSK(h,insertion='Star'),insertion='Star',
...              output='DecreasingHeckeFactorization')
(4, 2)()(4, 2)(3, 1)
>>> p,q = RSK(h, insertion='Star')
>>> ascii_art(p, q)
1  2  4  1  1  2
1  4     2  4
3        4
>>> RSK_inverse(p, q, insertion='Star')
[[1, 1, 2, 2, 4, 4], [1, 3, 2, 4, 2, 4]]
>>> f = RSK_inverse(p, q, output='DecreasingHeckeFactorization', insertion='Star')
>>> f == h
True
```

Warning

When `output` is set to `'DecreasingHeckeFactorization'`, the inverse of $\star$-insertion of (P, Q) returns a decreasing factorization whose number of factors is the maximum entry of Q:

```
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
sage: h1 = DecreasingHeckeFactorization([[],[3,1],[1]]); h1
()(3, 1)(1)
sage: P,Q = RSK(h1, insertion='Star')
sage: ascii_art(P, Q)
  1  3  1  2
  1     2
sage: h2 = RSK_inverse(P, Q, insertion='Star',
....: output='DecreasingHeckeFactorization'); h2
(3, 1)(1)
```

```
>>> from sage.all import *
>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
>>> h1 = DecreasingHeckeFactorization([[],[Integer(3),Integer(1)],
↪[Integer(1)]]); h1
()(3, 1)(1)
>>> P,Q = RSK(h1, insertion='Star')
>>> ascii_art(P, Q)
  1  3  1  2
  1     2
>>> h2 = RSK_inverse(P, Q, insertion='Star',
... output='DecreasingHeckeFactorization'); h2
(3, 1)(1)
```

backward_rule(*p*, *q*, *output='array'*)

Return the increasing Hecke biword obtained by applying reverse $\star$-insertion to a pair of tableaux `(p, q)`.

INPUT:

- `p`, `q` – two tableaux of the same shape, where `p` is the conjugate of a semistandard tableau, whose reading word is fully commutative and `q` is a semistandard tableau.
- `output` – (default: `'array'`) if `q` is semi-standard:
  - `'array'` – as a two-line array (i.e. generalized permutation or biword) that is an increasing Hecke biword
  - `'DecreasingHeckeFactorization'` – as a decreasing factorization in the 0-Hecke monoid

  and if `q` is standard:
  - `'word'` – as a (possibly non-reduced) word in the 0-Hecke monoid

Warning

When output is ‘DecreasingHeckeFactorization’, the number of factors in the output is the largest number in `obj1`.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleStar
sage: p,q = RuleStar().forward_rule([1,1,2,2,4,4], [1,3,2,4,2,4])
```

(continues on next page)

(continued from previous page)

```
sage: ascii_art(p, q)
  1  2  4  1  1  2
  1  4     2  4
  3        4
sage: line1,line2 = RuleStar().backward_rule(p, q); line1,line2
([1, 1, 2, 2, 4, 4], [1, 3, 2, 4, 2, 4])
sage: RuleStar().backward_rule(p, q, output = 'DecreasingHeckeFactorization')
(4, 2)()(4, 2)(3, 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleStar
>>> p,q = RuleStar().forward_rule([Integer(1),Integer(1),Integer(2),
↪Integer(2),Integer(4),Integer(4)], [Integer(1),Integer(3),Integer(2),
↪Integer(4),Integer(2),Integer(4)])
>>> ascii_art(p, q)
  1  2  4  1  1  2
  1  4     2  4
  3        4
>>> line1,line2 = RuleStar().backward_rule(p, q); line1,line2
([1, 1, 2, 2, 4, 4], [1, 3, 2, 4, 2, 4])
>>> RuleStar().backward_rule(p, q, output = 'DecreasingHeckeFactorization')
(4, 2)()(4, 2)(3, 1)
```

forward_rule(*obj1*, *obj2=None*, *check_braid=True*)

Return a pair of tableaux obtained by applying forward insertion to the increasing Hecke biword `[obj1, obj2]`.

INPUT:

- `obj1`, `obj2` – can be one of the following ways to represent a biword (or, equivalently, an increasing 0-Hecke factorization) that is fully commutative:
  - two lists `obj1` and `obj2` of equal length, to be interpreted as the top row and the bottom row of the biword.
  - a word `obj1` in an ordered alphabet, to be interpreted as the bottom row of the biword (in this case, `obj2` is `None`; the top row of the biword is understood to be $(1, 2, \ldots, n)$ by default).
  - a DecreasingHeckeFactorization `obj1`, the whose increasing Hecke biword will be interpreted as the bottom row; the top row is understood to be the indices of the factors for each letter in this biword.
- `check_braid` – boolean (default: `True`); indicator to validate that input is associated to a fully commutative word in the 0-Hecke monoid, validation is performed if set to `True`. Otherwise, this validation is ignored.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleStar
sage: p,q = RuleStar().forward_rule([1,1,2,3,3], [2,3,3,1,3]); p,q
([[1, 3], [2, 3], [2]], [[1, 1], [2, 3], [3]])
sage: p,q = RuleStar().forward_rule([2,3,3,1,3]); p,q
([[1, 3], [2, 3], [2]], [[1, 2], [3, 5], [4]])
sage: p,q = RSK([1,1,2,3,3], [2,3,3,1,3], insertion=RSK.rules.Star); p,q
([[1, 3], [2, 3], [2]], [[1, 1], [2, 3], [3]])
```

(continues on next page)

(continued from previous page)

```
sage: from sage.combinat.crystals.fully_commutative_stable_grothendieck␣
↪import DecreasingHeckeFactorization
sage: h = DecreasingHeckeFactorization([[3, 1], [3], [3, 2]])
sage: p,q = RSK(h, insertion=RSK.rules.Star); p,q
([[1, 3], [2, 3], [2]], [[1, 1], [2, 3], [3]])
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleStar
>>> p,q = RuleStar().forward_rule([Integer(1),Integer(1),Integer(2),
↪Integer(3),Integer(3)], [Integer(2),Integer(3),Integer(3),Integer(1),
↪Integer(3)]); p,q
([[1, 3], [2, 3], [2]], [[1, 1], [2, 3], [3]])
>>> p,q = RuleStar().forward_rule([Integer(2),Integer(3),Integer(3),
↪Integer(1),Integer(3)]); p,q
([[1, 3], [2, 3], [2]], [[1, 2], [3, 5], [4]])
>>> p,q = RSK([Integer(1),Integer(1),Integer(2),Integer(3),Integer(3)],␣
↪[Integer(2),Integer(3),Integer(3),Integer(1),Integer(3)], insertion=RSK.
↪rules.Star); p,q
([[1, 3], [2, 3], [2]], [[1, 1], [2, 3], [3]])

>>> from sage.combinat.crystals.fully_commutative_stable_grothendieck import␣
↪DecreasingHeckeFactorization
>>> h = DecreasingHeckeFactorization([[Integer(3), Integer(1)], [Integer(3)],␣
↪[Integer(3), Integer(2)]])
>>> p,q = RSK(h, insertion=RSK.rules.Star); p,q
([[1, 3], [2, 3], [2]], [[1, 1], [2, 3], [3]])
```

insertion(b, r)

Insert the letter `b` from the second row of the biword into the row `r` using $\star$-insertion defined in [MPPS2020].

The row r is modified in place if bumping occurs and b is not in row r. The bumped-out entry, if it exists, is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleStar
sage: RuleStar().insertion(3, [1,2,4,5])
4
sage: RuleStar().insertion(3, [1,2,3,5])
1
sage: RuleStar().insertion(6, [1,2,3,5]) is None
True
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleStar
>>> RuleStar().insertion(Integer(3), [Integer(1),Integer(2),Integer(4),
↪Integer(5)])
4
>>> RuleStar().insertion(Integer(3), [Integer(1),Integer(2),Integer(3),
↪Integer(5)])
1
>>> RuleStar().insertion(Integer(6), [Integer(1),Integer(2),Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(5)]) is None
True
```

reverse_insertion(*x*, *r*)

Reverse bump the row `r` of the current insertion tableau `p` with number `x`, provided that `r` is the `i`-th row of `p`.

The row `r` is modified in place. The bumped-out entry is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleStar
sage: RuleStar().reverse_insertion(4, [1,2,3,5])
3
sage: RuleStar().reverse_insertion(1, [1,2,3,5])
3
sage: RuleStar().reverse_insertion(5, [1,2,3,5])
5
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleStar
>>> RuleStar().reverse_insertion(Integer(4), [Integer(1),Integer(2),
↪Integer(3),Integer(5)])
3
>>> RuleStar().reverse_insertion(Integer(1), [Integer(1),Integer(2),
↪Integer(3),Integer(5)])
3
>>> RuleStar().reverse_insertion(Integer(5), [Integer(1),Integer(2),
↪Integer(3),Integer(5)])
5
```

class sage.combinat.rsk.**RuleSuperRSK**

Bases: *RuleRSK*

Rule for super RSK insertion.

Super RSK is based on ϵ-insertion, a combination of row and column classical RSK insertion.

Super RSK insertion differs from the classical RSK insertion in the following ways:

- The input (in terms of biwords) is no longer an arbitrary biword, but rather a restricted super biword (i.e., a pair of two lists $[a_1, a_2, \ldots, a_n]$ and $[b_1, b_2, \ldots, b_n]$ that contains entries with even and odd parity and pairs with mixed parity entries do not repeat).
- The output still consists of two tableaux (P, Q) of equal shapes, but rather than both of them being semistandard, now they are semistandard super tableaux.
- The main difference is in the way bumping works. Instead of having only row bumping super RSK uses ϵ-insertion, a combination of classical RSK bumping along the rows and a dual RSK like bumping (i.e. when a number k_i is inserted into the i-th row of P, it bumps out the first integer greater **or equal to** k_i in the column) along the column.

EXAMPLES:

```
sage: RSK([1], [1], insertion='superRSK')
[[[1]], [[1]]]
```

(continues on next page)

(continued from previous page)

```
sage: RSK([1, 2], [1, 3], insertion='superRSK')
[[[1, 3]], [[1, 2]]]
sage: RSK([1, 2, 3], [1, 3, "3p"], insertion='superRSK')
[[[1, 3], [3']], [[1, 2], [3]]]
sage: RSK([1, 3, "3p", "2p"], insertion='superRSK')
[[[1, 3', 3], [2']], [[1', 1, 2'], [2]]]
sage: RSK(["1p", "2p", 2, 2, "3p", "3p", 3, 3],
....:     ["1p", 1, "2p", 2, "3p", "3p", "3p", 3], insertion='superRSK')
[[[1', 2, 3', 3], [1, 3'], [2'], [3']], [[1', 2, 3', 3], [2', 3'], [2], [3]]]
sage: P = SemistandardSuperTableau([[1, '3p', 3], ['2p']])
sage: Q = SemistandardSuperTableau([['1p', 1, '2p'], [2]])
sage: RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1', 1, 2', 2], [1, 3, 3', 2']]
```

```
>>> from sage.all import *
>>> RSK([Integer(1)], [Integer(1)], insertion='superRSK')
[[[1]], [[1]]]
>>> RSK([Integer(1), Integer(2)], [Integer(1), Integer(3)], insertion='superRSK')
[[[1, 3]], [[1, 2]]]
>>> RSK([Integer(1), Integer(2), Integer(3)], [Integer(1), Integer(3), "3p"],␣
↪insertion='superRSK')
[[[1, 3], [3']], [[1, 2], [3]]]
>>> RSK([Integer(1), Integer(3), "3p", "2p"], insertion='superRSK')
[[[1, 3', 3], [2']], [[1', 1, 2'], [2]]]
>>> RSK(["1p", "2p", Integer(2), Integer(2), "3p", "3p", Integer(3), Integer(3)],
...     ["1p", Integer(1), "2p", Integer(2), "3p", "3p", "3p", Integer(3)],␣
↪insertion='superRSK')
[[[1', 2, 3', 3], [1, 3'], [2'], [3']], [[1', 2, 3', 3], [2', 3'], [2], [3]]]
>>> P = SemistandardSuperTableau([[Integer(1), '3p', Integer(3)], ['2p']])
>>> Q = SemistandardSuperTableau([['1p', Integer(1), '2p'], [Integer(2)]])
>>> RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1', 1, 2', 2], [1, 3, 3', 2']]
```

We apply super RSK on Example 5.1 in [Muth2019]:

```
sage: P,Q = RSK(["1p", "2p", 2, 2, "3p", "3p", 3, 3],
....:           ["3p", 1, 2, 3, "3p", "3p", "2p", "1p"], insertion='superRSK')
sage: (P, Q)
([[1', 2', 3', 3], [1, 2, 3'], [3']], [[1', 2, 2, 3'], [2', 3, 3], [3']])
sage: ascii_art((P, Q))
(  1' 2' 3'  3    1'  2  2 3' )
(   1  2 3'       2'  3  3    )
(  3'         ,   3'          )
sage: RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1', 2', 2, 2, 3', 3', 3, 3], [3', 1, 2, 3, 3', 3', 2', 1']]
```

```
>>> from sage.all import *
>>> P,Q = RSK(["1p", "2p", Integer(2), Integer(2), "3p", "3p", Integer(3),␣
↪Integer(3)],
...           ["3p", Integer(1), Integer(2), Integer(3), "3p", "3p", "2p", "1p"],␣
↪insertion='superRSK')
>>> (P, Q)
```

(continues on next page)

(continued from previous page)

```
([[1', 2', 3', 3], [1, 2, 3'], [3']], [[1', 2, 2, 3'], [2', 3, 3], [3']])
>>> ascii_art((P, Q))
(  1' 2' 3'  3   1'  2  2 3' )
(   1  2 3'      2'  3  3    )
(  3'         ,  3'          )
>>> RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1', 2', 2, 2, 3', 3', 3, 3], [3', 1, 2, 3, 3', 3', 2', 1']]
```

Example 6.1 in [Muth2019]:

```
sage: P,Q = RSK(["1p", "2p", 2, 2, "3p", "3p", 3, 3],
....:           ["3p", 1, 2, 3, "3p", "3p", "2p", "1p"], insertion='superRSK')
sage: ascii_art((P, Q))
(  1' 2' 3'  3   1'  2  2 3' )
(   1  2 3'      2'  3  3    )
(  3'         ,  3'          )
sage: RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1', 2', 2, 2, 3', 3', 3, 3], [3', 1, 2, 3, 3', 3', 2', 1']]

sage: P,Q = RSK(["1p", 1, "2p", 2, "3p", "3p", "3p", 3],
....:           [3, "2p", 3, 2, "3p", "3p", "1p", 2], insertion='superRSK')
sage: ascii_art((P, Q))
(  1'  2  2 3'   1' 2' 3'  3 )
(  2'  3  3       1  2 3'    )
(  3'         ,  3'          )
sage: RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1', 1, 2', 2, 3', 3', 3', 3], [3, 2', 3, 2, 3', 3', 1', 2]]
```

```
>>> from sage.all import *
>>> P,Q = RSK(["1p", "2p", Integer(2), Integer(2), "3p", "3p", Integer(3),␣
↪Integer(3)],
...           ["3p", Integer(1), Integer(2), Integer(3), "3p", "3p", "2p", "1p"],␣
↪insertion='superRSK')
>>> ascii_art((P, Q))
(  1' 2' 3'  3   1'  2  2 3' )
(   1  2 3'      2'  3  3    )
(  3'         ,  3'          )
>>> RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1', 2', 2, 2, 3', 3', 3, 3], [3', 1, 2, 3, 3', 3', 2', 1']]

>>> P,Q = RSK(["1p", Integer(1), "2p", Integer(2), "3p", "3p", "3p", Integer(3)],
...           [Integer(3), "2p", Integer(3), Integer(2), "3p", "3p", "1p",␣
↪Integer(2)], insertion='superRSK')
>>> ascii_art((P, Q))
(  1'  2  2 3'   1' 2' 3'  3 )
(  2'  3  3       1  2 3'    )
(  3'         ,  3'          )
>>> RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1', 1, 2', 2, 3', 3', 3', 3], [3, 2', 3, 2, 3', 3', 1', 2]]
```

Let us now call the inverse correspondence:

```
sage: P, Q = RSK([1, 2, 2, 2], [2, 1, 2, 3],
....:            insertion=RSK.rules.superRSK)
sage: RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1, 2, 2, 2], [2, 1, 2, 3]]
```

```
>>> from sage.all import *
>>> P, Q = RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2),␣
↪Integer(1), Integer(2), Integer(3)],
...            insertion=RSK.rules.superRSK)
>>> RSK_inverse(P, Q, insertion=RSK.rules.superRSK)
[[1, 2, 2, 2], [2, 1, 2, 3]]
```

When applied to two tableaux with only even parity elements, reverse super RSK insertion behaves identically to the usual reversel RSK insertion:

```
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: RSK_inverse(t1, t2, insertion=RSK.rules.RSK)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: t1 = SemistandardSuperTableau([[1, 2, 5], [3], [4]])
sage: t2 = SemistandardSuperTableau([[1, 2, 3], [4], [5]])
sage: RSK_inverse(t1, t2, insertion=RSK.rules.superRSK)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
```

```
>>> from sage.all import *
>>> t1 = Tableau([[Integer(1), Integer(2), Integer(5)], [Integer(3)],␣
↪[Integer(4)]])
>>> t2 = Tableau([[Integer(1), Integer(2), Integer(3)], [Integer(4)],␣
↪[Integer(5)]])
>>> RSK_inverse(t1, t2, insertion=RSK.rules.RSK)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
>>> t1 = SemistandardSuperTableau([[Integer(1), Integer(2), Integer(5)],␣
↪[Integer(3)], [Integer(4)]])
>>> t2 = SemistandardSuperTableau([[Integer(1), Integer(2), Integer(3)],␣
↪[Integer(4)], [Integer(5)]])
>>> RSK_inverse(t1, t2, insertion=RSK.rules.superRSK)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
```

backward_rule(*p*, *q*, *output='array'*)

Return the restricted super biword obtained by applying reverse super RSK insertion to a pair of tableaux `(p, q)`.

INPUT:

- `p`, `q` – two tableaux of the same shape
- `output` – (default: `'array'`) if `q` is row-strict:
  - `'array'` – as a two-line array (i.e. restricted super biword)

  and if `q` is standard, we can have the output:
  - `'word'` – as a word

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleSuperRSK
sage: t1 = SemistandardSuperTableau([['1p', '3p', '4p'], [2], [3]])
sage: t2 = SemistandardSuperTableau([[1, 2, 4], [3], [5]])
sage: RuleSuperRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [4', 3, 3', 2, 1']]
sage: t1 = SemistandardSuperTableau([[1, 3], ['3p']])
sage: t2 = SemistandardSuperTableau([[1, 2], [3]])
sage: RuleSuperRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3], [1, 3, 3']]
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleSuperRSK
>>> t1 = SemistandardSuperTableau([['1p', '3p', '4p'], [Integer(2)],
↪[Integer(3)]])
>>> t2 = SemistandardSuperTableau([[Integer(1), Integer(2), Integer(4)],
↪[Integer(3)], [Integer(5)]])
>>> RuleSuperRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3, 4, 5], [4', 3, 3', 2, 1']]
>>> t1 = SemistandardSuperTableau([[Integer(1), Integer(3)], ['3p']])
>>> t2 = SemistandardSuperTableau([[Integer(1), Integer(2)], [Integer(3)]])
>>> RuleSuperRSK().backward_rule(t1, t2, 'array')
[[1, 2, 3], [1, 3, 3']]
```

forward_rule(*obj1*, *obj2*, *check_standard=False*, *check=True*)

Return a pair of tableaux obtained by applying forward insertion to the restricted super biword `[obj1, obj2]`.

INPUT:

- `obj1`, `obj2` – can be one of the following ways to represent a generalized permutation (or, equivalently, biword):
  - two lists `obj1` and `obj2` of equal length, to be interpreted as the top row and the bottom row of the biword
  - a word `obj1` in an ordered alphabet, to be interpreted as the bottom row of the biword (in this case, `obj2` is `None`; the top row of the biword is understood to be $(1, 2, \ldots, n)$ by default)
  - any object `obj1` which has a method `_rsk_iter()`, as long as this method returns an iterator yielding pairs of numbers, which then are interperted as top entries and bottom entries in the biword (in this case, `obj2` is `None`)
- `check_standard` – boolean (default: `False`); check if either of the resulting tableaux is a standard super tableau, and if so, typecast it as such
- `check` – boolean (default: `True`); whether to check that `obj1` and `obj2` actually define a valid restricted super biword

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleSuperRSK
sage: p, q = RuleSuperRSK().forward_rule([1, 2], [1, 3]); p
[[1, 3]]
sage: q
[[1, 2]]
sage: isinstance(p, SemistandardSuperTableau)
```

(continues on next page)

(continued from previous page)

```
True
sage: isinstance(q, SemistandardSuperTableau)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleSuperRSK
>>> p, q = RuleSuperRSK().forward_rule([Integer(1), Integer(2)], [Integer(1),␣
↪Integer(3)]); p
[[1, 3]]
>>> q
[[1, 2]]
>>> isinstance(p, SemistandardSuperTableau)
True
>>> isinstance(q, SemistandardSuperTableau)
True
```

insertion(*j*, *r*, *epsilon=0*)

Insert the letter `j` from the second row of the biword into the row `r` using dual RSK insertion or classical Schensted insertion depending on the value of `epsilon`, if there is bumping to be done.

The row r is modified in place if bumping occurs. The bumped-out entry, if it exists, is returned.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleSuperRSK
sage: from bisect import bisect_left, bisect_right
sage: r = [1, 3, 3, 3, 4]
sage: j = 3
sage: j, y_pos = RuleSuperRSK().insertion(j, r, epsilon=0); r
[1, 3, 3, 3, 3]
sage: j
4
sage: y_pos
4
sage: r = [1, 3, 3, 3, 4]
sage: j = 3
sage: j, y_pos = RuleSuperRSK().insertion(j, r, epsilon=1); r
[1, 3, 3, 3, 4]
sage: j
3
sage: y_pos
1
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleSuperRSK
>>> from bisect import bisect_left, bisect_right
>>> r = [Integer(1), Integer(3), Integer(3), Integer(3), Integer(4)]
>>> j = Integer(3)
>>> j, y_pos = RuleSuperRSK().insertion(j, r, epsilon=Integer(0)); r
[1, 3, 3, 3, 3]
>>> j
4
```

(continues on next page)

(continued from previous page)

```
>>> y_pos
4
>>> r = [Integer(1), Integer(3), Integer(3), Integer(3), Integer(4)]
>>> j = Integer(3)
>>> j, y_pos = RuleSuperRSK().insertion(j, r, epsilon=Integer(1)); r
[1, 3, 3, 3, 4]
>>> j
3
>>> y_pos
1
```

reverse_insertion(*x*, *row*, *epsilon=0*)

Reverse bump the row `row` of the current insertion tableau with the number `x` using dual RSK insertion or classical Schensted insertion depending on the value of *epsilon*.

The row `row` is modified in place. The bumped-out entry is returned along with the bumped position.

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleSuperRSK
sage: from bisect import bisect_left, bisect_right
sage: r = [1, 3, 3, 3, 4]
sage: j = 2
sage: j, y = RuleSuperRSK().reverse_insertion(j, r, epsilon=0); r
[2, 3, 3, 3, 4]
sage: j
1
sage: y
0
sage: r = [1, 3, 3, 3, 4]
sage: j = 3
sage: j, y = RuleSuperRSK().reverse_insertion(j, r, epsilon=0); r
[3, 3, 3, 3, 4]
sage: j
1
sage: y
0
sage: r = [1, 3, 3, 3, 4]
sage: j = (3)
sage: j, y = RuleSuperRSK().reverse_insertion(j, r, epsilon=1); r
[1, 3, 3, 3, 4]
sage: j
3
sage: y
3
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleSuperRSK
>>> from bisect import bisect_left, bisect_right
>>> r = [Integer(1), Integer(3), Integer(3), Integer(3), Integer(4)]
>>> j = Integer(2)
>>> j, y = RuleSuperRSK().reverse_insertion(j, r, epsilon=Integer(0)); r
[2, 3, 3, 3, 4]
```

(continues on next page)

(continued from previous page)

```
>>> j
1
>>> y
0
>>> r = [Integer(1), Integer(3), Integer(3), Integer(3), Integer(4)]
>>> j = Integer(3)
>>> j, y = RuleSuperRSK().reverse_insertion(j, r, epsilon=Integer(0)); r
[3, 3, 3, 3, 4]
>>> j
1
>>> y
0
>>> r = [Integer(1), Integer(3), Integer(3), Integer(3), Integer(4)]
>>> j = (Integer(3))
>>> j, y = RuleSuperRSK().reverse_insertion(j, r, epsilon=Integer(1)); r
[1, 3, 3, 3, 4]
>>> j
3
>>> y
3
```

to_pairs(*obj1=None*, *obj2=None*, *check=True*)

Given a valid input for the super RSK algorithm, such as two n-tuples `obj1` $= [a_1, a_2, \ldots, a_n]$ and `obj2` $= [b_1, b_2, \ldots, b_n]$ forming a restricted super biword (i.e., entries with even and odd parity and no repetition of corresponding pairs with mixed parity entries) return the array $[(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]$.

INPUT:

- `obj1`, `obj2` – anything representing a restricted super biword (see the doc of *forward_rule()* for the encodings accepted)
- `check` – boolean (default: `True`); whether to check that `obj1` and `obj2` actually define a valid restricted super biword

EXAMPLES:

```
sage: from sage.combinat.rsk import RuleSuperRSK
sage: list(RuleSuperRSK().to_pairs([2, '1p', 1],[1, 1, '1p']))
[(2, 1), (1', 1), (1, 1')]
sage: list(RuleSuperRSK().to_pairs([1, '1p', '2p']))
[(1', 1), (1, 1'), (2', 2')]
sage: list(RuleSuperRSK().to_pairs([1, 1], ['1p', '1p']))
Traceback (most recent call last):
...
ValueError: invalid restricted superbiword
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import RuleSuperRSK
>>> list(RuleSuperRSK().to_pairs([Integer(2), '1p', Integer(1)],[Integer(1),␣
↪Integer(1), '1p']))
[(2, 1), (1', 1), (1, 1')]
>>> list(RuleSuperRSK().to_pairs([Integer(1), '1p', '2p']))
[(1', 1), (1, 1'), (2', 2')]
>>> list(RuleSuperRSK().to_pairs([Integer(1), Integer(1)], ['1p', '1p']))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: invalid restricted superbiword
```

sage.combinat.rsk.**robinson_schensted_knuth**(*obj1=None*, *obj2=None*, *insertion=<class 'sage.combinat.rsk.RuleRSK'>*, *check_standard=False*, ***options*)

Perform the Robinson-Schensted-Knuth (RSK) correspondence.

The Robinson-Schensted-Knuth (RSK) correspondence (also known as the RSK algorithm) is most naturally stated as a bijection between generalized permutations (also known as two-line arrays, biwords, …) and pairs of semi-standard Young tableaux (P, Q) of identical shape. The tableau P is known as the insertion tableau, and Q is known as the recording tableau.

The basic operation is known as row insertion $P \leftarrow k$ (where P is a given semi-standard Young tableau, and k is an integer). Row insertion is a recursive algorithm which starts by setting $k_0 = k$, and in its i-th step inserts the number k_i into the i-th row of P (we start counting the rows at 0) by replacing the first integer greater than k_i in the row by k_i and defines k_{i+1} as the integer that has been replaced. If no integer greater than k_i exists in the i-th row, then k_i is simply appended to the row and the algorithm terminates at this point.

A *generalized permutation* (or *biword*) is a list $((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$ of pairs such that the letters $j_0, j_1, \ldots, j_{\ell-1}$ are weakly increasing (that is, $j_0 \leq j_1 \leq \cdots \leq j_{\ell-1}$), whereas the letters k_i satisfy $k_i \leq k_{i+1}$ whenever $j_i = j_{i+1}$. The ℓ-tuple $(j_0, j_1, \ldots, j_{\ell-1})$ is called the *top line* of this generalized permutation, whereas the ℓ-tuple $(k_0, k_1, \ldots, k_{\ell-1})$ is called its *bottom line*.

Now the RSK algorithm, applied to a generalized permutation $p = ((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$ (encoded as a lexicographically sorted list of pairs) starts by initializing two semi-standard tableaux P_0 and Q_0 as empty tableaux. For each nonnegative integer t starting at 0, take the pair (j_t, k_t) from p and set $P_{t+1} = P_t \leftarrow k_t$, and define Q_{t+1} by adding a new box filled with j_t to the tableau Q_t at the same location the row insertion on P_t ended (that is to say, adding a new box with entry j_t such that P_{t+1} and Q_{t+1} have the same shape). The iterative process stops when t reaches the size of p, and the pair (P_t, Q_t) at this point is the image of p under the Robinson-Schensted-Knuth correspondence.

This correspondence has been introduced in [Knu1970], where it has been referred to as "Construction A".

For more information, see Chapter 7 in [Sta-EC2].

We also note that integer matrices are in bijection with generalized permutations. Furthermore, we can convert any word w (and, in particular, any permutation) to a generalized permutation by considering the top row to be $(1, 2, \ldots, n)$ where n is the length of w.

The optional argument `insertion` allows to specify an alternative insertion procedure to be used instead of the standard Robinson-Schensted-Knuth insertion.

INPUT:

- `obj1, obj2` – can be one of the following:
  - a word in an ordered alphabet (in this case, `obj1` is said word, and `obj2` is `None`)
  - an integer matrix
  - two lists of equal length representing a generalized permutation (namely, the lists $(j_0, j_1, \ldots, j_{\ell-1})$ and $(k_0, k_1, \ldots, k_{\ell-1})$ represent the generalized permutation $((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$)
  - any object which has a method `_rsk_iter()` which returns an iterator over the object represented as generalized permutation or a pair of lists (in this case, `obj1` is said object, and `obj2` is `None`).
- `insertion` – (default: `RSK.rules.RSK`) the following types of insertion are currently supported:

  - RSK.rules.RSK (or 'RSK') – Robinson-Schensted-Knuth insertion (*RuleRSK*)
  - RSK.rules.EG (or 'EG') – Edelman-Greene insertion (only for reduced words of permutations/elements of a type A Coxeter group) (*RuleEG*)
  - RSK.rules.Hecke – (or 'hecke') Hecke insertion (only guaranteed for generalized permutations whose top row is strictly increasing) (*RuleHecke*)
  - RSK.rules.dualRSK (or 'dualRSK') – Dual RSK insertion (only for strict biwords) (*RuleDualRSK*)
  - RSK.rules.coRSK (or 'coRSK') – CoRSK insertion (only for strict cobiwords) (*RuleCoRSK*)
  - RSK.rules.superRSK (or 'super') – Super RSK insertion (only for restricted super biwords) (*RuleSuperRSK*)
  - RSK.rules.Star (or 'Star') – $\star$-insertion (only for fully commutative words in the 0-Hecke monoid) (*RuleStar*)
- check_standard – boolean (default: False); check if either of the resulting tableaux is a standard tableau, and if so, typecast it as such

For precise information about constraints on the input and output, as well as the definition of the algorithm (if it is not standard RSK), see the particular *Rule* class.

EXAMPLES:

If we only input one row, it is understood that the top row should be $(1, 2, \ldots, n)$:

```
sage: RSK([3,3,2,4,1])
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RSK(Word([3,3,2,4,1]))
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RSK(Word([2,3,3,2,1,3,2,3]))
[[[1, 2, 2, 3, 3], [2, 3], [3]], [[1, 2, 3, 6, 8], [4, 7], [5]]]
```

```
>>> from sage.all import *
>>> RSK([Integer(3),Integer(3),Integer(2),Integer(4),Integer(1)])
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
>>> RSK(Word([Integer(3),Integer(3),Integer(2),Integer(4),Integer(1)]))
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
>>> RSK(Word([Integer(2),Integer(3),Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3)]))
[[[1, 2, 2, 3, 3], [2, 3], [3]], [[1, 2, 3, 6, 8], [4, 7], [5]]]
```

We can provide a generalized permutation:

```
sage: RSK([1, 2, 2, 2], [2, 1, 1, 2])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
sage: RSK(Word([1,1,3,4,4]), [1,4,2,1,3])
[[[1, 1, 3], [2], [4]], [[1, 1, 4], [3], [4]]]
sage: RSK([1,3,3,4,4], Word([6,2,2,1,7]))
[[[1, 2, 7], [2], [6]], [[1, 3, 4], [3], [4]]]
```

```
>>> from sage.all import *
>>> RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2), Integer(1),
↪ Integer(1), Integer(2)])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
>>> RSK(Word([Integer(1),Integer(1),Integer(3),Integer(4),Integer(4)]),␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(1),Integer(4),Integer(2),Integer(1),Integer(3)])
[[[1, 1, 3], [2], [4]], [[1, 1, 4], [3], [4]]]
>>> RSK([Integer(1),Integer(3),Integer(3),Integer(4),Integer(4)],␣
↪Word([Integer(6),Integer(2),Integer(2),Integer(1),Integer(7)]))
[[[1, 2, 7], [2], [6]], [[1, 3, 4], [3], [4]]]
```

We can provide a matrix:

```
sage: RSK(matrix([[0,1],[2,1]]))
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

```
>>> from sage.all import *
>>> RSK(matrix([[Integer(0),Integer(1)],[Integer(2),Integer(1)]]))
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

We can also provide something looking like a matrix:

```
sage: RSK([[0,1],[2,1]])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

```
>>> from sage.all import *
>>> RSK([[Integer(0),Integer(1)],[Integer(2),Integer(1)]])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

There is also `RSK_inverse()` which performs the inverse of the bijection on a pair of semistandard tableaux. We note that the inverse function takes 2 separate tableaux as inputs, so to compose with `RSK()`, we need to use the python `*` on the output:

```
sage: RSK_inverse(*RSK([1, 2, 2, 2], [2, 1, 1, 2]))
[[1, 2, 2, 2], [2, 1, 1, 2]]
sage: P,Q = RSK([1, 2, 2, 2], [2, 1, 1, 2])
sage: RSK_inverse(P, Q)
[[1, 2, 2, 2], [2, 1, 1, 2]]
```

```
>>> from sage.all import *
>>> RSK_inverse(*RSK([Integer(1), Integer(2), Integer(2), Integer(2)],␣
↪[Integer(2), Integer(1), Integer(1), Integer(2)]))
[[1, 2, 2, 2], [2, 1, 1, 2]]
>>> P,Q = RSK([Integer(1), Integer(2), Integer(2), Integer(2)], [Integer(2),␣
↪Integer(1), Integer(1), Integer(2)])
>>> RSK_inverse(P, Q)
[[1, 2, 2, 2], [2, 1, 1, 2]]
```

`sage.combinat.rsk.`**`robinson_schensted_knuth_inverse`**(*p*, *q*, *output='array'*, *insertion=<class 'sage.combinat.rsk.RuleRSK'>*)

Return the generalized permutation corresponding to the pair of tableaux (p, q) under the inverse of the Robinson-Schensted-Knuth correspondence.

For more information on the bijection, see `RSK()`.

INPUT:

- `p`, `q` – two semi-standard tableaux of the same shape, or (in the case when Hecke insertion is used) an increasing tableau and a set-valued tableau of the same shape (see the note below for the format of the set-valued tableau)

- `output` – (default: `'array'`) if `q` is semi-standard:
  - `'array'` – as a two-line array (i.e. generalized permutation or biword)
  - `'matrix'` – as an integer matrix

  and if `q` is standard, we can also have the output:
  - `'word'` – as a word

  and additionally if `p` is standard, we can also have the output:
  - `'permutation'` – as a permutation
- `insertion` – (default: `RSK.rules.RSK`) the insertion algorithm used in the bijection. Currently the following are supported:
  - `RSK.rules.RSK` (or `'RSK'`) – Robinson-Schensted-Knuth insertion (*RuleRSK*)
  - `RSK.rules.EG` (or `'EG'`) – Edelman-Greene insertion (only for reduced words of permutations/elements of a type A Coxeter group) (*RuleEG*)
  - `RSK.rules.Hecke` (or `'hecke'`) – Hecke insertion (only guaranteed for generalized permutations whose top row is strictly increasing) (*RuleHecke*)
  - `RSK.rules.dualRSK` (or `'dualRSK'`) – Dual RSK insertion (only for strict biwords) (*RuleDualRSK*)
  - `RSK.rules.coRSK` (or `'coRSK'`) – CoRSK insertion (only for strict cobiwords) (*RuleCoRSK*)
  - `RSK.rules.superRSK` (or `'super'`) – Super RSK insertion (only for restricted super biwords) (*RuleSuperRSK*)
  - `RSK.rules.Star` (or `'Star'`) – $\star$-insertion (only for fully commutative words in the 0-Hecke monoid) (*RuleStar*)

For precise information about constraints on the input and output, see the particular *Rule* class.

Note

In the case of Hecke insertion, the input variable `q` should be a set-valued tableau, encoded as a tableau whose entries are strictly increasing tuples of positive integers. Each such tuple encodes the set of its entries.

EXAMPLES:

If both `p` and `q` are standard:

```
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: RSK_inverse(t1, t2)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: RSK_inverse(t1, t2, 'word')
word: 14532
sage: RSK_inverse(t1, t2, 'matrix')
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
sage: RSK_inverse(t1, t2, 'permutation')
```

(continues on next page)

(continued from previous page)

```
[1, 4, 5, 3, 2]
sage: RSK_inverse(t1, t1, 'permutation')
[1, 4, 3, 2, 5]
sage: RSK_inverse(t2, t2, 'permutation')
[1, 2, 5, 4, 3]
sage: RSK_inverse(t2, t1, 'permutation')
[1, 5, 4, 2, 3]
```

```
>>> from sage.all import *
>>> t1 = Tableau([[Integer(1), Integer(2), Integer(5)], [Integer(3)],
↪[Integer(4)]])
>>> t2 = Tableau([[Integer(1), Integer(2), Integer(3)], [Integer(4)],
↪[Integer(5)]])
>>> RSK_inverse(t1, t2)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
>>> RSK_inverse(t1, t2, 'word')
word: 14532
>>> RSK_inverse(t1, t2, 'matrix')
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
>>> RSK_inverse(t1, t2, 'permutation')
[1, 4, 5, 3, 2]
>>> RSK_inverse(t1, t1, 'permutation')
[1, 4, 3, 2, 5]
>>> RSK_inverse(t2, t2, 'permutation')
[1, 2, 5, 4, 3]
>>> RSK_inverse(t2, t1, 'permutation')
[1, 5, 4, 2, 3]
```

If the first tableau is semistandard:

```
sage: p = Tableau([[1,2,2],[3]]); q = Tableau([[1,2,4],[3]])
sage: ret = RSK_inverse(p, q); ret
[[1, 2, 3, 4], [1, 3, 2, 2]]
sage: RSK_inverse(p, q, 'word')
word: 1322
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(3)]]); q =
↪Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3)]])
>>> ret = RSK_inverse(p, q); ret
[[1, 2, 3, 4], [1, 3, 2, 2]]
>>> RSK_inverse(p, q, 'word')
word: 1322
```

In general:

```
sage: p = Tableau([[1,2,2],[2]]); q = Tableau([[1,3,3],[2]])
sage: RSK_inverse(p, q)
```

(continues on next page)

(continued from previous page)

```
[[1, 2, 3, 3], [2, 1, 2, 2]]
sage: RSK_inverse(p, q, 'matrix')
[0 1]
[1 0]
[0 2]
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(2)]]); q =␣
↪Tableau([[Integer(1),Integer(3),Integer(3)],[Integer(2)]])
>>> RSK_inverse(p, q)
[[1, 2, 3, 3], [2, 1, 2, 2]]
>>> RSK_inverse(p, q, 'matrix')
[0 1]
[1 0]
[0 2]
```

Using Hecke insertion:

```
sage: w = [5, 4, 3, 1, 4, 2, 5, 5]
sage: pq = RSK(w, insertion=RSK.rules.Hecke)
sage: RSK_inverse(*pq, insertion=RSK.rules.Hecke, output='list')
[5, 4, 3, 1, 4, 2, 5, 5]
```

```
>>> from sage.all import *
>>> w = [Integer(5), Integer(4), Integer(3), Integer(1), Integer(4), Integer(2),␣
↪Integer(5), Integer(5)]
>>> pq = RSK(w, insertion=RSK.rules.Hecke)
>>> RSK_inverse(*pq, insertion=RSK.rules.Hecke, output='list')
[5, 4, 3, 1, 4, 2, 5, 5]
```

Note

The constructor of `Tableau` accepts not only semistandard tableaux, but also arbitrary lists that are fillings of a partition diagram. (And such lists are used, e.g., for the set-valued tableau `q` that is passed to `RSK_inverse(p, q, insertion='hecke')`.) The user is responsible for ensuring that the tableaux passed to `RSK_inverse` are of the right types (semistandard, standard, increasing, set-valued as needed).

`sage.combinat.rsk.`**`to_matrix`**`(t, b)`

Return the integer matrix corresponding to a two-line array.

INPUT:

- `t` – the top row of the array
- `b` – the bottom row of the array

OUTPUT:

An $m \times n$-matrix (where m and n are the maximum entries in t and b respectively) whose (i, j)-th entry, for any i and j, is the number of all positions k satisfying $t_k = i$ and $b_k = j$.

EXAMPLES:

```
sage: from sage.combinat.rsk import to_matrix
sage: to_matrix([1, 1, 3, 3, 4], [2, 3, 1, 1, 3])
[0 1 1]
[0 0 0]
[2 0 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.rsk import to_matrix
>>> to_matrix([Integer(1), Integer(1), Integer(3), Integer(3), Integer(4)],
↪[Integer(2), Integer(3), Integer(1), Integer(1), Integer(3)])
[0 1 1]
[0 0 0]
[2 0 0]
[0 0 1]
```

5.1.276 Schubert Polynomials

See Wikipedia article Schubert_polynomial and SymmetricFunctions.com. Schubert polynomials are representatives of cohomology classes in flag varieties. In n variables, they are indexed by permutations $w \in S_n$. They also form a basis for the coinvariant ring of the S_n action on $\mathbf{Z}[x_1, x_2, \ldots, x_n]$.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: w = [1,2,5,4,3];  # a list representing an element of `S_5`
sage: X(w)
X[1, 2, 5, 4, 3]
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(ZZ)
>>> w = [Integer(1),Integer(2),Integer(5),Integer(4),Integer(3)];  # a list
↪representing an element of `S_5`
>>> X(w)
X[1, 2, 5, 4, 3]
```

This can be expanded in terms of polynomial variables:

```
sage: X(w).expand()
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2
 + x0*x2^2 + x1*x2^2 + x0^2*x3 + x0*x1*x3 + x1^2*x3
 + x0*x2*x3 + x1*x2*x3 + x2^2*x3
```

```
>>> from sage.all import *
>>> X(w).expand()
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2
 + x0*x2^2 + x1*x2^2 + x0^2*x3 + x0*x1*x3 + x1^2*x3
 + x0*x2*x3 + x1*x2*x3 + x2^2*x3
```

We can also convert back from polynomial variables. For example, the longest permutation is a single term. In S_5, this is the element (in one line notation) $w_0 = 54321$:

```
sage: w0 = [5,4,3,2,1]
sage: R.<x0, x1, x2, x3, x4> = PolynomialRing(ZZ)
sage: Sw0 = X(x0^4*x1^3*x2^2*x3);  Sw0
X[5, 4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> w0 = [Integer(5),Integer(4),Integer(3),Integer(2),Integer(1)]
>>> R = PolynomialRing(ZZ, names=('x0', 'x1', 'x2', 'x3', 'x4',)); (x0, x1, x2, x3,␣
↪x4,) = R._first_ngens(5)
>>> Sw0 = X(x0**Integer(4)*x1**Integer(3)*x2**Integer(2)*x3);  Sw0
X[5, 4, 3, 2, 1]
```

The polynomials also have the property that if the indexing permutation w is multiplied by a simple transposition $s_i = (i, i+1)$ such that the length of w is more than the length of ws_i, then the Schubert polynomial of the permutation ws_i is computed by applying the divided difference operator *divided_difference()* to the polynomial indexed by w. For example, applying the divided difference operator ∂_2 to the Schubert polynomial $\mathfrak{S}_{w_0}$:

```
sage: Sw0.divided_difference(2)
X[5, 3, 4, 2, 1]
```

```
>>> from sage.all import *
>>> Sw0.divided_difference(Integer(2))
X[5, 3, 4, 2, 1]
```

We can also check the properties listed in Wikipedia article Schubert_polynomial:

```
sage: X([1,2,3,4,5])  # the identity in one-line notation
X[1]
sage: X([1,3,2,4,5]).expand()  # the transposition swapping 2 and 3
x0 + x1
sage: X([2,4,5,3,1]).expand()
x0^2*x1^2*x2*x3 + x0^2*x1*x2^2*x3 + x0*x1^2*x2^2*x3

sage: w = [4,5,1,2,3]
sage: s = SymmetricFunctions(QQ).schur()
sage: s[3,3].expand(2)
x0^3*x1^3
sage: X(w).expand()
x0^3*x1^3
```

```
>>> from sage.all import *
>>> X([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])  # the identity in␣
↪one-line notation
X[1]
>>> X([Integer(1),Integer(3),Integer(2),Integer(4),Integer(5)]).expand()  # the␣
↪transposition swapping 2 and 3
x0 + x1
>>> X([Integer(2),Integer(4),Integer(5),Integer(3),Integer(1)]).expand()
x0^2*x1^2*x2*x3 + x0^2*x1*x2^2*x3 + x0*x1^2*x2^2*x3

>>> w = [Integer(4),Integer(5),Integer(1),Integer(2),Integer(3)]
>>> s = SymmetricFunctions(QQ).schur()
```

(continues on next page)

(continued from previous page)

```
>>> s[Integer(3),Integer(3)].expand(Integer(2))
x0^3*x1^3
>>> X(w).expand()
x0^3*x1^3
```

sage.combinat.schubert_polynomial.**SchubertPolynomialRing**(*R*)

Return the Schubert polynomial ring over `R` on the X basis.

This is the basis made of the Schubert polynomials.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ); X
Schubert polynomial ring with X basis over Integer Ring
sage: TestSuite(X).run()
sage: X(1)
X[1]
sage: X([1,2,3])*X([2,1,3])
X[2, 1]
sage: X([2,1,3])*X([2,1,3])
X[3, 1, 2]
sage: X([2,1,3])+X([3,1,2,4])
X[2, 1] + X[3, 1, 2]
sage: a = X([2,1,3])+X([3,1,2,4])
sage: a^2
X[3, 1, 2] + 2*X[4, 1, 2, 3] + X[5, 1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(ZZ); X
Schubert polynomial ring with X basis over Integer Ring
>>> TestSuite(X).run()
>>> X(Integer(1))
X[1]
>>> X([Integer(1),Integer(2),Integer(3)])*X([Integer(2),Integer(1),Integer(3)])
X[2, 1]
>>> X([Integer(2),Integer(1),Integer(3)])*X([Integer(2),Integer(1),Integer(3)])
X[3, 1, 2]
>>> X([Integer(2),Integer(1),Integer(3)])+X([Integer(3),Integer(1),Integer(2),
→Integer(4)])
X[2, 1] + X[3, 1, 2]
>>> a = X([Integer(2),Integer(1),Integer(3)])+X([Integer(3),Integer(1),Integer(2),
→Integer(4)])
>>> a**Integer(2)
X[3, 1, 2] + 2*X[4, 1, 2, 3] + X[5, 1, 2, 3, 4]
```

class sage.combinat.schubert_polynomial.**SchubertPolynomialRing_xbasis**(*R*)

Bases: `CombinatorialFreeModule`

EXAMPLES:

```
sage: X = SchubertPolynomialRing(QQ)
sage: X == loads(dumps(X))
True
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(QQ)
>>> X == loads(dumps(X))
True
```

Element

alias of *SchubertPolynomial_class*

one_basis()

Return the index of the unit of this algebra.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(QQ)
sage: X.one()   # indirect doctest
X[1]
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(QQ)
>>> X.one()   # indirect doctest
X[1]
```

product_on_basis(*left*, *right*)

EXAMPLES:

```
sage: p1 = Permutation([3,2,1])
sage: p2 = Permutation([2,1,3])
sage: X = SchubertPolynomialRing(QQ)
sage: X.product_on_basis(p1,p2)
X[4, 2, 1, 3]
```

```
>>> from sage.all import *
>>> p1 = Permutation([Integer(3),Integer(2),Integer(1)])
>>> p2 = Permutation([Integer(2),Integer(1),Integer(3)])
>>> X = SchubertPolynomialRing(QQ)
>>> X.product_on_basis(p1,p2)
X[4, 2, 1, 3]
```

some_elements()

Return some elements.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(QQ)
sage: X.some_elements()
[X[1], X[1] + 2*X[2, 1], -X[3, 2, 1] + X[4, 2, 1, 3]]
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(QQ)
>>> X.some_elements()
[X[1], X[1] + 2*X[2, 1], -X[3, 2, 1] + X[4, 2, 1, 3]]
```

class sage.combinat.schubert_polynomial.**SchubertPolynomial_class**

Bases: IndexedFreeModuleElement

divided_difference(*i*, *algorithm='sage'*)

Return the i-th divided difference operator, applied to self.

Here, i can be either a permutation or a positive integer.

INPUT:

- i – permutation or positive integer
- algorithm – (default: 'sage') either 'sage' or 'symmetrica'; this determines which software is called for the computation

OUTPUT:

The result of applying the i-th divided difference operator to self.

If i is a positive integer, then the i-th divided difference operator δ_i is the linear operator sending each polynomial $f = f(x_1, x_2, \ldots, x_n)$ (in $n \geq i + 1$ variables) to the polynomial

$$\frac{f - f_i}{x_i - x_{i+1}}, \qquad \text{where } f_i = f(x_1, x_2, ..., x_{i-1}, x_{i+1}, x_i, x_{i+1}, ..., x_n).$$

If σ is a permutation in the n-th symmetric group, then the σ-th divided difference operator δ_σ is the composition $\delta_{i_1} \delta_{i_2} \cdots \delta_{i_k}$, where $\sigma = s_{i_1} \circ s_{i_2} \circ \cdots \circ s_{i_k}$ is any reduced expression for σ (the precise choice of reduced expression is immaterial).

Note: The *expand()* method results in a polynomial in n variables named x0, x1, ..., x(n-1) rather than $x_1, x_2, \ldots, x_n$. The variable named xi corresponds to x_{i+1}. Thus, self.divided_difference(i) involves the variables x(i-1) and xi getting switched (in the numerator).

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,1])
sage: a.divided_difference(1)
X[2, 3, 1]
sage: a.divided_difference([3,2,1])
X[1]
sage: a.divided_difference(5)
0
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(ZZ)
>>> a = X([Integer(3),Integer(2),Integer(1)])
>>> a.divided_difference(Integer(1))
X[2, 3, 1]
>>> a.divided_difference([Integer(3),Integer(2),Integer(1)])
X[1]
>>> a.divided_difference(Integer(5))
0
```

Any divided difference of 0 is 0:

```
sage: X.zero().divided_difference(2)
0
```

```
>>> from sage.all import *
>>> X.zero().divided_difference(Integer(2))
0
```

This is compatible when a permutation is given as input:

```
sage: a = X([3,2,4,1])
sage: a.divided_difference([2,3,1])
0
sage: a.divided_difference(1).divided_difference(2)
0
```

```
>>> from sage.all import *
>>> a = X([Integer(3),Integer(2),Integer(4),Integer(1)])
>>> a.divided_difference([Integer(2),Integer(3),Integer(1)])
0
>>> a.divided_difference(Integer(1)).divided_difference(Integer(2))
0
```

```
sage: a = X([4,3,2,1])
sage: a.divided_difference([2,3,1])
X[3, 2, 4, 1]
sage: a.divided_difference(1).divided_difference(2)
X[3, 2, 4, 1]
sage: a.divided_difference([4,1,3,2])
X[1, 4, 2, 3]
sage: b = X([4, 1, 3, 2])
sage: b.divided_difference(1).divided_difference(2)
X[1, 3, 4, 2]
sage: b.divided_difference(1).divided_difference(2).divided_difference(3)
X[1, 3, 2]
sage: b.divided_difference(1).divided_difference(2).divided_difference(3).
↪divided_difference(2)
X[1]
sage: b.divided_difference(1).divided_difference(2).divided_difference(3).
↪divided_difference(3)
0
sage: b.divided_difference(1).divided_difference(2).divided_difference(1)
0
```

```
>>> from sage.all import *
>>> a = X([Integer(4),Integer(3),Integer(2),Integer(1)])
>>> a.divided_difference([Integer(2),Integer(3),Integer(1)])
X[3, 2, 4, 1]
>>> a.divided_difference(Integer(1)).divided_difference(Integer(2))
X[3, 2, 4, 1]
>>> a.divided_difference([Integer(4),Integer(1),Integer(3),Integer(2)])
X[1, 4, 2, 3]
>>> b = X([Integer(4), Integer(1), Integer(3), Integer(2)])
>>> b.divided_difference(Integer(1)).divided_difference(Integer(2))
X[1, 3, 4, 2]
>>> b.divided_difference(Integer(1)).divided_difference(Integer(2)).divided_
```

(continues on next page)

(continued from previous page)

```
↪difference(Integer(3))
X[1, 3, 2]
>>> b.divided_difference(Integer(1)).divided_difference(Integer(2)).divided_
↪difference(Integer(3)).divided_difference(Integer(2))
X[1]
>>> b.divided_difference(Integer(1)).divided_difference(Integer(2)).divided_
↪difference(Integer(3)).divided_difference(Integer(3))
0
>>> b.divided_difference(Integer(1)).divided_difference(Integer(2)).divided_
↪difference(Integer(1))
0
```

expand()

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: X([2,1,3]).expand()
x0
sage: [X(p).expand() for p in Permutations(3)]
[1, x0 + x1, x0, x0*x1, x0^2, x0^2*x1]
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(ZZ)
>>> X([Integer(2),Integer(1),Integer(3)]).expand()
x0
>>> [X(p).expand() for p in Permutations(Integer(3))]
[1, x0 + x1, x0, x0*x1, x0^2, x0^2*x1]
```

multiply_variable(i)

Return the Schubert polynomial obtained by multiplying `self` by the variable x_i.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,4,1])
sage: a.multiply_variable(0)
X[4, 2, 3, 1]
sage: a.multiply_variable(1)
X[3, 4, 2, 1]
sage: a.multiply_variable(2)
X[3, 2, 5, 1, 4] - X[3, 4, 2, 1] - X[4, 2, 3, 1]
sage: a.multiply_variable(3)
X[3, 2, 4, 5, 1]
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(ZZ)
>>> a = X([Integer(3),Integer(2),Integer(4),Integer(1)])
>>> a.multiply_variable(Integer(0))
X[4, 2, 3, 1]
>>> a.multiply_variable(Integer(1))
X[3, 4, 2, 1]
>>> a.multiply_variable(Integer(2))
```

(continues on next page)

(continued from previous page)

```
X[3, 2, 5, 1, 4] - X[3, 4, 2, 1] - X[4, 2, 3, 1]
>>> a.multiply_variable(Integer(3))
X[3, 2, 4, 5, 1]
```

scalar_product(*x*)

Return the standard scalar product of `self` and `x`.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,4,1])
sage: a.scalar_product(a)
0
sage: b = X([4,3,2,1])
sage: b.scalar_product(a)
X[1, 3, 4, 6, 2, 5]
sage: Permutation([1, 3, 4, 6, 2, 5, 7]).to_lehmer_code()
[0, 1, 1, 2, 0, 0, 0]
sage: s = SymmetricFunctions(ZZ).schur()
sage: c = s([2,1,1])
sage: b.scalar_product(a).expand()
x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2 + x0^2*x1*x3 + x0*x1^2*x3
 + x0^2*x2*x3 + 3*x0*x1*x2*x3 + x1^2*x2*x3 + x0*x2^2*x3 + x1*x2^2*x3
 + x0*x1*x3^2 + x0*x2*x3^2 + x1*x2*x3^2
sage: c.expand(4)
x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2 + x0^2*x1*x3 + x0*x1^2*x3
 + x0^2*x2*x3 + 3*x0*x1*x2*x3 + x1^2*x2*x3 + x0*x2^2*x3 + x1*x2^2*x3
 + x0*x1*x3^2 + x0*x2*x3^2 + x1*x2*x3^2
```

```
>>> from sage.all import *
>>> X = SchubertPolynomialRing(ZZ)
>>> a = X([Integer(3),Integer(2),Integer(4),Integer(1)])
>>> a.scalar_product(a)
0
>>> b = X([Integer(4),Integer(3),Integer(2),Integer(1)])
>>> b.scalar_product(a)
X[1, 3, 4, 6, 2, 5]
>>> Permutation([Integer(1), Integer(3), Integer(4), Integer(6), Integer(2),
↪Integer(5), Integer(7)]).to_lehmer_code()
[0, 1, 1, 2, 0, 0, 0]
>>> s = SymmetricFunctions(ZZ).schur()
>>> c = s([Integer(2),Integer(1),Integer(1)])
>>> b.scalar_product(a).expand()
x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2 + x0^2*x1*x3 + x0*x1^2*x3
 + x0^2*x2*x3 + 3*x0*x1*x2*x3 + x1^2*x2*x3 + x0*x2^2*x3 + x1*x2^2*x3
 + x0*x1*x3^2 + x0*x2*x3^2 + x1*x2*x3^2
>>> c.expand(Integer(4))
x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2 + x0^2*x1*x3 + x0*x1^2*x3
 + x0^2*x2*x3 + 3*x0*x1*x2*x3 + x1^2*x2*x3 + x0*x2^2*x3 + x1*x2^2*x3
 + x0*x1*x3^2 + x0*x2*x3^2 + x1*x2*x3^2
```

5.1.277 Set Partitions

AUTHORS:

- Mike Hansen
- MuPAD-Combinat developers (for algorithms and design inspiration).
- Travis Scrimshaw (2013-02-28): Removed `CombinatorialClass` and added entry point through *SetPartition*.
- Martin Rubey (2017-10-10): Cleanup, add crossings and nestings, add random generation.

This module defines a class for immutable partitioning of a set. For mutable version see `DisjointSet()`.

class sage.combinat.set_partition.**AbstractSetPartition**

Bases: `ClonableArray`

Methods of set partitions which are independent of the base set

base_set()

Return the base set of `self`, which is the union of all parts of `self`.

EXAMPLES:

```
sage: SetPartition([[1], [2,3], [4]]).base_set()
{1, 2, 3, 4}
sage: SetPartition([[1,2,3,4]]).base_set()
{1, 2, 3, 4}
sage: SetPartition([]).base_set()
{}
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(1)], [Integer(2),Integer(3)], [Integer(4)]]).base_
↪set()
{1, 2, 3, 4}
>>> SetPartition([[Integer(1),Integer(2),Integer(3),Integer(4)]]).base_set()
{1, 2, 3, 4}
>>> SetPartition([]).base_set()
{}
```

base_set_cardinality()

Return the cardinality of the base set of `self`, which is the sum of the sizes of the parts of `self`.

This is also known as the *size* (sometimes the *weight*) of a set partition.

EXAMPLES:

```
sage: SetPartition([[1], [2,3], [4]]).base_set_cardinality()
4
sage: SetPartition([[1,2,3,4]]).base_set_cardinality()
4
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(1)], [Integer(2),Integer(3)], [Integer(4)]]).base_
↪set_cardinality()
4
>>> SetPartition([[Integer(1),Integer(2),Integer(3),Integer(4)]]).base_set_
```

(continues on next page)

(continued from previous page)

```
↪cardinality()
4
```

coarsenings()

Return a list of coarsenings of `self`.

See also:

`refinements()`

EXAMPLES:

```
sage: SetPartition([[1,3],[2,4]]).coarsenings()
[{{1, 2, 3, 4}}, {{1, 3}, {2, 4}}]
sage: SetPartition([[1],[2,4],[3]]).coarsenings()
[{{1, 2, 3, 4}},
 {{1, 2, 4}, {3}},
 {{1, 3}, {2, 4}},
 {{1}, {2, 3, 4}},
 {{1}, {2, 4}, {3}}]
sage: SetPartition([]).coarsenings()
[{}]
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(1),Integer(3)],[Integer(2),Integer(4)]]).
↪coarsenings()
[{{1, 2, 3, 4}}, {{1, 3}, {2, 4}}]
>>> SetPartition([[Integer(1)],[Integer(2),Integer(4)],[Integer(3)]]).
↪coarsenings()
[{{1, 2, 3, 4}},
 {{1, 2, 4}, {3}},
 {{1, 3}, {2, 4}},
 {{1}, {2, 3, 4}},
 {{1}, {2, 4}, {3}}]
>>> SetPartition([]).coarsenings()
[{}]
```

conjugate()

An involution exchanging singletons and circular adjacencies.

This method implements the definition of the conjugate of a set partition defined in [Cal2005].

INPUT:

- `self` – set partition of an ordered set

OUTPUT: a set partition

EXAMPLES:

```
sage: SetPartition([[1,6,7],[2,8],[3,4,5]]).conjugate()
{{1, 4, 7}, {2, 8}, {3}, {5}, {6}}
sage: all(sp.conjugate().conjugate()==sp for sp in SetPartitions([1,3,5,7]))
```

(continues on next page)

(continued from previous page)

```
True
sage: SetPartition([]).conjugate()
{}
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(1),Integer(6),Integer(7)],[Integer(2),Integer(8)],
↪[Integer(3),Integer(4),Integer(5)]]).conjugate()
{{1, 4, 7}, {2, 8}, {3}, {5}, {6}}
>>> all(sp.conjugate().conjugate()==sp for sp in SetPartitions([Integer(1),
↪Integer(3),Integer(5),Integer(7)]))
True
>>> SetPartition([]).conjugate()
{}
```

inf(*other*)

The product of the set partitions `self` and `other`.

The product of two set partitions B and C is defined as the set partition whose parts are the nonempty intersections between each part of B and each part of C. This product is also the infimum of B and C in the classical set partition lattice (that is, the coarsest set partition which is finer than each of B and C). Consequently, `inf` acts as an alias for this method.

See also

sup()

EXAMPLES:

```
sage: x = SetPartition([ [1,2], [3,5,4] ])
sage: y = SetPartition(( (3,1,2), (5,4) ))
sage: x * y
{{1, 2}, {3}, {4, 5}}

sage: S = SetPartitions(4)
sage: sp1 = S([[2,3,4], [1]])
sage: sp2 = S([[1,3], [2,4]])
sage: s = S([[2,4], [3], [1]])
sage: sp1.inf(sp2) == s
True
```

```
>>> from sage.all import *
>>> x = SetPartition([ [Integer(1),Integer(2)], [Integer(3),Integer(5),
↪Integer(4)] ])
>>> y = SetPartition(( (Integer(3),Integer(1),Integer(2)), (Integer(5),
↪Integer(4)) ))
>>> x * y
{{1, 2}, {3}, {4, 5}}

>>> S = SetPartitions(Integer(4))
>>> sp1 = S([[Integer(2),Integer(3),Integer(4)], [Integer(1)]])
>>> sp2 = S([[Integer(1),Integer(3)], [Integer(2),Integer(4)]])
```

(continues on next page)

(continued from previous page)

```
>>> s = S([[Integer(2),Integer(4)], [Integer(3)], [Integer(1)]])
>>> sp1.inf(sp2) == s
True
```

max_block_size()

The maximum block size of the diagram.

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.combinat.diagram_algebras import PartitionDiagram,␣
↪PartitionDiagrams
sage: pd = PartitionDiagram([[1,-3,-5],[2,4],[3,-1,-2],[5],[-4]])
sage: pd.max_block_size()
3
sage: sorted(d.max_block_size() for d in PartitionDiagrams(2))
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4]
sage: sorted(sp.max_block_size() for sp in SetPartitions(3))
[1, 2, 2, 2, 3]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.combinat.diagram_algebras import PartitionDiagram,␣
↪PartitionDiagrams
>>> pd = PartitionDiagram([[Integer(1),-Integer(3),-Integer(5)],[Integer(2),
↪Integer(4)],[Integer(3),-Integer(1),-Integer(2)],[Integer(5)],[-
↪Integer(4)]])
>>> pd.max_block_size()
3
>>> sorted(d.max_block_size() for d in PartitionDiagrams(Integer(2)))
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4]
>>> sorted(sp.max_block_size() for sp in SetPartitions(Integer(3)))
[1, 2, 2, 2, 3]
```

standard_form()

Return `self` as a list of lists.

When the ground set is totally ordered, the elements of each block are listed in increasing order.

This is not related to standard set partitions (which simply means set partitions of $[n] = \{1, 2, \ldots, n\}$ for some integer n) or standardization (`standardization()`).

EXAMPLES:

```
sage: [x.standard_form() for x in SetPartitions(4, [2,2])]                #␣
↪needs sage.graphs sage.rings.finite_rings
[[[1, 2], [3, 4]], [[1, 4], [2, 3]], [[1, 3], [2, 4]]]
```

```
>>> from sage.all import *
>>> [x.standard_form() for x in SetPartitions(Integer(4), [Integer(2),
↪Integer(2)])]                   # needs sage.graphs sage.rings.finite_rings
[[[1, 2], [3, 4]], [[1, 4], [2, 3]], [[1, 3], [2, 4]]]
```

sup(*t*)

Return the supremum of `self` and `t` in the classical set partition lattice.

The supremum of two set partitions B and C is obtained as the transitive closure of the relation which relates i to j if and only if i and j are in the same part in at least one of the set partitions B and C.

See also

`__mul__()`

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: sp1 = S([[2,3,4], [1]])
sage: sp2 = S([[1,3], [2,4]])
sage: s = S([[1,2,3,4]])
sage: sp1.sup(sp2) == s
True
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(4))
>>> sp1 = S([[Integer(2),Integer(3),Integer(4)], [Integer(1)]])
>>> sp2 = S([[Integer(1),Integer(3)], [Integer(2),Integer(4)]])
>>> s = S([[Integer(1),Integer(2),Integer(3),Integer(4)]])
>>> sp1.sup(sp2) == s
True
```

class sage.combinat.set_partition.**SetPartition**(*parent*, *s*, *check=True*)

Bases: *AbstractSetPartition*

A partition of a set.

A set partition p of a set S is a partition of S into subsets called parts and represented as a set of sets. By extension, a set partition of a nonnegative integer n is the set partition of the integers from 1 to n. The number of set partitions of n is called the n-th Bell number.

There is a natural integer partition associated with a set partition, namely the nonincreasing sequence of sizes of all its parts.

There is a classical lattice associated with all set partitions of n. The infimum of two set partitions is the set partition obtained by intersecting all the parts of both set partitions. The supremum is obtained by transitive closure of the relation i related to j if and only if they are in the same part in at least one of the set partitions.

We will use terminology from partitions, in particular the *length* of a set partition $A = \{A_1, \ldots, A_k\}$ is the number of parts of A and is denoted by $|A| := k$. The *size* of A is the cardinality of S. We will also sometimes use the notation $[n] := \{1, 2, \ldots, n\}$.

EXAMPLES:

There are 5 set partitions of the set $\{1, 2, 3\}$:

```
sage: SetPartitions(3).cardinality()                                    #␣
↪needs sage.libs.flint
5
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(3)).cardinality()                                    ␣
↪      # needs sage.libs.flint
5
```

Here is the list of them:

```
sage: SetPartitions(3).list()                                                 #␣
↪needs sage.graphs
[{{1, 2, 3}}, {{1, 2}, {3}}, {{1, 3}, {2}}, {{1}, {2, 3}}, {{1}, {2}, {3}}]
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(3)).list()                                           ␣
↪      # needs sage.graphs
[{{1, 2, 3}}, {{1, 2}, {3}}, {{1, 3}, {2}}, {{1}, {2, 3}}, {{1}, {2}, {3}}]
```

There are 6 set partitions of $\{1, 2, 3, 4\}$ whose underlying partition is $[2, 1, 1]$:

```
sage: SetPartitions(4, [2,1,1]).list()                                        #␣
↪needs sage.graphs sage.rings.finite_rings
[{{1}, {2, 4}, {3}},
 {{1}, {2}, {3, 4}},
 {{1, 4}, {2}, {3}},
 {{1, 3}, {2}, {4}},
 {{1, 2}, {3}, {4}},
 {{1}, {2, 3}, {4}}]
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(4), [Integer(2),Integer(1),Integer(1)]).list()       ␣
↪                               # needs sage.graphs sage.rings.finite_rings
[{{1}, {2, 4}, {3}},
 {{1}, {2}, {3, 4}},
 {{1, 4}, {2}, {3}},
 {{1, 3}, {2}, {4}},
 {{1, 2}, {3}, {4}},
 {{1}, {2, 3}, {4}}]
```

Since Issue #14140, we can create a set partition directly by `SetPartition`, which creates the base set by taking the union of the parts passed in:

```
sage: s = SetPartition([[1,3],[2,4]]); s
{{1, 3}, {2, 4}}
sage: s.parent()
Set partitions
```

```
>>> from sage.all import *
>>> s = SetPartition([[Integer(1),Integer(3)],[Integer(2),Integer(4)]]); s
{{1, 3}, {2, 4}}
>>> s.parent()
Set partitions
```

apply_permutation(*p*)

Apply `p` to the underlying set of `self`.

INPUT:

- `p` – a permutation

EXAMPLES:

```
sage: x = SetPartition([[1,2], [3,5,4]])
sage: p = Permutation([2,1,4,5,3])
sage: x.apply_permutation(p)
{{1, 2}, {3, 4, 5}}
sage: q = Permutation([3,2,1,5,4])
sage: x.apply_permutation(q)
{{1, 4, 5}, {2, 3}}

sage: m = PerfectMatching([(1,4),(2,6),(3,5)])
sage: m.apply_permutation(Permutation([4,1,5,6,3,2]))
[(1, 2), (3, 5), (4, 6)]
```

```
>>> from sage.all import *
>>> x = SetPartition([[Integer(1),Integer(2)], [Integer(3),Integer(5),
↪Integer(4)]])
>>> p = Permutation([Integer(2),Integer(1),Integer(4),Integer(5),Integer(3)])
>>> x.apply_permutation(p)
{{1, 2}, {3, 4, 5}}
>>> q = Permutation([Integer(3),Integer(2),Integer(1),Integer(5),Integer(4)])
>>> x.apply_permutation(q)
{{1, 4, 5}, {2, 3}}

>>> m = PerfectMatching([(Integer(1),Integer(4)),(Integer(2),Integer(6)),
↪(Integer(3),Integer(5))])
>>> m.apply_permutation(Permutation([Integer(4),Integer(1),Integer(5),
↪Integer(6),Integer(3),Integer(2)]))
[(1, 2), (3, 5), (4, 6)]
```

arcs()

Return `self` as a list of arcs.

Assuming that the blocks are sorted, the arcs are the pairs of consecutive elements in the blocks.

EXAMPLES:

```
sage: A = SetPartition([[1],[2,3],[4]])
sage: A.arcs()
[(2, 3)]
sage: B = SetPartition([[1,3,6,7],[2,5],[4]])
sage: B.arcs()
[(1, 3), (3, 6), (6, 7), (2, 5)]
```

```
>>> from sage.all import *
>>> A = SetPartition([[Integer(1)],[Integer(2),Integer(3)],[Integer(4)]])
>>> A.arcs()
[(2, 3)]
>>> B = SetPartition([[Integer(1),Integer(3),Integer(6),Integer(7)],
↪[Integer(2),Integer(5)],[Integer(4)]])
```

(continues on next page)

(continued from previous page)

```
>>> B.arcs()
[(1, 3), (3, 6), (6, 7), (2, 5)]
```

cardinality()

Return the `len` of `self`.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: len(IncreasingArrays()([1,2,3]))
3
```

```
>>> from sage.all import *
>>> from sage.structure.list_clone_demo import IncreasingArrays
>>> len(IncreasingArrays()([Integer(1),Integer(2),Integer(3)]))
3
```

check()

Check that we are a valid set partition.

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: s = S([[1, 3], [2, 4]])
sage: s.check()
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(4))
>>> s = S([[Integer(1), Integer(3)], [Integer(2), Integer(4)]])
>>> s.check()
```

closers()

Return the maximal elements of the blocks.

EXAMPLES:

```
sage: P = SetPartition([[1,2,4,7],[3,9],[5,6,10,11,13],[8],[12]])
sage: P.closers()
[7, 8, 9, 12, 13]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(2),Integer(4),Integer(7)],
↪[Integer(3),Integer(9)],[Integer(5),Integer(6),Integer(10),Integer(11),
↪Integer(13)],[Integer(8)],[Integer(12)]])
>>> P.closers()
[7, 8, 9, 12, 13]
```

crossings()

Return the crossing arcs of a set partition on a totally ordered set.

OUTPUT:

We place the elements of the ground set in order on a line and draw the set partition by linking consecutive elements of each block in the upper half-plane. This function returns a list of the pairs of crossing lines (as a line correspond to a pair, it returns a list of pairs of pairs).

EXAMPLES:

```
sage: p = SetPartition([[1,4],[2,5,7],[3,6]])
sage: p.crossings()
[((1, 4), (2, 5)), ((1, 4), (3, 6)), ((2, 5), (3, 6)), ((3, 6), (5, 7))]
```

```
>>> from sage.all import *
>>> p = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(5),
↪Integer(7)],[Integer(3),Integer(6)]])
>>> p.crossings()
[((1, 4), (2, 5)), ((1, 4), (3, 6)), ((2, 5), (3, 6)), ((3, 6), (5, 7))]
```

crossings_iterator()

Return the crossing arcs of a set partition on a totally ordered set.

OUTPUT:

We place the elements of the ground set in order on a line and draw the set partition by linking consecutive elements of each block in the upper half-plane. This function returns an iterator over the pairs of crossing lines (as a line correspond to a pair, the iterator produces pairs of pairs).

EXAMPLES:

```
sage: p = SetPartition([[1,4],[2,5,7],[3,6]])
sage: next(p.crossings_iterator())
((1, 4), (2, 5))
```

```
>>> from sage.all import *
>>> p = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(5),
↪Integer(7)],[Integer(3),Integer(6)]])
>>> next(p.crossings_iterator())
((1, 4), (2, 5))
```

is_atomic()

Return if `self` is an atomic set partition.

A (standard) set partition A can be split if there exist $j < i$ such that $\max(A_j) < \min(A_i)$ where A is ordered by minimal elements. This means we can write $A = B|C$ for some nonempty set partitions B and C. We call a set partition *atomic* if it cannot be split and is nonempty. Here, the pipe symbol | is as defined in method `pipe()`.

EXAMPLES:

```
sage: SetPartition([[1,3], [2]]).is_atomic()
True
sage: SetPartition([[1,3], [2], [4]]).is_atomic()
False
sage: SetPartition([[1], [2,4], [3]]).is_atomic()
False
sage: SetPartition([[1,2,3,4]]).is_atomic()
True
sage: SetPartition([[1, 4], [2], [3]]).is_atomic()
True
sage: SetPartition([]).is_atomic()
False
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(1),Integer(3)], [Integer(2)]]).is_atomic()
True
>>> SetPartition([[Integer(1),Integer(3)], [Integer(2)], [Integer(4)]]).is_
↪atomic()
False
>>> SetPartition([[Integer(1)], [Integer(2),Integer(4)], [Integer(3)]]).is_
↪atomic()
False
>>> SetPartition([[Integer(1),Integer(2),Integer(3),Integer(4)]]).is_atomic()
True
>>> SetPartition([[Integer(1), Integer(4)], [Integer(2)], [Integer(3)]]).is_
↪atomic()
True
>>> SetPartition([]).is_atomic()
False
```

is_noncrossing()

Check if `self` is noncrossing.

OUTPUT:

We place the elements of the ground set in order on a line and draw the set partition by linking consecutive elements of each block in the upper half-plane. This function returns `True` if the picture obtained this way has no crossings.

EXAMPLES:

```
sage: p = SetPartition([[1,4],[2,5,7],[3,6]])
sage: p.is_noncrossing()
False

sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.is_noncrossing()
False
sage: PerfectMatching([(1, 4), (2, 3), (5, 6)]).is_noncrossing()
True
```

```
>>> from sage.all import *
>>> p = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(5),
↪Integer(7)],[Integer(3),Integer(6)]])
>>> p.is_noncrossing()
False

>>> n = PerfectMatching([Integer(3),Integer(8),Integer(1),Integer(7),
↪Integer(6),Integer(5),Integer(4),Integer(2)]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
>>> n.is_noncrossing()
False
>>> PerfectMatching([(Integer(1), Integer(4)), (Integer(2), Integer(3)),␣
↪(Integer(5), Integer(6))]).is_noncrossing()
True
```

is_nonnesting()

Return if `self` is nonnesting or not.

OUTPUT:

We place the elements of the ground set in order on a line and draw the set partition by linking consecutive elements of each block in the upper half-plane. This function returns `True` if the picture obtained this way has no nestings.

EXAMPLES:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.is_nonnesting()
False
sage: PerfectMatching([(1, 3), (2, 5), (4, 6)]).is_nonnesting()
True
```

```
>>> from sage.all import *
>>> n = PerfectMatching([Integer(3),Integer(8),Integer(1),Integer(7),
↪Integer(6),Integer(5),Integer(4),Integer(2)]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
>>> n.is_nonnesting()
False
>>> PerfectMatching([(Integer(1), Integer(3)), (Integer(2), Integer(5)),␣
↪(Integer(4), Integer(6))]).is_nonnesting()
True
```

latex_options()

Return the latex options for use in the `_latex_` function as a dictionary. The default values are set using the global options.

Options can be found in *set_latex_options()*

EXAMPLES:

```
sage: SP = SetPartition([[1,6], [3,5,4]]); SP.latex_options()
{'angle': 0,
 'color': 'black',
 'fill': False,
 'plot': None,
 'radius': '1cm',
 'show_labels': True,
 'tikz_scale': 1}
```

```
>>> from sage.all import *
>>> SP = SetPartition([[Integer(1),Integer(6)], [Integer(3),Integer(5),
↪Integer(4)]]); SP.latex_options()
{'angle': 0,
 'color': 'black',
 'fill': False,
 'plot': None,
 'radius': '1cm',
 'show_labels': True,
 'tikz_scale': 1}
```

nestings()

Return the nestings of `self`.

OUTPUT:

We place the elements of the ground set in order on a line and draw the set partition by linking consecutive elements of each block in the upper half-plane. This function returns the list of the pairs of nesting lines (as a line correspond to a pair, it returns a list of pairs of pairs).

EXAMPLES:

```
sage: m = PerfectMatching([(1, 6), (2, 7), (3, 5), (4, 8)])
sage: m.nestings()
[((1, 6), (3, 5)), ((2, 7), (3, 5))]

sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.nestings()
[((2, 8), (4, 7)), ((2, 8), (5, 6)), ((4, 7), (5, 6))]
```

```
>>> from sage.all import *
>>> m = PerfectMatching([(Integer(1), Integer(6)), (Integer(2), Integer(7)),␣
↪(Integer(3), Integer(5)), (Integer(4), Integer(8))])
>>> m.nestings()
[((1, 6), (3, 5)), ((2, 7), (3, 5))]

>>> n = PerfectMatching([Integer(3),Integer(8),Integer(1),Integer(7),
↪Integer(6),Integer(5),Integer(4),Integer(2)]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
>>> n.nestings()
[((2, 8), (4, 7)), ((2, 8), (5, 6)), ((4, 7), (5, 6))]
```

nestings_iterator()

Iterate over the nestings of `self`.

OUTPUT:

We place the elements of the ground set in order on a line and draw the set partition by linking consecutive elements of each block in the upper half-plane. This function returns an iterator over the pairs of nesting lines (as a line correspond to a pair, the iterator produces pairs of pairs).

EXAMPLES:

```
sage: n = PerfectMatching([(1, 6), (2, 7), (3, 5), (4, 8)])
sage: it = n.nestings_iterator()
sage: next(it)
((1, 6), (3, 5))
sage: next(it)
((2, 7), (3, 5))
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

```
>>> from sage.all import *
>>> n = PerfectMatching([(Integer(1), Integer(6)), (Integer(2), Integer(7)),␣
```

(continues on next page)

(continued from previous page)

```
↪(Integer(3), Integer(5)), (Integer(4), Integer(8))])
>>> it = n.nestings_iterator()
>>> next(it)
((1, 6), (3, 5))
>>> next(it)
((2, 7), (3, 5))
>>> next(it)
Traceback (most recent call last):
...
StopIteration
```

number_of_crossings()

Return the number of crossings.

OUTPUT:

We place the elements of the ground set in order on a line and draw the set partition by linking consecutive elements of each block in the upper half-plane. This function returns the number the pairs of crossing lines.

EXAMPLES:

```
sage: p = SetPartition([[1,4],[2,5,7],[3,6]])
sage: p.number_of_crossings()
4

sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.number_of_crossings()
1
```

```
>>> from sage.all import *
>>> p = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(5),
↪Integer(7)],[Integer(3),Integer(6)]])
>>> p.number_of_crossings()
4

>>> n = PerfectMatching([Integer(3),Integer(8),Integer(1),Integer(7),
↪Integer(6),Integer(5),Integer(4),Integer(2)]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
>>> n.number_of_crossings()
1
```

number_of_nestings()

Return the number of nestings of `self`.

OUTPUT:

We place the elements of the ground set in order on a line and draw the set partition by linking consecutive elements of each block in the upper half-plane. This function returns the number the pairs of nesting lines.

EXAMPLES:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
```

(continues on next page)

(continued from previous page)

```
sage: n.number_of_nestings()
3
```

```
>>> from sage.all import *
>>> n = PerfectMatching([Integer(3),Integer(8),Integer(1),Integer(7),
↪Integer(6),Integer(5),Integer(4),Integer(2)]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
>>> n.number_of_nestings()
3
```

openers()

Return the minimal elements of the blocks.

EXAMPLES:

```
sage: P = SetPartition([[1,2,4,7],[3,9],[5,6,10,11,13],[8],[12]])
sage: P.openers()
[1, 3, 5, 8, 12]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(2),Integer(4),Integer(7)],
↪[Integer(3),Integer(9)],[Integer(5),Integer(6),Integer(10),Integer(11),
↪Integer(13)],[Integer(8)],[Integer(12)]])
>>> P.openers()
[1, 3, 5, 8, 12]
```

ordered_set_partition_action(*s*)

Return the action of an ordered set partition `s` on `self`.

Let $A = \{A_1, A_2, \ldots, A_k\}$ be a set partition of some set S and s be an ordered set partition (i.e., set composition) of a subset of $[k]$. Let $A^{\downarrow}$ denote the standardization of A, and $A_{\{i_1,i_2,\ldots,i_m\}}$ denote the sub-partition $\{A_{i_1}, A_{i_2}, \ldots, A_{i_m}\}$ for any subset $\{i_1, \ldots, i_m\}$ of $\{1, \ldots, k\}$. We define the set partition $s(A)$ by

$$s(A) = A^{\downarrow}_{s_1} | A^{\downarrow}_{s_2} | \cdots | A^{\downarrow}_{s_q}.$$

where $s = (s_1, s_2, \ldots, s_q)$. Here, the pipe symbol | is as defined in method `pipe()`.

This is $s[A]$ in section 2.3 in [LM2011].

INPUT:

- `s` – an ordered set partition with base set a subset of $\{1, \ldots, k\}$

EXAMPLES:

```
sage: A = SetPartition([[1], [2,4], [3]])
sage: s = OrderedSetPartition([[1,3], [2]])
sage: A.ordered_set_partition_action(s)
{{1}, {2}, {3, 4}}
sage: s = OrderedSetPartition([[2,3], [1]])
sage: A.ordered_set_partition_action(s)
{{1, 3}, {2}, {4}}
```

```
>>> from sage.all import *
>>> A = SetPartition([[Integer(1)], [Integer(2),Integer(4)], [Integer(3)]])
>>> s = OrderedSetPartition([[Integer(1),Integer(3)], [Integer(2)]])
>>> A.ordered_set_partition_action(s)
{{1}, {2}, {3, 4}}
>>> s = OrderedSetPartition([[Integer(2),Integer(3)], [Integer(1)]])
>>> A.ordered_set_partition_action(s)
{{1, 3}, {2}, {4}}
```

We create Figure 1 in [LM2011] (we note that there is a typo in the lower-left corner of the table in the published version of the paper, whereas the arXiv version gives the correct partition):

```
sage: A = SetPartition([[1,3], [2,9], [4,5,8], [7]])
sage: B = SetPartition([[1,3], [2,8], [4,5,6], [7]])
sage: C = SetPartition([[1,5], [2,8], [3,4,6], [7]])
sage: s = OrderedSetPartition([[1,3], [2]])
sage: t = OrderedSetPartition([[2], [3,4]])
sage: u = OrderedSetPartition([[1], [2,3,4]])
sage: A.ordered_set_partition_action(s)
{{1, 2}, {3, 4, 5}, {6, 7}}
sage: A.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 6}, {5}}
sage: A.ordered_set_partition_action(u)
{{1, 2}, {3, 8}, {4, 5, 7}, {6}}
sage: B.ordered_set_partition_action(s)
{{1, 2}, {3, 4, 5}, {6, 7}}
sage: B.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 5}, {6}}
sage: B.ordered_set_partition_action(u)
{{1, 2}, {3, 8}, {4, 5, 6}, {7}}
sage: C.ordered_set_partition_action(s)
{{1, 4}, {2, 3, 5}, {6, 7}}
sage: C.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 5}, {6}}
sage: C.ordered_set_partition_action(u)
{{1, 2}, {3, 8}, {4, 5, 6}, {7}}
```

```
>>> from sage.all import *
>>> A = SetPartition([[Integer(1),Integer(3)], [Integer(2),Integer(9)],
→[Integer(4),Integer(5),Integer(8)], [Integer(7)]])
>>> B = SetPartition([[Integer(1),Integer(3)], [Integer(2),Integer(8)],
→[Integer(4),Integer(5),Integer(6)], [Integer(7)]])
>>> C = SetPartition([[Integer(1),Integer(5)], [Integer(2),Integer(8)],
→[Integer(3),Integer(4),Integer(6)], [Integer(7)]])
>>> s = OrderedSetPartition([[Integer(1),Integer(3)], [Integer(2)]])
>>> t = OrderedSetPartition([[Integer(2)], [Integer(3),Integer(4)]])
>>> u = OrderedSetPartition([[Integer(1)], [Integer(2),Integer(3),
→Integer(4)]])
>>> A.ordered_set_partition_action(s)
{{1, 2}, {3, 4, 5}, {6, 7}}
>>> A.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 6}, {5}}
>>> A.ordered_set_partition_action(u)
```

(continues on next page)

(continued from previous page)

```
{{1, 2}, {3, 8}, {4, 5, 7}, {6}}
>>> B.ordered_set_partition_action(s)
{{1, 2}, {3, 4, 5}, {6, 7}}
>>> B.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 5}, {6}}
>>> B.ordered_set_partition_action(u)
{{1, 2}, {3, 8}, {4, 5, 6}, {7}}
>>> C.ordered_set_partition_action(s)
{{1, 4}, {2, 3, 5}, {6, 7}}
>>> C.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 5}, {6}}
>>> C.ordered_set_partition_action(u)
{{1, 2}, {3, 8}, {4, 5, 6}, {7}}
```

REFERENCES:

- [LM2011]

pipe(*other*)

Return the pipe of the set partitions `self` and `other`.

The pipe of two set partitions is defined as follows:

For any integer k and any subset I of $\mathbf{Z}$, let $I + k$ denote the subset of $\mathbf{Z}$ obtained by adding k to every element of k.

If B and C are set partitions of $[n]$ and $[m]$, respectively, then the pipe of B and C is defined as the set partition

$$\{B_1, B_2, \ldots, B_b, C_1 + n, C_2 + n, \ldots, C_c + n\}$$

of $[n + m]$, where $B = \{B_1, B_2, \ldots, B_b\}$ and $C = \{C_1, C_2, \ldots, C_c\}$. This pipe is denoted by $B|C$.

EXAMPLES:

```
sage: SetPartition([[1,3],[2,4]]).pipe(SetPartition([[1,3],[2]]))
{{1, 3}, {2, 4}, {5, 7}, {6}}
sage: SetPartition([]).pipe(SetPartition([[1,2],[3,5],[4]]))
{{1, 2}, {3, 5}, {4}}
sage: SetPartition([[1,2],[3,5],[4]]).pipe(SetPartition([]))
{{1, 2}, {3, 5}, {4}}
sage: SetPartition([[1,2],[3]]).pipe(SetPartition([[1]]))
{{1, 2}, {3}, {4}}
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(1),Integer(3)],[Integer(2),Integer(4)]]).
↪pipe(SetPartition([[Integer(1),Integer(3)],[Integer(2)]]))
{{1, 3}, {2, 4}, {5, 7}, {6}}
>>> SetPartition([]).pipe(SetPartition([[Integer(1),Integer(2)],[Integer(3),
↪Integer(5)],[Integer(4)]]))
{{1, 2}, {3, 5}, {4}}
>>> SetPartition([[Integer(1),Integer(2)],[Integer(3),Integer(5)],
↪[Integer(4)]]).pipe(SetPartition([]))
{{1, 2}, {3, 5}, {4}}
>>> SetPartition([[Integer(1),Integer(2)],[Integer(3)]]).
```

(continues on next page)

(continued from previous page)

```
↪pipe(SetPartition([[Integer(1)]]))
{{1, 2}, {3}, {4}}
```

plot(*angle=None*, *color='black'*, *base_set_dict=None*)

Return a plot of `self`.

INPUT:

- `angle` – (default: $\pi/4$) the angle at which the arcs take off (if angle is negative, the arcs are drawn below the horizontal line)
- `color` – (default: `'black'`) color of the arcs
- `base_set_dict` – (optional) dictionary with keys elements of `base_set()` and values as integer or float

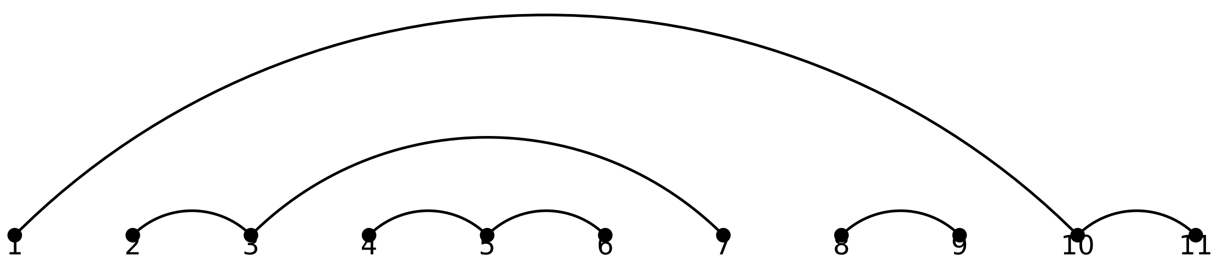
EXAMPLES:

```
sage: p = SetPartition([[1,10,11],[2,3,7],[4,5,6],[8,9]])
sage: p.plot()                                                        #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 29 graphics primitives
```

```
>>> from sage.all import *
>>> p = SetPartition([[Integer(1),Integer(10),Integer(11)],[Integer(2),
↪Integer(3),Integer(7)],[Integer(4),Integer(5),Integer(6)],[Integer(8),
↪Integer(9)]])
>>> p.plot()                                                        #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 29 graphics primitives
```

```
sage: p = SetPartition([[1,3,4],[2,5]])
sage: print(p.plot().description())                                   #␣
↪needs sage.plot sage.symbolic
Point set defined by 1 point(s):    [(0.0, 0.0)]
Point set defined by 1 point(s):    [(1.0, 0.0)]
Point set defined by 1 point(s):    [(2.0, 0.0)]
Point set defined by 1 point(s):    [(3.0, 0.0)]
Point set defined by 1 point(s):    [(4.0, 0.0)]
Text '1' at the point (0.0,-0.1)
Text '2' at the point (1.0,-0.1)
Text '3' at the point (2.0,-0.1)
Text '4' at the point (3.0,-0.1)
Text '5' at the point (4.0,-0.1)
Arc with center (1.0,-1.0) radii (1.41421356237...,1.41421356237...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
Arc with center (2.5,-0.5) radii (0.70710678118...,0.70710678118...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
Arc with center (2.5,-1.5) radii (2.1213203435...,2.1213203435...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
sage: p = SetPartition([['a','c'],['b','d'],['e']])
sage: print(p.plot().description())                                   #␣
↪needs sage.plot sage.symbolic
Point set defined by 1 point(s):  [(0.0, 0.0)]
```

(continues on next page)


1
2
3
4
5
6
7
8
9
10
11

(continued from previous page)

```
Point set defined by 1 point(s):    [(1.0, 0.0)]
Point set defined by 1 point(s):    [(2.0, 0.0)]
Point set defined by 1 point(s):    [(3.0, 0.0)]
Point set defined by 1 point(s):    [(4.0, 0.0)]
Text 'a' at the point (0.0,-0.1)
Text 'b' at the point (1.0,-0.1)
Text 'c' at the point (2.0,-0.1)
Text 'd' at the point (3.0,-0.1)
Text 'e' at the point (4.0,-0.1)
Arc with center (1.0,-1.0) radii (1.41421356237...,1.41421356237...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
Arc with center (2.0,-1.0) radii (1.41421356237...,1.41421356237...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
sage: p = SetPartition([['a','c'],['b','d'],['e']])
sage: print(p.plot(base_set_dict={'a':0,'b':1,'c':2,                        #␣
↪needs sage.plot sage.symbolic
....:                             'd':-2.3,'e':5.4}).description())
Point set defined by 1 point(s):    [(-2.3, 0.0)]
Point set defined by 1 point(s):    [(0.0, 0.0)]
Point set defined by 1 point(s):    [(1.0, 0.0)]
Point set defined by 1 point(s):    [(2.0, 0.0)]
Point set defined by 1 point(s):    [(5.4, 0.0)]
Text 'a' at the point (0.0,-0.1)
Text 'b' at the point (1.0,-0.1)
Text 'c' at the point (2.0,-0.1)
Text 'd' at the point (-2.3,-0.1)
Text 'e' at the point (5.4,-0.1)
Arc with center (-0.6...,-1.65) radii (2.3334523779...,2.3334523779...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
Arc with center (1.0,-1.0) radii (1.4142135623...,1.4142135623...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
```

```
>>> from sage.all import *
>>> p = SetPartition([[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(5)]])
>>> print(p.plot().description())                                       #␣
↪needs sage.plot sage.symbolic
Point set defined by 1 point(s):    [(0.0, 0.0)]
Point set defined by 1 point(s):    [(1.0, 0.0)]
Point set defined by 1 point(s):    [(2.0, 0.0)]
Point set defined by 1 point(s):    [(3.0, 0.0)]
Point set defined by 1 point(s):    [(4.0, 0.0)]
Text '1' at the point (0.0,-0.1)
Text '2' at the point (1.0,-0.1)
Text '3' at the point (2.0,-0.1)
Text '4' at the point (3.0,-0.1)
Text '5' at the point (4.0,-0.1)
Arc with center (1.0,-1.0) radii (1.41421356237...,1.41421356237...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
Arc with center (2.5,-0.5) radii (0.70710678118...,0.70710678118...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
Arc with center (2.5,-1.5) radii (2.1213203435...,2.1213203435...)
```

(continues on next page)

(continued from previous page)

```
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
>>> p = SetPartition([['a','c'],['b','d'],['e']])
>>> print(p.plot().description())                                    #␣
↪needs sage.plot sage.symbolic
Point set defined by 1 point(s):  [(0.0, 0.0)]
Point set defined by 1 point(s):    [(1.0, 0.0)]
Point set defined by 1 point(s):    [(2.0, 0.0)]
Point set defined by 1 point(s):    [(3.0, 0.0)]
Point set defined by 1 point(s):    [(4.0, 0.0)]
Text 'a' at the point (0.0,-0.1)
Text 'b' at the point (1.0,-0.1)
Text 'c' at the point (2.0,-0.1)
Text 'd' at the point (3.0,-0.1)
Text 'e' at the point (4.0,-0.1)
Arc with center (1.0,-1.0) radii (1.41421356237...,1.41421356237...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
Arc with center (2.0,-1.0) radii (1.41421356237...,1.41421356237...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
>>> p = SetPartition([['a','c'],['b','d'],['e']])
>>> print(p.plot(base_set_dict={'a':Integer(0),'b':Integer(1),'c':Integer(2),␣
↪                        # needs sage.plot sage.symbolic
...                                 'd':-RealNumber('2.3'),'e':RealNumber('5.4')}
↪).description())
Point set defined by 1 point(s):    [(-2.3, 0.0)]
Point set defined by 1 point(s):    [(0.0, 0.0)]
Point set defined by 1 point(s):    [(1.0, 0.0)]
Point set defined by 1 point(s):    [(2.0, 0.0)]
Point set defined by 1 point(s):    [(5.4, 0.0)]
Text 'a' at the point (0.0,-0.1)
Text 'b' at the point (1.0,-0.1)
Text 'c' at the point (2.0,-0.1)
Text 'd' at the point (-2.3,-0.1)
Text 'e' at the point (5.4,-0.1)
Arc with center (-0.6...,-1.65) radii (2.3334523779...,2.3334523779...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
Arc with center (1.0,-1.0) radii (1.4142135623...,1.4142135623...)
 angle 0.0 inside the sector (0.785398163397...,2.35619449019...)
```

refinements()

Return a list of refinements of `self`.

See also

`coarsenings()`

EXAMPLES:

```
sage: SetPartition([[1,3],[2,4]]).refinements()                          #␣
↪needs sage.graphs sage.libs.flint
[{{1, 3}, {2, 4}},
 {{1, 3}, {2}, {4}},
```

(continues on next page)

(continued from previous page)

```
 {{1}, {2, 4}, {3}},
 {{1}, {2}, {3}, {4}}]
sage: SetPartition([[1],[2,4],[3]]).refinements()                          #␣
↪needs sage.graphs sage.libs.flint
[{{1}, {2, 4}, {3}}, {{1}, {2}, {3}, {4}}]
sage: SetPartition([]).refinements()                                       #␣
↪needs sage.graphs sage.libs.flint
[{}]
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(1),Integer(3)],[Integer(2),Integer(4)]]).
↪refinements()                                # needs sage.graphs sage.libs.
↪flint
[{{1, 3}, {2, 4}},
 {{1, 3}, {2}, {4}},
 {{1}, {2, 4}, {3}},
 {{1}, {2}, {3}, {4}}]
>>> SetPartition([[Integer(1)],[Integer(2),Integer(4)],[Integer(3)]]).
↪refinements()                               # needs sage.graphs sage.libs.flint
[{{1}, {2, 4}, {3}}, {{1}, {2}, {3}, {4}}]
>>> SetPartition([]).refinements()                                         #␣
↪needs sage.graphs sage.libs.flint
[{}]
```

restriction(*I*)

Return the restriction of `self` to a subset `I` (which is given as a set or list or any other iterable).

EXAMPLES:

```
sage: A = SetPartition([[1], [2,3]])
sage: A.restriction([1,2])
{{1}, {2}}
sage: A.restriction([2,3])
{{2, 3}}
sage: A.restriction([])
{}
sage: A.restriction([4])
{}
```

```
>>> from sage.all import *
>>> A = SetPartition([[Integer(1)], [Integer(2),Integer(3)]])
>>> A.restriction([Integer(1),Integer(2)])
{{1}, {2}}
>>> A.restriction([Integer(2),Integer(3)])
{{2, 3}}
>>> A.restriction([])
{}
>>> A.restriction([Integer(4)])
{}
```

set_latex_options(***kwargs*)

Set the latex options for use in the `_latex_` function.

- tikz_scale – (default: 1) scale for use with tikz package
- plot – (default: None) None returns the set notation, linear returns a linear plot, cyclic returns a cyclic plot
- color – (default: 'black') the arc colors
- fill – boolean (default: False); if True then fills color, else you can pass in a color to alter the fill color - *only works with cyclic plot*
- show_labels – boolean (default: True); if True shows labels (*only works with plots*)
- radius – (default: '1cm') radius of circle for cyclic plot - *only works with cyclic plot*
- angle – (default: 0) angle for linear plot

EXAMPLES:

```
sage: SP = SetPartition([[1,6], [3,5,4]])
sage: SP.set_latex_options(tikz_scale=2,plot='linear',fill=True,color='blue',
↪angle=45)
sage: SP.set_latex_options(plot='cyclic')
sage: SP.latex_options()
{'angle': 45,
 'color': 'blue',
 'fill': True,
 'plot': 'cyclic',
 'radius': '1cm',
 'show_labels': True,
 'tikz_scale': 2}
```

```
>>> from sage.all import *
>>> SP = SetPartition([[Integer(1),Integer(6)], [Integer(3),Integer(5),
↪Integer(4)]])
>>> SP.set_latex_options(tikz_scale=Integer(2),plot='linear',fill=True,color=
↪'blue',angle=Integer(45))
>>> SP.set_latex_options(plot='cyclic')
>>> SP.latex_options()
{'angle': 45,
 'color': 'blue',
 'fill': True,
 'plot': 'cyclic',
 'radius': '1cm',
 'show_labels': True,
 'tikz_scale': 2}
```

shape()

Return the integer partition whose parts are the sizes of the sets in self.

EXAMPLES:

```
sage: S = SetPartitions(5)
sage: x = S([[1,2], [3,5,4]])
sage: x.shape()
[3, 2]
sage: y = S([[2], [3,1], [5,4]])
```

(continues on next page)

(continued from previous page)

```
sage: y.shape()
[2, 2, 1]
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(5))
>>> x = S([[Integer(1),Integer(2)], [Integer(3),Integer(5),Integer(4)]])
>>> x.shape()
[3, 2]
>>> y = S([[Integer(2)], [Integer(3),Integer(1)], [Integer(5),Integer(4)]])
>>> y.shape()
[2, 2, 1]
```

shape_partition()

Return the integer partition whose parts are the sizes of the sets in `self`.

EXAMPLES:

```
sage: S = SetPartitions(5)
sage: x = S([[1,2], [3,5,4]])
sage: x.shape()
[3, 2]
sage: y = S([[2], [3,1], [5,4]])
sage: y.shape()
[2, 2, 1]
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(5))
>>> x = S([[Integer(1),Integer(2)], [Integer(3),Integer(5),Integer(4)]])
>>> x.shape()
[3, 2]
>>> y = S([[Integer(2)], [Integer(3),Integer(1)], [Integer(5),Integer(4)]])
>>> y.shape()
[2, 2, 1]
```

size()

Return the cardinality of the base set of `self`, which is the sum of the sizes of the parts of `self`.

This is also known as the *size* (sometimes the *weight*) of a set partition.

EXAMPLES:

```
sage: SetPartition([[1], [2,3], [4]]).base_set_cardinality()
4
sage: SetPartition([[1,2,3,4]]).base_set_cardinality()
4
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(1)], [Integer(2),Integer(3)], [Integer(4)]]).base_
↪set_cardinality()
4
>>> SetPartition([[Integer(1),Integer(2),Integer(3),Integer(4)]]).base_set_
↪cardinality()
4
```

standardization()

Return the standardization of `self`.

Given a set partition $A = \{A_1, \ldots, A_n\}$ of an ordered set S, the standardization of A is the set partition of $\{1, 2, \ldots, |S|\}$ obtained by replacing the elements of the parts of A by the integers $1, 2, \ldots, |S|$ in such a way that their relative order is preserved (i. e., the smallest element in the whole set partition is replaced by 1, the next-smallest by 2, and so on).

EXAMPLES:

```
sage: SetPartition([[4], [1, 3]]).standardization()
{{1, 2}, {3}}
sage: SetPartition([[4], [6, 3]]).standardization()
{{1, 3}, {2}}
sage: SetPartition([]).standardization()
{}
sage: SetPartition([('c','b'),('d','f'),('e','a')]).standardization()
{{1, 5}, {2, 3}, {4, 6}}
```

```
>>> from sage.all import *
>>> SetPartition([[Integer(4)], [Integer(1), Integer(3)]]).standardization()
{{1, 2}, {3}}
>>> SetPartition([[Integer(4)], [Integer(6), Integer(3)]]).standardization()
{{1, 3}, {2}}
>>> SetPartition([]).standardization()
{}
>>> SetPartition([('c','b'),('d','f'),('e','a')]).standardization()
{{1, 5}, {2, 3}, {4, 6}}
```

strict_coarsenings()

Return all strict coarsenings of `self`.

Strict coarsening is the binary relation on set partitions defined as the transitive-and-reflexive closure of the relation $\prec$ defined as follows: For two set partitions A and B, we have $A \prec B$ if there exist parts A_i, A_j of A such that $\max(A_i) < \min(A_j)$ and $B = A \setminus \{A_i, A_j\} \cup \{A_i \cup A_j\}$.

EXAMPLES:

```
sage: A = SetPartition([[1],[2,3],[4]])
sage: A.strict_coarsenings()
[{{1}, {2, 3}, {4}}, {{1, 2, 3}, {4}}, {{1, 4}, {2, 3}},
 {{1}, {2, 3, 4}}, {{1, 2, 3, 4}}]
sage: SetPartition([[1],[2,4],[3]]).strict_coarsenings()
[{{1}, {2, 4}, {3}}, {{1, 2, 4}, {3}}, {{1, 3}, {2, 4}}]
sage: SetPartition([]).strict_coarsenings()
[{}]
```

```
>>> from sage.all import *
>>> A = SetPartition([[Integer(1)],[Integer(2),Integer(3)],[Integer(4)]])
>>> A.strict_coarsenings()
[{{1}, {2, 3}, {4}}, {{1, 2, 3}, {4}}, {{1, 4}, {2, 3}},
 {{1}, {2, 3, 4}}, {{1, 2, 3, 4}}]
>>> SetPartition([[Integer(1)],[Integer(2),Integer(4)],[Integer(3)]]).strict_
↪coarsenings()
[{{1}, {2, 4}, {3}}, {{1, 2, 4}, {3}}, {{1, 3}, {2, 4}}]
```

(continues on next page)

(continued from previous page)

```
>>> SetPartition([]).strict_coarsenings()
[{}]
```

to_partition()

Return the integer partition whose parts are the sizes of the sets in `self`.

EXAMPLES:

```
sage: S = SetPartitions(5)
sage: x = S([[1,2], [3,5,4]])
sage: x.shape()
[3, 2]
sage: y = S([[2], [3,1], [5,4]])
sage: y.shape()
[2, 2, 1]
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(5))
>>> x = S([[Integer(1),Integer(2)], [Integer(3),Integer(5),Integer(4)]])
>>> x.shape()
[3, 2]
>>> y = S([[Integer(2)], [Integer(3),Integer(1)], [Integer(5),Integer(4)]])
>>> y.shape()
[2, 2, 1]
```

to_permutation()

Convert a set partition of $\{1, ..., n\}$ to a permutation by considering the blocks of the partition as cycles.

The cycles are such that the number of excedences is maximised, that is, each cycle is of the form $(a_1, a_2, ..., a_k)$ with $a_1 < a_2 < ... < a_k$.

EXAMPLES:

```
sage: s = SetPartition([[1,3],[2,4]])
sage: s.to_permutation()
[3, 4, 1, 2]
```

```
>>> from sage.all import *
>>> s = SetPartition([[Integer(1),Integer(3)],[Integer(2),Integer(4)]])
>>> s.to_permutation()
[3, 4, 1, 2]
```

to_restricted_growth_word(*bijection='blocks'*)

Convert a set partition of $\{1, ..., n\}$ to a word of length n with letters in the nonnegative integers such that each letter is at most 1 larger than all the letters before.

INPUT:

- `bijection` – (default: `blocks`) defines the map from set partitions to restricted growth functions. These are currently:
  - `blocks`: *to_restricted_growth_word_blocks()*.
  - `intertwining`: *to_restricted_growth_word_intertwining()*.

OUTPUT: a restricted growth word

See also:

SetPartitions.from_restricted_growth_word()

EXAMPLES:

```
sage: P = SetPartition([[1,4],[2,8],[3,5,6,9],[7]])
sage: P.to_restricted_growth_word()
[0, 1, 2, 0, 2, 2, 3, 1, 2]

sage: P.to_restricted_growth_word("intertwining")
[0, 1, 2, 2, 1, 0, 3, 3, 2]

sage: P = SetPartition([[1,2,4,7],[3,9],[5,6,10,11,13],[8],[12]])
sage: P.to_restricted_growth_word()
[0, 0, 1, 0, 2, 2, 0, 3, 1, 2, 2, 4, 2]

sage: P.to_restricted_growth_word("intertwining")
[0, 0, 1, 1, 2, 0, 1, 3, 3, 3, 0, 4, 1]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(8)],
→[Integer(3),Integer(5),Integer(6),Integer(9)],[Integer(7)]])
>>> P.to_restricted_growth_word()
[0, 1, 2, 0, 2, 2, 3, 1, 2]

>>> P.to_restricted_growth_word("intertwining")
[0, 1, 2, 2, 1, 0, 3, 3, 2]

>>> P = SetPartition([[Integer(1),Integer(2),Integer(4),Integer(7)],
→[Integer(3),Integer(9)],[Integer(5),Integer(6),Integer(10),Integer(11),
→Integer(13)],[Integer(8)],[Integer(12)]])
>>> P.to_restricted_growth_word()
[0, 0, 1, 0, 2, 2, 0, 3, 1, 2, 2, 4, 2]

>>> P.to_restricted_growth_word("intertwining")
[0, 0, 1, 1, 2, 0, 1, 3, 3, 3, 0, 4, 1]
```

to_restricted_growth_word_blocks()

Convert a set partition of $\{1, ..., n\}$ to a word of length n with letters in the nonnegative integers such that each letter is at most 1 larger than all the letters before.

The word is obtained by sorting the blocks by their minimal element and setting the letters at the positions of the elements in the i-th block to i.

OUTPUT: a restricted growth word

See also:

to_restricted_growth_word() SetPartitions.from_restricted_growth_word()

EXAMPLES:

```
sage: P = SetPartition([[1,4],[2,8],[3,5,6,9],[7]])
sage: P.to_restricted_growth_word_blocks()
[0, 1, 2, 0, 2, 2, 3, 1, 2]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(8)],
↪[Integer(3),Integer(5),Integer(6),Integer(9)],[Integer(7)]])
>>> P.to_restricted_growth_word_blocks()
[0, 1, 2, 0, 2, 2, 3, 1, 2]
```

to_restricted_growth_word_intertwining()

Convert a set partition of $\{1, ..., n\}$ to a word of length n with letters in the nonnegative integers such that each letter is at most 1 larger than all the letters before.

The i-th letter of the word is the numbers of crossings of the arc (or half-arc) in the extended arc diagram ending at i, with arcs (or half-arcs) beginning at a smaller element and ending at a larger element.

OUTPUT: a restricted growth word

See also

to_restricted_growth_word() SetPartitions.from_restricted_growth_word()

EXAMPLES:

```
sage: P = SetPartition([[1,4],[2,8],[3,5,6,9],[7]])
sage: P.to_restricted_growth_word_intertwining()
[0, 1, 2, 2, 1, 0, 3, 3, 2]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(8)],
↪[Integer(3),Integer(5),Integer(6),Integer(9)],[Integer(7)]])
>>> P.to_restricted_growth_word_intertwining()
[0, 1, 2, 2, 1, 0, 3, 3, 2]
```

to_rook_placement(*bijection='arcs'*)

Return a set of pairs defining a placement of non-attacking rooks on a triangular board.

The cells of the board corresponding to a set partition of $\{1, ..., n\}$ are the pairs (i, j) with $0 < i < j < n+1$.

INPUT:

- `bijection` – (default: `arcs`) defines the bijection from set partitions to rook placements. These are currently:
  - `arcs`: *arcs()*
  - `gamma`: *to_rook_placement_gamma()*
  - `rho`: *to_rook_placement_rho()*
  - `psi`: *to_rook_placement_psi()*

See also

SetPartitions.from_rook_placement()

EXAMPLES:

```
sage: P = SetPartition([[1,2,4,7],[3,9],[5,6,10,11,13],[8],[12]])
sage: P.to_rook_placement()
[(1, 2), (2, 4), (4, 7), (3, 9), (5, 6), (6, 10), (10, 11), (11, 13)]
sage: P.to_rook_placement("gamma")
[(1, 4), (3, 5), (4, 6), (5, 8), (7, 11), (8, 9), (10, 12), (12, 13)]
sage: P.to_rook_placement("rho")
[(1, 2), (2, 6), (3, 4), (4, 10), (5, 9), (6, 7), (10, 11), (11, 13)]
sage: P.to_rook_placement("psi")
[(1, 2), (2, 6), (3, 4), (5, 9), (6, 7), (7, 10), (9, 11), (11, 13)]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(2),Integer(4),Integer(7)],
↪[Integer(3),Integer(9)],[Integer(5),Integer(6),Integer(10),Integer(11),
↪Integer(13)],[Integer(8)],[Integer(12)]])
>>> P.to_rook_placement()
[(1, 2), (2, 4), (4, 7), (3, 9), (5, 6), (6, 10), (10, 11), (11, 13)]
>>> P.to_rook_placement("gamma")
[(1, 4), (3, 5), (4, 6), (5, 8), (7, 11), (8, 9), (10, 12), (12, 13)]
>>> P.to_rook_placement("rho")
[(1, 2), (2, 6), (3, 4), (4, 10), (5, 9), (6, 7), (10, 11), (11, 13)]
>>> P.to_rook_placement("psi")
[(1, 2), (2, 6), (3, 4), (5, 9), (6, 7), (7, 10), (9, 11), (11, 13)]
```

to_rook_placement_gamma()

Return the rook diagram obtained by placing rooks according to Wachs and White's bijection gamma.

Note that our index convention differs from the convention in [WW1991]: regarding the rook board as a lower-right triangular grid, we refer with (i, j) to the cell in the i-th column from the right and the j-th row from the top.

The algorithm proceeds as follows: non-attacking rooks are placed beginning at the left column. If $n+1-i$ is an opener, column i remains empty. Otherwise, we place a rook into column i, such that the number of cells below the rook, which are not yet attacked by another rook, equals the index of the block to which $n+1-i$ belongs.

OUTPUT: list of coordinates

See also

- *to_rook_placement()*
- *SetPartitions.from_rook_placement()*
- *SetPartitions.from_rook_placement_gamma()*

EXAMPLES:

```
sage: P = SetPartition([[1,4],[2,8],[3,5,6,9],[7]])
sage: P.to_rook_placement_gamma()
[(1, 3), (2, 7), (4, 5), (5, 6), (6, 9)]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(8)],
↪[Integer(3),Integer(5),Integer(6),Integer(9)],[Integer(7)]])
>>> P.to_rook_placement_gamma()
[(1, 3), (2, 7), (4, 5), (5, 6), (6, 9)]
```

Figure 5 in [WW1991]:

```
sage: P = SetPartition([[1,2,4,7],[3,9],[5,6,10,11,13],[8],[12]])
sage: r = P.to_rook_placement_gamma(); r
[(1, 4), (3, 5), (4, 6), (5, 8), (7, 11), (8, 9), (10, 12), (12, 13)]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(2),Integer(4),Integer(7)],
↪[Integer(3),Integer(9)],[Integer(5),Integer(6),Integer(10),Integer(11),
↪Integer(13)],[Integer(8)],[Integer(12)]])
>>> r = P.to_rook_placement_gamma(); r
[(1, 4), (3, 5), (4, 6), (5, 8), (7, 11), (8, 9), (10, 12), (12, 13)]
```

to_rook_placement_psi()

Return the rook diagram obtained by placing rooks according to Yip's bijection psi.

OUTPUT: list of coordinates

See also

- *to_rook_placement()*
- *SetPartitions.from_rook_placement()*
- *SetPartitions.from_rook_placement_psi()*

EXAMPLES:

Example 36 (arXiv version: Example 4.5) in [Yip2018]:

```
sage: P = SetPartition([[1, 5], [2], [3, 8, 9], [4], [6, 7]])
sage: P.to_rook_placement_psi()
[(1, 7), (3, 8), (4, 5), (7, 9)]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1), Integer(5)], [Integer(2)], [Integer(3),␣
↪Integer(8), Integer(9)], [Integer(4)], [Integer(6), Integer(7)]])
>>> P.to_rook_placement_psi()
[(1, 7), (3, 8), (4, 5), (7, 9)]
```

Note that the columns corresponding to the minimal elements of the blocks remain empty.

to_rook_placement_rho()

Return the rook diagram obtained by placing rooks according to Wachs and White's bijection rho.

Note that our index convention differs from the convention in [WW1991]: regarding the rook board as a lower-right triangular grid, we refer with (i, j) to the cell in the i-th column from the right and the j-th row from the top.

The algorithm proceeds as follows: non-attacking rooks are placed beginning at the top row. The columns corresponding to the closers of the set partition remain empty. Let rs_j be the number of closers which are larger than j and whose block is before the block of j.

We then place a rook into row j, such that the number of cells to the left of the rook, which are not yet attacked by another rook and are not in a column corresponding to a closer, equals rs_j, unless there are not enough cells in this row available, in which case the row remains empty.

One can show that the precisely those rows which correspond to openers of the set partition remain empty.

OUTPUT: list of coordinates

See also

- `to_rook_placement()`
- `SetPartitions.from_rook_placement()`
- `SetPartitions.from_rook_placement_rho()`

EXAMPLES:

```
sage: P = SetPartition([[1,4],[2,8],[3,5,6,9],[7]])
sage: P.to_rook_placement_rho()
[(1, 5), (2, 6), (3, 4), (5, 9), (6, 8)]
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(4)],[Integer(2),Integer(8)],
↪[Integer(3),Integer(5),Integer(6),Integer(9)],[Integer(7)]])
>>> P.to_rook_placement_rho()
[(1, 5), (2, 6), (3, 4), (5, 9), (6, 8)]
```

Figure 6 in [WW1991]:

```
sage: P = SetPartition([[1,2,4,7],[3,9],[5,6,10,11,13],[8],[12]])
sage: r = P.to_rook_placement_rho(); r
[(1, 2), (2, 6), (3, 4), (4, 10), (5, 9), (6, 7), (10, 11), (11, 13)]

sage: sorted(P.closers() + [i for i, _ in r]) == list(range(1,14))
True
sage: sorted(P.openers() + [j for _, j in r]) == list(range(1,14))
True
```

```
>>> from sage.all import *
>>> P = SetPartition([[Integer(1),Integer(2),Integer(4),Integer(7)],
↪[Integer(3),Integer(9)],[Integer(5),Integer(6),Integer(10),Integer(11),
↪Integer(13)],[Integer(8)],[Integer(12)]])
>>> r = P.to_rook_placement_rho(); r
[(1, 2), (2, 6), (3, 4), (4, 10), (5, 9), (6, 7), (10, 11), (11, 13)]

>>> sorted(P.closers() + [i for i, _ in r]) == list(range(Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(14)))
True
>>> sorted(P.openers() + [j for _, j in r]) == list(range(Integer(1),
↪Integer(14)))
True
```

class sage.combinat.set_partition.**SetPartitions**

Bases: UniqueRepresentation, Parent

An (unordered) partition of a set S is a set of pairwise disjoint nonempty subsets with union S, and is represented by a sorted list of such subsets.

`SetPartitions(s)` returns the class of all set partitions of the set `s`, which can be given as a set or a string; if a string, each character is considered an element.

`SetPartitions(n)`, where `n` is an integer, returns the class of all set partitions of the set $\{1, 2, \ldots, n\}$.

You may specify a second argument k. If k is an integer, *SetPartitions* returns the class of set partitions into k parts; if it is an integer partition, *SetPartitions* returns the class of set partitions whose block sizes correspond to that integer partition.

The Bell number B_n, named in honor of Eric Temple Bell, is the number of different partitions of a set with n elements.

EXAMPLES:

```
sage: S = [1,2,3,4]
sage: SetPartitions(S, 2)
Set partitions of {1, 2, 3, 4} with 2 parts
sage: SetPartitions([1,2,3,4], [3,1]).list()                              #␣
↪needs sage.graphs sage.rings.finite_rings
[{{1}, {2, 3, 4}}, {{1, 2, 3}, {4}}, {{1, 2, 4}, {3}}, {{1, 3, 4}, {2}}]
sage: SetPartitions(7, [3,3,1]).cardinality()                             #␣
↪needs sage.libs.flint
70
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2),Integer(3),Integer(4)]
>>> SetPartitions(S, Integer(2))
Set partitions of {1, 2, 3, 4} with 2 parts
>>> SetPartitions([Integer(1),Integer(2),Integer(3),Integer(4)], [Integer(3),
↪Integer(1)]).list()                              # needs sage.graphs sage.
↪rings.finite_rings
[{{1}, {2, 3, 4}}, {{1, 2, 3}, {4}}, {{1, 2, 4}, {3}}, {{1, 3, 4}, {2}}]
>>> SetPartitions(Integer(7), [Integer(3),Integer(3),Integer(1)]).cardinality()  ␣
↪                              # needs sage.libs.flint
70
```

In strings, repeated letters are not considered distinct as of Issue #14140:

```
sage: SetPartitions('abcde').cardinality()                                #␣
↪needs sage.libs.flint
52
sage: SetPartitions('aabcd').cardinality()                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.flint
15
```

```
>>> from sage.all import *
>>> SetPartitions('abcde').cardinality()                                    #␣
↪needs sage.libs.flint
52
>>> SetPartitions('aabcd').cardinality()                                    #␣
↪needs sage.libs.flint
15
```

If the number of parts exceeds the length of the set, an empty iterator is returned (Issue #37643):

```
sage: SetPartitions(range(3), 4).list()
[]
sage: SetPartitions('abcd', 6).list()
[]
```

```
>>> from sage.all import *
>>> SetPartitions(range(Integer(3)), Integer(4)).list()
[]
>>> SetPartitions('abcd', Integer(6)).list()
[]
```

REFERENCES:

- Wikipedia article Partition_of_a_set

Element

alias of *SetPartition*

from_arcs(*arcs*, *n*)

Return the coarsest set partition of $\{1, ..., n\}$ such that any two elements connected by an arc are in the same block.

INPUT:

- `n` – integer specifying the size of the set partition to be produced
- `arcs` – list of pairs specifying which elements are in the same block

See also

- *from_rook_placement()*
- *SetPartition.to_rook_placement()*
- *SetPartition.arcs()*

EXAMPLES:

```
sage: SetPartitions().from_arcs([(2,3)], 5)
{{1}, {2, 3}, {4}, {5}}
```

```
>>> from sage.all import *
>>> SetPartitions().from_arcs([(Integer(2),Integer(3))], Integer(5))
{{1}, {2, 3}, {4}, {5}}
```

from_restricted_growth_word(*w*, *bijection='blocks'*)

Convert a word of length n with letters in the nonnegative integers such that each letter is at most 1 larger than all the letters before to a set partition of $\{1, ..., n\}$.

INPUT:

- `w` – a restricted growth word
- `bijection` – (default: `blocks`) defines the map from restricted growth functions to set partitions. These are currently:
  - `blocks`: .
  - `intertwining`: *from_restricted_growth_word_intertwining()*.

OUTPUT: a set partition

See also

SetPartition.to_restricted_growth_word()

EXAMPLES:

```
sage: SetPartitions().from_restricted_growth_word([0, 1, 2, 0, 2, 2, 3, 1, 2])
{{1, 4}, {2, 8}, {3, 5, 6, 9}, {7}}

sage: SetPartitions().from_restricted_growth_word([0, 0, 1, 0, 2, 2, 0, 3, 1,␣
↪2, 2, 4, 2])
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}

sage: SetPartitions().from_restricted_growth_word([0, 0, 1, 0, 2, 2, 0, 3, 1,␣
↪2, 2, 4, 2], "intertwining")
{{1, 2, 6, 7, 9}, {3, 4}, {5, 10, 13}, {8, 11}, {12}}
```

```
>>> from sage.all import *
>>> SetPartitions().from_restricted_growth_word([Integer(0), Integer(1),␣
↪Integer(2), Integer(0), Integer(2), Integer(2), Integer(3), Integer(1),␣
↪Integer(2)])
{{1, 4}, {2, 8}, {3, 5, 6, 9}, {7}}

>>> SetPartitions().from_restricted_growth_word([Integer(0), Integer(0),␣
↪Integer(1), Integer(0), Integer(2), Integer(2), Integer(0), Integer(3),␣
↪Integer(1), Integer(2), Integer(2), Integer(4), Integer(2)])
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}

>>> SetPartitions().from_restricted_growth_word([Integer(0), Integer(0),␣
↪Integer(1), Integer(0), Integer(2), Integer(2), Integer(0), Integer(3),␣
↪Integer(1), Integer(2), Integer(2), Integer(4), Integer(2)], "intertwining")
{{1, 2, 6, 7, 9}, {3, 4}, {5, 10, 13}, {8, 11}, {12}}
```

from_restricted_growth_word_blocks(*w*)

Convert a word of length n with letters in the nonnegative integers such that each letter is at most 1 larger than all the letters before to a set partition of $\{1, ..., n\}$.

`w[i]` is the index of the block containing `i+1` when sorting the blocks by their minimal element.

INPUT:

- `w` – a restricted growth word

OUTPUT: a set partition

See also

from_restricted_growth_word() SetPartition.to_restricted_growth_word()

EXAMPLES:

```
sage: SetPartitions().from_restricted_growth_word_blocks([0, 0, 1, 0, 2, 2, 0,
→ 3, 1, 2, 2, 4, 2])
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}
```

```
>>> from sage.all import *
>>> SetPartitions().from_restricted_growth_word_blocks([Integer(0),␣
→Integer(0), Integer(1), Integer(0), Integer(2), Integer(2), Integer(0),␣
→Integer(3), Integer(1), Integer(2), Integer(2), Integer(4), Integer(2)])
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}
```

from_restricted_growth_word_intertwining(*w*)

Convert a word of length n with letters in the nonnegative integers such that each letter is at most 1 larger than all the letters before to a set partition of $\{1, ..., n\}$.

The i-th letter of the word is the numbers of crossings of the arc (or half-arc) in the extended arc diagram ending at i, with arcs (or half-arcs) beginning at a smaller element and ending at a larger element.

INPUT:

- `w` – a restricted growth word

OUTPUT: a set partition

See also

from_restricted_growth_word() SetPartition.to_restricted_growth_word()

EXAMPLES:

```
sage: SetPartitions().from_restricted_growth_word_intertwining([0, 0, 1, 0, 2,
→ 2, 0, 3, 1, 2, 2, 4, 2])
{{1, 2, 6, 7, 9}, {3, 4}, {5, 10, 13}, {8, 11}, {12}}
```

```
>>> from sage.all import *
>>> SetPartitions().from_restricted_growth_word_intertwining([Integer(0),␣
→Integer(0), Integer(1), Integer(0), Integer(2), Integer(2), Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(3), Integer(1), Integer(2), Integer(2), Integer(4), Integer(2)])
{{1, 2, 6, 7, 9}, {3, 4}, {5, 10, 13}, {8, 11}, {12}}
```

from_rook_placement(*rooks*, *bijection='arcs'*, *n=None*)

Convert a rook placement of the triangular grid to a set partition of $\{1, ..., n\}$.

If `n` is not given, it is first checked whether it can be determined from the parent, otherwise it is the maximal occurring integer in the set of rooks.

INPUT:

- `rooks` – list of pairs (i, j) satisfying $0 < i < j < n + 1$
- `bijection` – (default: `arcs`) defines the map from rook placements to set partitions. These are currently:
  - `arcs`: *from_arcs()*.
  - `gamma`: *from_rook_placement_gamma()*.
  - `rho`: *from_rook_placement_rho()*.
  - `psi`: *from_rook_placement_psi()*.
- `n` – (optional) the size of the ground set

See also

SetPartition.to_rook_placement()

EXAMPLES:

```
sage: SetPartitions(9).from_rook_placement([[1,4],[2,8],[3,5],[5,6],[6,9]])
{{1, 4}, {2, 8}, {3, 5, 6, 9}, {7}}

sage: SetPartitions(13).from_rook_placement([[12,13],[10,12],[8,9],[7,11],[5,
↪8],[4,6],[3,5],[1,4]], "gamma")
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(9)).from_rook_placement([[Integer(1),Integer(4)],
↪[Integer(2),Integer(8)],[Integer(3),Integer(5)],[Integer(5),Integer(6)],
↪[Integer(6),Integer(9)]])
{{1, 4}, {2, 8}, {3, 5, 6, 9}, {7}}

>>> SetPartitions(Integer(13)).from_rook_placement([[Integer(12),Integer(13)],
↪[Integer(10),Integer(12)],[Integer(8),Integer(9)],[Integer(7),Integer(11)],
↪[Integer(5),Integer(8)],[Integer(4),Integer(6)],[Integer(3),Integer(5)],
↪[Integer(1),Integer(4)]], "gamma")
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}
```

from_rook_placement_gamma(*rooks*, *n*)

Return the set partition of $\{1, ..., n\}$ corresponding to the given rook placement by applying Wachs and White's bijection gamma.

Note that our index convention differs from the convention in [WW1991]: regarding the rook board as a lower-right triangular grid, we refer with (i, j) to the cell in the i-th column from the right and the j-th row from the top.

INPUT:

- n – integer specifying the size of the set partition to be produced
- rooks – list of pairs (i, j) such that $0 < i < j < n + 1$

OUTPUT: a set partition

See also:

- *from_rook_placement()*
- *SetPartition.to_rook_placement()*
- *SetPartition.to_rook_placement_gamma()*

EXAMPLES:

Figure 5 in [WW1991] concerns the following rook placement:

```
sage: r = [(1, 4), (3, 5), (4, 6), (5, 8), (7, 11), (8, 9), (10, 12), (12,
→13)]
```

```
>>> from sage.all import *
>>> r = [(Integer(1), Integer(4)), (Integer(3), Integer(5)), (Integer(4),
→Integer(6)), (Integer(5), Integer(8)), (Integer(7), Integer(11)),
→(Integer(8), Integer(9)), (Integer(10), Integer(12)), (Integer(12),
→Integer(13))]
```

Note that the rook $(1, 4)$, translated into Wachs and White's convention, is a rook in row 4 from the top and column 13 from the left. The corresponding set partition is:

```
sage: SetPartitions().from_rook_placement_gamma(r, 13)
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}
```

```
>>> from sage.all import *
>>> SetPartitions().from_rook_placement_gamma(r, Integer(13))
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}
```

from_rook_placement_psi(*rooks*, *n*)

Return the set partition of $\{1, ..., n\}$ corresponding to the given rook placement by applying Yip's bijection psi.

INPUT:

- n – integer specifying the size of the set partition to be produced
- rooks – list of pairs (i, j) such that $0 < i < j < n + 1$

OUTPUT: a set partition

See also:

- *from_rook_placement()*
- *SetPartition.to_rook_placement()*
- *SetPartition.to_rook_placement_psi()*

EXAMPLES:

Example 36 (arXiv version: Example 4.5) in [Yip2018] concerns the following rook placement:

```
sage: r = [(4,5), (1,7), (3, 8), (7,9)]
sage: SetPartitions().from_rook_placement_psi(r, 9)
{{1, 5}, {2}, {3, 8, 9}, {4}, {6, 7}}
```

```
>>> from sage.all import *
>>> r = [(Integer(4),Integer(5)), (Integer(1),Integer(7)), (Integer(3),
↪Integer(8)), (Integer(7),Integer(9))]
>>> SetPartitions().from_rook_placement_psi(r, Integer(9))
{{1, 5}, {2}, {3, 8, 9}, {4}, {6, 7}}
```

from_rook_placement_rho(*rooks*, *n*)

Return the set partition of $\{1, ..., n\}$ corresponding to the given rook placement by applying Wachs and White's bijection rho.

Note that our index convention differs from the convention in [WW1991]: regarding the rook board as a lower-right triangular grid, we refer with (i, j) to the cell in the i-th column from the right and the j-th row from the top.

INPUT:

- `n` – integer specifying the size of the set partition to be produced
- `rooks` – list of pairs (i, j) such that $0 < i < j < n + 1$

OUTPUT: a set partition

See also

- *from_rook_placement()*
- *SetPartition.to_rook_placement()*
- *SetPartition.to_rook_placement_rho()*

EXAMPLES:

Figure 5 in [WW1991] concerns the following rook placement:

```
sage: r = [(1, 2), (2, 6), (3, 4), (4, 10), (5, 9), (6, 7), (10, 11), (11,
↪13)]
```

```
>>> from sage.all import *
>>> r = [(Integer(1), Integer(2)), (Integer(2), Integer(6)), (Integer(3),
↪Integer(4)), (Integer(4), Integer(10)), (Integer(5), Integer(9)),
↪(Integer(6), Integer(7)), (Integer(10), Integer(11)), (Integer(11),
↪Integer(13))]
```

Note that the rook $(1, 2)$, translated into Wachs and White's convention, is a rook in row 2 from the top and column 13 from the left. The corresponding set partition is:

```
sage: SetPartitions().from_rook_placement_rho(r, 13)
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}
```

```
>>> from sage.all import *
>>> SetPartitions().from_rook_placement_rho(r, Integer(13))
{{1, 2, 4, 7}, {3, 9}, {5, 6, 10, 11, 13}, {8}, {12}}
```

is_less_than(*s*, *t*)

Check if $s < t$ in the refinement ordering on set partitions.

This means that s is a refinement of t and satisfies $s \neq t$.

A set partition s is said to be a refinement of a set partition t of the same set if and only if each part of s is a subset of a part of t.

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: s = S([[1,3],[2,4]])
sage: t = S([[1],[2],[3],[4]])
sage: S.is_less_than(t, s)
True
sage: S.is_less_than(s, t)
False
sage: S.is_less_than(s, s)
False
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(4))
>>> s = S([[Integer(1),Integer(3)],[Integer(2),Integer(4)]])
>>> t = S([[Integer(1)],[Integer(2)],[Integer(3)],[Integer(4)]])
>>> S.is_less_than(t, s)
True
>>> S.is_less_than(s, t)
False
>>> S.is_less_than(s, s)
False
```

is_strict_refinement(*s*, *t*)

Return `True` if `s` is a strict refinement of `t` and satisfies $s \neq t$.

A set partition s is said to be a strict refinement of a set partition t of the same set if and only if one can obtain t from s by repeatedly combining pairs of parts whose convex hulls don't intersect (i. e., whenever we are combining two parts, the maximum of each of them should be smaller than the minimum of the other).

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: s = S([[1],[2],[3],[4]])
sage: t = S([[1,3],[2,4]])
sage: u = S([[1,2,3,4]])
sage: S.is_strict_refinement(s, t)
```

(continues on next page)

(continued from previous page)

```
True
sage: S.is_strict_refinement(t, u)
False
sage: A = SetPartition([[1,3],[2,4]])
sage: B = SetPartition([[1,2,3,4]])
sage: S.is_strict_refinement(s, A)
True
sage: S.is_strict_refinement(t, B)
False
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(4))
>>> s = S([[Integer(1)],[Integer(2)],[Integer(3)],[Integer(4)]])
>>> t = S([[Integer(1),Integer(3)],[Integer(2),Integer(4)]])
>>> u = S([[Integer(1),Integer(2),Integer(3),Integer(4)]])
>>> S.is_strict_refinement(s, t)
True
>>> S.is_strict_refinement(t, u)
False
>>> A = SetPartition([[Integer(1),Integer(3)],[Integer(2),Integer(4)]])
>>> B = SetPartition([[Integer(1),Integer(2),Integer(3),Integer(4)]])
>>> S.is_strict_refinement(s, A)
True
>>> S.is_strict_refinement(t, B)
False
```

lt (*s*, *t*)

Check if $s < t$ in the refinement ordering on set partitions.

This means that s is a refinement of t and satisfies $s \neq t$.

A set partition s is said to be a refinement of a set partition t of the same set if and only if each part of s is a subset of a part of t.

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: s = S([[1,3],[2,4]])
sage: t = S([[1],[2],[3],[4]])
sage: S.is_less_than(t, s)
True
sage: S.is_less_than(s, t)
False
sage: S.is_less_than(s, s)
False
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(4))
>>> s = S([[Integer(1),Integer(3)],[Integer(2),Integer(4)]])
>>> t = S([[Integer(1)],[Integer(2)],[Integer(3)],[Integer(4)]])
>>> S.is_less_than(t, s)
True
>>> S.is_less_than(s, t)
```

(continues on next page)

(continued from previous page)

```
False
>>> S.is_less_than(s, s)
False
```

class sage.combinat.set_partition.**SetPartitions_all**

Bases: *SetPartitions*

All set partitions.

subset (*size=None*, ***kwargs*)

Return the subset of set partitions of a given size and additional keyword arguments.

EXAMPLES:

```
sage: P = SetPartitions()
sage: P.subset(4)
Set partitions of {1, 2, 3, 4}
```

```
>>> from sage.all import *
>>> P = SetPartitions()
>>> P.subset(Integer(4))
Set partitions of {1, 2, 3, 4}
```

class sage.combinat.set_partition.**SetPartitions_set** (*s*)

Bases: *SetPartitions*

Set partitions of a fixed set S.

base_set ()

Return the base set of `self`.

EXAMPLES:

```
sage: SetPartitions(3).base_set()
{1, 2, 3}

sage: sorted(SetPartitions(["a", "b", "c"]).base_set())
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(3)).base_set()
{1, 2, 3}

>>> sorted(SetPartitions(["a", "b", "c"]).base_set())
['a', 'b', 'c']
```

base_set_cardinality ()

Return the cardinality of the base set of `self`.

EXAMPLES:

```
sage: SetPartitions(3).base_set_cardinality()
3
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(3)).base_set_cardinality()
3
```

cardinality()

Return the number of set partitions of the set S.

The cardinality is given by the n-th Bell number where n is the number of elements in the set S.

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: SetPartitions([1,2,3,4]).cardinality()
15
sage: SetPartitions(3).cardinality()
5
sage: SetPartitions(3,2).cardinality()
3
sage: SetPartitions([]).cardinality()
1
```

```
>>> from sage.all import *
>>> # needs sage.libs.flint
>>> SetPartitions([Integer(1),Integer(2),Integer(3),Integer(4)]).cardinality()
15
>>> SetPartitions(Integer(3)).cardinality()
5
>>> SetPartitions(Integer(3),Integer(2)).cardinality()
3
>>> SetPartitions([]).cardinality()
1
```

random_element()

Return a random set partition.

This is a very naive implementation of Knuths outline in F3B, 7.2.1.5.

EXAMPLES:

```
sage: S = SetPartitions(10)
sage: s = S.random_element()                                                  #␣
↪needs sage.symbolic
sage: s.parent() is S                                                         #␣
↪needs sage.symbolic
True
sage: assert s in S, s                                                        #␣
↪needs sage.symbolic

sage: S = SetPartitions(["a", "b", "c"])
sage: s = S.random_element()                                                  #␣
↪needs sage.symbolic
sage: s.parent() is S                                                         #␣
↪needs sage.symbolic
True
```

(continues on next page)

(continued from previous page)

```
sage: assert s in S, s                                                          #␣
↪needs sage.symbolic
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(10))
>>> s = S.random_element()                                                      #␣
↪needs sage.symbolic
>>> s.parent() is S                                                             #␣
↪needs sage.symbolic
True
>>> assert s in S, s                                                            #␣
↪needs sage.symbolic

>>> S = SetPartitions(["a", "b", "c"])
>>> s = S.random_element()                                                      #␣
↪needs sage.symbolic
>>> s.parent() is S                                                             #␣
↪needs sage.symbolic
True
>>> assert s in S, s                                                            #␣
↪needs sage.symbolic
```

class sage.combinat.set_partition.**SetPartitions_setn**(s, k)

Bases: *SetPartitions_set*

Set partitions with a given number of blocks.

cardinality()

The Stirling number of the second kind is the number of partitions of a set of size n into k blocks.

EXAMPLES:

```
sage: SetPartitions(5, 3).cardinality()
25
sage: stirling_number2(5,3)
25
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(5), Integer(3)).cardinality()
25
>>> stirling_number2(Integer(5),Integer(3))
25
```

number_of_blocks()

Return the number of blocks of the set partitions in `self`.

EXAMPLES:

```
sage: SetPartitions(5, 3).number_of_blocks()
3
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(5), Integer(3)).number_of_blocks()
3
```

random_element()

Return a random set partition of `self`.

See https://mathoverflow.net/questions/141999.

EXAMPLES:

```
sage: S = SetPartitions(10, 4)
sage: s = S.random_element()
sage: s.parent() is S
True
sage: assert s in S, s

sage: S = SetPartitions(["a", "b", "c"], 2)
sage: s = S.random_element()
sage: s.parent() is S
True
sage: assert s in S, s
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(10), Integer(4))
>>> s = S.random_element()
>>> s.parent() is S
True
>>> assert s in S, s

>>> S = SetPartitions(["a", "b", "c"], Integer(2))
>>> s = S.random_element()
>>> s.parent() is S
True
>>> assert s in S, s
```

class sage.combinat.set_partition.**SetPartitions_setparts**(*s*, *parts*)

Bases: *SetPartitions_set*

Set partitions with fixed partition sizes corresponding to an integer partition λ.

cardinality()

Return the cardinality of `self`.

This algorithm counts for each block of the partition the number of ways to fill it using values from the set. Then, for each distinct value v of block size, we divide the result by the number of ways to arrange the blocks of size v in the set partition.

For example, if we want to count the number of set partitions of size 13 having [3,3,3,2,2] as underlying partition we compute the number of ways to fill each block of the partition, which is $\binom{13}{3}\binom{10}{3}\binom{7}{3}\binom{4}{2}\binom{2}{2}$ and as we have three blocks of size 3 and two blocks of size 2, we divide the result by 3!2! which gives us 600600.

EXAMPLES:

```
sage: SetPartitions(3, [2,1]).cardinality()
3
```

(continues on next page)

(continued from previous page)

```
sage: SetPartitions(13, Partition([3,3,3,2,2])).cardinality()
600600
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(3), [Integer(2),Integer(1)]).cardinality()
3
>>> SetPartitions(Integer(13), Partition([Integer(3),Integer(3),Integer(3),
↪Integer(2),Integer(2)])).cardinality()
600600
```

random_element()

Return a random set partition of `self`.

ALGORITHM:

Based on the cardinality method. For each block size k_i, we choose a uniformly random subset $X_i \subseteq S_i$ of size k_i of the elements S_i that have not yet been selected. Thus, we define $S_{i+1} = S_i \setminus X_i$ with $S_i = S$ being the defining set. This is not yet proven to be uniformly distributed, but numerical tests show this is likely uniform.

EXAMPLES:

```
sage: S = SetPartitions(10, [4,3,2,1])
sage: s = S.random_element()
sage: s.parent() is S
True
sage: assert s in S, s

sage: S = SetPartitions(["a", "b", "c", "d"], [2,2])
sage: s = S.random_element()
sage: s.parent() is S
True
sage: assert s in S, s
```

```
>>> from sage.all import *
>>> S = SetPartitions(Integer(10), [Integer(4),Integer(3),Integer(2),
↪Integer(1)])
>>> s = S.random_element()
>>> s.parent() is S
True
>>> assert s in S, s

>>> S = SetPartitions(["a", "b", "c", "d"], [Integer(2),Integer(2)])
>>> s = S.random_element()
>>> s.parent() is S
True
>>> assert s in S, s
```

shape()

Return the partition of block sizes of the set partitions in `self`.

EXAMPLES:

```
sage: SetPartitions(5, [2,2,1]).shape()
[2, 2, 1]
```

```
>>> from sage.all import *
>>> SetPartitions(Integer(5), [Integer(2),Integer(2),Integer(1)]).shape()
[2, 2, 1]
```

sage.combinat.set_partition.**cyclic_permutations_of_set_partition**(*set_part*)

Return all combinations of cyclic permutations of each cell of the set partition.

AUTHORS:

- Robert L. Miller

EXAMPLES:

```
sage: from sage.combinat.set_partition import cyclic_permutations_of_set_partition
sage: cyclic_permutations_of_set_partition([[1,2,3,4],[5,6,7]])
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.set_partition import cyclic_permutations_of_set_partition
>>> cyclic_permutations_of_set_partition([[Integer(1),Integer(2),Integer(3),
↪Integer(4)],[Integer(5),Integer(6),Integer(7)]])
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

sage.combinat.set_partition.**cyclic_permutations_of_set_partition_iterator**(*set_part*)

Iterate over all combinations of cyclic permutations of each cell of the set partition.

AUTHORS:

- Robert L. Miller

EXAMPLES:

```
sage: from sage.combinat.set_partition import cyclic_permutations_of_set_
↪partition_iterator
sage: list(cyclic_permutations_of_set_partition_iterator([[1,2,3,4],[5,6,7]]))
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.set_partition import cyclic_permutations_of_set_partition_
↪iterator
>>> list(cyclic_permutations_of_set_partition_iterator([[Integer(1),Integer(2),
↪Integer(3),Integer(4)],[Integer(5),Integer(6),Integer(7)]]))
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

5.1.278 Fast set partition iterators

sage.combinat.set_partition_iterator.**set_partition_iterator**(*base_set*)

A fast iterator for the set partitions of the base set, which returns lists of lists instead of set partitions types.

EXAMPLES:

```
sage: from sage.combinat.set_partition_iterator import set_partition_iterator
sage: list(set_partition_iterator([1,-1,x]))                              #␣
↪needs sage.symbolic
[[[1, -1, x]],
 [[1, -1], [x]],
 [[1, x], [-1]],
 [[1], [-1, x]],
 [[1], [-1], [x]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.set_partition_iterator import set_partition_iterator
```

(continues on next page)

(continued from previous page)

```
>>> list(set_partition_iterator([Integer(1),-Integer(1),x]))                   ␣
↪                # needs sage.symbolic
[[[1, -1, x]],
 [[1, -1], [x]],
 [[1, x], [-1]],
 [[1], [-1, x]],
 [[1], [-1], [x]]]
```

sage.combinat.set_partition_iterator.**set_partition_iterator_blocks**(*base_set*, *k*)

A fast iterator for the set partitions of the base set into the specified number of blocks, which returns lists of lists instead of set partitions types.

EXAMPLES:

```
sage: from sage.combinat.set_partition_iterator import set_partition_iterator_
↪blocks
sage: list(set_partition_iterator_blocks([1,-1,x], 2))                        #␣
↪needs sage.symbolic
[[[1, x], [-1]], [[1], [-1, x]], [[1, -1], [x]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.set_partition_iterator import set_partition_iterator_blocks
>>> list(set_partition_iterator_blocks([Integer(1),-Integer(1),x], Integer(2)))  ␣
↪                         # needs sage.symbolic
[[[1, x], [-1]], [[1], [-1, x]], [[1, -1], [x]]]
```

5.1.279 Ordered Set Partitions

AUTHORS:

- Mike Hansen
- MuPAD-Combinat developers (for algorithms and design inspiration)
- Travis Scrimshaw (2013-02-28): Removed `CombinatorialClass` and added entry point through *OrderedSetPartition*.

class sage.combinat.set_partition_ordered.**OrderedSetPartition**(*parent*, *s*, *check=True*)

Bases: `ClonableArray`

An ordered partition of a set.

An ordered set partition p of a set s is a list of pairwise disjoint nonempty subsets of s such that the union of these subsets is s. These subsets are called the parts of the partition.

We represent an ordered set partition as a list of sets. By extension, an ordered set partition of a nonnegative integer n is the set partition of the integers from 1 to n. The number of ordered set partitions of n is called the n-th ordered Bell number.

There is a natural integer composition associated with an ordered set partition, that is the sequence of sizes of all its parts in order.

The number T_n of ordered set partitions of $\{1, 2, \ldots, n\}$ is the so-called n-th *Fubini number* (also known as the n-th ordered Bell number; see Wikipedia article Ordered Bell number). Its exponential generating function is

$$\sum_n \frac{T_n}{n!} x^n = \frac{1}{2 - e^x}.$$

(See sequence OEIS sequence A000670 in OEIS.)

INPUT:

- `parts` – an object or iterable that defines an ordered set partition (e.g., a list of pairwise disjoint sets) or a packed word (e.g., a list of letters on some alphabet). If there is ambiguity and if the input should be treated as a packed word, the keyword `from_word` should be used.

EXAMPLES:

There are 13 ordered set partitions of $\{1, 2, 3\}$:

```
sage: OrderedSetPartitions(3).cardinality()
13
```

```
>>> from sage.all import *
>>> OrderedSetPartitions(Integer(3)).cardinality()
13
```

Here is the list of them:

```
sage: OrderedSetPartitions(3).list()
[[{1}, {2}, {3}],
 [{1}, {3}, {2}],
 [{2}, {1}, {3}],
 [{3}, {1}, {2}],
 [{2}, {3}, {1}],
 [{3}, {2}, {1}],
 [{1}, {2, 3}],
 [{2}, {1, 3}],
 [{3}, {1, 2}],
 [{1, 2}, {3}],
 [{1, 3}, {2}],
 [{2, 3}, {1}],
 [{1, 2, 3}]]
```

```
>>> from sage.all import *
>>> OrderedSetPartitions(Integer(3)).list()
[[{1}, {2}, {3}],
 [{1}, {3}, {2}],
 [{2}, {1}, {3}],
 [{3}, {1}, {2}],
 [{2}, {3}, {1}],
 [{3}, {2}, {1}],
 [{1}, {2, 3}],
 [{2}, {1, 3}],
 [{3}, {1, 2}],
 [{1, 2}, {3}],
 [{1, 3}, {2}],
 [{2, 3}, {1}],
 [{1, 2, 3}]]
```

There are 12 ordered set partitions of $\{1, 2, 3, 4\}$ whose underlying composition is $[1, 2, 1]$:

```
sage: OrderedSetPartitions(4,[1,2,1]).list()
[[{1}, {2, 3}, {4}],
 [{1}, {2, 4}, {3}],
 [{1}, {3, 4}, {2}],
 [{2}, {1, 3}, {4}],
 [{2}, {1, 4}, {3}],
 [{3}, {1, 2}, {4}],
 [{4}, {1, 2}, {3}],
 [{3}, {1, 4}, {2}],
 [{4}, {1, 3}, {2}],
 [{2}, {3, 4}, {1}],
 [{3}, {2, 4}, {1}],
 [{4}, {2, 3}, {1}]]
```

```
>>> from sage.all import *
>>> OrderedSetPartitions(Integer(4),[Integer(1),Integer(2),Integer(1)]).list()
[[{1}, {2, 3}, {4}],
 [{1}, {2, 4}, {3}],
 [{1}, {3, 4}, {2}],
 [{2}, {1, 3}, {4}],
 [{2}, {1, 4}, {3}],
 [{3}, {1, 2}, {4}],
 [{4}, {1, 2}, {3}],
 [{3}, {1, 4}, {2}],
 [{4}, {1, 3}, {2}],
 [{2}, {3, 4}, {1}],
 [{3}, {2, 4}, {1}],
 [{4}, {2, 3}, {1}]]
```

Since Issue #14140, we can create an ordered set partition directly by `OrderedSetPartition` which creates the parent object by taking the union of the partitions passed in. However it is recommended and (marginally) faster to create the parent first and then create the ordered set partition from that.

```
sage: s = OrderedSetPartition([[1,3],[2,4]]); s
[{1, 3}, {2, 4}]
sage: s.parent()
Ordered set partitions of {1, 2, 3, 4}
```

```
>>> from sage.all import *
>>> s = OrderedSetPartition([[Integer(1),Integer(3)],[Integer(2),Integer(4)]]); s
[{1, 3}, {2, 4}]
>>> s.parent()
Ordered set partitions of {1, 2, 3, 4}
```

We can construct the ordered set partition from a word, which we consider as packed:

```
sage: OrderedSetPartition([2,4,1,2])
[{3}, {1, 4}, {2}]
sage: OrderedSetPartition(from_word=[2,4,1,2])
[{3}, {1, 4}, {2}]
sage: OrderedSetPartition(from_word='bdab')
[{3}, {1, 4}, {2}]
```

```
>>> from sage.all import *
>>> OrderedSetPartition([Integer(2),Integer(4),Integer(1),Integer(2)])
[{3}, {1, 4}, {2}]
>>> OrderedSetPartition(from_word=[Integer(2),Integer(4),Integer(1),Integer(2)])
[{3}, {1, 4}, {2}]
>>> OrderedSetPartition(from_word='bdab')
[{3}, {1, 4}, {2}]
```

Warning

The elements of the underlying set should be hashable.

REFERENCES:

Wikipedia article Ordered_partition_of_a_set

base_set()

Return the base set of `self`.

This is the union of all parts of `self`.

EXAMPLES:

```
sage: OrderedSetPartition([[1], [2,3], [4]]).base_set()
frozenset({1, 2, 3, 4})
sage: OrderedSetPartition([[1,2,3,4]]).base_set()
frozenset({1, 2, 3, 4})
sage: OrderedSetPartition([]).base_set()
frozenset()
```

```
>>> from sage.all import *
>>> OrderedSetPartition([[Integer(1)], [Integer(2),Integer(3)],
→[Integer(4)]]).base_set()
frozenset({1, 2, 3, 4})
>>> OrderedSetPartition([[Integer(1),Integer(2),Integer(3),Integer(4)]]).base_
→set()
frozenset({1, 2, 3, 4})
>>> OrderedSetPartition([]).base_set()
frozenset()
```

base_set_cardinality()

Return the cardinality of the base set of `self`.

This is the sum of the sizes of the parts of `self`.

This is also known as the *size* (sometimes the *weight*) of an ordered set partition.

EXAMPLES:

```
sage: OrderedSetPartition([[1], [2,3], [4]]).base_set_cardinality()
4
sage: OrderedSetPartition([[1,2,3,4]]).base_set_cardinality()
4
```

```
>>> from sage.all import *
>>> OrderedSetPartition([[Integer(1)], [Integer(2),Integer(3)],␣
↪[Integer(4)]]).base_set_cardinality()
4
>>> OrderedSetPartition([[Integer(1),Integer(2),Integer(3),Integer(4)]]).base_
↪set_cardinality()
4
```

static bottom_up_osp(*X*, *comp*)

Return the ordered set partition obtained by listing the elements of the set `X` in increasing order, and placing bars between some of them according to the integer composition `comp` (namely, the bars are placed in such a way that the lengths of the resulting blocks are exactly the entries of `comp`).

INPUT:

- `X` – a finite set (or list or tuple)
- `comp` – a composition whose sum is the size of `X` (can be given as a list or tuple or composition)

EXAMPLES:

```
sage: buo = OrderedSetPartition.bottom_up_osp
sage: buo(Set([1, 4, 7, 9]), [2, 1, 1])
[{1, 4}, {7}, {9}]
sage: buo(Set([1, 4, 7, 9]), [1, 3])
[{1}, {4, 7, 9}]
sage: buo(Set([1, 4, 7, 9]), [1, 1, 1, 1])
[{1}, {4}, {7}, {9}]
sage: buo(range(8), [1, 4, 2, 1])
[{0}, {1, 2, 3, 4}, {5, 6}, {7}]
sage: buo([], [])
[]
```

```
>>> from sage.all import *
>>> buo = OrderedSetPartition.bottom_up_osp
>>> buo(Set([Integer(1), Integer(4), Integer(7), Integer(9)]), [Integer(2),␣
↪Integer(1), Integer(1)])
[{1, 4}, {7}, {9}]
>>> buo(Set([Integer(1), Integer(4), Integer(7), Integer(9)]), [Integer(1),␣
↪Integer(3)])
[{1}, {4, 7, 9}]
>>> buo(Set([Integer(1), Integer(4), Integer(7), Integer(9)]), [Integer(1),␣
↪Integer(1), Integer(1), Integer(1)])
[{1}, {4}, {7}, {9}]
>>> buo(range(Integer(8)), [Integer(1), Integer(4), Integer(2), Integer(1)])
[{0}, {1, 2, 3, 4}, {5, 6}, {7}]
>>> buo([], [])
[]
```

check()

Check that we are a valid ordered set partition.

EXAMPLES:

```
sage: OS = OrderedSetPartitions(4)
sage: s = OS([[1, 3], [2, 4]])
sage: s.check()
```

```
>>> from sage.all import *
>>> OS = OrderedSetPartitions(Integer(4))
>>> s = OS([[Integer(1), Integer(3)], [Integer(2), Integer(4)]])
>>> s.check()
```

complement()

Return the complement of the ordered set partition `self`.

This assumes that `self` is an ordered set partition of an interval of $\mathbf{Z}$.

Let $(P_1, P_2, \ldots, P_k)$ be an ordered set partition of some interval I of $\mathbf{Z}$. Let ω be the unique strictly decreasing bijection $I \to I$. Then, the *complement* of $(P_1, P_2, \ldots, P_k)$ is defined to be the ordered set partition $(\omega(P_1), \omega(P_2), \ldots, \omega(P_k))$.

EXAMPLES:

```
sage: OrderedSetPartition([[1, 2], [3]]).complement()
[{2, 3}, {1}]
sage: OrderedSetPartition([[1, 3], [2]]).complement()
[{1, 3}, {2}]
sage: OrderedSetPartition([[2, 3]]).complement()
[{2, 3}]
sage: OrderedSetPartition([[1, 5], [2, 3], [4]]).complement()
[{1, 5}, {3, 4}, {2}]
sage: OrderedSetPartition([[-1], [-2], [1, 2], [0]]).complement()
[{1}, {2}, {-2, -1}, {0}]
sage: OrderedSetPartition([]).complement()
[]
```

```
>>> from sage.all import *
>>> OrderedSetPartition([[Integer(1), Integer(2)], [Integer(3)]]).complement()
[{2, 3}, {1}]
>>> OrderedSetPartition([[Integer(1), Integer(3)], [Integer(2)]]).complement()
[{1, 3}, {2}]
>>> OrderedSetPartition([[Integer(2), Integer(3)]]).complement()
[{2, 3}]
>>> OrderedSetPartition([[Integer(1), Integer(5)], [Integer(2), Integer(3)],
↪[Integer(4)]]).complement()
[{1, 5}, {3, 4}, {2}]
>>> OrderedSetPartition([[-Integer(1)], [-Integer(2)], [Integer(1),
↪Integer(2)], [Integer(0)]]).complement()
[{1}, {2}, {-2, -1}, {0}]
>>> OrderedSetPartition([]).complement()
[]
```

fatten(*grouping*)

Return the ordered set partition fatter than `self`, obtained by grouping together consecutive parts according to the integer composition `grouping`.

See *finer()* for the definition of "fatter".

INPUT:

- `grouping` – a composition whose sum is the length of `self`

EXAMPLES:

Let us start with the ordered set partition:

```
sage: c = OrderedSetPartition([[2, 5], [1], [3, 4]])
```

```
>>> from sage.all import *
>>> c = OrderedSetPartition([[Integer(2), Integer(5)], [Integer(1)],␣
↪[Integer(3), Integer(4)]])
```

With `grouping` equal to $(1, \dots, 1)$, c is left unchanged:

```
sage: c.fatten(Composition([1,1,1]))
[{2, 5}, {1}, {3, 4}]
```

```
>>> from sage.all import *
>>> c.fatten(Composition([Integer(1),Integer(1),Integer(1)]))
[{2, 5}, {1}, {3, 4}]
```

With `grouping` equal to (ℓ) where ℓ is the length of c, this yields the coarsest ordered set partition above c:

```
sage: c.fatten(Composition([3]))
[{1, 2, 3, 4, 5}]
```

```
>>> from sage.all import *
>>> c.fatten(Composition([Integer(3)]))
[{1, 2, 3, 4, 5}]
```

Other values for `grouping` yield (all the) other ordered set partitions coarser than c:

```
sage: c.fatten(Composition([2,1]))
[{1, 2, 5}, {3, 4}]
sage: c.fatten(Composition([1,2]))
[{2, 5}, {1, 3, 4}]
```

```
>>> from sage.all import *
>>> c.fatten(Composition([Integer(2),Integer(1)]))
[{1, 2, 5}, {3, 4}]
>>> c.fatten(Composition([Integer(1),Integer(2)]))
[{2, 5}, {1, 3, 4}]
```

fatter()

Return the set of ordered set partitions which are fatter than `self`.

See *finer()* for the definition of "fatter".

EXAMPLES:

```
sage: C = OrderedSetPartition([[2, 5], [1], [3, 4]]).fatter()
sage: C.cardinality()
4
```

(continues on next page)

(continued from previous page)

```
sage: sorted(C)
[[{2, 5}, {1}, {3, 4}],
 [{2, 5}, {1, 3, 4}],
 [{1, 2, 5}, {3, 4}],
 [{1, 2, 3, 4, 5}]]

sage: OrderedSetPartition([[4, 9], [-1, 2]]).fatter().list()
[[{4, 9}, {-1, 2}], [{-1, 2, 4, 9}]]
```

```
>>> from sage.all import *
>>> C = OrderedSetPartition([[Integer(2), Integer(5)], [Integer(1)],␣
↪[Integer(3), Integer(4)]]).fatter()
>>> C.cardinality()
4
>>> sorted(C)
[[{2, 5}, {1}, {3, 4}],
 [{2, 5}, {1, 3, 4}],
 [{1, 2, 5}, {3, 4}],
 [{1, 2, 3, 4, 5}]]

>>> OrderedSetPartition([[Integer(4), Integer(9)], [-Integer(1),␣
↪Integer(2)]]).fatter().list()
[[{4, 9}, {-1, 2}], [{-1, 2, 4, 9}]]
```

Some extreme cases:

```
sage: list(OrderedSetPartition([[5]]).fatter())
[[{5}]]
sage: list(Composition([]).fatter())
[[]]
sage: sorted(OrderedSetPartition([[1], [2], [3], [4]]).fatter())
[[{1}, {2}, {3}, {4}],
 [{1}, {2}, {3, 4}],
 [{1}, {2, 3}, {4}],
 [{1}, {2, 3, 4}],
 [{1, 2}, {3}, {4}],
 [{1, 2}, {3, 4}],
 [{1, 2, 3}, {4}],
 [{1, 2, 3, 4}]]
```

```
>>> from sage.all import *
>>> list(OrderedSetPartition([[Integer(5)]]).fatter())
[[{5}]]
>>> list(Composition([]).fatter())
[[]]
>>> sorted(OrderedSetPartition([[Integer(1)], [Integer(2)], [Integer(3)],␣
↪[Integer(4)]]).fatter())
[[{1}, {2}, {3}, {4}],
 [{1}, {2}, {3, 4}],
 [{1}, {2, 3}, {4}],
 [{1}, {2, 3, 4}],
 [{1, 2}, {3}, {4}],
```

(continues on next page)

(continued from previous page)

```
 [{1, 2}, {3, 4}],
 [{1, 2, 3}, {4}],
 [{1, 2, 3, 4}]]
```

finer()

Return the set of ordered set partitions which are finer than `self`.

See *is_finer()* for the definition of "finer".

EXAMPLES:

```
sage: C = OrderedSetPartition([[1, 3], [2]]).finer()
sage: C.cardinality()
3
sage: C.list()
[[{1}, {3}, {2}], [{3}, {1}, {2}], [{1, 3}, {2}]]

sage: OrderedSetPartition([]).finer()
{[]}

sage: W = OrderedSetPartition([[4, 9], [-1, 2]])
sage: W.finer().list()
[[{9}, {4}, {2}, {-1}],
 [{9}, {4}, {-1}, {2}],
 [{9}, {4}, {-1, 2}],
 [{4}, {9}, {2}, {-1}],
 [{4}, {9}, {-1}, {2}],
 [{4}, {9}, {-1, 2}],
 [{4, 9}, {2}, {-1}],
 [{4, 9}, {-1}, {2}],
 [{4, 9}, {-1, 2}]]
```

```
>>> from sage.all import *
>>> C = OrderedSetPartition([[Integer(1), Integer(3)], [Integer(2)]]).finer()
>>> C.cardinality()
3
>>> C.list()
[[{1}, {3}, {2}], [{3}, {1}, {2}], [{1, 3}, {2}]]

>>> OrderedSetPartition([]).finer()
{[]}

>>> W = OrderedSetPartition([[Integer(4), Integer(9)], [-Integer(1),
↪Integer(2)]])
>>> W.finer().list()
[[{9}, {4}, {2}, {-1}],
 [{9}, {4}, {-1}, {2}],
 [{9}, {4}, {-1, 2}],
 [{4}, {9}, {2}, {-1}],
 [{4}, {9}, {-1}, {2}],
 [{4}, {9}, {-1, 2}],
 [{4, 9}, {2}, {-1}],
 [{4, 9}, {-1}, {2}],
```

(continues on next page)

(continued from previous page)

```
[{4, 9}, {-1, 2}]]
```

is_finer(*co2*)

Return `True` if the ordered set partition `self` is finer than the ordered set partition `co2`; otherwise, return `False`.

If A and B are two ordered set partitions of the same set, then A is said to be *finer* than B if B can be obtained from A by (repeatedly) merging consecutive parts. In this case, we say that B is *fatter* than A.

EXAMPLES:

```
sage: A = OrderedSetPartition([[1, 3], [2]])
sage: B = OrderedSetPartition([[1], [3], [2]])
sage: A.is_finer(B)
False
sage: B.is_finer(A)
True
sage: C = OrderedSetPartition([[3], [1], [2]])
sage: A.is_finer(C)
False
sage: C.is_finer(A)
True
sage: OrderedSetPartition([[2], [5], [1], [4]]).is_
↪finer(OrderedSetPartition([[2, 5], [1, 4]]))
True
sage: OrderedSetPartition([[5], [2], [1], [4]]).is_
↪finer(OrderedSetPartition([[2, 5], [1, 4]]))
True
sage: OrderedSetPartition([[2], [1], [5], [4]]).is_
↪finer(OrderedSetPartition([[2, 5], [1, 4]]))
False
sage: OrderedSetPartition([[2, 5, 1], [4]]).is_finer(OrderedSetPartition([[2,␣
↪5], [1, 4]]))
False
```

```
>>> from sage.all import *
>>> A = OrderedSetPartition([[Integer(1), Integer(3)], [Integer(2)]])
>>> B = OrderedSetPartition([[Integer(1)], [Integer(3)], [Integer(2)]])
>>> A.is_finer(B)
False
>>> B.is_finer(A)
True
>>> C = OrderedSetPartition([[Integer(3)], [Integer(1)], [Integer(2)]])
>>> A.is_finer(C)
False
>>> C.is_finer(A)
True
>>> OrderedSetPartition([[Integer(2)], [Integer(5)], [Integer(1)],␣
↪[Integer(4)]]).is_finer(OrderedSetPartition([[Integer(2), Integer(5)],␣
↪[Integer(1), Integer(4)]]))
True
>>> OrderedSetPartition([[Integer(5)], [Integer(2)], [Integer(1)],␣
↪[Integer(4)]]).is_finer(OrderedSetPartition([[Integer(2), Integer(5)],␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(1), Integer(4)]]))
True
>>> OrderedSetPartition([[Integer(2)], [Integer(1)], [Integer(5)],␣
↪[Integer(4)]]).is_finer(OrderedSetPartition([[Integer(2), Integer(5)],␣
↪[Integer(1), Integer(4)]]))
False
>>> OrderedSetPartition([[Integer(2), Integer(5), Integer(1)], [Integer(4)]]).
↪is_finer(OrderedSetPartition([[Integer(2), Integer(5)], [Integer(1),␣
↪Integer(4)]]))
False
```

is_strongly_finer(*co2*)

Return `True` if the ordered set partition `self` is strongly finer than the ordered set partition `co2`; otherwise, return `False`.

If A and B are two ordered set partitions of the same set, then A is said to be *strongly finer* than B if B can be obtained from A by (repeatedly) merging consecutive parts, provided that every time we merge two consecutive parts C_i and C_{i+1}, we have $\max C_i < \min C_{i+1}$. In this case, we say that B is *strongly fatter* than A.

EXAMPLES:

```
sage: A = OrderedSetPartition([[1, 3], [2]])
sage: B = OrderedSetPartition([[1], [3], [2]])
sage: A.is_strongly_finer(B)
False
sage: B.is_strongly_finer(A)
True
sage: C = OrderedSetPartition([[3], [1], [2]])
sage: A.is_strongly_finer(C)
False
sage: C.is_strongly_finer(A)
False
sage: OrderedSetPartition([[2], [5], [1], [4]]).is_strongly_
↪finer(OrderedSetPartition([[2, 5], [1, 4]]))
True
sage: OrderedSetPartition([[5], [2], [1], [4]]).is_strongly_
↪finer(OrderedSetPartition([[2, 5], [1, 4]]))
False
sage: OrderedSetPartition([[2], [1], [5], [4]]).is_strongly_
↪finer(OrderedSetPartition([[2, 5], [1, 4]]))
False
sage: OrderedSetPartition([[2, 5, 1], [4]]).is_strongly_
↪finer(OrderedSetPartition([[2, 5], [1, 4]]))
False
```

```
>>> from sage.all import *
>>> A = OrderedSetPartition([[Integer(1), Integer(3)], [Integer(2)]])
>>> B = OrderedSetPartition([[Integer(1)], [Integer(3)], [Integer(2)]])
>>> A.is_strongly_finer(B)
False
>>> B.is_strongly_finer(A)
True
```

(continues on next page)

(continued from previous page)

```
>>> C = OrderedSetPartition([[Integer(3)], [Integer(1)], [Integer(2)]])
>>> A.is_strongly_finer(C)
False
>>> C.is_strongly_finer(A)
False
>>> OrderedSetPartition([[Integer(2)], [Integer(5)], [Integer(1)],␣
↪[Integer(4)]]).is_strongly_finer(OrderedSetPartition([[Integer(2),␣
↪Integer(5)], [Integer(1), Integer(4)]]))
True
>>> OrderedSetPartition([[Integer(5)], [Integer(2)], [Integer(1)],␣
↪[Integer(4)]]).is_strongly_finer(OrderedSetPartition([[Integer(2),␣
↪Integer(5)], [Integer(1), Integer(4)]]))
False
>>> OrderedSetPartition([[Integer(2)], [Integer(1)], [Integer(5)],␣
↪[Integer(4)]]).is_strongly_finer(OrderedSetPartition([[Integer(2),␣
↪Integer(5)], [Integer(1), Integer(4)]]))
False
>>> OrderedSetPartition([[Integer(2), Integer(5), Integer(1)], [Integer(4)]]).
↪is_strongly_finer(OrderedSetPartition([[Integer(2), Integer(5)],␣
↪[Integer(1), Integer(4)]]))
False
```

length()

Return the number of parts of `self`.

EXAMPLES:

```
sage: OS = OrderedSetPartitions(4)
sage: s = OS([[1, 3], [2, 4]])
sage: s.length()
2
```

```
>>> from sage.all import *
>>> OS = OrderedSetPartitions(Integer(4))
>>> s = OS([[Integer(1), Integer(3)], [Integer(2), Integer(4)]])
>>> s.length()
2
```

number_of_inversions()

Return the number of inversions in `self`.

An inversion of an ordered set partition with blocks $[B_1, B_2, \ldots, B_k]$ is a pair of letters i and j with $i < j$ such that i is minimal in B_m, $j \in B_l$, and $l < m$.

REFERENCES:

- [Wilson2016]

EXAMPLES:

```
sage: OrderedSetPartition([{2,5},{4,6},{1,3}]).number_of_inversions()
5
sage: OrderedSetPartition([{1,3,8},{2,4},{5,6,7}]).number_of_inversions()
3
```

```
>>> from sage.all import *
>>> OrderedSetPartition([{Integer(2),Integer(5)},{Integer(4),Integer(6)},
↪{Integer(1),Integer(3)}]).number_of_inversions()
5
>>> OrderedSetPartition([{Integer(1),Integer(3),Integer(8)},{Integer(2),
↪Integer(4)},{Integer(5),Integer(6),Integer(7)}]).number_of_inversions()
3
```

reversed()

Return the reversal of the ordered set partition `self`.

The *reversal* of an ordered set partition $(P_1, P_2, \ldots, P_k)$ is defined to be the ordered set partition $(P_k, P_{k-1}, \ldots, P_1)$.

EXAMPLES:

```
sage: OrderedSetPartition([[1, 3], [2]]).reversed()
[{2}, {1, 3}]
sage: OrderedSetPartition([[1, 5], [2, 4]]).reversed()
[{2, 4}, {1, 5}]
sage: OrderedSetPartition([[-1], [-2], [3, 4], [0]]).reversed()
[{0}, {3, 4}, {-2}, {-1}]
sage: OrderedSetPartition([]).reversed()
[]
```

```
>>> from sage.all import *
>>> OrderedSetPartition([[Integer(1), Integer(3)], [Integer(2)]]).reversed()
[{2}, {1, 3}]
>>> OrderedSetPartition([[Integer(1), Integer(5)], [Integer(2), Integer(4)]]).
↪reversed()
[{2, 4}, {1, 5}]
>>> OrderedSetPartition([[-Integer(1)], [-Integer(2)], [Integer(3),␣
↪Integer(4)], [Integer(0)]]).reversed()
[{0}, {3, 4}, {-2}, {-1}]
>>> OrderedSetPartition([]).reversed()
[]
```

size()

Return the cardinality of the base set of `self`.

This is the sum of the sizes of the parts of `self`.

This is also known as the *size* (sometimes the *weight*) of an ordered set partition.

EXAMPLES:

```
sage: OrderedSetPartition([[1], [2,3], [4]]).base_set_cardinality()
4
sage: OrderedSetPartition([[1,2,3,4]]).base_set_cardinality()
4
```

```
>>> from sage.all import *
>>> OrderedSetPartition([[Integer(1)], [Integer(2),Integer(3)],␣
↪[Integer(4)]]).base_set_cardinality()
```

(continues on next page)

(continued from previous page)

```
4
>>> OrderedSetPartition([[Integer(1),Integer(2),Integer(3),Integer(4)]]).base_
↪set_cardinality()
4
```

strongly_fatter()

Return the set of ordered set partitions which are strongly fatter than `self`.

See *strongly_finer()* for the definition of "strongly fatter".

EXAMPLES:

```
sage: C = OrderedSetPartition([[2, 5], [1], [3, 4]]).strongly_fatter()
sage: C.cardinality()
2
sage: sorted(C)
[[{2, 5}, {1}, {3, 4}], [{2, 5}, {1, 3, 4}]]

sage: OrderedSetPartition([[4, 9], [-1, 2]]).strongly_fatter().list()
[[{4, 9}, {-1, 2}]]
```

```
>>> from sage.all import *
>>> C = OrderedSetPartition([[Integer(2), Integer(5)], [Integer(1)],
↪[Integer(3), Integer(4)]]).strongly_fatter()
>>> C.cardinality()
2
>>> sorted(C)
[[{2, 5}, {1}, {3, 4}], [{2, 5}, {1, 3, 4}]]

>>> OrderedSetPartition([[Integer(4), Integer(9)], [-Integer(1),
↪Integer(2)]]).strongly_fatter().list()
[[{4, 9}, {-1, 2}]]
```

Some extreme cases:

```
sage: list(OrderedSetPartition([[5]]).strongly_fatter())
[[{5}]]
sage: list(OrderedSetPartition([]).strongly_fatter())
[[]]
sage: sorted(OrderedSetPartition([[1], [2], [3], [4]]).strongly_fatter())
[[{1}, {2}, {3}, {4}],
 [{1}, {2}, {3, 4}],
 [{1}, {2, 3}, {4}],
 [{1}, {2, 3, 4}],
 [{1, 2}, {3}, {4}],
 [{1, 2}, {3, 4}],
 [{1, 2, 3}, {4}],
 [{1, 2, 3, 4}]]
sage: sorted(OrderedSetPartition([[1], [3], [2], [4]]).strongly_fatter())
[[{1}, {3}, {2}, {4}],
 [{1}, {3}, {2, 4}],
 [{1, 3}, {2}, {4}],
 [{1, 3}, {2, 4}]]
```

(continues on next page)

(continued from previous page)

```
sage: sorted(OrderedSetPartition([[4], [1], [5], [3]]).strongly_fatter())
[[{4}, {1}, {5}, {3}], [{4}, {1, 5}, {3}]]
```

```
>>> from sage.all import *
>>> list(OrderedSetPartition([[Integer(5)]]).strongly_fatter())
[[{5}]]
>>> list(OrderedSetPartition([]).strongly_fatter())
[[]]
>>> sorted(OrderedSetPartition([[Integer(1)], [Integer(2)], [Integer(3)],␣
↪[Integer(4)]]).strongly_fatter())
[[{1}, {2}, {3}, {4}],
 [{1}, {2}, {3, 4}],
 [{1}, {2, 3}, {4}],
 [{1}, {2, 3, 4}],
 [{1, 2}, {3}, {4}],
 [{1, 2}, {3, 4}],
 [{1, 2, 3}, {4}],
 [{1, 2, 3, 4}]]
>>> sorted(OrderedSetPartition([[Integer(1)], [Integer(3)], [Integer(2)],␣
↪[Integer(4)]]).strongly_fatter())
[[{1}, {3}, {2}, {4}],
 [{1}, {3}, {2, 4}],
 [{1, 3}, {2}, {4}],
 [{1, 3}, {2, 4}]]
>>> sorted(OrderedSetPartition([[Integer(4)], [Integer(1)], [Integer(5)],␣
↪[Integer(3)]]).strongly_fatter())
[[{4}, {1}, {5}, {3}], [{4}, {1, 5}, {3}]]
```

strongly_finer()

Return the set of ordered set partitions which are strongly finer than `self`.

See *is_strongly_finer()* for the definition of "strongly finer".

EXAMPLES:

```
sage: C = OrderedSetPartition([[1, 3], [2]]).strongly_finer()
sage: C.cardinality()
2
sage: C.list()
[[{1}, {3}, {2}], [{1, 3}, {2}]]

sage: OrderedSetPartition([]).strongly_finer()
{[]}

sage: W = OrderedSetPartition([[4, 9], [-1, 2]])
sage: W.strongly_finer().list()
[[{4}, {9}, {-1}, {2}],
 [{4}, {9}, {-1, 2}],
 [{4, 9}, {-1}, {2}],
 [{4, 9}, {-1, 2}]]
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> C = OrderedSetPartition([[Integer(1), Integer(3)], [Integer(2)]]).
↪strongly_finer()
>>> C.cardinality()
2
>>> C.list()
[[{1}, {3}, {2}], [{1, 3}, {2}]]

>>> OrderedSetPartition([]).strongly_finer()
{[]}

>>> W = OrderedSetPartition([[Integer(4), Integer(9)], [-Integer(1),␣
↪Integer(2)]])
>>> W.strongly_finer().list()
[[{4}, {9}, {-1}, {2}],
 [{4}, {9}, {-1, 2}],
 [{4, 9}, {-1}, {2}],
 [{4, 9}, {-1, 2}]]
```

static sum(*osps*)

Return the concatenation of the given ordered set partitions `osps` (provided they have no elements in common).

INPUT:

- `osps` – list (or iterable) of ordered set partitions

EXAMPLES:

```
sage: OrderedSetPartition.sum([OrderedSetPartition([[4, 1], [3]]),␣
↪OrderedSetPartition([[7], [2]]), OrderedSetPartition([[5, 6]])])
[{1, 4}, {3}, {7}, {2}, {5, 6}]
```

```
>>> from sage.all import *
>>> OrderedSetPartition.sum([OrderedSetPartition([[Integer(4), Integer(1)],␣
↪[Integer(3)]]), OrderedSetPartition([[Integer(7)], [Integer(2)]]),␣
↪OrderedSetPartition([[Integer(5), Integer(6)]])])
[{1, 4}, {3}, {7}, {2}, {5, 6}]
```

Any iterable can be provided as input:

```
sage: OrderedSetPartition.sum([OrderedSetPartition([[2*i,2*i+1]]) for i in [4,
↪1,3]])
[{8, 9}, {2, 3}, {6, 7}]
```

```
>>> from sage.all import *
>>> OrderedSetPartition.sum([OrderedSetPartition([[Integer(2)*i,
↪Integer(2)*i+Integer(1)]]) for i in [Integer(4),Integer(1),Integer(3)]])
[{8, 9}, {2, 3}, {6, 7}]
```

Empty inputs are handled gracefully:

```
sage: OrderedSetPartition.sum([]) == OrderedSetPartition([])
True
```

```
>>> from sage.all import *
>>> OrderedSetPartition.sum([]) == OrderedSetPartition([])
True
```

to_composition()

Return the integer composition whose parts are the sizes of the sets in `self`.

EXAMPLES:

```
sage: S = OrderedSetPartitions(5)
sage: x = S([[3,5,4], [1, 2]])
sage: x.to_composition()
[3, 2]
sage: y = S([[3,1], [2], [5,4]])
sage: y.to_composition()
[2, 1, 2]
```

```
>>> from sage.all import *
>>> S = OrderedSetPartitions(Integer(5))
>>> x = S([[Integer(3),Integer(5),Integer(4)], [Integer(1), Integer(2)]])
>>> x.to_composition()
[3, 2]
>>> y = S([[Integer(3),Integer(1)], [Integer(2)], [Integer(5),Integer(4)]])
>>> y.to_composition()
[2, 1, 2]
```

to_packed_word()

Return the packed word on alphabet $\{1, 2, 3, \ldots\}$ corresponding to `self`.

A *packed word* on alphabet $\{1, 2, 3, \ldots\}$ is any word whose maximum letter is the same as its total number of distinct letters. Let P be an ordered set partition of a set X. The corresponding packed word $w_1 w_2 \cdots w_n$ is constructed by having letter $w_i = j$ if the i-th smallest entry in X occurs in the j-th block of P.

See also

`Word.to_ordered_set_partition()`

Warning

This assumes there is a total order on the underlying set.

EXAMPLES:

```
sage: S = OrderedSetPartitions()
sage: x = S([[3,5], [2], [1,4,6]])
sage: x.to_packed_word()
word: 321313

sage: x = S([['a', 'c', 'e'], ['b', 'd']])
sage: x.to_packed_word()
word: 12121
```

```
>>> from sage.all import *
>>> S = OrderedSetPartitions()
>>> x = S([[Integer(3),Integer(5)], [Integer(2)], [Integer(1),Integer(4),
↪Integer(6)]])
>>> x.to_packed_word()
word: 321313

>>> x = S([['a', 'c', 'e'], ['b', 'd']])
>>> x.to_packed_word()
word: 12121
```

class sage.combinat.set_partition_ordered.**OrderedSetPartitions**(*s*)

Bases: UniqueRepresentation, Parent

Return the combinatorial class of ordered set partitions of s.

The optional argument c, if specified, restricts the parts of the partition to have certain sizes (the entries of c).

EXAMPLES:

```
sage: OS = OrderedSetPartitions([1,2,3,4]); OS
Ordered set partitions of {1, 2, 3, 4}
sage: OS.cardinality()
75
sage: OS.first()
[{1}, {2}, {3}, {4}]
sage: OS.last()
[{1, 2, 3, 4}]
sage: OS.random_element().parent() is OS
True
```

```
>>> from sage.all import *
>>> OS = OrderedSetPartitions([Integer(1),Integer(2),Integer(3),Integer(4)]); OS
Ordered set partitions of {1, 2, 3, 4}
>>> OS.cardinality()
75
>>> OS.first()
[{1}, {2}, {3}, {4}]
>>> OS.last()
[{1, 2, 3, 4}]
>>> OS.random_element().parent() is OS
True
```

```
sage: OS = OrderedSetPartitions([1,2,3,4], [2,2]); OS
Ordered set partitions of {1, 2, 3, 4} into parts of size [2, 2]
sage: OS.cardinality()
6
sage: OS.first()
[{1, 2}, {3, 4}]
sage: OS.last()
[{3, 4}, {1, 2}]
sage: OS.list()
[[{1, 2}, {3, 4}],
 [{1, 3}, {2, 4}],
```

(continues on next page)

(continued from previous page)

```
 [{1, 4}, {2, 3}],
 [{2, 3}, {1, 4}],
 [{2, 4}, {1, 3}],
 [{3, 4}, {1, 2}]]
```

```
>>> from sage.all import *
>>> OS = OrderedSetPartitions([Integer(1),Integer(2),Integer(3),Integer(4)],
↪[Integer(2),Integer(2)]); OS
Ordered set partitions of {1, 2, 3, 4} into parts of size [2, 2]
>>> OS.cardinality()
6
>>> OS.first()
[{1, 2}, {3, 4}]
>>> OS.last()
[{3, 4}, {1, 2}]
>>> OS.list()
[[{1, 2}, {3, 4}],
 [{1, 3}, {2, 4}],
 [{1, 4}, {2, 3}],
 [{2, 3}, {1, 4}],
 [{2, 4}, {1, 3}],
 [{3, 4}, {1, 2}]]
```

```
sage: OS = OrderedSetPartitions("cat")
sage: OS  # random
Ordered set partitions of {'a', 't', 'c'}
sage: sorted(OS.list(), key=str)
[[{'a', 'c', 't'}],
 [{'a', 'c'}, {'t'}],
 [{'a', 't'}, {'c'}],
 [{'a'}, {'c', 't'}],
 [{'a'}, {'c'}, {'t'}],
 [{'a'}, {'t'}, {'c'}],
 [{'c', 't'}, {'a'}],
 [{'c'}, {'a', 't'}],
 [{'c'}, {'a'}, {'t'}],
 [{'c'}, {'t'}, {'a'}],
 [{'t'}, {'a', 'c'}],
 [{'t'}, {'a'}, {'c'}],
 [{'t'}, {'c'}, {'a'}]]
```

```
>>> from sage.all import *
>>> OS = OrderedSetPartitions("cat")
>>> OS  # random
Ordered set partitions of {'a', 't', 'c'}
>>> sorted(OS.list(), key=str)
[[{'a', 'c', 't'}],
 [{'a', 'c'}, {'t'}],
 [{'a', 't'}, {'c'}],
 [{'a'}, {'c', 't'}],
 [{'a'}, {'c'}, {'t'}],
```

(continues on next page)

(continued from previous page)

```
[{'a'}, {'t'}, {'c'}],
[{'c', 't'}, {'a'}],
[{'c'}, {'a', 't'}],
[{'c'}, {'a'}, {'t'}],
[{'c'}, {'t'}, {'a'}],
[{'t'}, {'a', 'c'}],
[{'t'}, {'a'}, {'c'}],
[{'t'}, {'c'}, {'a'}]]
```

Element

alias of *OrderedSetPartition*

from_finite_word(*w*, *check=True*)

Return the unique ordered set partition of $\{1, 2, \ldots, n\}$ corresponding to a word w of length n.

See also

`Word.to_ordered_set_partition()`

EXAMPLES:

```
sage: A = OrderedSetPartitions().from_finite_word('abcabcabd'); A
[{1, 4, 7}, {2, 5, 8}, {3, 6}, {9}]
sage: B = OrderedSetPartitions().from_finite_word([1,2,3,1,2,3,1,2,4])
sage: A == B
True
```

```
>>> from sage.all import *
>>> A = OrderedSetPartitions().from_finite_word('abcabcabd'); A
[{1, 4, 7}, {2, 5, 8}, {3, 6}, {9}]
>>> B = OrderedSetPartitions().from_finite_word([Integer(1),Integer(2),
↪Integer(3),Integer(1),Integer(2),Integer(3),Integer(1),Integer(2),
↪Integer(4)])
>>> A == B
True
```

class sage.combinat.set_partition_ordered.**OrderedSetPartitions_all**

Bases: *OrderedSetPartitions*

Ordered set partitions of $\{1, \ldots, n\}$ for all $n \in \mathbf{Z}_{\geq 0}$.

class Element(*parent*, *s*, *check=True*)

Bases: *OrderedSetPartition*

subset(*size=None*, ***kwargs*)

Return the subset of ordered set partitions of a given size and additional keyword arguments.

EXAMPLES:

```
sage: P = OrderedSetPartitions()
sage: P.subset(4)
Ordered set partitions of {1, 2, 3, 4}
```

```
>>> from sage.all import *
>>> P = OrderedSetPartitions()
>>> P.subset(Integer(4))
Ordered set partitions of {1, 2, 3, 4}
```

class sage.combinat.set_partition_ordered.**OrderedSetPartitions_s**(*s*)

Bases: *OrderedSetPartitions*

Class of ordered partitions of a set S.

cardinality()

EXAMPLES:

```
sage: OrderedSetPartitions(0).cardinality()
1
sage: OrderedSetPartitions(1).cardinality()
1
sage: OrderedSetPartitions(2).cardinality()
3
sage: OrderedSetPartitions(3).cardinality()
13
sage: OrderedSetPartitions([1,2,3]).cardinality()
13
sage: OrderedSetPartitions(4).cardinality()
75
sage: OrderedSetPartitions(5).cardinality()
541
```

```
>>> from sage.all import *
>>> OrderedSetPartitions(Integer(0)).cardinality()
1
>>> OrderedSetPartitions(Integer(1)).cardinality()
1
>>> OrderedSetPartitions(Integer(2)).cardinality()
3
>>> OrderedSetPartitions(Integer(3)).cardinality()
13
>>> OrderedSetPartitions([Integer(1),Integer(2),Integer(3)]).cardinality()
13
>>> OrderedSetPartitions(Integer(4)).cardinality()
75
>>> OrderedSetPartitions(Integer(5)).cardinality()
541
```

class sage.combinat.set_partition_ordered.**OrderedSetPartitions_scomp**(*s*, *comp*)

Bases: *OrderedSetPartitions*

cardinality()

Return the cardinality of `self`.

The number of ordered set partitions of a set of length k with composition shape μ is equal to

$$\frac{k!}{\prod_{\mu_i \neq 0} \mu_i!}.$$

EXAMPLES:

```
sage: OrderedSetPartitions(5,[2,3]).cardinality()
10
sage: OrderedSetPartitions(0, []).cardinality()
1
sage: OrderedSetPartitions(0, [0]).cardinality()
1
sage: OrderedSetPartitions(0, [0,0]).cardinality()
1
sage: OrderedSetPartitions(5, [2,0,3]).cardinality()
10
```

```
>>> from sage.all import *
>>> OrderedSetPartitions(Integer(5),[Integer(2),Integer(3)]).cardinality()
10
>>> OrderedSetPartitions(Integer(0), []).cardinality()
1
>>> OrderedSetPartitions(Integer(0), [Integer(0)]).cardinality()
1
>>> OrderedSetPartitions(Integer(0), [Integer(0),Integer(0)]).cardinality()
1
>>> OrderedSetPartitions(Integer(5), [Integer(2),Integer(0),Integer(3)]).
↪cardinality()
10
```

class sage.combinat.set_partition_ordered.**OrderedSetPartitions_sn**(*s*, *n*)

Bases: *OrderedSetPartitions*

cardinality()

Return the cardinality of `self`.

The number of ordered partitions of a set of size n into k parts is equal to $k!S(n,k)$ where $S(n,k)$ denotes the Stirling number of the second kind.

EXAMPLES:

```
sage: OrderedSetPartitions(4,2).cardinality()
14
sage: OrderedSetPartitions(4,1).cardinality()
1
```

```
>>> from sage.all import *
>>> OrderedSetPartitions(Integer(4),Integer(2)).cardinality()
14
>>> OrderedSetPartitions(Integer(4),Integer(1)).cardinality()
1
```

class sage.combinat.set_partition_ordered.**SplitNK**(*s*, *comp*)

Bases: *OrderedSetPartitions_scomp*

sage.combinat.set_partition_ordered.**multiset_permutation_next_lex**(*l*)

Return the next multiset permutation after `l`.

EXAMPLES:

```
sage: from sage.combinat.set_partition_ordered import multiset_permutation_next_
↪lex
sage: l = [0, 0, 1, 1, 2]
sage: while multiset_permutation_next_lex(l):
....:     print(l)
[0, 0, 1, 2, 1]
[0, 0, 2, 1, 1]
[0, 1, 0, 1, 2]
[0, 1, 0, 2, 1]
[0, 1, 1, 0, 2]
[0, 1, 1, 2, 0]
...
[1, 1, 2, 0, 0]
[1, 2, 0, 0, 1]
[1, 2, 0, 1, 0]
[1, 2, 1, 0, 0]
[2, 0, 0, 1, 1]
[2, 0, 1, 0, 1]
[2, 0, 1, 1, 0]
[2, 1, 0, 0, 1]
[2, 1, 0, 1, 0]
[2, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.set_partition_ordered import multiset_permutation_next_lex
>>> l = [Integer(0), Integer(0), Integer(1), Integer(1), Integer(2)]
>>> while multiset_permutation_next_lex(l):
...     print(l)
[0, 0, 1, 2, 1]
[0, 0, 2, 1, 1]
[0, 1, 0, 1, 2]
[0, 1, 0, 2, 1]
[0, 1, 1, 0, 2]
[0, 1, 1, 2, 0]
...
[1, 1, 2, 0, 0]
[1, 2, 0, 0, 1]
[1, 2, 0, 1, 0]
[1, 2, 1, 0, 0]
[2, 0, 0, 1, 1]
[2, 0, 1, 0, 1]
[2, 0, 1, 1, 0]
[2, 1, 0, 0, 1]
[2, 1, 0, 1, 0]
[2, 1, 1, 0, 0]
```

sage.combinat.set_partition_ordered.**multiset_permutation_to_ordered_set_partition**(l, m)

Convert a multiset permutation to an ordered set partition.

INPUT:

- `l` – a multiset permutation
- `m` – number of parts

EXAMPLES:

```
sage: from sage.combinat.set_partition_ordered import multiset_permutation_to_
↪ordered_set_partition
sage: l = [0, 0, 1, 1, 2]
sage: multiset_permutation_to_ordered_set_partition(l, 3)
[[0, 1], [2, 3], [4]]
```

```
>>> from sage.all import *
>>> from sage.combinat.set_partition_ordered import multiset_permutation_to_
↪ordered_set_partition
>>> l = [Integer(0), Integer(0), Integer(1), Integer(1), Integer(2)]
>>> multiset_permutation_to_ordered_set_partition(l, Integer(3))
[[0, 1], [2, 3], [4]]
```

5.1.280 Symmetric Functions

- *Introduction to Symmetric Functions*
- *Symmetric Functions*
- *Symmetric functions, with their multiple realizations*
- *Classical symmetric functions*
- *Schur symmetric functions*
- *Monomial symmetric functions*
- *Multiplicative symmetric functions*
- *Elementary symmetric functions*
- *Homogeneous symmetric functions*
- *Power sum symmetric functions*
- *Characters of the symmetric group as bases of the symmetric functions*
- *Orthogonal Symmetric Functions*
- *Symplectic Symmetric Functions*
- *Generic dual bases symmetric functions*
- *Symmetric functions defined by orthogonality and triangularity*
- *Kostka-Foulkes Polynomials*
- *Hall-Littlewood Polynomials*
- *Hecke Character Basis*
- *Jack Symmetric Functions*
- *k-Schur Functions*
- *Quotient of symmetric function space by ideal generated by Hall-Littlewood symmetric functions*
- *LLT symmetric functions*
- *Macdonald Polynomials*
- *Non-symmetric Macdonald Polynomials*

- *Witt symmetric functions*

5.1.281 Characters of the symmetric group as bases of the symmetric functions

Just as the Schur functions are the irreducible characters of Gl_n and form a basis of the symmetric functions, the irreducible symmetric group character basis are the irreducible characters of of S_n when the group is realized as the permutation matrices.

REFERENCES:

class sage.combinat.sf.character.**Character_generic**(*Sym*, *basis_name=None*, *prefix=None*, *graded=True*)

Bases: *SymmetricFunctionAlgebra_generic*

class sage.combinat.sf.character.**InducedTrivialCharacterBasis**(*Sym*)

Bases: *Character_generic*

The induced trivial symmetric group character basis of the symmetric functions.

This is a basis of the symmetric functions that has the property that `self(la).character_to_frobenius_image(n)` is equal to `h([n-sum(la)]+la)`.

It has the property that the (outer) structure constants are the analogue of the stable Kronecker coefficients on the complete basis.

This basis is introduced in [OZ2015].

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: h = Sym.h()
sage: ht = SymmetricFunctions(QQ).ht()
sage: st = SymmetricFunctions(QQ).st()
sage: ht(s[2,1])
ht[1, 1] + ht[2, 1] - ht[3]
sage: s(ht[2,1])
s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1] + s[3]
sage: ht(h[2,1])
ht[1] + 2*ht[1, 1] + ht[2, 1]
sage: h(ht[2,1])
h[1] - 2*h[1, 1] + h[2, 1]
sage: st(ht[2,1])
st[] + 2*st[1] + st[1, 1] + 2*st[2] + st[2, 1] + st[3]
sage: ht(st[2,1])
ht[1] - ht[1, 1] + ht[2, 1] - ht[3]
sage: ht[2]*ht[1,1]
ht[1, 1] + 2*ht[1, 1, 1] + ht[2, 1, 1]
sage: h[4,2].kronecker_product(h[4,1,1])
h[2, 2, 1, 1] + 2*h[3, 1, 1, 1] + h[4, 1, 1]
sage: s(st[2,1])
3*s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1]
sage: st(s[2,1])
st[] + 3*st[1] + 2*st[1, 1] + 2*st[2] + st[2, 1]
sage: st[2]*st[1]
st[1] + st[1, 1] + st[2] + st[2, 1] + st[3]
```

(continues on next page)

(continued from previous page)

```
sage: s[4,2].kronecker_product(s[5,1])
s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2] + s[5, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> h = Sym.h()
>>> ht = SymmetricFunctions(QQ).ht()
>>> st = SymmetricFunctions(QQ).st()
>>> ht(s[Integer(2),Integer(1)])
ht[1, 1] + ht[2, 1] - ht[3]
>>> s(ht[Integer(2),Integer(1)])
s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1] + s[3]
>>> ht(h[Integer(2),Integer(1)])
ht[1] + 2*ht[1, 1] + ht[2, 1]
>>> h(ht[Integer(2),Integer(1)])
h[1] - 2*h[1, 1] + h[2, 1]
>>> st(ht[Integer(2),Integer(1)])
st[] + 2*st[1] + st[1, 1] + 2*st[2] + st[2, 1] + st[3]
>>> ht(st[Integer(2),Integer(1)])
ht[1] - ht[1, 1] + ht[2, 1] - ht[3]
>>> ht[Integer(2)]*ht[Integer(1),Integer(1)]
ht[1, 1] + 2*ht[1, 1, 1] + ht[2, 1, 1]
>>> h[Integer(4),Integer(2)].kronecker_product(h[Integer(4),Integer(1),
↪Integer(1)])
h[2, 2, 1, 1] + 2*h[3, 1, 1, 1] + h[4, 1, 1]
>>> s(st[Integer(2),Integer(1)])
3*s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1]
>>> st(s[Integer(2),Integer(1)])
st[] + 3*st[1] + 2*st[1, 1] + 2*st[2] + st[2, 1]
>>> st[Integer(2)]*st[Integer(1)]
st[1] + st[1, 1] + st[2] + st[2, 1] + st[3]
>>> s[Integer(4),Integer(2)].kronecker_product(s[Integer(5),Integer(1)])
s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2] + s[5, 1]
```

class sage.combinat.sf.character.**IrreducibleCharacterBasis**(*Sym*)

Bases: *Character_generic*

The irreducible symmetric group character basis of the symmetric functions.

This is a basis of the symmetric functions that has the property that `self(la).character_to_frobenius_image(n)` is equal to `s([n-sum(la)]+la)`.

It should also have the property that the (outer) structure constants are the analogue of the stable Kronecker coefficients on the Schur basis.

This basis is introduced in [OZ2015].

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: h = Sym.h()
sage: ht = SymmetricFunctions(QQ).ht()
```

(continues on next page)

(continued from previous page)

```
sage: st = SymmetricFunctions(QQ).st()
sage: st(ht[2,1])
st[] + 2*st[1] + st[1, 1] + 2*st[2] + st[2, 1] + st[3]
sage: ht(st[2,1])
ht[1] - ht[1, 1] + ht[2, 1] - ht[3]
sage: s(st[2,1])
3*s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1]
sage: st(s[2,1])
st[] + 3*st[1] + 2*st[1, 1] + 2*st[2] + st[2, 1]
sage: st[2]*st[1]
st[1] + st[1, 1] + st[2] + st[2, 1] + st[3]
sage: s[4,2].kronecker_product(s[5,1])
s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2] + s[5, 1]
sage: st[1,1,1].counit()
-1
sage: all(sum(c*st(la)*st(mu).antipode() for
....:     ((la,mu),c) in st(ga).coproduct())==st(st(ga).counit())
....:     for ga in Partitions(3))
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> h = Sym.h()
>>> ht = SymmetricFunctions(QQ).ht()
>>> st = SymmetricFunctions(QQ).st()
>>> st(ht[Integer(2),Integer(1)])
st[] + 2*st[1] + st[1, 1] + 2*st[2] + st[2, 1] + st[3]
>>> ht(st[Integer(2),Integer(1)])
ht[1] - ht[1, 1] + ht[2, 1] - ht[3]
>>> s(st[Integer(2),Integer(1)])
3*s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1]
>>> st(s[Integer(2),Integer(1)])
st[] + 3*st[1] + 2*st[1, 1] + 2*st[2] + st[2, 1]
>>> st[Integer(2)]*st[Integer(1)]
st[1] + st[1, 1] + st[2] + st[2, 1] + st[3]
>>> s[Integer(4),Integer(2)].kronecker_product(s[Integer(5),Integer(1)])
s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2] + s[5, 1]
>>> st[Integer(1),Integer(1),Integer(1)].counit()
-1
>>> all(sum(c*st(la)*st(mu).antipode() for
...     ((la,mu),c) in st(ga).coproduct())==st(st(ga).counit())
...     for ga in Partitions(Integer(3)))
True
```

5.1.282 Classical symmetric functions

class sage.combinat.sf.classical.**SymmetricFunctionAlgebra_classical**(*Sym*, *basis_name=None*, *prefix=None*, *graded=True*)

Bases: *SymmetricFunctionAlgebra_generic*

The class of classical symmetric functions.

Todo

delete this class once all coercions will be handled by Sage's coercion model

class Element

Bases: *SymmetricFunctionAlgebra_generic_Element*

A symmetric function.

sage.combinat.sf.classical.**init**()

Set up the conversion functions between the classical bases.

EXAMPLES:

```
sage: from sage.combinat.sf.classical import init
sage: sage.combinat.sf.classical.conversion_functions = {}
sage: init()
sage: sage.combinat.sf.classical.conversion_functions[('Schur', 'powersum')]
<built-in function t_SCHUR_POWSYM_symmetrica>
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.classical import init
>>> sage.combinat.sf.classical.conversion_functions = {}
>>> init()
>>> sage.combinat.sf.classical.conversion_functions[('Schur', 'powersum')]
<built-in function t_SCHUR_POWSYM_symmetrica>
```

The following checks if the bug described in Issue #15312 is fixed.

```
sage: change = sage.combinat.sf.classical.conversion_functions[('powersum', 'Schur
→')]
sage: hideme = change({Partition([1]*47):ZZ(1)}) # long time
sage: change({Partition([2,2]):QQ(1)})
s[1, 1, 1, 1] - s[2, 1, 1] + 2*s[2, 2] - s[3, 1] + s[4]
```

```
>>> from sage.all import *
>>> change = sage.combinat.sf.classical.conversion_functions[('powersum', 'Schur
→')]
>>> hideme = change({Partition([Integer(1)]*Integer(47)):ZZ(Integer(1))}) # long␣
→time
>>> change({Partition([Integer(2),Integer(2)]):QQ(Integer(1))})
s[1, 1, 1, 1] - s[2, 1, 1] + 2*s[2, 2] - s[3, 1] + s[4]
```

5.1.283 Generic dual bases symmetric functions

class sage.combinat.sf.dual.**DualBasisFunctor**(*basis*)

Bases: *SymmetricFunctionsFunctor*

A constructor for algebras of symmetric functions constructed by duality.

EXAMPLES:

```
sage: w = SymmetricFunctions(ZZ).witt()
sage: w.dual_basis().construction()
(SymmetricFunctionsFunctor[dual Witt], Integer Ring)
```

```
>>> from sage.all import *
>>> w = SymmetricFunctions(ZZ).witt()
>>> w.dual_basis().construction()
(SymmetricFunctionsFunctor[dual Witt], Integer Ring)
```

class sage.combinat.sf.dual.**SymmetricFunctionAlgebra_dual**(*dual_basis*, *scalar*, *scalar_name*, *basis_name*, *prefix*)

Bases: *SymmetricFunctionAlgebra_classical*

Generic dual basis of a basis of symmetric functions.

INPUT:

- `dual_basis` – a basis of the ring of symmetric functions
- `scalar` – a function z on partitions which determines the scalar product on the power sum basis by $\langle p_\mu, p_\mu \rangle = z(\mu)$. (Independently on the function chosen, the power sum basis will always be orthogonal; the function `scalar` only determines the norms of the basis elements.) This defaults to the function `zee` defined in `sage.combinat.sf.sfa`, that is, the function is defined by:

$$\lambda \mapsto \prod_{i=1}^{\infty} m_i(\lambda)! i^{m_i(\lambda)`},$$

  where $m_i(\lambda)$ means the number of times i appears in λ. This default function gives the standard Hall scalar product on the ring of symmetric functions.
- `scalar_name` – (default: the empty string) a string giving a description of the scalar product specified by the parameter `scalar`
- `basis_name` – (optional) a string to serve as name for the basis to be generated (such as "forgotten" in "the forgotten basis"); don't set it to any of the already existing basis names (such as `homogeneous`, `monomial`, `forgotten`, etc.).
- `prefix` – (default: `'d'` and the prefix for `dual_basis`) a string to use as the symbol for the basis

OUTPUT:

The basis of the ring of symmetric functions dual to the basis `dual_basis` with respect to the scalar product determined by `scalar`.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: f = e.dual_basis(prefix='m', basis_name="Forgotten symmetric functions"); f
Symmetric Functions over Rational Field in the Forgotten symmetric functions basis
sage: TestSuite(f).run(elements=[f[1,1]+2*f[2], f[1]+3*f[1,1]])
sage: TestSuite(f).run() # long time (11s on sage.math, 2011)
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> f = e.dual_basis(prefix='m', basis_name="Forgotten symmetric functions"); f
Symmetric Functions over Rational Field in the Forgotten symmetric functions basis
>>> TestSuite(f).run(elements=[f[Integer(1),Integer(1)]+Integer(2)*f[Integer(2)],␣
```

(continues on next page)

(continued from previous page)

```
↪f[Integer(1)]+Integer(3)*f[Integer(1),Integer(1)]])
>>> TestSuite(f).run() # long time (11s on sage.math, 2011)
```

This class defines canonical coercions between `self` and `self^*`, as follow:

Lookup for the canonical isomorphism from `self` to P (=powersum), and build the adjoint isomorphism from P^* to `self^*`. Since P is self-adjoint for this scalar product, derive an isomorphism from P to `self^*`, and by composition with the above get an isomorphism from `self` to `self^*` (and similarly for the isomorphism `self^*` to `self`).

This should be striped down to just (auto?) defining canonical isomorphism by adjunction (as in MuPAD-Combinat), and let the coercion handle the rest.

Inversions may not be possible if the base ring is not a field:

```
sage: m = SymmetricFunctions(ZZ).m()
sage: h = m.dual_basis(lambda x: 1)
sage: h[2,1]
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(ZZ).m()
>>> h = m.dual_basis(lambda x: Integer(1))
>>> h[Integer(2),Integer(1)]
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

By transitivity, this defines indirect coercions to and from all other bases:

```
sage: s = SymmetricFunctions(QQ['t'].fraction_field()).s()
sage: t = QQ['t'].fraction_field().gen()
sage: zee_hl = lambda x: x.centralizer_size(t=t)
sage: S = s.dual_basis(zee_hl)
sage: S(s([2,1]))
(-t/(t^5-2*t^4+t^3-t^2+2*t-1))*d_s[1, 1, 1] + ((-t^2-1)/(t^5-2*t^4+t^3-t^2+2*t-
↪1))*d_s[2, 1] + (-t/(t^5-2*t^4+t^3-t^2+2*t-1))*d_s[3]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ['t'].fraction_field()).s()
>>> t = QQ['t'].fraction_field().gen()
>>> zee_hl = lambda x: x.centralizer_size(t=t)
>>> S = s.dual_basis(zee_hl)
>>> S(s([Integer(2),Integer(1)]))
(-t/(t^5-2*t^4+t^3-t^2+2*t-1))*d_s[1, 1, 1] + ((-t^2-1)/(t^5-2*t^4+t^3-t^2+2*t-
↪1))*d_s[2, 1] + (-t/(t^5-2*t^4+t^3-t^2+2*t-1))*d_s[3]
```

class Element(*A*, *dictionary=None*, *dual=None*)

Bases: `Element`

An element in the dual basis.

INPUT:

At least one of the following must be specified. The one (if any) which is not provided will be computed.

- `dictionary` – an internal dictionary for the monomials and coefficients of `self`
- `dual` – self as an element of the dual basis

dual()

Return `self` in the dual basis.

OUTPUT:

- the element `self` expanded in the dual basis to `self.parent()`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.parent()
Dual basis to Symmetric Functions over Rational Field in the monomial␣
↪basis
sage: a.dual()
3*m[1, 1, 1] + 2*m[2, 1] + m[3]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).monomial()
>>> zee = sage.combinat.sf.sfa.zee
>>> h = m.dual_basis(scalar=zee)
>>> a = h([Integer(2),Integer(1)])
>>> a.parent()
Dual basis to Symmetric Functions over Rational Field in the monomial␣
↪basis
>>> a.dual()
3*m[1, 1, 1] + 2*m[2, 1] + m[3]
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(zee)
sage: a = h([2,1])+h([3])
sage: a.expand(2)
2*x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + 2*x1^3
sage: a.dual().expand(2)
2*x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + 2*x1^3
sage: a.expand(2,alphabet='y')
2*y0^3 + 3*y0^2*y1 + 3*y0*y1^2 + 2*y1^3
```

(continues on next page)

(continued from previous page)

```
sage: a.expand(2,alphabet='x,y')
2*x^3 + 3*x^2*y + 3*x*y^2 + 2*y^3
sage: h([1]).expand(0)
0
sage: (3*h([])).expand(0)
3
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).monomial()
>>> zee = sage.combinat.sf.sfa.zee
>>> h = m.dual_basis(zee)
>>> a = h([Integer(2),Integer(1)])+h([Integer(3)])
>>> a.expand(Integer(2))
2*x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + 2*x1^3
>>> a.dual().expand(Integer(2))
2*x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + 2*x1^3
>>> a.expand(Integer(2),alphabet='y')
2*y0^3 + 3*y0^2*y1 + 3*y0*y1^2 + 2*y1^3
>>> a.expand(Integer(2),alphabet='x,y')
2*x^3 + 3*x^2*y + 3*x*y^2 + 2*y^3
>>> h([Integer(1)]).expand(Integer(0))
0
>>> (Integer(3)*h([])).expand(Integer(0))
3
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)}p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT: the result of applying omega to `self`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(zee)
sage: hh = SymmetricFunctions(QQ).homogeneous()
sage: hh([2,1]).omega()
h[1, 1, 1] - h[2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: h([2,1]).omega()
d_m[1, 1, 1] - d_m[2, 1]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).monomial()
>>> zee = sage.combinat.sf.sfa.zee
>>> h = m.dual_basis(zee)
>>> hh = SymmetricFunctions(QQ).homogeneous()
>>> hh([Integer(2),Integer(1)]).omega()
h[1, 1, 1] - h[2, 1]
>>> h([Integer(2),Integer(1)]).omega()
d_m[1, 1, 1] - d_m[2, 1]
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT: the result of applying omega to `self`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(zee)
sage: hh = SymmetricFunctions(QQ).homogeneous()
sage: hh([2,1]).omega()
h[1, 1, 1] - h[2, 1]
sage: h([2,1]).omega()
d_m[1, 1, 1] - d_m[2, 1]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).monomial()
>>> zee = sage.combinat.sf.sfa.zee
>>> h = m.dual_basis(zee)
>>> hh = SymmetricFunctions(QQ).homogeneous()
>>> hh([Integer(2),Integer(1)]).omega()
h[1, 1, 1] - h[2, 1]
>>> h([Integer(2),Integer(1)]).omega()
d_m[1, 1, 1] - d_m[2, 1]
```

scalar(*x*)

Return the standard scalar product of `self` and `x`.

INPUT:

- `x` – element of the symmetric functions

OUTPUT: the scalar product between `x` and `self`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.scalar(a)
2
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).monomial()
>>> zee = sage.combinat.sf.sfa.zee
>>> h = m.dual_basis(scalar=zee)
>>> a = h([Integer(2),Integer(1)])
>>> a.scalar(a)
2
```

scalar_hl(*x*)

Return the Hall-Littlewood scalar product of `self` and `x`.

INPUT:

- `x` – element of the same dual basis as `self`

OUTPUT: the Hall-Littlewood scalar product between `x` and `self`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.scalar_hl(a)
(-t - 2)/(t^4 - 2*t^3 + 2*t - 1)
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).monomial()
>>> zee = sage.combinat.sf.sfa.zee
>>> h = m.dual_basis(scalar=zee)
>>> a = h([Integer(2),Integer(1)])
>>> a.scalar_hl(a)
(-t - 2)/(t^4 - 2*t^3 + 2*t - 1)
```

basis_name()

Return the name of the basis of `self`.

This is used for output and, for the classical bases of symmetric functions, to connect this basis with Symmetrica.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: f = Sym.f()
sage: f.basis_name()
'forgotten'
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> f = Sym.f()
>>> f.basis_name()
'forgotten'
```

construction()

Return a pair (F, R), where F is a *SymmetricFunctionsFunctor* and R is a ring, such that F(R) returns self.

EXAMPLES:

```
sage: w = SymmetricFunctions(ZZ).witt()
sage: w.dual_basis().construction()
(SymmetricFunctionsFunctor[dual Witt], Integer Ring)
```

```
>>> from sage.all import *
>>> w = SymmetricFunctions(ZZ).witt()
>>> w.dual_basis().construction()
(SymmetricFunctionsFunctor[dual Witt], Integer Ring)
```

product(*left*, *right*)

Return product of left and right.

Multiplication is done by performing the multiplication in the dual basis of self and then converting back to self.

INPUT:

- left, right – elements of self

OUTPUT: the product of left and right in the basis self

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2])
sage: b = a*a; b # indirect doctest
d_m[2, 2]
sage: b.dual()
6*m[1, 1, 1, 1] + 4*m[2, 1, 1] + 3*m[2, 2] + 2*m[3, 1] + m[4]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).monomial()
>>> zee = sage.combinat.sf.sfa.zee
>>> h = m.dual_basis(scalar=zee)
>>> a = h([Integer(2)])
>>> b = a*a; b # indirect doctest
```

(continues on next page)

(continued from previous page)

```
d_m[2, 2]
>>> b.dual()
6*m[1, 1, 1, 1] + 4*m[2, 1, 1] + 3*m[2, 2] + 2*m[3, 1] + m[4]
```

transition_matrix(*basis*, *n*)

Return the transition matrix between the n-th homogeneous components of `self` and `basis`.

INPUT:

- `basis` – a target basis of the ring of symmetric functions
- `n` – nonnegative integer

OUTPUT:

- A transition matrix from `self` to `basis` for the elements of degree `n`. The indexing order of the rows and columns is the order of `Partitions(n)`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: e = Sym.elementary()
sage: f = e.dual_basis()
sage: f.transition_matrix(s, 5)
[ 1 -1  0  1  0 -1  1]
[-2  1  1 -1 -1  1  0]
[-2  2 -1 -1  1  0  0]
[ 3 -1 -1  1  0  0  0]
[ 3 -2  1  0  0  0  0]
[-4  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
sage: Partitions(5).list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: s(f[2,2,1])
s[3, 2] - 2*s[4, 1] + 3*s[5]
sage: e.transition_matrix(s, 5).inverse().transpose()
[ 1 -1  0  1  0 -1  1]
[-2  1  1 -1 -1  1  0]
[-2  2 -1 -1  1  0  0]
[ 3 -1 -1  1  0  0  0]
[ 3 -2  1  0  0  0  0]
[-4  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.schur()
>>> e = Sym.elementary()
>>> f = e.dual_basis()
>>> f.transition_matrix(s, Integer(5))
[ 1 -1  0  1  0 -1  1]
[-2  1  1 -1 -1  1  0]
[-2  2 -1 -1  1  0  0]
[ 3 -1 -1  1  0  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 3 -2  1  0  0  0  0]
[-4  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
>>> Partitions(Integer(5)).list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
>>> s(f[Integer(2),Integer(2),Integer(1)])
s[3, 2] - 2*s[4, 1] + 3*s[5]
>>> e.transition_matrix(s, Integer(5)).inverse().transpose()
[ 1 -1  0  1  0 -1  1]
[-2  1  1 -1 -1  1  0]
[-2  2 -1 -1  1  0  0]
[ 3 -1 -1  1  0  0  0]
[ 3 -2  1  0  0  0  0]
[-4  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
```

5.1.284 Elementary symmetric functions

class sage.combinat.sf.elementary.**SymmetricFunctionAlgebra_elementary**(*Sym*)

Bases: *SymmetricFunctionAlgebra_multiplicative*

A class for methods for the elementary basis of the symmetric functions.

INPUT:

- `self` – an elementary basis of the symmetric functions
- `Sym` – an instance of the ring of symmetric functions

class Element

Bases: *Element*

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: e([2,1]).expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 3*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: e([1,1,1]).expand(2)
x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + x1^3
sage: e([3]).expand(2)
0
sage: e([2]).expand(3)
x0*x1 + x0*x2 + x1*x2
sage: e([3]).expand(4,alphabet='x,y,z,t')
x*y*z + x*y*t + x*z*t + y*z*t
```

(continues on next page)

(continued from previous page)

```
sage: e([3]).expand(4,alphabet='y')
y0*y1*y2 + y0*y1*y3 + y0*y2*y3 + y1*y2*y3
sage: e([]).expand(2)
1
sage: e([]).expand(0)
1
sage: (3*e([])).expand(0)
3
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> e([Integer(2),Integer(1)]).expand(Integer(3))
x0^2*x1 + x0*x1^2 + x0^2*x2 + 3*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
>>> e([Integer(1),Integer(1),Integer(1)]).expand(Integer(2))
x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + x1^3
>>> e([Integer(3)]).expand(Integer(2))
0
>>> e([Integer(2)]).expand(Integer(3))
x0*x1 + x0*x2 + x1*x2
>>> e([Integer(3)]).expand(Integer(4),alphabet='x,y,z,t')
x*y*z + x*y*t + x*z*t + y*z*t
>>> e([Integer(3)]).expand(Integer(4),alphabet='y')
y0*y1*y2 + y0*y1*y3 + y0*y2*y3 + y1*y2*y3
>>> e([]).expand(Integer(2))
1
>>> e([]).expand(Integer(0))
1
>>> (Integer(3)*e([])).expand(Integer(0))
3
```

exponential_specialization(*t=None*, *q=1*)

Return the exponential specialization of a symmetric function (when $q = 1$), or the q-exponential specialization (when $q \neq 1$).

The *exponential specialization* ex at t is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R. It is defined whenever the base ring K is a $\mathbf{Q}$-algebra and t is an element of R. The easiest way to define it is by specifying its values on the powersum symmetric functions to be $p_1 = t$ and $p_n = 0$ for $n > 1$. Equivalently, on the homogeneous functions it is given by $ex(h_n) = t^n/n!$; see Proposition 7.8.4 of [EnumComb2].

By analogy, the q-exponential specialization is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R that depends on two elements t and q of R for which the elements $1 - q^i$ for all positive integers i are invertible. It can be defined by specifying its values on the complete homogeneous symmetric functions to be

$$ex_q(h_n) = t^n/[n]_q!,$$

where $[n]_q!$ is the q-factorial. Equivalently, for $q \neq 1$ and a homogeneous symmetric function f of degree n, we have

$$ex_q(f) = (1 - q)^n t^n ps_q(f),$$

where $ps_q(f)$ is the stable principal specialization of f (see `principal_specialization()`). (See (7.29) in [EnumComb2].)

The limit of ex_q as $q \to 1$ is ex.

INPUT:

- `t` – (default: `None`) the value to use for t. The default is to create a ring of polynomials in t.
- `q` – (default: 1) the value to use for q. If `q` is `None`, then a ring (or fraction field) of polynomials in `q` is created.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: x = e[3,2]
sage: x.exponential_specialization()
1/12*t^5
sage: x = 5*e[2] + 3*e[1] + 1
sage: x.exponential_specialization(t=var("t"), q=var("q"))                #␣
↪needs sage.symbolic
5*q*t^2/(q + 1) + 3*t + 1
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> x = e[Integer(3),Integer(2)]
>>> x.exponential_specialization()
1/12*t^5
>>> x = Integer(5)*e[Integer(2)] + Integer(3)*e[Integer(1)] + Integer(1)
>>> x.exponential_specialization(t=var("t"), q=var("q"))                #␣
↪needs sage.symbolic
5*q*t^2/(q + 1) + 3*t + 1
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: a = e([2,1]); a
e[2, 1]
sage: a.omega()
e[1, 1, 1] - e[2, 1]
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
```

(continues on next page)

(continued from previous page)

```
>>> a = e([Integer(2),Integer(1)]); a
e[2, 1]
>>> a.omega()
e[1, 1, 1] - e[2, 1]
```

```
sage: h = SymmetricFunctions(QQ).h()
sage: h(e([2,1]).omega())
h[2, 1]
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> h(e([Integer(2),Integer(1)]).omega())
h[2, 1]
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (*length()*) of the partition λ, where λ' denotes the conjugate partition (*conjugate()*) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

omega_involution() is a synonym for the *omega()* method.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: a = e([2,1]); a
e[2, 1]
sage: a.omega()
e[1, 1, 1] - e[2, 1]
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> a = e([Integer(2),Integer(1)]); a
e[2, 1]
>>> a.omega()
e[1, 1, 1] - e[2, 1]
```

```
sage: h = SymmetricFunctions(QQ).h()
sage: h(e([2,1]).omega())
h[2, 1]
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> h(e([Integer(2),Integer(1)]).omega())
h[2, 1]
```

principal_specialization(*n=+Infinity*, *q=None*)

Return the principal specialization of a symmetric function.

The *principal specialization* of order n at q is the ring homomorphism $ps_{n,q}$ from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for $i \in \{1, \ldots, n\}$ and $x_i \mapsto 0$ for $i > n$. Here, q is a given element of R, and we assume that the variables of our symmetric functions are $x_1, x_2, x_3, \ldots$. (To be more precise, $ps_{n,q}$ is a K-algebra homomorphism, where K is the base ring.) See Section 7.8 of [EnumComb2].

The *stable principal specialization* at q is the ring homomorphism ps_q from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for all i. This is well-defined only if the resulting infinite sums converge; thus, in particular, setting $q = 1$ in the stable principal specialization is an invalid operation.

INPUT:

- `n` – (default: `infinity`) a nonnegative integer or `infinity`, specifying whether to compute the principal specialization of order `n` or the stable principal specialization.
- `q` – (default: `None`) the value to use for q; the default is to create a ring of polynomials in `q` (or a field of rational functions in `q`) over the given coefficient ring.

We use the formulas from Proposition 7.8.3 of [EnumComb2] (using Gaussian binomial coefficients $\binom{u}{v}_q$):

$$ps_{n,q}(e_\lambda) = \prod_i q^{\binom{\lambda_i}{2}} \binom{n}{\lambda_i}_q,$$

$$ps_{n,1}(e_\lambda) = \prod_i \binom{n}{\lambda_i},$$

$$ps_q(e_\lambda) = \prod_i q^{\binom{\lambda_i}{2}} / \prod_{j=1}^{\lambda_i} (1 - q^j).$$

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: x = e[3,1]
sage: x.principal_specialization(3)
q^5 + q^4 + q^3
sage: x = 5*e[1,1,1] + 3*e[2,1] + 1
sage: x.principal_specialization(3)
5*q^6 + 18*q^5 + 36*q^4 + 44*q^3 + 36*q^2 + 18*q + 6
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> x = e[Integer(3),Integer(1)]
>>> x.principal_specialization(Integer(3))
q^5 + q^4 + q^3
>>> x = Integer(5)*e[Integer(1),Integer(1),Integer(1)] +␣
↪Integer(3)*e[Integer(2),Integer(1)] + Integer(1)
>>> x.principal_specialization(Integer(3))
5*q^6 + 18*q^5 + 36*q^4 + 44*q^3 + 36*q^2 + 18*q + 6
```

By default, we return a rational functions in q. Sometimes it is better to obtain an element of the symbolic ring:

```
sage: x.principal_specialization(q=var("q"))                                    #␣
↪needs sage.symbolic
-3*q/((q^2 - 1)*(q - 1)^2) - 5/(q - 1)^3 + 1
```

```
>>> from sage.all import *
>>> x.principal_specialization(q=var("q"))                                  #␣
↪needs sage.symbolic
-3*q/((q^2 - 1)*(q - 1)^2) - 5/(q - 1)^3 + 1
```

verschiebung(*n*)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n} e_{r/n}$$

(where h is the complete homogeneous basis, p is the powersum basis, and e is the elementary basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for "shift") endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Frobenius operator (see *frobenius()* for its definition) with respect to the Hall scalar product (*scalar()*).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, ..., \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, ..., \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
```

(continues on next page)

(continued from previous page)

```
sage: e[3].verschiebung(2)
0
sage: e[4].verschiebung(4)
-e[1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> e = Sym.e()
>>> e[Integer(3)].verschiebung(Integer(2))
0
>>> e[Integer(4)].verschiebung(Integer(4))
-e[1]
```

The Verschiebung endomorphisms are multiplicative:

```
sage: all( all( e(lam).verschiebung(2) * e(mu).verschiebung(2)
....:             == (e(lam) * e(mu)).verschiebung(2)
....:             for mu in Partitions(4) )
....:        for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( e(lam).verschiebung(Integer(2)) * e(mu).
↪verschiebung(Integer(2))
...             == (e(lam) * e(mu)).verschiebung(Integer(2))
...             for mu in Partitions(Integer(4)) )
...        for lam in Partitions(Integer(4)) )
True
```

coproduct_on_generators(*i*)

Return the coproduct on `self[i]`.

INPUT:

- `self` – an elementary basis of the symmetric functions
- `i` – nonnegative integer

OUTPUT: the coproduct on the elementary generator $e(i)$

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: e = Sym.elementary()
sage: e.coproduct_on_generators(2)
e[] # e[2] + e[1] # e[1] + e[2] # e[]
sage: e.coproduct_on_generators(0)
e[] # e[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> e = Sym.elementary()
>>> e.coproduct_on_generators(Integer(2))
e[] # e[2] + e[1] # e[1] + e[2] # e[]
```

(continues on next page)

(continued from previous page)

```
>>> e.coproduct_on_generators(Integer(0))
e[] # e[]
```

5.1.285 Hall-Littlewood Polynomials

Notation used in the definitions follows mainly [Mac1995].

class sage.combinat.sf.hall_littlewood.**HallLittlewood**(*Sym*, *t*)

Bases: UniqueRepresentation

The family of Hall-Littlewood symmetric function bases.

The Hall-Littlewood symmetric functions are a family of symmetric functions that depend on a parameter t.

INPUT:

By default the parameter for these functions is t, and whatever the parameter is, it must be in the base ring.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).hall_littlewood(1)
Hall-Littlewood polynomials with t=1 over Rational Field
sage: SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood()
Hall-Littlewood polynomials over Fraction Field of Univariate Polynomial Ring in␣
↪t over Rational Field
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).hall_littlewood(Integer(1))
Hall-Littlewood polynomials with t=1 over Rational Field
>>> SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood()
Hall-Littlewood polynomials over Fraction Field of Univariate Polynomial Ring in␣
↪t over Rational Field
```

P()

Return the algebra of symmetric functions in the Hall-Littlewood P basis. This is the same as the HL basis in John Stembridge's SF examples file.

INPUT:

- self – a class of Hall-Littlewood symmetric function bases

OUTPUT: the class of the Hall-Littlewood P basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P(); HLP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood P basis
sage: SP = Sym.hall_littlewood(t=-1).P(); SP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood P with t=-1 basis
sage: s = Sym.schur()
sage: s(HLP([2,1]))
(-t^2-t)*s[1, 1, 1] + s[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP = Sym.hall_littlewood().P(); HLP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood P basis
>>> SP = Sym.hall_littlewood(t=-Integer(1)).P(); SP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood P with t=-1 basis
>>> s = Sym.schur()
>>> s(HLP([Integer(2),Integer(1)]))
(-t^2-t)*s[1, 1, 1] + s[2, 1]
```

The Hall-Littlewood polynomials in the P basis at $t = 0$ are the Schur functions:

```
sage: Sym = SymmetricFunctions(QQ)
sage: HLP = Sym.hall_littlewood(t=0).P()
sage: s = Sym.schur()
sage: s(HLP([2,1])) == s([2,1])
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> HLP = Sym.hall_littlewood(t=Integer(0)).P()
>>> s = Sym.schur()
>>> s(HLP([Integer(2),Integer(1)])) == s([Integer(2),Integer(1)])
True
```

The Hall-Littlewood polynomials in the P basis at $t = 1$ are the monomial symmetric functions:

```
sage: Sym = SymmetricFunctions(QQ)
sage: HLP = Sym.hall_littlewood(t=1).P()
sage: m = Sym.monomial()
sage: m(HLP([2,2,1])) == m([2,2,1])
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> HLP = Sym.hall_littlewood(t=Integer(1)).P()
>>> m = Sym.monomial()
>>> m(HLP([Integer(2),Integer(2),Integer(1)])) == m([Integer(2),Integer(2),
↪Integer(1)])
True
```

We end with some examples of coercions between:

1. Hall-Littlewood P basis.
2. Hall-Littlewood polynomials in the Q basis
3. Hall-Littlewood polynomials in the Q' basis (via the Schurs)
4. Classical symmetric functions

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP  = Sym.hall_littlewood().P()
```

(continues on next page)

(continued from previous page)

```
sage: HLQ  = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: s = Sym.schur()
sage: p = Sym.power()
sage: HLP(HLQ([2])) # indirect doctest
(-t+1)*HLP[2]
sage: HLP(HLQp([2]))
t*HLP[1, 1] + HLP[2]
sage: HLP(s([2]))
t*HLP[1, 1] + HLP[2]
sage: HLP(p([2]))
(t-1)*HLP[1, 1] + HLP[2]
sage: s = HLQp.symmetric_function_ring().s()
sage: HLQp.transition_matrix(s,3)
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
sage: s.transition_matrix(HLP,3)
[      1       t     t^3]
[      0       1 t^2 + t]
[      0       0       1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP  = Sym.hall_littlewood().P()
>>> HLQ  = Sym.hall_littlewood().Q()
>>> HLQp = Sym.hall_littlewood().Qp()
>>> s = Sym.schur()
>>> p = Sym.power()
>>> HLP(HLQ([Integer(2)])) # indirect doctest
(-t+1)*HLP[2]
>>> HLP(HLQp([Integer(2)]))
t*HLP[1, 1] + HLP[2]
>>> HLP(s([Integer(2)]))
t*HLP[1, 1] + HLP[2]
>>> HLP(p([Integer(2)]))
(t-1)*HLP[1, 1] + HLP[2]
>>> s = HLQp.symmetric_function_ring().s()
>>> HLQp.transition_matrix(s,Integer(3))
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
>>> s.transition_matrix(HLP,Integer(3))
[      1       t     t^3]
[      0       1 t^2 + t]
[      0       0       1]
```

The method *sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element.hl_creation_operator()* is a creation operator for the Q basis:

```
sage: HLQp[1].hl_creation_operator([3]).hl_creation_operator([3])
HLQp[3, 3, 1]
```

```
>>> from sage.all import *
>>> HLQp[Integer(1)].hl_creation_operator([Integer(3)]).hl_creation_
↪operator([Integer(3)])
HLQp[3, 3, 1]
```

Transitions between bases with the parameter t specialized:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['y','z']))
sage: (y,z) = Sym.base_ring().gens()
sage: HLy = Sym.hall_littlewood(t=y)
sage: HLz = Sym.hall_littlewood(t=z)
sage: Qpy = HLy.Qp()
sage: Qpz = HLz.Qp()
sage: s = Sym.schur()
sage: s( Qpy[3,1] + z*Qpy[2,2] )
z*s[2, 2] + (y*z+1)*s[3, 1] + (y^2*z+y)*s[4]
sage: s( Qpy[3,1] + y*Qpz[2,2] )
y*s[2, 2] + (y*z+1)*s[3, 1] + (y*z^2+y)*s[4]
sage: s( Qpy[3,1] + y*Qpy[2,2] )
y*s[2, 2] + (y^2+1)*s[3, 1] + (y^3+y)*s[4]

sage: Qy = HLy.Q()
sage: Qz = HLz.Q()
sage: Py = HLy.P()
sage: Pz = HLz.P()
sage: Pz(Qpy[2,1])
(y*z^3+z^2+z)*HLP[1, 1, 1] + (y*z+1)*HLP[2, 1] + y*HLP[3]
sage: Pz(Qz[2,1])
(z^2-2*z+1)*HLP[2, 1]
sage: Qz(Py[2])
-((y-z)/(z^3-z^2-z+1))*HLQ[1, 1] + (1/(-z+1))*HLQ[2]
sage: Qy(Pz[2])
((y-z)/(y^3-y^2-y+1))*HLQ[1, 1] + (1/(-y+1))*HLQ[2]
sage: Qy.hall_littlewood_family() == HLy
True
sage: Qy.hall_littlewood_family() == HLz
False
sage: Qz.symmetric_function_ring() == Qy.symmetric_function_ring()
True

sage: Sym = SymmetricFunctions(FractionField(QQ['q']))
sage: q = Sym.base_ring().gen()
sage: HL = Sym.hall_littlewood(t=q)
sage: HLQp = HL.Qp()
sage: HLQ = HL.Q()
sage: HLP = HL.P()
sage: s = Sym.schur()
sage: s(HLQp[3,2].plethysm((1-q)*s[1]))/(1-q)^2
(-q^5-q^4)*s[1, 1, 1, 1, 1] + (q^3+q^2)*s[2, 1, 1, 1] - q*s[2, 2, 1] - q*s[3,␣
↪1, 1] + s[3, 2]
sage: s(HLP[3,2])
(-q^5-q^4)*s[1, 1, 1, 1, 1] + (q^3+q^2)*s[2, 1, 1, 1] - q*s[2, 2, 1] - q*s[3,␣
↪1, 1] + s[3, 2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['y','z']))
>>> (y,z) = Sym.base_ring().gens()
>>> HLy = Sym.hall_littlewood(t=y)
>>> HLz = Sym.hall_littlewood(t=z)
>>> Qpy = HLy.Qp()
>>> Qpz = HLz.Qp()
>>> s = Sym.schur()
>>> s( Qpy[Integer(3),Integer(1)] + z*Qpy[Integer(2),Integer(2)] )
z*s[2, 2] + (y*z+1)*s[3, 1] + (y^2*z+y)*s[4]
>>> s( Qpy[Integer(3),Integer(1)] + y*Qpz[Integer(2),Integer(2)] )
y*s[2, 2] + (y*z+1)*s[3, 1] + (y*z^2+y)*s[4]
>>> s( Qpy[Integer(3),Integer(1)] + y*Qpy[Integer(2),Integer(2)] )
y*s[2, 2] + (y^2+1)*s[3, 1] + (y^3+y)*s[4]

>>> Qy = HLy.Q()
>>> Qz = HLz.Q()
>>> Py = HLy.P()
>>> Pz = HLz.P()
>>> Pz(Qpy[Integer(2),Integer(1)])
(y*z^3+z^2+z)*HLP[1, 1, 1] + (y*z+1)*HLP[2, 1] + y*HLP[3]
>>> Pz(Qz[Integer(2),Integer(1)])
(z^2-2*z+1)*HLP[2, 1]
>>> Qz(Py[Integer(2)])
-((y-z)/(z^3-z^2-z+1))*HLQ[1, 1] + (1/(-z+1))*HLQ[2]
>>> Qy(Pz[Integer(2)])
((y-z)/(y^3-y^2-y+1))*HLQ[1, 1] + (1/(-y+1))*HLQ[2]
>>> Qy.hall_littlewood_family() == HLy
True
>>> Qy.hall_littlewood_family() == HLz
False
>>> Qz.symmetric_function_ring() == Qy.symmetric_function_ring()
True

>>> Sym = SymmetricFunctions(FractionField(QQ['q']))
>>> q = Sym.base_ring().gen()
>>> HL = Sym.hall_littlewood(t=q)
>>> HLQp = HL.Qp()
>>> HLQ = HL.Q()
>>> HLP = HL.P()
>>> s = Sym.schur()
>>> s(HLQp[Integer(3),Integer(2)].plethysm((Integer(1)-q)*s[Integer(1)]))/
↪(Integer(1)-q)**Integer(2)
(-q^5-q^4)*s[1, 1, 1, 1, 1] + (q^3+q^2)*s[2, 1, 1, 1] - q*s[2, 2, 1] - q*s[3,␣
↪1, 1] + s[3, 2]
>>> s(HLP[Integer(3),Integer(2)])
(-q^5-q^4)*s[1, 1, 1, 1, 1] + (q^3+q^2)*s[2, 1, 1, 1] - q*s[2, 2, 1] - q*s[3,␣
↪1, 1] + s[3, 2]
```

The P and Q-Schur at $t = -1$ indexed by strict partitions are a basis for the space algebraically generated by the odd power sum symmetric functions:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q']))
sage: SP = Sym.hall_littlewood(t=-1).P()
sage: SQ = Sym.hall_littlewood(t=-1).Q()
sage: p = Sym.power()
sage: SP(SQ[3,2,1])
8*HLP[3, 2, 1]
sage: SP(SQ[2,2,1])
0
sage: p(SP[3,2,1])
1/45*p[1, 1, 1, 1, 1, 1] - 1/9*p[3, 1, 1, 1] - 1/9*p[3, 3] + 1/5*p[5, 1]
sage: SP(p[3,3])
-4*HLP[3, 2, 1] + 2*HLP[4, 2] - 2*HLP[5, 1] + HLP[6]
sage: SQ( SQ[1]*SQ[3] -2*(1-q)*SQ[4] )
HLQ[3, 1] + 2*q*HLQ[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q']))
>>> SP = Sym.hall_littlewood(t=-Integer(1)).P()
>>> SQ = Sym.hall_littlewood(t=-Integer(1)).Q()
>>> p = Sym.power()
>>> SP(SQ[Integer(3),Integer(2),Integer(1)])
8*HLP[3, 2, 1]
>>> SP(SQ[Integer(2),Integer(2),Integer(1)])
0
>>> p(SP[Integer(3),Integer(2),Integer(1)])
1/45*p[1, 1, 1, 1, 1, 1] - 1/9*p[3, 1, 1, 1] - 1/9*p[3, 3] + 1/5*p[5, 1]
>>> SP(p[Integer(3),Integer(3)])
-4*HLP[3, 2, 1] + 2*HLP[4, 2] - 2*HLP[5, 1] + HLP[6]
>>> SQ( SQ[Integer(1)]*SQ[Integer(3)] -Integer(2)*(Integer(1)-
↪q)*SQ[Integer(4)] )
HLQ[3, 1] + 2*q*HLQ[4]
```

Q()

Return the algebra of symmetric functions in Hall-Littlewood Q basis. This is the same as the Q basis in John Stembridge's SF examples file.

More extensive examples can be found in the documentation for the Hall-Littlewood P basis.

INPUT:

- `self` – a class of Hall-Littlewood symmetric function bases

OUTPUT: the class of the Hall-Littlewood Q basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLQ = Sym.hall_littlewood().Q(); HLQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood Q basis
sage: SQ = SymmetricFunctions(QQ).hall_littlewood(t=-1).Q(); SQ
Symmetric Functions over Rational Field in the Hall-Littlewood Q with t=-1␣
↪basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLQ = Sym.hall_littlewood().Q(); HLQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood Q basis
>>> SQ = SymmetricFunctions(QQ).hall_littlewood(t=-Integer(1)).Q(); SQ
Symmetric Functions over Rational Field in the Hall-Littlewood Q with t=-1␣
↪basis
```

Qp()

Return the algebra of symmetric functions in Hall-Littlewood Q' (Qp) basis. This is dual to the Hall-Littlewood P basis with respect to the standard scalar product.

More extensive examples can be found in the documentation for the Hall-Littlewood P basis.

INPUT:

- `self` – a class of Hall-Littlewood symmetric function bases

OUTPUT: the class of the Hall-Littlewood Qp-basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLQp = Sym.hall_littlewood().Qp(); HLQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood Qp basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLQp = Sym.hall_littlewood().Qp(); HLQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood Qp basis
```

base_ring()

Return the base ring of the symmetric functions where the Hall-Littlewood symmetric functions live.

INPUT:

- `self` – a class of Hall-Littlewood symmetric function bases

EXAMPLES:

```
sage: HL = SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood(t=1)
sage: HL.base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> HL = SymmetricFunctions(QQ['t'].fraction_field()).hall_
↪littlewood(t=Integer(1))
>>> HL.base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

symmetric_function_ring()

Return the ring of symmetric functions associated to the class of Hall-Littlewood symmetric functions.

INPUT:

- `self` – a class of Hall-Littlewood symmetric function bases

EXAMPLES:

```
sage: HL = SymmetricFunctions(FractionField(QQ['t'])).hall_littlewood()
sage: HL.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field
```

```
>>> from sage.all import *
>>> HL = SymmetricFunctions(FractionField(QQ['t'])).hall_littlewood()
>>> HL.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field
```

class sage.combinat.sf.hall_littlewood.**HallLittlewood_generic**(*hall_littlewood*)

Bases: *SymmetricFunctionAlgebra_generic*

A class with methods for working with Hall-Littlewood symmetric functions which are common to all bases.

INPUT:

- `self` – a Hall-Littlewood symmetric function basis
- `hall_littlewood` – a class of Hall-Littlewood bases

class Element

Bases: *SymmetricFunctionAlgebra_generic_Element*

Methods for elements of a Hall-Littlewood basis that are common to all bases.

expand(*n*, *alphabet='x'*)

Expand the symmetric function as a symmetric polynomial in `n` variables.

INPUT:

- `self` – an element of a Hall-Littlewood basis
- `n` – positive integer
- `alphabet` – string representing a variable name (default: `'x'`)

OUTPUT: a symmetric polynomial of `self` in `n` variables

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: HLP([2]).expand(2)
x0^2 + (-t + 1)*x0*x1 + x1^2
sage: HLQ([2]).expand(2)
(-t + 1)*x0^2 + (t^2 - 2*t + 1)*x0*x1 + (-t + 1)*x1^2
sage: HLQp([2]).expand(2)
x0^2 + x0*x1 + x1^2
sage: HLQp([2]).expand(2, 'y')
y0^2 + y0*y1 + y1^2
sage: HLQp([2]).expand(1)
x^2
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP = Sym.hall_littlewood().P()
>>> HLQ = Sym.hall_littlewood().Q()
>>> HLQp = Sym.hall_littlewood().Qp()
>>> HLP([Integer(2)]).expand(Integer(2))
x0^2 + (-t + 1)*x0*x1 + x1^2
>>> HLQ([Integer(2)]).expand(Integer(2))
(-t + 1)*x0^2 + (t^2 - 2*t + 1)*x0*x1 + (-t + 1)*x1^2
>>> HLQp([Integer(2)]).expand(Integer(2))
x0^2 + x0*x1 + x1^2
>>> HLQp([Integer(2)]).expand(Integer(2), 'y')
y0^2 + y0*y1 + y1^2
>>> HLQp([Integer(2)]).expand(Integer(1))
x^2
```

scalar(*x*, *zee=None*)

Return standard scalar product between `self` and `x`.

This is the default implementation that converts both `self` and `x` into Schur functions and performs the scalar product that basis.

The Hall-Littlewood P basis is dual to the Qp basis with respect to this scalar product.

INPUT:

- `self` – an element of a Hall-Littlewood basis
- `x` – another symmetric element of the symmetric functions

OUTPUT: the scalar product between `self` and `x`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: HLP([2]).scalar(HLQp([2]))
1
sage: HLP([2]).scalar(HLQp([1,1]))
0
sage: HLP([2]).scalar(HLQ([2]), lambda mu: mu.centralizer_size(t = HLP.t))
1
sage: HLP([2]).scalar(HLQ([1,1]), lambda mu: mu.centralizer_size(t = HLP.
↪t))
0
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP = Sym.hall_littlewood().P()
>>> HLQ = Sym.hall_littlewood().Q()
>>> HLQp = Sym.hall_littlewood().Qp()
>>> HLP([Integer(2)]).scalar(HLQp([Integer(2)]))
1
>>> HLP([Integer(2)]).scalar(HLQp([Integer(1),Integer(1)]))
0
```

(continues on next page)

(continued from previous page)

```
>>> HLP([Integer(2)]).scalar(HLQ([Integer(2)]), lambda mu: mu.centralizer_
↪size(t = HLP.t))
1
>>> HLP([Integer(2)]).scalar(HLQ([Integer(1),Integer(1)]), lambda mu: mu.
↪centralizer_size(t = HLP.t))
0
```

scalar_hl(*x*, *t=None*)

Return the Hall-Littlewood (with parameter `t`) scalar product of `self` and `x`.

The Hall-Littlewood scalar product is defined in Macdonald's book [Mac1995]. The power sum basis is orthogonal and $\langle p_\mu, p_\mu \rangle = z_\mu \prod_i 1/(1 - t^{\mu_i})$

The Hall-Littlewood P basis is dual to the Q basis with respect to this scalar product.

INPUT:

- `self` – an element of a Hall-Littlewood basis
- `x` – another symmetric element of the symmetric functions
- `t` – an optional parameter, if this parameter is not specified then the value of the `t` from the basis is used in the calculation

OUTPUT: the Hall-Littlewood scalar product between `self` and `x`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLP([2]).scalar_hl(HLQ([2]))
1
sage: HLP([2]).scalar_hl(HLQ([1,1]))
0
sage: HLQ([2]).scalar_hl(HLQ([2]))
-t + 1
sage: HLQ([2]).scalar_hl(HLQ([1,1]))
0
sage: HLP([2]).scalar_hl(HLP([2]))
-1/(t - 1)
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP = Sym.hall_littlewood().P()
>>> HLQ = Sym.hall_littlewood().Q()
>>> HLP([Integer(2)]).scalar_hl(HLQ([Integer(2)]))
1
>>> HLP([Integer(2)]).scalar_hl(HLQ([Integer(1),Integer(1)]))
0
>>> HLQ([Integer(2)]).scalar_hl(HLQ([Integer(2)]))
-t + 1
>>> HLQ([Integer(2)]).scalar_hl(HLQ([Integer(1),Integer(1)]))
0
>>> HLP([Integer(2)]).scalar_hl(HLP([Integer(2)]))
-1/(t - 1)
```

construction()

Return a pair (F, R), where F is a SymmetricFunctionsFunctor and R is a ring, such that F(R) returns self.

EXAMPLES:

```
sage: P = SymmetricFunctions(QQ).hall_littlewood(t=2).P()
sage: P.construction()
(SymmetricFunctionsFunctor[Hall-Littlewood P with t=2], Rational Field)
```

```
>>> from sage.all import *
>>> P = SymmetricFunctions(QQ).hall_littlewood(t=Integer(2)).P()
>>> P.construction()
(SymmetricFunctionsFunctor[Hall-Littlewood P with t=2], Rational Field)
```

hall_littlewood_family()

The family of Hall-Littlewood bases associated to self.

INPUT:

- self – a Hall-Littlewood symmetric function basis

OUTPUT: the class of Hall-Littlewood bases

EXAMPLES:

```
sage: HLP = SymmetricFunctions(FractionField(QQ['t'])).hall_littlewood(1).P()
sage: HLP.hall_littlewood_family()
Hall-Littlewood polynomials with t=1 over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> HLP = SymmetricFunctions(FractionField(QQ['t'])).hall_
↪littlewood(Integer(1)).P()
>>> HLP.hall_littlewood_family()
Hall-Littlewood polynomials with t=1 over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field
```

product(*left*, *right*)

Multiply an element of the Hall-Littlewood symmetric function basis self and another symmetric function

Convert to the Schur basis, do the multiplication there, and convert back to self basis.

INPUT:

- self – a Hall-Littlewood symmetric function basis
- left – an element of the basis self
- right – another symmetric function

OUTPUT: the product of left and right expanded in the basis self

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLP([2])^2 # indirect doctest
(t+1)*HLP[2, 2] + (-t+1)*HLP[3, 1] + HLP[4]
```

(continues on next page)

(continued from previous page)

```
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQ([2])^2 # indirect doctest
HLQ[2, 2] + (-t+1)*HLQ[3, 1] + (-t+1)*HLQ[4]

sage: HLQp = Sym.hall_littlewood().Qp()
sage: HLQp([2])^2 # indirect doctest
HLQp[2, 2] + (-t+1)*HLQp[3, 1] + (-t+1)*HLQp[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP = Sym.hall_littlewood().P()
>>> HLP([Integer(2)])**Integer(2) # indirect doctest
(t+1)*HLP[2, 2] + (-t+1)*HLP[3, 1] + HLP[4]

>>> HLQ = Sym.hall_littlewood().Q()
>>> HLQ([Integer(2)])**Integer(2) # indirect doctest
HLQ[2, 2] + (-t+1)*HLQ[3, 1] + (-t+1)*HLQ[4]

>>> HLQp = Sym.hall_littlewood().Qp()
>>> HLQp([Integer(2)])**Integer(2) # indirect doctest
HLQp[2, 2] + (-t+1)*HLQp[3, 1] + (-t+1)*HLQp[4]
```

transition_matrix(*basis*, *n*)

Return the transitions matrix between `self` and `basis` for the homogeneous component of degree `n`.

INPUT:

- `self` – a Hall-Littlewood symmetric function basis
- `basis` – another symmetric function basis
- `n` – nonnegative integer representing the degree

OUTPUT:

- Returns a $r \times r$ matrix of elements of the base ring of `self` where r is the number of partitions of `n`. The entry corresponding to row μ, column ν is the coefficient of `basis` (ν) in `self` (μ)

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: s   = Sym.schur()
sage: HLP.transition_matrix(s, 4)
[             1             -t              0            t^2           -t^3]
[             0              1             -t             -t      t^3 + t^2]
[             0              0              1             -t            t^3]
[             0              0              0              1 -t^3 - t^2 - t]
[             0              0              0              0              1]
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQ.transition_matrix(s,3)
[                        -t + 1                        t^2 - t                ␣
↪     -t^3 + t^2]
[                             0                  t^2 - 2*t + 1           -t^4␣
```

(continues on next page)

(continued from previous page)

```
↪+ t^3 + t^2 - t]
[                              0                              0 -t^6 + t^5 + t^
↪4 - t^2 - t + 1]
sage: HLQp = Sym.hall_littlewood().Qp()
sage: HLQp.transition_matrix(s,3)
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP = Sym.hall_littlewood().P()
>>> s   = Sym.schur()
>>> HLP.transition_matrix(s, Integer(4))
[             1             -t              0            t^2           -t^3]
[             0              1             -t             -t      t^3 + t^2]
[             0              0              1             -t            t^3]
[             0              0              0              1 -t^3 - t^2 - t]
[             0              0              0              0              1]
>>> HLQ = Sym.hall_littlewood().Q()
>>> HLQ.transition_matrix(s,Integer(3))
[                        -t + 1                        t^2 - t                    ␣
↪     -t^3 + t^2]
[                             0                  t^2 - 2*t + 1           -t^4␣
↪+ t^3 + t^2 - t]
[                              0                              0 -t^6 + t^5 + t^
↪4 - t^2 - t + 1]
>>> HLQp = Sym.hall_littlewood().Qp()
>>> HLQp.transition_matrix(s,Integer(3))
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
```

class sage.combinat.sf.hall_littlewood.**HallLittlewood_p**(*hall_littlewood*)

Bases: *HallLittlewood_generic*

A class representing the Hall-Littlewood P basis of symmetric functions

class Element

Bases: *Element*

class sage.combinat.sf.hall_littlewood.**HallLittlewood_q**(*hall_littlewood*)

Bases: *HallLittlewood_generic*

The Q basis is defined as a normalization of the P basis.

INPUT:

- self – an instance of the Hall-Littlewood P basis
- hall_littlewood – a class for the family of Hall-Littlewood bases

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: Q = Sym.hall_littlewood().Q()
sage: TestSuite(Q).run(skip=['_test_associativity', '_test_distributivity', '_
↪test_prod']) # products are too expensive, long time (3s on sage.math, 2012)
sage: TestSuite(Q).run(elements = [Q.t*Q[1,1]+Q[2], Q[1]+(1+Q.t)*Q[1,1]])  # long␣
↪time (depends on previous)

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: s = Sym.schur(); p = Sym.power()
sage: HLQ( HLP([2,1]) + HLP([3]) )
(1/(t^2-2*t+1))*HLQ[2, 1] - (1/(t-1))*HLQ[3]
sage: HLQ(HLQp([2])) # indirect doctest
(t/(t^3-t^2-t+1))*HLQ[1, 1] - (1/(t-1))*HLQ[2]
sage: HLQ(s([2]))
(t/(t^3-t^2-t+1))*HLQ[1, 1] - (1/(t-1))*HLQ[2]
sage: HLQ(p([2]))
(1/(t^2-1))*HLQ[1, 1] - (1/(t-1))*HLQ[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> Q = Sym.hall_littlewood().Q()
>>> TestSuite(Q).run(skip=['_test_associativity', '_test_distributivity', '_test_
↪prod']) # products are too expensive, long time (3s on sage.math, 2012)
>>> TestSuite(Q).run(elements = [Q.t*Q[Integer(1),Integer(1)]+Q[Integer(2)],␣
↪Q[Integer(1)]+(Integer(1)+Q.t)*Q[Integer(1),Integer(1)]])  # long time (depends␣
↪on previous)

>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP = Sym.hall_littlewood().P()
>>> HLQ = Sym.hall_littlewood().Q()
>>> HLQp = Sym.hall_littlewood().Qp()
>>> s = Sym.schur(); p = Sym.power()
>>> HLQ( HLP([Integer(2),Integer(1)]) + HLP([Integer(3)]) )
(1/(t^2-2*t+1))*HLQ[2, 1] - (1/(t-1))*HLQ[3]
>>> HLQ(HLQp([Integer(2)])) # indirect doctest
(t/(t^3-t^2-t+1))*HLQ[1, 1] - (1/(t-1))*HLQ[2]
>>> HLQ(s([Integer(2)]))
(t/(t^3-t^2-t+1))*HLQ[1, 1] - (1/(t-1))*HLQ[2]
>>> HLQ(p([Integer(2)]))
(1/(t^2-1))*HLQ[1, 1] - (1/(t-1))*HLQ[2]
```

class Element

Bases: `Element`

class `sage.combinat.sf.hall_littlewood.`**`HallLittlewood_qp`**(*hall_littlewood*)

Bases: `HallLittlewood_generic`

The Hall-Littlewood Qp basis is calculated through the symmetrica library (see the function `HallLittlewood_qp._to_s()`).

INPUT:

- `self` – an instance of the Hall-Littlewood P basis
- `hall_littlewood` – a class for the family of Hall-Littlewood bases

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: Qp = Sym.hall_littlewood().Q()
sage: TestSuite(Qp).run(skip=['_test_passociativity', '_test_distributivity', '_
↪test_prod']) # products are too expensive, long time (3s on sage.math, 2012)
sage: TestSuite(Qp).run(elements = [Qp.t*Qp[1,1]+Qp[2], Qp[1]+(1+Qp.t)*Qp[1,1]])
↪# long time (depends on previous)

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: s = Sym.schur(); p = Sym.power()
sage: HLQp(HLP([2])) # indirect doctest
-t*HLQp[1, 1] + (t^2+1)*HLQp[2]
sage: HLQp(s(HLQ([2]))) # work around bug reported in issue #12969
(t^2-t)*HLQp[1, 1] + (-t^3+t^2-t+1)*HLQp[2]
sage: HLQp(s([2]))
HLQp[2]
sage: HLQp(p([2]))
-HLQp[1, 1] + (t+1)*HLQp[2]
sage: s = HLQp.symmetric_function_ring().s()
sage: HLQp.transition_matrix(s,3)
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
sage: s.transition_matrix(HLP,3)
[      1       t     t^3]
[      0       1 t^2 + t]
[      0       0       1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> Qp = Sym.hall_littlewood().Q()
>>> TestSuite(Qp).run(skip=['_test_passociativity', '_test_distributivity', '_
↪test_prod']) # products are too expensive, long time (3s on sage.math, 2012)
>>> TestSuite(Qp).run(elements = [Qp.t*Qp[Integer(1),Integer(1)]+Qp[Integer(2)],␣
↪Qp[Integer(1)]+(Integer(1)+Qp.t)*Qp[Integer(1),Integer(1)]])   # long time␣
↪(depends on previous)

>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLP = Sym.hall_littlewood().P()
>>> HLQ = Sym.hall_littlewood().Q()
>>> HLQp = Sym.hall_littlewood().Qp()
>>> s = Sym.schur(); p = Sym.power()
>>> HLQp(HLP([Integer(2)])) # indirect doctest
-t*HLQp[1, 1] + (t^2+1)*HLQp[2]
>>> HLQp(s(HLQ([Integer(2)]))) # work around bug reported in issue #12969
(t^2-t)*HLQp[1, 1] + (-t^3+t^2-t+1)*HLQp[2]
```

(continues on next page)

(continued from previous page)

```
>>> HLQp(s([Integer(2)]))
HLQp[2]
>>> HLQp(p([Integer(2)]))
-HLQp[1, 1] + (t+1)*HLQp[2]
>>> s = HLQp.symmetric_function_ring().s()
>>> HLQp.transition_matrix(s,Integer(3))
[       1        0        0]
[       t        1        0]
[   t^3 t^2 + t        1]
>>> s.transition_matrix(HLP,Integer(3))
[       1        t      t^3]
[       0        1 t^2 + t]
[       0        0        1]
```

class Element

Bases: *Element*

5.1.286 Hecke Character Basis

The basis of symmetric functions given by characters of the Hecke algebra (of type A).

AUTHORS:

- Travis Scrimshaw (2017-08): Initial version

class sage.combinat.sf.hecke.**HeckeCharacter**(*sym*, *q*)

Bases: *SymmetricFunctionAlgebra_multiplicative*

Basis of the symmetric functions that gives the characters of the Hecke algebra in analogy to the Frobenius formula for the symmetric group.

Consider the Hecke algebra $H_n(q)$ with quadratic relations

$$T_i^2 = (q-1)T_i + q.$$

Let μ be a partition of n with length ℓ. The character χ of a $H_n(q)$-representation is completely determined by the elements T_{γ_μ}, where

$$\gamma_\mu = (\mu_1, \ldots, 1)(\mu_2 + \mu_1, \ldots, 1 + \mu_1) \cdots (n, \ldots, 1 + \sum_{i<\ell} \mu_i),$$

(written in cycle notation). We define a basis of the symmetric functions by

$$\bar{q}_\mu = \sum_{\lambda \vdash n} \chi^\lambda(T_{\gamma_\mu}) s_\lambda.$$

INPUT:

- `sym` – the ring of symmetric functions
- `q` – (default: `'q'`) the parameter q

EXAMPLES:

```
sage: q = ZZ['q'].fraction_field().gen()
sage: Sym = SymmetricFunctions(q.parent())
sage: qbar = Sym.hecke_character(q)
```

(continues on next page)

(continued from previous page)

```
sage: qbar[2] * qbar[3] * qbar[3,1]
qbar[3, 3, 2, 1]

sage: s = Sym.s()
sage: s(qbar([2]))
-s[1, 1] + q*s[2]
sage: s(qbar([4]))
-s[1, 1, 1, 1] + q*s[2, 1, 1] - q^2*s[3, 1] + q^3*s[4]
sage: qbar(s[2])
(1/(q+1))*qbar[1, 1] + (1/(q+1))*qbar[2]
sage: qbar(s[1,1])
(q/(q+1))*qbar[1, 1] - (1/(q+1))*qbar[2]

sage: s(qbar[2,1])
-s[1, 1, 1] + (q-1)*s[2, 1] + q*s[3]
sage: qbar(s[2,1])
(q/(q^2+q+1))*qbar[1, 1, 1] + ((q-1)/(q^2+q+1))*qbar[2, 1]
 - (1/(q^2+q+1))*qbar[3]
```

```
>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> Sym = SymmetricFunctions(q.parent())
>>> qbar = Sym.hecke_character(q)
>>> qbar[Integer(2)] * qbar[Integer(3)] * qbar[Integer(3),Integer(1)]
qbar[3, 3, 2, 1]

>>> s = Sym.s()
>>> s(qbar([Integer(2)]))
-s[1, 1] + q*s[2]
>>> s(qbar([Integer(4)]))
-s[1, 1, 1, 1] + q*s[2, 1, 1] - q^2*s[3, 1] + q^3*s[4]
>>> qbar(s[Integer(2)])
(1/(q+1))*qbar[1, 1] + (1/(q+1))*qbar[2]
>>> qbar(s[Integer(1),Integer(1)])
(q/(q+1))*qbar[1, 1] - (1/(q+1))*qbar[2]

>>> s(qbar[Integer(2),Integer(1)])
-s[1, 1, 1] + (q-1)*s[2, 1] + q*s[3]
>>> qbar(s[Integer(2),Integer(1)])
(q/(q^2+q+1))*qbar[1, 1, 1] + ((q-1)/(q^2+q+1))*qbar[2, 1]
 - (1/(q^2+q+1))*qbar[3]
```

We compute character tables for Hecke algebras, which correspond to the transition matrix from the $\bar{q}$ basis to the Schur basis:

```
sage: qbar.transition_matrix(s, 1)
[1]
sage: qbar.transition_matrix(s, 2)
[ q -1]
[ 1  1]
sage: qbar.transition_matrix(s, 3)
[  q^2    -q      1]
```

(continues on next page)

(continued from previous page)

```
[    q q - 1     -1]
[    1     2      1]
sage: qbar.transition_matrix(s, 4)
[      q^3      -q^2          0          q         -1]
[      q^2    q^2 - q        -q     -q + 1          1]
[      q^2 q^2 - 2*q   q^2 + 1   -2*q + 1          1]
[        q   2*q - 1     q - 1      q - 2         -1]
[        1         3         2          3          1]
```

```
>>> from sage.all import *
>>> qbar.transition_matrix(s, Integer(1))
[1]
>>> qbar.transition_matrix(s, Integer(2))
[ q -1]
[ 1  1]
>>> qbar.transition_matrix(s, Integer(3))
[  q^2     -q      1]
[    q q - 1     -1]
[    1     2      1]
>>> qbar.transition_matrix(s, Integer(4))
[      q^3      -q^2          0          q         -1]
[      q^2    q^2 - q        -q     -q + 1          1]
[      q^2 q^2 - 2*q   q^2 + 1   -2*q + 1          1]
[        q   2*q - 1     q - 1      q - 2         -1]
[        1         3         2          3          1]
```

We can do computations with a specialized q to a generic element of the base ring. We compute some examples with $q = 2$:

```
sage: qbar = Sym.qbar(q=2)
sage: s = Sym.schur()
sage: qbar(s[2,1])
2/7*qbar[1, 1, 1] + 1/7*qbar[2, 1] - 1/7*qbar[3]
sage: s(qbar[2,1])
-s[1, 1, 1] + s[2, 1] + 2*s[3]
```

```
>>> from sage.all import *
>>> qbar = Sym.qbar(q=Integer(2))
>>> s = Sym.schur()
>>> qbar(s[Integer(2),Integer(1)])
2/7*qbar[1, 1, 1] + 1/7*qbar[2, 1] - 1/7*qbar[3]
>>> s(qbar[Integer(2),Integer(1)])
-s[1, 1, 1] + s[2, 1] + 2*s[3]
```

REFERENCES:

- [Ram1991]
- [RR1997]

construction()

Return a pair `(F, R)`, where `F` is a `SymmetricFunctionsFunctor` and R is a ring, such that `F(R)` returns `self`.

EXAMPLES:

```
sage: qbar = SymmetricFunctions(QQ['q']).qbar('q')
sage: qbar.construction()
(SymmetricFunctionsFunctor[Hecke character with q=q],
 Univariate Polynomial Ring in q over Rational Field)
```

```
>>> from sage.all import *
>>> qbar = SymmetricFunctions(QQ['q']).qbar('q')
>>> qbar.construction()
(SymmetricFunctionsFunctor[Hecke character with q=q],
 Univariate Polynomial Ring in q over Rational Field)
```

coproduct_on_generators(*r*)

Return the coproduct on the generator $\bar{q}_r$ of `self`.

Define the coproduct on $\bar{q}_r$ by

$$\Delta(\bar{q}_r) = \bar{q}_0 \otimes \bar{q}_r + (q-1) \sum_{j=1}^{r-1} \bar{q}_j \otimes \bar{q}_{r-j} + \bar{q}_r \otimes \bar{q}_0.$$

EXAMPLES:

```
sage: q = ZZ['q'].fraction_field().gen()
sage: Sym = SymmetricFunctions(q.parent())
sage: qbar = Sym.hecke_character()
sage: s = Sym.s()
sage: qbar[2].coproduct()
qbar[] # qbar[2] + (q-1)*qbar[1] # qbar[1] + qbar[2] # qbar[]
```

```
>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> Sym = SymmetricFunctions(q.parent())
>>> qbar = Sym.hecke_character()
>>> s = Sym.s()
>>> qbar[Integer(2)].coproduct()
qbar[] # qbar[2] + (q-1)*qbar[1] # qbar[1] + qbar[2] # qbar[]
```

5.1.287 Homogeneous symmetric functions

By this we mean the basis formed of the complete homogeneous symmetric functions h_λ, not an arbitrary graded basis.

class sage.combinat.sf.homogeneous.**SymmetricFunctionAlgebra_homogeneous**(*Sym*)

Bases: *SymmetricFunctionAlgebra_multiplicative*

A class of methods specific to the homogeneous basis of symmetric functions.

INPUT:

- `self` – a homogeneous basis of symmetric functions
- `Sym` – an instance of the ring of symmetric functions

class Element

Bases: *Element*

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: h = SymmetricFunctions(QQ).h()
sage: h([3]).expand(2)
x0^3 + x0^2*x1 + x0*x1^2 + x1^3
sage: h([1,1,1]).expand(2)
x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + x1^3
sage: h([2,1]).expand(3)
x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3 + 2*x0^2*x2 + 3*x0*x1*x2 + 2*x1^2*x2
↪+ 2*x0*x2^2 + 2*x1*x2^2 + x2^3
sage: h([3]).expand(2,alphabet='y')
y0^3 + y0^2*y1 + y0*y1^2 + y1^3
sage: h([3]).expand(2,alphabet='x,y')
x^3 + x^2*y + x*y^2 + y^3
sage: h([3]).expand(3,alphabet='x,y,z')
x^3 + x^2*y + x*y^2 + y^3 + x^2*z + x*y*z + y^2*z + x*z^2 + y*z^2 + z^3
sage: (h([]) + 2*h([1])).expand(3)
2*x0 + 2*x1 + 2*x2 + 1
sage: h([1]).expand(0)
0
sage: (3*h([])).expand(0)
3
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> h([Integer(3)]).expand(Integer(2))
x0^3 + x0^2*x1 + x0*x1^2 + x1^3
>>> h([Integer(1),Integer(1),Integer(1)]).expand(Integer(2))
x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + x1^3
>>> h([Integer(2),Integer(1)]).expand(Integer(3))
x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3 + 2*x0^2*x2 + 3*x0*x1*x2 + 2*x1^2*x2
↪+ 2*x0*x2^2 + 2*x1*x2^2 + x2^3
>>> h([Integer(3)]).expand(Integer(2),alphabet='y')
y0^3 + y0^2*y1 + y0*y1^2 + y1^3
>>> h([Integer(3)]).expand(Integer(2),alphabet='x,y')
x^3 + x^2*y + x*y^2 + y^3
>>> h([Integer(3)]).expand(Integer(3),alphabet='x,y,z')
x^3 + x^2*y + x*y^2 + y^3 + x^2*z + x*y*z + y^2*z + x*z^2 + y*z^2 + z^3
>>> (h([]) + Integer(2)*h([Integer(1)])).expand(Integer(3))
2*x0 + 2*x1 + 2*x2 + 1
>>> h([Integer(1)]).expand(Integer(0))
0
>>> (Integer(3)*h([])).expand(Integer(0))
3
```

exponential_specialization(*t=None*, *q=1*)

Return the exponential specialization of a symmetric function (when $q = 1$), or the q-exponential specialization (when $q \neq 1$).

The *exponential specialization* ex at t is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R. It is defined whenever the base ring K is a **Q**-algebra and t is an element of R. The easiest way to define it is by specifying its values on the powersum symmetric functions to be $p_1 = t$ and $p_n = 0$ for $n > 1$. Equivalently, on the homogeneous functions it is given by $ex(h_n) = t^n/n!$; see Proposition 7.8.4 of [EnumComb2].

By analogy, the q-exponential specialization is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R that depends on two elements t and q of R for which the elements $1 - q^i$ for all positive integers i are invertible. It can be defined by specifying its values on the complete homogeneous symmetric functions to be

$$ex_q(h_n) = t^n/[n]_q!,$$

where $[n]_q!$ is the q-factorial. Equivalently, for $q \neq 1$ and a homogeneous symmetric function f of degree n, we have

$$ex_q(f) = (1-q)^n t^n ps_q(f),$$

where $ps_q(f)$ is the stable principal specialization of f (see `principal_specialization()`). (See (7.29) in [EnumComb2].)

The limit of ex_q as $q \to 1$ is ex.

INPUT:

- `t` – (default: `None`) the value to use for t; the default is to create a ring of polynomials in `t`
- `q` – (default: 1) the value to use for q; if `q` is `None`, then a ring (or fraction field) of polynomials in `q` is created

EXAMPLES:

```
sage: h = SymmetricFunctions(QQ).h()
sage: x = h[5,3]
sage: x.exponential_specialization()
1/720*t^8
sage: factorial(5)*factorial(3)
720

sage: x = 5*h[1,1,1] + 3*h[2,1] + 1
sage: x.exponential_specialization()
13/2*t^3 + 1
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> x = h[Integer(5),Integer(3)]
>>> x.exponential_specialization()
1/720*t^8
>>> factorial(Integer(5))*factorial(Integer(3))
720

>>> x = Integer(5)*h[Integer(1),Integer(1),Integer(1)] +␣
↪Integer(3)*h[Integer(2),Integer(1)] + Integer(1)
>>> x.exponential_specialization()
13/2*t^3 + 1
```

We also support the q-exponential_specialization:

```
sage: factor(h[3].exponential_specialization(q=var("q"), t=var("t")))    #␣
↪needs sage.symbolic
t^3/((q^2 + q + 1)*(q + 1))
```

```
>>> from sage.all import *
>>> factor(h[Integer(3)].exponential_specialization(q=var("q"), t=var("t
↪")))    # needs sage.symbolic
t^3/((q^2 + q + 1)*(q + 1))
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (*length()*) of the partition λ, where λ' denotes the conjugate partition (*conjugate()*) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

omega_involution() is a synonym for the *omega()* method.

OUTPUT: the image of `self` under the omega automorphism

EXAMPLES:

```
sage: h = SymmetricFunctions(QQ).h()
sage: a = h([2,1]); a
h[2, 1]
sage: a.omega()
h[1, 1, 1] - h[2, 1]
sage: e = SymmetricFunctions(QQ).e()
sage: e(h([2,1]).omega())
e[2, 1]
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> a = h([Integer(2),Integer(1)]); a
h[2, 1]
>>> a.omega()
h[1, 1, 1] - h[2, 1]
>>> e = SymmetricFunctions(QQ).e()
>>> e(h([Integer(2),Integer(1)]).omega())
e[2, 1]
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT: the image of `self` under the omega automorphism

EXAMPLES:

```
sage: h = SymmetricFunctions(QQ).h()
sage: a = h([2,1]); a
h[2, 1]
sage: a.omega()
h[1, 1, 1] - h[2, 1]
sage: e = SymmetricFunctions(QQ).e()
sage: e(h([2,1]).omega())
e[2, 1]
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> a = h([Integer(2),Integer(1)]); a
h[2, 1]
>>> a.omega()
h[1, 1, 1] - h[2, 1]
>>> e = SymmetricFunctions(QQ).e()
>>> e(h([Integer(2),Integer(1)]).omega())
e[2, 1]
```

principal_specialization(*n=+Infinity*, *q=None*)

Return the principal specialization of a symmetric function.

The *principal specialization* of order n at q is the ring homomorphism $ps_{n,q}$ from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for $i \in \{1, \dots, n\}$ and $x_i \mapsto 0$ for $i > n$. Here, q is a given element of R, and we assume that the variables of our symmetric functions are $x_1, x_2, x_3, \ldots$. (To be more precise, $ps_{n,q}$ is a K-algebra homomorphism, where K is the base ring.) See Section 7.8 of [EnumComb2].

The *stable principal specialization* at q is the ring homomorphism ps_q from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for all i. This is well-defined only if the resulting infinite sums converge; thus, in particular, setting $q = 1$ in the stable principal specialization is an invalid operation.

INPUT:

- `n` – (default: `infinity`) a nonnegative integer or `infinity`, specifying whether to compute the principal specialization of order `n` or the stable principal specialization.
- `q` – (default: `None`) the value to use for q; the default is to create a ring of polynomials in `q` (or a field of rational functions in `q`) over the given coefficient ring.

We use the formulas from Proposition 7.8.3 of [EnumComb2] (using Gaussian binomial coefficients $\binom{u}{v}_q$):

$$ps_{n,q}(h_\lambda) = \prod_i \binom{n+\lambda_i-1}{\lambda_i}_q,$$
$$ps_{n,1}(h_\lambda) = \prod_i \binom{n+\lambda_i-1}{\lambda_i},$$
$$ps_q(h_\lambda) = 1/\prod_i \prod_{j=1}^{\lambda_i}(1-q^j).$$

EXAMPLES:

```
sage: h = SymmetricFunctions(QQ).h()
sage: x = h[2,1]
sage: x.principal_specialization(3)
q^6 + 2*q^5 + 4*q^4 + 4*q^3 + 4*q^2 + 2*q + 1
sage: x = 3*h[2] + 2*h[1] + 1
sage: x.principal_specialization(3, q=var("q"))                           #␣
↪needs sage.symbolic
2*(q^3 - 1)/(q - 1) + 3*(q^4 - 1)*(q^3 - 1)/((q^2 - 1)*(q - 1)) + 1
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> x = h[Integer(2),Integer(1)]
>>> x.principal_specialization(Integer(3))
q^6 + 2*q^5 + 4*q^4 + 4*q^3 + 4*q^2 + 2*q + 1
>>> x = Integer(3)*h[Integer(2)] + Integer(2)*h[Integer(1)] + Integer(1)
>>> x.principal_specialization(Integer(3), q=var("q"))                    ␣
↪     # needs sage.symbolic
2*(q^3 - 1)/(q - 1) + 3*(q^4 - 1)*(q^3 - 1)/((q^2 - 1)*(q - 1)) + 1
```

coproduct_on_generators(*i*)

Return the coproduct on h_i.

INPUT:

- `self` – a homogeneous basis of symmetric functions
- `i` – nonnegative integer

OUTPUT: the sum $\sum_{r=0}^{i} h_r \otimes h_{i-r}$

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: h = Sym.homogeneous()
sage: h.coproduct_on_generators(2)
h[] # h[2] + h[1] # h[1] + h[2] # h[]
sage: h.coproduct_on_generators(0)
h[] # h[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> h = Sym.homogeneous()
```

(continues on next page)

(continued from previous page)

```
>>> h.coproduct_on_generators(Integer(2))
h[] # h[2] + h[1] # h[1] + h[2] # h[]
>>> h.coproduct_on_generators(Integer(0))
h[] # h[]
```

5.1.288 Jack Symmetric Functions

Jack's symmetric functions appear in [Ma1995] Chapter VI, section 10. Zonal polynomials are the subject of [Ma1995] Chapter VII. The parameter α in that reference is the parameter t in this implementation in sage.

REFERENCES:

class sage.combinat.sf.jack.**Jack**(*Sym*, *t*)

Bases: UniqueRepresentation

The family of Jack symmetric functions including the P, Q, J, Qp bases. The default parameter is `t`.

INPUT:

- `self` – the family of Jack symmetric function bases
- `Sym` – a ring of symmetric functions
- `t` – an optional parameter (default: `'t'`)

EXAMPLES:

```
sage: SymmetricFunctions(FractionField(QQ['t'])).jack()
Jack polynomials over Fraction Field of Univariate Polynomial Ring in t over␣
↪Rational Field
sage: SymmetricFunctions(QQ).jack(1)
Jack polynomials with t=1 over Rational Field
```

```
>>> from sage.all import *
>>> SymmetricFunctions(FractionField(QQ['t'])).jack()
Jack polynomials over Fraction Field of Univariate Polynomial Ring in t over␣
↪Rational Field
>>> SymmetricFunctions(QQ).jack(Integer(1))
Jack polynomials with t=1 over Rational Field
```

J()

Return the algebra of Jack polynomials in the J basis.

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT: the J basis of the Jack symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JJ = Sym.jack().J(); JJ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack J basis
sage: Sym = SymmetricFunctions(QQ)
sage: Sym.jack(t=-1).J()
Symmetric Functions over Rational Field in the Jack J with t=-1 basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JJ = Sym.jack().J(); JJ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack J basis
>>> Sym = SymmetricFunctions(QQ)
>>> Sym.jack(t=-Integer(1)).J()
Symmetric Functions over Rational Field in the Jack J with t=-1 basis
```

At $t = 1$, the Jack polynomials in the J basis are scalar multiples of the Schur functions with the scalar given by a Partition's `hook_product()` method at 1:

```
sage: Sym = SymmetricFunctions(QQ)
sage: JJ = Sym.jack(t=1).J()
sage: s = Sym.schur()
sage: p = Partition([3,2,1,1])
sage: s(JJ(p)) == p.hook_product(1)*s(p)  # long time (4s on sage.math, 2012)
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> JJ = Sym.jack(t=Integer(1)).J()
>>> s = Sym.schur()
>>> p = Partition([Integer(3),Integer(2),Integer(1),Integer(1)])
>>> s(JJ(p)) == p.hook_product(Integer(1))*s(p)  # long time (4s on sage.math,
↪ 2012)
True
```

At $t = 2$, the Jack polynomials in the J basis are scalar multiples of the zonal polynomials with the scalar given by a Partition's `hook_product()` method at 2.

```
sage: Sym = SymmetricFunctions(QQ)
sage: JJ = Sym.jack(t=2).J()
sage: Z = Sym.zonal()
sage: p = Partition([2,2,1])
sage: Z(JJ(p)) == p.hook_product(2)*Z(p)
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> JJ = Sym.jack(t=Integer(2)).J()
>>> Z = Sym.zonal()
>>> p = Partition([Integer(2),Integer(2),Integer(1)])
>>> Z(JJ(p)) == p.hook_product(Integer(2))*Z(p)
True
```

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JJ = Sym.jack().J()
sage: JP = Sym.jack().P()
sage: JJ(sum(JP(p) for p in Partitions(3)))
1/6*JackJ[1, 1, 1] + (1/(t+2))*JackJ[2, 1] + (1/2/(t^2+3/2*t+1/2))*JackJ[3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JJ = Sym.jack().J()
>>> JP = Sym.jack().P()
>>> JJ(sum(JP(p) for p in Partitions(Integer(3))))
1/6*JackJ[1, 1, 1] + (1/(t+2))*JackJ[2, 1] + (1/2/(t^2+3/2*t+1/2))*JackJ[3]
```

```
sage: s = Sym.schur()
sage: JJ(s([3])) # indirect doctest
((1/6*t^2-1/2*t+1/3)/(t^2+3*t+2))*JackJ[1, 1, 1] + ((t-1)/(t^2+5/
→2*t+1))*JackJ[2, 1] + (1/2/(t^2+3/2*t+1/2))*JackJ[3]
sage: JJ(s([2,1]))
((1/3*t-1/3)/(t+2))*JackJ[1, 1, 1] + (1/(t+2))*JackJ[2, 1]
sage: JJ(s([1,1,1]))
1/6*JackJ[1, 1, 1]
```

```
>>> from sage.all import *
>>> s = Sym.schur()
>>> JJ(s([Integer(3)])) # indirect doctest
((1/6*t^2-1/2*t+1/3)/(t^2+3*t+2))*JackJ[1, 1, 1] + ((t-1)/(t^2+5/
→2*t+1))*JackJ[2, 1] + (1/2/(t^2+3/2*t+1/2))*JackJ[3]
>>> JJ(s([Integer(2),Integer(1)]))
((1/3*t-1/3)/(t+2))*JackJ[1, 1, 1] + (1/(t+2))*JackJ[2, 1]
>>> JJ(s([Integer(1),Integer(1),Integer(1)]))
1/6*JackJ[1, 1, 1]
```

P()

Return the algebra of Jack polynomials in the P basis.

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT: the P basis of the Jack symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P(); JP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
→over Rational Field in the Jack P basis
sage: Sym.jack(t=-1).P()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
→over Rational Field in the Jack P with t=-1 basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JP = Sym.jack().P(); JP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
→over Rational Field in the Jack P basis
>>> Sym.jack(t=-Integer(1)).P()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
→over Rational Field in the Jack P with t=-1 basis
```

At $t = 1$, the Jack polynomials in the P basis are the Schur symmetric functions.

```
sage: Sym = SymmetricFunctions(QQ)
sage: JP = Sym.jack(t=1).P()
sage: s = Sym.schur()
sage: s(JP([2,2,1]))
s[2, 2, 1]
sage: JP(s([2,2,1]))
JackP[2, 2, 1]
sage: JP([2,1])^2
JackP[2, 2, 1, 1] + JackP[2, 2, 2] + JackP[3, 1, 1, 1] + 2*JackP[3, 2, 1] +␣
↪JackP[3, 3] + JackP[4, 1, 1] + JackP[4, 2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> JP = Sym.jack(t=Integer(1)).P()
>>> s = Sym.schur()
>>> s(JP([Integer(2),Integer(2),Integer(1)]))
s[2, 2, 1]
>>> JP(s([Integer(2),Integer(2),Integer(1)]))
JackP[2, 2, 1]
>>> JP([Integer(2),Integer(1)])**Integer(2)
JackP[2, 2, 1, 1] + JackP[2, 2, 2] + JackP[3, 1, 1, 1] + 2*JackP[3, 2, 1] +␣
↪JackP[3, 3] + JackP[4, 1, 1] + JackP[4, 2]
```

At $t = 2$, the Jack polynomials in the P basis are the zonal polynomials.

```
sage: Sym = SymmetricFunctions(QQ)
sage: JP = Sym.jack(t=2).P()
sage: Z = Sym.zonal()
sage: Z(JP([2,2,1]))
Z[2, 2, 1]
sage: JP(Z[2, 2, 1])
JackP[2, 2, 1]
sage: JP([2])^2
64/45*JackP[2, 2] + 16/21*JackP[3, 1] + JackP[4]
sage: Z([2])^2
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> JP = Sym.jack(t=Integer(2)).P()
>>> Z = Sym.zonal()
>>> Z(JP([Integer(2),Integer(2),Integer(1)]))
Z[2, 2, 1]
>>> JP(Z[Integer(2), Integer(2), Integer(1)])
JackP[2, 2, 1]
>>> JP([Integer(2)])**Integer(2)
64/45*JackP[2, 2] + 16/21*JackP[3, 1] + JackP[4]
>>> Z([Integer(2)])**Integer(2)
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
```

```
sage: Sym = SymmetricFunctions(QQ['a','b'].fraction_field())
sage: (a,b) = Sym.base_ring().gens()
```

(continues on next page)

(continued from previous page)

```
sage: Jacka = Sym.jack(t=a)
sage: Jackb = Sym.jack(t=b)
sage: m = Sym.monomial()
sage: JPa = Jacka.P()
sage: JPb = Jackb.P()
sage: m(JPa[2,1])
(6/(a+2))*m[1, 1, 1] + m[2, 1]
sage: m(JPb[2,1])
(6/(b+2))*m[1, 1, 1] + m[2, 1]
sage: m(a*JPb([2,1]) + b*JPa([2,1]))
((6*a^2+6*b^2+12*a+12*b)/(a*b+2*a+2*b+4))*m[1, 1, 1] + (a+b)*m[2, 1]
sage: JPa(JPb([2,1]))
((6*a-6*b)/(a*b+2*a+2*b+4))*JackP[1, 1, 1] + JackP[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['a','b'].fraction_field())
>>> (a,b) = Sym.base_ring().gens()
>>> Jacka = Sym.jack(t=a)
>>> Jackb = Sym.jack(t=b)
>>> m = Sym.monomial()
>>> JPa = Jacka.P()
>>> JPb = Jackb.P()
>>> m(JPa[Integer(2),Integer(1)])
(6/(a+2))*m[1, 1, 1] + m[2, 1]
>>> m(JPb[Integer(2),Integer(1)])
(6/(b+2))*m[1, 1, 1] + m[2, 1]
>>> m(a*JPb([Integer(2),Integer(1)]) + b*JPa([Integer(2),Integer(1)]))
((6*a^2+6*b^2+12*a+12*b)/(a*b+2*a+2*b+4))*m[1, 1, 1] + (a+b)*m[2, 1]
>>> JPa(JPb([Integer(2),Integer(1)]))
((6*a-6*b)/(a*b+2*a+2*b+4))*JackP[1, 1, 1] + JackP[2, 1]
```

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JQ = Sym.jack().Q()
sage: JP = Sym.jack().P()
sage: JJ = Sym.jack().J()
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JQ = Sym.jack().Q()
>>> JP = Sym.jack().P()
>>> JJ = Sym.jack().J()
```

```
sage: JP(JQ([2,1]))
((1/2*t+1)/(t^3+1/2*t^2))*JackP[2, 1]
sage: JP(JQ([3]))
((1/3*t^2+1/2*t+1/6)/t^3)*JackP[3]
sage: JP(JQ([1,1,1]))
(6/(t^3+3*t^2+2*t))*JackP[1, 1, 1]
```

```
>>> from sage.all import *
>>> JP(JQ([Integer(2),Integer(1)]))
```

(continues on next page)

(continued from previous page)

```
((1/2*t+1)/(t^3+1/2*t^2))*JackP[2, 1]
>>> JP(JQ([Integer(3)]))
((1/3*t^2+1/2*t+1/6)/t^3)*JackP[3]
>>> JP(JQ([Integer(1),Integer(1),Integer(1)]))
(6/(t^3+3*t^2+2*t))*JackP[1, 1, 1]
```

```
sage: JP(JJ([3]))
(2*t^2+3*t+1)*JackP[3]
sage: JP(JJ([2,1]))
(t+2)*JackP[2, 1]
sage: JP(JJ([1,1,1]))
6*JackP[1, 1, 1]
```

```
>>> from sage.all import *
>>> JP(JJ([Integer(3)]))
(2*t^2+3*t+1)*JackP[3]
>>> JP(JJ([Integer(2),Integer(1)]))
(t+2)*JackP[2, 1]
>>> JP(JJ([Integer(1),Integer(1),Integer(1)]))
6*JackP[1, 1, 1]
```

```
sage: s = Sym.schur()
sage: JP(s([2,1]))
((2*t-2)/(t+2))*JackP[1, 1, 1] + JackP[2, 1]
sage: s(_)
s[2, 1]
```

```
>>> from sage.all import *
>>> s = Sym.schur()
>>> JP(s([Integer(2),Integer(1)]))
((2*t-2)/(t+2))*JackP[1, 1, 1] + JackP[2, 1]
>>> s(_)
s[2, 1]
```

Q()

Return the algebra of Jack polynomials in the Q basis.

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT: the Q basis of the Jack symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JQ = Sym.jack().Q(); JQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack Q basis
sage: Sym = SymmetricFunctions(QQ)
sage: Sym.jack(t=-1).Q()
Symmetric Functions over Rational Field in the Jack Q with t=-1 basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JQ = Sym.jack().Q(); JQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack Q basis
>>> Sym = SymmetricFunctions(QQ)
>>> Sym.jack(t=-Integer(1)).Q()
Symmetric Functions over Rational Field in the Jack Q with t=-1 basis
```

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JQ = Sym.jack().Q()
sage: JP = Sym.jack().P()
sage: JQ(sum(JP(p) for p in Partitions(3)))
(1/6*t^3+1/2*t^2+1/3*t)*JackQ[1, 1, 1] + ((2*t^3+t^2)/(t+2))*JackQ[2, 1] +␣
↪(3*t^3/(t^2+3/2*t+1/2))*JackQ[3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JQ = Sym.jack().Q()
>>> JP = Sym.jack().P()
>>> JQ(sum(JP(p) for p in Partitions(Integer(3))))
(1/6*t^3+1/2*t^2+1/3*t)*JackQ[1, 1, 1] + ((2*t^3+t^2)/(t+2))*JackQ[2, 1] +␣
↪(3*t^3/(t^2+3/2*t+1/2))*JackQ[3]
```

```
sage: s = Sym.schur()
sage: JQ(s([3])) # indirect doctest
(1/6*t^3-1/2*t^2+1/3*t)*JackQ[1, 1, 1] + ((2*t^3-2*t^2)/(t+2))*JackQ[2, 1] +␣
↪(3*t^3/(t^2+3/2*t+1/2))*JackQ[3]
sage: JQ(s([2,1]))
(1/3*t^3-1/3*t)*JackQ[1, 1, 1] + ((2*t^3+t^2)/(t+2))*JackQ[2, 1]
sage: JQ(s([1,1,1]))
(1/6*t^3+1/2*t^2+1/3*t)*JackQ[1, 1, 1]
```

```
>>> from sage.all import *
>>> s = Sym.schur()
>>> JQ(s([Integer(3)])) # indirect doctest
(1/6*t^3-1/2*t^2+1/3*t)*JackQ[1, 1, 1] + ((2*t^3-2*t^2)/(t+2))*JackQ[2, 1] +␣
↪(3*t^3/(t^2+3/2*t+1/2))*JackQ[3]
>>> JQ(s([Integer(2),Integer(1)]))
(1/3*t^3-1/3*t)*JackQ[1, 1, 1] + ((2*t^3+t^2)/(t+2))*JackQ[2, 1]
>>> JQ(s([Integer(1),Integer(1),Integer(1)]))
(1/6*t^3+1/2*t^2+1/3*t)*JackQ[1, 1, 1]
```

Qp()

Return the algebra of Jack polynomials in the Qp, which is dual to the P basis with respect to the standard scalar product.

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT: the Q' basis of the Jack symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P()
sage: JQp = Sym.jack().Qp(); JQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack Qp basis
sage: a = JQp([2])
sage: a.scalar(JP([2]))
1
sage: a.scalar(JP([1,1]))
0
sage: JP(JQp([2]))                        # todo: missing auto normalization
((t-1)/(t+1))*JackP[1, 1] + JackP[2]
sage: JP._normalize(JP(JQp([2])))
((t-1)/(t+1))*JackP[1, 1] + JackP[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JP = Sym.jack().P()
>>> JQp = Sym.jack().Qp(); JQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack Qp basis
>>> a = JQp([Integer(2)])
>>> a.scalar(JP([Integer(2)]))
1
>>> a.scalar(JP([Integer(1),Integer(1)]))
0
>>> JP(JQp([Integer(2)]))                            # todo: missing auto␣
↪normalization
((t-1)/(t+1))*JackP[1, 1] + JackP[2]
>>> JP._normalize(JP(JQp([Integer(2)])))
((t-1)/(t+1))*JackP[1, 1] + JackP[2]
```

base_ring()

Return the base ring of the symmetric functions in which the Jack symmetric functions live.

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT: the base ring of the symmetric functions ring of `self`

EXAMPLES:

```
sage: J2 = SymmetricFunctions(QQ).jack(t=2)
sage: J2.base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> J2 = SymmetricFunctions(QQ).jack(t=Integer(2))
>>> J2.base_ring()
Rational Field
```

symmetric_function_ring()

Return the base ring of the symmetric functions of the Jack symmetric function bases

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT: the symmetric functions ring of `self`

EXAMPLES:

```
sage: Jacks = SymmetricFunctions(FractionField(QQ['t'])).jack()
sage: Jacks.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field
```

```
>>> from sage.all import *
>>> Jacks = SymmetricFunctions(FractionField(QQ['t'])).jack()
>>> Jacks.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field
```

class sage.combinat.sf.jack.**JackPolynomials_generic**(*jack*)

Bases: *SymmetricFunctionAlgebra_generic*

A class of methods which are common to all Jack bases of the symmetric functions.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `jack` – a family of Jack symmetric function bases

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P(); JP.base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
sage: Sym = SymmetricFunctions(QQ)
sage: JP = Sym.jack(t=2).P(); JP.base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JP = Sym.jack().P(); JP.base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
>>> Sym = SymmetricFunctions(QQ)
>>> JP = Sym.jack(t=Integer(2)).P(); JP.base_ring()
Rational Field
```

class Element

Bases: *SymmetricFunctionAlgebra_generic_Element*

scalar_jack(*x*, *t=None*)

A scalar product where the power sums are orthogonal and $\langle p_\mu, p_\mu \rangle = z_\mu t^{length(\mu)}$

INPUT:

- `self` – an element of a Jack basis of the symmetric functions
- `x` – an element of the symmetric functions
- `t` – an optional parameter (default: `None`; uses the parameter from the basis)

OUTPUT: the Jack scalar product between `x` and `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P()
sage: JQ = Sym.jack().Q()
sage: p = Partitions(3).list()
sage: matrix([[JP(a).scalar_jack(JQ(b)) for a in p] for b in p])
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JP = Sym.jack().P()
>>> JQ = Sym.jack().Q()
>>> p = Partitions(Integer(3)).list()
>>> matrix([[JP(a).scalar_jack(JQ(b)) for a in p] for b in p])
[1 0 0]
[0 1 0]
[0 0 1]
```

c1(*part*)

Return the t-Jack scalar product between `J(part)` and `P(part)`.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `part` – a partition
- `t` – an optional parameter (default: uses the parameter t from the Jack basis)

OUTPUT:

- a polynomial in the parameter `t` which is equal to the scalar product of `J(part)` and `P(part)`

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: JP.c1(Partition([2,1]))
t + 2
```

```
>>> from sage.all import *
>>> JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
>>> JP.c1(Partition([Integer(2),Integer(1)]))
t + 2
```

c2(*part*)

Return the t-Jack scalar product between `J(part)` and `Q(part)`.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `part` – a partition

- **t – an optional parameter (default: uses the parameter t from the** Jack basis)

OUTPUT:

- a polynomial in the parameter `t` which is equal to the scalar product of `J(part)` and `Q(part)`

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: JP.c2(Partition([2,1]))
2*t^3 + t^2
```

```
>>> from sage.all import *
>>> JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
>>> JP.c2(Partition([Integer(2),Integer(1)]))
2*t^3 + t^2
```

construction()

Return a pair `(F, R)`, where `F` is a `SymmetricFunctionsFunctor` and R is a ring, such that `F(R)` returns `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P()
sage: JP.construction()
(SymmetricFunctionsFunctor[Jack P],
 Fraction Field of Univariate Polynomial Ring in t over Rational Field)
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JP = Sym.jack().P()
>>> JP.construction()
(SymmetricFunctionsFunctor[Jack P],
 Fraction Field of Univariate Polynomial Ring in t over Rational Field)
```

coproduct_by_coercion(*elt*)

Return the coproduct of the element `elt` by coercion to the Schur basis.

INPUT:

- `self` – a Jack symmetric function basis
- `elt` – an instance of this basis

OUTPUT:

- The coproduct acting on `elt`, the result is an element of the tensor squared of the Jack symmetric function basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: Sym.jack().P()[2,2].coproduct() #indirect doctest
JackP[] # JackP[2, 2] + (2/(t+1))*JackP[1] # JackP[2, 1] + ((8*t+4)/(t^3+4*t^
↪2+5*t+2))*JackP[1, 1] # JackP[1, 1] + JackP[2] # JackP[2] + (2/
↪(t+1))*JackP[2, 1] # JackP[1] + JackP[2, 2] # JackP[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'].fraction_field())
>>> Sym.jack().P()[Integer(2),Integer(2)].coproduct() #indirect doctest
JackP[] # JackP[2, 2] + (2/(t+1))*JackP[1] # JackP[2, 1] + ((8*t+4)/(t^3+4*t^
→2+5*t+2))*JackP[1, 1] # JackP[1, 1] + JackP[2] # JackP[2] + (2/
→(t+1))*JackP[2, 1] # JackP[1] + JackP[2, 2] # JackP[]
```

jack_family()

Return the family of Jack bases associated to the basis `self`.

INPUT:

- `self` – a Jack basis of the symmetric functions

EXAMPLES:

```
sage: JackP = SymmetricFunctions(QQ).jack(t=2).P()
sage: JackP.jack_family()
Jack polynomials with t=2 over Rational Field
```

```
>>> from sage.all import *
>>> JackP = SymmetricFunctions(QQ).jack(t=Integer(2)).P()
>>> JackP.jack_family()
Jack polynomials with t=2 over Rational Field
```

product(*left*, *right*)

The product of two Jack symmetric functions is done by multiplying the elements in the P basis and then expressing the elements in the basis `self`.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `left`, `right` – symmetric function elements

OUTPUT: the product of `left` and `right` expanded in the basis `self`

EXAMPLES:

```
sage: JJ = SymmetricFunctions(FractionField(QQ['t'])).jack().J()
sage: JJ([1])^2              # indirect doctest
(t/(t+1))*JackJ[1, 1] + (1/(t+1))*JackJ[2]
sage: JJ([2])^2
(t^2/(t^2+3/2*t+1/2))*JackJ[2, 2] + (4/3*t/(t^2+4/3*t+1/3))*JackJ[3, 1] + ((1/
→6*t+1/6)/(t^2+5/6*t+1/6))*JackJ[4]
sage: JQ = SymmetricFunctions(FractionField(QQ['t'])).jack().Q()
sage: JQ([1])^2              # indirect doctest
JackQ[1, 1] + (2/(t+1))*JackQ[2]
sage: JQ([2])^2
JackQ[2, 2] + (2/(t+1))*JackQ[3, 1] + ((t+1)/(t^2+5/6*t+1/6))*JackQ[4]
```

```
>>> from sage.all import *
>>> JJ = SymmetricFunctions(FractionField(QQ['t'])).jack().J()
>>> JJ([Integer(1)])**Integer(2)              # indirect doctest
(t/(t+1))*JackJ[1, 1] + (1/(t+1))*JackJ[2]
>>> JJ([Integer(2)])**Integer(2)
```

(continues on next page)

(continued from previous page)

```
(t^2/(t^2+3/2*t+1/2))*JackJ[2, 2] + (4/3*t/(t^2+4/3*t+1/3))*JackJ[3, 1] + ((1/
↪6*t+1/6)/(t^2+5/6*t+1/6))*JackJ[4]
>>> JQ = SymmetricFunctions(FractionField(QQ['t'])).jack().Q()
>>> JQ([Integer(1)])**Integer(2)              # indirect doctest
JackQ[1, 1] + (2/(t+1))*JackQ[2]
>>> JQ([Integer(2)])**Integer(2)
JackQ[2, 2] + (2/(t+1))*JackQ[3, 1] + ((t+1)/(t^2+5/6*t+1/6))*JackQ[4]
```

class sage.combinat.sf.jack.**JackPolynomials_j**(*jack*)

Bases: *JackPolynomials_generic*

The J basis is a defined as a normalized form of the P basis.

INPUT:

- `self` – an instance of the Jack P basis of the symmetric functions
- `jack` – a family of Jack symmetric function bases

EXAMPLES:

```
sage: J = SymmetricFunctions(FractionField(QQ['t'])).jack().J()
sage: TestSuite(J).run(skip=['_test_associativity', '_test_distributivity', '_
↪test_prod']) # products are too expensive
sage: TestSuite(J).run(elements = [J.t*J[1,1]+J[2], J[1]+(1+J.t)*J[1,1]])  # long␣
↪time (3s on sage.math, 2012)
```

```
>>> from sage.all import *
>>> J = SymmetricFunctions(FractionField(QQ['t'])).jack().J()
>>> TestSuite(J).run(skip=['_test_associativity', '_test_distributivity', '_test_
↪prod']) # products are too expensive
>>> TestSuite(J).run(elements = [J.t*J[Integer(1),Integer(1)]+J[Integer(2)],␣
↪J[Integer(1)]+(Integer(1)+J.t)*J[Integer(1),Integer(1)]])  # long time (3s on␣
↪sage.math, 2012)
```

class Element

Bases: *Element*

class sage.combinat.sf.jack.**JackPolynomials_p**(*jack*)

Bases: *JackPolynomials_generic*

The P basis is uni-triangularly related to the monomial basis and orthogonal with respect to the Jack scalar product.

INPUT:

- `self` – an instance of the Jack P basis of the symmetric functions
- `jack` – a family of Jack symmetric function bases

EXAMPLES:

```
sage: P = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: TestSuite(P).run(skip=['_test_associativity', '_test_distributivity', '_
↪test_prod']) # products are too expensive
sage: TestSuite(P).run(elements = [P.t*P[1,1]+P[2], P[1]+(1+P.t)*P[1,1]])
```

```
>>> from sage.all import *
>>> P = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
>>> TestSuite(P).run(skip=['_test_associativity', '_test_distributivity', '_test_
↪prod']) # products are too expensive
>>> TestSuite(P).run(elements = [P.t*P[Integer(1),Integer(1)]+P[Integer(2)],␣
↪P[Integer(1)]+(Integer(1)+P.t)*P[Integer(1),Integer(1)]])
```

class Element

Bases: *Element*

scalar_jack(*x*, *t=None*)

The scalar product on the symmetric functions where the power sums are orthogonal and $\langle p_\mu, p_\mu \rangle = z_\mu t^{length(mu)}$ where the t parameter from the Jack symmetric function family.

INPUT:

- `self` – an element of the Jack P basis
- `x` – an element of the P basis

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: l = [JP(p) for p in Partitions(3)]
sage: matrix([[a.scalar_jack(b) for a in l] for b in l])
[3*t^3/(t^2 + 3/2*t + 1/2)                         0                         ␣
↪   0]
[                        0      (2*t^3 + t^2)/(t + 2)                         ␣
↪   0]
[                        0                         0 1/6*t^3 + 1/2*t^2 +␣
↪1/3*t]
```

```
>>> from sage.all import *
>>> JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
>>> l = [JP(p) for p in Partitions(Integer(3))]
>>> matrix([[a.scalar_jack(b) for a in l] for b in l])
[3*t^3/(t^2 + 3/2*t + 1/2)                         0                         ␣
↪   0]
[                        0      (2*t^3 + t^2)/(t + 2)                         ␣
↪   0]
[                        0                         0 1/6*t^3 + 1/2*t^2 +␣
↪1/3*t]
```

product(*left*, *right*)

The product of two Jack symmetric functions is done by multiplying the elements in the monomial basis and then expressing the elements the basis `self`.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `left`, `right` – symmetric function elements

OUTPUT: the product of `left` and `right` expanded in the basis `self`

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: m = JP.symmetric_function_ring().m()
```

(continues on next page)

(continued from previous page)

```
sage: JP([1])^2 # indirect doctest
(2*t/(t+1))*JackP[1, 1] + JackP[2]
sage: m(_)
2*m[1, 1] + m[2]
sage: JP = SymmetricFunctions(QQ).jack(t=2).P()
sage: JP([2,1])^2
125/63*JackP[2, 2, 1, 1] + 25/12*JackP[2, 2, 2] + 25/18*JackP[3, 1, 1, 1] +
↪12/5*JackP[3, 2, 1] + 4/3*JackP[3, 3] + 4/3*JackP[4, 1, 1] + JackP[4, 2]
sage: m(_)
45*m[1, 1, 1, 1, 1, 1] + 51/2*m[2, 1, 1, 1, 1] + 29/2*m[2, 2, 1, 1] + 33/
↪4*m[2, 2, 2] + 9*m[3, 1, 1, 1] + 5*m[3, 2, 1] + 2*m[3, 3] + 2*m[4, 1, 1] +
↪m[4, 2]
```

```
>>> from sage.all import *
>>> JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
>>> m = JP.symmetric_function_ring().m()
>>> JP([Integer(1)])**Integer(2) # indirect doctest
(2*t/(t+1))*JackP[1, 1] + JackP[2]
>>> m(_)
2*m[1, 1] + m[2]
>>> JP = SymmetricFunctions(QQ).jack(t=Integer(2)).P()
>>> JP([Integer(2),Integer(1)])**Integer(2)
125/63*JackP[2, 2, 1, 1] + 25/12*JackP[2, 2, 2] + 25/18*JackP[3, 1, 1, 1] +
↪12/5*JackP[3, 2, 1] + 4/3*JackP[3, 3] + 4/3*JackP[4, 1, 1] + JackP[4, 2]
>>> m(_)
45*m[1, 1, 1, 1, 1, 1] + 51/2*m[2, 1, 1, 1, 1] + 29/2*m[2, 2, 1, 1] + 33/
↪4*m[2, 2, 2] + 9*m[3, 1, 1, 1] + 5*m[3, 2, 1] + 2*m[3, 3] + 2*m[4, 1, 1] +
↪m[4, 2]
```

scalar_jack_basis(*part1*, *part2=None*)

Return the scalar product of $P(part1)$ and $P(part2)$.

This is equation (10.16) of [Mc1995] on page 380.

INPUT:

- `self` – an instance of the Jack P basis of the symmetric functions
- `part1` – a partition
- `part2` – an optional partition (default: `None`)

OUTPUT:

- the scalar product between $P(part1)$ and $P(part2)$ (or itself if $part2$ is None)

REFERENCES:

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: JJ = SymmetricFunctions(FractionField(QQ['t'])).jack().J()
sage: JP.scalar_jack_basis(Partition([2,1]), Partition([1,1,1]))
0
sage: JP._normalize_coefficients(JP.scalar_jack_basis(Partition([3,2,1]),
↪Partition([3,2,1])))
```

(continues on next page)

(continued from previous page)

```
(6*t^6 + 10*t^5 + 11/2*t^4 + t^3)/(t^3 + 11/2*t^2 + 10*t + 6)
sage: JJ(JP[3,2,1]).scalar_jack(JP[3,2,1])
(6*t^6 + 10*t^5 + 11/2*t^4 + t^3)/(t^3 + 11/2*t^2 + 10*t + 6)
```

```
>>> from sage.all import *
>>> JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
>>> JJ = SymmetricFunctions(FractionField(QQ['t'])).jack().J()
>>> JP.scalar_jack_basis(Partition([Integer(2),Integer(1)]),␣
↪Partition([Integer(1),Integer(1),Integer(1)]))
0
>>> JP._normalize_coefficients(JP.scalar_jack_basis(Partition([Integer(3),
↪Integer(2),Integer(1)]), Partition([Integer(3),Integer(2),Integer(1)])))
(6*t^6 + 10*t^5 + 11/2*t^4 + t^3)/(t^3 + 11/2*t^2 + 10*t + 6)
>>> JJ(JP[Integer(3),Integer(2),Integer(1)]).scalar_jack(JP[Integer(3),
↪Integer(2),Integer(1)])
(6*t^6 + 10*t^5 + 11/2*t^4 + t^3)/(t^3 + 11/2*t^2 + 10*t + 6)
```

With a single argument, takes $part2 = part1$:

```
sage: JP.scalar_jack_basis(Partition([2,1]), Partition([2,1]))
(2*t^3 + t^2)/(t + 2)
sage: JJ(JP[2,1]).scalar_jack(JP[2,1])
(2*t^3 + t^2)/(t + 2)
```

```
>>> from sage.all import *
>>> JP.scalar_jack_basis(Partition([Integer(2),Integer(1)]),␣
↪Partition([Integer(2),Integer(1)]))
(2*t^3 + t^2)/(t + 2)
>>> JJ(JP[Integer(2),Integer(1)]).scalar_jack(JP[Integer(2),Integer(1)])
(2*t^3 + t^2)/(t + 2)
```

class sage.combinat.sf.jack.**JackPolynomials_q**(*jack*)

Bases: *JackPolynomials_generic*

The Q basis is defined as a normalized form of the P basis.

INPUT:

- self – an instance of the Jack Q basis of the symmetric functions
- jack – a family of Jack symmetric function bases

EXAMPLES:

```
sage: Q = SymmetricFunctions(FractionField(QQ['t'])).jack().Q()
sage: TestSuite(Q).run(skip=['_test_associativity', '_test_distributivity', '_
↪test_prod']) # products are too expensive
sage: TestSuite(Q).run(elements = [Q.t*Q[1,1]+Q[2], Q[1]+(1+Q.t)*Q[1,1]])  # long␣
↪time (3s on sage.math, 2012)
```

```
>>> from sage.all import *
>>> Q = SymmetricFunctions(FractionField(QQ['t'])).jack().Q()
>>> TestSuite(Q).run(skip=['_test_associativity', '_test_distributivity', '_test_
↪prod']) # products are too expensive
```

(continues on next page)

(continued from previous page)

```
>>> TestSuite(Q).run(elements = [Q.t*Q[Integer(1),Integer(1)]+Q[Integer(2)],
↪Q[Integer(1)]+(Integer(1)+Q.t)*Q[Integer(1),Integer(1)]])  # long time (3s on
↪sage.math, 2012)
```

class Element

Bases: *Element*

class sage.combinat.sf.jack.**JackPolynomials_qp**(*jack*)

Bases: *JackPolynomials_generic*

The Qp basis is the dual basis to the P basis with respect to the standard scalar product

INPUT:

- `self` – an instance of the Jack Qp basis of the symmetric functions
- `jack` – a family of Jack symmetric function bases

EXAMPLES:

```
sage: Qp = SymmetricFunctions(FractionField(QQ['t'])).jack().Qp()
sage: TestSuite(Qp).run(skip=['_test_associativity', '_test_distributivity', '_
↪test_prod']) # products are too expensive
sage: TestSuite(Qp).run(elements = [Qp.t*Qp[1,1]+Qp[2], Qp[1]+(1+Qp.t)*Qp[1,1]])
↪# long time (3s on sage.math, 2012)
```

```
>>> from sage.all import *
>>> Qp = SymmetricFunctions(FractionField(QQ['t'])).jack().Qp()
>>> TestSuite(Qp).run(skip=['_test_associativity', '_test_distributivity', '_test_
↪prod']) # products are too expensive
>>> TestSuite(Qp).run(elements = [Qp.t*Qp[Integer(1),Integer(1)]+Qp[Integer(2)],
↪Qp[Integer(1)]+(Integer(1)+Qp.t)*Qp[Integer(1),Integer(1)]])  # long time (3s
↪on sage.math, 2012)
```

class Element

Bases: *Element*

coproduct_by_coercion(*elt*)

Return the coproduct of the element `elt` by coercion to the Schur basis.

INPUT:

- `elt` – an instance of the `Qp` basis

OUTPUT:

- The coproduct acting on `elt`, the result is an element of the tensor squared of the `Qp` symmetric function basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: JQp = Sym.jack().Qp()
sage: JQp[2,2].coproduct()   #indirect doctest
JackQp[] # JackQp[2, 2] + (2*t/(t+1))*JackQp[1] # JackQp[2, 1] + JackQp[1, 1]
↪# JackQp[1, 1] + ((2*t^3+4*t^2)/(t^3+5/2*t^2+2*t+1/2))*JackQp[2] #
↪JackQp[2] + (2*t/(t+1))*JackQp[2, 1] # JackQp[1] + JackQp[2, 2] # JackQp[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'].fraction_field())
>>> JQp = Sym.jack().Qp()
>>> JQp[Integer(2),Integer(2)].coproduct()   #indirect doctest
JackQp[] # JackQp[2, 2] + (2*t/(t+1))*JackQp[1] # JackQp[2, 1] + JackQp[1, 1]
↪# JackQp[1, 1] + ((2*t^3+4*t^2)/(t^3+5/2*t^2+2*t+1/2))*JackQp[2] #␣
↪JackQp[2] + (2*t/(t+1))*JackQp[2, 1] # JackQp[1] + JackQp[2, 2] # JackQp[]
```

product (*left*, *right*)

The product of two Jack symmetric functions is done by multiplying the elements in the monomial basis and then expressing the elements the basis `self`.

INPUT:

- `self` – an instance of the Jack Qp basis of the symmetric functions
- `left`, `right` – symmetric function elements

OUTPUT: the product of `left` and `right` expanded in the basis `self`

EXAMPLES:

```
sage: JQp = SymmetricFunctions(FractionField(QQ['t'])).jack().Qp()
sage: h = JQp.symmetric_function_ring().h()
sage: JQp([1])^2 # indirect doctest
JackQp[1, 1] + (2/(t+1))*JackQp[2]
sage: h(_)
h[1, 1]
sage: JQp = SymmetricFunctions(QQ).jack(t=2).Qp()
sage: h = SymmetricFunctions(QQ).h()
sage: JQp([2,1])^2
JackQp[2, 2, 1, 1] + 2/3*JackQp[2, 2, 2] + 2/3*JackQp[3, 1, 1, 1] + 48/
↪35*JackQp[3, 2, 1] + 28/75*JackQp[3, 3] + 128/225*JackQp[4, 1, 1] + 28/
↪75*JackQp[4, 2]
sage: h(_)
h[2, 2, 1, 1] - 6/5*h[3, 2, 1] + 9/25*h[3, 3]
```

```
>>> from sage.all import *
>>> JQp = SymmetricFunctions(FractionField(QQ['t'])).jack().Qp()
>>> h = JQp.symmetric_function_ring().h()
>>> JQp([Integer(1)])**Integer(2) # indirect doctest
JackQp[1, 1] + (2/(t+1))*JackQp[2]
>>> h(_)
h[1, 1]
>>> JQp = SymmetricFunctions(QQ).jack(t=Integer(2)).Qp()
>>> h = SymmetricFunctions(QQ).h()
>>> JQp([Integer(2),Integer(1)])**Integer(2)
JackQp[2, 2, 1, 1] + 2/3*JackQp[2, 2, 2] + 2/3*JackQp[3, 1, 1, 1] + 48/
↪35*JackQp[3, 2, 1] + 28/75*JackQp[3, 3] + 128/225*JackQp[4, 1, 1] + 28/
↪75*JackQp[4, 2]
>>> h(_)
h[2, 2, 1, 1] - 6/5*h[3, 2, 1] + 9/25*h[3, 3]
```

class sage.combinat.sf.jack.**SymmetricFunctionAlgebra_zonal** (*Sym*)

Bases: *SymmetricFunctionAlgebra_generic*

Return the algebra of zonal polynomials.

INPUT:

- `self` – a zonal basis of the symmetric functions
- `Sym` – a ring of the symmetric functions

EXAMPLES:

```
sage: Z = SymmetricFunctions(QQ).zonal()
sage: Z([2])^2
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
sage: Z = SymmetricFunctions(QQ).zonal()
sage: TestSuite(Z).run(skip=['_test_associativity', '_test_distributivity', '_
↪test_prod']) # products are too expensive
sage: TestSuite(Z).run(elements = [Z[1,1]+Z[2], Z[1]+2*Z[1,1]])
```

```
>>> from sage.all import *
>>> Z = SymmetricFunctions(QQ).zonal()
>>> Z([Integer(2)])**Integer(2)
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
>>> Z = SymmetricFunctions(QQ).zonal()
>>> TestSuite(Z).run(skip=['_test_associativity', '_test_distributivity', '_test_
↪prod']) # products are too expensive
>>> TestSuite(Z).run(elements = [Z[Integer(1),Integer(1)]+Z[Integer(2)],
↪Z[Integer(1)]+Integer(2)*Z[Integer(1),Integer(1)]])
```

class Element

Bases: *SymmetricFunctionAlgebra_generic_Element*

scalar_zonal(*x*)

The zonal scalar product has the power sum basis and the zonal symmetric functions are orthogonal. In particular, $\langle p_\mu, p_\mu \rangle = z_\mu 2^{length(\mu)}$.

INPUT:

- `self` – an element of the zonal basis
- `x` – an element of the symmetric function

OUTPUT: the scalar product between `self` and `x`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Z = Sym.zonal()
sage: parts = Partitions(3).list()
sage: matrix([[Z(a).scalar_zonal(Z(b)) for a in parts] for b in parts])
[16/5    0    0]
[   0    5    0]
[   0    0    4]
sage: p = Z.symmetric_function_ring().power()
sage: matrix([[Z(p(a)).scalar_zonal(p(b)) for a in parts] for b in parts])
[ 6  0  0]
[ 0  8  0]
[ 0  0 48]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Z = Sym.zonal()
>>> parts = Partitions(Integer(3)).list()
>>> matrix([[Z(a).scalar_zonal(Z(b)) for a in parts] for b in parts])
[16/5    0    0]
[   0    5    0]
[   0    0    4]
>>> p = Z.symmetric_function_ring().power()
>>> matrix([[Z(p(a)).scalar_zonal(p(b)) for a in parts] for b in parts])
[ 6  0  0]
[ 0  8  0]
[ 0  0 48]
```

product(*left*, *right*)

The product of two zonal symmetric functions is done by multiplying the elements in the monomial basis and then expressing the elements in the basis `self`.

INPUT:

- `self` – a zonal basis of the symmetric functions
- `left`, `right` – symmetric function elements

OUTPUT: the product of `left` and `right` expanded in the basis `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Z = Sym.zonal()
sage: JP = Sym.jack(t=1).P()
sage: Z([2])*Z([3])                          # indirect doctest
192/175*Z[3, 2] + 32/45*Z[4, 1] + Z[5]
sage: Z([2])*JP([2])
10/27*Z[2, 1, 1] + 64/45*Z[2, 2] + 23/21*Z[3, 1] + Z[4]
sage: JP = Sym.jack(t=2).P()
sage: Z([2])*JP([2])
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Z = Sym.zonal()
>>> JP = Sym.jack(t=Integer(1)).P()
>>> Z([Integer(2)])*Z([Integer(3)])                          # indirect doctest
192/175*Z[3, 2] + 32/45*Z[4, 1] + Z[5]
>>> Z([Integer(2)])*JP([Integer(2)])
10/27*Z[2, 1, 1] + 64/45*Z[2, 2] + 23/21*Z[3, 1] + Z[4]
>>> JP = Sym.jack(t=Integer(2)).P()
>>> Z([Integer(2)])*JP([Integer(2)])
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
```

`sage.combinat.sf.jack.`**c1**(*part*, *t*)

Return the t-Jack scalar product between `J(part)` and `P(part)`.

INPUT:

- `part` – a partition
- `t` – an optional parameter (default: uses the parameter t from the Jack basis)

OUTPUT:

- a polynomial in the parameter `t` which is equal to the scalar product of `J(part)` and `P(part)`

EXAMPLES:

```
sage: from sage.combinat.sf.jack import c1
sage: t = QQ['t'].gen()
sage: [c1(p,t) for p in Partitions(3)]
[2*t^2 + 3*t + 1, t + 2, 6]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.jack import c1
>>> t = QQ['t'].gen()
>>> [c1(p,t) for p in Partitions(Integer(3))]
[2*t^2 + 3*t + 1, t + 2, 6]
```

`sage.combinat.sf.jack.`**`c2`**`(`*part*, *t*`)`

Return the t-Jack scalar product between `J(part)` and `Q(part)`.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `part` – a partition
- `t` – an optional parameter (default: uses the parameter t from the Jack basis)

OUTPUT:

- a polynomial in the parameter `t` which is equal to the scalar product of `J(part)` and `Q(part)`

EXAMPLES:

```
sage: from sage.combinat.sf.jack import c2
sage: t = QQ['t'].gen()
sage: [c2(p,t) for p in Partitions(3)]
[6*t^3, 2*t^3 + t^2, t^3 + 3*t^2 + 2*t]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.jack import c2
>>> t = QQ['t'].gen()
>>> [c2(p,t) for p in Partitions(Integer(3))]
[6*t^3, 2*t^3 + t^2, t^3 + 3*t^2 + 2*t]
```

`sage.combinat.sf.jack.`**`normalize_coefficients`**`(`*self*, *c*`)`

If our coefficient ring is the field of fractions over a univariate polynomial ring over the rationals, then we should clear both the numerator and denominator of the denominators of their coefficients.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `c` – a coefficient in the base ring of `self`

OUTPUT: divide numerator and denominator by the greatest common divisor

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: t = JP.base_ring().gen()
sage: a = 2/(1/2*t+1/2)
sage: JP._normalize_coefficients(a)
4/(t + 1)
sage: a = 1/(1/3+1/6*t)
sage: JP._normalize_coefficients(a)
6/(t + 2)
sage: a = 24/(4*t^2 + 12*t + 8)
sage: JP._normalize_coefficients(a)
6/(t^2 + 3*t + 2)
```

```
>>> from sage.all import *
>>> JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
>>> t = JP.base_ring().gen()
>>> a = Integer(2)/(Integer(1)/Integer(2)*t+Integer(1)/Integer(2))
>>> JP._normalize_coefficients(a)
4/(t + 1)
>>> a = Integer(1)/(Integer(1)/Integer(3)+Integer(1)/Integer(6)*t)
>>> JP._normalize_coefficients(a)
6/(t + 2)
>>> a = Integer(24)/(Integer(4)*t**Integer(2) + Integer(12)*t + Integer(8))
>>> JP._normalize_coefficients(a)
6/(t^2 + 3*t + 2)
```

sage.combinat.sf.jack.**part_scalar_jack**(*part1*, *part2*, *t*)

Return the Jack scalar product between `p(part1)` and `p(part2)` where p is the power-sum basis.

INPUT:

- `part1`, `part2` – two partitions
- `t` – a parameter

OUTPUT: the scalar product between the power sum indexed by `part1` and `part2`

EXAMPLES:

```
sage: Q.<t> = QQ[]
sage: from sage.combinat.sf.jack import part_scalar_jack
sage: matrix([[part_scalar_jack(p1,p2,t) for p1 in Partitions(4)] for p2 in␣
↪Partitions(4)])
[   4*t       0       0       0       0]
[     0   3*t^2       0       0       0]
[     0       0   8*t^2       0       0]
[     0       0       0   4*t^3       0]
[     0       0       0       0 24*t^4]
```

```
>>> from sage.all import *
>>> Q = QQ['t']; (t,) = Q._first_ngens(1)
>>> from sage.combinat.sf.jack import part_scalar_jack
>>> matrix([[part_scalar_jack(p1,p2,t) for p1 in Partitions(Integer(4))] for p2␣
↪in Partitions(Integer(4))])
[   4*t       0       0       0       0]
[     0   3*t^2       0       0       0]
```

(continues on next page)

(continued from previous page)

```
[      0       0   8*t^2       0       0]
[      0       0       0   4*t^3       0]
[      0       0       0       0  24*t^4]
```

5.1.289 Quotient of symmetric function space by ideal generated by Hall-Littlewood symmetric functions

The quotient of symmetric functions by the ideal generated by the Hall-Littlewood P symmetric functions indexed by partitions with first part greater than k. When $t = 1$ this space is the quotient of the symmetric functions by the ideal generated by the monomial symmetric functions indexed by partitions with first part greater than k.

AUTHORS:

- Chris Berg (2012-12-01)
- Mike Zabrocki - k-bounded Hall Littlewood P and dual k-Schur functions (2012-12-02)

class sage.combinat.sf.k_dual.**AffineSchurFunctions**(*kBoundedRing*)

Bases: *KBoundedQuotientBasis*

This basis is dual to the k-Schur functions at $t = 1$. This realization follows the monomial expansion given by Lam [Lam2006].

REFERENCES:

class sage.combinat.sf.k_dual.**DualkSchurFunctions**(*kBoundedRing*)

Bases: *KBoundedQuotientBasis*

This basis is dual to the k-Schur functions. The expansion is given in Section 4.12 of [LLMSSZ]. When $t = 1$ this basis is equal to the *AffineSchurFunctions* and that basis is more efficient in this case.

REFERENCES:

class sage.combinat.sf.k_dual.**KBoundedQuotient**(*Sym*, *k*, *t='t'*)

Bases: UniqueRepresentation, Parent

Initialization of the ring of Symmetric functions modulo the ideal of monomial symmetric functions which are indexed by partitions whose first part is greater than k.

INPUT:

- Sym – an element of class *sage.combinat.sf.sf.SymmetricFunctions*
- k – positive integer
- R – a ring

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Q = Sym.kBoundedQuotient(3,t=1)
sage: Q
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1
sage: km = Q.km()
sage: km
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 3-
↪bounded monomial basis
sage: F = Q.affineSchur()
sage: F(km(F[3,1,1])) == F[3,1,1]
```

(continues on next page)

(continued from previous page)

```
True
sage: km(F(km([3,2]))) == km[3,2]
True
sage: F[3,2].lift()
m[1, 1, 1, 1, 1] + m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
sage: F[2,1]*F[2,1]
2*F3[1, 1, 1, 1, 1, 1] + 4*F3[2, 1, 1, 1, 1] + 4*F3[2, 2, 1, 1] + 4*F3[2, 2, 2] +
↪2*F3[3, 1, 1, 1] + 4*F3[3, 2, 1] + 2*F3[3, 3]
sage: F[1,2]
Traceback (most recent call last):
...
ValueError: [1, 2] is not an element of 3-Bounded Partitions
sage: F[4,2]
Traceback (most recent call last):
...
ValueError: [4, 2] is not an element of 3-Bounded Partitions
sage: km[2,1]*km[2,1]
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
sage: HLPk = Q.kHallLittlewoodP()
sage: HLPk[2,1]*HLPk[2,1]
4*HLP3[2, 2, 1, 1] + 6*HLP3[2, 2, 2] + 2*HLP3[3, 2, 1] + 2*HLP3[3, 3]
sage: dks = Q.dual_k_Schur()
sage: dks[2,1]*dks[2,1]
2*dks3[1, 1, 1, 1, 1, 1] + 4*dks3[2, 1, 1, 1, 1] + 4*dks3[2, 2, 1, 1] + 4*dks3[2,
↪2, 2] + 2*dks3[3, 1, 1, 1] + 4*dks3[3, 2, 1] + 2*dks3[3, 3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Q = Sym.kBoundedQuotient(Integer(3),t=Integer(1))
>>> Q
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1
>>> km = Q.km()
>>> km
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 3-
↪bounded monomial basis
>>> F = Q.affineSchur()
>>> F(km(F[Integer(3),Integer(1),Integer(1)])) == F[Integer(3),Integer(1),
↪Integer(1)]
True
>>> km(F(km([Integer(3),Integer(2)]))) == km[Integer(3),Integer(2)]
True
>>> F[Integer(3),Integer(2)].lift()
m[1, 1, 1, 1, 1] + m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
>>> F[Integer(2),Integer(1)]*F[Integer(2),Integer(1)]
2*F3[1, 1, 1, 1, 1, 1] + 4*F3[2, 1, 1, 1, 1] + 4*F3[2, 2, 1, 1] + 4*F3[2, 2, 2] +
↪2*F3[3, 1, 1, 1] + 4*F3[3, 2, 1] + 2*F3[3, 3]
>>> F[Integer(1),Integer(2)]
Traceback (most recent call last):
...
ValueError: [1, 2] is not an element of 3-Bounded Partitions
>>> F[Integer(4),Integer(2)]
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: [4, 2] is not an element of 3-Bounded Partitions
>>> km[Integer(2),Integer(1)]*km[Integer(2),Integer(1)]
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
>>> HLPk = Q.kHallLittlewoodP()
>>> HLPk[Integer(2),Integer(1)]*HLPk[Integer(2),Integer(1)]
4*HLP3[2, 2, 1, 1] + 6*HLP3[2, 2, 2] + 2*HLP3[3, 2, 1] + 2*HLP3[3, 3]
>>> dks = Q.dual_k_Schur()
>>> dks[Integer(2),Integer(1)]*dks[Integer(2),Integer(1)]
2*dks3[1, 1, 1, 1, 1, 1] + 4*dks3[2, 1, 1, 1, 1] + 4*dks3[2, 2, 1, 1] + 4*dks3[2,
↪2, 2] + 2*dks3[3, 1, 1, 1] + 4*dks3[3, 2, 1] + 2*dks3[3, 3]
```

```
sage: Q = Sym.kBoundedQuotient(3)
Traceback (most recent call last):
...
TypeError: unable to convert 't' to a rational
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: Q = Sym.kBoundedQuotient(3)
sage: km = Q.km()
sage: F = Q.affineSchur()
sage: F(km(F[3,1,1])) == F[3,1,1]
True
sage: km(F(km([3,2]))) == km[3,2]
True
sage: dks = Q.dual_k_Schur()
sage: HLPk = Q.kHallLittlewoodP()
sage: dks(HLPk(dks[3,1,1])) == dks[3,1,1]
True
sage: km(dks(km([3,2]))) == km[3,2]
True
sage: dks[2,1]*dks[2,1]
(t^3+t^2)*dks3[1, 1, 1, 1, 1, 1] + (2*t^2+2*t)*dks3[2, 1, 1, 1, 1] + (t^
↪2+2*t+1)*dks3[2, 2, 1, 1] + (t^2+2*t+1)*dks3[2, 2, 2] + (t+1)*dks3[3, 1, 1, 1]
↪+ (2*t+2)*dks3[3, 2, 1] + (t+1)*dks3[3, 3]
```

```
>>> from sage.all import *
>>> Q = Sym.kBoundedQuotient(Integer(3))
Traceback (most recent call last):
...
TypeError: unable to convert 't' to a rational
>>> Sym = SymmetricFunctions(QQ['t'].fraction_field())
>>> Q = Sym.kBoundedQuotient(Integer(3))
>>> km = Q.km()
>>> F = Q.affineSchur()
>>> F(km(F[Integer(3),Integer(1),Integer(1)])) == F[Integer(3),Integer(1),
↪Integer(1)]
True
>>> km(F(km([Integer(3),Integer(2)]))) == km[Integer(3),Integer(2)]
True
>>> dks = Q.dual_k_Schur()
>>> HLPk = Q.kHallLittlewoodP()
```

(continues on next page)

(continued from previous page)

```
>>> dks(HLPk(dks[Integer(3),Integer(1),Integer(1)])) == dks[Integer(3),Integer(1),
↪Integer(1)]
True
>>> km(dks(km([Integer(3),Integer(2)]))) == km[Integer(3),Integer(2)]
True
>>> dks[Integer(2),Integer(1)]*dks[Integer(2),Integer(1)]
(t^3+t^2)*dks3[1, 1, 1, 1, 1, 1] + (2*t^2+2*t)*dks3[2, 1, 1, 1, 1] + (t^
↪2+2*t+1)*dks3[2, 2, 1, 1] + (t^2+2*t+1)*dks3[2, 2, 2] + (t+1)*dks3[3, 1, 1, 1]␣
↪+ (2*t+2)*dks3[3, 2, 1] + (t+1)*dks3[3, 3]
```

AffineGrothendieckPolynomial(*la*, *m*)

Return the affine Grothendieck polynomial indexed by the partition `la`. Because this belongs to the completion of the algebra, and hence is an infinite sum, it computes only up to those symmetric functions of degree at most `m`. See `_AffineGrothendieckPolynomial()` for the code.

INPUT:

- `la` – a k-bounded partition
- `m` – integer

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.AffineGrothendieckPolynomial([2,1],4)
2*m3[1, 1, 1] - 8*m3[1, 1, 1, 1] + m3[2, 1] - 3*m3[2, 1, 1] - m3[2, 2]
```

```
>>> from sage.all import *
>>> Q = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1))
>>> Q.AffineGrothendieckPolynomial([Integer(2),Integer(1)],Integer(4))
2*m3[1, 1, 1] - 8*m3[1, 1, 1, 1] + m3[2, 1] - 3*m3[2, 1, 1] - m3[2, 2]
```

F()

The affine Schur basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. This is also equal to the affine Stanley symmetric functions (see `WeylGroups.ElementMethods.stanley_symmetric_function()`) indexed by an affine Grassmannian permutation.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedQuotient(2,t=1).affineSchur()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪2-bounded affine Schur basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).kBoundedQuotient(Integer(2),t=Integer(1)).
↪affineSchur()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪2-bounded affine Schur basis
```

a_realization()

Return a particular realization of `self` (the basis of k-bounded monomials if $t = 1$ and the basis of k-bounded Hall-Littlewood functions otherwise).

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Q = Sym.kBoundedQuotient(3,t=1)
sage: Q.a_realization()
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the
↪3-bounded monomial basis
sage: Q = Sym.kBoundedQuotient(3,t=2)
sage: Q.a_realization()
3-Bounded Quotient of Symmetric Functions over Rational Field with t=2 in the
↪3-bounded Hall-Littlewood P basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Q = Sym.kBoundedQuotient(Integer(3),t=Integer(1))
>>> Q.a_realization()
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the
↪3-bounded monomial basis
>>> Q = Sym.kBoundedQuotient(Integer(3),t=Integer(2))
>>> Q.a_realization()
3-Bounded Quotient of Symmetric Functions over Rational Field with t=2 in the
↪3-bounded Hall-Littlewood P basis
```

affineSchur()

The affine Schur basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. This is also equal to the affine Stanley symmetric functions (see `WeylGroups.ElementMethods.stanley_symmetric_function()`) indexed by an affine Grassmannian permutation.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedQuotient(2,t=1).affineSchur()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the
↪2-bounded affine Schur basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).kBoundedQuotient(Integer(2),t=Integer(1)).
↪affineSchur()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the
↪2-bounded affine Schur basis
```

ambient()

Returns the Symmetric Functions over the same ring as `self`. This is needed to realize our ring as a quotient.

an_element()

Return an element of the quotient ring of k-bounded symmetric functions. This method is here to make the TestSuite run properly.

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.an_element()
2*m3[] + 2*m3[1] + 3*m3[2]
```

```
>>> from sage.all import *
>>> Q = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> Q.an_element()
2*m3[] + 2*m3[1] + 3*m3[2]
```

dks()

The dual k-Schur basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. At $t = 1$ this is also equal to the affine Schur basis and calculations will be faster using elements in the *affineSchur()* basis.

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(2).dual_k_
↪Schur()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the dual 2-Schur basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(Integer(2)).
↪dual_k_Schur()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the dual 2-Schur basis
```

dual_k_Schur()

The dual k-Schur basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. At $t = 1$ this is also equal to the affine Schur basis and calculations will be faster using elements in the *affineSchur()* basis.

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(2).dual_k_
↪Schur()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the dual 2-Schur basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(Integer(2)).
↪dual_k_Schur()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the dual 2-Schur basis
```

kHLP()

The Hall-Littlewood P basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. At $t = 1$ this basis is equal to the k-bounded monomial basis and calculations will be faster using elements in the k-bounded monomial basis (see *kmonomial()*).

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(2).
↪kHallLittlewoodP()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 2-bounded Hall-Littlewood P␣
↪basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(Integer(2)).
↪kHallLittlewoodP()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 2-bounded Hall-Littlewood P␣
↪basis
```

kHallLittlewoodP()

The Hall-Littlewood P basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. At $t = 1$ this basis is equal to the k-bounded monomial basis and calculations will be faster using elements in the k-bounded monomial basis (see *kmonomial()*).

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(2).
↪kHallLittlewoodP()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 2-bounded Hall-Littlewood P␣
↪basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(Integer(2)).
↪kHallLittlewoodP()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 2-bounded Hall-Littlewood P␣
↪basis
```

km()

The monomial basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedQuotient(2,t=1).kmonomial()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪2-bounded monomial basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).kBoundedQuotient(Integer(2),t=Integer(1)).
↪kmonomial()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪2-bounded monomial basis
```

kmonomial()

The monomial basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedQuotient(2,t=1).kmonomial()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪2-bounded monomial basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).kBoundedQuotient(Integer(2),t=Integer(1)).
```

(continues on next page)

(continued from previous page)

```
↪kmonomial()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪2-bounded monomial basis
```

lift(*la*)

Give the lift map from the quotient ring of k-bounded symmetric functions to the symmetric functions. This method is here to make the TestSuite run properly.

INPUT:

- `la` – a k-bounded partition

OUTPUT:

- **The monomial element or a Hall-Littlewood P element of the symmetric functions** indexed by the partition `la`.

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.lift([2,1])
m[2, 1]
sage: Q = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3)
sage: Q.lift([2,1])
HLP[2, 1]
```

```
>>> from sage.all import *
>>> Q = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1))
>>> Q.lift([Integer(2),Integer(1)])
m[2, 1]
>>> Q = SymmetricFunctions(QQ['t'].fraction_field()).
↪kBoundedQuotient(Integer(3))
>>> Q.lift([Integer(2),Integer(1)])
HLP[2, 1]
```

one()

Return the unit of the quotient ring of k-bounded symmetric functions. This method is here to make the TestSuite run properly.

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.one()
m3[]
```

```
>>> from sage.all import *
>>> Q = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1))
>>> Q.one()
m3[]
```

realizations()

A list of realizations of the k-bounded quotient.

EXAMPLES:

```
sage: kQ = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3)
sage: kQ.realizations()
[3-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 3-bounded monomial basis, 3-
↪Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 3-bounded Hall-Littlewood P␣
↪basis, 3-Bounded Quotient of Symmetric Functions over Fraction Field of␣
↪Univariate Polynomial Ring in t over Rational Field in the 3-bounded affine␣
↪Schur basis, 3-Bounded Quotient of Symmetric Functions over Fraction Field␣
↪of Univariate Polynomial Ring in t over Rational Field in the dual 3-Schur␣
↪basis]
sage: HLP = kQ.ambient().hall_littlewood().P()
sage: all( rzn(HLP[3,2,1]).lift() == HLP[3,2,1] for rzn in kQ.realizations())
True
sage: kQ = SymmetricFunctions(QQ).kBoundedQuotient(3,1)
sage: kQ.realizations()
[3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in␣
↪the 3-bounded monomial basis, 3-Bounded Quotient of Symmetric Functions␣
↪over Rational Field with t=1 in the 3-bounded Hall-Littlewood P basis, 3-
↪Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪3-bounded affine Schur basis, 3-Bounded Quotient of Symmetric Functions␣
↪over Rational Field with t=1 in the dual 3-Schur basis]
sage: m = kQ.ambient().m()
sage: all( rzn(m[3,2,1]).lift() == m[3,2,1] for rzn in kQ.realizations())
True
```

```
>>> from sage.all import *
>>> kQ = SymmetricFunctions(QQ['t'].fraction_field()).
↪kBoundedQuotient(Integer(3))
>>> kQ.realizations()
[3-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 3-bounded monomial basis, 3-
↪Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 3-bounded Hall-Littlewood P␣
↪basis, 3-Bounded Quotient of Symmetric Functions over Fraction Field of␣
↪Univariate Polynomial Ring in t over Rational Field in the 3-bounded affine␣
↪Schur basis, 3-Bounded Quotient of Symmetric Functions over Fraction Field␣
↪of Univariate Polynomial Ring in t over Rational Field in the dual 3-Schur␣
↪basis]
>>> HLP = kQ.ambient().hall_littlewood().P()
>>> all( rzn(HLP[Integer(3),Integer(2),Integer(1)]).lift() == HLP[Integer(3),
↪Integer(2),Integer(1)] for rzn in kQ.realizations())
True
>>> kQ = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),Integer(1))
>>> kQ.realizations()
[3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in␣
↪the 3-bounded monomial basis, 3-Bounded Quotient of Symmetric Functions␣
↪over Rational Field with t=1 in the 3-bounded Hall-Littlewood P basis, 3-
↪Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪3-bounded affine Schur basis, 3-Bounded Quotient of Symmetric Functions␣
↪over Rational Field with t=1 in the dual 3-Schur basis]
>>> m = kQ.ambient().m()
```

(continues on next page)

(continued from previous page)

```
>>> all( rzn(m[Integer(3),Integer(2),Integer(1)]).lift() == m[Integer(3),
↪Integer(2),Integer(1)] for rzn in kQ.realizations())
True
```

retract(*la*)

Give the retract map from the symmetric functions to the quotient ring of k-bounded symmetric functions. This method is here to make the TestSuite run properly.

INPUT:

- `la` – a partition

OUTPUT: the monomial element of the k-bounded quotient indexed by `la`

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.retract([2,1])
m3[2, 1]
```

```
>>> from sage.all import *
>>> Q = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1))
>>> Q.retract([Integer(2),Integer(1)])
m3[2, 1]
```

class sage.combinat.sf.k_dual.**KBoundedQuotientBases**(*base*)

Bases: `Category_realization_of_parent`

The category of bases for the k-bounded subspace of symmetric functions.

class ElementMethods

Bases: `object`

class ParentMethods

Bases: `object`

ambient()

Return the symmetric functions.

EXAMPLES:

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.ambient()
Symmetric Functions over Rational Field
```

```
>>> from sage.all import *
>>> km = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).
↪km()
>>> km.ambient()
Symmetric Functions over Rational Field
```

antipode(*element*)

Return the antipode of `element` via lifting to the symmetric functions and then retracting into the k-bounded quotient basis.

INPUT:

- element – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: dks3 = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).dual_k_Schur()
sage: dks3[3,2].antipode()
-dks3[1, 1, 1, 1, 1]
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km[3,2].antipode()
m3[3, 2]
sage: km.antipode(km[3,2])
m3[3, 2]
sage: m = SymmetricFunctions(QQ).m()
sage: m[3,2].antipode()
m[3, 2] + 2*m[5]
```

```
>>> from sage.all import *
>>> dks3 = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),
↪t=Integer(1)).dual_k_Schur()
>>> dks3[Integer(3),Integer(2)].antipode()
-dks3[1, 1, 1, 1, 1]
>>> km = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).
↪km()
>>> km[Integer(3),Integer(2)].antipode()
m3[3, 2]
>>> km.antipode(km[Integer(3),Integer(2)])
m3[3, 2]
>>> m = SymmetricFunctions(QQ).m()
>>> m[Integer(3),Integer(2)].antipode()
m[3, 2] + 2*m[5]
```

```
sage: km = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3).
↪km()
sage: km[1,1,1,1].antipode()
(t^3-3*t^2+3*t)*m3[1, 1, 1, 1] + (-t^2+2*t)*m3[2, 1, 1] + t*m3[2, 2] +␣
↪t*m3[3, 1]
sage: kHP = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(3).kHLP()
sage: kHP[2,2].antipode()
(t^9-t^6-t^5+t^2)*HLP3[1, 1, 1, 1] + (t^6-t^3-t^2+t)*HLP3[2, 1, 1] + (t^5-
↪t^2+1)*HLP3[2, 2] + (t^4-t)*HLP3[3, 1]
sage: dks = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(3).dks()
sage: dks[2,2].antipode()
dks3[2, 2]
sage: dks[3,2].antipode()
-t^2*dks3[1, 1, 1, 1, 1] + (t^2-1)*dks3[2, 2, 1] + (-t^5+t)*dks3[3, 2]
```

```
>>> from sage.all import *
>>> km = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(Integer(3)).km()
>>> km[Integer(1),Integer(1),Integer(1),Integer(1)].antipode()
(t^3-3*t^2+3*t)*m3[1, 1, 1, 1] + (-t^2+2*t)*m3[2, 1, 1] + t*m3[2, 2] +␣
↪t*m3[3, 1]
```

(continues on next page)

(continued from previous page)

```
>>> kHP = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(Integer(3)).kHLP()
>>> kHP[Integer(2),Integer(2)].antipode()
(t^9-t^6-t^5+t^2)*HLP3[1, 1, 1, 1] + (t^6-t^3-t^2+t)*HLP3[2, 1, 1] + (t^5-
↪t^2+1)*HLP3[2, 2] + (t^4-t)*HLP3[3, 1]
>>> dks = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(Integer(3)).dks()
>>> dks[Integer(2),Integer(2)].antipode()
dks3[2, 2]
>>> dks[Integer(3),Integer(2)].antipode()
-t^2*dks3[1, 1, 1, 1, 1] + (t^2-1)*dks3[2, 2, 1] + (-t^5+t)*dks3[3, 2]
```

coproduct (*element*)

Return the coproduct of `element` via lifting to the symmetric functions and then returning to the k-bounded quotient basis. This method is implemented for all t but is (weakly) conjectured to not be the correct operation for arbitrary t because the coproduct on dual-k-Schur functions does not have a positive expansion.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Q3 = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: km = Q3.km()
sage: km[3,2].coproduct()
m3[] # m3[3, 2] + m3[2] # m3[3] + m3[3] # m3[2] + m3[3, 2] # m3[]
sage: dks3 = Q3.dual_k_Schur()
sage: dks3[2,2].coproduct()
dks3[] # dks3[2, 2] + dks3[1] # dks3[2, 1] + dks3[1, 1] # dks3[1, 1] +␣
↪dks3[2] # dks3[2] + dks3[2, 1] # dks3[1] + dks3[2, 2] # dks3[]
```

```
>>> from sage.all import *
>>> Q3 = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1))
>>> km = Q3.km()
>>> km[Integer(3),Integer(2)].coproduct()
m3[] # m3[3, 2] + m3[2] # m3[3] + m3[3] # m3[2] + m3[3, 2] # m3[]
>>> dks3 = Q3.dual_k_Schur()
>>> dks3[Integer(2),Integer(2)].coproduct()
dks3[] # dks3[2, 2] + dks3[1] # dks3[2, 1] + dks3[1, 1] # dks3[1, 1] +␣
↪dks3[2] # dks3[2] + dks3[2, 1] # dks3[1] + dks3[2, 2] # dks3[]
```

```
sage: Q3t = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3)
sage: km = Q3t.km()
sage: km[3,2].coproduct()
m3[] # m3[3, 2] + m3[2] # m3[3] + m3[3] # m3[2] + m3[3, 2] # m3[]
sage: dks = Q3t.dks()
sage: dks[2,1,1].coproduct()
dks3[] # dks3[2, 1, 1] + (-t+1)*dks3[1] # dks3[1, 1, 1] + dks3[1] #␣
↪dks3[2, 1] + (-t+1)*dks3[1, 1] # dks3[1, 1] + dks3[1, 1] # dks3[2] + (-
↪t+1)*dks3[1, 1, 1] # dks3[1] + dks3[2] # dks3[1, 1] + dks3[2, 1] #␣
↪dks3[1] + dks3[2, 1, 1] # dks3[]
sage: kHLP = Q3t.kHLP()
```

(continues on next page)

(continued from previous page)

```
sage: kHLP[2,1].coproduct()
HLP3[] # HLP3[2, 1] + (-t^2+1)*HLP3[1] # HLP3[1, 1] + HLP3[1] # HLP3[2] +
↪(-t^2+1)*HLP3[1, 1] # HLP3[1] + HLP3[2] # HLP3[1] + HLP3[2, 1] # HLP3[]
sage: km.coproduct(km[3,2])
m3[] # m3[3, 2] + m3[2] # m3[3] + m3[3] # m3[2] + m3[3, 2] # m3[]
```

```
>>> from sage.all import *
>>> Q3t = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(Integer(3))
>>> km = Q3t.km()
>>> km[Integer(3),Integer(2)].coproduct()
m3[] # m3[3, 2] + m3[2] # m3[3] + m3[3] # m3[2] + m3[3, 2] # m3[]
>>> dks = Q3t.dks()
>>> dks[Integer(2),Integer(1),Integer(1)].coproduct()
dks3[] # dks3[2, 1, 1] + (-t+1)*dks3[1] # dks3[1, 1, 1] + dks3[1] #
↪dks3[2, 1] + (-t+1)*dks3[1, 1] # dks3[1, 1] + dks3[1, 1] # dks3[2] + (-
↪t+1)*dks3[1, 1, 1] # dks3[1] + dks3[2] # dks3[1, 1] + dks3[2, 1] #
↪dks3[1] + dks3[2, 1, 1] # dks3[]
>>> kHLP = Q3t.kHLP()
>>> kHLP[Integer(2),Integer(1)].coproduct()
HLP3[] # HLP3[2, 1] + (-t^2+1)*HLP3[1] # HLP3[1, 1] + HLP3[1] # HLP3[2] +
↪(-t^2+1)*HLP3[1, 1] # HLP3[1] + HLP3[2] # HLP3[1] + HLP3[2, 1] # HLP3[]
>>> km.coproduct(km[Integer(3),Integer(2)])
m3[] # m3[3, 2] + m3[2] # m3[3] + m3[3] # m3[2] + m3[3, 2] # m3[]
```

counit(*element*)

Return the counit of `element`.

The counit is the constant term of `element`.

INPUT:

- `element` – an element in a basis

EXAMPLES:

```
sage: km = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3).
↪km()
sage: f = 2*km[2,1] - 3*km([])
sage: f.counit()
-3
sage: km.counit(f)
-3
```

```
>>> from sage.all import *
>>> km = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(Integer(3)).km()
>>> f = Integer(2)*km[Integer(2),Integer(1)] - Integer(3)*km([])
>>> f.counit()
-3
>>> km.counit(f)
-3
```

degree_on_basis(*b*)

Return the degree of the basis element indexed by `b`.

INPUT:

- b – a partition

EXAMPLES:

```
sage: F = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).affineSchur()
sage: F.degree_on_basis(Partition([3,2]))
5
```

```
>>> from sage.all import *
>>> F = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).
↪affineSchur()
>>> F.degree_on_basis(Partition([Integer(3),Integer(2)]))
5
```

indices()

The set of k-bounded partitions of all nonnegative integers.

EXAMPLES:

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.indices()
3-Bounded Partitions
```

```
>>> from sage.all import *
>>> km = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).
↪km()
>>> km.indices()
3-Bounded Partitions
```

lift(*la*)

Implement the lift map from the basis self to the monomial basis of symmetric functions.

INPUT:

- la – a k-bounded partition

OUTPUT: a symmetric function in the monomial basis

EXAMPLES:

```
sage: F = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).affineSchur()
sage: F.lift([3,1])
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1]
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: dks = Sym.kBoundedQuotient(3).dual_k_Schur()
sage: dks.lift([3,1])
t^5*HLP[1, 1, 1, 1] + t^2*HLP[2, 1, 1] + t*HLP[2, 2] + HLP[3, 1]
sage: dks = Sym.kBoundedQuotient(3,t=1).dual_k_Schur()
sage: dks.lift([3,1])
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1]
```

```
>>> from sage.all import *
>>> F = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).
↪affineSchur()
>>> F.lift([Integer(3),Integer(1)])
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1]
```

(continues on next page)

(continued from previous page)

```
>>> Sym = SymmetricFunctions(QQ['t'].fraction_field())
>>> dks = Sym.kBoundedQuotient(Integer(3)).dual_k_Schur()
>>> dks.lift([Integer(3),Integer(1)])
t^5*HLP[1, 1, 1, 1] + t^2*HLP[2, 1, 1] + t*HLP[2, 2] + HLP[3, 1]
>>> dks = Sym.kBoundedQuotient(Integer(3),t=Integer(1)).dual_k_Schur()
>>> dks.lift([Integer(3),Integer(1)])
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1]
```

one_basis()

Return the basis element indexing 1.

EXAMPLES:

```
sage: F = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).affineSchur()
sage: F.one()   # indirect doctest
F3[]
```

```
>>> from sage.all import *
>>> F = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).
↪affineSchur()
>>> F.one()   # indirect doctest
F3[]
```

product(*x*, *y*)

Return the product of two elements x and y.

INPUT:

- x, y – elements of the k-bounded quotient of symmetric functions

OUTPUT: a k-bounded symmetric function in the dual k-Schur function basis

EXAMPLES:

```
sage: dks3 = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).dual_k_Schur()
sage: dks3.product(dks3[2,1],dks3[1,1])
2*dks3[1, 1, 1, 1, 1] + 2*dks3[2, 1, 1, 1] + 2*dks3[2, 2, 1] + dks3[3, 1,␣
↪1] + dks3[3, 2]
sage: dks3.product(dks3[2,1]+dks3[1], dks3[1,1])
dks3[1, 1, 1] + 2*dks3[1, 1, 1, 1, 1] + dks3[2, 1] + 2*dks3[2, 1, 1, 1] +␣
↪2*dks3[2, 2, 1] + dks3[3, 1, 1] + dks3[3, 2]
sage: dks3.product(dks3[2,1]+dks3[1], dks3([]))
dks3[1] + dks3[2, 1]
sage: dks3.product(dks3([]), dks3([]))
dks3[]
sage: dks3.product(dks3([]), dks3([4,1]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪Bounded Quotient of Symmetric Functions over Rational Field with t=1 in␣
↪the dual 3-Schur basis)
```

```
>>> from sage.all import *
>>> dks3 = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),
↪t=Integer(1)).dual_k_Schur()
```

(continues on next page)

(continued from previous page)

```
>>> dks3.product(dks3[Integer(2),Integer(1)],dks3[Integer(1),Integer(1)])
2*dks3[1, 1, 1, 1, 1] + 2*dks3[2, 1, 1, 1] + 2*dks3[2, 2, 1] + dks3[3, 1,␣
↪1] + dks3[3, 2]
>>> dks3.product(dks3[Integer(2),Integer(1)]+dks3[Integer(1)],␣
↪dks3[Integer(1),Integer(1)])
dks3[1, 1, 1] + 2*dks3[1, 1, 1, 1, 1] + dks3[2, 1] + 2*dks3[2, 1, 1, 1] +␣
↪2*dks3[2, 2, 1] + dks3[3, 1, 1] + dks3[3, 2]
>>> dks3.product(dks3[Integer(2),Integer(1)]+dks3[Integer(1)], dks3([]))
dks3[1] + dks3[2, 1]
>>> dks3.product(dks3([]), dks3([]))
dks3[]
>>> dks3.product(dks3([]), dks3([Integer(4),Integer(1)]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪Bounded Quotient of Symmetric Functions over Rational Field with t=1 in␣
↪the dual 3-Schur basis)
```

```
sage: dks3 = SymmetricFunctions(QQ['t'].fraction_field()).
↪kBoundedQuotient(3).dual_k_Schur()
sage: dks3.product(dks3[2,1],dks3[1,1])
(t^2+t)*dks3[1, 1, 1, 1, 1] + (t+1)*dks3[2, 1, 1, 1] + (t+1)*dks3[2, 2,␣
↪1] + dks3[3, 1, 1] + dks3[3, 2]
sage: dks3.product(dks3[2,1]+dks3[1], dks3[1,1])
dks3[1, 1, 1] + (t^2+t)*dks3[1, 1, 1, 1, 1] + dks3[2, 1] + (t+1)*dks3[2,␣
↪1, 1, 1] + (t+1)*dks3[2, 2, 1] + dks3[3, 1, 1] + dks3[3, 2]
sage: dks3.product(dks3[2,1]+dks3[1], dks3([]))
dks3[1] + dks3[2, 1]
sage: dks3.product(dks3([]), dks3([]))
dks3[]
```

```
>>> from sage.all import *
>>> dks3 = SymmetricFunctions(QQ['t'].fraction_field()).
↪kBoundedQuotient(Integer(3)).dual_k_Schur()
>>> dks3.product(dks3[Integer(2),Integer(1)],dks3[Integer(1),Integer(1)])
(t^2+t)*dks3[1, 1, 1, 1, 1] + (t+1)*dks3[2, 1, 1, 1] + (t+1)*dks3[2, 2,␣
↪1] + dks3[3, 1, 1] + dks3[3, 2]
>>> dks3.product(dks3[Integer(2),Integer(1)]+dks3[Integer(1)],␣
↪dks3[Integer(1),Integer(1)])
dks3[1, 1, 1] + (t^2+t)*dks3[1, 1, 1, 1, 1] + dks3[2, 1] + (t+1)*dks3[2,␣
↪1, 1, 1] + (t+1)*dks3[2, 2, 1] + dks3[3, 1, 1] + dks3[3, 2]
>>> dks3.product(dks3[Integer(2),Integer(1)]+dks3[Integer(1)], dks3([]))
dks3[1] + dks3[2, 1]
>>> dks3.product(dks3([]), dks3([]))
dks3[]
```

```
sage: F = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).affineSchur()
sage: F.product(F[2,1],F[1,1])
2*F3[1, 1, 1, 1, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2, 2, 1] + F3[3, 1, 1] +␣
↪F3[3, 2]
sage: F.product(F[2,1]+F[1], F[1,1])
```

(continues on next page)

(continued from previous page)

```
F3[1, 1, 1] + 2*F3[1, 1, 1, 1, 1] + F3[2, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2,␣
↪2, 1] + F3[3, 1, 1] + F3[3, 2]
sage: F.product(F[2,1]+F[1], F([]))
F3[1] + F3[2, 1]
sage: F.product(F([]), F([]))
F3[]
sage: F.product(F([]), F([4,1]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪Bounded Quotient of Symmetric Functions over Rational Field with t=1 in␣
↪the 3-bounded affine Schur basis)
```

```
>>> from sage.all import *
>>> F = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).
↪affineSchur()
>>> F.product(F[Integer(2),Integer(1)],F[Integer(1),Integer(1)])
2*F3[1, 1, 1, 1, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2, 2, 1] + F3[3, 1, 1] +␣
↪F3[3, 2]
>>> F.product(F[Integer(2),Integer(1)]+F[Integer(1)], F[Integer(1),
↪Integer(1)])
F3[1, 1, 1] + 2*F3[1, 1, 1, 1, 1] + F3[2, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2,␣
↪2, 1] + F3[3, 1, 1] + F3[3, 2]
>>> F.product(F[Integer(2),Integer(1)]+F[Integer(1)], F([]))
F3[1] + F3[2, 1]
>>> F.product(F([]), F([]))
F3[]
>>> F.product(F([]), F([Integer(4),Integer(1)]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪Bounded Quotient of Symmetric Functions over Rational Field with t=1 in␣
↪the 3-bounded affine Schur basis)
```

```
sage: F = SymmetricFunctions(QQ['t'].fraction_field()).
↪kBoundedQuotient(3).affineSchur()
sage: F.product(F[2,1],F[1,1])
2*F3[1, 1, 1, 1, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2, 2, 1] + F3[3, 1, 1] +␣
↪F3[3, 2]
sage: F.product(F[2,1],F[2])
(t^4+t^3-2*t^2+1)*F3[1, 1, 1, 1, 1] + (-t^2+t+1)*F3[2, 1, 1, 1] + (-t^
↪2+t+2)*F3[2, 2, 1] + (t+1)*F3[3, 1, 1] + (t+1)*F3[3, 2]
sage: F.product(F[2,1]+F[1], F[1,1])
F3[1, 1, 1] + 2*F3[1, 1, 1, 1, 1] + F3[2, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2,␣
↪2, 1] + F3[3, 1, 1] + F3[3, 2]
sage: F.product(F[2,1]+F[1], F([]))
F3[1] + F3[2, 1]
sage: F.product(F([]), F([]))
F3[]
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> F = SymmetricFunctions(QQ['t'].fraction_field()).
↪kBoundedQuotient(Integer(3)).affineSchur()
>>> F.product(F[Integer(2),Integer(1)],F[Integer(1),Integer(1)])
2*F3[1, 1, 1, 1, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2, 2, 1] + F3[3, 1, 1] +␣
↪F3[3, 2]
>>> F.product(F[Integer(2),Integer(1)],F[Integer(2)])
(t^4+t^3-2*t^2+1)*F3[1, 1, 1, 1, 1] + (-t^2+t+1)*F3[2, 1, 1, 1] + (-t^
↪2+t+2)*F3[2, 2, 1] + (t+1)*F3[3, 1, 1] + (t+1)*F3[3, 2]
>>> F.product(F[Integer(2),Integer(1)]+F[Integer(1)], F[Integer(1),
↪Integer(1)])
F3[1, 1, 1] + 2*F3[1, 1, 1, 1, 1] + F3[2, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2,␣
↪2, 1] + F3[3, 1, 1] + F3[3, 2]
>>> F.product(F[Integer(2),Integer(1)]+F[Integer(1)], F([]))
F3[1] + F3[2, 1]
>>> F.product(F([]), F([]))
F3[]
```

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.product(km[2,1],km[2,1])
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
sage: Q3 = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3)
sage: km = Q3.km()
sage: km.product(km[2,1],km[2,1])
(t^5+7*t^4-8*t^3-28*t^2+47*t-19)*m3[1, 1, 1, 1, 1, 1] + (t^4-3*t^3-9*t^
↪2+23*t-12)*m3[2, 1, 1, 1, 1] + (-t^3-3*t^2+11*t-3)*m3[2, 2, 1, 1] + (-t^
↪2+5*t+2)*m3[2, 2, 2] + (6*t-6)*m3[3, 1, 1, 1] + (3*t-1)*m3[3, 2, 1] +␣
↪(t+1)*m3[3, 3]
sage: dks = Q3.dual_k_Schur()
sage: km.product(dks[2,1],dks[1,1])
20*m3[1, 1, 1, 1, 1] + 9*m3[2, 1, 1, 1] + 4*m3[2, 2, 1] + 2*m3[3, 1, 1] +␣
↪m3[3, 2]
```

```
>>> from sage.all import *
>>> km = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).
↪km()
>>> km.product(km[Integer(2),Integer(1)],km[Integer(2),Integer(1)])
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
>>> Q3 = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(Integer(3))
>>> km = Q3.km()
>>> km.product(km[Integer(2),Integer(1)],km[Integer(2),Integer(1)])
(t^5+7*t^4-8*t^3-28*t^2+47*t-19)*m3[1, 1, 1, 1, 1, 1] + (t^4-3*t^3-9*t^
↪2+23*t-12)*m3[2, 1, 1, 1, 1] + (-t^3-3*t^2+11*t-3)*m3[2, 2, 1, 1] + (-t^
↪2+5*t+2)*m3[2, 2, 2] + (6*t-6)*m3[3, 1, 1, 1] + (3*t-1)*m3[3, 2, 1] +␣
↪(t+1)*m3[3, 3]
>>> dks = Q3.dual_k_Schur()
>>> km.product(dks[Integer(2),Integer(1)],dks[Integer(1),Integer(1)])
20*m3[1, 1, 1, 1, 1] + 9*m3[2, 1, 1, 1] + 4*m3[2, 2, 1] + 2*m3[3, 1, 1] +␣
↪m3[3, 2]
```

retract(*la*)

Give the retract map from the symmetric functions to the quotient ring of k-bounded symmetric functions. This method is here to make the TestSuite run properly.

INPUT:

- `la` – a partition

OUTPUT: the monomial element of the k-bounded quotient indexed by `la`

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.retract([2,1])
m3[2, 1]
```

```
>>> from sage.all import *
>>> Q = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1))
>>> Q.retract([Integer(2),Integer(1)])
m3[2, 1]
```

super_categories()

The super categories of `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: from sage.combinat.sf.k_dual import KBoundedQuotientBases
sage: Q = Sym.kBoundedQuotient(3,t=1)
sage: KQB = KBoundedQuotientBases(Q)
sage: KQB.super_categories()
[Category of realizations of 3-Bounded Quotient of Symmetric Functions over
↪Univariate Polynomial Ring in t over Rational Field with t=1,
 Join of Category of graded Hopf algebras with basis over Univariate
↪Polynomial Ring in t over Rational Field
     and Category of quotients of algebras over Univariate Polynomial Ring in
↪t over Rational Field
     and Category of quotients of graded modules with basis over Univariate
↪Polynomial Ring in t over Rational Field]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'])
>>> from sage.combinat.sf.k_dual import KBoundedQuotientBases
>>> Q = Sym.kBoundedQuotient(Integer(3),t=Integer(1))
>>> KQB = KBoundedQuotientBases(Q)
>>> KQB.super_categories()
[Category of realizations of 3-Bounded Quotient of Symmetric Functions over
↪Univariate Polynomial Ring in t over Rational Field with t=1,
 Join of Category of graded Hopf algebras with basis over Univariate
↪Polynomial Ring in t over Rational Field
     and Category of quotients of algebras over Univariate Polynomial Ring in
↪t over Rational Field
     and Category of quotients of graded modules with basis over Univariate
↪Polynomial Ring in t over Rational Field]
```

class sage.combinat.sf.k_dual.**KBoundedQuotientBasis**(*kBoundedRing*, *prefix*)

Bases: *CombinatorialFreeModule*

Abstract base class for the bases of the k-bounded quotient.

class sage.combinat.sf.k_dual.**kMonomial**(*kBoundedRing*)

Bases: *KBoundedQuotientBasis*

The basis of monomial symmetric functions indexed by partitions with first part less than or equal to k.

lift(*la*)

Implement the lift function on the monomial basis. Given a k-bounded partition `la`, the lift will return the corresponding monomial basis element.

INPUT:

- `la` – a k-bounded partition

OUTPUT: a monomial symmetric function

EXAMPLES:

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.lift(Partition([3,1]))
m[3, 1]
sage: km.lift([])
m[]
sage: km.lift(Partition([4,1]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪3-bounded monomial basis)
```

```
>>> from sage.all import *
>>> km = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).km()
>>> km.lift(Partition([Integer(3),Integer(1)]))
m[3, 1]
>>> km.lift([])
m[]
>>> km.lift(Partition([Integer(4),Integer(1)]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the␣
↪3-bounded monomial basis)
```

retract(*la*)

Implement the retract function on the monomial basis. Given a partition `la`, the retract will return the corresponding k-bounded monomial basis element if `la` is k-bounded; zero otherwise.

INPUT:

- `la` – a partition

OUTPUT: a k-bounded monomial symmetric function in the k-quotient of symmetric functions

EXAMPLES:

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.retract(Partition([3,1]))
m3[3, 1]
sage: km.retract(Partition([4,1]))
```

(continues on next page)

(continued from previous page)

```
0
sage: km.retract([])
m3[]
sage: m = SymmetricFunctions(QQ).m()
sage: km(m[3, 1])
m3[3, 1]
sage: km(m[4, 1])
0
```

```
>>> from sage.all import *
>>> km = SymmetricFunctions(QQ).kBoundedQuotient(Integer(3),t=Integer(1)).km()
>>> km.retract(Partition([Integer(3),Integer(1)]))
m3[3, 1]
>>> km.retract(Partition([Integer(4),Integer(1)]))
0
>>> km.retract([])
m3[]
>>> m = SymmetricFunctions(QQ).m()
>>> km(m[Integer(3), Integer(1)])
m3[3, 1]
>>> km(m[Integer(4), Integer(1)])
0
```

```
sage: km = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3).km()
sage: km.retract(Partition([3,1]))
m3[3, 1]
sage: km.retract(Partition([4,1]))
(t^4+t^3-9*t^2+11*t-4)*m3[1, 1, 1, 1, 1] + (-3*t^2+6*t-3)*m3[2, 1, 1, 1] + (-
↪t^2+3*t-2)*m3[2, 2, 1] + (2*t-2)*m3[3, 1, 1] + (t-1)*m3[3, 2]
sage: m = SymmetricFunctions(FractionField(QQ['t'])).m()
sage: km(m[3, 1])
m3[3, 1]
sage: km(m[4, 1])
(t^4+t^3-9*t^2+11*t-4)*m3[1, 1, 1, 1, 1] + (-3*t^2+6*t-3)*m3[2, 1, 1, 1] + (-
↪t^2+3*t-2)*m3[2, 2, 1] + (2*t-2)*m3[3, 1, 1] + (t-1)*m3[3, 2]
```

```
>>> from sage.all import *
>>> km = SymmetricFunctions(FractionField(QQ['t'])).
↪kBoundedQuotient(Integer(3)).km()
>>> km.retract(Partition([Integer(3),Integer(1)]))
m3[3, 1]
>>> km.retract(Partition([Integer(4),Integer(1)]))
(t^4+t^3-9*t^2+11*t-4)*m3[1, 1, 1, 1, 1] + (-3*t^2+6*t-3)*m3[2, 1, 1, 1] + (-
↪t^2+3*t-2)*m3[2, 2, 1] + (2*t-2)*m3[3, 1, 1] + (t-1)*m3[3, 2]
>>> m = SymmetricFunctions(FractionField(QQ['t'])).m()
>>> km(m[Integer(3), Integer(1)])
m3[3, 1]
>>> km(m[Integer(4), Integer(1)])
(t^4+t^3-9*t^2+11*t-4)*m3[1, 1, 1, 1, 1] + (-3*t^2+6*t-3)*m3[2, 1, 1, 1] + (-
↪t^2+3*t-2)*m3[2, 2, 1] + (2*t-2)*m3[3, 1, 1] + (t-1)*m3[3, 2]
```

class sage.combinat.sf.k_dual.**kbounded_HallLittlewoodP**(*kBoundedRing*)

Bases: *KBoundedQuotientBasis*

The basis of P Hall-Littlewood symmetric functions indexed by partitions with first part less than or equal to k.

lift(*la*)

Implement the lift function on the Hall-Littlewood P basis. Given a k-bounded partition `la`, the lift will return the corresponding Hall-Littlewood P basis element.

INPUT:

- `la` – a k-bounded partition

OUTPUT: a Hall-Littlewood symmetric function

EXAMPLES:

```
sage: kHLP = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3).
↪kHallLittlewoodP()
sage: kHLP.lift(Partition([3,1]))
HLP[3, 1]
sage: kHLP.lift([])
HLP[]
sage: kHLP.lift(Partition([4,1]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 3-bounded Hall-Littlewood P␣
↪basis)
```

```
>>> from sage.all import *
>>> kHLP = SymmetricFunctions(QQ['t'].fraction_field()).
↪kBoundedQuotient(Integer(3)).kHallLittlewoodP()
>>> kHLP.lift(Partition([Integer(3),Integer(1)]))
HLP[3, 1]
>>> kHLP.lift([])
HLP[]
>>> kHLP.lift(Partition([Integer(4),Integer(1)]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field in the 3-bounded Hall-Littlewood P␣
↪basis)
```

retract(*la*)

Implement the retract function on the Hall-Littlewood P basis. Given a partition `la`, the retract will return the corresponding k-bounded Hall-Littlewood P basis element if `la` is k-bounded; zero otherwise.

INPUT:

- `la` – a partition

OUTPUT: a k-bounded Hall-Littlewood P symmetric function in the k-quotient of symmetric functions

EXAMPLES:

```
sage: kHLP = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3).
↪kHallLittlewoodP()
sage: kHLP.retract(Partition([3,1]))
HLP3[3, 1]
sage: kHLP.retract(Partition([4,1]))
0
sage: kHLP.retract([])
HLP3[]
sage: m = kHLP.realization_of().ambient().m()
sage: kHLP(m[2,2])
(t^4-t^3-t+1)*HLP3[1, 1, 1, 1] + (t-1)*HLP3[2, 1, 1] + HLP3[2, 2]
```

```
>>> from sage.all import *
>>> kHLP = SymmetricFunctions(QQ['t'].fraction_field()).
↪kBoundedQuotient(Integer(3)).kHallLittlewoodP()
>>> kHLP.retract(Partition([Integer(3),Integer(1)]))
HLP3[3, 1]
>>> kHLP.retract(Partition([Integer(4),Integer(1)]))
0
>>> kHLP.retract([])
HLP3[]
>>> m = kHLP.realization_of().ambient().m()
>>> kHLP(m[Integer(2),Integer(2)])
(t^4-t^3-t+1)*HLP3[1, 1, 1, 1] + (t-1)*HLP3[2, 1, 1] + HLP3[2, 2]
```

5.1.290 Kostka-Foulkes Polynomials

Based on the algorithms in John Stembridge's SF package for Maple which can be found at http://www.math.lsa.umich.edu/~jrs/maple.html .

sage.combinat.sf.kfpoly.**KostkaFoulkesPolynomial**(*mu*, *nu*, *t=None*)

Return the Kostka-Foulkes polynomial $K_{\mu,\nu}(t)$.

INPUT:

- mu, nu – partitions
- t – an optional parameter (default: None)

OUTPUT:

- the Koskta-Foulkes polynomial indexed by partitions mu and nu and evaluated at the parameter t. If t is None the resulting polynomial is in the polynomial ring $\mathbf{Z}['t']$.

EXAMPLES:

```
sage: KostkaFoulkesPolynomial([2,2],[2,2])
1
sage: KostkaFoulkesPolynomial([2,2],[4])
0
sage: KostkaFoulkesPolynomial([2,2],[1,1,1,1])
t^4 + t^2
sage: KostkaFoulkesPolynomial([2,2],[2,1,1])
t
sage: q = PolynomialRing(QQ,'q').gen()
```

(continues on next page)

(continued from previous page)

```
sage: KostkaFoulkesPolynomial([2,2],[2,1,1],q)
q
```

```
>>> from sage.all import *
>>> KostkaFoulkesPolynomial([Integer(2),Integer(2)],[Integer(2),Integer(2)])
1
>>> KostkaFoulkesPolynomial([Integer(2),Integer(2)],[Integer(4)])
0
>>> KostkaFoulkesPolynomial([Integer(2),Integer(2)],[Integer(1),Integer(1),
↪Integer(1),Integer(1)])
t^4 + t^2
>>> KostkaFoulkesPolynomial([Integer(2),Integer(2)],[Integer(2),Integer(1),
↪Integer(1)])
t
>>> q = PolynomialRing(QQ,'q').gen()
>>> KostkaFoulkesPolynomial([Integer(2),Integer(2)],[Integer(2),Integer(1),
↪Integer(1)],q)
q
```

sage.combinat.sf.kfpoly.**compat**(*n*, *mu*, *nu*)

Generate all possible partitions of n that can precede μ, ν in a rigging sequence.

INPUT:

- `n` – positive integer
- `mu`, `nu` – partitions

OUTPUT: list of partitions

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: compat(4, [1], [2,1])
[[1, 1, 1, 1], [2, 1, 1], [2, 2], [3, 1], [4]]
sage: compat(3, [1], [2,1])
[[1, 1, 1], [2, 1], [3]]
sage: compat(2, [1], [])
[[2]]
sage: compat(3, [1], [])
[[2, 1], [3]]
sage: compat(3, [2], [1])
[[3]]
sage: compat(4, [1,1], [])
[[2, 2], [3, 1], [4]]
sage: compat(4, [2], [])
[[4]]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.kfpoly import *
>>> compat(Integer(4), [Integer(1)], [Integer(2),Integer(1)])
[[1, 1, 1, 1], [2, 1, 1], [2, 2], [3, 1], [4]]
>>> compat(Integer(3), [Integer(1)], [Integer(2),Integer(1)])
[[1, 1, 1], [2, 1], [3]]
```

(continues on next page)

(continued from previous page)

```
>>> compat(Integer(2), [Integer(1)], [])
[[2]]
>>> compat(Integer(3), [Integer(1)], [])
[[2, 1], [3]]
>>> compat(Integer(3), [Integer(2)], [Integer(1)])
[[3]]
>>> compat(Integer(4), [Integer(1),Integer(1)], [])
[[2, 2], [3, 1], [4]]
>>> compat(Integer(4), [Integer(2)], [])
[[4]]
```

sage.combinat.sf.kfpoly.**dom**(*mup*, *snu*)

Return `True` if `sum(mu[:i+1]) >= snu[i]` for all `0 <= i < len(snu)`; otherwise, it returns `False`.

INPUT:

- `mup` – a partition conjugate to `mu`
- `snu` – a sequence of positive integers

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: dom([3,2,1],[2,4,5])
True
sage: dom([3,2,1],[2,4,7])
False
sage: dom([3,2,1],[2,6,5])
False
sage: dom([3,2,1],[4,4,4])
False
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.kfpoly import *
>>> dom([Integer(3),Integer(2),Integer(1)],[Integer(2),Integer(4),Integer(5)])
True
>>> dom([Integer(3),Integer(2),Integer(1)],[Integer(2),Integer(4),Integer(7)])
False
>>> dom([Integer(3),Integer(2),Integer(1)],[Integer(2),Integer(6),Integer(5)])
False
>>> dom([Integer(3),Integer(2),Integer(1)],[Integer(4),Integer(4),Integer(4)])
False
```

sage.combinat.sf.kfpoly.**kfpoly**(*mu*, *nu*, *t=None*)

Return the Kostka-Foulkes polynomial $K_{\mu,\nu}(t)$ by generating all rigging sequences for the shape μ, and then selecting those of content ν.

INPUT:

- `mu`, `nu` – partitions
- `t` – an optional parameter (default: `None`)

OUTPUT:

- the Koskta-Foulkes polynomial indexed by partitions `mu` and `nu` and evaluated at the parameter `t`. If `t` is `None` the resulting polynomial is in the polynomial ring $\mathbf{Z}['t']$.

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import kfpoly
sage: kfpoly([2,2], [2,1,1])
t
sage: kfpoly([4], [2,1,1])
t^3
sage: kfpoly([4], [2,2])
t^2
sage: kfpoly([1,1,1,1], [2,2])
0
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.kfpoly import kfpoly
>>> kfpoly([Integer(2),Integer(2)], [Integer(2),Integer(1),Integer(1)])
t
>>> kfpoly([Integer(4)], [Integer(2),Integer(1),Integer(1)])
t^3
>>> kfpoly([Integer(4)], [Integer(2),Integer(2)])
t^2
>>> kfpoly([Integer(1),Integer(1),Integer(1),Integer(1)], [Integer(2),Integer(2)])
0
```

`sage.combinat.sf.kfpoly.`**`riggings`**(*part*)

Generate all possible rigging sequences for a fixed partition `part`.

INPUT:

- `part` – a partition

OUTPUT: list of riggings associated to the partition `part`

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: riggings([3])
[[[1, 1, 1]], [[2, 1]], [[3]]]
sage: riggings([2,1])
[[[2, 1], [1]], [[3], [1]]]
sage: riggings([1,1,1])
[[[3], [2], [1]]]
sage: riggings([2,2])
[[[2, 2], [1, 1]], [[3, 1], [1, 1]], [[4], [1, 1]], [[4], [2]]]
sage: riggings([2,2,2])
[[[3, 3], [2, 2], [1, 1]],
 [[4, 2], [2, 2], [1, 1]],
 [[5, 1], [2, 2], [1, 1]],
 [[6], [2, 2], [1, 1]],
 [[5, 1], [3, 1], [1, 1]],
 [[6], [3, 1], [1, 1]],
 [[6], [4], [2]]]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.kfpoly import *
>>> riggings([Integer(3)])
[[[1, 1, 1]], [[2, 1]], [[3]]]
>>> riggings([Integer(2),Integer(1)])
[[[2, 1], [1]], [[3], [1]]]
>>> riggings([Integer(1),Integer(1),Integer(1)])
[[[3], [2], [1]]]
>>> riggings([Integer(2),Integer(2)])
[[[2, 2], [1, 1]], [[3, 1], [1, 1]], [[4], [1, 1]], [[4], [2]]]
>>> riggings([Integer(2),Integer(2),Integer(2)])
[[[3, 3], [2, 2], [1, 1]],
 [[4, 2], [2, 2], [1, 1]],
 [[5, 1], [2, 2], [1, 1]],
 [[6], [2, 2], [1, 1]],
 [[5, 1], [3, 1], [1, 1]],
 [[6], [3, 1], [1, 1]],
 [[6], [4], [2]]]
```

sage.combinat.sf.kfpoly.**schur_to_hl**(*mu*, *t=None*)

Return a dictionary corresponding to s_μ in Hall-Littlewood P basis.

INPUT:

- mu – a partition
- t – an optional parameter (default: the generator from $\mathbf{Z}['t']$)

OUTPUT:

- a dictionary with the coefficients $K_{\mu\nu}(t)$ for ν smaller in dominance order than μ

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: schur_to_hl([1,1,1])
{[1, 1, 1]: 1}
sage: a = schur_to_hl([2,1])
sage: for mc in sorted(a.items()): print(mc)
([1, 1, 1], t^2 + t)
([2, 1], 1)
sage: a = schur_to_hl([3])
sage: for mc in sorted(a.items()): print(mc)
([1, 1, 1], t^3)
([2, 1], t)
([3], 1)
sage: a = schur_to_hl([4])
sage: for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], t^6)
([2, 1, 1], t^3)
([2, 2], t^2)
([3, 1], t)
([4], 1)
sage: a = schur_to_hl([3,1])
sage: for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], t^5 + t^4 + t^3)
```

(continues on next page)

(continued from previous page)

```
([2, 1, 1], t^2 + t)
([2, 2], t)
([3, 1], 1)
sage: a = schur_to_hl([2,2])
sage: for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], t^4 + t^2)
([2, 1, 1], t)
([2, 2], 1)
sage: a = schur_to_hl([2,1,1])
sage: for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], t^3 + t^2 + t)
([2, 1, 1], 1)
sage: a = schur_to_hl([1,1,1,1])
sage: for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], 1)
sage: a = schur_to_hl([2,2,2])
sage: for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1, 1, 1], t^9 + t^7 + t^6 + t^5 + t^3)
([2, 1, 1, 1, 1], t^4 + t^2)
([2, 2, 1, 1], t)
([2, 2, 2], 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.kfpoly import *
>>> schur_to_hl([Integer(1),Integer(1),Integer(1)])
{[1, 1, 1]: 1}
>>> a = schur_to_hl([Integer(2),Integer(1)])
>>> for mc in sorted(a.items()): print(mc)
([1, 1, 1], t^2 + t)
([2, 1], 1)
>>> a = schur_to_hl([Integer(3)])
>>> for mc in sorted(a.items()): print(mc)
([1, 1, 1], t^3)
([2, 1], t)
([3], 1)
>>> a = schur_to_hl([Integer(4)])
>>> for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], t^6)
([2, 1, 1], t^3)
([2, 2], t^2)
([3, 1], t)
([4], 1)
>>> a = schur_to_hl([Integer(3),Integer(1)])
>>> for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], t^5 + t^4 + t^3)
([2, 1, 1], t^2 + t)
([2, 2], t)
([3, 1], 1)
>>> a = schur_to_hl([Integer(2),Integer(2)])
>>> for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], t^4 + t^2)
([2, 1, 1], t)
```

(continues on next page)

(continued from previous page)

```
([2, 2], 1)
>>> a = schur_to_hl([Integer(2),Integer(1),Integer(1)])
>>> for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], t^3 + t^2 + t)
([2, 1, 1], 1)
>>> a = schur_to_hl([Integer(1),Integer(1),Integer(1),Integer(1)])
>>> for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1], 1)
>>> a = schur_to_hl([Integer(2),Integer(2),Integer(2)])
>>> for mc in sorted(a.items()): print(mc)
([1, 1, 1, 1, 1, 1], t^9 + t^7 + t^6 + t^5 + t^3)
([2, 1, 1, 1, 1], t^4 + t^2)
([2, 2, 1, 1], t)
([2, 2, 2], 1)
```

sage.combinat.sf.kfpoly.**weight**(*rg*, *t=None*)

Return the weight of a rigging.

INPUT:

- `rg` – a rigging, a list of partitions
- `t` – an optional parameter, (default: the generator from $\mathbf{Z}['t']$)

OUTPUT: a polynomial in the parameter t

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import weight
sage: weight([[2,1], [1]])
1
sage: weight([[3], [1]])
t^2 + t
sage: weight([[2,1], [3]])
t^4
sage: weight([[2, 2], [1, 1]])
1
sage: weight([[3, 1], [1, 1]])
t
sage: weight([[4], [1, 1]], 2)
16
sage: weight([[4], [2]], t=2)
4
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.kfpoly import weight
>>> weight([[Integer(2),Integer(1)], [Integer(1)]])
1
>>> weight([[Integer(3)], [Integer(1)]])
t^2 + t
>>> weight([[Integer(2),Integer(1)], [Integer(3)]])
t^4
>>> weight([[Integer(2), Integer(2)], [Integer(1), Integer(1)]])
1
```

(continues on next page)

(continued from previous page)

```
>>> weight([[Integer(3), Integer(1)], [Integer(1), Integer(1)]])
t
>>> weight([[Integer(4)], [Integer(1), Integer(1)]], Integer(2))
16
>>> weight([[Integer(4)], [Integer(2)]], t=Integer(2))
4
```

5.1.291 LLT symmetric functions

REFERENCES:

class sage.combinat.sf.llt.**LLT_class**(*Sym*, *k*, *t*)

Bases: UniqueRepresentation

A class for working with LLT symmetric functions.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: L3 = Sym.llt(3); L3
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field
sage: L3.cospin([3,2,1])
(t+1)*m[1, 1] + m[2]
sage: HC3 = L3.hcospin(); HC3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over␣
↪Rational Field in the level 3 LLT cospin basis
sage: m = Sym.monomial()
sage: m( HC3[1,1] )
(t+1)*m[1, 1] + m[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> L3 = Sym.llt(Integer(3)); L3
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field
>>> L3.cospin([Integer(3),Integer(2),Integer(1)])
(t+1)*m[1, 1] + m[2]
>>> HC3 = L3.hcospin(); HC3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over␣
↪Rational Field in the level 3 LLT cospin basis
>>> m = Sym.monomial()
>>> m( HC3[Integer(1),Integer(1)] )
(t+1)*m[1, 1] + m[2]
```

We require that the parameter t must be in the base ring:

```
sage: Symxt = SymmetricFunctions(QQ['x','t'].fraction_field())
sage: (x,t) = Symxt.base_ring().gens()
sage: LLT3x = Symxt.llt(3,t=x)
sage: LLT3 = Symxt.llt(3)
sage: HS3x = LLT3x.hspin()
sage: HS3t = LLT3.hspin()
```

(continues on next page)

(continued from previous page)

```
sage: s = Symxt.schur()
sage: s(HS3x[2,1])
s[2, 1] + x*s[3]
sage: s(HS3t[2,1])
s[2, 1] + t*s[3]
sage: HS3x(HS3t[2,1])
HSp3[2, 1] - (x-t)*HSp3[3]
sage: s(HS3x(HS3t[2,1]))
s[2, 1] + t*s[3]
sage: LLT3t2 = Symxt.llt(3,t=2)
sage: HC3t2 = LLT3t2.hcospin()
sage: HS3x(HC3t2[3,1])
2*HSp3[3, 1] - (2*x-1)*HSp3[4]
```

```
>>> from sage.all import *
>>> Symxt = SymmetricFunctions(QQ['x','t'].fraction_field())
>>> (x,t) = Symxt.base_ring().gens()
>>> LLT3x = Symxt.llt(Integer(3),t=x)
>>> LLT3 = Symxt.llt(Integer(3))
>>> HS3x = LLT3x.hspin()
>>> HS3t = LLT3.hspin()
>>> s = Symxt.schur()
>>> s(HS3x[Integer(2),Integer(1)])
s[2, 1] + x*s[3]
>>> s(HS3t[Integer(2),Integer(1)])
s[2, 1] + t*s[3]
>>> HS3x(HS3t[Integer(2),Integer(1)])
HSp3[2, 1] - (x-t)*HSp3[3]
>>> s(HS3x(HS3t[Integer(2),Integer(1)]))
s[2, 1] + t*s[3]
>>> LLT3t2 = Symxt.llt(Integer(3),t=Integer(2))
>>> HC3t2 = LLT3t2.hcospin()
>>> HS3x(HC3t2[Integer(3),Integer(1)])
2*HSp3[3, 1] - (2*x-1)*HSp3[4]
```

base_ring()

Return the base ring of `self`.

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT: the base ring of the symmetric function ring associated to `self`

EXAMPLES:

```
sage: SymmetricFunctions(FractionField(QQ['t'])).llt(3).base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3)).base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

cospin(*skp*)

Calculate a single instance of the cospin symmetric functions.

These are the functions defined in [LLT1997] equation (26).

INPUT:

- `self` – a family of LLT symmetric functions bases
- `skp` – a partition or a list of partitions or a list of skew partitions

OUTPUT:

the monomial expansion of the LLT symmetric function cospin functions indexed by `skp`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: L3 = Sym.llt(3)
sage: L3.cospin([2,1])
m[1]
sage: L3.cospin([3,2,1])
(t+1)*m[1, 1] + m[2]
sage: s = Sym.schur()
sage: s(L3.cospin([[2],[1],[2]]))
t^4*s[2, 2, 1] + t^3*s[3, 1, 1] + (t^3+t^2)*s[3, 2] + (t^2+t)*s[4, 1] + s[5]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> L3 = Sym.llt(Integer(3))
>>> L3.cospin([Integer(2),Integer(1)])
m[1]
>>> L3.cospin([Integer(3),Integer(2),Integer(1)])
(t+1)*m[1, 1] + m[2]
>>> s = Sym.schur()
>>> s(L3.cospin([[Integer(2)],[Integer(1)],[Integer(2)]]))
t^4*s[2, 2, 1] + t^3*s[3, 1, 1] + (t^3+t^2)*s[3, 2] + (t^2+t)*s[4, 1] + s[5]
```

hcospin()

Return the HCospin basis. This basis is defined [LLT1997] equation (27).

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT: the h-cospin basis of the LLT symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HCosp3 = Sym.llt(3).hcospin(); HCosp3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT cospin basis
sage: HCosp3([1])^2
1/t*HCosp3[1, 1] + ((t-1)/t)*HCosp3[2]

sage: s = Sym.schur()
sage: HCosp3(s([2]))
HCosp3[2]
```

(continues on next page)

(continued from previous page)

```
sage: HCosp3(s([1,1]))
1/t*HCosp3[1, 1] - 1/t*HCosp3[2]
sage: s(HCosp3([2,1]))
t*s[2, 1] + s[3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HCosp3 = Sym.llt(Integer(3)).hcospin(); HCosp3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT cospin basis
>>> HCosp3([Integer(1)])**Integer(2)
1/t*HCosp3[1, 1] + ((t-1)/t)*HCosp3[2]

>>> s = Sym.schur()
>>> HCosp3(s([Integer(2)]))
HCosp3[2]
>>> HCosp3(s([Integer(1),Integer(1)]))
1/t*HCosp3[1, 1] - 1/t*HCosp3[2]
>>> s(HCosp3([Integer(2),Integer(1)]))
t*s[2, 1] + s[3]
```

hspin()

Return the HSpin basis.

This basis is defined [LLT1997] equation (28).

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT: the h-spin basis of the LLT symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HSp3 = Sym.llt(3).hspin(); HSp3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT spin basis
sage: HSp3([1])^2
HSp3[1, 1] + (-t+1)*HSp3[2]

sage: s = Sym.schur()
sage: HSp3(s([2]))
HSp3[2]
sage: HSp3(s([1,1]))
HSp3[1, 1] - t*HSp3[2]
sage: s(HSp3([2,1]))
s[2, 1] + t*s[3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HSp3 = Sym.llt(Integer(3)).hspin(); HSp3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT spin basis
```

(continues on next page)

(continued from previous page)

```
>>> HSp3([Integer(1)])**Integer(2)
HSp3[1, 1] + (-t+1)*HSp3[2]

>>> s = Sym.schur()
>>> HSp3(s([Integer(2)]))
HSp3[2]
>>> HSp3(s([Integer(1),Integer(1)]))
HSp3[1, 1] - t*HSp3[2]
>>> s(HSp3([Integer(2),Integer(1)]))
s[2, 1] + t*s[3]
```

level()

Return the level of `self`.

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT: the level is the parameter of k in the basis

EXAMPLES:

```
sage: SymmetricFunctions(FractionField(QQ['t'])).llt(3).level()
3
```

```
>>> from sage.all import *
>>> SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3)).level()
3
```

spin_square(*skp*)

Calculate a single instance of a spin squared LLT symmetric function associated with a partition, list of partitions, or a list of skew partitions.

This family of symmetric functions is defined in [LT2000] equation (43).

INPUT:

- `self` – a family of LLT symmetric functions bases
- `skp` – a partition of a list of partitions or a list of skew partitions

OUTPUT:

the monomial expansion of the LLT symmetric function spin-square functions indexed by `skp`

EXAMPLES:

```
sage: L3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3)
sage: L3.spin_square([2,1])
t*m[1]
sage: L3.spin_square([3,2,1])
(t^3+t)*m[1, 1] + t^3*m[2]
sage: L3.spin_square([[1],[1],[1]])
(t^6+2*t^4+2*t^2+1)*m[1, 1, 1] + (t^6+t^4+t^2)*m[2, 1] + t^6*m[3]
sage: L3.spin_square([[[2,2],[1]],[[2,1],[]]])
(2*t^4+3*t^2+1)*m[1, 1, 1, 1] + (t^4+t^2)*m[2, 1, 1] + t^4*m[2, 2]
```

```
>>> from sage.all import *
>>> L3 = SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3))
>>> L3.spin_square([Integer(2),Integer(1)])
t*m[1]
>>> L3.spin_square([Integer(3),Integer(2),Integer(1)])
(t^3+t)*m[1, 1] + t^3*m[2]
>>> L3.spin_square([[Integer(1)],[Integer(1)],[Integer(1)]])
(t^6+2*t^4+2*t^2+1)*m[1, 1, 1] + (t^6+t^4+t^2)*m[2, 1] + t^6*m[3]
>>> L3.spin_square([[[Integer(2),Integer(2)],[Integer(1)]],[[Integer(2),
↪Integer(1)],[]]])
(2*t^4+3*t^2+1)*m[1, 1, 1, 1] + (t^4+t^2)*m[2, 1, 1] + t^4*m[2, 2]
```

symmetric_function_ring()

The symmetric function algebra associated to the family of LLT symmetric function bases

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT: the symmetric function ring associated to `self`

EXAMPLES:

```
sage: L3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3)
sage: L3.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field
```

```
>>> from sage.all import *
>>> L3 = SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3))
>>> L3.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field
```

class sage.combinat.sf.llt.**LLT_cospin**(*llt*)

Bases: *LLT_generic*

A class of methods for the h-cospin LLT basis of the symmetric functions.

INPUT:

- `self` – an instance of the LLT hcospin basis
- `llt` – a family of LLT symmetric function bases

class Element

Bases: *Element*

class sage.combinat.sf.llt.**LLT_generic**(*llt*, *prefix*)

Bases: *SymmetricFunctionAlgebra_generic*

A class of methods which are common to both the hspin and hcospin of the LLT symmetric functions.

INPUT:

- `self` – an instance of the LLT hspin or hcospin basis
- `llt` – a family of LLT symmetric functions

EXAMPLES:

```
sage: SymmetricFunctions(FractionField(QQ['t'])).llt(3).hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over
↪Rational Field in the level 3 LLT spin basis
sage: SymmetricFunctions(QQ).llt(3,t=2).hspin()
Symmetric Functions over Rational Field in the level 3 LLT spin with t=2 basis
sage: QQz = FractionField(QQ['z']); z = QQz.gen()
sage: SymmetricFunctions(QQz).llt(3,t=z).hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in z over
↪Rational Field in the level 3 LLT spin with t=z basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3)).hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over
↪Rational Field in the level 3 LLT spin basis
>>> SymmetricFunctions(QQ).llt(Integer(3),t=Integer(2)).hspin()
Symmetric Functions over Rational Field in the level 3 LLT spin with t=2 basis
>>> QQz = FractionField(QQ['z']); z = QQz.gen()
>>> SymmetricFunctions(QQz).llt(Integer(3),t=z).hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in z over
↪Rational Field in the level 3 LLT spin with t=z basis
```

class Element

Bases: *SymmetricFunctionAlgebra_generic_Element*

construction()

Return a pair (F, R), where F is a SymmetricFunctionsFunctor and R is a ring, such that F(R) returns self.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HSp3 = Sym.llt(3).hspin()
sage: HSp3.construction()
(SymmetricFunctionsFunctor[level 3 LLT spin],
 Fraction Field of Univariate Polynomial Ring in t over Rational Field)
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HSp3 = Sym.llt(Integer(3)).hspin()
>>> HSp3.construction()
(SymmetricFunctionsFunctor[level 3 LLT spin],
 Fraction Field of Univariate Polynomial Ring in t over Rational Field)
```

level()

Return the level of self.

INPUT:

- self – an instance of the LLT hspin or hcospin basis

OUTPUT: the level associated to the basis self

EXAMPLES:

```
sage: HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3).hspin()
sage: HSp3.level()
3
```

```
>>> from sage.all import *
>>> HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3)).hspin()
>>> HSp3.level()
3
```

llt_family()

The family of the llt bases of the symmetric functions.

INPUT:

- `self` – an instance of the LLT hspin or hcospin basis

OUTPUT: an instance of the family of LLT bases associated to `self`

EXAMPLES:

```
sage: HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3).hspin()
sage: HSp3.llt_family()
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in␣
↪t over Rational Field
```

```
>>> from sage.all import *
>>> HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3)).hspin()
>>> HSp3.llt_family()
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in␣
↪t over Rational Field
```

product(*left*, *right*)

Convert to the monomial basis, do the multiplication there, and convert back to the basis `self`.

INPUT:

- `self` – an instance of the LLT hspin or hcospin basis
- `left`, `right` – elements of the symmetric functions

OUTPUT: the product of `left` and `right` expanded in the basis `self`

EXAMPLES:

```
sage: HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3).hspin()
sage: HSp3.product(HSp3([1]), HSp3([2]))
HSp3[2, 1] + (-t+1)*HSp3[3]
sage: HCosp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3).hcospin()
sage: HCosp3.product(HCosp3([1]), HSp3([2]))
1/t*HCosp3[2, 1] + ((t-1)/t)*HCosp3[3]
```

```
>>> from sage.all import *
>>> HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3)).hspin()
>>> HSp3.product(HSp3([Integer(1)]), HSp3([Integer(2)]))
HSp3[2, 1] + (-t+1)*HSp3[3]
>>> HCosp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(Integer(3)).
```

(continues on next page)

(continued from previous page)

```
↪hcospin()
>>> HCosp3.product(HCosp3([Integer(1)]), HSp3([Integer(2)]))
1/t*HCosp3[2, 1] + ((t-1)/t)*HCosp3[3]
```

class sage.combinat.sf.llt.**LLT_spin**(*llt*)

Bases: *LLT_generic*

A class of methods for the h-spin LLT basis of the symmetric functions.

INPUT:

- `self` – an instance of the LLT hcospin basis
- `llt` – a family of LLT symmetric function bases

class Element

Bases: *Element*

5.1.292 Macdonald Polynomials

Notation used in the definitions follows mainly [Mac1995].

The integral forms of the bases H and Ht do not appear in Macdonald's book. They correspond to the two bases $H_\mu[X;q,t] = \sum_\nu K_{\nu\mu}(q,t)s_\mu[X]$ and $\tilde{H}_\mu[X;q,t] = t^{n(\mu)}\sum_\nu K_{\nu\mu}(q,1/t)s_\nu[X]$ where $K_{\mu\nu}(q,t)$ are the Macdonald q,t-Koskta coefficients.

The Ht in this case is short for $\tilde{H}$ and is the basis which is the graded Frobenius image of the Garsia-Haiman modules [GH1993].

REFERENCES:

- [Mac1995]

class sage.combinat.sf.macdonald.**Macdonald**(*Sym*, *q*, *t*)

Bases: `UniqueRepresentation`

Macdonald Symmetric functions including P, Q, J, H, Ht bases also including the S basis which is the plethystic transformation of the Schur basis (that which is dual to the Schur basis with respect to the Macdonald q,t-scalar product)

INPUT:

- `self` – a family of Macdonald symmetric function bases

EXAMPLES:

```
sage: t = QQ['t'].gen(); SymmetricFunctions(QQ['t'].fraction_field()).
↪macdonald(q=t,t=1)
Macdonald polynomials with q=t and t=1 over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field
sage: Sym = SymmetricFunctions(FractionField(QQ['t'])).macdonald()
Traceback (most recent call last):
...
TypeError: unable to evaluate 'q' in Fraction Field of Univariate Polynomial Ring␣
↪in t over Rational Field
```

```
>>> from sage.all import *
>>> t = QQ['t'].gen(); SymmetricFunctions(QQ['t'].fraction_field()).macdonald(q=t,
```

(continues on next page)

(continued from previous page)

```
↪t=Integer(1))
Macdonald polynomials with q=t and t=1 over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field
>>> Sym = SymmetricFunctions(FractionField(QQ['t'])).macdonald()
Traceback (most recent call last):
...
TypeError: unable to evaluate 'q' in Fraction Field of Univariate Polynomial Ring␣
↪in t over Rational Field
```

H()

Return the Macdonald polynomials on the H basis. When the H basis is expanded on the Schur basis, the coefficients are the qt-Kostka numbers.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT: the H Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: H = Sym.macdonald().H(); H
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald H basis
sage: s = Sym.schur()
sage: s(H([2]))
q*s[1, 1] + s[2]
sage: s(H([1,1]))
s[1, 1] + t*s[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> H = Sym.macdonald().H(); H
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald H basis
>>> s = Sym.schur()
>>> s(H([Integer(2)]))
q*s[1, 1] + s[2]
>>> s(H([Integer(1),Integer(1)]))
s[1, 1] + t*s[2]
```

Coercions to/from the Schur basis are implemented:

```
sage: H = Sym.macdonald().H()
sage: s = Sym.schur()
sage: H(s([2]))
(q/(q*t-1))*McdH[1, 1] - (1/(q*t-1))*McdH[2]
```

```
>>> from sage.all import *
>>> H = Sym.macdonald().H()
>>> s = Sym.schur()
>>> H(s([Integer(2)]))
(q/(q*t-1))*McdH[1, 1] - (1/(q*t-1))*McdH[2]
```

Ht()

Return the Macdonald polynomials on the Ht basis. The elements of the Ht basis are eigenvectors of the $nabla$ operator. When expanded on the Schur basis, the coefficients are the modified qt-Kostka numbers.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT: the Ht Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Ht = Sym.macdonald().Ht(); Ht
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald Ht basis
sage: [Ht(p).nabla() for p in Partitions(3)]
[q^3*McdHt[3], q*t*McdHt[2, 1], t^3*McdHt[1, 1, 1]]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> Ht = Sym.macdonald().Ht(); Ht
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald Ht basis
>>> [Ht(p).nabla() for p in Partitions(Integer(3))]
[q^3*McdHt[3], q*t*McdHt[2, 1], t^3*McdHt[1, 1, 1]]
```

```
sage: s = Sym.schur()
sage: from sage.combinat.sf.macdonald import qt_kostka
sage: q,t = Ht.base_ring().gens()
sage: s(Ht([2,1]))
q*t*s[1, 1, 1] + (q+t)*s[2, 1] + s[3]
sage: qt_kostka([1,1,1],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
q*t
sage: qt_kostka([2,1],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
q + t
sage: qt_kostka([3],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
1
```

```
>>> from sage.all import *
>>> s = Sym.schur()
>>> from sage.combinat.sf.macdonald import qt_kostka
>>> q,t = Ht.base_ring().gens()
>>> s(Ht([Integer(2),Integer(1)]))
q*t*s[1, 1, 1] + (q+t)*s[2, 1] + s[3]
>>> qt_kostka([Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(1)]).
↪subs(t=Integer(1)/t)*t**Partition([Integer(2),Integer(1)]).weighted_size()
q*t
>>> qt_kostka([Integer(2),Integer(1)],[Integer(2),Integer(1)]).
↪subs(t=Integer(1)/t)*t**Partition([Integer(2),Integer(1)]).weighted_size()
q + t
>>> qt_kostka([Integer(3)],[Integer(2),Integer(1)]).subs(t=Integer(1)/
↪t)*t**Partition([Integer(2),Integer(1)]).weighted_size()
1
```

Coercions to/from the Schur basis are implemented:

```
sage: Ht = Sym.macdonald().Ht()
sage: s = Sym.schur()
sage: Ht(s([2,1]))
(q/(q*t^2-t^3-q^2+q*t))*McdHt[1, 1, 1] - ((q^2+q*t+t^2)/(q^2*t^2-q^3-t^
↪3+q*t))*McdHt[2, 1]
+ (t/(-q^3+q^2*t+q*t-t^2))*McdHt[3]
sage: Ht(s([2]))
-(q/(-q+t))*McdHt[1, 1] + (t/(-q+t))*McdHt[2]
```

```
>>> from sage.all import *
>>> Ht = Sym.macdonald().Ht()
>>> s = Sym.schur()
>>> Ht(s([Integer(2),Integer(1)]))
(q/(q*t^2-t^3-q^2+q*t))*McdHt[1, 1, 1] - ((q^2+q*t+t^2)/(q^2*t^2-q^3-t^
↪3+q*t))*McdHt[2, 1]
+ (t/(-q^3+q^2*t+q*t-t^2))*McdHt[3]
>>> Ht(s([Integer(2)]))
-(q/(-q+t))*McdHt[1, 1] + (t/(-q+t))*McdHt[2]
```

J()

Return the Macdonald polynomials on the J basis also known as the integral form of the Macdonald polynomials. These are scalar multiples of both the P and Q bases. When expressed in the P or Q basis, the scaling coefficients are polynomials in q and t rather than rational functions.

The J basis is calculated using determinantal formulas of Lapointe-Lascoux-Morse giving the action on the S-basis [LLM1998].

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT: the J Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: J = Sym.macdonald().J(); J
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald J basis
sage: P = Sym.macdonald().P()
sage: Q = Sym.macdonald().Q()
sage: P(J([2]))
(q*t^2-q*t-t+1)*McdP[2]
sage: P(J([1,1]))
(t^3-t^2-t+1)*McdP[1, 1]
sage: Q(J([2]))
(q^3-q^2-q+1)*McdQ[2]
sage: Q(J([1,1]))
(q^2*t-q*t-q+1)*McdQ[1, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> J = Sym.macdonald().J(); J
```

(continues on next page)

(continued from previous page)

```
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald J basis
>>> P = Sym.macdonald().P()
>>> Q = Sym.macdonald().Q()
>>> P(J([Integer(2)]))
(q*t^2-q*t-t+1)*McdP[2]
>>> P(J([Integer(1),Integer(1)]))
(t^3-t^2-t+1)*McdP[1, 1]
>>> Q(J([Integer(2)]))
(q^3-q^2-q+1)*McdQ[2]
>>> Q(J([Integer(1),Integer(1)]))
(q^2*t-q*t-q+1)*McdQ[1, 1]
```

Coercions from the Q and J basis (proportional) and to/from the Schur basis are implemented:

```
sage: P = Sym.macdonald().P()
sage: Q = Sym.macdonald().Q()
sage: J = Sym.macdonald().J()
sage: s = Sym.schur()
```

```
>>> from sage.all import *
>>> P = Sym.macdonald().P()
>>> Q = Sym.macdonald().Q()
>>> J = Sym.macdonald().J()
>>> s = Sym.schur()
```

```
sage: J(P([2]))
(1/(q*t^2-q*t-t+1))*McdJ[2]
```

```
>>> from sage.all import *
>>> J(P([Integer(2)]))
(1/(q*t^2-q*t-t+1))*McdJ[2]
```

```
sage: J(Q([2]))
(1/(q^3-q^2-q+1))*McdJ[2]
```

```
>>> from sage.all import *
>>> J(Q([Integer(2)]))
(1/(q^3-q^2-q+1))*McdJ[2]
```

```
sage: s(J([2]))
-(q*t-t^2-q+t)*s[1, 1] + (q*t^2-q*t-t+1)*s[2]
sage: J(s([2]))
((q-t)/(q*t^4-q*t^3-q*t^2-t^3+q*t+t^2+t-1))*McdJ[1, 1] + (1/(q*t^2-q*t-
↪t+1))*McdJ[2]
```

```
>>> from sage.all import *
>>> s(J([Integer(2)]))
-(q*t-t^2-q+t)*s[1, 1] + (q*t^2-q*t-t+1)*s[2]
>>> J(s([Integer(2)]))
```

(continues on next page)

(continued from previous page)

```
((q-t)/(q*t^4-q*t^3-q*t^2-t^3+q*t+t^2+t-1))*McdJ[1, 1] + (1/(q*t^2-q*t-
↪t+1))*McdJ[2]
```

P()

Return Macdonald polynomials in P basis. The P basis is defined here as a normalized form of the J basis.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT: the P Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald P basis
sage: P[2]
McdP[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald P basis
>>> P[Integer(2)]
McdP[2]
```

The P Macdonald basis is upper triangularly related to the monomial symmetric functions and are orthogonal with respect to the qt-Hall scalar product:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald P basis
sage: m = Sym.monomial()
sage: P.transition_matrix(m,2)
[                          1 (q*t - q + t - 1)/(q*t - 1)]
[                          0                           1]
sage: P([1,1]).scalar_qt(P([2]))
0
sage: P([2]).scalar_qt(P([2]))
(-q^3 + q^2 + q - 1)/(-q*t^2 + q*t + t - 1)
sage: P([1,1]).scalar_qt(P([1,1]))
(-q^2*t + q*t + q - 1)/(-t^3 + t^2 + t - 1)
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald P basis
>>> m = Sym.monomial()
>>> P.transition_matrix(m,Integer(2))
```

(continues on next page)

(continued from previous page)

```
[                          1  (q*t - q + t - 1)/(q*t - 1)]
[                          0                            1]
>>> P([Integer(1),Integer(1)]).scalar_qt(P([Integer(2)]))
0
>>> P([Integer(2)]).scalar_qt(P([Integer(2)]))
(-q^3 + q^2 + q - 1)/(-q*t^2 + q*t + t - 1)
>>> P([Integer(1),Integer(1)]).scalar_qt(P([Integer(1),Integer(1)]))
(-q^2*t + q*t + q - 1)/(-t^3 + t^2 + t - 1)
```

When $q = 0$, the Macdonald polynomials on the P basis are the same as the Hall-Littlewood polynomials on the P basis.

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: P = Sym.macdonald(q=0).P(); P
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Macdonald P with q=0 basis
sage: P([2])^2
(t+1)*McdP[2, 2] + (-t+1)*McdP[3, 1] + McdP[4]
sage: HLP = Sym.hall_littlewood().P()
sage: HLP([2])^2
(t+1)*HLP[2, 2] + (-t+1)*HLP[3, 1] + HLP[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> P = Sym.macdonald(q=Integer(0)).P(); P
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Macdonald P with q=0 basis
>>> P([Integer(2)])**Integer(2)
(t+1)*McdP[2, 2] + (-t+1)*McdP[3, 1] + McdP[4]
>>> HLP = Sym.hall_littlewood().P()
>>> HLP([Integer(2)])**Integer(2)
(t+1)*HLP[2, 2] + (-t+1)*HLP[3, 1] + HLP[4]
```

Coercions from the Q and J basis (proportional) are implemented:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: Q = Sym.macdonald().Q()
sage: J = Sym.macdonald().J()
sage: s = Sym.schur()
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> P = Sym.macdonald().P()
>>> Q = Sym.macdonald().Q()
>>> J = Sym.macdonald().J()
>>> s = Sym.schur()
```

```
sage: P(Q([2]))
((q*t^2-q*t-t+1)/(q^3-q^2-q+1))*McdP[2]
sage: P(Q([2,1]))
-((q*t^4-2*q*t^3+q*t^2-t^2+2*t-1)/(-q^4*t+2*q^3*t-q^2*t+q^2-2*q+1))*McdP[2, 1]
```

```
>>> from sage.all import *
>>> P(Q([Integer(2)]))
((q*t^2-q*t-t+1)/(q^3-q^2-q+1))*McdP[2]
>>> P(Q([Integer(2),Integer(1)]))
-((q*t^4-2*q*t^3+q*t^2-t^2+2*t-1)/(-q^4*t+2*q^3*t-q^2*t+q^2-2*q+1))*McdP[2, 1]
```

```
sage: P(J([2]))
(q*t^2-q*t-t+1)*McdP[2]
sage: P(J([2,1]))
-(q*t^4-2*q*t^3+q*t^2-t^2+2*t-1)*McdP[2, 1]
```

```
>>> from sage.all import *
>>> P(J([Integer(2)]))
(q*t^2-q*t-t+1)*McdP[2]
>>> P(J([Integer(2),Integer(1)]))
-(q*t^4-2*q*t^3+q*t^2-t^2+2*t-1)*McdP[2, 1]
```

By transitivity, one get coercions from the classical bases:

```
sage: P(s([2]))
((q-t)/(q*t-1))*McdP[1, 1] + McdP[2]
sage: P(s([2,1]))
((q*t-t^2+q-t)/(q*t^2-1))*McdP[1, 1, 1] + McdP[2, 1]
```

```
>>> from sage.all import *
>>> P(s([Integer(2)]))
((q-t)/(q*t-1))*McdP[1, 1] + McdP[2]
>>> P(s([Integer(2),Integer(1)]))
((q*t-t^2+q-t)/(q*t^2-1))*McdP[1, 1, 1] + McdP[2, 1]
```

```
sage: Sym = SymmetricFunctions(QQ['x','y','z'].fraction_field())
sage: (x,y,z) = Sym.base_ring().gens()
sage: Macxy = Sym.macdonald(q=x,t=y)
sage: Macyz = Sym.macdonald(q=y,t=z)
sage: Maczx = Sym.macdonald(q=z,t=x)
sage: P1 = Macxy.P()
sage: P2 = Macyz.P()
sage: P3 = Maczx.P()
sage: m(P1[2,1])
-((2*x*y^2-x*y+y^2-x+y-2)/(-x*y^2+1))*m[1, 1, 1] + m[2, 1]
sage: m(P2[2,1])
-((2*y*z^2-y*z+z^2-y+z-2)/(-y*z^2+1))*m[1, 1, 1] + m[2, 1]
sage: m(P1(P2(P3[2,1])))
-((2*x^2*z+x^2-x*z+x-z-2)/(-x^2*z+1))*m[1, 1, 1] + m[2, 1]
sage: P1(P2[2])
-((x*y^2-2*x*y*z+y^2*z+x-2*y+z)/(x*y^2*z-x*y-y*z+1))*McdP[1, 1] + McdP[2]
sage: m(z*P1[2]+x*P2[2])
((x^2*y^2*z+x*y^2*z^2-x^2*y^2+x^2*y*z-x*y*z^2+y^2*z^2-x^2*y-2*x*y*z-y*z^2+x*y-
↪y*z+x+z)/(x*y^2*z-x*y-y*z+1))*m[1, 1] + (x+z)*m[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['x','y','z'].fraction_field())
```

(continues on next page)

(continued from previous page)

```
>>> (x,y,z) = Sym.base_ring().gens()
>>> Macxy = Sym.macdonald(q=x,t=y)
>>> Macyz = Sym.macdonald(q=y,t=z)
>>> Maczx = Sym.macdonald(q=z,t=x)
>>> P1 = Macxy.P()
>>> P2 = Macyz.P()
>>> P3 = Maczx.P()
>>> m(P1[Integer(2),Integer(1)])
-((2*x*y^2-x*y+y^2-x+y-2)/(-x*y^2+1))*m[1, 1, 1] + m[2, 1]
>>> m(P2[Integer(2),Integer(1)])
-((2*y*z^2-y*z+z^2-y+z-2)/(-y*z^2+1))*m[1, 1, 1] + m[2, 1]
>>> m(P1(P2(P3[Integer(2),Integer(1)])))
-((2*x^2*z+x^2-x*z+x-z-2)/(-x^2*z+1))*m[1, 1, 1] + m[2, 1]
>>> P1(P2[Integer(2)])
-((x*y^2-2*x*y*z+y^2*z+x-2*y+z)/(x*y^2*z-x*y-y*z+1))*McdP[1, 1] + McdP[2]
>>> m(z*P1[Integer(2)]+x*P2[Integer(2)])
((x^2*y^2*z+x*y^2*z^2-x^2*y^2+x^2*y*z-x*y*z^2+y^2*z^2-x^2*y-2*x*y*z-y*z^2+x*y-
↪y*z+x+z)/(x*y^2*z-x*y-y*z+1))*m[1, 1] + (x+z)*m[2]
```

Q()

Return the Macdonald polynomials on the Q basis. These are dual to the Macdonald polynomials on the P basis with respect to the qt-Hall scalar product. The Q basis is defined to be a normalized form of the J basis.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT: the Q Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Q = Sym.macdonald().Q(); Q
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald Q basis
sage: P = Sym.macdonald().P()
sage: Q([2]).scalar_qt(P([2]))
1
sage: Q([2]).scalar_qt(P([1,1]))
0
sage: Q([1,1]).scalar_qt(P([2]))
0
sage: Q([1,1]).scalar_qt(P([1,1]))
1
sage: Q(P([2]))
((q^3-q^2-q+1)/(q*t^2-q*t-t+1))*McdQ[2]
sage: Q(P([1,1]))
((q^2*t-q*t-q+1)/(t^3-t^2-t+1))*McdQ[1, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> Q = Sym.macdonald().Q(); Q
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald Q basis
```

(continues on next page)

(continued from previous page)

```
>>> P = Sym.macdonald().P()
>>> Q([Integer(2)]).scalar_qt(P([Integer(2)]))
1
>>> Q([Integer(2)]).scalar_qt(P([Integer(1),Integer(1)]))
0
>>> Q([Integer(1),Integer(1)]).scalar_qt(P([Integer(2)]))
0
>>> Q([Integer(1),Integer(1)]).scalar_qt(P([Integer(1),Integer(1)]))
1
>>> Q(P([Integer(2)]))
((q^3-q^2-q+1)/(q*t^2-q*t-t+1))*McdQ[2]
>>> Q(P([Integer(1),Integer(1)]))
((q^2*t-q*t-q+1)/(t^3-t^2-t+1))*McdQ[1, 1]
```

Coercions from the P and J basis (proportional) are implemented:

```
sage: P = Sym.macdonald().P()
sage: Q = Sym.macdonald().Q()
sage: J = Sym.macdonald().J()
sage: s = Sym.schur()
```

```
>>> from sage.all import *
>>> P = Sym.macdonald().P()
>>> Q = Sym.macdonald().Q()
>>> J = Sym.macdonald().J()
>>> s = Sym.schur()
```

```
sage: Q(J([2]))
(q^3-q^2-q+1)*McdQ[2]
```

```
>>> from sage.all import *
>>> Q(J([Integer(2)]))
(q^3-q^2-q+1)*McdQ[2]
```

```
sage: Q(P([2]))
((q^3-q^2-q+1)/(q*t^2-q*t-t+1))*McdQ[2]
sage: P(Q(P([2])))
McdP[2]
sage: Q(P(Q([2])))
McdQ[2]
```

```
>>> from sage.all import *
>>> Q(P([Integer(2)]))
((q^3-q^2-q+1)/(q*t^2-q*t-t+1))*McdQ[2]
>>> P(Q(P([Integer(2)])))
McdP[2]
>>> Q(P(Q([Integer(2)])))
McdQ[2]
```

By transitivity, one gets coercions from the classical bases:

```
sage: Q(s([2]))
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdQ[1, 1] + ((q^3-q^2-q+1)/(q*t^2-q*t-
↪t+1))*McdQ[2]
```

```
>>> from sage.all import *
>>> Q(s([Integer(2)]))
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdQ[1, 1] + ((q^3-q^2-q+1)/(q*t^2-q*t-
↪t+1))*McdQ[2]
```

S()

Return the modified Schur functions defined by the plethystic substitution $S_\mu = s_\mu[X(1-t)/(1-q)]$. When the Macdonald polynomials in the J basis are expressed in terms of the modified Schur functions at $q = 0$, the coefficients are qt-Kostka numbers.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT: the S Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: S = Sym.macdonald().S(); S
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t
over Rational Field in the Macdonald S basis
sage: p = Sym.power()
sage: p(S[2,1])
((1/3*t^3-t^2+t-1/3)/(q^3-3*q^2+3*q-1))*p[1, 1, 1] - ((1/3*t^3-1/3)/(q^3-
↪1))*p[3]
sage: J = Sym.macdonald().J()
sage: S(J([2]))
(q^3-q^2-q+1)*McdS[2]
sage: S(J([1,1]))
(q^2*t-q*t-q+1)*McdS[1, 1] + (q^2-q*t-q+t)*McdS[2]
sage: S = Sym.macdonald(q=0).S()
sage: S(J[1,1])
McdS[1, 1] + t*McdS[2]
sage: S(J[2])
q*McdS[1, 1] + McdS[2]
sage: p(S[2,1])
-(1/3*t^3-t^2+t-1/3)*p[1, 1, 1] + (1/3*t^3-1/3)*p[3]

sage: from sage.combinat.sf.macdonald import qt_kostka
sage: qt_kostka([2],[1,1])
t
sage: qt_kostka([1,1],[2])
q
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> S = Sym.macdonald().S(); S
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
```

(continues on next page)

(continued from previous page)

```
↪t
over Rational Field in the Macdonald S basis
>>> p = Sym.power()
>>> p(S[Integer(2),Integer(1)])
((1/3*t^3-t^2+t-1/3)/(q^3-3*q^2+3*q-1))*p[1, 1, 1] - ((1/3*t^3-1/3)/(q^3-
↪1))*p[3]
>>> J = Sym.macdonald().J()
>>> S(J([Integer(2)]))
(q^3-q^2-q+1)*McdS[2]
>>> S(J([Integer(1),Integer(1)]))
(q^2*t-q*t-q+1)*McdS[1, 1] + (q^2-q*t-q+t)*McdS[2]
>>> S = Sym.macdonald(q=Integer(0)).S()
>>> S(J[Integer(1),Integer(1)])
McdS[1, 1] + t*McdS[2]
>>> S(J[Integer(2)])
q*McdS[1, 1] + McdS[2]
>>> p(S[Integer(2),Integer(1)])
-(1/3*t^3-t^2+t-1/3)*p[1, 1, 1] + (1/3*t^3-1/3)*p[3]

>>> from sage.combinat.sf.macdonald import qt_kostka
>>> qt_kostka([Integer(2)],[Integer(1),Integer(1)])
t
>>> qt_kostka([Integer(1),Integer(1)],[Integer(2)])
q
```

Coercions to/from the Schur basis are implemented:

```
sage: S = Sym.macdonald().S()
sage: s = Sym.schur()
sage: S(s([2]))
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdS[1, 1] - ((q^2*t-q*t-q+1)/(-t^3+t^2+t-
↪1))*McdS[2]
sage: s(S([1,1]))
-((q*t^2-q*t-t+1)/(-q^3+q^2+q-1))*s[1, 1] + ((q*t-t^2-q+t)/(-q^3+q^2+q-
↪1))*s[2]
```

```
>>> from sage.all import *
>>> S = Sym.macdonald().S()
>>> s = Sym.schur()
>>> S(s([Integer(2)]))
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdS[1, 1] - ((q^2*t-q*t-q+1)/(-t^3+t^2+t-
↪1))*McdS[2]
>>> s(S([Integer(1),Integer(1)]))
-((q*t^2-q*t-t+1)/(-q^3+q^2+q-1))*s[1, 1] + ((q*t-t^2-q+t)/(-q^3+q^2+q-
↪1))*s[2]
```

base_ring()

Return the base ring of the symmetric functions where the Macdonald symmetric functions live.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT: the base ring associated to the corresponding symmetric function ring

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['q'].fraction_field())
sage: Mac = Sym.macdonald(t=0)
sage: Mac.base_ring()
Fraction Field of Univariate Polynomial Ring in q over Rational Field
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['q'].fraction_field())
>>> Mac = Sym.macdonald(t=Integer(0))
>>> Mac.base_ring()
Fraction Field of Univariate Polynomial Ring in q over Rational Field
```

symmetric_function_ring()

Return the base ring of the symmetric functions where the Macdonald symmetric functions live.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT: the symmetric function ring associated to the Macdonald bases

EXAMPLES:

```
sage: Mac = SymmetricFunctions(QQ['q'].fraction_field()).macdonald(t=0)
sage: Mac.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in q␣
↪over Rational Field
```

```
>>> from sage.all import *
>>> Mac = SymmetricFunctions(QQ['q'].fraction_field()).macdonald(t=Integer(0))
>>> Mac.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in q␣
↪over Rational Field
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_generic**(*macdonald*)

Bases: *SymmetricFunctionAlgebra_generic*

A class for methods for one of the Macdonald bases of the symmetric functions.

INPUT:

- `self` – a Macdonald basis
- `macdonald` – a family of Macdonald symmetric function bases

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q,t'])); Sym.rename('Sym'); Sym
Sym
sage: Sym.macdonald().P()
Sym in the Macdonald P basis
sage: Sym.macdonald(t=2).P()
Sym in the Macdonald P with t=2 basis
sage: Sym.rename()
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q,t'])); Sym.rename('Sym'); Sym
Sym
>>> Sym.macdonald().P()
Sym in the Macdonald P basis
>>> Sym.macdonald(t=Integer(2)).P()
Sym in the Macdonald P with t=2 basis
>>> Sym.rename()
```

class Element

Bases: *SymmetricFunctionAlgebra_generic_Element*

nabla(*q=None*, *t=None*, *power=1*)

Return the value of the nabla operator applied to `self`.

The eigenvectors of the nabla operator are the Macdonald polynomials in the Ht basis. For more information see: [BGHT1999].

The operator nabla acts on symmetric functions and has the Macdonald Ht basis as eigenfunctions and the eigenvalues are $q^{n(\mu')}t^{n(\mu)}$ where $n(\mu) = \sum_i (i-1)\mu_i$ and μ' is the conjugate shape of μ.

If the parameter `power` is an integer then it calculates nabla to that integer. The default value of `power` is 1.

INPUT:

- `self` – an element of a Macdonald basis
- `q`, `t` – (optional) parameters to specialize
- `power` – integer (default: 1)

OUTPUT: the symmetric function of ∇ acting on `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: P([1,1]).nabla()
((q^2*t+q*t^2-2*t)/(q*t-1))*McdP[1, 1] + McdP[2]
sage: P([1,1]).nabla(t=1)
((q^2*t+q*t-t-1)/(q*t-1))*McdP[1, 1] + McdP[2]
sage: H = Sym.macdonald().H()
sage: H([1,1]).nabla()
t*McdH[1, 1] - (t^2-1)*McdH[2]
sage: H([1,1]).nabla(q=1)
((t^2+q-t-1)/(q*t-1))*McdH[1, 1] - ((t^3-t^2-t+1)/(q*t-1))*McdH[2]
sage: H(0).nabla()
0
sage: H([2,2,1]).nabla(t=1/H.t)
-(q^2/(-t^4))*McdH[2, 2, 1]
sage: H([2,2,1]).nabla(t=1/H.t,power=-1)
-(t^4/(-q^2))*McdH[2, 2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> P = Sym.macdonald().P()
>>> P([Integer(1),Integer(1)]).nabla()
((q^2*t+q*t^2-2*t)/(q*t-1))*McdP[1, 1] + McdP[2]
```

(continues on next page)

(continued from previous page)

```
>>> P([Integer(1),Integer(1)]).nabla(t=Integer(1))
((q^2*t+q*t-t-1)/(q*t-1))*McdP[1, 1] + McdP[2]
>>> H = Sym.macdonald().H()
>>> H([Integer(1),Integer(1)]).nabla()
t*McdH[1, 1] - (t^2-1)*McdH[2]
>>> H([Integer(1),Integer(1)]).nabla(q=Integer(1))
((t^2+q-t-1)/(q*t-1))*McdH[1, 1] - ((t^3-t^2-t+1)/(q*t-1))*McdH[2]
>>> H(Integer(0)).nabla()
0
>>> H([Integer(2),Integer(2),Integer(1)]).nabla(t=Integer(1)/H.t)
-(q^2/(-t^4))*McdH[2, 2, 1]
>>> H([Integer(2),Integer(2),Integer(1)]).nabla(t=Integer(1)/H.t,power=-
↪Integer(1))
-(t^4/(-q^2))*McdH[2, 2, 1]
```

c1 (*part*)

Return the qt-Hall scalar product between `J(part)` and `P(part)`.

INPUT:

- `self` – a Macdonald basis
- `part` – a partition

OUTPUT:

- returns the qt-Hall scalar product between `J(part)` and `P(part)`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: P.c1(Partition([2,1]))
-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> P = Sym.macdonald().P()
>>> P.c1(Partition([Integer(2),Integer(1)]))
-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1
```

c2 (*part*)

Return the qt-Hall scalar product between `J(part)` and `Q(part)`.

INPUT:

- `self` – a Macdonald basis
- `part` – a partition

OUTPUT:

- returns the qt-Hall scalar product between `J(part)` and `Q(part)`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: P.c2(Partition([2,1]))
-q*t^4 + 2*q*t^3 - q*t^2 + t^2 - 2*t + 1
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> P = Sym.macdonald().P()
>>> P.c2(Partition([Integer(2),Integer(1)]))
-q*t^4 + 2*q*t^3 - q*t^2 + t^2 - 2*t + 1
```

construction()

Return a pair `(F, R)`, where `F` is a `SymmetricFunctionsFunctor` and R is a ring, such that `F(R)` returns `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q']))
sage: J = Sym.macdonald(t=2).J()
sage: J.construction()
(SymmetricFunctionsFunctor[Macdonald J with t=2],
 Fraction Field of Univariate Polynomial Ring in q over Rational Field)
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q']))
>>> J = Sym.macdonald(t=Integer(2)).J()
>>> J.construction()
(SymmetricFunctionsFunctor[Macdonald J with t=2],
 Fraction Field of Univariate Polynomial Ring in q over Rational Field)
```

macdonald_family()

Return the family of Macdonald bases associated to the basis `self`.

INPUT:

- `self` – a Macdonald basis

OUTPUT: the family of Macdonald symmetric functions associated to `self`

EXAMPLES:

```
sage: MacP = SymmetricFunctions(QQ['q'].fraction_field()).macdonald(t=0).P()
sage: MacP.macdonald_family()
Macdonald polynomials with t=0 over Fraction Field of Univariate Polynomial␣
↪Ring in q over Rational Field
```

```
>>> from sage.all import *
>>> MacP = SymmetricFunctions(QQ['q'].fraction_field()).
↪macdonald(t=Integer(0)).P()
>>> MacP.macdonald_family()
Macdonald polynomials with t=0 over Fraction Field of Univariate Polynomial␣
↪Ring in q over Rational Field
```

product(*left*, *right*)

Multiply an element of the Macdonald symmetric function basis `self` and another symmetric function

Convert to the Schur basis, do the multiplication there, and convert back to `self` basis.

INPUT:

- `self` – a Macdonald symmetric function basis
- `left` – an element of the basis `self`
- `right` – another symmetric function

OUTPUT: the product of `left` and `right` expanded in the basis `self`

EXAMPLES:

```
sage: Mac = SymmetricFunctions(FractionField(QQ['q','t'])).macdonald()
sage: H = Mac.H()
sage: J = Mac.J()
sage: P = Mac.P()
sage: Q = Mac.Q()
sage: Ht = Mac.Ht()
sage: J([1])^2 #indirect doctest
((q-1)/(q*t-1))*McdJ[1, 1] + ((t-1)/(q*t-1))*McdJ[2]
sage: J.product( J[1], J[2] )
-((q^2-1)/(-q^2*t+1))*McdJ[2, 1] - ((t-1)/(-q^2*t+1))*McdJ[3]
sage: H.product( H[1], H[2] )
((q^2-1)/(q^2*t-1))*McdH[2, 1] - ((t-1)/(-q^2*t+1))*McdH[3]
sage: P.product( P[1], P[2] )
-((q^3*t^2-q*t^2-q^2+1)/(-q^3*t^2+q^2*t+q*t-1))*McdP[2, 1] + McdP[3]
sage: Q.product(Q[1],Q[2])
McdQ[2, 1] + ((q^2*t-q^2+q*t-q+t-1)/(q^2*t-1))*McdQ[3]
sage: Ht.product(Ht[1],Ht[2])
((q^2-1)/(q^2-t))*McdHt[2, 1] + ((t-1)/(-q^2+t))*McdHt[3]
```

```
>>> from sage.all import *
>>> Mac = SymmetricFunctions(FractionField(QQ['q','t'])).macdonald()
>>> H = Mac.H()
>>> J = Mac.J()
>>> P = Mac.P()
>>> Q = Mac.Q()
>>> Ht = Mac.Ht()
>>> J([Integer(1)])**Integer(2) #indirect doctest
((q-1)/(q*t-1))*McdJ[1, 1] + ((t-1)/(q*t-1))*McdJ[2]
>>> J.product( J[Integer(1)], J[Integer(2)] )
-((q^2-1)/(-q^2*t+1))*McdJ[2, 1] - ((t-1)/(-q^2*t+1))*McdJ[3]
>>> H.product( H[Integer(1)], H[Integer(2)] )
((q^2-1)/(q^2*t-1))*McdH[2, 1] - ((t-1)/(-q^2*t+1))*McdH[3]
>>> P.product( P[Integer(1)], P[Integer(2)] )
-((q^3*t^2-q*t^2-q^2+1)/(-q^3*t^2+q^2*t+q*t-1))*McdP[2, 1] + McdP[3]
>>> Q.product(Q[Integer(1)],Q[Integer(2)])
McdQ[2, 1] + ((q^2*t-q^2+q*t-q+t-1)/(q^2*t-1))*McdQ[3]
>>> Ht.product(Ht[Integer(1)],Ht[Integer(2)])
((q^2-1)/(q^2-t))*McdHt[2, 1] + ((t-1)/(-q^2+t))*McdHt[3]
```

class `sage.combinat.sf.macdonald.`**`MacdonaldPolynomials_h`**(*macdonald*)

Bases: *MacdonaldPolynomials_generic*

The H basis is defined as $H_\mu = \sum_\lambda K_{\lambda\mu}(q,t) s_\lambda$ where $K_{\lambda\mu}(q,t)$ are the Macdonald Kostka coefficients.

In this implementation, it is calculated by using the Macdonald Ht basis and substituting $t \to 1/t$ and multiplying by $t^{n(\mu)}$.

INPUT:

- `self` – a Macdonald H basis
- `macdonald` – a family of Macdonald bases

class Element

Bases: *Element*

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_ht**(*macdonald*)

Bases: *MacdonaldPolynomials_generic*

The Ht basis is defined as $\tilde{H}_\mu = t^{n(\mu)} \sum_\lambda K_{\lambda\mu}(q, t^{-1}) s_\lambda$ where $K_{\lambda\mu}(q,t)$ are the Macdonald (q,t)-Kostka coefficients and $n(\mu) = \sum_i (i-1)\mu_i$.

It is implemented here by using a Pieri formula due to F. Bergeron and M. Haiman [BH2013].

INPUT:

- `self` – a Macdonald Ht basis
- `macdonald` – a family of Macdonald bases

class Element

Bases: *Element*

nabla(*q=None*, *t=None*, *power=1*)

Return the value of the nabla operator applied to `self`. The eigenvectors of the $nabla$ operator are the Macdonald polynomials in the Ht basis. For more information see: [BGHT1999].

The operator $nabla$ acts on symmetric functions and has the Macdonald Ht basis as eigenfunctions and the eigenvalues are $q^{n(\mu')} t^{n(\mu)}$ where $n(\mu) = \sum_i (i-1)\mu_i$.

If the parameter `power` is an integer then it calculates nabla to that integer. The default value of `power` is 1.

INPUT:

- `self` – an element of the Macdonald Ht basis
- `q`, `t` – (optional) parameters to specialize
- `power` – integer (default: 1)

OUTPUT: the symmetric function of ∇ acting on `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Ht = Sym.macdonald().Ht()
sage: t = Ht.t; q = Ht.q
sage: s = Sym.schur()
sage: a = sum(Ht(p) for p in Partitions(3))
sage: Ht(0).nabla()
0
sage: a.nabla() == t^3*Ht([1,1,1])+q*t*Ht([2,1]) + q^3*Ht([3])
True
sage: a.nabla(t=3) == 27*Ht([1,1,1])+3*q*Ht([2,1]) + q^3*Ht([3])
True
sage: a.nabla(q=3) == t^3*Ht([1,1,1])+3*t*Ht([2,1]) + 27*Ht([3])
True
```

(continues on next page)

(continued from previous page)

```
sage: Ht[2,1].nabla(power=-1)
1/(q*t)*McdHt[2, 1]
sage: Ht[2,1].nabla(power=4)
q^4*t^4*McdHt[2, 1]
sage: s(a.nabla(q=3))
(t^6+27*q^3+3*q*t^2)*s[1, 1, 1] + (t^5+t^4+27*q^2+3*q*t+3*t^2+27*q)*s[2,␣
↪1] + (t^3+3*t+27)*s[3]
sage: Ht = Sym.macdonald(q=3).Ht()
sage: a = sum(Ht(p) for p in Partitions(3))
sage: s(a.nabla())
(t^6+9*t^2+729)*s[1, 1, 1] + (t^5+t^4+3*t^2+9*t+324)*s[2, 1] + (t^
↪3+3*t+27)*s[3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> Ht = Sym.macdonald().Ht()
>>> t = Ht.t; q = Ht.q
>>> s = Sym.schur()
>>> a = sum(Ht(p) for p in Partitions(Integer(3)))
>>> Ht(Integer(0)).nabla()
0
>>> a.nabla() == t**Integer(3)*Ht([Integer(1),Integer(1),
↪Integer(1)])+q*t*Ht([Integer(2),Integer(1)]) +␣
↪q**Integer(3)*Ht([Integer(3)])
True
>>> a.nabla(t=Integer(3)) == Integer(27)*Ht([Integer(1),Integer(1),
↪Integer(1)])+Integer(3)*q*Ht([Integer(2),Integer(1)]) +␣
↪q**Integer(3)*Ht([Integer(3)])
True
>>> a.nabla(q=Integer(3)) == t**Integer(3)*Ht([Integer(1),Integer(1),
↪Integer(1)])+Integer(3)*t*Ht([Integer(2),Integer(1)]) +␣
↪Integer(27)*Ht([Integer(3)])
True
>>> Ht[Integer(2),Integer(1)].nabla(power=-Integer(1))
1/(q*t)*McdHt[2, 1]
>>> Ht[Integer(2),Integer(1)].nabla(power=Integer(4))
q^4*t^4*McdHt[2, 1]
>>> s(a.nabla(q=Integer(3)))
(t^6+27*q^3+3*q*t^2)*s[1, 1, 1] + (t^5+t^4+27*q^2+3*q*t+3*t^2+27*q)*s[2,␣
↪1] + (t^3+3*t+27)*s[3]
>>> Ht = Sym.macdonald(q=Integer(3)).Ht()
>>> a = sum(Ht(p) for p in Partitions(Integer(3)))
>>> s(a.nabla())
(t^6+9*t^2+729)*s[1, 1, 1] + (t^5+t^4+3*t^2+9*t+324)*s[2, 1] + (t^
↪3+3*t+27)*s[3]
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_j**(*macdonald*)

Bases: *MacdonaldPolynomials_generic*

The J basis is calculated using determinantal formulas of Lapointe-Lascoux-Morse giving the action on the S-basis.

INPUT:

- `self` – a Macdonald J basis

- `macdonald` – a family of Macdonald bases

class Element

Bases: *Element*

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_p**(*macdonald*)

Bases: *MacdonaldPolynomials_generic*

The P basis is defined here as the J basis times a normalizing coefficient $c2$.

INPUT:

- `self` – a Macdonald P basis
- `macdonald` – a family of Macdonald bases

class Element

Bases: *Element*

scalar_qt_basis(*part1*, *part2=None*)

Return the scalar product of $P(part1)$ and $P(part2)$ This scalar product formula is given in equation (4.11) p.323 and (6.19) p.339 of Macdonald's book [Mac1995].

INPUT:

- `self` – a Macdonald P basis
- `part1`, `part2` – partitions

OUTPUT:

- returns the scalar product of `P(part1)` and `P(part2)`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: P.scalar_qt_basis(Partition([2,1]), Partition([1,1,1]))
0
sage: f = P.scalar_qt_basis(Partition([3,2,1]), Partition([3,2,1]))
sage: factor(f.numerator())
(q - 1)^3 * (q^2*t - 1)^2 * (q^3*t^2 - 1)
sage: factor(f.denominator())
(t - 1)^3 * (q*t^2 - 1)^2 * (q^2*t^3 - 1)
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> P = Sym.macdonald().P()
>>> P.scalar_qt_basis(Partition([Integer(2),Integer(1)]),␣
↪Partition([Integer(1),Integer(1),Integer(1)]))
0
>>> f = P.scalar_qt_basis(Partition([Integer(3),Integer(2),Integer(1)]),␣
↪Partition([Integer(3),Integer(2),Integer(1)]))
>>> factor(f.numerator())
(q - 1)^3 * (q^2*t - 1)^2 * (q^3*t^2 - 1)
>>> factor(f.denominator())
(t - 1)^3 * (q*t^2 - 1)^2 * (q^2*t^3 - 1)
```

With a single argument, takes $part2 = part1$:

```
sage: P.scalar_qt_basis(Partition([2,1]), Partition([2,1]))
(-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1)/(-q*t^4 + 2*q*t^3 - q*t^2 + t^2 -
↪2*t + 1)
```

```
>>> from sage.all import *
>>> P.scalar_qt_basis(Partition([Integer(2),Integer(1)]),
↪Partition([Integer(2),Integer(1)]))
(-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1)/(-q*t^4 + 2*q*t^3 - q*t^2 + t^2 -
↪2*t + 1)
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_q**(*macdonald*)

Bases: *MacdonaldPolynomials_generic*

The Q basis is defined here as the J basis times a normalizing coefficient.

INPUT:

- `self` – a Macdonald Q basis
- `macdonald` – a family of Macdonald bases

class Element

Bases: *Element*

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_s**(*macdonald*)

Bases: *MacdonaldPolynomials_generic*

An implementation of the basis $s_\lambda[(1-t)X/(1-q)]$.

This is perhaps misnamed as a 'Macdonald' basis for the symmetric functions but is used in the calculation of the Macdonald J basis (see method 'creation' below) but does use both of the two parameters and can be specialized to $s_\lambda[(1-t)X]$ and $s_\lambda[X/(1-t)]$.

INPUT:

- `self` – a Macdonald S basis
- `macdonald` – a family of Macdonald bases

class Element

Bases: *Element*

creation(*k*)

This function is a creation operator for the J-basis for which the action is known on the 'Macdonald' S-basis by formula from [LLM1998].

INPUT:
- `self` – an element of the Macdonald S basis
- `k` – positive integer

OUTPUT: the column adding operator on the J basis on `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: S = Sym.macdonald().S()
sage: a = S(1)
sage: a.creation(1)
-(q-1)*McdS[1]
```

(continues on next page)

(continued from previous page)

```
sage: a.creation(2)
(q^2*t-q*t-q+1)*McdS[1, 1] + (q^2-q*t-q+t)*McdS[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> S = Sym.macdonald().S()
>>> a = S(Integer(1))
>>> a.creation(Integer(1))
-(q-1)*McdS[1]
>>> a.creation(Integer(2))
(q^2*t-q*t-q+1)*McdS[1, 1] + (q^2-q*t-q+t)*McdS[2]
```

product(*left*, *right*)

The multiplication of the modified Schur functions behaves the same as the multiplication of the Schur functions.

INPUT:

- `self` – a Macdonald S basis
- `left`, `right` – a symmetric functions

OUTPUT: the product of `left` and `right`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: S = Sym.macdonald().S()
sage: S([2])^2 #indirect doctest
McdS[2, 2] + McdS[3, 1] + McdS[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> S = Sym.macdonald().S()
>>> S([Integer(2)])**Integer(2) #indirect doctest
McdS[2, 2] + McdS[3, 1] + McdS[4]
```

`sage.combinat.sf.macdonald.`**c1**(*part*, *q*, *t*)

This function returns the qt-Hall scalar product between `J(part)` and `P(part)`.

This coefficient is c_λ in equation (8.1') p. 352 of Macdonald's book [Mac1995].

INPUT:

- `part` – a partition
- `q`, `t` – parameters

OUTPUT: a polynomial of the scalar product between the J and P bases

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import c1
sage: R.<q,t> = QQ[]
sage: c1(Partition([2,1]),q,t)
-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1
```

(continues on next page)

(continued from previous page)

```
sage: c1(Partition([1,1]),q,t)
q^2*t - q*t - q + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.macdonald import c1
>>> R = QQ['q, t']; (q, t,) = R._first_ngens(2)
>>> c1(Partition([Integer(2),Integer(1)]),q,t)
-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1
>>> c1(Partition([Integer(1),Integer(1)]),q,t)
q^2*t - q*t - q + 1
```

sage.combinat.sf.macdonald.**c2**(*part*, *q*, *t*)

This function returns the qt-Hall scalar product between J(part) and Q(part).

This coefficient is c_λ in equation (8.1) p. 352 of Macdonald's book [Mac1995].

INPUT:

- part – a partition
- q, t – parameters

OUTPUT: a polynomial of the scalar product between the J and P bases

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import c2
sage: R.<q,t> = QQ[]
sage: c2(Partition([1,1]),q,t)
t^3 - t^2 - t + 1
sage: c2(Partition([2,1]),q,t)
-q*t^4 + 2*q*t^3 - q*t^2 + t^2 - 2*t + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.macdonald import c2
>>> R = QQ['q, t']; (q, t,) = R._first_ngens(2)
>>> c2(Partition([Integer(1),Integer(1)]),q,t)
t^3 - t^2 - t + 1
>>> c2(Partition([Integer(2),Integer(1)]),q,t)
-q*t^4 + 2*q*t^3 - q*t^2 + t^2 - 2*t + 1
```

sage.combinat.sf.macdonald.**cmunu**(*nu*)

Return the coefficient of $\tilde{H}_\nu$ in $h_r^\perp \tilde{H}_\mu$.

Proposition 5 of F. Bergeron and M. Haiman [BH2013] states

$$c_{\mu\nu} = \sum_{\alpha \leftarrow \nu} c_{\mu\alpha} c_{\alpha\nu} B_{\alpha/\nu} / B_{\mu/\nu}$$

where $c_{\mu\nu}$ is the coefficient of $\tilde{H}_\nu$ in $h_r^\perp \tilde{H}_\mu$ and $B_{\mu/\nu}$ is the bi-exponent generator implemented in the function sage.combinat.sf.macdonald.Bmu().

INPUT:

- mu, nu – partitions with nu contained in mu

OUTPUT: an element of the fraction field of polynomials in q and t

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import cmunu
sage: cmunu(Partition([2,1]),Partition([1]))
q + t + 1
sage: cmunu(Partition([2,2]),Partition([1,1]))
(-q^3 - q^2 + q*t + t)/(-q + t)
sage: Sym = SymmetricFunctions(QQ['q','t'].fraction_field())
sage: h = Sym.h()
sage: Ht = Sym.macdonald().Ht()
sage: all(Ht[2,2].skew_by(h[r]).coefficient(nu)
....:     == cmunu(Partition([2,2]),nu)
....:     for r in range(1,5) for nu in Partitions(4-r))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.macdonald import cmunu
>>> cmunu(Partition([Integer(2),Integer(1)]),Partition([Integer(1)]))
q + t + 1
>>> cmunu(Partition([Integer(2),Integer(2)]),Partition([Integer(1),Integer(1)]))
(-q^3 - q^2 + q*t + t)/(-q + t)
>>> Sym = SymmetricFunctions(QQ['q','t'].fraction_field())
>>> h = Sym.h()
>>> Ht = Sym.macdonald().Ht()
>>> all(Ht[Integer(2),Integer(2)].skew_by(h[r]).coefficient(nu)
...     == cmunu(Partition([Integer(2),Integer(2)]),nu)
...     for r in range(Integer(1),Integer(5)) for nu in Partitions(Integer(4)-r))
True
```

sage.combinat.sf.macdonald.**cmunu1**(*nu*)

Return the coefficient of $\tilde{H}_\nu$ in $h_1^\perp \tilde{H}_\mu$.

INPUT:

- `mu`, `nu` – partitions with `nu` precedes `mu`

OUTPUT: an element of the fraction field of polynomials in q and t

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import cmunu1
sage: cmunu1(Partition([2,1]),Partition([2]))
(-t^2 + q)/(q - t)
sage: cmunu1(Partition([2,1]),Partition([1,1]))
(-q^2 + t)/(-q + t)
sage: Sym = SymmetricFunctions(QQ['q','t'].fraction_field())
sage: h = Sym.h()
sage: Ht = Sym.macdonald().Ht()
sage: all(Ht[3,2,1].skew_by(h[1]).coefficient(nu)
....:     == cmunu1(Partition([3,2,1]),nu)
....:     for nu in Partition([3,2,1]).down_list())
True
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.macdonald import cmunu1
>>> cmunu1(Partition([Integer(2),Integer(1)]),Partition([Integer(2)]))
(-t^2 + q)/(q - t)
>>> cmunu1(Partition([Integer(2),Integer(1)]),Partition([Integer(1),Integer(1)]))
(-q^2 + t)/(-q + t)
>>> Sym = SymmetricFunctions(QQ['q','t'].fraction_field())
>>> h = Sym.h()
>>> Ht = Sym.macdonald().Ht()
>>> all(Ht[Integer(3),Integer(2),Integer(1)].skew_by(h[Integer(1)]).
↪coefficient(nu)
...     == cmunu1(Partition([Integer(3),Integer(2),Integer(1)]),nu)
...     for nu in Partition([Integer(3),Integer(2),Integer(1)]).down_list())
True
```

sage.combinat.sf.macdonald.**qt_kostka**(*lam*, *mu*)

Return the $K_{\lambda\mu}(q,t)$ by computing the change of basis from the Macdonald H basis to the Schurs.

INPUT:

- lam, mu – partitions of the same size

OUTPUT:

- returns the q,t-Kostka polynomial indexed by the partitions lam and mu

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import qt_kostka
sage: qt_kostka([2,1,1],[1,1,1,1])
t^3 + t^2 + t
sage: qt_kostka([1,1,1,1],[2,1,1])
q
sage: qt_kostka([1,1,1,1],[3,1])
q^3
sage: qt_kostka([1,1,1,1],[1,1,1,1])
1
sage: qt_kostka([2,1,1],[2,2])
q^2*t + q*t + q
sage: qt_kostka([2,2],[2,2])
q^2*t^2 + 1
sage: qt_kostka([4],[3,1])
t
sage: qt_kostka([2,2],[3,1])
q^2*t + q
sage: qt_kostka([3,1],[2,1,1])
q*t^3 + t^2 + t
sage: qt_kostka([2,1,1],[2,1,1])
q*t^2 + q*t + 1
sage: qt_kostka([2,1],[1,1,1,1])
0
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.macdonald import qt_kostka
>>> qt_kostka([Integer(2),Integer(1),Integer(1)],[Integer(1),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(1)])
t^3 + t^2 + t
>>> qt_kostka([Integer(1),Integer(1),Integer(1),Integer(1)],[Integer(2),
↪Integer(1),Integer(1)])
q
>>> qt_kostka([Integer(1),Integer(1),Integer(1),Integer(1)],[Integer(3),
↪Integer(1)])
q^3
>>> qt_kostka([Integer(1),Integer(1),Integer(1),Integer(1)],[Integer(1),
↪Integer(1),Integer(1),Integer(1)])
1
>>> qt_kostka([Integer(2),Integer(1),Integer(1)],[Integer(2),Integer(2)])
q^2*t + q*t + q
>>> qt_kostka([Integer(2),Integer(2)],[Integer(2),Integer(2)])
q^2*t^2 + 1
>>> qt_kostka([Integer(4)],[Integer(3),Integer(1)])
t
>>> qt_kostka([Integer(2),Integer(2)],[Integer(3),Integer(1)])
q^2*t + q
>>> qt_kostka([Integer(3),Integer(1)],[Integer(2),Integer(1),Integer(1)])
q*t^3 + t^2 + t
>>> qt_kostka([Integer(2),Integer(1),Integer(1)],[Integer(2),Integer(1),
↪Integer(1)])
q*t^2 + q*t + 1
>>> qt_kostka([Integer(2),Integer(1)],[Integer(1),Integer(1),Integer(1),
↪Integer(1)])
0
```

5.1.293 Monomial symmetric functions

class sage.combinat.sf.monomial.**SymmetricFunctionAlgebra_monomial**(*Sym*)

Bases: *SymmetricFunctionAlgebra_classical*

A class for methods related to monomial symmetric functions.

INPUT:

- `self` – a monomial symmetric function basis
- `Sym` – an instance of the ring of the symmetric functions

class Element

Bases: *Element*

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: m([2,1]).expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: m([1,1,1]).expand(2)
0
sage: m([2,1]).expand(3,alphabet='z')
z0^2*z1 + z0*z1^2 + z0^2*z2 + z1^2*z2 + z0*z2^2 + z1*z2^2
sage: m([2,1]).expand(3,alphabet='x,y,z')
x^2*y + x*y^2 + x^2*z + y^2*z + x*z^2 + y*z^2
sage: m([1]).expand(0)
0
sage: (3*m([])).expand(0)
3
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> m([Integer(2),Integer(1)]).expand(Integer(3))
x0^2*x1 + x0*x1^2 + x0^2*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
>>> m([Integer(1),Integer(1),Integer(1)]).expand(Integer(2))
0
>>> m([Integer(2),Integer(1)]).expand(Integer(3),alphabet='z')
z0^2*z1 + z0*z1^2 + z0^2*z2 + z1^2*z2 + z0*z2^2 + z1*z2^2
>>> m([Integer(2),Integer(1)]).expand(Integer(3),alphabet='x,y,z')
x^2*y + x*y^2 + x^2*z + y^2*z + x*z^2 + y*z^2
>>> m([Integer(1)]).expand(Integer(0))
0
>>> (Integer(3)*m([])).expand(Integer(0))
3
```

exponential_specialization(*t=None*, *q=1*)

Return the exponential specialization of a symmetric function (when $q = 1$), or the q-exponential specialization (when $q \neq 1$).

The *exponential specialization* ex at t is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R. It is defined whenever the base ring K is a **Q**-algebra and t is an element of R. The easiest way to define it is by specifying its values on the powersum symmetric functions to be $p_1 = t$ and $p_n = 0$ for $n > 1$. Equivalently, on the homogeneous functions it is given by $ex(h_n) = t^n/n!$; see Proposition 7.8.4 of [EnumComb2].

By analogy, the q-exponential specialization is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R that depends on two elements t and q of R for which the elements $1 - q^i$ for all positive integers i are invertible. It can be defined by specifying its values on the complete homogeneous symmetric functions to be

$$ex_q(h_n) = t^n/[n]_q!,$$

where $[n]_q!$ is the q-factorial. Equivalently, for $q \neq 1$ and a homogeneous symmetric function f of degree n, we have

$$ex_q(f) = (1-q)^n t^n ps_q(f),$$

where $ps_q(f)$ is the stable principal specialization of f (see `principal_specialization()`). (See (7.29) in [EnumComb2].)

The limit of ex_q as $q \to 1$ is ex.

INPUT:

- `t` – (default: `None`) the value to use for t; the default is to create a ring of polynomials in `t`
- `q` – (default: 1) the value to use for q; if `q` is `None`, then a ring (or fraction field) of polynomials in `q` is created

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: (m[3]+m[2,1]+m[1,1,1]).exponential_specialization()
1/6*t^3

sage: x = 5*m[1,1,1] + 3*m[2,1] + 1
sage: x.exponential_specialization()
5/6*t^3 + 1
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> (m[Integer(3)]+m[Integer(2),Integer(1)]+m[Integer(1),Integer(1),
↪Integer(1)]).exponential_specialization()
1/6*t^3

>>> x = Integer(5)*m[Integer(1),Integer(1),Integer(1)] +␣
↪Integer(3)*m[Integer(2),Integer(1)] + Integer(1)
>>> x.exponential_specialization()
5/6*t^3 + 1
```

We also support the q-exponential_specialization:

```
sage: factor(m[3].exponential_specialization(q=var("q"), t=var("t")))    #␣
↪needs sage.symbolic
(q - 1)^2*t^3/(q^2 + q + 1)
```

```
>>> from sage.all import *
>>> factor(m[Integer(3)].exponential_specialization(q=var("q"), t=var("t
↪")))    # needs sage.symbolic
(q - 1)^2*t^3/(q^2 + q + 1)
```

principal_specialization(*n=+Infinity*, *q=None*)

Return the principal specialization of a symmetric function.

The *principal specialization* of order n at q is the ring homomorphism $ps_{n,q}$ from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for $i \in \{1, \dots, n\}$ and $x_i \mapsto 0$ for $i > n$. Here, q is a given element of R, and we assume that the variables of our symmetric functions are $x_1, x_2, x_3, \dots$. (To be more precise, $ps_{n,q}$ is a K-algebra homomorphism, where K is the base ring.) See Section 7.8 of [EnumComb2].

The *stable principal specialization* at q is the ring homomorphism ps_q from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for all i. This is well-defined only if the resulting infinite sums converge; thus, in particular, setting $q = 1$ in the stable principal specialization is an invalid operation.

INPUT:

- `n` – (default: `infinity`) a nonnegative integer or `infinity`, specifying whether to compute the principal specialization of order `n` or the stable principal specialization.
- `q` – (default: `None`) the value to use for q; the default is to create a ring of polynomials in `q` (or a field of rational functions in `q`) over the given coefficient ring.

For q=1 and finite n we use the formula from Proposition 7.8.3 of [EnumComb2]:

$$ps_{n,1}(m_\lambda) = \binom{n}{\ell(\lambda)} \binom{\ell(\lambda)}{m_1(\lambda), m_2(\lambda), \ldots},$$

where $\ell(\lambda)$ denotes the length of λ.

In all other cases, we convert to complete homogeneous symmetric functions.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: x = m[3,1]
sage: x.principal_specialization(3)
q^7 + q^6 + q^5 + q^3 + q^2 + q

sage: x = 5*m[2] + 3*m[1] + 1
sage: x.principal_specialization(3, q=var("q"))                                #␣
↪needs sage.symbolic
-10*(q^3 - 1)*q/(q - 1) + 5*(q^3 - 1)^2/(q - 1)^2 + 3*(q^3 - 1)/(q - 1) +␣
↪1
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> x = m[Integer(3),Integer(1)]
>>> x.principal_specialization(Integer(3))
q^7 + q^6 + q^5 + q^3 + q^2 + q

>>> x = Integer(5)*m[Integer(2)] + Integer(3)*m[Integer(1)] + Integer(1)
>>> x.principal_specialization(Integer(3), q=var("q"))                          ␣
↪     # needs sage.symbolic
-10*(q^3 - 1)*q/(q - 1) + 5*(q^3 - 1)^2/(q - 1)^2 + 3*(q^3 - 1)/(q - 1) +␣
↪1
```

antipode_by_coercion(*element*)

The antipode of `element` via coercion to and from the power-sum basis or the Schur basis (depending on whether the power sums really form a basis over the given ground ring).

INPUT:

- `element` – element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: m[3,2].antipode()
m[3, 2] + 2*m[5]
sage: m.antipode_by_coercion(m[3,2])
m[3, 2] + 2*m[5]

sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.monomial()
sage: m[3,2].antipode()
m[3, 2] + 2*m[5]
sage: m.antipode_by_coercion(m[3,2])
m[3, 2] + 2*m[5]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.monomial()
>>> m[Integer(3),Integer(2)].antipode()
m[3, 2] + 2*m[5]
>>> m.antipode_by_coercion(m[Integer(3),Integer(2)])
m[3, 2] + 2*m[5]

>>> Sym = SymmetricFunctions(ZZ)
>>> m = Sym.monomial()
>>> m[Integer(3),Integer(2)].antipode()
m[3, 2] + 2*m[5]
>>> m.antipode_by_coercion(m[Integer(3),Integer(2)])
m[3, 2] + 2*m[5]
```

Todo

Is there a not too difficult way to get the power-sum computations to work over any ring, not just one with coercion from **Q**?

from_polynomial(*f*, *check=True*)

Return the symmetric function in the monomial basis corresponding to the polynomial `f`.

INPUT:

- `self` – a monomial symmetric function basis
- `f` – a polynomial in finitely many variables over the same base ring as `self`; it is assumed that this polynomial is symmetric
- `check` – boolean (default: `True`); checks whether the polynomial is indeed symmetric

OUTPUT:

- This function converts a symmetric polynomial f in a polynomial ring in finitely many variables to a symmetric function in the monomial basis of the ring of symmetric functions over the same base ring.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: P = PolynomialRing(QQ, 'x', 3)
sage: x = P.gens()
sage: f = x[0] + x[1] + x[2]
sage: m.from_polynomial(f)
m[1]
sage: f = x[0]**2+x[1]**2+x[2]**2
sage: m.from_polynomial(f)
m[2]
sage: f = x[0]^2+x[1]
sage: m.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a symmetric polynomial
sage: f = (m[2,1]+m[1,1]).expand(3)
sage: m.from_polynomial(f)
```

(continues on next page)

(continued from previous page)

```
m[1, 1] + m[2, 1]
sage: f = (2*m[2,1]+m[1,1]+3*m[3]).expand(3)
sage: m.from_polynomial(f)
m[1, 1] + 2*m[2, 1] + 3*m[3]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> P = PolynomialRing(QQ, 'x', Integer(3))
>>> x = P.gens()
>>> f = x[Integer(0)] + x[Integer(1)] + x[Integer(2)]
>>> m.from_polynomial(f)
m[1]
>>> f =␣
↪x[Integer(0)]**Integer(2)+x[Integer(1)]**Integer(2)+x[Integer(2)]**Integer(2)
>>> m.from_polynomial(f)
m[2]
>>> f = x[Integer(0)]**Integer(2)+x[Integer(1)]
>>> m.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a symmetric polynomial
>>> f = (m[Integer(2),Integer(1)]+m[Integer(1),Integer(1)]).expand(Integer(3))
>>> m.from_polynomial(f)
m[1, 1] + m[2, 1]
>>> f = (Integer(2)*m[Integer(2),Integer(1)]+m[Integer(1),
↪Integer(1)]+Integer(3)*m[Integer(3)]).expand(Integer(3))
>>> m.from_polynomial(f)
m[1, 1] + 2*m[2, 1] + 3*m[3]
```

from_polynomial_exp(p)

Conversion from polynomial in exponential notation.

INPUT:

- `self` – a monomial symmetric function basis
- `p` – a polynomial over the same base ring as `self`

OUTPUT:

- This returns a symmetric function by mapping each monomial of p with exponents `exp` into m_λ where λ is the partition with exponential notation `exp`.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: P = PolynomialRing(QQ,'x',5)
sage: x = P.gens()
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> P = PolynomialRing(QQ,'x',Integer(5))
>>> x = P.gens()
```

The exponential notation of the partition $(5, 5, 5, 3, 1, 1)$ is:

```
sage: Partition([5,5,5,3,1,1]).to_exp()
[2, 0, 1, 0, 3]
```

```
>>> from sage.all import *
>>> Partition([Integer(5),Integer(5),Integer(5),Integer(3),Integer(1),
↪Integer(1)]).to_exp()
[2, 0, 1, 0, 3]
```

Therefore, the monomial:

```
sage: f = x[0]^2 * x[2] * x[4]^3
```

```
>>> from sage.all import *
>>> f = x[Integer(0)]**Integer(2) * x[Integer(2)] * x[Integer(4)]**Integer(3)
```

is mapped to:

```
sage: m.from_polynomial_exp(f)
m[5, 5, 5, 3, 1, 1]
```

```
>>> from sage.all import *
>>> m.from_polynomial_exp(f)
m[5, 5, 5, 3, 1, 1]
```

Furthermore, this function is linear:

```
sage: f = 3 * x[3] + 2 * x[0]^2 * x[2] * x[4]^3
sage: m.from_polynomial_exp(f)
3*m[4] + 2*m[5, 5, 5, 3, 1, 1]
```

```
>>> from sage.all import *
>>> f = Integer(3) * x[Integer(3)] + Integer(2) * x[Integer(0)]**Integer(2) *␣
↪x[Integer(2)] * x[Integer(4)]**Integer(3)
>>> m.from_polynomial_exp(f)
3*m[4] + 2*m[5, 5, 5, 3, 1, 1]
```

See also:

Partition(), *Partition.to_exp()*

product (*left*, *right*)

Return the product of `left` and `right`.

- `left`, `right` – symmetric functions written in the monomial basis `self`

OUTPUT:

- the product of `left` and `right`, expanded in the monomial basis, as a dictionary whose keys are partitions and whose values are the coefficients of these partitions (more precisely, their respective monomial symmetric functions) in the product.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: a = m([2,1])
sage: a^2
4*m[2, 2, 1, 1] + 6*m[2, 2, 2] + 2*m[3, 2, 1] + 2*m[3, 3] + 2*m[4, 1, 1] +␣
↪m[4, 2]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> a = m([Integer(2),Integer(1)])
>>> a**Integer(2)
4*m[2, 2, 1, 1] + 6*m[2, 2, 2] + 2*m[3, 2, 1] + 2*m[3, 3] + 2*m[4, 1, 1] +␣
↪m[4, 2]
```

```
sage: QQx.<x> = QQ['x']
sage: m = SymmetricFunctions(QQx).m()
sage: a = m([2,1])+x
sage: 2*a # indirect doctest
2*x*m[] + 2*m[2, 1]
sage: a^2
x^2*m[] + 2*x*m[2, 1] + 4*m[2, 2, 1, 1] + 6*m[2, 2, 2] + 2*m[3, 2, 1] + 2*m[3,
↪ 3] + 2*m[4, 1, 1] + m[4, 2]
```

```
>>> from sage.all import *
>>> QQx = QQ['x']; (x,) = QQx._first_ngens(1)
>>> m = SymmetricFunctions(QQx).m()
>>> a = m([Integer(2),Integer(1)])+x
>>> Integer(2)*a # indirect doctest
2*x*m[] + 2*m[2, 1]
>>> a**Integer(2)
x^2*m[] + 2*x*m[2, 1] + 4*m[2, 2, 1, 1] + 6*m[2, 2, 2] + 2*m[3, 2, 1] + 2*m[3,
↪ 3] + 2*m[4, 1, 1] + m[4, 2]
```

5.1.294 Multiplicative symmetric functions

A realization h of the ring of symmetric functions is multiplicative if for a partition $\lambda = (\lambda_1, \lambda_2, \ldots)$ we have $h_\lambda = h_{\lambda_1} h_{\lambda_2} \cdots$.

class sage.combinat.sf.multiplicative.**SymmetricFunctionAlgebra_multiplicative**(*Sym*, *basis_name=None*, *prefix=None*, *graded=True*)

Bases: *SymmetricFunctionAlgebra_classical*

The class of multiplicative bases of the ring of symmetric functions.

A realization q of the ring of symmetric functions is multiplicative if for a partition $\lambda = (\lambda_1, \lambda_2, \ldots)$ we have $q_\lambda = q_{\lambda_1} q_{\lambda_2} \cdots$ (with q_0 meaning 1).

Examples of multiplicative realizations are the elementary symmetric basis, the complete homogeneous basis, the powersum basis (if the base ring is a **Q**-algebra), and the Witt basis (but not the Schur basis or the monomial basis).

coproduct_on_basis(*mu*)

Return the coproduct on a basis element for multiplicative bases.

INPUT:

- `mu` – a partition

OUTPUT:

- the image of `self[mu]` under comultiplication; this is an element of the tensor square of `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.powersum()
sage: p.coproduct_on_basis([2,1])
p[] # p[2, 1] + p[1] # p[2] + p[2] # p[1] + p[2, 1] # p[]

sage: e = Sym.elementary()
sage: e.coproduct_on_basis([3,1])
e[] # e[3, 1] + e[1] # e[2, 1] + e[1] # e[3] + e[1, 1] # e[2] + e[2] # e[1,
↪1] + e[2, 1] # e[1] + e[3] # e[1] + e[3, 1] # e[]

sage: h = Sym.homogeneous()
sage: h.coproduct_on_basis([3,1])
h[] # h[3, 1] + h[1] # h[2, 1] + h[1] # h[3] + h[1, 1] # h[2] + h[2] # h[1,
↪1] + h[2, 1] # h[1] + h[3] # h[1] + h[3, 1] # h[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> p = Sym.powersum()
>>> p.coproduct_on_basis([Integer(2),Integer(1)])
p[] # p[2, 1] + p[1] # p[2] + p[2] # p[1] + p[2, 1] # p[]

>>> e = Sym.elementary()
>>> e.coproduct_on_basis([Integer(3),Integer(1)])
e[] # e[3, 1] + e[1] # e[2, 1] + e[1] # e[3] + e[1, 1] # e[2] + e[2] # e[1,
↪1] + e[2, 1] # e[1] + e[3] # e[1] + e[3, 1] # e[]

>>> h = Sym.homogeneous()
>>> h.coproduct_on_basis([Integer(3),Integer(1)])
h[] # h[3, 1] + h[1] # h[2, 1] + h[1] # h[3] + h[1, 1] # h[2] + h[2] # h[1,
↪1] + h[2, 1] # h[1] + h[3] # h[1] + h[3, 1] # h[]
```

product_on_basis(*left*, *right*)

Return the product of `left` and `right`.

INPUT:

- `left`, `right` – partitions

OUTPUT: an element of `self`

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: e([2,1])^2  # indirect doctest
e[2, 2, 1, 1]
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> e([Integer(2),Integer(1)])**Integer(2)  # indirect doctest
e[2, 2, 1, 1]
```

```
sage: h = SymmetricFunctions(QQ).h()
sage: h([2,1])^2
h[2, 2, 1, 1]
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> h([Integer(2),Integer(1)])**Integer(2)
h[2, 2, 1, 1]
```

```
sage: p = SymmetricFunctions(QQ).p()
sage: p([2,1])^2
p[2, 2, 1, 1]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> p([Integer(2),Integer(1)])**Integer(2)
p[2, 2, 1, 1]
```

```
sage: QQx.<x> = QQ[]
sage: p = SymmetricFunctions(QQx).p()
sage: (x*p([2]))^2
x^2*p[2, 2]

sage: TestSuite(p).run() # to silence sage -coverage
```

```
>>> from sage.all import *
>>> QQx = QQ['x']; (x,) = QQx._first_ngens(1)
>>> p = SymmetricFunctions(QQx).p()
>>> (x*p([Integer(2)]))**Integer(2)
x^2*p[2, 2]

>>> TestSuite(p).run() # to silence sage -coverage
```

5.1.295 k-Schur Functions

class sage.combinat.sf.new_kschur.**KBoundedSubspace**(*Sym*, *k*, *t='t'*)

Bases: UniqueRepresentation, Parent

This class implements the subspace of the ring of symmetric functions spanned by $\{s_\lambda[X/(1-t)]\}_{\lambda_1\le k} = \{s_\lambda^{(k)}[X;t]\}_{\lambda_1\le k}$ over the base ring $\mathbf{Q}[t]$. When $t=1$, this space is in fact a subring of the ring of symmetric functions generated by the complete homogeneous symmetric functions h_i for $1\le i\le k$.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: KB = Sym.kBoundedSubspace(3,1); KB
```

(continues on next page)

(continued from previous page)

```
3-bounded Symmetric Functions over Rational Field with t=1

sage: Sym = SymmetricFunctions(QQ['t'])
sage: KB = Sym.kBoundedSubspace(3); KB
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational␣
↪Field
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> KB = Sym.kBoundedSubspace(Integer(3),Integer(1)); KB
3-bounded Symmetric Functions over Rational Field with t=1

>>> Sym = SymmetricFunctions(QQ['t'])
>>> KB = Sym.kBoundedSubspace(Integer(3)); KB
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational␣
↪Field
```

The k-Schur function basis can be constructed as follows:

```
sage: ks = KB.kschur(); ks
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational␣
↪Field in the 3-Schur basis
```

```
>>> from sage.all import *
>>> ks = KB.kschur(); ks
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational␣
↪Field in the 3-Schur basis
```

K_kschur()

Return the k-bounded basis called the K-k-Schur basis.

See [Morse11] and [LamSchillingShimozono10].

REFERENCES:

EXAMPLES:

```
sage: kB = SymmetricFunctions(QQ).kBoundedSubspace(3,1)
sage: g = kB.K_kschur()
sage: g
3-bounded Symmetric Functions over Rational Field with t=1 in the K-3-Schur␣
↪basis
sage: kB = SymmetricFunctions(QQ['t']).kBoundedSubspace(3)
sage: g = kB.K_kschur()
Traceback (most recent call last):
...
ValueError: This basis only exists for t=1
```

```
>>> from sage.all import *
>>> kB = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1))
>>> g = kB.K_kschur()
>>> g
3-bounded Symmetric Functions over Rational Field with t=1 in the K-3-Schur␣
```

(continues on next page)

(continued from previous page)

```
↪basis
>>> kB = SymmetricFunctions(QQ['t']).kBoundedSubspace(Integer(3))
>>> g = kB.K_kschur()
Traceback (most recent call last):
...
ValueError: This basis only exists for t=1
```

khomogeneous()

The homogeneous basis of this algebra.

See also

kHomogeneous()

EXAMPLES:

```
sage: kh3 = SymmetricFunctions(QQ).kBoundedSubspace(3,1).khomogeneous()
sage: TestSuite(kh3).run()
```

```
>>> from sage.all import *
>>> kh3 = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).
↪khomogeneous()
>>> TestSuite(kh3).run()
```

kschur()

The k-Schur basis of this algebra.

See also

kSchur()

EXAMPLES:

```
sage: ks3 = SymmetricFunctions(QQ).kBoundedSubspace(3,1).kschur()
sage: TestSuite(ks3).run()
```

```
>>> from sage.all import *
>>> ks3 = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).
↪kschur()
>>> TestSuite(ks3).run()
```

ksplit()

The k-split basis of this algebra.

See also

kSplit()

EXAMPLES:

```
sage: ksp3 = SymmetricFunctions(QQ).kBoundedSubspace(3,1).ksplit()
sage: TestSuite(ksp3).run()
```

```
>>> from sage.all import *
>>> ksp3 = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).
↪ksplit()
>>> TestSuite(ksp3).run()
```

realizations()

A list of realizations of this algebra.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedSubspace(3,1).realizations()
[3-bounded Symmetric Functions over Rational Field with t=1 in the 3-Schur␣
↪basis,
 3-bounded Symmetric Functions over Rational Field with t=1 in the 3-split␣
↪basis,
 3-bounded Symmetric Functions over Rational Field with t=1 in the 3-bounded␣
↪homogeneous basis,
 3-bounded Symmetric Functions over Rational Field with t=1 in the K-3-Schur␣
↪basis]
sage: SymmetricFunctions(QQ['t']).kBoundedSubspace(3).realizations()
[3-bounded Symmetric Functions over Univariate Polynomial Ring in t over␣
↪Rational Field in the 3-Schur basis,
 3-bounded Symmetric Functions over Univariate Polynomial Ring in t over␣
↪Rational Field in the 3-split basis]
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).
↪realizations()
[3-bounded Symmetric Functions over Rational Field with t=1 in the 3-Schur␣
↪basis,
 3-bounded Symmetric Functions over Rational Field with t=1 in the 3-split␣
↪basis,
 3-bounded Symmetric Functions over Rational Field with t=1 in the 3-bounded␣
↪homogeneous basis,
 3-bounded Symmetric Functions over Rational Field with t=1 in the K-3-Schur␣
↪basis]
>>> SymmetricFunctions(QQ['t']).kBoundedSubspace(Integer(3)).realizations()
[3-bounded Symmetric Functions over Univariate Polynomial Ring in t over␣
↪Rational Field in the 3-Schur basis,
 3-bounded Symmetric Functions over Univariate Polynomial Ring in t over␣
↪Rational Field in the 3-split basis]
```

retract(*sym*)

Return the retract of `sym` from the ring of symmetric functions to `self`.

INPUT:

- `sym` – a symmetric function

OUTPUT:

- the analogue of the symmetric function in the k-bounded subspace (if possible)

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: KB = Sym.kBoundedSubspace(3,1); KB
3-bounded Symmetric Functions over Rational Field with t=1
sage: KB.retract(s[2]+s[3])
ks3[2] + ks3[3]
sage: KB.retract(s[2,1,1])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1] is not in the image
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.schur()
>>> KB = Sym.kBoundedSubspace(Integer(3),Integer(1)); KB
3-bounded Symmetric Functions over Rational Field with t=1
>>> KB.retract(s[Integer(2)]+s[Integer(3)])
ks3[2] + ks3[3]
>>> KB.retract(s[Integer(2),Integer(1),Integer(1)])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1] is not in the image
```

class sage.combinat.sf.new_kschur.**KBoundedSubspaceBases**(*base*, *t='t'*)

Bases: Category_realization_of_parent

The category of bases for the k-bounded subspace of symmetric functions.

class ElementMethods

Bases: object

expand(**args*, ***kwargs*)

Return the monomial expansion of `self` in n variables.

INPUT:

- `n` – positive integer

OUTPUT: monomial expansion of `self` in n variables

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: ks[3,1].expand(2)
x0^4 + 2*x0^3*x1 + 2*x0^2*x1^2 + 2*x0*x1^3 + x1^4
sage: s = Sym.schur()
sage: ks[3,1].expand(2) == s(ks[3,1]).expand(2)
True

sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks = Sym.kschur(3)
sage: f = ks[3,2]-ks[1]
sage: f.expand(2)
t^2*x0^5 + (t^2 + t)*x0^4*x1 + (t^2 + t + 1)*x0^3*x1^2 + (t^2 + t + 1)*x0^
↪2*x1^3 + (t^2 + t)*x0*x1^4 + t^2*x1^5 - x0 - x1
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ks = Sym.kschur(Integer(3),Integer(1))
>>> ks[Integer(3),Integer(1)].expand(Integer(2))
x0^4 + 2*x0^3*x1 + 2*x0^2*x1^2 + 2*x0*x1^3 + x1^4
>>> s = Sym.schur()
>>> ks[Integer(3),Integer(1)].expand(Integer(2)) == s(ks[Integer(3),
↪Integer(1)]).expand(Integer(2))
True

>>> Sym = SymmetricFunctions(QQ['t'])
>>> ks = Sym.kschur(Integer(3))
>>> f = ks[Integer(3),Integer(2)]-ks[Integer(1)]
>>> f.expand(Integer(2))
t^2*x0^5 + (t^2 + t)*x0^4*x1 + (t^2 + t + 1)*x0^3*x1^2 + (t^2 + t + 1)*x0^
↪2*x1^3 + (t^2 + t)*x0*x1^4 + t^2*x1^5 - x0 - x1
```

hl_creation_operator(*nu*, *t=None*)

This is the vertex operator that generalizes Jing's operator.

It is a linear operator that raises the degree by $|\nu|$. This creation operator is a t-analogue of multiplication by `s(nu)` .

See also

Proposition 5 in [SZ2001].

INPUT:

- `nu` – a partition or a list of integers
- `t` – (default: `None`, in which case `t` is used) an element of the base ring

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: ks = Sym.kschur(4)
sage: s = Sym.schur()
sage: s(ks([3,1,1]).hl_creation_operator([1]))
(t-1)*s[2, 2, 1, 1] + t^2*s[3, 1, 1, 1] + (t^3+t^2-t)*s[3, 2, 1] + (t^3-t^
↪2)*s[3, 3] + (t^4+t^3)*s[4, 1, 1] + t^4*s[4, 2] + t^5*s[5, 1]
sage: ks([3,1,1]).hl_creation_operator([1])
(t-1)*ks4[2, 2, 1, 1] + t^2*ks4[3, 1, 1, 1] + t^3*ks4[3, 2, 1] + (t^3-t^
↪2)*ks4[3, 3] + t^4*ks4[4, 1, 1]

sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(4,t=1)
sage: ks([3,1,1]).hl_creation_operator([1])
ks4[3, 1, 1, 1] + ks4[3, 2, 1] + ks4[4, 1, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> ks = Sym.kschur(Integer(4))
>>> s = Sym.schur()
>>> s(ks([Integer(3),Integer(1),Integer(1)]).hl_creation_
```

(continues on next page)

(continued from previous page)

```
↪operator([Integer(1)]))
(t-1)*s[2, 2, 1, 1] + t^2*s[3, 1, 1, 1] + (t^3+t^2-t)*s[3, 2, 1] + (t^3-t^
↪2)*s[3, 3] + (t^4+t^3)*s[4, 1, 1] + t^4*s[4, 2] + t^5*s[5, 1]
>>> ks([Integer(3),Integer(1),Integer(1)]).hl_creation_
↪operator([Integer(1)])
(t-1)*ks4[2, 2, 1, 1] + t^2*ks4[3, 1, 1, 1] + t^3*ks4[3, 2, 1] + (t^3-t^
↪2)*ks4[3, 3] + t^4*ks4[4, 1, 1]

>>> Sym = SymmetricFunctions(QQ)
>>> ks = Sym.kschur(Integer(4),t=Integer(1))
>>> ks([Integer(3),Integer(1),Integer(1)]).hl_creation_
↪operator([Integer(1)])
ks4[3, 1, 1, 1] + ks4[3, 2, 1] + ks4[4, 1, 1]
```

is_schur_positive(**args*, ***kwargs*)

Return whether `self` is Schur positive.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: f = ks[3,2]+ks[1]
sage: f.is_schur_positive()
True
sage: f = ks[3,2]-ks[1]
sage: f.is_schur_positive()
False

sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks = Sym.kschur(3)
sage: f = ks[3,2]+ks[1]
sage: f.is_schur_positive()
True
sage: f = ks[3,2]-ks[1]
sage: f.is_schur_positive()
False
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ks = Sym.kschur(Integer(3),Integer(1))
>>> f = ks[Integer(3),Integer(2)]+ks[Integer(1)]
>>> f.is_schur_positive()
True
>>> f = ks[Integer(3),Integer(2)]-ks[Integer(1)]
>>> f.is_schur_positive()
False

>>> Sym = SymmetricFunctions(QQ['t'])
>>> ks = Sym.kschur(Integer(3))
>>> f = ks[Integer(3),Integer(2)]+ks[Integer(1)]
>>> f.is_schur_positive()
True
```

(continues on next page)

(continued from previous page)

```
>>> f = ks[Integer(3),Integer(2)]-ks[Integer(1)]
>>> f.is_schur_positive()
False
```

omega()

Return the ω operator on `self`.

At $t = 1$, ω maps the k-Schur function $s^{(k)}_\lambda$ to $s^{(k)}_{\lambda^{(k)}}$, where $\lambda^{(k)}$ is the k-conjugate of the partition λ.

See also:

k_conjugate().

For generic t, ω sends $s^{(k)}_\lambda[X; t]$ to $t^d s^{(k)}_{\lambda^{(k)}}[X; 1/t]$, where d is the size of the core of λ minus the size of λ. Most of the time, this result is not in the k-bounded subspace.

See also:

omega_t_inverse().

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: ks[2,2,1,1].omega()
ks3[2, 2, 2]
sage: kh = Sym.khomogeneous(3)
sage: kh[3].omega()
h3[1, 1, 1] - 2*h3[2, 1] + h3[3]

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: ks = Sym.kschur(3)
sage: ks[3,1,1].omega()
Traceback (most recent call last):
...
ValueError: t*s[2, 1, 1, 1] + s[3, 1, 1] is not in the image
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ks = Sym.kschur(Integer(3),Integer(1))
>>> ks[Integer(2),Integer(2),Integer(1),Integer(1)].omega()
ks3[2, 2, 2]
>>> kh = Sym.khomogeneous(Integer(3))
>>> kh[Integer(3)].omega()
h3[1, 1, 1] - 2*h3[2, 1] + h3[3]

>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> ks = Sym.kschur(Integer(3))
>>> ks[Integer(3),Integer(1),Integer(1)].omega()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: t*s[2, 1, 1, 1] + s[3, 1, 1] is not in the image
```

omega_t_inverse()

Return the map $t \to 1/t$ composed with ω on `self`.

Unlike the map *omega()*, the result of *omega_t_inverse()* lives in the k-bounded subspace and hence will return an element even for generic t. For $t = 1$, *omega()* and *omega_t_inverse()* return the same result.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: ks = Sym.kschur(3)
sage: ks[3,1,1].omega_t_inverse()
1/t*ks3[2, 1, 1, 1]
sage: ks[3,2].omega_t_inverse()
1/t^2*ks3[1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> ks = Sym.kschur(Integer(3))
>>> ks[Integer(3),Integer(1),Integer(1)].omega_t_inverse()
1/t*ks3[2, 1, 1, 1]
>>> ks[Integer(3),Integer(2)].omega_t_inverse()
1/t^2*ks3[1, 1, 1, 1, 1]
```

scalar(*x*, *zee=None*)

Return standard scalar product between `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `zee` – an optional function on partitions giving the value for the scalar product between p_μ and p_μ (default: use the standard *zee()* function)

See also

scalar()

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks3 = Sym.kschur(3)
sage: ks3[3,2,1].scalar( ks3[2,2,2] )
t^3 + t
sage: dks3 = Sym.kBoundedQuotient(3).dks()
sage: [ks3[3,2,1].scalar(dks3(la)) for la in Partitions(6, max_part=3)]
[0, 1, 0, 0, 0, 0, 0]
sage: dks3 = Sym.kBoundedQuotient(3,t=1).dks()
sage: [ks3[2,2,2].scalar(dks3(la)) for la in Partitions(6, max_part=3)]
[0, t - 1, 0, 1, 0, 0, 0]
sage: ks3 = Sym.kschur(3,t=1)
sage: [ks3[2,2,2].scalar(dks3(la)) for la in Partitions(6, max_part=3)]
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0, 1, 0, 0, 0]
sage: kH = Sym.khomogeneous(4)
sage: kH([2,2,1]).scalar(ks3[2,2,1])
3
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'])
>>> ks3 = Sym.kschur(Integer(3))
>>> ks3[Integer(3),Integer(2),Integer(1)].scalar( ks3[Integer(2),
↪Integer(2),Integer(2)] )
t^3 + t
>>> dks3 = Sym.kBoundedQuotient(Integer(3)).dks()
>>> [ks3[Integer(3),Integer(2),Integer(1)].scalar(dks3(la)) for la in␣
↪Partitions(Integer(6), max_part=Integer(3))]
[0, 1, 0, 0, 0, 0, 0]
>>> dks3 = Sym.kBoundedQuotient(Integer(3),t=Integer(1)).dks()
>>> [ks3[Integer(2),Integer(2),Integer(2)].scalar(dks3(la)) for la in␣
↪Partitions(Integer(6), max_part=Integer(3))]
[0, t - 1, 0, 1, 0, 0, 0]
>>> ks3 = Sym.kschur(Integer(3),t=Integer(1))
>>> [ks3[Integer(2),Integer(2),Integer(2)].scalar(dks3(la)) for la in␣
↪Partitions(Integer(6), max_part=Integer(3))]
[0, 0, 0, 1, 0, 0, 0]
>>> kH = Sym.khomogeneous(Integer(4))
>>> kH([Integer(2),Integer(2),Integer(1)]).scalar(ks3[Integer(2),
↪Integer(2),Integer(1)])
3
```

class ParentMethods

Bases: `object`

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t']).kschur(3).an_element()
2*ks3[] + 2*ks3[1] + 3*ks3[2]
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ['t']).kschur(Integer(3)).an_element()
2*ks3[] + 2*ks3[1] + 3*ks3[2]
```

antipode(*element*)

Return the antipode on `self` by lifting to the space of symmetric functions, computing the antipode, and then converting to `self.parent()`. This is only the antipode for $t = 1$ and for other values of t the result may not be in the space where the k-Schur functions live.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks3 = Sym.kschur(3,1)
sage: ks3[3,2].antipode()
-ks3[1, 1, 1, 1, 1]
sage: ks3.antipode(ks3[3,2])
-ks3[1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ks3 = Sym.kschur(Integer(3),Integer(1))
>>> ks3[Integer(3),Integer(2)].antipode()
-ks3[1, 1, 1, 1, 1]
>>> ks3.antipode(ks3[Integer(3),Integer(2)])
-ks3[1, 1, 1, 1, 1]
```

coproduct (*element*)

Return the coproduct operation on `element`.

The coproduct is first computed on the homogeneous basis if $t = 1$ and on the Hall-Littlewood `Qp` basis otherwise. The result is computed then converted to the tensor squared of `self.parent()`.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks3 = Sym.kschur(3,1)
sage: ks3[2,1].coproduct()
ks3[] # ks3[2, 1] + ks3[1] # ks3[1, 1] + ks3[1] # ks3[2] + ks3[1, 1] #
→ks3[1] + ks3[2] # ks3[1] + ks3[2, 1] # ks3[]
sage: h3 = Sym.khomogeneous(3)
sage: h3[2,1].coproduct()
h3[] # h3[2, 1] + h3[1] # h3[1, 1] + h3[1] # h3[2] + h3[1, 1] # h3[1] +
→h3[2] # h3[1] + h3[2, 1] # h3[]
sage: ks3t = SymmetricFunctions(FractionField(QQ['t'])).kschur(3)
sage: ks3t[2,1].coproduct()
ks3[] # ks3[2, 1] + ks3[1] # ks3[1, 1] + ks3[1] # ks3[2] + ks3[1, 1] #
→ks3[1] + ks3[2] # ks3[1] + ks3[2, 1] # ks3[]
sage: ks3t[3,1].coproduct()
ks3[] # ks3[3, 1] + ks3[1] # ks3[2, 1] + (t+1)*ks3[1] # ks3[3] + ks3[1,
→1] # ks3[2] + ks3[2] # ks3[1, 1]
+ (t+1)*ks3[2] # ks3[2] + ks3[2, 1] # ks3[1] + (t+1)*ks3[3] # ks3[1] +
→ks3[3, 1] # ks3[]
sage: h3.coproduct(h3[2,1])
h3[] # h3[2, 1] + h3[1] # h3[1, 1] + h3[1] # h3[2] + h3[1, 1] # h3[1] +
→h3[2] # h3[1] + h3[2, 1] # h3[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ks3 = Sym.kschur(Integer(3),Integer(1))
>>> ks3[Integer(2),Integer(1)].coproduct()
ks3[] # ks3[2, 1] + ks3[1] # ks3[1, 1] + ks3[1] # ks3[2] + ks3[1, 1] #
→ks3[1] + ks3[2] # ks3[1] + ks3[2, 1] # ks3[]
```

(continues on next page)

(continued from previous page)

```
>>> h3 = Sym.khomogeneous(Integer(3))
>>> h3[Integer(2),Integer(1)].coproduct()
h3[] # h3[2, 1] + h3[1] # h3[1, 1] + h3[1] # h3[2] + h3[1, 1] # h3[1] +
↪h3[2] # h3[1] + h3[2, 1] # h3[]
>>> ks3t = SymmetricFunctions(FractionField(QQ['t'])).kschur(Integer(3))
>>> ks3t[Integer(2),Integer(1)].coproduct()
ks3[] # ks3[2, 1] + ks3[1] # ks3[1, 1] + ks3[1] # ks3[2] + ks3[1, 1] #
↪ks3[1] + ks3[2] # ks3[1] + ks3[2, 1] # ks3[]
>>> ks3t[Integer(3),Integer(1)].coproduct()
ks3[] # ks3[3, 1] + ks3[1] # ks3[2, 1] + (t+1)*ks3[1] # ks3[3] + ks3[1,
↪1] # ks3[2] + ks3[2] # ks3[1, 1]
+ (t+1)*ks3[2] # ks3[2] + ks3[2, 1] # ks3[1] + (t+1)*ks3[3] # ks3[1] +
↪ks3[3, 1] # ks3[]
>>> h3.coproduct(h3[Integer(2),Integer(1)])
h3[] # h3[2, 1] + h3[1] # h3[1, 1] + h3[1] # h3[2] + h3[1, 1] # h3[1] +
↪h3[2] # h3[1] + h3[2, 1] # h3[]
```

counit(*element*)

Return the counit of `element`.

The counit is the constant term of `element`.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks3 = Sym.kschur(3,1)
sage: f = 2*ks3[2,1] + 3*ks3[[]]
sage: f.counit()
3
sage: ks3.counit(f)
3
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ks3 = Sym.kschur(Integer(3),Integer(1))
>>> f = Integer(2)*ks3[Integer(2),Integer(1)] + Integer(3)*ks3[[]]
>>> f.counit()
3
>>> ks3.counit(f)
3
```

degree_on_basis(*b*)

Return the degree of the basis element indexed by b.

INPUT:

- `b` – a partition

EXAMPLES:

```
sage: ks3 = SymmetricFunctions(QQ).kschur(3,1)
sage: ks3.degree_on_basis(Partition([3,2]))
5
```

```
>>> from sage.all import *
>>> ks3 = SymmetricFunctions(QQ).kschur(Integer(3),Integer(1))
>>> ks3.degree_on_basis(Partition([Integer(3),Integer(2)]))
5
```

one_basis()

Return the basis element indexing 1.

EXAMPLES:

```
sage: ks3 = SymmetricFunctions(QQ).kschur(3,1)
sage: ks3.one()  # indirect doctest
ks3[]
```

```
>>> from sage.all import *
>>> ks3 = SymmetricFunctions(QQ).kschur(Integer(3),Integer(1))
>>> ks3.one()  # indirect doctest
ks3[]
```

transition_matrix(*other*, *n*)

Return the degree n transition matrix between self and other.

INPUT:

- other – a basis in the ring of symmetric functions
- n – positive integer

The entry in the i-th row and j-th column is the coefficient obtained by writing the i-th element of the basis of self in terms of the basis other, and extracting the j-th coefficient.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ); s = Sym.schur()
sage: ks3 = Sym.kschur(3,1)
sage: ks3.transition_matrix(s,5)
[1 1 1 0 0 0 0]
[0 1 0 1 0 0 0]
[0 0 1 0 1 0 0]
[0 0 0 1 0 1 0]
[0 0 0 0 1 1 1]

sage: Sym = SymmetricFunctions(QQ['t'])
sage: s = Sym.schur()
sage: ks = Sym.kschur(3)
sage: ks.transition_matrix(s,5)
[t^2   t   1   0   0   0   0]
[  0   t   0   1   0   0   0]
[  0   0   t   0   1   0   0]
[  0   0   0   t   0   1   0]
[  0   0   0   0 t^2   t   1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ); s = Sym.schur()
>>> ks3 = Sym.kschur(Integer(3),Integer(1))
>>> ks3.transition_matrix(s,Integer(5))
[1 1 1 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 1 0 1 0 0 0]
[0 0 1 0 1 0 0]
[0 0 0 1 0 1 0]
[0 0 0 0 1 1 1]

>>> Sym = SymmetricFunctions(QQ['t'])
>>> s = Sym.schur()
>>> ks = Sym.kschur(Integer(3))
>>> ks.transition_matrix(s,Integer(5))
[t^2   t   1   0   0   0   0]
[  0   t   0   1   0   0   0]
[  0   0   t   0   1   0   0]
[  0   0   0   t   0   1   0]
[  0   0   0   0 t^2   t   1]
```

super_categories()

The super categories of `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: from sage.combinat.sf.new_kschur import KBoundedSubspaceBases
sage: KB = Sym.kBoundedSubspace(3)
sage: KBB = KBoundedSubspaceBases(KB); KBB
Category of k bounded subspace bases of 3-bounded Symmetric Functions over␣
↪Univariate Polynomial Ring in t over Rational Field
sage: KBB.super_categories()
[Category of realizations of 3-bounded Symmetric Functions over Univariate␣
↪Polynomial Ring in t over Rational Field,
 Join of Category of graded coalgebras with basis over Univariate Polynomial␣
↪Ring in t over Rational Field
     and Category of subobjects of filtered modules with basis over␣
↪Univariate Polynomial Ring in t over Rational Field]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'])
>>> from sage.combinat.sf.new_kschur import KBoundedSubspaceBases
>>> KB = Sym.kBoundedSubspace(Integer(3))
>>> KBB = KBoundedSubspaceBases(KB); KBB
Category of k bounded subspace bases of 3-bounded Symmetric Functions over␣
↪Univariate Polynomial Ring in t over Rational Field
>>> KBB.super_categories()
[Category of realizations of 3-bounded Symmetric Functions over Univariate␣
↪Polynomial Ring in t over Rational Field,
 Join of Category of graded coalgebras with basis over Univariate Polynomial␣
↪Ring in t over Rational Field
     and Category of subobjects of filtered modules with basis over␣
↪Univariate Polynomial Ring in t over Rational Field]
```

class sage.combinat.sf.new_kschur.**K_kSchur**(*kBoundedRing*)

Bases: *CombinatorialFreeModule*

This class implements the basis of the k-bounded subspace called the K-k-Schur basis.

See [Morse2011], [LamSchillingShimozono2010].

REFERENCES:

K_k_Schur_non_commutative_variables(*la*)

Return the K-k-Schur function, as embedded inside the affine zero Hecke algebra.

INPUT:

- `la` – a k-bounded Partition

OUTPUT: an element of the affine zero Hecke algebra

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: g.K_k_Schur_non_commutative_variables([2,1])
T[3,1,0] + T[1,2,0] + T[3,2,0] + T[0,1,0] + T[2,0,1] + T[0,3,0] + T[2,0,3] +␣
→T[0,3,1] + T[2,3,2] + T[2,3,1] + T[3,1,2] + T[1,2,1] - T[2,0] - T[3,1]
sage: g.K_k_Schur_non_commutative_variables([])
1
sage: g.K_k_Schur_non_commutative_variables([4,1])
Traceback (most recent call last):
...
ValueError: Partition should be 3-bounded
```

```
>>> from sage.all import *
>>> g = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).K_
→kschur()
>>> g.K_k_Schur_non_commutative_variables([Integer(2),Integer(1)])
T[3,1,0] + T[1,2,0] + T[3,2,0] + T[0,1,0] + T[2,0,1] + T[0,3,0] + T[2,0,3] +␣
→T[0,3,1] + T[2,3,2] + T[2,3,1] + T[3,1,2] + T[1,2,1] - T[2,0] - T[3,1]
>>> g.K_k_Schur_non_commutative_variables([])
1
>>> g.K_k_Schur_non_commutative_variables([Integer(4),Integer(1)])
Traceback (most recent call last):
...
ValueError: Partition should be 3-bounded
```

homogeneous_basis_noncommutative_variables_zero_Hecke(*la*)

Return the homogeneous basis element indexed by `la`, viewed as an element inside the affine zero Hecke algebra. For the code, see method _homogeneous_basis.

INPUT:

- `la` – a k-bounded partition

OUTPUT: an element of the affine zero Hecke algebra

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: g.homogeneous_basis_noncommutative_variables_zero_Hecke([2,1])
T[2,1,0] + T[3,1,0] + T[1,2,0] + T[3,2,0] + T[0,1,0] + T[2,0,1] + T[1,0,3] +␣
→T[0,3,0] + T[2,0,3] + T[0,3,2] + T[0,3,1] + T[2,3,2] + T[3,2,1] + T[2,3,1]␣
→+ T[3,1,2] + T[1,2,1] - T[1,0] - 2*T[2,0] - T[0,3] - T[3,2] - 2*T[3,1] -␣
→T[2,1]
```

(continues on next page)

(continued from previous page)

```
sage: g.homogeneous_basis_noncommutative_variables_zero_Hecke([])
1
```

```
>>> from sage.all import *
>>> g = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).K_
↪kschur()
>>> g.homogeneous_basis_noncommutative_variables_zero_Hecke([Integer(2),
↪Integer(1)])
T[2,1,0] + T[3,1,0] + T[1,2,0] + T[3,2,0] + T[0,1,0] + T[2,0,1] + T[1,0,3] +␣
↪T[0,3,0] + T[2,0,3] + T[0,3,2] + T[0,3,1] + T[2,3,2] + T[3,2,1] + T[2,3,1]␣
↪+ T[3,1,2] + T[1,2,1] - T[1,0] - 2*T[2,0] - T[0,3] - T[3,2] - 2*T[3,1] -␣
↪T[2,1]
>>> g.homogeneous_basis_noncommutative_variables_zero_Hecke([])
1
```

lift (x)

Return the lift of a k-bounded symmetric function.

INPUT:

- **x – an expression in the K-k-Schur basis. Equivalently, x can be a** k-bounded partition (then x corresponds to the basis element indexed by x)

OUTPUT: a symmetric function

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: g.lift([2,1])
h[2] + h[2, 1] - h[3]
sage: g.lift([])
h[]
sage: g.lift([4,1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪bounded Symmetric Functions over Rational Field with t=1 in the K-3-Schur␣
↪basis)
```

```
>>> from sage.all import *
>>> g = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).K_
↪kschur()
>>> g.lift([Integer(2),Integer(1)])
h[2] + h[2, 1] - h[3]
>>> g.lift([])
h[]
>>> g.lift([Integer(4),Integer(1)])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-
↪bounded Symmetric Functions over Rational Field with t=1 in the K-3-Schur␣
↪basis)
```

product (x, y)

Return the product of the two K-k-Schur functions.

INPUT:

- x, y – elements of the k-bounded subspace, in the K-k-Schur basis

OUTPUT: an element of the k-bounded subspace, in the K-k-Schur basis

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: g.product(g([2,1]), g[1])
-2*Kks3[2, 1] + Kks3[2, 1, 1] + Kks3[2, 2]
sage: g.product(g([2,1]), g([]))
Kks3[2, 1]
```

```
>>> from sage.all import *
>>> g = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).K_
↪kschur()
>>> g.product(g([Integer(2),Integer(1)]), g[Integer(1)])
-2*Kks3[2, 1] + Kks3[2, 1, 1] + Kks3[2, 2]
>>> g.product(g([Integer(2),Integer(1)]), g([]))
Kks3[2, 1]
```

retract(*x*)

Return the retract of a symmetric function.

INPUT:

- x – a symmetric function

OUTPUT: a k-bounded symmetric function in the K-k-Schur basis

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: m = SymmetricFunctions(QQ).m()
sage: g.retract(m[2,1])
-2*Kks3[1] + 4*Kks3[1, 1] - 2*Kks3[1, 1, 1] - Kks3[2] + Kks3[2, 1]
sage: g.retract(m([]))
Kks3[]
```

```
>>> from sage.all import *
>>> g = SymmetricFunctions(QQ).kBoundedSubspace(Integer(3),Integer(1)).K_
↪kschur()
>>> m = SymmetricFunctions(QQ).m()
>>> g.retract(m[Integer(2),Integer(1)])
-2*Kks3[1] + 4*Kks3[1, 1] - 2*Kks3[1, 1, 1] - Kks3[2] + Kks3[2, 1]
>>> g.retract(m([]))
Kks3[]
```

class sage.combinat.sf.new_kschur.**kHomogeneous**(*kBoundedRing*)

Bases: *CombinatorialFreeModule*

Space of k-bounded homogeneous symmetric functions.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: kH = Sym.khomogeneous(3)
sage: kH[2]
h3[2]
sage: kH[2].lift()
h[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> kH = Sym.khomogeneous(Integer(3))
>>> kH[Integer(2)]
h3[2]
>>> kH[Integer(2)].lift()
h[2]
```

class sage.combinat.sf.new_kschur.**kSchur**(*kBoundedRing*)

Bases: *CombinatorialFreeModule*

Space of k-Schur functions.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: KB = Sym.kBoundedSubspace(3); KB
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational
↪Field
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'])
>>> KB = Sym.kBoundedSubspace(Integer(3)); KB
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational
↪Field
```

The k-Schur function basis can be constructed as follows:

```
sage: ks3 = KB.kschur(); ks3
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational
↪Field in the 3-Schur basis
```

```
>>> from sage.all import *
>>> ks3 = KB.kschur(); ks3
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational
↪Field in the 3-Schur basis
```

We can convert to any basis of the ring of symmetric functions and, whenever it makes sense, also the other way round:

```
sage: s = Sym.schur()
sage: s(ks3([3,2,1]))
s[3, 2, 1] + t*s[4, 1, 1] + t*s[4, 2] + t^2*s[5, 1]
sage: t = Sym.base_ring().gen()
sage: ks3(s([3, 2, 1]) + t*s([4, 1, 1]) + t*s([4, 2]) + t^2*s([5, 1]))
ks3[3, 2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: s(ks3[2, 1, 1])
s[2, 1, 1] + t*s[3, 1]
sage: ks3(s[2, 1, 1] + t*s[3, 1])
ks3[2, 1, 1]
```

```
>>> from sage.all import *
>>> s = Sym.schur()
>>> s(ks3([Integer(3),Integer(2),Integer(1)]))
s[3, 2, 1] + t*s[4, 1, 1] + t*s[4, 2] + t^2*s[5, 1]
>>> t = Sym.base_ring().gen()
>>> ks3(s([Integer(3), Integer(2), Integer(1)]) + t*s([Integer(4), Integer(1),␣
↪Integer(1)]) + t*s([Integer(4), Integer(2)]) + t**Integer(2)*s([Integer(5),␣
↪Integer(1)]))
ks3[3, 2, 1]
>>> s(ks3[Integer(2), Integer(1), Integer(1)])
s[2, 1, 1] + t*s[3, 1]
>>> ks3(s[Integer(2), Integer(1), Integer(1)] + t*s[Integer(3), Integer(1)])
ks3[2, 1, 1]
```

k-Schur functions are indexed by partitions with first part $\leq k$. Constructing a k-Schur function for a larger partition raises an error:

```
sage: ks3([4,3,2,1]) #
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 3, 2, 1]) an element of self (=3-
↪bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational␣
↪Field in the 3-Schur basis)
```

```
>>> from sage.all import *
>>> ks3([Integer(4),Integer(3),Integer(2),Integer(1)]) #
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 3, 2, 1]) an element of self (=3-
↪bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational␣
↪Field in the 3-Schur basis)
```

Similarly, attempting to convert a function that is not in the linear span of the k-Schur functions raises an error:

```
sage: ks3(s([4]))
Traceback (most recent call last):
...
ValueError: s[4] is not in the image
```

```
>>> from sage.all import *
>>> ks3(s([Integer(4)]))
Traceback (most recent call last):
...
ValueError: s[4] is not in the image
```

Note that the product of k-Schur functions is not guaranteed to be in the space spanned by the k-Schurs. In general, we only have that a k-Schur times a j-Schur function is in the $(k+j)$-bounded subspace. The multiplication of two

k-Schur functions thus generally returns the product of the lift of the functions to the ambient symmetric function space. If the result happens to lie in the k-bounded subspace, then the result is cast into the k-Schur basis:

```
sage: ks2 = Sym.kBoundedSubspace(2).kschur()
sage: ks2[1] * ks2[1]
ks2[1, 1] + ks2[2]
sage: ks2[1] * ks2[2]
s[2, 1] + s[3]
```

```
>>> from sage.all import *
>>> ks2 = Sym.kBoundedSubspace(Integer(2)).kschur()
>>> ks2[Integer(1)] * ks2[Integer(1)]
ks2[1, 1] + ks2[2]
>>> ks2[Integer(1)] * ks2[Integer(2)]
s[2, 1] + s[3]
```

Because the target space of the product of a k-Schur and a j-Schur has several possibilities, the product of a k-Schur and j-Schur function is not implemented for distinct k and j. Let us show how to get around this 'manually':

```
sage: ks3 = Sym.kBoundedSubspace(3).kschur()
sage: ks2([2,1]) * ks3([3,1])
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: '2-bounded Symmetric Functions␣
↪over Univariate Polynomial Ring in t over Rational Field in the 2-Schur basis'␣
↪and '3-bounded Symmetric Functions over Univariate Polynomial Ring in t over␣
↪Rational Field in the 3-Schur basis'
```

```
>>> from sage.all import *
>>> ks3 = Sym.kBoundedSubspace(Integer(3)).kschur()
>>> ks2([Integer(2),Integer(1)]) * ks3([Integer(3),Integer(1)])
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: '2-bounded Symmetric Functions␣
↪over Univariate Polynomial Ring in t over Rational Field in the 2-Schur basis'␣
↪and '3-bounded Symmetric Functions over Univariate Polynomial Ring in t over␣
↪Rational Field in the 3-Schur basis'
```

The workaround:

```
sage: f = s(ks2([2,1])) * s(ks3([3,1])); f # Convert to Schur functions first and␣
↪multiply there.
s[3, 2, 1, 1] + s[3, 2, 2] + (t+1)*s[3, 3, 1] + s[4, 1, 1, 1]
+ (2*t+2)*s[4, 2, 1] + (t^2+t+1)*s[4, 3] + (2*t+1)*s[5, 1, 1]
+ (t^2+2*t+1)*s[5, 2] + (t^2+2*t)*s[6, 1] + t^2*s[7]
```

```
>>> from sage.all import *
>>> f = s(ks2([Integer(2),Integer(1)])) * s(ks3([Integer(3),Integer(1)])); f #␣
↪Convert to Schur functions first and multiply there.
s[3, 2, 1, 1] + s[3, 2, 2] + (t+1)*s[3, 3, 1] + s[4, 1, 1, 1]
+ (2*t+2)*s[4, 2, 1] + (t^2+t+1)*s[4, 3] + (2*t+1)*s[5, 1, 1]
+ (t^2+2*t+1)*s[5, 2] + (t^2+2*t)*s[6, 1] + t^2*s[7]
```

or:

```
sage: f = ks2[2,1].lift() * ks3[3,1].lift()
sage: ks5 = Sym.kBoundedSubspace(5).kschur()
sage: ks5(f) # The product of a 'ks2' with a 'ks3' is a 'ks5'.
ks5[3, 2, 1, 1] + ks5[3, 2, 2] + (t+1)*ks5[3, 3, 1] + ks5[4, 1, 1, 1]
+ (t+2)*ks5[4, 2, 1] + (t^2+t+1)*ks5[4, 3] + (t+1)*ks5[5, 1, 1] + ks5[5, 2]
```

```
>>> from sage.all import *
>>> f = ks2[Integer(2),Integer(1)].lift() * ks3[Integer(3),Integer(1)].lift()
>>> ks5 = Sym.kBoundedSubspace(Integer(5)).kschur()
>>> ks5(f) # The product of a 'ks2' with a 'ks3' is a 'ks5'.
ks5[3, 2, 1, 1] + ks5[3, 2, 2] + (t+1)*ks5[3, 3, 1] + ks5[4, 1, 1, 1]
+ (t+2)*ks5[4, 2, 1] + (t^2+t+1)*ks5[4, 3] + (t+1)*ks5[5, 1, 1] + ks5[5, 2]
```

For other technical reasons, taking powers of k-Schur functions is not implemented, even when the answer is still in the k-bounded subspace:

```
sage: ks2([1])^2
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for ^: '2-bounded Symmetric Functions␣
↪over Univariate Polynomial Ring in t over Rational Field in the 2-Schur basis'␣
↪and 'Integer Ring'
```

```
>>> from sage.all import *
>>> ks2([Integer(1)])**Integer(2)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for ^: '2-bounded Symmetric Functions␣
↪over Univariate Polynomial Ring in t over Rational Field in the 2-Schur basis'␣
↪and 'Integer Ring'
```

Todo

Get rid of said technical "reasons".

However, at $t = 1$, the product of k-Schur functions is in the span of the k-Schur functions always. Below are some examples at $t = 1$

```
sage: ks3 = Sym.kBoundedSubspace(3, t=1).kschur(); ks3
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational␣
↪Field with t=1 in the 3-Schur basis
sage: s = SymmetricFunctions(ks3.base_ring()).schur()
sage: ks3(s([3]))
ks3[3]
sage: s(ks3([3,2,1]))
s[3, 2, 1] + s[4, 1, 1] + s[4, 2] + s[5, 1]
sage: ks3([2,1])^2    # taking powers works for t=1
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
```

```
>>> from sage.all import *
>>> ks3 = Sym.kBoundedSubspace(Integer(3), t=Integer(1)).kschur(); ks3
```

(continues on next page)

(continued from previous page)

```
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational␣
↪Field with t=1 in the 3-Schur basis
>>> s = SymmetricFunctions(ks3.base_ring()).schur()
>>> ks3(s([Integer(3)]))
ks3[3]
>>> s(ks3([Integer(3),Integer(2),Integer(1)]))
s[3, 2, 1] + s[4, 1, 1] + s[4, 2] + s[5, 1]
>>> ks3([Integer(2),Integer(1)])**Integer(2)     # taking powers works for t=1
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
```

product_on_basis(*left*, *right*)

Take the product of two k-Schur functions.

If $t \neq 1$, then take the product by lifting to the Schur functions and then retracting back into the k-bounded subspace (if possible).

If $t = 1$, then the product calls `_product_on_basis_via_rectangles()`.

INPUT:

- `left`, `right` – partitions

OUTPUT: an element of the k-Schur functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks3 = Sym.kschur(3,1)
sage: kH = Sym.khomogeneous(3)
sage: ks3(kH[2,1,1])
ks3[2, 1, 1] + ks3[2, 2] + ks3[3, 1]
sage: ks3([])*kH[2,1,1]
ks3[2, 1, 1] + ks3[2, 2] + ks3[3, 1]
sage: ks3([3,3,3,2,2,1,1,1])^2
ks3[3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1]
sage: ks3([3,3,3,2,2,1,1,1])*ks3([2,2,2,2,2,1,1,1,1])
ks3[3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1]
sage: ks3([2,2,1,1,1,1])*ks3([2,2,2,1,1,1,1])
ks3[2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1] + ks3[2, 2, 2, 2, 2, 2, 1, 1, 1, 1,
↪ 1, 1]
sage: ks3[2,1]^2
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
sage: ks3 = Sym.kschur(3)
sage: ks3[2,1]*ks3[2,1]
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1,␣
↪1] + s[4, 2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'])
>>> ks3 = Sym.kschur(Integer(3),Integer(1))
>>> kH = Sym.khomogeneous(Integer(3))
>>> ks3(kH[Integer(2),Integer(1),Integer(1)])
ks3[2, 1, 1] + ks3[2, 2] + ks3[3, 1]
>>> ks3([])*kH[Integer(2),Integer(1),Integer(1)]
ks3[2, 1, 1] + ks3[2, 2] + ks3[3, 1]
```

(continues on next page)

(continued from previous page)

```
>>> ks3([Integer(3),Integer(3),Integer(3),Integer(2),Integer(2),Integer(1),
↪Integer(1),Integer(1)])**Integer(2)
ks3[3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1]
>>> ks3([Integer(3),Integer(3),Integer(3),Integer(2),Integer(2),Integer(1),
↪Integer(1),Integer(1)])*ks3([Integer(2),Integer(2),Integer(2),Integer(2),
↪Integer(2),Integer(1),Integer(1),Integer(1),Integer(1)])
ks3[3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1]
>>> ks3([Integer(2),Integer(2),Integer(1),Integer(1),Integer(1),
↪Integer(1)])*ks3([Integer(2),Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(1),Integer(1)])
ks3[2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1] + ks3[2, 2, 2, 2, 2, 2, 1, 1, 1, 1,
↪ 1, 1]
>>> ks3[Integer(2),Integer(1)]**Integer(2)
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
>>> ks3 = Sym.kschur(Integer(3))
>>> ks3[Integer(2),Integer(1)]*ks3[Integer(2),Integer(1)]
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1,␣
↪1] + s[4, 2]
```

class sage.combinat.sf.new_kschur.**kSplit**(*kBoundedRing*)

Bases: *CombinatorialFreeModule*

The k-split basis of the space of k-bounded-symmetric functions.

Fix `k` a positive integer and `t` an element of the base ring.

The k-split functions are a basis for the space of k-bounded symmetric functions that also have the bases

$$\{Q'_\lambda[X;t]\}_{\lambda_1\le k} = \{s_\lambda^{(k)}[X;t]\}_{\lambda_1\le k}$$

where $Q'_\lambda[X;t]$ are the Hall-Littlewood symmetric functions (using the notation of [MAC]) and $s_\lambda^{(k)}[X;t]$ are the k-Schur functions. If t is not a root of unity, then

$$\{s_\lambda[X/(1-t)]\}_{\lambda_1\le k}$$

is also a basis of this space.

The k-split basis has the property that $Q'_\lambda[X;t]$ expands positively in the k-split basis and the k-split basis conjecturally expands positively in the k-Schur functions. See [LLMSSZ] p. 81.

The k-split basis is defined recursively using the Hall-Littlewood creation operator defined in [SZ2001]. If a partition `la` is the concatenation (as lists) of a partition `mu` and `nu` where `mu` has maximal hook length equal to `k` then `ksp(la) = ksp(nu).hl_creation_operator(mu)`. If the hook length of `la` is less than or equal to `k`, then `ksp(la)` is equal to the Schur function indexed by `la`.

EXAMPLES:

```
sage: Symt = SymmetricFunctions(QQ['t'].fraction_field())
sage: kBS3 = Symt.kBoundedSubspace(3)
sage: ks3 = kBS3.kschur()
sage: ksp3 = kBS3.ksplit()
sage: ks3(ksp3[2,1,1])
ks3[2, 1, 1] + t*ks3[2, 2]
sage: ksp3(ks3[2,1,1])
ksp3[2, 1, 1] - t*ksp3[2, 2]
```

(continues on next page)

(continued from previous page)

```
sage: ksp3[2,1]*ksp3[1]
s[2, 1, 1] + s[2, 2] + s[3, 1]
sage: ksp3[2,1].hl_creation_operator([1])
t*ksp3[2, 1, 1] + (-t^2+t)*ksp3[2, 2]

sage: Qp = Symt.hall_littlewood().Qp()
sage: ksp3(Qp[3,2,1])
ksp3[3, 2, 1] + t*ksp3[3, 3]

sage: kBS4 = Symt.kBoundedSubspace(4)
sage: ksp4 = kBS4.ksplit()
sage: ksp4(ksp3([3,2,1]))
ksp4[3, 2, 1] - t*ksp4[3, 3] + t*ksp4[4, 1, 1]
sage: ks4 = kBS4.kschur()
sage: ks4(ksp4[3,2,2,1])
ks4[3, 2, 2, 1] + t*ks4[3, 3, 1, 1] + t*ks4[3, 3, 2]
```

```
>>> from sage.all import *
>>> Symt = SymmetricFunctions(QQ['t'].fraction_field())
>>> kBS3 = Symt.kBoundedSubspace(Integer(3))
>>> ks3 = kBS3.kschur()
>>> ksp3 = kBS3.ksplit()
>>> ks3(ksp3[Integer(2),Integer(1),Integer(1)])
ks3[2, 1, 1] + t*ks3[2, 2]
>>> ksp3(ks3[Integer(2),Integer(1),Integer(1)])
ksp3[2, 1, 1] - t*ksp3[2, 2]
>>> ksp3[Integer(2),Integer(1)]*ksp3[Integer(1)]
s[2, 1, 1] + s[2, 2] + s[3, 1]
>>> ksp3[Integer(2),Integer(1)].hl_creation_operator([Integer(1)])
t*ksp3[2, 1, 1] + (-t^2+t)*ksp3[2, 2]

>>> Qp = Symt.hall_littlewood().Qp()
>>> ksp3(Qp[Integer(3),Integer(2),Integer(1)])
ksp3[3, 2, 1] + t*ksp3[3, 3]

>>> kBS4 = Symt.kBoundedSubspace(Integer(4))
>>> ksp4 = kBS4.ksplit()
>>> ksp4(ksp3([Integer(3),Integer(2),Integer(1)]))
ksp4[3, 2, 1] - t*ksp4[3, 3] + t*ksp4[4, 1, 1]
>>> ks4 = kBS4.kschur()
>>> ks4(ksp4[Integer(3),Integer(2),Integer(2),Integer(1)])
ks4[3, 2, 2, 1] + t*ks4[3, 3, 1, 1] + t*ks4[3, 3, 2]
```

5.1.296 Non-symmetric Macdonald Polynomials

class sage.combinat.sf.ns_macdonald.**AugmentedLatticeDiagramFilling**(*l*, *pi=None*)

Bases: *CombinatorialObject*

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a == loads(dumps(a))
```

(continues on next page)

(continued from previous page)

```
True
sage: pi = Permutation([2,3,1]).to_permutation_group_element()
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]],pi)
sage: a == loads(dumps(a))
True
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a == loads(dumps(a))
True
>>> pi = Permutation([Integer(2),Integer(3),Integer(1)]).to_permutation_group_
↪element()
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]],pi)
>>> a == loads(dumps(a))
True
```

are_attacking(*i*, *j*, *ii*, *jj*)

Return `True` if the boxes `(i,j)` and `(ii,jj)` in `self` are attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: all( a.are_attacking(i,j,ii,jj) for (i,j),(ii,jj) in a.attacking_
↪boxes())
True
sage: a.are_attacking(1,1,3,2)
False
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> all( a.are_attacking(i,j,ii,jj) for (i,j),(ii,jj) in a.attacking_boxes())
True
>>> a.are_attacking(Integer(1),Integer(1),Integer(3),Integer(2))
False
```

attacking_boxes()

Return a list of pairs of boxes in `self` that are attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.attacking_boxes()[:5]
[((1, 1), (2, 1)),
 ((1, 1), (3, 1)),
 ((1, 1), (6, 1)),
 ((1, 1), (2, 0)),
 ((1, 1), (3, 0))]
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.attacking_boxes()[:Integer(5)]
[((1, 1), (2, 1)),
 ((1, 1), (3, 1)),
 ((1, 1), (6, 1)),
 ((1, 1), (2, 0)),
 ((1, 1), (3, 0))]
```

boxes()

Return a list of the coordinates of the boxes of `self`, including the 'basement row'.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.boxes()
[(1, 1),
 (1, 2),
 (2, 1),
 (3, 1),
 (3, 2),
 (3, 3),
 (6, 1),
 (6, 2),
 (1, 0),
 (2, 0),
 (3, 0),
 (4, 0),
 (5, 0),
 (6, 0)]
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.boxes()
[(1, 1),
 (1, 2),
 (2, 1),
 (3, 1),
 (3, 2),
 (3, 3),
 (6, 1),
 (6, 2),
 (1, 0),
 (2, 0),
 (3, 0),
 (4, 0),
 (5, 0),
 (6, 0)]
```

coeff(q, t)

Return the coefficient in front of `self` in the HHL formula for the expansion of the non-symmetric Macdonald polynomial E(self.shape()).

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: q,t = var('q,t')                                                  #␣
↪needs sage.symbolic
sage: a.coeff(q,t)                                                      #␣
↪needs sage.symbolic
(t - 1)^4/((q^2*t^3 - 1)^2*(q*t^2 - 1)^2)
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> q,t = var('q,t')                                                    #␣
↪needs sage.symbolic
>>> a.coeff(q,t)                                                        #␣
↪needs sage.symbolic
(t - 1)^4/((q^2*t^3 - 1)^2*(q*t^2 - 1)^2)
```

coeff_integral(*q*, *t*)

Return the coefficient in front of `self` in the HHL formula for the expansion of the integral non-symmetric Macdonald polynomial E(self.shape())

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: q,t = var('q,t')                                                  #␣
↪needs sage.symbolic
sage: a.coeff_integral(q,t)                                             #␣
↪needs sage.symbolic
(q^2*t^3 - 1)^2*(q*t^2 - 1)^2*(t - 1)^4
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> q,t = var('q,t')                                                    #␣
↪needs sage.symbolic
>>> a.coeff_integral(q,t)                                               #␣
↪needs sage.symbolic
(q^2*t^3 - 1)^2*(q*t^2 - 1)^2*(t - 1)^4
```

coinv()

Return `self`'s co-inversion statistic.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.coinv()
2
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.coinv()
2
```

descents()

Return a list of the descents of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.descents()
[(1, 2), (3, 2)]
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.descents()
[(1, 2), (3, 2)]
```

inv()

Return `self`'s inversion statistic.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.inv()
15
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.inv()
15
```

inversions()

Return a list of the inversions of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.inversions()[:5]
[((6, 2), (3, 2)),
 ((1, 2), (6, 1)),
 ((1, 2), (3, 1)),
 ((1, 2), (2, 1)),
 ((6, 1), (3, 1))]
sage: len(a.inversions())
25
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.inversions()[:Integer(5)]
[((6, 2), (3, 2)),
 ((1, 2), (6, 1)),
 ((1, 2), (3, 1)),
 ((1, 2), (2, 1)),
 ((6, 1), (3, 1))]
```

(continues on next page)

(continued from previous page)

```
>>> len(a.inversions())
25
```

is_non_attacking()

Return `True` if `self` is non-attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.is_non_attacking()
True
sage: a = AugmentedLatticeDiagramFilling([[1, 1, 1], [2, 3], [3]])
sage: a.is_non_attacking()
False
sage: a = AugmentedLatticeDiagramFilling([[2,2],[1]])
sage: a.is_non_attacking()
False
sage: pi = Permutation([2,1]).to_permutation_group_element()
sage: a = AugmentedLatticeDiagramFilling([[2,2],[1]],pi)
sage: a.is_non_attacking()
True
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.is_non_attacking()
True
>>> a = AugmentedLatticeDiagramFilling([[Integer(1), Integer(1), Integer(1)],␣
↪[Integer(2), Integer(3)], [Integer(3)]])
>>> a.is_non_attacking()
False
>>> a = AugmentedLatticeDiagramFilling([[Integer(2),Integer(2)],[Integer(1)]])
>>> a.is_non_attacking()
False
>>> pi = Permutation([Integer(2),Integer(1)]).to_permutation_group_element()
>>> a = AugmentedLatticeDiagramFilling([[Integer(2),Integer(2)],[Integer(1)]],
↪pi)
>>> a.is_non_attacking()
True
```

maj()

Return the major index of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.maj()
3
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
```

(continues on next page)

(continued from previous page)

```
>>> a.maj()
3
```

permuted_filling(*sigma*)

EXAMPLES:

```
sage: pi=Permutation([2,1,4,3]).to_permutation_group_element()
sage: fill=[[2],[1,2,3],[],[3,1]]
sage: AugmentedLatticeDiagramFilling(fill).permuted_filling(pi)
[[2, 1], [1, 2, 1, 4], [4], [3, 4, 2]]
```

```
>>> from sage.all import *
>>> pi=Permutation([Integer(2),Integer(1),Integer(4),Integer(3)]).to_
↪permutation_group_element()
>>> fill=[[Integer(2)],[Integer(1),Integer(2),Integer(3)],[],[Integer(3),
↪Integer(1)]]
>>> AugmentedLatticeDiagramFilling(fill).permuted_filling(pi)
[[2, 1], [1, 2, 1, 4], [4], [3, 4, 2]]
```

reading_order()

Return a list of coordinates of the boxes in `self`, starting from the top right, and reading from right to left.

Note that this includes the 'basement row' of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.reading_order()
[(3, 3),
 (6, 2),
 (3, 2),
 (1, 2),
 (6, 1),
 (3, 1),
 (2, 1),
 (1, 1),
 (6, 0),
 (5, 0),
 (4, 0),
 (3, 0),
 (2, 0),
 (1, 0)]
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.reading_order()
[(3, 3),
 (6, 2),
 (3, 2),
 (1, 2),
 (6, 1),
```

(continues on next page)

(continued from previous page)

```
 (3, 1),
 (2, 1),
 (1, 1),
 (6, 0),
 (5, 0),
 (4, 0),
 (3, 0),
 (2, 0),
 (1, 0)]
```

reading_word()

Return the reading word of `self`, obtained by reading the boxes entries of `self` from right to left, starting in the upper right.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.reading_word()
word: 25465321
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.reading_word()
word: 25465321
```

shape()

Return the shape of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.shape()
[2, 1, 3, 0, 0, 2]
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.shape()
[2, 1, 3, 0, 0, 2]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.weight()
[1, 2, 1, 1, 2, 1]
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
```

(continues on next page)

(continued from previous page)

```
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a.weight()
[1, 2, 1, 1, 2, 1]
```

sage.combinat.sf.ns_macdonald.**E**(*mu*, *q=None*, *t=None*, *pi=None*)

Return the non-symmetric Macdonald polynomial in type A corresponding to a shape `mu`, with basement permuted according to `pi`.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- J. Haglund, M. Haiman, N. Loehr. *A combinatorial formula for non-symmetric Macdonald polynomials*. arXiv math/0601693v3.

See also

NonSymmetricMacdonaldPolynomials for a type free implementation where the polynomials are constructed recursively by the application of intertwining operators.

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import E
sage: E([0,0,0])
1
sage: E([1,0,0])
x0
sage: E([0,1,0])
(t - 1)/(q*t^2 - 1)*x0 + x1
sage: E([0,0,1])
(t - 1)/(q*t - 1)*x0 + (t - 1)/(q*t - 1)*x1 + x2
sage: E([1,1,0])
x0*x1
sage: E([1,0,1])
(t - 1)/(q*t^2 - 1)*x0*x1 + x0*x2
sage: E([0,1,1])
(t - 1)/(q*t - 1)*x0*x1 + (t - 1)/(q*t - 1)*x0*x2 + x1*x2
sage: E([2,0,0])
x0^2 + (q*t - q)/(q*t - 1)*x0*x1 + (q*t - q)/(q*t - 1)*x0*x2
sage: E([0,2,0])
(t - 1)/(q^2*t^2 - 1)*x0^2 + (q^2*t^3 - q^2*t^2 + q*t^2 - 2*q*t + q - t + 1)/(q^
↪3*t^3 - q^2*t^2 - q*t + 1)*x0*x1 + x1^2 + (q*t^2 - 2*q*t + q)/(q^3*t^3 - q^2*t^
↪2 - q*t + 1)*x0*x2 + (q*t - q)/(q*t - 1)*x1*x2
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.ns_macdonald import E
>>> E([Integer(0),Integer(0),Integer(0)])
1
>>> E([Integer(1),Integer(0),Integer(0)])
x0
>>> E([Integer(0),Integer(1),Integer(0)])
(t - 1)/(q*t^2 - 1)*x0 + x1
```

(continues on next page)

(continued from previous page)

```
>>> E([Integer(0),Integer(0),Integer(1)])
(t - 1)/(q*t - 1)*x0 + (t - 1)/(q*t - 1)*x1 + x2
>>> E([Integer(1),Integer(1),Integer(0)])
x0*x1
>>> E([Integer(1),Integer(0),Integer(1)])
(t - 1)/(q*t^2 - 1)*x0*x1 + x0*x2
>>> E([Integer(0),Integer(1),Integer(1)])
(t - 1)/(q*t - 1)*x0*x1 + (t - 1)/(q*t - 1)*x0*x2 + x1*x2
>>> E([Integer(2),Integer(0),Integer(0)])
x0^2 + (q*t - q)/(q*t - 1)*x0*x1 + (q*t - q)/(q*t - 1)*x0*x2
>>> E([Integer(0),Integer(2),Integer(0)])
(t - 1)/(q^2*t^2 - 1)*x0^2 + (q^2*t^3 - q^2*t^2 + q*t^2 - 2*q*t + q - t + 1)/(q^
↪3*t^3 - q^2*t^2 - q*t + 1)*x0*x1 + x1^2 + (q*t^2 - 2*q*t + q)/(q^3*t^3 - q^2*t^
↪2 - q*t + 1)*x0*x2 + (q*t - q)/(q*t - 1)*x1*x2
```

sage.combinat.sf.ns_macdonald.**E_integral**(*mu*, *q=None*, *t=None*, *pi=None*)

Return the integral form for the non-symmetric Macdonald polynomial in type A corresponding to a shape mu.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- J. Haglund, M. Haiman, N. Loehr. *A combinatorial formula for non-symmetric Macdonald polynomials*. arXiv math/0601693v3.

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import E_integral
sage: E_integral([0,0,0])
1
sage: E_integral([1,0,0])
(-t + 1)*x0
sage: E_integral([0,1,0])
(-q*t^2 + 1)*x0 + (-t + 1)*x1
sage: E_integral([0,0,1])
(-q*t + 1)*x0 + (-q*t + 1)*x1 + (-t + 1)*x2
sage: E_integral([1,1,0])
(t^2 - 2*t + 1)*x0*x1
sage: E_integral([1,0,1])
(q*t^3 - q*t^2 - t + 1)*x0*x1 + (t^2 - 2*t + 1)*x0*x2
sage: E_integral([0,1,1])
(q^2*t^3 + q*t^4 - q*t^3 - q*t^2 - q*t - t^2 + t + 1)*x0*x1 + (q*t^2 - q*t - t +␣
↪1)*x0*x2 + (t^2 - 2*t + 1)*x1*x2
sage: E_integral([2,0,0])
(t^2 - 2*t + 1)*x0^2 + (q^2*t^2 - q^2*t - q*t + q)*x0*x1 + (q^2*t^2 - q^2*t - q*t␣
↪+ q)*x0*x2
sage: E_integral([0,2,0])
(q^2*t^3 - q^2*t^2 - t + 1)*x0^2 + (q^4*t^3 - q^3*t^2 - q^2*t + q*t^2 - q*t + q -␣
↪t + 1)*x0*x1 + (t^2 - 2*t + 1)*x1^2 + (q^4*t^3 - q^3*t^2 - q^2*t + q)*x0*x2 +␣
↪(q^2*t^2 - q^2*t - q*t + q)*x1*x2
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.ns_macdonald import E_integral
>>> E_integral([Integer(0),Integer(0),Integer(0)])
```

(continues on next page)

(continued from previous page)

```
1
>>> E_integral([Integer(1),Integer(0),Integer(0)])
(-t + 1)*x0
>>> E_integral([Integer(0),Integer(1),Integer(0)])
(-q*t^2 + 1)*x0 + (-t + 1)*x1
>>> E_integral([Integer(0),Integer(0),Integer(1)])
(-q*t + 1)*x0 + (-q*t + 1)*x1 + (-t + 1)*x2
>>> E_integral([Integer(1),Integer(1),Integer(0)])
(t^2 - 2*t + 1)*x0*x1
>>> E_integral([Integer(1),Integer(0),Integer(1)])
(q*t^3 - q*t^2 - t + 1)*x0*x1 + (t^2 - 2*t + 1)*x0*x2
>>> E_integral([Integer(0),Integer(1),Integer(1)])
(q^2*t^3 + q*t^4 - q*t^3 - q*t^2 - q*t - t^2 + t + 1)*x0*x1 + (q*t^2 - q*t - t +
↪1)*x0*x2 + (t^2 - 2*t + 1)*x1*x2
>>> E_integral([Integer(2),Integer(0),Integer(0)])
(t^2 - 2*t + 1)*x0^2 + (q^2*t^2 - q^2*t - q*t + q)*x0*x1 + (q^2*t^2 - q^2*t - q*t
↪+ q)*x0*x2
>>> E_integral([Integer(0),Integer(2),Integer(0)])
(q^2*t^3 - q^2*t^2 - t + 1)*x0^2 + (q^4*t^3 - q^3*t^2 - q^2*t + q*t^2 - q*t + q -
↪t + 1)*x0*x1 + (t^2 - 2*t + 1)*x1^2 + (q^4*t^3 - q^3*t^2 - q^2*t + q)*x0*x2 +
↪(q^2*t^2 - q^2*t - q*t + q)*x1*x2
```

`sage.combinat.sf.ns_macdonald.`**`Ht`**`(`*mu*, *q=None*, *t=None*, *pi=None*`)`

Return the symmetric Macdonald polynomial using the Haiman, Haglund, and Loehr formula.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- J. Haglund, M. Haiman, N. Loehr. *A combinatorial formula for non-symmetric Macdonald polynomials*. arXiv math/0601693v3.

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import Ht
sage: HHt = SymmetricFunctions(QQ['q','t'].fraction_field()).macdonald().Ht()
sage: Ht([0,0,1])
x0 + x1 + x2
sage: HHt([1]).expand(3)
x0 + x1 + x2
sage: Ht([0,0,2])
x0^2 + (q + 1)*x0*x1 + x1^2 + (q + 1)*x0*x2 + (q + 1)*x1*x2 + x2^2
sage: HHt([2]).expand(3)
x0^2 + (q + 1)*x0*x1 + x1^2 + (q + 1)*x0*x2 + (q + 1)*x1*x2 + x2^2
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.ns_macdonald import Ht
>>> HHt = SymmetricFunctions(QQ['q','t'].fraction_field()).macdonald().Ht()
>>> Ht([Integer(0),Integer(0),Integer(1)])
x0 + x1 + x2
>>> HHt([Integer(1)]).expand(Integer(3))
x0 + x1 + x2
>>> Ht([Integer(0),Integer(0),Integer(2)])
x0^2 + (q + 1)*x0*x1 + x1^2 + (q + 1)*x0*x2 + (q + 1)*x1*x2 + x2^2
```

(continues on next page)

(continued from previous page)

```
>>> HHt([Integer(2)]).expand(Integer(3))
x0^2 + (q + 1)*x0*x1 + x1^2 + (q + 1)*x0*x2 + (q + 1)*x1*x2 + x2^2
```

class sage.combinat.sf.ns_macdonald.**LatticeDiagram**(*l*, *copy=True*)

Bases: `CombinatorialObject`

a(*i*, *j*)

Return the length of the arm of the box `(i,j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.a(5,2)
3
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(1),Integer(2),Integer(4),
↪Integer(3),Integer(0),Integer(4),Integer(2),Integer(3)])
>>> a.a(Integer(5),Integer(2))
3
```

arm(*i*, *j*)

Return the arm of the box `(i,j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm(5,2)
[(1, 2), (3, 2), (8, 1)]
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(1),Integer(2),Integer(4),
↪Integer(3),Integer(0),Integer(4),Integer(2),Integer(3)])
>>> a.arm(Integer(5),Integer(2))
[(1, 2), (3, 2), (8, 1)]
```

arm_left(*i*, *j*)

Return the left arm of the box `(i,j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm_left(5,2)
[(1, 2), (3, 2)]
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(1),Integer(2),Integer(4),
↪Integer(3),Integer(0),Integer(4),Integer(2),Integer(3)])
>>> a.arm_left(Integer(5),Integer(2))
[(1, 2), (3, 2)]
```

arm_right(*i*, *j*)

Return the right arm of the box `(i,j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm_right(5,2)
[(8, 1)]
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(1),Integer(2),Integer(4),
↪Integer(3),Integer(0),Integer(4),Integer(2),Integer(3)])
>>> a.arm_right(Integer(5),Integer(2))
[(8, 1)]
```

boxes()

EXAMPLES:

```
sage: a = LatticeDiagram([3,0,2])
sage: a.boxes()
[(1, 1), (1, 2), (1, 3), (3, 1), (3, 2)]
sage: a = LatticeDiagram([2, 1, 3, 0, 0, 2])
sage: a.boxes()
[(1, 1), (1, 2), (2, 1), (3, 1), (3, 2), (3, 3), (6, 1), (6, 2)]
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(0),Integer(2)])
>>> a.boxes()
[(1, 1), (1, 2), (1, 3), (3, 1), (3, 2)]
>>> a = LatticeDiagram([Integer(2), Integer(1), Integer(3), Integer(0),␣
↪Integer(0), Integer(2)])
>>> a.boxes()
[(1, 1), (1, 2), (2, 1), (3, 1), (3, 2), (3, 3), (6, 1), (6, 2)]
```

boxes_same_and_lower_right(*ii*, *jj*)

Return an iterator of the boxes of `self` that are in row `jj` but not identical with `(ii, jj)`, or lie in the row `jj - 1` (the row directly below `jj`; this might be the basement) and strictly to the right of `(ii, jj)`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a = a.shape()
sage: list(a.boxes_same_and_lower_right(1,1))
[(2, 1), (3, 1), (6, 1), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0)]
sage: list(a.boxes_same_and_lower_right(1,2))
[(3, 2), (6, 2), (2, 1), (3, 1), (6, 1)]
sage: list(a.boxes_same_and_lower_right(3,3))
[(6, 2)]
sage: list(a.boxes_same_and_lower_right(2,3))
[(3, 3), (3, 2), (6, 2)]
```

```
>>> from sage.all import *
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> a = a.shape()
>>> list(a.boxes_same_and_lower_right(Integer(1),Integer(1)))
```

(continues on next page)

(continued from previous page)

```
[(2, 1), (3, 1), (6, 1), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0)]
>>> list(a.boxes_same_and_lower_right(Integer(1),Integer(2)))
[(3, 2), (6, 2), (2, 1), (3, 1), (6, 1)]
>>> list(a.boxes_same_and_lower_right(Integer(3),Integer(3)))
[(6, 2)]
>>> list(a.boxes_same_and_lower_right(Integer(2),Integer(3)))
[(3, 3), (3, 2), (6, 2)]
```

flip()

Return the flip of `self`, where flip is defined as follows. Let `r = max(self)`. Then `self.flip()[i] = r - self[i]`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,0,2])
sage: a.flip()
[0, 3, 1]
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(0),Integer(2)])
>>> a.flip()
[0, 3, 1]
```

l(*i*, *j*)

Return `self[i] - j`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.l(5,2)
1
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(1),Integer(2),Integer(4),
↪Integer(3),Integer(0),Integer(4),Integer(2),Integer(3)])
>>> a.l(Integer(5),Integer(2))
1
```

leg(*i*, *j*)

Return the leg of the box `(i,j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.leg(5,2)
[(5, 3)]
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(1),Integer(2),Integer(4),
↪Integer(3),Integer(0),Integer(4),Integer(2),Integer(3)])
>>> a.leg(Integer(5),Integer(2))
[(5, 3)]
```

size()

Return the number of boxes in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.size()
22
```

```
>>> from sage.all import *
>>> a = LatticeDiagram([Integer(3),Integer(1),Integer(2),Integer(4),
↪Integer(3),Integer(0),Integer(4),Integer(2),Integer(3)])
>>> a.size()
22
```

class sage.combinat.sf.ns_macdonald.**NonattackingBacktracker**(*shape*, *pi=None*)

Bases: *GenericBacktracker*

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import NonattackingBacktracker
sage: n = NonattackingBacktracker(LatticeDiagram([0,1,2]))
sage: n._ending_position
(3, 2)
sage: n._initial_state
(2, 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.ns_macdonald import NonattackingBacktracker
>>> n = NonattackingBacktracker(LatticeDiagram([Integer(0),Integer(1),
↪Integer(2)]))
>>> n._ending_position
(3, 2)
>>> n._initial_state
(2, 1)
```

get_next_pos(*ii*, *jj*)

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import NonattackingBacktracker
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: n = NonattackingBacktracker(a.shape())
sage: n.get_next_pos(1, 1)
(2, 1)
sage: n.get_next_pos(6, 1)
(1, 2)
sage: n = NonattackingBacktracker(LatticeDiagram([2,2,2]))
sage: n.get_next_pos(3, 1)
(1, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.ns_macdonald import NonattackingBacktracker
>>> a = AugmentedLatticeDiagramFilling([[Integer(1),Integer(6)],[Integer(2)],
```

(continues on next page)

(continued from previous page)

```
↪[Integer(3),Integer(4),Integer(2)],[],[],[Integer(5),Integer(5)]])
>>> n = NonattackingBacktracker(a.shape())
>>> n.get_next_pos(Integer(1), Integer(1))
(2, 1)
>>> n.get_next_pos(Integer(6), Integer(1))
(1, 2)
>>> n = NonattackingBacktracker(LatticeDiagram([Integer(2),Integer(2),
↪Integer(2)]))
>>> n.get_next_pos(Integer(3), Integer(1))
(1, 2)
```

sage.combinat.sf.ns_macdonald.**NonattackingFillings**(*shape*, *pi=None*)

Returning the finite set of nonattacking fillings of a given shape.

EXAMPLES:

```
sage: NonattackingFillings([0,1,2])
Nonattacking fillings of [0, 1, 2]
sage: NonattackingFillings([0,1,2]).list()
[[[1], [2, 1], [3, 2, 1]],
 [[1], [2, 1], [3, 2, 2]],
 [[1], [2, 1], [3, 2, 3]],
 [[1], [2, 1], [3, 3, 1]],
 [[1], [2, 1], [3, 3, 2]],
 [[1], [2, 1], [3, 3, 3]],
 [[1], [2, 2], [3, 1, 1]],
 [[1], [2, 2], [3, 1, 2]],
 [[1], [2, 2], [3, 1, 3]],
 [[1], [2, 2], [3, 3, 1]],
 [[1], [2, 2], [3, 3, 2]],
 [[1], [2, 2], [3, 3, 3]]]
```

```
>>> from sage.all import *
>>> NonattackingFillings([Integer(0),Integer(1),Integer(2)])
Nonattacking fillings of [0, 1, 2]
>>> NonattackingFillings([Integer(0),Integer(1),Integer(2)]).list()
[[[1], [2, 1], [3, 2, 1]],
 [[1], [2, 1], [3, 2, 2]],
 [[1], [2, 1], [3, 2, 3]],
 [[1], [2, 1], [3, 3, 1]],
 [[1], [2, 1], [3, 3, 2]],
 [[1], [2, 1], [3, 3, 3]],
 [[1], [2, 2], [3, 1, 1]],
 [[1], [2, 2], [3, 1, 2]],
 [[1], [2, 2], [3, 1, 3]],
 [[1], [2, 2], [3, 3, 1]],
 [[1], [2, 2], [3, 3, 2]],
 [[1], [2, 2], [3, 3, 3]]]
```

class sage.combinat.sf.ns_macdonald.**NonattackingFillings_shape**(*shape*, *pi=None*)

Bases: Parent, UniqueRepresentation

EXAMPLES:

```
sage: n = NonattackingFillings([0,1,2])
sage: n == loads(dumps(n))
True
```

```
>>> from sage.all import *
>>> n = NonattackingFillings([Integer(0),Integer(1),Integer(2)])
>>> n == loads(dumps(n))
True
```

flip()

Return the nonattacking fillings of the flipped shape.

EXAMPLES:

```
sage: NonattackingFillings([0,1,2]).flip()
Nonattacking fillings of [2, 1, 0]
```

```
>>> from sage.all import *
>>> NonattackingFillings([Integer(0),Integer(1),Integer(2)]).flip()
Nonattacking fillings of [2, 1, 0]
```

5.1.297 Orthogonal Symmetric Functions

AUTHORS:

- Travis Scrimshaw (2013-11-10): Initial version

class sage.combinat.sf.orthogonal.**SymmetricFunctionAlgebra_orthogonal**(*Sym*)

Bases: *SymmetricFunctionAlgebra_generic*

The orthogonal symmetric function basis (or orthogonal basis, to be short).

The orthogonal basis $\{o_\lambda\}$ where λ is taken over all partitions is defined by the following change of basis with the Schur functions:

$$s_\lambda = \sum_\mu \left(\sum_{\nu \in H} c^\lambda_{\mu\nu} \right) o_\mu$$

where H is the set of all partitions with even-width rows and $c^\lambda_{\mu\nu}$ is the usual Littlewood-Richardson (LR) coefficients. By the properties of LR coefficients, this can be shown to be a upper unitriangular change of basis.

Note

This is only a filtered basis, not a $\mathbf{Z}$-graded basis. However this does respect the induced $(\mathbf{Z}/2\mathbf{Z})$-grading.

INPUT:

- `Sym` – an instance of the ring of the symmetric functions

REFERENCES:

- [ChariKleber2000]
- [KoikeTerada1987]
- [ShimozonoZabrocki2006]

EXAMPLES:

Here are the first few orthogonal symmetric functions, in various bases:

```
sage: Sym = SymmetricFunctions(QQ)
sage: o = Sym.o()
sage: e = Sym.e()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()

sage: p(o([1]))
p[1]
sage: m(o([1]))
m[1]
sage: e(o([1]))
e[1]
sage: h(o([1]))
h[1]
sage: s(o([1]))
s[1]

sage: p(o([2]))
-p[] + 1/2*p[1, 1] + 1/2*p[2]
sage: m(o([2]))
-m[] + m[1, 1] + m[2]
sage: e(o([2]))
-e[] + e[1, 1] - e[2]
sage: h(o([2]))
-h[] + h[2]
sage: s(o([2]))
-s[] + s[2]

sage: p(o([3]))
-p[1] + 1/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3]
sage: m(o([3]))
-m[1] + m[1, 1, 1] + m[2, 1] + m[3]
sage: e(o([3]))
-e[1] + e[1, 1, 1] - 2*e[2, 1] + e[3]
sage: h(o([3]))
-h[1] + h[3]
sage: s(o([3]))
-s[1] + s[3]

sage: Sym = SymmetricFunctions(ZZ)
sage: o = Sym.o()
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: p = Sym.p()
sage: m(o([4]))
-m[1, 1] + m[1, 1, 1, 1] - m[2] + m[2, 1, 1] + m[2, 2] + m[3, 1] + m[4]
```

(continues on next page)

(continued from previous page)

```
sage: e(o([4]))
-e[1, 1] + e[1, 1, 1, 1] + e[2] - 3*e[2, 1, 1] + e[2, 2] + 2*e[3, 1] - e[4]
sage: h(o([4]))
-h[2] + h[4]
sage: s(o([4]))
-s[2] + s[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> o = Sym.o()
>>> e = Sym.e()
>>> h = Sym.h()
>>> p = Sym.p()
>>> s = Sym.s()
>>> m = Sym.m()

>>> p(o([Integer(1)]))
p[1]
>>> m(o([Integer(1)]))
m[1]
>>> e(o([Integer(1)]))
e[1]
>>> h(o([Integer(1)]))
h[1]
>>> s(o([Integer(1)]))
s[1]

>>> p(o([Integer(2)]))
-p[] + 1/2*p[1, 1] + 1/2*p[2]
>>> m(o([Integer(2)]))
-m[] + m[1, 1] + m[2]
>>> e(o([Integer(2)]))
-e[] + e[1, 1] - e[2]
>>> h(o([Integer(2)]))
-h[] + h[2]
>>> s(o([Integer(2)]))
-s[] + s[2]

>>> p(o([Integer(3)]))
-p[1] + 1/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3]
>>> m(o([Integer(3)]))
-m[1] + m[1, 1, 1] + m[2, 1] + m[3]
>>> e(o([Integer(3)]))
-e[1] + e[1, 1, 1] - 2*e[2, 1] + e[3]
>>> h(o([Integer(3)]))
-h[1] + h[3]
>>> s(o([Integer(3)]))
-s[1] + s[3]

>>> Sym = SymmetricFunctions(ZZ)
>>> o = Sym.o()
>>> e = Sym.e()
```

(continues on next page)

(continued from previous page)

```
>>> h = Sym.h()
>>> s = Sym.s()
>>> m = Sym.m()
>>> p = Sym.p()
>>> m(o([Integer(4)]))
-m[1, 1] + m[1, 1, 1, 1] - m[2] + m[2, 1, 1] + m[2, 2] + m[3, 1] + m[4]
>>> e(o([Integer(4)]))
-e[1, 1] + e[1, 1, 1, 1] + e[2] - 3*e[2, 1, 1] + e[2, 2] + 2*e[3, 1] - e[4]
>>> h(o([Integer(4)]))
-h[2] + h[4]
>>> s(o([Integer(4)]))
-s[2] + s[4]
```

Some examples of conversions the other way:

```
sage: o(h[3])
o[1] + o[3]
sage: o(e[3])
o[1, 1, 1]
sage: o(m[2,1])
o[1] - 2*o[1, 1, 1] + o[2, 1]
sage: o(p[3])
o[1, 1, 1] - o[2, 1] + o[3]
```

```
>>> from sage.all import *
>>> o(h[Integer(3)])
o[1] + o[3]
>>> o(e[Integer(3)])
o[1, 1, 1]
>>> o(m[Integer(2),Integer(1)])
o[1] - 2*o[1, 1, 1] + o[2, 1]
>>> o(p[Integer(3)])
o[1, 1, 1] - o[2, 1] + o[3]
```

Some multiplication:

```
sage: o([2]) * o([1,1])
o[1, 1] + o[2] + o[2, 1, 1] + o[3, 1]
sage: o([2,1,1]) * o([2])
o[1, 1] + o[1, 1, 1, 1] + 2*o[2, 1, 1] + o[2, 2] + o[2, 2, 1, 1]
 + o[3, 1] + o[3, 1, 1, 1] + o[3, 2, 1] + o[4, 1, 1]
sage: o([1,1]) * o([2,1])
o[1] + o[1, 1, 1] + 2*o[2, 1] + o[2, 1, 1, 1] + o[2, 2, 1]
 + o[3] + o[3, 1, 1] + o[3, 2]
```

```
>>> from sage.all import *
>>> o([Integer(2)]) * o([Integer(1),Integer(1)])
o[1, 1] + o[2] + o[2, 1, 1] + o[3, 1]
>>> o([Integer(2),Integer(1),Integer(1)]) * o([Integer(2)])
o[1, 1] + o[1, 1, 1, 1] + 2*o[2, 1, 1] + o[2, 2] + o[2, 2, 1, 1]
 + o[3, 1] + o[3, 1, 1, 1] + o[3, 2, 1] + o[4, 1, 1]
>>> o([Integer(1),Integer(1)]) * o([Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
o[1] + o[1, 1, 1] + 2*o[2, 1] + o[2, 1, 1, 1] + o[2, 2, 1]
 + o[3] + o[3, 1, 1] + o[3, 2]
```

Examples of the Hopf algebra structure:

```
sage: o([1]).antipode()
-o[1]
sage: o([2]).antipode()
-o[] + o[1, 1]
sage: o([1]).coproduct()
o[] # o[1] + o[1] # o[]
sage: o([2]).coproduct()
o[] # o[] + o[] # o[2] + o[1] # o[1] + o[2] # o[]
sage: o([1]).counit()
0
sage: o.one().counit()
1
```

```
>>> from sage.all import *
>>> o([Integer(1)]).antipode()
-o[1]
>>> o([Integer(2)]).antipode()
-o[] + o[1, 1]
>>> o([Integer(1)]).coproduct()
o[] # o[1] + o[1] # o[]
>>> o([Integer(2)]).coproduct()
o[] # o[] + o[] # o[2] + o[1] # o[1] + o[2] # o[]
>>> o([Integer(1)]).counit()
0
>>> o.one().counit()
1
```

5.1.298 Symmetric functions defined by orthogonality and triangularity

One characterization of Schur functions is that they are upper triangularly related to the monomial symmetric functions and orthogonal with respect to the Hall scalar product. We can use the class SymmetricFunctionAlgebra_orthotriang to obtain the Schur functions from this definition.

```
sage: from sage.combinat.sf.sfa import zee
sage: from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_orthotriang
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur functions')
sage: s([2,1])^2
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] +
↪s[4, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import zee
>>> from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_orthotriang
>>> Sym = SymmetricFunctions(QQ)
```

(continues on next page)

(continued from previous page)

```
>>> m = Sym.m()
>>> s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur functions')
>>> s([Integer(2),Integer(1)])**Integer(2)
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] +
↪s[4, 2]
```

```
sage: s2 = SymmetricFunctions(QQ).s()
sage: s2([2,1])^2
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] +
↪s[4, 2]
```

```
>>> from sage.all import *
>>> s2 = SymmetricFunctions(QQ).s()
>>> s2([Integer(2),Integer(1)])**Integer(2)
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] +
↪s[4, 2]
```

class sage.combinat.sf.orthotriang.**OrthotriangBasisFunctor**(*basis*)

Bases: *SymmetricFunctionsFunctor*

A constructor for algebras of symmetric functions constructed by orthogonality and triangularity.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import zee
sage: from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_
↪orthotriang
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur')
sage: s.construction()
(SymmetricFunctionsFunctor[Schur], Rational Field)
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import zee
>>> from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_orthotriang
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.m()
>>> s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur')
>>> s.construction()
(SymmetricFunctionsFunctor[Schur], Rational Field)
```

class sage.combinat.sf.orthotriang.**SymmetricFunctionAlgebra_orthotriang**(*Sym*, *base*, *scalar*, *prefix*, *basis_name*, *leading_coeff*)

Bases: *SymmetricFunctionAlgebra_generic*

Initialization of the symmetric function algebra defined via orthotriangular rules.

INPUT:

- `self` – a basis determined by an orthotriangular definition
- `Sym` – ring of symmetric functions

- `base` – an instance of a basis of the ring of symmetric functions (e.g. the Schur functions)
- `scalar` – a function `zee` on partitions. The function `zee` determines the scalar product on the power sum basis with normalization $\langle p_\mu, p_\mu \rangle = \mathrm{zee}(\mu)$.
- `prefix` – the prefix used to display the basis
- `basis_name` – the name used for the basis

Note

The base ring is required to be a **Q**-algebra for this method to be usable, since the scalar product is defined by its values on the power sum basis.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import zee
sage: from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_
↪orthotriang
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur'); s
Symmetric Functions over Rational Field in the Schur basis
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import zee
>>> from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_orthotriang
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.m()
>>> s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur'); s
Symmetric Functions over Rational Field in the Schur basis
```

class Element

Bases: *SymmetricFunctionAlgebra_generic_Element*

construction()

Return a pair `(F, R)`, where `F` is a *SymmetricFunctionsFunctor* and R is a ring, such that `F(R)` returns `self`.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import zee
sage: from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_
↪orthotriang
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur')
sage: s.construction()
(SymmetricFunctionsFunctor[Schur], Rational Field)
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import zee
>>> from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_
↪orthotriang
```

(continues on next page)

(continued from previous page)

```
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.m()
>>> s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur')
>>> s.construction()
(SymmetricFunctionsFunctor[Schur], Rational Field)
```

product (*left*, *right*)

Return `left * right` by converting both to the base and then converting back to `self`.

INPUT:

- `self` – a basis determined by an orthotriangular definition
- `left`, `right` – elements in `self`

OUTPUT: the expansion of the product of `left` and `right` in the basis `self`

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import zee
sage: from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_
↪orthotriang
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur␣
↪functions')
sage: s([1])*s([2,1]) #indirect doctest
s[2, 1, 1] + s[2, 2] + s[3, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import zee
>>> from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_
↪orthotriang
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.m()
>>> s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur␣
↪functions')
>>> s([Integer(1)])*s([Integer(2),Integer(1)]) #indirect doctest
s[2, 1, 1] + s[2, 2] + s[3, 1]
```

5.1.299 Power sum symmetric functions

class sage.combinat.sf.powersum.**SymmetricFunctionAlgebra_power** (*Sym*)

Bases: *SymmetricFunctionAlgebra_multiplicative*

A class for methods associated to the power sum basis of the symmetric functions.

INPUT:

- `self` – the power sum basis of the symmetric functions
- `Sym` – an instance of the ring of symmetric functions

class Element

Bases: *Element*

adams_operation(**args*, ***kwds*)

Deprecated: Use `adams_operator()` instead. See Issue #36396 for details.

adams_operator(*n*)

Return the image of the symmetric function `self` under the n-th Adams operator.

The n-th Adams operator $\mathbf{f}_n$ is defined to be the map from the ring of symmetric functions to itself that sends every symmetric function $P(x_1, x_2, x_3, \ldots)$ to $P(x_1^n, x_2^n, x_3^n, \ldots)$. This operator $\mathbf{f}_n$ is a Hopf algebra endomorphism, and satisfies

$$\mathbf{f}_n m_{(\lambda_1,\lambda_2,\lambda_3,\ldots)} = m_{(n\lambda_1,n\lambda_2,n\lambda_3,\ldots)}$$

for every partition $(\lambda_1, \lambda_2, \lambda_3, \ldots)$ (where m means the monomial basis). Moreover, $\mathbf{f}_n(p_r) = p_{nr}$ for every positive integer r (where p_k denotes the k-th powersum symmetric function).

The n-th Adams operator is also called the n-th Frobenius endomorphism. It is not related to the Frobenius map which connects the ring of symmetric functions with the representation theory of the symmetric group.

The n-th Adams operator is the n-th Adams operator of the Λ-ring of symmetric functions over the integers.

The n-th Adams operator can also be described via plethysm: Every symmetric function P satisfies $\mathbf{f}_n(P) = p_n \circ P = P \circ p_n$, where p_n is the n-th powersum symmetric function, and $\circ$ denotes (outer) plethysm.

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Adams operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: p[3].adams_operator(2)
p[6]
sage: p[4,2,1].adams_operator(3)
p[12, 6, 3]
sage: p([]).adams_operator(4)
p[]
sage: p[3].adams_operator(1)
p[3]
sage: (p([3]) - p([2]) + p([])).adams_operator(3)
p[] - p[6] + p[9]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> p = Sym.p()
>>> p[Integer(3)].adams_operator(Integer(2))
p[6]
>>> p[Integer(4),Integer(2),Integer(1)].adams_operator(Integer(3))
p[12, 6, 3]
>>> p([]).adams_operator(Integer(4))
p[]
>>> p[Integer(3)].adams_operator(Integer(1))
```

(continues on next page)

(continued from previous page)

```
p[3]
>>> (p([Integer(3)]) - p([Integer(2)]) + p([])).adams_operator(Integer(3))
p[] - p[6] + p[9]
```

See also

plethysm()

eval_at_permutation_roots(*rho*)

Evaluate at eigenvalues of a permutation matrix.

Evaluate an element of the power sum basis at the eigenvalues of a permutation matrix with cycle structure ρ.

This function evaluates an element at the roots of unity

$$\Xi_{\rho_1}, \Xi_{\rho_2}, \ldots, \Xi_{\rho_\ell}$$

where

$$\Xi_m = 1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1}$$

and ζ_m is an m root of unity. These roots of unity represent the eigenvalues of permutation matrix with cycle structure ρ.

INPUT:

- `rho` – a partition or a list of nonnegative integers

OUTPUT: an element of the base ring

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p([3,3]).eval_at_permutation_roots([6])
0
sage: p([3,3]).eval_at_permutation_roots([3])
9
sage: p([3,3]).eval_at_permutation_roots([1])
1
sage: p([3,3]).eval_at_permutation_roots([3,3])
36
sage: p([3,3]).eval_at_permutation_roots([1,1,1,1,1])
25
sage: (p[1]+p[2]+p[3]).eval_at_permutation_roots([3,2])
5
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> p([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(6)])
0
>>> p([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(3)])
9
>>> p([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(1)])
1
```

(continues on next page)

(continued from previous page)

```
>>> p([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(3),
↪Integer(3)])
36
>>> p([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1)])
25
>>> (p[Integer(1)]+p[Integer(2)]+p[Integer(3)]).eval_at_permutation_
↪roots([Integer(3),Integer(2)])
5
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: a = p([2])
sage: a.expand(2)
x0^2 + x1^2
sage: a.expand(3, alphabet=['a','b','c'])
a^2 + b^2 + c^2
sage: p([2,1,1]).expand(2)
x0^4 + 2*x0^3*x1 + 2*x0^2*x1^2 + 2*x0*x1^3 + x1^4
sage: p([7]).expand(4)
x0^7 + x1^7 + x2^7 + x3^7
sage: p([7]).expand(4,alphabet='t')
t0^7 + t1^7 + t2^7 + t3^7
sage: p([7]).expand(4,alphabet='x,y,z,t')
x^7 + y^7 + z^7 + t^7
sage: p(1).expand(4)
1
sage: p(0).expand(4)
0
sage: (p([]) + 2*p([1])).expand(3)
2*x0 + 2*x1 + 2*x2 + 1
sage: p([1]).expand(0)
0
sage: (3*p([])).expand(0)
3
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> a = p([Integer(2)])
>>> a.expand(Integer(2))
x0^2 + x1^2
>>> a.expand(Integer(3), alphabet=['a','b','c'])
```

(continues on next page)

(continued from previous page)

```
a^2 + b^2 + c^2
>>> p([Integer(2),Integer(1),Integer(1)]).expand(Integer(2))
x0^4 + 2*x0^3*x1 + 2*x0^2*x1^2 + 2*x0*x1^3 + x1^4
>>> p([Integer(7)]).expand(Integer(4))
x0^7 + x1^7 + x2^7 + x3^7
>>> p([Integer(7)]).expand(Integer(4),alphabet='t')
t0^7 + t1^7 + t2^7 + t3^7
>>> p([Integer(7)]).expand(Integer(4),alphabet='x,y,z,t')
x^7 + y^7 + z^7 + t^7
>>> p(Integer(1)).expand(Integer(4))
1
>>> p(Integer(0)).expand(Integer(4))
0
>>> (p([]) + Integer(2)*p([Integer(1)])).expand(Integer(3))
2*x0 + 2*x1 + 2*x2 + 1
>>> p([Integer(1)]).expand(Integer(0))
0
>>> (Integer(3)*p([])).expand(Integer(0))
3
```

exponential_specialization(*t=None*, *q=1*)

Return the exponential specialization of a symmetric function (when $q = 1$), or the q-exponential specialization (when $q \neq 1$).

The *exponential specialization* ex at t is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R. It is defined whenever the base ring K is a **Q**-algebra and t is an element of R. The easiest way to define it is by specifying its values on the powersum symmetric functions to be $p_1 = t$ and $p_n = 0$ for $n > 1$. Equivalently, on the homogeneous functions it is given by $ex(h_n) = t^n/n!$; see Proposition 7.8.4 of [EnumComb2].

By analogy, the q-exponential specialization is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R that depends on two elements t and q of R for which the elements $1 - q^i$ for all positive integers i are invertible. It can be defined by specifying its values on the complete homogeneous symmetric functions to be

$$ex_q(h_n) = t^n/[n]_q!,$$

where $[n]_q!$ is the q-factorial. Equivalently, for $q \neq 1$ and a homogeneous symmetric function f of degree n, we have

$$ex_q(f) = (1-q)^n t^n ps_q(f),$$

where $ps_q(f)$ is the stable principal specialization of f (see `principal_specialization()`). (See (7.29) in [EnumComb2].)

The limit of ex_q as $q \to 1$ is ex.

INPUT:

- `t` – (default: `None`) the value to use for t; the default is to create a ring of polynomials in `t`
- `q` – (default: 1) the value to use for q. If `q` is `None`, then a ring (or fraction field) of polynomials in `q` is created.

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: x = p[8,7,3,1]
```

(continues on next page)

(continued from previous page)

```
sage: x.exponential_specialization()
0
sage: x = p[3] + 5*p[1,1] + 2*p[1] + 1
sage: x.exponential_specialization(t=var("t"))                          #␣
↪needs sage.symbolic
5*t^2 + 2*t + 1
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> x = p[Integer(8),Integer(7),Integer(3),Integer(1)]
>>> x.exponential_specialization()
0
>>> x = p[Integer(3)] + Integer(5)*p[Integer(1),Integer(1)] +␣
↪Integer(2)*p[Integer(1)] + Integer(1)
>>> x.exponential_specialization(t=var("t"))                          #␣
↪needs sage.symbolic
5*t^2 + 2*t + 1
```

We also support the q-exponential_specialization:

```
sage: factor(p[3].exponential_specialization(q=var("q"), t=var("t")))    #␣
↪needs sage.symbolic
(q - 1)^2*t^3/(q^2 + q + 1)
```

```
>>> from sage.all import *
>>> factor(p[Integer(3)].exponential_specialization(q=var("q"), t=var("t
↪")))    # needs sage.symbolic
(q - 1)^2*t^3/(q^2 + q + 1)
```

frobenius(**args*, ***kwds*)

Deprecated: Use `adams_operator()` instead. See Issue #36396 for details.

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)}p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT: the image of `self` under the omega automorphism

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: a = p([2,1]); a
p[2, 1]
sage: a.omega()
-p[2, 1]
sage: p([]).omega()
p[]
sage: p(0).omega()
0
sage: p = SymmetricFunctions(ZZ).p()
sage: (p([3,1,1]) - 2 * p([2,1])).omega()
2*p[2, 1] + p[3, 1, 1]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> a = p([Integer(2),Integer(1)]); a
p[2, 1]
>>> a.omega()
-p[2, 1]
>>> p([]).omega()
p[]
>>> p(Integer(0)).omega()
0
>>> p = SymmetricFunctions(ZZ).p()
>>> (p([Integer(3),Integer(1),Integer(1)]) - Integer(2) * p([Integer(2),
↪Integer(1)])).omega()
2*p[2, 1] + p[3, 1, 1]
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT: the image of `self` under the omega automorphism

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: a = p([2,1]); a
p[2, 1]
sage: a.omega()
```

(continues on next page)

(continued from previous page)

```
-p[2, 1]
sage: p([]).omega()
p[]
sage: p(0).omega()
0
sage: p = SymmetricFunctions(ZZ).p()
sage: (p([3,1,1]) - 2 * p([2,1])).omega()
2*p[2, 1] + p[3, 1, 1]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> a = p([Integer(2),Integer(1)]); a
p[2, 1]
>>> a.omega()
-p[2, 1]
>>> p([]).omega()
p[]
>>> p(Integer(0)).omega()
0
>>> p = SymmetricFunctions(ZZ).p()
>>> (p([Integer(3),Integer(1),Integer(1)]) - Integer(2) * p([Integer(2),
↪Integer(1)])).omega()
2*p[2, 1] + p[3, 1, 1]
```

principal_specialization(*n=+Infinity*, *q=None*)

Return the principal specialization of a symmetric function.

The *principal specialization* of order n at q is the ring homomorphism $ps_{n,q}$ from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for $i \in \{1,\dots,n\}$ and $x_i \mapsto 0$ for $i > n$. Here, q is a given element of R, and we assume that the variables of our symmetric functions are $x_1, x_2, x_3, \dots$. (To be more precise, $ps_{n,q}$ is a K-algebra homomorphism, where K is the base ring.) See Section 7.8 of [EnumComb2].

The *stable principal specialization* at q is the ring homomorphism ps_q from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for all i. This is well-defined only if the resulting infinite sums converge; thus, in particular, setting $q = 1$ in the stable principal specialization is an invalid operation.

INPUT:

- `n` – (default: `infinity`) a nonnegative integer or `infinity`, specifying whether to compute the principal specialization of order `n` or the stable principal specialization.
- `q` – (default: `None`) the value to use for q; the default is to create a ring of polynomials in `q` (or a field of rational functions in `q`) over the given coefficient ring.

We use the formulas from Proposition 7.8.3 of [EnumComb2]:

$$ps_{n,q}(p_\lambda) = \prod_i (1 - q^{n\lambda_i})/(1 - q^{\lambda_i}),$$

$$ps_{n,1}(p_\lambda) = n^{\ell(\lambda)},$$

$$ps_q(p_\lambda) = 1/\prod_i (1 - q^{\lambda_i}),$$

where $\ell(\lambda)$ denotes the length of λ, and where the products range from $i = 1$ to $i = \ell(\lambda)$.

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: x = p[8,7,3,1]
sage: x.principal_specialization(3, q=var("q"))                          #␣
↪needs sage.symbolic
(q^24 - 1)*(q^21 - 1)*(q^9 - 1)/((q^8 - 1)*(q^7 - 1)*(q - 1))

sage: x = 5*p[1,1,1] + 3*p[2,1] + 1
sage: x.principal_specialization(3, q=var("q"))                          #␣
↪needs sage.symbolic
5*(q^3 - 1)^3/(q - 1)^3 + 3*(q^6 - 1)*(q^3 - 1)/((q^2 - 1)*(q - 1)) + 1
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> x = p[Integer(8),Integer(7),Integer(3),Integer(1)]
>>> x.principal_specialization(Integer(3), q=var("q"))                   ␣
↪     # needs sage.symbolic
(q^24 - 1)*(q^21 - 1)*(q^9 - 1)/((q^8 - 1)*(q^7 - 1)*(q - 1))

>>> x = Integer(5)*p[Integer(1),Integer(1),Integer(1)] +␣
↪Integer(3)*p[Integer(2),Integer(1)] + Integer(1)
>>> x.principal_specialization(Integer(3), q=var("q"))                   ␣
↪     # needs sage.symbolic
5*(q^3 - 1)^3/(q - 1)^3 + 3*(q^6 - 1)*(q^3 - 1)/((q^2 - 1)*(q - 1)) + 1
```

By default, we return a rational function in q:

```
sage: x.principal_specialization(3)
8*q^6 + 18*q^5 + 36*q^4 + 38*q^3 + 36*q^2 + 18*q + 9
```

```
>>> from sage.all import *
>>> x.principal_specialization(Integer(3))
8*q^6 + 18*q^5 + 36*q^4 + 38*q^3 + 36*q^2 + 18*q + 9
```

If n is not given we return the stable principal specialization:

```
sage: x.principal_specialization(q=var("q"))                             #␣
↪needs sage.symbolic
3/((q^2 - 1)*(q - 1)) - 5/(q - 1)^3 + 1
```

```
>>> from sage.all import *
>>> x.principal_specialization(q=var("q"))                             #␣
↪needs sage.symbolic
3/((q^2 - 1)*(q - 1)) - 5/(q - 1)^3 + 1
```

scalar(*x*, *zee=None*)

Return the standard scalar product of self and x.

INPUT:

- x – a power sum symmetric function
- zee – (default: uses standard zee function) optional input specifying the scalar product on the power sum basis with normalization $\langle p_\mu, p_\mu \rangle = \mathrm{zee}(\mu)$. zee should be a function on partitions.

Note that the power-sum symmetric functions are orthogonal under this scalar product. With the default value of zee, the value of $\langle p_\lambda, p_\lambda \rangle$ is given by the size of the centralizer in S_n of a permutation of cycle type λ.

OUTPUT:

- the standard scalar product between `self` and `x`, or, if the optional parameter `zee` is specified, then the scalar product with respect to the normalization $\langle p_\mu, p_\mu \rangle = \mathrm{zee}(\mu)$ with the power sum basis elements being orthogonal

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p4 = Partitions(4)
sage: matrix([ [p(a).scalar(p(b)) for a in p4] for b in p4])
[ 4  0  0  0  0]
[ 0  3  0  0  0]
[ 0  0  8  0  0]
[ 0  0  0  4  0]
[ 0  0  0  0 24]
sage: p(0).scalar(p(1))
0
sage: p(1).scalar(p(2))
2

sage: zee = lambda x : 1
sage: matrix( [[p[la].scalar(p[mu], zee) for la in Partitions(3)] for mu␣
↪in Partitions(3)])
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> p4 = Partitions(Integer(4))
>>> matrix([ [p(a).scalar(p(b)) for a in p4] for b in p4])
[ 4  0  0  0  0]
[ 0  3  0  0  0]
[ 0  0  8  0  0]
[ 0  0  0  4  0]
[ 0  0  0  0 24]
>>> p(Integer(0)).scalar(p(Integer(1)))
0
>>> p(Integer(1)).scalar(p(Integer(2)))
2

>>> zee = lambda x : Integer(1)
>>> matrix( [[p[la].scalar(p[mu], zee) for la in Partitions(Integer(3))]␣
↪for mu in Partitions(Integer(3))])
[1 0 0]
[0 1 0]
[0 0 1]
```

verschiebung(n)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n}e_{r/n}$$

(where h is the complete homogeneous basis, p is the powersum basis, and e is the elementary basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for "shift") endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Adams operator (see `adams_operator()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley's [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, \ldots, \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, \ldots, \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: p[3].verschiebung(2)
0
sage: p[4].verschiebung(4)
4*p[1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> p = Sym.p()
>>> p[Integer(3)].verschiebung(Integer(2))
0
>>> p[Integer(4)].verschiebung(Integer(4))
4*p[1]
```

The Verschiebung endomorphisms are multiplicative:

```
sage: all( all( p(lam).verschiebung(2) * p(mu).verschiebung(2)
....:           == (p(lam) * p(mu)).verschiebung(2)
....:           for mu in Partitions(4) )
```

(continues on next page)

(continued from previous page)

```
....:       for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( p(lam).verschiebung(Integer(2)) * p(mu).
↪verschiebung(Integer(2))
...             == (p(lam) * p(mu)).verschiebung(Integer(2))
...             for mu in Partitions(Integer(4)) )
...       for lam in Partitions(Integer(4)) )
True
```

Testing the adjointness between the Adams operators $\mathbf{f}_n$ and the Verschiebung operators $\mathbf{V}_n$:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: all( all( p(lam).verschiebung(2).scalar(p(mu))
....:             == p(lam).scalar(p(mu).adams_operator(2))
....:             for mu in Partitions(2) )
....:       for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> p = Sym.p()
>>> all( all( p(lam).verschiebung(Integer(2)).scalar(p(mu))
...             == p(lam).scalar(p(mu).adams_operator(Integer(2)))
...             for mu in Partitions(Integer(2)) )
...       for lam in Partitions(Integer(4)) )
True
```

antipode_on_basis(*partition*)

Return the antipode of `self[partition]`.

The antipode on the generator p_i (for $i > 0$) is $-p_i$, and the antipode on p_μ is $(-1)^{length(\mu)} p_\mu$.

INPUT:

- `self` – the power sum basis of the symmetric functions
- `partition` – a partition

OUTPUT:

- the result of the antipode on `self(partition)`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: p.antipode_on_basis([2])
-p[2]
sage: p.antipode_on_basis([3])
-p[3]
sage: p.antipode_on_basis([2,2])
p[2, 2]
```

(continues on next page)

(continued from previous page)

```
sage: p.antipode_on_basis([])
p[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> p = Sym.p()
>>> p.antipode_on_basis([Integer(2)])
-p[2]
>>> p.antipode_on_basis([Integer(3)])
-p[3]
>>> p.antipode_on_basis([Integer(2),Integer(2)])
p[2, 2]
>>> p.antipode_on_basis([])
p[]
```

bottom_schur_function(*partition*, *degree=None*)

Return the least-degree component of `s[partition]`, where `s` denotes the Schur basis of the symmetric functions, and the grading is not the usual grading on the symmetric functions but rather the grading which gives every p_i degree 1.

This least-degree component has its degree equal to the Frobenius rank of `partition`, while the degree with respect to the usual grading is still the size of `partition`.

This method requires the base ring to be a (commutative) **Q**-algebra. This restriction is unavoidable, since the least-degree component (in general) has noninteger coefficients in all classical bases of the symmetric functions.

The optional keyword `degree` allows taking any homogeneous component rather than merely the least-degree one. Specifically, if `degree` is set, then the `degree`-th component will be returned.

REFERENCES:

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: p.bottom_schur_function([2,2,1])
-1/6*p[3, 2] + 1/4*p[4, 1]
sage: p.bottom_schur_function([2,1])
-1/3*p[3]
sage: p.bottom_schur_function([3])
1/3*p[3]
sage: p.bottom_schur_function([1,1,1])
1/3*p[3]
sage: p.bottom_schur_function(Partition([1,1,1]))
1/3*p[3]
sage: p.bottom_schur_function([2,1], degree=1)
-1/3*p[3]
sage: p.bottom_schur_function([2,1], degree=2)
0
sage: p.bottom_schur_function([2,1], degree=3)
1/3*p[1, 1, 1]
sage: p.bottom_schur_function([2,2,1], degree=3)
1/8*p[2, 2, 1] - 1/6*p[3, 1, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> p = Sym.p()
>>> p.bottom_schur_function([Integer(2),Integer(2),Integer(1)])
-1/6*p[3, 2] + 1/4*p[4, 1]
>>> p.bottom_schur_function([Integer(2),Integer(1)])
-1/3*p[3]
>>> p.bottom_schur_function([Integer(3)])
1/3*p[3]
>>> p.bottom_schur_function([Integer(1),Integer(1),Integer(1)])
1/3*p[3]
>>> p.bottom_schur_function(Partition([Integer(1),Integer(1),Integer(1)]))
1/3*p[3]
>>> p.bottom_schur_function([Integer(2),Integer(1)], degree=Integer(1))
-1/3*p[3]
>>> p.bottom_schur_function([Integer(2),Integer(1)], degree=Integer(2))
0
>>> p.bottom_schur_function([Integer(2),Integer(1)], degree=Integer(3))
1/3*p[1, 1, 1]
>>> p.bottom_schur_function([Integer(2),Integer(2),Integer(1)],␣
↪degree=Integer(3))
1/8*p[2, 2, 1] - 1/6*p[3, 1, 1]
```

coproduct_on_generators(*i*)

Return coproduct on generators for power sums p_i (for $i > 0$).

The elements p_i are primitive elements.

INPUT:

- `self` – the power sum basis of the symmetric functions
- `i` – positive integer

OUTPUT: the result of the coproduct on the generator $p(i)$

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.powersum()
sage: p.coproduct_on_generators(2)
p[] # p[2] + p[2] # p[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> p = Sym.powersum()
>>> p.coproduct_on_generators(Integer(2))
p[] # p[2] + p[2] # p[]
```

eval_at_permutation_roots_on_generators(*k*, *rho*)

Evaluate p_k at eigenvalues of permutation matrix.

This function evaluates a symmetric function `p([k])` at the eigenvalues of a permutation matrix with cycle structure `\rho`.

This function evaluates a p_k at the roots of unity

$$\Xi_{\rho_1}, \Xi_{\rho_2}, \ldots, \Xi_{\rho_\ell}$$

where

$$\Xi_m = 1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1}$$

and ζ_m is an m root of unity. This is characterized by $p_k[A,B] = p_k[A] + p_k[B]$ and $p_k[\Xi_m] = 0$ unless m divides k and $p_{rm}[\Xi_m] = m$.

INPUT:

- `k` – nonnegative integer
- `rho` – a partition or a list of nonnegative integers

OUTPUT: an element of the base ring

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p.eval_at_permutation_roots_on_generators(3, [6])
0
sage: p.eval_at_permutation_roots_on_generators(3, [3])
3
sage: p.eval_at_permutation_roots_on_generators(3, [1])
1
sage: p.eval_at_permutation_roots_on_generators(3, [3,3])
6
sage: p.eval_at_permutation_roots_on_generators(3, [1,1,1,1,1])
5
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> p.eval_at_permutation_roots_on_generators(Integer(3), [Integer(6)])
0
>>> p.eval_at_permutation_roots_on_generators(Integer(3), [Integer(3)])
3
>>> p.eval_at_permutation_roots_on_generators(Integer(3), [Integer(1)])
1
>>> p.eval_at_permutation_roots_on_generators(Integer(3), [Integer(3),
↪Integer(3)])
6
>>> p.eval_at_permutation_roots_on_generators(Integer(3), [Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1)])
5
```

5.1.300 Schur symmetric functions

class sage.combinat.sf.schur.**SymmetricFunctionAlgebra_schur**(*Sym*)

Bases: *SymmetricFunctionAlgebra_classical*

A class for methods related to the Schur symmetric function basis.

INPUT:

- `self` – a Schur symmetric function basis

- Sym – an instance of the ring of the symmetric functions

class Element

Bases: *Element*

expand(*n*, *alphabet='x'*)

Expand the symmetric function self as a symmetric polynomial in n variables.

INPUT:

- n – nonnegative integer
- alphabet – (default: 'x') a variable for the expansion

OUTPUT:

A monomial expansion of self in the n variables labelled by alphabet.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: a.expand(2)
x0^2*x1 + x0*x1^2
sage: a.expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: a.expand(4)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2 +
↪x0^2*x3 + 2*x0*x1*x3 + x1^2*x3 + 2*x0*x2*x3 + 2*x1*x2*x3 + x2^2*x3 +
↪x0*x3^2 + x1*x3^2 + x2*x3^2
sage: a.expand(2, alphabet='y')
y0^2*y1 + y0*y1^2
sage: a.expand(2, alphabet=['a','b'])
a^2*b + a*b^2
sage: s([1,1,1,1]).expand(3)
0
sage: (s([]) + 2*s([1])).expand(3)
2*x0 + 2*x1 + 2*x2 + 1
sage: s([1]).expand(0)
0
sage: (3*s([])).expand(0)
3
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> a = s([Integer(2),Integer(1)])
>>> a.expand(Integer(2))
x0^2*x1 + x0*x1^2
>>> a.expand(Integer(3))
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
>>> a.expand(Integer(4))
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2 +
↪x0^2*x3 + 2*x0*x1*x3 + x1^2*x3 + 2*x0*x2*x3 + 2*x1*x2*x3 + x2^2*x3 +
↪x0*x3^2 + x1*x3^2 + x2*x3^2
>>> a.expand(Integer(2), alphabet='y')
y0^2*y1 + y0*y1^2
>>> a.expand(Integer(2), alphabet=['a','b'])
a^2*b + a*b^2
```

(continues on next page)

(continued from previous page)

```
>>> s([Integer(1),Integer(1),Integer(1),Integer(1)]).expand(Integer(3))
0
>>> (s([]) + Integer(2)*s([Integer(1)])).expand(Integer(3))
2*x0 + 2*x1 + 2*x2 + 1
>>> s([Integer(1)]).expand(Integer(0))
0
>>> (Integer(3)*s([])).expand(Integer(0))
3
```

exponential_specialization(*t=None*, *q=1*)

Return the exponential specialization of a symmetric function (when $q = 1$), or the q-exponential specialization (when $q \neq 1$).

The *exponential specialization* ex at t is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R. It is defined whenever the base ring K is a **Q**-algebra and t is an element of R. The easiest way to define it is by specifying its values on the powersum symmetric functions to be $p_1 = t$ and $p_n = 0$ for $n > 1$. Equivalently, on the homogeneous functions it is given by $ex(h_n) = t^n/n!$; see Proposition 7.8.4 of [EnumComb2].

By analogy, the q-exponential specialization is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R that depends on two elements t and q of R for which the elements $1 - q^i$ for all positive integers i are invertible. It can be defined by specifying its values on the complete homogeneous symmetric functions to be

$$ex_q(h_n) = t^n/[n]_q!,$$

where $[n]_q!$ is the q-factorial. Equivalently, for $q \neq 1$ and a homogeneous symmetric function f of degree n, we have

$$ex_q(f) = (1-q)^n t^n ps_q(f),$$

where $ps_q(f)$ is the stable principal specialization of f (see `principal_specialization()`). (See (7.29) in [EnumComb2].)

The limit of ex_q as $q \to 1$ is ex.

INPUT:

- `t` – (default: `None`) the value to use for t; the default is to create a ring of polynomials in `t`
- `q` – (default: 1) the value to use for q. If `q` is `None`, then a ring (or fraction field) of polynomials in `q` is created.

We use the formula in the proof of Corollary 7.21.6 of [EnumComb2]

$$ex_q(s_\lambda) = t^{|\lambda|} q^{\sum_i (i-1)\lambda_i} / \prod_{u\in\lambda} (1 + q + q^2 + \cdots + q^{h(u)-1})$$

where $h(u)$ is the hook length of a cell u in λ.

As a limit case, we obtain a formula for $q = 1$

$$ex_1(s_\lambda) = f^\lambda t^{|\lambda|}/|\lambda|!$$

where f^λ is the number of standard Young tableaux of shape λ.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: x = s[5,3]
sage: x.exponential_specialization()
1/1440*t^8

sage: x = 5*s[1,1,1] + 3*s[2,1] + 1
sage: x.exponential_specialization()
11/6*t^3 + 1
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> x = s[Integer(5),Integer(3)]
>>> x.exponential_specialization()
1/1440*t^8

>>> x = Integer(5)*s[Integer(1),Integer(1),Integer(1)] +␣
↪Integer(3)*s[Integer(2),Integer(1)] + Integer(1)
>>> x.exponential_specialization()
11/6*t^3 + 1
```

We also support the q-exponential_specialization:

```
sage: factor(s[3].exponential_specialization(q=var("q"), t=var("t")))    #␣
↪needs sage.symbolic
t^3/((q^2 + q + 1)*(q + 1))
```

```
>>> from sage.all import *
>>> factor(s[Integer(3)].exponential_specialization(q=var("q"), t=var("t
↪")))    # needs sage.symbolic
t^3/((q^2 + q + 1)*(q + 1))
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT: the image of `self` under the omega automorphism

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([2,1]).omega()
s[2, 1]
sage: s([2,1,1]).omega()
s[3, 1]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s([Integer(2),Integer(1)]).omega()
s[2, 1]
>>> s([Integer(2),Integer(1),Integer(1)]).omega()
s[3, 1]
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT: the image of `self` under the omega automorphism

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([2,1]).omega()
s[2, 1]
sage: s([2,1,1]).omega()
s[3, 1]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s([Integer(2),Integer(1)]).omega()
s[2, 1]
>>> s([Integer(2),Integer(1),Integer(1)]).omega()
s[3, 1]
```

principal_specialization(*n=+Infinity*, *q=None*)

Return the principal specialization of a symmetric function.

The *principal specialization* of order n at q is the ring homomorphism $ps_{n,q}$ from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for $i \in \{1, \ldots, n\}$ and $x_i \mapsto 0$ for $i > n$. Here, q is a given element of R, and we assume that the variables of our symmetric functions are

$x_1, x_2, x_3, \ldots$. (To be more precise, $ps_{n,q}$ is a K-algebra homomorphism, where K is the base ring.) See Section 7.8 of [EnumComb2].

The *stable principal specialization* at q is the ring homomorphism ps_q from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for all i. This is well-defined only if the resulting infinite sums converge; thus, in particular, setting $q = 1$ in the stable principal specialization is an invalid operation.

INPUT:

- `n` – (default: `infinity`) a nonnegative integer or `infinity`, specifying whether to compute the principal specialization of order `n` or the stable principal specialization.
- `q` – (default: `None`) the value to use for q; the default is to create a ring of polynomials in `q` (or a field of rational functions in `q`) over the given coefficient ring.

For $q = 1$ we use the formula from Corollary 7.21.4 of [EnumComb2]:

$$ps_{n,1}(s_\lambda) = \prod_{u\in\lambda} (n + c(u))/h(u),$$

where $h(u)$ is the hook length of a cell u in λ, and where $c(u)$ is the content of a cell u in λ.

For $n = infinity$ we use the formula from Corollary 7.21.3 of [EnumComb2]

$$ps_q(s_\lambda) = q^{\sum_i (i-1)\lambda_i} / \prod_{u\in\lambda} (1 - q^{h(u)}).$$

Otherwise, we use the formula from Theorem 7.21.2 of [EnumComb2],

$$ps_{n,q}(s_\lambda) = q^{\sum_i (i-1)\lambda_i} \prod_{u\in\lambda} (1 - q^{n+c(u)})/(1 - q^{h(u)}).$$

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: x = s[2]
sage: x.principal_specialization(3)
q^4 + q^3 + 2*q^2 + q + 1

sage: x = 3*s[2,2] + 2*s[1] + 1
sage: x.principal_specialization(3, q=var("q"))                               #␣
↪needs sage.symbolic
3*(q^4 - 1)*(q^3 - 1)*q^2/((q^2 - 1)*(q - 1)) + 2*(q^3 - 1)/(q - 1) + 1

sage: x.principal_specialization(q=var("q"))                                  #␣
↪needs sage.symbolic
-2/(q - 1) + 3*q^2/((q^3 - 1)*(q^2 - 1)^2*(q - 1)) + 1
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> x = s[Integer(2)]
>>> x.principal_specialization(Integer(3))
q^4 + q^3 + 2*q^2 + q + 1

>>> x = Integer(3)*s[Integer(2),Integer(2)] + Integer(2)*s[Integer(1)] +␣
↪Integer(1)
>>> x.principal_specialization(Integer(3), q=var("q"))                        ␣
↪    # needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
3*(q^4 - 1)*(q^3 - 1)*q^2/((q^2 - 1)*(q - 1)) + 2*(q^3 - 1)/(q - 1) + 1

>>> x.principal_specialization(q=var("q"))                         #␣
↪needs sage.symbolic
-2/(q - 1) + 3*q^2/((q^3 - 1)*(q^2 - 1)^2*(q - 1)) + 1
```

scalar(*x*, *zee=None*)

Return the standard scalar product between `self` and x.

Note that the Schur functions are self-dual with respect to this scalar product. They are also lower-triangularly related to the monomial symmetric functions with respect to this scalar product.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `zee` – an optional function on partitions giving the value for the scalar product between the power-sum symmetric function p_μ and itself (the default value is the standard *zee()* function)

OUTPUT: the scalar product between `self` and `x`

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([2,1])
sage: b = s([1,1,1])
sage: c = 2*s([1,1,1])
sage: d = a + b
sage: a.scalar(a)
1
sage: b.scalar(b)
1
sage: b.scalar(a)
0
sage: b.scalar(c)
2
sage: c.scalar(c)
4
sage: d.scalar(a)
1
sage: d.scalar(b)
1
sage: d.scalar(c)
2
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> a = s([Integer(2),Integer(1)])
>>> b = s([Integer(1),Integer(1),Integer(1)])
>>> c = Integer(2)*s([Integer(1),Integer(1),Integer(1)])
>>> d = a + b
>>> a.scalar(a)
1
>>> b.scalar(b)
1
>>> b.scalar(a)
```

(continues on next page)

(continued from previous page)

```
0
>>> b.scalar(c)
2
>>> c.scalar(c)
4
>>> d.scalar(a)
1
>>> d.scalar(b)
1
>>> d.scalar(c)
2
```

```
sage: m = SymmetricFunctions(ZZ).monomial()
sage: p4 = Partitions(4)
sage: l = [ [s(p).scalar(m(q)) for q in p4] for p in p4]
sage: matrix(l)
[ 1  0  0  0  0]
[-1  1  0  0  0]
[ 0 -1  1  0  0]
[ 1 -1 -1  1  0]
[-1  2  1 -3  1]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(ZZ).monomial()
>>> p4 = Partitions(Integer(4))
>>> l = [ [s(p).scalar(m(q)) for q in p4] for p in p4]
>>> matrix(l)
[ 1  0  0  0  0]
[-1  1  0  0  0]
[ 0 -1  1  0  0]
[ 1 -1 -1  1  0]
[-1  2  1 -3  1]
```

verschiebung(n)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n} e_{r/n}$$

(where h is the complete homogeneous basis, p is the powersum basis, and e is the elementary basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for "shift") endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Frobenius operator (see `frobenius()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley's [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, \ldots, \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, \ldots, \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: s = Sym.s()
sage: s[5].verschiebung(2)
0
sage: s[6].verschiebung(6)
s[1]
sage: s[6,3].verschiebung(3)
s[2, 1] + s[3]
sage: s[6,3,1].verschiebung(2)
-s[3, 2]
sage: s[3,2,1].verschiebung(1)
s[3, 2, 1]
sage: s([]).verschiebung(1)
s[]
sage: s([]).verschiebung(4)
s[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> s = Sym.s()
>>> s[Integer(5)].verschiebung(Integer(2))
0
>>> s[Integer(6)].verschiebung(Integer(6))
s[1]
>>> s[Integer(6),Integer(3)].verschiebung(Integer(3))
s[2, 1] + s[3]
>>> s[Integer(6),Integer(3),Integer(1)].verschiebung(Integer(2))
-s[3, 2]
>>> s[Integer(3),Integer(2),Integer(1)].verschiebung(Integer(1))
s[3, 2, 1]
>>> s([]).verschiebung(Integer(1))
s[]
>>> s([]).verschiebung(Integer(4))
```

(continues on next page)

(continued from previous page)

```
s[]
```

coproduct_on_basis(*mu*)

Return the coproduct of `self(mu)`.

Here `self` is the basis of Schur functions in the ring of symmetric functions.

INPUT:

- `self` – a Schur symmetric function basis
- `mu` – a partition

OUTPUT:

- the image of the `mu`-th Schur function under the comultiplication of the Hopf algebra of symmetric functions; this is an element of the tensor square of the Schur basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: s.coproduct_on_basis([2])
s[] # s[2] + s[1] # s[1] + s[2] # s[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.schur()
>>> s.coproduct_on_basis([Integer(2)])
s[] # s[2] + s[1] # s[1] + s[2] # s[]
```

product_on_basis(*left*, *right*)

Return the product of `left` and `right`.

INPUT:

- `self` – a Schur symmetric function basis
- `left`, `right` – partitions

OUTPUT: an element of the Schur basis, the product of `left` and `right`

5.1.301 Symplectic Symmetric Functions

AUTHORS:

- Travis Scrimshaw (2013-11-10): Initial version

class sage.combinat.sf.symplectic.**SymmetricFunctionAlgebra_symplectic**(*Sym*)

Bases: *SymmetricFunctionAlgebra_generic*

The symplectic symmetric function basis (or symplectic basis, to be short).

The symplectic basis $\{sp_\lambda\}$ where λ is taken over all partitions is defined by the following change of basis with the Schur functions:

$$s_\lambda = \sum_\mu \left(\sum_{\nu \in V} c^\lambda_{\mu\nu} \right) sp_\mu$$

where V is the set of all partitions with even-height columns and $c^{\lambda}_{\mu\nu}$ is the usual Littlewood-Richardson (LR) coefficients. By the properties of LR coefficients, this can be shown to be a upper unitriangular change of basis.

Note: This is only a filtered basis, not a $\mathbf{Z}$-graded basis. However this does respect the induced $(\mathbf{Z}/2\mathbf{Z})$-grading.

INPUT:

- `Sym` – an instance of the ring of the symmetric functions

REFERENCES:

EXAMPLES:

Here are the first few symplectic symmetric functions, in various bases:

```
sage: Sym = SymmetricFunctions(QQ)
sage: sp = Sym.sp()
sage: e = Sym.e()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()

sage: p(sp([1]))
p[1]
sage: m(sp([1]))
m[1]
sage: e(sp([1]))
e[1]
sage: h(sp([1]))
h[1]
sage: s(sp([1]))
s[1]

sage: p(sp([2]))
1/2*p[1, 1] + 1/2*p[2]
sage: m(sp([2]))
m[1, 1] + m[2]
sage: e(sp([2]))
e[1, 1] - e[2]
sage: h(sp([2]))
h[2]
sage: s(sp([2]))
s[2]

sage: p(sp([3]))
1/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3]
sage: m(sp([3]))
m[1, 1, 1] + m[2, 1] + m[3]
sage: e(sp([3]))
e[1, 1, 1] - 2*e[2, 1] + e[3]
sage: h(sp([3]))
```

(continues on next page)

(continued from previous page)

```
h[3]
sage: s(sp([3]))
s[3]

sage: Sym = SymmetricFunctions(ZZ)
sage: sp = Sym.sp()
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: p = Sym.p()
sage: m(sp([4]))
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1] + m[4]
sage: e(sp([4]))
e[1, 1, 1, 1] - 3*e[2, 1, 1] + e[2, 2] + 2*e[3, 1] - e[4]
sage: h(sp([4]))
h[4]
sage: s(sp([4]))
s[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> sp = Sym.sp()
>>> e = Sym.e()
>>> h = Sym.h()
>>> p = Sym.p()
>>> s = Sym.s()
>>> m = Sym.m()

>>> p(sp([Integer(1)]))
p[1]
>>> m(sp([Integer(1)]))
m[1]
>>> e(sp([Integer(1)]))
e[1]
>>> h(sp([Integer(1)]))
h[1]
>>> s(sp([Integer(1)]))
s[1]

>>> p(sp([Integer(2)]))
1/2*p[1, 1] + 1/2*p[2]
>>> m(sp([Integer(2)]))
m[1, 1] + m[2]
>>> e(sp([Integer(2)]))
e[1, 1] - e[2]
>>> h(sp([Integer(2)]))
h[2]
>>> s(sp([Integer(2)]))
s[2]

>>> p(sp([Integer(3)]))
```

(continues on next page)

(continued from previous page)

```
1/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3]
>>> m(sp([Integer(3)]))
m[1, 1, 1] + m[2, 1] + m[3]
>>> e(sp([Integer(3)]))
e[1, 1, 1] - 2*e[2, 1] + e[3]
>>> h(sp([Integer(3)]))
h[3]
>>> s(sp([Integer(3)]))
s[3]

>>> Sym = SymmetricFunctions(ZZ)
>>> sp = Sym.sp()
>>> e = Sym.e()
>>> h = Sym.h()
>>> s = Sym.s()
>>> m = Sym.m()
>>> p = Sym.p()
>>> m(sp([Integer(4)]))
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1] + m[4]
>>> e(sp([Integer(4)]))
e[1, 1, 1, 1] - 3*e[2, 1, 1] + e[2, 2] + 2*e[3, 1] - e[4]
>>> h(sp([Integer(4)]))
h[4]
>>> s(sp([Integer(4)]))
s[4]
```

Some examples of conversions the other way:

```
sage: sp(h[3])
sp[3]
sage: sp(e[3])
sp[1] + sp[1, 1, 1]
sage: sp(m[2,1])
-sp[1] - 2*sp[1, 1, 1] + sp[2, 1]
sage: sp(p[3])
sp[1, 1, 1] - sp[2, 1] + sp[3]
```

```
>>> from sage.all import *
>>> sp(h[Integer(3)])
sp[3]
>>> sp(e[Integer(3)])
sp[1] + sp[1, 1, 1]
>>> sp(m[Integer(2),Integer(1)])
-sp[1] - 2*sp[1, 1, 1] + sp[2, 1]
>>> sp(p[Integer(3)])
sp[1, 1, 1] - sp[2, 1] + sp[3]
```

Some multiplication:

```
sage: sp([2]) * sp([1,1])
sp[1, 1] + sp[2] + sp[2, 1, 1] + sp[3, 1]
sage: sp([2,1,1]) * sp([2])
```

(continues on next page)

(continued from previous page)

```
sp[1, 1] + sp[1, 1, 1, 1] + 2*sp[2, 1, 1] + sp[2, 2] + sp[2, 2, 1, 1]
 + sp[3, 1] + sp[3, 1, 1, 1] + sp[3, 2, 1] + sp[4, 1, 1]
sage: sp([1,1]) * sp([2,1])
sp[1] + sp[1, 1, 1] + 2*sp[2, 1] + sp[2, 1, 1, 1] + sp[2, 2, 1]
 + sp[3] + sp[3, 1, 1] + sp[3, 2]
```

```
>>> from sage.all import *
>>> sp([Integer(2)]) * sp([Integer(1),Integer(1)])
sp[1, 1] + sp[2] + sp[2, 1, 1] + sp[3, 1]
>>> sp([Integer(2),Integer(1),Integer(1)]) * sp([Integer(2)])
sp[1, 1] + sp[1, 1, 1, 1] + 2*sp[2, 1, 1] + sp[2, 2] + sp[2, 2, 1, 1]
 + sp[3, 1] + sp[3, 1, 1, 1] + sp[3, 2, 1] + sp[4, 1, 1]
>>> sp([Integer(1),Integer(1)]) * sp([Integer(2),Integer(1)])
sp[1] + sp[1, 1, 1] + 2*sp[2, 1] + sp[2, 1, 1, 1] + sp[2, 2, 1]
 + sp[3] + sp[3, 1, 1] + sp[3, 2]
```

Examples of the Hopf algebra structure:

```
sage: sp([1]).antipode()
-sp[1]
sage: sp([2]).antipode()
sp[] + sp[1, 1]
sage: sp([1]).coproduct()
sp[] # sp[1] + sp[1] # sp[]
sage: sp([2]).coproduct()
sp[] # sp[2] + sp[1] # sp[1] + sp[2] # sp[]
sage: sp([1]).counit()
0
sage: sp.one().counit()
1
```

```
>>> from sage.all import *
>>> sp([Integer(1)]).antipode()
-sp[1]
>>> sp([Integer(2)]).antipode()
sp[] + sp[1, 1]
>>> sp([Integer(1)]).coproduct()
sp[] # sp[1] + sp[1] # sp[]
>>> sp([Integer(2)]).coproduct()
sp[] # sp[2] + sp[1] # sp[1] + sp[2] # sp[]
>>> sp([Integer(1)]).counit()
0
>>> sp.one().counit()
1
```

5.1.302 Symmetric functions, with their multiple realizations

class sage.combinat.sf.sf.**SymmetricFunctions**(R)

Bases: UniqueRepresentation, Parent

The abstract algebra of commutative symmetric functions.

Symmetric Functions in Sage

Author: Jason Bandlow, Anne Schilling, Nicolas M. Thiery, Mike Zabrocki

This document is an introduction to working with symmetric function theory in Sage. It is not intended to be an introduction to the theory of symmetric functions ([MAC] and [STA], Chapter 7, are two excellent references.) The reader is also expected to be familiar with Sage.

The algebra of symmetric functions

The algebra of symmetric functions is the unique free commutative graded connected algebra over the given ring, with one generator in each degree. It can also be thought of as the inverse limit (in the category of graded algebras) of the algebra of symmetric polynomials in n variables as $n \to \infty$. Sage allows us to construct the algebra of symmetric functions over any ring. We will use a base ring of rational numbers in these first examples:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sym
Symmetric Functions over Rational Field
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Sym
Symmetric Functions over Rational Field
```

Sage knows certain categorical information about this algebra:

```
sage: Sym.category()
Join of Category of Hopf algebras over Rational Field
    and Category of unique factorization domains
    and Category of graded algebras over Rational Field
    and Category of commutative algebras over Rational Field
    and Category of monoids with realizations
    and Category of graded coalgebras over Rational Field
    and Category of coalgebras over Rational Field with realizations
    and Category of cocommutative coalgebras over Rational Field
```

```
>>> from sage.all import *
>>> Sym.category()
Join of Category of Hopf algebras over Rational Field
    and Category of unique factorization domains
    and Category of graded algebras over Rational Field
    and Category of commutative algebras over Rational Field
    and Category of monoids with realizations
    and Category of graded coalgebras over Rational Field
    and Category of coalgebras over Rational Field with realizations
    and Category of cocommutative coalgebras over Rational Field
```

Notice that `Sym` is an *abstract* algebra. This reflects the fact that there are multiple natural bases. To work with specific elements, we need a *realization* of this algebra. In practice, this means we need to specify a basis.

An example basis - power sums

Here is an example of how one might use the power sum realization:

```
sage: p = Sym.powersum()
sage: p
Symmetric Functions over Rational Field in the powersum basis
```

```
>>> from sage.all import *
>>> p = Sym.powersum()
>>> p
Symmetric Functions over Rational Field in the powersum basis
```

`p` now represents the realization of the symmetric function algebra on the power sum basis. The basis itself is accessible through:

```
sage: p.basis()
Lazy family (Term map from Partitions to Symmetric Functions over Rational Field␣
↪in the powersum basis(i))_{i in Partitions}
sage: p.basis().keys()
Partitions
```

```
>>> from sage.all import *
>>> p.basis()
Lazy family (Term map from Partitions to Symmetric Functions over Rational Field␣
↪in the powersum basis(i))_{i in Partitions}
>>> p.basis().keys()
Partitions
```

This last line means that `p.basis()` is an association between the set of Partitions and the basis elements of the algebra `p`. To construct a specific element one can therefore do:

```
sage: p.basis()[Partition([2,1,1])]
p[2, 1, 1]
```

```
>>> from sage.all import *
>>> p.basis()[Partition([Integer(2),Integer(1),Integer(1)])]
p[2, 1, 1]
```

As this is rather cumbersome, realizations of the symmetric function algebra allow for the following abuses of notation:

```
sage: p[Partition([2, 1, 1])]
p[2, 1, 1]
sage: p[[2, 1, 1]]
p[2, 1, 1]
sage: p[2, 1, 1]
p[2, 1, 1]
```

```
>>> from sage.all import *
>>> p[Partition([Integer(2), Integer(1), Integer(1)])]
p[2, 1, 1]
>>> p[[Integer(2), Integer(1), Integer(1)]]
```

(continues on next page)

(continued from previous page)

```
p[2, 1, 1]
>>> p[Integer(2), Integer(1), Integer(1)]
p[2, 1, 1]
```

or even:

```
sage: p[(i for i in [2, 1, 1])]
p[2, 1, 1]
```

```
>>> from sage.all import *
>>> p[(i for i in [Integer(2), Integer(1), Integer(1)])]
p[2, 1, 1]
```

In the special case of the empty partition, due to a limitation in Python syntax, one cannot use:

```
sage: p[]       # todo: not implemented
```

```
>>> from sage.all import *
>>> p[]       # todo: not implemented
```

Please use instead:

```
sage: p[[]]
p[]
```

```
>>> from sage.all import *
>>> p[[]]
p[]
```

Note: When elements are constructed using the `p[something ]` syntax , an error will be raised if the input cannot be interpreted as a partition. This is *not* the case when `p.basis()` is used:

```
sage: p['something']
Traceback (most recent call last):
...
ValueError: all parts of 'something' should be nonnegative integers
sage: p.basis()['something']
p'something'
```

```
>>> from sage.all import *
>>> p['something']
Traceback (most recent call last):
...
ValueError: all parts of 'something' should be nonnegative integers
>>> p.basis()['something']
p'something'
```

Elements of `p` are linear combinations of such compositions:

```
sage: p.an_element()
2*p[] + 2*p[1] + 3*p[2]
```

```
>>> from sage.all import *
>>> p.an_element()
2*p[] + 2*p[1] + 3*p[2]
```

Algebra structure

Algebraic combinations of basis elements can be entered in a natural way:

```
sage: p[2,1,1] + 2 * p[1] * (p[4] + p[2,1])
3*p[2, 1, 1] + 2*p[4, 1]
```

```
>>> from sage.all import *
>>> p[Integer(2),Integer(1),Integer(1)] + Integer(2) * p[Integer(1)] *␣
↪(p[Integer(4)] + p[Integer(2),Integer(1)])
3*p[2, 1, 1] + 2*p[4, 1]
```

Let us explore the other operations of `p`. We can ask for the mathematical properties of `p`:

```
sage: p.categories()
[Category of graded bases of Symmetric Functions over Rational Field,
 Category of filtered bases of Symmetric Functions over Rational Field,
 Category of bases of Symmetric Functions over Rational Field,
 Category of graded Hopf algebras with basis over Rational Field,
 ...]
```

```
>>> from sage.all import *
>>> p.categories()
[Category of graded bases of Symmetric Functions over Rational Field,
 Category of filtered bases of Symmetric Functions over Rational Field,
 Category of bases of Symmetric Functions over Rational Field,
 Category of graded Hopf algebras with basis over Rational Field,
 ...]
```

To start with, `p` is a graded algebra, the grading being induced by the size of the partitions. Due to this, the one is the basis element indexed by the empty partition:

```
sage: p.one()
p[]
```

```
>>> from sage.all import *
>>> p.one()
p[]
```

The `p` basis is multiplicative; that is, multiplication is induced by linearity from the (nonincreasingly sorted) concatenation of partitions:

```
sage: p[3,1] * p[2,1]
p[3, 2, 1, 1]
```

(continues on next page)

(continued from previous page)

```
sage: (p.one() + 2 * p[3,1]) * p[4, 2]
p[4, 2] + 2*p[4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> p[Integer(3),Integer(1)] * p[Integer(2),Integer(1)]
p[3, 2, 1, 1]

>>> (p.one() + Integer(2) * p[Integer(3),Integer(1)]) * p[Integer(4), Integer(2)]
p[4, 2] + 2*p[4, 3, 2, 1]
```

The classical bases

In addition to the power sum basis, other classical bases of the symmetric function algebra include the elementary, complete homogeneous, monomial, and Schur bases:

```
sage: e = Sym.elementary()
sage: h = Sym.homogeneous()
sage: m = Sym.monomial()
sage: s = Sym.schur()
```

```
>>> from sage.all import *
>>> e = Sym.elementary()
>>> h = Sym.homogeneous()
>>> m = Sym.monomial()
>>> s = Sym.schur()
```

These and others can be defined all at once with the single command:

```
sage: Sym.inject_shorthands()
Defining e as shorthand for Symmetric Functions over Rational Field in the␣
↪elementary basis
Defining f as shorthand for Symmetric Functions over Rational Field in the␣
↪forgotten basis
Defining h as shorthand for Symmetric Functions over Rational Field in the␣
↪homogeneous basis
Defining m as shorthand for Symmetric Functions over Rational Field in the␣
↪monomial basis
Defining p as shorthand for Symmetric Functions over Rational Field in the␣
↪powersum basis
Defining s as shorthand for Symmetric Functions over Rational Field in the Schur␣
↪basis
```

```
>>> from sage.all import *
>>> Sym.inject_shorthands()
Defining e as shorthand for Symmetric Functions over Rational Field in the␣
↪elementary basis
Defining f as shorthand for Symmetric Functions over Rational Field in the␣
↪forgotten basis
Defining h as shorthand for Symmetric Functions over Rational Field in the␣
↪homogeneous basis
Defining m as shorthand for Symmetric Functions over Rational Field in the␣
```

(continues on next page)

(continued from previous page)

```
↪monomial basis
Defining p as shorthand for Symmetric Functions over Rational Field in the␣
↪powersum basis
Defining s as shorthand for Symmetric Functions over Rational Field in the Schur␣
↪basis
```

We can then do conversions from one basis to another:

```
sage: s(p[2,1])
-s[1, 1, 1] + s[3]

sage: m(p[3])
m[3]
sage: m(p[3,2])
m[3, 2] + m[5]
```

```
>>> from sage.all import *
>>> s(p[Integer(2),Integer(1)])
-s[1, 1, 1] + s[3]

>>> m(p[Integer(3)])
m[3]
>>> m(p[Integer(3),Integer(2)])
m[3, 2] + m[5]
```

For computations which mix bases, Sage will return a result with respect to a single (not necessarily predictable) basis:

```
sage: p[2] * s[2] - m[4]
1/2*p[2, 1, 1] + 1/2*p[2, 2] - p[4]

sage: p( m[1] * ( e[3]*s[2] + 1 ))
p[1] + 1/12*p[1, 1, 1, 1, 1, 1] - 1/6*p[2, 1, 1, 1, 1] - 1/4*p[2, 2, 1, 1] + 1/
↪6*p[3, 1, 1, 1] + 1/6*p[3, 2, 1]
```

```
>>> from sage.all import *
>>> p[Integer(2)] * s[Integer(2)] - m[Integer(4)]
1/2*p[2, 1, 1] + 1/2*p[2, 2] - p[4]

>>> p( m[Integer(1)] * ( e[Integer(3)]*s[Integer(2)] + Integer(1) ))
p[1] + 1/12*p[1, 1, 1, 1, 1, 1] - 1/6*p[2, 1, 1, 1, 1] - 1/4*p[2, 2, 1, 1] + 1/
↪6*p[3, 1, 1, 1] + 1/6*p[3, 2, 1]
```

The one for different bases such as the power sum and Schur function is the same:

```
sage: s.one() == p.one()
True
```

```
>>> from sage.all import *
>>> s.one() == p.one()
True
```

Basic computations

In this section, we explore some of the many methods that can be applied to an arbitrary symmetric function:

```
sage: f = s[2]^2; f
s[2, 2] + s[3, 1] + s[4]
```

```
>>> from sage.all import *
>>> f = s[Integer(2)]**Integer(2); f
s[2, 2] + s[3, 1] + s[4]
```

For more methods than discussed here, create a symmetric function as above, and use `f.<tab>`.

Representation theory of the symmetric group

The Schur functions s_λ can also be interpreted as irreducible characters of the symmetric group S_n, where n is the size of the partition λ. Since the Schur functions of degree n form a basis of the symmetric functions of degree n, it follows that an arbitrary symmetric function (homogeneous of degree n) may be interpreted as a function on the symmetric group. In this interpretation the power sum symmetric function p_λ is the characteristic function of the conjugacy class with shape λ, multiplied by the order of the centralizer of an element. Hence the irreducible characters can be computed as follows:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: p = Sym.power()
sage: P = Partitions(5).list()
sage: P = [P[i] for i in range(len(P)-1,-1,-1)]
sage: M = matrix([[s[P[i]].scalar(p[P[j]]) for j in range(len(P))] for i in␣
↪range(len(P))])
sage: M
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.schur()
>>> p = Sym.power()
>>> P = Partitions(Integer(5)).list()
>>> P = [P[i] for i in range(len(P)-Integer(1),-Integer(1),-Integer(1))]
>>> M = matrix([[s[P[i]].scalar(p[P[j]]) for j in range(len(P))] for i in␣
↪range(len(P))])
>>> M
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
```

We can indeed check that this agrees with the character table of S_5:

```
sage: SymmetricGroup(5).character_table() == M
True
```

```
>>> from sage.all import *
>>> SymmetricGroup(Integer(5)).character_table() == M
True
```

In this interpretation of symmetric functions as characters on the symmetric group, the multiplication and comultiplication are interpreted as induction (from $S_n \times S_m$ to S_{n+m}) and restriction, respectively. The Schur functions can also be interpreted as characters of GL_n, see Partitions and Schur functions.

The omega involution

The ω involution is the linear extension of the map which sends e_λ to h_λ:

```
sage: h(f)
h[2, 2]
sage: e(f.omega())
e[2, 2]
```

```
>>> from sage.all import *
>>> h(f)
h[2, 2]
>>> e(f.omega())
e[2, 2]
```

The Hall scalar product

The Hall scalar product on the algebra of symmetric functions makes the Schur functions into an orthonormal basis:

```
sage: f.scalar(f)
3
```

```
>>> from sage.all import *
>>> f.scalar(f)
3
```

Skewing

Skewing is the adjoint operation to multiplication with respect to this scalar product:

```
sage: f.skew_by(s[1])
2*s[2, 1] + 2*s[3]
```

```
>>> from sage.all import *
>>> f.skew_by(s[Integer(1)])
2*s[2, 1] + 2*s[3]
```

In general, `s[la].skew_by(s[mu])` is the symmetric function typically denoted $s_{\lambda\backslash\mu}$ or $s_{\lambda/\mu}$.

Expanding into variables

We can expand a symmetric function into a symmetric polynomial in a specified number of variables:

```
sage: f.expand(2)
x0^4 + 2*x0^3*x1 + 3*x0^2*x1^2 + 2*x0*x1^3 + x1^4
```

```
>>> from sage.all import *
>>> f.expand(Integer(2))
x0^4 + 2*x0^3*x1 + 3*x0^2*x1^2 + 2*x0*x1^3 + x1^4
```

See the documentation for `expand` for more examples.

The Kronecker product

As in the section on the *Representation theory of the symmetric group*, a symmetric function may be considered as a class function on the symmetric group where the elements p_μ/z_μ are the indicators of a permutation having cycle structure μ. The Kronecker product of two symmetric functions corresponds to the pointwise product of these class functions.

Since the Schur functions are the irreducible characters of the symmetric group under this identification, the Kronecker product of two Schur functions corresponds to the internal tensor product of two irreducible symmetric group representations.

Under this identification, the Kronecker product of p_μ/z_μ and p_ν/z_ν is p_μ/z_μ if $\mu = \nu$, and the result is equal to 0 otherwise.

`internal_product`, `kronecker_product`, `inner_tensor` and `itensor` are different names for the same function.

```
sage: f.kronecker_product(f)
s[1, 1, 1, 1] + 3*s[2, 1, 1] + 4*s[2, 2] + 5*s[3, 1] + 3*s[4]
```

```
>>> from sage.all import *
>>> f.kronecker_product(f)
s[1, 1, 1, 1] + 3*s[2, 1, 1] + 4*s[2, 2] + 5*s[3, 1] + 3*s[4]
```

Plethysm

The *plethysm* of symmetric functions is the operation corresponding to composition of representations of the general linear group. See [STA] Chapter 7, Appendix 2 for details.

```
sage: s[2].plethysm(s[2])
s[2, 2] + s[4]
```

```
>>> from sage.all import *
>>> s[Integer(2)].plethysm(s[Integer(2)])
s[2, 2] + s[4]
```

Plethysm can also be written as a composition of functions:

```
sage: s[2]( s[2] )
s[2, 2] + s[4]
```

```
>>> from sage.all import *
>>> s[Integer(2)]( s[Integer(2)] )
s[2, 2] + s[4]
```

If the coefficient ring contains degree 1 elements, these are handled properly by plethysm:

```
sage: R.<t> = QQ[]; s = SymmetricFunctions(R).schur()
sage: s[2]( (1-t)*s[1] )
(t^2-t)*s[1, 1] + (-t+1)*s[2]
```

```
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1); s = SymmetricFunctions(R).schur()
>>> s[Integer(2)]( (Integer(1)-t)*s[Integer(1)] )
(t^2-t)*s[1, 1] + (-t+1)*s[2]
```

See the documentation for `plethysm` for more information.

Inner plethysm

The operation of inner plethysm `f.inner_plethysm(g)` models the composition of the S_n representation represented by g with the GL_m representation whose character is f. See the documentation of `inner_plethysm`, [ST94] or [STA], exercise 7.74 solutions for more information:

```
sage: s = SymmetricFunctions(QQ).schur()
sage: f = s[2]^2
sage: f.inner_plethysm(s[2])
s[2]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).schur()
>>> f = s[Integer(2)]**Integer(2)
>>> f.inner_plethysm(s[Integer(2)])
s[2]
```

Hopf algebra structure

The ring of symmetric functions is further endowed with a coalgebra structure. The coproduct is an algebra morphism, and therefore determined by its values on the generators; the power sum generators are primitive:

```
sage: p[1].coproduct()
p[] # p[1] + p[1] # p[]
sage: p[2].coproduct()
p[] # p[2] + p[2] # p[]
```

```
>>> from sage.all import *
>>> p[Integer(1)].coproduct()
p[] # p[1] + p[1] # p[]
>>> p[Integer(2)].coproduct()
p[] # p[2] + p[2] # p[]
```

The coproduct, being cocommutative on the generators, is cocommutative everywhere:

```
sage: p[2, 1].coproduct()
p[] # p[2, 1] + p[1] # p[2] + p[2] # p[1] + p[2, 1] # p[]
```

```
>>> from sage.all import *
>>> p[Integer(2), Integer(1)].coproduct()
p[] # p[2, 1] + p[1] # p[2] + p[2] # p[1] + p[2, 1] # p[]
```

This coproduct, along with the counit which sends every symmetric function to its 0-th homogeneous component, makes the ring of symmetric functions into a graded connected bialgebra. It is known that every graded connected bialgebra has an antipode. For the ring of symmetric functions, the antipode can be characterized explicitly: The antipode is an anti-algebra morphism (thus an algebra morphism, since our algebra is commutative) which sends p_λ to $(-1)^{\mathrm{length}(\lambda)} p_\lambda$ for every partition λ. Thus, in particular, it sends the generators on the `p` basis to their opposites:

```
sage: p[3].antipode()
-p[3]
sage: p[3,2,1].antipode()
-p[3, 2, 1]
```

```
>>> from sage.all import *
>>> p[Integer(3)].antipode()
-p[3]
>>> p[Integer(3),Integer(2),Integer(1)].antipode()
-p[3, 2, 1]
```

The graded connected bialgebra of symmetric functions over a **Q**-algebra has a rather simply-understood structure: It is (isomorphic to) the symmetric algebra of its space of primitives (which is spanned by the power-sum symmetric functions).

Here are further examples:

```
sage: f = s[2]^2
sage: f.antipode()
s[1, 1, 1, 1] + s[2, 1, 1] + s[2, 2]
sage: f.coproduct()
s[] # s[2, 2] + s[] # s[3, 1] + s[] # s[4] + 2*s[1] # s[2, 1] + 2*s[1] # s[3] +␣
↪s[1, 1] # s[1, 1]
+ s[1, 1] # s[2] + s[2] # s[1, 1] + 3*s[2] # s[2] + 2*s[2, 1] # s[1] + s[2, 2] #␣
↪s[] + 2*s[3] # s[1]
+ s[3, 1] # s[] + s[4] # s[]
sage: f.coproduct().apply_multilinear_morphism( lambda x,y: x*y.antipode() )
0
```

```
>>> from sage.all import *
>>> f = s[Integer(2)]**Integer(2)
>>> f.antipode()
s[1, 1, 1, 1] + s[2, 1, 1] + s[2, 2]
>>> f.coproduct()
s[] # s[2, 2] + s[] # s[3, 1] + s[] # s[4] + 2*s[1] # s[2, 1] + 2*s[1] # s[3] +␣
↪s[1, 1] # s[1, 1]
+ s[1, 1] # s[2] + s[2] # s[1, 1] + 3*s[2] # s[2] + 2*s[2, 1] # s[1] + s[2, 2] #␣
↪s[] + 2*s[3] # s[1]
+ s[3, 1] # s[] + s[4] # s[]
```

(continues on next page)

(continued from previous page)

```
>>> f.coproduct().apply_multilinear_morphism( lambda x,y: x*y.antipode() )
0
```

Transformations of symmetric functions

There are many methods in Sage which make it easy to manipulate symmetric functions. For example, if we have some function which acts on partitions (say, conjugation), it is a simple matter to apply it to the support of a symmetric function. Here is an example:

```
sage: conj = lambda mu: mu.conjugate()
sage: f = h[4] + 2*h[3,1]
sage: f.map_support(conj)
h[1, 1, 1, 1] + 2*h[2, 1, 1]
```

```
>>> from sage.all import *
>>> conj = lambda mu: mu.conjugate()
>>> f = h[Integer(4)] + Integer(2)*h[Integer(3),Integer(1)]
>>> f.map_support(conj)
h[1, 1, 1, 1] + 2*h[2, 1, 1]
```

We can also easily modify the coefficients:

```
sage: def foo(mu, coeff): return mu.conjugate(), -coeff
sage: f.map_item(foo)
-h[1, 1, 1, 1] - 2*h[2, 1, 1]
```

```
>>> from sage.all import *
>>> def foo(mu, coeff): return mu.conjugate(), -coeff
>>> f.map_item(foo)
-h[1, 1, 1, 1] - 2*h[2, 1, 1]
```

See also `map_coefficients`.

There are also methods for building functions directly:

```
sage: s.sum_of_monomials(mu for mu in Partitions(3))
s[1, 1, 1] + s[2, 1] + s[3]
sage: s.sum_of_monomials(Partitions(3))
s[1, 1, 1] + s[2, 1] + s[3]
sage: s.sum_of_terms( (mu, mu[0]) for mu in Partitions(3))
s[1, 1, 1] + 2*s[2, 1] + 3*s[3]
```

```
>>> from sage.all import *
>>> s.sum_of_monomials(mu for mu in Partitions(Integer(3)))
s[1, 1, 1] + s[2, 1] + s[3]
>>> s.sum_of_monomials(Partitions(Integer(3)))
s[1, 1, 1] + s[2, 1] + s[3]
>>> s.sum_of_terms( (mu, mu[Integer(0)]) for mu in Partitions(Integer(3)))
s[1, 1, 1] + 2*s[2, 1] + 3*s[3]
```

These are the preferred way to build elements within a program; the result will usually be faster than using `sum()`. It also guarantees that empty sums yields the zero of `s` (see also `s.sum`).

Note also that it is a good idea to use:

```
sage: s.one()
s[]
sage: s.zero()
0
```

```
>>> from sage.all import *
>>> s.one()
s[]
>>> s.zero()
0
```

instead of `s(1)` and `s(0)` within programs where speed is important, in order to prevent unnecessary coercions.

Different base rings

Depending on the base ring, the different realizations of the symmetric function algebra may not span the same space:

```
sage: SZ = SymmetricFunctions(ZZ)
sage: p = SZ.power(); s = SZ.schur()
sage: p(s[1,1,1])
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

```
>>> from sage.all import *
>>> SZ = SymmetricFunctions(ZZ)
>>> p = SZ.power(); s = SZ.schur()
>>> p(s[Integer(1),Integer(1),Integer(1)])
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Because of this, some functions may not behave as expected when working over the integers, even though they make mathematical sense:

```
sage: s[1,1,1].plethysm(s[1,1,1])
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

```
>>> from sage.all import *
>>> s[Integer(1),Integer(1),Integer(1)].plethysm(s[Integer(1),Integer(1),
↪Integer(1)])
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

It is possible to work over different base rings simultaneously:

```
sage: s = SymmetricFunctions(QQ).schur()
sage: p = SymmetricFunctions(QQ).power()
sage: sz = SymmetricFunctions(ZZ).schur(); sz._prefix = 'sz'
sage: pz = SymmetricFunctions(ZZ).power(); pz._prefix = 'pz'
sage: p(sz[1,1,1])
1/6*p[1, 1, 1] - 1/2*p[2, 1] + 1/3*p[3]
sage: sz( 1/6*p[1, 1, 1] - 1/2*p[2, 1] + 1/3*p[3] )
sz[1, 1, 1]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).schur()
>>> p = SymmetricFunctions(QQ).power()
>>> sz = SymmetricFunctions(ZZ).schur(); sz._prefix = 'sz'
>>> pz = SymmetricFunctions(ZZ).power(); pz._prefix = 'pz'
>>> p(sz[Integer(1),Integer(1),Integer(1)])
1/6*p[1, 1, 1] - 1/2*p[2, 1] + 1/3*p[3]
>>> sz( Integer(1)/Integer(6)*p[Integer(1), Integer(1), Integer(1)] - Integer(1)/
↪Integer(2)*p[Integer(2), Integer(1)] + Integer(1)/Integer(3)*p[Integer(3)] )
sz[1, 1, 1]
```

As shown in this example, if you are working over multiple base rings simultaneously, it is a good idea to change the prefix in some cases, so that you can tell from the output which realization your result is in.

Let us change the notation back for the remainder of this tutorial:

```
sage: sz._prefix = 's'
sage: pz._prefix = 'p'
```

```
>>> from sage.all import *
>>> sz._prefix = 's'
>>> pz._prefix = 'p'
```

One can also use the Sage standard renaming idiom to get shorter outputs:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sym.rename('Sym')
sage: Sym
Sym
sage: Sym.rename()
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Sym.rename('Sym')
>>> Sym
Sym
>>> Sym.rename()
```

And we name it back:

```
sage: Sym.rename('Symmetric Functions over Rational Field'); Sym
Symmetric Functions over Rational Field
```

```
>>> from sage.all import *
>>> Sym.rename('Symmetric Functions over Rational Field'); Sym
Symmetric Functions over Rational Field
```

Other bases

There are two additional basis of the symmetric functions which are not considered as classical bases:

- forgotten basis
- Witt basis

The forgotten basis is the dual basis of the elementary symmetric functions basis with respect to the Hall scalar product. The Witt basis can be constructed by

$$\prod_{d=1}^{\infty}(1-w_d t^d)^{-1} = \sum_{n=0}^{\infty} h_n t^n$$

where t is a formal variable.

There are further bases of the ring of symmetric functions, in general over fields with parameters such as q and t:

- Hall-Littlewood bases
- Jack bases
- Macdonald bases
- k-Schur functions
- Hecke character basis

We briefly demonstrate how to access these bases. For more information, see the documentation of the individual bases.

The *Jack polynomials* can be obtained as:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: Jack = Sym.jack()
sage: P = Jack.P(); J = Jack.J(); Q = Jack.Q()
sage: J(P[2,1])
(1/(t+2))*JackJ[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> Jack = Sym.jack()
>>> P = Jack.P(); J = Jack.J(); Q = Jack.Q()
>>> J(P[Integer(2),Integer(1)])
(1/(t+2))*JackJ[2, 1]
```

The parameter t can be specialized as follows:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Jack = Sym.jack(t = 1)
sage: P = Jack.P(); J = Jack.J(); Q = Jack.Q()
sage: J(P[2,1])
1/3*JackJ[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Jack = Sym.jack(t = Integer(1))
>>> P = Jack.P(); J = Jack.J(); Q = Jack.Q()
>>> J(P[Integer(2),Integer(1)])
1/3*JackJ[2, 1]
```

Similarly one can access the Hall-Littlewood and Macdonald polynomials, etc:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Mcd = Sym.macdonald()
sage: P = Mcd.P(); J = Mcd.J(); Q = Mcd.Q()
sage: J(P[2,1])
(1/(-q*t^4+2*q*t^3-q*t^2+t^2-2*t+1))*McdJ[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> Mcd = Sym.macdonald()
>>> P = Mcd.P(); J = Mcd.J(); Q = Mcd.Q()
>>> J(P[Integer(2),Integer(1)])
(1/(-q*t^4+2*q*t^3-q*t^2+t^2-2*t+1))*McdJ[2, 1]
```

We can also construct the $\bar{q}$ basis that can be used to determine character tables for Hecke algebras (with quadratic relation $T_i^2 = (1-q)T_i + q$):

```
sage: Sym = SymmetricFunctions(ZZ['q'].fraction_field())
sage: qbar = Sym.hecke_character()
sage: s = Sym.s()
sage: s(qbar[2,1])
-s[1, 1, 1] + (q-1)*s[2, 1] + q*s[3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ['q'].fraction_field())
>>> qbar = Sym.hecke_character()
>>> s = Sym.s()
>>> s(qbar[Integer(2),Integer(1)])
-s[1, 1, 1] + (q-1)*s[2, 1] + q*s[3]
```

k-Schur functions

The k-Schur functions live in the k-bounded subspace of the ring of symmetric functions. It is possible to compute in the k-bounded subspace directly:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: f = ks[2,1]*ks[2,1]; f
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ks = Sym.kschur(Integer(3),Integer(1))
>>> f = ks[Integer(2),Integer(1)]*ks[Integer(2),Integer(1)]; f
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
```

or to lift to the ring of symmetric functions:

```
sage: f.lift()
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1]␣
↪+ s[4, 2]
```

```
>>> from sage.all import *
>>> f.lift()
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1]␣
↪+ s[4, 2]
```

However, it is not always possible to convert a symmetric function to the k-bounded subspace:

```
sage: s = Sym.schur()
sage: ks(s[2,1,1])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1] is not in the image
```

```
>>> from sage.all import *
>>> s = Sym.schur()
>>> ks(s[Integer(2),Integer(1),Integer(1)])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1] is not in the image
```

The k-Schur functions are more generally defined with a parameter t and they are a basis of the subspace spanned by the Hall-Littlewood `Qp` symmetric functions indexed by partitions whose first part is less than or equal to k:

```
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: SymS3 = Sym.kBoundedSubspace(3) # default t='t'
sage: ks = SymS3.kschur()
sage: Qp = Sym.hall_littlewood().Qp()
sage: ks(Qp[2,1,1,1])
ks3[2, 1, 1, 1] + (t^2+t)*ks3[2, 2, 1] + (t^3+t^2)*ks3[3, 1, 1] + t^4*ks3[3, 2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'].fraction_field())
>>> SymS3 = Sym.kBoundedSubspace(Integer(3)) # default t='t'
>>> ks = SymS3.kschur()
>>> Qp = Sym.hall_littlewood().Qp()
>>> ks(Qp[Integer(2),Integer(1),Integer(1),Integer(1)])
ks3[2, 1, 1, 1] + (t^2+t)*ks3[2, 2, 1] + (t^3+t^2)*ks3[3, 1, 1] + t^4*ks3[3, 2]
```

The subspace spanned by the k-Schur functions with a parameter t are not known to form a natural algebra. However it is known that the product of a k-Schur function and an ℓ-Schur function is in the linear span of the $k+\ell$-Schur functions:

```
sage: ks(ks[2,1]*ks[1,1])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] is not in the image
sage: ks[2,1]*ks[1,1]
```

(continues on next page)

(continued from previous page)

```
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2]
sage: ks6 = Sym.kBoundedSubspace(6).kschur()
sage: ks6(ks[3,1,1]*ks[3])
ks6[3, 3, 1, 1] + ks6[4, 2, 1, 1] + (t+1)*ks6[4, 3, 1] + t*ks6[4, 4]
+ ks6[5, 1, 1, 1] + ks6[5, 2, 1] + t*ks6[5, 3] + ks6[6, 1, 1]
```

```
>>> from sage.all import *
>>> ks(ks[Integer(2),Integer(1)]*ks[Integer(1),Integer(1)])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] is not in the image
>>> ks[Integer(2),Integer(1)]*ks[Integer(1),Integer(1)]
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2]
>>> ks6 = Sym.kBoundedSubspace(Integer(6)).kschur()
>>> ks6(ks[Integer(3),Integer(1),Integer(1)]*ks[Integer(3)])
ks6[3, 3, 1, 1] + ks6[4, 2, 1, 1] + (t+1)*ks6[4, 3, 1] + t*ks6[4, 4]
+ ks6[5, 1, 1, 1] + ks6[5, 2, 1] + t*ks6[5, 3] + ks6[6, 1, 1]
```

The k-split basis is a second basis of the ring spanned by the k-Schur functions with a parameter t. The k-split basis has the property that $Q'_\lambda[X;t]$ expands positively in the k-split basis and the k-split basis conjecturally expands positively in the k-Schur functions. The definition can be found in [LLMSSZ] p. 81.:

```
sage: ksp3 = SymS3.ksplit()
sage: ksp3(Qp[2,1,1,1])
ksp3[2, 1, 1, 1] + t^2*ksp3[2, 2, 1] + (t^3+t^2)*ksp3[3, 1, 1] + t^4*ksp3[3, 2]
sage: [ks(ksp3(la)) for la in sorted(ksp3(Qp[2,1,1,1]).support())]
[ks3[2, 1, 1, 1] + t*ks3[2, 2, 1], ks3[2, 2, 1], ks3[3, 1, 1], ks3[3, 2]]
```

```
>>> from sage.all import *
>>> ksp3 = SymS3.ksplit()
>>> ksp3(Qp[Integer(2),Integer(1),Integer(1),Integer(1)])
ksp3[2, 1, 1, 1] + t^2*ksp3[2, 2, 1] + (t^3+t^2)*ksp3[3, 1, 1] + t^4*ksp3[3, 2]
>>> [ks(ksp3(la)) for la in sorted(ksp3(Qp[Integer(2),Integer(1),Integer(1),
→Integer(1)]).support())]
[ks3[2, 1, 1, 1] + t*ks3[2, 2, 1], ks3[2, 2, 1], ks3[3, 1, 1], ks3[3, 2]]
```

dual k-Schur functions

The dual space to the subspace spanned by the k-Schur functions is most naturally realized as a quotient of the ring of symmetric functions by an ideal. When $t = 1$ the ideal is generated by the monomial symmetric functions indexed by partitions whose first part is greater than k.:

```
sage: Sym = SymmetricFunctions(QQ)
sage: SymQ3 = Sym.kBoundedQuotient(3,t=1)
sage: km = SymQ3.kmonomial()
sage: km[2,1]*km[2,1]
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
sage: F = SymQ3.affineSchur()
sage: F[2,1]*F[2,1]
2*F3[1, 1, 1, 1, 1, 1] + 4*F3[2, 1, 1, 1, 1] + 4*F3[2, 2, 1, 1] + 4*F3[2, 2, 2]
+ 2*F3[3, 1, 1, 1] + 4*F3[3, 2, 1] + 2*F3[3, 3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> SymQ3 = Sym.kBoundedQuotient(Integer(3),t=Integer(1))
>>> km = SymQ3.kmonomial()
>>> km[Integer(2),Integer(1)]*km[Integer(2),Integer(1)]
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
>>> F = SymQ3.affineSchur()
>>> F[Integer(2),Integer(1)]*F[Integer(2),Integer(1)]
2*F3[1, 1, 1, 1, 1, 1] + 4*F3[2, 1, 1, 1, 1] + 4*F3[2, 2, 1, 1] + 4*F3[2, 2, 2]
+ 2*F3[3, 1, 1, 1] + 4*F3[3, 2, 1] + 2*F3[3, 3]
```

When t is not equal to 1, the subspace spanned by the k-Schur functions is realized as a quotient of the ring of symmetric functions by the ideal generated by the Hall-Littlewood symmetric functions in the P basis indexed by partitions with first part greater than k.:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: SymQ3 = Sym.kBoundedQuotient(3)
sage: kHLP = SymQ3.kHallLittlewoodP()
sage: kHLP[2,1]*kHLP[2,1]
(t^2+2*t+1)*HLP3[2, 2, 1, 1] + (t^3+2*t^2+2*t+1)*HLP3[2, 2, 2]
+ (-t^4-t^3+t+1)*HLP3[3, 1, 1, 1] + (-t^2+t+2)*HLP3[3, 2, 1] + (t+1)*HLP3[3, 3]
sage: HLP = Sym.hall_littlewood().P()
sage: kHLP(HLP[3,1])
HLP3[3, 1]
sage: kHLP(HLP[4])
0
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> SymQ3 = Sym.kBoundedQuotient(Integer(3))
>>> kHLP = SymQ3.kHallLittlewoodP()
>>> kHLP[Integer(2),Integer(1)]*kHLP[Integer(2),Integer(1)]
(t^2+2*t+1)*HLP3[2, 2, 1, 1] + (t^3+2*t^2+2*t+1)*HLP3[2, 2, 2]
+ (-t^4-t^3+t+1)*HLP3[3, 1, 1, 1] + (-t^2+t+2)*HLP3[3, 2, 1] + (t+1)*HLP3[3, 3]
>>> HLP = Sym.hall_littlewood().P()
>>> kHLP(HLP[Integer(3),Integer(1)])
HLP3[3, 1]
>>> kHLP(HLP[Integer(4)])
0
```

In this space, the basis which is dual to the k-Schur functions conjecturally expands positively in the k-bounded Hall-Littlewood functions and has positive structure coefficients.:

```
sage: dks = SymQ3.dual_k_Schur()
sage: kHLP(dks[2,2])
(t^4+t^2)*HLP3[1, 1, 1, 1] + t*HLP3[2, 1, 1] + HLP3[2, 2]
sage: dks[2,1]*dks[1,1]
(t^2+t)*dks3[1, 1, 1, 1, 1] + (t+1)*dks3[2, 1, 1, 1] + (t+1)*dks3[2, 2, 1]
+ dks3[3, 1, 1] + dks3[3, 2]
```

```
>>> from sage.all import *
>>> dks = SymQ3.dual_k_Schur()
>>> kHLP(dks[Integer(2),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
(t^4+t^2)*HLP3[1, 1, 1, 1] + t*HLP3[2, 1, 1] + HLP3[2, 2]
>>> dks[Integer(2),Integer(1)]*dks[Integer(1),Integer(1)]
(t^2+t)*dks3[1, 1, 1, 1, 1] + (t+1)*dks3[2, 1, 1, 1] + (t+1)*dks3[2, 2, 1]
+ dks3[3, 1, 1] + dks3[3, 2]
```

At $t = 1$ the k-bounded Hall-Littlewood basis is equal to the k-bounded monomial basis and the dual k-Schur elements are equal to the affine Schur basis. The k-bounded monomial basis and affine Schur functions are faster and should be used instead of the k-bounded Hall-Littlewood P basis and dual k-Schur functions when $t = 1$.:

```
sage: SymQ3 = Sym.kBoundedQuotient(3,t=1)
sage: dks = SymQ3.dual_k_Schur()
sage: F = SymQ3.affineSchur()
sage: F[3,1]==dks[3,1]
True
```

```
>>> from sage.all import *
>>> SymQ3 = Sym.kBoundedQuotient(Integer(3),t=Integer(1))
>>> dks = SymQ3.dual_k_Schur()
>>> F = SymQ3.affineSchur()
>>> F[Integer(3),Integer(1)]==dks[Integer(3),Integer(1)]
True
```

Implementing new bases

In order to implement a new symmetric function basis, Sage will need to know at a minimum how to change back and forth between at least one other basis (although they do not necessarily have to be the same basis). All of the standard functions associated with the basis will have a default implementation (although a more specific implementation may be more efficient).

To present an idea of how this is done, we will create here the example of how to implement the basis $s_\mu[X(1-t)]$.

To begin, we import the class *sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic()*. Our new basis will inherit all of the default methods from this class:

```
sage: from sage.combinat.sf.sfa import SymmetricFunctionAlgebra_generic as SFA_
↪generic
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import SymmetricFunctionAlgebra_generic as SFA_
↪generic
```

Now the basis we are creating has a parameter t which is possible to specialize. In this example we will convert to and from the Schur basis. For this we implement methods `_self_to_s` and `_s_to_self`. By registering these two functions as coercions, Sage then knows automatically how it possible to change between any two bases for which there is a path of changes of bases.

```
sage: from sage.categories.morphism import SetMorphism
sage: class SFA_st(SFA_generic):
....:     def __init__(self, Sym, t):
....:         SFA_generic.__init__(self, Sym, basis_name=
....:           "Schur functions with a plethystic substitution of X -> X(1-t)",
....:           prefix='st')
....:         self._s = Sym.s()
```

(continues on next page)

(continued from previous page)

```
....:         self.t = Sym.base_ring()(t)
....:         cat = HopfAlgebras(Sym.base_ring()).WithBasis()
....:         self.register_coercion(
....:           SetMorphism(Hom(self._s, self, cat), self._s_to_self))
....:         self._s.register_coercion(
....:           SetMorphism(Hom(self, self._s, cat), self._self_to_s))
....:     def _s_to_self(self, f):
....:         # f is a Schur function and the output is in the st basis
....:         return self._from_dict(f.theta_qt(0,self.t)._monomial_coefficients)
....:     def _self_to_s(self, f):
....:         # f is in the st basis and the output is in the Schur basis
....:         return self._s.sum(cmu*self._s(mu).theta_qt(self.t,0) for mu,cmu in␣
↪f)
....:     class Element(SFA_generic.Element):
....:         pass
```

```
>>> from sage.all import *
>>> from sage.categories.morphism import SetMorphism
>>> class SFA_st(SFA_generic):
...     def __init__(self, Sym, t):
...         SFA_generic.__init__(self, Sym, basis_name=
...           "Schur functions with a plethystic substitution of X -> X(1-t)",
...           prefix='st')
...         self._s = Sym.s()
...         self.t = Sym.base_ring()(t)
...         cat = HopfAlgebras(Sym.base_ring()).WithBasis()
...         self.register_coercion(
...           SetMorphism(Hom(self._s, self, cat), self._s_to_self))
...         self._s.register_coercion(
...           SetMorphism(Hom(self, self._s, cat), self._self_to_s))
...     def _s_to_self(self, f):
...         # f is a Schur function and the output is in the st basis
...         return self._from_dict(f.theta_qt(Integer(0),self.t)._monomial_
↪coefficients)
...     def _self_to_s(self, f):
...         # f is in the st basis and the output is in the Schur basis
...         return self._s.sum(cmu*self._s(mu).theta_qt(self.t,Integer(0)) for mu,
↪cmu in f)
...     class Element(SFA_generic.Element):
...         pass
```

An instance of this basis is created by calling it with a symmetric function ring `Sym` and a parameter `t` which is in the base ring of `Sym`. The `Element` class inherits all of the methods from `sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element`.

In the reference [MAC] on page 354, this basis is denoted $S_\lambda(x;t)$ and the change of basis coefficients of the Macdonald `J` basis are the coefficients $K_{\lambda\mu}(q,t)$. Here is an example of its use:

```
sage: QQqt = QQ['q','t'].fraction_field()
sage: (q,t) = QQqt.gens()
sage: st = SFA_st(SymmetricFunctions(QQqt),t)
sage: st
```

(continues on next page)

(continued from previous page)

```
Symmetric Functions over Fraction Field of Multivariate Polynomial
 Ring in q, t over Rational Field in the Schur functions with a
 plethystic substitution of X -> X(1-t) basis
sage: st[2,1] * st[1]
st[2, 1, 1] + st[2, 2] + st[3, 1]
sage: st([2]).coproduct()
st[] # st[2] + st[1] # st[1] + st[2] # st[]
sage: J = st.symmetric_function_ring().macdonald().J()
sage: st(J[2,1])
q*st[1, 1, 1] + (q*t+1)*st[2, 1] + t*st[3]
```

```
>>> from sage.all import *
>>> QQqt = QQ['q','t'].fraction_field()
>>> (q,t) = QQqt.gens()
>>> st = SFA_st(SymmetricFunctions(QQqt),t)
>>> st
Symmetric Functions over Fraction Field of Multivariate Polynomial
 Ring in q, t over Rational Field in the Schur functions with a
 plethystic substitution of X -> X(1-t) basis
>>> st[Integer(2),Integer(1)] * st[Integer(1)]
st[2, 1, 1] + st[2, 2] + st[3, 1]
>>> st([Integer(2)]).coproduct()
st[] # st[2] + st[1] # st[1] + st[2] # st[]
>>> J = st.symmetric_function_ring().macdonald().J()
>>> st(J[Integer(2),Integer(1)])
q*st[1, 1, 1] + (q*t+1)*st[2, 1] + t*st[3]
```

Acknowledgements

The design is heavily inspired from the implementation of symmetric functions in MuPAD-Combinat (see [HT04] and [FD06]).

REFERENCES:

Further tests

Todo

- Introduce fields with degree 1 elements as in MuPAD-Combinat, to get proper plethysm.
- Use UniqueRepresentation to get rid of all the manual cache handling for the bases
- Devise a mechanism so that pickling bases of symmetric functions pickles the coercions which have a cache.

Schur()

The Schur basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

Witt (*coerce_h=None*, *coerce_e=None*, *coerce_p=None*)

The Witt basis of the symmetric functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
```

a_realization ()

Return a particular realization of `self` (the Schur basis).

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sym.a_realization()
Symmetric Functions over Rational Field in the Schur basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> Sym.a_realization()
Symmetric Functions over Rational Field in the Schur basis
```

complete ()

The complete basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

e ()

The elementary basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).elementary()
Symmetric Functions over Rational Field in the elementary basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).elementary()
Symmetric Functions over Rational Field in the elementary basis
```

elementary()

The elementary basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).elementary()
Symmetric Functions over Rational Field in the elementary basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).elementary()
Symmetric Functions over Rational Field in the elementary basis
```

f()

The forgotten basis of the Symmetric Functions (or the basis dual to the elementary basis with respect to the Hall scalar product).

EXAMPLES:

```
sage: SymmetricFunctions(QQ).forgotten()
Symmetric Functions over Rational Field in the forgotten basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).forgotten()
Symmetric Functions over Rational Field in the forgotten basis
```

forgotten()

The forgotten basis of the Symmetric Functions (or the basis dual to the elementary basis with respect to the Hall scalar product).

EXAMPLES:

```
sage: SymmetricFunctions(QQ).forgotten()
Symmetric Functions over Rational Field in the forgotten basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).forgotten()
Symmetric Functions over Rational Field in the forgotten basis
```

from_polynomial(f)

Convert a symmetric polynomial `f` to a symmetric function.

INPUT:

- `f` – a symmetric polynomial

This function converts a symmetric polynomial f in a polynomial ring in finitely many variables to a symmetric function in the monomial basis of the ring of symmetric functions over the same base ring.

EXAMPLES:

```
sage: P = PolynomialRing(QQ, 'x', 3)
sage: x = P.gens()
sage: f = x[0] + x[1] + x[2]
sage: S = SymmetricFunctions(QQ)
sage: S.from_polynomial(f)
m[1]

sage: f = x[0] + 2*x[1] + x[2]
sage: S.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0 + 2*x1 + x2 is not a symmetric polynomial
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ, 'x', Integer(3))
>>> x = P.gens()
>>> f = x[Integer(0)] + x[Integer(1)] + x[Integer(2)]
>>> S = SymmetricFunctions(QQ)
>>> S.from_polynomial(f)
m[1]

>>> f = x[Integer(0)] + Integer(2)*x[Integer(1)] + x[Integer(2)]
>>> S.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0 + 2*x1 + x2 is not a symmetric polynomial
```

h()

The complete basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

hall_littlewood(*t='t'*)

Return the entry point for the various Hall-Littlewood bases.

INPUT:

- `t` – parameter

Hall-Littlewood symmetric functions including bases P, Q, Qp. The Hall-Littlewood P and Q functions at $t = -1$ are the Schur-P and Schur-Q functions when indexed by strict partitions.

The parameter t must be in the base ring of parent.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: P = Sym.hall_littlewood().P(); P
```

(continues on next page)

(continued from previous page)

```
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood P basis
sage: P[2]
HLP[2]
sage: Q = Sym.hall_littlewood().Q(); Q
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood Q basis
sage: Q[2]
HLQ[2]
sage: Qp = Sym.hall_littlewood().Qp(); Qp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood Qp basis
sage: Qp[2]
HLQp[2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> P = Sym.hall_littlewood().P(); P
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood P basis
>>> P[Integer(2)]
HLP[2]
>>> Q = Sym.hall_littlewood().Q(); Q
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood Q basis
>>> Q[Integer(2)]
HLQ[2]
>>> Qp = Sym.hall_littlewood().Qp(); Qp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Hall-Littlewood Qp basis
>>> Qp[Integer(2)]
HLQp[2]
```

hecke_character(*q='q'*)

The basis of symmetric functions that determines the character tables for Hecke algebras.

EXAMPLES:

```
sage: SymmetricFunctions(ZZ['q'].fraction_field()).hecke_character()
Symmetric Functions over
 Fraction Field of Univariate Polynomial Ring in q over Integer Ring
 in the Hecke character with q=q basis
sage: SymmetricFunctions(QQ).hecke_character(1/2)
Symmetric Functions over Rational Field in the Hecke character with q=1/2␣
↪basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(ZZ['q'].fraction_field()).hecke_character()
Symmetric Functions over
 Fraction Field of Univariate Polynomial Ring in q over Integer Ring
 in the Hecke character with q=q basis
>>> SymmetricFunctions(QQ).hecke_character(Integer(1)/Integer(2))
```

(continues on next page)

(continued from previous page)

```
Symmetric Functions over Rational Field in the Hecke character with q=1/2␣
↪basis
```

homogeneous()

The complete basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

ht()

The induced trivial character basis of the Symmetric Functions.

The trivial character of

$$S_{n-|\lambda|} \times S_{\lambda_1} \times S_{\lambda_2} \times \cdots \times S_{\lambda_{\ell(\lambda)}}$$

induced to the group S_n is a symmetric function in the eigenvalues of a permutation matrix. This basis is that character.

It has the property that if the element indexed by the partition λ is evaluated at the roots of a permutation of cycle structure ρ then the value is the coefficient $\langle h_{(n-|\lambda|,\lambda)}, p_\rho \rangle$.

In terms of methods that are implemented in Sage, if `n` is a sufficiently large integer, then `ht(lam).character_to_frobenius_image(n)` is equal the complete function indexed by `[n-sum(lam)]+lam`.

This basis is introduced in [OZ2015].

See also

character_to_frobenius_image(), *eval_at_permutation_roots()*

EXAMPLES:

```
sage: SymmetricFunctions(QQ).induced_trivial_character()
Symmetric Functions over Rational Field in the induced trivial symmetric␣
↪group character basis
sage: ht = SymmetricFunctions(QQ).ht()
sage: h = SymmetricFunctions(QQ).h()
sage: h(ht([3,2]).character_to_frobenius_image(9))
h[4, 3, 2]
sage: h(ht([3,2]).character_to_frobenius_image(7))
h[3, 2, 2]
sage: h(ht([3,2]).character_to_frobenius_image(5))
h[3, 2]
sage: h(ht([3,2]).character_to_frobenius_image(4))
0
sage: p = SymmetricFunctions(QQ).p()
sage: [h([4,1]).scalar(p(rho)) for rho in Partitions(5)]
```

(continues on next page)

(continued from previous page)

```
[0, 1, 0, 2, 1, 3, 5]
sage: [ht([1]).eval_at_permutation_roots(rho) for rho in Partitions(5)]
[0, 1, 0, 2, 1, 3, 5]
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).induced_trivial_character()
Symmetric Functions over Rational Field in the induced trivial symmetric␣
↪group character basis
>>> ht = SymmetricFunctions(QQ).ht()
>>> h = SymmetricFunctions(QQ).h()
>>> h(ht([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(9)))
h[4, 3, 2]
>>> h(ht([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(7)))
h[3, 2, 2]
>>> h(ht([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(5)))
h[3, 2]
>>> h(ht([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(4)))
0
>>> p = SymmetricFunctions(QQ).p()
>>> [h([Integer(4),Integer(1)]).scalar(p(rho)) for rho in␣
↪Partitions(Integer(5))]
[0, 1, 0, 2, 1, 3, 5]
>>> [ht([Integer(1)]).eval_at_permutation_roots(rho) for rho in␣
↪Partitions(Integer(5))]
[0, 1, 0, 2, 1, 3, 5]
```

induced_trivial_character()

The induced trivial character basis of the Symmetric Functions.

The trivial character of

$$S_{n-|\lambda|} \times S_{\lambda_1} \times S_{\lambda_2} \times \cdots \times S_{\lambda_{\ell(\lambda)}}$$

induced to the group S_n is a symmetric function in the eigenvalues of a permutation matrix. This basis is that character.

It has the property that if the element indexed by the partition λ is evaluated at the roots of a permutation of cycle structure ρ then the value is the coefficient $\langle h_{(n-|\lambda|,\lambda)}, p_\rho \rangle$.

In terms of methods that are implemented in Sage, if `n` is a sufficiently large integer, then `ht(lam).character_to_frobenius_image(n)` is equal the complete function indexed by `[n-sum(lam)]+lam`.

This basis is introduced in [OZ2015].

See also

character_to_frobenius_image(), *eval_at_permutation_roots()*

EXAMPLES:

```
sage: SymmetricFunctions(QQ).induced_trivial_character()
Symmetric Functions over Rational Field in the induced trivial symmetric␣
↪group character basis
```

(continues on next page)

(continued from previous page)

```
sage: ht = SymmetricFunctions(QQ).ht()
sage: h = SymmetricFunctions(QQ).h()
sage: h(ht([3,2]).character_to_frobenius_image(9))
h[4, 3, 2]
sage: h(ht([3,2]).character_to_frobenius_image(7))
h[3, 2, 2]
sage: h(ht([3,2]).character_to_frobenius_image(5))
h[3, 2]
sage: h(ht([3,2]).character_to_frobenius_image(4))
0
sage: p = SymmetricFunctions(QQ).p()
sage: [h([4,1]).scalar(p(rho)) for rho in Partitions(5)]
[0, 1, 0, 2, 1, 3, 5]
sage: [ht([1]).eval_at_permutation_roots(rho) for rho in Partitions(5)]
[0, 1, 0, 2, 1, 3, 5]
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).induced_trivial_character()
Symmetric Functions over Rational Field in the induced trivial symmetric␣
↪group character basis
>>> ht = SymmetricFunctions(QQ).ht()
>>> h = SymmetricFunctions(QQ).h()
>>> h(ht([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(9)))
h[4, 3, 2]
>>> h(ht([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(7)))
h[3, 2, 2]
>>> h(ht([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(5)))
h[3, 2]
>>> h(ht([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(4)))
0
>>> p = SymmetricFunctions(QQ).p()
>>> [h([Integer(4),Integer(1)]).scalar(p(rho)) for rho in␣
↪Partitions(Integer(5))]
[0, 1, 0, 2, 1, 3, 5]
>>> [ht([Integer(1)]).eval_at_permutation_roots(rho) for rho in␣
↪Partitions(Integer(5))]
[0, 1, 0, 2, 1, 3, 5]
```

irreducible_symmetric_group_character()

The irreducible S_n character basis of the Symmetric Functions.

This basis has the property that if the element indexed by the partition λ is evaluated at the roots of a permutation of cycle structure ρ then the value is the irreducible character $\chi^{(|\rho|-|\lambda|,\lambda)}(\rho)$.

In terms of methods that are implemented in Sage, if `n` is a sufficiently large integer, then `st(lam).character_to_frobenius_image(n)` is equal the Schur function indexed by `[n-sum(lam)]+lam`.

This basis is introduced in [OZ2015].

See also

character_to_frobenius_image(), *eval_at_permutation_roots()*

EXAMPLES:

```
sage: SymmetricFunctions(QQ).irreducible_symmetric_group_character()
Symmetric Functions over Rational Field in the irreducible symmetric group␣
↪character basis
sage: st = SymmetricFunctions(QQ).st()
sage: s = SymmetricFunctions(QQ).s()
sage: s(st([3,2]).character_to_frobenius_image(9))
s[4, 3, 2]
sage: s(st([3,2]).character_to_frobenius_image(7))
0
sage: s(st([3,2]).character_to_frobenius_image(6))
-s[2, 2, 2]
sage: list(SymmetricGroup(5).character_table()[-2])
[4, 2, 0, 1, -1, 0, -1]
sage: list(reversed([st([1]).eval_at_permutation_roots(rho)
....:    for rho in Partitions(5)]))
[4, 2, 0, 1, -1, 0, -1]
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).irreducible_symmetric_group_character()
Symmetric Functions over Rational Field in the irreducible symmetric group␣
↪character basis
>>> st = SymmetricFunctions(QQ).st()
>>> s = SymmetricFunctions(QQ).s()
>>> s(st([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(9)))
s[4, 3, 2]
>>> s(st([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(7)))
0
>>> s(st([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(6)))
-s[2, 2, 2]
>>> list(SymmetricGroup(Integer(5)).character_table()[-Integer(2)])
[4, 2, 0, 1, -1, 0, -1]
>>> list(reversed([st([Integer(1)]).eval_at_permutation_roots(rho)
...    for rho in Partitions(Integer(5))]))
[4, 2, 0, 1, -1, 0, -1]
```

jack (*t='t'*)

Return the entry point for the various Jack bases.

INPUT:

- `t` – parameter

Jack symmetric functions including bases P, Q, Qp.

The parameter t must be in the base ring of parent.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P(); JP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack P basis
sage: JQ = Sym.jack().Q(); JQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
```

(continues on next page)

(continued from previous page)

```
↪over Rational Field in the Jack Q basis
sage: JJ = Sym.jack().J(); JJ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack J basis
sage: JQp = Sym.jack().Qp(); JQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack Qp basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> JP = Sym.jack().P(); JP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack P basis
>>> JQ = Sym.jack().Q(); JQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack Q basis
>>> JJ = Sym.jack().J(); JJ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack J basis
>>> JQp = Sym.jack().Qp(); JQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the Jack Qp basis
```

kBoundedQuotient(*k*, *t='t'*)

Return the k-bounded quotient space of the ring of symmetric functions.

INPUT:

- k – positive integer

The quotient of the ring of symmetric functions …

See also

sage.combinat.sf.k_dual.KBoundedQuotient()

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: KQ = Sym.kBoundedQuotient(3); KQ
Traceback (most recent call last):
...
TypeError: unable to convert 't' to a rational
sage: KQ = Sym.kBoundedQuotient(3,t=1); KQ
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: KQ = Sym.kBoundedQuotient(3); KQ
3-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
```

(continues on next page)

(continued from previous page)

```
>>> KQ = Sym.kBoundedQuotient(Integer(3)); KQ
Traceback (most recent call last):
...
TypeError: unable to convert 't' to a rational
>>> KQ = Sym.kBoundedQuotient(Integer(3),t=Integer(1)); KQ
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1
>>> Sym = SymmetricFunctions(QQ['t'].fraction_field())
>>> KQ = Sym.kBoundedQuotient(Integer(3)); KQ
3-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate␣
↪Polynomial Ring in t over Rational Field
```

kBoundedSubspace(k, t='t')

Return the k-bounded subspace of the ring of symmetric functions.

INPUT:

- `k` – positive integer
- `t` – a formal parameter; $t = 1$ yields a subring

The subspace of the ring of symmetric functions spanned by $\{s_\lambda[X/(1-t)]\}_{\lambda_1\le k} = \{s_\lambda^{(k)}[X,t]\}_{\lambda_1\le k}$ over the base ring $\mathbf{Q}[t]$. When $t = 1$, this space is in fact a subalgebra of the ring of symmetric functions generated by the complete homogeneous symmetric functions h_i for $1 \le i \le k$.

See also

sage.combinat.sf.new_kschur.KBoundedSubspace()

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: KB = Sym.kBoundedSubspace(3,1); KB
3-bounded Symmetric Functions over Rational Field with t=1

sage: Sym = SymmetricFunctions(QQ['t'])
sage: Sym.kBoundedSubspace(3)
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over␣
↪Rational Field

sage: Sym = SymmetricFunctions(QQ['z'])
sage: z = Sym.base_ring().gens()[0]
sage: Sym.kBoundedSubspace(3,t=z)
3-bounded Symmetric Functions over Univariate Polynomial Ring in z over␣
↪Rational Field with t=z
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> KB = Sym.kBoundedSubspace(Integer(3),Integer(1)); KB
3-bounded Symmetric Functions over Rational Field with t=1

>>> Sym = SymmetricFunctions(QQ['t'])
>>> Sym.kBoundedSubspace(Integer(3))
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over␣
```

(continues on next page)

(continued from previous page)

```
↪Rational Field

>>> Sym = SymmetricFunctions(QQ['z'])
>>> z = Sym.base_ring().gens()[Integer(0)]
>>> Sym.kBoundedSubspace(Integer(3),t=z)
3-bounded Symmetric Functions over Univariate Polynomial Ring in z over␣
↪Rational Field with t=z
```

khomogeneous(*k*)

Return the homogeneous symmetric functions in the k-bounded subspace.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: kh = Sym.khomogeneous(4)
sage: kh[3]*kh[4]
h4[4, 3]
sage: kh[4].lift()
h[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> kh = Sym.khomogeneous(Integer(4))
>>> kh[Integer(3)]*kh[Integer(4)]
h4[4, 3]
>>> kh[Integer(4)].lift()
h[4]
```

kschur(*k*, *t='t'*)

Return the k-Schur functions.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: ks[2]*ks[2]
ks3[2, 2] + ks3[3, 1]
sage: ks[2,1,1].lift()
s[2, 1, 1] + s[3, 1]

sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks = Sym.kschur(3)
sage: ks[2,2,1].lift()
s[2, 2, 1] + t*s[3, 2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ks = Sym.kschur(Integer(3),Integer(1))
>>> ks[Integer(2)]*ks[Integer(2)]
ks3[2, 2] + ks3[3, 1]
>>> ks[Integer(2),Integer(1),Integer(1)].lift()
s[2, 1, 1] + s[3, 1]
```

(continues on next page)

(continued from previous page)

```
>>> Sym = SymmetricFunctions(QQ['t'])
>>> ks = Sym.kschur(Integer(3))
>>> ks[Integer(2),Integer(2),Integer(1)].lift()
s[2, 2, 1] + t*s[3, 2]
```

ksplit(*k*, *t='t'*)

Return the k-split basis of the k-bounded subspace.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ksp = Sym.ksplit(3,1)
sage: ksp[2]*ksp[2]
ksp3[2, 2] + ksp3[3, 1]
sage: ksp[2,1,1].lift()
s[2, 1, 1] + s[2, 2] + s[3, 1]

sage: Sym = SymmetricFunctions(QQ['t'])
sage: ksp = Sym.ksplit(3)
sage: ksp[2,1,1].lift()
s[2, 1, 1] + t*s[2, 2] + t*s[3, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> ksp = Sym.ksplit(Integer(3),Integer(1))
>>> ksp[Integer(2)]*ksp[Integer(2)]
ksp3[2, 2] + ksp3[3, 1]
>>> ksp[Integer(2),Integer(1),Integer(1)].lift()
s[2, 1, 1] + s[2, 2] + s[3, 1]

>>> Sym = SymmetricFunctions(QQ['t'])
>>> ksp = Sym.ksplit(Integer(3))
>>> ksp[Integer(2),Integer(1),Integer(1)].lift()
s[2, 1, 1] + t*s[2, 2] + t*s[3, 1]
```

llt(*k*, *t='t'*)

The LLT symmetric functions.

INPUT:

- `k` – positive integer indicating the level
- `t` – a parameter (default: t)

LLT polynomials in $hspin$ and $hcospin$ bases.

EXAMPLES:

```
sage: llt3 = SymmetricFunctions(QQ['t'].fraction_field()).llt(3); llt3
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in␣
↪t over Rational Field
sage: llt3.hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT spin basis
```

(continues on next page)

(continued from previous page)

```
sage: llt3.hcospin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT cospin basis
sage: llt3.hcospin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT cospin basis
```

```
>>> from sage.all import *
>>> llt3 = SymmetricFunctions(QQ['t'].fraction_field()).llt(Integer(3)); llt3
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in␣
↪t over Rational Field
>>> llt3.hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT spin basis
>>> llt3.hcospin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT cospin basis
>>> llt3.hcospin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t␣
↪over Rational Field in the level 3 LLT cospin basis
```

m()

The monomial basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).monomial()
Symmetric Functions over Rational Field in the monomial basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).monomial()
Symmetric Functions over Rational Field in the monomial basis
```

macdonald(*q='q'*, *t='t'*)

Return the entry point for the various Macdonald bases.

INPUT:

- `q`, `t` – parameters

Macdonald symmetric functions including bases P, Q, J, H, Ht. This also contains the S basis which is dual to the Schur basis with respect to the q,t scalar product.

The parameters q and t must be in the base_ring of parent.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,␣
↪t over Rational Field in the Macdonald P basis
sage: P[2]
McdP[2]
sage: Q = Sym.macdonald().Q(); Q
```

(continues on next page)

(continued from previous page)

```
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,
↪t over Rational Field in the Macdonald Q basis
sage: S = Sym.macdonald().S()
sage: s = Sym.schur()
sage: matrix([[S(la).scalar_qt(s(mu)) for la in Partitions(3)] for mu in
↪Partitions(3)])
[1 0 0]
[0 1 0]
[0 0 1]
sage: H = Sym.macdonald().H()
sage: s(H[2,2])
q^2*s[1, 1, 1, 1] + (q^2*t+q*t+q)*s[2, 1, 1] + (q^2*t^2+1)*s[2, 2] + (q*t^
↪2+q*t+t)*s[3, 1] + t^2*s[4]

sage: Sym = SymmetricFunctions(QQ['z','q'].fraction_field())
sage: (z,q) = Sym.base_ring().gens()
sage: Hzq = Sym.macdonald(q=z,t=q).H()
sage: H1z = Sym.macdonald(q=1,t=z).H()
sage: s = Sym.schur()
sage: s(H1z([2,2]))
s[1, 1, 1, 1] + (2*z+1)*s[2, 1, 1] + (z^2+1)*s[2, 2] + (z^2+2*z)*s[3, 1] + z^
↪2*s[4]
sage: s(Hzq[2,2])
z^2*s[1, 1, 1, 1] + (z^2*q+z*q+z)*s[2, 1, 1] + (z^2*q^2+1)*s[2, 2] + (z*q^
↪2+z*q+q)*s[3, 1] + q^2*s[4]
sage: s(H1z(Hzq[2,2]))
z^2*s[1, 1, 1, 1] + (z^2*q+z*q+z)*s[2, 1, 1] + (z^2*q^2+1)*s[2, 2] + (z*q^
↪2+z*q+q)*s[3, 1] + q^2*s[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,
↪t over Rational Field in the Macdonald P basis
>>> P[Integer(2)]
McdP[2]
>>> Q = Sym.macdonald().Q(); Q
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q,
↪t over Rational Field in the Macdonald Q basis
>>> S = Sym.macdonald().S()
>>> s = Sym.schur()
>>> matrix([[S(la).scalar_qt(s(mu)) for la in Partitions(Integer(3))] for mu
↪in Partitions(Integer(3))])
[1 0 0]
[0 1 0]
[0 0 1]
>>> H = Sym.macdonald().H()
>>> s(H[Integer(2),Integer(2)])
q^2*s[1, 1, 1, 1] + (q^2*t+q*t+q)*s[2, 1, 1] + (q^2*t^2+1)*s[2, 2] + (q*t^
↪2+q*t+t)*s[3, 1] + t^2*s[4]

>>> Sym = SymmetricFunctions(QQ['z','q'].fraction_field())
```

(continues on next page)

(continued from previous page)

```
>>> (z,q) = Sym.base_ring().gens()
>>> Hzq = Sym.macdonald(q=z,t=q).H()
>>> H1z = Sym.macdonald(q=Integer(1),t=z).H()
>>> s = Sym.schur()
>>> s(H1z([Integer(2),Integer(2)]))
s[1, 1, 1, 1] + (2*z+1)*s[2, 1, 1] + (z^2+1)*s[2, 2] + (z^2+2*z)*s[3, 1] + z^
↪2*s[4]
>>> s(Hzq[Integer(2),Integer(2)])
z^2*s[1, 1, 1, 1] + (z^2*q+z*q+z)*s[2, 1, 1] + (z^2*q^2+1)*s[2, 2] + (z*q^
↪2+z*q+q)*s[3, 1] + q^2*s[4]
>>> s(H1z(Hzq[Integer(2),Integer(2)]))
z^2*s[1, 1, 1, 1] + (z^2*q+z*q+z)*s[2, 1, 1] + (z^2*q^2+1)*s[2, 2] + (z*q^
↪2+z*q+q)*s[3, 1] + q^2*s[4]
```

monomial()

The monomial basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).monomial()
Symmetric Functions over Rational Field in the monomial basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).monomial()
Symmetric Functions over Rational Field in the monomial basis
```

o()

The orthogonal basis of the symmetric functions.

See also

SymmetricFunctionAlgebra_orthogonal

EXAMPLES:

```
sage: SymmetricFunctions(QQ).orthogonal()
Symmetric Functions over Rational Field in the orthogonal basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).orthogonal()
Symmetric Functions over Rational Field in the orthogonal basis
```

orthogonal()

The orthogonal basis of the symmetric functions.

See also

SymmetricFunctionAlgebra_orthogonal

EXAMPLES:

```
sage: SymmetricFunctions(QQ).orthogonal()
Symmetric Functions over Rational Field in the orthogonal basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).orthogonal()
Symmetric Functions over Rational Field in the orthogonal basis
```

p()

The power sum basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

power()

The power sum basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

powersum()

The power sum basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

qbar(*q='q'*)

The basis of symmetric functions that determines the character tables for Hecke algebras.

EXAMPLES:

```
sage: SymmetricFunctions(ZZ['q'].fraction_field()).hecke_character()
Symmetric Functions over
 Fraction Field of Univariate Polynomial Ring in q over Integer Ring
 in the Hecke character with q=q basis
sage: SymmetricFunctions(QQ).hecke_character(1/2)
```

(continues on next page)

(continued from previous page)

```
Symmetric Functions over Rational Field in the Hecke character with q=1/2␣
↪basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(ZZ['q'].fraction_field()).hecke_character()
Symmetric Functions over
 Fraction Field of Univariate Polynomial Ring in q over Integer Ring
 in the Hecke character with q=q basis
>>> SymmetricFunctions(QQ).hecke_character(Integer(1)/Integer(2))
Symmetric Functions over Rational Field in the Hecke character with q=1/2␣
↪basis
```

register_isomorphism(*morphism*, *only_conversion=False*)

Register an isomorphism between two bases of `self`, as a canonical coercion (unless the optional keyword `only_conversion` is set to `True`, in which case the isomorphism is registered as conversion only).

EXAMPLES:

We override the canonical coercion from the Schur basis to the powersum basis by a (stupid!) map $s_\lambda \mapsto 2p_\lambda$.

```
sage: Sym = SymmetricFunctions(QQ['zorglub']) # make sure we are not going to␣
↪screw up later tests
sage: s = Sym.s(); p = Sym.p().dual_basis()
sage: phi = s.module_morphism(diagonal = lambda t: 2, codomain = p)
sage: phi(s[2, 1])
2*d_p[2, 1]
sage: Sym.register_isomorphism(phi)
sage: p(s[2,1])
2*d_p[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['zorglub']) # make sure we are not going to␣
↪screw up later tests
>>> s = Sym.s(); p = Sym.p().dual_basis()
>>> phi = s.module_morphism(diagonal = lambda t: Integer(2), codomain = p)
>>> phi(s[Integer(2), Integer(1)])
2*d_p[2, 1]
>>> Sym.register_isomorphism(phi)
>>> p(s[Integer(2),Integer(1)])
2*d_p[2, 1]
```

The map is supposed to implement the canonical isomorphism between the two bases. Otherwise, the results will be mathematically wrong, as above. Use with care!

s()

The Schur basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

schur()

The Schur basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

sp()

The symplectic basis of the symmetric functions.

See also

SymmetricFunctionAlgebra_symplectic

EXAMPLES:

```
sage: SymmetricFunctions(QQ).symplectic()
Symmetric Functions over Rational Field in the symplectic basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).symplectic()
Symmetric Functions over Rational Field in the symplectic basis
```

st()

The irreducible S_n character basis of the Symmetric Functions.

This basis has the property that if the element indexed by the partition λ is evaluated at the roots of a permutation of cycle structure ρ then the value is the irreducible character $\chi^{(|\rho|-|\lambda|,\lambda)}(\rho)$.

In terms of methods that are implemented in Sage, if `n` is a sufficiently large integer, then `st(lam).character_to_frobenius_image(n)` is equal the Schur function indexed by `[n-sum(lam)]+lam`.

This basis is introduced in [OZ2015].

See also

character_to_frobenius_image(), *eval_at_permutation_roots()*

EXAMPLES:

```
sage: SymmetricFunctions(QQ).irreducible_symmetric_group_character()
Symmetric Functions over Rational Field in the irreducible symmetric group␣
↪character basis
```

(continues on next page)

(continued from previous page)

```
sage: st = SymmetricFunctions(QQ).st()
sage: s = SymmetricFunctions(QQ).s()
sage: s(st([3,2]).character_to_frobenius_image(9))
s[4, 3, 2]
sage: s(st([3,2]).character_to_frobenius_image(7))
0
sage: s(st([3,2]).character_to_frobenius_image(6))
-s[2, 2, 2]
sage: list(SymmetricGroup(5).character_table()[-2])
[4, 2, 0, 1, -1, 0, -1]
sage: list(reversed([st([1]).eval_at_permutation_roots(rho)
....:   for rho in Partitions(5)]))
[4, 2, 0, 1, -1, 0, -1]
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).irreducible_symmetric_group_character()
Symmetric Functions over Rational Field in the irreducible symmetric group␣
↪character basis
>>> st = SymmetricFunctions(QQ).st()
>>> s = SymmetricFunctions(QQ).s()
>>> s(st([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(9)))
s[4, 3, 2]
>>> s(st([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(7)))
0
>>> s(st([Integer(3),Integer(2)]).character_to_frobenius_image(Integer(6)))
-s[2, 2, 2]
>>> list(SymmetricGroup(Integer(5)).character_table()[-Integer(2)])
[4, 2, 0, 1, -1, 0, -1]
>>> list(reversed([st([Integer(1)]).eval_at_permutation_roots(rho)
...   for rho in Partitions(Integer(5))]))
[4, 2, 0, 1, -1, 0, -1]
```

symplectic()

The symplectic basis of the symmetric functions.

See also

SymmetricFunctionAlgebra_symplectic

EXAMPLES:

```
sage: SymmetricFunctions(QQ).symplectic()
Symmetric Functions over Rational Field in the symplectic basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).symplectic()
Symmetric Functions over Rational Field in the symplectic basis
```

w(*coerce_h=None*, *coerce_e=None*, *coerce_p=None*)

The Witt basis of the symmetric functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
```

witt(*coerce_h=None*, *coerce_e=None*, *coerce_p=None*)

The Witt basis of the symmetric functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
```

zonal()

The zonal basis of the Symmetric Functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).zonal()
Symmetric Functions over Rational Field in the zonal basis
```

```
>>> from sage.all import *
>>> SymmetricFunctions(QQ).zonal()
Symmetric Functions over Rational Field in the zonal basis
```

class `sage.combinat.sf.sf.`**SymmetricaConversionOnBasis**(*t*, *domain*, *codomain*)

Bases: `object`

Initialization of `self`.

INPUT:

- `t` – a function taking a monomial in CombinatorialFreeModule(QQ, Partitions()), and returning a (partition, coefficient) list
- `domain`, `codomain` – parents

Construct a function mapping a partition to an element of `codomain`.

This is a temporary quick hack to wrap around the existing symmetrica conversions, without changing their specs.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['x'])
sage: p = Sym.p(); s = Sym.s()
sage: def t(x) : [(p, c)] = x; return [ (p, 2*c), (p.conjugate(), c) ]
sage: f = sage.combinat.sf.sf.SymmetricaConversionOnBasis(t, p, s)
sage: f(Partition([3,1]))
s[2, 1, 1] + 2*s[3, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['x'])
>>> p = Sym.p(); s = Sym.s()
>>> def t(x) : [(p, c)] = x; return [ (p, Integer(2)*c), (p.conjugate(), c) ]
>>> f = sage.combinat.sf.sf.SymmetricaConversionOnBasis(t, p, s)
>>> f(Partition([Integer(3),Integer(1)]))
s[2, 1, 1] + 2*s[3, 1]
```

5.1.303 Symmetric Functions

For a comprehensive tutorial on how to use symmetric functions in Sage

See also

SymmetricFunctions()

We define the algebra of symmetric functions in the Schur and elementary bases:

```
sage: s = SymmetricFunctions(QQ).schur()
sage: e = SymmetricFunctions(QQ).elementary()
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).schur()
>>> e = SymmetricFunctions(QQ).elementary()
```

Each is actually a graded Hopf algebra whose basis is indexed by integer partitions:

```
sage: s.category()
Category of graded bases of Symmetric Functions over Rational Field
sage: s.basis().keys()
Partitions
```

```
>>> from sage.all import *
>>> s.category()
Category of graded bases of Symmetric Functions over Rational Field
>>> s.basis().keys()
Partitions
```

Let us compute with some elements in different bases:

```
sage: f1 = s([2,1]); f1
s[2, 1]
sage: f2 = e(f1); f2 # basis conversion
e[2, 1] - e[3]
sage: f1 == f2
True
sage: f1.expand(3, alphabet=['x','y','z'])
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
sage: f2.expand(3, alphabet=['x','y','z'])
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

```
>>> from sage.all import *
>>> f1 = s([Integer(2),Integer(1)]); f1
s[2, 1]
>>> f2 = e(f1); f2 # basis conversion
e[2, 1] - e[3]
>>> f1 == f2
True
>>> f1.expand(Integer(3), alphabet=['x','y','z'])
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
>>> f2.expand(Integer(3), alphabet=['x','y','z'])
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: m([3,1])
m[3, 1]
sage: m(4) # This is the constant 4, not the partition 4.
4*m[]
sage: m([4]) # This is the partition 4.
m[4]
sage: 3*m([3,1])-1/2*m([4])
3*m[3, 1] - 1/2*m[4]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).monomial()
>>> m([Integer(3),Integer(1)])
m[3, 1]
>>> m(Integer(4)) # This is the constant 4, not the partition 4.
4*m[]
>>> m([Integer(4)]) # This is the partition 4.
m[4]
>>> Integer(3)*m([Integer(3),Integer(1)])-Integer(1)/Integer(2)*m([Integer(4)])
3*m[3, 1] - 1/2*m[4]
```

```
sage: p = SymmetricFunctions(QQ).power()
sage: f = p(3)
sage: f
3*p[]
sage: f.parent()
Symmetric Functions over Rational Field in the powersum basis
sage: f + p([3,2])
3*p[] + p[3, 2]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).power()
>>> f = p(Integer(3))
>>> f
3*p[]
>>> f.parent()
Symmetric Functions over Rational Field in the powersum basis
>>> f + p([Integer(3),Integer(2)])
3*p[] + p[3, 2]
```

One can convert symmetric functions to symmetric polynomials and vice versa:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.powersum()
sage: h = Sym.homogeneous()
sage: f = h[2,1] + 2*p[3,1]
sage: poly = f.expand(3); poly
2*x0^4 + 2*x0^3*x1 + 2*x0*x1^3 + 2*x1^4 + 2*x0^3*x2 + 2*x1^3*x2 + 2*x0*x2^3 + 2*x1*x2^
↪3 + 2*x2^4
+ x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3 + 2*x0^2*x2 + 3*x0*x1*x2 + 2*x1^2*x2 + 2*x0*x2^
↪2 + 2*x1*x2^2 + x2^3
sage: Sym.from_polynomial(poly)
3*m[1, 1, 1] + 2*m[2, 1] + m[3] + 2*m[3, 1] + 2*m[4]
sage: Sym.from_polynomial(poly) == f
True
sage: g = h[1,1,1,1]
sage: poly = g.expand(3)
sage: Sym.from_polynomial(poly) == g
False
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> p = Sym.powersum()
>>> h = Sym.homogeneous()
>>> f = h[Integer(2),Integer(1)] + Integer(2)*p[Integer(3),Integer(1)]
>>> poly = f.expand(Integer(3)); poly
2*x0^4 + 2*x0^3*x1 + 2*x0*x1^3 + 2*x1^4 + 2*x0^3*x2 + 2*x1^3*x2 + 2*x0*x2^3 + 2*x1*x2^
↪3 + 2*x2^4
+ x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3 + 2*x0^2*x2 + 3*x0*x1*x2 + 2*x1^2*x2 + 2*x0*x2^
↪2 + 2*x1*x2^2 + x2^3
>>> Sym.from_polynomial(poly)
3*m[1, 1, 1] + 2*m[2, 1] + m[3] + 2*m[3, 1] + 2*m[4]
>>> Sym.from_polynomial(poly) == f
True
>>> g = h[Integer(1),Integer(1),Integer(1),Integer(1)]
>>> poly = g.expand(Integer(3))
>>> Sym.from_polynomial(poly) == g
False
```

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: h = Sym.h()
sage: p = Sym.p()
sage: e = Sym.e()
sage: m = Sym.m()
sage: a = s([3,1])
sage: s(a)
s[3, 1]
sage: h(a)
h[3, 1] - h[4]
sage: p(a)
1/8*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] - 1/8*p[2, 2] - 1/4*p[4]
sage: e(a)
```

(continues on next page)

(continued from previous page)

```
e[2, 1, 1] - e[2, 2] - e[3, 1] + e[4]
sage: m(a)
3*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2] + m[3, 1]
sage: a.expand(4)
x0^3*x1 + x0^2*x1^2 + x0*x1^3 + x0^3*x2 + 2*x0^2*x1*x2 + 2*x0*x1^2*x2 + x1^3*x2 + x0^
↪2*x2^2 + 2*x0*x1*x2^2 + x1^2*x2^2 + x0*x2^3 + x1*x2^3 + x0^3*x3 + 2*x0^2*x1*x3 +␣
↪2*x0*x1^2*x3 + x1^3*x3 + 2*x0^2*x2*x3 + 3*x0*x1*x2*x3 + 2*x1^2*x2*x3 + 2*x0*x2^2*x3␣
↪+ 2*x1*x2^2*x3 + x2^3*x3 + x0^2*x3^2 + 2*x0*x1*x3^2 + x1^2*x3^2 + 2*x0*x2*x3^2 +␣
↪2*x1*x2*x3^2 + x2^2*x3^2 + x0*x3^3 + x1*x3^3 + x2*x3^3
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> h = Sym.h()
>>> p = Sym.p()
>>> e = Sym.e()
>>> m = Sym.m()
>>> a = s([Integer(3),Integer(1)])
>>> s(a)
s[3, 1]
>>> h(a)
h[3, 1] - h[4]
>>> p(a)
1/8*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] - 1/8*p[2, 2] - 1/4*p[4]
>>> e(a)
e[2, 1, 1] - e[2, 2] - e[3, 1] + e[4]
>>> m(a)
3*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2] + m[3, 1]
>>> a.expand(Integer(4))
x0^3*x1 + x0^2*x1^2 + x0*x1^3 + x0^3*x2 + 2*x0^2*x1*x2 + 2*x0*x1^2*x2 + x1^3*x2 + x0^
↪2*x2^2 + 2*x0*x1*x2^2 + x1^2*x2^2 + x0*x2^3 + x1*x2^3 + x0^3*x3 + 2*x0^2*x1*x3 +␣
↪2*x0*x1^2*x3 + x1^3*x3 + 2*x0^2*x2*x3 + 3*x0*x1*x2*x3 + 2*x1^2*x2*x3 + 2*x0*x2^2*x3␣
↪+ 2*x1*x2^2*x3 + x2^3*x3 + x0^2*x3^2 + 2*x0*x1*x3^2 + x1^2*x3^2 + 2*x0*x2*x3^2 +␣
↪2*x1*x2*x3^2 + x2^2*x3^2 + x0*x3^3 + x1*x3^3 + x2*x3^3
```

Here are further examples:

```
sage: h(m([1]))
h[1]
sage: h( m([2]) +m([1,1]) )
h[2]
sage: h( m([3]) + m([2,1]) + m([1,1,1]) )
h[3]
sage: h( m([4]) + m([3,1]) + m([2,2]) + m([2,1,1]) + m([1,1,1,1]) )
h[4]
sage: k = 5
sage: h( sum([ m(part) for part in Partitions(k)]) )
h[5]
sage: k = 10
sage: h( sum([ m(part) for part in Partitions(k)]) )
h[10]
```

```
>>> from sage.all import *
>>> h(m([Integer(1)]))
h[1]
>>> h( m([Integer(2)]) +m([Integer(1),Integer(1)]) )
h[2]
>>> h( m([Integer(3)]) + m([Integer(2),Integer(1)]) + m([Integer(1),Integer(1),
↪Integer(1)]) )
h[3]
>>> h( m([Integer(4)]) + m([Integer(3),Integer(1)]) + m([Integer(2),Integer(2)]) +␣
↪m([Integer(2),Integer(1),Integer(1)]) + m([Integer(1),Integer(1),Integer(1),
↪Integer(1)]) )
h[4]
>>> k = Integer(5)
>>> h( sum([ m(part) for part in Partitions(k)]) )
h[5]
>>> k = Integer(10)
>>> h( sum([ m(part) for part in Partitions(k)]) )
h[10]
```

```
sage: P3 = Partitions(3)
sage: P3.list()
[[3], [2, 1], [1, 1, 1]]
sage: m = SymmetricFunctions(QQ).monomial()
sage: f = sum([m(p) for p in P3])
sage: m.get_print_style()
'lex'
sage: f
m[1, 1, 1] + m[2, 1] + m[3]
sage: m.set_print_style('length')
sage: f
m[3] + m[2, 1] + m[1, 1, 1]
sage: m.set_print_style('maximal_part')
sage: f
m[1, 1, 1] + m[2, 1] + m[3]
sage: m.set_print_style('lex')
```

```
>>> from sage.all import *
>>> P3 = Partitions(Integer(3))
>>> P3.list()
[[3], [2, 1], [1, 1, 1]]
>>> m = SymmetricFunctions(QQ).monomial()
>>> f = sum([m(p) for p in P3])
>>> m.get_print_style()
'lex'
>>> f
m[1, 1, 1] + m[2, 1] + m[3]
>>> m.set_print_style('length')
>>> f
m[3] + m[2, 1] + m[1, 1, 1]
>>> m.set_print_style('maximal_part')
>>> f
m[1, 1, 1] + m[2, 1] + m[3]
```

(continues on next page)

(continued from previous page)

```
>>> m.set_print_style('lex')
```

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: m = Sym.m()
sage: m([3])*s([2,1])
2*m[3, 1, 1, 1] + m[3, 2, 1] + 2*m[4, 1, 1] + m[4, 2] + m[5, 1]
sage: s(m([3])*s([2,1]))
s[2, 1, 1, 1, 1] - s[2, 2, 2] - s[3, 3] + s[5, 1]
sage: s(s([2,1])*m([3]))
s[2, 1, 1, 1, 1] - s[2, 2, 2] - s[3, 3] + s[5, 1]
sage: e = Sym.e()
sage: e([4])*e([3])*e([1])
e[4, 3, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> m = Sym.m()
>>> m([Integer(3)])*s([Integer(2),Integer(1)])
2*m[3, 1, 1, 1] + m[3, 2, 1] + 2*m[4, 1, 1] + m[4, 2] + m[5, 1]
>>> s(m([Integer(3)])*s([Integer(2),Integer(1)]))
s[2, 1, 1, 1, 1] - s[2, 2, 2] - s[3, 3] + s[5, 1]
>>> s(s([Integer(2),Integer(1)])*m([Integer(3)]))
s[2, 1, 1, 1, 1] - s[2, 2, 2] - s[3, 3] + s[5, 1]
>>> e = Sym.e()
>>> e([Integer(4)])*e([Integer(3)])*e([Integer(1)])
e[4, 3, 1]
```

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([2,1]) + s([1,1,1])
sage: z.coefficient([2,1])
1
sage: z.length()
2
sage: sorted(z.support())
[[1, 1, 1], [2, 1]]
sage: z.degree()
3
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> z = s([Integer(2),Integer(1)]) + s([Integer(1),Integer(1),Integer(1)])
>>> z.coefficient([Integer(2),Integer(1)])
1
>>> z.length()
2
>>> sorted(z.support())
[[1, 1, 1], [2, 1]]
>>> z.degree()
3
```

AUTHORS:

- Mike Hansen (2007-06-15)
- Nicolas M. Thiery (partial refactoring)
- Mike Zabrocki, Anne Schilling (2012)
- Darij Grinberg (2013) Sym over rings that are not characteristic 0

class sage.combinat.sf.sfa.**FilteredSymmetricFunctionsBases**(*parent_with_realization*)

Bases: Category_realization_of_parent

The category of filtered bases of the ring of symmetric functions.

super_categories()

The super categories of self.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import FilteredSymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = FilteredSymmetricFunctionsBases(Sym)
sage: bases.super_categories()
[Category of bases of Symmetric Functions over Rational Field,
 Category of commutative filtered Hopf algebras with basis over Rational␣
↪Field]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import FilteredSymmetricFunctionsBases
>>> Sym = SymmetricFunctions(QQ)
>>> bases = FilteredSymmetricFunctionsBases(Sym)
>>> bases.super_categories()
[Category of bases of Symmetric Functions over Rational Field,
 Category of commutative filtered Hopf algebras with basis over Rational␣
↪Field]
```

class sage.combinat.sf.sfa.**GradedSymmetricFunctionsBases**(*parent_with_realization*)

Bases: Category_realization_of_parent

The category of graded bases of the ring of symmetric functions.

These are further required to have the property that the basis element indexed by the empty partition is 1.

class ElementMethods

Bases: object

degree_negation()

Return the image of self under the degree negation automorphism of the ring of symmetric functions.

The degree negation is the automorphism which scales every homogeneous element of degree k by $(-1)^k$ (for all k).

Calling degree_negation(self) is equivalent to calling self.parent().degree_negation(self).

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.monomial()
sage: f = 2*m[2,1] + 4*m[1,1] - 5*m[1] - 3*m[[]]
sage: f.degree_negation()
-3*m[] + 5*m[1] + 4*m[1, 1] - 2*m[2, 1]
sage: x = m.zero().degree_negation(); x
0
sage: parent(x) is m
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> m = Sym.monomial()
>>> f = Integer(2)*m[Integer(2),Integer(1)] + Integer(4)*m[Integer(1),
↪Integer(1)] - Integer(5)*m[Integer(1)] - Integer(3)*m[[]]
>>> f.degree_negation()
-3*m[] + 5*m[1] + 4*m[1, 1] - 2*m[2, 1]
>>> x = m.zero().degree_negation(); x
0
>>> parent(x) is m
True
```

degree_zero_coefficient()

Return the degree zero coefficient of `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: f = 2*m[2,1] + 3*m[[]]
sage: f.degree_zero_coefficient()
3
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.monomial()
>>> f = Integer(2)*m[Integer(2),Integer(1)] + Integer(3)*m[[]]
>>> f.degree_zero_coefficient()
3
```

is_unit()

Return whether this element is a unit in the ring.

EXAMPLES:

```
sage: m = SymmetricFunctions(ZZ).monomial()
sage: (2*m[2,1] + m[[]]).is_unit()
False

sage: m = SymmetricFunctions(QQ).monomial()
sage: (3/2*m([])).is_unit()
True
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(ZZ).monomial()
>>> (Integer(2)*m[Integer(2),Integer(1)] + m[[]]).is_unit()
False

>>> m = SymmetricFunctions(QQ).monomial()
>>> (Integer(3)/Integer(2)*m([])).is_unit()
True
```

class ParentMethods

Bases: `object`

antipode_by_coercion(*element*)

The antipode of `element`.

INPUT:

- `element` – element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: s = Sym.s()
sage: e = Sym.e()
sage: h = Sym.h()
sage: (h([]) + h([1])).antipode() # indirect doctest
h[] - h[1]
sage: (s([]) + s([1]) + s[2]).antipode()
s[] - s[1] + s[1, 1]
sage: (p([2]) + p([3])).antipode()
-p[2] - p[3]
sage: (e([2]) + e([3])).antipode()
e[1, 1] - e[1, 1, 1] - e[2] + 2*e[2, 1] - e[3]
sage: f = Sym.f()
sage: f([3,2,1]).antipode()
-f[3, 2, 1] - 4*f[3, 3] - 2*f[4, 2] - 2*f[5, 1] - 6*f[6]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> p = Sym.p()
>>> s = Sym.s()
>>> e = Sym.e()
>>> h = Sym.h()
>>> (h([]) + h([Integer(1)])).antipode() # indirect doctest
h[] - h[1]
>>> (s([]) + s([Integer(1)]) + s[Integer(2)]).antipode()
s[] - s[1] + s[1, 1]
>>> (p([Integer(2)]) + p([Integer(3)])).antipode()
-p[2] - p[3]
>>> (e([Integer(2)]) + e([Integer(3)])).antipode()
e[1, 1] - e[1, 1, 1] - e[2] + 2*e[2, 1] - e[3]
>>> f = Sym.f()
>>> f([Integer(3),Integer(2),Integer(1)]).antipode()
-f[3, 2, 1] - 4*f[3, 3] - 2*f[4, 2] - 2*f[5, 1] - 6*f[6]
```

The antipode is an involution:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: s = Sym.s()
sage: all( s[u].antipode().antipode() == s[u] for u in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> s = Sym.s()
>>> all( s[u].antipode().antipode() == s[u] for u in␣
↪Partitions(Integer(4)) )
True
```

The antipode is an algebra homomorphism:

```
sage: Sym = SymmetricFunctions(FiniteField(23))
sage: h = Sym.h()
sage: all( all( (s[u] * s[v]).antipode() == s[u].antipode() * s[v].
↪antipode()
....:            for u in Partitions(3) )
....:       for v in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FiniteField(Integer(23)))
>>> h = Sym.h()
>>> all( all( (s[u] * s[v]).antipode() == s[u].antipode() * s[v].
↪antipode()
...            for u in Partitions(Integer(3)) )
...       for v in Partitions(Integer(3)) )
True
```

counit(*element*)

Return the counit of `element`.

The counit is the constant term of `element`.

INPUT:

- `element` – element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: f = 2*m[2,1] + 3*m[[]]
sage: f.counit()
3
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.monomial()
>>> f = Integer(2)*m[Integer(2),Integer(1)] + Integer(3)*m[[]]
>>> f.counit()
3
```

degree_negation(*element*)

Return the image of `element` under the degree negation automorphism of the ring of symmetric functions.

The degree negation is the automorphism which scales every homogeneous element of degree k by $(-1)^k$ (for all k).

INPUT:

- `element` – symmetric function written in `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.monomial()
sage: f = 2*m[2,1] + 4*m[1,1] - 5*m[1] - 3*m[[]]
sage: m.degree_negation(f)
-3*m[] + 5*m[1] + 4*m[1, 1] - 2*m[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> m = Sym.monomial()
>>> f = Integer(2)*m[Integer(2),Integer(1)] + Integer(4)*m[Integer(1),
↪Integer(1)] - Integer(5)*m[Integer(1)] - Integer(3)*m[[]]
>>> m.degree_negation(f)
-3*m[] + 5*m[1] + 4*m[1, 1] - 2*m[2, 1]
```

super_categories()

The super categories of `self`.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import GradedSymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = GradedSymmetricFunctionsBases(Sym)
sage: bases.super_categories()
[Category of filtered bases of Symmetric Functions over Rational Field,
 Category of commutative graded Hopf algebras with basis over Rational Field]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import GradedSymmetricFunctionsBases
>>> Sym = SymmetricFunctions(QQ)
>>> bases = GradedSymmetricFunctionsBases(Sym)
>>> bases.super_categories()
[Category of filtered bases of Symmetric Functions over Rational Field,
 Category of commutative graded Hopf algebras with basis over Rational Field]
```

class sage.combinat.sf.sfa.**SymmetricFunctionAlgebra_generic**(*Sym*, *basis_name=None*, *prefix=None*, *graded=True*)

Bases: `CombinatorialFreeModule`

Abstract base class for symmetric function algebras.

Todo

Most of the methods in this class are generic (manipulations of morphisms, …) and should be generalized (or removed)

Element

alias of *SymmetricFunctionAlgebra_generic_Element*

basis_name()

Return the name of the basis of `self`.

This is used for output and, for the classical bases of symmetric functions, to connect this basis with Symmetrica.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: s.basis_name()
'Schur'
sage: p = Sym.p()
sage: p.basis_name()
'powersum'
sage: h = Sym.h()
sage: h.basis_name()
'homogeneous'
sage: e = Sym.e()
sage: e.basis_name()
'elementary'
sage: m = Sym.m()
sage: m.basis_name()
'monomial'
sage: f = Sym.f()
sage: f.basis_name()
'forgotten'
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> s.basis_name()
'Schur'
>>> p = Sym.p()
>>> p.basis_name()
'powersum'
>>> h = Sym.h()
>>> h.basis_name()
'homogeneous'
>>> e = Sym.e()
>>> e.basis_name()
'elementary'
>>> m = Sym.m()
>>> m.basis_name()
'monomial'
>>> f = Sym.f()
>>> f.basis_name()
```

(continues on next page)

(continued from previous page)

```
'forgotten'
```

change_ring(*R*)

Return the base change of `self` to R.

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s.change_ring(QQ)
Symmetric Functions over Rational Field in the Schur basis
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> s.change_ring(QQ)
Symmetric Functions over Rational Field in the Schur basis
```

construction()

Return a pair `(F, R)`, where `F` is a *SymmetricFunctionsFunctor* and R is a ring, such that `F(R)` returns `self`.

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s.construction()
(SymmetricFunctionsFunctor[Schur], Integer Ring)
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> s.construction()
(SymmetricFunctionsFunctor[Schur], Integer Ring)
```

coproduct_by_coercion(*elt*)

Return the coproduct of the element `elt` by coercion to the Schur basis.

INPUT:

- `elt` – an instance of this basis

OUTPUT:

- The image of `elt` under the comultiplication (=coproduct) of the coalgebra of symmetric functions. The result is an element of the tensor squared of the basis `self`.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: m[3,1,1].coproduct()
m[] # m[3, 1, 1] + m[1] # m[3, 1] + m[1, 1] # m[3] + m[3] # m[1, 1] + m[3, 1]
↪# m[1] + m[3, 1, 1] # m[]
sage: m.coproduct_by_coercion(m[2,1])
m[] # m[2, 1] + m[1] # m[2] + m[2] # m[1] + m[2, 1] # m[]
sage: m.coproduct_by_coercion(m[2,1]) == m([2,1]).coproduct()
True
sage: McdH = SymmetricFunctions(QQ['q','t'].fraction_field()).macdonald().H()
sage: McdH[2,1].coproduct()
```

(continues on next page)

(continued from previous page)

```
McdH[] # McdH[2, 1] + ((q^2*t-1)/(q*t-1))*McdH[1] # McdH[1, 1] + ((q*t^2-1)/
↪(q*t-1))*McdH[1] # McdH[2] + ((q^2*t-1)/(q*t-1))*McdH[1, 1] # McdH[1] +␣
↪((q*t^2-1)/(q*t-1))*McdH[2] # McdH[1] + McdH[2, 1] # McdH[]
sage: HLQp = SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood().
↪Qp()
sage: HLQp[2,1].coproduct()
HLQp[] # HLQp[2, 1] + HLQp[1] # HLQp[1, 1] + HLQp[1] # HLQp[2] + HLQp[1, 1] #␣
↪HLQp[1] + HLQp[2] # HLQp[1] + HLQp[2, 1] # HLQp[]
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: LLT = Sym.llt(3)
sage: LLT.cospin([3,2,1]).coproduct()
(t+1)*m[] # m[1, 1] + m[] # m[2] + (t+1)*m[1] # m[1] + (t+1)*m[1, 1] # m[] +␣
↪m[2] # m[]
sage: f = SymmetricFunctions(ZZ).f()
sage: f[3].coproduct()
f[] # f[3] + f[3] # f[]
sage: f[3,2,1].coproduct()
f[] # f[3, 2, 1] + f[1] # f[3, 2] + f[2] # f[3, 1] + f[2, 1] # f[3] + f[3] #␣
↪f[2, 1] + f[3, 1] # f[2] + f[3, 2] # f[1] + f[3, 2, 1] # f[]
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> m[Integer(3),Integer(1),Integer(1)].coproduct()
m[] # m[3, 1, 1] + m[1] # m[3, 1] + m[1, 1] # m[3] + m[3] # m[1, 1] + m[3, 1]
↪# m[1] + m[3, 1, 1] # m[]
>>> m.coproduct_by_coercion(m[Integer(2),Integer(1)])
m[] # m[2, 1] + m[1] # m[2] + m[2] # m[1] + m[2, 1] # m[]
>>> m.coproduct_by_coercion(m[Integer(2),Integer(1)]) == m([Integer(2),
↪Integer(1)]).coproduct()
True
>>> McdH = SymmetricFunctions(QQ['q','t'].fraction_field()).macdonald().H()
>>> McdH[Integer(2),Integer(1)].coproduct()
McdH[] # McdH[2, 1] + ((q^2*t-1)/(q*t-1))*McdH[1] # McdH[1, 1] + ((q*t^2-1)/
↪(q*t-1))*McdH[1] # McdH[2] + ((q^2*t-1)/(q*t-1))*McdH[1, 1] # McdH[1] +␣
↪((q*t^2-1)/(q*t-1))*McdH[2] # McdH[1] + McdH[2, 1] # McdH[]
>>> HLQp = SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood().Qp()
>>> HLQp[Integer(2),Integer(1)].coproduct()
HLQp[] # HLQp[2, 1] + HLQp[1] # HLQp[1, 1] + HLQp[1] # HLQp[2] + HLQp[1, 1] #␣
↪HLQp[1] + HLQp[2] # HLQp[1] + HLQp[2, 1] # HLQp[]
>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> LLT = Sym.llt(Integer(3))
>>> LLT.cospin([Integer(3),Integer(2),Integer(1)]).coproduct()
(t+1)*m[] # m[1, 1] + m[] # m[2] + (t+1)*m[1] # m[1] + (t+1)*m[1, 1] # m[] +␣
↪m[2] # m[]
>>> f = SymmetricFunctions(ZZ).f()
>>> f[Integer(3)].coproduct()
f[] # f[3] + f[3] # f[]
>>> f[Integer(3),Integer(2),Integer(1)].coproduct()
f[] # f[3, 2, 1] + f[1] # f[3, 2] + f[2] # f[3, 1] + f[2, 1] # f[3] + f[3] #␣
↪f[2, 1] + f[3, 1] # f[2] + f[3, 2] # f[1] + f[3, 2, 1] # f[]
```

dual_basis(*scalar=None*, *scalar_name=''*, *basis_name=None*, *prefix=None*)

Return the dual basis of `self` with respect to the scalar product `scalar`.

INPUT:

- `scalar` – a function `zee` from partitions to the base ring which specifies the scalar product by $\langle p_\lambda, p_\lambda \rangle = \mathrm{zee}(\lambda)$. (Independently on the function chosen, the power sum basis will always be orthogonal; the function `scalar` only determines the norms of the basis elements.) If `scalar` is None, then the standard (Hall) scalar product is used.
- `scalar_name` – name of the scalar function
- `prefix` – prefix used to display the basis

EXAMPLES:

The duals of the elementary symmetric functions with respect to the Hall scalar product are the forgotten symmetric functions.

```
sage: e = SymmetricFunctions(QQ).e()
sage: f = e.dual_basis(prefix='f'); f
Dual basis to Symmetric Functions over Rational Field in the elementary basis␣
↪with respect to the Hall scalar product
sage: f([2,1])^2
4*f[2, 2, 1, 1] + 6*f[2, 2, 2] + 2*f[3, 2, 1] + 2*f[3, 3] + 2*f[4, 1, 1] +␣
↪f[4, 2]
sage: f([2,1]).scalar(e([2,1]))
1
sage: f([2,1]).scalar(e([1,1,1]))
0
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> f = e.dual_basis(prefix='f'); f
Dual basis to Symmetric Functions over Rational Field in the elementary basis␣
↪with respect to the Hall scalar product
>>> f([Integer(2),Integer(1)])**Integer(2)
4*f[2, 2, 1, 1] + 6*f[2, 2, 2] + 2*f[3, 2, 1] + 2*f[3, 3] + 2*f[4, 1, 1] +␣
↪f[4, 2]
>>> f([Integer(2),Integer(1)]).scalar(e([Integer(2),Integer(1)]))
1
>>> f([Integer(2),Integer(1)]).scalar(e([Integer(1),Integer(1),Integer(1)]))
0
```

Since the power-sum symmetric functions are orthogonal, their duals with respect to the Hall scalar product are scalar multiples of themselves.

```
sage: p = SymmetricFunctions(QQ).p()
sage: q = p.dual_basis(prefix='q'); q
Dual basis to Symmetric Functions over Rational Field in the powersum basis␣
↪with respect to the Hall scalar product
sage: q([2,1])^2
4*q[2, 2, 1, 1]
sage: p([2,1]).scalar(q([2,1]))
1
sage: p([2,1]).scalar(q([1,1,1]))
0
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> q = p.dual_basis(prefix='q'); q
Dual basis to Symmetric Functions over Rational Field in the powersum basis␣
↪with respect to the Hall scalar product
>>> q([Integer(2),Integer(1)])**Integer(2)
4*q[2, 2, 1, 1]
>>> p([Integer(2),Integer(1)]).scalar(q([Integer(2),Integer(1)]))
1
>>> p([Integer(2),Integer(1)]).scalar(q([Integer(1),Integer(1),Integer(1)]))
0
```

from_polynomial(*poly*, *check=True*)

Convert polynomial to a symmetric function in the monomial basis and then to the basis `self`.

INPUT:

- `poly` – a symmetric polynomial
- `check` – boolean (default: `True`); specifies whether the computation checks that the polynomial is indeed symmetric

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: h = Sym.homogeneous()
sage: f = (h([]) + h([2,1]) + h([3])).expand(3)
sage: h.from_polynomial(f)
h[] + h[2, 1] + h[3]
sage: s = Sym.s()
sage: g = (s([]) + s([2,1])).expand(3); g
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2 + 1
sage: s.from_polynomial(g)
s[] + s[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> h = Sym.homogeneous()
>>> f = (h([]) + h([Integer(2),Integer(1)]) + h([Integer(3)])).
↪expand(Integer(3))
>>> h.from_polynomial(f)
h[] + h[2, 1] + h[3]
>>> s = Sym.s()
>>> g = (s([]) + s([Integer(2),Integer(1)])).expand(Integer(3)); g
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2 + 1
>>> s.from_polynomial(g)
s[] + s[2, 1]
```

get_print_style()

Return the value of the current print style for `self`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.get_print_style()
```

(continues on next page)

(continued from previous page)

```
'lex'
sage: s.set_print_style('length')
sage: s.get_print_style()
'length'
sage: s.set_print_style('lex')
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s.get_print_style()
'lex'
>>> s.set_print_style('length')
>>> s.get_print_style()
'length'
>>> s.set_print_style('lex')
```

prefix()

Return the prefix on the elements of `self`.

EXAMPLES:

```
sage: schur = SymmetricFunctions(QQ).schur()
sage: schur([3,2,1])
s[3, 2, 1]
sage: schur.prefix()
's'
```

```
>>> from sage.all import *
>>> schur = SymmetricFunctions(QQ).schur()
>>> schur([Integer(3),Integer(2),Integer(1)])
s[3, 2, 1]
>>> schur.prefix()
's'
```

product_by_coercion(*left*, *right*)

Return the product of elements `left` and `right` by coercion to the Schur basis.

INPUT:

- `left`, `right` – instances of this basis

OUTPUT: the product of `left` and `right` expressed in the basis `self`

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p.product_by_coercion(p[3,1,1], p[2,2])
p[3, 2, 2, 1, 1]
sage: m = SymmetricFunctions(QQ).m()
sage: m.product_by_coercion(m[2,1],m[1,1]) == m[2,1]*m[1,1]
True
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> p.product_by_coercion(p[Integer(3),Integer(1),Integer(1)], p[Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)])
p[3, 2, 2, 1, 1]
>>> m = SymmetricFunctions(QQ).m()
>>> m.product_by_coercion(m[Integer(2),Integer(1)],m[Integer(1),Integer(1)])␣
↪== m[Integer(2),Integer(1)]*m[Integer(1),Integer(1)]
True
```

set_print_style(*ps*)

Set the value of the current print style to `ps`.

INPUT:

- `ps` – string specifying the printing style

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.get_print_style()
'lex'
sage: s.set_print_style('length')
sage: s.get_print_style()
'length'
sage: s.set_print_style('lex')
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s.get_print_style()
'lex'
>>> s.set_print_style('length')
>>> s.get_print_style()
'length'
>>> s.set_print_style('lex')
```

symmetric_function_ring()

Return the family of symmetric functions associated to the basis `self`.

OUTPUT: an instance of the ring of symmetric functions

EXAMPLES:

```
sage: schur = SymmetricFunctions(QQ).schur()
sage: schur.symmetric_function_ring()
Symmetric Functions over Rational Field
sage: power = SymmetricFunctions(QQ['t']).power()
sage: power.symmetric_function_ring()
Symmetric Functions over Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> schur = SymmetricFunctions(QQ).schur()
>>> schur.symmetric_function_ring()
Symmetric Functions over Rational Field
>>> power = SymmetricFunctions(QQ['t']).power()
>>> power.symmetric_function_ring()
Symmetric Functions over Univariate Polynomial Ring in t over Rational Field
```

transition_matrix(*basis*, *n*)

Return the transition matrix between `self` and `basis` for the homogeneous component of degree `n`.

INPUT:

- `basis` – a basis of the ring of symmetric functions
- `n` – nonnegative integer

OUTPUT:

A matrix of coefficients giving the expansion of the homogeneous degree-n elements of `self` in the degree-n elements of `basis`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: m = SymmetricFunctions(QQ).m()
sage: s.transition_matrix(m,5)
[1 1 1 1 1 1 1]
[0 1 1 2 2 3 4]
[0 0 1 1 2 3 5]
[0 0 0 1 1 3 6]
[0 0 0 0 1 2 5]
[0 0 0 0 0 1 4]
[0 0 0 0 0 0 1]
sage: s.transition_matrix(m,1)
[1]
sage: s.transition_matrix(m,0)
[1]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> m = SymmetricFunctions(QQ).m()
>>> s.transition_matrix(m,Integer(5))
[1 1 1 1 1 1 1]
[0 1 1 2 2 3 4]
[0 0 1 1 2 3 5]
[0 0 0 1 1 3 6]
[0 0 0 0 1 2 5]
[0 0 0 0 0 1 4]
[0 0 0 0 0 0 1]
>>> s.transition_matrix(m,Integer(1))
[1]
>>> s.transition_matrix(m,Integer(0))
[1]
```

```
sage: p = SymmetricFunctions(QQ).p()
sage: s.transition_matrix(p, 4)
[ 1/4  1/3  1/8  1/4 1/24]
[-1/4    0 -1/8  1/4  1/8]
[   0 -1/3  1/4    0 1/12]
[ 1/4    0 -1/8 -1/4  1/8]
[-1/4  1/3  1/8 -1/4 1/24]
sage: StoP = s.transition_matrix(p,4)
sage: a = s([3,1])+5*s([1,1,1,1])-s([4])
```

(continues on next page)

(continued from previous page)

```
sage: a
5*s[1, 1, 1, 1] + s[3, 1] - s[4]
sage: mon = sorted(a.support())
sage: coeffs = [a[i] for i in mon]
sage: coeffs
[5, 1, -1]
sage: mon
[[1, 1, 1, 1], [3, 1], [4]]
sage: cm = matrix([[-1,1,0,0,5]])
sage: cm * StoP
[-7/4  4/3  3/8 -5/4 7/24]
sage: p(a)
7/24*p[1, 1, 1, 1] - 5/4*p[2, 1, 1] + 3/8*p[2, 2] + 4/3*p[3, 1] - 7/4*p[4]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> s.transition_matrix(p, Integer(4))
[ 1/4  1/3  1/8  1/4 1/24]
[-1/4    0 -1/8  1/4  1/8]
[   0 -1/3  1/4    0 1/12]
[ 1/4    0 -1/8 -1/4  1/8]
[-1/4  1/3  1/8 -1/4 1/24]
>>> StoP = s.transition_matrix(p,Integer(4))
>>> a = s([Integer(3),Integer(1)])+Integer(5)*s([Integer(1),Integer(1),
↪Integer(1),Integer(1)])-s([Integer(4)])
>>> a
5*s[1, 1, 1, 1] + s[3, 1] - s[4]
>>> mon = sorted(a.support())
>>> coeffs = [a[i] for i in mon]
>>> coeffs
[5, 1, -1]
>>> mon
[[1, 1, 1, 1], [3, 1], [4]]
>>> cm = matrix([[-Integer(1),Integer(1),Integer(0),Integer(0),Integer(5)]])
>>> cm * StoP
[-7/4  4/3  3/8 -5/4 7/24]
>>> p(a)
7/24*p[1, 1, 1, 1] - 5/4*p[2, 1, 1] + 3/8*p[2, 2] + 4/3*p[3, 1] - 7/4*p[4]
```

```
sage: h = SymmetricFunctions(QQ).h()
sage: e = SymmetricFunctions(QQ).e()
sage: s.transition_matrix(m,7) == h.transition_matrix(s,7).transpose()
True
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> e = SymmetricFunctions(QQ).e()
>>> s.transition_matrix(m,Integer(7)) == h.transition_matrix(s,Integer(7)).
↪transpose()
True
```

```
sage: h.transition_matrix(m, 7) == h.transition_matrix(m, 7).transpose()
True
```

```
>>> from sage.all import *
>>> h.transition_matrix(m, Integer(7)) == h.transition_matrix(m, Integer(7)).
↪transpose()
True
```

```
sage: h.transition_matrix(e, 7) == e.transition_matrix(h, 7)
True
```

```
>>> from sage.all import *
>>> h.transition_matrix(e, Integer(7)) == e.transition_matrix(h, Integer(7))
True
```

```
sage: p.transition_matrix(s, 5)
[ 1 -1  0  1  0 -1  1]
[ 1  0 -1  0  1  0 -1]
[ 1 -1  1  0 -1  1 -1]
[ 1  1 -1  0 -1  1  1]
[ 1  0  1 -2  1  0  1]
[ 1  2  1  0 -1 -2 -1]
[ 1  4  5  6  5  4  1]
```

```
>>> from sage.all import *
>>> p.transition_matrix(s, Integer(5))
[ 1 -1  0  1  0 -1  1]
[ 1  0 -1  0  1  0 -1]
[ 1 -1  1  0 -1  1 -1]
[ 1  1 -1  0 -1  1  1]
[ 1  0  1 -2  1  0  1]
[ 1  2  1  0 -1 -2 -1]
[ 1  4  5  6  5  4  1]
```

```
sage: e.transition_matrix(m,7) == e.transition_matrix(m,7).transpose()
True
```

```
>>> from sage.all import *
>>> e.transition_matrix(m,Integer(7)) == e.transition_matrix(m,Integer(7)).
↪transpose()
True
```

class sage.combinat.sf.sfa.**SymmetricFunctionAlgebra_generic_Element**

Bases: IndexedFreeModuleElement

Class of generic elements for the symmetric function algebra.

adams_operator(*n*)

Return the image of the symmetric function `self` under the n-th Adams operator.

The n-th Adams operator $\mathbf{f}_n$ is defined to be the map from the ring of symmetric functions to itself that sends every symmetric function $P(x_1, x_2, x_3, \ldots)$ to $P(x_1^n, x_2^n, x_3^n, \ldots)$. This operator $\mathbf{f}_n$ is a Hopf algebra endomorphism, and satisfies

$$\mathbf{f}_n m_{(\lambda_1,\lambda_2,\lambda_3,\dots)} = m_{(n\lambda_1,n\lambda_2,n\lambda_3,\dots)}$$

for every partition $(\lambda_1, \lambda_2, \lambda_3, \dots)$ (where m means the monomial basis). Moreover, $\mathbf{f}_n(p_r) = p_{nr}$ for every positive integer r (where p_k denotes the k-th powersum symmetric function).

The n-th Adams operator is also called the n-th Frobenius endomorphism. It is not related to the Frobenius map which connects the ring of symmetric functions with the representation theory of the symmetric group.

The n-th Adams operator is also the n-th Adams operator of the Λ-ring of symmetric functions over the integers.

The n-th Adams operator can also be described via plethysm: Every symmetric function P satisfies $\mathbf{f}_n(P) = p_n \circ P = P \circ p_n$, where p_n is the n-th powersum symmetric function, and $\circ$ denotes (outer) plethysm.

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Adams operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: s[3].adams_operator(2)
-s[3, 3] + s[4, 2] - s[5, 1] + s[6]
sage: m[4,2,1].adams_operator(3)
m[12, 6, 3]
sage: p[4,2,1].adams_operator(3)
p[12, 6, 3]
sage: h[4].adams_operator(2)
h[4, 4] - 2*h[5, 3] + 2*h[6, 2] - 2*h[7, 1] + 2*h[8]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> p = Sym.p()
>>> h = Sym.h()
>>> s = Sym.s()
>>> m = Sym.m()
>>> s[Integer(3)].adams_operator(Integer(2))
-s[3, 3] + s[4, 2] - s[5, 1] + s[6]
>>> m[Integer(4),Integer(2),Integer(1)].adams_operator(Integer(3))
m[12, 6, 3]
>>> p[Integer(4),Integer(2),Integer(1)].adams_operator(Integer(3))
p[12, 6, 3]
>>> h[Integer(4)].adams_operator(Integer(2))
h[4, 4] - 2*h[5, 3] + 2*h[6, 2] - 2*h[7, 1] + 2*h[8]
```

The Adams endomorphisms are multiplicative:

```
sage: all( all( s(lam).adams_operator(3) * s(mu).adams_operator(3) # long time
....:           == (s(lam) * s(mu)).adams_operator(3)
....:           for mu in Partitions(3) )
....:      for lam in Partitions(3) )
True
sage: all( all( m(lam).adams_operator(2) * m(mu).adams_operator(2)
....:           == (m(lam) * m(mu)).adams_operator(2)
....:           for mu in Partitions(4) )
....:      for lam in Partitions(4) )
True
sage: all( all( p(lam).adams_operator(2) * p(mu).adams_operator(2)
....:           == (p(lam) * p(mu)).adams_operator(2)
....:           for mu in Partitions(3) )
....:      for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( s(lam).adams_operator(Integer(3)) * s(mu).adams_
↪operator(Integer(3)) # long time
...           == (s(lam) * s(mu)).adams_operator(Integer(3))
...           for mu in Partitions(Integer(3)) )
...      for lam in Partitions(Integer(3)) )
True
>>> all( all( m(lam).adams_operator(Integer(2)) * m(mu).adams_
↪operator(Integer(2))
...           == (m(lam) * m(mu)).adams_operator(Integer(2))
...           for mu in Partitions(Integer(4)) )
...      for lam in Partitions(Integer(4)) )
True
>>> all( all( p(lam).adams_operator(Integer(2)) * p(mu).adams_
↪operator(Integer(2))
...           == (p(lam) * p(mu)).adams_operator(Integer(2))
...           for mu in Partitions(Integer(3)) )
...      for lam in Partitions(Integer(4)) )
True
```

Being Hopf algebra endomorphisms, the Adams operators commute with the antipode:

```
sage: all( p(lam).adams_operator(4).antipode()
....:      == p(lam).antipode().adams_operator(4)
....:      for lam in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> all( p(lam).adams_operator(Integer(4)).antipode()
...      == p(lam).antipode().adams_operator(Integer(4))
...      for lam in Partitions(Integer(3)) )
True
```

Testing the $\mathbf{f}_n(P) = p_n \circ P = P \circ p_n$ equality (over $\mathbf{Q}$, since plethysm is currently not defined over $\mathbf{Z}$ in Sage):

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: p = Sym.p()
sage: all( s(lam).adams_operator(3) == s(lam).plethysm(p[3])
....:      == s(p[3].plethysm(s(lam)))
....:      for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> p = Sym.p()
>>> all( s(lam).adams_operator(Integer(3)) == s(lam).plethysm(p[Integer(3)])
...      == s(p[Integer(3)].plethysm(s(lam)))
...      for lam in Partitions(Integer(4)) )
True
```

By Exercise 7.61 in Stanley's EC2 [STA] (see the errata on his website), $\mathbf{f}_n(h_m)$ is a linear combination of Schur polynomials (of straight shapes) using coefficients 0, 1 and -1 only; moreover, all partitions whose Schur polynomials occur with coefficient $\neq 0$ in this combination have empty n-cores. Let us check this on examples:

```
sage: all( all( all( (coeff == -1 or coeff == 1)
....:                and lam.core(n) == Partition([])
....:                for lam, coeff in s([m]).adams_operator(n) )
....:           for n in range(2, 4) )
....:      for m in range(4) )
True
```

```
>>> from sage.all import *
>>> all( all( all( (coeff == -Integer(1) or coeff == Integer(1))
...                and lam.core(n) == Partition([])
...                for lam, coeff in s([m]).adams_operator(n) )
...           for n in range(Integer(2), Integer(4)) )
...      for m in range(Integer(4)) )
True
```

See also:

plethysm()

Todo: This method is fast on the monomial and the powersum bases, while all other bases get converted to the monomial basis. For most bases, this is probably the quickest way to do, but at least the Schur basis should have a better option. (Quoting from Stanley's EC2 [STA]: "D. G. Duncan, J. London Math. Soc. 27 (1952), 235-236, or Y. M. Chen, A. M. Garsia, and J. B. Remmel, Contemp. Math. 34 (1984), 109-153".)

arithmetic_product(*x*)

Return the arithmetic product of `self` and `x` in the basis of `self`.

The arithmetic product is a binary operation $\boxdot$ on the ring of symmetric functions which is bilinear in its two arguments and satisfies

$$p_\lambda \boxdot p_\mu = \prod_{i \geq 1, j \geq 1} p_{\mathrm{lcm}(\lambda_i, \mu_j)}^{\mathrm{gcd}(\lambda_i, \mu_j)}$$

for any two partitions $\lambda = (\lambda_1, \lambda_2, \lambda_3, \dots)$ and $\mu = (\mu_1, \mu_2, \mu_3, \dots)$ (where p_ν denotes the power-sum symmetric function indexed by the partition ν, and p_i denotes the i-th power-sum symmetric function). This is enough to define the arithmetic product if the base ring is torsion-free as a **Z**-module; for all other cases the arithmetic product is uniquely determined by requiring it to be functorial in the base ring. See http://mathoverflow.net/questions/138148/ for a discussion of this arithmetic product.

If f and g are two symmetric functions which are homogeneous of degrees a and b, respectively, then $f \boxdot g$ is homogeneous of degree ab.

The arithmetic product is commutative and associative and has unity $e_1 = p_1 = h_1$.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

Arithmetic product of `self` with `x`; this is a symmetric function over the same base ring as `self`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([2]).arithmetic_product(s([2]))
s[1, 1, 1, 1] + 2*s[2, 2] + s[4]
sage: s([2]).arithmetic_product(s([1,1]))
s[2, 1, 1] + s[3, 1]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s([Integer(2)]).arithmetic_product(s([Integer(2)]))
s[1, 1, 1, 1] + 2*s[2, 2] + s[4]
>>> s([Integer(2)]).arithmetic_product(s([Integer(1),Integer(1)]))
s[2, 1, 1] + s[3, 1]
```

The symmetric function `e[1]` is the unity for the arithmetic product:

```
sage: e = SymmetricFunctions(ZZ).e()
sage: all( e([1]).arithmetic_product(e(q)) == e(q) for q in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(ZZ).e()
>>> all( e([Integer(1)]).arithmetic_product(e(q)) == e(q) for q in␣
↪Partitions(Integer(4)) )
True
```

The arithmetic product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).arithmetic_product(m(q)) == m(q).arithmetic_
↪product(e(p))  # long time (26s on sage.math, 2013)
....:           for q in Partitions(4) )
....:       for p in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(19))).e()
>>> m = SymmetricFunctions(FiniteField(Integer(19))).m()
>>> all( all( e(p).arithmetic_product(m(q)) == m(q).arithmetic_product(e(p))
↪# long time (26s on sage.math, 2013)
...           for q in Partitions(Integer(4)) )
...       for p in Partitions(Integer(4)) )
True
```

Note

The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a **Q**-algebra (in which case the arithmetic product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over **Q** (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

bernstein_creation_operator(*n*)

Return the image of `self` under the n-th Bernstein creation operator.

Let n be an integer. The n-th Bernstein creation operator $\mathbf{B}_n$ is defined as the endomorphism of the space Sym of symmetric functions which sends every f to

$$\sum_{i\geq 0}(-1)^i h_{n+i}e_i^\perp,$$

where usual notations are in place (h stands for the complete homogeneous symmetric functions, e for the elementary ones, and $e_i^\perp$ means skewing (*skew_by()*) by e_i).

This has been studied in [BBSSZ2012], section 2.2, where the following rule is given for computing $\mathbf{B}_n$ on a Schur function: If $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ is an n-tuple of integers (positive or not), then

$$\mathbf{B}_n s_{(\alpha_1,\alpha_2,\ldots,\alpha_n)} = s_{(n,\alpha_1,\alpha_2,\ldots,\alpha_n)}.$$

Here, $s_{(\alpha_1,\alpha_2,\ldots,\alpha_n)}$ is the "Schur function" associated to the n-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, and defined by literally applying the Jacobi-Trudi identity, i.e., by

$$s_{(\alpha_1,\alpha_2,\ldots,\alpha_n)} = \det\left((h_{\alpha_i-i+j})_{i,j=1,2,\ldots,n}\right).$$

This notion of a Schur function clearly extends the classical notion of Schur function corresponding to a partition, but is easily reduced to the latter (in fact, for any n-tuple α of integers, one easily sees that s_α is either 0 or minus-plus a Schur function corresponding to a partition; and it is easy to determine which of these is the case and find the partition by a combinatorial algorithm).

EXAMPLES:

Let us check that what this method computes agrees with the definition:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: def bernstein_creation_by_def(n, f):
....:     # `n`-th Bernstein creation operator applied to `f`
....:     # computed according to its definition.
....:     res = f.parent().zero()
....:     if not f:
....:         return res
....:     max_degree = max(sum(m) for m, c in f)
....:     for i in range(max_degree + 1):
....:         if n + i >= 0:
....:             res += (-1) ** i * h[n + i] * f.skew_by(e[i])
....:     return res
sage: all( bernstein_creation_by_def(n, s[l]) == s[l].bernstein_creation_
↪operator(n)
....:      for n in range(-2, 3) for l in Partitions(4) )
True
sage: all( bernstein_creation_by_def(n, s[l]) == s[l].bernstein_creation_
↪operator(n)
....:      for n in range(-3, 4) for l in Partitions(3) )
True
sage: all( bernstein_creation_by_def(n, e[l]) == e[l].bernstein_creation_
↪operator(n)
....:      for n in range(-3, 4) for k in range(3) for l in Partitions(k) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> e = Sym.e()
>>> h = Sym.h()
>>> s = Sym.s()
>>> def bernstein_creation_by_def(n, f):
...     # `n`-th Bernstein creation operator applied to `f`
...     # computed according to its definition.
...     res = f.parent().zero()
...     if not f:
...         return res
...     max_degree = max(sum(m) for m, c in f)
...     for i in range(max_degree + Integer(1)):
...         if n + i >= Integer(0):
...             res += (-Integer(1)) ** i * h[n + i] * f.skew_by(e[i])
...     return res
>>> all( bernstein_creation_by_def(n, s[l]) == s[l].bernstein_creation_
↪operator(n)
...      for n in range(-Integer(2), Integer(3)) for l in␣
↪Partitions(Integer(4)) )
True
>>> all( bernstein_creation_by_def(n, s[l]) == s[l].bernstein_creation_
↪operator(n)
...      for n in range(-Integer(3), Integer(4)) for l in␣
```

(continues on next page)

(continued from previous page)

```
↪Partitions(Integer(3)) )
True
>>> all( bernstein_creation_by_def(n, e[l]) == e[l].bernstein_creation_
↪operator(n)
...      for n in range(-Integer(3), Integer(4)) for k in range(Integer(3))␣
↪for l in Partitions(k) )
True
```

Some examples:

```
sage: s[3,2].bernstein_creation_operator(3)
s[3, 3, 2]
sage: s[3,2].bernstein_creation_operator(1)
-s[2, 2, 2]
sage: h[3,2].bernstein_creation_operator(-2)
h[2, 1]
sage: h[3,2].bernstein_creation_operator(-1)
h[2, 1, 1] - h[2, 2] - h[3, 1]
sage: h[3,2].bernstein_creation_operator(0)
-h[3, 1, 1] + h[3, 2]
sage: h[3,2].bernstein_creation_operator(1)
-h[2, 2, 2] + h[3, 2, 1]
sage: h[3,2].bernstein_creation_operator(2)
-h[3, 3, 1] + h[4, 2, 1]
```

```
>>> from sage.all import *
>>> s[Integer(3),Integer(2)].bernstein_creation_operator(Integer(3))
s[3, 3, 2]
>>> s[Integer(3),Integer(2)].bernstein_creation_operator(Integer(1))
-s[2, 2, 2]
>>> h[Integer(3),Integer(2)].bernstein_creation_operator(-Integer(2))
h[2, 1]
>>> h[Integer(3),Integer(2)].bernstein_creation_operator(-Integer(1))
h[2, 1, 1] - h[2, 2] - h[3, 1]
>>> h[Integer(3),Integer(2)].bernstein_creation_operator(Integer(0))
-h[3, 1, 1] + h[3, 2]
>>> h[Integer(3),Integer(2)].bernstein_creation_operator(Integer(1))
-h[2, 2, 2] + h[3, 2, 1]
>>> h[Integer(3),Integer(2)].bernstein_creation_operator(Integer(2))
-h[3, 3, 1] + h[4, 2, 1]
```

character_to_frobenius_image(n)

Interpret `self` as a GL_n character and then take the Frobenius image of this character of the permutation matrices S_n which naturally sit inside of GL_n.

To know the value of this character at a permutation of cycle structure ρ the symmetric function `self` is evaluated at the eigenvalues of a permutation of cycle structure ρ. The Frobenius image is then defined as $\sum_{\rho \vdash n} f[\Xi_\rho] p_\rho / z_\rho$.

See also

eval_at_permutation_roots()

INPUT:

- `n` – nonnegative integer to interpret `self` as a character of GL_n

OUTPUT: a symmetric function of degree `n`

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([1,1]).character_to_frobenius_image(5)
s[3, 1, 1] + s[4, 1]
sage: s([2,1]).character_to_frobenius_image(5)
s[2, 2, 1] + 2*s[3, 1, 1] + 2*s[3, 2] + 3*s[4, 1] + s[5]
sage: s([2,2,2]).character_to_frobenius_image(3)
s[3]
sage: s([2,2,2]).character_to_frobenius_image(4)
s[2, 2] + 2*s[3, 1] + 2*s[4]
sage: s([2,2,2]).character_to_frobenius_image(5)
2*s[2, 2, 1] + s[3, 1, 1] + 4*s[3, 2] + 3*s[4, 1] + 2*s[5]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s([Integer(1),Integer(1)]).character_to_frobenius_image(Integer(5))
s[3, 1, 1] + s[4, 1]
>>> s([Integer(2),Integer(1)]).character_to_frobenius_image(Integer(5))
s[2, 2, 1] + 2*s[3, 1, 1] + 2*s[3, 2] + 3*s[4, 1] + s[5]
>>> s([Integer(2),Integer(2),Integer(2)]).character_to_frobenius_
↪image(Integer(3))
s[3]
>>> s([Integer(2),Integer(2),Integer(2)]).character_to_frobenius_
↪image(Integer(4))
s[2, 2] + 2*s[3, 1] + 2*s[4]
>>> s([Integer(2),Integer(2),Integer(2)]).character_to_frobenius_
↪image(Integer(5))
2*s[2, 2, 1] + s[3, 1, 1] + 4*s[3, 2] + 3*s[4, 1] + 2*s[5]
```

degree()

Return the degree of `self` (which is defined to be 0 for the zero element).

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1]) + 3
sage: z.degree()
4
sage: s(1).degree()
0
sage: s(0).degree()
0
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> z = s([Integer(4)]) + s([Integer(2),Integer(1)]) + s([Integer(1),
↪Integer(1),Integer(1)]) + s([Integer(1)]) + Integer(3)
>>> z.degree()
```

(continues on next page)

(continued from previous page)

```
4
>>> s(Integer(1)).degree()
0
>>> s(Integer(0)).degree()
0
```

derivative_with_respect_to_p1(*n=1*)

Return the symmetric function obtained by taking the derivative of `self` with respect to the power-sum symmetric function p_1 when the expansion of `self` in the power-sum basis is considered as a polynomial in p_k's (with $k \geq 1$).

This is the same as skewing `self` by the first power-sum symmetric function p_1.

INPUT:

- `n` – (default: 1) nonnegative integer which determines which power of the derivative is taken

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: a = p([1,1,1])
sage: a.derivative_with_respect_to_p1()
3*p[1, 1]
sage: a.derivative_with_respect_to_p1(1)
3*p[1, 1]
sage: a.derivative_with_respect_to_p1(2)
6*p[1]
sage: a.derivative_with_respect_to_p1(3)
6*p[]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).p()
>>> a = p([Integer(1),Integer(1),Integer(1)])
>>> a.derivative_with_respect_to_p1()
3*p[1, 1]
>>> a.derivative_with_respect_to_p1(Integer(1))
3*p[1, 1]
>>> a.derivative_with_respect_to_p1(Integer(2))
6*p[1]
>>> a.derivative_with_respect_to_p1(Integer(3))
6*p[]
```

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([3]).derivative_with_respect_to_p1()
s[2]
sage: s([2,1]).derivative_with_respect_to_p1()
s[1, 1] + s[2]
sage: s([1,1,1]).derivative_with_respect_to_p1()
s[1, 1]
sage: s(0).derivative_with_respect_to_p1()
0
sage: s(1).derivative_with_respect_to_p1()
0
```

(continues on next page)

(continued from previous page)

```
sage: s([1]).derivative_with_respect_to_p1()
s[]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s([Integer(3)]).derivative_with_respect_to_p1()
s[2]
>>> s([Integer(2),Integer(1)]).derivative_with_respect_to_p1()
s[1, 1] + s[2]
>>> s([Integer(1),Integer(1),Integer(1)]).derivative_with_respect_to_p1()
s[1, 1]
>>> s(Integer(0)).derivative_with_respect_to_p1()
0
>>> s(Integer(1)).derivative_with_respect_to_p1()
0
>>> s([Integer(1)]).derivative_with_respect_to_p1()
s[]
```

Let us check that taking the derivative with respect to `p[1]` is equivalent to skewing by `p[1]`:

```
sage: p1 = s([1])
sage: all( s(lam).derivative_with_respect_to_p1()
....:      == s(lam).skew_by(p1) for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> p1 = s([Integer(1)])
>>> all( s(lam).derivative_with_respect_to_p1()
...      == s(lam).skew_by(p1) for lam in Partitions(Integer(4)) )
True
```

eval_at_permutation_roots(*rho*)

Evaluate at eigenvalues of a permutation matrix.

Evaluate a symmetric function at the eigenvalues of a permutation matrix whose cycle structure is `rho`. This computation is computed by coercing to the power sum basis where the value may be computed on the generators.

This function evaluates an element at the roots of unity

$$\Xi_{\rho_1}, \Xi_{\rho_2}, \ldots, \Xi_{\rho_\ell}$$

where

$$\Xi_m = 1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1}$$

and ζ_m is an m root of unity. These roots of unity represent the eigenvalues of permutation matrix with cycle structure ρ.

INPUT:

- `rho` – a partition or a list of nonnegative integers

OUTPUT: an element of the base ring

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([3,3]).eval_at_permutation_roots([6])
0
sage: s([3,3]).eval_at_permutation_roots([3])
1
sage: s([3,3]).eval_at_permutation_roots([1])
0
sage: s([3,3]).eval_at_permutation_roots([3,3])
4
sage: s([3,3]).eval_at_permutation_roots([1,1,1,1,1])
175
sage: (s[1]+s[2]+s[3]).eval_at_permutation_roots([3,2])
2
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(6)])
0
>>> s([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(3)])
1
>>> s([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(1)])
0
>>> s([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(3),
↪Integer(3)])
4
>>> s([Integer(3),Integer(3)]).eval_at_permutation_roots([Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1)])
175
>>> (s[Integer(1)]+s[Integer(2)]+s[Integer(3)]).eval_at_permutation_
↪roots([Integer(3),Integer(2)])
2
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled `x0`, `x1`, …, `x{n-1}` (or just `x` if $n = 1$), where `x` is `alphabet`.

EXAMPLES:

```
sage: J = SymmetricFunctions(QQ).jack(t=2).J()
sage: J([2,1]).expand(3)
4*x0^2*x1 + 4*x0*x1^2 + 4*x0^2*x2 + 6*x0*x1*x2 + 4*x1^2*x2 + 4*x0*x2^2 +␣
↪4*x1*x2^2
sage: (2*J([2])).expand(0)
0
sage: (3*J([])).expand(0)
3
```

```
>>> from sage.all import *
>>> J = SymmetricFunctions(QQ).jack(t=Integer(2)).J()
>>> J([Integer(2),Integer(1)]).expand(Integer(3))
4*x0^2*x1 + 4*x0*x1^2 + 4*x0^2*x2 + 6*x0*x1*x2 + 4*x1^2*x2 + 4*x0*x2^2 +
↪4*x1*x2^2
>>> (Integer(2)*J([Integer(2)])).expand(Integer(0))
0
>>> (Integer(3)*J([])).expand(Integer(0))
3
```

exponential_specialization(*t=None*, *q=1*)

Return the exponential specialization of a symmetric function (when $q = 1$), or the q-exponential specialization (when $q \neq 1$).

The *exponential specialization* ex at t is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R. It is defined whenever the base ring K is a **Q**-algebra and t is an element of R. The easiest way to define it is by specifying its values on the powersum symmetric functions to be $p_1 = t$ and $p_n = 0$ for $n > 1$. Equivalently, on the homogeneous functions it is given by $ex(h_n) = t^n/n!$; see Proposition 7.8.4 of [EnumComb2].

By analogy, the q-exponential specialization is a K-algebra homomorphism from the K-algebra of symmetric functions to another K-algebra R that depends on two elements t and q of R for which the elements $1 - q^i$ for all positive integers i are invertible. It can be defined by specifying its values on the complete homogeneous symmetric functions to be

$$ex_q(h_n) = t^n/[n]_q!,$$

where $[n]_q!$ is the q-factorial. Equivalently, for $q \neq 1$ and a homogeneous symmetric function f of degree n, we have

$$ex_q(f) = (1 - q)^n t^n ps_q(f),$$

where $ps_q(f)$ is the stable principal specialization of f (see `principal_specialization()`). (See (7.29) in [EnumComb2].)

The limit of ex_q as $q \to 1$ is ex.

INPUT:

- `t` – (default: `None`) the value to use for t; the default is to create a ring of polynomials in `t`
- `q` – (default: 1) the value to use for q. If `q` is `None`, then a ring (or fraction field) of polynomials in `q` is created.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: (m[2,1]+m[1,1]).exponential_specialization()
1/2*t^2
sage: (m[2,1]+m[1,1]).exponential_specialization(q=1)
1/2*t^2
sage: m[1,1].exponential_specialization(q=None)
(q/(q + 1))*t^2
sage: Qq = PolynomialRing(QQ, "q"); q = Qq.gen()
sage: m[1,1].exponential_specialization(q=q)
(q/(q + 1))*t^2
sage: Qt = PolynomialRing(QQ, "t"); t = Qt.gen()
```

(continues on next page)

(continued from previous page)

```
sage: m[1,1].exponential_specialization(t=t)
1/2*t^2
sage: Qqt = PolynomialRing(QQ, ["q", "t"]); q, t = Qqt.gens()
sage: m[1,1].exponential_specialization(q=q, t=t)
q*t^2/(q + 1)

sage: x = m[3]+m[2,1]+m[1,1,1]
sage: d = x.homogeneous_degree()
sage: var("q t")                                                        #␣
↪needs sage.symbolic
(q, t)
sage: factor((x.principal_specialization()*(1-q)^d*t^d))               #␣
↪needs sage.symbolic
t^3/((q^2 + q + 1)*(q + 1))
sage: factor(x.exponential_specialization(q=q, t=t))                   #␣
↪needs sage.symbolic
t^3/((q^2 + q + 1)*(q + 1))
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> (m[Integer(2),Integer(1)]+m[Integer(1),Integer(1)]).exponential_
↪specialization()
1/2*t^2
>>> (m[Integer(2),Integer(1)]+m[Integer(1),Integer(1)]).exponential_
↪specialization(q=Integer(1))
1/2*t^2
>>> m[Integer(1),Integer(1)].exponential_specialization(q=None)
(q/(q + 1))*t^2
>>> Qq = PolynomialRing(QQ, "q"); q = Qq.gen()
>>> m[Integer(1),Integer(1)].exponential_specialization(q=q)
(q/(q + 1))*t^2
>>> Qt = PolynomialRing(QQ, "t"); t = Qt.gen()
>>> m[Integer(1),Integer(1)].exponential_specialization(t=t)
1/2*t^2
>>> Qqt = PolynomialRing(QQ, ["q", "t"]); q, t = Qqt.gens()
>>> m[Integer(1),Integer(1)].exponential_specialization(q=q, t=t)
q*t^2/(q + 1)

>>> x = m[Integer(3)]+m[Integer(2),Integer(1)]+m[Integer(1),Integer(1),
↪Integer(1)]
>>> d = x.homogeneous_degree()
>>> var("q t")                                                        #␣
↪needs sage.symbolic
(q, t)
>>> factor((x.principal_specialization()*(Integer(1)-q)**d*t**d))     ␣
↪       # needs sage.symbolic
t^3/((q^2 + q + 1)*(q + 1))
>>> factor(x.exponential_specialization(q=q, t=t))                   #␣
↪needs sage.symbolic
t^3/((q^2 + q + 1)*(q + 1))
```

factor()

Return the factorization of this symmetric function.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: factor((5*e[3] + e[2,1] + e[1])*(7*e[2] + e[5,1]))
(e[1] + e[2, 1] + 5*e[3]) * (7*e[2] + e[5, 1])

sage: R.<x, y> = QQ[]
sage: s = SymmetricFunctions(R.fraction_field()).s()
sage: factor((s[3] + x*s[2,1] + 1)*(3*y*s[2] + s[4,1] + x*y))
(-s[] - x*s[2, 1] - s[3]) * (-x*y*s[] - 3*y*s[2] - s[4, 1])
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> factor((Integer(5)*e[Integer(3)] + e[Integer(2),Integer(1)] +␣
↪e[Integer(1)])*(Integer(7)*e[Integer(2)] + e[Integer(5),Integer(1)]))
(e[1] + e[2, 1] + 5*e[3]) * (7*e[2] + e[5, 1])

>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> s = SymmetricFunctions(R.fraction_field()).s()
>>> factor((s[Integer(3)] + x*s[Integer(2),Integer(1)] +␣
↪Integer(1))*(Integer(3)*y*s[Integer(2)] + s[Integer(4),Integer(1)] + x*y))
(-s[] - x*s[2, 1] - s[3]) * (-x*y*s[] - 3*y*s[2] - s[4, 1])
```

frobenius(**args*, ***kwds*)

Deprecated: Use `adams_operator()` instead. See Issue #36396 for details.

gcd(*other*)

Return the greatest common divisor with `other`.

INPUT:

- `other` – the other symmetric function

EXAMPLES:

```
sage: e = SymmetricFunctions(ZZ).e()
sage: A = 5*e[3] + e[2,1] + e[1]
sage: B = 7*e[2] + e[5,1]
sage: C = 3*e[1,1] + e[2]
sage: gcd(A*B^2, B*C)
7*e[2] + e[5, 1]

sage: p = SymmetricFunctions(ZZ).p()
sage: gcd(e[2,1], p[1,1]-p[2])
e[2]
sage: gcd(p[2,1], p[3,2]-p[2,1])
p[2]
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(ZZ).e()
>>> A = Integer(5)*e[Integer(3)] + e[Integer(2),Integer(1)] + e[Integer(1)]
>>> B = Integer(7)*e[Integer(2)] + e[Integer(5),Integer(1)]
>>> C = Integer(3)*e[Integer(1),Integer(1)] + e[Integer(2)]
```

(continues on next page)

(continued from previous page)

```
>>> gcd(A*B**Integer(2), B*C)
7*e[2] + e[5, 1]

>>> p = SymmetricFunctions(ZZ).p()
>>> gcd(e[Integer(2),Integer(1)], p[Integer(1),Integer(1)]-p[Integer(2)])
e[2]
>>> gcd(p[Integer(2),Integer(1)], p[Integer(3),Integer(2)]-p[Integer(2),
↪Integer(1)])
p[2]
```

hl_creation_operator(*nu*, *t=None*)

This is the vertex operator that generalizes Jing's operator.

It is a linear operator that raises the degree by $|\nu|$. This creation operator is a t-analogue of multiplication by `s(nu)` .

See also

Proposition 5 in [SZ2001].

INPUT:

- `nu` – a partition or a list of integers
- `t` – (default: `None`, in which case `t` is used) an element of the base ring

REFERENCES:

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ['t']).s()
sage: s([2]).hl_creation_operator([3,2])
s[3, 2, 2] + t*s[3, 3, 1] + t*s[4, 2, 1] + t^2*s[4, 3] + t^2*s[5, 2]

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLQp = Sym.hall_littlewood().Qp()
sage: s = Sym.s()
sage: HLQp(s([2]).hl_creation_operator([2]).hl_creation_operator([3]))
HLQp[3, 2, 2]
sage: s([2,2]).hl_creation_operator([2,1])
t*s[2, 2, 2, 1] + t^2*s[3, 2, 1, 1] + t^2*s[3, 2, 2] + t^3*s[3, 3, 1] + t^
↪3*s[4, 2, 1] + t^4*s[4, 3]
sage: s(1).hl_creation_operator([2,1,1])
s[2, 1, 1]
sage: s(0).hl_creation_operator([2,1,1])
0
sage: s([3,2]).hl_creation_operator([2,1,1])
(t^2-t)*s[2, 2, 2, 2, 1] + t^3*s[3, 2, 2, 1, 1]
 + (t^3-t^2)*s[3, 2, 2, 2] + t^3*s[3, 3, 1, 1, 1]
 + t^4*s[3, 3, 2, 1] + t^3*s[4, 2, 1, 1, 1] + t^4*s[4, 2, 2, 1]
 + 2*t^4*s[4, 3, 1, 1] + t^5*s[4, 3, 2] + t^5*s[4, 4, 1]
 + t^4*s[5, 2, 1, 1] + t^5*s[5, 3, 1]
sage: s([3,2]).hl_creation_operator([-2])
```

(continues on next page)

(continued from previous page)

```
(-t^2+t)*s[1, 1, 1] + (-t^2+1)*s[2, 1]
sage: s([3,2]).hl_creation_operator(-2)
Traceback (most recent call last):
...
ValueError: nu must be a list of integers
sage: s = SymmetricFunctions(FractionField(ZZ['t'])).schur()
sage: s[2].hl_creation_operator([3])
s[3, 2] + t*s[4, 1] + t^2*s[5]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ['t']).s()
>>> s([Integer(2)]).hl_creation_operator([Integer(3),Integer(2)])
s[3, 2, 2] + t*s[3, 3, 1] + t*s[4, 2, 1] + t^2*s[4, 3] + t^2*s[5, 2]

>>> Sym = SymmetricFunctions(FractionField(QQ['t']))
>>> HLQp = Sym.hall_littlewood().Qp()
>>> s = Sym.s()
>>> HLQp(s([Integer(2)]).hl_creation_operator([Integer(2)]).hl_creation_
↪operator([Integer(3)]))
HLQp[3, 2, 2]
>>> s([Integer(2),Integer(2)]).hl_creation_operator([Integer(2),Integer(1)])
t*s[2, 2, 2, 1] + t^2*s[3, 2, 1, 1] + t^2*s[3, 2, 2] + t^3*s[3, 3, 1] + t^
↪3*s[4, 2, 1] + t^4*s[4, 3]
>>> s(Integer(1)).hl_creation_operator([Integer(2),Integer(1),Integer(1)])
s[2, 1, 1]
>>> s(Integer(0)).hl_creation_operator([Integer(2),Integer(1),Integer(1)])
0
>>> s([Integer(3),Integer(2)]).hl_creation_operator([Integer(2),Integer(1),
↪Integer(1)])
(t^2-t)*s[2, 2, 2, 2, 1] + t^3*s[3, 2, 2, 1, 1]
 + (t^3-t^2)*s[3, 2, 2, 2] + t^3*s[3, 3, 1, 1, 1]
 + t^4*s[3, 3, 2, 1] + t^3*s[4, 2, 1, 1, 1] + t^4*s[4, 2, 2, 1]
 + 2*t^4*s[4, 3, 1, 1] + t^5*s[4, 3, 2] + t^5*s[4, 4, 1]
 + t^4*s[5, 2, 1, 1] + t^5*s[5, 3, 1]
>>> s([Integer(3),Integer(2)]).hl_creation_operator([-Integer(2)])
(-t^2+t)*s[1, 1, 1] + (-t^2+1)*s[2, 1]
>>> s([Integer(3),Integer(2)]).hl_creation_operator(-Integer(2))
Traceback (most recent call last):
...
ValueError: nu must be a list of integers
>>> s = SymmetricFunctions(FractionField(ZZ['t'])).schur()
>>> s[Integer(2)].hl_creation_operator([Integer(3)])
s[3, 2] + t*s[4, 1] + t^2*s[5]
```

inner_plethysm(*x*)

Return the inner plethysm of `self` with `x`.

Whenever R is a $\mathbf{Q}$-algebra, and f and g are two symmetric functions over R such that the constant term of f is zero, the inner plethysm of f with g is a symmetric function over R, and the degree of this symmetric function is the same as the degree of g. We will denote the inner plethysm of f with g by $f\{g\}$ (in contrast to the notation of outer plethysm which is generally denoted $f[g]$); in Sage syntax, it is `f.inner_plethysm(g)`.

First we describe the axiomatic definition of the operation; see below for a representation-theoretic interpretation. In the following equations, we denote the outer product (i.e., the standard product on the ring of symmetric functions, `product()`) by $\cdot$ and the Kronecker product (`itensor()`) by $*$).

$$(f + g)\{h\} = f\{h\} + g\{h\}$$
$$(f \cdot g)\{h\} = (f\{h\}) * (g\{h\})$$
$$p_k\{f + g\} = p_k\{f\} + p_k\{g\}$$

where p_k is the k-th power-sum symmetric function for every $k > 0$.

Let σ be a permutation of cycle type μ and let μ^k be the cycle type of σ^k. Then,

$$p_k\{p_\mu/z_\mu\} = \sum_{\nu:\nu^k=\mu} p_\nu/z_\nu$$

Since $(p_\mu/z_\mu)_\mu$ is a basis for the symmetric functions, these four formulas define the symmetric function operation $f\{g\}$ for any symmetric functions f and g (where f has constant term 0) by expanding f in the power sum basis and g in the dual basis p_μ/z_μ.

See also

`itensor()`, `partition_power()`, `plethysm()`

This operation admits a representation-theoretic interpretation in the case where f is a Schur function s_λ and g is a homogeneous degree n symmetric function with nonnegative integral coefficients in the Schur basis. The symmetric function $f\{g\}$ is the Frobenius image of the S_n-representation constructed as follows.

The assumptions on g imply that g is the Frobenius image of a representation ρ of the symmetric group S_n:

$$\rho : S_n \to GL_N.$$

If the degree N of this representation is greater than or equal to the number of parts of λ, then f, which denotes s_λ, corresponds to the character of some irreducible GL_N-representation, say

$$\sigma : GL_N \to GL_M.$$

The composition $\sigma \circ \rho : S_n \to GL_M$ is a representation of S_n whose Frobenius image is precisely $f\{g\}$.

If N is less than the number of parts of λ, then $f\{g\}$ is 0 by definition.

When f is a symmetric function with constant term $\neq 0$, the inner plethysm $f\{g\}$ isn't well-defined in the ring of symmetric functions. Indeed, it is not clear how to define $1\{g\}$. The most sensible way to get around this probably is defining it as the infinite sum $h_0 + h_1 + h_2 + \cdots$ (where h_i means the i-th complete homogeneous symmetric function) in the completion of this ring with respect to its grading. This is how [SchaThi1994] defines $1\{g\}$. The present method, however, sets it to be the sum of h_i over all i for which the i-th homogeneous component of g is nonzero. This is rather a hack than a reasonable definition. Use with caution!

Note

If a symmetric function g is written in the form $g = g_0 + g_1 + g_2 + \cdots$ with each g_i homogeneous of degree i, then $f\{g\} = f\{g_0\} + f\{g_1\} + f\{g_2\} + \cdots$ for every f with constant term 0. But in general, inner plethysm is not linear in the second variable.

REFERENCES:

INPUT:

- x – element of the ring of symmetric functions over the same base ring as self

OUTPUT: an element of symmetric functions in the parent of self

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: p = Sym.power()
sage: h = Sym.complete()
sage: s([2,1]).inner_plethysm(s([1,1,1]))
0
sage: s([2]).inner_plethysm(s([2,1]))
s[2, 1] + s[3]
sage: s([1,1]).inner_plethysm(s([2,1]))
s[1, 1, 1]
sage: s[2,1].inner_tensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.schur()
>>> p = Sym.power()
>>> h = Sym.complete()
>>> s([Integer(2),Integer(1)]).inner_plethysm(s([Integer(1),Integer(1),
↪Integer(1)]))
0
>>> s([Integer(2)]).inner_plethysm(s([Integer(2),Integer(1)]))
s[2, 1] + s[3]
>>> s([Integer(1),Integer(1)]).inner_plethysm(s([Integer(2),Integer(1)]))
s[1, 1, 1]
>>> s[Integer(2),Integer(1)].inner_tensor(s[Integer(2),Integer(1)])
s[1, 1, 1] + s[2, 1] + s[3]
```

```
sage: f = s([2,1]) + 2*s([3,1])
sage: f.itensor(f)
s[1, 1, 1] + s[2, 1] + 4*s[2, 1, 1] + 4*s[2, 2] + s[3] + 4*s[3, 1] + 4*s[4]
sage: s( h([1,1]).inner_plethysm(f) )
s[1, 1, 1] + s[2, 1] + 4*s[2, 1, 1] + 4*s[2, 2] + s[3] + 4*s[3, 1] + 4*s[4]
```

```
>>> from sage.all import *
>>> f = s([Integer(2),Integer(1)]) + Integer(2)*s([Integer(3),Integer(1)])
>>> f.itensor(f)
s[1, 1, 1] + s[2, 1] + 4*s[2, 1, 1] + 4*s[2, 2] + s[3] + 4*s[3, 1] + 4*s[4]
>>> s( h([Integer(1),Integer(1)]).inner_plethysm(f) )
s[1, 1, 1] + s[2, 1] + 4*s[2, 1, 1] + 4*s[2, 2] + s[3] + 4*s[3, 1] + 4*s[4]
```

```
sage: s([]).inner_plethysm(s([1,1]) + 2*s([2,1])+s([3]))
s[2] + s[3]
sage: [s([]).inner_plethysm(s(la)) for la in Partitions(4)]
[s[4], s[4], s[4], s[4], s[4]]
sage: s([3]).inner_plethysm(s([]))
s[]
```

(continues on next page)

(continued from previous page)

```
sage: s[1,1,1,1].inner_plethysm(s[2,1])
0
sage: s[1,1,1,1].inner_plethysm(2*s[2,1])
s[3]
```

```
>>> from sage.all import *
>>> s([]).inner_plethysm(s([Integer(1),Integer(1)]) +␣
↪Integer(2)*s([Integer(2),Integer(1)])+s([Integer(3)]))
s[2] + s[3]
>>> [s([]).inner_plethysm(s(la)) for la in Partitions(Integer(4))]
[s[4], s[4], s[4], s[4], s[4]]
>>> s([Integer(3)]).inner_plethysm(s([]))
s[]
>>> s[Integer(1),Integer(1),Integer(1),Integer(1)].inner_
↪plethysm(s[Integer(2),Integer(1)])
0
>>> s[Integer(1),Integer(1),Integer(1),Integer(1)].inner_
↪plethysm(Integer(2)*s[Integer(2),Integer(1)])
s[3]
```

```
sage: p[3].inner_plethysm(p[3])
0
sage: p[3,3].inner_plethysm(p[3])
0
sage: p[3].inner_plethysm(p[1,1,1])
p[1, 1, 1] + 2*p[3]
sage: p[4].inner_plethysm(p[1,1,1,1]/24)
1/24*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] + 1/8*p[2, 2] + 1/4*p[4]
sage: p[3,3].inner_plethysm(p[1,1,1])
6*p[1, 1, 1] + 12*p[3]
```

```
>>> from sage.all import *
>>> p[Integer(3)].inner_plethysm(p[Integer(3)])
0
>>> p[Integer(3),Integer(3)].inner_plethysm(p[Integer(3)])
0
>>> p[Integer(3)].inner_plethysm(p[Integer(1),Integer(1),Integer(1)])
p[1, 1, 1] + 2*p[3]
>>> p[Integer(4)].inner_plethysm(p[Integer(1),Integer(1),Integer(1),
↪Integer(1)]/Integer(24))
1/24*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] + 1/8*p[2, 2] + 1/4*p[4]
>>> p[Integer(3),Integer(3)].inner_plethysm(p[Integer(1),Integer(1),
↪Integer(1)])
6*p[1, 1, 1] + 12*p[3]
```

inner_tensor(*x*)

Return the internal (tensor) product of `self` and `x` in the basis of `self`.

The internal tensor product can be defined as the linear extension of the definition on power sums $p_\lambda * p_\mu = \delta_{\lambda,\mu} z_\lambda p_\lambda$, where $z_\lambda = (1^{r_1} r_1!)(2^{r_2} r_2!)\cdots$ for $\lambda = (1^{r_1} 2^{r_2} \cdots)$ and where $*$ denotes the internal tensor product. The internal tensor product is also known as the Kronecker product, or as the second multiplication on the ring of symmetric functions.

Note that the internal product of any two homogeneous symmetric functions of equal degrees is a homogeneous symmetric function of the same degree. On the other hand, the internal product of two homogeneous symmetric functions of distinct degrees is 0.

Note: The internal product is sometimes referred to as "inner product" in the literature, but unfortunately this name is shared by a different operation, namely the Hall inner product (see *scalar()*).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the internal product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

The methods *itensor()*, *internal_product()*, *kronecker_product()*, *inner_tensor()* are all synonyms.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> a = s([Integer(2),Integer(1)])
>>> b = s([Integer(3)])
>>> a.itensor(b)
s[2, 1]
>>> c = s([Integer(3),Integer(2),Integer(1)])
>>> c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

There are few quantitative results pertaining to Kronecker products in general, which makes their computation so difficult. Let us test a few of them in different bases.

The Kronecker product of any homogeneous symmetric function f of degree n with the n-th complete homogeneous symmetric function `h[n]` (a.k.a. `s[n]`) is f:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: all( h([5]).itensor(h(p)) == h(p) for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(ZZ).h()
>>> all( h([Integer(5)]).itensor(h(p)) == h(p) for p in␣
↪Partitions(Integer(5)) )
True
```

The Kronecker product of a Schur function s_λ with the n-th elementary symmetric function `e[n]`, where $n = |\lambda|$, is $s_{\lambda'}$ (where λ' is the conjugate partition of λ):

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: all( e([5]).itensor(s(p)) == s(p.conjugate()) for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> F = CyclotomicField(Integer(12))
>>> s = SymmetricFunctions(F).s()
>>> e = SymmetricFunctions(F).e()
>>> all( e([Integer(5)]).itensor(s(p)) == s(p.conjugate()) for p in␣
↪Partitions(Integer(5)) )
True
```

The Kronecker product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in␣
↪Partitions(4) )
....:      for p in Partitions(4) )
True

sage: F = FractionField(QQ['q','t'])
sage: mq = SymmetricFunctions(F).macdonald().Q()
sage: mh = SymmetricFunctions(F).macdonald().H()
sage: all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
....:           for r in Partitions(4) )
....:      for p in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(19))).e()
>>> m = SymmetricFunctions(FiniteField(Integer(19))).m()
>>> all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in␣
↪Partitions(Integer(4)) )
...      for p in Partitions(Integer(4)) )
True

>>> F = FractionField(QQ['q','t'])
>>> mq = SymmetricFunctions(F).macdonald().Q()
>>> mh = SymmetricFunctions(F).macdonald().H()
>>> all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
...           for r in Partitions(Integer(4)) )
```

(continues on next page)

(continued from previous page)

```
...       for p in Partitions(Integer(3)) )
True
```

Let us check (on examples) Proposition 5.2 of Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, "Noncommutative symmetric functions", arXiv hep-th/9407124, for $r = 2$:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w))␣
↪with
....:                            # * meaning Kronecker product and \mu␣
↪meaning the
....:                            # usual multiplication.
....:     result = w.parent().zero()
....:     for partition_pair, coeff in w.coproduct():
....:         result += coeff * w.parent()(u).itensor(partition_pair[0]) * w.
↪parent()(v).itensor(partition_pair[1])
....:     return result
sage: all( all( all( tensor_copr(e[u], s[v], m[w])   # long time
....:                == (e[u] * s[v]).itensor(m[w])
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:      for u in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(29))).e()
>>> s = SymmetricFunctions(FiniteField(Integer(29))).s()
>>> m = SymmetricFunctions(FiniteField(Integer(29))).m()
>>> def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w)) with
...                            # * meaning Kronecker product and \mu meaning␣
↪the
...                            # usual multiplication.
...     result = w.parent().zero()
...     for partition_pair, coeff in w.coproduct():
...         result += coeff * w.parent()(u).itensor(partition_
↪pair[Integer(0)]) * w.parent()(v).itensor(partition_pair[Integer(1)])
...     return result
>>> all( all( all( tensor_copr(e[u], s[v], m[w])   # long time
...                == (e[u] * s[v]).itensor(m[w])
...                for w in Partitions(Integer(5)) )
...           for v in Partitions(Integer(2)) )
...      for u in Partitions(Integer(3)) )
True
```

Some examples from Briand, Orellana, Rosas, "The stability of the Kronecker products of Schur functions." arXiv 0907.4652:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s[2,2].itensor(s[2,2])
s[1, 1, 1, 1] + s[2, 2] + s[4]
```

(continues on next page)

(continued from previous page)

```
sage: s[3,2].itensor(s[3,2])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s[4,2].itensor(s[4,2])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1]
↪+ s[6]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> s[Integer(2),Integer(2)].itensor(s[Integer(2),Integer(2)])
s[1, 1, 1, 1] + s[2, 2] + s[4]
>>> s[Integer(3),Integer(2)].itensor(s[Integer(3),Integer(2)])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
>>> s[Integer(4),Integer(2)].itensor(s[Integer(4),Integer(2)])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1]
↪+ s[6]
```

An example from p. 220 of Thibon, “Hopf algebras of symmetric functions and tensor products of symmetric group representations”, International Journal of Algebra and Computation, 1991:

```
sage: s = SymmetricFunctions(QQbar).s()
sage: s[2,1].itensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQbar).s()
>>> s[Integer(2),Integer(1)].itensor(s[Integer(2),Integer(1)])
s[1, 1, 1] + s[2, 1] + s[3]
```

Note

The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a $\mathbf{Q}$-algebra (in which case the Kronecker product can be easily computed using the power sum basis) from the case where it isn’t. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the “corresponding” basis of the symmetric function algebra over $\mathbf{Q}$ (using the `corresponding_basis_over` hack) from the case when it isn’t (in which case it transforms everything into the Schur basis, which is slow).

internal_coproduct()

Return the inner coproduct of `self` in the basis of `self`.

The inner coproduct (also known as the Kronecker coproduct, as the internal coproduct, or as the second comultiplication on the ring of symmetric functions) is a ring homomorphism $\Delta^\times$ from the ring of symmetric functions to the tensor product (over the base ring) of this ring with itself. It is uniquely characterized by the formula

$$\Delta^\times(h_n) = \sum_{\lambda \vdash n} s_\lambda \otimes s_\lambda = \sum_{\lambda \vdash n} h_\lambda \otimes m_\lambda = \sum_{\lambda \vdash n} m_\lambda \otimes h_\lambda,$$

where $\lambda \vdash n$ means λ is a partition of n, and n is any nonnegative integer. It also satisfies

$$\Delta^\times(p_n) = p_n \otimes p_n$$

for any positive integer n. If the base ring is a $\mathbf{Q}$-algebra, it also satisfies

$$\Delta^{\times}(h_n) = \sum_{\lambda \vdash n} z_{\lambda}^{-1} p_{\lambda} \otimes p_{\lambda},$$

where

$$z_{\lambda} = \prod_{i=1}^{\infty} i^{m_i(\lambda)} m_i(\lambda)!$$

with $m_i(\lambda)$ meaning the number of appearances of i in λ (see `zee()`).

The method `kronecker_coproduct()` is a synonym of `internal_coproduct()`.

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([2,1])
sage: a.internal_coproduct()
s[1, 1, 1] # s[2, 1] + s[2, 1] # s[1, 1, 1] + s[2, 1] # s[2, 1] + s[2, 1] #
↪s[3] + s[3] # s[2, 1]

sage: e = SymmetricFunctions(QQ).e()
sage: b = e([2])
sage: b.internal_coproduct()
e[1, 1] # e[2] + e[2] # e[1, 1] - 2*e[2] # e[2]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> a = s([Integer(2),Integer(1)])
>>> a.internal_coproduct()
s[1, 1, 1] # s[2, 1] + s[2, 1] # s[1, 1, 1] + s[2, 1] # s[2, 1] + s[2, 1] #
↪s[3] + s[3] # s[2, 1]

>>> e = SymmetricFunctions(QQ).e()
>>> b = e([Integer(2)])
>>> b.internal_coproduct()
e[1, 1] # e[2] + e[2] # e[1, 1] - 2*e[2] # e[2]
```

The internal coproduct is adjoint to the internal product with respect to the Hall inner product: Any three symmetric functions f, g and h satisfy $\langle f * g, h \rangle = \sum_i \langle f, h'_i \rangle \langle g, h''_i \rangle$, where we write $\Delta^{\times}(h)$ as $\sum_i h'_i \otimes h''_i$. Let us check this in degree 4:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \
↪left< g, h''_i \right>
....:     result = h.base_ring().zero()
....:     for partition_pair, coeff in h.internal_coproduct():
....:         result += coeff * h.parent()(f).scalar(partition_pair[0]) * h.
↪parent()(g).scalar(partition_pair[1])
....:     return result
sage: all( all( all( tensor_incopr(e[u], s[v], m[w]) == (e[u].itensor(s[v])).
↪scalar(m[w])  # long time (10s on sage.math, 2013)
```

(continues on next page)

(continued from previous page)

```
....:                 for w in Partitions(5) )
....:            for v in Partitions(2) )
....:       for u in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(29))).e()
>>> s = SymmetricFunctions(FiniteField(Integer(29))).s()
>>> m = SymmetricFunctions(FiniteField(Integer(29))).m()
>>> def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \
↪left< g, h''_i \right>
...     result = h.base_ring().zero()
...     for partition_pair, coeff in h.internal_coproduct():
...         result += coeff * h.parent()(f).scalar(partition_
↪pair[Integer(0)]) * h.parent()(g).scalar(partition_pair[Integer(1)])
...     return result
>>> all( all( all( tensor_incopr(e[u], s[v], m[w]) == (e[u].itensor(s[v])).
↪scalar(m[w])  # long time (10s on sage.math, 2013)
...                 for w in Partitions(Integer(5)) )
...            for v in Partitions(Integer(2)) )
...       for u in Partitions(Integer(3)) )
True
```

Let us check the formulas for $\Delta^{\times}(h_n)$ and $\Delta^{\times}(p_n)$ given in the description of this method:

```
sage: e = SymmetricFunctions(QQ).e()
sage: p = SymmetricFunctions(QQ).p()
sage: h = SymmetricFunctions(QQ).h()
sage: s = SymmetricFunctions(QQ).s()
sage: all( s(h([n])).internal_coproduct() == sum([tensor([s(lam), s(lam)])␣
↪for lam in Partitions(n)])
....:       for n in range(6) )
True
sage: all( h([n]).internal_coproduct() == sum([tensor([h(lam), h(m(lam))])␣
↪for lam in Partitions(n)])
....:       for n in range(6) )
True
sage: all( factorial(n) * h([n]).internal_coproduct()
....:       == sum([lam.conjugacy_class_size() * tensor([h(p(lam)), h(p(lam))])
....:               for lam in Partitions(n)])
....:       for n in range(6) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> p = SymmetricFunctions(QQ).p()
>>> h = SymmetricFunctions(QQ).h()
>>> s = SymmetricFunctions(QQ).s()
>>> all( s(h([n])).internal_coproduct() == sum([tensor([s(lam), s(lam)]) for␣
↪lam in Partitions(n)])
...       for n in range(Integer(6)) )
True
```

(continues on next page)

(continued from previous page)

```
>>> all( h([n]).internal_coproduct() == sum([tensor([h(lam), h(m(lam))]) for␣
↪lam in Partitions(n)])
...       for n in range(Integer(6)) )
True
>>> all( factorial(n) * h([n]).internal_coproduct()
...       == sum([lam.conjugacy_class_size() * tensor([h(p(lam)), h(p(lam))])
...               for lam in Partitions(n)])
...       for n in range(Integer(6)) )
True
```

internal_product(*x*)

Return the internal (tensor) product of `self` and `x` in the basis of `self`.

The internal tensor product can be defined as the linear extension of the definition on power sums $p_\lambda * p_\mu = \delta_{\lambda,\mu} z_\lambda p_\lambda$, where $z_\lambda = (1^{r_1} r_1!)(2^{r_2} r_2!) \cdots$ for $\lambda = (1^{r_1} 2^{r_2} \cdots)$ and where $*$ denotes the internal tensor product. The internal tensor product is also known as the Kronecker product, or as the second multiplication on the ring of symmetric functions.

Note that the internal product of any two homogeneous symmetric functions of equal degrees is a homogeneous symmetric function of the same degree. On the other hand, the internal product of two homogeneous symmetric functions of distinct degrees is 0.

Note

The internal product is sometimes referred to as "inner product" in the literature, but unfortunately this name is shared by a different operation, namely the Hall inner product (see `scalar()`).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the internal product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

The methods `itensor()`, `internal_product()`, `kronecker_product()`, `inner_tensor()` are all synonyms.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
```

(continues on next page)

(continued from previous page)

```
>>> a = s([Integer(2),Integer(1)])
>>> b = s([Integer(3)])
>>> a.itensor(b)
s[2, 1]
>>> c = s([Integer(3),Integer(2),Integer(1)])
>>> c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

There are few quantitative results pertaining to Kronecker products in general, which makes their computation so difficult. Let us test a few of them in different bases.

The Kronecker product of any homogeneous symmetric function f of degree n with the n-th complete homogeneous symmetric function `h[n]` (a.k.a. `s[n]`) is f:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: all( h([5]).itensor(h(p)) == h(p) for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(ZZ).h()
>>> all( h([Integer(5)]).itensor(h(p)) == h(p) for p in␣
↪Partitions(Integer(5)) )
True
```

The Kronecker product of a Schur function s_λ with the n-th elementary symmetric function `e[n]`, where $n = |\lambda|$, is $s_{\lambda'}$ (where λ' is the conjugate partition of λ):

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: all( e([5]).itensor(s(p)) == s(p.conjugate()) for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> F = CyclotomicField(Integer(12))
>>> s = SymmetricFunctions(F).s()
>>> e = SymmetricFunctions(F).e()
>>> all( e([Integer(5)]).itensor(s(p)) == s(p.conjugate()) for p in␣
↪Partitions(Integer(5)) )
True
```

The Kronecker product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in␣
↪Partitions(4) )
....:      for p in Partitions(4) )
True

sage: F = FractionField(QQ['q','t'])
```

(continues on next page)

(continued from previous page)

```
sage: mq = SymmetricFunctions(F).macdonald().Q()
sage: mh = SymmetricFunctions(F).macdonald().H()
sage: all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
....:           for r in Partitions(4) )
....:       for p in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(19))).e()
>>> m = SymmetricFunctions(FiniteField(Integer(19))).m()
>>> all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in␣
↪Partitions(Integer(4)) )
...       for p in Partitions(Integer(4)) )
True

>>> F = FractionField(QQ['q','t'])
>>> mq = SymmetricFunctions(F).macdonald().Q()
>>> mh = SymmetricFunctions(F).macdonald().H()
>>> all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
...           for r in Partitions(Integer(4)) )
...       for p in Partitions(Integer(3)) )
True
```

Let us check (on examples) Proposition 5.2 of Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, “Noncommutative symmetric functions”, arXiv hep-th/9407124, for $r = 2$:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w))␣
↪with
....:                            # * meaning Kronecker product and \mu␣
↪meaning the
....:                            # usual multiplication.
....:     result = w.parent().zero()
....:     for partition_pair, coeff in w.coproduct():
....:         result += coeff * w.parent()(u).itensor(partition_pair[0]) * w.
↪parent()(v).itensor(partition_pair[1])
....:     return result
sage: all( all( all( tensor_copr(e[u], s[v], m[w])    # long time
....:                == (e[u] * s[v]).itensor(m[w])
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:       for u in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(29))).e()
>>> s = SymmetricFunctions(FiniteField(Integer(29))).s()
>>> m = SymmetricFunctions(FiniteField(Integer(29))).m()
>>> def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w)) with
...                            # * meaning Kronecker product and \mu meaning␣
```

(continues on next page)

(continued from previous page)

```
↪the
...                                 # usual multiplication.
...     result = w.parent().zero()
...     for partition_pair, coeff in w.coproduct():
...         result += coeff * w.parent()(u).itensor(partition_
↪pair[Integer(0)]) * w.parent()(v).itensor(partition_pair[Integer(1)])
...     return result
>>> all( all( all( tensor_copr(e[u], s[v], m[w])    # long time
...                  == (e[u] * s[v]).itensor(m[w])
...                  for w in Partitions(Integer(5)) )
...             for v in Partitions(Integer(2)) )
...        for u in Partitions(Integer(3)) )
True
```

Some examples from Briand, Orellana, Rosas, "The stability of the Kronecker products of Schur functions." arXiv 0907.4652:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s[2,2].itensor(s[2,2])
s[1, 1, 1, 1] + s[2, 2] + s[4]
sage: s[3,2].itensor(s[3,2])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s[4,2].itensor(s[4,2])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1]␣
↪+ s[6]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> s[Integer(2),Integer(2)].itensor(s[Integer(2),Integer(2)])
s[1, 1, 1, 1] + s[2, 2] + s[4]
>>> s[Integer(3),Integer(2)].itensor(s[Integer(3),Integer(2)])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
>>> s[Integer(4),Integer(2)].itensor(s[Integer(4),Integer(2)])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1]␣
↪+ s[6]
```

An example from p. 220 of Thibon, "Hopf algebras of symmetric functions and tensor products of symmetric group representations", International Journal of Algebra and Computation, 1991:

```
sage: s = SymmetricFunctions(QQbar).s()
sage: s[2,1].itensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQbar).s()
>>> s[Integer(2),Integer(1)].itensor(s[Integer(2),Integer(1)])
s[1, 1, 1] + s[2, 1] + s[3]
```

Note

The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a **Q**-algebra (in which case the Kronecker product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over **Q** (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

is_schur_positive()

Return `True` if and only if `self` is Schur positive.

If s is the space of Schur functions over `self`'s base ring, then this is the same as `self._is_positive(s)`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1]) + s([3])
sage: a.is_schur_positive()
True
sage: a = s([2,1]) - s([3])
sage: a.is_schur_positive()
False
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> a = s([Integer(2),Integer(1)]) + s([Integer(3)])
>>> a.is_schur_positive()
True
>>> a = s([Integer(2),Integer(1)]) - s([Integer(3)])
>>> a.is_schur_positive()
False
```

```
sage: QQx = QQ['x']
sage: s = SymmetricFunctions(QQx).s()
sage: x = QQx.gen()
sage: a = (1+x)*s([2,1])
sage: a.is_schur_positive()
True
sage: a = (1-x)*s([2,1])
sage: a.is_schur_positive()
False
sage: s(0).is_schur_positive()
True
sage: s(1+x).is_schur_positive()
True
```

```
>>> from sage.all import *
>>> QQx = QQ['x']
>>> s = SymmetricFunctions(QQx).s()
>>> x = QQx.gen()
>>> a = (Integer(1)+x)*s([Integer(2),Integer(1)])
>>> a.is_schur_positive()
True
>>> a = (Integer(1)-x)*s([Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> a.is_schur_positive()
False
>>> s(Integer(0)).is_schur_positive()
True
>>> s(Integer(1)+x).is_schur_positive()
True
```

itensor(*x*)

Return the internal (tensor) product of `self` and `x` in the basis of `self`.

The internal tensor product can be defined as the linear extension of the definition on power sums $p_\lambda * p_\mu = \delta_{\lambda,\mu} z_\lambda p_\lambda$, where $z_\lambda = (1^{r_1} r_1!)(2^{r_2} r_2!) \cdots$ for $\lambda = (1^{r_1} 2^{r_2} \cdots)$ and where $*$ denotes the internal tensor product. The internal tensor product is also known as the Kronecker product, or as the second multiplication on the ring of symmetric functions.

Note that the internal product of any two homogeneous symmetric functions of equal degrees is a homogeneous symmetric function of the same degree. On the other hand, the internal product of two homogeneous symmetric functions of distinct degrees is 0.

Note

The internal product is sometimes referred to as "inner product" in the literature, but unfortunately this name is shared by a different operation, namely the Hall inner product (see `scalar()`).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the internal product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

The methods `itensor()`, `internal_product()`, `kronecker_product()`, `inner_tensor()` are all synonyms.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> a = s([Integer(2),Integer(1)])
>>> b = s([Integer(3)])
>>> a.itensor(b)
```

(continues on next page)

(continued from previous page)

```
s[2, 1]
>>> c = s([Integer(3),Integer(2),Integer(1)])
>>> c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

There are few quantitative results pertaining to Kronecker products in general, which makes their computation so difficult. Let us test a few of them in different bases.

The Kronecker product of any homogeneous symmetric function f of degree n with the n-th complete homogeneous symmetric function `h[n]` (a.k.a. `s[n]`) is f:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: all( h([5]).itensor(h(p)) == h(p) for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(ZZ).h()
>>> all( h([Integer(5)]).itensor(h(p)) == h(p) for p in␣
↪Partitions(Integer(5)) )
True
```

The Kronecker product of a Schur function s_λ with the n-th elementary symmetric function `e[n]`, where $n = |\lambda|$, is $s_{\lambda'}$ (where λ' is the conjugate partition of λ):

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: all( e([5]).itensor(s(p)) == s(p.conjugate()) for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> F = CyclotomicField(Integer(12))
>>> s = SymmetricFunctions(F).s()
>>> e = SymmetricFunctions(F).e()
>>> all( e([Integer(5)]).itensor(s(p)) == s(p.conjugate()) for p in␣
↪Partitions(Integer(5)) )
True
```

The Kronecker product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in␣
↪Partitions(4) )
....:       for p in Partitions(4) )
True

sage: F = FractionField(QQ['q','t'])
sage: mq = SymmetricFunctions(F).macdonald().Q()
sage: mh = SymmetricFunctions(F).macdonald().H()
sage: all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
```

(continues on next page)

(continued from previous page)

```
....:            for r in Partitions(4) )
....:       for p in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(19))).e()
>>> m = SymmetricFunctions(FiniteField(Integer(19))).m()
>>> all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in␣
↪Partitions(Integer(4)) )
...       for p in Partitions(Integer(4)) )
True

>>> F = FractionField(QQ['q','t'])
>>> mq = SymmetricFunctions(F).macdonald().Q()
>>> mh = SymmetricFunctions(F).macdonald().H()
>>> all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
...            for r in Partitions(Integer(4)) )
...       for p in Partitions(Integer(3)) )
True
```

Let us check (on examples) Proposition 5.2 of Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, "Noncommutative symmetric functions", arXiv hep-th/9407124, for $r = 2$:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w))␣
↪with
....:                            # * meaning Kronecker product and \mu␣
↪meaning the
....:                            # usual multiplication.
....:     result = w.parent().zero()
....:     for partition_pair, coeff in w.coproduct():
....:         result += coeff * w.parent()(u).itensor(partition_pair[0]) * w.
↪parent()(v).itensor(partition_pair[1])
....:     return result
sage: all( all( all( tensor_copr(e[u], s[v], m[w])    # long time
....:                == (e[u] * s[v]).itensor(m[w])
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:      for u in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(29))).e()
>>> s = SymmetricFunctions(FiniteField(Integer(29))).s()
>>> m = SymmetricFunctions(FiniteField(Integer(29))).m()
>>> def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w)) with
...                            # * meaning Kronecker product and \mu meaning␣
↪the
...                            # usual multiplication.
...     result = w.parent().zero()
```

(continues on next page)

(continued from previous page)

```
...     for partition_pair, coeff in w.coproduct():
...         result += coeff * w.parent()(u).itensor(partition_
↪pair[Integer(0)]) * w.parent()(v).itensor(partition_pair[Integer(1)])
...     return result
>>> all( all( all( tensor_copr(e[u], s[v], m[w])   # long time
...                == (e[u] * s[v]).itensor(m[w])
...                for w in Partitions(Integer(5)) )
...           for v in Partitions(Integer(2)) )
...      for u in Partitions(Integer(3)) )
True
```

Some examples from Briand, Orellana, Rosas, "The stability of the Kronecker products of Schur functions." arXiv 0907.4652:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s[2,2].itensor(s[2,2])
s[1, 1, 1, 1] + s[2, 2] + s[4]
sage: s[3,2].itensor(s[3,2])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s[4,2].itensor(s[4,2])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1]␣
↪+ s[6]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> s[Integer(2),Integer(2)].itensor(s[Integer(2),Integer(2)])
s[1, 1, 1, 1] + s[2, 2] + s[4]
>>> s[Integer(3),Integer(2)].itensor(s[Integer(3),Integer(2)])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
>>> s[Integer(4),Integer(2)].itensor(s[Integer(4),Integer(2)])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1]␣
↪+ s[6]
```

An example from p. 220 of Thibon, "Hopf algebras of symmetric functions and tensor products of symmetric group representations", International Journal of Algebra and Computation, 1991:

```
sage: s = SymmetricFunctions(QQbar).s()
sage: s[2,1].itensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQbar).s()
>>> s[Integer(2),Integer(1)].itensor(s[Integer(2),Integer(1)])
s[1, 1, 1] + s[2, 1] + s[3]
```

Note

The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a **Q**-algebra (in which case the Kronecker product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over **Q** (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

kronecker_coproduct()

Return the inner coproduct of `self` in the basis of `self`.

The inner coproduct (also known as the Kronecker coproduct, as the internal coproduct, or as the second comultiplication on the ring of symmetric functions) is a ring homomorphism $\Delta^\times$ from the ring of symmetric functions to the tensor product (over the base ring) of this ring with itself. It is uniquely characterized by the formula

$$\Delta^\times(h_n) = \sum_{\lambda \vdash n} s_\lambda \otimes s_\lambda = \sum_{\lambda \vdash n} h_\lambda \otimes m_\lambda = \sum_{\lambda \vdash n} m_\lambda \otimes h_\lambda,$$

where $\lambda \vdash n$ means λ is a partition of n, and n is any nonnegative integer. It also satisfies

$$\Delta^\times(p_n) = p_n \otimes p_n$$

for any positive integer n. If the base ring is a **Q**-algebra, it also satisfies

$$\Delta^\times(h_n) = \sum_{\lambda \vdash n} z_\lambda^{-1} p_\lambda \otimes p_\lambda,$$

where

$$z_\lambda = \prod_{i=1}^{\infty} i^{m_i(\lambda)} m_i(\lambda)!$$

with $m_i(\lambda)$ meaning the number of appearances of i in λ (see *zee()*).

The method *kronecker_coproduct()* is a synonym of *internal_coproduct()*.

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([2,1])
sage: a.internal_coproduct()
s[1, 1, 1] # s[2, 1] + s[2, 1] # s[1, 1, 1] + s[2, 1] # s[2, 1] + s[2, 1] #␣
→s[3] + s[3] # s[2, 1]

sage: e = SymmetricFunctions(QQ).e()
sage: b = e([2])
sage: b.internal_coproduct()
e[1, 1] # e[2] + e[2] # e[1, 1] - 2*e[2] # e[2]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> a = s([Integer(2),Integer(1)])
>>> a.internal_coproduct()
s[1, 1, 1] # s[2, 1] + s[2, 1] # s[1, 1, 1] + s[2, 1] # s[2, 1] + s[2, 1] #␣
→s[3] + s[3] # s[2, 1]

>>> e = SymmetricFunctions(QQ).e()
>>> b = e([Integer(2)])
>>> b.internal_coproduct()
e[1, 1] # e[2] + e[2] # e[1, 1] - 2*e[2] # e[2]
```

The internal coproduct is adjoint to the internal product with respect to the Hall inner product: Any three symmetric functions f, g and h satisfy $\langle f * g, h\rangle = \sum_i \langle f, h'_i\rangle \langle g, h''_i\rangle$, where we write $\Delta^{\times}(h)$ as $\sum_i h'_i \otimes h''_i$. Let us check this in degree 4:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \
↪left< g, h''_i \right>
....:     result = h.base_ring().zero()
....:     for partition_pair, coeff in h.internal_coproduct():
....:         result += coeff * h.parent()(f).scalar(partition_pair[0]) * h.
↪parent()(g).scalar(partition_pair[1])
....:     return result
sage: all( all( all( tensor_incopr(e[u], s[v], m[w]) == (e[u].itensor(s[v])).
↪scalar(m[w])  # long time (10s on sage.math, 2013)
....:                  for w in Partitions(5) )
....:            for v in Partitions(2) )
....:      for u in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(29))).e()
>>> s = SymmetricFunctions(FiniteField(Integer(29))).s()
>>> m = SymmetricFunctions(FiniteField(Integer(29))).m()
>>> def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \
↪left< g, h''_i \right>
...     result = h.base_ring().zero()
...     for partition_pair, coeff in h.internal_coproduct():
...         result += coeff * h.parent()(f).scalar(partition_
↪pair[Integer(0)]) * h.parent()(g).scalar(partition_pair[Integer(1)])
...     return result
>>> all( all( all( tensor_incopr(e[u], s[v], m[w]) == (e[u].itensor(s[v])).
↪scalar(m[w])  # long time (10s on sage.math, 2013)
...                  for w in Partitions(Integer(5)) )
...            for v in Partitions(Integer(2)) )
...      for u in Partitions(Integer(3)) )
True
```

Let us check the formulas for $\Delta^{\times}(h_n)$ and $\Delta^{\times}(p_n)$ given in the description of this method:

```
sage: e = SymmetricFunctions(QQ).e()
sage: p = SymmetricFunctions(QQ).p()
sage: h = SymmetricFunctions(QQ).h()
sage: s = SymmetricFunctions(QQ).s()
sage: all( s(h([n])).internal_coproduct() == sum([tensor([s(lam), s(lam)])␣
↪for lam in Partitions(n)])
....:      for n in range(6) )
True
sage: all( h([n]).internal_coproduct() == sum([tensor([h(lam), h(m(lam))])␣
↪for lam in Partitions(n)])
....:      for n in range(6) )
True
```

(continues on next page)

(continued from previous page)

```
sage: all( factorial(n) * h([n]).internal_coproduct()
....:      == sum([lam.conjugacy_class_size() * tensor([h(p(lam)), h(p(lam))])
....:              for lam in Partitions(n)])
....:      for n in range(6) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> p = SymmetricFunctions(QQ).p()
>>> h = SymmetricFunctions(QQ).h()
>>> s = SymmetricFunctions(QQ).s()
>>> all( s(h([n])).internal_coproduct() == sum([tensor([s(lam), s(lam)]) for␣
↪lam in Partitions(n)])
...      for n in range(Integer(6)) )
True
>>> all( h([n]).internal_coproduct() == sum([tensor([h(lam), h(m(lam))]) for␣
↪lam in Partitions(n)])
...      for n in range(Integer(6)) )
True
>>> all( factorial(n) * h([n]).internal_coproduct()
...      == sum([lam.conjugacy_class_size() * tensor([h(p(lam)), h(p(lam))])
...              for lam in Partitions(n)])
...      for n in range(Integer(6)) )
True
```

kronecker_product(*x*)

Return the internal (tensor) product of `self` and `x` in the basis of `self`.

The internal tensor product can be defined as the linear extension of the definition on power sums $p_\lambda * p_\mu = \delta_{\lambda,\mu} z_\lambda p_\lambda$, where $z_\lambda = (1^{r_1} r_1!)(2^{r_2} r_2!) \cdots$ for $\lambda = (1^{r_1} 2^{r_2} \cdots)$ and where $*$ denotes the internal tensor product. The internal tensor product is also known as the Kronecker product, or as the second multiplication on the ring of symmetric functions.

Note that the internal product of any two homogeneous symmetric functions of equal degrees is a homogeneous symmetric function of the same degree. On the other hand, the internal product of two homogeneous symmetric functions of distinct degrees is 0.

Note

The internal product is sometimes referred to as "inner product" in the literature, but unfortunately this name is shared by a different operation, namely the Hall inner product (see `scalar()`).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the internal product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

The methods `itensor()`, `internal_product()`, `kronecker_product()`, `inner_tensor()` are all synonyms.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> a = s([Integer(2),Integer(1)])
>>> b = s([Integer(3)])
>>> a.itensor(b)
s[2, 1]
>>> c = s([Integer(3),Integer(2),Integer(1)])
>>> c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

There are few quantitative results pertaining to Kronecker products in general, which makes their computation so difficult. Let us test a few of them in different bases.

The Kronecker product of any homogeneous symmetric function f of degree n with the n-th complete homogeneous symmetric function `h[n]` (a.k.a. `s[n]`) is f:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: all( h([5]).itensor(h(p)) == h(p) for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(ZZ).h()
>>> all( h([Integer(5)]).itensor(h(p)) == h(p) for p in␣
↪Partitions(Integer(5)) )
True
```

The Kronecker product of a Schur function s_λ with the n-th elementary symmetric function `e[n]`, where $n = |\lambda|$, is $s_{\lambda'}$ (where λ' is the conjugate partition of λ):

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: all( e([5]).itensor(s(p)) == s(p.conjugate()) for p in Partitions(5) )
True
```

```
>>> from sage.all import *
>>> F = CyclotomicField(Integer(12))
>>> s = SymmetricFunctions(F).s()
>>> e = SymmetricFunctions(F).e()
>>> all( e([Integer(5)]).itensor(s(p)) == s(p.conjugate()) for p in␣
```

(continues on next page)

(continued from previous page)

```
↪Partitions(Integer(5)) )
True
```

The Kronecker product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in␣
↪Partitions(4) )
....:      for p in Partitions(4) )
True

sage: F = FractionField(QQ['q','t'])
sage: mq = SymmetricFunctions(F).macdonald().Q()
sage: mh = SymmetricFunctions(F).macdonald().H()
sage: all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
....:           for r in Partitions(4) )
....:      for p in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(19))).e()
>>> m = SymmetricFunctions(FiniteField(Integer(19))).m()
>>> all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in␣
↪Partitions(Integer(4)) )
...      for p in Partitions(Integer(4)) )
True

>>> F = FractionField(QQ['q','t'])
>>> mq = SymmetricFunctions(F).macdonald().Q()
>>> mh = SymmetricFunctions(F).macdonald().H()
>>> all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
...           for r in Partitions(Integer(4)) )
...      for p in Partitions(Integer(3)) )
True
```

Let us check (on examples) Proposition 5.2 of Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, “Noncommutative symmetric functions”, arXiv hep-th/9407124, for $r = 2$:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w))␣
↪with
....:                            # * meaning Kronecker product and \mu␣
↪meaning the
....:                            # usual multiplication.
....:     result = w.parent().zero()
....:     for partition_pair, coeff in w.coproduct():
....:         result += coeff * w.parent()(u).itensor(partition_pair[0]) * w.
↪parent()(v).itensor(partition_pair[1])
....:     return result
```

(continues on next page)

(continued from previous page)

```
sage: all( all( all( tensor_copr(e[u], s[v], m[w])   # long time
....:                == (e[u] * s[v]).itensor(m[w])
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:      for u in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(FiniteField(Integer(29))).e()
>>> s = SymmetricFunctions(FiniteField(Integer(29))).s()
>>> m = SymmetricFunctions(FiniteField(Integer(29))).m()
>>> def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w)) with
...                            # * meaning Kronecker product and \mu meaning␣
↪the
...                            # usual multiplication.
...     result = w.parent().zero()
...     for partition_pair, coeff in w.coproduct():
...         result += coeff * w.parent()(u).itensor(partition_
↪pair[Integer(0)]) * w.parent()(v).itensor(partition_pair[Integer(1)])
...     return result
>>> all( all( all( tensor_copr(e[u], s[v], m[w])   # long time
...                == (e[u] * s[v]).itensor(m[w])
...                for w in Partitions(Integer(5)) )
...           for v in Partitions(Integer(2)) )
...      for u in Partitions(Integer(3)) )
True
```

Some examples from Briand, Orellana, Rosas, "The stability of the Kronecker products of Schur functions." arXiv 0907.4652:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s[2,2].itensor(s[2,2])
s[1, 1, 1, 1] + s[2, 2] + s[4]
sage: s[3,2].itensor(s[3,2])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s[4,2].itensor(s[4,2])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1]␣
↪+ s[6]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> s[Integer(2),Integer(2)].itensor(s[Integer(2),Integer(2)])
s[1, 1, 1, 1] + s[2, 2] + s[4]
>>> s[Integer(3),Integer(2)].itensor(s[Integer(3),Integer(2)])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
>>> s[Integer(4),Integer(2)].itensor(s[Integer(4),Integer(2)])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1]␣
↪+ s[6]
```

An example from p. 220 of Thibon, "Hopf algebras of symmetric functions and tensor products of symmetric group representations", International Journal of Algebra and Computation, 1991:

```
sage: s = SymmetricFunctions(QQbar).s()
sage: s[2,1].itensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQbar).s()
>>> s[Integer(2),Integer(1)].itensor(s[Integer(2),Integer(1)])
s[1, 1, 1] + s[2, 1] + s[3]
```

Note

The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a **Q**-algebra (in which case the Kronecker product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over **Q** (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

left_padded_kronecker_product(*x*)

Return the left-padded Kronecker product of `self` and `x` in the basis of `self`.

The left-padded Kronecker product is a bilinear map mapping two symmetric functions to another, not necessarily preserving degree. It can be defined as follows: Let $*$ denote the Kronecker product (*itensor()*) on the space of symmetric functions. For any partitions α, β, γ, let $g^{\gamma}_{\alpha,\beta}$ denote the coefficient of the complete homogeneous symmetric function h_γ in the Kronecker product $h_\alpha * h_\beta$. For every partition $\lambda = (\lambda_1, \lambda_2, \lambda_3, \ldots)$ and every integer $n > |\lambda| + \lambda_1$, let $\lambda[n]$ denote the n-completion of λ (this is the partition $(n - |\lambda|, \lambda_1, \lambda_2, \lambda_3, \ldots)$; see *t_completion()*). Then, for any partitions α and β and every integer $n \geq |\alpha| + |\beta| + \alpha_1 + \beta_1$, we can write the Kronecker product $h_{\alpha[n]} * h_{\beta[n]}$ in the form

$$h_{\alpha[n]} * h_{\beta[n]} = \sum_{\gamma} g^{\gamma[n]}_{\alpha[n],\beta[n]} h_{\gamma[n]}$$

with γ ranging over all partitions. The coefficients $g^{\gamma[n]}_{\alpha[n],\beta[n]}$ are independent on n. These coefficients $g^{\gamma[n]}_{\alpha[n],\beta[n]}$ are denoted by $\overline{g}^{\gamma}_{\alpha,\beta}$, and the symmetric function

$$\sum_{\gamma} \overline{g}^{\gamma}_{\alpha,\beta} h_\gamma$$

is said to be the *left-padded Kronecker product* of h_α and h_β. By bilinearity, this extends to a definition of a left-padded Kronecker product of any two symmetric functions.

This notion of left-padded Kronecker product can be lifted to the non-commutative symmetric functions (*left_padded_kronecker_product()*).

Warning

Do not mistake this product for the reduced Kronecker product (*reduced_kronecker_product()*), which uses the Schur functions instead of the complete homogeneous functions in its definition.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the left-padded Kronecker product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: h = Sym.h()
sage: h[2,1].left_padded_kronecker_product(h[3])
h[1, 1, 1, 1] + h[2, 1] + h[2, 1, 1] + h[2, 1, 1, 1] + h[2, 2, 1] + h[3, 2, 1]
sage: h[2,1].left_padded_kronecker_product(h[1])
h[1, 1, 1] + h[2, 1] + h[2, 1, 1]
sage: h[1].left_padded_kronecker_product(h[2,1])
h[1, 1, 1] + h[2, 1] + h[2, 1, 1]
sage: h[1,1].left_padded_kronecker_product(h[2])
h[1, 1] + 2*h[1, 1, 1] + h[2, 1, 1]
sage: h[1].left_padded_kronecker_product(h[2,1,1])
h[1, 1, 1, 1] + 2*h[2, 1, 1] + h[2, 1, 1, 1]
sage: h[2].left_padded_kronecker_product(h[3])
h[2, 1] + h[2, 1, 1] + h[3, 2]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> h = Sym.h()
>>> h[Integer(2),Integer(1)].left_padded_kronecker_product(h[Integer(3)])
h[1, 1, 1, 1] + h[2, 1] + h[2, 1, 1] + h[2, 1, 1, 1] + h[2, 2, 1] + h[3, 2, 1]
>>> h[Integer(2),Integer(1)].left_padded_kronecker_product(h[Integer(1)])
h[1, 1, 1] + h[2, 1] + h[2, 1, 1]
>>> h[Integer(1)].left_padded_kronecker_product(h[Integer(2),Integer(1)])
h[1, 1, 1] + h[2, 1] + h[2, 1, 1]
>>> h[Integer(1),Integer(1)].left_padded_kronecker_product(h[Integer(2)])
h[1, 1] + 2*h[1, 1, 1] + h[2, 1, 1]
>>> h[Integer(1)].left_padded_kronecker_product(h[Integer(2),Integer(1),
↪Integer(1)])
h[1, 1, 1, 1] + 2*h[2, 1, 1] + h[2, 1, 1, 1]
>>> h[Integer(2)].left_padded_kronecker_product(h[Integer(3)])
h[2, 1] + h[2, 1, 1] + h[3, 2]
```

Taking the left-padded Kronecker product with $1 = h_\emptyset$ is the identity map on the ring of symmetric functions:

```
sage: all( h[Partition([])].left_padded_kronecker_product(h[lam])
....:      == h[lam] for i in range(4)
....:      for lam in Partitions(i) )
True
```

```
>>> from sage.all import *
>>> all( h[Partition([])].left_padded_kronecker_product(h[lam])
...      == h[lam] for i in range(Integer(4))
...      for lam in Partitions(i) )
True
```

Here is a rule for the left-padded Kronecker product of h_1 (this is the same as $h_{(1)}$) with any complete homogeneous function: Let λ be a partition. Then, the left-padded Kronecker product of h_1 and h_λ is

$\sum_\mu a_\mu h_\mu$, where the sum runs over all partitions μ, and the coefficient a_μ is defined as the number of ways to obtain μ from λ by one of the following two operations:

- Insert a 1 into λ.
- Subtract 1 from one of the entries of λ (and remove the entry if it thus becomes 0), and insert a 1 into λ.

We check this for partitions of size ≤ 4:

```
sage: def mults1(I):
....:     # Left-padded Kronecker multiplication by h[1].
....:     res = h[I[:] + [1]]
....:     for k in range(len(I)):
....:         I2 = I[:]
....:         if I2[k] == 1:
....:             I2 = I2[:k] + I2[k+1:]
....:         else:
....:             I2[k] -= 1
....:         res += h[sorted(I2 + [1], reverse=True)]
....:     return res
sage: all( mults1(I) == h[1].left_padded_kronecker_product(h[I])
....:                == h[I].left_padded_kronecker_product(h[1])
....:       for i in range(5) for I in Partitions(i) )
True
```

```
>>> from sage.all import *
>>> def mults1(I):
...     # Left-padded Kronecker multiplication by h[1].
...     res = h[I[:] + [Integer(1)]]
...     for k in range(len(I)):
...         I2 = I[:]
...         if I2[k] == Integer(1):
...             I2 = I2[:k] + I2[k+Integer(1):]
...         else:
...             I2[k] -= Integer(1)
...         res += h[sorted(I2 + [Integer(1)], reverse=True)]
...     return res
>>> all( mults1(I) == h[Integer(1)].left_padded_kronecker_product(h[I])
...                == h[I].left_padded_kronecker_product(h[Integer(1)])
...       for i in range(Integer(5)) for I in Partitions(i) )
True
```

The left-padded Kronecker product is commutative:

```
sage: all( h[lam].left_padded_kronecker_product(h[mu])
....:      == h[mu].left_padded_kronecker_product(h[lam])
....:      for lam in Partitions(3) for mu in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> all( h[lam].left_padded_kronecker_product(h[mu])
...      == h[mu].left_padded_kronecker_product(h[lam])
...      for lam in Partitions(Integer(3)) for mu in Partitions(Integer(3)) )
True
```

nabla(*q=None*, *t=None*, *power=1*)

Return the value of the nabla operator applied to `self`.

The eigenvectors of the nabla operator are the Macdonald polynomials in the Ht basis.

If the parameter `power` is an integer then it calculates nabla to that integer. The default value of `power` is 1.

INPUT:

- `q`, `t` – parameters (default: `None`, in which case `q` and `t` are used)
- `power` – (default: `1`) an integer indicating how many times to apply the operator ∇. Negative values of `power` indicate powers of ∇^{-1}.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: p = Sym.power()
sage: p([1,1]).nabla()
-(1/2*q*t-1/2*q-1/2*t-1/2)*p[1, 1] + (1/2*q*t-1/2*q-1/2*t+1/2)*p[2]
sage: p([2,1]).nabla(q=1)
-(t+1)*p[1, 1, 1] + t*p[2, 1]
sage: p([2]).nabla(q=1)*p([1]).nabla(q=1)
-(t+1)*p[1, 1, 1] + t*p[2, 1]
sage: s = Sym.schur()
sage: s([2,1]).nabla()
-(q^3*t+q^2*t^2+q*t^3)*s[1, 1, 1] - (q^2*t+q*t^2)*s[2, 1]
sage: s([1,1,1]).nabla()
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
sage: s([1,1,1]).nabla(t=1)
(q^3+q^2+2*q+1)*s[1, 1, 1] + (q^2+2*q+2)*s[2, 1] + s[3]
sage: s(0).nabla()
0
sage: s(1).nabla()
s[]
sage: s([2,1]).nabla(power=-1)
-((q+t)/(q^2*t^2))*s[2, 1] + ((q^2+q*t+t^2)/(-q^3*t^3))*s[3]
sage: (s([2])+s([3])).nabla()
-q*t*s[1, 1] + (q^3*t^2+q^2*t^3)*s[1, 1, 1] + q^2*t^2*s[2, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> p = Sym.power()
>>> p([Integer(1),Integer(1)]).nabla()
-(1/2*q*t-1/2*q-1/2*t-1/2)*p[1, 1] + (1/2*q*t-1/2*q-1/2*t+1/2)*p[2]
>>> p([Integer(2),Integer(1)]).nabla(q=Integer(1))
-(t+1)*p[1, 1, 1] + t*p[2, 1]
>>> p([Integer(2)]).nabla(q=Integer(1))*p([Integer(1)]).nabla(q=Integer(1))
-(t+1)*p[1, 1, 1] + t*p[2, 1]
>>> s = Sym.schur()
>>> s([Integer(2),Integer(1)]).nabla()
-(q^3*t+q^2*t^2+q*t^3)*s[1, 1, 1] - (q^2*t+q*t^2)*s[2, 1]
>>> s([Integer(1),Integer(1),Integer(1)]).nabla()
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
>>> s([Integer(1),Integer(1),Integer(1)]).nabla(t=Integer(1))
(q^3+q^2+2*q+1)*s[1, 1, 1] + (q^2+2*q+2)*s[2, 1] + s[3]
```

(continues on next page)

(continued from previous page)

```
>>> s(Integer(0)).nabla()
0
>>> s(Integer(1)).nabla()
s[]
>>> s([Integer(2),Integer(1)]).nabla(power=-Integer(1))
-((q+t)/(q^2*t^2))*s[2, 1] + ((q^2+q*t+t^2)/(-q^3*t^3))*s[3]
>>> (s([Integer(2)])+s([Integer(3)])).nabla()
-q*t*s[1, 1] + (q^3*t^2+q^2*t^3)*s[1, 1, 1] + q^2*t^2*s[2, 1]
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

The default implementation converts to the Schur basis, then performs the automorphism and changes back.

`omega_involution()` is a synonym for the `omega()` method.

EXAMPLES:

```
sage: J = SymmetricFunctions(QQ).jack(t=1).P()
sage: a = J([2,1]) + J([1,1,1])
sage: a.omega()
JackP[2, 1] + JackP[3]
sage: J(0).omega()
0
sage: J(1).omega()
JackP[]
```

```
>>> from sage.all import *
>>> J = SymmetricFunctions(QQ).jack(t=Integer(1)).P()
>>> a = J([Integer(2),Integer(1)]) + J([Integer(1),Integer(1),Integer(1)])
>>> a.omega()
JackP[2, 1] + JackP[3]
>>> J(Integer(0)).omega()
0
>>> J(Integer(1)).omega()
JackP[]
```

The forgotten symmetric functions are the images of the monomial symmetric functions under omega:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.m()
```

(continues on next page)

(continued from previous page)

```
sage: f = Sym.f()
sage: all( f(lam) == m(lam).omega() for lam in Partitions(3) )
True
sage: all( m(lam) == f(lam).omega() for lam in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> m = Sym.m()
>>> f = Sym.f()
>>> all( f(lam) == m(lam).omega() for lam in Partitions(Integer(3)) )
True
>>> all( m(lam) == f(lam).omega() for lam in Partitions(Integer(3)) )
True
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

The default implementation converts to the Schur basis, then performs the automorphism and changes back.

`omega_involution()` is a synonym for the `omega()` method.

EXAMPLES:

```
sage: J = SymmetricFunctions(QQ).jack(t=1).P()
sage: a = J([2,1]) + J([1,1,1])
sage: a.omega()
JackP[2, 1] + JackP[3]
sage: J(0).omega()
0
sage: J(1).omega()
JackP[]
```

```
>>> from sage.all import *
>>> J = SymmetricFunctions(QQ).jack(t=Integer(1)).P()
>>> a = J([Integer(2),Integer(1)]) + J([Integer(1),Integer(1),Integer(1)])
>>> a.omega()
JackP[2, 1] + JackP[3]
>>> J(Integer(0)).omega()
0
```

(continues on next page)

(continued from previous page)

```
>>> J(Integer(1)).omega()
JackP[]
```

The forgotten symmetric functions are the images of the monomial symmetric functions under omega:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.m()
sage: f = Sym.f()
sage: all( f(lam) == m(lam).omega() for lam in Partitions(3) )
True
sage: all( m(lam) == f(lam).omega() for lam in Partitions(3) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> m = Sym.m()
>>> f = Sym.f()
>>> all( f(lam) == m(lam).omega() for lam in Partitions(Integer(3)) )
True
>>> all( m(lam) == f(lam).omega() for lam in Partitions(Integer(3)) )
True
```

omega_qt(*q=None*, *t=None*)

Return the image of `self` under the q,t-deformed omega automorphism which sends p_k to $(-1)^{k-1} \cdot \frac{1-q^k}{1-t^k} \cdot p_k$ for all positive integers k.

In general, this is well-defined outside of the powersum basis only if the base ring is a **Q**-algebra.

If $q = t$, then this is the omega automorphism (`omega()`).

INPUT:

- `q`, `t` – parameters (default: `None`, in which case `'q'` and `'t'` are used)

EXAMPLES:

```
sage: QQqt = QQ['q,t'].fraction_field()
sage: q,t = QQqt.gens()
sage: p = SymmetricFunctions(QQqt).p()
sage: p[5].omega_qt()
-((q^5-1)/(-t^5+1))*p[5]
sage: p[5].omega_qt(q,t)
-((q^5-1)/(-t^5+1))*p[5]
sage: p([2]).omega_qt(q,t)
((q^2-1)/(-t^2+1))*p[2]
sage: p([2,1]).omega_qt(q,t)
-((q^3-q^2-q+1)/(t^3-t^2-t+1))*p[2, 1]
sage: p([3,2]).omega_qt(5,q)
-(2976/(q^5-q^3-q^2+1))*p[3, 2]
sage: p(0).omega_qt()
0
sage: p(1).omega_qt()
p[]
sage: H = SymmetricFunctions(QQqt).macdonald().H()
```

(continues on next page)

(continued from previous page)

```
sage: H([1,1]).omega_qt()
((2*q^2-2*q*t-2*q+2*t)/(t^3-t^2-t+1))*McdH[1, 1] + ((q-1)/(t-1))*McdH[2]
sage: H([1,1]).omega_qt(q,t)
((2*q^2-2*q*t-2*q+2*t)/(t^3-t^2-t+1))*McdH[1, 1] + ((q-1)/(t-1))*McdH[2]
sage: H([1,1]).omega_qt(t,q)
-((t^3-t^2-t+1)/(-q^3+q^2+q-1))*McdH[2]
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: S = Sym.macdonald().S()
sage: S([1,1]).omega_qt()
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdS[1, 1] - ((q^2*t-q*t-q+1)/(-t^3+t^2+t-
↪1))*McdS[2]
sage: s = Sym.schur()
sage: s(S([1,1]).omega_qt())
s[2]
```

```
>>> from sage.all import *
>>> QQqt = QQ['q,t'].fraction_field()
>>> q,t = QQqt.gens()
>>> p = SymmetricFunctions(QQqt).p()
>>> p[Integer(5)].omega_qt()
-((q^5-1)/(-t^5+1))*p[5]
>>> p[Integer(5)].omega_qt(q,t)
-((q^5-1)/(-t^5+1))*p[5]
>>> p([Integer(2)]).omega_qt(q,t)
((q^2-1)/(-t^2+1))*p[2]
>>> p([Integer(2),Integer(1)]).omega_qt(q,t)
-((q^3-q^2-q+1)/(t^3-t^2-t+1))*p[2, 1]
>>> p([Integer(3),Integer(2)]).omega_qt(Integer(5),q)
-(2976/(q^5-q^3-q^2+1))*p[3, 2]
>>> p(Integer(0)).omega_qt()
0
>>> p(Integer(1)).omega_qt()
p[]
>>> H = SymmetricFunctions(QQqt).macdonald().H()
>>> H([Integer(1),Integer(1)]).omega_qt()
((2*q^2-2*q*t-2*q+2*t)/(t^3-t^2-t+1))*McdH[1, 1] + ((q-1)/(t-1))*McdH[2]
>>> H([Integer(1),Integer(1)]).omega_qt(q,t)
((2*q^2-2*q*t-2*q+2*t)/(t^3-t^2-t+1))*McdH[1, 1] + ((q-1)/(t-1))*McdH[2]
>>> H([Integer(1),Integer(1)]).omega_qt(t,q)
-((t^3-t^2-t+1)/(-q^3+q^2+q-1))*McdH[2]
>>> Sym = SymmetricFunctions(FractionField(QQ['q','t']))
>>> S = Sym.macdonald().S()
>>> S([Integer(1),Integer(1)]).omega_qt()
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdS[1, 1] - ((q^2*t-q*t-q+1)/(-t^3+t^2+t-
↪1))*McdS[2]
>>> s = Sym.schur()
>>> s(S([Integer(1),Integer(1)]).omega_qt())
s[2]
```

plethysm(*x*, *include=None*, *exclude=None*)

Return the outer plethysm of `self` with `x`.

This is implemented only over base rings which are **Q**-algebras. (To compute outer plethysms over general binomial rings, change bases to the fraction field.)

The outer plethysm of f with g is commonly denoted by $f[g]$ or by $f \circ g$. It is an algebra map in f, but not (generally) in g.

By default, the degree one elements are taken to be the generators for the `self`'s base ring. This setting can be modified by specifying the `include` and `exclude` keywords.

INPUT:

- `x` – a symmetric function over the same base ring as `self`
- `include` – list of variables to be treated as degree one elements instead of the default degree one elements
- `exclude` – list of variables to be excluded from the default degree one elements

OUTPUT:

An element in the parent of `x` or the base ring R of `self` when `x` is in R.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: h = Sym.h()
sage: h3h2 = h[3](h[2]); h3h2
h[2, 2, 2] - 2*h[3, 2, 1] + h[3, 3] + h[4, 1, 1] - h[5, 1] + h[6]
sage: s(h3h2)
s[2, 2, 2] + s[4, 2] + s[6]
sage: p = Sym.p()
sage: p3s21 = p[3](s[2,1]); p3s21
s[2, 2, 2, 1, 1, 1] - s[2, 2, 2, 2, 1] - s[3, 2, 1, 1, 1, 1]
 + s[3, 2, 2, 2] + s[3, 3, 1, 1, 1] - s[3, 3, 2, 1] + 2*s[3, 3, 3]
 + s[4, 1, 1, 1, 1, 1] - s[4, 3, 2] + s[4, 4, 1] - s[5, 1, 1, 1, 1]
 + s[5, 2, 2] - s[5, 4] + s[6, 1, 1, 1] - s[6, 2, 1] + s[6, 3]
sage: p(p3s21)
1/3*p[3, 3, 3] - 1/3*p[9]
sage: e = Sym.e()
sage: e[3](e[2])
e[3, 3] + e[4, 1, 1] - 2*e[4, 2] - e[5, 1] + e[6]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> h = Sym.h()
>>> h3h2 = h[Integer(3)](h[Integer(2)]); h3h2
h[2, 2, 2] - 2*h[3, 2, 1] + h[3, 3] + h[4, 1, 1] - h[5, 1] + h[6]
>>> s(h3h2)
s[2, 2, 2] + s[4, 2] + s[6]
>>> p = Sym.p()
>>> p3s21 = p[Integer(3)](s[Integer(2),Integer(1)]); p3s21
s[2, 2, 2, 1, 1, 1] - s[2, 2, 2, 2, 1] - s[3, 2, 1, 1, 1, 1]
 + s[3, 2, 2, 2] + s[3, 3, 1, 1, 1] - s[3, 3, 2, 1] + 2*s[3, 3, 3]
 + s[4, 1, 1, 1, 1, 1] - s[4, 3, 2] + s[4, 4, 1] - s[5, 1, 1, 1, 1]
 + s[5, 2, 2] - s[5, 4] + s[6, 1, 1, 1] - s[6, 2, 1] + s[6, 3]
>>> p(p3s21)
1/3*p[3, 3, 3] - 1/3*p[9]
>>> e = Sym.e()
```

(continues on next page)

(continued from previous page)

```
>>> e[Integer(3)](e[Integer(2)])
e[3, 3] + e[4, 1, 1] - 2*e[4, 2] - e[5, 1] + e[6]
```

Note that the output is in the basis of the input `x`:

```
sage: s[2,1](h[3])
h[4, 3, 2] - h[4, 4, 1] - h[5, 2, 2] + h[5, 3, 1] + h[5, 4]
 + h[6, 2, 1] - 2*h[6, 3] - h[7, 1, 1] + h[7, 2] + h[8, 1] - h[9]

sage: h[2,1](s[3])
s[4, 3, 2] + s[4, 4, 1] + s[5, 2, 2] + s[5, 3, 1] + s[5, 4]
 + s[6, 2, 1] + 2*s[6, 3] + 2*s[7, 2] + s[8, 1] + s[9]
```

```
>>> from sage.all import *
>>> s[Integer(2),Integer(1)](h[Integer(3)])
h[4, 3, 2] - h[4, 4, 1] - h[5, 2, 2] + h[5, 3, 1] + h[5, 4]
 + h[6, 2, 1] - 2*h[6, 3] - h[7, 1, 1] + h[7, 2] + h[8, 1] - h[9]

>>> h[Integer(2),Integer(1)](s[Integer(3)])
s[4, 3, 2] + s[4, 4, 1] + s[5, 2, 2] + s[5, 3, 1] + s[5, 4]
 + s[6, 2, 1] + 2*s[6, 3] + 2*s[7, 2] + s[8, 1] + s[9]
```

Examples over a polynomial ring:

```
sage: R.<t> = QQ[]
sage: s = SymmetricFunctions(R).s()
sage: a = s([3])
sage: f = t * s([2])
sage: a(f)
t^3*s[2, 2, 2] + t^3*s[4, 2] + t^3*s[6]
sage: f(a)
t*s[4, 2] + t*s[6]
sage: s(0).plethysm(s[1])
0
sage: s(1).plethysm(s[1])
s[]
sage: s(1).plethysm(s(0))
s[]
```

```
>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> s = SymmetricFunctions(R).s()
>>> a = s([Integer(3)])
>>> f = t * s([Integer(2)])
>>> a(f)
t^3*s[2, 2, 2] + t^3*s[4, 2] + t^3*s[6]
>>> f(a)
t*s[4, 2] + t*s[6]
>>> s(Integer(0)).plethysm(s[Integer(1)])
0
>>> s(Integer(1)).plethysm(s[Integer(1)])
s[]
```

(continues on next page)

(continued from previous page)

```
>>> s(Integer(1)).plethysm(s(Integer(0)))
s[]
```

When x is a constant, then it is returned as an element of the base ring:

```
sage: s[3](2).parent() is R
True
```

```
>>> from sage.all import *
>>> s[Integer(3)](Integer(2)).parent() is R
True
```

Sage also handles plethysm of tensor products of symmetric functions:

```
sage: s = SymmetricFunctions(QQ).s()
sage: X = tensor([s[1],s[[]]])
sage: Y = tensor([s[[]],s[1]])
sage: s[1,1,1](X+Y)
s[] # s[1, 1, 1] + s[1] # s[1, 1] + s[1, 1] # s[1] + s[1, 1, 1] # s[]
sage: s[1,1,1](X*Y)
s[1, 1, 1] # s[3] + s[2, 1] # s[2, 1] + s[3] # s[1, 1, 1]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> X = tensor([s[Integer(1)],s[[]]])
>>> Y = tensor([s[[]],s[Integer(1)]])
>>> s[Integer(1),Integer(1),Integer(1)](X+Y)
s[] # s[1, 1, 1] + s[1] # s[1, 1] + s[1, 1] # s[1] + s[1, 1, 1] # s[]
>>> s[Integer(1),Integer(1),Integer(1)](X*Y)
s[1, 1, 1] # s[3] + s[2, 1] # s[2, 1] + s[3] # s[1, 1, 1]
```

One can use this to work with symmetric functions in two sets of commuting variables. For example, we verify the Cauchy identities (in degree 5):

```
sage: m = SymmetricFunctions(QQ).m()
sage: P5 = Partitions(5)
sage: sum(s[mu](X)*s[mu](Y) for mu in P5) == sum(m[mu](X)*h[mu](Y) for mu in␣
↪P5)
True
sage: sum(s[mu](X)*s[mu.conjugate()](Y) for mu in P5) ==␣
↪sum(m[mu](X)*e[mu](Y) for mu in P5)
True
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> P5 = Partitions(Integer(5))
>>> sum(s[mu](X)*s[mu](Y) for mu in P5) == sum(m[mu](X)*h[mu](Y) for mu in P5)
True
>>> sum(s[mu](X)*s[mu.conjugate()](Y) for mu in P5) == sum(m[mu](X)*e[mu](Y)␣
↪for mu in P5)
True
```

Sage can also do the plethysm with an element in the completion:

```
sage: s = SymmetricFunctions(QQ).s()
sage: L = LazySymmetricFunctions(s)
sage: f = s[2,1]
sage: g = L(s[1]) / (1 - L(s[1])); g
s[1] + (s[1,1]+s[2]) + (s[1,1,1]+2*s[2,1]+s[3])
 + (s[1,1,1,1]+3*s[2,1,1]+2*s[2,2]+3*s[3,1]+s[4])
 + (s[1,1,1,1,1]+4*s[2,1,1,1]+5*s[2,2,1]+6*s[3,1,1]+5*s[3,2]+4*s[4,1]+s[5])
 + ... + O^8
sage: fog = f(g)
sage: fog[:8]
[s[2, 1],
 s[1, 1, 1, 1] + 3*s[2, 1, 1] + 2*s[2, 2] + 3*s[3, 1] + s[4],
 2*s[1, 1, 1, 1, 1] + 8*s[2, 1, 1, 1] + 10*s[2, 2, 1]
 + 12*s[3, 1, 1] + 10*s[3, 2] + 8*s[4, 1] + 2*s[5],
 3*s[1, 1, 1, 1, 1, 1] + 17*s[2, 1, 1, 1, 1] + 30*s[2, 2, 1, 1]
 + 16*s[2, 2, 2] + 33*s[3, 1, 1, 1] + 54*s[3, 2, 1] + 16*s[3, 3]
 + 33*s[4, 1, 1] + 30*s[4, 2] + 17*s[5, 1] + 3*s[6],
 5*s[1, 1, 1, 1, 1, 1, 1] + 30*s[2, 1, 1, 1, 1, 1] + 70*s[2, 2, 1, 1, 1]
 + 70*s[2, 2, 2, 1] + 75*s[3, 1, 1, 1, 1] + 175*s[3, 2, 1, 1]
 + 105*s[3, 2, 2] + 105*s[3, 3, 1] + 100*s[4, 1, 1, 1] + 175*s[4, 2, 1]
 + 70*s[4, 3] + 75*s[5, 1, 1] + 70*s[5, 2] + 30*s[6, 1] + 5*s[7]]
sage: parent(fog)
Lazy completion of Symmetric Functions over Rational Field in the Schur basis
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> L = LazySymmetricFunctions(s)
>>> f = s[Integer(2),Integer(1)]
>>> g = L(s[Integer(1)]) / (Integer(1) - L(s[Integer(1)])); g
s[1] + (s[1,1]+s[2]) + (s[1,1,1]+2*s[2,1]+s[3])
 + (s[1,1,1,1]+3*s[2,1,1]+2*s[2,2]+3*s[3,1]+s[4])
 + (s[1,1,1,1,1]+4*s[2,1,1,1]+5*s[2,2,1]+6*s[3,1,1]+5*s[3,2]+4*s[4,1]+s[5])
 + ... + O^8
>>> fog = f(g)
>>> fog[:Integer(8)]
[s[2, 1],
 s[1, 1, 1, 1] + 3*s[2, 1, 1] + 2*s[2, 2] + 3*s[3, 1] + s[4],
 2*s[1, 1, 1, 1, 1] + 8*s[2, 1, 1, 1] + 10*s[2, 2, 1]
 + 12*s[3, 1, 1] + 10*s[3, 2] + 8*s[4, 1] + 2*s[5],
 3*s[1, 1, 1, 1, 1, 1] + 17*s[2, 1, 1, 1, 1] + 30*s[2, 2, 1, 1]
 + 16*s[2, 2, 2] + 33*s[3, 1, 1, 1] + 54*s[3, 2, 1] + 16*s[3, 3]
 + 33*s[4, 1, 1] + 30*s[4, 2] + 17*s[5, 1] + 3*s[6],
 5*s[1, 1, 1, 1, 1, 1, 1] + 30*s[2, 1, 1, 1, 1, 1] + 70*s[2, 2, 1, 1, 1]
 + 70*s[2, 2, 2, 1] + 75*s[3, 1, 1, 1, 1] + 175*s[3, 2, 1, 1]
 + 105*s[3, 2, 2] + 105*s[3, 3, 1] + 100*s[4, 1, 1, 1] + 175*s[4, 2, 1]
 + 70*s[4, 3] + 75*s[5, 1, 1] + 70*s[5, 2] + 30*s[6, 1] + 5*s[7]]
>>> parent(fog)
Lazy completion of Symmetric Functions over Rational Field in the Schur basis
```

See also

adams_operator()

Todo

The implementation of plethysm in `sage.data_structures.stream.Stream_plethysm` seems to be faster. This should be investigated.

principal_specialization(*n=+Infinity*, *q=None*)

Return the principal specialization of a symmetric function.

The *principal specialization* of order n at q is the ring homomorphism $ps_{n,q}$ from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for $i \in \{1, \ldots, n\}$ and $x_i \mapsto 0$ for $i > n$. Here, q is a given element of R, and we assume that the variables of our symmetric functions are $x_1, x_2, x_3, \ldots$. (To be more precise, $ps_{n,q}$ is a K-algebra homomorphism, where K is the base ring.) See Section 7.8 of [EnumComb2].

The *stable principal specialization* at q is the ring homomorphism ps_q from the ring of symmetric functions to another commutative ring R given by $x_i \mapsto q^{i-1}$ for all i. This is well-defined only if the resulting infinite sums converge; thus, in particular, setting $q = 1$ in the stable principal specialization is an invalid operation.

INPUT:

- `n` – (default: `infinity`) a nonnegative integer or `infinity`, specifying whether to compute the principal specialization of order `n` or the stable principal specialization.
- `q` – (default: `None`) the value to use for q; the default is to create a ring of polynomials in `q` (or a field of rational functions in `q`) over the given coefficient ring.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: x = m[1,1]
sage: x.principal_specialization(3)
q^3 + q^2 + q
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
>>> x = m[Integer(1),Integer(1)]
>>> x.principal_specialization(Integer(3))
q^3 + q^2 + q
```

By default we return a rational function in `q`. Sometimes it is better to obtain an element of the symbolic ring:

```
sage: h = SymmetricFunctions(QQ).h()
sage: (h[3]+h[2]).principal_specialization(q=var("q"))                    #␣
↪needs sage.symbolic
1/((q^2 - 1)*(q - 1)) - 1/((q^3 - 1)*(q^2 - 1)*(q - 1))
```

```
>>> from sage.all import *
>>> h = SymmetricFunctions(QQ).h()
>>> (h[Integer(3)]+h[Integer(2)]).principal_specialization(q=var("q"))     ␣
↪              # needs sage.symbolic
1/((q^2 - 1)*(q - 1)) - 1/((q^3 - 1)*(q^2 - 1)*(q - 1))
```

In case `q` is in the base ring, it must be passed explicitly:

```
sage: R = QQ['q,t']
sage: Ht = SymmetricFunctions(R).macdonald().Ht()
sage: Ht[2].principal_specialization()
Traceback (most recent call last):
...
ValueError: the variable q is in the base ring, pass it explicitly

sage: Ht[2].principal_specialization(q=R("q"))
(q^2 + 1)/(q^3 - q^2 - q + 1)
```

```
>>> from sage.all import *
>>> R = QQ['q,t']
>>> Ht = SymmetricFunctions(R).macdonald().Ht()
>>> Ht[Integer(2)].principal_specialization()
Traceback (most recent call last):
...
ValueError: the variable q is in the base ring, pass it explicitly

>>> Ht[Integer(2)].principal_specialization(q=R("q"))
(q^2 + 1)/(q^3 - q^2 - q + 1)
```

Note that the principal specialization can be obtained as a plethysm:

```
sage: R = QQ['q'].fraction_field()
sage: s = SymmetricFunctions(R).s()
sage: one = s.one()
sage: q = R("q")
sage: f = s[3,2,2]
sage: f.principal_specialization(q=q) == f(one/(1-q)).coefficient([])
True
sage: f.principal_specialization(n=4, q=q) == f(one*(1-q^4)/(1-q)).
↪coefficient([])
True
```

```
>>> from sage.all import *
>>> R = QQ['q'].fraction_field()
>>> s = SymmetricFunctions(R).s()
>>> one = s.one()
>>> q = R("q")
>>> f = s[Integer(3),Integer(2),Integer(2)]
>>> f.principal_specialization(q=q) == f(one/(Integer(1)-q)).coefficient([])
True
>>> f.principal_specialization(n=Integer(4), q=q) == f(one*(Integer(1)-
↪q**Integer(4))/(Integer(1)-q)).coefficient([])
True
```

reduced_kronecker_product(*x*)

Return the reduced Kronecker product of `self` and `x` in the basis of `self`.

The reduced Kronecker product is a bilinear map mapping two symmetric functions to another, not necessarily preserving degree. It can be defined as follows: Let $*$ denote the Kronecker product (*itensor()*) on the space of symmetric functions. For any partitions α, β, γ, let $g^{\gamma}_{\alpha,\beta}$ denote the coefficient of the Schur function s_γ in the Kronecker product $s_\alpha * s_\beta$ (this is called a Kronecker coefficient). For every partition

$\lambda = (\lambda_1, \lambda_2, \lambda_3, \ldots)$ and every integer $n > |\lambda| + \lambda_1$, let $\lambda[n]$ denote the n-completion of λ (this is the partition $(n - |\lambda|, \lambda_1, \lambda_2, \lambda_3, \ldots)$; see `t_completion()`). Then, Theorem 1.2 of [BOR2009] shows that for any partitions α and β and every integer $n \geq |\alpha| + |\beta| + \alpha_1 + \beta_1$, we can write the Kronecker product $s_{\alpha[n]} * s_{\beta[n]}$ in the form

$$s_{\alpha[n]} * s_{\beta[n]} = \sum_{\gamma} g_{\alpha[n],\beta[n]}^{\gamma[n]} s_{\gamma[n]}$$

with γ ranging over all partitions. The coefficients $g_{\alpha[n],\beta[n]}^{\gamma[n]}$ are independent on n. These coefficients $g_{\alpha[n],\beta[n]}^{\gamma[n]}$ are denoted by $\overline{g}_{\alpha,\beta}^{\gamma}$, and the symmetric function

$$\sum_{\gamma} \overline{g}_{\alpha,\beta}^{\gamma} s_{\gamma}$$

is said to be the *reduced Kronecker product* of s_α and s_β. By bilinearity, this extends to a definition of a reduced Kronecker product of any two symmetric functions.

The definition of the reduced Kronecker product goes back to Murnaghan, and has recently been studied in [BOR2009], [BdVO2012] and other places (our notation $\overline{g}_{\alpha,\beta}^{\gamma}$ appears in these two sources).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the reduced Kronecker product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

EXAMPLES:

The example from page 2 of [BOR2009]:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: s[2].reduced_kronecker_product(s[2])
s[] + s[1] + s[1, 1] + s[1, 1, 1] + 2*s[2] + 2*s[2, 1] + s[2, 2] + s[3] + s[3,
↪ 1] + s[4]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.schur()
>>> s[Integer(2)].reduced_kronecker_product(s[Integer(2)])
s[] + s[1] + s[1, 1] + s[1, 1, 1] + 2*s[2] + 2*s[2, 1] + s[2, 2] + s[3] + s[3,
↪ 1] + s[4]
```

Taking the reduced Kronecker product with $1 = s_{\emptyset}$ is the identity map on the ring of symmetric functions:

```
sage: all( s[Partition([])].reduced_kronecker_product(s[lam])
....:      == s[lam] for i in range(4)
....:      for lam in Partitions(i) )
True
```

```
>>> from sage.all import *
>>> all( s[Partition([])].reduced_kronecker_product(s[lam])
...      == s[lam] for i in range(Integer(4))
...      for lam in Partitions(i) )
True
```

While reduced Kronecker products are hard to compute in general, there is a rule for taking reduced Kronecker products with s_1. Namely, for every partition λ, the reduced Kronecker product of s_λ with s_1 is $\sum_\mu a_\mu s_\mu$, where the sum runs over all partitions μ, and the coefficient a_μ is defined as the number of ways to obtain μ from λ by one of the following three operations:

- Add an addable cell (`addable_cells()`) to λ.
- Remove a removable cell (`removable_cells()`) from λ.
- First remove a removable cell from λ, then add an addable cell to the resulting Young diagram.

This is, in fact, Proposition 5.15 of [CO2010] in an elementary wording. We check this for partitions of size ≤ 4:

```
sage: def mults1(lam):
....:     # Reduced Kronecker multiplication by s[1], according
....:     # to [CO2010]_.
....:     res = s.zero()
....:     for mu in lam.up_list():
....:         res += s(mu)
....:     for mu in lam.down_list():
....:         res += s(mu)
....:         for nu in mu.up_list():
....:             res += s(nu)
....:     return res
sage: all( mults1(lam) == s[1].reduced_kronecker_product(s[lam])
....:      for i in range(5) for lam in Partitions(i) )
True
```

```
>>> from sage.all import *
>>> def mults1(lam):
...     # Reduced Kronecker multiplication by s[1], according
...     # to [CO2010]_.
...     res = s.zero()
...     for mu in lam.up_list():
...         res += s(mu)
...     for mu in lam.down_list():
...         res += s(mu)
...         for nu in mu.up_list():
...             res += s(nu)
...     return res
>>> all( mults1(lam) == s[Integer(1)].reduced_kronecker_product(s[lam])
...      for i in range(Integer(5)) for lam in Partitions(i) )
True
```

Here is the example on page 3 of Christian Gutschwager's arXiv 0912.4411v3:

```
sage: s[1,1].reduced_kronecker_product(s[2])
s[1] + 2*s[1, 1] + s[1, 1, 1] + s[2] + 2*s[2, 1] + s[2, 1, 1] + s[3] + s[3, 1]
```

```
>>> from sage.all import *
>>> s[Integer(1),Integer(1)].reduced_kronecker_product(s[Integer(2)])
s[1] + 2*s[1, 1] + s[1, 1, 1] + s[2] + 2*s[2, 1] + s[2, 1, 1] + s[3] + s[3, 1]
```

Example 39 from F. D. Murnaghan, "The analysis of the Kronecker product of irreducible representations of the symmetric group", American Journal of Mathematics, Vol. 60, No. 3, Jul. 1938:

```
sage: s[3].reduced_kronecker_product(s[2,1])
s[1] + 2*s[1, 1] + 2*s[1, 1, 1] + s[1, 1, 1, 1] + 2*s[2] + 5*s[2, 1] + 4*s[2,␣
↪1, 1]
+ s[2, 1, 1, 1] + 3*s[2, 2] + 2*s[2, 2, 1] + 2*s[3] + 5*s[3, 1] + 3*s[3, 1, 1]
+ 3*s[3, 2] + s[3, 2, 1] + 2*s[4] + 3*s[4, 1] + s[4, 1, 1] + s[4, 2] + s[5]
+ s[5, 1]
```

```
>>> from sage.all import *
>>> s[Integer(3)].reduced_kronecker_product(s[Integer(2),Integer(1)])
s[1] + 2*s[1, 1] + 2*s[1, 1, 1] + s[1, 1, 1, 1] + 2*s[2] + 5*s[2, 1] + 4*s[2,␣
↪1, 1]
+ s[2, 1, 1, 1] + 3*s[2, 2] + 2*s[2, 2, 1] + 2*s[3] + 5*s[3, 1] + 3*s[3, 1, 1]
+ 3*s[3, 2] + s[3, 2, 1] + 2*s[4] + 3*s[4, 1] + s[4, 1, 1] + s[4, 2] + s[5]
+ s[5, 1]
```

Todo

This implementation of the reduced Kronecker product is painfully slow.

restrict_degree(*d*, *exact=True*)

Return the degree `d` component of `self`.

INPUT:

- `d` – positive integer, degree of the terms to be returned
- `exact` – boolean, if `True`, returns the terms of degree exactly `d`, otherwise returns all terms of degree less than or equal to `d`

OUTPUT: the homogeneous component of `self` of degree `d`

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_degree(2)
0
sage: z.restrict_degree(1)
s[1]
sage: z.restrict_degree(3)
s[1, 1, 1] + s[2, 1]
sage: z.restrict_degree(3, exact=False)
s[1] + s[1, 1, 1] + s[2, 1]
sage: z.restrict_degree(0)
0
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> z = s([Integer(4)]) + s([Integer(2),Integer(1)]) + s([Integer(1),
↪Integer(1),Integer(1)]) + s([Integer(1)])
>>> z.restrict_degree(Integer(2))
0
>>> z.restrict_degree(Integer(1))
```

(continues on next page)

(continued from previous page)

```
s[1]
>>> z.restrict_degree(Integer(3))
s[1, 1, 1] + s[2, 1]
>>> z.restrict_degree(Integer(3), exact=False)
s[1] + s[1, 1, 1] + s[2, 1]
>>> z.restrict_degree(Integer(0))
0
```

restrict_partition_lengths(*l*, *exact=True*)

Return the terms of `self` labelled by partitions of length `l`.

INPUT:

- `l` – nonnegative integer
- `exact` – boolean, defaulting to `True`

OUTPUT:

- if `True`, returns the terms labelled by partitions of length precisely `l`; otherwise returns all terms labelled by partitions of length less than or equal to `l`

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_partition_lengths(2)
s[2, 1]
sage: z.restrict_partition_lengths(0)
0
sage: z.restrict_partition_lengths(2, exact = False)
s[1] + s[2, 1] + s[4]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> z = s([Integer(4)]) + s([Integer(2),Integer(1)]) + s([Integer(1),
↪Integer(1),Integer(1)]) + s([Integer(1)])
>>> z.restrict_partition_lengths(Integer(2))
s[2, 1]
>>> z.restrict_partition_lengths(Integer(0))
0
>>> z.restrict_partition_lengths(Integer(2), exact = False)
s[1] + s[2, 1] + s[4]
```

restrict_parts(*n*)

Return the terms of `self` labelled by partitions λ with $\lambda_1 \leq n$.

INPUT:

- `n` – positive integer, to restrict the parts of the partitions of the terms to be returned

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_parts(2)
```

(continues on next page)

(continued from previous page)

```
s[1] + s[1, 1, 1] + s[2, 1]
sage: z.restrict_parts(1)
s[1] + s[1, 1, 1]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> z = s([Integer(4)]) + s([Integer(2),Integer(1)]) + s([Integer(1),
↪Integer(1),Integer(1)]) + s([Integer(1)])
>>> z.restrict_parts(Integer(2))
s[1] + s[1, 1, 1] + s[2, 1]
>>> z.restrict_parts(Integer(1))
s[1] + s[1, 1, 1]
```

scalar(*x*, *zee=None*)

Return the standard scalar product between `self` and `x`.

This is also known as the "Hall inner product" or the "Hall scalar product".

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `zee` – an optional function on partitions giving the value for the scalar product between p_μ and p_μ (default: the standard `zee()` function)

This is the default implementation that converts both `self` and `x` into either Schur functions (if `zee` is not specified) or power-sum functions (if `zee` is specified) and performs the scalar product in that basis.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: h = SymmetricFunctions(QQ).h()
sage: m = SymmetricFunctions(QQ).m()
sage: p4 = Partitions(4)
sage: matrix([ [e(a).scalar(h(b)) for a in p4] for b in p4])
[ 0  0  0  0  1]
[ 0  0  0  1  4]
[ 0  0  1  2  6]
[ 0  1  2  5 12]
[ 1  4  6 12 24]
sage: matrix([ [h(a).scalar(e(b)) for a in p4] for b in p4])
[ 0  0  0  0  1]
[ 0  0  0  1  4]
[ 0  0  1  2  6]
[ 0  1  2  5 12]
[ 1  4  6 12 24]
sage: matrix([ [m(a).scalar(e(b)) for a in p4] for b in p4])
[-1  2  1 -3  1]
[ 0  1  0 -2  1]
[ 0  0  1 -2  1]
[ 0  0  0 -1  1]
[ 0  0  0  0  1]
sage: matrix([ [m(a).scalar(h(b)) for a in p4] for b in p4])
[1 0 0 0 0]
[0 1 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]

sage: p = SymmetricFunctions(QQ).p()
sage: m(p[3,2]).scalar(p[3,2], zee=lambda mu: 2**mu.length())
4
sage: m(p[3,2]).scalar(p[2,2,1], lambda mu: 1)
0
sage: m[3,2].scalar(h[3,2], zee=lambda mu: 2**mu.length())
2/3
```

```
>>> from sage.all import *
>>> e = SymmetricFunctions(QQ).e()
>>> h = SymmetricFunctions(QQ).h()
>>> m = SymmetricFunctions(QQ).m()
>>> p4 = Partitions(Integer(4))
>>> matrix([ [e(a).scalar(h(b)) for a in p4] for b in p4])
[ 0  0  0  0  1]
[ 0  0  0  1  4]
[ 0  0  1  2  6]
[ 0  1  2  5 12]
[ 1  4  6 12 24]
>>> matrix([ [h(a).scalar(e(b)) for a in p4] for b in p4])
[ 0  0  0  0  1]
[ 0  0  0  1  4]
[ 0  0  1  2  6]
[ 0  1  2  5 12]
[ 1  4  6 12 24]
>>> matrix([ [m(a).scalar(e(b)) for a in p4] for b in p4])
[-1  2  1 -3  1]
[ 0  1  0 -2  1]
[ 0  0  1 -2  1]
[ 0  0  0 -1  1]
[ 0  0  0  0  1]
>>> matrix([ [m(a).scalar(h(b)) for a in p4] for b in p4])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]

>>> p = SymmetricFunctions(QQ).p()
>>> m(p[Integer(3),Integer(2)]).scalar(p[Integer(3),Integer(2)], zee=lambda␣
↪mu: Integer(2)**mu.length())
4
>>> m(p[Integer(3),Integer(2)]).scalar(p[Integer(2),Integer(2),Integer(1)],␣
↪lambda mu: Integer(1))
0
>>> m[Integer(3),Integer(2)].scalar(h[Integer(3),Integer(2)], zee=lambda mu:␣
↪Integer(2)**mu.length())
2/3
```

scalar_hl(*x*, *t=None*)

Return the t-deformed standard Hall-Littlewood scalar product of `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `t` – parameter (default: `None`, in which case `t` is used)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: sp = a.scalar_t(a); sp
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
sage: sp.parent()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> a = s([Integer(2),Integer(1)])
>>> sp = a.scalar_t(a); sp
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
>>> sp.parent()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

scalar_jack(*x*, *t=None*)

Return the Jack-scalar product between `self` and `x`.

This scalar product is defined so that the power sum elements p_μ are orthogonal and $\langle p_\mu, p_\mu \rangle = z_\mu t^{\ell(\mu)}$, where $\ell(\mu)$ denotes the length of μ.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `t` – an optional parameter (default: `None` in which case `t` is used)

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ['t']).power()
sage: matrix([[p(mu).scalar_jack(p(nu)) for nu in Partitions(4)] for mu in␣
→Partitions(4)])
[   4*t       0       0       0       0]
[     0   3*t^2       0       0       0]
[     0       0   8*t^2       0       0]
[     0       0       0   4*t^3       0]
[     0       0       0       0 24*t^4]
sage: matrix([[p(mu).scalar_jack(p(nu),2) for nu in Partitions(4)] for mu in␣
→Partitions(4)])
[  8   0   0   0   0]
[  0  12   0   0   0]
[  0   0  32   0   0]
[  0   0   0  32   0]
[  0   0   0   0 384]
sage: JQ = SymmetricFunctions(QQ['t'].fraction_field()).jack().Q()
sage: matrix([[JQ(mu).scalar_jack(JQ(nu)) for nu in Partitions(3)] for mu in␣
```

(continues on next page)

(continued from previous page)

```
↪Partitions(3)])
[(1/3*t^2 + 1/2*t + 1/6)/t^3                           0                       ␣
↪      0]
[                          0  (1/2*t + 1)/(t^3 + 1/2*t^2)                      ␣
↪      0]
[                          0                           0        6/(t^3 + 3*t^2␣
↪+ 2*t)]
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ['t']).power()
>>> matrix([[p(mu).scalar_jack(p(nu)) for nu in Partitions(Integer(4))] for␣
↪mu in Partitions(Integer(4))])
[   4*t       0       0       0       0]
[     0   3*t^2       0       0       0]
[     0       0   8*t^2       0       0]
[     0       0       0   4*t^3       0]
[     0       0       0       0  24*t^4]
>>> matrix([[p(mu).scalar_jack(p(nu),Integer(2)) for nu in␣
↪Partitions(Integer(4))] for mu in Partitions(Integer(4))])
[  8   0   0   0   0]
[  0  12   0   0   0]
[  0   0  32   0   0]
[  0   0   0  32   0]
[  0   0   0   0 384]
>>> JQ = SymmetricFunctions(QQ['t'].fraction_field()).jack().Q()
>>> matrix([[JQ(mu).scalar_jack(JQ(nu)) for nu in Partitions(Integer(3))] for␣
↪mu in Partitions(Integer(3))])
[(1/3*t^2 + 1/2*t + 1/6)/t^3                           0                       ␣
↪      0]
[                          0  (1/2*t + 1)/(t^3 + 1/2*t^2)                      ␣
↪      0]
[                          0                           0        6/(t^3 + 3*t^2␣
↪+ 2*t)]
```

scalar_qt(*x*, *q=None*, *t=None*)

Return the q, t-deformed standard Hall-Littlewood scalar product of `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `q`, `t` – parameters (default: `None` in which case `q` and `t` are used)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: sp = a.scalar_qt(a); factor(sp)
(t - 1)^-3 * (q - 1) * (t^2 + t + 1)^-1 * (q^2*t^2 - q*t^2 + q^2 - 2*q*t + t^
↪2 - q + 1)
sage: sp.parent()
Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field
sage: a.scalar_qt(a,q=0)
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
```

(continues on next page)

(continued from previous page)

```
sage: a.scalar_qt(a,t=0)
-q^3 + 2*q^2 - 2*q + 1
sage: a.scalar_qt(a,5,7) # q=5 and t=7
490/1539
sage: (x,y) = var('x,y')                                                    #␣
↪needs sage.symbolic
sage: a.scalar_qt(a, q=x, t=y)                                              #␣
↪needs sage.symbolic
1/3*(x^3 - 1)/(y^3 - 1) + 2/3*(x - 1)^3/(y - 1)^3
sage: Rn = QQ['q','t','y','z'].fraction_field()
sage: (q,t,y,z) = Rn.gens()
sage: Mac = SymmetricFunctions(Rn).macdonald(q=y,t=z)
sage: a = Mac._sym.schur()([2,1])
sage: factor(Mac.P()(a).scalar_qt(Mac.Q()(a),q,t))
(t - 1)^-3 * (q - 1) * (t^2 + t + 1)^-1 * (q^2*t^2 - q*t^2 + q^2 - 2*q*t + t^
↪2 - q + 1)
sage: factor(Mac.P()(a).scalar_qt(Mac.Q()(a)))
(z - 1)^-3 * (y - 1) * (z^2 + z + 1)^-1 * (y^2*z^2 - y*z^2 + y^2 - 2*y*z + z^
↪2 - y + 1)
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> a = s([Integer(2),Integer(1)])
>>> sp = a.scalar_qt(a); factor(sp)
(t - 1)^-3 * (q - 1) * (t^2 + t + 1)^-1 * (q^2*t^2 - q*t^2 + q^2 - 2*q*t + t^
↪2 - q + 1)
>>> sp.parent()
Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field
>>> a.scalar_qt(a,q=Integer(0))
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
>>> a.scalar_qt(a,t=Integer(0))
-q^3 + 2*q^2 - 2*q + 1
>>> a.scalar_qt(a,Integer(5),Integer(7)) # q=5 and t=7
490/1539
>>> (x,y) = var('x,y')                                                    #␣
↪needs sage.symbolic
>>> a.scalar_qt(a, q=x, t=y)                                              #␣
↪needs sage.symbolic
1/3*(x^3 - 1)/(y^3 - 1) + 2/3*(x - 1)^3/(y - 1)^3
>>> Rn = QQ['q','t','y','z'].fraction_field()
>>> (q,t,y,z) = Rn.gens()
>>> Mac = SymmetricFunctions(Rn).macdonald(q=y,t=z)
>>> a = Mac._sym.schur()([Integer(2),Integer(1)])
>>> factor(Mac.P()(a).scalar_qt(Mac.Q()(a),q,t))
(t - 1)^-3 * (q - 1) * (t^2 + t + 1)^-1 * (q^2*t^2 - q*t^2 + q^2 - 2*q*t + t^
↪2 - q + 1)
>>> factor(Mac.P()(a).scalar_qt(Mac.Q()(a)))
(z - 1)^-3 * (y - 1) * (z^2 + z + 1)^-1 * (y^2*z^2 - y*z^2 + y^2 - 2*y*z + z^
↪2 - y + 1)
```

scalar_t(*x*, *t=None*)

Return the t-deformed standard Hall-Littlewood scalar product of `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `t` – parameter (default: `None`, in which case `t` is used)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: sp = a.scalar_t(a); sp
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
sage: sp.parent()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> a = s([Integer(2),Integer(1)])
>>> sp = a.scalar_t(a); sp
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
>>> sp.parent()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

skew_by(*x*)

Return the result of skewing `self` by `x`. (Skewing by `x` is the endomorphism (as additive group) of the ring of symmetric functions adjoint to multiplication by `x` with respect to the Hall inner product.)

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([3,2]).skew_by(s([2]))
s[2, 1] + s[3]
sage: s([3,2]).skew_by(s([1,1,1]))
0
sage: s([3,2,1]).skew_by(s([2,1]))
s[1, 1, 1] + 2*s[2, 1] + s[3]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s([Integer(3),Integer(2)]).skew_by(s([Integer(2)]))
s[2, 1] + s[3]
>>> s([Integer(3),Integer(2)]).skew_by(s([Integer(1),Integer(1),Integer(1)]))
0
>>> s([Integer(3),Integer(2),Integer(1)]).skew_by(s([Integer(2),Integer(1)]))
s[1, 1, 1] + 2*s[2, 1] + s[3]
```

```
sage: p = SymmetricFunctions(QQ).powersum()
sage: p([4,3,3,2,2,1]).skew_by(p([2,1]))
4*p[4, 3, 3, 2]
sage: zee = sage.combinat.sf.sfa.zee
sage: zee([4,3,3,2,2,1])/zee([4,3,3,2])
```

(continues on next page)

(continued from previous page)

```
4
sage: s(0).skew_by(s([1]))
0
sage: s(1).skew_by(s([1]))
0
sage: s([]).skew_by(s([]))
s[]
sage: s([]).skew_by(s[1])
0
```

```
>>> from sage.all import *
>>> p = SymmetricFunctions(QQ).powersum()
>>> p([Integer(4),Integer(3),Integer(3),Integer(2),Integer(2),Integer(1)]).
↪skew_by(p([Integer(2),Integer(1)]))
4*p[4, 3, 3, 2]
>>> zee = sage.combinat.sf.sfa.zee
>>> zee([Integer(4),Integer(3),Integer(3),Integer(2),Integer(2),Integer(1)])/
↪zee([Integer(4),Integer(3),Integer(3),Integer(2)])
4
>>> s(Integer(0)).skew_by(s([Integer(1)]))
0
>>> s(Integer(1)).skew_by(s([Integer(1)]))
0
>>> s([]).skew_by(s([]))
s[]
>>> s([]).skew_by(s[Integer(1)])
0
```

theta(a)

Return the image of `self` under the theta endomorphism which sends p_k to $a \cdot p_k$ for every positive integer k.

In general, this is well-defined outside of the powersum basis only if the base ring is a **Q**-algebra.

INPUT:

- `a` – an element of the base ring

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([2,1]).theta(2)
2*s[1, 1, 1] + 6*s[2, 1] + 2*s[3]
sage: p = SymmetricFunctions(QQ).p()
sage: p([2]).theta(2)
2*p[2]
sage: p(0).theta(2)
0
sage: p(1).theta(2)
p[]
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
```

(continues on next page)

(continued from previous page)

```
>>> s([Integer(2),Integer(1)]).theta(Integer(2))
2*s[1, 1, 1] + 6*s[2, 1] + 2*s[3]
>>> p = SymmetricFunctions(QQ).p()
>>> p([Integer(2)]).theta(Integer(2))
2*p[2]
>>> p(Integer(0)).theta(Integer(2))
0
>>> p(Integer(1)).theta(Integer(2))
p[]
```

theta_qt(*q=None*, *t=None*)

Return the image of `self` under the q,t-deformed theta endomorphism which sends p_k to $\frac{1-q^k}{1-t^k} \cdot p_k$ for all positive integers k.

In general, this is well-defined outside of the powersum basis only if the base ring is a **Q**-algebra.

INPUT:

- `q`, `t` – parameters (default: `None`, in which case 'q' and 't' are used)

EXAMPLES:

```
sage: QQqt = QQ['q,t'].fraction_field()
sage: q,t = QQqt.gens()
sage: p = SymmetricFunctions(QQqt).p()
sage: p([2]).theta_qt(q,t)
-((q^2-1)/(-t^2+1))*p[2]
sage: p([2,1]).theta_qt(q,t)
((q^3-q^2-q+1)/(t^3-t^2-t+1))*p[2, 1]
sage: p(0).theta_qt(q=1,t=3)
0
sage: p([2,1]).theta_qt(q=2,t=3)
3/16*p[2, 1]
sage: s = p.realization_of().schur()
sage: s([3]).theta_qt(q=0)*(1-t)*(1-t^2)*(1-t^3)
t^3*s[1, 1, 1] + (t^2+t)*s[2, 1] + s[3]
sage: p(1).theta_qt()
p[]
```

```
>>> from sage.all import *
>>> QQqt = QQ['q,t'].fraction_field()
>>> q,t = QQqt.gens()
>>> p = SymmetricFunctions(QQqt).p()
>>> p([Integer(2)]).theta_qt(q,t)
-((q^2-1)/(-t^2+1))*p[2]
>>> p([Integer(2),Integer(1)]).theta_qt(q,t)
((q^3-q^2-q+1)/(t^3-t^2-t+1))*p[2, 1]
>>> p(Integer(0)).theta_qt(q=Integer(1),t=Integer(3))
0
>>> p([Integer(2),Integer(1)]).theta_qt(q=Integer(2),t=Integer(3))
3/16*p[2, 1]
>>> s = p.realization_of().schur()
>>> s([Integer(3)]).theta_qt(q=Integer(0))*(Integer(1)-t)*(Integer(1)-
```

(continues on next page)

(continued from previous page)

```
↪t**Integer(2))*(Integer(1)-t**Integer(3))
t^3*s[1, 1, 1] + (t^2+t)*s[2, 1] + s[3]
>>> p(Integer(1)).theta_qt()
p[]
```

verschiebung(*n*)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n}e_{r/n}$$

(where h is the complete homogeneous basis, p is the powersum basis, and e is the elementary basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for "shift") endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Adams operator (see `adams_operator()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley's [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, \ldots, \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, \ldots, \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi\prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: s[3].verschiebung(2)
0
```

(continues on next page)

(continued from previous page)

```
sage: s[3].verschiebung(3)
s[1]
sage: p[3].verschiebung(3)
3*p[1]
sage: m[3,2,1].verschiebung(3)
-18*m[1, 1] - 3*m[2]
sage: p[3,2,1].verschiebung(3)
0
sage: h[4].verschiebung(2)
h[2]
sage: p[2].verschiebung(2)
2*p[1]
sage: m[3,2,1].verschiebung(6)
12*m[1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> p = Sym.p()
>>> h = Sym.h()
>>> s = Sym.s()
>>> m = Sym.m()
>>> s[Integer(3)].verschiebung(Integer(2))
0
>>> s[Integer(3)].verschiebung(Integer(3))
s[1]
>>> p[Integer(3)].verschiebung(Integer(3))
3*p[1]
>>> m[Integer(3),Integer(2),Integer(1)].verschiebung(Integer(3))
-18*m[1, 1] - 3*m[2]
>>> p[Integer(3),Integer(2),Integer(1)].verschiebung(Integer(3))
0
>>> h[Integer(4)].verschiebung(Integer(2))
h[2]
>>> p[Integer(2)].verschiebung(Integer(2))
2*p[1]
>>> m[Integer(3),Integer(2),Integer(1)].verschiebung(Integer(6))
12*m[1]
```

The Verschiebung endomorphisms are multiplicative:

```
sage: all( all( s(lam).verschiebung(2) * s(mu).verschiebung(2)
....:           == (s(lam) * s(mu)).verschiebung(2)
....:           for mu in Partitions(4) )
....:      for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> all( all( s(lam).verschiebung(Integer(2)) * s(mu).verschiebung(Integer(2))
...           == (s(lam) * s(mu)).verschiebung(Integer(2))
...           for mu in Partitions(Integer(4)) )
...      for lam in Partitions(Integer(4)) )
True
```

Being Hopf algebra endomorphisms, the Verschiebung operators commute with the antipode:

```
sage: all( p(lam).verschiebung(3).antipode()
....:      == p(lam).antipode().verschiebung(3)
....:      for lam in Partitions(6) )
True
```

```
>>> from sage.all import *
>>> all( p(lam).verschiebung(Integer(3)).antipode()
...      == p(lam).antipode().verschiebung(Integer(3))
...      for lam in Partitions(Integer(6)) )
True
```

Testing the adjointness between the Adams operators $\mathbf{f}_n$ and the Verschiebung operators $\mathbf{V}_n$:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: p = Sym.p()
sage: all( all( s(lam).verschiebung(2).scalar(p(mu))
....:           == s(lam).scalar(p(mu).adams_operator(2))
....:           for mu in Partitions(3) )
....:      for lam in Partitions(6) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> p = Sym.p()
>>> all( all( s(lam).verschiebung(Integer(2)).scalar(p(mu))
...           == s(lam).scalar(p(mu).adams_operator(Integer(2)))
...           for mu in Partitions(Integer(3)) )
...      for lam in Partitions(Integer(6)) )
True
```

class sage.combinat.sf.sfa.**SymmetricFunctionsBases**(*parent_with_realization*)

Bases: `Category_realization_of_parent`

The category of bases of the ring of symmetric functions.

INPUT:

- `self` – a category of bases for the symmetric functions
- `base` – ring of symmetric functions

class ParentMethods

Bases: `object`

Eulerian(*n, j, k=None*)

Return the Eulerian symmetric function $Q_{n,j}$ (with n either an integer or a partition) or $Q_{n,j,k}$ (if the optional argument `k` is specified) in terms of the basis `self`.

It is known that the Eulerian quasisymmetric functions are in fact symmetric functions [SW2010]. For more information, see `QuasiSymmetricFunctions.Fundamental.Eulerian()`, which accepts the same syntax as this method.

INPUT:

- n – the nonnegative integer n or a partition
- j – the number of excedances
- k – (optional) if specified, determines the number of fixed points of the permutations which are being summed over

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: m.Eulerian(3, 1)
4*m[1, 1, 1] + 3*m[2, 1] + 2*m[3]
sage: h = Sym.h()
sage: h.Eulerian(4, 2)
h[2, 2] + h[3, 1] + h[4]
sage: s = Sym.s()
sage: s.Eulerian(5, 2)
s[2, 2, 1] + s[3, 1, 1] + 5*s[3, 2] + 6*s[4, 1] + 6*s[5]
sage: s.Eulerian([2,2,1], 2)
s[2, 2, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s.Eulerian(5, 2, 2)
s[3, 2] + s[4, 1] + s[5]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.m()
>>> m.Eulerian(Integer(3), Integer(1))
4*m[1, 1, 1] + 3*m[2, 1] + 2*m[3]
>>> h = Sym.h()
>>> h.Eulerian(Integer(4), Integer(2))
h[2, 2] + h[3, 1] + h[4]
>>> s = Sym.s()
>>> s.Eulerian(Integer(5), Integer(2))
s[2, 2, 1] + s[3, 1, 1] + 5*s[3, 2] + 6*s[4, 1] + 6*s[5]
>>> s.Eulerian([Integer(2),Integer(2),Integer(1)], Integer(2))
s[2, 2, 1] + s[3, 2] + s[4, 1] + s[5]
>>> s.Eulerian(Integer(5), Integer(2), Integer(2))
s[3, 2] + s[4, 1] + s[5]
```

We check Equation (5.4) in [SW2010]:

```
sage: h.Eulerian([6], 3)
h[3, 2, 1] - h[4, 1, 1] + 2*h[4, 2] + h[5, 1]
sage: s.Eulerian([6], 3)
s[3, 2, 1] + s[3, 3] + 3*s[4, 2] + 3*s[5, 1] + 3*s[6]
```

```
>>> from sage.all import *
>>> h.Eulerian([Integer(6)], Integer(3))
h[3, 2, 1] - h[4, 1, 1] + 2*h[4, 2] + h[5, 1]
>>> s.Eulerian([Integer(6)], Integer(3))
s[3, 2, 1] + s[3, 3] + 3*s[4, 2] + 3*s[5, 1] + 3*s[6]
```

carlitz_shareshian_wachs(*n*, *d*, *s*, *comparison=None*)

Return the Carlitz-Shareshian-Wachs symmetric function $X_{n,d,s}$ (if comparison is None), or $U_{n,d,s}$ (if comparison is -1), or $V_{n,d,s}$ (if comparison is 0), or $W_{n,d,s}$ (if comparison is 1) written in the basis self. These functions are defined below.

The Carlitz-Shareshian-Wachs symmetric functions have been introduced in [GriRei18], Exercise 2.9.11, as refinements of a certain particular case of chromatic quasisymmetric functions defined by Shareshian and Wachs. Their definitions are as follows:

Let n, d and s be three nonnegative integers. Let $W(n,d,s)$ denote the set of all n-tuples $(w_1, w_2, \ldots, w_n)$ of positive integers having the property that there exist precisely d elements i of $\{1, 2, \ldots, n-1\}$ satisfying $w_i > w_{i+1}$, and precisely s elements i of $\{1, 2, \ldots, n-1\}$ satisfying $w_i = w_{i+1}$. For every $w = (w_1, w_2, \ldots, w_n) \in W(n,d,s)$, let x_w be the monomial $x_{w_1} x_{w_2} \cdots x_{w_n}$. We then define the power series $X_{n,d,s}$ by

$$X_{n,d,s} = \sum_{w \in W(n,d,s)} x_w.$$

This is a symmetric function (according to [GriRei18], Exercise 2.9.11(b)), and for $s = 0$ equals the t^d-coefficient of the descent enumerator of Smirnov words of length n (an example of a chromatic quasisymmetric function which happens to be symmetric – see [ShaWach2014], Example 2.5).

Assume that $n > 0$. Then, we can define three further power series as follows:

$$U_{n,d,s} = \sum_{w_1 < w_n} x_w; \qquad V_{n,d,s} = \sum_{w_1 = w_n} x_w; \qquad W_{n,d,s} = \sum_{w_1 > w_n} x_w,$$

where all three sums range over $w = (w_1, w_2, \ldots, w_n) \in W(n,d,s)$. These three power series $U_{n,d,s}$, $V_{n,d,s}$ and $W_{n,d,s}$ are symmetric functions as well ([GriRei18], Exercise 2.9.11(c)). Their sum is $X_{n,d,s}$.

REFERENCES:

INPUT:

- `n` – nonnegative integer
- `d` – nonnegative integer
- `s` – nonnegative integer
- `comparison` – (default: `None`) a variable which can take the forms `None`, `-1`, `0` and `1`

OUTPUT:

The Carlitz-Shareshian-Wachs symmetric function $X_{n,d,s}$ (if `comparison` is `None`), or $U_{n,d,s}$ (if `comparison` is `-1`), or $V_{n,d,s}$ (if `comparison` is `0`), or $W_{n,d,s}$ (if `comparison` is `1`) written in the basis `self`.

EXAMPLES:

The power series $X_{n,d,s}$:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.m()
sage: m.carlitz_shareshian_wachs(3, 2, 1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 1)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 2, 0)
m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 2)
m[3]
sage: m.carlitz_shareshian_wachs(3, 1, 0)
4*m[1, 1, 1] + m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 1)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 0)
m[1, 1, 1]
```

(continues on next page)

(continued from previous page)

```
sage: m.carlitz_shareshian_wachs(5, 2, 2)
m[2, 2, 1] + m[3, 1, 1]
sage: m.carlitz_shareshian_wachs(1, 0, 0)
m[1]
sage: m.carlitz_shareshian_wachs(0, 0, 0)
m[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> m = Sym.m()
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(2), Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(1), Integer(1))
m[2, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(2), Integer(0))
m[1, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(2))
m[3]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(1), Integer(0))
4*m[1, 1, 1] + m[2, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(1))
m[2, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(0))
m[1, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(5), Integer(2), Integer(2))
m[2, 2, 1] + m[3, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(1), Integer(0), Integer(0))
m[1]
>>> m.carlitz_shareshian_wachs(Integer(0), Integer(0), Integer(0))
m[]
```

The power series $U_{n,d,s}$:

```
sage: m.carlitz_shareshian_wachs(3, 2, 1, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 1, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(3, 2, 0, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(3, 0, 2, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 0, comparison=-1)
2*m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 1, comparison=-1)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 0, comparison=-1)
m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(5, 2, 2, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(4, 2, 0, comparison=-1)
3*m[1, 1, 1, 1]
sage: m.carlitz_shareshian_wachs(1, 0, 0, comparison=-1)
```

(continues on next page)

(continued from previous page)

```
0
```

```
>>> from sage.all import *
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(2), Integer(1),␣
↪comparison=-Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(1), Integer(1),␣
↪comparison=-Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(2), Integer(0),␣
↪comparison=-Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(2),␣
↪comparison=-Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(1), Integer(0),␣
↪comparison=-Integer(1))
2*m[1, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(1),␣
↪comparison=-Integer(1))
m[2, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(0),␣
↪comparison=-Integer(1))
m[1, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(5), Integer(2), Integer(2),␣
↪comparison=-Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(4), Integer(2), Integer(0),␣
↪comparison=-Integer(1))
3*m[1, 1, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(1), Integer(0), Integer(0),␣
↪comparison=-Integer(1))
0
```

The power series $V_{n,d,s}$:

```
sage: m.carlitz_shareshian_wachs(3, 2, 1, comparison=0)
0
sage: m.carlitz_shareshian_wachs(3, 1, 1, comparison=0)
0
sage: m.carlitz_shareshian_wachs(3, 2, 0, comparison=0)
0
sage: m.carlitz_shareshian_wachs(3, 0, 2, comparison=0)
m[3]
sage: m.carlitz_shareshian_wachs(3, 1, 0, comparison=0)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 1, comparison=0)
0
sage: m.carlitz_shareshian_wachs(3, 0, 0, comparison=0)
0
sage: m.carlitz_shareshian_wachs(5, 2, 2, comparison=0)
0
```

(continues on next page)

(continued from previous page)

```
sage: m.carlitz_shareshian_wachs(4, 2, 0, comparison=0)
m[2, 1, 1]
sage: m.carlitz_shareshian_wachs(1, 0, 0, comparison=0)
m[1]
```

```
>>> from sage.all import *
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(2), Integer(1),␣
↪comparison=Integer(0))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(1), Integer(1),␣
↪comparison=Integer(0))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(2), Integer(0),␣
↪comparison=Integer(0))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(2),␣
↪comparison=Integer(0))
m[3]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(1), Integer(0),␣
↪comparison=Integer(0))
m[2, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(1),␣
↪comparison=Integer(0))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(0),␣
↪comparison=Integer(0))
0
>>> m.carlitz_shareshian_wachs(Integer(5), Integer(2), Integer(2),␣
↪comparison=Integer(0))
0
>>> m.carlitz_shareshian_wachs(Integer(4), Integer(2), Integer(0),␣
↪comparison=Integer(0))
m[2, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(1), Integer(0), Integer(0),␣
↪comparison=Integer(0))
m[1]
```

The power series $W_{n,d,s}$:

```
sage: m.carlitz_shareshian_wachs(3, 2, 1, comparison=1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 1, comparison=1)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 2, 0, comparison=1)
m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 2, comparison=1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 0, comparison=1)
2*m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 1, comparison=1)
0
sage: m.carlitz_shareshian_wachs(3, 0, 0, comparison=1)
```

(continues on next page)

(continued from previous page)

```
0
sage: m.carlitz_shareshian_wachs(5, 2, 2, comparison=1)
m[2, 2, 1] + m[3, 1, 1]
sage: m.carlitz_shareshian_wachs(4, 2, 0, comparison=1)
8*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2]
sage: m.carlitz_shareshian_wachs(1, 0, 0, comparison=1)
0
```

```
>>> from sage.all import *
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(2), Integer(1),␣
↪comparison=Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(1), Integer(1),␣
↪comparison=Integer(1))
m[2, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(2), Integer(0),␣
↪comparison=Integer(1))
m[1, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(2),␣
↪comparison=Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(1), Integer(0),␣
↪comparison=Integer(1))
2*m[1, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(1),␣
↪comparison=Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(3), Integer(0), Integer(0),␣
↪comparison=Integer(1))
0
>>> m.carlitz_shareshian_wachs(Integer(5), Integer(2), Integer(2),␣
↪comparison=Integer(1))
m[2, 2, 1] + m[3, 1, 1]
>>> m.carlitz_shareshian_wachs(Integer(4), Integer(2), Integer(0),␣
↪comparison=Integer(1))
8*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2]
>>> m.carlitz_shareshian_wachs(Integer(1), Integer(0), Integer(0),␣
↪comparison=Integer(1))
0
```

corresponding_basis_over(R)

Return the realization of symmetric functions corresponding to `self` but over the base ring `R`. Only works when `self` is one of the classical bases, not one of the q,t-dependent ones. In the latter case, `None` is returned instead.

INPUT:

- `R` – a commutative ring

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: m.corresponding_basis_over(ZZ)
```

(continues on next page)

(continued from previous page)

```
doctest:warning
...
DeprecationWarning: S.corresponding_basis_over(R) is deprecated.
Use S.change_ring(R) instead.
See https://github.com/sagemath/sage/issues/37220 for details.
Symmetric Functions over Integer Ring in the monomial basis

sage: Sym = SymmetricFunctions(CyclotomicField())
sage: s = Sym.schur()
sage: s.corresponding_basis_over(Integers(13))
Symmetric Functions over Ring of integers modulo 13 in the Schur basis

sage: P = ZZ['q','t']
sage: Sym = SymmetricFunctions(P)
sage: mj = Sym.macdonald().J()
sage: mj.corresponding_basis_over(Integers(13)['q','t'])
Symmetric Functions over Multivariate Polynomial Ring in q, t over
 Ring of integers modulo 13 in the Macdonald J basis
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.monomial()
>>> m.corresponding_basis_over(ZZ)
doctest:warning
...
DeprecationWarning: S.corresponding_basis_over(R) is deprecated.
Use S.change_ring(R) instead.
See https://github.com/sagemath/sage/issues/37220 for details.
Symmetric Functions over Integer Ring in the monomial basis

>>> Sym = SymmetricFunctions(CyclotomicField())
>>> s = Sym.schur()
>>> s.corresponding_basis_over(Integers(Integer(13)))
Symmetric Functions over Ring of integers modulo 13 in the Schur basis

>>> P = ZZ['q','t']
>>> Sym = SymmetricFunctions(P)
>>> mj = Sym.macdonald().J()
>>> mj.corresponding_basis_over(Integers(Integer(13))['q','t'])
Symmetric Functions over Multivariate Polynomial Ring in q, t over
 Ring of integers modulo 13 in the Macdonald J basis
```

Todo

This function is an ugly hack using strings. It should be rewritten as soon as the bases of `SymmetricFunctions` are put on a more robust and systematic footing.

degree_on_basis(*b*)

Return the degree of the basis element indexed by `b`.

INPUT:

- `self` – a basis of the symmetric functions
- `b` – a partition

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['q,t'].fraction_field())
sage: m = Sym.monomial()
sage: m.degree_on_basis(Partition([3,2]))
5
sage: P = Sym.macdonald().P()
sage: P.degree_on_basis(Partition([]))
0
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['q,t'].fraction_field())
>>> m = Sym.monomial()
>>> m.degree_on_basis(Partition([Integer(3),Integer(2)]))
5
>>> P = Sym.macdonald().P()
>>> P.degree_on_basis(Partition([]))
0
```

formal_series_ring()

Return the completion of all formal linear combinations of `self` with finite linear combinations in each homogeneous degree (computed lazily).

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: L = s.formal_series_ring()
sage: L
Lazy completion of Symmetric Functions over Integer Ring in the Schur
↪basis
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(ZZ).s()
>>> L = s.formal_series_ring()
>>> L
Lazy completion of Symmetric Functions over Integer Ring in the Schur
↪basis
```

fraction_field()

Return the fraction field of `self`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.fraction_field()
Fraction Field of Symmetric Functions over Rational Field in the Schur
↪basis
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s.fraction_field()
```

(continues on next page)

(continued from previous page)

```
Fraction Field of Symmetric Functions over Rational Field in the Schur␣
↪basis
```

gessel_reutenauer(*lam*)

Return the Gessel-Reutenauer symmetric function corresponding to the partition `lam` written in the basis `self`.

Let λ be a partition. The *Gessel-Reutenauer symmetric function* $\mathbf{GR}_\lambda$ corresponding to λ is the symmetric function denoted L_λ in [GR1993] and in Exercise 7.89 of [STA] and denoted $\mathbf{GR}_\lambda$ in Definition 6.6.34 of [GriRei18]. It is also called the *higher Lie character*, for instance in [Sch2003b]. It can be defined in several ways:

- It is the sum of the monomials $\mathbf{x}_w$ over all words w over the alphabet $\{1, 2, 3, \ldots\}$ which have CFL type λ. Here, the monomial $\mathbf{x}_w$ for a word $w = (w_1, w_2, \ldots, w_k)$ is defined as $x_{w_1} x_{w_2} \cdots x_{w_k}$, and the *CFL type* of a word w is defined as the partition obtained by sorting (in decreasing order) the lengths of the factors in the Lyndon factorization (`lyndon_factorization()`) of w. The fact that this power series $\mathbf{GR}_\lambda$ is symmetric is not obvious.
- It is the sum of the fundamental quasisymmetric functions $F_{\mathrm{Des}\,\sigma}$ over all permutations σ that have cycle type λ. See `sage.combinat.ncsf_qsym.qsym.QuasiSymmetricFunctions.Fundamental` for the definition of fundamental quasisymmetric functions, and `cycle_type()` for that of cycle type. For a permutation σ, we use $\mathrm{Des}\,\sigma$ to denote the descent composition (`descents_composition()`) of σ. Again, this definition does not make the symmetry of $\mathbf{GR}_\lambda$ obvious.
- For every positive integer n, we have

$$\mathbf{GR}_{(n)} = \frac{1}{n} \sum_{d|n} \mu(d) p_d^{n/d},$$

  where p_d denotes the d-th power-sum symmetric function. This $\mathbf{GR}_{(n)}$ is also denoted by L_n, and is called the Lie character. Now, the higher Lie character $\mathbf{GR}_\lambda$ is defined as the product:

$$h_{m_1}[L_1] \cdot h_{m_2}[L_2] \cdot h_{m_3}[L_3] \cdots ,$$

  where m_i denotes the multiplicity of the part i in λ, and where the square brackets stand for plethysm (`plethysm()`). This definition makes the symmetry (but not the integrality!) of $\mathbf{GR}_\lambda$ obvious.

The equivalences of these three definitions are proven in [GR1993] Sections 2-3. (See also [GriRei18] Subsection 6.6.2 for the equivalence of the first two definitions and further formulas.)

$\mathbf{GR}_\lambda$ has further significance in representations afforded by the tensor algebra $T(V)$ of a finite dimensional vector space. The Poincaré-Birkhoff-Witt theorem describes the universal enveloping algebra of a Lie algebra. It gives a decomposition of the degree-n component $T_n(V)$ of $T(V)$ into $GL(V)$ representations indexed by partitions. The higher Lie characters are the symmetric group S_n characters corresponding to this decomposition via Schur-Weyl duality.

Another important question, *Thrall's problem* (see e.g. [Sch2003b]) asks, for λ a partition of n, can we combinatorially interpret the coefficients α_μ^λ in the Schur-expansion of $\mathbf{GR}_\lambda$:

$$\mathbf{GR}_\lambda = \sum_{\mu \vdash n} \alpha_\mu^\lambda s_\mu.$$

INPUT:

- `lam` – a partition or a positive integer (in the latter case, it is understood to mean the partition `[lam]`)

OUTPUT:

The Gessel-Reutenauer symmetric function $\mathbf{GR}_\lambda$, where λ is `lam`, expanded in the basis `self`.

EXAMPLES:

The first few values of $\mathbf{GR}_{(n)} = L_n$:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: h = Sym.h()
sage: h.gessel_reutenauer(1)
h[1]
sage: h.gessel_reutenauer(2)
h[1, 1] - h[2]
sage: h.gessel_reutenauer(3)
h[2, 1] - h[3]
sage: h.gessel_reutenauer(4)
h[2, 1, 1] - h[2, 2]
sage: h.gessel_reutenauer(5)
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
sage: h.gessel_reutenauer(6)
h[2, 1, 1, 1, 1] - h[2, 2, 1, 1] - h[2, 2, 2]
 - 2*h[3, 1, 1, 1] + 5*h[3, 2, 1] - 2*h[3, 3] + h[4, 1, 1]
 - h[4, 2] - h[5, 1] + h[6]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> h = Sym.h()
>>> h.gessel_reutenauer(Integer(1))
h[1]
>>> h.gessel_reutenauer(Integer(2))
h[1, 1] - h[2]
>>> h.gessel_reutenauer(Integer(3))
h[2, 1] - h[3]
>>> h.gessel_reutenauer(Integer(4))
h[2, 1, 1] - h[2, 2]
>>> h.gessel_reutenauer(Integer(5))
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
>>> h.gessel_reutenauer(Integer(6))
h[2, 1, 1, 1, 1] - h[2, 2, 1, 1] - h[2, 2, 2]
 - 2*h[3, 1, 1, 1] + 5*h[3, 2, 1] - 2*h[3, 3] + h[4, 1, 1]
 - h[4, 2] - h[5, 1] + h[6]
```

Gessel-Reutenauer functions indexed by partitions:

```
sage: h.gessel_reutenauer([2, 1])
h[1, 1, 1] - h[2, 1]
sage: h.gessel_reutenauer([2, 2])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 2*h[2, 2] + h[3, 1] - h[4]
```

```
>>> from sage.all import *
>>> h.gessel_reutenauer([Integer(2), Integer(1)])
h[1, 1, 1] - h[2, 1]
>>> h.gessel_reutenauer([Integer(2), Integer(2)])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 2*h[2, 2] + h[3, 1] - h[4]
```

The Gessel-Reutenauer functions are Schur-positive:

```
sage: s = Sym.s()
sage: s.gessel_reutenauer([2, 1])
s[1, 1, 1] + s[2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: s.gessel_reutenauer([2, 2, 1])
s[1, 1, 1, 1, 1] + s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 2]
```

```
>>> from sage.all import *
>>> s = Sym.s()
>>> s.gessel_reutenauer([Integer(2), Integer(1)])
s[1, 1, 1] + s[2, 1]
>>> s.gessel_reutenauer([Integer(2), Integer(2), Integer(1)])
s[1, 1, 1, 1, 1] + s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 2]
```

They do not form a basis, as the following example (from [GR1993] p. 201) shows:

```
sage: s.gessel_reutenauer([4]) == s.gessel_reutenauer([2, 1, 1])
True
```

```
>>> from sage.all import *
>>> s.gessel_reutenauer([Integer(4)]) == s.gessel_reutenauer([Integer(2),␣
→Integer(1), Integer(1)])
True
```

They also go by the name *higher Lie character*:

```
sage: s.higher_lie_character([2, 2, 1]) == s.gessel_reutenauer([2, 2, 1])
True
```

```
>>> from sage.all import *
>>> s.higher_lie_character([Integer(2), Integer(2), Integer(1)]) == s.
→gessel_reutenauer([Integer(2), Integer(2), Integer(1)])
True
```

Of the above three equivalent definitions of $\mathbf{GR}_\lambda$, we use the third one for computations. Let us check that the second one gives the same results:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F() # fundamental basis
sage: def GR_def2(lam): # `\mathbf{GR}_\lambda`
....:     n = lam.size()
....:     r = F.sum_of_monomials([sigma.descents_composition()
....:                             for sigma in Permutations(n)
....:                             if sigma.cycle_type() == lam])
....:     return r.to_symmetric_function()
sage: all( GR_def2(lam) == h.gessel_reutenauer(lam)
....:      for n in range(5) for lam in Partitions(n) )
True
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> F = QSym.F() # fundamental basis
>>> def GR_def2(lam): # `\mathbf{GR}_\lambda`
...     n = lam.size()
...     r = F.sum_of_monomials([sigma.descents_composition()
```

(continues on next page)

(continued from previous page)

```
...                             for sigma in Permutations(n)
...                             if sigma.cycle_type() == lam])
...     return r.to_symmetric_function()
>>> all( GR_def2(lam) == h.gessel_reutenauer(lam)
...      for n in range(Integer(5)) for lam in Partitions(n) )
True
```

And the first one, too (assuming symmetry):

```
sage: m = Sym.m()
sage: def GR_def1(lam): # `\mathbf{GR}_\lambda`
....:     n = lam.size()
....:     Permus_mset = sage.combinat.permutation.Permutations_mset
....:     def coeff_of_m_mu_in_result(mu):
....:         words_to_check = Permus_mset([i for (i, l) in enumerate(mu)
....:                                       for _ in range(l)])
....:         return sum((1 for w in words_to_check if
....:                     Partition(list(reversed(sorted([len(v) for v in␣
↪Word(w).lyndon_factorization()]))))
....:                     == lam))
....:     r = m.sum_of_terms([(mu, coeff_of_m_mu_in_result(mu))
....:                         for mu in Partitions(n)],
....:                        distinct=True)
....:     return r
sage: all( GR_def1(lam) == h.gessel_reutenauer(lam)
....:      for n in range(5) for lam in Partitions(n) )
True
```

```
>>> from sage.all import *
>>> m = Sym.m()
>>> def GR_def1(lam): # `\mathbf{GR}_\lambda`
...     n = lam.size()
...     Permus_mset = sage.combinat.permutation.Permutations_mset
...     def coeff_of_m_mu_in_result(mu):
...         words_to_check = Permus_mset([i for (i, l) in enumerate(mu)
...                                       for _ in range(l)])
...         return sum((Integer(1) for w in words_to_check if
...                     Partition(list(reversed(sorted([len(v) for v in␣
↪Word(w).lyndon_factorization()]))))
...                     == lam))
...     r = m.sum_of_terms([(mu, coeff_of_m_mu_in_result(mu))
...                         for mu in Partitions(n)],
...                        distinct=True)
...     return r
>>> all( GR_def1(lam) == h.gessel_reutenauer(lam)
...      for n in range(Integer(5)) for lam in Partitions(n) )
True
```

Note

The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a **Q**-algebra from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over **Q** (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

higher_lie_character(*lam*)

Return the Gessel-Reutenauer symmetric function corresponding to the partition `lam` written in the basis `self`.

Let λ be a partition. The *Gessel-Reutenauer symmetric function* $\mathbf{GR}_\lambda$ corresponding to λ is the symmetric function denoted L_λ in [GR1993] and in Exercise 7.89 of [STA] and denoted $\mathbf{GR}_\lambda$ in Definition 6.6.34 of [GriRei18]. It is also called the *higher Lie character*, for instance in [Sch2003b]. It can be defined in several ways:

- It is the sum of the monomials $\mathbf{x}_w$ over all words w over the alphabet $\{1, 2, 3, \ldots\}$ which have CFL type λ. Here, the monomial $\mathbf{x}_w$ for a word $w = (w_1, w_2, \ldots, w_k)$ is defined as $x_{w_1} x_{w_2} \cdots x_{w_k}$, and the *CFL type* of a word w is defined as the partition obtained by sorting (in decreasing order) the lengths of the factors in the Lyndon factorization (`lyndon_factorization()`) of w. The fact that this power series $\mathbf{GR}_\lambda$ is symmetric is not obvious.
- It is the sum of the fundamental quasisymmetric functions $F_{\mathrm{Des}\,\sigma}$ over all permutations σ that have cycle type λ. See `sage.combinat.ncsf_qsym.qsym.QuasiSymmetricFunctions.Fundamental` for the definition of fundamental quasisymmetric functions, and `cycle_type()` for that of cycle type. For a permutation σ, we use $\mathrm{Des}\,\sigma$ to denote the descent composition (`descents_composition()`) of σ. Again, this definition does not make the symmetry of $\mathbf{GR}_\lambda$ obvious.
- For every positive integer n, we have

$$\mathbf{GR}_{(n)} = \frac{1}{n} \sum_{d|n} \mu(d) p_d^{n/d},$$

  where p_d denotes the d-th power-sum symmetric function. This $\mathbf{GR}_{(n)}$ is also denoted by L_n, and is called the Lie character. Now, the higher Lie character $\mathbf{GR}_\lambda$ is defined as the product:

$$h_{m_1}[L_1] \cdot h_{m_2}[L_2] \cdot h_{m_3}[L_3] \cdots ,$$

  where m_i denotes the multiplicity of the part i in λ, and where the square brackets stand for plethysm (`plethysm()`). This definition makes the symmetry (but not the integrality!) of $\mathbf{GR}_\lambda$ obvious.

The equivalences of these three definitions are proven in [GR1993] Sections 2-3. (See also [GriRei18] Subsection 6.6.2 for the equivalence of the first two definitions and further formulas.)

$\mathbf{GR}_\lambda$ has further significance in representations afforded by the tensor algebra $T(V)$ of a finite dimensional vector space. The Poincaré-Birkhoff-Witt theorem describes the universal enveloping algebra of a Lie algebra. It gives a decomposition of the degree-n component $T_n(V)$ of $T(V)$ into $GL(V)$ representations indexed by partitions. The higher Lie characters are the symmetric group S_n characters corresponding to this decomposition via Schur-Weyl duality.

Another important question, *Thrall's problem* (see e.g. [Sch2003b]) asks, for λ a partition of n, can we combinatorially interpret the coefficients α_μ^λ in the Schur-expansion of $\mathbf{GR}_\lambda$:

$$\mathbf{GR}_\lambda = \sum_{\mu \vdash n} \alpha_\mu^\lambda s_\mu.$$

INPUT:

- `lam` – a partition or a positive integer (in the latter case, it is understood to mean the partition `[lam]`)

OUTPUT:

The Gessel-Reutenauer symmetric function $\mathbf{GR}_\lambda$, where λ is `lam`, expanded in the basis `self`.

EXAMPLES:

The first few values of $\mathbf{GR}_{(n)} = L_n$:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: h = Sym.h()
sage: h.gessel_reutenauer(1)
h[1]
sage: h.gessel_reutenauer(2)
h[1, 1] - h[2]
sage: h.gessel_reutenauer(3)
h[2, 1] - h[3]
sage: h.gessel_reutenauer(4)
h[2, 1, 1] - h[2, 2]
sage: h.gessel_reutenauer(5)
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
sage: h.gessel_reutenauer(6)
h[2, 1, 1, 1, 1] - h[2, 2, 1, 1] - h[2, 2, 2]
 - 2*h[3, 1, 1, 1] + 5*h[3, 2, 1] - 2*h[3, 3] + h[4, 1, 1]
 - h[4, 2] - h[5, 1] + h[6]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> h = Sym.h()
>>> h.gessel_reutenauer(Integer(1))
h[1]
>>> h.gessel_reutenauer(Integer(2))
h[1, 1] - h[2]
>>> h.gessel_reutenauer(Integer(3))
h[2, 1] - h[3]
>>> h.gessel_reutenauer(Integer(4))
h[2, 1, 1] - h[2, 2]
>>> h.gessel_reutenauer(Integer(5))
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
>>> h.gessel_reutenauer(Integer(6))
h[2, 1, 1, 1, 1] - h[2, 2, 1, 1] - h[2, 2, 2]
 - 2*h[3, 1, 1, 1] + 5*h[3, 2, 1] - 2*h[3, 3] + h[4, 1, 1]
 - h[4, 2] - h[5, 1] + h[6]
```

Gessel-Reutenauer functions indexed by partitions:

```
sage: h.gessel_reutenauer([2, 1])
h[1, 1, 1] - h[2, 1]
sage: h.gessel_reutenauer([2, 2])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 2*h[2, 2] + h[3, 1] - h[4]
```

```
>>> from sage.all import *
>>> h.gessel_reutenauer([Integer(2), Integer(1)])
h[1, 1, 1] - h[2, 1]
>>> h.gessel_reutenauer([Integer(2), Integer(2)])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 2*h[2, 2] + h[3, 1] - h[4]
```

The Gessel-Reutenauer functions are Schur-positive:

```
sage: s = Sym.s()
sage: s.gessel_reutenauer([2, 1])
s[1, 1, 1] + s[2, 1]
sage: s.gessel_reutenauer([2, 2, 1])
s[1, 1, 1, 1, 1] + s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 2]
```

```
>>> from sage.all import *
>>> s = Sym.s()
>>> s.gessel_reutenauer([Integer(2), Integer(1)])
s[1, 1, 1] + s[2, 1]
>>> s.gessel_reutenauer([Integer(2), Integer(2), Integer(1)])
s[1, 1, 1, 1, 1] + s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 2]
```

They do not form a basis, as the following example (from [GR1993] p. 201) shows:

```
sage: s.gessel_reutenauer([4]) == s.gessel_reutenauer([2, 1, 1])
True
```

```
>>> from sage.all import *
>>> s.gessel_reutenauer([Integer(4)]) == s.gessel_reutenauer([Integer(2),␣
↪Integer(1), Integer(1)])
True
```

They also go by the name *higher Lie character*:

```
sage: s.higher_lie_character([2, 2, 1]) == s.gessel_reutenauer([2, 2, 1])
True
```

```
>>> from sage.all import *
>>> s.higher_lie_character([Integer(2), Integer(2), Integer(1)]) == s.
↪gessel_reutenauer([Integer(2), Integer(2), Integer(1)])
True
```

Of the above three equivalent definitions of $\mathbf{GR}_\lambda$, we use the third one for computations. Let us check that the second one gives the same results:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F() # fundamental basis
sage: def GR_def2(lam): # `\mathbf{GR}_\lambda`
....:     n = lam.size()
....:     r = F.sum_of_monomials([sigma.descents_composition()
....:                            for sigma in Permutations(n)
....:                            if sigma.cycle_type() == lam])
....:     return r.to_symmetric_function()
sage: all( GR_def2(lam) == h.gessel_reutenauer(lam)
....:      for n in range(5) for lam in Partitions(n) )
True
```

```
>>> from sage.all import *
>>> QSym = QuasiSymmetricFunctions(ZZ)
>>> F = QSym.F() # fundamental basis
```

(continues on next page)

(continued from previous page)

```
>>> def GR_def2(lam): # `\mathbf{GR}_\lambda`
...     n = lam.size()
...     r = F.sum_of_monomials([sigma.descents_composition()
...                             for sigma in Permutations(n)
...                             if sigma.cycle_type() == lam])
...     return r.to_symmetric_function()
>>> all( GR_def2(lam) == h.gessel_reutenauer(lam)
...      for n in range(Integer(5)) for lam in Partitions(n) )
True
```

And the first one, too (assuming symmetry):

```
sage: m = Sym.m()
sage: def GR_def1(lam): # `\mathbf{GR}_\lambda`
....:     n = lam.size()
....:     Permus_mset = sage.combinat.permutation.Permutations_mset
....:     def coeff_of_m_mu_in_result(mu):
....:         words_to_check = Permus_mset([i for (i, l) in enumerate(mu)
....:                                       for _ in range(l)])
....:         return sum((1 for w in words_to_check if
....:                     Partition(list(reversed(sorted([len(v) for v in␣
↪Word(w).lyndon_factorization()]))))
....:                     == lam))
....:     r = m.sum_of_terms([(mu, coeff_of_m_mu_in_result(mu))
....:                         for mu in Partitions(n)],
....:                        distinct=True)
....:     return r
sage: all( GR_def1(lam) == h.gessel_reutenauer(lam)
....:      for n in range(5) for lam in Partitions(n) )
True
```

```
>>> from sage.all import *
>>> m = Sym.m()
>>> def GR_def1(lam): # `\mathbf{GR}_\lambda`
...     n = lam.size()
...     Permus_mset = sage.combinat.permutation.Permutations_mset
...     def coeff_of_m_mu_in_result(mu):
...         words_to_check = Permus_mset([i for (i, l) in enumerate(mu)
...                                       for _ in range(l)])
...         return sum((Integer(1) for w in words_to_check if
...                     Partition(list(reversed(sorted([len(v) for v in␣
↪Word(w).lyndon_factorization()]))))
...                     == lam))
...     r = m.sum_of_terms([(mu, coeff_of_m_mu_in_result(mu))
...                         for mu in Partitions(n)],
...                        distinct=True)
...     return r
>>> all( GR_def1(lam) == h.gessel_reutenauer(lam)
...      for n in range(Integer(5)) for lam in Partitions(n) )
True
```

Note

The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a **Q**-algebra from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over **Q** (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

is_commutative()

Return whether this symmetric function algebra is commutative.

INPUT:

- `self` – a basis of the symmetric functions

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.is_commutative()
True
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s.is_commutative()
True
```

is_field(*proof=True*)

Return whether `self` is a field. (It is not.)

INPUT:

- `self` – a basis of the symmetric functions
- `proof` – an optional argument (default: value: `True`)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.is_field()
False
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s.is_field()
False
```

is_integral_domain(*proof=True*)

Return whether `self` is an integral domain. (It is if and only if the base ring is an integral domain.)

INPUT:

- `self` – a basis of the symmetric functions
- `proof` – an optional argument (default: value: `True`)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.is_integral_domain()
True
```

(continues on next page)

(continued from previous page)

```
sage: s = SymmetricFunctions(Zmod(14)).s()
sage: s.is_integral_domain()
False
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s.is_integral_domain()
True

>>> s = SymmetricFunctions(Zmod(Integer(14))).s()
>>> s.is_integral_domain()
False
```

lehrer_solomon(*lam*)

Return the Lehrer-Solomon symmetric function (also known as the Whitney homology character) corresponding to the partition `lam` written in the basis `self`.

Let $\lambda \vdash n$ be a partition. The *Lehrer-Solomon symmetric function* $\mathbf{LS}_\lambda$ corresponding to λ is the Frobenius characteristic of the representation denoted $\mathrm{Ind}_{Z_\lambda}^{S_n}(\xi_\lambda)$ in Theorem 4.5 of [LS1986] or W_λ in Theorem 2.7 of [HR2017]. It was first computed as a symmetric function in [Sun1994].

It is the symmetric group representation corresponding to a summand of the Whitney homology of the set partition lattice. The summand comes from the orbit of set partitions with block sizes corresponding to λ (after reordering appropriately).

It can be computed using Sundaram's plethystic formula (see [Sun1994] Theorem 1.8):

$$\mathbf{LS}_\lambda = \prod_{\text{odd } j \geq 1} h_{m_j}[\pi_j] \prod_{\text{even } j \geq 2} e_{m_j}[\pi_j],$$

where h_{m_j} are complete homogeneous symmetric functions, e_{m_j} are elementary symmetric functions, and π_j are the images of the Gessel-Reutenauer symmetric function $\mathbf{GR}_{(j)}$ (see `gessel_reutenauer()`) under the involution ω (i.e. `omega_involution()`):

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: pi_2 = (s.gessel_reutenauer(2)).omega_involution()
sage: pi_1 = (s.gessel_reutenauer(1)).omega_involution()
sage: s.lehrer_solomon([2,1]) == pi_2 * pi_1 # since h_1, e_1 are␣
↪plethystic identities
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> pi_2 = (s.gessel_reutenauer(Integer(2))).omega_involution()
>>> pi_1 = (s.gessel_reutenauer(Integer(1))).omega_involution()
>>> s.lehrer_solomon([Integer(2),Integer(1)]) == pi_2 * pi_1 # since h_1,␣
↪e_1 are plethystic identities
True
```

Note that this also gives the S_n-equivariant structure of the Orlik-Solomon algebra of the braid arrangement (also known as the type-A reflection arrangement).

The representation corresponding to $\mathbf{LS}_\lambda$ exhibits representation stability [Chu2012], and a sharp bound is given in [HR2017].

INPUT:

- `lam` – a partition or a positive integer (in the latter case, it is understood to mean the partition `[lam]`)

OUTPUT:

The Lehrer-Solomon symmetric function $\mathbf{LS}_\lambda$, where λ is `lam`, expanded in the basis `self`.

EXAMPLES:

The first few values of $\mathbf{LS}_{(n)}$:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: h = Sym.h()
sage: h.lehrer_solomon(1)
h[1]
sage: h.lehrer_solomon(2)
h[2]
sage: h.lehrer_solomon(3)
h[2, 1] - h[3]
sage: h.lehrer_solomon(4)
h[2, 1, 1] - h[2, 2]
sage: h.lehrer_solomon(5)
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> h = Sym.h()
>>> h.lehrer_solomon(Integer(1))
h[1]
>>> h.lehrer_solomon(Integer(2))
h[2]
>>> h.lehrer_solomon(Integer(3))
h[2, 1] - h[3]
>>> h.lehrer_solomon(Integer(4))
h[2, 1, 1] - h[2, 2]
>>> h.lehrer_solomon(Integer(5))
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
```

The *`whitney_homology_character()`* method is an alias:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: s = Sym.schur()
sage: s.lehrer_solomon([2, 2, 1]) == s.whitney_homology_character([2, 2,␣
↪1])
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> s = Sym.schur()
>>> s.lehrer_solomon([Integer(2), Integer(2), Integer(1)]) == s.whitney_
↪homology_character([Integer(2), Integer(2), Integer(1)])
True
```

Lehrer-Solomon functions indexed by partitions:

```
sage: h.lehrer_solomon([2, 1])
h[2, 1]
sage: h.lehrer_solomon([2, 2])
h[3, 1] - h[4]
```

```
>>> from sage.all import *
>>> h.lehrer_solomon([Integer(2), Integer(1)])
h[2, 1]
>>> h.lehrer_solomon([Integer(2), Integer(2)])
h[3, 1] - h[4]
```

The Lehrer-Solomon functions are Schur-positive:

```
sage: s = Sym.s()
sage: s.lehrer_solomon([2, 1])
s[2, 1] + s[3]
sage: s.lehrer_solomon([2, 2, 1])
s[3, 1, 1] + s[3, 2] + s[4, 1]
sage: s.lehrer_solomon([4, 1])
s[2, 1, 1, 1] + s[2, 2, 1] + 2*s[3, 1, 1] + s[3, 2] + s[4, 1]
```

```
>>> from sage.all import *
>>> s = Sym.s()
>>> s.lehrer_solomon([Integer(2), Integer(1)])
s[2, 1] + s[3]
>>> s.lehrer_solomon([Integer(2), Integer(2), Integer(1)])
s[3, 1, 1] + s[3, 2] + s[4, 1]
>>> s.lehrer_solomon([Integer(4), Integer(1)])
s[2, 1, 1, 1] + s[2, 2, 1] + 2*s[3, 1, 1] + s[3, 2] + s[4, 1]
```

one_basis()

Return the empty partition, as per `AlgebrasWithBasis.ParentMethods.one_basis`.

INPUT:

- `self` – a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: s = Sym.s()
sage: s.one_basis()
[]
sage: Q = Sym.hall_littlewood().Q()
sage: Q.one_basis()
[]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ['t'].fraction_field())
>>> s = Sym.s()
>>> s.one_basis()
[]
>>> Q = Sym.hall_littlewood().Q()
>>> Q.one_basis()
[]
```

Todo

generalize to Modules.Graded.Connected.ParentMethods

skew_schur(*x*)

Return the skew Schur function indexed by `x` in `self`.

INPUT:

- `x` – a skew partition

EXAMPLES:

```
sage: sp = SkewPartition([[5,3,3,1], [3,2,1]])
sage: s = SymmetricFunctions(QQ).s()
sage: s.skew_schur(sp)
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 3*s[3, 2, 1]
 + s[3, 3] + 2*s[4, 1, 1] + 2*s[4, 2] + s[5, 1]

sage: e = SymmetricFunctions(QQ).e()
sage: ess = e.skew_schur(sp); ess
e[2, 1, 1, 1, 1] - e[2, 2, 1, 1] - e[3, 1, 1, 1] + e[3, 2, 1]
sage: ess == e(s.skew_schur(sp))
True
```

```
>>> from sage.all import *
>>> sp = SkewPartition([[Integer(5),Integer(3),Integer(3),Integer(1)],␣
↪[Integer(3),Integer(2),Integer(1)]])
>>> s = SymmetricFunctions(QQ).s()
>>> s.skew_schur(sp)
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 3*s[3, 2, 1]
 + s[3, 3] + 2*s[4, 1, 1] + 2*s[4, 2] + s[5, 1]

>>> e = SymmetricFunctions(QQ).e()
>>> ess = e.skew_schur(sp); ess
e[2, 1, 1, 1, 1] - e[2, 2, 1, 1] - e[3, 1, 1, 1] + e[3, 2, 1]
>>> ess == e(s.skew_schur(sp))
True
```

whitney_homology_character(*lam*)

Return the Lehrer-Solomon symmetric function (also known as the Whitney homology character) corresponding to the partition `lam` written in the basis `self`.

Let $\lambda \vdash n$ be a partition. The *Lehrer-Solomon symmetric function* $\mathbf{LS}_\lambda$ corresponding to λ is the Frobenius characteristic of the representation denoted $\mathrm{Ind}_{Z_\lambda}^{S_n}(\xi_\lambda)$ in Theorem 4.5 of [LS1986] or W_λ in Theorem 2.7 of [HR2017]. It was first computed as a symmetric function in [Sun1994].

It is the symmetric group representation corresponding to a summand of the Whitney homology of the set partition lattice. The summand comes from the orbit of set partitions with block sizes corresponding to λ (after reordering appropriately).

It can be computed using Sundaram's plethystic formula (see [Sun1994] Theorem 1.8):

$$\mathbf{LS}_\lambda = \prod_{\text{odd } j\geq 1} h_{m_j}[\pi_j] \prod_{\text{even } j\geq 2} e_{m_j}[\pi_j],$$

where h_{m_j} are complete homogeneous symmetric functions, e_{m_j} are elementary symmetric functions, and π_j are the images of the Gessel-Reutenauer symmetric function $\mathbf{GR}_{(j)}$ (see gessel_reutenauer()) under the involution ω (i.e. omega_involution()):

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: pi_2 = (s.gessel_reutenauer(2)).omega_involution()
sage: pi_1 = (s.gessel_reutenauer(1)).omega_involution()
sage: s.lehrer_solomon([2,1]) == pi_2 * pi_1 # since h_1, e_1 are␣
↪plethystic identities
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> pi_2 = (s.gessel_reutenauer(Integer(2))).omega_involution()
>>> pi_1 = (s.gessel_reutenauer(Integer(1))).omega_involution()
>>> s.lehrer_solomon([Integer(2),Integer(1)]) == pi_2 * pi_1 # since h_1,␣
↪e_1 are plethystic identities
True
```

Note that this also gives the S_n-equivariant structure of the Orlik-Solomon algebra of the braid arrangement (also known as the type-A reflection arrangement).

The representation corresponding to $\mathbf{LS}_\lambda$ exhibits representation stability [Chu2012], and a sharp bound is given in [HR2017].

INPUT:

- lam – a partition or a positive integer (in the latter case, it is understood to mean the partition [lam])

OUTPUT:

The Lehrer-Solomon symmetric function $\mathbf{LS}_\lambda$, where λ is lam, expanded in the basis self.

EXAMPLES:

The first few values of $\mathbf{LS}_{(n)}$:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: h = Sym.h()
sage: h.lehrer_solomon(1)
h[1]
sage: h.lehrer_solomon(2)
h[2]
sage: h.lehrer_solomon(3)
h[2, 1] - h[3]
sage: h.lehrer_solomon(4)
h[2, 1, 1] - h[2, 2]
sage: h.lehrer_solomon(5)
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> h = Sym.h()
>>> h.lehrer_solomon(Integer(1))
h[1]
>>> h.lehrer_solomon(Integer(2))
```

(continues on next page)

(continued from previous page)

```
h[2]
>>> h.lehrer_solomon(Integer(3))
h[2, 1] - h[3]
>>> h.lehrer_solomon(Integer(4))
h[2, 1, 1] - h[2, 2]
>>> h.lehrer_solomon(Integer(5))
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
```

The `whitney_homology_character()` method is an alias:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: s = Sym.schur()
sage: s.lehrer_solomon([2, 2, 1]) == s.whitney_homology_character([2, 2,␣
↪1])
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> s = Sym.schur()
>>> s.lehrer_solomon([Integer(2), Integer(2), Integer(1)]) == s.whitney_
↪homology_character([Integer(2), Integer(2), Integer(1)])
True
```

Lehrer-Solomon functions indexed by partitions:

```
sage: h.lehrer_solomon([2, 1])
h[2, 1]
sage: h.lehrer_solomon([2, 2])
h[3, 1] - h[4]
```

```
>>> from sage.all import *
>>> h.lehrer_solomon([Integer(2), Integer(1)])
h[2, 1]
>>> h.lehrer_solomon([Integer(2), Integer(2)])
h[3, 1] - h[4]
```

The Lehrer-Solomon functions are Schur-positive:

```
sage: s = Sym.s()
sage: s.lehrer_solomon([2, 1])
s[2, 1] + s[3]
sage: s.lehrer_solomon([2, 2, 1])
s[3, 1, 1] + s[3, 2] + s[4, 1]
sage: s.lehrer_solomon([4, 1])
s[2, 1, 1, 1] + s[2, 2, 1] + 2*s[3, 1, 1] + s[3, 2] + s[4, 1]
```

```
>>> from sage.all import *
>>> s = Sym.s()
>>> s.lehrer_solomon([Integer(2), Integer(1)])
s[2, 1] + s[3]
>>> s.lehrer_solomon([Integer(2), Integer(2), Integer(1)])
s[3, 1, 1] + s[3, 2] + s[4, 1]
```

(continues on next page)

(continued from previous page)

```
>>> s.lehrer_solomon([Integer(4), Integer(1)])
s[2, 1, 1, 1] + s[2, 2, 1] + 2*s[3, 1, 1] + s[3, 2] + s[4, 1]
```

super_categories()

The super categories of `self`.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import SymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = SymmetricFunctionsBases(Sym)
sage: bases.super_categories()
[Category of realizations of Symmetric Functions over Rational Field,
 Category of commutative Hopf algebras with basis over Rational Field,
 Join of Category of realizations of Hopf algebras over Rational Field
     and Category of graded algebras over Rational Field
     and Category of graded coalgebras over Rational Field,
 Category of unique factorization domains]

sage: Sym = SymmetricFunctions(ZZ["x"])
sage: bases = SymmetricFunctionsBases(Sym)
sage: bases.super_categories()
[Category of realizations of Symmetric Functions over Univariate Polynomial␣
↪Ring in x over Integer Ring,
 Category of commutative Hopf algebras with basis over Univariate Polynomial␣
↪Ring in x over Integer Ring,
 Join of Category of realizations of Hopf algebras over Univariate Polynomial␣
↪Ring in x over Integer Ring
     and Category of graded algebras over Univariate Polynomial Ring in x␣
↪over Integer Ring
     and Category of graded coalgebras over Univariate Polynomial Ring in x␣
↪over Integer Ring]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import SymmetricFunctionsBases
>>> Sym = SymmetricFunctions(QQ)
>>> bases = SymmetricFunctionsBases(Sym)
>>> bases.super_categories()
[Category of realizations of Symmetric Functions over Rational Field,
 Category of commutative Hopf algebras with basis over Rational Field,
 Join of Category of realizations of Hopf algebras over Rational Field
     and Category of graded algebras over Rational Field
     and Category of graded coalgebras over Rational Field,
 Category of unique factorization domains]

>>> Sym = SymmetricFunctions(ZZ["x"])
>>> bases = SymmetricFunctionsBases(Sym)
>>> bases.super_categories()
[Category of realizations of Symmetric Functions over Univariate Polynomial␣
↪Ring in x over Integer Ring,
 Category of commutative Hopf algebras with basis over Univariate Polynomial␣
↪Ring in x over Integer Ring,
```

(continues on next page)

(continued from previous page)

```
Join of Category of realizations of Hopf algebras over Univariate Polynomial␣
↪Ring in x over Integer Ring
    and Category of graded algebras over Univariate Polynomial Ring in x␣
↪over Integer Ring
    and Category of graded coalgebras over Univariate Polynomial Ring in x␣
↪over Integer Ring]
```

class sage.combinat.sf.sfa.**SymmetricFunctionsFamilyFunctor**(*basis*, *family*, *name*, **args*)

Bases: *SymmetricFunctionsFunctor*

Initialize the functor.

INPUT:

- basis – the basis of the symmetric function algebra

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import SymmetricFunctionsFamilyFunctor
sage: R.<t> = ZZ[]
sage: basis = SymmetricFunctions(R).macdonald(q=1).H()
sage: family = sage.combinat.sf.macdonald.Macdonald
sage: name = basis.basis_name()
sage: SymmetricFunctionsFamilyFunctor(basis, family, name, 1, t)
SymmetricFunctionsFunctor[Macdonald H with q=1]
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import SymmetricFunctionsFamilyFunctor
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> basis = SymmetricFunctions(R).macdonald(q=Integer(1)).H()
>>> family = sage.combinat.sf.macdonald.Macdonald
>>> name = basis.basis_name()
>>> SymmetricFunctionsFamilyFunctor(basis, family, name, Integer(1), t)
SymmetricFunctionsFunctor[Macdonald H with q=1]
```

class sage.combinat.sf.sfa.**SymmetricFunctionsFunctor**(*basis*, *name*, **args*)

Bases: ConstructionFunctor

A constructor for algebras of symmetric functions.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.construction()
(SymmetricFunctionsFunctor[Schur], Rational Field)
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).s()
>>> s.construction()
(SymmetricFunctionsFunctor[Schur], Rational Field)
```

rank = 9

sage.combinat.sf.sfa.**is_SymmetricFunction**(*x*)

Check whether x is a symmetric function.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import is_SymmetricFunction
sage: s = SymmetricFunctions(QQ).s()
sage: is_SymmetricFunction(2)
doctest:warning...
DeprecationWarning: The function is_SymmetricFunction is deprecated;
use 'isinstance(..., SymmetricFunctionAlgebra_generic.Element)' instead.
See https://github.com/sagemath/sage/issues/38279 for details.
False
sage: is_SymmetricFunction(s(2))
True
sage: is_SymmetricFunction(s([2,1]))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import is_SymmetricFunction
>>> s = SymmetricFunctions(QQ).s()
>>> is_SymmetricFunction(Integer(2))
doctest:warning...
DeprecationWarning: The function is_SymmetricFunction is deprecated;
use 'isinstance(..., SymmetricFunctionAlgebra_generic.Element)' instead.
See https://github.com/sagemath/sage/issues/38279 for details.
False
>>> is_SymmetricFunction(s(Integer(2)))
True
>>> is_SymmetricFunction(s([Integer(2),Integer(1)]))
True
```

sage.combinat.sf.sfa.**is_SymmetricFunctionAlgebra**(*x*)

Check whether x is a symmetric function algebra.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import is_SymmetricFunctionAlgebra
sage: is_SymmetricFunctionAlgebra(5)
doctest:warning...
DeprecationWarning: the function is_SymmetricFunctionAlgebra is deprecated;
use 'isinstance(..., SymmetricFunctionAlgebra_generic)' instead
See https://github.com/sagemath/sage/issues/37896 for details.
False
sage: is_SymmetricFunctionAlgebra(ZZ)
False
sage: is_SymmetricFunctionAlgebra(SymmetricFunctions(ZZ).schur())
True
sage: is_SymmetricFunctionAlgebra(SymmetricFunctions(QQ).e())
True
sage: is_SymmetricFunctionAlgebra(SymmetricFunctions(QQ).macdonald(q=1,t=1).P())
True
sage: is_SymmetricFunctionAlgebra(SymmetricFunctions(FractionField(QQ['q','t'])).
↪macdonald().P())
True
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import is_SymmetricFunctionAlgebra
>>> is_SymmetricFunctionAlgebra(Integer(5))
doctest:warning...
DeprecationWarning: the function is_SymmetricFunctionAlgebra is deprecated;
use 'isinstance(..., SymmetricFunctionAlgebra_generic)' instead
See https://github.com/sagemath/sage/issues/37896 for details.
False
>>> is_SymmetricFunctionAlgebra(ZZ)
False
>>> is_SymmetricFunctionAlgebra(SymmetricFunctions(ZZ).schur())
True
>>> is_SymmetricFunctionAlgebra(SymmetricFunctions(QQ).e())
True
>>> is_SymmetricFunctionAlgebra(SymmetricFunctions(QQ).macdonald(q=Integer(1),
↪t=Integer(1)).P())
True
>>> is_SymmetricFunctionAlgebra(SymmetricFunctions(FractionField(QQ['q','t'])).
↪macdonald().P())
True
```

sage.combinat.sf.sfa.**zee**(*part*)

Return the size of the centralizer of any permutation of cycle type `part`.

Note that the size of the centralizer is the inner product between `p(part)` and itself, where p is the power-sum symmetric functions.

INPUT:

- `part` – integer partition (for example, `[2,1,1]`)

OUTPUT:

- the integer $\prod_i i^{m_i(part)} m_i(part)!$ where $m_i(part)$ is the number of parts in the partition `part` equal to i

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import zee
sage: zee([2,1,1])
4
```

```
>>> from sage.all import *
>>> from sage.combinat.sf.sfa import zee
>>> zee([Integer(2),Integer(1),Integer(1)])
4
```

5.1.304 Witt symmetric functions

class sage.combinat.sf.witt.**SymmetricFunctionAlgebra_witt**(*Sym*)

Bases: *SymmetricFunctionAlgebra_multiplicative*

The Witt symmetric function basis (or Witt basis, to be short).

The Witt basis of the ring of symmetric functions is denoted by (x_λ) in [HazWitt1], section 9.63, and by (q_λ) in [DoranIV1996]. We will denote this basis by (w_λ) (which is precisely how it is denoted in [GriRei18], Exercise 2.9.3(d)). It is a multiplicative basis (meaning that $w_\emptyset = 1$ and that every partition λ satisfies $w_\lambda = w_{\lambda_1} w_{\lambda_2} w_{\lambda_3} \cdots$, where w_i means $w_{(i)}$ for every nonnegative integer i).

This basis can be defined in various ways. Probably the most well-known one is using the equation

$$\prod_{d=1}^{\infty} (1 - w_d t^d)^{-1} = \sum_{n=0}^{\infty} h_n t^n$$

where t is a formal variable and h_n are the complete homogeneous symmetric functions, extended to 0 by $h_0 = 1$. This equation allows one to uniquely determine the functions $w_1, w_2, w_3, \ldots$ by recursion; one consequently extends the definition to all w_λ by requiring multiplicativity.

A way to rewrite the above equation without power series is:

$$h_n = \sum_{\lambda \vdash n} w_\lambda$$

for all nonnegative integers n, where $\lambda \vdash n$ means that λ is a partition of n.

A similar equation (which is easily seen to be equivalent to the former) is

$$e_n = \sum_{\lambda} (-1)^{n - \ell(\lambda)} w_\lambda,$$

with the sum running only over *strict* partitions λ of n this time. This equation can also be used to recursively define the w_n. Furthermore, every positive integer n satisfies

$$p_n = \sum_{d \mid n} d w_d^{n/d},$$

and this can be used to define the w_n recursively over any ring which is torsion-free as a $\mathbf{Z}$-module. While these equations all yield easy formulas for classical bases of the ring of symmetric functions in terms of the Witt symmetric functions, it seems difficult to obtain explicit formulas in the other direction.

The Witt symmetric functions owe their name to the fact that the ring of symmetric functions can be viewed as the coordinate ring of the group scheme of Witt vectors, and the Witt symmetric functions are the functions that send a Witt vector to its components (whereas the powersum symmetric functions send a Witt vector to its ghost components). Details can be found in [HazWitt1] or section 3.2 of [BorWi2004].

INPUT:

- `Sym` – an instance of the ring of the symmetric functions

REFERENCES:

EXAMPLES:

Here are the first few Witt symmetric functions, in various bases:

```
sage: Sym = SymmetricFunctions(QQ)
sage: w = Sym.w()
sage: e = Sym.e()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()

sage: p(w([1]))
p[1]
```

(continues on next page)

(continued from previous page)

```
sage: m(w([1]))
m[1]
sage: e(w([1]))
e[1]
sage: h(w([1]))
h[1]
sage: s(w([1]))
s[1]

sage: p(w([2]))
-1/2*p[1, 1] + 1/2*p[2]
sage: m(w([2]))
-m[1, 1]
sage: e(w([2]))
-e[2]
sage: h(w([2]))
-h[1, 1] + h[2]
sage: s(w([2]))
-s[1, 1]

sage: p(w([3]))
-1/3*p[1, 1, 1] + 1/3*p[3]
sage: m(w([3]))
-2*m[1, 1, 1] - m[2, 1]
sage: e(w([3]))
-e[2, 1] + e[3]
sage: h(w([3]))
-h[2, 1] + h[3]
sage: s(w([3]))
-s[2, 1]

sage: Sym = SymmetricFunctions(ZZ)
sage: w = Sym.w()
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: p = Sym.p()
sage: m(w([4]))
-9*m[1, 1, 1, 1] - 4*m[2, 1, 1] - 2*m[2, 2] - m[3, 1]
sage: e(w([4]))
-e[2, 1, 1] + e[3, 1] - e[4]
sage: h(w([4]))
-h[1, 1, 1, 1] + 2*h[2, 1, 1] - h[2, 2] - h[3, 1] + h[4]
sage: s(w([4]))
-s[1, 1, 1, 1] - s[2, 1, 1] - s[2, 2] - s[3, 1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> w = Sym.w()
>>> e = Sym.e()
>>> h = Sym.h()
```

(continues on next page)

(continued from previous page)

```
>>> p = Sym.p()
>>> s = Sym.s()
>>> m = Sym.m()

>>> p(w([Integer(1)]))
p[1]
>>> m(w([Integer(1)]))
m[1]
>>> e(w([Integer(1)]))
e[1]
>>> h(w([Integer(1)]))
h[1]
>>> s(w([Integer(1)]))
s[1]

>>> p(w([Integer(2)]))
-1/2*p[1, 1] + 1/2*p[2]
>>> m(w([Integer(2)]))
-m[1, 1]
>>> e(w([Integer(2)]))
-e[2]
>>> h(w([Integer(2)]))
-h[1, 1] + h[2]
>>> s(w([Integer(2)]))
-s[1, 1]

>>> p(w([Integer(3)]))
-1/3*p[1, 1, 1] + 1/3*p[3]
>>> m(w([Integer(3)]))
-2*m[1, 1, 1] - m[2, 1]
>>> e(w([Integer(3)]))
-e[2, 1] + e[3]
>>> h(w([Integer(3)]))
-h[2, 1] + h[3]
>>> s(w([Integer(3)]))
-s[2, 1]

>>> Sym = SymmetricFunctions(ZZ)
>>> w = Sym.w()
>>> e = Sym.e()
>>> h = Sym.h()
>>> s = Sym.s()
>>> m = Sym.m()
>>> p = Sym.p()
>>> m(w([Integer(4)]))
-9*m[1, 1, 1, 1] - 4*m[2, 1, 1] - 2*m[2, 2] - m[3, 1]
>>> e(w([Integer(4)]))
-e[2, 1, 1] + e[3, 1] - e[4]
>>> h(w([Integer(4)]))
-h[1, 1, 1, 1] + 2*h[2, 1, 1] - h[2, 2] - h[3, 1] + h[4]
>>> s(w([Integer(4)]))
-s[1, 1, 1, 1] - s[2, 1, 1] - s[2, 2] - s[3, 1]
```

Some examples of conversions the other way:

```
sage: w(h[3])
w[1, 1, 1] + w[2, 1] + w[3]
sage: w(e[3])
-w[2, 1] + w[3]
sage: w(m[2,1])
2*w[2, 1] - 3*w[3]
sage: w(p[3])
w[1, 1, 1] + 3*w[3]
```

```
>>> from sage.all import *
>>> w(h[Integer(3)])
w[1, 1, 1] + w[2, 1] + w[3]
>>> w(e[Integer(3)])
-w[2, 1] + w[3]
>>> w(m[Integer(2),Integer(1)])
2*w[2, 1] - 3*w[3]
>>> w(p[Integer(3)])
w[1, 1, 1] + 3*w[3]
```

Antipodes:

```
sage: w([1]).antipode()
-w[1]
sage: w([2]).antipode()
-w[1, 1] - w[2]
```

```
>>> from sage.all import *
>>> w([Integer(1)]).antipode()
-w[1]
>>> w([Integer(2)]).antipode()
-w[1, 1] - w[2]
```

The following holds for all odd i and is easily proven by induction:

```
sage: all(w([i]).antipode() == -w([i]) for i in range(1, 10, 2))
True
```

```
>>> from sage.all import *
>>> all(w([i]).antipode() == -w([i]) for i in range(Integer(1), Integer(10),
↪Integer(2)))
True
```

The Witt basis does not allow for simple expressions for comultiplication and antipode in general (this is related to the fact that the sum of two Witt vectors isn't easily described in terms of the components). Therefore, a number of computations with Witt symmetric functions pass through the complete homogeneous symmetric functions by default.

class Element

Bases: *Element*

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: w = Sym.w()
sage: a = w([4,3,1,1]); a
w[4, 3, 1, 1]
sage: a.omega()
-w[3, 1, 1, 1, 1, 1, 1] - w[3, 2, 1, 1, 1, 1]
 - w[3, 2, 2, 1, 1] - w[4, 3, 1, 1]

sage: h = Sym.h()
sage: all(w(h(w[la]).omega()) == w[la].omega()
....:     for n in range(6) for la in Partitions(n))
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> w = Sym.w()
>>> a = w([Integer(4),Integer(3),Integer(1),Integer(1)]); a
w[4, 3, 1, 1]
>>> a.omega()
-w[3, 1, 1, 1, 1, 1, 1] - w[3, 2, 1, 1, 1, 1]
 - w[3, 2, 2, 1, 1] - w[4, 3, 1, 1]

>>> h = Sym.h()
>>> all(w(h(w[la]).omega()) == w[la].omega()
...     for n in range(Integer(6)) for la in Partitions(n))
True
```

coproduct(*elt*)

Return the coproduct of the element `elt`.

INPUT:

- `elt` – a symmetric function written in this basis

OUTPUT:

The coproduct acting on `elt`; the result is an element of the tensor squared of the basis `self`.

EXAMPLES:

```
sage: w = SymmetricFunctions(QQ).w()
sage: w[2].coproduct()
w[] # w[2] - w[1] # w[1] + w[2] # w[]
sage: w.coproduct(w[2])
w[] # w[2] - w[1] # w[1] + w[2] # w[]
sage: w[2,1].coproduct()
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] +
→w[2, 1] # w[]
sage: w.coproduct(w[2,1])
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] +
→w[2, 1] # w[]
```

```
>>> from sage.all import *
>>> w = SymmetricFunctions(QQ).w()
>>> w[Integer(2)].coproduct()
w[] # w[2] - w[1] # w[1] + w[2] # w[]
>>> w.coproduct(w[Integer(2)])
w[] # w[2] - w[1] # w[1] + w[2] # w[]
>>> w[Integer(2),Integer(1)].coproduct()
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] +
→w[2, 1] # w[]
>>> w.coproduct(w[Integer(2),Integer(1)])
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] +
→w[2, 1] # w[]
```

verschiebung(*n*)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n}e_{r/n}, \quad \mathbf{V}_n(w_r) = w_{r/n},$$

(where h is the complete homogeneous basis, p is the powersum basis, e is the elementary basis, and w is the Witt basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = \mathbf{V}_n(w_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for “shift”) endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Frobenius operator (see `frobenius()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley’s [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, \ldots, \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, \ldots, \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: w = Sym.w()
sage: w[3].verschiebung(2)
0
sage: w[4].verschiebung(4)
w[1]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(ZZ)
>>> w = Sym.w()
>>> w[Integer(3)].verschiebung(Integer(2))
0
>>> w[Integer(4)].verschiebung(Integer(4))
w[1]
```

5.1.305 Shard intersection order

This file builds a combinatorial version of the shard intersection order of type A (in the classification of finite Coxeter groups). This is a lattice on the set of permutations, closely related to noncrossing partitions and the weak order.

For technical reasons, the elements of the posets are not permutations, but can be easily converted from and to permutations:

```
sage: from sage.combinat.shard_order import ShardPosetElement
sage: p0 = Permutation([1,3,4,2])
sage: e0 = ShardPosetElement(p0); e0
(1, 3, 4, 2)
sage: Permutation(list(e0)) == p0
True
```

```
>>> from sage.all import *
>>> from sage.combinat.shard_order import ShardPosetElement
>>> p0 = Permutation([Integer(1),Integer(3),Integer(4),Integer(2)])
>>> e0 = ShardPosetElement(p0); e0
(1, 3, 4, 2)
>>> Permutation(list(e0)) == p0
True
```

See also

A general implementation for all finite Coxeter groups is available as `shard_poset()`

REFERENCES:

`class sage.combinat.shard_order.ShardPosetElement(p)`

Bases: `tuple`

An element of the shard poset.

This is basically a permutation with extra stored arguments:

- `p` – the permutation itself as a tuple
- `runs` – the decreasing runs as a tuple of tuples
- `run_indices` – list; `integer -> index of the run`
- `dpg` – the transitive closure of the shard preorder graph
- `spg` – the transitive reduction of the shard preorder graph

These elements can easily be converted from and to permutations:

```
sage: from sage.combinat.shard_order import ShardPosetElement
sage: p0 = Permutation([1,3,4,2])
sage: e0 = ShardPosetElement(p0); e0
(1, 3, 4, 2)
sage: Permutation(list(e0)) == p0
True
```

```
>>> from sage.all import *
>>> from sage.combinat.shard_order import ShardPosetElement
>>> p0 = Permutation([Integer(1),Integer(3),Integer(4),Integer(2)])
>>> e0 = ShardPosetElement(p0); e0
(1, 3, 4, 2)
>>> Permutation(list(e0)) == p0
True
```

`sage.combinat.shard_order.shard_poset(n)`

Return the shard intersection order on permutations of size n.

This is defined on the set of permutations. To every permutation, one can attach a pre-order, using the descending runs and their relative positions.

The shard intersection order is given by the implication (or refinement) order on the set of pre-orders defined from all permutations.

This can also be seen in a geometrical way. Every pre-order defines a cone in a vector space of dimension n. The shard poset is given by the inclusion of these cones.

See also

shard_preorder_graph()

EXAMPLES:

```
sage: P = posets.ShardPoset(4); P  # indirect doctest
Finite poset containing 24 elements
sage: P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
sage: P.characteristic_polynomial()
q^3 - 11*q^2 + 23*q - 13
sage: P.zeta_polynomial()
17/3*q^3 - 6*q^2 + 4/3*q
sage: P.is_self_dual()
False
```

```
>>> from sage.all import *
>>> P = posets.ShardPoset(Integer(4)); P  # indirect doctest
Finite poset containing 24 elements
>>> P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
>>> P.characteristic_polynomial()
q^3 - 11*q^2 + 23*q - 13
>>> P.zeta_polynomial()
17/3*q^3 - 6*q^2 + 4/3*q
>>> P.is_self_dual()
False
```

sage.combinat.shard_order.**shard_preorder_graph**(*runs*)

Return the preorder attached to a tuple of decreasing runs.

This is a directed graph, whose vertices correspond to the runs.

There is an edge from a run R to a run S if R is before S in the list of runs and the two intervals defined by the initial and final indices of R and S overlap.

This only depends on the initial and final indices of the runs. For this reason, this input can also be given in that shorten way.

INPUT:

- `runs` – either
  - a tuple of tuples, the runs of a permutation, or
  - a tuple of pairs (i, j), each one standing for a run from i to j

OUTPUT: a directed graph, with vertices labelled by integers

EXAMPLES:

```
sage: from sage.combinat.shard_order import shard_preorder_graph
sage: s = Permutation([2,8,3,9,6,4,5,1,7])
sage: def cut(lr):
....:     return tuple((r[0], r[-1]) for r in lr)
sage: shard_preorder_graph(cut(s.decreasing_runs()))
Digraph on 5 vertices
sage: s = Permutation([9,4,3,2,8,6,5,1,7])
sage: P = shard_preorder_graph(s.decreasing_runs())
sage: P.is_isomorphic(digraphs.TransitiveTournament(3))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.shard_order import shard_preorder_graph
>>> s = Permutation([Integer(2),Integer(8),Integer(3),Integer(9),Integer(6),
↪Integer(4),Integer(5),Integer(1),Integer(7)])
>>> def cut(lr):
...     return tuple((r[Integer(0)], r[-Integer(1)]) for r in lr)
>>> shard_preorder_graph(cut(s.decreasing_runs()))
Digraph on 5 vertices
>>> s = Permutation([Integer(9),Integer(4),Integer(3),Integer(2),Integer(8),
↪Integer(6),Integer(5),Integer(1),Integer(7)])
>>> P = shard_preorder_graph(s.decreasing_runs())
>>> P.is_isomorphic(digraphs.TransitiveTournament(Integer(3)))
True
```

5.1.306 Shifted primed tableaux

AUTHORS:

- Kirill Paramonov (2017-08-18): initial implementation
- Chaman Agrawal (2019-08-12): add parameter to allow primed diagonal entry

class sage.combinat.shifted_primed_tableau.**CrystalElementShiftedPrimedTableau**(*parent*, *T*, *skew=None*, *check=True*, *preprocessed=False*)

Bases: *ShiftedPrimedTableau*

Class for elements of `crystals.ShiftedPrimedTableau`.

e(*ind*)

Compute the action of the crystal operator e_i on a shifted primed tableau using cases from the papers [HPS2017] and [AO2018].

INPUT:

- `ind` – an element in the index set of the crystal

OUTPUT: primed tableau or `None`

EXAMPLES:

```
sage: SPT = ShiftedPrimedTableaux([5,4,2])
sage: t = SPT([[1,1,1,'2p','3p'], [2,'3p',3,3],[3,4]])
sage: t.pp()
1  1  1  2' 3'
   2  3' 3  3
      3  4
sage: s = t.e(2)
sage: s.pp()
1  1  1  2' 3'
   2  2  3  3
      3  4
sage: t == s.f(2)
True
```

(continues on next page)

(continued from previous page)

```
sage: SPT = ShiftedPrimedTableaux([2,1])
sage: t = SPT([[2,'3p'],[3]])
sage: t.e(-1).pp()
1  3'
   3
sage: t.e(1).pp()
1  3'
   3
sage: t.e(2).pp()
2  2
   3

sage: r = SPT([[2, 2],[3]])
sage: r.e(-1).pp()
1  2
   3
sage: r.e(1).pp()
1  2
   3
sage: r.e(2) is None
True

sage: r = SPT([[1,'3p'],[3]])
sage: r.e(-1) is None
True
sage: r.e(1) is None
True
sage: r.e(2).pp()
1  2'
   3
sage: r = SPT([[1,'2p'],[3]])
sage: r.e(-1).pp()
1  1
   3
sage: r.e(1) is None
True
sage: r.e(2).pp()
1  2'
   2
sage: t = SPT([[2,'3p'],[3]])
sage: t.e(-1).e(2).e(2).e(-1) == t.e(2).e(1).e(1).e(2)
True
sage: t.e(-1).e(2).e(2).e(-1).pp()
1  1
   2
sage: all(t.e(-1).e(2).e(2).e(-1).e(i) is None for i in {-1, 1, 2})
True

sage: SPT = ShiftedPrimedTableaux([4])
sage: t = SPT([[2,2,2,2]])
sage: t.e(-1).pp()
```

(continues on next page)

(continued from previous page)

```
1  2  2  2
sage: t.e(1).pp()
1  2  2  2
sage: t.e(-1).e(-1) is None
True
sage: t.e(1).e(1).pp()
1  1  2  2
```

```
>>> from sage.all import *
>>> SPT = ShiftedPrimedTableaux([Integer(5),Integer(4),Integer(2)])
>>> t = SPT([[Integer(1),Integer(1),Integer(1),'2p','3p'], [Integer(2),'3p',
↪Integer(3),Integer(3)],[Integer(3),Integer(4)]])
>>> t.pp()
1  1  1  2' 3'
   2  3' 3  3
      3  4
>>> s = t.e(Integer(2))
>>> s.pp()
1  1  1  2' 3'
   2  2  3  3
      3  4
>>> t == s.f(Integer(2))
True

>>> SPT = ShiftedPrimedTableaux([Integer(2),Integer(1)])
>>> t = SPT([[Integer(2),'3p'],[Integer(3)]])
>>> t.e(-Integer(1)).pp()
1  3'
   3
>>> t.e(Integer(1)).pp()
1  3'
   3
>>> t.e(Integer(2)).pp()
2  2
   3

>>> r = SPT([[Integer(2), Integer(2)],[Integer(3)]])
>>> r.e(-Integer(1)).pp()
1  2
   3
>>> r.e(Integer(1)).pp()
1  2
   3
>>> r.e(Integer(2)) is None
True

>>> r = SPT([[Integer(1),'3p'],[Integer(3)]])
>>> r.e(-Integer(1)) is None
True
>>> r.e(Integer(1)) is None
True
>>> r.e(Integer(2)).pp()
```

(continues on next page)

(continued from previous page)

```
1  2'
   3
>>> r = SPT([[Integer(1),'2p'],[Integer(3)]])
>>> r.e(-Integer(1)).pp()
1  1
   3
>>> r.e(Integer(1)) is None
True
>>> r.e(Integer(2)).pp()
1  2'
   2
>>> t = SPT([[Integer(2),'3p'],[Integer(3)]])
>>> t.e(-Integer(1)).e(Integer(2)).e(Integer(2)).e(-Integer(1)) == t.
↪e(Integer(2)).e(Integer(1)).e(Integer(1)).e(Integer(2))
True
>>> t.e(-Integer(1)).e(Integer(2)).e(Integer(2)).e(-Integer(1)).pp()
1  1
   2
>>> all(t.e(-Integer(1)).e(Integer(2)).e(Integer(2)).e(-Integer(1)).e(i) is␣
↪None for i in {-Integer(1), Integer(1), Integer(2)})
True

>>> SPT = ShiftedPrimedTableaux([Integer(4)])
>>> t = SPT([[Integer(2),Integer(2),Integer(2),Integer(2)]])
>>> t.e(-Integer(1)).pp()
1  2  2  2
>>> t.e(Integer(1)).pp()
1  2  2  2
>>> t.e(-Integer(1)).e(-Integer(1)) is None
True
>>> t.e(Integer(1)).e(Integer(1)).pp()
1  1  2  2
```

f(*ind*)

Compute the action of the crystal operator f_i on a shifted primed tableau using cases from the papers [HPS2017] and [AO2018].

INPUT:

- `ind` – element in the index set of the crystal

OUTPUT: primed tableau or `None`

EXAMPLES:

```
sage: SPT = ShiftedPrimedTableaux([5,4,2])
sage: t = SPT([[1,1,1,1,'3p'],[2,2,2,'3p'],[3,3]])
sage: t.pp()
1  1  1  1  3'
   2  2  2  3'
      3  3
sage: s = t.f(2)
sage: s is None
True
```

(continues on next page)

(continued from previous page)

```
sage: t = SPT([[1,1,1,'2p','3p'],[2,2,3,3],[3,4]])
sage: t.pp()
1  1  1  2' 3'
   2  2  3  3
      3  4
sage: s = t.f(2)
sage: s.pp()
1  1  1  2' 3'
   2  3' 3  3
      3  4

sage: SPT = ShiftedPrimedTableaux([2,1])
sage: t = SPT([[1,1],[2]])
sage: t.f(-1).pp()
1  2'
   2
sage: t.f(1).pp()
1  2'
   2
sage: t.f(2).pp()
1  1
   3

sage: r = SPT([[1,'2p'],[2]])
sage: r.f(-1) is None
True
sage: r.f(1) is None
True
sage: r.f(2).pp()
1  2'
   3

sage: r = SPT([[1,1],[3]])
sage: r.f(-1).pp()
1  2'
   3
sage: r.f(1).pp()
1  2
   3
sage: r.f(2) is None
True

sage: r = SPT([[1,2],[3]])
sage: r.f(-1).pp()
2  2
   3
sage: r.f(1).pp()
2  2
   3
sage: r.f(2) is None
True
```

(continues on next page)

(continued from previous page)

```
sage: t = SPT([[1,1],[2]])
sage: t.f(-1).f(2).f(2).f(-1) == t.f(2).f(1).f(-1).f(2)
True
sage: t.f(-1).f(2).f(2).f(-1).pp()
2  3'
   3
sage: all(t.f(-1).f(2).f(2).f(-1).f(i) is None for i in {-1, 1, 2})
True

sage: SPT = ShiftedPrimedTableaux([4])
sage: t = SPT([[1,1,1,1]])
sage: t.f(-1).pp()
1  1  1  2'
sage: t.f(1).pp()
1  1  1  2
sage: t.f(-1).f(-1) is None
True
sage: t.f(1).f(-1).pp()
1  1  2' 2
sage: t.f(1).f(1).pp()
1  1  2  2
sage: t.f(1).f(1).f(-1).pp()
1  2' 2  2
sage: t.f(1).f(1).f(1).pp()
1  2  2  2
sage: t.f(1).f(1).f(1).f(-1).pp()
2  2  2  2
sage: t.f(1).f(1).f(1).f(1).pp()
2  2  2  2
sage: t.f(1).f(1).f(1).f(1).f(-1) is None
True
```

```
>>> from sage.all import *
>>> SPT = ShiftedPrimedTableaux([Integer(5),Integer(4),Integer(2)])
>>> t = SPT([[Integer(1),Integer(1),Integer(1),Integer(1),'3p'],[Integer(2),
↪Integer(2),Integer(2),'3p'],[Integer(3),Integer(3)]])
>>> t.pp()
1  1  1  1  3'
   2  2  2  3'
      3  3
>>> s = t.f(Integer(2))
>>> s is None
True

>>> t = SPT([[Integer(1),Integer(1),Integer(1),'2p','3p'],[Integer(2),
↪Integer(2),Integer(3),Integer(3)],[Integer(3),Integer(4)]])
>>> t.pp()
1  1  1  2' 3'
   2  2  3  3
      3  4
>>> s = t.f(Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> s.pp()
1  1  1  2' 3'
   2  3' 3  3
      3  4

>>> SPT = ShiftedPrimedTableaux([Integer(2),Integer(1)])
>>> t = SPT([[Integer(1),Integer(1)],[Integer(2)]])
>>> t.f(-Integer(1)).pp()
1  2'
   2
>>> t.f(Integer(1)).pp()
1  2'
   2
>>> t.f(Integer(2)).pp()
1  1
   3

>>> r = SPT([[Integer(1),'2p'],[Integer(2)]])
>>> r.f(-Integer(1)) is None
True
>>> r.f(Integer(1)) is None
True
>>> r.f(Integer(2)).pp()
1  2'
   3

>>> r = SPT([[Integer(1),Integer(1)],[Integer(3)]])
>>> r.f(-Integer(1)).pp()
1  2'
   3
>>> r.f(Integer(1)).pp()
1  2
   3
>>> r.f(Integer(2)) is None
True

>>> r = SPT([[Integer(1),Integer(2)],[Integer(3)]])
>>> r.f(-Integer(1)).pp()
2  2
   3
>>> r.f(Integer(1)).pp()
2  2
   3
>>> r.f(Integer(2)) is None
True

>>> t = SPT([[Integer(1),Integer(1)],[Integer(2)]])
>>> t.f(-Integer(1)).f(Integer(2)).f(Integer(2)).f(-Integer(1)) == t.
↪f(Integer(2)).f(Integer(1)).f(-Integer(1)).f(Integer(2))
True
>>> t.f(-Integer(1)).f(Integer(2)).f(Integer(2)).f(-Integer(1)).pp()
2  3'
```

(continues on next page)

(continued from previous page)

```
   3
>>> all(t.f(-Integer(1)).f(Integer(2)).f(Integer(2)).f(-Integer(1)).f(i) is␣
↪None for i in {-Integer(1), Integer(1), Integer(2)})
True

>>> SPT = ShiftedPrimedTableaux([Integer(4)])
>>> t = SPT([[Integer(1),Integer(1),Integer(1),Integer(1)]])
>>> t.f(-Integer(1)).pp()
1  1  1  2'
>>> t.f(Integer(1)).pp()
1  1  1  2
>>> t.f(-Integer(1)).f(-Integer(1)) is None
True
>>> t.f(Integer(1)).f(-Integer(1)).pp()
1  1  2' 2
>>> t.f(Integer(1)).f(Integer(1)).pp()
1  1  2  2
>>> t.f(Integer(1)).f(Integer(1)).f(-Integer(1)).pp()
1  2' 2  2
>>> t.f(Integer(1)).f(Integer(1)).f(Integer(1)).pp()
1  2  2  2
>>> t.f(Integer(1)).f(Integer(1)).f(Integer(1)).f(-Integer(1)).pp()
2  2  2  2
>>> t.f(Integer(1)).f(Integer(1)).f(Integer(1)).f(Integer(1)).pp()
2  2  2  2
>>> t.f(Integer(1)).f(Integer(1)).f(Integer(1)).f(Integer(1)).f(-Integer(1))␣
↪is None
True
```

is_highest_weight(*index_set=None*)

Return whether `self` is a highest weight element of the crystal.

An element is highest weight if it vanishes under all crystal operators e_i.

EXAMPLES:

```
sage: SPT = ShiftedPrimedTableaux([5,4,2])
sage: t = SPT([(1, 1, 1, 1, 1), (2, 2, 2, "3p"), (3, 3)])
sage: t.is_highest_weight()
True

sage: SPT = ShiftedPrimedTableaux([5,4])
sage: s = SPT([(1, 1, 1, 1, 1), (2, 2, "3p", 3)])
sage: s.is_highest_weight(index_set=[1])
True
```

```
>>> from sage.all import *
>>> SPT = ShiftedPrimedTableaux([Integer(5),Integer(4),Integer(2)])
>>> t = SPT([(Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)),␣
↪(Integer(2), Integer(2), Integer(2), "3p"), (Integer(3), Integer(3))])
>>> t.is_highest_weight()
True
```

(continues on next page)

(continued from previous page)

```
>>> SPT = ShiftedPrimedTableaux([Integer(5),Integer(4)])
>>> s = SPT([(Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)),
↪(Integer(2), Integer(2), "3p", Integer(3))])
>>> s.is_highest_weight(index_set=[Integer(1)])
True
```

reading_word()

Return the reading word of `self`.

The reading word of a shifted primed tableau is constructed as follows:

1. List all primed entries in the tableau, column by column, in decreasing order within each column, moving from the rightmost column to the left, and with all the primes removed (i.e. all entries are increased by half a unit).
2. Then list all unprimed entries, row by row, in increasing order within each row, moving from the bottommost row to the top.

EXAMPLES:

```
sage: SPT = ShiftedPrimedTableaux([4,2])
sage: t = SPT([[1,'2p',2,2],[2,'3p']])
sage: t.reading_word()
[3, 2, 2, 1, 2, 2]
```

```
>>> from sage.all import *
>>> SPT = ShiftedPrimedTableaux([Integer(4),Integer(2)])
>>> t = SPT([[Integer(1),'2p',Integer(2),Integer(2)],[Integer(2),'3p']])
>>> t.reading_word()
[3, 2, 2, 1, 2, 2]
```

weight()

Return the weight of `self`.

The weight of a shifted primed tableau is defined to be the vector with i-th component equal to the number of entries i and i' in the tableau.

EXAMPLES:

```
sage: t = ShiftedPrimedTableau([[1,'2p',2,2],[2,'3p']])
sage: t.weight()
(1, 4, 1)
```

```
>>> from sage.all import *
>>> t = ShiftedPrimedTableau([[Integer(1),'2p',Integer(2),Integer(2)],
↪[Integer(2),'3p']])
>>> t.weight()
(1, 4, 1)
```

class sage.combinat.shifted_primed_tableau.**PrimedEntry**(*entry=None*, *double=None*)

Bases: `SageObject`

The class of entries in shifted primed tableaux.

An entry in a shifted primed tableau is an element in the alphabet $\{1' < 1 < 2' < 2 < \cdots < n' < n\}$. The difference between two elements i and $i-1$ counts as a whole unit, whereas the difference between i and i' counts as half a unit. Internally, we represent an unprimed element x as $2x$ and the primed elements as the corresponding odd integer that respects the total order.

INPUT:

- `entry` – half integer or string of an integer possibly ending in `p` or `'`
- `double` – the doubled value

decrease_half()

Decrease `self` by half a unit.

decrease_one()

Decrease `self` by one unit.

increase_half()

Increase `self` by half a unit.

increase_one()

Increase `self` by one unit.

integer()

Return the corresponding integer i for primed entries of the form i or i'.

is_primed()

Check if `self` is a primed element.

is_unprimed()

Check if `self` is an unprimed element.

primed()

Prime `self` if it is an unprimed element.

unprimed()

Unprime `self` if it is a primed element.

class sage.combinat.shifted_primed_tableau.**ShiftedPrimedTableau**(*parent*, *T*, *skew=None*, *check=True*, *preprocessed=False*)

Bases: `ClonableArray`

A shifted primed tableau.

A primed tableau is a tableau of shifted shape in the alphabet $X' = \{1' < 1 < 2' < 2 < \cdots < n' < n\}$ such that

1. the entries are weakly increasing along rows and columns;
2. a row cannot have two repeated primed elements, and a column cannot have two repeated non-primed elements;

Skew shape of the shifted primed tableaux is specified either with an optional argument `skew` or with `None` entries.

Primed entries in the main diagonal can be allowed with the optional boolean parameter `primed_diagonal`` (default: ``False`).

EXAMPLES:

```
sage: T = ShiftedPrimedTableaux([4,2])
sage: T([[1,"2'","3'",3],[2,"3'"]])[1]
(2, 3')
sage: t = ShiftedPrimedTableau([[1,"2p",2.5,3],[2,2.5]])
```

(continues on next page)

(continued from previous page)

```
sage: t[1]
(2, 3')
sage: ShiftedPrimedTableau([["2p",2,3],["2p","3p"],[2]], skew=[2,1])
[(None, None, 2', 2, 3), (None, 2', 3'), (2,)]
sage: ShiftedPrimedTableau([[None,None,"2p"],[None,"2p"]])
[(None, None, 2'), (None, 2')]
sage: T = ShiftedPrimedTableaux([4,2], primed_diagonal=True)
sage: T([[1,"2'","3'",3],["2'","3'"]])[1] # With primed diagonal entry
(2', 3')
```

```
>>> from sage.all import *
>>> T = ShiftedPrimedTableaux([Integer(4),Integer(2)])
>>> T([[Integer(1),"2'","3'",Integer(3)],[Integer(2),"3'"]])[Integer(1)]
(2, 3')
>>> t = ShiftedPrimedTableau([[Integer(1),"2p",RealNumber('2.5'),Integer(3)],
→[Integer(2),RealNumber('2.5')]])
>>> t[Integer(1)]
(2, 3')
>>> ShiftedPrimedTableau([["2p",Integer(2),Integer(3)],["2p","3p"],[Integer(2)]],␣
→skew=[Integer(2),Integer(1)])
[(None, None, 2', 2, 3), (None, 2', 3'), (2,)]
>>> ShiftedPrimedTableau([[None,None,"2p"],[None,"2p"]])
[(None, None, 2'), (None, 2')]
>>> T = ShiftedPrimedTableaux([Integer(4),Integer(2)], primed_diagonal=True)
>>> T([[Integer(1),"2'","3'",Integer(3)],["2'","3'"]])[Integer(1)] # With primed␣
→diagonal entry
(2', 3')
```

check()

Check that `self` is a valid primed tableau.

EXAMPLES:

```
sage: T = ShiftedPrimedTableaux([4,2])
sage: t = T([[1,'2p',2,2],[2,'3p']])
sage: t.check()
sage: s = ShiftedPrimedTableau([["2p",2,3],["2p"],[2]],skew=[2,1])
sage: s.check()
sage: t = T([['1p','2p',2,2],[2,'3p']])
Traceback (most recent call last):
...
ValueError: [['1p', '2p', 2, 2], [2, '3p']] is not an element of
 Shifted Primed Tableaux of shape [4, 2]

sage: T = ShiftedPrimedTableaux([4,2], primed_diagonal=True)
sage: t = T([['1p','2p',2,2],[2,'3p']])  # primed_diagonal allowed
sage: t.check()
sage: t = T([['1p','1p',2,2],[2,'3p']])
Traceback (most recent call last):
...
ValueError: [['1p', '1p', 2, 2], [2, '3p']] is not an element of
 Shifted Primed Tableaux of shape [4, 2] and maximum entry 6
```

```
>>> from sage.all import *
>>> T = ShiftedPrimedTableaux([Integer(4),Integer(2)])
>>> t = T([[Integer(1),'2p',Integer(2),Integer(2)],[Integer(2),'3p']])
>>> t.check()
>>> s = ShiftedPrimedTableau([["2p",Integer(2),Integer(3)],["2p"],
↪[Integer(2)]],skew=[Integer(2),Integer(1)])
>>> s.check()
>>> t = T([['1p','2p',Integer(2),Integer(2)],[Integer(2),'3p']])
Traceback (most recent call last):
...
ValueError: [['1p', '2p', 2, 2], [2, '3p']] is not an element of
 Shifted Primed Tableaux of shape [4, 2]

>>> T = ShiftedPrimedTableaux([Integer(4),Integer(2)], primed_diagonal=True)
>>> t = T([['1p','2p',Integer(2),Integer(2)],[Integer(2),'3p']])   # primed_
↪diagonal allowed
>>> t.check()
>>> t = T([['1p','1p',Integer(2),Integer(2)],[Integer(2),'3p']])
Traceback (most recent call last):
...
ValueError: [['1p', '1p', 2, 2], [2, '3p']] is not an element of
 Shifted Primed Tableaux of shape [4, 2] and maximum entry 6
```

is_standard()

Return `True` if the entries of `self` are in bijection with positive primed integers $1', 1, 2', \ldots, n$.

EXAMPLES:

```
sage: ShiftedPrimedTableau([["1'", 1, "2'"], [2, "3'"]],
....:                      primed_diagonal=True).is_standard()
True
sage: ShiftedPrimedTableau([["1'", 1, 2], ["2'", "3'"]],
....:                      primed_diagonal=True).is_standard()
True
sage: ShiftedPrimedTableau([["1'", 1, 1], ["2'", 2]],
....:                      primed_diagonal=True).is_standard()
False
sage: ShiftedPrimedTableau([[1, "2'"], [2]]).is_standard()
False
sage: s = ShiftedPrimedTableau([[None, None,"1p","2p",2],[None,"1"]])
sage: s.is_standard()
True
```

```
>>> from sage.all import *
>>> ShiftedPrimedTableau([["1'", Integer(1), "2'"], [Integer(2), "3'"]],
...                      primed_diagonal=True).is_standard()
True
>>> ShiftedPrimedTableau([["1'", Integer(1), Integer(2)], ["2'", "3'"]],
...                      primed_diagonal=True).is_standard()
True
>>> ShiftedPrimedTableau([["1'", Integer(1), Integer(1)], ["2'", Integer(2)]],
...                      primed_diagonal=True).is_standard()
False
```

(continues on next page)

(continued from previous page)

```
>>> ShiftedPrimedTableau([[Integer(1), "2'"], [Integer(2)]]).is_standard()
False
>>> s = ShiftedPrimedTableau([[None, None,"1p","2p",Integer(2)],[None,"1"]])
>>> s.is_standard()
True
```

max_entry()

Return the minimum unprimed letter $x > y$ for all y in `self`.

EXAMPLES:

```
sage: Tab = ShiftedPrimedTableau([(1,1,'2p','3p'),(2,2)])
sage: Tab.max_entry()
3
```

```
>>> from sage.all import *
>>> Tab = ShiftedPrimedTableau([(Integer(1),Integer(1),'2p','3p'),(Integer(2),
↪Integer(2))])
>>> Tab.max_entry()
3
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: t = ShiftedPrimedTableau([[1,'2p',2,2],[2,'3p']])
sage: t.pp()
1  2' 2  2
   2  3'
sage: t = ShiftedPrimedTableau([[10,'11p',11,11],[11,'12']])
sage: t.pp()
10  11' 11  11
    11  12
sage: s = ShiftedPrimedTableau([['2p',2,3],['2p']],skew=[2,1])
sage: s.pp()
 .  .  2' 2  3
    .  2'
```

```
>>> from sage.all import *
>>> t = ShiftedPrimedTableau([[Integer(1),'2p',Integer(2),Integer(2)],
↪[Integer(2),'3p']])
>>> t.pp()
1  2' 2  2
   2  3'
>>> t = ShiftedPrimedTableau([[Integer(10),'11p',Integer(11),Integer(11)],
↪[Integer(11),'12']])
>>> t.pp()
10  11' 11  11
    11  12
>>> s = ShiftedPrimedTableau([['2p',Integer(2),Integer(3)],['2p']],
↪skew=[Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> s.pp()
 .  .  2' 2  3
    .  2'
```

restrict(*n*)

Return the restriction of the shifted tableau to all the numbers less than or equal to n.

Note: If only the outer shape of the restriction, rather than the whole restriction, is needed, then the faster method *restriction_outer_shape()* is preferred. Similarly if only the skew shape is needed, use *restriction_shape()*.

EXAMPLES:

```
sage: t = ShiftedPrimedTableau([[1,'2p',2,2],[2,'3p']])
sage: t.restrict(2).pp()
1  2' 2  2
   2

sage: t.restrict("2p").pp()
1  2'

sage: s = ShiftedPrimedTableau([["2p",2,3],["2p"]], skew=[2,1])
sage: s.restrict(2).pp()
.  .  2' 2
   .  2'
sage: s.restrict(1.5).pp()
.  .  2'
   .  2'
```

```
>>> from sage.all import *
>>> t = ShiftedPrimedTableau([[Integer(1),'2p',Integer(2),Integer(2)],
↪[Integer(2),'3p']])
>>> t.restrict(Integer(2)).pp()
1  2' 2  2
   2

>>> t.restrict("2p").pp()
1  2'

>>> s = ShiftedPrimedTableau([["2p",Integer(2),Integer(3)],["2p"]],␣
↪skew=[Integer(2),Integer(1)])
>>> s.restrict(Integer(2)).pp()
.  .  2' 2
   .  2'
>>> s.restrict(RealNumber('1.5')).pp()
.  .  2'
   .  2'
```

restriction_outer_shape(*n*)

Return the outer shape of the restriction of the shifted tableau `self` to n.

If T is a (skew) shifted tableau and n is a half-integer, then the restriction of T to n is defined as the (skew) shifted tableau obtained by removing all cells filled with entries greater than n from T.

This method computes merely the outer shape of the restriction. For the restriction itself, use `restrict()`.

EXAMPLES:

```
sage: s = ShiftedPrimedTableau([["2p",2,3],["2p"]], skew=[2,1])
sage: s.pp()
.  .  2' 2  3
   .  2'
sage: s.restriction_outer_shape(2)
[4, 2]
sage: s.restriction_outer_shape("2p")
[3, 2]
```

```
>>> from sage.all import *
>>> s = ShiftedPrimedTableau([["2p",Integer(2),Integer(3)],["2p"]],
↪skew=[Integer(2),Integer(1)])
>>> s.pp()
.  .  2' 2  3
   .  2'
>>> s.restriction_outer_shape(Integer(2))
[4, 2]
>>> s.restriction_outer_shape("2p")
[3, 2]
```

restriction_shape(*n*)

Return the skew shape of the restriction of the skew tableau `self` to `n`.

If T is a shifted tableau and n is a half-integer, then the restriction of T to n is defined as the (skew) shifted tableau obtained by removing all cells filled with entries greater than n from T.

This method computes merely the skew shape of the restriction. For the restriction itself, use `restrict()`.

EXAMPLES:

```
sage: s = ShiftedPrimedTableau([["2p",2,3],["2p"]], skew=[2,1])
sage: s.pp()
 .  .  2' 2  3
    .  2'

sage: s.restriction_shape(2)
[4, 2] / [2, 1]
```

```
>>> from sage.all import *
>>> s = ShiftedPrimedTableau([["2p",Integer(2),Integer(3)],["2p"]],
↪skew=[Integer(2),Integer(1)])
>>> s.pp()
 .  .  2' 2  3
    .  2'

>>> s.restriction_shape(Integer(2))
[4, 2] / [2, 1]
```

shape()

Return the shape of the underlying partition of `self`.

EXAMPLES:

```
sage: t = ShiftedPrimedTableau([[1,'2p',2,2],[2,'3p']])
sage: t.shape()
[4, 2]
sage: s = ShiftedPrimedTableau([["2p",2,3],["2p"]],skew=[2,1])
sage: s.shape()
[5, 2] / [2, 1]
```

```
>>> from sage.all import *
>>> t = ShiftedPrimedTableau([[Integer(1),'2p',Integer(2),Integer(2)],
↪[Integer(2),'3p']])
>>> t.shape()
[4, 2]
>>> s = ShiftedPrimedTableau([["2p",Integer(2),Integer(3)],["2p"]],
↪skew=[Integer(2),Integer(1)])
>>> s.shape()
[5, 2] / [2, 1]
```

to_chain()

Return the chain of partitions corresponding to the (skew) shifted tableau `self`, interlaced by one of the colours `1` is the added cell is on the diagonal, `2` if an ordinary entry is added and `3` if a primed entry is added.

EXAMPLES:

```
sage: s = ShiftedPrimedTableau([(1, 2, 3.5, 5, 6.5), (3, 5.5)])
sage: s.pp()
 1  2  4' 5  7'
    3  6'

sage: s.to_chain()
[[], 1, [1], 2, [2], 1, [2, 1], 3, [3, 1], 2, [4, 1], 3, [4, 2], 3, [5, 2]]


sage: s = ShiftedPrimedTableau([(1, 3.5), (2.5,), (6,)], skew=[2,1])
sage: s.pp()
 .  .  1  4'
    .  3'
       6

sage: s.to_chain()
[[2, 1], 2, [3, 1], 0, [3, 1], 3, [3, 2], 3, [4, 2], 0, [4, 2], 1, [4, 2, 1]]
```

```
>>> from sage.all import *
>>> s = ShiftedPrimedTableau([(Integer(1), Integer(2), RealNumber('3.5'),␣
↪Integer(5), RealNumber('6.5')), (Integer(3), RealNumber('5.5'))])
>>> s.pp()
 1  2  4' 5  7'
    3  6'

>>> s.to_chain()
```

(continues on next page)

(continued from previous page)

```
[[], 1, [1], 2, [2], 1, [2, 1], 3, [3, 1], 2, [4, 1], 3, [4, 2], 3, [5, 2]]


>>> s = ShiftedPrimedTableau([(Integer(1), RealNumber('3.5')), (RealNumber('2.
↪5'),), (Integer(6),)], skew=[Integer(2),Integer(1)])
>>> s.pp()
 .  .  1  4'
    .  3'
       6

>>> s.to_chain()
[[2, 1], 2, [3, 1], 0, [3, 1], 3, [3, 2], 3, [4, 2], 0, [4, 2], 1, [4, 2, 1]]
```

weight()

Return the weight of `self`.

The weight of a shifted primed tableau is defined to be the vector with i-th component equal to the number of entries i and i' in the tableau.

EXAMPLES:

```
sage: t = ShiftedPrimedTableau([['2p',2,2],[2,'3p']], skew=[1])
sage: t.weight()
(0, 4, 1)
```

```
>>> from sage.all import *
>>> t = ShiftedPrimedTableau([['2p',Integer(2),Integer(2)],[Integer(2),'3p']],
↪ skew=[Integer(1)])
>>> t.weight()
(0, 4, 1)
```

class sage.combinat.shifted_primed_tableau.**ShiftedPrimedTableaux**(*skew=None*, *primed_diagonal=False*)

Bases: `UniqueRepresentation`, `Parent`

Return the combinatorial class of shifted primed tableaux subject to the constraints given by the arguments.

A primed tableau is a tableau of shifted shape on the alphabet $X' = \{1' < 1 < 2' < 2 < \cdots < n' < n\}$ such that

1. the entries are weakly increasing along rows and columns
2. a row cannot have two repeated primed entries, and a column cannot have two repeated non-primed entries

INPUT:

Valid optional keywords:

- `shape` – the (outer skew) shape of tableaux
- `weight` – the weight of tableaux
- `max_entry` – the maximum entry of tableaux
- `skew` – the inner skew shape of tableaux
- `primed_diagonal` – allow primed entries in main diagonal of tableaux

The weight of a tableau is defined to be the vector with i-th component equal to the number of entries i and i' in the tableau. The sum of the coordinates in the weight vector must be equal to the number of entries in the partition.

The shape and skew must be strictly decreasing partitions. The primed_diagonal is a boolean (default: False).

EXAMPLES:

```
sage: SPT = ShiftedPrimedTableaux(weight=(1,2,2), shape=[3,2]); SPT
Shifted Primed Tableaux of weight (1, 2, 2) and shape [3, 2]
sage: SPT.list()
[[(1, 2, 2), (3, 3)],
 [(1, 2', 3'), (2, 3)],
 [(1, 2', 3'), (2, 3')],
 [(1, 2', 2), (3, 3)]]
sage: SPT = ShiftedPrimedTableaux(weight=(1,2,2), shape=[3,2],
....:                              primed_diagonal=True); SPT
Shifted Primed Tableaux of weight (1, 2, 2) and shape [3, 2]
sage: SPT.list()
[[(1, 2, 2), (3, 3)],
 [(1, 2, 2), (3', 3)],
 [(1, 2', 3'), (2, 3)],
 [(1, 2', 3'), (2, 3')],
 [(1, 2', 3'), (2', 3)],
 [(1, 2', 3'), (2', 3')],
 [(1, 2', 2), (3, 3)],
 [(1, 2', 2), (3', 3)],
 [(1', 2, 2), (3, 3)],
 [(1', 2, 2), (3', 3)],
 [(1', 2', 3'), (2, 3)],
 [(1', 2', 3'), (2, 3')],
 [(1', 2', 3'), (2', 3)],
 [(1', 2', 3'), (2', 3')],
 [(1', 2', 2), (3, 3)],
 [(1', 2', 2), (3', 3)]]
sage: SPT = ShiftedPrimedTableaux(weight=(1,2)); SPT
Shifted Primed Tableaux of weight (1, 2)
sage: list(SPT)
[[(1, 2, 2)], [(1, 2', 2)], [(1, 2'), (2,)]]
sage: SPT = ShiftedPrimedTableaux(weight=(1,2), primed_diagonal=True)
sage: list(SPT)
[[(1, 2, 2)],
 [(1, 2', 2)],
 [(1', 2, 2)],
 [(1', 2', 2)],
 [(1, 2'), (2,)],
 [(1, 2'), (2',)],
 [(1', 2'), (2,)],
 [(1', 2'), (2',)]]
sage: SPT = ShiftedPrimedTableaux([3,2], max_entry=2); SPT
Shifted Primed Tableaux of shape [3, 2] and maximum entry 2
sage: list(SPT)
[[(1, 1, 1), (2, 2)], [(1, 1, 2'), (2, 2)]]
sage: SPT = ShiftedPrimedTableaux([3,2], max_entry=2,
....:                              primed_diagonal=True)
sage: list(SPT)
[[(1, 1, 1), (2, 2)],
 [(1, 1, 1), (2', 2)],
```

(continues on next page)

(continued from previous page)

```
 [(1', 1, 1), (2, 2)],
 [(1', 1, 1), (2', 2)],
 [(1, 1, 2'), (2, 2)],
 [(1, 1, 2'), (2', 2)],
 [(1', 1, 2'), (2, 2)],
 [(1', 1, 2'), (2', 2)]]
```

```
>>> from sage.all import *
>>> SPT = ShiftedPrimedTableaux(weight=(Integer(1),Integer(2),Integer(2)),
↪shape=[Integer(3),Integer(2)]); SPT
Shifted Primed Tableaux of weight (1, 2, 2) and shape [3, 2]
>>> SPT.list()
[[(1, 2, 2), (3, 3)],
 [(1, 2', 3'), (2, 3)],
 [(1, 2', 3'), (2, 3')],
 [(1, 2', 2), (3, 3)]]
>>> SPT = ShiftedPrimedTableaux(weight=(Integer(1),Integer(2),Integer(2)),
↪shape=[Integer(3),Integer(2)],
...                                 primed_diagonal=True); SPT
Shifted Primed Tableaux of weight (1, 2, 2) and shape [3, 2]
>>> SPT.list()
[[(1, 2, 2), (3, 3)],
 [(1, 2, 2), (3', 3)],
 [(1, 2', 3'), (2, 3)],
 [(1, 2', 3'), (2, 3')],
 [(1, 2', 3'), (2', 3)],
 [(1, 2', 3'), (2', 3')],
 [(1, 2', 2), (3, 3)],
 [(1, 2', 2), (3', 3)],
 [(1', 2, 2), (3, 3)],
 [(1', 2, 2), (3', 3)],
 [(1', 2', 3'), (2, 3)],
 [(1', 2', 3'), (2, 3')],
 [(1', 2', 3'), (2', 3)],
 [(1', 2', 3'), (2', 3')],
 [(1', 2', 2), (3, 3)],
 [(1', 2', 2), (3', 3)]]
>>> SPT = ShiftedPrimedTableaux(weight=(Integer(1),Integer(2))); SPT
Shifted Primed Tableaux of weight (1, 2)
>>> list(SPT)
[[(1, 2, 2)], [(1, 2', 2)], [(1, 2'), (2,)]]
>>> SPT = ShiftedPrimedTableaux(weight=(Integer(1),Integer(2)), primed_
↪diagonal=True)
>>> list(SPT)
[[(1, 2, 2)],
 [(1, 2', 2)],
 [(1', 2, 2)],
 [(1', 2', 2)],
 [(1, 2'), (2,)],
 [(1, 2'), (2',)],
 [(1', 2'), (2,)],
 [(1', 2'), (2',)]]
```

(continues on next page)

(continued from previous page)

```
>>> SPT = ShiftedPrimedTableaux([Integer(3),Integer(2)], max_entry=Integer(2));␣
↪SPT
Shifted Primed Tableaux of shape [3, 2] and maximum entry 2
>>> list(SPT)
[[(1, 1, 1), (2, 2)], [(1, 1, 2'), (2, 2)]]
>>> SPT = ShiftedPrimedTableaux([Integer(3),Integer(2)], max_entry=Integer(2),
...                             primed_diagonal=True)
>>> list(SPT)
[[(1, 1, 1), (2, 2)],
 [(1, 1, 1), (2', 2)],
 [(1', 1, 1), (2, 2)],
 [(1', 1, 1), (2', 2)],
 [(1, 1, 2'), (2, 2)],
 [(1, 1, 2'), (2', 2)],
 [(1', 1, 2'), (2, 2)],
 [(1', 1, 2'), (2', 2)]]
```

See also

- *ShiftedPrimedTableau*

Element

alias of *ShiftedPrimedTableau*

options = Current options for Tableaux - ascii_art: repr - convention: English - display: list - latex: diagram

class sage.combinat.shifted_primed_tableau.**ShiftedPrimedTableaux_all**(*skew=None*, *primed_diagonal=False*)

Bases: *ShiftedPrimedTableaux*

The class of all shifted primed tableaux.

class sage.combinat.shifted_primed_tableau.**ShiftedPrimedTableaux_shape**(*shape*, *max_entry=None*, *skew=None*, *primed_diagonal=False*)

Bases: *ShiftedPrimedTableaux*

Shifted primed tableaux of a fixed shape.

Shifted primed tableaux admit a type A_n classical crystal structure with highest weights corresponding to a given shape.

The list of module generators consists of all elements of the crystal with nonincreasing weight entries.

The crystal is constructed following operations described in [HPS2017] and [AO2018].

The optional `primed_diagonal` allows primed entries in the main diagonal of all the Shifted primed tableaux of a fixed shape. If the `max_entry` is `None` then `max_entry` is set to the total number of entries in the tableau if `primed_diagonal` is `True`.

EXAMPLES:

```
sage: ShiftedPrimedTableaux([4,3,1], max_entry=4)
Shifted Primed Tableaux of shape [4, 3, 1] and maximum entry 4
sage: ShiftedPrimedTableaux([4,3,1], max_entry=4).cardinality()
384
```

```
>>> from sage.all import *
>>> ShiftedPrimedTableaux([Integer(4),Integer(3),Integer(1)], max_
↪entry=Integer(4))
Shifted Primed Tableaux of shape [4, 3, 1] and maximum entry 4
>>> ShiftedPrimedTableaux([Integer(4),Integer(3),Integer(1)], max_
↪entry=Integer(4)).cardinality()
384
```

We compute some of the crystal structure:

```
sage: SPTC = crystals.ShiftedPrimedTableaux([3,2], 3)
sage: T = SPTC.module_generators[-1]
sage: T
[(1, 1, 2'), (2, 3')]
sage: T.f(2)
[(1, 1, 3'), (2, 3')]
sage: len(SPTC.module_generators)
7
sage: SPTC[0]
[(1, 1, 1), (2, 2)]
sage: SPTC.cardinality()
24
```

```
>>> from sage.all import *
>>> SPTC = crystals.ShiftedPrimedTableaux([Integer(3),Integer(2)], Integer(3))
>>> T = SPTC.module_generators[-Integer(1)]
>>> T
[(1, 1, 2'), (2, 3')]
>>> T.f(Integer(2))
[(1, 1, 3'), (2, 3')]
>>> len(SPTC.module_generators)
7
>>> SPTC[Integer(0)]
[(1, 1, 1), (2, 2)]
>>> SPTC.cardinality()
24
```

We compare this implementation with the $q(n)$-crystal on (tensor products) of letters:

```
sage: tableau_crystal = crystals.ShiftedPrimedTableaux([4,1], 3)
sage: tableau_digraph = tableau_crystal.digraph()
sage: c = crystals.Letters(['Q', 3])
sage: tensor_crystal = tensor([c]*5)
sage: u = tensor_crystal(c(1), c(1), c(1), c(2), c(1))
sage: subcrystal = tensor_crystal.subcrystal(generators=[u],
....:                                        index_set=[1,2,-1])
sage: tensor_digraph = subcrystal.digraph()
```

(continues on next page)

(continued from previous page)

```
sage: tensor_digraph.is_isomorphic(tableau_digraph, edge_labels=True)
True
```

```
>>> from sage.all import *
>>> tableau_crystal = crystals.ShiftedPrimedTableaux([Integer(4),Integer(1)],␣
↪Integer(3))
>>> tableau_digraph = tableau_crystal.digraph()
>>> c = crystals.Letters(['Q', Integer(3)])
>>> tensor_crystal = tensor([c]*Integer(5))
>>> u = tensor_crystal(c(Integer(1)), c(Integer(1)), c(Integer(1)), c(Integer(2)),
↪ c(Integer(1)))
>>> subcrystal = tensor_crystal.subcrystal(generators=[u],
...                                        index_set=[Integer(1),Integer(2),-
↪Integer(1)])
>>> tensor_digraph = subcrystal.digraph()
>>> tensor_digraph.is_isomorphic(tableau_digraph, edge_labels=True)
True
```

If we allow primed entries in the main diagonal:

```
sage: ShiftedPrimedTableaux([4,3,1], max_entry=4,
....:                       primed_diagonal=True)
Shifted Primed Tableaux of shape [4, 3, 1] and maximum entry 4
sage: ShiftedPrimedTableaux([4,3,1], max_entry=4,
....:                      primed_diagonal=True).cardinality()
3072
sage: SPTC = ShiftedPrimedTableaux([3,2], max_entry=3,
....:                              primed_diagonal=True)
sage: T = SPTC[-1]
sage: T
[(1', 2', 2), (3', 3)]
sage: SPTC[0]
[(1, 1, 1), (2, 2)]
sage: SPTC.cardinality()
96
```

```
>>> from sage.all import *
>>> ShiftedPrimedTableaux([Integer(4),Integer(3),Integer(1)], max_
↪entry=Integer(4),
...                       primed_diagonal=True)
Shifted Primed Tableaux of shape [4, 3, 1] and maximum entry 4
>>> ShiftedPrimedTableaux([Integer(4),Integer(3),Integer(1)], max_
↪entry=Integer(4),
...                      primed_diagonal=True).cardinality()
3072
>>> SPTC = ShiftedPrimedTableaux([Integer(3),Integer(2)], max_entry=Integer(3),
...                              primed_diagonal=True)
>>> T = SPTC[-Integer(1)]
>>> T
[(1', 2', 2), (3', 3)]
>>> SPTC[Integer(0)]
[(1, 1, 1), (2, 2)]
```

(continues on next page)

(continued from previous page)

```
>>> SPTC.cardinality()
96
```

module_generators()

Return the generators of `self` as a crystal.

shape()

Return the shape of the shifted tableaux `self`.

class sage.combinat.shifted_primed_tableau.**ShiftedPrimedTableaux_weight**(*weight*, *skew=None*, *primed_diagonal=False*)

Bases: *ShiftedPrimedTableaux*

Shifted primed tableaux of fixed weight.

EXAMPLES:

```
sage: ShiftedPrimedTableaux(weight=(2,3,1))
Shifted Primed Tableaux of weight (2, 3, 1)
sage: ShiftedPrimedTableaux(weight=(2,3,1)).cardinality()
17
sage: SPT = ShiftedPrimedTableaux(weight=(2,3,1), primed_diagonal=True)
sage: SPT.cardinality()
64
sage: T = ShiftedPrimedTableaux(weight=(3,2), primed_diagonal=True)
sage: T[:5]
[[(1, 1, 1, 2, 2)],
 [(1, 1, 1, 2', 2)],
 [(1', 1, 1, 2, 2)],
 [(1', 1, 1, 2', 2)],
 [(1, 1, 1, 2), (2,)]]
sage: T.cardinality()
16
```

```
>>> from sage.all import *
>>> ShiftedPrimedTableaux(weight=(Integer(2),Integer(3),Integer(1)))
Shifted Primed Tableaux of weight (2, 3, 1)
>>> ShiftedPrimedTableaux(weight=(Integer(2),Integer(3),Integer(1))).cardinality()
17
>>> SPT = ShiftedPrimedTableaux(weight=(Integer(2),Integer(3),Integer(1)), primed_
↪diagonal=True)
>>> SPT.cardinality()
64
>>> T = ShiftedPrimedTableaux(weight=(Integer(3),Integer(2)), primed_
↪diagonal=True)
>>> T[:Integer(5)]
[[(1, 1, 1, 2, 2)],
 [(1, 1, 1, 2', 2)],
 [(1', 1, 1, 2, 2)],
 [(1', 1, 1, 2', 2)],
 [(1, 1, 1, 2), (2,)]]
>>> T.cardinality()
16
```

class sage.combinat.shifted_primed_tableau.**ShiftedPrimedTableaux_weight_shape**(*weight*, *shape*, *skew=None*, *primed_diagonal=False*)

Bases: *ShiftedPrimedTableaux*

Shifted primed tableaux of the fixed weight and shape.

EXAMPLES:

```
sage: ShiftedPrimedTableaux([4,2,1], weight=(2,3,2))
Shifted Primed Tableaux of weight (2, 3, 2) and shape [4, 2, 1]
sage: ShiftedPrimedTableaux([4,2,1], weight=(2,3,2)).cardinality()
4
sage: T = ShiftedPrimedTableaux([4,2,1], weight=(2,3,2),
....:                            primed_diagonal=True)
sage: T[:6]
[[(1, 1, 2, 2), (2, 3'), (3,)],
 [(1, 1, 2, 2), (2, 3'), (3',)],
 [(1, 1, 2, 2), (2', 3'), (3,)],
 [(1, 1, 2, 2), (2', 3'), (3',)],
 [(1, 1, 2', 3), (2, 2), (3,)],
 [(1, 1, 2', 3), (2, 2), (3',)]]
sage: T.cardinality()
32
```

```
>>> from sage.all import *
>>> ShiftedPrimedTableaux([Integer(4),Integer(2),Integer(1)], weight=(Integer(2),
↪Integer(3),Integer(2)))
Shifted Primed Tableaux of weight (2, 3, 2) and shape [4, 2, 1]
>>> ShiftedPrimedTableaux([Integer(4),Integer(2),Integer(1)], weight=(Integer(2),
↪Integer(3),Integer(2))).cardinality()
4
>>> T = ShiftedPrimedTableaux([Integer(4),Integer(2),Integer(1)],␣
↪weight=(Integer(2),Integer(3),Integer(2)),
...                           primed_diagonal=True)
>>> T[:Integer(6)]
[[(1, 1, 2, 2), (2, 3'), (3,)],
 [(1, 1, 2, 2), (2, 3'), (3',)],
 [(1, 1, 2, 2), (2', 3'), (3,)],
 [(1, 1, 2, 2), (2', 3'), (3',)],
 [(1, 1, 2', 3), (2, 2), (3,)],
 [(1, 1, 2', 3), (2, 2), (3',)]]
>>> T.cardinality()
32
```

5.1.307 Shuffle product of iterables

The shuffle product of two sequences of lengths m and n is a sum over the $\binom{m+n}{n}$ ways of interleaving the two sequences.

That could be defined inductively by:

$$(a_n)_{n\geqslant 0} \Cup (b_m)_{m\geqslant 0} = a_0 \cdot ((a_n)_{n\geqslant 1} \Cup (b_m)_{m\geqslant 0}) + b_0 \cdot ((a_n)_{n\geqslant 0} \Cup (b_m)_{m\geqslant 1})$$

with (a_n) and (b_m) two non-empty sequences and if one of them is empty then the product is equals to the other.

The shuffle product has been introduced by S. Eilenberg and S. Mac Lane in 1953 [EilLan53].

EXAMPLES:

```
sage: from sage.combinat.shuffle import ShuffleProduct
sage: list(ShuffleProduct([1,2], ["a", "b", "c"]))
[[1, 2, 'a', 'b', 'c'],
 ['a', 1, 2, 'b', 'c'],
 [1, 'a', 2, 'b', 'c'],
 ['a', 'b', 1, 2, 'c'],
 ['a', 1, 'b', 2, 'c'],
 [1, 'a', 'b', 2, 'c'],
 ['a', 'b', 'c', 1, 2],
 ['a', 'b', 1, 'c', 2],
 ['a', 1, 'b', 'c', 2],
 [1, 'a', 'b', 'c', 2]]
```

```
>>> from sage.all import *
>>> from sage.combinat.shuffle import ShuffleProduct
>>> list(ShuffleProduct([Integer(1),Integer(2)], ["a", "b", "c"]))
[[1, 2, 'a', 'b', 'c'],
 ['a', 1, 2, 'b', 'c'],
 [1, 'a', 2, 'b', 'c'],
 ['a', 'b', 1, 2, 'c'],
 ['a', 1, 'b', 2, 'c'],
 [1, 'a', 'b', 2, 'c'],
 ['a', 'b', 'c', 1, 2],
 ['a', 'b', 1, 'c', 2],
 ['a', 1, 'b', 'c', 2],
 [1, 'a', 'b', 'c', 2]]
```

References:

Author:

- Jean-Baptiste Priez

class sage.combinat.shuffle.**SetShuffleProduct**(*l1*, *l2*, *element_constructor=None*)

Bases: *ShuffleProduct_abstract*

The union of all possible shuffle products of two sets of iterables.

EXAMPLES:

```
sage: from sage.combinat.shuffle import SetShuffleProduct
sage: sorted(SetShuffleProduct({(1,), (2,3)}, {(4,5), (6,)}))
[[1, 4, 5],
 [1, 6],
 [2, 3, 4, 5],
 [2, 3, 6],
 [2, 4, 3, 5],
 [2, 4, 5, 3],
 [2, 6, 3],
 [4, 1, 5],
 [4, 2, 3, 5],
```

(continues on next page)

(continued from previous page)

```
 [4, 2, 5, 3],
 [4, 5, 1],
 [4, 5, 2, 3],
 [6, 1],
 [6, 2, 3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.shuffle import SetShuffleProduct
>>> sorted(SetShuffleProduct({(Integer(1),), (Integer(2),Integer(3))},
↪{(Integer(4),Integer(5)), (Integer(6),)}))
[[1, 4, 5],
 [1, 6],
 [2, 3, 4, 5],
 [2, 3, 6],
 [2, 4, 3, 5],
 [2, 4, 5, 3],
 [2, 6, 3],
 [4, 1, 5],
 [4, 2, 3, 5],
 [4, 2, 5, 3],
 [4, 5, 1],
 [4, 5, 2, 3],
 [6, 1],
 [6, 2, 3]]
```

cardinality()

The cardinality is defined by the sum of the cardinality of all shuffles. That means by a sum of binomials.

class sage.combinat.shuffle.**ShuffleProduct**(*l1*, *l2*, *element_constructor=None*)

Bases: *ShuffleProduct_abstract*

Shuffle product of two iterables.

EXAMPLES:

```
sage: from sage.combinat.shuffle import ShuffleProduct
sage: list(ShuffleProduct("abc", "de", element_constructor="".join))
['abcde',
 'adbce',
 'dabce',
 'abdce',
 'adebc',
 'daebc',
 'deabc',
 'adbec',
 'dabec',
 'abdec']
sage: list(ShuffleProduct("", "de", element_constructor="".join))
['de']
```

```
>>> from sage.all import *
>>> from sage.combinat.shuffle import ShuffleProduct
>>> list(ShuffleProduct("abc", "de", element_constructor="".join))
```

(continues on next page)

(continued from previous page)

```
['abcde',
 'adbce',
 'dabce',
 'abdce',
 'adebc',
 'daebc',
 'deabc',
 'adbec',
 'dabec',
 'abdec']
>>> list(ShuffleProduct("", "de", element_constructor="".join))
['de']
```

cardinality()

Return the number of shuffles of l_1 and l_2, respectively of lengths m and n, which is $\binom{m+n}{n}$.

class sage.combinat.shuffle.**ShuffleProduct_abstract**(*l1*, *l2*, *element_constructor=None*)

Bases: Parent

Abstract base class for shuffle products.

class sage.combinat.shuffle.**ShuffleProduct_overlapping**(*w1*, *w2*, *element_constructor=None*, *add=<built-in function add>*)

Bases: *ShuffleProduct_abstract*

The overlapping shuffle product of the two words w1 and w2.

If u and v are two words whose letters belong to an additive monoid or to another kind of alphabet on which addition is well-defined, then the *overlapping shuffle product* of u and v is a certain multiset of words defined as follows: Let a and b be the lengths of u and v, respectively. Let A be the set $\{(0,1),(0,2),\cdots,(0,a)\}$, and let B be the set $\{(1,1),(1,2),\cdots,(1,b)\}$. Notice that the sets A and B are disjoint. We can make A and B into posets by setting $(k,i)\leq(k,j)$ for all $k\in\{0,1\}$ and $i\leq j$. Then, $A\cup B$ becomes a poset by disjoint union (we don't set $(0,i)\leq(1,i)$). Let p be the map from $A\cup B$ to the set of all letters which sends every $(0,i)$ to the i-th letter of u, and every $(1,j)$ to the j-th letter of v. For every nonnegative integer c and every surjective map $f:A\cup B\to\{1,2,\cdots,c\}$ for which both restrictions $f\mid_A$ and $f\mid_B$ are strictly increasing, let $w(f)$ be the length-c word such that for every $1\leq k\leq c$, the k-th letter of $w(f)$ equals $\sum_{j\in f^{-1}(k)}p(j)$ (this sum always has either one or two addends). The overlapping shuffle product of u and v is then the multiset of all $w(f)$ with c ranging over all nonnegative integers and f ranging over the surjective maps $f:A\cup B\to\{1,2,\cdots,c\}$ for which both restrictions $f\mid_A$ and $f\mid_B$ are strictly increasing.

If one restricts c to a particular fixed nonnegative integer, then the multiset is instead called the *overlapping shuffle product with precisely `a + b - c` overlaps*. This is nonempty only if $\max\{a,b\}\leq c\leq a+b$.

If $c=a+b$, then the overlapping shuffle product with precisely $a+b-c$ overlaps is plainly the shuffle product (ShuffleProduct_w1w2).

INPUT:

- w1, w2 – iterables
- element_constructor – (default: the parent of w1) the function used to construct the output
- add – (default: +) the addition function

EXAMPLES:

```
sage: from sage.combinat.shuffle import ShuffleProduct_overlapping
sage: w, u = [[2, 9], [9, 1]]
sage: S = ShuffleProduct_overlapping(w, u)
sage: sorted(S)
[[2, 9, 1, 9],
 [2, 9, 9, 1],
 [2, 9, 9, 1],
 [2, 9, 10],
 [2, 18, 1],
 [9, 1, 2, 9],
 [9, 2, 1, 9],
 [9, 2, 9, 1],
 [9, 2, 10],
 [9, 3, 9],
 [11, 1, 9],
 [11, 9, 1],
 [11, 10]]
sage: A = [{1,2}, {3,4}]
sage: B = [{2,3}, {4,5,6}]
sage: S = ShuffleProduct_overlapping(A, B, add=lambda X,Y: X.union(Y))
sage: list(S)
[[{1, 2}, {3, 4}, {2, 3}, {4, 5, 6}],
 [{1, 2}, {2, 3}, {3, 4}, {4, 5, 6}],
 [{1, 2}, {2, 3}, {4, 5, 6}, {3, 4}],
 [{2, 3}, {1, 2}, {3, 4}, {4, 5, 6}],
 [{2, 3}, {1, 2}, {4, 5, 6}, {3, 4}],
 [{2, 3}, {4, 5, 6}, {1, 2}, {3, 4}],
 [{1, 2, 3}, {3, 4}, {4, 5, 6}],
 [{1, 2}, {2, 3, 4}, {4, 5, 6}],
 [{1, 2, 3}, {4, 5, 6}, {3, 4}],
 [{1, 2}, {2, 3}, {3, 4, 5, 6}],
 [{2, 3}, {1, 2, 4, 5, 6}, {3, 4}],
 [{2, 3}, {1, 2}, {3, 4, 5, 6}],
 [{1, 2, 3}, {3, 4, 5, 6}]]
```

```
>>> from sage.all import *
>>> from sage.combinat.shuffle import ShuffleProduct_overlapping
>>> w, u = [[Integer(2), Integer(9)], [Integer(9), Integer(1)]]
>>> S = ShuffleProduct_overlapping(w, u)
>>> sorted(S)
[[2, 9, 1, 9],
 [2, 9, 9, 1],
 [2, 9, 9, 1],
 [2, 9, 10],
 [2, 18, 1],
 [9, 1, 2, 9],
 [9, 2, 1, 9],
 [9, 2, 9, 1],
 [9, 2, 10],
 [9, 3, 9],
 [11, 1, 9],
 [11, 9, 1],
```

(continues on next page)

(continued from previous page)

```
 [11, 10]]
>>> A = [{Integer(1),Integer(2)}, {Integer(3),Integer(4)}]
>>> B = [{Integer(2),Integer(3)}, {Integer(4),Integer(5),Integer(6)}]
>>> S = ShuffleProduct_overlapping(A, B, add=lambda X,Y: X.union(Y))
>>> list(S)
[[{1, 2}, {3, 4}, {2, 3}, {4, 5, 6}],
 [{1, 2}, {2, 3}, {3, 4}, {4, 5, 6}],
 [{1, 2}, {2, 3}, {4, 5, 6}, {3, 4}],
 [{2, 3}, {1, 2}, {3, 4}, {4, 5, 6}],
 [{2, 3}, {1, 2}, {4, 5, 6}, {3, 4}],
 [{2, 3}, {4, 5, 6}, {1, 2}, {3, 4}],
 [{1, 2, 3}, {3, 4}, {4, 5, 6}],
 [{1, 2}, {2, 3, 4}, {4, 5, 6}],
 [{1, 2, 3}, {4, 5, 6}, {3, 4}],
 [{1, 2}, {2, 3}, {3, 4, 5, 6}],
 [{2, 3}, {1, 2, 4, 5, 6}, {3, 4}],
 [{2, 3}, {1, 2}, {3, 4, 5, 6}],
 [{1, 2, 3}, {3, 4, 5, 6}]]
```

class sage.combinat.shuffle.**ShuffleProduct_overlapping_r**(*w1*, *w2*, *r*, *element_constructor=None*, *add=<built-in function add>*)

Bases: *ShuffleProduct_abstract*

The overlapping shuffle product of the two words `w1` and `w2` with precisely `r` overlaps.

See *ShuffleProduct_overlapping* for a definition.

EXAMPLES:

```
sage: from sage.combinat.shuffle import ShuffleProduct_overlapping_r
sage: w, u = map(Words(range(20)), [[2, 9], [9, 1]])
sage: S = ShuffleProduct_overlapping_r(w,u,1)
sage: list(S)
[word: 11,9,1,
 word: 2,18,1,
 word: 11,1,9,
 word: 2,9,10,
 word: 939,
 word: 9,2,10]
```

```
>>> from sage.all import *
>>> from sage.combinat.shuffle import ShuffleProduct_overlapping_r
>>> w, u = map(Words(range(Integer(20))), [[Integer(2), Integer(9)], [Integer(9),
↪Integer(1)]])
>>> S = ShuffleProduct_overlapping_r(w,u,Integer(1))
>>> list(S)
[word: 11,9,1,
 word: 2,18,1,
 word: 11,1,9,
 word: 2,9,10,
 word: 939,
 word: 9,2,10]
```

5.1.308 Sidon sets and their generalizations, Sidon g-sets

AUTHORS:

- Martin Raum (07-25-2011)

`sage.combinat.sidon_sets.`**`sidon_sets`**`(N, g=1)`

Return the set of all Sidon-g sets that have elements less than or equal to N.

A Sidon-g set is a set of positive integers $A \subset [1, N]$ such that any integer M can be obtain at most g times as sums of unordered pairs of elements of A (the two elements are not necessary distinct):

$$\#\{(a_i, a_j) | a_i, a_j \in A, a_i + a_j = M, a_i \leq a_j\} \leq g$$

INPUT:

- `N` – positive integer
- `g` – positive integer (default: 1)

OUTPUT: a Sage set with categories whose element are also set of integers

EXAMPLES:

```
sage: S = sidon_sets(3, 2)
sage: sorted(S, key=str)
[{1, 2, 3}, {1, 2}, {1, 3}, {1}, {2, 3}, {2}, {3}, {}]
sage: S.cardinality()
8
sage: S.category()
Category of finite enumerated sets
sage: sid = S.an_element()
sage: sid
{2}
sage: sid.category()
Category of finite enumerated sets
```

```
>>> from sage.all import *
>>> S = sidon_sets(Integer(3), Integer(2))
>>> sorted(S, key=str)
[{1, 2, 3}, {1, 2}, {1, 3}, {1}, {2, 3}, {2}, {3}, {}]
>>> S.cardinality()
8
>>> S.category()
Category of finite enumerated sets
>>> sid = S.an_element()
>>> sid
{2}
>>> sid.category()
Category of finite enumerated sets
```

`sage.combinat.sidon_sets.`**`sidon_sets_rec`**`(g=1)`

Return the set of all Sidon-g sets that have elements less than or equal to N without checking the arguments. This internal function should not be call directly by user.

5.1.309 Similarity class types of matrices with entries in a finite field

The notion of a matrix conjugacy class type was introduced by J. A. Green in [Green55], in the context of computing the irreducible characters of finite general linear groups. The class types are equivalence classes of similarity classes of square matrices with entries in a finite field which, roughly speaking, have the same qualitative properties.

For example, all similarity classes of the same class type have centralizers of the same cardinality and the same degrees of elementary divisors. Qualitative properties of similarity classes such as semisimplicity and regularity descend to class types.

The most important feature of similarity class types is that, for any n, the number of similarity class types of $n \times n$ matrices is independent of q. This makes it possible to perform many combinatorial calculations treating q as a formal variable.

In order to define similarity class types, recall that similarity classes of $n \times n$ matrices with entries in $\mathbf{F}_q$ correspond to functions

$$c : \mathrm{Irr}\mathbf{F}_{q[t]} \to \Lambda$$

such that

$$\sum_{f \in \mathrm{Irr}\mathbf{F}_{q[t]}} |c(f)| \deg f = n,$$

where we denote the set of irreducible monic polynomials in $\mathbf{F}_{q[t]}$ by $\mathrm{Irr}\mathbf{F}_{q[t]}$, the set of all partitions by Λ, and the size of $\lambda \in \Lambda$ by $|\lambda|$.

Similarity classes indexed by functions c_1 and c_2 as above are said to be of the same type if there exists a degree-preserving self-bijection σ of $\mathrm{Irr}\mathbf{F}_{q[t]}$ such that $c_2 = c_1 \circ \sigma$. Thus, the type of c remembers only the degrees of the polynomials (and not the polynomials themselves) for which c takes a certain value λ. Replacing each irreducible polynomial of degree d for which c takes a non-trivial value λ by the pair (d, λ), we obtain a multiset of such pairs. Clearly, c_1 and c_2 have the same type if and only if these multisets are equal. Thus a similarity class type may be viewed as a multiset of pairs of the form (d, λ).

For 2×2 matrices there are four types:

```
sage: for tau in SimilarityClassTypes(2):
....:     print(tau)
[[1, [1]], [1, [1]]]
[[1, [2]]]
[[1, [1, 1]]]
[[2, [1]]]
```

```
>>> from sage.all import *
>>> for tau in SimilarityClassTypes(Integer(2)):
...     print(tau)
[[1, [1]], [1, [1]]]
[[1, [2]]]
[[1, [1, 1]]]
[[2, [1]]]
```

These four types correspond to the regular split semisimple matrices, the non-semisimple matrices, the central matrices and the irreducible matrices respectively.

For any matrix A in a given similarity class type, it is possible to calculate the number elements in the similarity class of A, the dimension of the algebra of matrices in $M_n(A)$ that commute with A, and the cardinality of the subgroup of $GL_n(\mathbf{F}_q)$ that commute with A. For each similarity class type, it is also possible to compute the number of classes of that type (and hence, the total number of matrices of that type). All these calculations treat the cardinality q of the finite field as a formal variable:

```
sage: M = SimilarityClassType([[1, [1]], [1, [1]]])
sage: M.class_card()
q^2 + q
sage: M.centralizer_algebra_dim()
2
sage: M.centralizer_group_card()
q^2 - 2*q + 1
sage: M.number_of_classes()
1/2*q^2 - 1/2*q
sage: M.number_of_matrices()
1/2*q^4 - 1/2*q^2
```

```
>>> from sage.all import *
>>> M = SimilarityClassType([[Integer(1), [Integer(1)]], [Integer(1), [Integer(1)]]])
>>> M.class_card()
q^2 + q
>>> M.centralizer_algebra_dim()
2
>>> M.centralizer_group_card()
q^2 - 2*q + 1
>>> M.number_of_classes()
1/2*q^2 - 1/2*q
>>> M.number_of_matrices()
1/2*q^4 - 1/2*q^2
```

We now describe two applications of similarity class types.

We say that an $n \times n$ matrix has rational canonical form type λ for some partition λ of n if the diagonal blocks in the rational canonical form have sizes given by the parts of λ. Thus the matrices with rational canonical type (n) are the regular ones, while the matrices with rational canonical type (1^n) are the central ones.

Using similarity class types, it becomes easy to get a formula for the number of matrices with a given rational canonical type:

```
sage: def matrices_with_rcf(la):
....:     return sum([tau.number_of_matrices() for tau in filter(lambda tau:tau.
↪rcf()==la, SimilarityClassTypes(la.size()))])
sage: matrices_with_rcf(Partition([2,1]))
q^6 + q^5 + q^4 - q^3 - q^2 - q
```

```
>>> from sage.all import *
>>> def matrices_with_rcf(la):
...     return sum([tau.number_of_matrices() for tau in filter(lambda tau:tau.
↪rcf()==la, SimilarityClassTypes(la.size()))])
>>> matrices_with_rcf(Partition([Integer(2),Integer(1)]))
q^6 + q^5 + q^4 - q^3 - q^2 - q
```

Similarity class types can also be used to calculate the number of simultaneous similarity classes of k-tuples of $n \times n$ matrices with entries in $\mathbf{F}_q$ by using Burnside's lemma:

```
sage: from sage.combinat.similarity_class_type import order_of_general_linear_group,␣
↪centralizer_algebra_dim
sage: q = ZZ['q'].gen()
```

(continues on next page)

(continued from previous page)

```
sage: def simultaneous_similarity_classes(n, k):
....:     return SimilarityClassTypes(n).sum(lambda la: q**(k*centralizer_algebra_
↪dim(la)), invertible = True)/order_of_general_linear_group(n)
sage: simultaneous_similarity_classes(3, 2)
q^10 + q^8 + 2*q^7 + 2*q^6 + 2*q^5 + q^4
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import order_of_general_linear_group,␣
↪centralizer_algebra_dim
>>> q = ZZ['q'].gen()
>>> def simultaneous_similarity_classes(n, k):
...     return SimilarityClassTypes(n).sum(lambda la: q**(k*centralizer_algebra_
↪dim(la)), invertible = True)/order_of_general_linear_group(n)
>>> simultaneous_similarity_classes(Integer(3), Integer(2))
q^10 + q^8 + 2*q^7 + 2*q^6 + 2*q^5 + q^4
```

Similarity class types can be used to compute the coefficients of generating functions coming from the cycle index type techniques of Kung and Stong (see Morrison [Morrison06]).

They can also be used to compute the number of invariant subspaces for a matrix over a finite field of any given dimension. For this we use the elegant recursive formula of Ramaré [R17] (see also [PR22]).

Along with the results of [PSS13], similarity class types can be used to calculate the number of similarity classes of matrices of order n with entries in a principal ideal local ring of length two with residue field of cardinality q with centralizer of any given cardinality up to $n = 4$. Among these, the classes which are selftranspose can also be counted:

```
sage: from sage.combinat.similarity_class_type import matrix_centralizer_
↪cardinalities_length_two
sage: list(matrix_centralizer_cardinalities_length_two(3))
[(q^6 - 3*q^5 + 3*q^4 - q^3, 1/6*q^6 - 1/2*q^5 + 1/3*q^4),
(q^6 - 2*q^5 + q^4, q^5 - q^4),
(q^8 - 3*q^7 + 3*q^6 - q^5, 1/2*q^5 - q^4 + 1/2*q^3),
(q^8 - 2*q^7 + q^6, q^4 - q^3),
(q^10 - 2*q^9 + 2*q^7 - q^6, q^4 - q^3),
(q^8 - q^7 - q^6 + q^5, 1/2*q^5 - q^4 + 1/2*q^3),
(q^6 - q^5 - q^4 + q^3, 1/2*q^6 - 1/2*q^5),
(q^6 - q^5, q^4),
(q^10 - 2*q^9 + q^8, q^3),
(q^8 - 2*q^7 + q^6, q^4 - q^3),
(q^8 - q^7, q^3 + q^2),
(q^12 - 3*q^11 + 3*q^10 - q^9, 1/6*q^4 - 1/2*q^3 + 1/3*q^2),
(q^12 - 2*q^11 + q^10, q^3 - q^2),
(q^14 - 2*q^13 + 2*q^11 - q^10, q^3 - q^2),
(q^12 - q^11 - q^10 + q^9, 1/2*q^4 - 1/2*q^3),
(q^12 - q^11, q^2),
(q^14 - 2*q^13 + q^12, q^2),
(q^18 - q^17 - q^16 + q^14 + q^13 - q^12, q^2),
(q^12 - q^9, 1/3*q^4 - 1/3*q^2),
(q^6 - q^3, 1/3*q^6 - 1/3*q^4)]
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import matrix_centralizer_cardinalities_
↪length_two
```

(continues on next page)

(continued from previous page)

```
>>> list(matrix_centralizer_cardinalities_length_two(Integer(3)))
[(q^6 - 3*q^5 + 3*q^4 - q^3, 1/6*q^6 - 1/2*q^5 + 1/3*q^4),
(q^6 - 2*q^5 + q^4, q^5 - q^4),
(q^8 - 3*q^7 + 3*q^6 - q^5, 1/2*q^5 - q^4 + 1/2*q^3),
(q^8 - 2*q^7 + q^6, q^4 - q^3),
(q^10 - 2*q^9 + 2*q^7 - q^6, q^4 - q^3),
(q^8 - q^7 - q^6 + q^5, 1/2*q^5 - q^4 + 1/2*q^3),
(q^6 - q^5 - q^4 + q^3, 1/2*q^6 - 1/2*q^5),
(q^6 - q^5, q^4),
(q^10 - 2*q^9 + q^8, q^3),
(q^8 - 2*q^7 + q^6, q^4 - q^3),
(q^8 - q^7, q^3 + q^2),
(q^12 - 3*q^11 + 3*q^10 - q^9, 1/6*q^4 - 1/2*q^3 + 1/3*q^2),
(q^12 - 2*q^11 + q^10, q^3 - q^2),
(q^14 - 2*q^13 + 2*q^11 - q^10, q^3 - q^2),
(q^12 - q^11 - q^10 + q^9, 1/2*q^4 - 1/2*q^3),
(q^12 - q^11, q^2),
(q^14 - 2*q^13 + q^12, q^2),
(q^18 - q^17 - q^16 + q^14 + q^13 - q^12, q^2),
(q^12 - q^9, 1/3*q^4 - 1/3*q^2),
(q^6 - q^3, 1/3*q^6 - 1/3*q^4)]
```

REFERENCES:

AUTHOR:

- Amritanshu Prasad (2013-07-18): initial implementation
- Amritanshu Prasad (2013-09-09): added functions for similarity classes over rings of length two
- Amritanshu Prasad (2022-07-31): added computation of similarity class type of a given matrix and invariant subspace generating function

class sage.combinat.similarity_class_type.**PrimarySimilarityClassType**(*parent*, *deg*, *par*)

Bases: `Element`

A primary similarity class type is a pair consisting of a partition and a positive integer.

For a partition λ and a positive integer d, the primary similarity class type (d, λ) represents similarity classes of square matrices of order $|\lambda| \cdot d$ with entries in a finite field of order q which correspond to the $\mathbf{F}_{q[t]}$-module

$$\frac{\mathbf{F}_{q[t]}}{p(t)^{\lambda_1}} \oplus \frac{\mathbf{F}_{q[t]}}{p(t)^{\lambda_2}} \oplus \cdots$$

for some irreducible polynomial $p(t)$ of degree d.

centralizer_algebra_dim()

Return the dimension of the algebra of matrices which commute with a matrix of type `self`.

For a partition (d, λ) this dimension is given by $d(\lambda_1 + 3\lambda_2 + 5\lambda_3 + \cdots)$.

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 2, 1])
sage: PT.centralizer_algebra_dim()
28
```

```
>>> from sage.all import *
>>> PT = PrimarySimilarityClassType(Integer(2), [Integer(3), Integer(2),␣
↪Integer(1)])
>>> PT.centralizer_algebra_dim()
28
```

centralizer_group_card(*q=None*)

Return the cardinality of the centralizer group of a matrix of type `self` in a field of order `q`.

INPUT:

- `q` – integer or an indeterminate

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(1, [])
sage: PT.centralizer_group_card()
1
sage: PT = PrimarySimilarityClassType(2, [1, 1])
sage: PT.centralizer_group_card()
q^8 - q^6 - q^4 + q^2
```

```
>>> from sage.all import *
>>> PT = PrimarySimilarityClassType(Integer(1), [])
>>> PT.centralizer_group_card()
1
>>> PT = PrimarySimilarityClassType(Integer(2), [Integer(1), Integer(1)])
>>> PT.centralizer_group_card()
q^8 - q^6 - q^4 + q^2
```

degree()

Return degree of `self`.

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 2, 1])
sage: PT.degree()
2
```

```
>>> from sage.all import *
>>> PT = PrimarySimilarityClassType(Integer(2), [Integer(3), Integer(2),␣
↪Integer(1)])
>>> PT.degree()
2
```

invariant_subspace_generating_function(*q=None*, *t=None*)

Return the invariant subspace generating function of `self`.

INPUT:

- `q` – (optional) an integer or an inderminate
- `t` – (optional) an indeterminate

EXAMPLES:

```
sage: PrimarySimilarityClassType(1, [2, 2]).invariant_subspace_generating_
→function()
t^4 + (q + 1)*t^3 + (q^2 + q + 1)*t^2 + (q + 1)*t + 1
```

```
>>> from sage.all import *
>>> PrimarySimilarityClassType(Integer(1), [Integer(2), Integer(2)]).
→invariant_subspace_generating_function()
t^4 + (q + 1)*t^3 + (q^2 + q + 1)*t^2 + (q + 1)*t + 1
```

partition()

Return partition corresponding to `self`.

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 2, 1])
sage: PT.partition()
[3, 2, 1]
```

```
>>> from sage.all import *
>>> PT = PrimarySimilarityClassType(Integer(2), [Integer(3), Integer(2),␣
→Integer(1)])
>>> PT.partition()
[3, 2, 1]
```

size()

Return the size of `self`.

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 2, 1])
sage: PT.size()
12
```

```
>>> from sage.all import *
>>> PT = PrimarySimilarityClassType(Integer(2), [Integer(3), Integer(2),␣
→Integer(1)])
>>> PT.size()
12
```

statistic(*func*, *q=None*)

Return $n_\lambda(q^d)$ where n_λ is the value returned by `func` upon input λ, if `self` is (d, λ).

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 1])
sage: q = ZZ['q'].gen()
sage: PT.statistic(lambda la:q**la.size(), q = q)
q^8
```

```
>>> from sage.all import *
>>> PT = PrimarySimilarityClassType(Integer(2), [Integer(3), Integer(1)])
>>> q = ZZ['q'].gen()
```

(continues on next page)

(continued from previous page)

```
>>> PT.statistic(lambda la:q**la.size(), q = q)
q^8
```

class sage.combinat.similarity_class_type.**PrimarySimilarityClassTypes**(*n*, *min*)

Bases: UniqueRepresentation, Parent

All primary similarity class types of size n whose degree is greater than that of min or whose degree is that of min and whose partition is less than of min in lexicographic order.

A primary similarity class type of size n is a pair (λ, d) consisting of a partition λ and a positive integer d such that $|\lambda|d = n$.

INPUT:

- n – positive integer
- min – a primary matrix type of size n

EXAMPLES:

If min is not specified, then the class of all primary similarity class types of size n is created:

```
sage: PTC = PrimarySimilarityClassTypes(2)
sage: for PT in PTC:
....:     print(PT)
[1, [2]]
[1, [1, 1]]
[2, [1]]
```

```
>>> from sage.all import *
>>> PTC = PrimarySimilarityClassTypes(Integer(2))
>>> for PT in PTC:
...     print(PT)
[1, [2]]
[1, [1, 1]]
[2, [1]]
```

If min is specified, then the class consists of only those primary similarity class types whose degree is greater than that of min or whose degree is that of min and whose partition is less than of min in lexicographic order:

```
sage: PTC = PrimarySimilarityClassTypes(2, min = PrimarySimilarityClassType(1, [1,
↪ 1]))
sage: for PT in PTC:
....:     print(PT)
[1, [1, 1]]
[2, [1]]
```

```
>>> from sage.all import *
>>> PTC = PrimarySimilarityClassTypes(Integer(2), min =␣
↪PrimarySimilarityClassType(Integer(1), [Integer(1), Integer(1)]))
>>> for PT in PTC:
...     print(PT)
[1, [1, 1]]
[2, [1]]
```

Element

alias of *PrimarySimilarityClassType*

size()

Return size of elements of `self`.

The size of a primary similarity class type (d, λ) is $d|\lambda|$.

EXAMPLES:

```
sage: PTC = PrimarySimilarityClassTypes(2)
sage: PTC.size()
2
```

```
>>> from sage.all import *
>>> PTC = PrimarySimilarityClassTypes(Integer(2))
>>> PTC.size()
2
```

class sage.combinat.similarity_class_type.**SimilarityClassType**(*parent*, *tau*)

Bases: *CombinatorialElement*

A similarity class type.

A matrix type is a multiset of primary similarity class types.

INPUT:

- `tau` – list of primary similarity class types or a square matrix over a finite field

EXAMPLES:

```
sage: tau1 = SimilarityClassType([[3, [3, 2, 1]], [2, [2, 1]]]); tau1
[[2, [2, 1]], [3, [3, 2, 1]]]

sage: SimilarityClassType(Matrix(GF(2), [[1,1],[0,1]]))
[[1, [2]]]
```

```
>>> from sage.all import *
>>> tau1 = SimilarityClassType([[Integer(3), [Integer(3), Integer(2),␣
↪Integer(1)]], [Integer(2), [Integer(2), Integer(1)]]]); tau1
[[2, [2, 1]], [3, [3, 2, 1]]]

>>> SimilarityClassType(Matrix(GF(Integer(2)), [[Integer(1),Integer(1)],
↪[Integer(0),Integer(1)]]))
[[1, [2]]]
```

as_partition_dictionary()

Return a dictionary whose keys are the partitions of types occurring in `self` and the value at the key λ is the partition formed by sorting the degrees of primary types with partition λ.

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.as_partition_dictionary()
{[1]: [1, 1]}
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(1), [Integer(1)]], [Integer(1),
↪[Integer(1)]]])
>>> tau.as_partition_dictionary()
{[1]: [1, 1]}
```

centralizer_algebra_dim()

Return the dimension of the algebra of matrices which commute with a matrix of type `self`.

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.centralizer_algebra_dim()
2
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(1), [Integer(1)]], [Integer(1),
↪[Integer(1)]]])
>>> tau.centralizer_algebra_dim()
2
```

centralizer_group_card(*q=None*)

Return the cardinality of the group of matrices in $GL_n(\mathbf{F}_q)$ which commute with a matrix of type `self`.

INPUT:

- `q` – integer or an indeterminate

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.centralizer_group_card()
q^2 - 2*q + 1
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(1), [Integer(1)]], [Integer(1),
↪[Integer(1)]]])
>>> tau.centralizer_group_card()
q^2 - 2*q + 1
```

class_card(*q=None*)

Return the number of matrices in each similarity class of type `self`.

INPUT:

- `q` – integer or an indeterminate

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1, 1, 1, 1]]])
sage: tau.class_card()
1
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.class_card()
q^2 + q
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(1), [Integer(1), Integer(1),␣
↪Integer(1), Integer(1)]]])
>>> tau.class_card()
1
>>> tau = SimilarityClassType([[Integer(1), [Integer(1)]], [Integer(1),␣
↪[Integer(1)]]])
>>> tau.class_card()
q^2 + q
```

invariant_subspace_generating_function(*q=None*, *t=None*)

Return the invariant subspace generating function of `self`.

The invariant subspace generating function is the function is the polynomial

$$\sum_{j\geq 0} a_j(q)t^j,$$

where $a_j(q)$ denotes the number of j-dimensional invariant subspaces of dimensiona j for any matrix with the similarity class type `self` with entries in a field of order q.

EXAMPLES:

```
sage: SimilarityClassType([[1, [2, 2]]]).invariant_subspace_generating_
↪function()
t^4 + (q + 1)*t^3 + (q^2 + q + 1)*t^2 + (q + 1)*t + 1
sage: A = Matrix(GF(2),[(0, 1, 0, 0), (0, 1, 1, 1), (1, 0, 1, 0), (1, 1, 0,␣
↪0)])
sage: SimilarityClassType(A).invariant_subspace_generating_function()
t^4 + 1
```

```
>>> from sage.all import *
>>> SimilarityClassType([[Integer(1), [Integer(2), Integer(2)]]]).invariant_
↪subspace_generating_function()
t^4 + (q + 1)*t^3 + (q^2 + q + 1)*t^2 + (q + 1)*t + 1
>>> A = Matrix(GF(Integer(2)),[(Integer(0), Integer(1), Integer(0),␣
↪Integer(0)), (Integer(0), Integer(1), Integer(1), Integer(1)), (Integer(1),␣
↪Integer(0), Integer(1), Integer(0)), (Integer(1), Integer(1), Integer(0),␣
↪Integer(0))])
>>> SimilarityClassType(A).invariant_subspace_generating_function()
t^4 + 1
```

is_regular()

Return `True` if every primary type in `self` has partition with one part.

EXAMPLES:

```
sage: tau = SimilarityClassType([[2, [1]], [1, [3]]])
sage: tau.is_regular()
True
sage: tau = SimilarityClassType([[2, [1, 1]], [1, [3]]])
sage: tau.is_regular()
False
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(2), [Integer(1)]], [Integer(1),␣
↪[Integer(3)]]])
>>> tau.is_regular()
True
>>> tau = SimilarityClassType([[Integer(2), [Integer(1), Integer(1)]],␣
↪[Integer(1), [Integer(3)]]])
>>> tau.is_regular()
False
```

is_semisimple()

Return True if every primary similarity class type in self has all parts equal to 1.

EXAMPLES:

```
sage: tau = SimilarityClassType([[2, [1, 1]], [1, [1]]])
sage: tau.is_semisimple()
True
sage: tau = SimilarityClassType([[2, [1, 1]], [1, [2]]])
sage: tau.is_semisimple()
False
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(2), [Integer(1), Integer(1)]],␣
↪[Integer(1), [Integer(1)]]])
>>> tau.is_semisimple()
True
>>> tau = SimilarityClassType([[Integer(2), [Integer(1), Integer(1)]],␣
↪[Integer(1), [Integer(2)]]])
>>> tau.is_semisimple()
False
```

number_of_classes(*invertible=False*, *q=None*)

Return the number of similarity classes of matrices of type self.

INPUT:

- invertible – boolean; return number of invertible classes if set to True
- q – integer or an indeterminate

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.number_of_classes()
1/2*q^2 - 1/2*q
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(1), [Integer(1)]], [Integer(1),␣
↪[Integer(1)]]])
>>> tau.number_of_classes()
1/2*q^2 - 1/2*q
```

number_of_matrices(*invertible=False*, *q=None*)

Return the number of matrices of type self.

INPUT:

- `invertible` – a boolean; return the number of invertible matrices if set

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]]])
sage: tau.number_of_matrices()
q
sage: tau.number_of_matrices(invertible = True)
q - 1
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.number_of_matrices()
1/2*q^4 - 1/2*q^2
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(1), [Integer(1)]]])
>>> tau.number_of_matrices()
q
>>> tau.number_of_matrices(invertible = True)
q - 1
>>> tau = SimilarityClassType([[Integer(1), [Integer(1)]], [Integer(1),␣
↪[Integer(1)]]])
>>> tau.number_of_matrices()
1/2*q^4 - 1/2*q^2
```

rcf()

Return the partition corresponding to the rational canonical form of a matrix of type `self`.

EXAMPLES:

```
sage: tau = SimilarityClassType([[2, [1, 1, 1]], [1, [3, 2]]])
sage: tau.rcf()
[5, 4, 2]
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(2), [Integer(1), Integer(1),␣
↪Integer(1)]], [Integer(1), [Integer(3), Integer(2)]]])
>>> tau.rcf()
[5, 4, 2]
```

size()

Return the sum of the sizes of the primary parts of `self`.

EXAMPLES:

```
sage: tau = SimilarityClassType([[3, [3, 2, 1]], [2, [2, 1]]])
sage: tau.size()
24
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(3), [Integer(3), Integer(2),␣
↪Integer(1)]], [Integer(2), [Integer(2), Integer(1)]]])
>>> tau.size()
24
```

statistic(*func*, *q=None*)

Return.

$$\prod_{(d,\lambda)\in\tau} n_\lambda(q^d)$$

where $n_\lambda(q)$ is the value returned by `func` on the input λ.

INPUT:

- `func` – a function that takes a partition to a polynomial in `q`
- `q` – integer or an indeterminate

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [2, 1]], [2, [1, 1]]])
sage: from sage.combinat.similarity_class_type import fq
sage: tau.statistic(lambda la: prod([fq(m) for m in la.to_exp()]))
(q^9 - 3*q^8 + 2*q^7 + 2*q^6 - 4*q^5 + 4*q^4 - 2*q^3 - 2*q^2 + 3*q - 1)/q^9
sage: q = ZZ['q'].gen()
sage: tau.statistic(lambda la: q**la.size(), q = q)
q^8
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(1), [Integer(1)]], [Integer(1),␣
↪[Integer(2), Integer(1)]], [Integer(2), [Integer(1), Integer(1)]]])
>>> from sage.combinat.similarity_class_type import fq
>>> tau.statistic(lambda la: prod([fq(m) for m in la.to_exp()]))
(q^9 - 3*q^8 + 2*q^7 + 2*q^6 - 4*q^5 + 4*q^4 - 2*q^3 - 2*q^2 + 3*q - 1)/q^9
>>> q = ZZ['q'].gen()
>>> tau.statistic(lambda la: q**la.size(), q = q)
q^8
```

class sage.combinat.similarity_class_type.**SimilarityClassTypes**(*n*, *min*)

Bases: `UniqueRepresentation`, `Parent`

Class of all similarity class types of size `n` with all primary matrix types greater than or equal to the primary matrix type `min`.

A similarity class type is a multiset of primary matrix types.

INPUT:

- `n` – nonnegative integer
- `min` – a primary similarity class type

EXAMPLES:

If `min` is not specified, then the class of all matrix types of size `n` is constructed:

```
sage: M = SimilarityClassTypes(2)
sage: for tau in M:
....:     print(tau)
[[1, [1]], [1, [1]]]
[[1, [2]]]
[[1, [1, 1]]]
[[2, [1]]]
```

```
>>> from sage.all import *
>>> M = SimilarityClassTypes(Integer(2))
>>> for tau in M:
...     print(tau)
[[1, [1]], [1, [1]]]
[[1, [2]]]
[[1, [1, 1]]]
[[2, [1]]]
```

If `min` is specified, then the class consists of only those similarity class types which are multisets of primary matrix types which either have size greater than that of `min`, or if they have size equal to that of `min`, then they occur after `min` in the iterator for `PrimarySimilarityClassTypes(n)`, where `n` is the size of `min`:

```
sage: M = SimilarityClassTypes(2, min = [1, [1, 1]])
sage: for tau in M:
....:     print(tau)
[[1, [1, 1]]]
[[2, [1]]]
```

```
>>> from sage.all import *
>>> M = SimilarityClassTypes(Integer(2), min = [Integer(1), [Integer(1),
→Integer(1)]])
>>> for tau in M:
...     print(tau)
[[1, [1, 1]]]
[[2, [1]]]
```

Element

alias of *SimilarityClassType*

size()

Return size of `self`.

EXAMPLES:

```
sage: tau = SimilarityClassType([[3, [3, 2, 1]], [2, [2, 1]]])
sage: tau.parent().size()
24
```

```
>>> from sage.all import *
>>> tau = SimilarityClassType([[Integer(3), [Integer(3), Integer(2),
→Integer(1)]], [Integer(2), [Integer(2), Integer(1)]]])
>>> tau.parent().size()
24
```

sum(*stat*, *sumover='matrices'*, *invertible=False*, *q=None*)

Return the sum of a local statistic over all types.

Given a set of functions $n_\lambda(q)$ (these could be polynomials or rational functions in q, for each similarity class type τ define

$$n_\tau(q) = \prod_{(d,\lambda)\in\tau} n_\lambda(q^d).$$

This function returns

$$\sum n_{\tau(g)}(q)$$

where $\tau(g)$ denotes the type of a matrix g, and the sum is over all $n \times n$ matrices if `sumover` is set to `'matrices'`, is over all $n \times n$ similarity classes if `sumover` is set to `'classes'`, and over all $n \times n$ types if `sumover` is set to `'types'`. If `invertible` is set to `True`, then the sum is only over invertible matrices or classes.

INPUT:

- `stat` – a function which takes partitions and returns a function of `q`
- `sumover` – can be one of the following:
  - `'matrices'`
  - `'classes'`
  - `'types'`
- `q` – integer or an indeterminate

OUTPUT: a function of `q`

EXAMPLES:

```
sage: M = SimilarityClassTypes(2)
sage: M.sum(lambda la:1)
q^4
sage: M.sum(lambda la:1, invertible = True)
q^4 - q^3 - q^2 + q
sage: M.sum(lambda la:1, sumover = "classes")
q^2 + q
sage: M.sum(lambda la:1, sumover = "classes", invertible = True)
q^2 - 1
```

```
>>> from sage.all import *
>>> M = SimilarityClassTypes(Integer(2))
>>> M.sum(lambda la:Integer(1))
q^4
>>> M.sum(lambda la:Integer(1), invertible = True)
q^4 - q^3 - q^2 + q
>>> M.sum(lambda la:Integer(1), sumover = "classes")
q^2 + q
>>> M.sum(lambda la:Integer(1), sumover = "classes", invertible = True)
q^2 - 1
```

Burside's lemma can be used to calculate the number of similarity classes of matrices:

```
sage: from sage.combinat.similarity_class_type import centralizer_algebra_dim,
↪ order_of_general_linear_group
sage: q = ZZ['q'].gen()
sage: M.sum(lambda la:q**centralizer_algebra_dim(la), invertible = True)/
↪order_of_general_linear_group(2)
q^2 + q
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import centralizer_algebra_dim,␣
↪order_of_general_linear_group
>>> q = ZZ['q'].gen()
>>> M.sum(lambda la:q**centralizer_algebra_dim(la), invertible = True)/order_
↪of_general_linear_group(Integer(2))
q^2 + q
```

sage.combinat.similarity_class_type.**centralizer_algebra_dim**()

Return the dimension of the centralizer algebra in $M_n(\mathbf{F}_q)$ of a nilpotent matrix whose Jordan blocks are given by `la`.

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import centralizer_algebra_dim
sage: centralizer_algebra_dim(Partition([2, 1]))
5
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import centralizer_algebra_dim
>>> centralizer_algebra_dim(Partition([Integer(2), Integer(1)]))
5
```

Note

If it is a list, `la` is expected to be sorted in decreasing order.

sage.combinat.similarity_class_type.**centralizer_group_cardinality**(*q=None*)

Return the cardinality of the centralizer group in $GL_n(\mathbf{F}_q)$ of a nilpotent matrix whose Jordan blocks are given by `la`.

INPUT:

- `lambda` – a partition
- `q` – an integer or an indeterminate

OUTPUT: a polynomial function of `q`

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import centralizer_group_
↪cardinality
sage: q = ZZ['q'].gen()
sage: centralizer_group_cardinality(Partition([2, 1]))
q^5 - 2*q^4 + q^3
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import centralizer_group_cardinality
>>> q = ZZ['q'].gen()
>>> centralizer_group_cardinality(Partition([Integer(2), Integer(1)]))
q^5 - 2*q^4 + q^3
```

sage.combinat.similarity_class_type.**dictionary_from_generator**(*gen*)

Given a generator for a list of pairs (c, f), construct a dictionary whose keys are the distinct values for c and whose value at c is the sum of f over all pairs of the form (c', f) such that $c = c'$.

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import dictionary_from_generator
sage: dictionary_from_generator(((x // 2, x) for x in range(10)))
{0: 1, 1: 5, 2: 9, 3: 13, 4: 17}
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import dictionary_from_generator
>>> dictionary_from_generator(((x // Integer(2), x) for x in range(Integer(10))))
{0: 1, 1: 5, 2: 9, 3: 13, 4: 17}
```

It also works with lists:

```
sage: dictionary_from_generator([(x // 2, x) for x in range(10)])
{0: 1, 1: 5, 2: 9, 3: 13, 4: 17}
```

```
>>> from sage.all import *
>>> dictionary_from_generator([(x // Integer(2), x) for x in range(Integer(10))])
{0: 1, 1: 5, 2: 9, 3: 13, 4: 17}
```

Note

Since the generator is first converted to a list, memory usage could be high.

sage.combinat.similarity_class_type.**ext_orbit_centralizers**(*input_data*, *q=None*, *selftranspose=False*)

Generate pairs consisting of centralizer cardinalities of orbits in $\mathrm{Ext}^1(M, M)$ for the action of $\mathrm{Aut}(M, M)$, where M is the $\mathbf{F}_{q[t]}$-module constructed from `input` and their frequencies.

INPUT:

- `input_data` – input for *input_parsing()*
- `q` – (default: q) an integer or an indeterminate
- `selftranspose` – boolean (default: `False`); stating if we only want selftranspose type

sage.combinat.similarity_class_type.**ext_orbits**(*input_data*, *q=None*, *selftranspose=False*)

Return the number of orbits in $\mathrm{Ext}^1(M, M)$ for the action of $\mathrm{Aut}(M, M)$, where M is the $\mathbf{F}_{q[t]}$-module constructed from `input_data`.

INPUT:

- `input_data` – input for *input_parsing()*
- `q` – (default: q) an integer or an indeterminate
- `selftranspose` – boolean (default: `False`); stating if we only want selftranspose type

sage.combinat.similarity_class_type.**fq**(*q=None*)

Return $(1 - q^{-1})(1 - q^{-2}) \cdots (1 - q^{-n})$.

INPUT:

- `n` – nonnegative integer
- `q` – integer or an indeterminate

OUTPUT: a rational function in `q`

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import fq
sage: fq(0)
1
sage: fq(3)
(q^6 - q^5 - q^4 + q^2 + q - 1)/q^6
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import fq
>>> fq(Integer(0))
1
>>> fq(Integer(3))
(q^6 - q^5 - q^4 + q^2 + q - 1)/q^6
```

`sage.combinat.similarity_class_type.`**`input_parsing`**(*data*)

Recognize and return the intended type of `input`.

`sage.combinat.similarity_class_type.`**`invariant_subspace_generating_function`**(*la*, *q=None*, *t=None*)

Return the invariant subspace generating function of a nilpotent matrix with Jordan block sizes given by `la`.

INPUT:

- `la` – a partition
- `q` – (optional) an integer or an inderminate
- `t` – (optional) an indeterminate

OUTPUT: a polynomial in `t` whose coefficients are polynomials in `q`

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import invariant_subspace_
↪generating_function
sage: invariant_subspace_generating_function([2,2])
t^4 + (q + 1)*t^3 + (q^2 + q + 1)*t^2 + (q + 1)*t + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import invariant_subspace_generating_
↪function
>>> invariant_subspace_generating_function([Integer(2),Integer(2)])
t^4 + (q + 1)*t^3 + (q^2 + q + 1)*t^2 + (q + 1)*t + 1
```

`sage.combinat.similarity_class_type.`**`matrix_centralizer_cardinalities`**(*n*, *q=None*, *invertible=False*)

Generate pairs consisting of centralizer cardinalities of matrices over a finite field and their frequencies.

sage.combinat.similarity_class_type.**matrix_centralizer_cardinalities_length_two**(*n*, *q=None*, *selftranspose=False*, *invertible=False*)

Generate pairs consisting of centralizer cardinalities of matrices over a principal ideal local ring of length two with residue field of order q and their frequencies.

INPUT:

- n – the order
- q – (default: q) an integer or an indeterminate
- selftranspose – boolean (default: False); stating if we only want selftranspose type
- invertible – boolean (default: False); stating if we only want invertible type

sage.combinat.similarity_class_type.**matrix_similarity_classes**(*n*, *q=None*, *invertible=False*)

Return the number of matrix similarity classes over a finite field of order q.

sage.combinat.similarity_class_type.**matrix_similarity_classes_length_two**(*n*, *q=None*, *selftranspose=False*, *invertible=False*)

Return the number of similarity classes of matrices of order n with entries in a principal ideal local ring of length two.

INPUT:

- n – the order
- q – (default: q) an integer or an indeterminate
- selftranspose – boolean (default: False); stating if we only want selftranspose type
- invertible – boolean (default: False); stating if we only want invertible type

EXAMPLES:

We can generate Table 6 of [PSS13]:

```
sage: from sage.combinat.similarity_class_type import matrix_similarity_classes_
↪length_two
sage: matrix_similarity_classes_length_two(2)
q^4 + q^3 + q^2
sage: matrix_similarity_classes_length_two(2, invertible = True)
q^4 - q
sage: matrix_similarity_classes_length_two(3)
q^6 + q^5 + 2*q^4 + q^3 + 2*q^2
sage: matrix_similarity_classes_length_two(3, invertible = true)
q^6 - q^3 + 2*q^2 - 2*q
sage: matrix_similarity_classes_length_two(4)
q^8 + q^7 + 3*q^6 + 3*q^5 + 5*q^4 + 3*q^3 + 3*q^2
sage: matrix_similarity_classes_length_two(4, invertible = True)
q^8 + q^6 - q^5 + 2*q^4 - 2*q^3 + 2*q^2 - 3*q
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import matrix_similarity_classes_
```

(continues on next page)

(continued from previous page)

```
↪length_two
>>> matrix_similarity_classes_length_two(Integer(2))
q^4 + q^3 + q^2
>>> matrix_similarity_classes_length_two(Integer(2), invertible = True)
q^4 - q
>>> matrix_similarity_classes_length_two(Integer(3))
q^6 + q^5 + 2*q^4 + q^3 + 2*q^2
>>> matrix_similarity_classes_length_two(Integer(3), invertible = true)
q^6 - q^3 + 2*q^2 - 2*q
>>> matrix_similarity_classes_length_two(Integer(4))
q^8 + q^7 + 3*q^6 + 3*q^5 + 5*q^4 + 3*q^3 + 3*q^2
>>> matrix_similarity_classes_length_two(Integer(4), invertible = True)
q^8 + q^6 - q^5 + 2*q^4 - 2*q^3 + 2*q^2 - 3*q
```

And also Table 7:

```
sage: matrix_similarity_classes_length_two(2, selftranspose = True)
q^4 + q^3 + q^2
sage: matrix_similarity_classes_length_two(2, selftranspose = True, invertible =␣
↪True)
q^4 - q
sage: matrix_similarity_classes_length_two(3, selftranspose = True)
q^6 + q^5 + 2*q^4 + q^3
sage: matrix_similarity_classes_length_two(3, selftranspose = True, invertible =␣
↪True)
q^6 - q^3
sage: matrix_similarity_classes_length_two(4, selftranspose = True)
q^8 + q^7 + 3*q^6 + 3*q^5 + 3*q^4 + q^3 + q^2
sage: matrix_similarity_classes_length_two(4, selftranspose = True, invertible =␣
↪True)
q^8 + q^6 - q^5 - q
```

```
>>> from sage.all import *
>>> matrix_similarity_classes_length_two(Integer(2), selftranspose = True)
q^4 + q^3 + q^2
>>> matrix_similarity_classes_length_two(Integer(2), selftranspose = True,␣
↪invertible = True)
q^4 - q
>>> matrix_similarity_classes_length_two(Integer(3), selftranspose = True)
q^6 + q^5 + 2*q^4 + q^3
>>> matrix_similarity_classes_length_two(Integer(3), selftranspose = True,␣
↪invertible = True)
q^6 - q^3
>>> matrix_similarity_classes_length_two(Integer(4), selftranspose = True)
q^8 + q^7 + 3*q^6 + 3*q^5 + 3*q^4 + q^3 + q^2
>>> matrix_similarity_classes_length_two(Integer(4), selftranspose = True,␣
↪invertible = True)
q^8 + q^6 - q^5 - q
```

sage.combinat.similarity_class_type.**order_of_general_linear_group**(*q=None*)

Return the cardinality of the group of $n \times n$ invertible matrices with entries in a field of order q.

INPUT:

- n – nonnegative integer
- q – integer or an indeterminate

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import order_of_general_linear_
↪group
sage: order_of_general_linear_group(0)
1
sage: order_of_general_linear_group(2)
q^4 - q^3 - q^2 + q
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import order_of_general_linear_group
>>> order_of_general_linear_group(Integer(0))
1
>>> order_of_general_linear_group(Integer(2))
q^4 - q^3 - q^2 + q
```

sage.combinat.similarity_class_type.**primitives**(*invertible=False*, *q=None*)

Return the number of similarity classes of simple matrices of order n with entries in a finite field of order q. This is the same as the number of irreducible polynomials of degree d.

If invertible is True, then only the number of similarity classes of invertible matrices is returned.

Note

All primitive classes are invertible unless n is 1.

INPUT:

- n – positive integer
- invertible – boolean; if set, only number of nonzero classes is returned
- q – integer or an indeterminate

OUTPUT: a rational function of the variable q

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import primitives
sage: primitives(1)
q
sage: primitives(1, invertible = True)
q - 1
sage: primitives(4)
1/4*q^4 - 1/4*q^2
sage: primitives(4, invertible = True)
1/4*q^4 - 1/4*q^2
```

```
>>> from sage.all import *
>>> from sage.combinat.similarity_class_type import primitives
>>> primitives(Integer(1))
q
```

(continues on next page)

(continued from previous page)

```
>>> primitives(Integer(1), invertible = True)
q - 1
>>> primitives(Integer(4))
1/4*q^4 - 1/4*q^2
>>> primitives(Integer(4), invertible = True)
1/4*q^4 - 1/4*q^2
```

5.1.310 sine-Gordon Y-system plotter

This class builds the triangulations associated to sine-Gordon and reduced sine-Gordon Y-systems as constructed in [NS].

AUTHORS:

- Salvatore Stella (2014-07-18): initial version

EXAMPLES:

A reduced sine-Gordon example with 3 generations:

```
sage: Y = SineGordonYsystem('A',(6,4,3)); Y
A sine-Gordon Y-system of type A with defining integer tuple (6, 4, 3)
sage: Y.plot()      #not tested
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3))); Y
A sine-Gordon Y-system of type A with defining integer tuple (6, 4, 3)
>>> Y.plot()      #not tested
```

The same integer tuple but for the non-reduced case:

```
sage: Y = SineGordonYsystem('D',(6,4,3)); Y
A sine-Gordon Y-system of type D with defining integer tuple (6, 4, 3)
sage: Y.plot()      #not tested
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('D',(Integer(6),Integer(4),Integer(3))); Y
A sine-Gordon Y-system of type D with defining integer tuple (6, 4, 3)
>>> Y.plot()      #not tested
```

Todo

The code for plotting is extremely slow.

REFERENCES:

class sage.combinat.sine_gordon.**SineGordonYsystem**(X, na)

Bases: SageObject

A class to model a (reduced) sine-Gordon Y-system.

Note that the generations, together with all integer tuples, in this implementation are numbered from 0 while in [NS] they are numbered from 1

INPUT:

- `X` – the type of the Y-system to construct (either 'A' or 'D')
- `na` – the tuple of positive integers defining the Y-system with `na[0] > 2`

See [NS]

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3)); Y
A sine-Gordon Y-system of type A with defining integer tuple (6, 4, 3)
sage: Y.intervals()
(((0, 0, 'R'),),
 ((0, 17, 'L'),
  (17, 34, 'L'),
...
  (104, 105, 'R'),
  (105, 0, 'R')))
sage: Y.triangulation()
((17, 89),
 (17, 72),
 (34, 72),
...
 (102, 105),
 (103, 105))
sage: Y.plot()      #not tested
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3))); Y
A sine-Gordon Y-system of type A with defining integer tuple (6, 4, 3)
>>> Y.intervals()
(((0, 0, 'R'),),
 ((0, 17, 'L'),
  (17, 34, 'L'),
...
  (104, 105, 'R'),
  (105, 0, 'R')))
>>> Y.triangulation()
((17, 89),
 (17, 72),
 (34, 72),
...
 (102, 105),
 (103, 105))
>>> Y.plot()      #not tested
```

F()

Return the number of generations in `self`.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.F()
3
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.F()
3
```

intervals()

Return, divided by generation, the list of intervals used to construct the initial triangulation.

Each such interval is a triple `(p, q, X)` where `p` and `q` are the two extremal vertices of the interval and `X` is the type of the interval (one of 'L', 'R', 'NL', 'NR').

ALGORITHM:

The algorithm used here is the one described in section 5.1 of [NS]. The only difference is that we get rid of the special case of the first generation by treating the whole disk as a type 'R' interval.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.intervals()
(((0, 0, 'R'),),
 ((0, 17, 'L'),
  (17, 34, 'L'),
...
  (104, 105, 'R'),
  (105, 0, 'R')))
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.intervals()
(((0, 0, 'R'),),
 ((0, 17, 'L'),
  (17, 34, 'L'),
...
  (104, 105, 'R'),
  (105, 0, 'R')))
```

na()

Return the sequence of the integers n_a defining `self`.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.na()
(6, 4, 3)
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.na()
(6, 4, 3)
```

pa()

Return the sequence of integers `p_a`, i.e. the total number of intervals of types 'NL' and 'NR' in the `(a+1)`-th generation.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.pa()
(1, 6, 25)
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.pa()
(1, 6, 25)
```

plot(***kwds*)

Plot the initial triangulation associated to `self`.

INPUT:

- `radius` – the radius of the disk; by default the length of the circle is the number of vertices
- `points_color` – the color of the vertices; default 'black'
- `points_size` – the size of the vertices; default 7
- `triangulation_color` – the color of the arcs; default 'black'
- `triangulation_thickness` – the thickness of the arcs; default 0.5
- `shading_color` – the color of the shading used on neuter intervals; default 'lightgray'
- `reflections_color` – the color of the reflection axes; default 'blue'
- `reflections_thickness` – the thickness of the reflection axes; default 1

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.plot()                          # long time (2s)                        #␣
↪needs sage.plot
Graphics object consisting of 219 graphics primitives
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.plot()                          # long time (2s)                        #␣
↪needs sage.plot
Graphics object consisting of 219 graphics primitives
```

qa()

Return the sequence of integers `q_a`, i.e. the total number of intervals of types 'L' and 'R' in the `(a+1)`-th generation.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.qa()
(6, 25, 81)
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.qa()
(6, 25, 81)
```

r()

Return the number of vertices in the polygon realizing self.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.r()
106
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.r()
106
```

rk()

Return the sequence of integers r^{(k)}, i.e. the width of an interval of type 'L' or 'R' in the k-th generation.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.rk()
(106, 17, 4)
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.rk()
(106, 17, 4)
```

triangulation()

Return the initial triangulation of the polygon realizing self as a tuple of pairs of vertices.

Warning

In type 'D' the returned triangulation does NOT contain the two radii.

ALGORITHM:

We implement the four cases described by Figure 14 in [NS].

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.triangulation()
((17, 89),
 (17, 72),
...
 (102, 105),
 (103, 105))
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.triangulation()
((17, 89),
```

(continues on next page)

(continued from previous page)

```
 (17, 72),
...
 (102, 105),
 (103, 105))
```

type()

Return the type of `self`.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.type()
'A'
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.type()
'A'
```

vertices()

Return the vertices of the polygon realizing `self` as the ring of integers modulo `self.r()`.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3))
sage: Y.vertices()
Ring of integers modulo 106
```

```
>>> from sage.all import *
>>> Y = SineGordonYsystem('A',(Integer(6),Integer(4),Integer(3)))
>>> Y.vertices()
Ring of integers modulo 106
```

5.1.311 Six Vertex Model

class sage.combinat.six_vertex_model.**SixVertexConfiguration**

Bases: `ClonableArray`

A configuration in the six vertex model.

check()

Check if `self` is a valid 6 vertex configuration.

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: M[0].check()
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(3), boundary_conditions='ice')
>>> M[Integer(0)].check()
```

energy(*epsilon*)

Return the energy of the configuration.

The energy of a configuration ν is defined as

$$E(\nu) = n_0\epsilon_0 + n_1\epsilon_1 + \cdots + n_5\epsilon_5$$

where n_i is the number of vertices of type i and ϵ_i is the i-th energy constant.

Note

We number our configurations as:

0. LR
1. LU
2. LD
3. UD
4. UR
5. RD

which differs from Wikipedia article Ice-type_model.

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: nu = M[2]; nu
    ^    ^    ^
    |    |    |
--> # -> # <- # <--
    ^    |    ^
    |    V    |
--> # <- # -> # <--
    |    ^    |
    V    |    V
--> # -> # <- # <--
    |    |    |
    V    V    V
sage: nu.energy([1,2,1,2,1,2])
15
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(3), boundary_conditions='ice')
>>> nu = M[Integer(2)]; nu
    ^    ^    ^
    |    |    |
--> # -> # <- # <--
    ^    |    ^
    |    V    |
--> # <- # -> # <--
    |    ^    |
    V    |    V
```

(continues on next page)

(continued from previous page)

```
--> # -> # <- # <--
    |    |    |
    V    V    V
>>> nu.energy([Integer(1),Integer(2),Integer(1),Integer(2),Integer(1),
↪Integer(2)])
15
```

A KDP energy:

```
sage: nu.energy([1,1,0,1,0,1])
7
```

```
>>> from sage.all import *
>>> nu.energy([Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1)])
7
```

A Rys F energy:

```
sage: nu.energy([0,1,1,0,1,1])
4
```

```
>>> from sage.all import *
>>> nu.energy([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(1)])
4
```

The zero field assumption:

```
sage: nu.energy([1,2,3,1,3,2])
15
```

```
>>> from sage.all import *
>>> nu.energy([Integer(1),Integer(2),Integer(3),Integer(1),Integer(3),
↪Integer(2)])
15
```

plot(*color='sign'*)

Return a plot of `self`.

INPUT:

- `color` – can be any of the following:
  - `4` – use 4 colors: black, red, blue, and green with each corresponding to up, right, down, and left respectively
  - `2` – use 2 colors: red for horizontal, blue for vertical arrows
  - `'sign'` – use red for right and down arrows, blue for left and up arrows
  - a list of 4 colors for each direction
  - a function which takes a direction and a boolean corresponding to the sign

EXAMPLES:

```
sage: M = SixVertexModel(2, boundary_conditions='ice')
sage: print(M[0].plot().description())                                  #␣
↪needs sage.plot
Arrow from (-1.0,0.0) to (0.0,0.0)
Arrow from (-1.0,1.0) to (0.0,1.0)
Arrow from (0.0,0.0) to (0.0,-1.0)
Arrow from (0.0,0.0) to (1.0,0.0)
Arrow from (0.0,1.0) to (0.0,0.0)
Arrow from (0.0,1.0) to (0.0,2.0)
Arrow from (1.0,0.0) to (1.0,-1.0)
Arrow from (1.0,0.0) to (1.0,1.0)
Arrow from (1.0,1.0) to (0.0,1.0)
Arrow from (1.0,1.0) to (1.0,2.0)
Arrow from (2.0,0.0) to (1.0,0.0)
Arrow from (2.0,1.0) to (1.0,1.0)
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(2), boundary_conditions='ice')
>>> print(M[Integer(0)].plot().description())                          ␣
↪     # needs sage.plot
Arrow from (-1.0,0.0) to (0.0,0.0)
Arrow from (-1.0,1.0) to (0.0,1.0)
Arrow from (0.0,0.0) to (0.0,-1.0)
Arrow from (0.0,0.0) to (1.0,0.0)
Arrow from (0.0,1.0) to (0.0,0.0)
Arrow from (0.0,1.0) to (0.0,2.0)
Arrow from (1.0,0.0) to (1.0,-1.0)
Arrow from (1.0,0.0) to (1.0,1.0)
Arrow from (1.0,1.0) to (0.0,1.0)
Arrow from (1.0,1.0) to (1.0,2.0)
Arrow from (2.0,0.0) to (1.0,0.0)
Arrow from (2.0,1.0) to (1.0,1.0)
```

to_signed_matrix()

Return the signed matrix of `self`.

The signed matrix corresponding to a six vertex configuration is given by 0 if there is a cross flow, a 1 if the outward arrows are vertical and -1 if the outward arrows are horizontal.

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: [x.to_signed_matrix() for x in M]                                 #␣
↪needs sage.modules
[
[1 0 0]  [1 0 0]  [ 0  1  0]  [0 1 0]  [0 1 0]  [0 0 1]  [0 0 1]
[0 1 0]  [0 0 1]  [ 1 -1  1]  [1 0 0]  [0 0 1]  [1 0 0]  [0 1 0]
[0 0 1], [0 1 0], [ 0  1  0], [0 0 1], [1 0 0], [0 1 0], [1 0 0]
]
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(3), boundary_conditions='ice')
>>> [x.to_signed_matrix() for x in M]                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
[
[1 0 0]  [1 0 0]  [ 0  1  0]  [0 1 0]  [0 1 0]  [0 0 1]  [0 0 1]
[0 1 0]  [0 0 1]  [ 1 -1  1]  [1 0 0]  [0 0 1]  [1 0 0]  [0 1 0]
[0 0 1], [0 1 0], [ 0  1  0], [0 0 1], [1 0 0], [0 1 0], [1 0 0]
]
```

class sage.combinat.six_vertex_model.**SixVertexModel**(*n*, *m*, *boundary_conditions*)

Bases: `UniqueRepresentation`, `Parent`

The six vertex model.

We model a configuration by indicating which configuration by the following six configurations which are determined by the two outgoing arrows in the Up, Right, Down, Left directions:

1. LR:

```
     |
     V
<-- # -->
     ^
     |
```

2. LU:

```
     ^
     |
<-- # <--
     ^
     |
```

3. LD:

```
     |
     V
<-- # <--
     |
     V
```

4. UD:

```
     ^
     |
--> # <--
     |
     V
```

5. UR:

```
     ^
     |
--> # -->
     ^
     |
```

6. RD:

```
     |
     V
--> # -->
     |
     V
```

INPUT:

- n – the number of rows
- m – (optional) the number of columns, if not specified, then the number of columns is the number of rows
- boundary_conditions – (optional) a quadruple of tuples whose entries are either:
  - True for an inward arrow,
  - False for an outward arrow, or
  - None for no boundary condition.

  There are also the following predefined boundary conditions:
  - 'ice' – the top and bottom boundary conditions are outward and the left and right boundary conditions are inward; this gives the square ice model. Also called domain wall boundary conditions.
  - 'domain wall' – same as 'ice'.
  - 'alternating' – the boundary conditions alternate between inward and outward.
  - 'free' – there are no boundary conditions.

EXAMPLES:

Here are the six types of vertices that can be created:

```
sage: M = SixVertexModel(1)
sage: list(M)
[
    |             ^             |             ^             ^             |
    V             |             V             |             |             V
<-- # -->     <-- # <--     <-- # <--     --> # <--     --> # -->     --> # -->
    ^             ^             |             |             ^             |
    |     ,       |     ,       V     ,       V     ,       |     ,       V
]
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(1))
>>> list(M)
[
    |             ^             |             ^             ^             |
    V             |             V             |             |             V
<-- # -->     <-- # <--     <-- # <--     --> # <--     --> # -->     --> # -->
    ^             ^             |             |             ^             |
    |     ,       |     ,       V     ,       V     ,       |     ,       V
]
```

When using the square ice model, it is known that the number of configurations is equal to the number of alternating sign matrices:

```
sage: M = SixVertexModel(1, boundary_conditions='ice')
sage: len(M)
1
sage: M = SixVertexModel(4, boundary_conditions='ice')
sage: len(M)
42
sage: all(len(SixVertexModel(n, boundary_conditions='ice'))                  #␣
↪needs sage.modules
....:     == AlternatingSignMatrices(n).cardinality() for n in range(1, 7))
True
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(1), boundary_conditions='ice')
>>> len(M)
1
>>> M = SixVertexModel(Integer(4), boundary_conditions='ice')
>>> len(M)
42
>>> all(len(SixVertexModel(n, boundary_conditions='ice'))                  #␣
↪needs sage.modules
...     == AlternatingSignMatrices(n).cardinality() for n in range(Integer(1),␣
↪Integer(7)))
True
```

An example with a specified non-standard boundary condition and non-rectangular shape:

```
sage: M = SixVertexModel(2, 1, [[None], [True,True], [None], [None,None]])
sage: list(M)
[
    ^           ^           |           ^
    |           |           V           |
<-- # <--   <-- # <--   <-- # <--   --> # <--
    ^           ^           |           |
    |           |           V           V
<-- # <--   --> # <--   <-- # <--   <-- # <--
    ^           |           |           |
    |     ,     V     ,     V     ,     V
]
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(2), Integer(1), [[None], [True,True], [None],␣
↪[None,None]])
>>> list(M)
[
    ^           ^           |           ^
    |           |           V           |
<-- # <--   <-- # <--   <-- # <--   --> # <--
    ^           ^           |           |
    |           |           V           V
<-- # <--   --> # <--   <-- # <--   <-- # <--
    ^           |           |           |
    |     ,     V     ,     V     ,     V
]
```

REFERENCES:

- Wikipedia article Vertex_model
- Wikipedia article Ice-type_model

Element

alias of *SixVertexConfiguration*

boundary_conditions()

Return the boundary conditions of `self`.

EXAMPLES:

```
sage: M = SixVertexModel(2, boundary_conditions='ice')
sage: M.boundary_conditions()
((False, False), (True, True), (False, False), (True, True))
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(2), boundary_conditions='ice')
>>> M.boundary_conditions()
((False, False), (True, True), (False, False), (True, True))
```

partition_function(*beta*, *epsilon*)

Return the partition function of `self`.

The partition function of a 6 vertex model is defined by:

$$Z = \sum_{\nu} e^{-\beta E(\nu)}$$

where we sum over all configurations and E is the energy function. The constant β is known as the *inverse temperature* and is equal to $1/k_B T$ where k_B is Boltzmann's constant and T is the system's temperature.

INPUT:

- `beta` – the inverse temperature constant β
- `epsilon` – the energy constants, see *energy()*

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: M.partition_function(2, [1,2,1,2,1,2])                                    #␣
↪needs sage.symbolic
e^(-24) + 2*e^(-28) + e^(-30) + 2*e^(-32) + e^(-36)
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(3), boundary_conditions='ice')
>>> M.partition_function(Integer(2), [Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(1),Integer(2)])                                # needs␣
↪sage.symbolic
e^(-24) + 2*e^(-28) + e^(-30) + 2*e^(-32) + e^(-36)
```

REFERENCES:

Wikipedia article Partition_function_(statistical_mechanics)

class sage.combinat.six_vertex_model.**SquareIceModel**(*n*)

Bases: *SixVertexModel*

The square ice model.

The square ice model is a 6 vertex model on an $n \times n$ grid with the boundary conditions that the top and bottom boundaries are pointing outward and the left and right boundaries are pointing inward. These boundary conditions are also called domain wall boundary conditions.

Configurations of the 6 vertex model with domain wall boundary conditions are in bijection with alternating sign matrices.

class Element

Bases: *SixVertexConfiguration*

An element in the square ice model.

to_alternating_sign_matrix()

Return an alternating sign matrix of `self`.

See also

to_signed_matrix()

EXAMPLES:

```
sage: M = SixVertexModel(4, boundary_conditions='ice')
sage: M[6].to_alternating_sign_matrix()                                  #␣
↪needs sage.modules
[1 0 0 0]
[0 0 0 1]
[0 0 1 0]
[0 1 0 0]
sage: M[7].to_alternating_sign_matrix()                                  #␣
↪needs sage.modules
[ 0  1  0  0]
[ 1 -1  1  0]
[ 0  1 -1  1]
[ 0  0  1  0]
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(4), boundary_conditions='ice')
>>> M[Integer(6)].to_alternating_sign_matrix()                             ␣
↪     # needs sage.modules
[1 0 0 0]
[0 0 0 1]
[0 0 1 0]
[0 1 0 0]
>>> M[Integer(7)].to_alternating_sign_matrix()                             ␣
↪     # needs sage.modules
[ 0  1  0  0]
[ 1 -1  1  0]
[ 0  1 -1  1]
[ 0  0  1  0]
```

from_alternating_sign_matrix(*asm*)

Return a configuration from the alternating sign matrix asm.

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: asm = AlternatingSignMatrix([[0,1,0],[1,-1,1],[0,1,0]])                #␣
↪needs sage.modules
sage: M.from_alternating_sign_matrix(asm)                                    #␣
↪needs sage.modules
    ^    ^    ^
    |    |    |
--> # -> # <- # <--
    ^    |    ^
    |    V    |
--> # <- # -> # <--
    |    ^    |
    V    |    V
--> # -> # <- # <--
    |    |    |
    V    V    V
```

```
>>> from sage.all import *
>>> M = SixVertexModel(Integer(3), boundary_conditions='ice')
>>> asm = AlternatingSignMatrix([[Integer(0),Integer(1),Integer(0)],
↪[Integer(1),-Integer(1),Integer(1)],[Integer(0),Integer(1),Integer(0)]])   ␣
↪           # needs sage.modules
>>> M.from_alternating_sign_matrix(asm)                                    #␣
↪needs sage.modules
    ^    ^    ^
    |    |    |
--> # -> # <- # <--
    ^    |    ^
    |    V    |
--> # <- # -> # <--
    |    ^    |
    V    |    V
--> # -> # <- # <--
    |    |    |
    V    V    V
```

5.1.312 Skew Partitions

A skew partition skp of size n is a pair of partitions $[p_1, p_2]$ where p_1 is a partition of the integer n_1, p_2 is a partition of the integer n_2, p_2 is an inner partition of p_1, and $n = n_1 - n_2$. We say that p_1 and p_2 are respectively the *inner* and *outer* partitions of skp.

A skew partition can be depicted by a diagram made of rows of cells, in the same way as a partition. Only the cells of the outer partition p_1 which are not in the inner partition p_2 appear in the picture. For example, this is the diagram of the skew partition [[5,4,3,1],[3,3,1]].

```
sage: print(SkewPartition([[5,4,3,1],[3,3,1]]).diagram())
   **
   *
```

(continues on next page)

(continued from previous page)

```
 **
*
```

```
>>> from sage.all import *
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(3),Integer(1)]]).diagram())
   **
   *
 **
*
```

A skew partition can be *connected*, which can easily be described in graphic terms: for each pair of consecutive rows, there are at least two cells (one in each row) which have a common edge. This is the diagram of the connected skew partition `[[5,4,3,1],[3,1]]`:

```
sage: print(SkewPartition([[5,4,3,1],[3,1]]).diagram())
   **
 ***
***
*
sage: SkewPartition([[5,4,3,1],[3,1]]).is_connected()
True
```

```
>>> from sage.all import *
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(1)]]).diagram())
   **
 ***
***
*
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(1)]]).is_connected()
True
```

The first example of a skew partition is not a connected one.

Applying a reflection with respect to the main diagonal yields the diagram of the *conjugate skew partition*, here `[[4,3,3,2,1],[3,2,2]]`:

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).conjugate()
[4, 3, 3, 2, 1] / [3, 2, 2]
sage: print(SkewPartition([[5,4,3,1],[3,3,1]]).conjugate().diagram())
   *
  *
  *
**
*
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(3),Integer(1)]]).conjugate()
[4, 3, 3, 2, 1] / [3, 2, 2]
```

(continues on next page)

(continued from previous page)

```
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(3),Integer(1)]]).conjugate().diagram())
   *
  *
  *
**
*
```

The *outer corners* of a skew partition are the corners of its outer partition. The *inner corners* are the internal corners of the outer partition when the inner partition is taken off. Shown below are the coordinates of the inner and outer corners.

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).outer_corners()
[(0, 4), (1, 3), (2, 2), (3, 0)]
sage: SkewPartition([[5,4,3,1],[3,3,1]]).inner_corners()
[(0, 3), (2, 1), (3, 0)]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(3),Integer(1)]]).outer_corners()
[(0, 4), (1, 3), (2, 2), (3, 0)]
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(3),Integer(1)]]).inner_corners()
[(0, 3), (2, 1), (3, 0)]
```

EXAMPLES:

There are 9 skew partitions of size 3, with no empty row nor empty column:

```
sage: SkewPartitions(3).cardinality()
9
sage: SkewPartitions(3).list()
[[3] / [],
 [2, 1] / [],
 [3, 1] / [1],
 [2, 2] / [1],
 [3, 2] / [2],
 [1, 1, 1] / [],
 [2, 2, 1] / [1, 1],
 [2, 1, 1] / [1],
 [3, 2, 1] / [2, 1]]
```

```
>>> from sage.all import *
>>> SkewPartitions(Integer(3)).cardinality()
9
>>> SkewPartitions(Integer(3)).list()
[[3] / [],
 [2, 1] / [],
 [3, 1] / [1],
 [2, 2] / [1],
 [3, 2] / [2],
 [1, 1, 1] / [],
 [2, 2, 1] / [1, 1],
```

(continues on next page)

(continued from previous page)

```
[2, 1, 1] / [1],
[3, 2, 1] / [2, 1]]
```

There are 4 connected skew partitions of size 3:

```
sage: SkewPartitions(3, overlap=1).cardinality()
4
sage: SkewPartitions(3, overlap=1).list()
[[3] / [], [2, 1] / [], [2, 2] / [1], [1, 1, 1] / []]
```

```
>>> from sage.all import *
>>> SkewPartitions(Integer(3), overlap=Integer(1)).cardinality()
4
>>> SkewPartitions(Integer(3), overlap=Integer(1)).list()
[[3] / [], [2, 1] / [], [2, 2] / [1], [1, 1, 1] / []]
```

This is the conjugate of the skew partition `[[4,3,1], [2]]`

```
sage: SkewPartition([[4,3,1], [2]]).conjugate()
[3, 2, 2, 1] / [1, 1]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(4),Integer(3),Integer(1)], [Integer(2)]]).conjugate()
[3, 2, 2, 1] / [1, 1]
```

Geometrically, we just applied a reflection with respect to the main diagonal on the diagram of the partition. Of course, this operation is an involution:

```
sage: SkewPartition([[4,3,1],[2]]).conjugate().conjugate()
[4, 3, 1] / [2]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(4),Integer(3),Integer(1)],[Integer(2)]]).conjugate().
↪conjugate()
[4, 3, 1] / [2]
```

The `jacobi_trudi()` method computes the Jacobi-Trudi matrix. See [Mac1995] for a definition and discussion.

```
sage: SkewPartition([[4,3,1],[2]]).jacobi_trudi()                          #␣
↪needs sage.modules
[h[2]  h[]    0]
[h[5] h[3]  h[]]
[h[6] h[4] h[1]]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(4),Integer(3),Integer(1)],[Integer(2)]]).jacobi_trudi()  ␣
↪                                  # needs sage.modules
[h[2]  h[]    0]
[h[5] h[3]  h[]]
[h[6] h[4] h[1]]
```

This example shows how to compute the corners of a skew partition.

```
sage: SkewPartition([[4,3,1],[2]]).inner_corners()
[(0, 2), (1, 0)]
sage: SkewPartition([[4,3,1],[2]]).outer_corners()
[(0, 3), (1, 2), (2, 0)]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(4),Integer(3),Integer(1)],[Integer(2)]]).inner_corners()
[(0, 2), (1, 0)]
>>> SkewPartition([[Integer(4),Integer(3),Integer(1)],[Integer(2)]]).outer_corners()
[(0, 3), (1, 2), (2, 0)]
```

AUTHORS:

- Mike Hansen: Initial version
- Travis Scrimshaw (2013-02-11): Factored out `CombinatorialClass`
- Trevor K. Karn (2022-08-03): Add `outside_corners`

class `sage.combinat.skew_partition.`**`SkewPartition`**(*parent*, *skp*)

Bases: *CombinatorialElement*

A skew partition.

A skew partition of shape λ/μ is the Young diagram from the partition λ and removing the partition μ from the upper-left corner in English convention.

`cell_poset`(*orientation='SE'*)

Return the Young diagram of `self` as a poset. The optional keyword variable `orientation` determines the order relation of the poset.

The poset always uses the set of cells of the Young diagram of `self` as its ground set. The order relation of the poset depends on the `orientation` variable (which defaults to `'SE'`). Concretely, `orientation` has to be specified to one of the strings `'NW'`, `'NE'`, `'SW'`, and `'SE'`, standing for "northwest", "northeast", "southwest" and "southeast", respectively. If `orientation` is `'SE'`, then the order relation of the poset is such that a cell u is greater or equal to a cell v in the poset if and only if u lies weakly southeast of v (this means that u can be reached from v by a sequence of south and east steps; the sequence is allowed to consist of south steps only, or of east steps only, or even be empty). Similarly the order relation is defined for the other three orientations. The Young diagram is supposed to be drawn in English notation.

The elements of the poset are the cells of the Young diagram of `self`, written as tuples of zero-based coordinates (so that $(3, 7)$ stands for the 8-th cell of the 4-th row, etc.).

EXAMPLES:

```
sage: # needs sage.graphs
sage: p = SkewPartition([[3,3,1], [2,1]])
sage: Q = p.cell_poset(); Q
Finite poset containing 4 elements
sage: sorted(Q)
[(0, 2), (1, 1), (1, 2), (2, 0)]
sage: sorted(Q.maximal_elements())
[(1, 2), (2, 0)]
sage: sorted(Q.minimal_elements())
[(0, 2), (1, 1), (2, 0)]
sage: sorted(Q.upper_covers((1, 1)))
[(1, 2)]
```

(continues on next page)

(continued from previous page)

```
sage: sorted(Q.upper_covers((0, 2)))
[(1, 2)]

sage: # needs sage.graphs
sage: P = p.cell_poset(orientation="NW"); P
Finite poset containing 4 elements
sage: sorted(P)
[(0, 2), (1, 1), (1, 2), (2, 0)]
sage: sorted(P.minimal_elements())
[(1, 2), (2, 0)]
sage: sorted(P.maximal_elements())
[(0, 2), (1, 1), (2, 0)]
sage: sorted(P.upper_covers((1, 2)))
[(0, 2), (1, 1)]

sage: # needs sage.graphs
sage: R = p.cell_poset(orientation="NE"); R
Finite poset containing 4 elements
sage: sorted(R)
[(0, 2), (1, 1), (1, 2), (2, 0)]
sage: R.maximal_elements()
[(0, 2)]
sage: R.minimal_elements()
[(2, 0)]
sage: R.upper_covers((2, 0))
[(1, 1)]
sage: sorted([len(R.upper_covers(v)) for v in R])
[0, 1, 1, 1]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> p = SkewPartition([[Integer(3),Integer(3),Integer(1)], [Integer(2),
↪Integer(1)]])
>>> Q = p.cell_poset(); Q
Finite poset containing 4 elements
>>> sorted(Q)
[(0, 2), (1, 1), (1, 2), (2, 0)]
>>> sorted(Q.maximal_elements())
[(1, 2), (2, 0)]
>>> sorted(Q.minimal_elements())
[(0, 2), (1, 1), (2, 0)]
>>> sorted(Q.upper_covers((Integer(1), Integer(1))))
[(1, 2)]
>>> sorted(Q.upper_covers((Integer(0), Integer(2))))
[(1, 2)]

>>> # needs sage.graphs
>>> P = p.cell_poset(orientation="NW"); P
Finite poset containing 4 elements
>>> sorted(P)
[(0, 2), (1, 1), (1, 2), (2, 0)]
>>> sorted(P.minimal_elements())
```

(continues on next page)

(continued from previous page)

```
[(1, 2), (2, 0)]
>>> sorted(P.maximal_elements())
[(0, 2), (1, 1), (2, 0)]
>>> sorted(P.upper_covers((Integer(1), Integer(2))))
[(0, 2), (1, 1)]

>>> # needs sage.graphs
>>> R = p.cell_poset(orientation="NE"); R
Finite poset containing 4 elements
>>> sorted(R)
[(0, 2), (1, 1), (1, 2), (2, 0)]
>>> R.maximal_elements()
[(0, 2)]
>>> R.minimal_elements()
[(2, 0)]
>>> R.upper_covers((Integer(2), Integer(0)))
[(1, 1)]
>>> sorted([len(R.upper_covers(v)) for v in R])
[0, 1, 1, 1]
```

cells()

Return the coordinates of the cells of `self`. Coordinates are given as `(row-index, column-index)` and are 0 based.

EXAMPLES:

```
sage: SkewPartition([[4, 3, 1], [2]]).cells()
[(0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: SkewPartition([[4, 3, 1], []]).cells()
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: SkewPartition([[2], []]).cells()
[(0, 0), (0, 1)]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(4), Integer(3), Integer(1)], [Integer(2)]]).
→cells()
[(0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (2, 0)]
>>> SkewPartition([[Integer(4), Integer(3), Integer(1)], []]).cells()
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (2, 0)]
>>> SkewPartition([[Integer(2)], []]).cells()
[(0, 0), (0, 1)]
```

column_lengths()

Return the column lengths of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).column_lengths()
[1, 2, 1]
sage: SkewPartition([[5,2,2,2],[2,1]]).column_lengths()
[2, 3, 1, 1, 1]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(1),
↪Integer(1)]]).column_lengths()
[1, 2, 1]
>>> SkewPartition([[Integer(5),Integer(2),Integer(2),Integer(2)],[Integer(2),
↪Integer(1)]]).column_lengths()
[2, 3, 1, 1, 1]
```

columns_intersection_set()

Return the set of cells in the columns of the outer shape of `self` which columns intersect the skew diagram of `self`.

EXAMPLES:

```
sage: skp = SkewPartition([[3,2,1],[2,1]])
sage: cells = Set([ (0,0), (0, 1), (0,2), (1, 0), (1, 1), (2, 0)])
sage: skp.columns_intersection_set() == cells
True
```

```
>>> from sage.all import *
>>> skp = SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(1)]])
>>> cells = Set([ (Integer(0),Integer(0)), (Integer(0), Integer(1)),␣
↪(Integer(0),Integer(2)), (Integer(1), Integer(0)), (Integer(1), Integer(1)),
↪ (Integer(2), Integer(0))])
>>> skp.columns_intersection_set() == cells
True
```

conjugate()

Return the conjugate of the skew partition skp.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2]]).conjugate()
[3, 2, 1] / [1, 1]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2)]]).
↪conjugate()
[3, 2, 1] / [1, 1]
```

diagram()

Return the Ferrers diagram of `self`.

EXAMPLES:

```
sage: print(SkewPartition([[5,4,3,1],[3,3,1]]).ferrers_diagram())
   **
   *
 **
*
sage: print(SkewPartition([[5,4,3,1],[3,1]]).diagram())
   **
```

(continues on next page)

(continued from previous page)

```
 ***
***
*
sage: SkewPartitions.options(diagram_str='#', convention='French')
sage: print(SkewPartition([[5,4,3,1],[3,1]]).diagram())
#
###
 ###
   ##
sage: SkewPartitions.options._reset()
```

```
>>> from sage.all import *
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],
↪[Integer(3),Integer(3),Integer(1)]]).ferrers_diagram())
   **
   *
 **
*
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],
↪[Integer(3),Integer(1)]]).diagram())
   **
 ***
***
*
>>> SkewPartitions.options(diagram_str='#', convention='French')
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],
↪[Integer(3),Integer(1)]]).diagram())
#
###
 ###
   ##
>>> SkewPartitions.options._reset()
```

ferrers_diagram()

Return the Ferrers diagram of `self`.

EXAMPLES:

```
sage: print(SkewPartition([[5,4,3,1],[3,3,1]]).ferrers_diagram())
   **
   *
 **
*
sage: print(SkewPartition([[5,4,3,1],[3,1]]).diagram())
   **
 ***
***
*
sage: SkewPartitions.options(diagram_str='#', convention='French')
sage: print(SkewPartition([[5,4,3,1],[3,1]]).diagram())
#
###
```

(continues on next page)

(continued from previous page)

```
 ###
   ##
sage: SkewPartitions.options._reset()
```

```
>>> from sage.all import *
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],
↪[Integer(3),Integer(3),Integer(1)]]).ferrers_diagram())
   **
   *
 **
*
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],
↪[Integer(3),Integer(1)]]).diagram())
   **
 ***
***
*
>>> SkewPartitions.options(diagram_str='#', convention='French')
>>> print(SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],
↪[Integer(3),Integer(1)]]).diagram())
#
###
 ###
   ##
>>> SkewPartitions.options._reset()
```

frobenius_rank()

Return the Frobenius rank of the skew partition `self`.

The Frobenius rank of a skew partition λ/μ can be defined in various ways. The quickest one is probably the following: Writing λ as $(\lambda_1, \lambda_2, \cdots, \lambda_N)$, and writing μ as $(\mu_1, \mu_2, \cdots, \mu_N)$, we define the Frobenius rank of λ/μ to be the number of all $1 \leq i \leq N$ such that

$$\lambda_i - i \not\in \{\mu_1 - 1, \mu_2 - 2, \cdots, \mu_N - N\}.$$

In other words, the Frobenius rank of λ/μ is the number of rows in the Jacobi-Trudi matrix of λ/μ which don't contain h_0. Further definitions have been considered in [Sta2002] (where Frobenius rank is just being called rank).

If μ is the empty shape, then the Frobenius rank of λ/μ is just the usual Frobenius rank of the partition λ (see *frobenius_rank()*).

EXAMPLES:

```
sage: SkewPartition([[8,8,7,4], [4,1,1]]).frobenius_rank()
4
sage: SkewPartition([[2,1], [1]]).frobenius_rank()
2
sage: SkewPartition([[2,1,1], [1]]).frobenius_rank()
2
sage: SkewPartition([[2,1,1], [1,1]]).frobenius_rank()
2
sage: SkewPartition([[5,4,3,2], [2,1,1]]).frobenius_rank()
```

(continues on next page)

(continued from previous page)

```
3
sage: SkewPartition([[4,2,1], [3,1,1]]).frobenius_rank()
2
sage: SkewPartition([[4,2,1], [3,2,1]]).frobenius_rank()
1
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(8),Integer(8),Integer(7),Integer(4)], [Integer(4),
↪Integer(1),Integer(1)]]).frobenius_rank()
4
>>> SkewPartition([[Integer(2),Integer(1)], [Integer(1)]]).frobenius_rank()
2
>>> SkewPartition([[Integer(2),Integer(1),Integer(1)], [Integer(1)]]).
↪frobenius_rank()
2
>>> SkewPartition([[Integer(2),Integer(1),Integer(1)], [Integer(1),
↪Integer(1)]]).frobenius_rank()
2
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(2)], [Integer(2),
↪Integer(1),Integer(1)]]).frobenius_rank()
3
>>> SkewPartition([[Integer(4),Integer(2),Integer(1)], [Integer(3),Integer(1),
↪Integer(1)]]).frobenius_rank()
2
>>> SkewPartition([[Integer(4),Integer(2),Integer(1)], [Integer(3),Integer(2),
↪Integer(1)]]).frobenius_rank()
1
```

If the inner shape is empty, then the Frobenius rank of the skew partition is just the standard Frobenius rank of the partition:

```
sage: all( SkewPartition([lam, Partition([])]).frobenius_rank()
....:      == lam.frobenius_rank() for i in range(6)
....:      for lam in Partitions(i) )
True
```

```
>>> from sage.all import *
>>> all( SkewPartition([lam, Partition([])]).frobenius_rank()
...      == lam.frobenius_rank() for i in range(Integer(6))
...      for lam in Partitions(i) )
True
```

If the inner and outer shapes are equal, then the Frobenius rank is zero:

```
sage: all( SkewPartition([lam, lam]).frobenius_rank() == 0
....:      for i in range(6) for lam in Partitions(i) )
True
```

```
>>> from sage.all import *
>>> all( SkewPartition([lam, lam]).frobenius_rank() == Integer(0)
...      for i in range(Integer(6)) for lam in Partitions(i) )
True
```

inner()

Return the inner partition of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).inner()
[1, 1]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(1),
↪Integer(1)]]).inner()
[1, 1]
```

inner_corners()

Return a list of the inner corners of `self`.

EXAMPLES:

```
sage: SkewPartition([[4, 3, 1], [2]]).inner_corners()
[(0, 2), (1, 0)]
sage: SkewPartition([[4, 3, 1], []]).inner_corners()
[(0, 0)]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(4), Integer(3), Integer(1)], [Integer(2)]]).inner_
↪corners()
[(0, 2), (1, 0)]
>>> SkewPartition([[Integer(4), Integer(3), Integer(1)], []]).inner_corners()
[(0, 0)]
```

is_connected()

Return `True` if `self` is a connected skew partition.

A skew partition is said to be *connected* if for each pair of consecutive rows, there are at least two cells (one in each row) which have a common edge.

EXAMPLES:

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).is_connected()
False
sage: SkewPartition([[5,4,3,1],[3,1]]).is_connected()
True
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(3),Integer(1)]]).is_connected()
False
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(1)]]).is_connected()
True
```

is_overlap(*n*)

Return `True` if the overlap of `self` is at most `n`.

> **See also**
>
> *overlap()*

EXAMPLES:

```
sage: SkewPartition([[5,4,3,1],[3,1]]).is_overlap(1)
True
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(1)]]).is_overlap(Integer(1))
True
```

is_ribbon()

Return `True` if and only if `self` is a ribbon.

This means that if it has exactly one cell in each of q consecutive diagonals for some nonnegative integer q.

EXAMPLES:

```
sage: P = SkewPartition([[4,4,3,3],[3,2,2]])
sage: P.pp()
   *
  **
  *
***
sage: P.is_ribbon()
True

sage: P = SkewPartition([[4,3,3],[1,1]])
sage: P.pp()
 ***
 **
***
sage: P.is_ribbon()
False

sage: P = SkewPartition([[4,4,3,2],[3,2,2]])
sage: P.pp()
   *
  **
  *
**
sage: P.is_ribbon()
False

sage: P = SkewPartition([[4,4,3,3],[4,2,2,1]])
sage: P.pp()

  **
  *
 **
```

(continues on next page)

(continued from previous page)

```
sage: P.is_ribbon()
True

sage: P = SkewPartition([[4,4,3,3],[4,2,2]])
sage: P.pp()

  **
  *
***
sage: P.is_ribbon()
True

sage: SkewPartition([[2,2,1],[2,2,1]]).is_ribbon()
True
```

```
>>> from sage.all import *
>>> P = SkewPartition([[Integer(4),Integer(4),Integer(3),Integer(3)],
↪[Integer(3),Integer(2),Integer(2)]])
>>> P.pp()
   *
  **
  *
***
>>> P.is_ribbon()
True

>>> P = SkewPartition([[Integer(4),Integer(3),Integer(3)],[Integer(1),
↪Integer(1)]])
>>> P.pp()
 ***
 **
***
>>> P.is_ribbon()
False

>>> P = SkewPartition([[Integer(4),Integer(4),Integer(3),Integer(2)],
↪[Integer(3),Integer(2),Integer(2)]])
>>> P.pp()
   *
  **
  *
**
>>> P.is_ribbon()
False

>>> P = SkewPartition([[Integer(4),Integer(4),Integer(3),Integer(3)],
↪[Integer(4),Integer(2),Integer(2),Integer(1)]])
>>> P.pp()
<BLANKLINE>
  **
  *
 **
```

(continues on next page)

(continued from previous page)

```
>>> P.is_ribbon()
True

>>> P = SkewPartition([[Integer(4),Integer(4),Integer(3),Integer(3)],
↪[Integer(4),Integer(2),Integer(2)]])
>>> P.pp()
<BLANKLINE>
  **
  *
***
>>> P.is_ribbon()
True

>>> SkewPartition([[Integer(2),Integer(2),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).is_ribbon()
True
```

jacobi_trudi()

Return the Jacobi-Trudi matrix of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).jacobi_trudi()                          #␣
↪needs sage.modules
[h[1]    0    0]
[h[3] h[1]    0]
[h[5] h[3] h[1]]
sage: SkewPartition([[4,3,2],[2,1]]).jacobi_trudi()                          #␣
↪needs sage.modules
[h[2]  h[]    0]
[h[4] h[2]  h[]]
[h[6] h[4] h[2]]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(1)]]).jacobi_trudi()                          # needs sage.modules
[h[1]    0    0]
[h[3] h[1]    0]
[h[5] h[3] h[1]]
>>> SkewPartition([[Integer(4),Integer(3),Integer(2)],[Integer(2),
↪Integer(1)]]).jacobi_trudi()                          # needs sage.modules
[h[2]  h[]    0]
[h[4] h[2]  h[]]
[h[6] h[4] h[2]]
```

k_conjugate(k)

Return the k-conjugate of the skew partition.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(3)
[2, 1, 1, 1, 1] / [2, 1]
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(4)
```

(continues on next page)

(continued from previous page)

```
[2, 2, 1, 1] / [2, 1]
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(5)
[3, 2, 1] / [2, 1]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(1)]]).k_conjugate(Integer(3))
[2, 1, 1, 1, 1] / [2, 1]
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(1)]]).k_conjugate(Integer(4))
[2, 2, 1, 1] / [2, 1]
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(1)]]).k_conjugate(Integer(5))
[3, 2, 1] / [2, 1]
```

outer()

Return the outer partition of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).outer()
[3, 2, 1]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(1),
↪Integer(1)]]).outer()
[3, 2, 1]
```

outer_corners()

Return a list of the outer corners of `self`.

These are corners that are contained inside of the shape. For the corners which are outside of the shape, use *outside_corners()*.

Warning

In the case that `self` is an honest (rather than skew) partition, these are the *corners()* of the outer partition. In the language of [Sag2001] these would be the "inner corners" of the outer partition.

See also

- *sage.combinat.skew_partition.SkewPartition.outside_corners()*
- *sage.combinat.partition.Partition.outside_corners()*

EXAMPLES:

```
sage: SkewPartition([[4, 3, 1], [2]]).outer_corners()
[(0, 3), (1, 2), (2, 0)]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(4), Integer(3), Integer(1)], [Integer(2)]]).outer_
↪corners()
[(0, 3), (1, 2), (2, 0)]
```

outside_corners()

Return the outside corners of `self`.

The outside corners are corners which are outside of the shape. This should not be confused with *outer_corners()* which consists of corners inside the shape. It returns a result analogous to the . `outside_corners()` method on (non-skew) `Partitions`.

See also

- *sage.combinat.skew_partition.SkewPartition.outer_corners()*
- *sage.combinat.partition.Partition.outside_corners()*

EXAMPLES:

```
sage: mu = SkewPartition([[3,2,1],[2,1]])
sage: mu.pp()
  *
 *
*
sage: mu.outside_corners()
[(0, 3), (1, 2), (2, 1), (3, 0)]
```

```
>>> from sage.all import *
>>> mu = SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(1)]])
>>> mu.pp()
  *
 *
*
>>> mu.outside_corners()
[(0, 3), (1, 2), (2, 1), (3, 0)]
```

overlap()

Return the overlap of `self`.

The overlap of two consecutive rows in a skew partition is the number of pairs of cells (one in each row) that share a common edge. This number can be positive, zero, or negative.

The overlap of a skew partition is the minimum of the overlap of the consecutive rows, or infinity in the case of at most one row. If the overlap is positive, then the skew partition is called *connected*.

EXAMPLES:

```
sage: SkewPartition([[],[]]).overlap()
+Infinity
sage: SkewPartition([[1],[]]).overlap()
+Infinity
```

(continues on next page)

(continued from previous page)

```
sage: SkewPartition([[10],[]]).overlap()
+Infinity
sage: SkewPartition([[10],[2]]).overlap()
+Infinity
sage: SkewPartition([[10,1],[2]]).overlap()
-1
sage: SkewPartition([[10,10],[1]]).overlap()
9
```

```
>>> from sage.all import *
>>> SkewPartition([[],[]]).overlap()
+Infinity
>>> SkewPartition([[Integer(1)],[]]).overlap()
+Infinity
>>> SkewPartition([[Integer(10)],[]]).overlap()
+Infinity
>>> SkewPartition([[Integer(10)],[Integer(2)]]).overlap()
+Infinity
>>> SkewPartition([[Integer(10),Integer(1)],[Integer(2)]]).overlap()
-1
>>> SkewPartition([[Integer(10),Integer(10)],[Integer(1)]]).overlap()
9
```

pieri_macdonald_coeffs()

Computation of the coefficients which appear in the Pieri formula for Macdonald polynomials given in his book (Chapter 6.6 formula 6.24(ii))

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).pieri_macdonald_coeffs()
1
sage: SkewPartition([[3,2,1],[2,2]]).pieri_macdonald_coeffs()
(q^2*t^3 - q^2*t - t^2 + 1)/(q^2*t^3 - q*t^2 - q*t + 1)
sage: SkewPartition([[3,2,1],[2,2,1]]).pieri_macdonald_coeffs()
(q^6*t^8 - q^6*t^6 - q^4*t^7 - q^5*t^5 + q^4*t^5 - q^3*t^6 + q^5*t^3 + 2*q^
↪3*t^4 + q*t^5 - q^3*t^2 + q^2*t^3 - q*t^3 - q^2*t - t^2 + 1)/(q^6*t^8 - q^
↪5*t^7 - q^5*t^6 - q^4*t^6 + q^3*t^5 + 2*q^3*t^4 + q^3*t^3 - q^2*t^2 - q*t^2␣
↪- q*t + 1)
sage: SkewPartition([[3,3,2,2],[3,2,2,1]]).pieri_macdonald_coeffs()
(q^5*t^6 - q^5*t^5 + q^4*t^6 - q^4*t^5 - q^4*t^3 + q^4*t^2 - q^3*t^3 - q^2*t^
↪4 + q^3*t^2 + q^2*t^3 - q*t^4 + q*t^3 + q*t - q + t - 1)/(q^5*t^6 - q^4*t^5␣
↪- q^3*t^4 - q^3*t^3 + q^2*t^3 + q^2*t^2 + q*t - 1)
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(1)]]).pieri_macdonald_coeffs()
1
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(2)]]).pieri_macdonald_coeffs()
(q^2*t^3 - q^2*t - t^2 + 1)/(q^2*t^3 - q*t^2 - q*t + 1)
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).pieri_macdonald_coeffs()
```

(continues on next page)

(continued from previous page)

```
(q^6*t^8 - q^6*t^6 - q^4*t^7 - q^5*t^5 + q^4*t^5 - q^3*t^6 + q^5*t^3 + 2*q^
↪3*t^4 + q*t^5 - q^3*t^2 + q^2*t^3 - q*t^3 - q^2*t - t^2 + 1)/(q^6*t^8 - q^
↪5*t^7 - q^5*t^6 - q^4*t^6 + q^3*t^5 + 2*q^3*t^4 + q^3*t^3 - q^2*t^2 - q*t^2␣
↪- q*t + 1)
>>> SkewPartition([[Integer(3),Integer(3),Integer(2),Integer(2)],[Integer(3),
↪Integer(2),Integer(2),Integer(1)]]).pieri_macdonald_coeffs()
(q^5*t^6 - q^5*t^5 + q^4*t^6 - q^4*t^5 - q^4*t^3 + q^4*t^2 - q^3*t^3 - q^2*t^
↪4 + q^3*t^2 + q^2*t^3 - q*t^4 + q*t^3 + q*t - q + t - 1)/(q^5*t^6 - q^4*t^5␣
↪- q^3*t^4 - q^3*t^3 + q^2*t^3 + q^2*t^2 + q*t - 1)
```

pp()

Pretty-print `self`.

EXAMPLES:

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).pp()
   **
   *
 **
*
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(3),Integer(1)]]).pp()
   **
   *
 **
*
```

quotient(k)

The quotient map extended to skew partitions.

EXAMPLES:

```
sage: SkewPartition([[3, 3, 2, 1], [2, 1]]).quotient(2)
[[3] / [], [] / []]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3), Integer(3), Integer(2), Integer(1)],␣
↪[Integer(2), Integer(1)]]).quotient(Integer(2))
[[3] / [], [] / []]
```

row_lengths()

Return the row lengths of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).row_lengths()
[2, 1, 1]
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)]]).row_lengths()
[2, 1, 1]
```

rows_intersection_set()

Return the set of cells in the rows of the outer shape of `self` which rows intersect the skew diagram of `self`.

EXAMPLES:

```
sage: skp = SkewPartition([[3,2,1],[2,1]])
sage: cells = Set([ (0,0), (0, 1), (0,2), (1, 0), (1, 1), (2, 0)])
sage: skp.rows_intersection_set() == cells
True
```

```
>>> from sage.all import *
>>> skp = SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(2),
↪Integer(1)]])
>>> cells = Set([ (Integer(0),Integer(0)), (Integer(0), Integer(1)),␣
↪(Integer(0),Integer(2)), (Integer(1), Integer(0)), (Integer(1), Integer(1)),
↪ (Integer(2), Integer(0))])
>>> skp.rows_intersection_set() == cells
True
```

size()

Return the size of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).size()
4
```

```
>>> from sage.all import *
>>> SkewPartition([[Integer(3),Integer(2),Integer(1)],[Integer(1),
↪Integer(1)]]).size()
4
```

specht_module(*base_ring=None*)

Return the Specht module corresponding to `self`.

EXAMPLES:

```
sage: mu = SkewPartition([[3,2,1], [2]])
sage: SM = mu.specht_module(QQ)                                         #␣
↪needs sage.modules
sage: s = SymmetricFunctions(QQ).s()                                    #␣
↪needs sage.modules
sage: s(SM.frobenius_image())                                           #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + s[3, 1]
```

```
>>> from sage.all import *
>>> mu = SkewPartition([[Integer(3),Integer(2),Integer(1)], [Integer(2)]])
>>> SM = mu.specht_module(QQ)                                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
>>> s = SymmetricFunctions(QQ).s()                                    #␣
↪needs sage.modules
>>> s(SM.frobenius_image())                                           #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + s[3, 1]
```

We verify that the Frobenius image is the corresponding skew Schur function:

```
sage: s[3,2,1].skew_by(s[2])                                            #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + s[3, 1]
```

```
>>> from sage.all import *
>>> s[Integer(3),Integer(2),Integer(1)].skew_by(s[Integer(2)])          ␣
↪                               # needs sage.modules
s[2, 1, 1] + s[2, 2] + s[3, 1]
```

```
sage: mu = SkewPartition([[4,2,1], [2,1]])
sage: SM = mu.specht_module(QQ)                                         #␣
↪needs sage.modules
sage: s(SM.frobenius_image())                                           #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + 2*s[3, 1] + s[4]
sage: s(mu)                                                             #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + 2*s[3, 1] + s[4]
```

```
>>> from sage.all import *
>>> mu = SkewPartition([[Integer(4),Integer(2),Integer(1)], [Integer(2),
↪Integer(1)]])
>>> SM = mu.specht_module(QQ)                                         #␣
↪needs sage.modules
>>> s(SM.frobenius_image())                                           #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + 2*s[3, 1] + s[4]
>>> s(mu)                                                             #␣
↪needs sage.modules
s[2, 1, 1] + s[2, 2] + 2*s[3, 1] + s[4]
```

specht_module_dimension(*base_ring=None*)

Return the dimension of the Specht module corresponding to `self`.

This is equal to the number of standard (skew) tableaux of shape `self`.

EXAMPLES:

```
sage: mu = SkewPartition([[3,2,1], [2]])
sage: mu.specht_module_dimension()                                      #␣
↪needs sage.modules
8
sage: mu.specht_module_dimension(GF(2))                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules sage.rings.finite_rings
8
```

```
>>> from sage.all import *
>>> mu = SkewPartition([[Integer(3),Integer(2),Integer(1)], [Integer(2)]])
>>> mu.specht_module_dimension()                                          #␣
↪needs sage.modules
8
>>> mu.specht_module_dimension(GF(Integer(2)))                            ␣
↪     # needs sage.modules sage.rings.finite_rings
8
```

to_dag(*format='string'*)

Return a directed acyclic graph corresponding to the skew partition `self`.

The directed acyclic graph corresponding to a skew partition p is the digraph whose vertices are the cells of p, and whose edges go from each cell to its lower and right neighbors (in English notation).

INPUT:

- `format` – either `'string'` or `'tuple'` (default: `'string'`); determines whether the vertices of the resulting dag will be strings or 2-tuples of coordinates

EXAMPLES:

```
sage: # needs sage.graphs
sage: dag = SkewPartition([[3, 3, 1], [1, 1]]).to_dag()
sage: dag.edges(sort=True)
[('0,1', '0,2', None),
('0,1', '1,1', None),
('0,2', '1,2', None),
('1,1', '1,2', None)]
sage: dag.vertices(sort=True)
['0,1', '0,2', '1,1', '1,2', '2,0']
sage: dag = SkewPartition([[3, 2, 1], [1, 1]]).to_dag(format='tuple')
sage: dag.edges(sort=True)
[((0, 1), (0, 2), None), ((0, 1), (1, 1), None)]
sage: dag.vertices(sort=True)
[(0, 1), (0, 2), (1, 1), (2, 0)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> dag = SkewPartition([[Integer(3), Integer(3), Integer(1)], [Integer(1),␣
↪Integer(1)]]).to_dag()
>>> dag.edges(sort=True)
[('0,1', '0,2', None),
('0,1', '1,1', None),
('0,2', '1,2', None),
('1,1', '1,2', None)]
>>> dag.vertices(sort=True)
['0,1', '0,2', '1,1', '1,2', '2,0']
>>> dag = SkewPartition([[Integer(3), Integer(2), Integer(1)], [Integer(1),␣
↪Integer(1)]]).to_dag(format='tuple')
```

(continues on next page)

(continued from previous page)

```
>>> dag.edges(sort=True)
[((0, 1), (0, 2), None), ((0, 1), (1, 1), None)]
>>> dag.vertices(sort=True)
[(0, 1), (0, 2), (1, 1), (2, 0)]
```

to_list()

Return `self` as a list of lists.

EXAMPLES:

```
sage: s = SkewPartition([[4,3,1],[2]])
sage: s.to_list()
[[4, 3, 1], [2]]
sage: type(s.to_list())
<class 'list'>
```

```
>>> from sage.all import *
>>> s = SkewPartition([[Integer(4),Integer(3),Integer(1)],[Integer(2)]])
>>> s.to_list()
[[4, 3, 1], [2]]
>>> type(s.to_list())
<class 'list'>
```

class sage.combinat.skew_partition.**SkewPartitions**(*is_infinite=False*)

Bases: `UniqueRepresentation`, `Parent`

Skew partitions.

Warning

The iterator of this class only yields skew partitions which are reduced, in the sense that there are no empty rows before the last nonempty row, and there are no empty columns before the last nonempty column.

EXAMPLES:

```
sage: SkewPartitions(4)
Skew partitions of 4
sage: SkewPartitions(4).cardinality()
28
sage: SkewPartitions(row_lengths=[2,1,2])
Skew partitions with row lengths [2, 1, 2]
sage: SkewPartitions(4, overlap=2)
Skew partitions of 4 with a minimum overlap of 2
sage: SkewPartitions(4, overlap=2).list()
[[4] / [], [2, 2] / []]
```

```
>>> from sage.all import *
>>> SkewPartitions(Integer(4))
Skew partitions of 4
>>> SkewPartitions(Integer(4)).cardinality()
28
```

(continues on next page)

(continued from previous page)

```
>>> SkewPartitions(row_lengths=[Integer(2),Integer(1),Integer(2)])
Skew partitions with row lengths [2, 1, 2]
>>> SkewPartitions(Integer(4), overlap=Integer(2))
Skew partitions of 4 with a minimum overlap of 2
>>> SkewPartitions(Integer(4), overlap=Integer(2)).list()
[[4] / [], [2, 2] / []]
```

Element

alias of *SkewPartition*

from_row_and_column_length(*rowL*, *colL*)

Construct a partition from its row lengths and column lengths.

INPUT:

- rowL – a composition or a list of positive integers
- colL – a composition or a list of positive integers

OUTPUT:

- If it exists the unique skew-partitions with row lengths rowL and column lengths colL.
- Raise a ValueError if rowL and colL are not compatible.

EXAMPLES:

```
sage: S = SkewPartitions()
sage: print(S.from_row_and_column_length([3,1,2,2],[2,3,1,1,1]).diagram())
  ***
 *
**
**
sage: S.from_row_and_column_length([],[])
[] / []
sage: S.from_row_and_column_length([1],[1])
[1] / []
sage: S.from_row_and_column_length([2,1],[2,1])
[2, 1] / []
sage: S.from_row_and_column_length([1,2],[1,2])
[2, 2] / [1]
sage: S.from_row_and_column_length([1,2],[1,3])
Traceback (most recent call last):
...
ValueError: sum mismatch: [1, 2] and [1, 3]
sage: S.from_row_and_column_length([3,2,1,2],[2,3,1,1,1])
Traceback (most recent call last):
...
ValueError: incompatible row and column length : [3, 2, 1, 2] and [2, 3, 1, 1,
↪ 1]
```

```
>>> from sage.all import *
>>> S = SkewPartitions()
>>> print(S.from_row_and_column_length([Integer(3),Integer(1),Integer(2),
↪Integer(2)],[Integer(2),Integer(3),Integer(1),Integer(1),Integer(1)]).
```

(continues on next page)

(continued from previous page)

```
↪diagram())
  ***
 *
**
**
>>> S.from_row_and_column_length([],[])
[] / []
>>> S.from_row_and_column_length([Integer(1)],[Integer(1)])
[1] / []
>>> S.from_row_and_column_length([Integer(2),Integer(1)],[Integer(2),
↪Integer(1)])
[2, 1] / []
>>> S.from_row_and_column_length([Integer(1),Integer(2)],[Integer(1),
↪Integer(2)])
[2, 2] / [1]
>>> S.from_row_and_column_length([Integer(1),Integer(2)],[Integer(1),
↪Integer(3)])
Traceback (most recent call last):
...
ValueError: sum mismatch: [1, 2] and [1, 3]
>>> S.from_row_and_column_length([Integer(3),Integer(2),Integer(1),
↪Integer(2)],[Integer(2),Integer(3),Integer(1),Integer(1),Integer(1)])
Traceback (most recent call last):
...
ValueError: incompatible row and column length : [3, 2, 1, 2] and [2, 3, 1, 1,
↪ 1]
```

Warning: If some rows and columns have length zero, there is no way to retrieve unambiguously the skew partition. We therefore raise a `ValueError`. For examples here are two skew partitions with the same row and column lengths:

```
sage: skp1 = SkewPartition([[2,2],[2,2]])
sage: skp2 = SkewPartition([[2,1],[2,1]])
sage: skp1.row_lengths(), skp1.column_lengths()
([0, 0], [0, 0])
sage: skp2.row_lengths(), skp2.column_lengths()
([0, 0], [0, 0])
sage: SkewPartitions().from_row_and_column_length([0,0], [0,0])
Traceback (most recent call last):
...
ValueError: row and column length must be positive
```

```
>>> from sage.all import *
>>> skp1 = SkewPartition([[Integer(2),Integer(2)],[Integer(2),Integer(2)]])
>>> skp2 = SkewPartition([[Integer(2),Integer(1)],[Integer(2),Integer(1)]])
>>> skp1.row_lengths(), skp1.column_lengths()
([0, 0], [0, 0])
>>> skp2.row_lengths(), skp2.column_lengths()
([0, 0], [0, 0])
>>> SkewPartitions().from_row_and_column_length([Integer(0),Integer(0)],␣
↪[Integer(0),Integer(0)])
Traceback (most recent call last):
...
ValueError: row and column length must be positive
```

options = Current options for SkewPartitions - convention: English - diagram_str: * - display: quotient - latex: young_diagram - latex_diagram_str: \ast - latex_marking_str: X

class sage.combinat.skew_partition.**SkewPartitions_all**

Bases: *SkewPartitions*

Class of all skew partitions.

class sage.combinat.skew_partition.**SkewPartitions_n**(*n*, *overlap*)

Bases: *SkewPartitions*

The set of skew partitions of `n` with overlap at least `overlap` and no empty row.

INPUT:

- `n` – nonnegative integer
- `overlap` – integer (default: 0)

Caveat: this set is stable under conjugation only for `overlap` equal to 0 or 1. What exactly happens for negative overlaps is not yet well specified and subject to change (we may want to introduce vertical overlap constraints as well).

Todo

As is, this set is essentially the composition of `Compositions(n)` (which give the row lengths) and `SkewPartition(n, row_lengths=...)`, and one would want to "inherit" list and cardinality from this composition.

cardinality()

Return the number of skew partitions of the integer n (with given overlap, if specified; and with no empty rows before the last row).

EXAMPLES:

```
sage: SkewPartitions(0).cardinality()
1
sage: SkewPartitions(4).cardinality()
28
sage: SkewPartitions(5).cardinality()
87
sage: SkewPartitions(4, overlap=1).cardinality()
9
sage: SkewPartitions(5, overlap=1).cardinality()
20
sage: s = SkewPartitions(5, overlap=-1)
sage: s.cardinality() == len(s.list())
True
```

```
>>> from sage.all import *
>>> SkewPartitions(Integer(0)).cardinality()
1
```

(continues on next page)

(continued from previous page)

```
>>> SkewPartitions(Integer(4)).cardinality()
28
>>> SkewPartitions(Integer(5)).cardinality()
87
>>> SkewPartitions(Integer(4), overlap=Integer(1)).cardinality()
9
>>> SkewPartitions(Integer(5), overlap=Integer(1)).cardinality()
20
>>> s = SkewPartitions(Integer(5), overlap=-Integer(1))
>>> s.cardinality() == len(s.list())
True
```

class sage.combinat.skew_partition.**SkewPartitions_rowlengths**(*co*, *overlap*)

Bases: *SkewPartitions*

All skew partitions with given row lengths.

sage.combinat.skew_partition.**row_lengths_aux**(*skp*)

EXAMPLES:

```
sage: from sage.combinat.skew_partition import row_lengths_aux
sage: row_lengths_aux([[5,4,3,1],[3,3,1]])
[2, 1, 2]
sage: row_lengths_aux([[5,4,3,1],[3,1]])
[2, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.skew_partition import row_lengths_aux
>>> row_lengths_aux([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(3),Integer(1)]])
[2, 1, 2]
>>> row_lengths_aux([[Integer(5),Integer(4),Integer(3),Integer(1)],[Integer(3),
↪Integer(1)]])
[2, 3]
```

5.1.313 Skew Tableaux

AUTHORS:

- Mike Hansen: Initial version
- Travis Scrimshaw, Arthur Lubovsky (2013-02-11): Factored out `CombinatorialClass`
- Trevor K. Karn (2022-08-03): added `backward_slide`

class sage.combinat.skew_tableau.**SemistandardSkewTableaux**(*category=None*)

Bases: *SkewTableaux*

Semistandard skew tableaux.

This class can be initialized with several optional variables: the size of the skew tableaux (as a nameless integer variable), their shape (as a nameless skew partition variable), their weight (*weight()*, as a nameless second variable after either the size or the shape) and their maximum entry (as an optional keyword variable called `max_entry`, unless the weight has been specified). If neither the weight nor the maximum entry is specified, the maximum entry defaults to the size of the tableau.

Note that "maximum entry" does not literally mean the highest entry; instead it is just an upper bound that no entry is allowed to surpass.

EXAMPLES:

The (infinite) class of all semistandard skew tableaux:

```
sage: SemistandardSkewTableaux()
Semistandard skew tableaux
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux()
Semistandard skew tableaux
```

The (still infinite) class of all semistandard skew tableaux with maximum entry 2:

```
sage: SemistandardSkewTableaux(max_entry=2)
Semistandard skew tableaux with maximum entry 2
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux(max_entry=Integer(2))
Semistandard skew tableaux with maximum entry 2
```

The class of all semistandard skew tableaux of given size 3 and maximum entry 3:

```
sage: SemistandardSkewTableaux(3)
Semistandard skew tableaux of size 3 and maximum entry 3
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux(Integer(3))
Semistandard skew tableaux of size 3 and maximum entry 3
```

To set a different maximum entry:

```
sage: SemistandardSkewTableaux(3, max_entry = 7)
Semistandard skew tableaux of size 3 and maximum entry 7
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux(Integer(3), max_entry = Integer(7))
Semistandard skew tableaux of size 3 and maximum entry 7
```

Specifying a shape:

```
sage: SemistandardSkewTableaux([[2,1],[]])
Semistandard skew tableaux of shape [2, 1] / [] and maximum entry 3
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux([[Integer(2),Integer(1)],[]])
Semistandard skew tableaux of shape [2, 1] / [] and maximum entry 3
```

Specifying both a shape and a maximum entry:

```
sage: S = SemistandardSkewTableaux([[2,1],[1]], max_entry = 3); S
Semistandard skew tableaux of shape [2, 1] / [1] and maximum entry 3
```

(continues on next page)

(continued from previous page)

```
sage: S.list()
[[[None, 1], [1]],
 [[None, 2], [1]],
 [[None, 1], [2]],
 [[None, 3], [1]],
 [[None, 1], [3]],
 [[None, 2], [2]],
 [[None, 3], [2]],
 [[None, 2], [3]],
 [[None, 3], [3]]]

sage: for n in range(5):
....:     print("{} {}".format(n, len(SemistandardSkewTableaux([[2,2,1],[1]], max_
↪entry = n))))
0 0
1 0
2 1
3 9
4 35
```

```
>>> from sage.all import *
>>> S = SemistandardSkewTableaux([[Integer(2),Integer(1)],[Integer(1)]], max_
↪entry = Integer(3)); S
Semistandard skew tableaux of shape [2, 1] / [1] and maximum entry 3
>>> S.list()
[[[None, 1], [1]],
 [[None, 2], [1]],
 [[None, 1], [2]],
 [[None, 3], [1]],
 [[None, 1], [3]],
 [[None, 2], [2]],
 [[None, 3], [2]],
 [[None, 2], [3]],
 [[None, 3], [3]]]

>>> for n in range(Integer(5)):
...     print("{} {}".format(n, len(SemistandardSkewTableaux([[Integer(2),
↪Integer(2),Integer(1)],[Integer(1)]], max_entry = n))))
0 0
1 0
2 1
3 9
4 35
```

Specifying a shape and a weight:

```
sage: SemistandardSkewTableaux([[2,1],[]],[2,1])
Semistandard skew tableaux of shape [2, 1] / [] and weight [2, 1]
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux([[Integer(2),Integer(1)],[]],[Integer(2),Integer(1)])
Semistandard skew tableaux of shape [2, 1] / [] and weight [2, 1]
```

(the maximum entry is redundant in this case and thus is ignored).

Specifying a size and a weight:

```
sage: SemistandardSkewTableaux(3, [2,1])
Semistandard skew tableaux of size 3 and weight [2, 1]
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux(Integer(3), [Integer(2),Integer(1)])
Semistandard skew tableaux of size 3 and weight [2, 1]
```

Warning

If the shape is not specified, the iterator of this class yields only skew tableaux whose shape is reduced, in the sense that there are no empty rows before the last nonempty row, and there are no empty columns before the last nonempty column. (Otherwise it would go on indefinitely.)

Warning

This class acts as a factory. The resulting classes are mainly useful for iteration. Do not rely on their containment tests, as they are not correct, e. g.:

```
sage: SkewTableau([[None]]) in SemistandardSkewTableaux(2)
True
```

```
>>> from sage.all import *
>>> SkewTableau([[None]]) in SemistandardSkewTableaux(Integer(2))
True
```

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_all**(*max_entry*)

Bases: *SemistandardSkewTableaux*

Class of all semistandard skew tableaux, possibly with a given maximum entry.

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_shape**(*p*, *max_entry*)

Bases: *SemistandardSkewTableaux*

Class of semistandard skew tableaux of a fixed skew shape λ/μ with a given max entry.

A semistandard skew tableau with max entry i is required to have all its entries less or equal to i. It is not required to actually contain an entry i.

INPUT:

- p – a skew partition
- max_entry – the max entry; defaults to the size of p

Warning

Input is not checked; please use *SemistandardSkewTableaux* to ensure the options are properly parsed.

cardinality()

EXAMPLES:

```
sage: SemistandardSkewTableaux([[2,1],[]]).cardinality()
8
sage: SemistandardSkewTableaux([[2,1],[]], max_entry=2).cardinality()
2
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux([[Integer(2),Integer(1)],[]]).cardinality()
8
>>> SemistandardSkewTableaux([[Integer(2),Integer(1)],[]], max_
↪entry=Integer(2)).cardinality()
2
```

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_shape_weight**(*p*, *mu*)

Bases: *SemistandardSkewTableaux*

Class of semistandard skew tableaux of a fixed skew shape λ/ν and weight μ.

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_size**(*n*, *max_entry*)

Bases: *SemistandardSkewTableaux*

Class of all semistandard skew tableaux of a fixed size n, possibly with a given maximum entry.

cardinality()

EXAMPLES:

```
sage: SemistandardSkewTableaux(2).cardinality()
8
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux(Integer(2)).cardinality()
8
```

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_size_weight**(*n*, *mu*)

Bases: *SemistandardSkewTableaux*

Class of semistandard tableaux of a fixed size n and weight μ.

cardinality()

EXAMPLES:

```
sage: SemistandardSkewTableaux(2,[1,1]).cardinality()
4
```

```
>>> from sage.all import *
>>> SemistandardSkewTableaux(Integer(2),[Integer(1),Integer(1)]).cardinality()
4
```

class sage.combinat.skew_tableau.**SkewTableau**(*parent*, *st*)

Bases: ClonableList

A skew tableau.

Note that Sage by default uses the English convention for partitions and tableaux. To change this, see `Tableaux.options()`.

EXAMPLES:

```
sage: st = SkewTableau([[None, 1],[2,3]]); st
[[None, 1], [2, 3]]
sage: st.inner_shape()
[1]
sage: st.outer_shape()
[2, 2]
```

```
>>> from sage.all import *
>>> st = SkewTableau([[None, Integer(1)],[Integer(2),Integer(3)]]); st
[[None, 1], [2, 3]]
>>> st.inner_shape()
[1]
>>> st.outer_shape()
[2, 2]
```

The `expr` form of a skew tableau consists of the inner partition followed by a list of the entries in each row from bottom to top:

```
sage: SkewTableau(expr=[[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

```
>>> from sage.all import *
>>> SkewTableau(expr=[[Integer(1),Integer(1)],[[Integer(5)],[Integer(3),
↪Integer(4)],[Integer(1),Integer(2)]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

The `chain` form of a skew tableau consists of a list of partitions $\lambda_1, \lambda_2, \ldots,$, such that all cells in λ_{i+1} that are not in λ_i have entry i:

```
sage: SkewTableau(chain=[[2], [2, 1], [3, 1], [4, 3, 2, 1]])
[[None, None, 2, 3], [1, 3, 3], [3, 3], [3]]
```

```
>>> from sage.all import *
>>> SkewTableau(chain=[[Integer(2)], [Integer(2), Integer(1)], [Integer(3),␣
↪Integer(1)], [Integer(4), Integer(3), Integer(2), Integer(1)]])
[[None, None, 2, 3], [1, 3, 3], [3, 3], [3]]
```

backward_slide(*corner=None*)

Apply a backward jeu de taquin slide on the specified outside `corner` of `self`.

Backward jeu de taquin slides are defined in Section 3.7 of [Sag2001].

> **Warning**
>
> The `inner_corners()` and `outer_corners()` are the `sage.combinat.partition.Partition.corners()` of the inner and outer partitions of the skew shape. They are different from the inner/outer corners defined in [Sag2001].

The "inner corners" of [Sag2001] may be found by calling `outer_corners()`. The "outer corners" of [Sag2001] may be found by calling `self.outer_shape().outside_corners()`.

EXAMPLES:

```
sage: T = SkewTableaux()([[2, 2], [4, 4], [5]])
sage: Tableaux.options.display='array'
sage: Q = T.backward_slide(); Q
. 2 2
4 4
5
sage: Q.backward_slide((1, 2))
. 2 2
. 4 4
5
sage: Q.reverse_slide((1, 2)) == Q.backward_slide((1, 2))
True

sage: T = SkewTableaux()([[1, 3],[3],[5]]); T
1 3
3
5
sage: T.reverse_slide((1,1))
. 1
3 3
5
```

```
>>> from sage.all import *
>>> T = SkewTableaux()([[Integer(2), Integer(2)], [Integer(4), Integer(4)],␣
↪[Integer(5)]])
>>> Tableaux.options.display='array'
>>> Q = T.backward_slide(); Q
. 2 2
4 4
5
>>> Q.backward_slide((Integer(1), Integer(2)))
. 2 2
. 4 4
5
>>> Q.reverse_slide((Integer(1), Integer(2))) == Q.backward_slide((Integer(1),
↪ Integer(2)))
True

>>> T = SkewTableaux()([[Integer(1), Integer(3)],[Integer(3)],[Integer(5)]]);␣
↪T
1 3
3
5
>>> T.reverse_slide((Integer(1),Integer(1)))
. 1
3 3
5
```

bender_knuth_involution(*k*, *rows=None*, *check=True*)

Return the image of `self` under the k-th Bender–Knuth involution, assuming `self` is a skew semistandard tableau.

Let T be a tableau, then a *lower free `k` in `T`* means a cell of T which is filled with the integer k and whose direct lower neighbor is not filled with the integer $k + 1$ (in particular, this lower neighbor might not exist at all). Let an *upper free `k + 1` in `T`* mean a cell of T which is filled with the integer $k + 1$ and whose direct upper neighbor is not filled with the integer k (in particular, this neighbor might not exist at all). It is clear that for any row r of T, the lower free k's and the upper free $k + 1$'s in r together form a contiguous interval or r.

The *`k`-th Bender–Knuth switch at row `i`* changes the entries of the cells in this interval in such a way that if it used to have a entries of k and b entries of $k + 1$, it will now have b entries of k and a entries of $k + 1$. For fixed k, the k-th Bender–Knuth switches for different i commute. The composition of the k-th Bender–Knuth switches for all rows is called the *`k`-th Bender–Knuth involution*. This is used to show that the Schur functions defined by semistandard (skew) tableaux are symmetric functions.

INPUT:

- `k` – integer
- `rows` – (default: `None`) when set to `None`, the method computes the k-th Bender–Knuth involution as defined above. When an iterable, this computes the composition of the k-th Bender–Knuth switches at row i over all i in `rows`. When set to an integer i, the method computes the k-th Bender–Knuth switch at row i. Note the indexing of the rows starts with 1.
- `check` – (default: `True`) check to make sure `self` is semistandard; set to `False` to avoid this check

OUTPUT:

The image of `self` under either the k-th Bender–Knuth involution, the k-th Bender–Knuth switch at a certain row, or the composition of such switches, as detailed in the INPUT section.

EXAMPLES:

```
sage: t = SkewTableau([[None,None,None,4,4,5,6,7],[None,2,4,6,7,7,7],
....:                  [None,4,5,8,8,9],[None,6,7,10],[None,8,8,11],[None],
↪[4]])
sage: t
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7],
 [None, 4, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [4]]
sage: t.bender_knuth_involution(1)
[[None, None, None, 4, 4, 5, 6, 7], [None, 1, 4, 6, 7, 7, 7],
 [None, 4, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [4]]
sage: t.bender_knuth_involution(4)
[[None, None, None, 4, 5, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7],
 [None, 5, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [5]]
sage: t.bender_knuth_involution(5)
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 5, 7, 7, 7],
 [None, 4, 6, 8, 8, 9], [None, 5, 7, 10], [None, 8, 8, 11], [None], [4]]
sage: t.bender_knuth_involution(6)
[[None, None, None, 4, 4, 5, 6, 6], [None, 2, 4, 6, 6, 7, 7],
 [None, 4, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [4]]
sage: t.bender_knuth_involution(666) == t
True
sage: t.bender_knuth_involution(4, 2) == t
True
sage: t.bender_knuth_involution(4, 3)
```

(continues on next page)

(continued from previous page)

```
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7],
 [None, 5, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [4]]
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None,None,None,Integer(4),Integer(4),Integer(5),
↪Integer(6),Integer(7)],[None,Integer(2),Integer(4),Integer(6),Integer(7),
↪Integer(7),Integer(7)],
...                     [None,Integer(4),Integer(5),Integer(8),Integer(8),
↪Integer(9)],[None,Integer(6),Integer(7),Integer(10)],[None,Integer(8),
↪Integer(8),Integer(11)],[None],[Integer(4)]])
>>> t
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7],
 [None, 4, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [4]]
>>> t.bender_knuth_involution(Integer(1))
[[None, None, None, 4, 4, 5, 6, 7], [None, 1, 4, 6, 7, 7, 7],
 [None, 4, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [4]]
>>> t.bender_knuth_involution(Integer(4))
[[None, None, None, 4, 5, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7],
 [None, 5, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [5]]
>>> t.bender_knuth_involution(Integer(5))
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 5, 7, 7, 7],
 [None, 4, 6, 8, 8, 9], [None, 5, 7, 10], [None, 8, 8, 11], [None], [4]]
>>> t.bender_knuth_involution(Integer(6))
[[None, None, None, 4, 4, 5, 6, 6], [None, 2, 4, 6, 6, 7, 7],
 [None, 4, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [4]]
>>> t.bender_knuth_involution(Integer(666)) == t
True
>>> t.bender_knuth_involution(Integer(4), Integer(2)) == t
True
>>> t.bender_knuth_involution(Integer(4), Integer(3))
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7],
 [None, 5, 5, 8, 8, 9], [None, 6, 7, 10], [None, 8, 8, 11], [None], [4]]
```

The Bender–Knuth involution is an involution:

```
sage: t = SkewTableau([[None,3,4,4],[None,6,10],[7,7,11],[18]])
sage: all(t.bender_knuth_involution(k).bender_knuth_involution(k)
....:     == t for k in range(1,4))
True
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None,Integer(3),Integer(4),Integer(4)],[None,Integer(6),
↪Integer(10)],[Integer(7),Integer(7),Integer(11)],[Integer(18)]])
>>> all(t.bender_knuth_involution(k).bender_knuth_involution(k)
...     == t for k in range(Integer(1),Integer(4)))
True
```

The same for the single switches:

```
sage: all(t.bender_knuth_involution(k, j).bender_knuth_involution(k, j)
....:     == t for k in range(1,5) for j in range(1, 5))
True
```

```
>>> from sage.all import *
>>> all(t.bender_knuth_involution(k, j).bender_knuth_involution(k, j)
...     == t for k in range(Integer(1),Integer(5)) for j in range(Integer(1),
→Integer(5)))
True
```

Locality of the Bender–Knuth involutions:

```
sage: all(t.bender_knuth_involution(k).bender_knuth_involution(l)
....:     == t.bender_knuth_involution(l).bender_knuth_involution(k)
....:     for k in range(1,5) for l in range(1,5) if abs(k - l) > 1)
True
```

```
>>> from sage.all import *
>>> all(t.bender_knuth_involution(k).bender_knuth_involution(l)
...     == t.bender_knuth_involution(l).bender_knuth_involution(k)
...     for k in range(Integer(1),Integer(5)) for l in range(Integer(1),
→Integer(5)) if abs(k - l) > Integer(1))
True
```

AUTHORS:

- Darij Grinberg (2013-05-14)

cells()

Return the cells in `self`.

EXAMPLES:

```
sage: s = SkewTableau([[None,1,2],[3],[6]])
sage: s.cells()
[(0, 1), (0, 2), (1, 0), (2, 0)]
```

```
>>> from sage.all import *
>>> s = SkewTableau([[None,Integer(1),Integer(2)],[Integer(3)],[Integer(6)]])
>>> s.cells()
[(0, 1), (0, 2), (1, 0), (2, 0)]
```

cells_by_content(*c*)

Return the coordinates of the cells in `self` with content `c`.

EXAMPLES:

```
sage: s = SkewTableau([[None,1,2],[3,4,5],[6]])
sage: s.cells_by_content(0)
[(1, 1)]
sage: s.cells_by_content(1)
[(0, 1), (1, 2)]
sage: s.cells_by_content(2)
[(0, 2)]
sage: s.cells_by_content(-1)
[(1, 0)]
sage: s.cells_by_content(-2)
[(2, 0)]
```

```
>>> from sage.all import *
>>> s = SkewTableau([[None,Integer(1),Integer(2)],[Integer(3),Integer(4),
↪Integer(5)],[Integer(6)]])
>>> s.cells_by_content(Integer(0))
[(1, 1)]
>>> s.cells_by_content(Integer(1))
[(0, 1), (1, 2)]
>>> s.cells_by_content(Integer(2))
[(0, 2)]
>>> s.cells_by_content(-Integer(1))
[(1, 0)]
>>> s.cells_by_content(-Integer(2))
[(2, 0)]
```

cells_containing(*i*)

Return the list of cells in which the letter `i` appears in the tableau `self`. The list is ordered with cells appearing from left to right.

Cells are given as pairs of coordinates (a, b), where both rows and columns are counted from 0 (so $a = 0$ means the cell lies in the leftmost column of the tableau, etc.).

EXAMPLES:

```
sage: t = SkewTableau([[None,None,3],[None,3,5],[4,5]])
sage: t.cells_containing(5)
[(2, 1), (1, 2)]
sage: t.cells_containing(4)
[(2, 0)]
sage: t.cells_containing(2)
[]

sage: t = SkewTableau([[None,None,None,None],[None,4,5],[None,5,6],[None,9],
↪[None]])
sage: t.cells_containing(2)
[]
sage: t.cells_containing(4)
[(1, 1)]
sage: t.cells_containing(5)
[(2, 1), (1, 2)]

sage: SkewTableau([]).cells_containing(3)
[]

sage: SkewTableau([[None,None],[None]]).cells_containing(3)
[]
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None,None,Integer(3)],[None,Integer(3),Integer(5)],
↪[Integer(4),Integer(5)]])
>>> t.cells_containing(Integer(5))
[(2, 1), (1, 2)]
>>> t.cells_containing(Integer(4))
[(2, 0)]
```

(continues on next page)

(continued from previous page)

```
>>> t.cells_containing(Integer(2))
[]

>>> t = SkewTableau([[None,None,None,None],[None,Integer(4),Integer(5)],[None,
↪Integer(5),Integer(6)],[None,Integer(9)],[None]])
>>> t.cells_containing(Integer(2))
[]
>>> t.cells_containing(Integer(4))
[(1, 1)]
>>> t.cells_containing(Integer(5))
[(2, 1), (1, 2)]

>>> SkewTableau([]).cells_containing(Integer(3))
[]

>>> SkewTableau([[None,None],[None]]).cells_containing(Integer(3))
[]
```

check()

Check that `self` is a valid skew tableau. This is currently far too liberal, and only checks some trivial things.

EXAMPLES:

```
sage: t = SkewTableau([[None,1,1],[2]])
sage: t.check()

sage: t = SkewTableau([[None, None, 1], [2, 4], [], [3, 4, 5]])
Traceback (most recent call last):
...
TypeError: a skew tableau cannot have an empty list for a row

sage: s = SkewTableau([[1, None, None],[2, None],[3]])
Traceback (most recent call last):
...
TypeError: not a valid skew tableau
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None,Integer(1),Integer(1)],[Integer(2)]])
>>> t.check()

>>> t = SkewTableau([[None, None, Integer(1)], [Integer(2), Integer(4)], [],␣
↪[Integer(3), Integer(4), Integer(5)]])
Traceback (most recent call last):
...
TypeError: a skew tableau cannot have an empty list for a row

>>> s = SkewTableau([[Integer(1), None, None],[Integer(2), None],
↪[Integer(3)]])
Traceback (most recent call last):
...
TypeError: not a valid skew tableau
```

column_stabilizer()

Return the `PermutationGroup()` corresponding to the column stabilizer of `self`.

This assumes that every integer from 1 to the size of `self` appears exactly once in `self`.

EXAMPLES:

```
sage: # needs sage.groups
sage: cs = SkewTableau([[None,2,3],[1,5],[4]]).column_stabilizer()
sage: cs.order() == factorial(2) * factorial(2)
True
sage: PermutationGroupElement([(1,3,2),(4,5)]) in cs
False
sage: PermutationGroupElement([(1,4)]) in cs
True
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> cs = SkewTableau([[None,Integer(2),Integer(3)],[Integer(1),Integer(5)],
↪[Integer(4)]]).column_stabilizer()
>>> cs.order() == factorial(Integer(2)) * factorial(Integer(2))
True
>>> PermutationGroupElement([(Integer(1),Integer(3),Integer(2)),(Integer(4),
↪Integer(5))]) in cs
False
>>> PermutationGroupElement([(Integer(1),Integer(4))]) in cs
True
```

conjugate()

Return the conjugate of `self`.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2,3]]).conjugate()
[[None, 2], [1, 3]]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(1)],[Integer(2),Integer(3)]]).conjugate()
[[None, 2], [1, 3]]
```

entries_by_content(*c*)

Return the entries in `self` with content `c`.

EXAMPLES:

```
sage: s = SkewTableau([[None,1,2],[3,4,5],[6]])
sage: s.entries_by_content(0)
[4]
sage: s.entries_by_content(1)
[1, 5]
sage: s.entries_by_content(2)
[2]
sage: s.entries_by_content(-1)
[3]
```

(continues on next page)

(continued from previous page)

```
sage: s.entries_by_content(-2)
[6]
```

```
>>> from sage.all import *
>>> s = SkewTableau([[None,Integer(1),Integer(2)],[Integer(3),Integer(4),
↪Integer(5)],[Integer(6)]])
>>> s.entries_by_content(Integer(0))
[4]
>>> s.entries_by_content(Integer(1))
[1, 5]
>>> s.entries_by_content(Integer(2))
[2]
>>> s.entries_by_content(-Integer(1))
[3]
>>> s.entries_by_content(-Integer(2))
[6]
```

evaluation()

Return the weight (aka evaluation) of the tableau `self`. Trailing zeroes are omitted when returning the weight.

The weight of a skew tableau T is the sequence $(a_1, a_2, a_3, \ldots)$, where a_k is the number of entries of T equal to k. This sequence contains only finitely many nonzero entries.

The weight of a skew tableau T is the same as the weight of the reading word of T, for any reading order.

evaluation() is a synonym for this method.

EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).weight()
[1, 1, 1, 1]

sage: SkewTableau([[None,2],[None,4],[None,5],[None]]).weight()
[0, 1, 0, 1, 1]

sage: SkewTableau([]).weight()
[]

sage: SkewTableau([[None,None,None],[None]]).weight()
[]

sage: SkewTableau([[None,3,4],[None,6,7],[4,8],[5,13],[6],[7]]).weight()
[0, 0, 1, 2, 1, 2, 2, 1, 0, 0, 0, 0, 1]
```

```
>>> from sage.all import *
>>> SkewTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).weight()
[1, 1, 1, 1]

>>> SkewTableau([[None,Integer(2)],[None,Integer(4)],[None,Integer(5)],
↪[None]]).weight()
[0, 1, 0, 1, 1]

>>> SkewTableau([]).weight()
```

(continues on next page)

(continued from previous page)

```
[]

>>> SkewTableau([[None,None,None],[None]]).weight()
[]

>>> SkewTableau([[None,Integer(3),Integer(4)],[None,Integer(6),Integer(7)],
↪[Integer(4),Integer(8)],[Integer(5),Integer(13)],[Integer(6)],
↪[Integer(7)]]).weight()
[0, 0, 1, 2, 1, 2, 2, 1, 0, 0, 0, 0, 1]
```

filling()

Return a list of the non-empty entries in `self`.

EXAMPLES:

```
sage: t = SkewTableau([[None,1],[2,3]])
sage: t.filling()
[[1], [2, 3]]
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None,Integer(1)],[Integer(2),Integer(3)]])
>>> t.filling()
[[1], [2, 3]]
```

inner_shape()

Return the inner shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).inner_shape()
[1, 1]
sage: SkewTableau([[1,2],[3,4],[7]]).inner_shape()
[]
sage: SkewTableau([[None,None,None,2,3],[None,1],[None],[2]]).inner_shape()
[3, 1, 1]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(1),Integer(2)],[None,Integer(3)],
↪[Integer(4)]]).inner_shape()
[1, 1]
>>> SkewTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)],
↪[Integer(7)]]).inner_shape()
[]
>>> SkewTableau([[None,None,None,Integer(2),Integer(3)],[None,Integer(1)],
↪[None],[Integer(2)]]).inner_shape()
[3, 1, 1]
```

inner_size()

Return the size of the inner shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).inner_size()
2
sage: SkewTableau([[None, 2], [1, 3]]).inner_size()
1
```

```
>>> from sage.all import *
>>> SkewTableau([[None, Integer(2), Integer(4)], [None, Integer(3)],␣
↪[Integer(1)]]).inner_size()
2
>>> SkewTableau([[None, Integer(2)], [Integer(1), Integer(3)]]).inner_size()
1
```

is_k_tableau(*k*)

Check whether `self` is a valid skew weak k-tableau.

EXAMPLES:

```
sage: t = SkewTableau([[None,2,3],[2,3],[3]])
sage: t.is_k_tableau(3)
True
sage: t = SkewTableau([[None,1,3],[2,2],[3]])
sage: t.is_k_tableau(3)
False
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None,Integer(2),Integer(3)],[Integer(2),Integer(3)],
↪[Integer(3)]])
>>> t.is_k_tableau(Integer(3))
True
>>> t = SkewTableau([[None,Integer(1),Integer(3)],[Integer(2),Integer(2)],
↪[Integer(3)]])
>>> t.is_k_tableau(Integer(3))
False
```

is_ribbon()

Return `True` if and only if the shape of `self` is a ribbon.

This means that it has exactly one cell in each of q consecutive diagonals for some nonnegative integer q.

EXAMPLES:

```
sage: S = SkewTableau([[None, None, 1, 2],[None, None, 3],[1, 3, 4]])
sage: S.pp()
  .  .  1  2
  .  .  3
  1  3  4
sage: S.is_ribbon()
True

sage: S = SkewTableau([[None, 1, 1, 2],[None, 2, 3],[1, 3, 4]])
sage: S.pp()
  .  1  1  2
  .  2  3
  1  3  4
```

(continues on next page)

(continued from previous page)

```
sage: S.is_ribbon()
False

sage: S = SkewTableau([[None, None, 1, 2],[None, None, 3],[1]])
sage: S.pp()
  .   .   1   2
  .   .   3
  1
sage: S.is_ribbon()
False

sage: S = SkewTableau([[None, None, None, None],[None, None, 3],[1, 2, 4]])
sage: S.pp()
  .   .   .   .
  .   .   3
  1   2   4
sage: S.is_ribbon()
True

sage: S = SkewTableau([[None, None, None, None],[None, None, 3],[None, 2, 4]])
sage: S.pp()
  .   .   .   .
  .   .   3
  .   2   4
sage: S.is_ribbon()
True

sage: S = SkewTableau([[None, None],[None]])
sage: S.pp()
  .   .
  .
sage: S.is_ribbon()
True
```

```
>>> from sage.all import *
>>> S = SkewTableau([[None, None, Integer(1), Integer(2)],[None, None,␣
↪Integer(3)],[Integer(1), Integer(3), Integer(4)]])
>>> S.pp()
  .   .   1   2
  .   .   3
  1   3   4
>>> S.is_ribbon()
True

>>> S = SkewTableau([[None, Integer(1), Integer(1), Integer(2)],[None,␣
↪Integer(2), Integer(3)],[Integer(1), Integer(3), Integer(4)]])
>>> S.pp()
  .   1   1   2
  .   2   3
  1   3   4
>>> S.is_ribbon()
False
```

(continues on next page)

(continued from previous page)

```
>>> S = SkewTableau([[None, None, Integer(1), Integer(2)],[None, None,␣
↪Integer(3)],[Integer(1)]])
>>> S.pp()
  .  .  1  2
  .  .  3
  1
>>> S.is_ribbon()
False

>>> S = SkewTableau([[None, None, None, None],[None, None, Integer(3)],
↪[Integer(1), Integer(2), Integer(4)]])
>>> S.pp()
  .  .  .  .
  .  .  3
  1  2  4
>>> S.is_ribbon()
True

>>> S = SkewTableau([[None, None, None, None],[None, None, Integer(3)],[None,␣
↪Integer(2), Integer(4)]])
>>> S.pp()
  .  .  .  .
  .  .  3
  .  2  4
>>> S.is_ribbon()
True

>>> S = SkewTableau([[None, None],[None]])
>>> S.pp()
  .  .
  .
>>> S.is_ribbon()
True
```

is_semistandard()

Return `True` if `self` is a semistandard skew tableau and `False` otherwise.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 2], [1, 3]]).is_semistandard()
True
sage: SkewTableau([[None, 2], [2, 4]]).is_semistandard()
True
sage: SkewTableau([[None, 3], [2, 4]]).is_semistandard()
True
sage: SkewTableau([[None, 2], [1, 2]]).is_semistandard()
False
sage: SkewTableau([[None, 2, 3]]).is_semistandard()
True
sage: SkewTableau([[None, 3, 2]]).is_semistandard()
False
```

(continues on next page)

(continued from previous page)

```
sage: SkewTableau([[None, 2, 3], [1, 4]]).is_semistandard()
True
sage: SkewTableau([[None, 2, 3], [1, 2]]).is_semistandard()
False
sage: SkewTableau([[None, 2, 3], [None, None, 4]]).is_semistandard()
False
```

```
>>> from sage.all import *
>>> SkewTableau([[None, Integer(2), Integer(2)], [Integer(1), Integer(3)]]).
→is_semistandard()
True
>>> SkewTableau([[None, Integer(2)], [Integer(2), Integer(4)]]).is_
→semistandard()
True
>>> SkewTableau([[None, Integer(3)], [Integer(2), Integer(4)]]).is_
→semistandard()
True
>>> SkewTableau([[None, Integer(2)], [Integer(1), Integer(2)]]).is_
→semistandard()
False
>>> SkewTableau([[None, Integer(2), Integer(3)]]).is_semistandard()
True
>>> SkewTableau([[None, Integer(3), Integer(2)]]).is_semistandard()
False
>>> SkewTableau([[None, Integer(2), Integer(3)], [Integer(1), Integer(4)]]).
→is_semistandard()
True
>>> SkewTableau([[None, Integer(2), Integer(3)], [Integer(1), Integer(2)]]).
→is_semistandard()
False
>>> SkewTableau([[None, Integer(2), Integer(3)], [None, None, Integer(4)]]).
→is_semistandard()
False
```

is_standard()

Return `True` if `self` is a standard skew tableau and `False` otherwise.

EXAMPLES:

```
sage: SkewTableau([[None, 2], [1, 3]]).is_standard()
True
sage: SkewTableau([[None, 2], [2, 4]]).is_standard()
False
sage: SkewTableau([[None, 3], [2, 4]]).is_standard()
False
sage: SkewTableau([[None, 2], [2, 4]]).is_standard()
False
```

```
>>> from sage.all import *
>>> SkewTableau([[None, Integer(2)], [Integer(1), Integer(3)]]).is_standard()
True
>>> SkewTableau([[None, Integer(2)], [Integer(2), Integer(4)]]).is_standard()
```

(continues on next page)

(continued from previous page)

```
False
>>> SkewTableau([[None, Integer(3)], [Integer(2), Integer(4)]]).is_standard()
False
>>> SkewTableau([[None, Integer(2)], [Integer(2), Integer(4)]]).is_standard()
False
```

outer_shape()

Return the outer shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).outer_shape()
[3, 2, 1]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(1),Integer(2)],[None,Integer(3)],
↪[Integer(4)]]).outer_shape()
[3, 2, 1]
```

outer_size()

Return the size of the outer shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).outer_size()
6
sage: SkewTableau([[None, 2], [1, 3]]).outer_size()
4
```

```
>>> from sage.all import *
>>> SkewTableau([[None, Integer(2), Integer(4)], [None, Integer(3)],␣
↪[Integer(1)]]).outer_size()
6
>>> SkewTableau([[None, Integer(2)], [Integer(1), Integer(3)]]).outer_size()
4
```

pp()

Return a pretty print string of the tableau.

EXAMPLES:

```
sage: SkewTableau([[None,2,3],[None,4],[5]]).pp()
  .  2  3
  .  4
  5
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(2),Integer(3)],[None,Integer(4)],
↪[Integer(5)]]).pp()
  .  2  3
  .  4
  5
```

rectify(*algorithm=None*)

Return a *StandardTableau*, *SemistandardTableau*, or just *Tableau* formed by applying the jeu de taquin process to `self`.

See page 15 of [Ful1997].

INPUT:

- `algorithm` – (optional) if set to `'jdt'`, rectifies by jeu de taquin; if set to `'schensted'`, rectifies by Schensted insertion of the reading word. Otherwise, guesses which will be faster.

EXAMPLES:

```
sage: S = SkewTableau([[None,1],[2,3]])
sage: S.rectify()
[[1, 3], [2]]
sage: T = SkewTableau([[None, None, None, 4],[None,None,1,6],[None,None,5],[2,
↪3]])
sage: T.rectify()
[[1, 3, 4, 6], [2, 5]]
sage: T.rectify(algorithm='jdt')
[[1, 3, 4, 6], [2, 5]]
sage: T.rectify(algorithm='schensted')
[[1, 3, 4, 6], [2, 5]]
sage: T.rectify(algorithm='spaghetti')
Traceback (most recent call last):
...
ValueError: algorithm must be 'jdt', 'schensted', or None
```

```
>>> from sage.all import *
>>> S = SkewTableau([[None,Integer(1)],[Integer(2),Integer(3)]])
>>> S.rectify()
[[1, 3], [2]]
>>> T = SkewTableau([[None, None, None, Integer(4)],[None,None,Integer(1),
↪Integer(6)],[None,None,Integer(5)],[Integer(2),Integer(3)]])
>>> T.rectify()
[[1, 3, 4, 6], [2, 5]]
>>> T.rectify(algorithm='jdt')
[[1, 3, 4, 6], [2, 5]]
>>> T.rectify(algorithm='schensted')
[[1, 3, 4, 6], [2, 5]]
>>> T.rectify(algorithm='spaghetti')
Traceback (most recent call last):
...
ValueError: algorithm must be 'jdt', 'schensted', or None
```

restrict(*n*)

Return the restriction of the (semi)standard skew tableau to all the numbers less than or equal to `n`.

Note

If only the outer shape of the restriction, rather than the whole restriction, is needed, then the faster method *restriction_outer_shape()* is preferred. Similarly if only the skew shape is needed, use *restriction_shape()*.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2],[3]]).restrict(2)
[[None, 1], [2]]
sage: SkewTableau([[None,1],[2],[3]]).restrict(1)
[[None, 1]]
sage: SkewTableau([[None,1],[1],[2]]).restrict(1)
[[None, 1], [1]]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(1)],[Integer(2)],[Integer(3)]]).
↪restrict(Integer(2))
[[None, 1], [2]]
>>> SkewTableau([[None,Integer(1)],[Integer(2)],[Integer(3)]]).
↪restrict(Integer(1))
[[None, 1]]
>>> SkewTableau([[None,Integer(1)],[Integer(1)],[Integer(2)]]).
↪restrict(Integer(1))
[[None, 1], [1]]
```

restriction_outer_shape(n)

Return the outer shape of the restriction of the semistandard skew tableau `self` to n.

If T is a semistandard skew tableau and n is a nonnegative integer, then the restriction of T to n is defined as the (semistandard) skew tableau obtained by removing all cells filled with entries greater than n from T.

This method computes merely the outer shape of the restriction. For the restriction itself, use *restrict()*.

EXAMPLES:

```
sage: SkewTableau([[None,None],[2,3],[3,4]]).restriction_outer_shape(3)
[2, 2, 1]
sage: SkewTableau([[None,2],[None],[4],[5]]).restriction_outer_shape(2)
[2, 1]
sage: T = SkewTableau([[None,None,3,5],[None,4,4],[17]])
sage: T.restriction_outer_shape(0)
[2, 1]
sage: T.restriction_outer_shape(2)
[2, 1]
sage: T.restriction_outer_shape(3)
[3, 1]
sage: T.restriction_outer_shape(4)
[3, 3]
sage: T.restriction_outer_shape(19)
[4, 3, 1]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,None],[Integer(2),Integer(3)],[Integer(3),
↪Integer(4)]]).restriction_outer_shape(Integer(3))
[2, 2, 1]
>>> SkewTableau([[None,Integer(2)],[None],[Integer(4)],[Integer(5)]]).
↪restriction_outer_shape(Integer(2))
[2, 1]
>>> T = SkewTableau([[None,None,Integer(3),Integer(5)],[None,Integer(4),
```

(continues on next page)

(continued from previous page)

```
↪Integer(4)],[Integer(17)]])
>>> T.restriction_outer_shape(Integer(0))
[2, 1]
>>> T.restriction_outer_shape(Integer(2))
[2, 1]
>>> T.restriction_outer_shape(Integer(3))
[3, 1]
>>> T.restriction_outer_shape(Integer(4))
[3, 3]
>>> T.restriction_outer_shape(Integer(19))
[4, 3, 1]
```

restriction_shape(n)

Return the skew shape of the restriction of the semistandard skew tableau `self` to `n`.

If T is a semistandard skew tableau and n is a nonnegative integer, then the restriction of T to n is defined as the (semistandard) skew tableau obtained by removing all cells filled with entries greater than n from T.

This method computes merely the skew shape of the restriction. For the restriction itself, use `restrict()`.

EXAMPLES:

```
sage: SkewTableau([[None,None],[2,3],[3,4]]).restriction_shape(3)
[2, 2, 1] / [2]
sage: SkewTableau([[None,2],[None],[4],[5]]).restriction_shape(2)
[2, 1] / [1, 1]
sage: T = SkewTableau([[None,None,3,5],[None,4,4],[17]])
sage: T.restriction_shape(0)
[2, 1] / [2, 1]
sage: T.restriction_shape(2)
[2, 1] / [2, 1]
sage: T.restriction_shape(3)
[3, 1] / [2, 1]
sage: T.restriction_shape(4)
[3, 3] / [2, 1]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,None],[Integer(2),Integer(3)],[Integer(3),
↪Integer(4)]]).restriction_shape(Integer(3))
[2, 2, 1] / [2]
>>> SkewTableau([[None,Integer(2)],[None],[Integer(4)],[Integer(5)]]).
↪restriction_shape(Integer(2))
[2, 1] / [1, 1]
>>> T = SkewTableau([[None,None,Integer(3),Integer(5)],[None,Integer(4),
↪Integer(4)],[Integer(17)]])
>>> T.restriction_shape(Integer(0))
[2, 1] / [2, 1]
>>> T.restriction_shape(Integer(2))
[2, 1] / [2, 1]
>>> T.restriction_shape(Integer(3))
[3, 1] / [2, 1]
>>> T.restriction_shape(Integer(4))
[3, 3] / [2, 1]
```

reverse_slide(*corner=None*)

Apply a backward jeu de taquin slide on the specified outside `corner` of `self`.

Backward jeu de taquin slides are defined in Section 3.7 of [Sag2001].

> **Warning**
>
> The `inner_corners()` and `outer_corners()` are the *sage.combinat.partition.Partition.corners()* of the inner and outer partitions of the skew shape. They are different from the inner/outer corners defined in [Sag2001].
>
> The "inner corners" of [Sag2001] may be found by calling `outer_corners()`. The "outer corners" of [Sag2001] may be found by calling `self.outer_shape().outside_corners()`.

EXAMPLES:

```
sage: T = SkewTableaux()([[2, 2], [4, 4], [5]])
sage: Tableaux.options.display='array'
sage: Q = T.backward_slide(); Q
. 2 2
4 4
5
sage: Q.backward_slide((1, 2))
. 2 2
. 4 4
5
sage: Q.reverse_slide((1, 2)) == Q.backward_slide((1, 2))
True

sage: T = SkewTableaux()([[1, 3],[3],[5]]); T
1 3
3
5
sage: T.reverse_slide((1,1))
. 1
3 3
5
```

```
>>> from sage.all import *
>>> T = SkewTableaux()([[Integer(2), Integer(2)], [Integer(4), Integer(4)],␣
→[Integer(5)]])
>>> Tableaux.options.display='array'
>>> Q = T.backward_slide(); Q
. 2 2
4 4
5
>>> Q.backward_slide((Integer(1), Integer(2)))
. 2 2
. 4 4
5
>>> Q.reverse_slide((Integer(1), Integer(2))) == Q.backward_slide((Integer(1),
→ Integer(2)))
True
```

(continues on next page)

(continued from previous page)

```
>>> T = SkewTableaux()([[Integer(1), Integer(3)],[Integer(3)],[Integer(5)]]);␣
↪T
1 3
3
5
>>> T.reverse_slide((Integer(1),Integer(1)))
. 1
3 3
5
```

row_stabilizer()

Return the PermutationGroup() corresponding to the row stabilizer of self.

This assumes that every integer from 1 to the size of self appears exactly once in self.

EXAMPLES:

```
sage: # needs sage.groups
sage: rs = SkewTableau([[None,1,2,3],[4,5]]).row_stabilizer()
sage: rs.order() == factorial(3) * factorial(2)
True
sage: PermutationGroupElement([(1,3,2),(4,5)]) in rs
True
sage: PermutationGroupElement([(1,4)]) in rs
False
sage: rs = SkewTableau([[None,1,2],[3]]).row_stabilizer()
sage: PermutationGroupElement([(1,2),(3,)]) in rs
True
sage: rs.one().domain()
[1, 2, 3]
sage: rs = SkewTableau([[None,None,1],[None,2],[3]]).row_stabilizer()
sage: rs.order()
1
sage: rs = SkewTableau([[None,None,2,4,5],[1,3]]).row_stabilizer()
sage: rs.order()
12
sage: rs = SkewTableau([]).row_stabilizer()
sage: rs.order()
1
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> rs = SkewTableau([[None,Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]]).row_stabilizer()
>>> rs.order() == factorial(Integer(3)) * factorial(Integer(2))
True
>>> PermutationGroupElement([(Integer(1),Integer(3),Integer(2)),(Integer(4),
↪Integer(5))]) in rs
True
>>> PermutationGroupElement([(Integer(1),Integer(4))]) in rs
False
>>> rs = SkewTableau([[None,Integer(1),Integer(2)],[Integer(3)]]).row_
```

(continues on next page)

(continued from previous page)

```
↪stabilizer()
>>> PermutationGroupElement([(Integer(1),Integer(2)),(Integer(3),)]) in rs
True
>>> rs.one().domain()
[1, 2, 3]
>>> rs = SkewTableau([[None,None,Integer(1)],[None,Integer(2)],[Integer(3)]]).
↪row_stabilizer()
>>> rs.order()
1
>>> rs = SkewTableau([[None,None,Integer(2),Integer(4),Integer(5)],
↪[Integer(1),Integer(3)]]).row_stabilizer()
>>> rs.order()
12
>>> rs = SkewTableau([]).row_stabilizer()
>>> rs.order()
1
```

shape()

Return the shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).shape()
[3, 2, 1] / [1, 1]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(1),Integer(2)],[None,Integer(3)],
↪[Integer(4)]]).shape()
[3, 2, 1] / [1, 1]
```

shuffle(*t2*)

Shuffle the standard tableaux `self` and `t2`.

Let `t1 = self`. The shape of `t2` must extend the shape of `t1`, that is, `self.outer_shape() == t2.inner_shape()`. Then this function computes the pair of tableaux `(t2_new, t1_new)` obtained by using jeu de taquin slides to move the boxes of `t2` behind the boxes of `self`.

The entries of `t2_new` are obtained by performing successive inwards jeu de taquin slides on `t2` in the order indicated by the entries of `t1`, from largest to smallest. The entries of `t1` then slide outwards one by one and land in the squares vacated successively by `t2`, forming `t1_new`.

Note

Equivalently, the entries of `t1_new` are obtained by performing outer jeu de taquin slides on `t1` in the order indicated by the entries of `t2`, from smallest to largest. In this case the entries of `t2` slide backwards and fill the squares successively vacated by `t1` and so form `t2_new`. (This is not how the algorithm is implemented.)

INPUT:

- `self`, `t2` – a pair of standard SkewTableaux with `self.outer_shape() == t2.inner_shape()`

OUTPUT:

- `t2_new, t1_new` – a pair of standard *SkewTableaux* with `t2_new.outer_shape() == t1_new.inner_shape()`

EXAMPLES:

```
sage: t1 = SkewTableau([[None, 1, 2], [3, 4]])
sage: t2 = SkewTableau([[None, None, None, 3], [None, None, 4], [1, 2, 5]])
sage: (t2_new, t1_new) = t1.shuffle(t2)
sage: t1_new
[[None, None, None, 2], [None, None, 1], [None, 3, 4]]
sage: t2_new
[[None, 2, 3], [1, 4], [5]]
sage: t1_new.outer_shape() == t2.outer_shape()
True
sage: t2_new.inner_shape() == t1.inner_shape()
True
```

```
>>> from sage.all import *
>>> t1 = SkewTableau([[None, Integer(1), Integer(2)], [Integer(3),
↪Integer(4)]])
>>> t2 = SkewTableau([[None, None, None, Integer(3)], [None, None,
↪Integer(4)], [Integer(1), Integer(2), Integer(5)]])
>>> (t2_new, t1_new) = t1.shuffle(t2)
>>> t1_new
[[None, None, None, 2], [None, None, 1], [None, 3, 4]]
>>> t2_new
[[None, 2, 3], [1, 4], [5]]
>>> t1_new.outer_shape() == t2.outer_shape()
True
>>> t2_new.inner_shape() == t1.inner_shape()
True
```

Shuffling is an involution:

```
sage: t1 = SkewTableau([[None, 1, 2], [3, 4]])
sage: t2 = SkewTableau([[None, None, None, 3], [None, None, 4], [1, 2, 5]])
sage: sh = lambda x,y : x.shuffle(y)
sage: (t1, t2) == sh(*sh(t1, t2))
True
```

```
>>> from sage.all import *
>>> t1 = SkewTableau([[None, Integer(1), Integer(2)], [Integer(3),
↪Integer(4)]])
>>> t2 = SkewTableau([[None, None, None, Integer(3)], [None, None,
↪Integer(4)], [Integer(1), Integer(2), Integer(5)]])
>>> sh = lambda x,y : x.shuffle(y)
>>> (t1, t2) == sh(*sh(t1, t2))
True
```

Both tableaux must be standard:

```
sage: t1 = SkewTableau([[None, 1, 2], [2, 4]])
sage: t2 = SkewTableau([[None, None, None, 3], [None, None, 4], [1, 2, 5]])
sage: t1.shuffle(t2)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: the tableaux must be standard
sage: t1 = SkewTableau([[None, 1, 2], [3, 4]])
sage: t2 = SkewTableau([[None, None, None, 3], [None, None, 4], [1, 2, 6]])
sage: t1.shuffle(t2)
Traceback (most recent call last):
...
ValueError: the tableaux must be standard
```

```
>>> from sage.all import *
>>> t1 = SkewTableau([[None, Integer(1), Integer(2)], [Integer(2),␣
↪Integer(4)]])
>>> t2 = SkewTableau([[None, None, None, Integer(3)], [None, None,␣
↪Integer(4)], [Integer(1), Integer(2), Integer(5)]])
>>> t1.shuffle(t2)
Traceback (most recent call last):
...
ValueError: the tableaux must be standard
>>> t1 = SkewTableau([[None, Integer(1), Integer(2)], [Integer(3),␣
↪Integer(4)]])
>>> t2 = SkewTableau([[None, None, None, Integer(3)], [None, None,␣
↪Integer(4)], [Integer(1), Integer(2), Integer(6)]])
>>> t1.shuffle(t2)
Traceback (most recent call last):
...
ValueError: the tableaux must be standard
```

The shapes (not just the nonempty cells) must be adjacent:

```
sage: t1 = SkewTableau([[None, None, None], [1]])
sage: t2 = SkewTableau([[None], [None], [1]])
sage: t1.shuffle(t2)
Traceback (most recent call last):
...
ValueError: the shapes must be adjacent
```

```
>>> from sage.all import *
>>> t1 = SkewTableau([[None, None, None], [Integer(1)]])
>>> t2 = SkewTableau([[None], [None], [Integer(1)]])
>>> t1.shuffle(t2)
Traceback (most recent call last):
...
ValueError: the shapes must be adjacent
```

size()

Return the number of cells in `self`.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).size()
4
```

(continues on next page)

(continued from previous page)

```
sage: SkewTableau([[None, 2], [1, 3]]).size()
3
```

```
>>> from sage.all import *
>>> SkewTableau([[None, Integer(2), Integer(4)], [None, Integer(3)],␣
↪[Integer(1)]]).size()
4
>>> SkewTableau([[None, Integer(2)], [Integer(1), Integer(3)]]).size()
3
```

slide(*corner=None*, *return_vacated=False*)

Apply a jeu de taquin slide to `self` on the specified inner corner and return the resulting tableau.

If no corner is given, the topmost inner corner is chosen.

The optional parameter `return_vacated=True` causes the output to be the pair `(t, (i, j))` where `t` is the new tableau and `(i, j)` are the coordinates of the vacated square.

See [Ful1997] p12-13.

EXAMPLES:

```
sage: st = SkewTableau([[None, None, None, None, 2], [None, None, None, None,␣
↪6], [None, 2, 4, 4], [2, 3, 6], [5, 5]])
sage: st.slide((2, 0))
[[None, None, None, None, 2], [None, None, None, None, 6], [2, 2, 4, 4], [3,␣
↪5, 6], [5]]
sage: st2 = SkewTableau([[None, None, 3], [None, 2, 4], [1, 5]])
sage: st2.slide((1, 0), True)
([[None, None, 3], [1, 2, 4], [5]], (2, 1))
```

```
>>> from sage.all import *
>>> st = SkewTableau([[None, None, None, None, Integer(2)], [None, None, None,
↪ None, Integer(6)], [None, Integer(2), Integer(4), Integer(4)], [Integer(2),
↪ Integer(3), Integer(6)], [Integer(5), Integer(5)]])
>>> st.slide((Integer(2), Integer(0)))
[[None, None, None, None, 2], [None, None, None, None, 6], [2, 2, 4, 4], [3,␣
↪5, 6], [5]]
>>> st2 = SkewTableau([[None, None, Integer(3)], [None, Integer(2),␣
↪Integer(4)], [Integer(1), Integer(5)]])
>>> st2.slide((Integer(1), Integer(0)), True)
([[None, None, 3], [1, 2, 4], [5]], (2, 1))
```

standardization(*check=True*)

Return the standardization of `self`, assuming `self` is a semistandard skew tableau.

The standardization of a semistandard skew tableau T is the standard skew tableau st(T) of the same shape as T whose reversed reading word is the standardization of the reversed reading word of T.

The standardization of a word w can be formed by replacing all 1's in w by $1, 2, \ldots, k_1$ from left to right, all 2's in w by $k_1 + 1, k_1 + 2, \ldots, k_2$, and repeating for all letters that appear in w. See also `Word.standard_permutation()`.

INPUT:

- `check` – (default: `True`) check to make sure `self` is semistandard; set to `False` to avoid this check

EXAMPLES:

```
sage: t = SkewTableau([[None,None,3,4,7,19],[None,4,4,8],[None,5,16,17],
↪[None],[2],[3]])
sage: t.standardization()
[[None, None, 3, 6, 8, 12], [None, 4, 5, 9], [None, 7, 10, 11], [None], [1],␣
↪[2]]
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None,None,Integer(3),Integer(4),Integer(7),Integer(19)],
↪[None,Integer(4),Integer(4),Integer(8)],[None,Integer(5),Integer(16),
↪Integer(17)],[None],[Integer(2)],[Integer(3)]])
>>> t.standardization()
[[None, None, 3, 6, 8, 12], [None, 4, 5, 9], [None, 7, 10, 11], [None], [1],␣
↪[2]]
```

Standard skew tableaux are fixed under standardization:

```
sage: p = Partition([4,3,3,2])
sage: q = Partitions(3).random_element()                                    #␣
↪needs sage.libs.flint
sage: all(t == t.standardization()                                          #␣
↪needs sage.libs.flint
....:     for t in StandardSkewTableaux([p, q]))
True
```

```
>>> from sage.all import *
>>> p = Partition([Integer(4),Integer(3),Integer(3),Integer(2)])
>>> q = Partitions(Integer(3)).random_element()                              ␣
↪     # needs sage.libs.flint
>>> all(t == t.standardization()                                          #␣
↪needs sage.libs.flint
...     for t in StandardSkewTableaux([p, q]))
True
```

The reading word of the standardization is the standardization of the reading word:

```
sage: t = SkewTableau([[None,3,4,4],[None,6,10],[7,7,11],[18]])
sage: t.to_word().standard_permutation() == t.standardization().to_
↪permutation()
True
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None,Integer(3),Integer(4),Integer(4)],[None,Integer(6),
↪Integer(10)],[Integer(7),Integer(7),Integer(11)],[Integer(18)]])
>>> t.to_word().standard_permutation() == t.standardization().to_permutation()
True
```

to_chain(*max_entry=None*)

Return the chain of partitions corresponding to the (semi)standard skew tableau `self`.

The optional keyword parameter `max_entry` can be used to customize the length of the chain. Specifically, if this parameter is set to a nonnegative integer `n`, then the chain is constructed from the positions of the letters $1, 2, \ldots, n$ in the tableau.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2],[3]]).to_chain()
[[1], [2], [2, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain()
[[1], [2, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain(max_entry=2)
[[1], [2, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain(max_entry=3)
[[1], [2, 1], [2, 1, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain(max_entry=1)
[[1], [2, 1]]
sage: SkewTableau([[None,None,2],[None,3],[None,5]]).to_chain(max_entry=6)
[[2, 1, 1], [2, 1, 1], [3, 1, 1], [3, 2, 1], [3, 2, 1], [3, 2, 2], [3, 2, 2]]
sage: SkewTableau([]).to_chain()
[[]]
sage: SkewTableau([]).to_chain(max_entry=1)
[[], []]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(1)],[Integer(2)],[Integer(3)]]).to_chain()
[[1], [2], [2, 1], [2, 1, 1]]
>>> SkewTableau([[None,Integer(1)],[Integer(1)],[Integer(2)]]).to_chain()
[[1], [2, 1], [2, 1, 1]]
>>> SkewTableau([[None,Integer(1)],[Integer(1)],[Integer(2)]]).to_chain(max_
↪entry=Integer(2))
[[1], [2, 1], [2, 1, 1]]
>>> SkewTableau([[None,Integer(1)],[Integer(1)],[Integer(2)]]).to_chain(max_
↪entry=Integer(3))
[[1], [2, 1], [2, 1, 1], [2, 1, 1]]
>>> SkewTableau([[None,Integer(1)],[Integer(1)],[Integer(2)]]).to_chain(max_
↪entry=Integer(1))
[[1], [2, 1]]
>>> SkewTableau([[None,None,Integer(2)],[None,Integer(3)],[None,Integer(5)]]).
↪to_chain(max_entry=Integer(6))
[[2, 1, 1], [2, 1, 1], [3, 1, 1], [3, 2, 1], [3, 2, 1], [3, 2, 2], [3, 2, 2]]
>>> SkewTableau([]).to_chain()
[[]]
>>> SkewTableau([]).to_chain(max_entry=Integer(1))
[[], []]
```

to_expr()

The first list in a result corresponds to the inner partition of the skew shape. The second list is a list of the rows in the skew tableau read from the bottom up.

Provided for compatibility with MuPAD-Combinat. In MuPAD-Combinat, if `t` is a skew tableau, then to_expr gives the same result as `expr(t)` would give in MuPAD-Combinat.

EXAMPLES:

```
sage: SkewTableau([[None,1,1,3],[None,2,2],[1]]).to_expr()
[[1, 1], [[1], [2, 2], [1, 1, 3]]]
sage: SkewTableau([]).to_expr()
[[], []]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(1),Integer(1),Integer(3)],[None,Integer(2),
↪Integer(2)],[Integer(1)]]).to_expr()
[[1, 1], [[1], [2, 2], [1, 1, 3]]]
>>> SkewTableau([]).to_expr()
[[], []]
```

to_list()

Return a (mutable) list representation of `self`.

EXAMPLES:

```
sage: stlist = [[None, None, 3], [None, 1, 3], [2, 2]]
sage: st = SkewTableau(stlist)
sage: st.to_list()
[[None, None, 3], [None, 1, 3], [2, 2]]
sage: st.to_list() == stlist
True
```

```
>>> from sage.all import *
>>> stlist = [[None, None, Integer(3)], [None, Integer(1), Integer(3)],␣
↪[Integer(2), Integer(2)]]
>>> st = SkewTableau(stlist)
>>> st.to_list()
[[None, None, 3], [None, 1, 3], [2, 2]]
>>> st.to_list() == stlist
True
```

to_permutation()

Return a permutation with the entries of `self` obtained by reading `self` row by row, from the bottommost to the topmost row, with each row being read from left to right, in English convention. See *to_word_by_row()*.

EXAMPLES:

```
sage: SkewTableau([[None,2],[3,4],[None],[1]]).to_permutation()
[1, 3, 4, 2]
sage: SkewTableau([[None,2],[None,4],[1],[3]]).to_permutation()
[3, 1, 4, 2]
sage: SkewTableau([[None]]).to_permutation()
[]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(2)],[Integer(3),Integer(4)],[None],
↪[Integer(1)]]).to_permutation()
[1, 3, 4, 2]
>>> SkewTableau([[None,Integer(2)],[None,Integer(4)],[Integer(1)],
↪[Integer(3)]]).to_permutation()
[3, 1, 4, 2]
>>> SkewTableau([[None]]).to_permutation()
[]
```

to_ribbon(*check_input=True*)

Return `self` as a ribbon-shaped tableau (*RibbonShapedTableau*), provided that the shape of `self` is a ribbon.

INPUT:

- `check_input` – boolean (default: `True`); whether or not to check that `self` indeed has ribbon shape

EXAMPLES:

```
sage: SkewTableau([[None,1],[2,3]]).to_ribbon()
[[None, 1], [2, 3]]
```

```
>>> from sage.all import *
>>> SkewTableau([[None,Integer(1)],[Integer(2),Integer(3)]]).to_ribbon()
[[None, 1], [2, 3]]
```

to_tableau()

Return a tableau with the same filling. This only works if the inner shape of the skew tableau has size zero.

EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).to_tableau()
[[1, 2], [3, 4]]
```

```
>>> from sage.all import *
>>> SkewTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).to_
↪tableau()
[[1, 2], [3, 4]]
```

to_word()

Return a word obtained from a row reading of `self`.

This is the word obtained by concatenating the rows from the bottommost one (in English notation) to the topmost one.

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.pp()
  .  1
  2  3
sage: s.to_word_by_row()
word: 231
sage: s = SkewTableau([[None, 2, 4], [None, 3], [1]])
sage: s.pp()
  .  2  4
  .  3
  1
sage: s.to_word_by_row()
word: 1324
```

```
>>> from sage.all import *
>>> s = SkewTableau([[None,Integer(1)],[Integer(2),Integer(3)]])
>>> s.pp()
  .  1
  2  3
```

(continues on next page)

(continued from previous page)

```
>>> s.to_word_by_row()
word: 231
>>> s = SkewTableau([[None, Integer(2), Integer(4)], [None, Integer(3)],
↪[Integer(1)]])
>>> s.pp()
  .  2  4
  .  3
  1
>>> s.to_word_by_row()
word: 1324
```

to_word_by_column()

Return the word obtained from a column reading of the skew tableau.

This is the word obtained by concatenating the columns from the rightmost one (in English notation) to the leftmost one.

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.pp()
  .  1
  2  3
sage: s.to_word_by_column()
word: 132
```

```
>>> from sage.all import *
>>> s = SkewTableau([[None,Integer(1)],[Integer(2),Integer(3)]])
>>> s.pp()
  .  1
  2  3
>>> s.to_word_by_column()
word: 132
```

```
sage: s = SkewTableau([[None, 2, 4], [None, 3], [1]])
sage: s.pp()
.  2  4
.  3
1
sage: s.to_word_by_column()
word: 4231
```

```
>>> from sage.all import *
>>> s = SkewTableau([[None, Integer(2), Integer(4)], [None, Integer(3)],
↪[Integer(1)]])
>>> s.pp()
.  2  4
.  3
1
>>> s.to_word_by_column()
word: 4231
```

to_word_by_row()

Return a word obtained from a row reading of `self`.

This is the word obtained by concatenating the rows from the bottommost one (in English notation) to the topmost one.

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.pp()
  .  1
  2  3
sage: s.to_word_by_row()
word: 231
sage: s = SkewTableau([[None, 2, 4], [None, 3], [1]])
sage: s.pp()
  .  2  4
  .  3
  1
sage: s.to_word_by_row()
word: 1324
```

```
>>> from sage.all import *
>>> s = SkewTableau([[None,Integer(1)],[Integer(2),Integer(3)]])
>>> s.pp()
  .  1
  2  3
>>> s.to_word_by_row()
word: 231
>>> s = SkewTableau([[None, Integer(2), Integer(4)], [None, Integer(3)],
↪[Integer(1)]])
>>> s.pp()
  .  2  4
  .  3
  1
>>> s.to_word_by_row()
word: 1324
```

weight()

Return the weight (aka evaluation) of the tableau `self`. Trailing zeroes are omitted when returning the weight.

The weight of a skew tableau T is the sequence $(a_1, a_2, a_3, \ldots)$, where a_k is the number of entries of T equal to k. This sequence contains only finitely many nonzero entries.

The weight of a skew tableau T is the same as the weight of the reading word of T, for any reading order.

evaluation() is a synonym for this method.

EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).weight()
[1, 1, 1, 1]

sage: SkewTableau([[None,2],[None,4],[None,5],[None]]).weight()
[0, 1, 0, 1, 1]
```

(continues on next page)

(continued from previous page)

```
sage: SkewTableau([]).weight()
[]

sage: SkewTableau([[None,None,None], [None]]).weight()
[]

sage: SkewTableau([[None,3,4],[None,6,7],[4,8],[5,13],[6],[7]]).weight()
[0, 0, 1, 2, 1, 2, 2, 1, 0, 0, 0, 0, 1]
```

```
>>> from sage.all import *
>>> SkewTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).weight()
[1, 1, 1, 1]

>>> SkewTableau([[None,Integer(2)],[None,Integer(4)],[None,Integer(5)],
↪[None]]).weight()
[0, 1, 0, 1, 1]

>>> SkewTableau([]).weight()
[]

>>> SkewTableau([[None,None,None], [None]]).weight()
[]

>>> SkewTableau([[None,Integer(3),Integer(4)],[None,Integer(6),Integer(7)],
↪[Integer(4),Integer(8)],[Integer(5),Integer(13)],[Integer(6)],
↪[Integer(7)]]).weight()
[0, 0, 1, 2, 1, 2, 2, 1, 0, 0, 0, 0, 1]
```

class sage.combinat.skew_tableau.**SkewTableau_class**(*parent*, *st*)

Bases: *SkewTableau*

This exists solely for unpickling `SkewTableau_class` objects.

class sage.combinat.skew_tableau.**SkewTableaux**(*category=None*)

Bases: `UniqueRepresentation`, `Parent`

Class of all skew tableaux.

Element

alias of *SkewTableau*

from_chain(*chain*)

Return the tableau corresponding to the chain of partitions.

EXAMPLES:

```
sage: SkewTableaux().from_chain([[1,1],[2,1],[3,1],[3,2],[3,3],[3,3,1]])
[[None, 1, 2], [None, 3, 4], [5]]
```

```
>>> from sage.all import *
>>> SkewTableaux().from_chain([[Integer(1),Integer(1)],[Integer(2),
↪Integer(1)],[Integer(3),Integer(1)],[Integer(3),Integer(2)],[Integer(3),
↪Integer(3)],[Integer(3),Integer(3),Integer(1)]])
[[None, 1, 2], [None, 3, 4], [5]]
```

from_expr(*expr*)

Return a *SkewTableau* from a MuPAD-Combinat expr for a skew tableau.

The first list in `expr` is the inner shape of the skew tableau. The second list are the entries in the rows of the skew tableau from bottom to top.

Provided primarily for compatibility with MuPAD-Combinat.

EXAMPLES:

```
sage: SkewTableaux().from_expr([[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

```
>>> from sage.all import *
>>> SkewTableaux().from_expr([[Integer(1),Integer(1)],[[Integer(5)],
→[Integer(3),Integer(4)],[Integer(1),Integer(2)]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

from_shape_and_word(*shape*, *word*)

Return the skew tableau corresponding to the skew partition `shape` and the word `word` obtained from the row reading.

EXAMPLES:

```
sage: t = SkewTableau([[None, 1, 3], [None, 2], [4]])
sage: shape = t.shape()
sage: word  = t.to_word()
sage: SkewTableaux().from_shape_and_word(shape, word)
[[None, 1, 3], [None, 2], [4]]
```

```
>>> from sage.all import *
>>> t = SkewTableau([[None, Integer(1), Integer(3)], [None, Integer(2)],␣
→[Integer(4)]])
>>> shape = t.shape()
>>> word  = t.to_word()
>>> SkewTableaux().from_shape_and_word(shape, word)
[[None, 1, 3], [None, 2], [4]]
```

options = Current options for Tableaux - ascii_art: repr - convention: English - display: list - latex: diagram

class sage.combinat.skew_tableau.**StandardSkewTableaux**(*category=None*)

Bases: *SkewTableaux*

Standard skew tableaux.

EXAMPLES:

```
sage: S = StandardSkewTableaux(); S
Standard skew tableaux
sage: S.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> S = StandardSkewTableaux(); S
```

(continues on next page)

(continued from previous page)

```
Standard skew tableaux
>>> S.cardinality()
+Infinity
```

```
sage: S = StandardSkewTableaux(2); S
Standard skew tableaux of size 2
sage: S.cardinality()                                                           #␣
↪needs sage.modules
4
```

```
>>> from sage.all import *
>>> S = StandardSkewTableaux(Integer(2)); S
Standard skew tableaux of size 2
>>> S.cardinality()                                                         #␣
↪needs sage.modules
4
```

```
sage: # needs sage.graphs sage.modules
sage: StandardSkewTableaux([[3, 2, 1], [1, 1]]).list()
[[[None, 2, 3], [None, 4], [1]],
 [[None, 1, 2], [None, 3], [4]],
 [[None, 1, 2], [None, 4], [3]],
 [[None, 1, 3], [None, 4], [2]],
 [[None, 1, 4], [None, 3], [2]],
 [[None, 1, 4], [None, 2], [3]],
 [[None, 1, 3], [None, 2], [4]],
 [[None, 2, 4], [None, 3], [1]]]
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.modules
>>> StandardSkewTableaux([[Integer(3), Integer(2), Integer(1)], [Integer(1),␣
↪Integer(1)]]).list()
[[[None, 2, 3], [None, 4], [1]],
 [[None, 1, 2], [None, 3], [4]],
 [[None, 1, 2], [None, 4], [3]],
 [[None, 1, 3], [None, 4], [2]],
 [[None, 1, 4], [None, 3], [2]],
 [[None, 1, 4], [None, 2], [3]],
 [[None, 1, 3], [None, 2], [4]],
 [[None, 2, 4], [None, 3], [1]]]
```

class sage.combinat.skew_tableau.**StandardSkewTableaux_all**

Bases: *StandardSkewTableaux*

Class of all standard skew tableaux.

class sage.combinat.skew_tableau.**StandardSkewTableaux_shape**(*skp*)

Bases: *StandardSkewTableaux*

Standard skew tableaux of a fixed skew shape λ/μ.

cardinality()

Return the number of standard skew tableaux with shape of the skew partition skp. This uses a formula due to Aitken (see Cor. 7.16.3 of [Sta-EC2]).

EXAMPLES:

```
sage: StandardSkewTableaux([[3, 2, 1], [1, 1]]).cardinality()              #␣
↪needs sage.modules
8
```

```
>>> from sage.all import *
>>> StandardSkewTableaux([[Integer(3), Integer(2), Integer(1)], [Integer(1),␣
↪Integer(1)]]).cardinality()              # needs sage.modules
8
```

class sage.combinat.skew_tableau.**StandardSkewTableaux_size**(*n*)

Bases: *StandardSkewTableaux*

Standard skew tableaux of a fixed size n.

cardinality()

EXAMPLES:

```
sage: # needs sage.modules
sage: StandardSkewTableaux(1).cardinality()
1
sage: StandardSkewTableaux(2).cardinality()
4
sage: StandardSkewTableaux(3).cardinality()
24
sage: StandardSkewTableaux(4).cardinality()
194
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> StandardSkewTableaux(Integer(1)).cardinality()
1
>>> StandardSkewTableaux(Integer(2)).cardinality()
4
>>> StandardSkewTableaux(Integer(3)).cardinality()
24
>>> StandardSkewTableaux(Integer(4)).cardinality()
194
```

5.1.314 Functions that compute some of the sequences in Sloane's tables

EXAMPLES:

Type `sloane.[tab]` to see a list of the sequences that are defined.

```
sage: a = sloane.A000005; a
The integer sequence tau(n), which is the number of divisors of n.
sage: a(1)
1
sage: a(6)
4
```

(continues on next page)

(continued from previous page)

```
sage: a(100)
9
```

```
>>> from sage.all import *
>>> a = sloane.A000005; a
The integer sequence tau(n), which is the number of divisors of n.
>>> a(Integer(1))
1
>>> a(Integer(6))
4
>>> a(Integer(100))
9
```

Type `d._eval??` to see how the function that computes an individual term of the sequence is implemented.

The input must be a positive integer:

```
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

```
>>> from sage.all import *
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1)/Integer(3))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

You can also change how a sequence prints:

```
sage: a = sloane.A000005; a
The integer sequence tau(n), which is the number of divisors of n.
sage: a.rename('(..., tau(n), ...)')
sage: a
(..., tau(n), ...)
sage: a.reset_name()
sage: a
The integer sequence tau(n), which is the number of divisors of n.
```

```
>>> from sage.all import *
>>> a = sloane.A000005; a
The integer sequence tau(n), which is the number of divisors of n.
>>> a.rename('(..., tau(n), ...)')
>>> a
```

(continues on next page)

(continued from previous page)

```
(..., tau(n), ...)
>>> a.reset_name()
>>> a
The integer sequence tau(n), which is the number of divisors of n.
```

See also:

- If you want to get more informations relative to a sequence (references, links, examples, programs, …), you can use the On-Line Encyclopedia of Integer Sequences provided by the `OEIS` module.
- If you plan to do a lot of automatic searches for subsequences, you should consider installing `SloaneEncyclopedia`, a local partial copy of the OEIS.

AUTHORS:

- William Stein: framework
- Jaap Spies: most sequences
- Nick Alexander: updated framework

class `sage.combinat.sloane_functions.`**`A000001`**

Bases: *`SloaneSequence`*

Number of groups of order n.

INPUT:

- `n` – positive integer

OUTPUT: integer

EXAMPLES:

```
sage: a = sloane.A000001;a
Number of groups of order n.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
1
sage: a(9)
2
sage: a.list(16)
[1, 1, 1, 2, 1, 2, 1, 5, 2, 2, 1, 5, 1, 2, 1, 14]
sage: a(60)
13
```

```
>>> from sage.all import *
>>> a = sloane.A000001;a
Number of groups of order n.
>>> a(Integer(0))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
1
>>> a(Integer(9))
2
>>> a.list(Integer(16))
[1, 1, 1, 2, 1, 2, 1, 5, 2, 2, 1, 5, 1, 2, 1, 14]
>>> a(Integer(60))
13
```

AUTHORS:

- Jaap Spies (2007-02-04)

class sage.combinat.sloane_functions.**A000004**

Bases: *SloaneSequence*

The zero sequence.

INPUT:

- n – nonnegative integer

EXAMPLES:

```
sage: a = sloane.A000004; a
The zero sequence.
sage: a(1)
0
sage: a(2007)
0
sage: a.list(12)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> a = sloane.A000004; a
The zero sequence.
>>> a(Integer(1))
0
>>> a(Integer(2007))
0
>>> a.list(Integer(12))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

AUTHORS:

- Jaap Spies (2006-12-10)

class sage.combinat.sloane_functions.**A000005**

Bases: *SloaneSequence*

The sequence $tau(n)$, which is the number of divisors of n.

This sequence is also denoted $d(n)$ (also called $\tau(n)$ or $\sigma_0(n)$), the number of divisors of n.

INPUT:

- n – positive integer

EXAMPLES:

```
sage: d = sloane.A000005; d
The integer sequence tau(n), which is the number of divisors of n.
sage: d(1)
1
sage: d(6)
4
sage: d(51)
4
sage: d(100)
9
sage: d(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: d.list(10)
[1, 2, 2, 3, 2, 4, 2, 4, 3, 4]
```

```
>>> from sage.all import *
>>> d = sloane.A000005; d
The integer sequence tau(n), which is the number of divisors of n.
>>> d(Integer(1))
1
>>> d(Integer(6))
4
>>> d(Integer(51))
4
>>> d(Integer(100))
9
>>> d(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> d.list(Integer(10))
[1, 2, 2, 3, 2, 4, 2, 4, 3, 4]
```

AUTHORS:

- Jaap Spies (2006-12-10)
- William Stein (2007-01-08)

class sage.combinat.sloane_functions.**A000007**

Bases: *SloaneSequence*

The characteristic function of 0: $a(n) = 0^n$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000007;a
The characteristic function of 0: a(n) = 0^n.
sage: a(0)
1
sage: a(2)
0
sage: a(12)
0
sage: a.list(12)
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> a = sloane.A000007;a
The characteristic function of 0: a(n) = 0^n.
>>> a(Integer(0))
1
>>> a(Integer(2))
0
>>> a(Integer(12))
0
>>> a.list(Integer(12))
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000008**

Bases: *SloaneSequence*

Number of ways of making change for n cents using coins of 1, 2, 5, 10 cents.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000008;a
Number of ways of making change for n cents using coins of 1, 2, 5, 10 cents.
sage: a(0)
1
sage: a(1)
1
sage: a(13)
16
sage: a.list(14)
[1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 11, 12, 15, 16]
```

```
>>> from sage.all import *
>>> a = sloane.A000008;a
```

(continues on next page)

(continued from previous page)

```
Number of ways of making change for n cents using coins of 1, 2, 5, 10 cents.
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(13))
16
>>> a.list(Integer(14))
[1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 11, 12, 15, 16]
```

AUTHOR:

- J. Gaski (2009-05-29)

class sage.combinat.sloane_functions.**A000009**

Bases: *SloaneSequence*

Number of partitions of n into odd parts.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000009;a
Number of partitions of n into odd parts.
sage: a(0)
1
sage: a(1)
1
sage: a(13)
18
sage: a.list(14)
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

```
>>> from sage.all import *
>>> a = sloane.A000009;a
Number of partitions of n into odd parts.
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(13))
18
>>> a.list(Integer(14))
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

AUTHOR:

- Jaap Spies (2007-01-30)

cf()

EXAMPLES:

```
sage: it = sloane.A000009.cf()
sage: [next(it) for i in range(14)]
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

```
>>> from sage.all import *
>>> it = sloane.A000009.cf()
>>> [next(it) for i in range(Integer(14))]
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

list(n)

EXAMPLES:

```
sage: sloane.A000009.list(14)
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

```
>>> from sage.all import *
>>> sloane.A000009.list(Integer(14))
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

class sage.combinat.sloane_functions.**A000010**

Bases: *SloaneSequence*

The integer sequence A000010 is Euler's totient function.

Number of positive integers $i < n$ that are relative prime to n. Number of totatives of n.

Euler totient function $\phi(n)$: count numbers n and prime to n. `euler_phi` is a standard Sage function implemented in PARI.

INPUT:

- `n` – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000010; a
Euler's totient function
sage: a(1)
1
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(11)
10
sage: a.list(12)
[1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

```
>>> from sage.all import *
>>> a = sloane.A000010; a
Euler's totient function
>>> a(Integer(1))
1
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(11))
10
>>> a.list(Integer(12))
[1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4]
>>> a(Integer(1)/Integer(3))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000012**

Bases: *SloaneSequence*

The all 1s sequence.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000012; a
The all 1s sequence.
sage: a(1)
1
sage: a(2007)
1
sage: a.list(12)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> a = sloane.A000012; a
The all 1s sequence.
>>> a(Integer(1))
1
>>> a(Integer(2007))
1
>>> a.list(Integer(12))
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000015**

Bases: *SloaneSequence*

Smallest prime power $\geq n$ (where 1 is considered a prime power).

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000015; a
Smallest prime power >= n.
sage: a(1)
1
sage: a(8)
8
sage: a(305)
307
sage: a(-4)
Traceback (most recent call last):
...
ValueError: input n (=-4) must be a positive integer
sage: a.list(12)
[1, 2, 3, 4, 5, 7, 7, 8, 9, 11, 11, 13]
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

```
>>> from sage.all import *
>>> a = sloane.A000015; a
Smallest prime power >= n.
>>> a(Integer(1))
1
>>> a(Integer(8))
8
>>> a(Integer(305))
307
>>> a(-Integer(4))
Traceback (most recent call last):
...
ValueError: input n (=-4) must be a positive integer
>>> a.list(Integer(12))
[1, 2, 3, 4, 5, 7, 7, 8, 9, 11, 11, 13]
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000016**

Bases: *SloaneSequence*

Sloane's A000016.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000016; a
Sloane's A000016.
sage: a(1)
1
sage: a(0)
1
sage: a(8)
16
sage: a(75)
251859545753048193000
sage: a(-4)
Traceback (most recent call last):
...
ValueError: input n (=-4) must be an integer >= 0
sage: a.list(12)
[1, 1, 1, 2, 2, 4, 6, 10, 16, 30, 52, 94]
```

```
>>> from sage.all import *
>>> a = sloane.A000016; a
Sloane's A000016.
>>> a(Integer(1))
1
>>> a(Integer(0))
1
>>> a(Integer(8))
16
>>> a(Integer(75))
251859545753048193000
>>> a(-Integer(4))
Traceback (most recent call last):
...
ValueError: input n (=-4) must be an integer >= 0
>>> a.list(Integer(12))
[1, 1, 1, 2, 2, 4, 6, 10, 16, 30, 52, 94]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000027**

Bases: *SloaneSequence*

The natural numbers. Also called the whole numbers, the counting numbers or the positive integers.

The following examples are tests of SloaneSequence more than A000027.

EXAMPLES:

```
sage: s = sloane.A000027; s
The natural numbers.
sage: s(10)
10
```

```
>>> from sage.all import *
>>> s = sloane.A000027; s
The natural numbers.
>>> s(Integer(10))
10
```

Index n is interpreted as _eval(n):

```
sage: s[10]
10
```

```
>>> from sage.all import *
>>> s[Integer(10)]
10
```

Slices are interpreted with absolute offsets, so the following returns the terms of the sequence up to but not including the third term:

```
sage: s[:3]
[1, 2]
sage: s[3:6]
[3, 4, 5]
sage: s.list(5)
[1, 2, 3, 4, 5]
```

```
>>> from sage.all import *
>>> s[:Integer(3)]
[1, 2]
>>> s[Integer(3):Integer(6)]
[3, 4, 5]
>>> s.list(Integer(5))
[1, 2, 3, 4, 5]
```

link = 'http://oeis.org/classic/A000027'

class sage.combinat.sloane_functions.**A000030**

Bases: *SloaneSequence*

Initial digit of n.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000030; a
Initial digit of n
sage: a(0)
0
sage: a(1)
1
sage: a(8)
8
sage: a(454)
4
sage: a(-4)
Traceback (most recent call last):
...
ValueError: input n (=-4) must be an integer >= 0
sage: a.list(12)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1]
```

```
>>> from sage.all import *
>>> a = sloane.A000030; a
Initial digit of n
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(8))
8
>>> a(Integer(454))
4
>>> a(-Integer(4))
Traceback (most recent call last):
...
ValueError: input n (=-4) must be an integer >= 0
>>> a.list(Integer(12))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000032**

Bases: *SloaneSequence*

Lucas numbers (beginning at 2): $L(n) = L(n-1) + L(n-2)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000032; a
Lucas numbers (beginning at 2): L(n) = L(n-1) + L(n-2).
sage: a(0)
2
```

(continues on next page)

(continued from previous page)

```
sage: a(1)
1
sage: a(8)
47
sage: a(200)
627376215338105766356982006981782561278127
sage: a(-4)
Traceback (most recent call last):
...
ValueError: input n (=-4) must be an integer >= 0
sage: a.list(12)
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

```
>>> from sage.all import *
>>> a = sloane.A000032; a
Lucas numbers (beginning at 2): L(n) = L(n-1) + L(n-2).
>>> a(Integer(0))
2
>>> a(Integer(1))
1
>>> a(Integer(8))
47
>>> a(Integer(200))
627376215338105766356982006981782561278127
>>> a(-Integer(4))
Traceback (most recent call last):
...
ValueError: input n (=-4) must be an integer >= 0
>>> a.list(Integer(12))
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000035**

Bases: *SloaneSequence*

A simple periodic sequence.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000035;a
A simple periodic sequence.
sage: a(0.0)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
sage: a(1)
```

(continues on next page)

(continued from previous page)

```
1
sage: a(2)
0
sage: a(9)
1
sage: a.list(10)
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

```
>>> from sage.all import *
>>> a = sloane.A000035;a
A simple periodic sequence.
>>> a(RealNumber('0.0'))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
>>> a(Integer(1))
1
>>> a(Integer(2))
0
>>> a(Integer(9))
1
>>> a.list(Integer(10))
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A000040**

Bases: *SloaneSequence*

The prime numbers.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000040; a
The prime numbers.
sage: a(1)
2
sage: a(8)
19
sage: a(305)
2011
sage: a.list(12)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

```
>>> from sage.all import *
>>> a = sloane.A000040; a
The prime numbers.
>>> a(Integer(1))
2
>>> a(Integer(8))
19
>>> a(Integer(305))
2011
>>> a.list(Integer(12))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-17)

class sage.combinat.sloane_functions.**A000041**

Bases: *SloaneSequence*

$a(n)$ = number of partitions of n (the partition numbers).

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000041;a
a(n) = number of partitions of n (the partition numbers).
sage: a(0)
1
sage: a(2)
2
sage: a(8)
22
sage: a(200)
3972999029388
sage: a.list(9)
[1, 1, 2, 3, 5, 7, 11, 15, 22]
```

```
>>> from sage.all import *
>>> a = sloane.A000041;a
a(n) = number of partitions of n (the partition numbers).
>>> a(Integer(0))
1
>>> a(Integer(2))
2
>>> a(Integer(8))
22
>>> a(Integer(200))
```

(continues on next page)

(continued from previous page)

```
3972999029388
>>> a.list(Integer(9))
[1, 1, 2, 3, 5, 7, 11, 15, 22]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000043**

Bases: *SloaneSequence*

Primes p such that $2^p - 1$ is prime. $2^p - 1$ is then called a Mersenne prime.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000043;a
Primes p such that 2^p - 1 is prime. 2^p - 1 is then called a Mersenne prime.
sage: a(1)
2
sage: a(2)
3
sage: a(39)
13466917
sage: a(40)
Traceback (most recent call last):
...
IndexError: list index out of range
sage: a.list(12)
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127]
```

```
>>> from sage.all import *
>>> a = sloane.A000043;a
Primes p such that 2^p - 1 is prime. 2^p - 1 is then called a Mersenne prime.
>>> a(Integer(1))
2
>>> a(Integer(2))
3
>>> a(Integer(39))
13466917
>>> a(Integer(40))
Traceback (most recent call last):
...
IndexError: list index out of range
>>> a.list(Integer(12))
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000045**

Bases: *SloaneSequence*

Sequence of Fibonacci numbers, offset 0,4.

REFERENCES:

- S. Plouffe, Project Gutenberg, The First 1001 Fibonacci Numbers, http://ibiblio.org/pub/docs/books/gutenberg/etext01/fbncc10.txt

We have one more. Our first Fibonacci number is 0.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000045; a
Fibonacci numbers with index n >= 0
sage: a(0)
0
sage: a(1)
1
sage: a.list(12)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

```
>>> from sage.all import *
>>> a = sloane.A000045; a
Fibonacci numbers with index n >= 0
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a.list(Integer(12))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> a(Integer(1)/Integer(3))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

AUTHORS:

- Jaap Spies (2007-01-13)

fib()

Return a generator over all Fibonacci numbers, starting with 0.

EXAMPLES:

```
sage: it = sloane.A000045.fib()
sage: [next(it) for i in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
>>> from sage.all import *
>>> it = sloane.A000045.fib()
>>> [next(it) for i in range(Integer(10))]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

list(*n*)

EXAMPLES:

```
sage: sloane.A000045.list(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
>>> from sage.all import *
>>> sloane.A000045.list(Integer(10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

class sage.combinat.sloane_functions.**A000069**

Bases: *SloaneSequence*

Odious numbers: odd number of 1s in binary expansion.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000069; a
Odious numbers: odd number of 1s in binary expansion.
sage: a(0)
1
sage: a(2)
4
sage: a.list(9)
[1, 2, 4, 7, 8, 11, 13, 14, 16]
```

```
>>> from sage.all import *
>>> a = sloane.A000069; a
Odious numbers: odd number of 1s in binary expansion.
>>> a(Integer(0))
1
>>> a(Integer(2))
4
>>> a.list(Integer(9))
[1, 2, 4, 7, 8, 11, 13, 14, 16]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A000073**

Bases: *SloaneSequence*

Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3). Starting with 0, 0, 1, …

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000073;a
Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3).
sage: a(0)
0
sage: a(1)
0
sage: a(2)
1
sage: a(11)
149
sage: a.list(12)
[0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149]
```

```
>>> from sage.all import *
>>> a = sloane.A000073;a
Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3).
>>> a(Integer(0))
0
>>> a(Integer(1))
0
>>> a(Integer(2))
1
>>> a(Integer(11))
149
>>> a.list(Integer(12))
[0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149]
```

AUTHORS:

- Jaap Spies (2007-01-19)

list(*n*)

EXAMPLES:

```
sage: sloane.A000073.list(10)
[0, 0, 1, 1, 2, 4, 7, 13, 24, 44]
```

```
>>> from sage.all import *
>>> sloane.A000073.list(Integer(10))
[0, 0, 1, 1, 2, 4, 7, 13, 24, 44]
```

class sage.combinat.sloane_functions.**A000079**

Bases: *SloaneSequence*

Powers of 2: $a(n) = 2^n$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000079;a
Powers of 2: a(n) = 2^n.
sage: a(0)
1
sage: a(2)
4
sage: a(8)
256
sage: a(100)
1267650600228229401496703205376
sage: a.list(9)
[1, 2, 4, 8, 16, 32, 64, 128, 256]
```

```
>>> from sage.all import *
>>> a = sloane.A000079;a
Powers of 2: a(n) = 2^n.
>>> a(Integer(0))
1
>>> a(Integer(2))
4
>>> a(Integer(8))
256
>>> a(Integer(100))
1267650600228229401496703205376
>>> a.list(Integer(9))
[1, 2, 4, 8, 16, 32, 64, 128, 256]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000085**

Bases: *SloaneSequence*

Number of self-inverse permutations on n letters, also known as involutions; number of Young tableaux with n cells.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000085;a
Number of self-inverse permutations on n letters.
sage: a(0)
1
sage: a(1)
1
sage: a(2)
2
sage: a(12)
140152
```

(continues on next page)

(continued from previous page)

```
sage: a.list(13)
[1, 1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, 140152]
```

```
>>> from sage.all import *
>>> a = sloane.A000085;a
Number of self-inverse permutations on n letters.
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(2))
2
>>> a(Integer(12))
140152
>>> a.list(Integer(13))
[1, 1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, 140152]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A000100**

Bases: *SloaneSequence*

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000100;a
Number of compositions of n in which the maximum part size is 3.
sage: a(0)
0
sage: a(1)
0
sage: a(2)
0
sage: a(3)
1
sage: a(11)
360
sage: a.list(12)
[0, 0, 0, 1, 2, 5, 11, 23, 47, 94, 185, 360]
```

```
>>> from sage.all import *
>>> a = sloane.A000100;a
Number of compositions of n in which the maximum part size is 3.
>>> a(Integer(0))
0
>>> a(Integer(1))
0
```

(continues on next page)

(continued from previous page)

```
>>> a(Integer(2))
0
>>> a(Integer(3))
1
>>> a(Integer(11))
360
>>> a.list(Integer(12))
[0, 0, 0, 1, 2, 5, 11, 23, 47, 94, 185, 360]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000108**

Bases: *SloaneSequence*

Catalan numbers: $C_n = \frac{\binom{2n}{n}}{n+1} = \frac{(2n)!}{n!(n+1)!}$.

Also called Segner numbers.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000108;a
Catalan numbers: C(n) = binomial(2n,n)/(n+1) = (2n)!/(n!(n+1)!). Also called
↪Segner numbers.
sage: a(0)
1
sage: a.offset
0
sage: a(8)
1430
sage: a(40)
2622127042276492108820
sage: a.list(9)
[1, 1, 2, 5, 14, 42, 132, 429, 1430]
```

```
>>> from sage.all import *
>>> a = sloane.A000108;a
Catalan numbers: C(n) = binomial(2n,n)/(n+1) = (2n)!/(n!(n+1)!). Also called
↪Segner numbers.
>>> a(Integer(0))
1
>>> a.offset
0
>>> a(Integer(8))
1430
>>> a(Integer(40))
2622127042276492108820
>>> a.list(Integer(9))
[1, 1, 2, 5, 14, 42, 132, 429, 1430]
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000110**

Bases: *ExponentialNumbers*

The sequence of Bell numbers.

The Bell number B_n counts the number of ways to put n distinguishable things into indistinguishable boxes such that no box is empty.

Let $S(n,k)$ denote the Stirling number of the second kind. Then

$$B_n = \sum k = 0^n S(n,k).$$

INPUT:

- n – nonnegative integer

OUTPUT: integer; B_n

EXAMPLES:

```
sage: a = sloane.A000110; a
Sequence of Bell numbers
sage: a.offset
0
sage: a(0)
1
sage: a(100)
47585391276764833658790768841387207826363669686825611466616334637559114497892442622
↪672724044217756306953557882560751
sage: a.list(10)
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

```
>>> from sage.all import *
>>> a = sloane.A000110; a
Sequence of Bell numbers
>>> a.offset
0
>>> a(Integer(0))
1
>>> a(Integer(100))
47585391276764833658790768841387207826363669686825611466616334637559114497892442622
↪672724044217756306953557882560751
>>> a.list(Integer(10))
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

AUTHORS:

- Nick Alexander

class sage.combinat.sloane_functions.**A000120**

Bases: *SloaneSequence*

1s-counting sequence: number of 1s in binary expansion of n.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000120;a
1s-counting sequence: number of 1s in binary expansion of n.
sage: a(0)
0
sage: a(2)
1
sage: a(12)
2
sage: a.list(12)
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3]
```

```
>>> from sage.all import *
>>> a = sloane.A000120;a
1s-counting sequence: number of 1s in binary expansion of n.
>>> a(Integer(0))
0
>>> a(Integer(2))
1
>>> a(Integer(12))
2
>>> a.list(Integer(12))
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3]
```

AUTHORS:

- Jaap Spies (2007-01-26)

f(*n*)

EXAMPLES:

```
sage: [sloane.A000120.f(n) for n in range(10)]
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2]
```

```
>>> from sage.all import *
>>> [sloane.A000120.f(n) for n in range(Integer(10))]
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2]
```

class sage.combinat.sloane_functions.**A000124**

Bases: *SloaneSequence*

Central polygonal numbers (the Lazy Caterer's sequence): $n(n+1)/2+1$.

Or, maximal number of pieces formed when slicing a pancake with n cuts.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000124;a
Central polygonal numbers (the Lazy Caterer's sequence): n(n+1)/2 + 1.
sage: a(0)
1
sage: a(1)
2
sage: a(2)
4
sage: a(9)
46
sage: a.list(10)
[1, 2, 4, 7, 11, 16, 22, 29, 37, 46]
```

```
>>> from sage.all import *
>>> a = sloane.A000124;a
Central polygonal numbers (the Lazy Caterer's sequence): n(n+1)/2 + 1.
>>> a(Integer(0))
1
>>> a(Integer(1))
2
>>> a(Integer(2))
4
>>> a(Integer(9))
46
>>> a.list(Integer(10))
[1, 2, 4, 7, 11, 16, 22, 29, 37, 46]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000129**

Bases: *RecurrenceSequence2*

Pell numbers: $a(0) = 0$, $a(1) = 1$; for $n > 1$, $a(n) = 2a(n-1) + a(n-2)$.

Denominators of continued fraction convergents to $\sqrt{2}$.

See also A001333

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000129;a
Pell numbers: a(0) = 0, a(1) = 1; for n > 1, a(n) = 2*a(n-1) + a(n-2).
sage: a(0)
0
sage: a(2)
2
sage: a(12)
13860
```

(continues on next page)

(continued from previous page)

```
sage: a.list(12)
[0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741]
```

```
>>> from sage.all import *
>>> a = sloane.A000129;a
Pell numbers: a(0) = 0, a(1) = 1; for n > 1, a(n) = 2*a(n-1) + a(n-2).
>>> a(Integer(0))
0
>>> a(Integer(2))
2
>>> a(Integer(12))
13860
>>> a.list(Integer(12))
[0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000142**

Bases: *SloaneSequence*

Factorial numbers: $n! = 1 \cdot 2 \cdot 3 \cdots n$.

Order of symmetric group S_n, number of permutations of n letters.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000142;a
Factorial numbers: n! = 1*2*3*4*...*n (order of symmetric group S_n, number of
→permutations of n letters).
sage: a(0)
1
sage: a(8)
40320
sage: a(40)
815915283247897734345611269596115894272000000000
sage: a.list(9)
[1, 1, 2, 6, 24, 120, 720, 5040, 40320]
```

```
>>> from sage.all import *
>>> a = sloane.A000142;a
Factorial numbers: n! = 1*2*3*4*...*n (order of symmetric group S_n, number of
→permutations of n letters).
>>> a(Integer(0))
1
>>> a(Integer(8))
40320
>>> a(Integer(40))
```

(continues on next page)

(continued from previous page)

```
815915283247897734345611269596115894272000000000
>>> a.list(Integer(9))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320]
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000153**

Bases: *ExtremesOfPermanentsSequence*

$a(n) = n * a(n-1) + (n-2) * a(n-2)$, with $a(0) = 0$, $a(1) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 2$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000153; a
a(n) = n*a(n-1) + (n-2)*a(n-2), with a(0) = 0, a(1) = 1.
sage: a(0)
0
sage: a(1)
1
sage: a(8)
82508
sage: a(20)
10315043624498196944
sage: a.list(8)
[0, 1, 2, 7, 32, 181, 1214, 9403]
```

```
>>> from sage.all import *
>>> a = sloane.A000153; a
a(n) = n*a(n-1) + (n-2)*a(n-2), with a(0) = 0, a(1) = 1.
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(8))
82508
>>> a(Integer(20))
10315043624498196944
>>> a.list(Integer(8))
[0, 1, 2, 7, 32, 181, 1214, 9403]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A000165**

Bases: *SloaneSequence*

Double factorial numbers: $(2n)!! = 2^n * n!$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000165;a
Double factorial numbers: (2n)!! = 2^n*n!.
sage: a(0)
1
sage: a.offset
0
sage: a(8)
10321920
sage: a(20)
2551082656125828464640000
sage: a.list(9)
[1, 2, 8, 48, 384, 3840, 46080, 645120, 10321920]
```

```
>>> from sage.all import *
>>> a = sloane.A000165;a
Double factorial numbers: (2n)!! = 2^n*n!.
>>> a(Integer(0))
1
>>> a.offset
0
>>> a(Integer(8))
10321920
>>> a(Integer(20))
2551082656125828464640000
>>> a.list(Integer(9))
[1, 2, 8, 48, 384, 3840, 46080, 645120, 10321920]
```

AUTHORS:

- Jaap Spies (2007-01-24)

class sage.combinat.sloane_functions.**A000166**

Bases: *SloaneSequence*

Subfactorial or rencontres numbers, or derangements: number of permutations of n elements with no fixed points.

With offset 1 also the permanent of a (0,1)-matrix of order n with n 0s not on a line.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000166;a
Subfactorial or rencontres numbers, or derangements: number of permutations of $n
↪$ elements with no fixed points.
```

(continues on next page)

(continued from previous page)

```
sage: a(0)
1
sage: a(1)
0
sage: a(2)
1
sage: a.offset
0
sage: a(8)
14833
sage: a(20)
895014631192902121
sage: a.list(9)
[1, 0, 1, 2, 9, 44, 265, 1854, 14833]
```

```
>>> from sage.all import *
>>> a = sloane.A000166;a
Subfactorial or rencontres numbers, or derangements: number of permutations of $n
↪$ elements with no fixed points.
>>> a(Integer(0))
1
>>> a(Integer(1))
0
>>> a(Integer(2))
1
>>> a.offset
0
>>> a(Integer(8))
14833
>>> a(Integer(20))
895014631192902121
>>> a.list(Integer(9))
[1, 0, 1, 2, 9, 44, 265, 1854, 14833]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A000169**

Bases: *SloaneSequence*

Number of labeled rooted trees with n nodes: $n^{(n-1)}$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000169;a
Number of labeled rooted trees with n nodes: n^(n-1).
sage: a(0)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
2
sage: a(10)
1000000000
sage: a.list(11)
[1, 2, 9, 64, 625, 7776, 117649, 2097152, 43046721, 1000000000, 25937424601]
```

```
>>> from sage.all import *
>>> a = sloane.A000169;a
Number of labeled rooted trees with n nodes: n^(n-1).
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
2
>>> a(Integer(10))
1000000000
>>> a.list(Integer(11))
[1, 2, 9, 64, 625, 7776, 117649, 2097152, 43046721, 1000000000, 25937424601]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000203**

Bases: *SloaneSequence*

The sequence $\sigma(n)$, where $\sigma(n)$ is the sum of the divisors of n. Also called $\sigma_1(n)$.

The function `sigma(n, k)` implements $\sigma_k(n)$ in Sage.

INPUT:

- `n` – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000203; a
sigma(n) = sum of divisors of n. Also called sigma_1(n).
sage: a(1)
1
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(256)
511
```

(continues on next page)

(continued from previous page)

```
sage: a.list(12)
[1, 3, 4, 7, 6, 12, 8, 15, 13, 18, 12, 28]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

```
>>> from sage.all import *
>>> a = sloane.A000203; a
sigma(n) = sum of divisors of n. Also called sigma_1(n).
>>> a(Integer(1))
1
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(256))
511
>>> a.list(Integer(12))
[1, 3, 4, 7, 6, 12, 8, 15, 13, 18, 12, 28]
>>> a(Integer(1)/Integer(3))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A000204**

Bases: *SloaneSequence*

Lucas numbers (beginning with 1): $L(n) = L(n-1) + L(n-2)$ with $L(1) = 1$, $L(2) = 3$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000204; a
Lucas numbers (beginning at 1): L(n) = L(n-1) + L(n-2), L(2) = 3.
sage: a(1)
1
sage: a(8)
47
sage: a(200)
627376215338105766356982006981782561278127
sage: a(-4)
Traceback (most recent call last):
...
ValueError: input n (=-4) must be a positive integer
sage: a.list(12)
```

(continues on next page)

(continued from previous page)

```
[1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322]
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

```
>>> from sage.all import *
>>> a = sloane.A000204; a
Lucas numbers (beginning at 1): L(n) = L(n-1) + L(n-2), L(2) = 3.
>>> a(Integer(1))
1
>>> a(Integer(8))
47
>>> a(Integer(200))
627376215338105766356982006981782561278127
>>> a(-Integer(4))
Traceback (most recent call last):
...
ValueError: input n (=-4) must be a positive integer
>>> a.list(Integer(12))
[1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322]
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000213**

Bases: *SloaneSequence*

Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3). Starting with 1, 1, 1, …

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000213;a
Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3).
sage: a(0)
1
sage: a(1)
1
sage: a(2)
1
sage: a(11)
355
sage: a.list(12)
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105, 193, 355]
```

```
>>> from sage.all import *
>>> a = sloane.A000213;a
Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3).
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(2))
1
>>> a(Integer(11))
355
>>> a.list(Integer(12))
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105, 193, 355]
```

AUTHORS:

- Jaap Spies (2007-01-19)

list(n)

EXAMPLES:

```
sage: sloane.A000213.list(10)
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105]
```

```
>>> from sage.all import *
>>> sloane.A000213.list(Integer(10))
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105]
```

class sage.combinat.sloane_functions.**A000217**

Bases: *SloaneSequence*

Triangular numbers: $a(n) = \binom{n+1}{2} = n(n+1)/2$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000217;a
Triangular numbers: a(n) = C(n+1,2) = n(n+1)/2 = 0+1+2+...+n.
sage: a(0)
0
sage: a(2)
3
sage: a(8)
36
sage: a(2000)
2001000
sage: a.list(9)
[0, 1, 3, 6, 10, 15, 21, 28, 36]
```

```
>>> from sage.all import *
>>> a = sloane.A000217;a
Triangular numbers: a(n) = C(n+1,2) = n(n+1)/2 = 0+1+2+...+n.
>>> a(Integer(0))
0
>>> a(Integer(2))
3
>>> a(Integer(8))
36
>>> a(Integer(2000))
2001000
>>> a.list(Integer(9))
[0, 1, 3, 6, 10, 15, 21, 28, 36]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000225**

Bases: *SloaneSequence*

$2^n - 1$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000225;a
2^n - 1.
sage: a(0)
0
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(12)
4095
sage: a.list(12)
[0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047]
```

```
>>> from sage.all import *
>>> a = sloane.A000225;a
2^n - 1.
>>> a(Integer(0))
0
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(12))
4095
```

(continues on next page)

(continued from previous page)

```
>>> a.list(Integer(12))
[0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000244**

Bases: *SloaneSequence*

Powers of 3: $a(n) = 3^n$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000244;a
Powers of 3: a(n) = 3^n.
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
1
sage: a(3)
27
sage: a(11)
177147
sage: a.list(12)
[1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, 177147]
```

```
>>> from sage.all import *
>>> a = sloane.A000244;a
Powers of 3: a(n) = 3^n.
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(0))
1
>>> a(Integer(3))
27
>>> a(Integer(11))
177147
>>> a.list(Integer(12))
[1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, 177147]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000255**

Bases: *ExtremesOfPermanentsSequence*

$a(n) = n * a(n-1) + (n-1) * a(n-2)$, with $a(0) = 1$, $a(1) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 1$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

INPUT:

- `n` – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000255;a
a(n) = n*a(n-1) + (n-1)*a(n-2), a(0) = 1, a(1) = 1.
sage: a(0)
1
sage: a(1)
1
sage: a.offset
0
sage: a(8)
148329
sage: a(22)
9923922230666898717143
sage: a.list(9)
[1, 1, 3, 11, 53, 309, 2119, 16687, 148329]
```

```
>>> from sage.all import *
>>> a = sloane.A000255;a
a(n) = n*a(n-1) + (n-1)*a(n-2), a(0) = 1, a(1) = 1.
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a.offset
0
>>> a(Integer(8))
148329
>>> a(Integer(22))
9923922230666898717143
>>> a.list(Integer(9))
[1, 1, 3, 11, 53, 309, 2119, 16687, 148329]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class `sage.combinat.sloane_functions.`**`A000261`**

Bases: *ExtremesOfPermanentsSequence*

$a(n) = n * a(n-1) + (n-3) * a(n-2)$, with $a(1) = 1$, $a(2) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 3$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000261;a
a(n) = n*a(n-1) + (n-3)*a(n-2), a(1) = 0, a(2) = 1.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a.offset
1
sage: a(8)
30637
sage: a(22)
1801366114380914335441
sage: a.list(9)
[0, 1, 3, 13, 71, 465, 3539, 30637, 296967]
```

```
>>> from sage.all import *
>>> a = sloane.A000261;a
a(n) = n*a(n-1) + (n-3)*a(n-2), a(1) = 0, a(2) = 1.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
0
>>> a.offset
1
>>> a(Integer(8))
30637
>>> a(Integer(22))
1801366114380914335441
>>> a.list(Integer(9))
[0, 1, 3, 13, 71, 465, 3539, 30637, 296967]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A000272**

Bases: *SloaneSequence*

Number of labeled rooted trees on n nodes: $n^{(n-2)}$.

INPUT:

- n – integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000272;a
Number of labeled rooted trees with n nodes: n^(n-2).
sage: a(0)
1
sage: a(1)
1
sage: a(2)
1
sage: a(10)
100000000
sage: a.list(12)
[1, 1, 1, 3, 16, 125, 1296, 16807, 262144, 4782969, 100000000, 2357947691]
```

```
>>> from sage.all import *
>>> a = sloane.A000272;a
Number of labeled rooted trees with n nodes: n^(n-2).
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(2))
1
>>> a(Integer(10))
100000000
>>> a.list(Integer(12))
[1, 1, 1, 3, 16, 125, 1296, 16807, 262144, 4782969, 100000000, 2357947691]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000290**

Bases: *SloaneSequence*

The squares: $a(n) = n^2$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000290;a
The squares: a(n) = n^2.
sage: a(0)
0
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(16)
256
sage: a.list(17)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256]
```

```
>>> from sage.all import *
>>> a = sloane.A000290;a
The squares: a(n) = n^2.
>>> a(Integer(0))
0
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(16))
256
>>> a.list(Integer(17))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000292**

Bases: *SloaneSequence*

Tetrahedral (or pyramidal) numbers: $\binom{n+2}{3} = n(n+1)(n+2)/6$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000292;a
Tetrahedral (or pyramidal) numbers: C(n+2,3) = n(n+1)(n+2)/6.
sage: a(0)
0
sage: a(2)
4
sage: a(11)
286
sage: a.list(12)
[0, 1, 4, 10, 20, 35, 56, 84, 120, 165, 220, 286]
```

```
>>> from sage.all import *
>>> a = sloane.A000292;a
Tetrahedral (or pyramidal) numbers: C(n+2,3) = n(n+1)(n+2)/6.
>>> a(Integer(0))
0
>>> a(Integer(2))
4
>>> a(Integer(11))
286
>>> a.list(Integer(12))
[0, 1, 4, 10, 20, 35, 56, 84, 120, 165, 220, 286]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000302**

Bases: *SloaneSequence*

Powers of 4: $a(n) = 4^n$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000302;a
Powers of 4: a(n) = 4^n.
sage: a(0)
1
sage: a(1)
4
sage: a(2)
16
sage: a(10)
1048576
sage: a.list(12)
[1, 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, 4194304]
```

```
>>> from sage.all import *
>>> a = sloane.A000302;a
Powers of 4: a(n) = 4^n.
>>> a(Integer(0))
1
>>> a(Integer(1))
4
>>> a(Integer(2))
16
>>> a(Integer(10))
1048576
>>> a.list(Integer(12))
[1, 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, 4194304]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000312**

Bases: *SloaneSequence*

Number of labeled mappings from n points to themselves (endofunctions): n^n.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000312;a
Number of labeled mappings from n points to themselves (endofunctions): n^n.
```

(continues on next page)

(continued from previous page)

```
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
1
sage: a(1)
1
sage: a(9)
387420489
sage: a.list(11)
[1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489, 10000000000]
```

```
>>> from sage.all import *
>>> a = sloane.A000312;a
Number of labeled mappings from n points to themselves (endofunctions): n^n.
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(9))
387420489
>>> a.list(Integer(11))
[1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489, 10000000000]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000326**

Bases: *SloaneSequence*

Pentagonal numbers: $n(3n - 1)/2$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000326;a
Pentagonal numbers: n(3n-1)/2.
sage: a(0)
0
sage: a(1)
1
sage: a(2)
5
sage: a(10)
```

(continues on next page)

(continued from previous page)

```
145
sage: a.list(12)
[0, 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

```
>>> from sage.all import *
>>> a = sloane.A000326;a
Pentagonal numbers: n(3n-1)/2.
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(2))
5
>>> a(Integer(10))
145
>>> a.list(Integer(12))
[0, 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176]
>>> a(Integer(1)/Integer(3))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000330**

Bases: *SloaneSequence*

Square pyramidal numbers" $0^2 + 1^2 \cdots n^2 = n(n+1)(2n+1)/6$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000330;a
Square pyramidal numbers: 0^2+1^2+2^2+...+n^2 = n(n+1)(2n+1)/6.
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
0
sage: a(3)
14
sage: a(11)
506
```

(continues on next page)

(continued from previous page)

```
sage: a.list(12)
[0, 1, 5, 14, 30, 55, 91, 140, 204, 285, 385, 506]
```

```
>>> from sage.all import *
>>> a = sloane.A000330;a
Square pyramidal numbers: 0^2+1^2+2^2+...+n^2 = n(n+1)(2n+1)/6.
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(0))
0
>>> a(Integer(3))
14
>>> a(Integer(11))
506
>>> a.list(Integer(12))
[0, 1, 5, 14, 30, 55, 91, 140, 204, 285, 385, 506]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000396**

Bases: *SloaneSequence*

Perfect numbers: equal to sum of proper divisors.

INPUT:

- `n` – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000396;a
Perfect numbers: equal to sum of proper divisors.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
6
sage: a(2)
28
sage: a(7)
137438691328
sage: a.list(7)
[6, 28, 496, 8128, 33550336, 8589869056, 137438691328]
```

```
>>> from sage.all import *
>>> a = sloane.A000396;a
Perfect numbers: equal to sum of proper divisors.
>>> a(Integer(0))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
6
>>> a(Integer(2))
28
>>> a(Integer(7))
137438691328
>>> a.list(Integer(7))
[6, 28, 496, 8128, 33550336, 8589869056, 137438691328]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000578**

Bases: *SloaneSequence*

The cubes: $a(n) = n^3$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000578;a
The cubes: n^3
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
0
sage: a(3)
27
sage: a(11)
1331
sage: a.list(12)
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331]
```

```
>>> from sage.all import *
>>> a = sloane.A000578;a
The cubes: n^3
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(0))
0
>>> a(Integer(3))
27
```

(continues on next page)

(continued from previous page)

```
>>> a(Integer(11))
1331
>>> a.list(Integer(12))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000583**

Bases: *SloaneSequence*

Fourth powers: $a(n) = n^4$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000583;a
Fourth powers: n^4.
sage: a(0.0)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
sage: a(1)
1
sage: a(2)
16
sage: a(9)
6561
sage: a.list(10)
[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561]
```

```
>>> from sage.all import *
>>> a = sloane.A000583;a
Fourth powers: n^4.
>>> a(RealNumber('0.0'))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
>>> a(Integer(1))
1
>>> a(Integer(2))
16
>>> a(Integer(9))
6561
>>> a.list(Integer(10))
[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561]
```

AUTHORS:

- Jaap Spies (2007-02-04)

class sage.combinat.sloane_functions.**A000587**

Bases: *ExponentialNumbers*

The sequence of Uppuluri-Carpenter numbers.

The Uppuluri-Carpenter number C_n counts the imbalance in the number of ways to put n distinguishable things into an even number of indistinguishable boxes versus into an odd number of indistinguishable boxes, such that no box is empty.

Let $S(n, k)$ denote the Stirling number of the second kind. Then

$$C_n = \sum k = 0^n (-1)^k S(n, k).$$

INPUT:

- n – nonnegative integer

OUTPUT: integer; C_n

EXAMPLES:

```
sage: a = sloane.A000587; a
Sequence of Uppuluri-Carpenter numbers
sage: a.offset
0
sage: a(0)
1
sage: a(100)
397577026456518507969762382254187048845620355238545130875069912944235105204434466
↪095862371032124545552161
sage: a.list(10)
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]
```

```
>>> from sage.all import *
>>> a = sloane.A000587; a
Sequence of Uppuluri-Carpenter numbers
>>> a.offset
0
>>> a(Integer(0))
1
>>> a(Integer(100))
397577026456518507969762382254187048845620355238545130875069912944235105204434466
↪095862371032124545552161
>>> a.list(Integer(10))
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]
```

AUTHORS:

- Nick Alexander

class sage.combinat.sloane_functions.**A000668**

Bases: *SloaneSequence*

Mersenne primes (of form $2^p - 1$ where p is a prime).

(See A000043 for the values of p.)

Warning: a(39) has 4,053,946 digits!

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000668;a
Mersenne primes (of form 2^p - 1 where p is a prime). (See A000043 for the values␣
↪of p.)
sage: a(1)
3
sage: a(2)
7
sage: a(12)
170141183460469231731687303715884105727
```

```
>>> from sage.all import *
>>> a = sloane.A000668;a
Mersenne primes (of form 2^p - 1 where p is a prime). (See A000043 for the values␣
↪of p.)
>>> a(Integer(1))
3
>>> a(Integer(2))
7
>>> a(Integer(12))
170141183460469231731687303715884105727
```

Warning: a(39) has 4,053,946 digits!

```
sage: a(40)
Traceback (most recent call last):
...
IndexError: list index out of range
sage: a.list(8)
[3, 7, 31, 127, 8191, 131071, 524287, 2147483647]
```

```
>>> from sage.all import *
>>> a(Integer(40))
Traceback (most recent call last):
...
IndexError: list index out of range
>>> a.list(Integer(8))
[3, 7, 31, 127, 8191, 131071, 524287, 2147483647]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000670**

Bases: *SloaneSequence*

Number of preferential arrangements of n labeled elements; or number of weak orders on n labeled elements.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000670;a
Number of preferential arrangements of n labeled elements.
sage: a(0)
1
sage: a(1)
1
sage: a(2)
3
sage: a(9)
7087261
sage: a.list(10)
[1, 1, 3, 13, 75, 541, 4683, 47293, 545835, 7087261]
```

```
>>> from sage.all import *
>>> a = sloane.A000670;a
Number of preferential arrangements of n labeled elements.
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(2))
3
>>> a(Integer(9))
7087261
>>> a.list(Integer(10))
[1, 1, 3, 13, 75, 541, 4683, 47293, 545835, 7087261]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A000720**

Bases: *SloaneSequence*

$pi(n)$, the number of primes $\leq n$. Sometimes called $PrimePi(n)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000720;a
pi(n), the number of primes <= n. Sometimes called PrimePi(n)
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
1
sage: a(8)
```

(continues on next page)

(continued from previous page)

```
4
sage: a(1000)
168
sage: a.list(12)
[0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5]
```

```
>>> from sage.all import *
>>> a = sloane.A000720;a
pi(n), the number of primes <= n. Sometimes called PrimePi(n)
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(2))
1
>>> a(Integer(8))
4
>>> a(Integer(1000))
168
>>> a.list(Integer(12))
[0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000796**

Bases: *SloaneSequence*

Decimal expansion of π.

INPUT:

- n – positive integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000796;a
Decimal expansion of Pi.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
3
sage: a(13)
9
sage: a.list(14)
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7]
sage: a(100)
7
```

```
>>> from sage.all import *
>>> a = sloane.A000796;a
Decimal expansion of Pi.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
3
>>> a(Integer(13))
9
>>> a.list(Integer(14))
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7]
>>> a(Integer(100))
7
```

AUTHOR:

- Jaap Spies (2007-01-30)

list(*n*)

EXAMPLES:

```
sage: sloane.A000796.list(10)
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

```
>>> from sage.all import *
>>> sloane.A000796.list(Integer(10))
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

pi()

Based on an algorithm of Lambert Meertens The ABC-programming language!!!

EXAMPLES:

```
sage: it = sloane.A000796.pi()
sage: [next(it) for i in range(10)]
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

```
>>> from sage.all import *
>>> it = sloane.A000796.pi()
>>> [next(it) for i in range(Integer(10))]
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

class sage.combinat.sloane_functions.**A000961**

Bases: *SloaneSequence*

Prime powers.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000961;a
Prime powers.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
2
sage: a(12)
17
sage: a.list(12)
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17]
```

```
>>> from sage.all import *
>>> a = sloane.A000961;a
Prime powers.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(2))
2
>>> a(Integer(12))
17
>>> a.list(Integer(12))
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17]
```

AUTHORS:

- Jaap Spies (2007-01-25)

list(*n*)

EXAMPLES:

```
sage: sloane.A000961.list(10)
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13]
```

```
>>> from sage.all import *
>>> sloane.A000961.list(Integer(10))
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13]
```

class sage.combinat.sloane_functions.**A000984**

Bases: *SloaneSequence*

Central binomial coefficients: $\binom{2n}{n} = \frac{(2n)!}{(n!)^2}$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A000984;a
Central binomial coefficients: C(2n,n) = (2n)!/(n!)^2
sage: a(0)
1
sage: a(2)
6
sage: a(8)
12870
sage: a.list(9)
[1, 2, 6, 20, 70, 252, 924, 3432, 12870]
```

```
>>> from sage.all import *
>>> a = sloane.A000984;a
Central binomial coefficients: C(2n,n) = (2n)!/(n!)^2
>>> a(Integer(0))
1
>>> a(Integer(2))
6
>>> a(Integer(8))
12870
>>> a.list(Integer(9))
[1, 2, 6, 20, 70, 252, 924, 3432, 12870]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A001006**

Bases: *SloaneSequence*

Motzkin numbers: number of ways of drawing any number of nonintersecting chords among n points on a circle.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001006;a
Motzkin numbers: number of ways of drawing any number of nonintersecting chords␣
↪among n points on a circle.
sage: a(0)
1
sage: a(1)
1
sage: a(2)
2
sage: a(12)
15511
sage: a.list(13)
[1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, 5798, 15511]
```

```
>>> from sage.all import *
>>> a = sloane.A001006;a
```

(continues on next page)

(continued from previous page)

```
Motzkin numbers: number of ways of drawing any number of nonintersecting chords␣
↪among n points on a circle.
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(2))
2
>>> a(Integer(12))
15511
>>> a.list(Integer(13))
[1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, 5798, 15511]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A001045**

Bases: *RecurrenceSequence2*

Jacobsthal sequence: $a(n) = a(n-1) + 2a(n-2)$, $a(0) = 0$ and $a(1) = 1$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001045;a
Jacobsthal sequence: a(n) = a(n-1) + 2a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(2)
1
sage: a(11)
683
sage: a.list(12)
[0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341, 683]
```

```
>>> from sage.all import *
>>> a = sloane.A001045;a
Jacobsthal sequence: a(n) = a(n-1) + 2a(n-2).
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(2))
1
>>> a(Integer(11))
683
```

(continues on next page)

(continued from previous page)

```
>>> a.list(Integer(12))
[0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341, 683]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A001055**

Bases: *SloaneSequence*

Number of ways of factoring n with all factors 1.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001055;a
Number of ways of factoring n with all factors >1.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
1
sage: a(9)
2
sage: a.list(16)
[1, 1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 4, 1, 2, 2, 5]
```

```
>>> from sage.all import *
>>> a = sloane.A001055;a
Number of ways of factoring n with all factors >1.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
1
>>> a(Integer(9))
2
>>> a.list(Integer(16))
[1, 1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 4, 1, 2, 2, 5]
```

AUTHORS:

- Jaap Spies (2007-02-04)

nwf(n, m)

EXAMPLES:

```
sage: sloane.A001055.nwf(4,1)
0
sage: sloane.A001055.nwf(4,2)
1
sage: sloane.A001055.nwf(4,3)
1
sage: sloane.A001055.nwf(4,4)
2
```

```
>>> from sage.all import *
>>> sloane.A001055.nwf(Integer(4),Integer(1))
0
>>> sloane.A001055.nwf(Integer(4),Integer(2))
1
>>> sloane.A001055.nwf(Integer(4),Integer(3))
1
>>> sloane.A001055.nwf(Integer(4),Integer(4))
2
```

class sage.combinat.sloane_functions.**A001109**

Bases: *RecurrenceSequence2*

$a(n)^2$ is a triangular number: $a(n) = 6 * a(n-1) - a(n-2)$ with $a(0) = 0$, $a(1) = 1$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001109;a
a(n)^2 is a triangular number: a(n) = 6*a(n-1) - a(n-2) with a(0)=0, a(1)=1
sage: a(0)
0
sage: a(1)
1
sage: a(2)
6
sage: a.offset
0
sage: a(8)
235416
sage: a(60)
1515330104844857898115857393785728383101709300
sage: a.list(9)
[0, 1, 6, 35, 204, 1189, 6930, 40391, 235416]
```

```
>>> from sage.all import *
>>> a = sloane.A001109;a
a(n)^2 is a triangular number: a(n) = 6*a(n-1) - a(n-2) with a(0)=0, a(1)=1
>>> a(Integer(0))
0
>>> a(Integer(1))
```

(continues on next page)

(continued from previous page)

```
1
>>> a(Integer(2))
6
>>> a.offset
0
>>> a(Integer(8))
235416
>>> a(Integer(60))
1515330104844857898115857393785728383101709300
>>> a.list(Integer(9))
[0, 1, 6, 35, 204, 1189, 6930, 40391, 235416]
```

AUTHORS:

- Jaap Spies (2007-01-24)

class sage.combinat.sloane_functions.**A001110**

Bases: *RecurrenceSequence*

Numbers that are both triangular and square: $a(n) = 34a(n-1) - a(n-2) + 2$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001110; a
Numbers that are both triangular and square: a(n) = 34a(n-1) - a(n-2) + 2.
sage: a(0)
0
sage: a(1)
1
sage: a(8)
55420693056
sage: a(21)
4446390382511295358038307980025
sage: a.list(8)
[0, 1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

```
>>> from sage.all import *
>>> a = sloane.A001110; a
Numbers that are both triangular and square: a(n) = 34a(n-1) - a(n-2) + 2.
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(8))
55420693056
>>> a(Integer(21))
4446390382511295358038307980025
>>> a.list(Integer(8))
[0, 1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

AUTHORS:

- Jaap Spies (2007-01-19)

g(*k*)

EXAMPLES:

```
sage: sloane.A001110.g(2)
2
sage: sloane.A001110.g(1)
0
```

```
>>> from sage.all import *
>>> sloane.A001110.g(Integer(2))
2
>>> sloane.A001110.g(Integer(1))
0
```

link = 'http://oeis.org/classic/A001110'

class sage.combinat.sloane_functions.**A001147**

Bases: *SloaneSequence*

Double factorial numbers: $(2n-1)!! = 1 \cdot 3 \cdot 5 \cdots (2n-1)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001147;a
Double factorial numbers: (2n-1)!! = 1.3.5....(2n-1).
sage: a(0)
1
sage: a.offset
0
sage: a(8)
2027025
sage: a(20)
319830986772877770815625
sage: a.list(9)
[1, 1, 3, 15, 105, 945, 10395, 135135, 2027025]
```

```
>>> from sage.all import *
>>> a = sloane.A001147;a
Double factorial numbers: (2n-1)!! = 1.3.5....(2n-1).
>>> a(Integer(0))
1
>>> a.offset
0
>>> a(Integer(8))
2027025
>>> a(Integer(20))
```

(continues on next page)

(continued from previous page)

```
319830986772877770815625
>>> a.list(Integer(9))
[1, 1, 3, 15, 105, 945, 10395, 135135, 2027025]
```

AUTHORS:

- Jaap Spies (2007-01-24)

class sage.combinat.sloane_functions.**A001157**

Bases: *SloaneSequence*

The sequence $\sigma_2(n)$, sum of squares of divisors of n.

The function sigma(n, k) implements σ_k* in Sage.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001157;a
sigma_2(n): sum of squares of divisors of n
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
5
sage: a(8)
85
sage: a.list(9)
[1, 5, 10, 21, 26, 50, 50, 85, 91]
```

```
>>> from sage.all import *
>>> a = sloane.A001157;a
sigma_2(n): sum of squares of divisors of n
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(2))
5
>>> a(Integer(8))
85
>>> a.list(Integer(9))
[1, 5, 10, 21, 26, 50, 50, 85, 91]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A001189**

Bases: *SloaneSequence*

Number of degree-n permutations of order exactly 2.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001189;a
Number of degree-n permutations of order exactly 2.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(2)
1
sage: a(12)
140151
sage: a.list(13)
[0, 1, 3, 9, 25, 75, 231, 763, 2619, 9495, 35695, 140151, 568503]
```

```
>>> from sage.all import *
>>> a = sloane.A001189;a
Number of degree-n permutations of order exactly 2.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
0
>>> a(Integer(2))
1
>>> a(Integer(12))
140151
>>> a.list(Integer(13))
[0, 1, 3, 9, 25, 75, 231, 763, 2619, 9495, 35695, 140151, 568503]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A001221**

Bases: *SloaneSequence*

Number of different prime divisors of n.

Also called omega(n) or $\omega(n)$. Maximal number of terms in any factorization of n. Number of prime powers that divide n.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001221; a
Number of distinct primes dividing n (also called omega(n)).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(8)
1
sage: a(41)
1
sage: a(84792)
3
sage: a.list(12)
[0, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2]
```

```
>>> from sage.all import *
>>> a = sloane.A001221; a
Number of distinct primes dividing n (also called omega(n)).
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
0
>>> a(Integer(8))
1
>>> a(Integer(41))
1
>>> a(Integer(84792))
3
>>> a.list(Integer(12))
[0, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A001222**

Bases: *SloaneSequence*

Number of prime divisors of n (counted with multiplicity).

Also called bigomega(n) or $\Omega(n)$. Maximal number of terms in any factorization of n. Number of prime powers that divide n.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001222; a
Number of prime divisors of n (counted with multiplicity).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(8)
3
sage: a(41)
1
sage: a(84792)
5
sage: a.list(12)
[0, 1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 3]
```

```
>>> from sage.all import *
>>> a = sloane.A001222; a
Number of prime divisors of n (counted with multiplicity).
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
0
>>> a(Integer(8))
3
>>> a(Integer(41))
1
>>> a(Integer(84792))
5
>>> a.list(Integer(12))
[0, 1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 3]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A001227**

Bases: *SloaneSequence*

Number of odd divisors of n.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001227; a
Number of odd divisors of n
sage: a.offset
1
```

(continues on next page)

(continued from previous page)

```
sage: a(1)
1
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(100)
3
sage: a(256)
1
sage: a(29)
2
sage: a.list(20)
[1, 1, 2, 1, 2, 2, 2, 1, 3, 2, 2, 2, 2, 2, 4, 1, 2, 3, 2, 2]
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

```
>>> from sage.all import *
>>> a = sloane.A001227; a
Number of odd divisors of n
>>> a.offset
1
>>> a(Integer(1))
1
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(100))
3
>>> a(Integer(256))
1
>>> a(Integer(29))
2
>>> a.list(Integer(20))
[1, 1, 2, 1, 2, 2, 2, 1, 3, 2, 2, 2, 2, 2, 4, 1, 2, 3, 2, 2]
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

class sage.combinat.sloane_functions.**A001333**

Bases: *RecurrenceSequence2*

Numerators of continued fraction convergents to $\sqrt{2}$.

See also A000129

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001333;a
Numerators of continued fraction convergents to sqrt(2).
sage: a(0)
1
sage: a(1)
1
sage: a(2)
3
sage: a(3)
7
sage: a(11)
8119
sage: a.list(12)
[1, 1, 3, 7, 17, 41, 99, 239, 577, 1393, 3363, 8119]
```

```
>>> from sage.all import *
>>> a = sloane.A001333;a
Numerators of continued fraction convergents to sqrt(2).
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(2))
3
>>> a(Integer(3))
7
>>> a(Integer(11))
8119
>>> a.list(Integer(12))
[1, 1, 3, 7, 17, 41, 99, 239, 577, 1393, 3363, 8119]
```

AUTHORS:

- Jaap Spies (2007-02-01)

class sage.combinat.sloane_functions.**A001358**

Bases: *SloaneSequence*

Products of two primes.

These numbers have been called semiprimes (or semi-primes), biprimes or 2-almost primes.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001358;a
Products of two primes.
```

(continues on next page)

(continued from previous page)

```
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
6
sage: a(8)
22
sage: a(200)
669
sage: a.list(9)
[4, 6, 9, 10, 14, 15, 21, 22, 25]
```

```
>>> from sage.all import *
>>> a = sloane.A001358;a
Products of two primes.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(2))
6
>>> a(Integer(8))
22
>>> a(Integer(200))
669
>>> a.list(Integer(9))
[4, 6, 9, 10, 14, 15, 21, 22, 25]
```

AUTHORS:

- Jaap Spies (2007-01-25)

list(*n*)

EXAMPLES:

```
sage: sloane.A001358.list(9)
[4, 6, 9, 10, 14, 15, 21, 22, 25]
```

```
>>> from sage.all import *
>>> sloane.A001358.list(Integer(9))
[4, 6, 9, 10, 14, 15, 21, 22, 25]
```

class sage.combinat.sloane_functions.**A001405**

Bases: *SloaneSequence*

Central binomial coefficients: $\binom{n}{\lfloor \frac{n}{2} \rfloor}$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001405;a
Central binomial coefficients: C(n,floor(n/2)).
sage: a(0)
1
sage: a(2)
2
sage: a(12)
924
sage: a.list(12)
[1, 1, 2, 3, 6, 10, 20, 35, 70, 126, 252, 462]
```

```
>>> from sage.all import *
>>> a = sloane.A001405;a
Central binomial coefficients: C(n,floor(n/2)).
>>> a(Integer(0))
1
>>> a(Integer(2))
2
>>> a(Integer(12))
924
>>> a.list(Integer(12))
[1, 1, 2, 3, 6, 10, 20, 35, 70, 126, 252, 462]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A001477**

Bases: *SloaneSequence*

The nonnegative integers.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001477;a
The nonnegative integers.
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
0
sage: a(3382789)
3382789
sage: a(11)
11
sage: a.list(12)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
>>> from sage.all import *
>>> a = sloane.A001477;a
The nonnegative integers.
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(0))
0
>>> a(Integer(3382789))
3382789
>>> a(Integer(11))
11
>>> a.list(Integer(12))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A001694**

Bases: *SloaneSequence*

This function returns the n-th Powerful Number:

A positive integer n is powerful if for every prime p dividing n, p^2 also divides n.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001694; a
Powerful Numbers (also called squarefull, square-full or 2-full numbers).
sage: a.offset
1
sage: a(1)
1
sage: a(4)
9
sage: a(100)
3136
sage: a(156)
7225
sage: a.list(19)
[1, 4, 8, 9, 16, 25, 27, 32, 36, 49, 64, 72, 81, 100, 108, 121, 125, 128, 144]
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

```
>>> from sage.all import *
>>> a = sloane.A001694; a
```

(continues on next page)

(continued from previous page)

```
Powerful Numbers (also called squarefull, square-full or 2-full numbers).
>>> a.offset
1
>>> a(Integer(1))
1
>>> a(Integer(4))
9
>>> a(Integer(100))
3136
>>> a(Integer(156))
7225
>>> a.list(Integer(19))
[1, 4, 8, 9, 16, 25, 27, 32, 36, 49, 64, 72, 81, 100, 108, 121, 125, 128, 144]
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

is_powerful(*n*)

Return `True` if and only if n is a powerful number.

A positive integer n is powerful if for every prime p dividing n, p^2 also divides n.

See OEIS sequence A001694.

INPUT:

- `n` – integer

OUTPUT: `True` if n is a powerful number, else `False`

EXAMPLES:

```
sage: a = sloane.A001694
sage: a.is_powerful(2500)
True
sage: a.is_powerful(20)
False
```

```
>>> from sage.all import *
>>> a = sloane.A001694
>>> a.is_powerful(Integer(2500))
True
>>> a.is_powerful(Integer(20))
False
```

AUTHORS:

- Jaap Spies (2006-12-07)

list(*n*)

EXAMPLES:

```
sage: sloane.A001694.list(9)
[1, 4, 8, 9, 16, 25, 27, 32, 36]
```

```
>>> from sage.all import *
>>> sloane.A001694.list(Integer(9))
[1, 4, 8, 9, 16, 25, 27, 32, 36]
```

class sage.combinat.sloane_functions.**A001836**

Bases: *SloaneSequence*

Numbers n such that $\phi(2n-1) < \phi(2n)$, where ϕ is Euler's totient function.

Euler's totient function is also known as `euler_phi`; `euler_phi` is a standard Sage function.

INPUT:

- `n` – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001836; a
Numbers n such that phi(2n-1) < phi(2n), where phi is Euler's totient function␣
↪A000010.
sage: a.offset
1
sage: a(1)
53
sage: a(8)
683
sage: a(300)
17798
sage: a.list(12)
[53, 83, 158, 263, 293, 368, 578, 683, 743, 788, 878, 893]
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

```
>>> from sage.all import *
>>> a = sloane.A001836; a
Numbers n such that phi(2n-1) < phi(2n), where phi is Euler's totient function␣
↪A000010.
>>> a.offset
1
>>> a(Integer(1))
53
>>> a(Integer(8))
683
>>> a(Integer(300))
17798
>>> a.list(Integer(12))
[53, 83, 158, 263, 293, 368, 578, 683, 743, 788, 878, 893]
>>> a(Integer(0))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

Compare: Searching Sloane's online database… Numbers n such that phi(2n-1) phi(2n), where phi is Euler's totient function A000010. [53, 83, 158, 263, 293, 368, 578, 683, 743, 788, 878, 893]

AUTHORS:

- Jaap Spies (2007-01-17)

list(n)

EXAMPLES:

```
sage: sloane.A001836.list(9)
[53, 83, 158, 263, 293, 368, 578, 683, 743]
```

```
>>> from sage.all import *
>>> sloane.A001836.list(Integer(9))
[53, 83, 158, 263, 293, 368, 578, 683, 743]
```

class sage.combinat.sloane_functions.**A001906**

Bases: *RecurrenceSequence2*

$F(2n)$ = bisection of Fibonacci sequence: $a(n) = 3a(n-1) - a(n-2)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001906; a
F(2n) = bisection of Fibonacci sequence: a(n)=3a(n-1)-a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(8)
987
sage: a(22)
701408733
sage: a.list(12)
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584, 6765, 17711]
```

```
>>> from sage.all import *
>>> a = sloane.A001906; a
F(2n) = bisection of Fibonacci sequence: a(n)=3a(n-1)-a(n-2).
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(8))
```

(continues on next page)

(continued from previous page)

```
987
>>> a(Integer(22))
701408733
>>> a.list(Integer(12))
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584, 6765, 17711]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A001909**

Bases: *ExtremesOfPermanentsSequence*

$a(n) = n * a(n-1) + (n-4) * a(n-2)$, with $a(2) = 0$, $a(3) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 4$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – positive integer >= 2

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001909;a
a(n) = n*a(n-1) + (n-4)*a(n-2), a(2) = 0, a(3) = 1.
sage: a(1)
Traceback (most recent call last):
...
ValueError: input n (=1) must be an integer >= 2
sage: a.offset
2
sage: a(2)
0
sage: a(8)
8544
sage: a(22)
470033715095287415734
sage: a.list(9)
[0, 1, 4, 21, 134, 1001, 8544, 81901, 870274]
```

```
>>> from sage.all import *
>>> a = sloane.A001909;a
a(n) = n*a(n-1) + (n-4)*a(n-2), a(2) = 0, a(3) = 1.
>>> a(Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=1) must be an integer >= 2
>>> a.offset
2
>>> a(Integer(2))
```

(continues on next page)

(continued from previous page)

```
0
>>> a(Integer(8))
8544
>>> a(Integer(22))
470033715095287415734
>>> a.list(Integer(9))
[0, 1, 4, 21, 134, 1001, 8544, 81901, 870274]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A001910**

Bases: *ExtremesOfPermanentsSequence*

$a(n) = n * a(n-1) + (n-5) * a(n-2)$, with $a(3) = 0$, $a(4) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 5$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – positive integer ≥ 3

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001910;a
a(n) = n*a(n-1) + (n-5)*a(n-2), a(3) = 0, a(4) = 1.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be an integer >= 3
sage: a(3)
0
sage: a.offset
3
sage: a(8)
1909
sage: a(22)
98125321641110663023
sage: a.list(9)
[0, 1, 5, 31, 227, 1909, 18089, 190435, 2203319]
```

```
>>> from sage.all import *
>>> a = sloane.A001910;a
a(n) = n*a(n-1) + (n-5)*a(n-2), a(3) = 0, a(4) = 1.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be an integer >= 3
>>> a(Integer(3))
```

(continues on next page)

(continued from previous page)

```
0
>>> a.offset
3
>>> a(Integer(8))
1909
>>> a(Integer(22))
98125321641110663023
>>> a.list(Integer(9))
[0, 1, 5, 31, 227, 1909, 18089, 190435, 2203319]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A001969**

Bases: *SloaneSequence*

Evil numbers: even number of 1s in binary expansion.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A001969;a
Evil numbers: even number of 1s in binary expansion.
sage: a(0)
0
sage: a(1)
3
sage: a(2)
5
sage: a(12)
24
sage: a.list(13)
[0, 3, 5, 6, 9, 10, 12, 15, 17, 18, 20, 23, 24]
```

```
>>> from sage.all import *
>>> a = sloane.A001969;a
Evil numbers: even number of 1s in binary expansion.
>>> a(Integer(0))
0
>>> a(Integer(1))
3
>>> a(Integer(2))
5
>>> a(Integer(12))
24
>>> a.list(Integer(13))
[0, 3, 5, 6, 9, 10, 12, 15, 17, 18, 20, 23, 24]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A002110**

Bases: *SloaneSequence*

Primorial numbers (first definition): product of first n primes. Sometimes written $p\#$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A002110;a
Primorial numbers (first definition): product of first n primes. Sometimes␣
↪written p#.
sage: a(0)
1
sage: a(2)
6
sage: a(8)
9699690
sage: a(17)
1922760350154212639070
sage: a.list(9)
[1, 2, 6, 30, 210, 2310, 30030, 510510, 9699690]
```

```
>>> from sage.all import *
>>> a = sloane.A002110;a
Primorial numbers (first definition): product of first n primes. Sometimes␣
↪written p#.
>>> a(Integer(0))
1
>>> a(Integer(2))
6
>>> a(Integer(8))
9699690
>>> a(Integer(17))
1922760350154212639070
>>> a.list(Integer(9))
[1, 2, 6, 30, 210, 2310, 30030, 510510, 9699690]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A002113**

Bases: *SloaneSequence*

Palindromes in base 10.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A002113;a
Palindromes in base 10.
sage: a(0)
0
sage: a(1)
1
sage: a(2)
2
sage: a(12)
33
sage: a.list(13)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33]
```

```
>>> from sage.all import *
>>> a = sloane.A002113;a
Palindromes in base 10.
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(2))
2
>>> a(Integer(12))
33
>>> a.list(Integer(13))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33]
```

AUTHORS:

- Jaap Spies (2007-02-02)

list(*n*)

EXAMPLES:

```
sage: sloane.A002113.list(15)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55]
```

```
>>> from sage.all import *
>>> sloane.A002113.list(Integer(15))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55]
```

class sage.combinat.sloane_functions.**A002275**

Bases: *SloaneSequence*

Repunits: $\frac{(10^n-1)}{9}$. Often denoted by R_n.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A002275;a
Repunits: (10^n - 1)/9. Often denoted by R_n.
sage: a(0)
0
sage: a(2)
11
sage: a(8)
11111111
sage: a(20)
11111111111111111111
sage: a.list(9)
[0, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111]
```

```
>>> from sage.all import *
>>> a = sloane.A002275;a
Repunits: (10^n - 1)/9. Often denoted by R_n.
>>> a(Integer(0))
0
>>> a(Integer(2))
11
>>> a(Integer(8))
11111111
>>> a(Integer(20))
11111111111111111111
>>> a.list(Integer(9))
[0, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A002378**

Bases: *SloaneSequence*

Oblong (or pronic, or heteromecic) numbers: $n(n+1)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A002378;a
Oblong (or pronic, or heteromecic) numbers: n(n+1).
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
0
sage: a(1)
2
sage: a(11)
132
```

(continues on next page)

(continued from previous page)

```
sage: a.list(12)
[0, 2, 6, 12, 20, 30, 42, 56, 72, 90, 110, 132]
```

```
>>> from sage.all import *
>>> a = sloane.A002378;a
Oblong (or pronic, or heteromecic) numbers: n(n+1).
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(0))
0
>>> a(Integer(1))
2
>>> a(Integer(11))
132
>>> a.list(Integer(12))
[0, 2, 6, 12, 20, 30, 42, 56, 72, 90, 110, 132]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A002620**

Bases: *SloaneSequence*

Quarter-squares: floor(n/2)*ceiling(n/2). Equivalently, $\lfloor n^2/4 \rfloor$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A002620;a
Quarter-squares: floor(n/2)*ceiling(n/2). Equivalently, floor(n^2/4).
sage: a(0)
0
sage: a(1)
0
sage: a(2)
1
sage: a(10)
25
sage: a.list(12)
[0, 0, 1, 2, 4, 6, 9, 12, 16, 20, 25, 30]
```

```
>>> from sage.all import *
>>> a = sloane.A002620;a
Quarter-squares: floor(n/2)*ceiling(n/2). Equivalently, floor(n^2/4).
>>> a(Integer(0))
0
>>> a(Integer(1))
```

(continues on next page)

(continued from previous page)

```
0
>>> a(Integer(2))
1
>>> a(Integer(10))
25
>>> a.list(Integer(12))
[0, 0, 1, 2, 4, 6, 9, 12, 16, 20, 25, 30]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A002808**

Bases: *SloaneSequence*

The composite numbers: numbers n of the form xy for $x > 1$ and $y > 1$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A002808;a
The composite numbers: numbers n of the form x*y for x > 1 and y > 1.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
6
sage: a(11)
20
sage: a.list(12)
[4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21]
```

```
>>> from sage.all import *
>>> a = sloane.A002808;a
The composite numbers: numbers n of the form x*y for x > 1 and y > 1.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(2))
6
>>> a(Integer(11))
20
>>> a.list(Integer(12))
[4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21]
```

AUTHORS:

- Jaap Spies (2007-01-26)

list(*n*)

EXAMPLES:

```
sage: sloane.A002808.list(10)
[4, 6, 8, 9, 10, 12, 14, 15, 16, 18]
```

```
>>> from sage.all import *
>>> sloane.A002808.list(Integer(10))
[4, 6, 8, 9, 10, 12, 14, 15, 16, 18]
```

class sage.combinat.sloane_functions.**A003418**

Bases: *SloaneSequence*

Least common multiple (or lcm) of $\{1, 2, \ldots, n\}$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A003418;a
Least common multiple (or lcm) of {1, 2, ..., n}.
sage: a(0)
1
sage: a(1)
1
sage: a(13)
360360
sage: a.list(14)
[1, 1, 2, 6, 12, 60, 60, 420, 840, 2520, 2520, 27720, 27720, 360360]
sage: a(20.0)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

```
>>> from sage.all import *
>>> a = sloane.A003418;a
Least common multiple (or lcm) of {1, 2, ..., n}.
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(13))
360360
>>> a.list(Integer(14))
[1, 1, 2, 6, 12, 60, 60, 420, 840, 2520, 2520, 27720, 27720, 360360]
>>> a(RealNumber('20.0'))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

AUTHOR:

- Jaap Spies (2007-01-31)

class sage.combinat.sloane_functions.**A004086**

Bases: *SloaneSequence*

Read n backwards (referred to as $R(n)$ in many sequences).

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A004086;a
Read n backwards (referred to as R(n) in many sequences).
sage: a(0)
0
sage: a(1)
1
sage: a(2)
2
sage: a(3333)
3333
sage: a(12345)
54321
sage: a.list(13)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 11, 21]
```

```
>>> from sage.all import *
>>> a = sloane.A004086;a
Read n backwards (referred to as R(n) in many sequences).
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(2))
2
>>> a(Integer(3333))
3333
>>> a(Integer(12345))
54321
>>> a.list(Integer(13))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 11, 21]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A004526**

Bases: *SloaneSequence*

The nonnegative integers repeated.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A004526;a
The nonnegative integers repeated.
sage: a(0)
0
sage: a(1)
0
sage: a(2)
1
sage: a(10)
5
sage: a.list(12)
[0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
```

```
>>> from sage.all import *
>>> a = sloane.A004526;a
The nonnegative integers repeated.
>>> a(Integer(0))
0
>>> a(Integer(1))
0
>>> a(Integer(2))
1
>>> a(Integer(10))
5
>>> a.list(Integer(12))
[0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A005100**

Bases: *SloaneSequence*

Deficient numbers: $\sigma(n) < 2n$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A005100;a
Deficient numbers: sigma(n) < 2n
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
```

(continues on next page)

(continued from previous page)

```
2
sage: a(12)
14
sage: a.list(12)
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14]
```

```
>>> from sage.all import *
>>> a = sloane.A005100;a
Deficient numbers: sigma(n) < 2n
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
2
>>> a(Integer(12))
14
>>> a.list(Integer(12))
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14]
```

AUTHORS:

- Jaap Spies (2007-01-26)

list(n)

EXAMPLES:

```
sage: sloane.A005100.list(10)
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
```

```
>>> from sage.all import *
>>> sloane.A005100.list(Integer(10))
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
```

class sage.combinat.sloane_functions.**A005101**

Bases: *SloaneSequence*

Abundant numbers (sum of divisors of n exceeds $2n$).

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A005101;a
Abundant numbers (sum of divisors of n exceeds 2n).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

(continues on next page)

(continued from previous page)

```
sage: a(1)
12
sage: a(2)
18
sage: a(12)
60
sage: a.list(12)
[12, 18, 20, 24, 30, 36, 40, 42, 48, 54, 56, 60]
```

```
>>> from sage.all import *
>>> a = sloane.A005101;a
Abundant numbers (sum of divisors of n exceeds 2n).
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
12
>>> a(Integer(2))
18
>>> a(Integer(12))
60
>>> a.list(Integer(12))
[12, 18, 20, 24, 30, 36, 40, 42, 48, 54, 56, 60]
```

AUTHORS:

- Jaap Spies (2007-01-26)

list(*n*)

EXAMPLES:

```
sage: sloane.A005101.list(10)
[12, 18, 20, 24, 30, 36, 40, 42, 48, 54]
```

```
>>> from sage.all import *
>>> sloane.A005101.list(Integer(10))
[12, 18, 20, 24, 30, 36, 40, 42, 48, 54]
```

class sage.combinat.sloane_functions.**A005117**

Bases: *SloaneSequence*

Square-free numbers.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A005117;a
Square-free numbers.
sage: a(0)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
2
sage: a(12)
17
sage: a.list(12)
[1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17]
```

```
>>> from sage.all import *
>>> a = sloane.A005117;a
Square-free numbers.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(2))
2
>>> a(Integer(12))
17
>>> a.list(Integer(12))
[1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17]
```

AUTHORS:

- Jaap Spies (2007-01-25)

list(*n*)

EXAMPLES:

```
sage: sloane.A005117.list(10)
[1, 2, 3, 5, 6, 7, 10, 11, 13, 14]
```

```
>>> from sage.all import *
>>> sloane.A005117.list(Integer(10))
[1, 2, 3, 5, 6, 7, 10, 11, 13, 14]
```

class sage.combinat.sloane_functions.**A005408**

Bases: *SloaneSequence*

The odd numbers a(n) = 2n + 1.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A005408;a
The odd numbers a(n) = 2n + 1.
sage: a(-1)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
1
sage: a(4)
9
sage: a(11)
23
sage: a.list(12)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

```
>>> from sage.all import *
>>> a = sloane.A005408;a
The odd numbers a(n) = 2n + 1.
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
>>> a(Integer(0))
1
>>> a(Integer(4))
9
>>> a(Integer(11))
23
>>> a.list(Integer(12))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A005843**

Bases: *SloaneSequence*

The even numbers: $a(n) = 2n$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A005843;a
The even numbers: a(n) = 2n.
sage: a(0.0)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
sage: a(1)
2
sage: a(2)
4
sage: a(9)
```

(continues on next page)

(continued from previous page)

```
18
sage: a.list(10)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
>>> from sage.all import *
>>> a = sloane.A005843;a
The even numbers: a(n) = 2n.
>>> a(RealNumber('0.0'))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
>>> a(Integer(1))
2
>>> a(Integer(2))
4
>>> a(Integer(9))
18
>>> a.list(Integer(10))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A006318**

Bases: *SloaneSequence*

Large Schroeder numbers.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A006318;a
Large Schroeder numbers.
sage: a(0)
1
sage: a(1)
2
sage: a(2)
6
sage: a(9)
206098
sage: a.list(10)
[1, 2, 6, 22, 90, 394, 1806, 8558, 41586, 206098]
```

```
>>> from sage.all import *
>>> a = sloane.A006318;a
Large Schroeder numbers.
>>> a(Integer(0))
1
```

(continues on next page)

(continued from previous page)

```
>>> a(Integer(1))
2
>>> a(Integer(2))
6
>>> a(Integer(9))
206098
>>> a.list(Integer(10))
[1, 2, 6, 22, 90, 394, 1806, 8558, 41586, 206098]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A006530**

Bases: *SloaneSequence*

Largest prime dividing n (with $a(1) = 1$).

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A006530;a
Largest prime dividing n (with a(1)=1).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
2
sage: a(8)
2
sage: a(11)
11
sage: a.list(15)
[1, 2, 3, 2, 5, 3, 7, 2, 3, 5, 11, 3, 13, 7, 5]
```

```
>>> from sage.all import *
>>> a = sloane.A006530;a
Largest prime dividing n (with a(1)=1).
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
2
>>> a(Integer(8))
```

(continues on next page)

(continued from previous page)

```
2
>>> a(Integer(11))
11
>>> a.list(Integer(15))
[1, 2, 3, 2, 5, 3, 7, 2, 3, 5, 11, 3, 13, 7, 5]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A006882**

Bases: *SloaneSequence*

Double factorials $n!!$: $a(n) = n \cdot a(n-2)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A006882;a
Double factorials n!!: a(n)=n*a(n-2).
sage: a(0)
1
sage: a(2)
2
sage: a(8)
384
sage: a(20)
3715891200
sage: a.list(9)
[1, 1, 2, 3, 8, 15, 48, 105, 384]
```

```
>>> from sage.all import *
>>> a = sloane.A006882;a
Double factorials n!!: a(n)=n*a(n-2).
>>> a(Integer(0))
1
>>> a(Integer(2))
2
>>> a(Integer(8))
384
>>> a(Integer(20))
3715891200
>>> a.list(Integer(9))
[1, 1, 2, 3, 8, 15, 48, 105, 384]
```

AUTHORS:

- Jaap Spies (2007-01-24)

df()

Double factorials n!!: a(n)=n*a(n-2).

EXAMPLES:

```
sage: it = sloane.A006882.df()
sage: [next(it) for i in range(10)]
[1, 1, 2, 3, 8, 15, 48, 105, 384, 945]
```

```
>>> from sage.all import *
>>> it = sloane.A006882.df()
>>> [next(it) for i in range(Integer(10))]
[1, 1, 2, 3, 8, 15, 48, 105, 384, 945]
```

list (n)

EXAMPLES:

```
sage: sloane.A006882.list(10)
[1, 1, 2, 3, 8, 15, 48, 105, 384, 945]
```

```
>>> from sage.all import *
>>> sloane.A006882.list(Integer(10))
[1, 1, 2, 3, 8, 15, 48, 105, 384, 945]
```

class sage.combinat.sloane_functions.**A007318**

Bases: *SloaneSequence*

Pascal's triangle read by rows: $C(n,k) = \binom{n}{k} = \frac{n!}{(k!(n-k)!)}$, $0 \le k \le n$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A007318
sage: a(0)
1
sage: a(1)
1
sage: a(13)
4
sage: a.list(15)
[1, 1, 1, 1, 2, 1, 1, 3, 3, 1, 1, 4, 6, 4, 1]
sage: a(100)
715
```

```
>>> from sage.all import *
>>> a = sloane.A007318
>>> a(Integer(0))
1
>>> a(Integer(1))
1
>>> a(Integer(13))
4
>>> a.list(Integer(15))
```

(continues on next page)

(continued from previous page)

```
[1, 1, 1, 1, 2, 1, 1, 3, 3, 1, 1, 4, 6, 4, 1]
>>> a(Integer(100))
715
```

AUTHORS:

- Jaap Spies (2007-01-31)

keyword = ['nonn', 'tabl', 'nice', 'easy', 'core', 'triangle']

class sage.combinat.sloane_functions.**A008275**

Bases: *SloaneSequence*

Triangle of Stirling numbers of first kind, $s(n, k)$, $n \geq 1$, $1 \leq k \leq n$.

The unsigned numbers are also called Stirling cycle numbers:

$|s(n, k)|$ = number of permutations of n objects with exactly k cycles.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A008275;a
Triangle of Stirling numbers of first kind, s(n,k), n >= 1, 1<=k<=n.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
-1
sage: a(3)
1
sage: a(11)
24
sage: a.list(12)
[1, -1, 1, 2, -3, 1, -6, 11, -6, 1, 24, -50]
```

```
>>> from sage.all import *
>>> a = sloane.A008275;a
Triangle of Stirling numbers of first kind, s(n,k), n >= 1, 1<=k<=n.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
-1
>>> a(Integer(3))
```

(continues on next page)

(continued from previous page)

```
1
>>> a(Integer(11))
24
>>> a.list(Integer(12))
[1, -1, 1, 2, -3, 1, -6, 11, -6, 1, 24, -50]
```

AUTHORS:

- Jaap Spies (2007-02-02)

keyword = ['sign', 'tabl', 'nice', 'core', 'triangle']

s (*n*, *k*)

EXAMPLES:

```
sage: sloane.A008275.s(4,2)
11
sage: sloane.A008275.s(5,2)
-50
sage: sloane.A008275.s(5,3)
35
```

```
>>> from sage.all import *
>>> sloane.A008275.s(Integer(4),Integer(2))
11
>>> sloane.A008275.s(Integer(5),Integer(2))
-50
>>> sloane.A008275.s(Integer(5),Integer(3))
35
```

class sage.combinat.sloane_functions.**A008277**

Bases: *SloaneSequence*

Triangle of Stirling numbers of 2nd kind, $S2(n, k)$, $n \geq 1$, $1 \leq k \leq n$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A008277;a
Triangle of Stirling numbers of 2nd kind, S2(n,k), n >= 1, 1<=k<=n.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
1
sage: a(3)
1
```

(continues on next page)

(continued from previous page)

```
sage: a(4.0)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
sage: a.list(15)
[1, 1, 1, 1, 3, 1, 1, 7, 6, 1, 1, 15, 25, 10, 1]
```

```
>>> from sage.all import *
>>> a = sloane.A008277;a
Triangle of Stirling numbers of 2nd kind, S2(n,k), n >= 1, 1<=k<=n.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
1
>>> a(Integer(3))
1
>>> a(RealNumber('4.0'))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
>>> a.list(Integer(15))
[1, 1, 1, 1, 3, 1, 1, 7, 6, 1, 1, 15, 25, 10, 1]
```

AUTHORS:

- Jaap Spies (2007-01-31)

keyword = ['nonn', 'tabl', 'nice', 'core', 'triangle']

s2 (n, k)

Return the Stirling number S2(n,k) of the 2nd kind.

EXAMPLES:

```
sage: sloane.A008277.s2(4,2)
7
```

```
>>> from sage.all import *
>>> sloane.A008277.s2(Integer(4),Integer(2))
7
```

class sage.combinat.sloane_functions.**A008683**

Bases: *SloaneSequence*

Möbius function $\mu(n)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A008683;a
Moebius function mu(n).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
-1
sage: a(12)
0
sage: a.list(12)
[1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0]
```

```
>>> from sage.all import *
>>> a = sloane.A008683;a
Moebius function mu(n).
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(2))
-1
>>> a(Integer(12))
0
>>> a.list(Integer(12))
[1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class `sage.combinat.sloane_functions.`**`A010060`**

Bases: `SloaneSequence`

Thue-Morse sequence.

Let A_k denote the first 2^k terms; then $A_0 = 0$, and for $k \geq 0$, $A_{k+1} = A_k B_k$, where B_k is obtained from A_k by interchanging 0s and 1s.

INPUT:

- `n` – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A010060;a
Thue-Morse sequence.
sage: a(0)
0
sage: a(1)
1
sage: a(2)
1
```

(continues on next page)

(continued from previous page)

```
sage: a(12)
0
sage: a.list(13)
[0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0]
```

```
>>> from sage.all import *
>>> a = sloane.A010060;a
Thue-Morse sequence.
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(2))
1
>>> a(Integer(12))
0
>>> a.list(Integer(13))
[0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A015521**

Bases: *RecurrenceSequence2*

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 3a(n-1) + 4a(n-2)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A015521; a
Linear 2nd order recurrence, a(n) = 3 a(n-1) + 4 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(8)
13107
sage: a(41)
967140655691703339764941
sage: a.list(12)
[0, 1, 3, 13, 51, 205, 819, 3277, 13107, 52429, 209715, 838861]
```

```
>>> from sage.all import *
>>> a = sloane.A015521; a
Linear 2nd order recurrence, a(n) = 3 a(n-1) + 4 a(n-2).
>>> a(Integer(0))
0
>>> a(Integer(1))
```

(continues on next page)

(continued from previous page)

```
1
>>> a(Integer(8))
13107
>>> a(Integer(41))
967140655691703339764941
>>> a.list(Integer(12))
[0, 1, 3, 13, 51, 205, 819, 3277, 13107, 52429, 209715, 838861]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A015523**

Bases: *RecurrenceSequence2*

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 3a(n-1) + 5a(n-2)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A015523; a
Linear 2nd order recurrence, a(n) = 3 a(n-1) + 5 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(8)
17727
sage: a(41)
6173719566474529739091481
sage: a.list(12)
[0, 1, 3, 14, 57, 241, 1008, 4229, 17727, 74326, 311613, 1306469]
```

```
>>> from sage.all import *
>>> a = sloane.A015523; a
Linear 2nd order recurrence, a(n) = 3 a(n-1) + 5 a(n-2).
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(8))
17727
>>> a(Integer(41))
6173719566474529739091481
>>> a.list(Integer(12))
[0, 1, 3, 14, 57, 241, 1008, 4229, 17727, 74326, 311613, 1306469]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A015530**

Bases: *RecurrenceSequence2*

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 4a(n-1) + 3a(n-2)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A015530;a
Linear 2nd order recurrence, a(n) = 4 a(n-1) + 3 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(2)
4
sage: a.offset
0
sage: a(8)
41008
sage: a.list(9)
[0, 1, 4, 19, 88, 409, 1900, 8827, 41008]
```

```
>>> from sage.all import *
>>> a = sloane.A015530;a
Linear 2nd order recurrence, a(n) = 4 a(n-1) + 3 a(n-2).
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(2))
4
>>> a.offset
0
>>> a(Integer(8))
41008
>>> a.list(Integer(9))
[0, 1, 4, 19, 88, 409, 1900, 8827, 41008]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A015531**

Bases: *RecurrenceSequence2*

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 4a(n-1) + 5a(n-2)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A015531;a
Linear 2nd order recurrence, a(n) = 4 a(n-1) + 5 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(2)
4
sage: a.offset
0
sage: a(8)
65104
sage: a(60)
144560289664733924534327040115992228190104
sage: a.list(9)
[0, 1, 4, 21, 104, 521, 2604, 13021, 65104]
```

```
>>> from sage.all import *
>>> a = sloane.A015531;a
Linear 2nd order recurrence, a(n) = 4 a(n-1) + 5 a(n-2).
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(2))
4
>>> a.offset
0
>>> a(Integer(8))
65104
>>> a(Integer(60))
144560289664733924534327040115992228190104
>>> a.list(Integer(9))
[0, 1, 4, 21, 104, 521, 2604, 13021, 65104]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A015551**

Bases: *RecurrenceSequence2*

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 6a(n-1) + 5a(n-2)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A015551;a
Linear 2nd order recurrence, a(n) = 6 a(n-1) + 5 a(n-2).
sage: a(0)
0
```

(continues on next page)

(continued from previous page)

```
sage: a(1)
1
sage: a(2)
6
sage: a.offset
0
sage: a(8)
570216
sage: a(60)
7110606606530059736761484557155863822531970573036
sage: a.list(9)
[0, 1, 6, 41, 276, 1861, 12546, 84581, 570216]
```

```
>>> from sage.all import *
>>> a = sloane.A015551;a
Linear 2nd order recurrence, a(n) = 6 a(n-1) + 5 a(n-2).
>>> a(Integer(0))
0
>>> a(Integer(1))
1
>>> a(Integer(2))
6
>>> a.offset
0
>>> a(Integer(8))
570216
>>> a(Integer(60))
7110606606530059736761484557155863822531970573036
>>> a.list(Integer(9))
[0, 1, 6, 41, 276, 1861, 12546, 84581, 570216]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A018252**

Bases: *SloaneSequence*

The nonprime numbers, starting with 1.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A018252;a
The nonprime numbers.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
```

(continues on next page)

(continued from previous page)

```
1
sage: a(2)
4
sage: a(9)
15
sage: a.list(10)
[1, 4, 6, 8, 9, 10, 12, 14, 15, 16]
```

```
>>> from sage.all import *
>>> a = sloane.A018252;a
The nonprime numbers.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
4
>>> a(Integer(9))
15
>>> a.list(Integer(10))
[1, 4, 6, 8, 9, 10, 12, 14, 15, 16]
```

AUTHORS:

- Jaap Spies (2007-02-04)

class sage.combinat.sloane_functions.**A020639**

Bases: *SloaneSequence*

Least prime dividing n with $a(1) = 1$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A020639;a
Least prime dividing n (a(1)=1).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(13)
13
sage: a.list(14)
[1, 2, 3, 2, 5, 2, 7, 2, 3, 2, 11, 2, 13, 2]
```

```
>>> from sage.all import *
>>> a = sloane.A020639;a
Least prime dividing n (a(1)=1).
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(13))
13
>>> a.list(Integer(14))
[1, 2, 3, 2, 5, 2, 7, 2, 3, 2, 11, 2, 13, 2]
```

AUTHORS:

- Jaap Spies (2007-01-25)

list(*n*)

EXAMPLES:

```
sage: sloane.A020639.list(10)
[1, 2, 3, 2, 5, 2, 7, 2, 3, 2]
```

```
>>> from sage.all import *
>>> sloane.A020639.list(Integer(10))
[1, 2, 3, 2, 5, 2, 7, 2, 3, 2]
```

class sage.combinat.sloane_functions.**A046660**(*offset=1*)

Bases: *SloaneSequence*

Excess of n = number of prime divisors (with multiplicity) - number of prime divisors (without multiplicity).

$\Omega(n) - \omega(n)$.

INPUT:

- n – positive integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A046660; a
Excess of n = Bigomega (with multiplicity) - omega (without multiplicity).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(8)
2
sage: a(41)
0
sage: a(84792)
```

(continues on next page)

(continued from previous page)

```
2
sage: a.list(12)
[0, 0, 0, 1, 0, 0, 0, 2, 1, 0, 0, 1]
```

```
>>> from sage.all import *
>>> a = sloane.A046660; a
Excess of n = Bigomega (with multiplicity) - omega (without multiplicity).
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
0
>>> a(Integer(8))
2
>>> a(Integer(41))
0
>>> a(Integer(84792))
2
>>> a.list(Integer(12))
[0, 0, 0, 1, 0, 0, 0, 2, 1, 0, 0, 1]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A049310**

Bases: *SloaneSequence*

Triangle of coefficients of Chebyshev's $S(n, x)$: $U(n, \frac{x}{2})$ polynomials (exponents in increasing order).

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A049310;a
Triangle of coefficients of Chebyshev's S(n,x) := U(n,x/2) polynomials (exponents␣
↪in increasing order).
sage: a(0)
1
sage: a(1)
0
sage: a(13)
0
sage: a.list(15)
[1, 0, 1, -1, 0, 1, 0, -2, 0, 1, 1, 0, -3, 0, 1]
sage: a(200)
0
sage: a.keyword
['sign', 'tabl', 'nice', 'easy', 'core', 'triangle']
```

```
>>> from sage.all import *
>>> a = sloane.A049310;a
Triangle of coefficients of Chebyshev's S(n,x) := U(n,x/2) polynomials (exponents␣
↪in increasing order).
>>> a(Integer(0))
1
>>> a(Integer(1))
0
>>> a(Integer(13))
0
>>> a.list(Integer(15))
[1, 0, 1, -1, 0, 1, 0, -2, 0, 1, 1, 0, -3, 0, 1]
>>> a(Integer(200))
0
>>> a.keyword
['sign', 'tabl', 'nice', 'easy', 'core', 'triangle']
```

AUTHORS:

- Jaap Spies (2007-01-31)

keyword = ['sign', 'tabl', 'nice', 'easy', 'core', 'triangle']

class sage.combinat.sloane_functions.**A051959**

Bases: *RecurrenceSequence*

Linear second order recurrence. A051959.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A051959; a
Linear second order recurrence. A051959.
sage: a(0)
1
sage: a(1)
10
sage: a(8)
9969
sage: a(41)
42834431872413650
sage: a.list(12)
[1, 10, 36, 104, 273, 686, 1688, 4112, 9969, 24114, 58268, 140728]
```

```
>>> from sage.all import *
>>> a = sloane.A051959; a
Linear second order recurrence. A051959.
>>> a(Integer(0))
1
>>> a(Integer(1))
10
```

(continues on next page)

(continued from previous page)

```
>>> a(Integer(8))
9969
>>> a(Integer(41))
42834431872413650
>>> a.list(Integer(12))
[1, 10, 36, 104, 273, 686, 1688, 4112, 9969, 24114, 58268, 140728]
```

AUTHORS:

- Jaap Spies (2007-01-19)

g(k)

EXAMPLES:

```
sage: sloane.A051959.g(2)
15
sage: sloane.A051959.g(1)
0
```

```
>>> from sage.all import *
>>> sloane.A051959.g(Integer(2))
15
>>> sloane.A051959.g(Integer(1))
0
```

class sage.combinat.sloane_functions.**A055790**

Bases: *ExtremesOfPermanentsSequence2*

$a(n) = n * a(n-1) + (n-2) * a(n-2)[a(0) = 0, a(1) = 2]$.

With offset 1, permanent of (0,1)-matrix of size n X (n+d) with d=1 and n-1 zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A055790;a
a(n) = n*a(n-1) + (n-2)*a(n-2) [a(0) = 0, a(1) = 2].
sage: a(0)
0
sage: a(1)
2
sage: a(2)
4
sage: a.offset
0
sage: a(8)
```

(continues on next page)

(continued from previous page)

```
165016
sage: a(22)
10356214297533070441564
sage: a.list(9)
[0, 2, 4, 14, 64, 362, 2428, 18806, 165016]
```

```
>>> from sage.all import *
>>> a = sloane.A055790;a
a(n) = n*a(n-1) + (n-2)*a(n-2) [a(0) = 0, a(1) = 2].
>>> a(Integer(0))
0
>>> a(Integer(1))
2
>>> a(Integer(2))
4
>>> a.offset
0
>>> a(Integer(8))
165016
>>> a(Integer(22))
10356214297533070441564
>>> a.list(Integer(9))
[0, 2, 4, 14, 64, 362, 2428, 18806, 165016]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A061084**

Bases: *SloaneSequence*

Fibonacci-type sequence based on subtraction: $a(0) = 1$, $a(1) = 2$ and $a(n) = a(n-2) - a(n-1)$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A061084; a
Fibonacci-type sequence based on subtraction: a(0) = 1, a(1) = 2 and a(n) = a(n-
→2)-a(n-1).
sage: a(0)
1
sage: a(1)
2
sage: a(8)
-29
sage: a(22)
-24476
sage: a.list(12)
[1, 2, -1, 3, -4, 7, -11, 18, -29, 47, -76, 123]
```

(continues on next page)

(continued from previous page)

```
sage: a.keyword
['sign', 'easy', 'nice']
```

```
>>> from sage.all import *
>>> a = sloane.A061084; a
Fibonacci-type sequence based on subtraction: a(0) = 1, a(1) = 2 and a(n) = a(n-
↪2)-a(n-1).
>>> a(Integer(0))
1
>>> a(Integer(1))
2
>>> a(Integer(8))
-29
>>> a(Integer(22))
-24476
>>> a.list(Integer(12))
[1, 2, -1, 3, -4, 7, -11, 18, -29, 47, -76, 123]
>>> a.keyword
['sign', 'easy', 'nice']
```

AUTHORS:

- Jaap Spies (2007-01-18)

keyword = ['sign', 'easy', 'nice']

class sage.combinat.sloane_functions.**A064553**

Bases: *SloaneSequence*

$a(1) = 1$, $a(prime(i)) = i + 1$ for $i > 0$ and $a(u \cdot v) = a(u) \cdot a(v)$ for $u, v > 0$.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A064553;a
a(1) = 1, a(prime(i)) = i+1 for i > 0 and a(u*v) = a(u)*a(v) for u,v > 0
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
2
sage: a(9)
9
sage: a.list(16)
[1, 2, 3, 4, 4, 6, 5, 8, 9, 8, 6, 12, 7, 10, 12, 16]
```

```
>>> from sage.all import *
>>> a = sloane.A064553;a
a(1) = 1, a(prime(i)) = i+1 for i > 0 and a(u*v) = a(u)*a(v) for u,v > 0
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
1
>>> a(Integer(2))
2
>>> a(Integer(9))
9
>>> a.list(Integer(16))
[1, 2, 3, 4, 4, 6, 5, 8, 9, 8, 6, 12, 7, 10, 12, 16]
```

AUTHORS:

- Jaap Spies (2007-02-04)

class sage.combinat.sloane_functions.**A079922**(*offset=1*)

Bases: *SloaneSequence*

Function returns solutions to the Dancing School problem with n girls and $n + 3$ boys.

The value is $per(B)$, the permanent of the (0,1)-matrix B of size $n \times n + 3$ with $b(i, j) = 1$ if and only if $i \leq j \leq i + n$.

REFERENCES:

- Jaap Spies, Nieuw Archief voor Wiskunde, 5/7 nr 4, December 2006

INPUT:

- n – positive integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A079922; a
Solutions to the Dancing School problem with n girls and n+3 boys
sage: a.offset
1
sage: a(1)
4
sage: a(8)
2227
sage: a.list(8)
[4, 13, 36, 90, 212, 478, 1044, 2227]
```

```
>>> from sage.all import *
>>> a = sloane.A079922; a
Solutions to the Dancing School problem with n girls and n+3 boys
>>> a.offset
1
>>> a(Integer(1))
```

(continues on next page)

(continued from previous page)

```
4
>>> a(Integer(8))
2227
>>> a.list(Integer(8))
[4, 13, 36, 90, 212, 478, 1044, 2227]
```

Compare: Searching Sloane's online database… Solution to the Dancing School Problem with n girls and n+3 boys: f(n,3). [4, 13, 36, 90, 212, 478, 1044, 2227]

```
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

```
>>> from sage.all import *
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

class sage.combinat.sloane_functions.**A079923**(*offset=1*)

Bases: *SloaneSequence*

Function returns solutions to the Dancing School problem with n girls and $n + 4$ boys.

The value is $per(B)$, the permanent of the (0,1)-matrix B of size $n \times n + 3$ with $b(i, j) = 1$ if and only if $i \leq j \leq i + n$.

REFERENCES:

- Jaap Spies, Nieuw Archief voor Wiskunde, 5/7 nr 4, December 2006

INPUT:

- `n` – positive integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A079923; a
Solutions to the Dancing School problem with n girls and n+4 boys
sage: a.offset
1
sage: a(1)
5
sage: a(8)
15458
sage: a.list(8)
[5, 21, 76, 246, 738, 2108, 5794, 15458]
```

```
>>> from sage.all import *
>>> a = sloane.A079923; a
Solutions to the Dancing School problem with n girls and n+4 boys
>>> a.offset
1
>>> a(Integer(1))
5
>>> a(Integer(8))
15458
>>> a.list(Integer(8))
[5, 21, 76, 246, 738, 2108, 5794, 15458]
```

Compare: Searching Sloane's online database… Solution to the Dancing School Problem with n girls and n+4 boys: f(n,4). [5, 21, 76, 246, 738, 2108, 5794, 15458]

```
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

```
>>> from sage.all import *
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-17)

class sage.combinat.sloane_functions.**A082411**

Bases: *RecurrenceSequence2*

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 407389224418$, $a(1) = 76343678551$. This is the second-order linear recurrence sequence with $a(0)$ and $a(1)$ co-prime, that R. L. Graham in 1964 stated did not contain any primes.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A082411;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(1)
76343678551
sage: a(2)
483732902969
sage: a(3)
560076581520
sage: a(20)
2219759332689173
```

(continues on next page)

(continued from previous page)

```
sage: a.list(4)
[407389224418, 76343678551, 483732902969, 560076581520]
```

```
>>> from sage.all import *
>>> a = sloane.A082411;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
>>> a(Integer(1))
76343678551
>>> a(Integer(2))
483732902969
>>> a(Integer(3))
560076581520
>>> a(Integer(20))
2219759332689173
>>> a.list(Integer(4))
[407389224418, 76343678551, 483732902969, 560076581520]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A083103**

Bases: *RecurrenceSequence2*

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 1786772701928802632268715130455793$, $a(1) = 1059683225053915111058165141686995$. This is the second-order linear recurrence sequence with $a(0)$ and $a(1)$ co- prime, that R. L. Graham in 1964 stated did not contain any primes. It has not been verified. Graham made a mistake in the calculation that was corrected by D. E. Knuth in 1990.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A083103;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(1)
1059683225053915111058165141686995
sage: a(2)
2846455926982717743326880272142788
sage: a(3)
3906139152036632854385045413829783
sage: a.offset
0
sage: a(8)
45481392851206651551714764671352204
sage: a(20)
14639253684254059531823985143948191708
sage: a.list(4)
[1786772701928802632268715130455793, 1059683225053915111058165141686995,␣
↪2846455926982717743326880272142788, 3906139152036632854385045413829783]
```

```
>>> from sage.all import *
>>> a = sloane.A083103;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
>>> a(Integer(1))
1059683225053915111058165141686995
>>> a(Integer(2))
2846455926982717743326880272142788
>>> a(Integer(3))
3906139152036632854385045413829783
>>> a.offset
0
>>> a(Integer(8))
45481392851206651551714764671352204
>>> a(Integer(20))
14639253684254059531823985143948191708
>>> a.list(Integer(4))
[1786772701928802632268715130455793, 1059683225053915111058165141686995,
↪2846455926982717743326880272142788, 3906139152036632854385045413829783]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A083104**

Bases: *RecurrenceSequence2*

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 331635635998274737472200656430763$, $a(1) = 1510028911088401971189590305498785$. This is the second-order linear recurrence sequence with $a(0)$ and $a(1)$ co-prime. It was found by Ronald Graham in 1990.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A083104;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(3)
3351693458175078679851381267428333
sage: a.offset
0
sage: a(8)
36021870400834012982120004949074404
sage: a(20)
11601914177621826012468849361236300628
```

```
>>> from sage.all import *
>>> a = sloane.A083104;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
>>> a(Integer(3))
3351693458175078679851381267428333
```

(continues on next page)

(continued from previous page)

```
>>> a.offset
0
>>> a(Integer(8))
36021870400834012982120004949074404
>>> a(Integer(20))
11601914177621826012468849361236300628
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A083105**

Bases: *RecurrenceSequence2*

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 62638280004239857$, $a(1) = 49463435743205655$. This is the second-order linear recurrence sequence with $a(0)$ and $a(1)$ co-prime. It was found by Donald Knuth in 1990.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A083105;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(1)
49463435743205655
sage: a(2)
112101715747445512
sage: a(3)
161565151490651167
sage: a.offset
0
sage: a(8)
1853029790662436896
sage: a(20)
596510791500513098192
sage: a.list(4)
[62638280004239857, 49463435743205655, 112101715747445512, 161565151490651167]
```

```
>>> from sage.all import *
>>> a = sloane.A083105;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
>>> a(Integer(1))
49463435743205655
>>> a(Integer(2))
112101715747445512
>>> a(Integer(3))
161565151490651167
>>> a.offset
0
>>> a(Integer(8))
```

(continues on next page)

(continued from previous page)

```
18530297906624368 96
>>> a(Integer(20))
5965107915005130981 92
>>> a.list(Integer(4))
[62638280004239857, 49463435743205655, 112101715747445512, 161565151490651167]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A083216**

Bases: *RecurrenceSequence2*

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 20615674205555510$, $a(1) = 3794765361567513$. This is a second-order linear recurrence sequence with $a(0)$ and $a(1)$ co-prime that does not contain any primes. It was found by Herbert Wilf in 1990.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A083216; a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(0)
20615674205555510
sage: a(1)
3794765361567513
sage: a(8)
347693837265139403
sage: a(41)
2738025383211084205003383
sage: a.list(4)
[20615674205555510, 3794765361567513, 24410439567123023, 28205204928690536]
```

```
>>> from sage.all import *
>>> a = sloane.A083216; a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
>>> a(Integer(0))
20615674205555510
>>> a(Integer(1))
3794765361567513
>>> a(Integer(8))
347693837265139403
>>> a(Integer(41))
2738025383211084205003383
>>> a.list(Integer(4))
[20615674205555510, 3794765361567513, 24410439567123023, 28205204928690536]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A090010**

Bases: *ExtremesOfPermanentsSequence2*

Permanent of (0,1)-matrix of size $n \times (n + d)$ with $d = 6$ and n zeros not on a line.

` a(n) = (n+5)*a(n-1) + (n-1)*a(n-2), a(1)=6, a(2)=43`.

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A090010;a
Permanent of (0,1)-matrix of size n X (n+d) with d=6 and n zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
6
sage: a(2)
43
sage: a.offset
1
sage: a(8)
67741129
sage: a(22)
192416593029158989003270143
sage: a.list(9)
[6, 43, 356, 3333, 34754, 398959, 4996032, 67741129, 988344062]
```

```
>>> from sage.all import *
>>> a = sloane.A090010;a
Permanent of (0,1)-matrix of size n X (n+d) with d=6 and n zeros not on a line.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
6
>>> a(Integer(2))
43
>>> a.offset
1
>>> a(Integer(8))
67741129
```

(continues on next page)

(continued from previous page)

```
>>> a(Integer(22))
192416593029158989003270143
>>> a.list(Integer(9))
[6, 43, 356, 3333, 34754, 398959, 4996032, 67741129, 988344062]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090012**

Bases: *SloaneSequence*

Permanent of (0,1)-matrix of size $n \times (n + d)$ with $d = 2$ and $n - 1$ zeros not on a line.

$a(n) = (n + 1) * a(n - 1) + (n - 2) * a(n - 2)$, $a(1) = 3$ and $a(2) = 9$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A090012;a
Permanent of (0,1)-matrix of size n X (n+d) with d=2 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
3
sage: a(2)
9
sage: a.offset
1
sage: a(8)
890901
sage: a(22)
129020386652297208795129
sage: a.list(9)
[3, 9, 39, 213, 1395, 10617, 91911, 890901, 9552387]
```

```
>>> from sage.all import *
>>> a = sloane.A090012;a
Permanent of (0,1)-matrix of size n X (n+d) with d=2 and n-1 zeros not on a line.
>>> a(Integer(0))
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
3
>>> a(Integer(2))
9
>>> a.offset
1
>>> a(Integer(8))
890901
>>> a(Integer(22))
129020386652297208795129
>>> a.list(Integer(9))
[3, 9, 39, 213, 1395, 10617, 91911, 890901, 9552387]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090013**

Bases: *SloaneSequence*

Permanent of (0,1)-matrix of size $n \times (n + d)$ with $d = 3$ and $n - 1$ zeros not on a line.

$a(n) = (n + 1) * a(n - 1) + (n - 2) * a(n - 2)[a(1) = 4, a(2) = 16]$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A090013;a
Permanent of (0,1)-matrix of size n X (n+d) with d=3 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
4
sage: a(2)
16
sage: a.offset
1
sage: a(8)
3481096
sage: a(22)
1112998577171142607670336
```

(continues on next page)

(continued from previous page)

```
sage: a.list(9)
[4, 16, 84, 536, 4004, 34176, 327604, 3481096, 40585284]
```

```
>>> from sage.all import *
>>> a = sloane.A090013;a
Permanent of (0,1)-matrix of size n X (n+d) with d=3 and n-1 zeros not on a line.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
4
>>> a(Integer(2))
16
>>> a.offset
1
>>> a(Integer(8))
3481096
>>> a(Integer(22))
111299857717114260767033б
>>> a.list(Integer(9))
[4, 16, 84, 536, 4004, 34176, 327604, 3481096, 40585284]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090014**

Bases: *SloaneSequence*

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 4$ and $n - 1$ zeros not on a line.

$a(n) = (n+1) * a(n-1) + (n-2) * a(n-2)[a(1) = 5, a(2) = 25]$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A090014;a
Permanent of (0,1)-matrix of size n X (n+d) with d=4 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
```

(continues on next page)

(continued from previous page)

```
5
sage: a(2)
25
sage: a.offset
1
sage: a(8)
11016595
sage: a(22)
7469733600354446865509725
sage: a.list(9)
[5, 25, 155, 1135, 9545, 90445, 952175, 11016595, 138864365]
```

```
>>> from sage.all import *
>>> a = sloane.A090014;a
Permanent of (0,1)-matrix of size n X (n+d) with d=4 and n-1 zeros not on a line.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
5
>>> a(Integer(2))
25
>>> a.offset
1
>>> a(Integer(8))
11016595
>>> a(Integer(22))
7469733600354446865509725
>>> a.list(Integer(9))
[5, 25, 155, 1135, 9545, 90445, 952175, 11016595, 138864365]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090015**

Bases: *SloaneSequence*

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 5$ and $n - 1$ zeros not on a line.

$a(n) = (n+1) * a(n-1) + (n-2) * a(n-2)[a(1) = 6, a(2) = 36]$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A090015;a
Permanent of (0,1)-matrix of size n X (n+d) with d=3 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
6
sage: a(2)
36
sage: a.offset
1
sage: a(8)
29976192
sage: a(22)
415522585176921167949368 76
sage: a.list(9)
[6, 36, 258, 2136, 19998, 208524, 2393754, 29976192, 406446774]
```

```
>>> from sage.all import *
>>> a = sloane.A090015;a
Permanent of (0,1)-matrix of size n X (n+d) with d=3 and n-1 zeros not on a line.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
6
>>> a(Integer(2))
36
>>> a.offset
1
>>> a(Integer(8))
29976192
>>> a(Integer(22))
41552258517692116794936876
>>> a.list(Integer(9))
[6, 36, 258, 2136, 19998, 208524, 2393754, 29976192, 406446774]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090016**

Bases: *SloaneSequence*

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 6$ and $n - 1$ zeros not on a line.

$a(n) = (n+1) * a(n-1) + (n-2) * a(n-2)[a(1) = 7, a(2) = 49]$

$A090016a(n) = A090010(n-1) + A090010(n), a(1) = 7$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- `n` – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A090016;a
Permanent of (0,1)-matrix of size n X (n+d) with d=6 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
7
sage: a(2)
49
sage: a.offset
1
sage: a(8)
72737161
sage: a(22)
199341969448774341802426289
sage: a.list(9)
[7, 49, 399, 3689, 38087, 433713, 5394991, 72737161, 1056085191]
```

```
>>> from sage.all import *
>>> a = sloane.A090016;a
Permanent of (0,1)-matrix of size n X (n+d) with d=6 and n-1 zeros not on a line.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(1))
7
>>> a(Integer(2))
49
>>> a.offset
1
>>> a(Integer(8))
72737161
>>> a(Integer(22))
199341969448774341802426289
>>> a.list(Integer(9))
[7, 49, 399, 3689, 38087, 433713, 5394991, 72737161, 1056085191]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A109814**

Bases: *SloaneSequence*

The n th term of the sequence $a(n)$ is the largest k such that n can be written as sum of k consecutive integers.

By definition, n is the sum of at most $a(n)$ consecutive positive integers. Suppose n is to be written as sum of k consecutive integers starting with m, then $2n = k(2m + k - 1)$. Only one of the factors is odd. For each odd divisor d of n there is a unique corresponding $k = min(d, 2n/d)$. $a(n)$ can be alternatively defined as the largest among those k .

See also

- Wikipedia article Polite_number
- An exercise sheet (with answers) about sums of consecutive integers.

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A109814; a
a(n) is the largest k such that n can be written as sum of k consecutive positive
↪integers.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
1
sage: a.list(9)
[1, 1, 2, 1, 2, 3, 2, 1, 3]
```

```
>>> from sage.all import *
>>> a = sloane.A109814; a
a(n) is the largest k such that n can be written as sum of k consecutive positive
↪integers.
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(2))
1
>>> a.list(Integer(9))
[1, 1, 2, 1, 2, 3, 2, 1, 3]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A111774**

Bases: *SloaneSequence*

Sequence of numbers of the third kind, i.e., numbers that can be written as a sum of at least three consecutive positive integers.

Odd primes can only be written as a sum of two consecutive integers. Powers of 2 do not have a representation as a sum of k consecutive integers (other than the trivial $n = n$ for $k = 1$).

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A111774; a
Numbers that can be written as a sum of at least three consecutive positive␣
↪integers.
sage: a(1)
6
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(100)
141
sage: a(156)
209
sage: a(302)
386
sage: a.list(12)
[6, 9, 10, 12, 14, 15, 18, 20, 21, 22, 24, 25]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

```
>>> from sage.all import *
>>> a = sloane.A111774; a
Numbers that can be written as a sum of at least three consecutive positive␣
↪integers.
>>> a(Integer(1))
6
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(100))
141
>>> a(Integer(156))
209
>>> a(Integer(302))
386
>>> a.list(Integer(12))
[6, 9, 10, 12, 14, 15, 18, 20, 21, 22, 24, 25]
>>> a(Integer(1)/Integer(3))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: input must be an int or Integer
```

AUTHORS:

- Jaap Spies (2007-01-13)

is_number_of_the_third_kind(n)

Return `True` if and only if n is a number of the third kind.

A number is of the third kind if it can be written as a sum of at least three consecutive positive integers. Odd primes can only be written as a sum of two consecutive integers. Powers of 2 do not have a representation as a sum of k consecutive integers (other than the trivial $n = n$ for $k = 1$).

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

- `n` – positive integer

OUTPUT: `True` if n is not prime and not a power of 2

EXAMPLES:

```
sage: a = sloane.A111774
sage: a.is_number_of_the_third_kind(6)
True
sage: a.is_number_of_the_third_kind(100)
True
sage: a.is_number_of_the_third_kind(16)
False
sage: a.is_number_of_the_third_kind(97)
False
```

```
>>> from sage.all import *
>>> a = sloane.A111774
>>> a.is_number_of_the_third_kind(Integer(6))
True
>>> a.is_number_of_the_third_kind(Integer(100))
True
>>> a.is_number_of_the_third_kind(Integer(16))
False
>>> a.is_number_of_the_third_kind(Integer(97))
False
```

AUTHORS:

- Jaap Spies (2006-12-09)

list(n)

EXAMPLES:

```
sage: sloane.A111774.list(12)
[6, 9, 10, 12, 14, 15, 18, 20, 21, 22, 24, 25]
```

```
>>> from sage.all import *
>>> sloane.A111774.list(Integer(12))
[6, 9, 10, 12, 14, 15, 18, 20, 21, 22, 24, 25]
```

class sage.combinat.sloane_functions.**A111775**

Bases: *SloaneSequence*

Number of ways n can be written as a sum of at least three consecutive integers.

Powers of 2 and (odd) primes can not be written as a sum of at least three consecutive integers. $a(n)$ strongly depends on the number of odd divisors of n (A001227): Suppose n is to be written as sum of k consecutive integers starting with m, then $2n = k(2m + k - 1)$. Only one of the factors is odd. For each odd divisor of n there is a unique corresponding k, $k = 1$ and $k = 2$ must be excluded.

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A111775; a
Number of ways n can be written as a sum of at least three consecutive integers.
```

```
>>> from sage.all import *
>>> a = sloane.A111775; a
Number of ways n can be written as a sum of at least three consecutive integers.
```

```
sage: a(1)
0
sage: a(0)
0
```

```
>>> from sage.all import *
>>> a(Integer(1))
0
>>> a(Integer(0))
0
```

We have a(15)=2 because 15 = 4+5+6 and 15 = 1+2+3+4+5. The number of odd divisors of 15 is 4.

```
sage: a(15)
2
```

```
>>> from sage.all import *
>>> a(Integer(15))
2
```

```
sage: a(100)
2
sage: a(256)
0
```

(continues on next page)

(continued from previous page)

```
sage: a(29)
0
sage: a.list(20)
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 2, 0, 0, 2, 0]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

```
>>> from sage.all import *
>>> a(Integer(100))
2
>>> a(Integer(256))
0
>>> a(Integer(29))
0
>>> a.list(Integer(20))
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 2, 0, 0, 2, 0]
>>> a(Integer(1)/Integer(3))
Traceback (most recent call last):
...
TypeError: input must be an int or Integer
```

AUTHORS:

- Jaap Spies (2006-12-09)

class sage.combinat.sloane_functions.**A111787**

Bases: *SloaneSequence*

This function returns the n-th number of Sloane's sequence A111787

$a(n) = 0$ if n is an odd prime or a power of 2. For numbers of the third kind (see A111774) we proceed as follows: suppose n is to be written as sum of k consecutive integers starting with m, then $2n = k(2m + k - 1)$. Let p be the smallest odd prime divisor of n then $a(n) = min(p, 2n/p)$.

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

- n – nonnegative integer

OUTPUT: integer; function value

EXAMPLES:

```
sage: a = sloane.A111787; a
a(n) is the least k >= 3 such that n can be written as sum of k consecutive␣
↪integers. a(n)=0 if such a k does not exist.
sage: a.offset
1
sage: a(1)
0
sage: a(0)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: input n (=0) must be a positive integer
sage: a(100)
5
sage: a(256)
0
sage: a(29)
0
sage: a.list(20)
[0, 0, 0, 0, 0, 3, 0, 0, 3, 4, 0, 3, 0, 4, 3, 0, 0, 3, 0, 5]
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

```
>>> from sage.all import *
>>> a = sloane.A111787; a
a(n) is the least k >= 3 such that n can be written as sum of k consecutive␣
↪integers. a(n)=0 if such a k does not exist.
>>> a.offset
1
>>> a(Integer(1))
0
>>> a(Integer(0))
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
>>> a(Integer(100))
5
>>> a(Integer(256))
0
>>> a(Integer(29))
0
>>> a.list(Integer(20))
[0, 0, 0, 0, 0, 3, 0, 0, 3, 4, 0, 3, 0, 4, 3, 0, 0, 3, 0, 5]
>>> a(-Integer(1))
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

class sage.combinat.sloane_functions.**ExponentialNumbers**(*a*)

Bases: *SloaneSequence*

A sequence of Exponential numbers.

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import ExponentialNumbers
sage: ExponentialNumbers(0)
Sequence of Exponential numbers around 0
```

```
>>> from sage.all import *
>>> from sage.combinat.sloane_functions import ExponentialNumbers
>>> ExponentialNumbers(Integer(0))
Sequence of Exponential numbers around 0
```

class sage.combinat.sloane_functions.**ExtremesOfPermanentsSequence**(*offset=1*)

Bases: *SloaneSequence*

gen(*a0*, *a1*, *d*)

EXAMPLES:

```
sage: it = sloane.A000153.gen(0,1,2)
sage: [next(it) for i in range(5)]
[0, 1, 2, 7, 32]
```

```
>>> from sage.all import *
>>> it = sloane.A000153.gen(Integer(0),Integer(1),Integer(2))
>>> [next(it) for i in range(Integer(5))]
[0, 1, 2, 7, 32]
```

list(*n*)

EXAMPLES:

```
sage: sloane.A000153.list(8)
[0, 1, 2, 7, 32, 181, 1214, 9403]
```

```
>>> from sage.all import *
>>> sloane.A000153.list(Integer(8))
[0, 1, 2, 7, 32, 181, 1214, 9403]
```

class sage.combinat.sloane_functions.**ExtremesOfPermanentsSequence2**(*offset=1*)

Bases: *ExtremesOfPermanentsSequence*

gen(*a0*, *a1*, *d*)

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import ExtremesOfPermanentsSequence2
sage: e = ExtremesOfPermanentsSequence2()
sage: it = e.gen(6,43,6)
sage: [next(it) for i in range(5)]
[6, 43, 307, 2542, 23799]
```

```
>>> from sage.all import *
>>> from sage.combinat.sloane_functions import ExtremesOfPermanentsSequence2
>>> e = ExtremesOfPermanentsSequence2()
>>> it = e.gen(Integer(6),Integer(43),Integer(6))
>>> [next(it) for i in range(Integer(5))]
[6, 43, 307, 2542, 23799]
```

class sage.combinat.sloane_functions.**RecurrenceSequence**(*offset=1*)

Bases: *SloaneSequence*

list(*n*)

EXAMPLES:

```
sage: sloane.A001110.list(8)
[0, 1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

```
>>> from sage.all import *
>>> sloane.A001110.list(Integer(8))
[0, 1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

class sage.combinat.sloane_functions.**RecurrenceSequence2**(*offset=1*)

Bases: *SloaneSequence*

list(*n*)

EXAMPLES:

```
sage: sloane.A001906.list(10)
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584]
```

```
>>> from sage.all import *
>>> sloane.A001906.list(Integer(10))
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584]
```

class sage.combinat.sloane_functions.**Sloane**

Bases: SageObject

A collection of Sloane generating functions.

This class inspects sage.combinat.sloane_functions, accumulating all the SloaneSequence classes starting with 'A'. These are listed for tab completion, but not instantiated until requested.

EXAMPLES:

Ensure we have lots of entries:

```
sage: len(sloane.__dir__()) > 100
True
```

```
>>> from sage.all import *
>>> len(sloane.__dir__()) > Integer(100)
True
```

Ensure none are being incorrectly returned:

```
sage: [ None for n in sloane.__dir__() if not n.startswith('A') ]
[]
```

```
>>> from sage.all import *
>>> [ None for n in sloane.__dir__() if not n.startswith('A') ]
[]
```

Ensure we can access dynamic constructions and cache correctly:

```
sage: s = sloane.A000587
sage: s is sloane.A000587
True
```

```
>>> from sage.all import *
>>> s = sloane.A000587
>>> s is sloane.A000587
True
```

Ensure that we can access other functions in parent classes:

```
sage: sloane.__class__
<class 'sage.combinat.sloane_functions.Sloane'>
```

```
>>> from sage.all import *
>>> sloane.__class__
<class 'sage.combinat.sloane_functions.Sloane'>
```

AUTHORS:

- Nick Alexander

class sage.combinat.sloane_functions.**SloaneSequence**(*offset=1*)

Bases: SageObject

Base class for a Sloane integer sequence.

list(*n*)

Return n terms of the sequence:

```
sequence[offset], sequence[offset+1], ..., sequence[offset+n-1].
```

EXAMPLES:

```
sage: sloane.A000012.list(4)
[1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> sloane.A000012.list(Integer(4))
[1, 1, 1, 1]
```

sage.combinat.sloane_functions.**perm_mh**(*m*, *h*)

This functions calculates $f(g, h)$ from Sloane's sequences A079908-A079928

INPUT:

- m – positive integer
- h – nonnegative integer

OUTPUT: permanent of the $m \times (m + h)$ matrix, etc.

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import perm_mh
sage: perm_mh(3,3)
```

(continues on next page)

(continued from previous page)

```
36
sage: perm_mh(3,4)
76
```

```
>>> from sage.all import *
>>> from sage.combinat.sloane_functions import perm_mh
>>> perm_mh(Integer(3),Integer(3))
36
>>> perm_mh(Integer(3),Integer(4))
76
```

AUTHORS:

- Jaap Spies (2006)

sage.combinat.sloane_functions.**recur_gen2**(*a0*, *a1*, *a2*, *a3*)

Homogeneous general second-order linear recurrence generator with fixed coefficients.

a(0) = a0, a(1) = a1, a(n) = a2*a(n-1) + a3*a(n-2)

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen2
sage: it = recur_gen2(1,1,1,1)
sage: [next(it) for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
>>> from sage.all import *
>>> from sage.combinat.sloane_functions import recur_gen2
>>> it = recur_gen2(Integer(1),Integer(1),Integer(1),Integer(1))
>>> [next(it) for i in range(Integer(10))]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

sage.combinat.sloane_functions.**recur_gen2b**(*a0*, *a1*, *a2*, *a3*, *b*)

Inhomogeneous second-order linear recurrence generator with fixed coefficients and $b = f(n)$

$a(0) = a0$, $a(1) = a1$, $a(n) = a2 * a(n-1) + a3 * a(n-2) + f(n)$.

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen2b
sage: it = recur_gen2b(1,1,1,1, lambda n: 0)
sage: [next(it) for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
>>> from sage.all import *
>>> from sage.combinat.sloane_functions import recur_gen2b
>>> it = recur_gen2b(Integer(1),Integer(1),Integer(1),Integer(1), lambda n:␣
→Integer(0))
>>> [next(it) for i in range(Integer(10))]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

sage.combinat.sloane_functions.**recur_gen3**(*a0*, *a1*, *a2*, *a3*, *a4*, *a5*)

Homogeneous general third-order linear recurrence generator with fixed coefficients a(0) = a0, a(1) = a1, a(2) = a2, a(n) = a3*a(n-1) + a4*a(n-2) + a5*a(n-3)

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen3
sage: it = recur_gen3(1,1,1,1,1,1)
sage: [next(it) for i in range(10)]
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105]
```

```
>>> from sage.all import *
>>> from sage.combinat.sloane_functions import recur_gen3
>>> it = recur_gen3(Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1))
>>> [next(it) for i in range(Integer(10))]
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105]
```

5.1.315 Combinatorial species

Todo

Short blurb about species

Todo

Proofread / point to the main classes rather than the modules?

Introductory material

- *Enumeration of trees using generating functions*
- *Species, decomposable combinatorial classes*

Basic Species

- *Combinatorial Species*
- *Empty Species*
- *Recursive Species*
- *Characteristic Species*
- *Cycle Species*
- *Partition Species*
- *Permutation species*
- *Linear-order Species*
- *Set Species*
- *Subset Species*
- *Examples of Combinatorial Species*

Operations on Species

- *Sum species*
- *Product species*
- *Composition species*
- *Functorial composition species*

Miscellaneous

- *Species structures*
- *Miscellaneous Functions*

5.1.316 Characteristic Species

class sage.combinat.species.characteristic_species.**CharacteristicSpecies**(*n*, *min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the characteristic species of order n.

This species has exactly one structure on a set of size n and no structures on sets of any other size.

EXAMPLES:

```
sage: X = species.CharacteristicSpecies(1)
sage: X.structures([1]).list()
[1]
sage: X.structures([1,2]).list()
[]
sage: X.generating_series()[0:4]
[0, 1, 0, 0]
sage: X.isotype_generating_series()[0:4]
[0, 1, 0, 0]
sage: X.cycle_index_series()[0:4]                                    #␣
↪needs sage.modules
[0, p[1], 0, 0]

sage: F = species.CharacteristicSpecies(3)
sage: c = F.generating_series()[0:4]
sage: F._check()
True
sage: F == loads(dumps(F))
True
```

```
>>> from sage.all import *
>>> X = species.CharacteristicSpecies(Integer(1))
>>> X.structures([Integer(1)]).list()
[1]
>>> X.structures([Integer(1),Integer(2)]).list()
[]
>>> X.generating_series()[Integer(0):Integer(4)]
[0, 1, 0, 0]
```

(continues on next page)

(continued from previous page)

```
>>> X.isotype_generating_series()[Integer(0):Integer(4)]
[0, 1, 0, 0]
>>> X.cycle_index_series()[Integer(0):Integer(4)]                                   ␣
↪           # needs sage.modules
[0, p[1], 0, 0]

>>> F = species.CharacteristicSpecies(Integer(3))
>>> c = F.generating_series()[Integer(0):Integer(4)]
>>> F._check()
True
>>> F == loads(dumps(F))
True
```

class sage.combinat.species.characteristic_species.**CharacteristicSpeciesStructure**(*parent*, *labels*, *list*)

Bases: *GenericSpeciesStructure*

automorphism_group()

Return the group of permutations whose action on this structure leave it fixed. For the characteristic species, there is only one structure, so every permutation is in its automorphism group.

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.automorphism_group()                                                        #␣
↪needs sage.groups
Symmetric group of order 3! as a permutation group
```

```
>>> from sage.all import *
>>> F = species.CharacteristicSpecies(Integer(3))
>>> a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
>>> a.automorphism_group()                                                          #␣
↪needs sage.groups
Symmetric group of order 3! as a permutation group
```

canonical_label()

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.canonical_label()
{'a', 'b', 'c'}
```

```
>>> from sage.all import *
>>> F = species.CharacteristicSpecies(Integer(3))
>>> a = F.structures(["a", "b", "c"]).random_element(); a
```

(continues on next page)

(continued from previous page)

```
{'a', 'b', 'c'}
>>> a.canonical_label()
{'a', 'b', 'c'}
```

transport(*perm*)

Return the transport of this structure along the permutation `perm`.

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: p = PermutationGroupElement((1,2))                                    #␣
↪needs sage.groups
sage: a.transport(p)                                                        #␣
↪needs sage.groups
{'a', 'b', 'c'}
```

```
>>> from sage.all import *
>>> F = species.CharacteristicSpecies(Integer(3))
>>> a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
>>> p = PermutationGroupElement((Integer(1),Integer(2)))                     ␣
↪              # needs sage.groups
>>> a.transport(p)                                                         #␣
↪needs sage.groups
{'a', 'b', 'c'}
```

sage.combinat.species.characteristic_species.**CharacteristicSpecies_class**

alias of *CharacteristicSpecies*

class sage.combinat.species.characteristic_species.**EmptySetSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *CharacteristicSpecies*

Return the empty set species.

This species has exactly one structure on the empty set. It is the same (and is implemented) as `CharacteristicSpecies(0)`.

EXAMPLES:

```
sage: X = species.EmptySetSpecies()
sage: X.structures([]).list()
[{}]
sage: X.structures([1,2]).list()
[]
sage: X.generating_series()[0:4]
[1, 0, 0, 0]
sage: X.isotype_generating_series()[0:4]
[1, 0, 0, 0]
sage: X.cycle_index_series()[0:4]                                         #␣
↪needs sage.modules
[p[], 0, 0, 0]
```

```
>>> from sage.all import *
>>> X = species.EmptySetSpecies()
>>> X.structures([]).list()
[{}]
>>> X.structures([Integer(1),Integer(2)]).list()
[]
>>> X.generating_series()[Integer(0):Integer(4)]
[1, 0, 0, 0]
>>> X.isotype_generating_series()[Integer(0):Integer(4)]
[1, 0, 0, 0]
>>> X.cycle_index_series()[Integer(0):Integer(4)]                              ␣
↪          # needs sage.modules
[p[], 0, 0, 0]
```

sage.combinat.species.characteristic_species.**EmptySetSpecies_class**

alias of *EmptySetSpecies*

class sage.combinat.species.characteristic_species.**SingletonSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *CharacteristicSpecies*

Return the species of singletons.

This species has exactly one structure on a set of size 1. It is the same (and is implemented) as `CharacteristicSpecies(1)`.

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: X.structures([1]).list()
[1]
sage: X.structures([1,2]).list()
[]
sage: X.generating_series()[0:4]
[0, 1, 0, 0]
sage: X.isotype_generating_series()[0:4]
[0, 1, 0, 0]
sage: X.cycle_index_series()[0:4]                                          #␣
↪needs sage.modules
[0, p[1], 0, 0]
```

```
>>> from sage.all import *
>>> X = species.SingletonSpecies()
>>> X.structures([Integer(1)]).list()
[1]
>>> X.structures([Integer(1),Integer(2)]).list()
[]
>>> X.generating_series()[Integer(0):Integer(4)]
[0, 1, 0, 0]
>>> X.isotype_generating_series()[Integer(0):Integer(4)]
[0, 1, 0, 0]
>>> X.cycle_index_series()[Integer(0):Integer(4)]                              ␣
↪          # needs sage.modules
[0, p[1], 0, 0]
```

sage.combinat.species.characteristic_species.**SingletonSpecies_class**

alias of *SingletonSpecies*

5.1.317 Composition species

class sage.combinat.species.composition_species.**CompositionSpecies**(*F*, *G*, *min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the composition of two species.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: C = species.CycleSpecies()
sage: S = E(C)
sage: S.generating_series()[:5]
[1, 1, 1, 1, 1]
sage: E(C) is S
True
```

```
>>> from sage.all import *
>>> E = species.SetSpecies()
>>> C = species.CycleSpecies()
>>> S = E(C)
>>> S.generating_series()[:Integer(5)]
[1, 1, 1, 1, 1]
>>> E(C) is S
True
```

weight_ring()

Return the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: L.weight_ring()
Rational Field
```

```
>>> from sage.all import *
>>> E = species.SetSpecies(); C = species.CycleSpecies()
>>> L = E(C)
>>> L.weight_ring()
Rational Field
```

class sage.combinat.species.composition_species.**CompositionSpeciesStructure**(*parent*, *labels*, *pi*, *f*, *gs*)

Bases: *GenericSpeciesStructure*

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- `labels` – list of labels

OUTPUT:

A structure with the i-th label of `self` replaced with the i-th label of the list.

EXAMPLES:

```
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: S = L.structures(['a','b','c']).list()                               #␣
↪needs sage.libs.flint
sage: a = S[2]; a                                                          #␣
↪needs sage.libs.flint
F-structure: {{'a', 'c'}, {'b'}}; G-structures: (('a', 'c'), ('b'))
sage: a.change_labels([1,2,3])                                             #␣
↪needs sage.libs.flint
F-structure: {{1, 3}, {2}}; G-structures: [(1, 3), (2)]
```

```
>>> from sage.all import *
>>> E = species.SetSpecies(); C = species.CycleSpecies()
>>> L = E(C)
>>> S = L.structures(['a','b','c']).list()                               #␣
↪needs sage.libs.flint
>>> a = S[Integer(2)]; a                                                  ␣
↪     # needs sage.libs.flint
F-structure: {{'a', 'c'}, {'b'}}; G-structures: (('a', 'c'), ('b'))
>>> a.change_labels([Integer(1),Integer(2),Integer(3)])                   ␣
↪                         # needs sage.libs.flint
F-structure: {{1, 3}, {2}}; G-structures: [(1, 3), (2)]
```

transport(*perm*)

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))                                   #␣
↪needs sage.groups
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: S = L.structures(['a','b','c']).list()                               #␣
↪needs sage.libs.flint
sage: a = S[2]; a                                                          #␣
↪needs sage.libs.flint
F-structure: {{'a', 'c'}, {'b'}}; G-structures: (('a', 'c'), ('b'))
sage: a.transport(p)                                                       #␣
↪needs sage.groups sage.libs.flint
F-structure: {{'a', 'b'}, {'c'}}; G-structures: (('a', 'c'), ('b'))
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement((Integer(2),Integer(3)))                  ␣
↪                # needs sage.groups
>>> E = species.SetSpecies(); C = species.CycleSpecies()
>>> L = E(C)
>>> S = L.structures(['a','b','c']).list()                               #␣
↪needs sage.libs.flint
```

(continues on next page)

(continued from previous page)

```
>>> a = S[Integer(2)]; a                                                        ␣
↪     # needs sage.libs.flint
F-structure: {{'a', 'c'}, {'b'}}; G-structures: (('a', 'c'), ('b'))
>>> a.transport(p)                                                            #␣
↪needs sage.groups sage.libs.flint
F-structure: {{'a', 'b'}, {'c'}}; G-structures: (('a', 'c'), ('b'))
```

sage.combinat.species.composition_species.**CompositionSpecies_class**

alias of *CompositionSpecies*

5.1.318 Cycle Species

class sage.combinat.species.cycle_species.**CycleSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the species of cycles.

EXAMPLES:

```
sage: C = species.CycleSpecies(); C
Cyclic permutation species
sage: C.structures([1,2,3,4]).list()
[(1, 2, 3, 4),
 (1, 2, 4, 3),
 (1, 3, 2, 4),
 (1, 3, 4, 2),
 (1, 4, 2, 3),
 (1, 4, 3, 2)]
```

```
>>> from sage.all import *
>>> C = species.CycleSpecies(); C
Cyclic permutation species
>>> C.structures([Integer(1),Integer(2),Integer(3),Integer(4)]).list()
[(1, 2, 3, 4),
 (1, 2, 4, 3),
 (1, 3, 2, 4),
 (1, 3, 4, 2),
 (1, 4, 2, 3),
 (1, 4, 3, 2)]
```

class sage.combinat.species.cycle_species.**CycleSpeciesStructure**(*parent*, *labels*, *list*)

Bases: *GenericSpeciesStructure*

automorphism_group()

Return the group of permutations whose action on this structure leave it fixed.

EXAMPLES:

```
sage: P = species.CycleSpecies()
sage: a = P.structures([1, 2, 3, 4])[0]; a
(1, 2, 3, 4)
sage: a.automorphism_group()                                                  #␣
↪needs sage.groups
Permutation Group with generators [(1,2,3,4)]
```

```
>>> from sage.all import *
>>> P = species.CycleSpecies()
>>> a = P.structures([Integer(1), Integer(2), Integer(3),␣
↪Integer(4)])[Integer(0)]; a
(1, 2, 3, 4)
>>> a.automorphism_group()                                          #␣
↪needs sage.groups
Permutation Group with generators [(1,2,3,4)]
```

```
sage: [a.transport(perm) for perm in a.automorphism_group()]         #␣
↪needs sage.groups
[(1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)]
```

```
>>> from sage.all import *
>>> [a.transport(perm) for perm in a.automorphism_group()]          #␣
↪needs sage.groups
[(1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)]
```

canonical_label()

EXAMPLES:

```
sage: P = species.CycleSpecies()
sage: P.structures(["a","b","c"]).random_element().canonical_label()
('a', 'b', 'c')
```

```
>>> from sage.all import *
>>> P = species.CycleSpecies()
>>> P.structures(["a","b","c"]).random_element().canonical_label()
('a', 'b', 'c')
```

permutation_group_element()

Return this cycle as a permutation group element.

EXAMPLES:

```
sage: F = species.CycleSpecies()
sage: a = F.structures(["a", "b", "c"])[0]; a
('a', 'b', 'c')
sage: a.permutation_group_element()                                  #␣
↪needs sage.groups
(1,2,3)
```

```
>>> from sage.all import *
>>> F = species.CycleSpecies()
>>> a = F.structures(["a", "b", "c"])[Integer(0)]; a
('a', 'b', 'c')
>>> a.permutation_group_element()                                   #␣
↪needs sage.groups
(1,2,3)
```

transport(*perm*)

Return the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: F = species.CycleSpecies()
sage: a = F.structures(["a", "b", "c"])[0]; a
('a', 'b', 'c')
sage: p = PermutationGroupElement((1,2))                                    #␣
↪needs sage.groups
sage: a.transport(p)                                                        #␣
↪needs sage.groups
('a', 'c', 'b')
```

```
>>> from sage.all import *
>>> F = species.CycleSpecies()
>>> a = F.structures(["a", "b", "c"])[Integer(0)]; a
('a', 'b', 'c')
>>> p = PermutationGroupElement((Integer(1),Integer(2)))                    ␣
↪              # needs sage.groups
>>> a.transport(p)                                                        #␣
↪needs sage.groups
('a', 'c', 'b')
```

sage.combinat.species.cycle_species.**CycleSpecies_class**

alias of *CycleSpecies*

5.1.319 Empty Species

class sage.combinat.species.empty_species.**EmptySpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the empty species. This species has no structure at all. It is the zero of the semi-ring of species.

EXAMPLES:

```
sage: X = species.EmptySpecies(); X
Empty species
sage: X.structures([]).list()
[]
sage: X.structures([1]).list()
[]
sage: X.structures([1,2]).list()
[]
sage: X.generating_series()[0:4]
[0, 0, 0, 0]
sage: X.isotype_generating_series()[0:4]
[0, 0, 0, 0]
sage: X.cycle_index_series()[0:4]                                          #␣
↪needs sage.modules
[0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> X = species.EmptySpecies(); X
Empty species
>>> X.structures([]).list()
```

(continues on next page)

(continued from previous page)

```
[]
>>> X.structures([Integer(1)]).list()
[]
>>> X.structures([Integer(1),Integer(2)]).list()
[]
>>> X.generating_series()[Integer(0):Integer(4)]
[0, 0, 0, 0]
>>> X.isotype_generating_series()[Integer(0):Integer(4)]
[0, 0, 0, 0]
>>> X.cycle_index_series()[Integer(0):Integer(4)]                          ␣
↪              # needs sage.modules
[0, 0, 0, 0]
```

The empty species is the zero of the semi-ring of species. The following tests that it is neutral with respect to addition:

```
sage: Empt  = species.EmptySpecies()
sage: S = species.CharacteristicSpecies(2)
sage: X = S + Empt
sage: X == S     # TODO: Not Implemented
True
sage: (X.generating_series()[0:4] ==
....:  S.generating_series()[0:4])
True
sage: (X.isotype_generating_series()[0:4] ==
....:  S.isotype_generating_series()[0:4])
True
sage: (X.cycle_index_series()[0:4] ==                                      #␣
↪needs sage.modules
....:  S.cycle_index_series()[0:4])
True
```

```
>>> from sage.all import *
>>> Empt  = species.EmptySpecies()
>>> S = species.CharacteristicSpecies(Integer(2))
>>> X = S + Empt
>>> X == S     # TODO: Not Implemented
True
>>> (X.generating_series()[Integer(0):Integer(4)] ==
...  S.generating_series()[Integer(0):Integer(4)])
True
>>> (X.isotype_generating_series()[Integer(0):Integer(4)] ==
...  S.isotype_generating_series()[Integer(0):Integer(4)])
True
>>> (X.cycle_index_series()[Integer(0):Integer(4)] ==                      ␣
↪              # needs sage.modules
...  S.cycle_index_series()[Integer(0):Integer(4)])
True
```

The following tests that it is the zero element with respect to multiplication:

```
sage: Y = Empt*S
```

(continues on next page)

(continued from previous page)

```
sage: Y == Empt    # TODO: Not Implemented
True
sage: Y.generating_series()[0:4]
[0, 0, 0, 0]
sage: Y.isotype_generating_series()[0:4]
[0, 0, 0, 0]
sage: Y.cycle_index_series()[0:4]                                        #␣
↪needs sage.modules
[0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> Y = Empt*S
>>> Y == Empt    # TODO: Not Implemented
True
>>> Y.generating_series()[Integer(0):Integer(4)]
[0, 0, 0, 0]
>>> Y.isotype_generating_series()[Integer(0):Integer(4)]
[0, 0, 0, 0]
>>> Y.cycle_index_series()[Integer(0):Integer(4)]                        ␣
↪               # needs sage.modules
[0, 0, 0, 0]
```

sage.combinat.species.empty_species.**EmptySpecies_class**

alias of *EmptySpecies*

5.1.320 Functorial composition species

class sage.combinat.species.functorial_composition_species.**FunctorialCompositionSpecies**(*F*, *G*, *min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*

Return the functorial composition of two species.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: E2 = species.SetSpecies(size=2)
sage: WP = species.SubsetSpecies()
sage: P2 = E2*E
sage: G = WP.functorial_composition(P2)
sage: G.isotype_generating_series()[0:5]                             #␣
↪needs sage.modules
[1, 1, 2, 4, 11]

sage: G = species.SimpleGraphSpecies()
sage: c = G.generating_series()[0:2]
sage: type(G)
<class 'sage.combinat.species.functorial_composition_species.
↪FunctorialCompositionSpecies'>
```

(continues on next page)

(continued from previous page)

```
sage: G == loads(dumps(G))
True
sage: G._check()   # False due to isomorphism types not being implemented    #␣
↪needs sage.modules
False
```

```
>>> from sage.all import *
>>> E = species.SetSpecies()
>>> E2 = species.SetSpecies(size=Integer(2))
>>> WP = species.SubsetSpecies()
>>> P2 = E2*E
>>> G = WP.functorial_composition(P2)
>>> G.isotype_generating_series()[Integer(0):Integer(5)]                     ␣
↪          # needs sage.modules
[1, 1, 2, 4, 11]

>>> G = species.SimpleGraphSpecies()
>>> c = G.generating_series()[Integer(0):Integer(2)]
>>> type(G)
<class 'sage.combinat.species.functorial_composition_species.
↪FunctorialCompositionSpecies'>
>>> G == loads(dumps(G))
True
>>> G._check()   # False due to isomorphism types not being implemented    # needs␣
↪sage.modules
False
```

weight_ring()

Return the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: G = species.SimpleGraphSpecies()
sage: G.weight_ring()
Rational Field
```

```
>>> from sage.all import *
>>> G = species.SimpleGraphSpecies()
>>> G.weight_ring()
Rational Field
```

sage.combinat.species.functorial_composition_species.**FunctorialCompositionSpecies_class**

alias of *FunctorialCompositionSpecies*

class sage.combinat.species.functorial_composition_species.**FunctorialCompositionStructure**(*parent*, *labels*, *list*)

Bases: *GenericSpeciesStructure*

5.1.321 Generating Series

This file makes a number of extensions to lazy power series by endowing them with some semantic content for how they're to be interpreted.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse10.html. One notable difference is that we use power-sum symmetric functions as the coefficients of our cycle index series.

class sage.combinat.species.generating_series.**CycleIndexSeries**(*parent*, *coeff_stream*)

Bases: LazySymmetricFunction

coefficient_cycle_type(*t*)

Return the coefficient of a cycle type `t` in `self`.

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SymmetricFunctions(QQ).power()
sage: CIS = CycleIndexSeriesRing(QQ)
sage: f = CIS([0, p([1]), 2*p([1,1]),3*p([2,1])])
sage: f.coefficient_cycle_type([1])
1
sage: f.coefficient_cycle_type([1,1])
2
sage: f.coefficient_cycle_type([2,1])
3
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.combinat.species.generating_series import CycleIndexSeriesRing
>>> p = SymmetricFunctions(QQ).power()
>>> CIS = CycleIndexSeriesRing(QQ)
>>> f = CIS([Integer(0), p([Integer(1)]), Integer(2)*p([Integer(1),
↪Integer(1)]),Integer(3)*p([Integer(2),Integer(1)])])
>>> f.coefficient_cycle_type([Integer(1)])
1
>>> f.coefficient_cycle_type([Integer(1),Integer(1)])
2
>>> f.coefficient_cycle_type([Integer(2),Integer(1)])
3
```

count(*t*)

Return the number of structures corresponding to a certain cycle type `t`.

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SymmetricFunctions(QQ).power()
sage: CIS = CycleIndexSeriesRing(QQ)
sage: f = CIS([0, p([1]), 2*p([1,1]), 3*p([2,1])])
sage: f.count([1])
1
```

(continues on next page)

(continued from previous page)

```
sage: f.count([1,1])
4
sage: f.count([2,1])
6
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.combinat.species.generating_series import CycleIndexSeriesRing
>>> p = SymmetricFunctions(QQ).power()
>>> CIS = CycleIndexSeriesRing(QQ)
>>> f = CIS([Integer(0), p([Integer(1)]), Integer(2)*p([Integer(1),
↪Integer(1)]), Integer(3)*p([Integer(2),Integer(1)])])
>>> f.count([Integer(1)])
1
>>> f.count([Integer(1),Integer(1)])
4
>>> f.count([Integer(2),Integer(1)])
6
```

derivative(*n=1*)

Return the species-theoretic n-th derivative of `self`.

For a cycle index series $F(p_1, p_2, p_3, \ldots)$, its derivative is the cycle index series $F' = D_{p_1} F$ (that is, the formal derivative of F with respect to the variable p_1).

If F is the cycle index series of a species S then F' is the cycle index series of an associated species S' of S-structures with a "hole".

EXAMPLES:

The species E of sets satisfies the relationship $E' = E$:

```
sage: E = species.SetSpecies().cycle_index_series()                          #␣
↪needs sage.modules
sage: E[:8] == E.derivative()[:8]                                            #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> E = species.SetSpecies().cycle_index_series()                          #␣
↪needs sage.modules
>>> E[:Integer(8)] == E.derivative()[:Integer(8)]                            ␣
↪               # needs sage.modules
True
```

The species C of cyclic orderings and the species L of linear orderings satisfy the relationship $C' = L$:

```
sage: C = species.CycleSpecies().cycle_index_series()                        #␣
↪needs sage.modules
sage: L = species.LinearOrderSpecies().cycle_index_series()                  #␣
↪needs sage.modules
sage: L[:8] == C.derivative()[:8]                                            #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> C = species.CycleSpecies().cycle_index_series()                    #␣
↪needs sage.modules
>>> L = species.LinearOrderSpecies().cycle_index_series()              #␣
↪needs sage.modules
>>> L[:Integer(8)] == C.derivative()[:Integer(8)]                      ␣
↪                # needs sage.modules
True
```

exponential()

Return the species-theoretic exponential of `self`.

For a cycle index Z_F of a species F, its exponential is the cycle index series $Z_E \circ Z_F$, where Z_E is the *ExponentialCycleIndexSeries()*.

The exponential $Z_E \circ Z_F$ is then the cycle index series of the species $E \circ F$ of "sets of F-structures".

EXAMPLES:

Let BT be the species of binary trees, BF the species of binary forests, and E the species of sets. Then we have $BF = E \circ BT$:

```
sage: BT = species.BinaryTreeSpecies().cycle_index_series()              #␣
↪needs sage.modules
sage: BF = species.BinaryForestSpecies().cycle_index_series()            #␣
↪needs sage.modules
sage: BT.exponential().isotype_generating_series()[:8] == BF.isotype_
↪generating_series()[:8]                # needs sage.modules
True
```

```
>>> from sage.all import *
>>> BT = species.BinaryTreeSpecies().cycle_index_series()              #␣
↪needs sage.modules
>>> BF = species.BinaryForestSpecies().cycle_index_series()            #␣
↪needs sage.modules
>>> BT.exponential().isotype_generating_series()[:Integer(8)] == BF.isotype_
↪generating_series()[:Integer(8)]                # needs sage.modules
True
```

generating_series()

Return the generating series of `self`.

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: cis = P.cycle_index_series()                                       #␣
↪needs sage.modules
sage: f = cis.generating_series()                                        #␣
↪needs sage.modules
sage: f[:5]                                                              #␣
↪needs sage.modules
[1, 1, 1, 5/6, 5/8]
```

```
>>> from sage.all import *
>>> P = species.PartitionSpecies()
```

(continues on next page)

(continued from previous page)

```
>>> cis = P.cycle_index_series()                                    #␣
↪needs sage.modules
>>> f = cis.generating_series()                                     #␣
↪needs sage.modules
>>> f[:Integer(5)]                                                    ␣
↪      # needs sage.modules
[1, 1, 1, 5/6, 5/8]
```

isotype_generating_series()

Return the isotype generating series of `self`.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: cis = P.cycle_index_series()                                    #␣
↪needs sage.modules
sage: f = cis.isotype_generating_series()                             #␣
↪needs sage.modules
sage: f[:10]                                                          #␣
↪needs sage.modules
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> cis = P.cycle_index_series()                                    #␣
↪needs sage.modules
>>> f = cis.isotype_generating_series()                             #␣
↪needs sage.modules
>>> f[:Integer(10)]                                                   ␣
↪      # needs sage.modules
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

logarithm()

Return the combinatorial logarithm of `self`.

For a cycle index Z_F of a species F, its logarithm is the cycle index series $Z_\Omega \circ Z_F$, where Z_Ω is the *LogarithmCycleIndexSeries()*.

The logarithm $Z_\Omega \circ Z_F$ is then the cycle index series of the (virtual) species $\Omega \circ F$ of "connected F-structures". In particular, if $F = E^+ \circ G$ for E^+ the species of nonempty sets and G some other species, then $\Omega \circ F = G$.

EXAMPLES:

Let G be the species of nonempty graphs and CG be the species of nonempty connected graphs. Then $G = E^+ \circ CG$, so $CG = \Omega \circ G$:

```
sage: G = species.SimpleGraphSpecies().cycle_index_series() - 1       #␣
↪needs sage.modules
sage: from sage.combinat.species.generating_series import␣
↪LogarithmCycleIndexSeries
sage: CG = LogarithmCycleIndexSeries()(G)                             #␣
↪needs sage.modules
sage: CG.isotype_generating_series()[0:8]                             #␣
↪needs sage.modules
[0, 1, 1, 2, 6, 21, 112, 853]
```

```
>>> from sage.all import *
>>> G = species.SimpleGraphSpecies().cycle_index_series() - Integer(1)      ␣
↪     # needs sage.modules
>>> from sage.combinat.species.generating_series import␣
↪LogarithmCycleIndexSeries
>>> CG = LogarithmCycleIndexSeries()(G)                                    #␣
↪needs sage.modules
>>> CG.isotype_generating_series()[Integer(0):Integer(8)]                  ␣
↪              # needs sage.modules
[0, 1, 1, 2, 6, 21, 112, 853]
```

pointing()

Return the species-theoretic pointing of `self`.

For a cycle index F, its pointing is the cycle index series $F^\bullet = p_1 \cdot F'$.

If F is the cycle index series of a species S then $F^\bullet$ is the cycle index series of an associated species $S^\bullet$ of S-structures with a marked "root".

EXAMPLES:

The species $E^\bullet$ of "pointed sets" satisfies $E^\bullet = X \cdot E$:

```
sage: E = species.SetSpecies().cycle_index_series()                          #␣
↪needs sage.modules
sage: X = species.SingletonSpecies().cycle_index_series()                    #␣
↪needs sage.modules
sage: E.pointing()[:8] == (X*E)[:8]                                          #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> E = species.SetSpecies().cycle_index_series()                          #␣
↪needs sage.modules
>>> X = species.SingletonSpecies().cycle_index_series()                    #␣
↪needs sage.modules
>>> E.pointing()[:Integer(8)] == (X*E)[:Integer(8)]                          ␣
↪              # needs sage.modules
True
```

class sage.combinat.species.generating_series.**CycleIndexSeriesRing**(*base_ring*, *sparse=True*)

Bases: `LazySymmetricFunctions`

Return the ring of cycle index series over `R`.

This is the ring of formal power series $\Lambda[x]$, where Λ is the ring of symmetric functions over `R` in the p-basis. Its purpose is to house the cycle index series of species (in a somewhat nonstandard notation tailored to Sage): If F is a species, then the *cycle index series* of F is defined to be the formal power series

$$\sum_{n \geq 0} \frac{1}{n!} \Big(\sum_{\sigma \in S_n} \operatorname{fix} F[\sigma] \prod_{z \text{ is a cycle of } \sigma} p_{\text{length of } z} \Big) x^n \in \Lambda_{\mathbf{Q}}[x],$$

where $\operatorname{fix} F[\sigma]$ denotes the number of fixed points of the permutation $F[\sigma]$ of $F[n]$. We notice that this power series is "equigraded" (meaning that its x^n-coefficient is homogeneous of degree n). A more standard convention in combinatorics would be to use x_i instead of p_i, and drop the x (that is, evaluate the above power series at $x = 1$);

but this would be more difficult to implement in Sage, as it would be an element of a power series ring in infinitely many variables.

Note that it is just a `LazyPowerSeriesRing` (whose base ring is Λ) whose elements have some extra methods.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: R = CycleIndexSeriesRing(QQ); R                                          #␣
↪needs sage.modules
Cycle Index Series Ring over Rational Field
sage: p = SymmetricFunctions(QQ).p()                                           #␣
↪needs sage.modules
sage: R(lambda n: p[n])                                                        #␣
↪needs sage.modules
p[] + p[1] + p[2] + p[3] + p[4] + p[5] + p[6] + O^7
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import CycleIndexSeriesRing
>>> R = CycleIndexSeriesRing(QQ); R                                          #␣
↪needs sage.modules
Cycle Index Series Ring over Rational Field
>>> p = SymmetricFunctions(QQ).p()                                           #␣
↪needs sage.modules
>>> R(lambda n: p[n])                                                        #␣
↪needs sage.modules
p[] + p[1] + p[2] + p[3] + p[4] + p[5] + p[6] + O^7
```

Element

alias of `CycleIndexSeries`

sage.combinat.species.generating_series.**ExponentialCycleIndexSeries**()

Return the cycle index series of the species E of sets.

This cycle index satisfies

$$Z_E = \sum_{n \geq 0} \sum_{\lambda \vdash n} \frac{p_\lambda}{z_\lambda}.$$

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪ExponentialCycleIndexSeries
sage: ExponentialCycleIndexSeries()[:5]                                        #␣
↪needs sage.modules
[p[], p[1], 1/2*p[1, 1] + 1/2*p[2], 1/6*p[1, 1, 1] + 1/2*p[2, 1]
 + 1/3*p[3], 1/24*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] + 1/8*p[2, 2]
 + 1/3*p[3, 1] + 1/4*p[4]]
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪ExponentialCycleIndexSeries
>>> ExponentialCycleIndexSeries()[:Integer(5)]                                 ␣
↪     # needs sage.modules
[p[], p[1], 1/2*p[1, 1] + 1/2*p[2], 1/6*p[1, 1, 1] + 1/2*p[2, 1]
```

(continues on next page)

(continued from previous page)

```
+ 1/3*p[3], 1/24*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] + 1/8*p[2, 2]
+ 1/3*p[3, 1] + 1/4*p[4]]
```

class sage.combinat.species.generating_series.**ExponentialGeneratingSeries**(*parent*, *coeff_stream*)

Bases: LazyPowerSeries

A class for ordinary generating series.

Note that it is just a LazyPowerSeries whose elements have some extra methods.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: f = R(lambda n: n)
sage: f
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
>>> R = OrdinaryGeneratingSeriesRing(QQ)
>>> f = R(lambda n: n)
>>> f
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
```

count(*n*)

Return the number of structures of size n.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ)
sage: f = R(lambda n: 1)
sage: [f.count(i) for i in range(7)]
[1, 1, 2, 6, 24, 120, 720]
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪ExponentialGeneratingSeriesRing
>>> R = ExponentialGeneratingSeriesRing(QQ)
>>> f = R(lambda n: Integer(1))
>>> [f.count(i) for i in range(Integer(7))]
[1, 1, 2, 6, 24, 120, 720]
```

counts(*n*)

Return the number of structures on a set for size i for each i in range(n).

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.counts(5)
[0, 1, 4, 18, 96]
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪ExponentialGeneratingSeriesRing
>>> R = ExponentialGeneratingSeriesRing(QQ)
>>> f = R(range(Integer(20)))
>>> f.counts(Integer(5))
[0, 1, 4, 18, 96]
```

functorial_composition(*y*)

Return the exponential generating series which is the functorial composition of `self` with `y`.

If $f = \sum_{n=0}^{\infty} f_n \frac{x^n}{n!}$ and $g = \sum_{n=0}^{\infty} g_n \frac{x^n}{n!}$, then functorial composition $f \Box g$ is defined as

$$f \Box g = \sum_{n=0}^{\infty} f_{g_n} \frac{x^n}{n!}.$$

REFERENCES:

- Section 2.2 of [BLL1998].

EXAMPLES:

```
sage: G = species.SimpleGraphSpecies()
sage: g = G.generating_series()
sage: [g.coefficient(i) for i in range(10)]
[1, 1, 1, 4/3, 8/3, 128/15, 2048/45, 131072/315, 2097152/315, 536870912/2835]

sage: E = species.SetSpecies()
sage: E2 = E.restricted(min=2, max=3)
sage: WP = species.SubsetSpecies()
sage: P2 = E2*E
sage: g1 = WP.generating_series()
sage: g2 = P2.generating_series()
sage: g1.functorial_composition(g2)[:10]
[1, 1, 1, 4/3, 8/3, 128/15, 2048/45, 131072/315, 2097152/315, 536870912/2835]
```

```
>>> from sage.all import *
>>> G = species.SimpleGraphSpecies()
>>> g = G.generating_series()
>>> [g.coefficient(i) for i in range(Integer(10))]
[1, 1, 1, 4/3, 8/3, 128/15, 2048/45, 131072/315, 2097152/315, 536870912/2835]

>>> E = species.SetSpecies()
>>> E2 = E.restricted(min=Integer(2), max=Integer(3))
>>> WP = species.SubsetSpecies()
>>> P2 = E2*E
>>> g1 = WP.generating_series()
```

(continues on next page)

(continued from previous page)

```
>>> g2 = P2.generating_series()
>>> g1.functorial_composition(g2)[:Integer(10)]
[1, 1, 1, 4/3, 8/3, 128/15, 2048/45, 131072/315, 2097152/315, 536870912/2835]
```

class sage.combinat.species.generating_series.**ExponentialGeneratingSeriesRing**(*base_ring*)

Bases: LazyPowerSeriesRing

Return the ring of exponential generating series over R.

Note that it is just a LazyPowerSeriesRing whose elements have some extra methods.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ); R
Lazy Taylor Series Ring in z over Rational Field
sage: [R(lambda n: 1).coefficient(i) for i in range(4)]
[1, 1, 1, 1]
sage: R(lambda n: 1).counts(4)
[1, 1, 2, 6]
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪ExponentialGeneratingSeriesRing
>>> R = ExponentialGeneratingSeriesRing(QQ); R
Lazy Taylor Series Ring in z over Rational Field
>>> [R(lambda n: Integer(1)).coefficient(i) for i in range(Integer(4))]
[1, 1, 1, 1]
>>> R(lambda n: Integer(1)).counts(Integer(4))
[1, 1, 2, 6]
```

Element

alias of *ExponentialGeneratingSeries*

sage.combinat.species.generating_series.**LogarithmCycleIndexSeries**()

Return the cycle index series of the virtual species Ω, the compositional inverse of the species E^+ of nonempty sets.

The notion of virtual species is treated thoroughly in [BLL1998]. The specific algorithm used here to compute the cycle index of Ω is found in [Labelle2008].

EXAMPLES:

The virtual species Ω is 'properly virtual', in the sense that its cycle index has negative coefficients:

```
sage: from sage.combinat.species.generating_series import␣
↪LogarithmCycleIndexSeries
sage: LogarithmCycleIndexSeries()[0:4]                                     #␣
↪needs sage.modules
[0, p[1], -1/2*p[1, 1] - 1/2*p[2], 1/3*p[1, 1, 1] - 1/3*p[3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import LogarithmCycleIndexSeries
>>> LogarithmCycleIndexSeries()[Integer(0):Integer(4)]                    ␣
```

(continues on next page)

(continued from previous page)

```
↪               # needs sage.modules
[0, p[1], -1/2*p[1, 1] - 1/2*p[2], 1/3*p[1, 1, 1] - 1/3*p[3]]
```

Its defining property is that $\Omega \circ E^{+} = E^{+} \circ \Omega = X$ (that is, that composition with E^{+} in both directions yields the multiplicative identity X):

```
sage: Eplus = sage.combinat.species.set_species.SetSpecies(min=1).cycle_index_
↪series()       # needs sage.modules
sage: LogarithmCycleIndexSeries()(Eplus)[0:4]                              #␣
↪needs sage.modules
[0, p[1], 0, 0]
```

```
>>> from sage.all import *
>>> Eplus = sage.combinat.species.set_species.SetSpecies(min=Integer(1)).cycle_
↪index_series()       # needs sage.modules
>>> LogarithmCycleIndexSeries()(Eplus)[Integer(0):Integer(4)]              ␣
↪               # needs sage.modules
[0, p[1], 0, 0]
```

class sage.combinat.species.generating_series.**OrdinaryGeneratingSeries**(*parent*, *coeff_stream*)

Bases: LazyPowerSeries

A class for ordinary generating series.

Note that it is just a LazyPowerSeries whose elements have some extra methods.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: f = R(lambda n: n)
sage: f
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
>>> R = OrdinaryGeneratingSeriesRing(QQ)
>>> f = R(lambda n: n)
>>> f
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
```

count(*n*)

Return the number of structures on a set of size n.

INPUT:

- n – the size of the set

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
```

(continues on next page)

(continued from previous page)

```
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.count(10)
10
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
>>> R = OrdinaryGeneratingSeriesRing(QQ)
>>> f = R(range(Integer(20)))
>>> f.count(Integer(10))
10
```

counts(*n*)

Return the number of structures on a set for size `i` for each `i` in `range(n)`.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.counts(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
>>> R = OrdinaryGeneratingSeriesRing(QQ)
>>> f = R(range(Integer(20)))
>>> f.counts(Integer(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

class `sage.combinat.species.generating_series.`**OrdinaryGeneratingSeriesRing**(*base_ring*)

Bases: `LazyPowerSeriesRing`

Return the ring of ordinary generating series over `R`.

Note that it is just a `LazyPowerSeriesRing` whose elements have some extra methods.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ); R
Lazy Taylor Series Ring in z over Rational Field
sage: [R(lambda n: 1).coefficient(i) for i in range(4)]
[1, 1, 1, 1]
sage: R(lambda n: 1).counts(4)
[1, 1, 1, 1]
sage: R == loads(dumps(R))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.species.generating_series import␣
↪OrdinaryGeneratingSeriesRing
>>> R = OrdinaryGeneratingSeriesRing(QQ); R
Lazy Taylor Series Ring in z over Rational Field
>>> [R(lambda n: Integer(1)).coefficient(i) for i in range(Integer(4))]
[1, 1, 1, 1]
>>> R(lambda n: Integer(1)).counts(Integer(4))
[1, 1, 1, 1]
>>> R == loads(dumps(R))
True
```

Element

alias of *OrdinaryGeneratingSeries*

5.1.322 Examples of Combinatorial Species

sage.combinat.species.library.**BinaryForestSpecies**()

Return the species of binary forests.

Binary forests are defined as sets of binary trees.

EXAMPLES:

```
sage: F = species.BinaryForestSpecies()
sage: F.generating_series().counts(10)
[1, 1, 3, 19, 193, 2721, 49171, 1084483, 28245729, 848456353]
sage: F.isotype_generating_series().counts(10)                              #␣
↪needs sage.modules
[1, 1, 2, 4, 10, 26, 77, 235, 758, 2504]
sage: F.cycle_index_series()[:7]                                            #␣
↪needs sage.modules
[p[],
 p[1],
 3/2*p[1, 1] + 1/2*p[2],
 19/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3],
 193/24*p[1, 1, 1, 1] + 3/4*p[2, 1, 1] + 5/8*p[2, 2] + 1/3*p[3, 1] + 1/4*p[4],
 907/40*p[1, 1, 1, 1, 1] + 19/12*p[2, 1, 1, 1] + 5/8*p[2, 2, 1] + 1/2*p[3, 1, 1]␣
↪+ 1/6*p[3, 2] + 1/4*p[4, 1] + 1/5*p[5],
 49171/720*p[1, 1, 1, 1, 1, 1] + 193/48*p[2, 1, 1, 1, 1] + 15/16*p[2, 2, 1, 1] +␣
↪61/48*p[2, 2, 2] + 19/18*p[3, 1, 1, 1] + 1/6*p[3, 2, 1] + 7/18*p[3, 3] + 3/
↪8*p[4, 1, 1] + 1/8*p[4, 2] + 1/5*p[5, 1] + 1/6*p[6]]
```

```
>>> from sage.all import *
>>> F = species.BinaryForestSpecies()
>>> F.generating_series().counts(Integer(10))
[1, 1, 3, 19, 193, 2721, 49171, 1084483, 28245729, 848456353]
>>> F.isotype_generating_series().counts(Integer(10))                           ␣
↪     # needs sage.modules
[1, 1, 2, 4, 10, 26, 77, 235, 758, 2504]
>>> F.cycle_index_series()[:Integer(7)]                                         ␣
↪     # needs sage.modules
[p[],
```

(continues on next page)

(continued from previous page)

```
p[1],
3/2*p[1, 1] + 1/2*p[2],
19/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3],
193/24*p[1, 1, 1, 1] + 3/4*p[2, 1, 1] + 5/8*p[2, 2] + 1/3*p[3, 1] + 1/4*p[4],
907/40*p[1, 1, 1, 1, 1] + 19/12*p[2, 1, 1, 1] + 5/8*p[2, 2, 1] + 1/2*p[3, 1, 1]␣
↪+ 1/6*p[3, 2] + 1/4*p[4, 1] + 1/5*p[5],
49171/720*p[1, 1, 1, 1, 1, 1] + 193/48*p[2, 1, 1, 1, 1] + 15/16*p[2, 2, 1, 1] +␣
↪61/48*p[2, 2, 2] + 19/18*p[3, 1, 1, 1] + 1/6*p[3, 2, 1] + 7/18*p[3, 3] + 3/
↪8*p[4, 1, 1] + 1/8*p[4, 2] + 1/5*p[5, 1] + 1/6*p[6]]
```

sage.combinat.species.library.**BinaryTreeSpecies**()

Return the species of binary trees on n leaves.

The species of binary trees B is defined by $B = X + B \cdot B$, where X is the singleton species.

EXAMPLES:

```
sage: B = species.BinaryTreeSpecies()
sage: B.generating_series().counts(10)
[0, 1, 2, 12, 120, 1680, 30240, 665280, 17297280, 518918400]
sage: B.isotype_generating_series().counts(10)
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430]
sage: B._check()
True
```

```
>>> from sage.all import *
>>> B = species.BinaryTreeSpecies()
>>> B.generating_series().counts(Integer(10))
[0, 1, 2, 12, 120, 1680, 30240, 665280, 17297280, 518918400]
>>> B.isotype_generating_series().counts(Integer(10))
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430]
>>> B._check()
True
```

```
sage: B = species.BinaryTreeSpecies()
sage: a = B.structures([1,2,3,4,5])[187]; a
2*((5*3)*(4*1))
sage: a.automorphism_group()                                                    #␣
↪needs sage.groups
Permutation Group with generators [()]
```

```
>>> from sage.all import *
>>> B = species.BinaryTreeSpecies()
>>> a = B.structures([Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)])[Integer(187)]; a
2*((5*3)*(4*1))
>>> a.automorphism_group()                                                    #␣
↪needs sage.groups
Permutation Group with generators [()]
```

sage.combinat.species.library.**SimpleGraphSpecies**()

Return the species of simple graphs.

EXAMPLES:

```
sage: S = species.SimpleGraphSpecies()
sage: S.generating_series().counts(10)
[1, 1, 2, 8, 64, 1024, 32768, 2097152, 268435456, 68719476736]
sage: S.cycle_index_series()[:5]                                    #␣
↪needs sage.modules
[p[],
 p[1],
 p[1, 1] + p[2],
 4/3*p[1, 1, 1] + 2*p[2, 1] + 2/3*p[3],
 8/3*p[1, 1, 1, 1] + 4*p[2, 1, 1] + 2*p[2, 2] + 4/3*p[3, 1] + p[4]]
sage: S.isotype_generating_series()[:6]                              #␣
↪needs sage.modules
[1, 1, 2, 4, 11, 34]
```

```
>>> from sage.all import *
>>> S = species.SimpleGraphSpecies()
>>> S.generating_series().counts(Integer(10))
[1, 1, 2, 8, 64, 1024, 32768, 2097152, 268435456, 68719476736]
>>> S.cycle_index_series()[:Integer(5)]                               ␣
↪    # needs sage.modules
[p[],
 p[1],
 p[1, 1] + p[2],
 4/3*p[1, 1, 1] + 2*p[2, 1] + 2/3*p[3],
 8/3*p[1, 1, 1, 1] + 4*p[2, 1, 1] + 2*p[2, 2] + 4/3*p[3, 1] + p[4]]
>>> S.isotype_generating_series()[:Integer(6)]                        ␣
↪    # needs sage.modules
[1, 1, 2, 4, 11, 34]
```

5.1.323 Linear-order Species

class sage.combinat.species.linear_order_species.**LinearOrderSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the species of linear orders.

EXAMPLES:

```
sage: L = species.LinearOrderSpecies()
sage: L.generating_series()[0:5]
[1, 1, 1, 1, 1]

sage: L = species.LinearOrderSpecies()
sage: L._check()
True
sage: L == loads(dumps(L))
True
```

```
>>> from sage.all import *
>>> L = species.LinearOrderSpecies()
```

(continues on next page)

(continued from previous page)

```
>>> L.generating_series()[Integer(0):Integer(5)]
[1, 1, 1, 1, 1]

>>> L = species.LinearOrderSpecies()
>>> L._check()
True
>>> L == loads(dumps(L))
True
```

class sage.combinat.species.linear_order_species.**LinearOrderSpeciesStructure**(*parent*, *labels*, *list*)

Bases: *GenericSpeciesStructure*

automorphism_group()

Return the group of permutations whose action on this structure leave it fixed. For the species of linear orders, there is no non-trivial automorphism.

EXAMPLES:

```
sage: F = species.LinearOrderSpecies()
sage: a = F.structures(["a", "b", "c"])[0]; a
['a', 'b', 'c']
sage: a.automorphism_group()                                          #␣
↪needs sage.groups
Symmetric group of order 1! as a permutation group
```

```
>>> from sage.all import *
>>> F = species.LinearOrderSpecies()
>>> a = F.structures(["a", "b", "c"])[Integer(0)]; a
['a', 'b', 'c']
>>> a.automorphism_group()                                            #␣
↪needs sage.groups
Symmetric group of order 1! as a permutation group
```

canonical_label()

EXAMPLES:

```
sage: P = species.LinearOrderSpecies()
sage: s = P.structures(["a", "b", "c"]).random_element()
sage: s.canonical_label()
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> P = species.LinearOrderSpecies()
>>> s = P.structures(["a", "b", "c"]).random_element()
>>> s.canonical_label()
['a', 'b', 'c']
```

transport(*perm*)

Return the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: F = species.LinearOrderSpecies()
sage: a = F.structures(["a", "b", "c"])[0]; a
['a', 'b', 'c']
sage: p = PermutationGroupElement((1,2))                                      #␣
↪needs sage.groups
sage: a.transport(p)                                                          #␣
↪needs sage.groups
['b', 'a', 'c']
```

```
>>> from sage.all import *
>>> F = species.LinearOrderSpecies()
>>> a = F.structures(["a", "b", "c"])[Integer(0)]; a
['a', 'b', 'c']
>>> p = PermutationGroupElement((Integer(1),Integer(2)))                       ␣
↪             # needs sage.groups
>>> a.transport(p)                                                           #␣
↪needs sage.groups
['b', 'a', 'c']
```

sage.combinat.species.linear_order_species.**LinearOrderSpecies_class**

alias of *LinearOrderSpecies*

5.1.324 Miscellaneous Functions

sage.combinat.species.misc.**accept_size**(*f*)

The purpose of this decorator is to change calls like species.SetSpecies(size=1) to species.SetSpecies(min=1, max=2). This is to make caching species easier and to restrict the number of parameters that the lower level code needs to know about.

EXAMPLES:

```
sage: from sage.combinat.species.misc import accept_size
sage: def f(*args, **kwds):
....:     print("{} {}".format(args, sorted(kwds.items())))
sage: f = accept_size(f)
sage: f(min=1)
() [('min', 1)]
sage: f(size=2)
() [('max', 3), ('min', 2)]
```

```
>>> from sage.all import *
>>> from sage.combinat.species.misc import accept_size
>>> def f(*args, **kwds):
...     print("{} {}".format(args, sorted(kwds.items())))
>>> f = accept_size(f)
>>> f(min=Integer(1))
() [('min', 1)]
>>> f(size=Integer(2))
() [('max', 3), ('min', 2)]
```

sage.combinat.species.misc.**change_support**(*perm*, *support*, *change_perm=None*)

Changes the support of a permutation defined on [1, …, n] to support.

EXAMPLES:

```
sage: from sage.combinat.species.misc import change_support
sage: p = PermutationGroupElement((1,2,3)); p
(1,2,3)
sage: change_support(p, [3,4,5])
(3,4,5)
```

```
>>> from sage.all import *
>>> from sage.combinat.species.misc import change_support
>>> p = PermutationGroupElement((Integer(1),Integer(2),Integer(3))); p
(1,2,3)
>>> change_support(p, [Integer(3),Integer(4),Integer(5)])
(3,4,5)
```

5.1.325 Partition Species

class sage.combinat.species.partition_species.**PartitionSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*

Return the species of partitions.

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: P.generating_series()[0:5]
[1, 1, 1, 5/6, 5/8]
sage: P.isotype_generating_series()[0:5]
[1, 1, 2, 3, 5]

sage: P = species.PartitionSpecies()
sage: P._check()
True
sage: P == loads(dumps(P))
True
```

```
>>> from sage.all import *
>>> P = species.PartitionSpecies()
>>> P.generating_series()[Integer(0):Integer(5)]
[1, 1, 1, 5/6, 5/8]
>>> P.isotype_generating_series()[Integer(0):Integer(5)]
[1, 1, 2, 3, 5]

>>> P = species.PartitionSpecies()
>>> P._check()
True
>>> P == loads(dumps(P))
True
```

class sage.combinat.species.partition_species.**PartitionSpeciesStructure**(*parent*, *labels*, *list*)

Bases: *GenericSpeciesStructure*

EXAMPLES:

```
sage: from sage.combinat.species.partition_species import␣
↪PartitionSpeciesStructure
sage: P = species.PartitionSpecies()
sage: s = PartitionSpeciesStructure(P, ['a','b','c'], [[1,2],[3]]); s
{{'a', 'b'}, {'c'}}
sage: s == loads(dumps(s))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.species.partition_species import PartitionSpeciesStructure
>>> P = species.PartitionSpecies()
>>> s = PartitionSpeciesStructure(P, ['a','b','c'], [[Integer(1),Integer(2)],
↪[Integer(3)]]); s
{{'a', 'b'}, {'c'}}
>>> s == loads(dumps(s))
True
```

automorphism_group()

Return the group of permutations whose action on this set partition leave it fixed.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import␣
↪PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
{{2, 3}, {4}}
sage: a.automorphism_group()
Permutation Group with generators [(1,2)]
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement((Integer(2),Integer(3)))
>>> from sage.combinat.species.partition_species import␣
↪PartitionSpeciesStructure
>>> a = PartitionSpeciesStructure(None, [Integer(2),Integer(3),Integer(4)],␣
↪[[Integer(1),Integer(2)],[Integer(3)]]); a
{{2, 3}, {4}}
>>> a.automorphism_group()
Permutation Group with generators [(1,2)]
```

canonical_label()

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[{{'a', 'b', 'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a'}, {'b'}, {'c'}}]
```

```
>>> from sage.all import *
>>> P = species.PartitionSpecies()
>>> S = P.structures(["a", "b", "c"])
>>> [s.canonical_label() for s in S]
[{{'a', 'b', 'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a'}, {'b'}, {'c'}}]
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- `labels` – list of labels

OUTPUT:

A structure with the i-th label of `self` replaced with the i-th label of the list.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import␣
↪PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
{{2, 3}, {4}}
sage: a.change_labels([1,2,3])
{{1, 2}, {3}}
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement((Integer(2),Integer(3)))
>>> from sage.combinat.species.partition_species import␣
↪PartitionSpeciesStructure
>>> a = PartitionSpeciesStructure(None, [Integer(2),Integer(3),Integer(4)],␣
↪[[Integer(1),Integer(2)],[Integer(3)]]); a
{{2, 3}, {4}}
>>> a.change_labels([Integer(1),Integer(2),Integer(3)])
{{1, 2}, {3}}
```

transport(*perm*)

Return the transport of this set partition along the permutation perm. For set partitions, this is the direct product of the automorphism groups for each of the blocks.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import␣
↪PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
{{2, 3}, {4}}
sage: a.transport(p)
{{2, 4}, {3}}
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement((Integer(2),Integer(3)))
>>> from sage.combinat.species.partition_species import␣
↪PartitionSpeciesStructure
>>> a = PartitionSpeciesStructure(None, [Integer(2),Integer(3),Integer(4)],␣
↪[[Integer(1),Integer(2)],[Integer(3)]]); a
{{2, 3}, {4}}
>>> a.transport(p)
{{2, 4}, {3}}
```

sage.combinat.species.partition_species.**PartitionSpecies_class**

alias of *PartitionSpecies*

5.1.326 Permutation species

class sage.combinat.species.permutation_species.**PermutationSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the species of permutations.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: P.generating_series()[0:5]
[1, 1, 1, 1, 1]
sage: P.isotype_generating_series()[0:5]
[1, 1, 2, 3, 5]

sage: P = species.PermutationSpecies()
sage: c = P.generating_series()[0:3]
sage: P._check()
True
sage: P == loads(dumps(P))
True
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> P.generating_series()[Integer(0):Integer(5)]
[1, 1, 1, 1, 1]
>>> P.isotype_generating_series()[Integer(0):Integer(5)]
[1, 1, 2, 3, 5]

>>> P = species.PermutationSpecies()
>>> c = P.generating_series()[Integer(0):Integer(3)]
>>> P._check()
True
>>> P == loads(dumps(P))
True
```

class sage.combinat.species.permutation_species.**PermutationSpeciesStructure**(*parent*, *labels*, *list*)

Bases: *GenericSpeciesStructure*

automorphism_group()

Return the group of permutations whose action on this structure leave it fixed.

EXAMPLES:

```
sage: set_random_seed(0)
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"])[2]; a
['a', 'c', 'b', 'd']
sage: a.automorphism_group()
Permutation Group with generators [(2,3), (1,4)]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> p = PermutationGroupElement((Integer(2),Integer(3),Integer(4)))
>>> P = species.PermutationSpecies()
>>> a = P.structures(["a", "b", "c", "d"])[Integer(2)]; a
['a', 'c', 'b', 'd']
>>> a.automorphism_group()
Permutation Group with generators [(2,3), (1,4)]
```

```
sage: [a.transport(perm) for perm in a.automorphism_group()]
[['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd']]
```

```
>>> from sage.all import *
>>> [a.transport(perm) for perm in a.automorphism_group()]
[['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd']]
```

canonical_label()

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[['a', 'b', 'c'],
 ['b', 'a', 'c'],
 ['b', 'a', 'c'],
 ['b', 'c', 'a'],
 ['b', 'c', 'a'],
 ['b', 'a', 'c']]
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> S = P.structures(["a", "b", "c"])
>>> [s.canonical_label() for s in S]
[['a', 'b', 'c'],
```

(continues on next page)

(continued from previous page)

```
 ['b', 'a', 'c'],
 ['b', 'a', 'c'],
 ['b', 'c', 'a'],
 ['b', 'c', 'a'],
 ['b', 'a', 'c']]
```

permutation_group_element()

Return `self` as a permutation group element.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"])[2]; a
['a', 'c', 'b', 'd']
sage: a.permutation_group_element()
(2,3)
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement((Integer(2),Integer(3),Integer(4)))
>>> P = species.PermutationSpecies()
>>> a = P.structures(["a", "b", "c", "d"])[Integer(2)]; a
['a', 'c', 'b', 'd']
>>> a.permutation_group_element()
(2,3)
```

transport(*perm*)

Return the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"])[2]; a
['a', 'c', 'b', 'd']
sage: a.transport(p)
['a', 'd', 'c', 'b']
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement((Integer(2),Integer(3),Integer(4)))
>>> P = species.PermutationSpecies()
>>> a = P.structures(["a", "b", "c", "d"])[Integer(2)]; a
['a', 'c', 'b', 'd']
>>> a.transport(p)
['a', 'd', 'c', 'b']
```

sage.combinat.species.permutation_species.**PermutationSpecies_class**

alias of *PermutationSpecies*

5.1.327 Product species

class sage.combinat.species.product_species.**ProductSpecies**(*F*, *G*, *min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: A = X*X
sage: A.generating_series()[0:4]
[0, 0, 1, 0]

sage: P = species.PermutationSpecies()
sage: F = P * P; F
Product of (Permutation species) and (Permutation species)
sage: F == loads(dumps(F))
True
sage: F._check()                                                    #␣
↪needs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> X = species.SingletonSpecies()
>>> A = X*X
>>> A.generating_series()[Integer(0):Integer(4)]
[0, 0, 1, 0]

>>> P = species.PermutationSpecies()
>>> F = P * P; F
Product of (Permutation species) and (Permutation species)
>>> F == loads(dumps(F))
True
>>> F._check()                                                 # needs␣
↪sage.libs.flint
True
```

left_factor()

Return the left factor of this product.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: X = species.SingletonSpecies()
sage: F = P*X
sage: F.left_factor()
Permutation species
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> X = species.SingletonSpecies()
>>> F = P*X
>>> F.left_factor()
Permutation species
```

right_factor()

Return the right factor of this product.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: X = species.SingletonSpecies()
sage: F = P*X
sage: F.right_factor()
Singleton species
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> X = species.SingletonSpecies()
>>> F = P*X
>>> F.right_factor()
Singleton species
```

weight_ring()

Return the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: C = S*S
sage: C.weight_ring()
Rational Field
```

```
>>> from sage.all import *
>>> S = species.SetSpecies()
>>> C = S*S
>>> C.weight_ring()
Rational Field
```

```
sage: S = species.SetSpecies(weight=QQ['t'].gen())
sage: C = S*S
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> S = species.SetSpecies(weight=QQ['t'].gen())
>>> C = S*S
>>> C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

```
sage: S = species.SetSpecies()
sage: C = (S*S).weighted(QQ['t'].gen())
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> S = species.SetSpecies()
```

(continues on next page)

(continued from previous page)

```
>>> C = (S*S).weighted(QQ['t'].gen())
>>> C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

class sage.combinat.species.product_species.**ProductSpeciesStructure**(*parent*, *labels*, *subset*, *left*, *right*)

Bases: *GenericSpeciesStructure*

automorphism_group()

EXAMPLES:

```
sage: # needs sage.groups
sage: p = PermutationGroupElement((2,3))
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures([1,2,3,4])[1]; a
{1}*{2, 3, 4}
sage: a.automorphism_group()
Permutation Group with generators [(2,3), (2,3,4)]
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> p = PermutationGroupElement((Integer(2),Integer(3)))
>>> S = species.SetSpecies()
>>> F = S * S
>>> a = F.structures([Integer(1),Integer(2),Integer(3),
↪Integer(4)])[Integer(1)]; a
{1}*{2, 3, 4}
>>> a.automorphism_group()
Permutation Group with generators [(2,3), (2,3,4)]
```

```
sage: [a.transport(g) for g in a.automorphism_group()]                    #␣
↪needs sage.groups
[{1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4}]
```

```
>>> from sage.all import *
>>> [a.transport(g) for g in a.automorphism_group()]                    #␣
↪needs sage.groups
[{1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4}]
```

```
sage: a = F.structures([1,2,3,4])[8]; a                                          #␣
↪needs sage.groups
{2, 3}*{1, 4}
sage: [a.transport(g) for g in a.automorphism_group()]                           #␣
↪needs sage.groups
[{2, 3}*{1, 4}, {2, 3}*{1, 4}, {2, 3}*{1, 4}, {2, 3}*{1, 4}]
```

```
>>> from sage.all import *
>>> a = F.structures([Integer(1),Integer(2),Integer(3),
↪Integer(4)])[Integer(8)]; a                                         # needs␣
↪sage.groups
{2, 3}*{1, 4}
>>> [a.transport(g) for g in a.automorphism_group()]                           #␣
↪needs sage.groups
[{2, 3}*{1, 4}, {2, 3}*{1, 4}, {2, 3}*{1, 4}, {2, 3}*{1, 4}]
```

canonical_label()

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: F = S * S
sage: S = F.structures(['a','b','c']).list(); S
[{}*{'a', 'b', 'c'},
 {'a'}*{'b', 'c'},
 {'b'}*{'a', 'c'},
 {'c'}*{'a', 'b'},
 {'a', 'b'}*{'c'},
 {'a', 'c'}*{'b'},
 {'b', 'c'}*{'a'},
 {'a', 'b', 'c'}*{}]
```

```
>>> from sage.all import *
>>> S = species.SetSpecies()
>>> F = S * S
>>> S = F.structures(['a','b','c']).list(); S
[{}*{'a', 'b', 'c'},
 {'a'}*{'b', 'c'},
 {'b'}*{'a', 'c'},
 {'c'}*{'a', 'b'},
 {'a', 'b'}*{'c'},
 {'a', 'c'}*{'b'},
 {'b', 'c'}*{'a'},
 {'a', 'b', 'c'}*{}]
```

```
sage: F.isotypes(['a','b','c']).cardinality()
4
sage: [s.canonical_label() for s in S]
[{}*{'a', 'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a', 'b'}*{'c'},
```

(continues on next page)

(continued from previous page)

```
 {'a', 'b'}*{'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b', 'c'}*{}]
```

```
>>> from sage.all import *
>>> F.isotypes(['a','b','c']).cardinality()
4
>>> [s.canonical_label() for s in S]
[{}*{'a', 'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b', 'c'}*{}]
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- `labels` – list of labels

OUTPUT:

A structure with the i-th label of `self` replaced with the i-th label of the list.

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures(['a','b','c'])[0]; a
{}*{'a', 'b', 'c'}
sage: a.change_labels([1,2,3])
{}*{1, 2, 3}
```

```
>>> from sage.all import *
>>> S = species.SetSpecies()
>>> F = S * S
>>> a = F.structures(['a','b','c'])[Integer(0)]; a
{}*{'a', 'b', 'c'}
>>> a.change_labels([Integer(1),Integer(2),Integer(3)])
{}*{1, 2, 3}
```

transport(*perm*)

EXAMPLES:

```
sage: # needs sage.groups
sage: p = PermutationGroupElement((2,3))
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures(['a','b','c'])[4]; a
{'a', 'b'}*{'c'}
```

(continues on next page)

(continued from previous page)

```
sage: a.transport(p)
{'a', 'c'}*{'b'}
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> p = PermutationGroupElement((Integer(2),Integer(3)))
>>> S = species.SetSpecies()
>>> F = S * S
>>> a = F.structures(['a','b','c'])[Integer(4)]; a
{'a', 'b'}*{'c'}
>>> a.transport(p)
{'a', 'c'}*{'b'}
```

sage.combinat.species.product_species.**ProductSpecies_class**

alias of *ProductSpecies*

5.1.328 Recursive Species

class sage.combinat.species.recursive_species.**CombinatorialSpecies**(*min=None*)

Bases: *GenericCombinatorialSpecies*

EXAMPLES:

```
sage: F = CombinatorialSpecies()
sage: loads(dumps(F))
Combinatorial species
```

```
>>> from sage.all import *
>>> F = CombinatorialSpecies()
>>> loads(dumps(F))
Combinatorial species
```

```
sage: X = species.SingletonSpecies()
sage: E = species.EmptySetSpecies()
sage: L = CombinatorialSpecies()
sage: L.define(E+X*L)
sage: L.generating_series()[0:4]
[1, 1, 1, 1]
sage: LL = loads(dumps(L))
sage: LL.generating_series()[0:4]
[1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> X = species.SingletonSpecies()
>>> E = species.EmptySetSpecies()
>>> L = CombinatorialSpecies()
>>> L.define(E+X*L)
>>> L.generating_series()[Integer(0):Integer(4)]
[1, 1, 1, 1]
>>> LL = loads(dumps(L))
>>> LL.generating_series()[Integer(0):Integer(4)]
[1, 1, 1, 1]
```

define(*x*)

Define `self` to be equal to the combinatorial species `x`.

This is used to define combinatorial species recursively. All of the real work is done by calling the `.set()` method for each of the series associated to `self`.

EXAMPLES: The species of linear orders L can be recursively defined by $L = 1 + X * L$ where 1 represents the empty set species and X represents the singleton species:

```
sage: X = species.SingletonSpecies()
sage: E = species.EmptySetSpecies()
sage: L = CombinatorialSpecies()
sage: L.define(E+X*L)
sage: L.generating_series()[0:4]
[1, 1, 1, 1]
sage: L.structures([1,2,3]).cardinality()
6
sage: L.structures([1,2,3]).list()
[1*(2*(3*{})),
 1*(3*(2*{})),
 2*(1*(3*{})),
 2*(3*(1*{})),
 3*(1*(2*{})),
 3*(2*(1*{}))]
```

```
>>> from sage.all import *
>>> X = species.SingletonSpecies()
>>> E = species.EmptySetSpecies()
>>> L = CombinatorialSpecies()
>>> L.define(E+X*L)
>>> L.generating_series()[Integer(0):Integer(4)]
[1, 1, 1, 1]
>>> L.structures([Integer(1),Integer(2),Integer(3)]).cardinality()
6
>>> L.structures([Integer(1),Integer(2),Integer(3)]).list()
[1*(2*(3*{})),
 1*(3*(2*{})),
 2*(1*(3*{})),
 2*(3*(1*{})),
 3*(1*(2*{})),
 3*(2*(1*{}))]
```

```
sage: L = species.LinearOrderSpecies()
sage: L.generating_series()[0:4]
[1, 1, 1, 1]
sage: L.structures([1,2,3]).cardinality()
6
sage: L.structures([1,2,3]).list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

```
>>> from sage.all import *
>>> L = species.LinearOrderSpecies()
>>> L.generating_series()[Integer(0):Integer(4)]
```

(continues on next page)

(continued from previous page)

```
[1, 1, 1, 1]
>>> L.structures([Integer(1),Integer(2),Integer(3)]).cardinality()
6
>>> L.structures([Integer(1),Integer(2),Integer(3)]).list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

weight_ring()

EXAMPLES:

```
sage: F = species.CombinatorialSpecies()
sage: F.weight_ring()
Rational Field
```

```
>>> from sage.all import *
>>> F = species.CombinatorialSpecies()
>>> F.weight_ring()
Rational Field
```

```
sage: X = species.SingletonSpecies()
sage: E = species.EmptySetSpecies()
sage: L = CombinatorialSpecies()
sage: L.define(E+X*L)
sage: L.weight_ring()
Rational Field
```

```
>>> from sage.all import *
>>> X = species.SingletonSpecies()
>>> E = species.EmptySetSpecies()
>>> L = CombinatorialSpecies()
>>> L.define(E+X*L)
>>> L.weight_ring()
Rational Field
```

class sage.combinat.species.recursive_species.**CombinatorialSpeciesStructure**(*parent*, *s*, ***options*)

Bases: *SpeciesStructureWrapper*

5.1.329 Set Species

class sage.combinat.species.set_species.**SetSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the species of sets.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: E.structures([1,2,3]).list()
[{1, 2, 3}]
sage: E.isotype_generating_series()[0:4]
[1, 1, 1, 1]
```

(continues on next page)

(continued from previous page)

```
sage: S = species.SetSpecies()
sage: c = S.generating_series()[0:3]
sage: S._check()
True
sage: S == loads(dumps(S))
True
```

```
>>> from sage.all import *
>>> E = species.SetSpecies()
>>> E.structures([Integer(1),Integer(2),Integer(3)]).list()
[{1, 2, 3}]
>>> E.isotype_generating_series()[Integer(0):Integer(4)]
[1, 1, 1, 1]

>>> S = species.SetSpecies()
>>> c = S.generating_series()[Integer(0):Integer(3)]
>>> S._check()
True
>>> S == loads(dumps(S))
True
```

class sage.combinat.species.set_species.**SetSpeciesStructure**(*parent*, *labels*, *list*)

Bases: *GenericSpeciesStructure*

automorphism_group()

Return the group of permutations whose action on this set leave it fixed. For the species of sets, there is only one isomorphism class, so every permutation is in its automorphism group.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.automorphism_group()                                          #␣
↪needs sage.groups
Symmetric group of order 3! as a permutation group
```

```
>>> from sage.all import *
>>> F = species.SetSpecies()
>>> a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
>>> a.automorphism_group()                                            #␣
↪needs sage.groups
Symmetric group of order 3! as a permutation group
```

canonical_label()

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: a = S.structures(["a","b","c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.canonical_label()
{'a', 'b', 'c'}
```

```
>>> from sage.all import *
>>> S = species.SetSpecies()
>>> a = S.structures(["a","b","c"]).random_element(); a
{'a', 'b', 'c'}
>>> a.canonical_label()
{'a', 'b', 'c'}
```

transport (*perm*)

Return the transport of this set along the permutation perm.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: p = PermutationGroupElement((1,2))                                    #␣
↪needs sage.groups
sage: a.transport(p)                                                        #␣
↪needs sage.groups
{'a', 'b', 'c'}
```

```
>>> from sage.all import *
>>> F = species.SetSpecies()
>>> a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
>>> p = PermutationGroupElement((Integer(1),Integer(2)))                    ␣
↪             # needs sage.groups
>>> a.transport(p)                                                        #␣
↪needs sage.groups
{'a', 'b', 'c'}
```

sage.combinat.species.set_species.**SetSpecies_class**

alias of *SetSpecies*

5.1.330 Combinatorial Species

This file defines the main classes for working with combinatorial species, operations on them, as well as some implementations of basic species required for other constructions.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse8.html.

Weighted Species:

As a first application of weighted species, we count unlabeled ordered trees by total number of nodes and number of internal nodes. To achieve this, we assign a weight of 1 to the leaves and of q to internal nodes:

```
sage: q = QQ['q'].gen()
sage: leaf = species.SingletonSpecies()
sage: internal_node = species.SingletonSpecies(weight=q)
sage: L = species.LinearOrderSpecies(min=1)
sage: T = species.CombinatorialSpecies(min=1)
sage: T.define(leaf + internal_node*L(T))
```

(continues on next page)

(continued from previous page)

```
sage: T.isotype_generating_series()[0:6]                                  #␣
↪needs sage.modules
[0, 1, q, q^2 + q, q^3 + 3*q^2 + q, q^4 + 6*q^3 + 6*q^2 + q]
```

```
>>> from sage.all import *
>>> q = QQ['q'].gen()
>>> leaf = species.SingletonSpecies()
>>> internal_node = species.SingletonSpecies(weight=q)
>>> L = species.LinearOrderSpecies(min=Integer(1))
>>> T = species.CombinatorialSpecies(min=Integer(1))
>>> T.define(leaf + internal_node*L(T))
>>> T.isotype_generating_series()[Integer(0):Integer(6)]                   ␣
↪              # needs sage.modules
[0, 1, q, q^2 + q, q^3 + 3*q^2 + q, q^4 + 6*q^3 + 6*q^2 + q]
```

Consider the following:

```
sage: T.isotype_generating_series().coefficient(4)                        #␣
↪needs sage.modules
q^3 + 3*q^2 + q
```

```
>>> from sage.all import *
>>> T.isotype_generating_series().coefficient(Integer(4))                  ␣
↪     # needs sage.modules
q^3 + 3*q^2 + q
```

This means that, among the trees on 4 nodes, one has a single internal node, three have two internal nodes, and one has three internal nodes.

class sage.combinat.species.species.**GenericCombinatorialSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: SageObject

algebraic_equation_system()

Return a system of algebraic equations satisfied by this species.

The nodes are numbered in the order that they appear as vertices of the associated digraph.

EXAMPLES:

```
sage: B = species.BinaryTreeSpecies()
sage: B.algebraic_equation_system()                                      #␣
↪needs sage.graphs
[-node3^2 + node1, -node1 + node3 + (-z)]
```

```
>>> from sage.all import *
>>> B = species.BinaryTreeSpecies()
>>> B.algebraic_equation_system()                                      #␣
↪needs sage.graphs
[-node3^2 + node1, -node1 + node3 + (-z)]
```

```
sage: sorted(B.digraph().vertex_iterator(), key=str)                     #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
[Combinatorial species with min=1,
 Product of (Combinatorial species with min=1)
        and (Combinatorial species with min=1),
 Singleton species,
 Sum of (Singleton species)
    and (Product of (Combinatorial species with min=1)
    and (Combinatorial species with min=1))]
```

```
>>> from sage.all import *
>>> sorted(B.digraph().vertex_iterator(), key=str)                          #␣
↪needs sage.graphs
[Combinatorial species with min=1,
 Product of (Combinatorial species with min=1)
        and (Combinatorial species with min=1),
 Singleton species,
 Sum of (Singleton species)
    and (Product of (Combinatorial species with min=1)
    and (Combinatorial species with min=1))]
```

```
sage: B.algebraic_equation_system()[0].parent()                              #␣
↪needs sage.graphs
Multivariate Polynomial Ring in node0, node1, node2, node3 over
 Fraction Field of Univariate Polynomial Ring in z over Rational Field
```

```
>>> from sage.all import *
>>> B.algebraic_equation_system()[Integer(0)].parent()                        ␣
↪     # needs sage.graphs
Multivariate Polynomial Ring in node0, node1, node2, node3 over
 Fraction Field of Univariate Polynomial Ring in z over Rational Field
```

composition(*g*)

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: S(S)
Composition of (Set species) and (Set species)
```

```
>>> from sage.all import *
>>> S = species.SetSpecies()
>>> S(S)
Composition of (Set species) and (Set species)
```

cycle_index_series(*base_ring=None*)

Return the cycle index series for this species.

The cycle index series is a sequence of symmetric functions.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.cycle_index_series()                                             #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
sage: g[0:4]                                                            #␣
↪needs sage.modules
[p[], p[1], p[1, 1] + p[2], p[1, 1, 1] + p[2, 1] + p[3]]
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> g = P.cycle_index_series()                                        #␣
↪needs sage.modules
>>> g[Integer(0):Integer(4)]                                            ␣
↪              # needs sage.modules
[p[], p[1], p[1, 1] + p[2], p[1, 1, 1] + p[2, 1] + p[3]]
```

digraph()

Return a directed graph where the vertices are the individual species that make up this one.

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: B = species.CombinatorialSpecies()
sage: B.define(X+B*B)
sage: g = B.digraph(); g                                                #␣
↪needs sage.graphs
Multi-digraph on 4 vertices

sage: sorted(g, key=str)                                                #␣
↪needs sage.graphs
[Combinatorial species,
 Product of (Combinatorial species) and (Combinatorial species),
 Singleton species,
 Sum of (Singleton species) and
  (Product of (Combinatorial species) and (Combinatorial species))]

sage: d = {sp: i for i, sp in enumerate(g)}                             #␣
↪needs sage.graphs
sage: g.relabel(d)                                                      #␣
↪needs sage.graphs
sage: g.canonical_label().edges(sort=True)                              #␣
↪needs sage.graphs
[(0, 3, None), (2, 0, None), (2, 0, None), (3, 1, None), (3, 2, None)]
```

```
>>> from sage.all import *
>>> X = species.SingletonSpecies()
>>> B = species.CombinatorialSpecies()
>>> B.define(X+B*B)
>>> g = B.digraph(); g                                                #␣
↪needs sage.graphs
Multi-digraph on 4 vertices

>>> sorted(g, key=str)                                                #␣
↪needs sage.graphs
[Combinatorial species,
 Product of (Combinatorial species) and (Combinatorial species),
```

(continues on next page)

(continued from previous page)

```
 Singleton species,
 Sum of (Singleton species) and
  (Product of (Combinatorial species) and (Combinatorial species))]

>>> d = {sp: i for i, sp in enumerate(g)}                                   #␣
↪needs sage.graphs
>>> g.relabel(d)                                                            #␣
↪needs sage.graphs
>>> g.canonical_label().edges(sort=True)                                    #␣
↪needs sage.graphs
[(0, 3, None), (2, 0, None), (2, 0, None), (3, 1, None), (3, 2, None)]
```

functorial_composition(*g*)

Return the functorial composition of `self` with `g`.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: E2 = E.restricted(min=2, max=3)
sage: WP = species.SubsetSpecies()
sage: P2 = E2*E
sage: G = WP.functorial_composition(P2)
sage: G.isotype_generating_series()[0:5]                                    #␣
↪needs sage.modules
[1, 1, 2, 4, 11]
```

```
>>> from sage.all import *
>>> E = species.SetSpecies()
>>> E2 = E.restricted(min=Integer(2), max=Integer(3))
>>> WP = species.SubsetSpecies()
>>> P2 = E2*E
>>> G = WP.functorial_composition(P2)
>>> G.isotype_generating_series()[Integer(0):Integer(5)]                    ␣
↪             # needs sage.modules
[1, 1, 2, 4, 11]
```

generating_series(*base_ring=None*)

Return the generating series for this species.

This is an exponential generating series so the n-th coefficient of the series corresponds to the number of labeled structures with n labels divided by $n!$.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.generating_series()
sage: g[:4]
[1, 1, 1, 1]
sage: g.counts(4)
[1, 1, 2, 6]
sage: P.structures([1,2,3]).list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
sage: len(_)
6
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> g = P.generating_series()
>>> g[:Integer(4)]
[1, 1, 1, 1]
>>> g.counts(Integer(4))
[1, 1, 2, 6]
>>> P.structures([Integer(1),Integer(2),Integer(3)]).list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
>>> len(_)
6
```

is_weighted()

Return True if this species has a nontrivial weighting associated with it.

EXAMPLES:

```
sage: C = species.CycleSpecies()
sage: C.is_weighted()
False
```

```
>>> from sage.all import *
>>> C = species.CycleSpecies()
>>> C.is_weighted()
False
```

isotype_generating_series(*base_ring=None*)

Return the isotype generating series for this species.

The n-th coefficient of this series corresponds to the number of isomorphism types for the structures on n labels.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.isotype_generating_series()
sage: g[0:4]                                                            #␣
↪needs sage.libs.flint
[1, 1, 2, 3]
sage: g.counts(4)                                                       #␣
↪needs sage.libs.flint
[1, 1, 2, 3]
sage: P.isotypes([1,2,3]).list()                                        #␣
↪needs sage.libs.flint
[[2, 3, 1], [2, 1, 3], [1, 2, 3]]
sage: len(_)                                                            #␣
↪needs sage.libs.flint
3
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> g = P.isotype_generating_series()
>>> g[Integer(0):Integer(4)]                                            ␣
↪              # needs sage.libs.flint
```

(continues on next page)

(continued from previous page)

```
[1, 1, 2, 3]
>>> g.counts(Integer(4))                                                    ␣
↪     # needs sage.libs.flint
[1, 1, 2, 3]
>>> P.isotypes([Integer(1),Integer(2),Integer(3)]).list()                   ␣
↪                         # needs sage.libs.flint
[[2, 3, 1], [2, 1, 3], [1, 2, 3]]
>>> len(_)                                                                 #␣
↪needs sage.libs.flint
3
```

isotypes(*labels*, *structure_class=None*)

EXAMPLES:

```
sage: F = CombinatorialSpecies()
sage: F.isotypes([1,2,3]).list()
Traceback (most recent call last):
...
ValueError: Stream is not yet defined
```

```
>>> from sage.all import *
>>> F = CombinatorialSpecies()
>>> F.isotypes([Integer(1),Integer(2),Integer(3)]).list()
Traceback (most recent call last):
...
ValueError: Stream is not yet defined
```

product(*g*)

Return the product of `self` and `g`.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P * P; F
Product of (Permutation species) and (Permutation species)
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> F = P * P; F
Product of (Permutation species) and (Permutation species)
```

restricted(**args*, ***kwds*)

Return the restriction of the species.

INPUT:

- `min` – (optional) integer
- `max` – (optional) integer

EXAMPLES:

```
sage: S = species.SetSpecies().restricted(min=3); S
Set species with min=3
```

(continues on next page)

(continued from previous page)

```
sage: S.structures([1,2]).list()
[]
sage: S.generating_series()[0:5]
[0, 0, 0, 1/6, 1/24]
```

```
>>> from sage.all import *
>>> S = species.SetSpecies().restricted(min=Integer(3)); S
Set species with min=3
>>> S.structures([Integer(1),Integer(2)]).list()
[]
>>> S.generating_series()[Integer(0):Integer(5)]
[0, 0, 0, 1/6, 1/24]
```

structures(*labels*, *structure_class=None*)

EXAMPLES:

```
sage: F = CombinatorialSpecies()
sage: F.structures([1,2,3]).list()
Traceback (most recent call last):
...
ValueError: Stream is not yet defined
```

```
>>> from sage.all import *
>>> F = CombinatorialSpecies()
>>> F.structures([Integer(1),Integer(2),Integer(3)]).list()
Traceback (most recent call last):
...
ValueError: Stream is not yet defined
```

sum(*g*)

Return the sum of `self` and `g`.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P + P; F
Sum of (Permutation species) and (Permutation species)
sage: F.structures([1,2]).list()
[[1, 2], [2, 1], [1, 2], [2, 1]]
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> F = P + P; F
Sum of (Permutation species) and (Permutation species)
>>> F.structures([Integer(1),Integer(2)]).list()
[[1, 2], [2, 1], [1, 2], [2, 1]]
```

weight_ring()

Return the ring in which the weights of this species occur.

By default, this is just the field of rational numbers.

EXAMPLES:

```
sage: species.SetSpecies().weight_ring()
Rational Field
```

```
>>> from sage.all import *
>>> species.SetSpecies().weight_ring()
Rational Field
```

weighted(*weight*)

Return a version of this species with the specified weight.

EXAMPLES:

```
sage: t = ZZ['t'].gen()
sage: C = species.CycleSpecies(); C
Cyclic permutation species
sage: C.weighted(t)
Cyclic permutation species with weight=t
```

```
>>> from sage.all import *
>>> t = ZZ['t'].gen()
>>> C = species.CycleSpecies(); C
Cyclic permutation species
>>> C.weighted(t)
Cyclic permutation species with weight=t
```

5.1.331 Species structures

We will illustrate the use of the structure classes using the "balls and bars" model for integer compositions. An integer composition of 6 such as [2, 1, 3] can be represented in this model as 'oooooo' where the 6 o's correspond to the balls and the 2 's correspond to the bars. If BB is our species for this model, the it satisfies the following recursive definition:

BB = o + o*BB + o*|*BB

Here we define this species using the default structures:

```
sage: ball = species.SingletonSpecies()
sage: bar = species.EmptySetSpecies()
sage: BB = CombinatorialSpecies()
sage: BB.define(ball + ball*BB + ball*bar*BB)
sage: o = var('o')                                                          #␣
↪needs sage.symbolic
sage: BB.isotypes([o]*3).list()                                             #␣
↪needs sage.symbolic
[o*(o*o), o*((o*{})*o), (o*{})*(o*o), (o*{})*((o*{})*o)]
```

```
>>> from sage.all import *
>>> ball = species.SingletonSpecies()
>>> bar = species.EmptySetSpecies()
>>> BB = CombinatorialSpecies()
>>> BB.define(ball + ball*BB + ball*bar*BB)
>>> o = var('o')                                                        #␣
↪needs sage.symbolic
>>> BB.isotypes([o]*Integer(3)).list()                                    ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.symbolic
[o*(o*o), o*((o*{})*o), (o*{})*(o*o), (o*{})*((o*{})*o)]
```

If we ignore the parentheses, we can read off that the integer compositions are [3], [2, 1], [1, 2], and [1, 1, 1].

class sage.combinat.species.structure.**GenericSpeciesStructure**(*parent*, *labels*, *list*)

Bases: *CombinatorialObject*

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.species.structure import GenericSpeciesStructure
>>> a = GenericSpeciesStructure(None, [Integer(2),Integer(3),Integer(4)],␣
↪[Integer(1),Integer(2),Integer(3)])
>>> a
[2, 3, 4]
>>> a.parent() is None
True
>>> a == loads(dumps(a))
True
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- labels – list of labels

OUTPUT:

A structure with the i-th label of self replaced with the i-th label of the list.

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.change_labels([1,2,3]) for s in S]
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

```
>>> from sage.all import *
>>> P = species.SubsetSpecies()
>>> S = P.structures(["a", "b", "c"])
>>> [s.change_labels([Integer(1),Integer(2),Integer(3)]) for s in S]
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

is_isomorphic(*x*)

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: a = S.structures([1,2,3]).random_element(); a
{1, 2, 3}
sage: b = S.structures(['a','b','c']).random_element(); b
{'a', 'b', 'c'}
sage: a.is_isomorphic(b)
True
```

```
>>> from sage.all import *
>>> S = species.SetSpecies()
>>> a = S.structures([Integer(1),Integer(2),Integer(3)]).random_element(); a
{1, 2, 3}
>>> b = S.structures(['a','b','c']).random_element(); b
{'a', 'b', 'c'}
>>> a.is_isomorphic(b)
True
```

labels()

Return the labels used for this structure.

Note

This includes labels which may not "appear" in this particular structure.

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: s = P.structures(["a", "b", "c"]).random_element()
sage: s.labels()
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> P = species.SubsetSpecies()
>>> s = P.structures(["a", "b", "c"]).random_element()
>>> s.labels()
['a', 'b', 'c']
```

parent()

Return the species that this structure is associated with.

EXAMPLES:

```
sage: L = species.LinearOrderSpecies()
sage: a,b = L.structures([1,2])
sage: a.parent()
Linear order species
```

```
>>> from sage.all import *
>>> L = species.LinearOrderSpecies()
```

(continues on next page)

(continued from previous page)

```
>>> a,b = L.structures([Integer(1),Integer(2)])
>>> a.parent()
Linear order species
```

class sage.combinat.species.structure.**IsotypesWrapper**(*species*, *labels*, *structure_class*)

Bases: *SpeciesWrapper*

A base class for the set of isotypes of a species with given set of labels. An object of this type is returned when you call the `isotypes()` method of a species.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: S = F.isotypes([1,2,3])
sage: S == loads(dumps(S))
True
```

```
>>> from sage.all import *
>>> F = species.SetSpecies()
>>> S = F.isotypes([Integer(1),Integer(2),Integer(3)])
>>> S == loads(dumps(S))
True
```

class sage.combinat.species.structure.**SimpleIsotypesWrapper**(*species*, *labels*, *structure_class*)

Bases: *SpeciesWrapper*

> **Warning**
>
> This is deprecated and currently not used for anything.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: S = F.structures([1,2,3])
sage: S == loads(dumps(S))
True
```

```
>>> from sage.all import *
>>> F = species.SetSpecies()
>>> S = F.structures([Integer(1),Integer(2),Integer(3)])
>>> S == loads(dumps(S))
True
```

class sage.combinat.species.structure.**SimpleStructuresWrapper**(*species*, *labels*, *structure_class*)

Bases: *SpeciesWrapper*

> **Warning**
>
> This is deprecated and currently not used for anything.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: S = F.structures([1,2,3])
sage: S == loads(dumps(S))
True
```

```
>>> from sage.all import *
>>> F = species.SetSpecies()
>>> S = F.structures([Integer(1),Integer(2),Integer(3)])
>>> S == loads(dumps(S))
True
```

sage.combinat.species.structure.**SpeciesStructure**

alias of *GenericSpeciesStructure*

class sage.combinat.species.structure.**SpeciesStructureWrapper**(*parent*, *s*, ***options*)

Bases: *GenericSpeciesStructure*

This is a class for the structures of species such as the sum species that do not provide "additional" structure. For example, if you have the sum C of species A and B, then a structure of C will either be either something from A or B. Instead of just returning one of these directly, a "wrapper" is put around them so that they have their parent is C rather than A or B:

```
sage: X = species.SingletonSpecies()
sage: X2 = X+X
sage: s = X2.structures([1]).random_element(); s
1
sage: s.parent()
Sum of (Singleton species) and (Singleton species)
sage: from sage.combinat.species.structure import SpeciesStructureWrapper
sage: issubclass(type(s), SpeciesStructureWrapper)
True
```

```
>>> from sage.all import *
>>> X = species.SingletonSpecies()
>>> X2 = X+X
>>> s = X2.structures([Integer(1)]).random_element(); s
1
>>> s.parent()
Sum of (Singleton species) and (Singleton species)
>>> from sage.combinat.species.structure import SpeciesStructureWrapper
>>> issubclass(type(s), SpeciesStructureWrapper)
True
```

EXAMPLES:

```
sage: E = species.SetSpecies(); B = E+E
sage: s = B.structures([1,2,3]).random_element()
sage: s.parent()
Sum of (Set species) and (Set species)
sage: s == loads(dumps(s))
True
```

```
>>> from sage.all import *
>>> E = species.SetSpecies(); B = E+E
>>> s = B.structures([Integer(1),Integer(2),Integer(3)]).random_element()
>>> s.parent()
Sum of (Set species) and (Set species)
>>> s == loads(dumps(s))
True
```

canonical_label()

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: s = (P+P).structures([1,2,3])[1]; s                               #␣
↪needs sage.libs.flint
{{1, 3}, {2}}
sage: s.canonical_label()                                                #␣
↪needs sage.libs.flint
{{1, 2}, {3}}
```

```
>>> from sage.all import *
>>> P = species.PartitionSpecies()
>>> s = (P+P).structures([Integer(1),Integer(2),Integer(3)])[Integer(1)]; s  ␣
↪                               # needs sage.libs.flint
{{1, 3}, {2}}
>>> s.canonical_label()                                                #␣
↪needs sage.libs.flint
{{1, 2}, {3}}
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- `labels` – list of labels

OUTPUT:

A structure with the i-th label of `self` replaced with the i-th label of the list.

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: X2 = X+X
sage: s = X2.structures([1]).random_element(); s
1
sage: s.change_labels(['a'])
'a'
```

```
>>> from sage.all import *
>>> X = species.SingletonSpecies()
>>> X2 = X+X
>>> s = X2.structures([Integer(1)]).random_element(); s
1
>>> s.change_labels(['a'])
'a'
```

transport(*perm*)

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: s = (P+P).structures([1,2,3])[1]; s                                    #␣
↪needs sage.libs.flint
{{1, 3}, {2}}
sage: s.transport(PermutationGroupElement((2,3)))                            #␣
↪needs sage.groups sage.libs.flint
{{1, 2}, {3}}
```

```
>>> from sage.all import *
>>> P = species.PartitionSpecies()
>>> s = (P+P).structures([Integer(1),Integer(2),Integer(3)])[Integer(1)]; s  ␣
↪                                 # needs sage.libs.flint
{{1, 3}, {2}}
>>> s.transport(PermutationGroupElement((Integer(2),Integer(3))))           ␣
↪                  # needs sage.groups sage.libs.flint
{{1, 2}, {3}}
```

class sage.combinat.species.structure.**SpeciesWrapper**(*species*, *labels*, *iterator*, *generating_series*, *name*, *structure_class*)

Bases: `Parent`

This is a abstract base class for the set of structures of a species as well as the set of isotypes of the species.

Note: One typically does not use *SpeciesWrapper* directly, but instead instantiates one of its subclasses: *StructuresWrapper* or *IsotypesWrapper*.

EXAMPLES:

```
sage: from sage.combinat.species.structure import SpeciesWrapper
sage: F = species.SetSpecies()
sage: S = SpeciesWrapper(F, [1,2,3], "_structures", "generating_series",
↪'Structures', None)
sage: S
Structures for Set species with labels [1, 2, 3]
sage: S.list()
[{1, 2, 3}]
sage: S.cardinality()
1
```

```
>>> from sage.all import *
>>> from sage.combinat.species.structure import SpeciesWrapper
>>> F = species.SetSpecies()
>>> S = SpeciesWrapper(F, [Integer(1),Integer(2),Integer(3)], "_structures",
↪"generating_series", 'Structures', None)
>>> S
Structures for Set species with labels [1, 2, 3]
>>> S.list()
```

(continues on next page)

(continued from previous page)

```
[{1, 2, 3}]
>>> S.cardinality()
1
```

cardinality()

Return the number of structures in this set.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: F.structures([1,2,3]).cardinality()
1
```

```
>>> from sage.all import *
>>> F = species.SetSpecies()
>>> F.structures([Integer(1),Integer(2),Integer(3)]).cardinality()
1
```

labels()

Return the labels used on these structures. If X is the species, then *labels()* returns the preimage of these structures under the functor X.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: F.structures([1,2,3]).labels()
[1, 2, 3]
```

```
>>> from sage.all import *
>>> F = species.SetSpecies()
>>> F.structures([Integer(1),Integer(2),Integer(3)]).labels()
[1, 2, 3]
```

class sage.combinat.species.structure.**StructuresWrapper**(*species*, *labels*, *structure_class*)

Bases: *SpeciesWrapper*

A base class for the set of structures of a species with given set of labels. An object of this type is returned when you call the `structures()` method of a species.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: S = F.structures([1,2,3])
sage: S == loads(dumps(S))
True
```

```
>>> from sage.all import *
>>> F = species.SetSpecies()
>>> S = F.structures([Integer(1),Integer(2),Integer(3)])
>>> S == loads(dumps(S))
True
```

5.1.332 Subset Species

class sage.combinat.species.subset_species.**SubsetSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the species of subsets.

EXAMPLES:

```
sage: S = species.SubsetSpecies()
sage: S.generating_series()[0:5]
[1, 2, 2, 4/3, 2/3]
sage: S.isotype_generating_series()[0:5]
[1, 2, 3, 4, 5]

sage: S = species.SubsetSpecies()
sage: c = S.generating_series()[0:3]
sage: S._check()
True
sage: S == loads(dumps(S))
True
```

```
>>> from sage.all import *
>>> S = species.SubsetSpecies()
>>> S.generating_series()[Integer(0):Integer(5)]
[1, 2, 2, 4/3, 2/3]
>>> S.isotype_generating_series()[Integer(0):Integer(5)]
[1, 2, 3, 4, 5]

>>> S = species.SubsetSpecies()
>>> c = S.generating_series()[Integer(0):Integer(3)]
>>> S._check()
True
>>> S == loads(dumps(S))
True
```

class sage.combinat.species.subset_species.**SubsetSpeciesStructure**(*parent*, *labels*, *list*)

Bases: *GenericSpeciesStructure*

automorphism_group()

Return the group of permutations whose action on this subset leave it fixed.

EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures([1,2,3,4])[6]; a
{1, 3}
sage: a.automorphism_group()                                                    #␣
↪needs sage.groups
Permutation Group with generators [(2,4), (1,3)]
```

```
>>> from sage.all import *
>>> F = species.SubsetSpecies()
>>> a = F.structures([Integer(1),Integer(2),Integer(3),
↪Integer(4)])[Integer(6)]; a
```

(continues on next page)

(continued from previous page)

```
{1, 3}
>>> a.automorphism_group()                                                   #␣
↪needs sage.groups
Permutation Group with generators [(2,4), (1,3)]
```

```
sage: [a.transport(g) for g in a.automorphism_group()]                          #␣
↪needs sage.groups
[{1, 3}, {1, 3}, {1, 3}, {1, 3}]
```

```
>>> from sage.all import *
>>> [a.transport(g) for g in a.automorphism_group()]                        #␣
↪needs sage.groups
[{1, 3}, {1, 3}, {1, 3}, {1, 3}]
```

canonical_label()

Return the canonical label of `self`.

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[{}, {'a'}, {'a'}, {'a'}, {'a', 'b'}, {'a', 'b'}, {'a', 'b'}, {'a', 'b', 'c'}]
```

```
>>> from sage.all import *
>>> P = species.SubsetSpecies()
>>> S = P.structures(["a", "b", "c"])
>>> [s.canonical_label() for s in S]
[{}, {'a'}, {'a'}, {'a'}, {'a', 'b'}, {'a', 'b'}, {'a', 'b'}, {'a', 'b', 'c'}]
```

complement()

Return the complement of `self`.

EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures(["a", "b", "c"])[5]; a
{'a', 'c'}
sage: a.complement()
{'b'}
```

```
>>> from sage.all import *
>>> F = species.SubsetSpecies()
>>> a = F.structures(["a", "b", "c"])[Integer(5)]; a
{'a', 'c'}
>>> a.complement()
{'b'}
```

label_subset()

Return a subset of the labels that “appear” in this structure.

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.label_subset() for s in S]
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

```
>>> from sage.all import *
>>> P = species.SubsetSpecies()
>>> S = P.structures(["a", "b", "c"])
>>> [s.label_subset() for s in S]
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

transport(*perm*)

Return the transport of this subset along the permutation perm.

EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures(["a", "b", "c"])[5]; a
{'a', 'c'}
sage: p = PermutationGroupElement((1,2))                                    #␣
↪needs sage.groups
sage: a.transport(p)                                                        #␣
↪needs sage.groups
{'b', 'c'}
sage: p = PermutationGroupElement((1,3))                                    #␣
↪needs sage.groups
sage: a.transport(p)                                                        #␣
↪needs sage.groups
{'a', 'c'}
```

```
>>> from sage.all import *
>>> F = species.SubsetSpecies()
>>> a = F.structures(["a", "b", "c"])[Integer(5)]; a
{'a', 'c'}
>>> p = PermutationGroupElement((Integer(1),Integer(2)))                    ␣
↪               # needs sage.groups
>>> a.transport(p)                                                        #␣
↪needs sage.groups
{'b', 'c'}
>>> p = PermutationGroupElement((Integer(1),Integer(3)))                    ␣
↪               # needs sage.groups
>>> a.transport(p)                                                        #␣
↪needs sage.groups
{'a', 'c'}
```

sage.combinat.species.subset_species.**SubsetSpecies_class**

alias of `SubsetSpecies`

5.1.333 Sum species

class sage.combinat.species.sum_species.**SumSpecies**(*F*, *G*, *min=None*, *max=None*, *weight=None*)

Bases: *GenericCombinatorialSpecies*, UniqueRepresentation

Return the sum of two species.

EXAMPLES:

```
sage: S = species.PermutationSpecies()
sage: A = S+S
sage: A.generating_series()[:5]
[2, 2, 2, 2, 2]

sage: P = species.PermutationSpecies()
sage: F = P + P
sage: F._check()                                                    #␣
↪needs sage.libs.flint
True
sage: F == loads(dumps(F))
True
```

```
>>> from sage.all import *
>>> S = species.PermutationSpecies()
>>> A = S+S
>>> A.generating_series()[:Integer(5)]
[2, 2, 2, 2, 2]

>>> P = species.PermutationSpecies()
>>> F = P + P
>>> F._check()                                                  # needs␣
↪sage.libs.flint
True
>>> F == loads(dumps(F))
True
```

left_summand()

Return the left summand of this species.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P + P*P
sage: F.left_summand()
Permutation species
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> F = P + P*P
>>> F.left_summand()
Permutation species
```

right_summand()

Return the right summand of this species.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P + P*P
sage: F.right_summand()
Product of (Permutation species) and (Permutation species)
```

```
>>> from sage.all import *
>>> P = species.PermutationSpecies()
>>> F = P + P*P
>>> F.right_summand()
Product of (Permutation species) and (Permutation species)
```

weight_ring()

Return the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you add elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: C = S+S
sage: C.weight_ring()
Rational Field
```

```
>>> from sage.all import *
>>> S = species.SetSpecies()
>>> C = S+S
>>> C.weight_ring()
Rational Field
```

```
sage: S = species.SetSpecies(weight=QQ['t'].gen())
sage: C = S + S
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> S = species.SetSpecies(weight=QQ['t'].gen())
>>> C = S + S
>>> C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

class sage.combinat.species.sum_species.**SumSpeciesStructure**(*parent*, *s*, ***options*)

Bases: *SpeciesStructureWrapper*

sage.combinat.species.sum_species.**SumSpecies_class**

alias of *SumSpecies*

5.1.334 Specht Modules

AUTHORS:

- Travis Scrimshaw (2023-1-22): initial version
- Travis Scrimshaw (2023-11-23): added simple modules based on code from Sacha Goldman

Todo

Integrate this with the implementations in `sage.modules.with_basis.representation`.

class `sage.combinat.specht_module.`**`MaximalSpechtSubmodule`**(*specht_module*)

Bases: *SymmetricGroupRepresentation*, `SubmoduleWithBasis`

The maximal submodule U^λ of the Specht module S^λ.

ALGORITHM:

We construct U^λ as the intersection $S \cap S^\perp$, where $S^\perp$ is the orthogonal complement of the Specht module S inside of the tabloid module T (with respect to the natural bilinear form on T).

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 5)
sage: SM = SGA.specht_module([3,2])
sage: U = SM.maximal_submodule()
sage: u = U.an_element(); u
2*U[0] + 2*U[1]
sage: [p * u for p in list(SGA.basis())[:4]]
[2*U[0] + 2*U[1], 2*U[2] + 2*U[3], 2*U[0] + 2*U[1], U[0] + 2*U[2]]
sage: sum(SGA.basis()) * u
0
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(5))
>>> SM = SGA.specht_module([Integer(3),Integer(2)])
>>> U = SM.maximal_submodule()
>>> u = U.an_element(); u
2*U[0] + 2*U[1]
>>> [p * u for p in list(SGA.basis())[:Integer(4)]]
[2*U[0] + 2*U[1], 2*U[2] + 2*U[3], 2*U[0] + 2*U[1], U[0] + 2*U[2]]
>>> sum(SGA.basis()) * u
0
```

`Element`

alias of *Element*

class `sage.combinat.specht_module.`**`SimpleModule`**(*specht_module*)

Bases: *SymmetricGroupRepresentation*, `QuotientModuleWithBasis`

The simple S_n-module associated with a partition λ.

The simple module D^λ is the quotient of the Specht module S^λ by its *maximal submodule* U^λ.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 5)
sage: SM = SGA.specht_module([3,1,1])
sage: D = SM.simple_module()
sage: v = D.an_element(); v
2*D[[[1, 3, 5], [2], [4]]] + 2*D[[[1, 4, 5], [2], [3]]]
sage: SGA.an_element() * v
2*D[[[1, 2, 4], [3], [5]]] + 2*D[[[1, 3, 5], [2], [4]]]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(5))
>>> SM = SGA.specht_module([Integer(3),Integer(1),Integer(1)])
>>> D = SM.simple_module()
>>> v = D.an_element(); v
2*D[[[1, 3, 5], [2], [4]]] + 2*D[[[1, 4, 5], [2], [3]]]
>>> SGA.an_element() * v
2*D[[[1, 2, 4], [3], [5]]] + 2*D[[[1, 3, 5], [2], [4]]]
```

We give an example on how to construct the decomposition matrix (the Specht modules are a complete set of irreducible projective modules) and the Cartan matrix of a symmetric group algebra:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 4)
sage: BM = matrix(SGA.simple_module(la).brauer_character()
....:               for la in Partitions(4, regular=3))
sage: SBT = matrix(SGA.specht_module(la).brauer_character()
....:                for la in Partitions(4))
sage: D = SBT * ~BM; D
[1 0 0 0]
[0 1 0 0]
[1 0 1 0]
[0 0 0 1]
[0 0 1 0]
sage: D.transpose() * D
[2 0 1 0]
[0 1 0 0]
[1 0 2 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(4))
>>> BM = matrix(SGA.simple_module(la).brauer_character()
...               for la in Partitions(Integer(4), regular=Integer(3)))
>>> SBT = matrix(SGA.specht_module(la).brauer_character()
...                for la in Partitions(Integer(4)))
>>> D = SBT * ~BM; D
[1 0 0 0]
[0 1 0 0]
[1 0 1 0]
[0 0 0 1]
[0 0 1 0]
>>> D.transpose() * D
[2 0 1 0]
[0 1 0 0]
[1 0 2 0]
[0 0 0 1]
```

We verify this against the direct computation (up to reindexing the rows and columns):

```
sage: SGA.cartan_invariants_matrix()  # long time
[1 0 0 0]
[0 1 0 0]
[0 0 2 1]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 2]
```

```
>>> from sage.all import *
>>> SGA.cartan_invariants_matrix()  # long time
[1 0 0 0]
[0 1 0 0]
[0 0 2 1]
[0 0 1 2]
```

Element

alias of `Element`

class sage.combinat.specht_module.**SpechtModule**(*SGA*, *D*)

Bases: `SymmetricGroupRepresentation`, `SubmoduleWithBasis`

A Specht module.

Let S_n be the symmetric group on n letters and R be a commutative ring. The *Specht module* S^D for a diagram D is an S_n-module defined as follows. Let

$$R(D) := \sum_{w \in R_D} w, \qquad C(D) := \sum_{w \in C_D} (-1)^w w,$$

where R_D (resp. C_D) is the row (resp. column) stabilizer of D. Then, we construct the Specht module S^D as the left ideal

$$S^D = R[S_n]C(D)R(D),$$

where $R[S_n]$ is the group algebra of S_n over R.

INPUT:

- `SGA` – a symmetric group algebra
- `D` – a diagram

EXAMPLES:

We begin by constructing all irreducible Specht modules for the symmetric group S_4 and show that they give a full set of irreducible representations both by having distinct Frobenius characters and the sum of the square of their dimensions is equal to $4!$:

```
sage: SP = [la.specht_module(QQ) for la in Partitions(4)]
sage: s = SymmetricFunctions(QQ).s()
sage: [s(S.frobenius_image()) for S in SP]
[s[4], s[3, 1], s[2, 2], s[2, 1, 1], s[1, 1, 1, 1]]
sage: sum(S.dimension()^2 for S in SP)
24
```

```
>>> from sage.all import *
>>> SP = [la.specht_module(QQ) for la in Partitions(Integer(4))]
>>> s = SymmetricFunctions(QQ).s()
>>> [s(S.frobenius_image()) for S in SP]
[s[4], s[3, 1], s[2, 2], s[2, 1, 1], s[1, 1, 1, 1]]
>>> sum(S.dimension()**Integer(2) for S in SP)
24
```

Next, we compute the Specht module for a more general diagram for S_5 and compute its irreducible decomposition by using its Frobenius character:

```
sage: D = [(0,0), (0,1), (1,1), (1,2), (0,3)]
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SM = SGA.specht_module(D)
sage: SM.dimension()
9
sage: s(SM.frobenius_image())
s[3, 2] + s[4, 1]
```

```
>>> from sage.all import *
>>> D = [(Integer(0),Integer(0)), (Integer(0),Integer(1)), (Integer(1),
↪Integer(1)), (Integer(1),Integer(2)), (Integer(0),Integer(3))]
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SM = SGA.specht_module(D)
>>> SM.dimension()
9
>>> s(SM.frobenius_image())
s[3, 2] + s[4, 1]
```

This carries a natural (left) action of the symmetric group (algebra):

```
sage: S5 = SGA.group()
sage: v = SM.an_element(); v
2*S[0] + 2*S[1] + 3*S[2]
sage: S5([2,1,5,3,4]) * v
3*S[0] + 2*S[1] + 2*S[2]
sage: x = SGA.an_element(); x
[1, 2, 3, 4, 5] + 2*[1, 2, 3, 5, 4] + 3*[1, 2, 4, 3, 5] + [5, 1, 2, 3, 4]
sage: x * v
15*S[0] + 14*S[1] + 16*S[2] - 7*S[5] + 2*S[6] + 2*S[7]
```

```
>>> from sage.all import *
>>> S5 = SGA.group()
>>> v = SM.an_element(); v
2*S[0] + 2*S[1] + 3*S[2]
>>> S5([Integer(2),Integer(1),Integer(5),Integer(3),Integer(4)]) * v
3*S[0] + 2*S[1] + 2*S[2]
>>> x = SGA.an_element(); x
[1, 2, 3, 4, 5] + 2*[1, 2, 3, 5, 4] + 3*[1, 2, 4, 3, 5] + [5, 1, 2, 3, 4]
>>> x * v
15*S[0] + 14*S[1] + 16*S[2] - 7*S[5] + 2*S[6] + 2*S[7]
```

See also

SpechtRepresentation for an implementation of the representation by matrices.

class Element

Bases: IndexedFreeModuleElement

class sage.combinat.specht_module.**SpechtModuleTableauxBasis**(*ambient*)

Bases: *SpechtModule*

A Specht module of a partition in the classical standard tableau basis.

This is constructed as a S_n-submodule of the `TabloidModule` (also referred to as the standard module).

See also:

- `SpechtModule` for the generic diagram implementation constructed as a left ideal of the group algebra
- `SpechtRepresentation` for an implementation of the representation by matrices.

bilinear_form(*u*, *v*)

Return the natural bilinear form of `self` applied to `u` and `v`.

The natural bilinear form is given by the pullback of the natural bilinear form on the tabloid module (where the tabloid basis is an orthonormal basis).

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SM = SGA.specht_module([2,2,1])
sage: u = SM.an_element(); u
3*S[[1, 2], [3, 5], [4]] + 2*S[[1, 3], [2, 5], [4]] + 2*S[[1, 4], [2, 5], [3]]
sage: v = sum(SM.basis())
sage: SM.bilinear_form(u, v)
140
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SM = SGA.specht_module([Integer(2),Integer(2),Integer(1)])
>>> u = SM.an_element(); u
3*S[[1, 2], [3, 5], [4]] + 2*S[[1, 3], [2, 5], [4]] + 2*S[[1, 4], [2, 5], [3]]
>>> v = sum(SM.basis())
>>> SM.bilinear_form(u, v)
140
```

gram_matrix()

Return the Gram matrix of the natural bilinear form of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SM = SGA.specht_module([2,2,1])
sage: M = SM.gram_matrix(); M
[12  4 -4 -4  4]
[ 4 12  4  4  4]
[-4  4 12  4  4]
[-4  4  4 12  4]
[ 4  4  4  4 12]
sage: M.det() != 0
True
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SM = SGA.specht_module([Integer(2),Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> M = SM.gram_matrix(); M
[12  4 -4 -4  4]
[ 4 12  4  4  4]
[-4  4 12  4  4]
[-4  4  4 12  4]
[ 4  4  4  4 12]
>>> M.det() != Integer(0)
True
```

intrinsic_arrangement(*base_ring=None*)

Return the intrinsic arrangement of `self`.

Consider the Specht module S^λ with λ a (integer) partition of n (i.e., S^λ is an S_n-module). The *intrinsic arrangement* of S^λ is the central hyperplane arrangement in S^λ given by the hyperplanes H_α, indexed by a set partition α of $\{1, \ldots, n\}$ of size λ, defined by

$$H_\alpha := \bigoplus_{\tau \in T_\alpha} (S^\lambda)^\tau,$$

where T_α is some set of generating transpositions of the Young subgroup S_α and V^τ denotes the τ-invariant subspace of V. (These hyperplanes do not depend on the choice of T_α.)

This was introduced in [TVY2020] as a generalization of the braid arrangement, which is the case when $\lambda = (n-1, 1)$ (equivalently, for the irreducible representation of S_n given by the type A_{n-1} root system).

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 4)
sage: SM = SGA.specht_module([2, 1, 1])
sage: A = SM.intrinsic_arrangement()
sage: A.hyperplanes()
(Hyperplane T0 - T1 - 3*T2 + 0,
 Hyperplane T0 - T1 + T2 + 0,
 Hyperplane T0 + 3*T1 + T2 + 0,
 Hyperplane 3*T0 + T1 - T2 + 0)
sage: A.is_free()
False
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(4))
>>> SM = SGA.specht_module([Integer(2), Integer(1), Integer(1)])
>>> A = SM.intrinsic_arrangement()
>>> A.hyperplanes()
(Hyperplane T0 - T1 - 3*T2 + 0,
 Hyperplane T0 - T1 + T2 + 0,
 Hyperplane T0 + 3*T1 + T2 + 0,
 Hyperplane 3*T0 + T1 - T2 + 0)
>>> A.is_free()
False
```

We reproduce Example 3 of [TVY2020]:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: for la in Partitions(5):
```

(continues on next page)

(continued from previous page)

```
....:     SM = SGA.specht_module(la)
....:     A = SM.intrinsic_arrangement()
....:     print(la, A.characteristic_polynomial())
[5] 1
[4, 1] x^4 - 10*x^3 + 35*x^2 - 50*x + 24
[3, 2] x^5 - 15*x^4 + 90*x^3 - 260*x^2 + 350*x - 166
[3, 1, 1] x^6 - 10*x^5 + 45*x^4 - 115*x^3 + 175*x^2 - 147*x + 51
[2, 2, 1] x^5 - 10*x^4 + 45*x^3 - 105*x^2 + 120*x - 51
[2, 1, 1, 1] x^4 - 5*x^3 + 10*x^2 - 10*x + 4
[1, 1, 1, 1, 1] 1

sage: A = SGA.specht_module([4, 1]).intrinsic_arrangement()
sage: A.characteristic_polynomial().factor()
(x - 4) * (x - 3) * (x - 2) * (x - 1)
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> for la in Partitions(Integer(5)):
...     SM = SGA.specht_module(la)
...     A = SM.intrinsic_arrangement()
...     print(la, A.characteristic_polynomial())
[5] 1
[4, 1] x^4 - 10*x^3 + 35*x^2 - 50*x + 24
[3, 2] x^5 - 15*x^4 + 90*x^3 - 260*x^2 + 350*x - 166
[3, 1, 1] x^6 - 10*x^5 + 45*x^4 - 115*x^3 + 175*x^2 - 147*x + 51
[2, 2, 1] x^5 - 10*x^4 + 45*x^3 - 105*x^2 + 120*x - 51
[2, 1, 1, 1] x^4 - 5*x^3 + 10*x^2 - 10*x + 4
[1, 1, 1, 1, 1] 1

>>> A = SGA.specht_module([Integer(4), Integer(1)]).intrinsic_arrangement()
>>> A.characteristic_polynomial().factor()
(x - 4) * (x - 3) * (x - 2) * (x - 1)
```

lift()

The lift (embedding) map from `self` to the ambient space.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SM = SGA.specht_module([3, 1, 1])
sage: SM.lift
Generic morphism:
  From: Specht module of [3, 1, 1] over Rational Field
  To:   Tabloid module of [3, 1, 1] over Rational Field
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SM = SGA.specht_module([Integer(3), Integer(1), Integer(1)])
>>> SM.lift
Generic morphism:
  From: Specht module of [3, 1, 1] over Rational Field
  To:   Tabloid module of [3, 1, 1] over Rational Field
```

maximal_submodule()

Return the maximal submodule of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 5)
sage: SM = SGA.specht_module([3,2])
sage: U = SM.maximal_submodule()
sage: U.dimension()
4
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(5))
>>> SM = SGA.specht_module([Integer(3),Integer(2)])
>>> U = SM.maximal_submodule()
>>> U.dimension()
4
```

retract()

The retract map from the ambient space.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: X = SGA.tabloid_module([2,2,1])
sage: Y = X.specht_module()
sage: Y.retract
Generic morphism:
  From: Tabloid module of [2, 2, 1] over Rational Field
  To:   Specht module of [2, 2, 1] over Rational Field
sage: all(Y.retract(u.lift()) == u for u in Y.basis())
True

sage: Y.retract(X.zero())
0
sage: Y.retract(sum(X.basis()))
Traceback (most recent call last):
...
ValueError: ... is not in the image
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> X = SGA.tabloid_module([Integer(2),Integer(2),Integer(1)])
>>> Y = X.specht_module()
>>> Y.retract
Generic morphism:
  From: Tabloid module of [2, 2, 1] over Rational Field
  To:   Specht module of [2, 2, 1] over Rational Field
>>> all(Y.retract(u.lift()) == u for u in Y.basis())
True

>>> Y.retract(X.zero())
0
>>> Y.retract(sum(X.basis()))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: ... is not in the image
```

simple_module()

Return the simple (or irreducible) S_n-submodule of `self`.

See also

SimpleModule

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 5)
sage: SM = SGA.specht_module([3,2])
sage: L = SM.simple_module()
sage: L.dimension()
1

sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SM = SGA.specht_module([3,2])
sage: SM.simple_module() is SM
True
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(5))
>>> SM = SGA.specht_module([Integer(3),Integer(2)])
>>> L = SM.simple_module()
>>> L.dimension()
1

>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SM = SGA.specht_module([Integer(3),Integer(2)])
>>> SM.simple_module() is SM
True
```

class sage.combinat.specht_module.**SymmetricGroupRepresentation**(*SGA*)

Bases: `Representation_abstract`

Mixin class for symmetric group (algebra) representations.

frobenius_image()

Return the Frobenius image of `self`.

The Frobenius map is defined as the map to symmetric functions

$$F(\chi) = \frac{1}{n!} \sum_{w \in S_n} \chi(w) p_{\rho(w)},$$

where χ is the character of the S_n-module `self`, p_λ is the powersum symmetric function basis element indexed by λ, and $\rho(w)$ is the cycle type of w as a partition. Specifically, this map takes irreducible representations indexed by λ to the Schur function s_λ.

EXAMPLES:

```
sage: SM = Partition([2,2,1]).specht_module(QQ)
sage: SM.frobenius_image()
s[2, 2, 1]
sage: SM = Partition([4,1]).specht_module(CyclotomicField(5))
sage: SM.frobenius_image()
s[4, 1]
```

```
>>> from sage.all import *
>>> SM = Partition([Integer(2),Integer(2),Integer(1)]).specht_module(QQ)
>>> SM.frobenius_image()
s[2, 2, 1]
>>> SM = Partition([Integer(4),Integer(1)]).specht_
↪module(CyclotomicField(Integer(5)))
>>> SM.frobenius_image()
s[4, 1]
```

We verify the regular representation:

```
sage: from sage.combinat.diagram import Diagram
sage: D = Diagram([(0,0), (1,1), (2,2), (3,3), (4,4)])
sage: F = D.specht_module(QQ).frobenius_image(); F
s[1, 1, 1, 1, 1] + 4*s[2, 1, 1, 1] + 5*s[2, 2, 1]
 + 6*s[3, 1, 1] + 5*s[3, 2] + 4*s[4, 1] + s[5]
sage: s = SymmetricFunctions(QQ).s()
sage: F == sum(StandardTableaux(la).cardinality() * s[la]
....:          for la in Partitions(5))
True
sage: all(s[la] == la.specht_module(QQ).frobenius_image()
....:     for n in range(1, 5) for la in Partitions(n))
True

sage: D = Diagram([(0,0), (1,1), (1,2), (2,3), (2,4)])
sage: SM = D.specht_module(QQ)
sage: SM.frobenius_image()
s[2, 2, 1] + s[3, 1, 1] + 2*s[3, 2] + 2*s[4, 1] + s[5]
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram import Diagram
>>> D = Diagram([(Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(2)), (Integer(3),Integer(3)), (Integer(4),Integer(4))])
>>> F = D.specht_module(QQ).frobenius_image(); F
s[1, 1, 1, 1, 1] + 4*s[2, 1, 1, 1] + 5*s[2, 2, 1]
 + 6*s[3, 1, 1] + 5*s[3, 2] + 4*s[4, 1] + s[5]
>>> s = SymmetricFunctions(QQ).s()
>>> F == sum(StandardTableaux(la).cardinality() * s[la]
...          for la in Partitions(Integer(5)))
True
>>> all(s[la] == la.specht_module(QQ).frobenius_image()
...     for n in range(Integer(1), Integer(5)) for la in Partitions(n))
True

>>> D = Diagram([(Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
```

(continues on next page)

(continued from previous page)

```
↪(Integer(1),Integer(2)), (Integer(2),Integer(3)), (Integer(2),Integer(4))])
>>> SM = D.specht_module(QQ)
>>> SM.frobenius_image()
s[2, 2, 1] + s[3, 1, 1] + 2*s[3, 2] + 2*s[4, 1] + s[5]
```

An example using the tabloid module:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: TM = SGA.tabloid_module([2, 2, 1])
sage: TM.frobenius_image()
s[2, 2, 1] + s[3, 1, 1] + 2*s[3, 2] + 2*s[4, 1] + s[5]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> TM = SGA.tabloid_module([Integer(2), Integer(2), Integer(1)])
>>> TM.frobenius_image()
s[2, 2, 1] + s[3, 1, 1] + 2*s[3, 2] + 2*s[4, 1] + s[5]
```

class sage.combinat.specht_module.**TabloidModule**(*SGA*, *shape*)

Bases: *SymmetricGroupRepresentation*, *CombinatorialFreeModule*

The vector space of all tabloids of a fixed shape with the natural symmetric group action.

A *tabloid* is an *OrderedSetPartition* whose underlying set is $\{1, \ldots, n\}$. The symmetric group acts by permuting the entries of the set. Hence, this is a representation of the symmetric group defined over any field.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 5)
sage: TM = SGA.tabloid_module([2, 2, 1])
sage: TM.dimension()
30
sage: TM.brauer_character()
(30, 6, 2, 0, 0)
sage: IM = TM.invariant_module()
sage: IM.dimension()
1
sage: IM.basis()[0].lift() == sum(TM.basis())
True
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(5))
>>> TM = SGA.tabloid_module([Integer(2), Integer(2), Integer(1)])
>>> TM.dimension()
30
>>> TM.brauer_character()
(30, 6, 2, 0, 0)
>>> IM = TM.invariant_module()
>>> IM.dimension()
1
>>> IM.basis()[Integer(0)].lift() == sum(TM.basis())
True
```

class Element

Bases: IndexedFreeModuleElement

bilinear_form(*u*, *v*)

Return the natural bilinear form of `self` applied to `u` and `v`.

The natural bilinear form is given by defining the tabloid basis to be orthonormal.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: TM = SGA.tabloid_module([2,2,1])
sage: u = TM.an_element(); u
2*T[{1, 2}, {3, 4}, {5}] + 2*T[{1, 2}, {3, 5}, {4}] + 3*T[{1, 2}, {4, 5}, {3}]
sage: v = sum(TM.basis())
sage: TM.bilinear_form(u, v)
7
sage: TM.bilinear_form(u, TM.zero())
0
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> TM = SGA.tabloid_module([Integer(2),Integer(2),Integer(1)])
>>> u = TM.an_element(); u
2*T[{1, 2}, {3, 4}, {5}] + 2*T[{1, 2}, {3, 5}, {4}] + 3*T[{1, 2}, {4, 5}, {3}]
>>> v = sum(TM.basis())
>>> TM.bilinear_form(u, v)
7
>>> TM.bilinear_form(u, TM.zero())
0
```

specht_module()

Return the Specht submodule of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: TM = SGA.tabloid_module([2,2,1])
sage: TM.specht_module() is SGA.specht_module([2,2,1])
True
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> TM = SGA.tabloid_module([Integer(2),Integer(2),Integer(1)])
>>> TM.specht_module() is SGA.specht_module([Integer(2),Integer(2),
↪Integer(1)])
True
```

sage.combinat.specht_module.**polytabloid**(T)

Compute the polytabloid element associated to a tableau `T`.

For a tableau T, the polytabloid associated to T is

$$e_T = \sum_{\sigma \in C_T} (-1)^\sigma \{\sigma T\},$$

where $\{\}$ is the row-equivalence class, i.e. a tabloid, and C_T is the column stabilizer of T. The sum takes place in the module spanned by tabloids $\{T\}$.

OUTPUT:

A `dict` whose keys are tabloids represented by tuples of frozensets and whose values are the coefficient.

EXAMPLES:

```
sage: from sage.combinat.specht_module import polytabloid
sage: T = StandardTableau([[1,3,4],[2,5]])
sage: polytabloid(T)
{(frozenset({1, 3, 4}), frozenset({2, 5})): 1,
 (frozenset({1, 4, 5}), frozenset({2, 3})): -1,
 (frozenset({2, 3, 4}), frozenset({1, 5})): -1,
 (frozenset({2, 4, 5}), frozenset({1, 3})): 1}
```

```
>>> from sage.all import *
>>> from sage.combinat.specht_module import polytabloid
>>> T = StandardTableau([[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(5)]])
>>> polytabloid(T)
{(frozenset({1, 3, 4}), frozenset({2, 5})): 1,
 (frozenset({1, 4, 5}), frozenset({2, 3})): -1,
 (frozenset({2, 3, 4}), frozenset({1, 5})): -1,
 (frozenset({2, 4, 5}), frozenset({1, 3})): 1}
```

`sage.combinat.specht_module.`**`simple_module_rank`**(*la*, *base_ring*)

Return the rank of the simple S_n-module corresponding to the partition `la` of size n over `base_ring`.

EXAMPLES:

```
sage: from sage.combinat.specht_module import simple_module_rank
sage: simple_module_rank([3,2,1,1], GF(3))
13
```

```
>>> from sage.all import *
>>> from sage.combinat.specht_module import simple_module_rank
>>> simple_module_rank([Integer(3),Integer(2),Integer(1),Integer(1)],␣
↪GF(Integer(3)))
13
```

`sage.combinat.specht_module.`**`specht_module_rank`**(*D*, *base_ring=None*)

Return the rank of the Specht module of diagram `D`.

EXAMPLES:

```
sage: from sage.combinat.specht_module import specht_module_rank
sage: specht_module_rank([(0,0), (1,1), (2,2)])
6
```

```
>>> from sage.all import *
>>> from sage.combinat.specht_module import specht_module_rank
>>> specht_module_rank([(Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(2))])
6
```

sage.combinat.specht_module.**specht_module_spanning_set**(*D*, *SGA=None*)

Return a spanning set of the Specht module of diagram D.

INPUT:

- D – list of cells (r,c) for row r and column c
- SGA – (optional) a symmetric group algebra

EXAMPLES:

```
sage: from sage.combinat.specht_module import specht_module_spanning_set
sage: specht_module_spanning_set([(0,0), (1,1), (2,2)])
([1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1])
sage: specht_module_spanning_set([(0,0), (1,1), (2,1)])
([1, 2, 3] - [1, 3, 2], -[1, 2, 3] + [1, 3, 2], [2, 1, 3] - [3, 1, 2],
 [2, 3, 1] - [3, 2, 1], -[2, 1, 3] + [3, 1, 2], -[2, 3, 1] + [3, 2, 1])

sage: SGA = SymmetricGroup(3).algebra(QQ)
sage: specht_module_spanning_set([(0,0), (1,1), (2,1)], SGA)
(() - (2,3), -(1,2) + (1,3,2), (1,2,3) - (1,3),
 -() + (2,3), -(1,2,3) + (1,3), (1,2) - (1,3,2))
```

```
>>> from sage.all import *
>>> from sage.combinat.specht_module import specht_module_spanning_set
>>> specht_module_spanning_set([(Integer(0),Integer(0)), (Integer(1),Integer(1)),
↪(Integer(2),Integer(2))])
([1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1])
>>> specht_module_spanning_set([(Integer(0),Integer(0)), (Integer(1),Integer(1)),
↪(Integer(2),Integer(1))])
([1, 2, 3] - [1, 3, 2], -[1, 2, 3] + [1, 3, 2], [2, 1, 3] - [3, 1, 2],
 [2, 3, 1] - [3, 2, 1], -[2, 1, 3] + [3, 1, 2], -[2, 3, 1] + [3, 2, 1])

>>> SGA = SymmetricGroup(Integer(3)).algebra(QQ)
>>> specht_module_spanning_set([(Integer(0),Integer(0)), (Integer(1),Integer(1)),
↪(Integer(2),Integer(1))], SGA)
(() - (2,3), -(1,2) + (1,3,2), (1,2,3) - (1,3),
 -() + (2,3), -(1,2,3) + (1,3), (1,2) - (1,3,2))
```

sage.combinat.specht_module.**tabloid_gram_matrix**(*la*, *base_ring*)

Compute the Gram matrix of the bilinear form of a Specht module pulled back from the tabloid module.

For the module spanned by all tabloids, we define an bilinear form by having the tabloids be an orthonormal basis. We then pull this bilinear form back across the natural injection of the Specht module into the tabloid module.

EXAMPLES:

```
sage: from sage.combinat.specht_module import tabloid_gram_matrix
sage: tabloid_gram_matrix([3,2], GF(5))
[4 2 2 1 4]
[2 4 1 2 1]
[2 1 4 2 1]
[1 2 2 4 2]
[4 1 1 2 4]
```

```
>>> from sage.all import *
>>> from sage.combinat.specht_module import tabloid_gram_matrix
>>> tabloid_gram_matrix([Integer(3),Integer(2)], GF(Integer(5)))
[4 2 2 1 4]
[2 4 1 2 1]
[2 1 4 2 1]
[1 2 2 4 2]
[4 1 1 2 4]
```

5.1.335 Subsets

The set of subsets of a finite set. The set can be given as a list or a `Set` or else as an integer n which encodes the set $\{1, 2, ..., n\}$. See *Subsets* for more information and examples.

AUTHORS:

- Mike Hansen: initial version
- Florent Hivert (2009/02/06): doc improvements + new methods

class sage.combinat.subset.**SubMultiset_s**(*s*)

Bases: `Parent`

The combinatorial class of the sub multisets of `s`.

EXAMPLES:

```
sage: S = Subsets([1,2,2,3], submultiset=True)
sage: S.cardinality()
12
sage: S.list()
[[],
 [1],
 [2],
 [3],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
 [1, 2, 2],
 [1, 2, 3],
 [2, 2, 3],
 [1, 2, 2, 3]]
sage: S.first()
[]
sage: S.last()
[1, 2, 2, 3]
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(1),Integer(2),Integer(2),Integer(3)], submultiset=True)
>>> S.cardinality()
12
>>> S.list()
[[],
 [1],
```

(continues on next page)

(continued from previous page)

```
 [2],
 [3],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
 [1, 2, 2],
 [1, 2, 3],
 [2, 2, 3],
 [1, 2, 2, 3]]
>>> S.first()
[]
>>> S.last()
[1, 2, 2, 3]
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: S = Subsets([1,1,2,3],submultiset=True)
sage: S.cardinality()
12
sage: len(S.list())
12

sage: S = Subsets([1,1,2,2,3],submultiset=True)
sage: S.cardinality()
18
sage: len(S.list())
18

sage: S = Subsets([1,1,1,2,2,3],submultiset=True)
sage: S.cardinality()
24
sage: len(S.list())
24
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(1),Integer(1),Integer(2),Integer(3)],
↪submultiset=True)
>>> S.cardinality()
12
>>> len(S.list())
12

>>> S = Subsets([Integer(1),Integer(1),Integer(2),Integer(2),Integer(3)],
↪submultiset=True)
>>> S.cardinality()
18
>>> len(S.list())
18
```

(continues on next page)

(continued from previous page)

```
>>> S = Subsets([Integer(1),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3)],submultiset=True)
>>> S.cardinality()
24
>>> len(S.list())
24
```

element_class

alias of `list`

generating_serie(*variable='x'*)

Return the polynomial associated to the counting of the elements of `self` weighted by the number of element they contain.

EXAMPLES:

```
sage: Subsets([1,1],submultiset=True).generating_serie()
x^2 + x + 1
sage: Subsets([1,1,2,3],submultiset=True).generating_serie()
x^4 + 3*x^3 + 4*x^2 + 3*x + 1
sage: Subsets([1,1,1,2,2,3,3,4],submultiset=True).generating_serie()
x^8 + 4*x^7 + 9*x^6 + 14*x^5 + 16*x^4 + 14*x^3 + 9*x^2 + 4*x + 1

sage: S = Subsets([1,1,1,2,2,3,3,4],submultiset=True)
sage: S.cardinality()
72
sage: sum(S.generating_serie())
72
```

```
>>> from sage.all import *
>>> Subsets([Integer(1),Integer(1)],submultiset=True).generating_serie()
x^2 + x + 1
>>> Subsets([Integer(1),Integer(1),Integer(2),Integer(3)],submultiset=True).
↪generating_serie()
x^4 + 3*x^3 + 4*x^2 + 3*x + 1
>>> Subsets([Integer(1),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3),Integer(4)],submultiset=True).generating_serie()
x^8 + 4*x^7 + 9*x^6 + 14*x^5 + 16*x^4 + 14*x^3 + 9*x^2 + 4*x + 1

>>> S = Subsets([Integer(1),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3),Integer(4)],submultiset=True)
>>> S.cardinality()
72
>>> sum(S.generating_serie())
72
```

random_element()

Return a random element of `self` with uniform law.

EXAMPLES:

```
sage: S = Subsets([1,1,2,3], submultiset=True)
sage: s = S.random_element()
sage: s in S
True
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(1),Integer(1),Integer(2),Integer(3)],␣
↪submultiset=True)
>>> s = S.random_element()
>>> s in S
True
```

class sage.combinat.subset.**SubMultiset_sk**(*s*, *k*)

Bases: *SubMultiset_s*

The combinatorial class of the subsets of size k of a multiset s. Note that each subset is represented by a list of the elements rather than a set since we can have multiplicities (no multiset data structure yet in sage).

EXAMPLES:

```
sage: S = Subsets([1,2,3,3],2,submultiset=True)
sage: S._k
2
sage: S.cardinality()
4
sage: S.first()
[1, 2]
sage: S.last()
[3, 3]
sage: [sub for sub in S]
[[1, 2], [1, 3], [2, 3], [3, 3]]
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(1),Integer(2),Integer(3),Integer(3)],Integer(2),
↪submultiset=True)
>>> S._k
2
>>> S.cardinality()
4
>>> S.first()
[1, 2]
>>> S.last()
[3, 3]
>>> [sub for sub in S]
[[1, 2], [1, 3], [2, 3], [3, 3]]
```

cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: S = Subsets([1,2,2,3,3,3],4,submultiset=True)
sage: S.cardinality()
5
```

(continues on next page)

(continued from previous page)

```
sage: len(list(S))
5

sage: S = Subsets([1,2,2,3,3,3],3,submultiset=True)
sage: S.cardinality()
6
sage: len(list(S))
6
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(1),Integer(2),Integer(2),Integer(3),Integer(3),
↪Integer(3)],Integer(4),submultiset=True)
>>> S.cardinality()
5
>>> len(list(S))
5

>>> S = Subsets([Integer(1),Integer(2),Integer(2),Integer(3),Integer(3),
↪Integer(3)],Integer(3),submultiset=True)
>>> S.cardinality()
6
>>> len(list(S))
6
```

generating_serie(*variable='x'*)

Return the polynomial associated to the counting of the elements of `self` weighted by the number of elements they contains

EXAMPLES:

```
sage: x = ZZ['x'].gen()
sage: l = [1,1,1,1,2,2,3]
sage: for k in range(len(l)):
....:     S = Subsets(l,k,submultiset=True)
....:     print(S.generating_serie('x') == S.cardinality()*x**k)
True
True
True
True
True
True
True
```

```
>>> from sage.all import *
>>> x = ZZ['x'].gen()
>>> l = [Integer(1),Integer(1),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3)]
>>> for k in range(len(l)):
...     S = Subsets(l,k,submultiset=True)
...     print(S.generating_serie('x') == S.cardinality()*x**k)
True
True
```

(continues on next page)

(continued from previous page)

```
True
True
True
True
True
```

random_element()

Return a random submultiset of given length.

EXAMPLES:

```
sage: s = Subsets(7,3).random_element()
sage: s in Subsets(7,3)
True

sage: s = Subsets(7,5).random_element()
sage: s in Subsets(7,5)
True
```

```
>>> from sage.all import *
>>> s = Subsets(Integer(7),Integer(3)).random_element()
>>> s in Subsets(Integer(7),Integer(3))
True

>>> s = Subsets(Integer(7),Integer(5)).random_element()
>>> s in Subsets(Integer(7),Integer(5))
True
```

sage.combinat.subset.**Subsets**(*s*, *k=None*, *submultiset=False*)

Return the combinatorial class of the subsets of the finite set `s`. The set can be given as a list, Set or any iterable convertible to a set. Alternatively, a nonnegative integer n can be provided in place of `s`; in this case, the result is the combinatorial class of the subsets of the set $\{1, 2, \dots, n\}$ (i.e. of the Sage `range(1,n+1)`).

A second optional parameter `k` can be given. In this case, `Subsets` returns the combinatorial class of subsets of `s` of size `k`.

> **Warning**
>
> The subsets are returned as Sets. Do not assume that these Sets are ordered; they often are not! (E.g., `Subsets(10).list()[619]` returns `{10, 4, 5, 6, 7}` on my system.) See *SubsetsSorted* for a similar class which returns the subsets as sorted tuples.

Finally the option `submultiset` allows one to deal with sets with repeated elements, usually called multisets. The method then returns the class of all multisets in which every element is contained at most as often as it is contained in `s`. These multisets are encoded as lists.

EXAMPLES:

```
sage: S = Subsets([1, 2, 3]); S
Subsets of {1, 2, 3}
sage: S.cardinality()
8
```

(continues on next page)

(continued from previous page)

```
sage: S.first()
{}
sage: S.last()
{1, 2, 3}
sage: S.random_element() in S
True
sage: S.list()
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(1), Integer(2), Integer(3)]); S
Subsets of {1, 2, 3}
>>> S.cardinality()
8
>>> S.first()
{}
>>> S.last()
{1, 2, 3}
>>> S.random_element() in S
True
>>> S.list()
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

Here is the same example where the set is given as an integer:

```
sage: S = Subsets(3)
sage: S.list()
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

```
>>> from sage.all import *
>>> S = Subsets(Integer(3))
>>> S.list()
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

We demonstrate various the effect of the various options:

```
sage: S = Subsets(3, 2); S
Subsets of {1, 2, 3} of size 2
sage: S.list()
[{1, 2}, {1, 3}, {2, 3}]

sage: S = Subsets([1, 2, 2], submultiset=True); S
SubMultiset of [1, 2, 2]
sage: S.list()
[[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]

sage: S = Subsets([1, 2, 2, 3], 3, submultiset=True); S
SubMultiset of [1, 2, 2, 3] of size 3
sage: S.list()
[[1, 2, 2], [1, 2, 3], [2, 2, 3]]

sage: S = Subsets(['a','b','a','b'], 2, submultiset=True); S.list()
```

(continues on next page)

(continued from previous page)

```
[['a', 'a'], ['a', 'b'], ['b', 'b']]
```

```
>>> from sage.all import *
>>> S = Subsets(Integer(3), Integer(2)); S
Subsets of {1, 2, 3} of size 2
>>> S.list()
[{1, 2}, {1, 3}, {2, 3}]

>>> S = Subsets([Integer(1), Integer(2), Integer(2)], submultiset=True); S
SubMultiset of [1, 2, 2]
>>> S.list()
[[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]

>>> S = Subsets([Integer(1), Integer(2), Integer(2), Integer(3)], Integer(3),␣
→submultiset=True); S
SubMultiset of [1, 2, 2, 3] of size 3
>>> S.list()
[[1, 2, 2], [1, 2, 3], [2, 2, 3]]

>>> S = Subsets(['a','b','a','b'], Integer(2), submultiset=True); S.list()
[['a', 'a'], ['a', 'b'], ['b', 'b']]
```

And it is possible to play with subsets of subsets:

```
sage: S = Subsets(3)
sage: S2 = Subsets(S); S2
Subsets of Subsets of {1, 2, 3}
sage: S2.cardinality()
256
sage: it = iter(S2)
sage: [next(it) for _ in range(8)]
[{}, {{}}, {{1}}, {{2}}, {{3}}, {{1, 2}},  {{1, 3}}, {{2, 3}}]
sage: S2.random_element()     # random
{{2}, {1, 2, 3}, {}}
sage: [S2.unrank(k) for k in range(256)] == S2.list()
True

sage: S3 = Subsets(S2)
sage: S3.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
sage: S3.unrank(14123091480)  # random
{{{2}, {1, 2, 3}, {1, 2}, {3}, {}},
 {{1, 2, 3}, {2}, {1}, {1, 3}},
 {{}, {2}, {2, 3}, {1, 2}},
 {{}, {2}, {1, 2, 3}, {1, 2}},
 {},
 {{}, {1}, {1, 2, 3}}}

sage: T = Subsets(S2, 10)
sage: T.cardinality()
278826214642518400
sage: T.unrank(1441231049)  # random
```

(continues on next page)

(continued from previous page)

```
{{{1, 2, 3}, {2}, {2, 3}}, {{3}, {1, 3}, ..., {3}, {1}, {}, {1, 3}}}
```

```
>>> from sage.all import *
>>> S = Subsets(Integer(3))
>>> S2 = Subsets(S); S2
Subsets of Subsets of {1, 2, 3}
>>> S2.cardinality()
256
>>> it = iter(S2)
>>> [next(it) for _ in range(Integer(8))]
[{}, {{}}, {{1}}, {{2}}, {{3}}, {{1, 2}},  {{1, 3}}, {{2, 3}}]
>>> S2.random_element()     # random
{{2}, {1, 2, 3}, {}}
>>> [S2.unrank(k) for k in range(Integer(256))] == S2.list()
True

>>> S3 = Subsets(S2)
>>> S3.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
>>> S3.unrank(Integer(14123091480))  # random
{{{2}, {1, 2, 3}, {1, 2}, {3}, {}},
 {{1, 2, 3}, {2}, {1}, {1, 3}},
 {{}, {2}, {2, 3}, {1, 2}},
 {{}, {2}, {1, 2, 3}, {1, 2}},
 {},
 {{}, {1}, {1, 2, 3}}}

>>> T = Subsets(S2, Integer(10))
>>> T.cardinality()
278826214642518400
>>> T.unrank(Integer(1441231049))  # random
{{{1, 2, 3}, {2}, {2, 3}}, {{3}, {1, 3}, ..., {3}, {1}, {}, {1, 3}}}
```

class sage.combinat.subset.**SubsetsSorted**(s)

Bases: *Subsets_s*

Lightweight class of all subsets of some set S, with each subset being encoded as a sorted tuple.

Used to model indices of algebras given by subsets (so we don't have to explicitly build all 2^n subsets in memory). For example, CliffordAlgebra.

element_class

alias of tuple

first()

Return the first element of self.

EXAMPLES:

```
sage: from sage.combinat.subset import SubsetsSorted
sage: S = SubsetsSorted(range(3))
sage: S.first()
()
```

```
>>> from sage.all import *
>>> from sage.combinat.subset import SubsetsSorted
>>> S = SubsetsSorted(range(Integer(3)))
>>> S.first()
()
```

last()

Return the last element of `self`.

EXAMPLES:

```
sage: from sage.combinat.subset import SubsetsSorted
sage: S = SubsetsSorted(range(3))
sage: S.last()
(0, 1, 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.subset import SubsetsSorted
>>> S = SubsetsSorted(range(Integer(3)))
>>> S.last()
(0, 1, 2)
```

random_element()

Return a random element of `self`.

EXAMPLES:

```
sage: from sage.combinat.subset import SubsetsSorted
sage: S = SubsetsSorted(range(3))
sage: isinstance(S.random_element(), tuple)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.subset import SubsetsSorted
>>> S = SubsetsSorted(range(Integer(3)))
>>> isinstance(S.random_element(), tuple)
True
```

unrank(*r*)

Return the subset which has rank `r`.

EXAMPLES:

```
sage: from sage.combinat.subset import SubsetsSorted
sage: S = SubsetsSorted(range(3))
sage: S.unrank(4)
(0, 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.subset import SubsetsSorted
>>> S = SubsetsSorted(range(Integer(3)))
>>> S.unrank(Integer(4))
(0, 1)
```

class sage.combinat.subset.**Subsets_s**(*s*)

Bases: Parent

Subsets of a given set.

EXAMPLES:

```
sage: S = Subsets(4); S
Subsets of {1, 2, 3, 4}
sage: S.cardinality()
16
sage: Subsets(4).list()
[{}, {1}, {2}, {3}, {4},
 {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4},
 {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4},
 {1, 2, 3, 4}]

sage: S = Subsets(Subsets(Subsets(GF(3)))); S
Subsets of Subsets of Subsets of Finite Field of size 3
sage: S.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
sage: S.unrank(3149254230)  # random
{{{1}, {0, 2}}, {{0, 1, 2}, {0, 1}, {1}, {1, 2}},
 {{2}, {1, 2}, {0, 1, 2}, {0, 2}, {1}, {}},
 {{1, 2}, {0}},
 {{0, 1, 2}, {0, 1}, {0, 2}, {1, 2}}}
```

```
>>> from sage.all import *
>>> S = Subsets(Integer(4)); S
Subsets of {1, 2, 3, 4}
>>> S.cardinality()
16
>>> Subsets(Integer(4)).list()
[{}, {1}, {2}, {3}, {4},
 {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4},
 {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4},
 {1, 2, 3, 4}]

>>> S = Subsets(Subsets(Subsets(GF(Integer(3))))); S
Subsets of Subsets of Subsets of Finite Field of size 3
>>> S.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
>>> S.unrank(Integer(3149254230))  # random
{{{1}, {0, 2}}, {{0, 1, 2}, {0, 1}, {1}, {1, 2}},
 {{2}, {1, 2}, {0, 1, 2}, {0, 2}, {1}, {}},
 {{1, 2}, {0}},
 {{0, 1, 2}, {0, 1}, {0, 2}, {1, 2}}}
```

cardinality()

Return the number of subsets of the set `s`.

This is given by $2^{|s|}$.

EXAMPLES:

```
sage: Subsets(Set([1,2,3])).cardinality()
8
sage: Subsets([1,2,3,3]).cardinality()
8
sage: Subsets(3).cardinality()
8
```

```
>>> from sage.all import *
>>> Subsets(Set([Integer(1),Integer(2),Integer(3)])).cardinality()
8
>>> Subsets([Integer(1),Integer(2),Integer(3),Integer(3)]).cardinality()
8
>>> Subsets(Integer(3)).cardinality()
8
```

element_class

alias of `Set_object_enumerated`

first()

Return the first subset of `s`. Since we aren't restricted to subsets of a certain size, this is always the empty set.

EXAMPLES:

```
sage: Subsets([1,2,3]).first()
{}
sage: Subsets(3).first()
{}
```

```
>>> from sage.all import *
>>> Subsets([Integer(1),Integer(2),Integer(3)]).first()
{}
>>> Subsets(Integer(3)).first()
{}
```

last()

Return the last subset of `s`. Since we aren't restricted to subsets of a certain size, this is always the set `s` itself.

EXAMPLES:

```
sage: Subsets([1,2,3]).last()
{1, 2, 3}
sage: Subsets(3).last()
{1, 2, 3}
```

```
>>> from sage.all import *
>>> Subsets([Integer(1),Integer(2),Integer(3)]).last()
{1, 2, 3}
>>> Subsets(Integer(3)).last()
{1, 2, 3}
```

lattice()

Return the lattice of subsets ordered by containment.

EXAMPLES:

```
sage: X = Subsets([7,8,9])
sage: X.lattice()                                                        #␣
↪needs sage.combinat sage.graphs
Finite lattice containing 8 elements
sage: Y = Subsets(0)
sage: Y.lattice()                                                        #␣
↪needs sage.combinat sage.graphs
Finite lattice containing 1 elements
```

```
>>> from sage.all import *
>>> X = Subsets([Integer(7),Integer(8),Integer(9)])
>>> X.lattice()                                                          #␣
↪needs sage.combinat sage.graphs
Finite lattice containing 8 elements
>>> Y = Subsets(Integer(0))
>>> Y.lattice()                                                          #␣
↪needs sage.combinat sage.graphs
Finite lattice containing 1 elements
```

random_element()

Return a random element of the class of subsets of s (in other words, a random subset of s).

EXAMPLES:

```
sage: Subsets(3).random_element()           # random
{2}
sage: Subsets([4,5,6]).random_element()     # random
{5}

sage: S = Subsets(Subsets(Subsets([0,1,2])))
sage: S.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
sage: s = S.random_element()
sage: s     # random
{{{1, 2}, {2}, {0}, {1}}, {{1, 2}, {0, 1, 2}, {0, 2}, {0}, {0, 1}}, ..., {{1,␣
↪2}, {2}, {1}}, {{2}, {0, 2}, {}, {1}}}
sage: s in S
True
```

```
>>> from sage.all import *
>>> Subsets(Integer(3)).random_element()           # random
{2}
>>> Subsets([Integer(4),Integer(5),Integer(6)]).random_element()     # random
{5}

>>> S = Subsets(Subsets(Subsets([Integer(0),Integer(1),Integer(2)])))
>>> S.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
>>> s = S.random_element()
>>> s     # random
{{{1, 2}, {2}, {0}, {1}}, {{1, 2}, {0, 1, 2}, {0, 2}, {0}, {0, 1}}, ..., {{1,␣
↪2}, {2}, {1}}, {{2}, {0, 2}, {}, {1}}}
>>> s in S
```

(continues on next page)

(continued from previous page)

```
True
```

rank(*sub*)

Return the rank of `sub` as a subset of `s`.

EXAMPLES:

```
sage: Subsets(3).rank([])
0
sage: Subsets(3).rank([1,2])
4
sage: Subsets(3).rank([1,2,3])
7
sage: Subsets(3).rank([2,3,4])
Traceback (most recent call last):
...
ValueError: {2, 3, 4} is not a subset of {1, 2, 3}
```

```
>>> from sage.all import *
>>> Subsets(Integer(3)).rank([])
0
>>> Subsets(Integer(3)).rank([Integer(1),Integer(2)])
4
>>> Subsets(Integer(3)).rank([Integer(1),Integer(2),Integer(3)])
7
>>> Subsets(Integer(3)).rank([Integer(2),Integer(3),Integer(4)])
Traceback (most recent call last):
...
ValueError: {2, 3, 4} is not a subset of {1, 2, 3}
```

underlying_set()

Return the set of elements.

EXAMPLES:

```
sage: Subsets(GF(13)).underlying_set()
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

```
>>> from sage.all import *
>>> Subsets(GF(Integer(13))).underlying_set()
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

unrank(*r*)

Return the subset of `s` that has rank `k`.

EXAMPLES:

```
sage: Subsets(3).unrank(0)
{}
sage: Subsets([2,4,5]).unrank(1)
{2}
sage: Subsets([1,2,3]).unrank(257)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
IndexError: index out of range
```

```
>>> from sage.all import *
>>> Subsets(Integer(3)).unrank(Integer(0))
{}
>>> Subsets([Integer(2),Integer(4),Integer(5)]).unrank(Integer(1))
{2}
>>> Subsets([Integer(1),Integer(2),Integer(3)]).unrank(Integer(257))
Traceback (most recent call last):
...
IndexError: index out of range
```

class sage.combinat.subset.**Subsets_sk**(*s*, *k*)

Bases: *Subsets_s*

Subsets of fixed size of a set.

EXAMPLES:

```
sage: S = Subsets([0,1,2,5,7], 3); S
Subsets of {0, 1, 2, 5, 7} of size 3
sage: S.cardinality()
10
sage: S.first(), S.last()
({0, 1, 2}, {2, 5, 7})
sage: S.random_element()  # random
{0, 5, 7}
sage: S([0,2,7])
{0, 2, 7}
sage: S([0,3,5])
Traceback (most recent call last):
...
ValueError: {0, 3, 5} not in Subsets of {0, 1, 2, 5, 7} of size 3
sage: S([0])
Traceback (most recent call last):
...
ValueError: {0} not in Subsets of {0, 1, 2, 5, 7} of size 3
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(0),Integer(1),Integer(2),Integer(5),Integer(7)],␣
↪Integer(3)); S
Subsets of {0, 1, 2, 5, 7} of size 3
>>> S.cardinality()
10
>>> S.first(), S.last()
({0, 1, 2}, {2, 5, 7})
>>> S.random_element()  # random
{0, 5, 7}
>>> S([Integer(0),Integer(2),Integer(7)])
{0, 2, 7}
>>> S([Integer(0),Integer(3),Integer(5)])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: {0, 3, 5} not in Subsets of {0, 1, 2, 5, 7} of size 3
>>> S([Integer(0)])
Traceback (most recent call last):
...
ValueError: {0} not in Subsets of {0, 1, 2, 5, 7} of size 3
```

an_element()

Return an example of subset.

EXAMPLES:

```
sage: Subsets(0,0).an_element()
{}
sage: Subsets(3,2).an_element()
{1, 3}
sage: Subsets([2,4,5],2).an_element()
{2, 5}
```

```
>>> from sage.all import *
>>> Subsets(Integer(0),Integer(0)).an_element()
{}
>>> Subsets(Integer(3),Integer(2)).an_element()
{1, 3}
>>> Subsets([Integer(2),Integer(4),Integer(5)],Integer(2)).an_element()
{2, 5}
```

cardinality()

EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).cardinality()
3
sage: Subsets([1,2,3,3], 2).cardinality()
3
sage: Subsets([1,2,3], 1).cardinality()
3
sage: Subsets([1,2,3], 3).cardinality()
1
sage: Subsets([1,2,3], 0).cardinality()
1
sage: Subsets([1,2,3], 4).cardinality()
0
sage: Subsets(3,2).cardinality()
3
sage: Subsets(3,4).cardinality()
0
```

```
>>> from sage.all import *
>>> Subsets(Set([Integer(1),Integer(2),Integer(3)]), Integer(2)).cardinality()
3
>>> Subsets([Integer(1),Integer(2),Integer(3),Integer(3)], Integer(2)).
```

(continues on next page)

(continued from previous page)

```
↪cardinality()
3
>>> Subsets([Integer(1),Integer(2),Integer(3)], Integer(1)).cardinality()
3
>>> Subsets([Integer(1),Integer(2),Integer(3)], Integer(3)).cardinality()
1
>>> Subsets([Integer(1),Integer(2),Integer(3)], Integer(0)).cardinality()
1
>>> Subsets([Integer(1),Integer(2),Integer(3)], Integer(4)).cardinality()
0
>>> Subsets(Integer(3),Integer(2)).cardinality()
3
>>> Subsets(Integer(3),Integer(4)).cardinality()
0
```

first()

Return the first subset of s of size k.

EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).first()
{1, 2}
sage: Subsets([1,2,3,3], 2).first()
{1, 2}
sage: Subsets(3,2).first()
{1, 2}
sage: Subsets(3,4).first()
Traceback (most recent call last):
...
EmptySetError
```

```
>>> from sage.all import *
>>> Subsets(Set([Integer(1),Integer(2),Integer(3)]), Integer(2)).first()
{1, 2}
>>> Subsets([Integer(1),Integer(2),Integer(3),Integer(3)], Integer(2)).first()
{1, 2}
>>> Subsets(Integer(3),Integer(2)).first()
{1, 2}
>>> Subsets(Integer(3),Integer(4)).first()
Traceback (most recent call last):
...
EmptySetError
```

last()

Return the last subset of s of size k.

EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).last()
{2, 3}
sage: Subsets([1,2,3,3], 2).last()
{2, 3}
sage: Subsets(3,2).last()
```

(continues on next page)

(continued from previous page)

```
{2, 3}
sage: Subsets(3,4).last()
Traceback (most recent call last):
...
EmptySetError
```

```
>>> from sage.all import *
>>> Subsets(Set([Integer(1),Integer(2),Integer(3)]), Integer(2)).last()
{2, 3}
>>> Subsets([Integer(1),Integer(2),Integer(3),Integer(3)], Integer(2)).last()
{2, 3}
>>> Subsets(Integer(3),Integer(2)).last()
{2, 3}
>>> Subsets(Integer(3),Integer(4)).last()
Traceback (most recent call last):
...
EmptySetError
```

random_element()

Return a random element of the class of subsets of `s` of size `k` (in other words, a random subset of `s` of size `k`).

EXAMPLES:

```
sage: s = Subsets(3, 2).random_element()
sage: s in Subsets(3, 2)
True

sage: Subsets(3,4).random_element()
Traceback (most recent call last):
...
EmptySetError
```

```
>>> from sage.all import *
>>> s = Subsets(Integer(3), Integer(2)).random_element()
>>> s in Subsets(Integer(3), Integer(2))
True

>>> Subsets(Integer(3),Integer(4)).random_element()
Traceback (most recent call last):
...
EmptySetError
```

rank(*sub*)

Return the rank of `sub` as a subset of `s` of size `k`.

EXAMPLES:

```
sage: Subsets(3,2).rank([1,2])
0
sage: Subsets([2,3,4],2).rank([3,4])
2
```

(continues on next page)

(continued from previous page)

```
sage: Subsets([2,3,4],2).rank([2])
Traceback (most recent call last):
...
ValueError: {2} is not a subset of length 2 of {2, 3, 4}
sage: Subsets([2,3,4],4).rank([2,3,4,5])
Traceback (most recent call last):
...
ValueError: {2, 3, 4, 5} is not a subset of length 4 of {2, 3, 4}
```

```
>>> from sage.all import *
>>> Subsets(Integer(3),Integer(2)).rank([Integer(1),Integer(2)])
0
>>> Subsets([Integer(2),Integer(3),Integer(4)],Integer(2)).rank([Integer(3),
↪Integer(4)])
2
>>> Subsets([Integer(2),Integer(3),Integer(4)],Integer(2)).rank([Integer(2)])
Traceback (most recent call last):
...
ValueError: {2} is not a subset of length 2 of {2, 3, 4}
>>> Subsets([Integer(2),Integer(3),Integer(4)],Integer(4)).rank([Integer(2),
↪Integer(3),Integer(4),Integer(5)])
Traceback (most recent call last):
...
ValueError: {2, 3, 4, 5} is not a subset of length 4 of {2, 3, 4}
```

unrank(*r*)

Return the subset of `s` of size `k` that has rank `r`.

EXAMPLES:

```
sage: Subsets(3,2).unrank(0)
{1, 2}
sage: Subsets([2,4,5],2).unrank(0)
{2, 4}
sage: Subsets([1,2,8],3).unrank(42)
Traceback (most recent call last):
...
IndexError: index out of range
```

```
>>> from sage.all import *
>>> Subsets(Integer(3),Integer(2)).unrank(Integer(0))
{1, 2}
>>> Subsets([Integer(2),Integer(4),Integer(5)],Integer(2)).unrank(Integer(0))
{2, 4}
>>> Subsets([Integer(1),Integer(2),Integer(8)],Integer(3)).unrank(Integer(42))
Traceback (most recent call last):
...
IndexError: index out of range
```

`sage.combinat.subset.`**dict_to_list**(*d*)

Return a list whose elements are the elements of `i` of `d` repeated with multiplicity `d[i]`.

EXAMPLES:

```
sage: from sage.combinat.subset import dict_to_list
sage: dict_to_list({'a':1, 'b':3})
['a', 'b', 'b', 'b']
```

```
>>> from sage.all import *
>>> from sage.combinat.subset import dict_to_list
>>> dict_to_list({'a':Integer(1), 'b':Integer(3)})
['a', 'b', 'b', 'b']
```

sage.combinat.subset.**list_to_dict**(*l*)

Return a dictionary of multiplicities and the list of its keys.

INPUT:

- l – list with possibly repeated elements

The keys are the elements of l (in the same order in which they appear) and values are the multiplicities of each element in l.

EXAMPLES:

```
sage: from sage.combinat.subset import list_to_dict
sage: list_to_dict(['a', 'b', 'b', 'b'])
({'a': 1, 'b': 3}, ['a', 'b'])
```

```
>>> from sage.all import *
>>> from sage.combinat.subset import list_to_dict
>>> list_to_dict(['a', 'b', 'b', 'b'])
({'a': 1, 'b': 3}, ['a', 'b'])
```

sage.combinat.subset.**powerset**(*X*)

Iterator over the *list* of all subsets of the iterable X, in no particular order. Each list appears exactly once, up to order.

INPUT:

- X – an iterable

OUTPUT: iterator of lists

EXAMPLES:

```
sage: list(powerset([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
sage: [z for z in powerset([0,[1,2]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

```
>>> from sage.all import *
>>> list(powerset([Integer(1),Integer(2),Integer(3)]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
>>> [z for z in powerset([Integer(0),[Integer(1),Integer(2)]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

Iterating over the power set of an infinite set is also allowed:

```
sage: i = 0
sage: L = []
sage: for x in powerset(ZZ):
....:     if i > 10:
....:         break
....:     else:
....:         i += 1
....:     L.append(x)
sage: print(" ".join(str(x) for x in L))
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]
```

```
>>> from sage.all import *
>>> i = Integer(0)
>>> L = []
>>> for x in powerset(ZZ):
...     if i > Integer(10):
...         break
...     else:
...         i += Integer(1)
...     L.append(x)
>>> print(" ".join(str(x) for x in L))
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]
```

You may also use subsets as an alias for powerset:

```
sage: subsets([1,2,3])
<generator object ...powerset at 0x...>
sage: list(subsets([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of
sets must be hashable and many structures on which one wants the
powerset consist of non-hashable objects.
```

```
>>> from sage.all import *
>>> subsets([Integer(1),Integer(2),Integer(3)])
<generator object ...powerset at 0x...>
>>> list(subsets([Integer(1),Integer(2),Integer(3)]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of
sets must be hashable and many structures on which one wants the
powerset consist of non-hashable objects.
```

AUTHORS:

- William Stein
- Nils Bruin (2006-12-19): rewrite to work for not-necessarily finite objects X.

`sage.combinat.subset.`**`subsets`**`(X)`

Iterator over the *list* of all subsets of the iterable `X`, in no particular order. Each list appears exactly once, up to order.

INPUT:

- x – an iterable

OUTPUT: iterator of lists

EXAMPLES:

```
sage: list(powerset([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
sage: [z for z in powerset([0,[1,2]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

```
>>> from sage.all import *
>>> list(powerset([Integer(1),Integer(2),Integer(3)]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
>>> [z for z in powerset([Integer(0),[Integer(1),Integer(2)]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

Iterating over the power set of an infinite set is also allowed:

```
sage: i = 0
sage: L = []
sage: for x in powerset(ZZ):
....:     if i > 10:
....:         break
....:     else:
....:         i += 1
....:     L.append(x)
sage: print(" ".join(str(x) for x in L))
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]
```

```
>>> from sage.all import *
>>> i = Integer(0)
>>> L = []
>>> for x in powerset(ZZ):
...     if i > Integer(10):
...         break
...     else:
...         i += Integer(1)
...     L.append(x)
>>> print(" ".join(str(x) for x in L))
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]
```

You may also use subsets as an alias for powerset:

```
sage: subsets([1,2,3])
<generator object ...powerset at 0x...>
sage: list(subsets([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of
sets must be hashable and many structures on which one wants the
powerset consist of non-hashable objects.
```

```
>>> from sage.all import *
>>> subsets([Integer(1),Integer(2),Integer(3)])
<generator object ...powerset at 0x...>
>>> list(subsets([Integer(1),Integer(2),Integer(3)]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of
sets must be hashable and many structures on which one wants the
powerset consist of non-hashable objects.
```

AUTHORS:

- William Stein
- Nils Bruin (2006-12-19): rewrite to work for not-necessarily finite objects X.

sage.combinat.subset.**uniq**(*L*)

Iterate over the elements of `L`, yielding every element at most once: keep only the first occurrence of any item.

The items must be hashable.

INPUT:

- `L` – iterable

EXAMPLES:

```
sage: L = [1, 1, 8, -5, 3, -5, 'a', 'x', 'a']
sage: it = uniq(L); it
<generator object uniq at ...>
sage: list(it)
[1, 8, -5, 3, 'a', 'x']
```

```
>>> from sage.all import *
>>> L = [Integer(1), Integer(1), Integer(8), -Integer(5), Integer(3), -Integer(5),
↪ 'a', 'x', 'a']
>>> it = uniq(L); it
<generator object uniq at ...>
>>> list(it)
[1, 8, -5, 3, 'a', 'x']
```

5.1.336 Subsets satisfying a hereditary property

sage.combinat.subsets_hereditary.**subsets_with_hereditary_property**(*f*, *X*, *max_obstruction_size=None*, *ncpus=1*)

Return all subsets S of X such that $f(S)$ is true.

The boolean function f must be decreasing, i.e. $f(S) \Rightarrow f(S')$ if $S' \subseteq S$.

This function is implemented to call f as few times as possible. More precisely, f will be called on all sets S such that $f(S)$ is true, as well as on all inclusionwise minimal sets S such that $f(S)$ is false.

The problem that this function answers is also known as the learning problem on monotone boolean functions, or as computing the set of winning coalitions in a simple game.

INPUT:

- f – boolean function which takes as input a list of elements from X
- X – list/iterable
- max_obstruction_size – integer; if you know that there is a k such that $f(S)$ is true if and only if $f(S')$ is true for all $S' \subseteq S$ with $S' \leq k$, set max_obstruction_size=k. It may dramatically decrease the number of calls to f. Set to None by default, meaning $k = |X|$.
- ncpus – number of cpus to use for this computation. Note that changing the value from 1 (default) to anything different *enables* parallel computations which can have a cost by itself, so it is not necessarily a good move. In some cases, however, it is a *great* move. Set to None to automatically detect and use the maximum number of cpus available.

> **Note**
>
> Parallel computations are performed through the parallel() decorator. See its documentation for more information, in particular with respect to the memory context.

EXAMPLES:

Sets whose elements all have the same remainder mod 2:

```
sage: from sage.combinat.subsets_hereditary import subsets_with_hereditary_
↪property
sage: def f(x):
....:     return (not x) or all(xx % 2 == x[0] % 2 for xx in x)
sage: list(subsets_with_hereditary_property(f, range(4)))
[[], [0], [1], [2], [3], [0, 2], [1, 3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.subsets_hereditary import subsets_with_hereditary_property
>>> def f(x):
...     return (not x) or all(xx % Integer(2) == x[Integer(0)] % Integer(2) for␣
↪xx in x)
>>> list(subsets_with_hereditary_property(f, range(Integer(4))))
[[], [0], [1], [2], [3], [0, 2], [1, 3]]
```

Same, on two threads:

```
sage: sorted(subsets_with_hereditary_property(f, range(4), ncpus=2))
[[], [0], [0, 2], [1], [1, 3], [2], [3]]
```

```
>>> from sage.all import *
>>> sorted(subsets_with_hereditary_property(f, range(Integer(4)),␣
↪ncpus=Integer(2)))
[[], [0], [0, 2], [1], [1, 3], [2], [3]]
```

One can use this function to compute the independent sets of a graph. We know, however, that in this case the maximum obstructions are the edges, and have size 2. We can thus set max_obstruction_size=2, which reduces the number of calls to f from 91 to 56:

```
sage: # needs sage.graphs
sage: num_calls = 0
sage: g = graphs.PetersenGraph()
```

(continues on next page)

(continued from previous page)

```
sage: def is_independent_set(S):
....:     global num_calls
....:     num_calls += 1
....:     return g.subgraph(S).size() == 0
sage: l1 = list(subsets_with_hereditary_property(is_independent_set,
....:                                              g.vertices(sort=False)))
sage: num_calls
91
sage: num_calls = 0
sage: l2 = list(subsets_with_hereditary_property(is_independent_set,
....:                                              g.vertices(sort=False),
....:                                              max_obstruction_size=2))
sage: num_calls
56
sage: l1 == l2
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> num_calls = Integer(0)
>>> g = graphs.PetersenGraph()
>>> def is_independent_set(S):
...     global num_calls
...     num_calls += Integer(1)
...     return g.subgraph(S).size() == Integer(0)
>>> l1 = list(subsets_with_hereditary_property(is_independent_set,
...                                              g.vertices(sort=False)))
>>> num_calls
91
>>> num_calls = Integer(0)
>>> l2 = list(subsets_with_hereditary_property(is_independent_set,
...                                              g.vertices(sort=False),
...                                              max_obstruction_size=Integer(2)))
>>> num_calls
56
>>> l1 == l2
True
```

5.1.337 Subsets whose elements satisfy a predicate pairwise

class sage.combinat.subsets_pairwise.**PairwiseCompatibleSubsets**(*ambient*, *predicate*, *maximal=False*, *element_class=<class 'sage.sets.set.Set_object_enumerated'>*)

Bases: `RecursivelyEnumeratedSet_forest`

The set of all subsets of `ambient` whose elements satisfy `predicate` pairwise

INPUT:

- `ambient` – set (or iterable)
- `predicate` – a binary predicate

Assumptions: `predicate` is symmetric (`predicate(x,y) == predicate(y,x)`) and reflexive (`predicate(x,x) == True`).

Note

in fact, `predicate(x,x)` is never called.

Warning

The current name is suboptimal and is subject to change. Suggestions for a good name, and a good user entry point are welcome. Maybe `Subsets(..., independent = predicate)`.

EXAMPLES:

We construct the set of all subsets of $\{4,5,6,8,9\}$ whose elements are pairwise relatively prime:

```
sage: from sage.combinat.subsets_pairwise import PairwiseCompatibleSubsets
sage: def predicate(x, y): return gcd(x, y) == 1
sage: P = PairwiseCompatibleSubsets( [4,5,6,8,9], predicate); P
An enumerated set with a forest structure
sage: P.list()
[{}, {4}, {4, 5}, {9, 4, 5}, {9, 4}, {5}, {5, 6}, {8, 5}, {8, 9, 5}, {9, 5}, {6},
→{8}, {8, 9}, {9}]
sage: P.cardinality()
14
sage: P.category()
Category of finite enumerated sets
```

```
>>> from sage.all import *
>>> from sage.combinat.subsets_pairwise import PairwiseCompatibleSubsets
>>> def predicate(x, y): return gcd(x, y) == Integer(1)
>>> P = PairwiseCompatibleSubsets( [Integer(4),Integer(5),Integer(6),Integer(8),
→Integer(9)], predicate); P
An enumerated set with a forest structure
>>> P.list()
[{}, {4}, {4, 5}, {9, 4, 5}, {9, 4}, {5}, {5, 6}, {8, 5}, {8, 9, 5}, {9, 5}, {6},
→{8}, {8, 9}, {9}]
>>> P.cardinality()
14
>>> P.category()
Category of finite enumerated sets
```

Here we consider only those subsets which are maximal for inclusion (not yet implemented):

```
sage: P = PairwiseCompatibleSubsets( [4,5,6,8,9], predicate, maximal = True); P
An enumerated set with a forest structure
sage: P.list()              # todo: not implemented
[{9, 4, 5}, {5, 6}, {8, 9, 5}]
sage: P.cardinality()       # todo: not implemented
14
sage: P.category()
Category of finite enumerated sets
```

```
>>> from sage.all import *
>>> P = PairwiseCompatibleSubsets( [Integer(4),Integer(5),Integer(6),Integer(8),
↪Integer(9)], predicate, maximal = True); P
An enumerated set with a forest structure
>>> P.list()                    # todo: not implemented
[{9, 4, 5}, {5, 6}, {8, 9, 5}]
>>> P.cardinality()             # todo: not implemented
14
>>> P.category()
Category of finite enumerated sets
```

Algorithm

In the following, we order the elements of the ambient set by order of apparition. The elements of `self` are generated by organizing them in a search tree. Each node of this tree is of the form `(subset, rest)`, where:

- `subset` represents an element of `self`, represented by an increasing tuple
- `rest` is the set of all y's such that y appears after x in the ambient set and `predicate(x,y)` holds, represented by a decreasing tuple

The root of this tree is `( (), ambient )`. All the other elements are generated by recursive depth first search, which gives lexicographic order.

children(*subset_rest*)

Return the children of a node in the tree.

post_process(*subset_rest*)

5.1.338 Subwords

A subword of a word w is a word obtained by deleting the letters at some (non necessarily adjacent) positions in w. It is not to be confused with the notion of factor where one keeps adjacent positions in w. Sometimes it is useful to allow repeated uses of the same letter of w in a "generalized" subword. We call this a subword with repetitions.

For example:

- "bnjr" is a subword of the word "bonjour" but not a factor;
- "njo" is both a factor and a subword of the word "bonjour";
- "nr" is a subword of "bonjour";
- "rn" is not a subword of "bonjour";
- "nnu" is not a subword of "bonjour";
- "nnu" is a subword with repetitions of "bonjour";

A word can be given either as a string, as a list or as a tuple.

As repetition can occur in the initial word, in general subwords of a given word form an enumerated multiset rather than a set!

Todo

- implement subwords with repetitions

- implement the category of `EnumeratedMultiset` and inheritate from it when needed (i.e. the initial word has repeated letters)

AUTHORS:

- Mike Hansen: initial version
- Florent Hivert (2009/02/06): doc improvements + new methods + bug fixes

`sage.combinat.subword.`**`Subwords`**`(`*w*, *k=None*, *element_constructor=None*`)`

Return the set of subwords of `w`.

INPUT:

- `w` – a word (can be a list, a string, a tuple or a word)
- `k` – an optional integer to specify the length of subwords
- `element_constructor` – an optional function that will be used to build the subwords

EXAMPLES:

```
sage: S = Subwords(['a','b','c']); S
Subwords of ['a', 'b', 'c']
sage: S.first()
[]
sage: S.last()
['a', 'b', 'c']
sage: S.list()
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

```
>>> from sage.all import *
>>> S = Subwords(['a','b','c']); S
Subwords of ['a', 'b', 'c']
>>> S.first()
[]
>>> S.last()
['a', 'b', 'c']
>>> S.list()
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

The same example using string, tuple or a word:

```
sage: S = Subwords('abc'); S
Subwords of 'abc'
sage: S.list()
['', 'a', 'b', 'c', 'ab', 'ac', 'bc', 'abc']

sage: S = Subwords((1,2,3)); S
Subwords of (1, 2, 3)
sage: S.list()
[(), (1,), (2,), (3,), (1, 2), (1, 3), (2, 3), (1, 2, 3)]

sage: w = Word([1,2,3])
sage: S = Subwords(w); S
Subwords of word: 123
```

(continues on next page)

(continued from previous page)

```
sage: S.list()
[word: , word: 1, word: 2, word: 3, word: 12, word: 13, word: 23, word: 123]
```

```
>>> from sage.all import *
>>> S = Subwords('abc'); S
Subwords of 'abc'
>>> S.list()
['', 'a', 'b', 'c', 'ab', 'ac', 'bc', 'abc']

>>> S = Subwords((Integer(1),Integer(2),Integer(3))); S
Subwords of (1, 2, 3)
>>> S.list()
[(), (1,), (2,), (3,), (1, 2), (1, 3), (2, 3), (1, 2, 3)]

>>> w = Word([Integer(1),Integer(2),Integer(3)])
>>> S = Subwords(w); S
Subwords of word: 123
>>> S.list()
[word: , word: 1, word: 2, word: 3, word: 12, word: 13, word: 23, word: 123]
```

Using word with specified length:

```
sage: S = Subwords(['a','b','c'], 2); S
Subwords of ['a', 'b', 'c'] of length 2
sage: S.list()
[['a', 'b'], ['a', 'c'], ['b', 'c']]
```

```
>>> from sage.all import *
>>> S = Subwords(['a','b','c'], Integer(2)); S
Subwords of ['a', 'b', 'c'] of length 2
>>> S.list()
[['a', 'b'], ['a', 'c'], ['b', 'c']]
```

An example that uses the `element_constructor` argument:

```
sage: p = Permutation([3,2,1])
sage: Subwords(p, element_constructor=tuple).list()
[(), (3,), (2,), (1,), (3, 2), (3, 1), (2, 1), (3, 2, 1)]
sage: Subwords(p, 2, element_constructor=tuple).list()
[(3, 2), (3, 1), (2, 1)]
```

```
>>> from sage.all import *
>>> p = Permutation([Integer(3),Integer(2),Integer(1)])
>>> Subwords(p, element_constructor=tuple).list()
[(), (3,), (2,), (1,), (3, 2), (3, 1), (2, 1), (3, 2, 1)]
>>> Subwords(p, Integer(2), element_constructor=tuple).list()
[(3, 2), (3, 1), (2, 1)]
```

class sage.combinat.subword.**Subwords_w**(*w*, *element_constructor*)

Bases: `Parent`

Subwords of a given word.

cardinality()

EXAMPLES:

```
sage: Subwords([1,2,3]).cardinality()
8
```

```
>>> from sage.all import *
>>> Subwords([Integer(1),Integer(2),Integer(3)]).cardinality()
8
```

first()

EXAMPLES:

```
sage: Subwords([1,2,3]).first()
[]
sage: Subwords((1,2,3)).first()
()
sage: Subwords('123').first()
''
```

```
>>> from sage.all import *
>>> Subwords([Integer(1),Integer(2),Integer(3)]).first()
[]
>>> Subwords((Integer(1),Integer(2),Integer(3))).first()
()
>>> Subwords('123').first()
''
```

last()

EXAMPLES:

```
sage: Subwords([1,2,3]).last()
[1, 2, 3]
sage: Subwords((1,2,3)).last()
(1, 2, 3)
sage: Subwords('123').last()
'123'
```

```
>>> from sage.all import *
>>> Subwords([Integer(1),Integer(2),Integer(3)]).last()
[1, 2, 3]
>>> Subwords((Integer(1),Integer(2),Integer(3))).last()
(1, 2, 3)
>>> Subwords('123').last()
'123'
```

random_element()

Return a random subword with uniform law.

EXAMPLES:

```
sage: S1 = Subwords([1,2,3,2,1,3])
sage: S2 = Subwords([4,6,6,6,7,4,5,5])
```

(continues on next page)

(continued from previous page)

```
sage: for i in range(100):
....:     w = S1.random_element()
....:     if w in S2:
....:         assert not w
sage: for i in range(100):
....:     w = S2.random_element()
....:     if w in S1:
....:         assert not w
```

```
>>> from sage.all import *
>>> S1 = Subwords([Integer(1),Integer(2),Integer(3),Integer(2),Integer(1),
↪Integer(3)])
>>> S2 = Subwords([Integer(4),Integer(6),Integer(6),Integer(6),Integer(7),
↪Integer(4),Integer(5),Integer(5)])
>>> for i in range(Integer(100)):
...     w = S1.random_element()
...     if w in S2:
...         assert not w
>>> for i in range(Integer(100)):
...     w = S2.random_element()
...     if w in S1:
...         assert not w
```

class sage.combinat.subword.**Subwords_wk**(*w*, *k*, *element_constructor*)

Bases: *Subwords_w*

Subwords with fixed length of a given word.

cardinality()

Return the number of subwords of w of length k.

EXAMPLES:

```
sage: Subwords([1,2,3], 2).cardinality()
3
```

```
>>> from sage.all import *
>>> Subwords([Integer(1),Integer(2),Integer(3)], Integer(2)).cardinality()
3
```

first()

EXAMPLES:

```
sage: Subwords([1,2,3],2).first()
[1, 2]
sage: Subwords([1,2,3],0).first()
[]
sage: Subwords((1,2,3),2).first()
(1, 2)
sage: Subwords((1,2,3),0).first()
()
sage: Subwords('123',2).first()
'12'
```

(continues on next page)

(continued from previous page)

```
sage: Subwords('123',0).first()
''
```

```
>>> from sage.all import *
>>> Subwords([Integer(1),Integer(2),Integer(3)],Integer(2)).first()
[1, 2]
>>> Subwords([Integer(1),Integer(2),Integer(3)],Integer(0)).first()
[]
>>> Subwords((Integer(1),Integer(2),Integer(3)),Integer(2)).first()
(1, 2)
>>> Subwords((Integer(1),Integer(2),Integer(3)),Integer(0)).first()
()
>>> Subwords('123',Integer(2)).first()
'12'
>>> Subwords('123',Integer(0)).first()
''
```

last()

EXAMPLES:

```
sage: Subwords([1,2,3],2).last()
[2, 3]
sage: Subwords([1,2,3],0).last()
[]
sage: Subwords((1,2,3),2).last()
(2, 3)
sage: Subwords((1,2,3),0).last()
()
sage: Subwords('123',2).last()
'23'
sage: Subwords('123',0).last()
''
```

```
>>> from sage.all import *
>>> Subwords([Integer(1),Integer(2),Integer(3)],Integer(2)).last()
[2, 3]
>>> Subwords([Integer(1),Integer(2),Integer(3)],Integer(0)).last()
[]
>>> Subwords((Integer(1),Integer(2),Integer(3)),Integer(2)).last()
(2, 3)
>>> Subwords((Integer(1),Integer(2),Integer(3)),Integer(0)).last()
()
>>> Subwords('123',Integer(2)).last()
'23'
>>> Subwords('123',Integer(0)).last()
''
```

random_element()

Return a random subword of given length with uniform law.

EXAMPLES:

```
sage: S1 = Subwords([1,2,3,2,1],3)
sage: S2 = Subwords([4,4,5,5,4,5,4,4],3)
sage: for i in range(100):
....:   w = S1.random_element()
....:   if w in S2:
....:       assert not w
sage: for i in range(100):
....:   w = S2.random_element()
....:   if w in S1:
....:       assert not w
```

```
>>> from sage.all import *
>>> S1 = Subwords([Integer(1),Integer(2),Integer(3),Integer(2),Integer(1)],
↪Integer(3))
>>> S2 = Subwords([Integer(4),Integer(4),Integer(5),Integer(5),Integer(4),
↪Integer(5),Integer(4),Integer(4)],Integer(3))
>>> for i in range(Integer(100)):
...   w = S1.random_element()
...   if w in S2:
...       assert not w
>>> for i in range(Integer(100)):
...   w = S2.random_element()
...   if w in S1:
...       assert not w
```

sage.combinat.subword.**smallest_positions**(*word*, *subword*, *pos=0*)

Return the smallest positions for which `subword` appears as a subword of `word`.

If `pos` is specified, then it returns the positions of the first appearance of subword starting at `pos`.

If `subword` is not found in `word`, then return `False`.

EXAMPLES:

```
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [2,4])
[1, 3]
sage: sage.combinat.subword.smallest_positions([1,2,3,4,4], [2,4])
[1, 3]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4])
[2, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4],2)
[2, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4],3)
[3, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [2,3])
[1, 2]
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [5,5])
False
sage: sage.combinat.subword.smallest_positions([1,3,3,4,5],[3,5])
[1, 4]
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3])
[1, 3, 6]
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3],2)
[2, 3, 6]
```

(continues on next page)

(continued from previous page)

```
sage: sage.combinat.subword.smallest_positions([1,2,3,4,3,4,4],[2,3,3,1])
False
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3],3)
False
```

```
>>> from sage.all import *
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(2),Integer(3),
↪Integer(4)], [Integer(2),Integer(4)])
[1, 3]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(4)], [Integer(2),Integer(4)])
[1, 3]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(2),Integer(3),
↪Integer(3),Integer(4),Integer(4)], [Integer(3),Integer(4)])
[2, 4]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(2),Integer(3),
↪Integer(3),Integer(4),Integer(4)], [Integer(3),Integer(4)],Integer(2))
[2, 4]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(2),Integer(3),
↪Integer(3),Integer(4),Integer(4)], [Integer(3),Integer(4)],Integer(3))
[3, 4]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(2),Integer(3),
↪Integer(4)], [Integer(2),Integer(3)])
[1, 2]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(2),Integer(3),
↪Integer(4)], [Integer(5),Integer(5)])
False
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(3),Integer(3),
↪Integer(4),Integer(5)],[Integer(3),Integer(5)])
[1, 4]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(3),Integer(3),
↪Integer(5),Integer(4),Integer(5),Integer(3),Integer(5)],[Integer(3),Integer(5),
↪Integer(3)])
[1, 3, 6]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(3),Integer(3),
↪Integer(5),Integer(4),Integer(5),Integer(3),Integer(5)],[Integer(3),Integer(5),
↪Integer(3)],Integer(2))
[2, 3, 6]
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(3),Integer(4),Integer(4)],[Integer(2),Integer(3),Integer(3),
↪Integer(1)])
False
>>> sage.combinat.subword.smallest_positions([Integer(1),Integer(3),Integer(3),
↪Integer(5),Integer(4),Integer(5),Integer(3),Integer(5)],[Integer(3),Integer(5),
↪Integer(3)],Integer(3))
False
```

5.1.339 Subword complex

Fix a Coxeter system (W, S). The subword complex $\mathcal{SC}(Q, w)$ associated to a word $Q \in S^*$ and an element $w \in W$ is the simplicial complex whose ground set is the set of positions in Q and whose facets are complements of sets of positions defining a reduced expression for w.

A subword complex is a shellable sphere if and only if the Demazure product of Q equals w, otherwise it is a shellable ball.

The code is optimized to be used with `ReflectionGroup`, it works as well with `CoxeterGroup`, but many methods fail for `WeylGroup`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',3]); I = list(W.index_set())          # optional - gap3
sage: Q = I + W.w0.coxeter_sorting_word(I); Q                        # optional - gap3
[1, 2, 3, 1, 2, 3, 1, 2, 1]

sage: S = SubwordComplex(Q,W.w0)                                     # optional - gap3
sage: for F in S: print("{} {}".format(F, F.root_configuration()))            #␣
↪optional - gap3
(0, 1, 2) [(1, 0, 0), (0, 1, 0), (0, 0, 1)]
(0, 1, 8) [(1, 0, 0), (0, 1, 0), (0, 0, -1)]
(0, 2, 6) [(1, 0, 0), (0, 1, 1), (0, -1, 0)]
(0, 6, 7) [(1, 0, 0), (0, 0, 1), (0, -1, -1)]
(0, 7, 8) [(1, 0, 0), (0, -1, 0), (0, 0, -1)]
(1, 2, 3) [(1, 1, 0), (0, 0, 1), (-1, 0, 0)]
(1, 3, 8) [(1, 1, 0), (-1, 0, 0), (0, 0, -1)]
(2, 3, 4) [(1, 1, 1), (0, 1, 0), (-1, -1, 0)]
(2, 4, 6) [(1, 1, 1), (-1, 0, 0), (0, -1, 0)]
(3, 4, 5) [(0, 1, 0), (0, 0, 1), (-1, -1, -1)]
(3, 5, 8) [(0, 1, 0), (-1, -1, 0), (0, 0, -1)]
(4, 5, 6) [(0, 1, 1), (-1, -1, -1), (0, -1, 0)]
(5, 6, 7) [(-1, 0, 0), (0, 0, 1), (0, -1, -1)]
(5, 7, 8) [(-1, 0, 0), (0, -1, 0), (0, 0, -1)]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(3)]); I = list(W.index_set())        # optional␣
↪- gap3
>>> Q = I + W.w0.coxeter_sorting_word(I); Q                          # optional - gap3
[1, 2, 3, 1, 2, 3, 1, 2, 1]

>>> S = SubwordComplex(Q,W.w0)                                       # optional - gap3
>>> for F in S: print("{} {}".format(F, F.root_configuration()))              #␣
↪optional - gap3
(0, 1, 2) [(1, 0, 0), (0, 1, 0), (0, 0, 1)]
(0, 1, 8) [(1, 0, 0), (0, 1, 0), (0, 0, -1)]
(0, 2, 6) [(1, 0, 0), (0, 1, 1), (0, -1, 0)]
(0, 6, 7) [(1, 0, 0), (0, 0, 1), (0, -1, -1)]
(0, 7, 8) [(1, 0, 0), (0, -1, 0), (0, 0, -1)]
(1, 2, 3) [(1, 1, 0), (0, 0, 1), (-1, 0, 0)]
(1, 3, 8) [(1, 1, 0), (-1, 0, 0), (0, 0, -1)]
(2, 3, 4) [(1, 1, 1), (0, 1, 0), (-1, -1, 0)]
(2, 4, 6) [(1, 1, 1), (-1, 0, 0), (0, -1, 0)]
(3, 4, 5) [(0, 1, 0), (0, 0, 1), (-1, -1, -1)]
```

(continues on next page)

(continued from previous page)

```
(3, 5, 8) [(0, 1, 0), (-1, -1, 0), (0, 0, -1)]
(4, 5, 6) [(0, 1, 1), (-1, -1, -1), (0, -1, 0)]
(5, 6, 7) [(-1, 0, 0), (0, 0, 1), (0, -1, -1)]
(5, 7, 8) [(-1, 0, 0), (0, -1, 0), (0, 0, -1)]
```

Testing that the implementation also works with CoxeterGroup:

```
sage: W = CoxeterGroup(['A',3]); I = list(W.index_set())
sage: Q = I + W.w0.coxeter_sorting_word(I); Q
[1, 2, 3, 1, 2, 3, 1, 2, 1]
sage: S = SubwordComplex(Q,W.w0); S
Subword complex of type ['A', 3] for Q = (1, 2, 3, 1, 2, 3, 1, 2, 1) and pi = [1, 2,␣
↪3, 1, 2, 1]
sage: P = S.increasing_flip_poset(); P; len(P.cover_relations())
Finite poset containing 14 elements
21
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)]); I = list(W.index_set())
>>> Q = I + W.w0.coxeter_sorting_word(I); Q
[1, 2, 3, 1, 2, 3, 1, 2, 1]
>>> S = SubwordComplex(Q,W.w0); S
Subword complex of type ['A', 3] for Q = (1, 2, 3, 1, 2, 3, 1, 2, 1) and pi = [1, 2,␣
↪3, 1, 2, 1]
>>> P = S.increasing_flip_poset(); P; len(P.cover_relations())
Finite poset containing 14 elements
21
```

The root configuration works:

```
sage: for F in S: print("{} {}".format(F, F.root_configuration()))
(0, 1, 2) [(1, 0, 0), (0, 1, 0), (0, 0, 1)]
(0, 1, 8) [(1, 0, 0), (0, 1, 0), (0, 0, -1)]
(0, 2, 6) [(1, 0, 0), (0, 1, 1), (0, -1, 0)]
(0, 6, 7) [(1, 0, 0), (0, 0, 1), (0, -1, -1)]
(0, 7, 8) [(1, 0, 0), (0, -1, 0), (0, 0, -1)]
(1, 2, 3) [(1, 1, 0), (0, 0, 1), (-1, 0, 0)]
(1, 3, 8) [(1, 1, 0), (-1, 0, 0), (0, 0, -1)]
(2, 3, 4) [(1, 1, 1), (0, 1, 0), (-1, -1, 0)]
(2, 4, 6) [(1, 1, 1), (-1, 0, 0), (0, -1, 0)]
(3, 4, 5) [(0, 1, 0), (0, 0, 1), (-1, -1, -1)]
(3, 5, 8) [(0, 1, 0), (-1, -1, 0), (0, 0, -1)]
(4, 5, 6) [(0, 1, 1), (-1, -1, -1), (0, -1, 0)]
(5, 6, 7) [(-1, 0, 0), (0, 0, 1), (0, -1, -1)]
(5, 7, 8) [(-1, 0, 0), (0, -1, 0), (0, 0, -1)]
```

```
>>> from sage.all import *
>>> for F in S: print("{} {}".format(F, F.root_configuration()))
(0, 1, 2) [(1, 0, 0), (0, 1, 0), (0, 0, 1)]
(0, 1, 8) [(1, 0, 0), (0, 1, 0), (0, 0, -1)]
(0, 2, 6) [(1, 0, 0), (0, 1, 1), (0, -1, 0)]
(0, 6, 7) [(1, 0, 0), (0, 0, 1), (0, -1, -1)]
```

(continues on next page)

(continued from previous page)

```
(0, 7, 8) [(1, 0, 0), (0, -1, 0), (0, 0, -1)]
(1, 2, 3) [(1, 1, 0), (0, 0, 1), (-1, 0, 0)]
(1, 3, 8) [(1, 1, 0), (-1, 0, 0), (0, 0, -1)]
(2, 3, 4) [(1, 1, 1), (0, 1, 0), (-1, -1, 0)]
(2, 4, 6) [(1, 1, 1), (-1, 0, 0), (0, -1, 0)]
(3, 4, 5) [(0, 1, 0), (0, 0, 1), (-1, -1, -1)]
(3, 5, 8) [(0, 1, 0), (-1, -1, 0), (0, 0, -1)]
(4, 5, 6) [(0, 1, 1), (-1, -1, -1), (0, -1, 0)]
(5, 6, 7) [(-1, 0, 0), (0, 0, 1), (0, -1, -1)]
(5, 7, 8) [(-1, 0, 0), (0, -1, 0), (0, 0, -1)]
```

And the weight configuration also works:

```
sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2])
sage: F.extended_weight_configuration()
[(4/3, 2/3), (2/3, 4/3), (-2/3, 2/3), (2/3, 4/3), (-2/3, 2/3)]
sage: F.extended_weight_configuration(coefficients=(1,2))
[(4/3, 2/3), (4/3, 8/3), (-2/3, 2/3), (4/3, 8/3), (-2/3, 2/3)]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)])
>>> F.extended_weight_configuration()
[(4/3, 2/3), (2/3, 4/3), (-2/3, 2/3), (2/3, 4/3), (-2/3, 2/3)]
>>> F.extended_weight_configuration(coefficients=(Integer(1),Integer(2)))
[(4/3, 2/3), (4/3, 8/3), (-2/3, 2/3), (4/3, 8/3), (-2/3, 2/3)]
```

One finally can compute the brick polytope, using all functionality on weight configurations, though it does not realize to live in real space:

```
sage: W = CoxeterGroup(['A',3]); I = list(W.index_set())
sage: Q = I + W.w0.coxeter_sorting_word(I)
sage: S = SubwordComplex(Q,W.w0)
sage: S.brick_polytope()                                                    #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 14 vertices

sage: W = CoxeterGroup(['H',3]); I = list(W.index_set())
sage: Q = I + W.w0.coxeter_sorting_word(I)
sage: S = SubwordComplex(Q,W.w0)
sage: S.brick_polytope()                                                    #␣
↪needs sage.geometry.polyhedron
doctest:...: RuntimeWarning: the polytope is built with rational vertices
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 32 vertices
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)]); I = list(W.index_set())
```

(continues on next page)

(continued from previous page)

```
>>> Q = I + W.w0.coxeter_sorting_word(I)
>>> S = SubwordComplex(Q,W.w0)
>>> S.brick_polytope()                                                    #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 14 vertices

>>> W = CoxeterGroup(['H',Integer(3)]); I = list(W.index_set())
>>> Q = I + W.w0.coxeter_sorting_word(I)
>>> S = SubwordComplex(Q,W.w0)
>>> S.brick_polytope()                                                    #␣
↪needs sage.geometry.polyhedron
doctest:...: RuntimeWarning: the polytope is built with rational vertices
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 32 vertices
```

AUTHORS:

- Christian Stump: initial version
- Vincent Pilaud: greedy flip algorithm, minor improvements, documentation

REFERENCES:

class sage.combinat.subword_complex.**SubwordComplex**(*Q*, *w*, *algorithm='inductive'*)

Bases: UniqueRepresentation, SimplicialComplex

Fix a Coxeter system (W, S). The subword complex $\mathcal{SC}(Q, w)$ associated to a word $Q \in S^*$ and an element $w \in W$ is the simplicial complex whose ground set is the set of positions in Q and whose facets are complements of sets of positions defining a reduced expression for w.

A subword complex is a shellable sphere if and only if the Demazure product of Q equals w, otherwise it is a shellable ball.

> **Warning**
>
> This implementation only works for groups build using `CoxeterGroup`, and does not work with groups build using `WeylGroup`.

EXAMPLES:

As an example, dual associahedra are subword complexes in type A_{n-1} given by the word $[1, \dots, n, 1, \dots, n, 1, \dots, n-1, \dots, 1, 2, 1]$ and the permutation w_0.

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w); SC
Subword complex of type ['A', 2] for Q = (1, 2, 1, 2, 1) and pi = [1, 2, 1]
sage: SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w); SC
Subword complex of type ['A', 2] for Q = (1, 2, 1, 2, 1) and pi = [1, 2, 1]
```

(continues on next page)

(continued from previous page)

```
sage: SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),Integer(1)],
↪w); SC
Subword complex of type ['A', 2] for Q = (1, 2, 1, 2, 1) and pi = [1, 2, 1]
>>> SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),Integer(1)],
↪w); SC
Subword complex of type ['A', 2] for Q = (1, 2, 1, 2, 1) and pi = [1, 2, 1]
>>> SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]
```

REFERENCES: [KnuMil], [PilStu]

Element

alias of *SubwordComplexFacet*

barycenter()

Return the barycenter of the brick polytope of `self`.

> **See also**
>
> *brick_polytope()*

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                          # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)                # optional - gap3
sage: SC.barycenter()                                       # optional - gap3
(2/3, 4/3)

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.barycenter()
(4/3, 8/3)
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                              # optional
↪- gap3
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)                 # optional - gap3
>>> SC.barycenter()                                             # optional - gap3
```

(continues on next page)

(continued from previous page)

```
(2/3, 4/3)

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> SC.barycenter()
(4/3, 8/3)
```

brick_fan()

Return the brick fan of `self`.

It is the normal fan of the brick polytope of `self`. It is formed by the cones generated by the weight configurations of the facets of `self`.

See also

weight_cone

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.brick_fan()
Rational polyhedral fan in 2-d lattice N

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.brick_fan()
Rational polyhedral fan in 2-d lattice N
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.brick_fan()
Rational polyhedral fan in 2-d lattice N

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.brick_fan()
Rational polyhedral fan in 2-d lattice N
```

brick_polytope(*coefficients=None*)

Return the brick polytope of `self`.

This polytope is the convex hull of the brick vectors of `self`.

INPUT:

- `coefficients` – (optional) a list of coefficients used to scale the fundamental weights

See also:

`brick_vectors()`

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                           # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)                 # optional - gap3
sage: X = SC.brick_polytope(); X                             # optional - gap3
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices

sage: Y = SC.brick_polytope(coefficients=[1,2]); Y           # optional - gap3
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices

sage: X == Y                                                 # optional - gap3
False

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: X = SC.brick_polytope(); X
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices

sage: # optional - gap3
sage: W = ReflectionGroup(['H',3])
sage: c = W.index_set(); Q = c + tuple(W.w0.coxeter_sorting_word(c))
sage: SC = SubwordComplex(Q,W.w0)
sage: SC.brick_polytope()
doctest:...:
RuntimeWarning: the polytope is built with rational vertices
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 32 vertices
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                            # optional␣
↪- gap3
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)                   # optional - gap3
>>> X = SC.brick_polytope(); X                             # optional - gap3
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices

>>> Y = SC.brick_polytope(coefficients=[Integer(1),Integer(2)]); Y          #␣
↪optional - gap3
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices

>>> X == Y                                                 # optional - gap3
False

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
```

(continues on next page)

(continued from previous page)

```
>>> X = SC.brick_polytope(); X
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices

>>> # optional - gap3
>>> W = ReflectionGroup(['H',Integer(3)])
>>> c = W.index_set(); Q = c + tuple(W.w0.coxeter_sorting_word(c))
>>> SC = SubwordComplex(Q,W.w0)
>>> SC.brick_polytope()
doctest:...:
RuntimeWarning: the polytope is built with rational vertices
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 32 vertices
```

brick_vectors(*coefficients=None*)

Return the list of all brick vectors of facets of `self`.

INPUT:

- `coefficients` – (optional) a list of coefficients used to scale the fundamental weights

See also

brick_vector

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.brick_vectors()
[(5/3, 7/3), (5/3, 1/3), (2/3, 7/3), (-1/3, 4/3), (-1/3, 1/3)]
sage: SC.brick_vectors(coefficients=(1,2))
[(7/3, 11/3), (7/3, 2/3), (4/3, 11/3), (-2/3, 5/3), (-2/3, 2/3)]

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.brick_vectors()
[(10/3, 14/3), (10/3, 2/3), (4/3, 14/3), (-2/3, 8/3), (-2/3, 2/3)]
sage: SC.brick_vectors(coefficients=(1,2))
[(14/3, 22/3), (14/3, 4/3), (8/3, 22/3), (-4/3, 10/3), (-4/3, 4/3)]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> SC.brick_vectors()
[(5/3, 7/3), (5/3, 1/3), (2/3, 7/3), (-1/3, 4/3), (-1/3, 1/3)]
>>> SC.brick_vectors(coefficients=(Integer(1),Integer(2)))
[(7/3, 11/3), (7/3, 2/3), (4/3, 11/3), (-2/3, 5/3), (-2/3, 2/3)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)], W.w0)
>>> SC.brick_vectors()
[(10/3, 14/3), (10/3, 2/3), (4/3, 14/3), (-2/3, 8/3), (-2/3, 2/3)]
>>> SC.brick_vectors(coefficients=(Integer(1),Integer(2)))
[(14/3, 22/3), (14/3, 4/3), (8/3, 22/3), (-4/3, 10/3), (-4/3, 4/3)]
```

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.cartan_type()
['A', 2]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.cartan_type()
['A', 2]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.cartan_type()
['A', 2]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.cartan_type()
['A', 2]
```

cover_relations(*label=False*)

Return the set of cover relations in the associated poset.

INPUT:

- `label` – boolean (default: `False`); whether or not to label the cover relations by the position of flip

OUTPUT: list of pairs of facets

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                          # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)                # optional - gap3
sage: sorted(SC.cover_relations())                          # optional - gap3
```

(continues on next page)

(continued from previous page)

```
[((0, 1), (0, 4)),
 ((0, 1), (1, 2)),
 ((0, 4), (3, 4)),
 ((1, 2), (2, 3)),
 ((2, 3), (3, 4))]

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: sorted(SC.cover_relations())
[((0, 1), (0, 4)),
 ((0, 1), (1, 2)),
 ((0, 4), (3, 4)),
 ((1, 2), (2, 3)),
 ((2, 3), (3, 4))]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                             # optional␣
↪- gap3
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)                     # optional - gap3
>>> sorted(SC.cover_relations())                           # optional - gap3
[((0, 1), (0, 4)),
 ((0, 1), (1, 2)),
 ((0, 4), (3, 4)),
 ((1, 2), (2, 3)),
 ((2, 3), (3, 4))]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> sorted(SC.cover_relations())
[((0, 1), (0, 4)),
 ((0, 1), (1, 2)),
 ((0, 4), (3, 4)),
 ((1, 2), (2, 3)),
 ((2, 3), (3, 4))]
```

dimension()

Return the dimension of `self`.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                         # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)               # optional - gap3
sage: SC.dimension()                                       # optional - gap3
1

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.dimension()
1
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                          # optional␣
↪- gap3
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)                    # optional - gap3
>>> SC.dimension()                                          # optional - gap3
1

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> SC.dimension()
1
```

facets()

Return all facets of `self`.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]
```

greedy_facet(*side='positive'*)

Return the negative (or positive) greedy facet of `self`.

This is the lexicographically last (or first) facet of `self`.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.greedy_facet(side='positive')
(0, 1)
sage: SC.greedy_facet(side='negative')
(3, 4)

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.greedy_facet(side='positive')
(0, 1)
sage: SC.greedy_facet(side='negative')
(3, 4)
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.greedy_facet(side='positive')
(0, 1)
>>> SC.greedy_facet(side='negative')
(3, 4)

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.greedy_facet(side='positive')
(0, 1)
>>> SC.greedy_facet(side='negative')
(3, 4)
```

group()

Return the group associated to `self`.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.group()
Irreducible real reflection group of rank 2 and type A2

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.group()
```

(continues on next page)

(continued from previous page)

```
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3]
[3 1]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.group()
Irreducible real reflection group of rank 2 and type A2

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.group()
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3]
[3 1]
```

increasing_flip_graph(*label=True*)

Return the increasing flip graph of the subword complex.

OUTPUT: a directed graph

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                          # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)                # optional - gap3
sage: SC.increasing_flip_graph()                            # optional - gap3
Digraph on 5 vertices

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.increasing_flip_graph()
Digraph on 5 vertices
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                          # optional␣
↪- gap3
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)                # optional - gap3
>>> SC.increasing_flip_graph()                            # optional - gap3
Digraph on 5 vertices

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> SC.increasing_flip_graph()
Digraph on 5 vertices
```

increasing_flip_poset()

Return the increasing flip poset of the subword complex.

OUTPUT: a poset

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                          # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)                # optional - gap3
sage: SC.increasing_flip_poset()                            # optional - gap3
Finite poset containing 5 elements

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.increasing_flip_poset()
Finite poset containing 5 elements
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                          # optional␣
↪- gap3
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)                # optional - gap3
>>> SC.increasing_flip_poset()                            # optional - gap3
Finite poset containing 5 elements

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> SC.increasing_flip_poset()
Finite poset containing 5 elements
```

interval(*I*, *J*)

Return the interval [I,J] in the increasing flip graph subword complex.

INPUT:

- I, J – two facets

OUTPUT: a set of facets

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: F = SC([1,2])
sage: SC.interval(F, F)
{(1, 2)}

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: F = SC([1,2])
sage: SC.interval(F, F)
{(1, 2)}
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> F = SC([Integer(1),Integer(2)])
>>> SC.interval(F, F)
{(1, 2)}

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> F = SC([Integer(1),Integer(2)])
>>> SC.interval(F, F)
{(1, 2)}
```

is_ball()

Return `True` if the subword complex `self` is a ball.

This is the case if and only if it is not a sphere.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_ball()
False

sage: SC = SubwordComplex([3,2,1,3,2,3], w)                    # optional - gap3
sage: SC.is_ball()                                              # optional - gap3
True

sage: W = CoxeterGroup(['A',3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_ball()
False
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(3)])
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(2)])
>>> SC = SubwordComplex([Integer(3),Integer(2),Integer(3),Integer(2),
↪Integer(3)], w)
>>> SC.is_ball()
False

>>> SC = SubwordComplex([Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3)], w)                    # optional - gap3
>>> SC.is_ball()                                        # optional - gap3
True
```

(continues on next page)

(continued from previous page)

```
>>> W = CoxeterGroup(['A',Integer(3)])
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(2)])
>>> SC = SubwordComplex([Integer(3),Integer(2),Integer(3),Integer(2),
↪Integer(3)], w)
>>> SC.is_ball()
False
```

is_double_root_free()

Return `True` if `self` is double-root-free.

This means that the root configurations of all facets do not contain a root twice.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.is_double_root_free()
True

sage: SC = SubwordComplex([1,1,2,2,1,1], w)                    # optional - gap3
sage: SC.is_double_root_free()                                 # optional - gap3
True

sage: SC = SubwordComplex([1,2,1,2,1,2], w)                    # optional - gap3
sage: SC.is_double_root_free()                                 # optional - gap3
False

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.is_double_root_free()
True
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.is_double_root_free()
True

>>> SC = SubwordComplex([Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(1),Integer(1)], w)                    # optional - gap3
>>> SC.is_double_root_free()                                # optional - gap3
True

>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1),Integer(2)], w)                    # optional - gap3
>>> SC.is_double_root_free()                                # optional - gap3
```

(continues on next page)

(continued from previous page)

```
False

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.is_double_root_free()
True
```

is_pure()

Return `True` since all subword complexes are pure.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_pure()
True

sage: W = CoxeterGroup(['A',3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_pure()
True
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(3)])
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(2)])
>>> SC = SubwordComplex([Integer(3),Integer(2),Integer(3),Integer(2),
↪Integer(3)], w)
>>> SC.is_pure()
True

>>> W = CoxeterGroup(['A',Integer(3)])
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(2)])
>>> SC = SubwordComplex([Integer(3),Integer(2),Integer(3),Integer(2),
↪Integer(3)], w)
>>> SC.is_pure()
True
```

is_root_independent()

Return `True` if `self` is root-independent.

This means that the root configuration of any (or equivalently all) facets is linearly independent.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                              # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)                    # optional - gap3
sage: SC.is_root_independent()                                  # optional - gap3
```

(continues on next page)

(continued from previous page)

```
True

sage: SC = SubwordComplex([1,2,1,2,1,2], W.w0)               # optional - gap3
sage: SC.is_root_independent()                               # optional - gap3
False

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.is_root_independent()
True
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                          # optional␣
↪- gap3
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)                   # optional - gap3
>>> SC.is_root_independent()                              # optional - gap3
True

>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1),Integer(2)], W.w0)             # optional - gap3
>>> SC.is_root_independent()                              # optional - gap3
False

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> SC.is_root_independent()
True
```

is_sphere()

Return `True` if the subword complex `self` is a sphere.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_sphere()
True

sage: SC = SubwordComplex([3,2,1,3,2,3], w)                  # optional - gap3
sage: SC.is_sphere()                                         # optional - gap3
False

sage: W = CoxeterGroup(['A',3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_sphere()
True
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(3)])
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(2)])
>>> SC = SubwordComplex([Integer(3),Integer(2),Integer(3),Integer(2),
↪Integer(3)], w)
>>> SC.is_sphere()
True

>>> SC = SubwordComplex([Integer(3),Integer(2),Integer(1),Integer(3),
↪Integer(2),Integer(3)], w)                    # optional - gap3
>>> SC.is_sphere()                                         # optional - gap3
False

>>> W = CoxeterGroup(['A',Integer(3)])
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(2)])
>>> SC = SubwordComplex([Integer(3),Integer(2),Integer(3),Integer(2),
↪Integer(3)], w)
>>> SC.is_sphere()
True
```

kappa_preimages()

Return a dictionary containing facets of `self` as keys, and list of elements of `self.group()` as values.

See also

kappa_preimage

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: kappa = SC.kappa_preimages()
sage: for F in SC: print("{} {}".format(F, [w.reduced_word() for w in␣
↪kappa[F]]))
(0, 1) [[]]
(0, 4) [[2], [2, 1]]
(1, 2) [[1]]
(2, 3) [[1, 2]]
(3, 4) [[1, 2, 1]]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: kappa = SC.kappa_preimages()
sage: for F in SC: print("{} {}".format(F, [w.reduced_word() for w in␣
↪kappa[F]]))
(0, 1) [[]]
(0, 4) [[2], [2, 1]]
(1, 2) [[1]]
```

(continues on next page)

(continued from previous page)

```
(2, 3) [[1, 2]]
(3, 4) [[1, 2, 1]]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> kappa = SC.kappa_preimages()
>>> for F in SC: print("{} {}".format(F, [w.reduced_word() for w in␣
↪kappa[F]]))
(0, 1) [[]]
(0, 4) [[2], [2, 1]]
(1, 2) [[1]]
(2, 3) [[1, 2]]
(3, 4) [[1, 2, 1]]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> kappa = SC.kappa_preimages()
>>> for F in SC: print("{} {}".format(F, [w.reduced_word() for w in␣
↪kappa[F]]))
(0, 1) [[]]
(0, 4) [[2], [2, 1]]
(1, 2) [[1]]
(2, 3) [[1, 2]]
(3, 4) [[1, 2, 1]]
```

minkowski_summand(i)

Return the i th Minkowski summand of `self`.

INPUT:

- `i` – an integer defining a position in the word Q

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                          # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)                # optional - gap3
sage: SC.minkowski_summand(1)                               # optional - gap3
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex

sage: W = CoxeterGroup(['A',2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.minkowski_summand(1)
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                              # optional␣
↪- gap3
```

(continues on next page)

(continued from previous page)

```
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)                    # optional - gap3
>>> SC.minkowski_summand(Integer(1))                              # optional␣
↪- gap3
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex

>>> W = CoxeterGroup(['A',Integer(2)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], W.w0)
>>> SC.minkowski_summand(Integer(1))
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
```

pi()

Return the element in the Coxeter group associated to `self`.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.pi().reduced_word()
[1, 2, 1]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.pi().reduced_word()
[1, 2, 1]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.pi().reduced_word()
[1, 2, 1]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.pi().reduced_word()
[1, 2, 1]
```

word()

Return the word in the simple generators associated to `self`.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
```

(continues on next page)

(continued from previous page)

```
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.word()
(1, 2, 1, 2, 1)

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.word()
(1, 2, 1, 2, 1)
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.word()
(1, 2, 1, 2, 1)

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)], w)
>>> SC.word()
(1, 2, 1, 2, 1)
```

class sage.combinat.subword_complex.**SubwordComplexFacet**(*parent*, *positions*, *facet_test=True*)

Bases: `Simplex`, `Element`

A facet of a subword complex.

Facets of the subword complex $\mathcal{SC}(Q, w)$ are complements of sets of positions in Q defining a reduced expression for w.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: F = SC[0]; F
(0, 1)

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: F = SC[0]; F
(0, 1)
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),Integer(1)],␣
↪w)
>>> F = SC[Integer(0)]; F
(0, 1)

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),Integer(1)],␣
↪w)
>>> F = SC[Integer(0)]; F
(0, 1)
```

brick_vector(*coefficients=None*)

Return the brick vector of `self`.

This is the sum of the weight vectors in the extended weight configuration.

INPUT:

- `coefficients` – (optional) a list of coefficients used to scale the fundamental weights

See also

extended_weight_configuration()

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.extended_weight_configuration()
[(2/3, 1/3), (1/3, 2/3), (-1/3, 1/3), (1/3, 2/3), (-1/3, 1/3)]
sage: F.brick_vector()
(2/3, 7/3)
sage: F.brick_vector(coefficients=[1,2])
(4/3, 11/3)

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2])
sage: F.brick_vector()
(4/3, 14/3)
sage: F.brick_vector(coefficients=[1,2])
(8/3, 22/3)
```

```
>>> from sage.all import *
>>> # optional - gap3
```

(continues on next page)

(continued from previous page)

```
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.extended_weight_configuration()
[(2/3, 1/3), (1/3, 2/3), (-1/3, 1/3), (1/3, 2/3), (-1/3, 1/3)]
>>> F.brick_vector()
(2/3, 7/3)
>>> F.brick_vector(coefficients=[Integer(1),Integer(2)])
(4/3, 11/3)

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)])
>>> F.brick_vector()
(4/3, 14/3)
>>> F.brick_vector(coefficients=[Integer(1),Integer(2)])
(8/3, 22/3)
```

extended_root_configuration()

Return the extended root configuration of `self`.

Let $Q = q_1 \ldots q_m \in S^*$ and $w \in W$. The extended root configuration of a facet I of $\mathcal{SC}(Q,w)$ is the sequence $\mathsf{r}(I,1),\ldots,\mathsf{r}(I,m)$ of roots defined by $\mathsf{r}(I,k) = \Pi Q_{[k-1]\setminus I}(\alpha_{q_k})$, where $\Pi Q_{[k-1]\setminus I}$ is the product of the simple reflections q_i for $i \in [k-1] \setminus I$ in this order.

The extended root configuration is used to perform flips efficiently.

See also

flip()

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.extended_root_configuration()
[(1, 0), (1, 1), (-1, 0), (1, 1), (0, 1)]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
```

(continues on next page)

(continued from previous page)

```
sage: F.extended_root_configuration()
[(1, 0), (1, 1), (-1, 0), (1, 1), (0, 1)]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.extended_root_configuration()
[(1, 0), (1, 1), (-1, 0), (1, 1), (0, 1)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.extended_root_configuration()
[(1, 0), (1, 1), (-1, 0), (1, 1), (0, 1)]
```

extended_weight_configuration(*coefficients=None*)

Return the extended weight configuration of `self`.

Let $Q = q_1 \dots q_m \in S^*$ and $w \in W$. The extended weight configuration of a facet I of $\mathcal{SC}(Q, w)$ is the sequence $\mathsf{w}(I, 1), \dots, \mathsf{w}(I, m)$ of weights defined by $\mathsf{w}(I, k) = \Pi Q_{[k-1] \smallsetminus I}(\omega_{q_k})$, where $\Pi Q_{[k-1] \smallsetminus I}$ is the product of the simple reflections q_i for $i \in [k-1] \smallsetminus I$ in this order.

The extended weight configuration is used to compute the brick vector.

INPUT:

- `coefficients` – (optional) a list of coefficients used to scale the fundamental weights

See also

brick_vector()

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2])
sage: F.extended_weight_configuration()
[(2/3, 1/3), (1/3, 2/3), (-1/3, 1/3), (1/3, 2/3), (-1/3, 1/3)]
sage: F.extended_weight_configuration(coefficients=(1,2))
[(2/3, 1/3), (2/3, 4/3), (-1/3, 1/3), (2/3, 4/3), (-1/3, 1/3)]

sage: W = CoxeterGroup(['A',2])
```

(continues on next page)

(continued from previous page)

```
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2])
sage: F.extended_weight_configuration()
[(4/3, 2/3), (2/3, 4/3), (-2/3, 2/3), (2/3, 4/3), (-2/3, 2/3)]
sage: F.extended_weight_configuration(coefficients=(1,2))
[(4/3, 2/3), (4/3, 8/3), (-2/3, 2/3), (4/3, 8/3), (-2/3, 2/3)]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)])
>>> F.extended_weight_configuration()
[(2/3, 1/3), (1/3, 2/3), (-1/3, 1/3), (1/3, 2/3), (-1/3, 1/3)]
>>> F.extended_weight_configuration(coefficients=(Integer(1),Integer(2)))
[(2/3, 1/3), (2/3, 4/3), (-1/3, 1/3), (2/3, 4/3), (-1/3, 1/3)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)])
>>> F.extended_weight_configuration()
[(4/3, 2/3), (2/3, 4/3), (-2/3, 2/3), (2/3, 4/3), (-2/3, 2/3)]
>>> F.extended_weight_configuration(coefficients=(Integer(1),Integer(2)))
[(4/3, 2/3), (4/3, 8/3), (-2/3, 2/3), (4/3, 8/3), (-2/3, 2/3)]
```

flip(*i*, *return_position=False*)

Return the facet obtained after flipping position `i` in `self`.

INPUT:

- `i` – integer; position in the word Q
- `return_position` – boolean (default: `False`); tells whether the new position should be returned as well

OUTPUT:

- The new subword complex facet.
- The new position if `return_position` is `True`.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.flip(1)
```

(continues on next page)

(continued from previous page)

```
(2, 3)
sage: F.flip(1, return_position=True)
((2, 3), 3)

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.flip(1)
(2, 3)
sage: F.flip(1, return_position=True)
((2, 3), 3)
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.flip(Integer(1))
(2, 3)
>>> F.flip(Integer(1), return_position=True)
((2, 3), 3)

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.flip(Integer(1))
(2, 3)
>>> F.flip(Integer(1), return_position=True)
((2, 3), 3)
```

is_vertex()

Return `True` if `self` is a vertex of the brick polytope of `self.parent`.

A facet is a vertex of the brick polytope if its root cone is pointed. Note that this property is always satisfied for root-independent subword complexes.

See also

root_cone()

EXAMPLES:

```
sage: # optional - gap3
```

(continues on next page)

(continued from previous page)

```
sage: W = ReflectionGroup(['A',1])
sage: w = W.from_reduced_word([1])
sage: SC = SubwordComplex([1,1,1],w)
sage: F = SC([0,1]); F.is_vertex()
True
sage: F = SC([0,2]); F.is_vertex()
False

sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1,2,1],w)
sage: F = SC([0,1,2,3]); F.is_vertex()
True
sage: F = SC([0,1,2,6]); F.is_vertex()
False

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1,2,1],w)
sage: F = SC([0,1,2,3]); F.is_vertex()
True
sage: F = SC([0,1,2,6]); F.is_vertex()
False
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(1)])
>>> w = W.from_reduced_word([Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(1),Integer(1)],w)
>>> F = SC([Integer(0),Integer(1)]); F.is_vertex()
True
>>> F = SC([Integer(0),Integer(2)]); F.is_vertex()
False

>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1),Integer(2),Integer(1)],w)
>>> F = SC([Integer(0),Integer(1),Integer(2),Integer(3)]); F.is_vertex()
True
>>> F = SC([Integer(0),Integer(1),Integer(2),Integer(6)]); F.is_vertex()
False

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1),Integer(2),Integer(1)],w)
>>> F = SC([Integer(0),Integer(1),Integer(2),Integer(3)]); F.is_vertex()
True
>>> F = SC([Integer(0),Integer(1),Integer(2),Integer(6)]); F.is_vertex()
```

(continues on next page)

(continued from previous page)

```
False
```

kappa_preimage()

Return the fiber of `self` under the κ map.

The κ map sends an element $w \in W$ to the unique facet of $I \in \mathcal{SC}(Q, w)$ such that the root configuration of I is contained in $w(\Phi^+)$. In other words, w is in the preimage of `self` under κ if and only if w^{-1} sends every root in the root configuration to a positive root.

EXAMPLES:

```
sage: W = ReflectionGroup(['A',2])                              # optional - gap3
sage: w = W.from_reduced_word([1,2,1])                          # optional - gap3
sage: SC = SubwordComplex([1,2,1,2,1],w)                        # optional - gap3

sage: F = SC([1,2]); F                                          # optional - gap3
(1, 2)
sage: F.kappa_preimage()                                        # optional - gap3
[(1,4)(2,3)(5,6)]

sage: F = SC([0,4]); F                                          # optional - gap3
(0, 4)
sage: F.kappa_preimage()                                        # optional - gap3
[(1,3)(2,5)(4,6), (1,2,6)(3,4,5)]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)

sage: F = SC([1,2]); F
(1, 2)
sage: F.kappa_preimage()
[
[-1  1]
[ 0  1]
]

sage: F = SC([0,4]); F
(0, 4)
sage: F.kappa_preimage()
[
[ 1  0]  [-1  1]
[ 1 -1], [-1  0]
]
```

```
>>> from sage.all import *
>>> W = ReflectionGroup(['A',Integer(2)])                                 # optional␣
↪- gap3
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])                 ␣
↪        # optional - gap3
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)                     # optional - gap3
```

(continues on next page)

(continued from previous page)

```
>>> F = SC([Integer(1),Integer(2)]); F                                       #␣
↪optional - gap3
(1, 2)
>>> F.kappa_preimage()                                    # optional - gap3
[(1,4)(2,3)(5,6)]

>>> F = SC([Integer(0),Integer(4)]); F                                       #␣
↪optional - gap3
(0, 4)
>>> F.kappa_preimage()                                    # optional - gap3
[(1,3)(2,5)(4,6), (1,2,6)(3,4,5)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)

>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.kappa_preimage()
[
[-1  1]
[ 0  1]
]

>>> F = SC([Integer(0),Integer(4)]); F
(0, 4)
>>> F.kappa_preimage()
[
[ 1  0]  [-1  1]
[ 1 -1], [-1  0]
]
```

plot (*list_colors=None*, *labels=[]*, *thickness=3*, *fontsize=14*, *shift=(0, 0)*, *compact=False*, *roots=True*, ***args*)

In type A or B, plot a pseudoline arrangement representing the facet `self`.

Pseudoline arrangements are graphical representations of facets of types A or B subword complexes.

INPUT:

- `list_colors` – list (default: `[]`); to change the colors of the pseudolines
- `labels` – list (default: `[]`); to change the labels of the pseudolines
- `thickness` – integer (default: 3); for the thickness of the pseudolines
- `fontsize` – integer (default: 14); for the size of the font used for labels
- `shift` – couple of coordinates (default: `(0,0)`) to change the origin
- `compact` – boolean (default: `False`); to require a more compact representation
- `roots` – boolean (default: `True`); whether to print the extended root configuration

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F.plot()                                               #␣
↪needs sage.plot
Graphics object consisting of 26 graphics primitives

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F.plot()                                               #␣
↪needs sage.plot
Graphics object consisting of 26 graphics primitives

sage: # optional - gap3
sage: W = ReflectionGroup(['B',3])
sage: c = W.from_reduced_word([1,2,3])
sage: Q = c.reduced_word()*2 + W.w0.coxeter_sorting_word(c)
sage: SC = SubwordComplex(Q, W.w0)
sage: F = SC[15]; F.plot()                                                  #␣
↪needs sage.plot
Graphics object consisting of 53 graphics primitives
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F.plot()                               ␣
↪              # needs sage.plot
Graphics object consisting of 26 graphics primitives

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F.plot()                               ␣
↪              # needs sage.plot
Graphics object consisting of 26 graphics primitives

>>> # optional - gap3
>>> W = ReflectionGroup(['B',Integer(3)])
>>> c = W.from_reduced_word([Integer(1),Integer(2),Integer(3)])
>>> Q = c.reduced_word()*Integer(2) + W.w0.coxeter_sorting_word(c)
>>> SC = SubwordComplex(Q, W.w0)
>>> F = SC[Integer(15)]; F.plot()                                           ␣
↪      # needs sage.plot
Graphics object consisting of 53 graphics primitives
```

REFERENCES: [PilStu]

root_cone()

Return the polyhedral cone generated by the root configuration of `self`.

See also:

root_configuration()

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',1])
sage: w = W.from_reduced_word([1])
sage: SC = SubwordComplex([1,1,1],w)
sage: F = SC([0,2]); F.root_cone()
1-d cone in 1-d lattice N

sage: W = CoxeterGroup(['A',1])
sage: w = W.from_reduced_word([1])
sage: SC = SubwordComplex([1,1,1],w)
sage: F = SC([0,2]); F.root_cone()
1-d cone in 1-d lattice N
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(1)])
>>> w = W.from_reduced_word([Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(1),Integer(1)],w)
>>> F = SC([Integer(0),Integer(2)]); F.root_cone()
1-d cone in 1-d lattice N

>>> W = CoxeterGroup(['A',Integer(1)])
>>> w = W.from_reduced_word([Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(1),Integer(1)],w)
>>> F = SC([Integer(0),Integer(2)]); F.root_cone()
1-d cone in 1-d lattice N
```

root_configuration()

Return the root configuration of `self`.

Let $Q = q_1 \dots q_m \in S^*$ and $w \in W$. The root configuration of a facet $I = [i_1, \dots, i_n]$ of $\mathcal{SC}(Q, w)$ is the sequence $\mathsf{r}(I, i_1), \dots, \mathsf{r}(I, i_n)$ of roots defined by $\mathsf{r}(I, k) = \Pi Q_{[k-1] \smallsetminus I}(\alpha_{q_k})$, where $\Pi Q_{[k-1] \smallsetminus I}$ is the product of the simple reflections q_i for $i \in [k-1] \smallsetminus I$ in this order.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.root_configuration()
[(1, 1), (-1, 0)]
```

(continues on next page)

(continued from previous page)

```
sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.root_configuration()                                    # optional - gap3
[(1, 1), (-1, 0)]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.root_configuration()
[(1, 1), (-1, 0)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.root_configuration()                                    # optional - gap3
[(1, 1), (-1, 0)]
```

show (**kwds*, ***args*)

Show the facet `self`.

See also

plot()

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F.show()
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
```

(continues on next page)

(continued from previous page)

```
>>> F = SC([Integer(1),Integer(2)]); F.show()
<BLANKLINE>
```

upper_root_configuration()

Return the positive roots of the root configuration of `self`.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.root_configuration()
[(1, 1), (-1, 0)]
sage: F.upper_root_configuration()
[(1, 0)]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.upper_root_configuration()
[(1, 0)]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.root_configuration()
[(1, 1), (-1, 0)]
>>> F.upper_root_configuration()
[(1, 0)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.upper_root_configuration()
[(1, 0)]
```

weight_cone()

Return the polyhedral cone generated by the weight configuration of `self`.

See also

weight_configuration()

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: WC = F.weight_cone(); WC
2-d cone in 2-d lattice N
sage: WC.rays()
N( 1, 2),
N(-1, 1)
in 2-d lattice N

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: WC = F.weight_cone(); WC
2-d cone in 2-d lattice N
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> WC = F.weight_cone(); WC
2-d cone in 2-d lattice N
>>> WC.rays()
N( 1, 2),
N(-1, 1)
in 2-d lattice N

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> WC = F.weight_cone(); WC
2-d cone in 2-d lattice N
```

weight_configuration()

Return the weight configuration of `self`.

Let $Q = q_1 \dots q_m \in S^*$ and $w \in W$. The weight configuration of a facet $I = [i_1, \dots, i_n]$ of $\mathcal{SC}(Q, w)$ is the sequence $\mathsf{w}(I, i_1), \dots, \mathsf{w}(I, i_n)$ of weights defined by $\mathsf{w}(I, k) = \Pi Q_{[k-1] \smallsetminus I}(\omega_{q_k})$, where $\Pi Q_{[k-1] \smallsetminus I}$ is the product of the simple reflections q_i for $i \in [k-1] \smallsetminus I$ in this order.

EXAMPLES:

```
sage: # optional - gap3
sage: W = ReflectionGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.weight_configuration()
[(1/3, 2/3), (-1/3, 1/3)]

sage: W = CoxeterGroup(['A',2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.weight_configuration()
[(2/3, 4/3), (-2/3, 2/3)]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.weight_configuration()
[(1/3, 2/3), (-1/3, 1/3)]

>>> W = CoxeterGroup(['A',Integer(2)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> SC = SubwordComplex([Integer(1),Integer(2),Integer(1),Integer(2),
↪Integer(1)],w)
>>> F = SC([Integer(1),Integer(2)]); F
(1, 2)
>>> F.weight_configuration()
[(2/3, 4/3), (-2/3, 2/3)]
```

5.1.340 Super Tableaux

AUTHORS:

- Matthew Lancellotti (2007): initial version
- Chaman Agrawal (2019-07-23): Modify standard and semistandard tableaux for super tableaux.

class sage.combinat.super_tableau.**SemistandardSuperTableau**(*parent*, *t*, *check=True*, *preprocessed=False*)

Bases: *Tableau*

A semistandard super tableau.

A semistandard super tableau is a tableau with primed positive integer entries. As defined in [Muth2019], a semistandard super tableau weakly increases along the rows and down the columns. Also, the letters of even parity (unprimed) strictly increases down the columns, and letters of oddd parity (primed) strictly increases along the rows. Note that Sage uses the English convention for partitions and tableaux; the longer rows are displayed on top.

INPUT:

- `t` – a tableau, a list of iterables, or an empty list

EXAMPLES:

```
sage: t = SemistandardSuperTableau([['1p',2,"3'"],[2,3]]); t
[[1', 2, 3'], [2, 3]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty printing
1' 2 3'
2 3
sage: t = Tableau([["1p",2],[2]])
sage: s = SemistandardSuperTableau(t); s
[[1', 2], [2]]
sage: SemistandardSuperTableau([]) # The empty tableau
[]
```

```
>>> from sage.all import *
>>> t = SemistandardSuperTableau([['1p',Integer(2),"3'"],[Integer(2),
↪Integer(3)]]); t
[[1', 2, 3'], [2, 3]]
>>> t.shape()
[3, 2]
>>> t.pp() # pretty printing
1' 2 3'
2 3
>>> t = Tableau([["1p",Integer(2)],[Integer(2)]])
>>> s = SemistandardSuperTableau(t); s
[[1', 2], [2]]
>>> SemistandardSuperTableau([]) # The empty tableau
[]
```

check()

Check that `self` is a valid semistandard super tableau.

class sage.combinat.super_tableau.**SemistandardSuperTableaux**(***kwds*)

Bases: *SemistandardTableaux*

The set of semistandard super tableaux.

A semistandard super tableau is a tableau with primed positive integer entries. As defined in [Muth2019], a semistandard super tableau weakly increases along the rows and down the columns. Also, the letters of even parity (unprimed) strictly increases down the columns, and letters of oddd parity (primed) strictly increases along the rows. Note that Sage uses the English convention for partitions and tableaux; the longer rows are displayed on top.

EXAMPLES:

```
sage: SST = SemistandardSuperTableaux(); SST
Semistandard super tableaux
```

```
>>> from sage.all import *
>>> SST = SemistandardSuperTableaux(); SST
Semistandard super tableaux
```

Element

alias of *SemistandardSuperTableau*

class sage.combinat.super_tableau.**SemistandardSuperTableaux_all**

Bases: *SemistandardSuperTableaux*

All semistandard super tableaux.

class sage.combinat.super_tableau.**StandardSuperTableau**(*parent*, *t*, *check=True*, *preprocessed=False*)

Bases: *SemistandardSuperTableau*

A standard super tableau.

A standard super tableau is a semistandard super tableau whose entries are in bijection with positive primed integers $1', 1, 2' \ldots n$.

For more information refer [Muth2019].

INPUT:

- t – a Tableau, a list of iterables, or an empty list

EXAMPLES:

```
sage: t = StandardSuperTableau([["1'",1,"2'",2,"3'"],[3,"4'"]]); t
[[1', 1, 2', 2, 3'], [3, 4']]
sage: t.shape()
[5, 2]
sage: t.pp() # pretty printing
1' 1 2' 2 3'
3 4'
sage: t.is_standard()
True
sage: StandardSuperTableau([]) # The empty tableau
[]
```

```
>>> from sage.all import *
>>> t = StandardSuperTableau([["1'",Integer(1),"2'",Integer(2),"3'"],[Integer(3),
↪"4'"]]); t
[[1', 1, 2', 2, 3'], [3, 4']]
>>> t.shape()
[5, 2]
>>> t.pp() # pretty printing
1' 1 2' 2 3'
3 4'
>>> t.is_standard()
True
>>> StandardSuperTableau([]) # The empty tableau
[]
```

check()

Check that self is a standard tableau.

is_standard()

Return `True` since `self` is a standard super tableau.

EXAMPLES:

```
sage: StandardSuperTableau([['1p', 1], ['2p', 2]]).is_standard()
True
```

```
>>> from sage.all import *
>>> StandardSuperTableau([['1p', Integer(1)], ['2p', Integer(2)]]).is_
↪standard()
True
```

class sage.combinat.super_tableau.**StandardSuperTableaux**(***kwds*)

Bases: *SemistandardSuperTableaux*, `Parent`

The set of standard super tableaux.

A standard super tableau is a tableau whose entries are primed positive integers, which are strictly increasing in rows and down columns and contains each letters from 1',1,2'…n exactly once.

For more information refer [Muth2019].

INPUT:

- `n` – a nonnegative integer or a partition

EXAMPLES:

```
sage: SST = StandardSuperTableaux()
sage: SST
Standard super tableaux
sage: SST([["1'",1,"2'",2,"3'"],[3,"4'"]])
[[1', 1, 2', 2, 3'], [3, 4']]
sage: SST = StandardSuperTableaux(3)
sage: SST
Standard super tableaux of size 3
sage: SST.first()
[[1', 1, 2']]
sage: SST.last()
[[1'], [1], [2']]
sage: SST.cardinality()
4
sage: SST.list()
[[[1', 1, 2']], [[1', 2'], [1]], [[1', 1], [2']], [[1'], [1], [2']]]
sage: SST = StandardSuperTableaux([3,2])
sage: SST
Standard super tableaux of shape [3, 2]
```

```
>>> from sage.all import *
>>> SST = StandardSuperTableaux()
>>> SST
Standard super tableaux
>>> SST([["1'",Integer(1),"2'",Integer(2),"3'"],[Integer(3),"4'"]])
[[1', 1, 2', 2, 3'], [3, 4']]
>>> SST = StandardSuperTableaux(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> SST
Standard super tableaux of size 3
>>> SST.first()
[[1', 1, 2']]
>>> SST.last()
[[1'], [1], [2']]
>>> SST.cardinality()
4
>>> SST.list()
[[[1', 1, 2']], [[1', 2'], [1]], [[1', 1], [2']], [[1'], [1], [2']]]
>>> SST = StandardSuperTableaux([Integer(3),Integer(2)])
>>> SST
Standard super tableaux of shape [3, 2]
```

Element

alias of *StandardSuperTableau*

class sage.combinat.super_tableau.**StandardSuperTableaux_all**

Bases: *StandardSuperTableaux*, DisjointUnionEnumeratedSets

All standard super tableaux.

class sage.combinat.super_tableau.**StandardSuperTableaux_shape**(*p*)

Bases: *StandardSuperTableaux*

Standard super tableaux of a fixed shape p.

cardinality()

Return the number of standard super tableaux of given shape.

The standard super tableaux of a fixed shape p are in bijection with the corresponding standard tableaux (under the alphabet relabeling). Refer *sage.combinat.tableau.StandardTableaux_shape* for more details.

EXAMPLES:

```
sage: StandardSuperTableaux([3,2,1]).cardinality()
16
sage: StandardSuperTableaux([2,2]).cardinality()
2
sage: StandardSuperTableaux([5]).cardinality()
1
sage: StandardSuperTableaux([6,5,5,3]).cardinality()
6651216
sage: StandardSuperTableaux([]).cardinality()
1
```

```
>>> from sage.all import *
>>> StandardSuperTableaux([Integer(3),Integer(2),Integer(1)]).cardinality()
16
>>> StandardSuperTableaux([Integer(2),Integer(2)]).cardinality()
2
>>> StandardSuperTableaux([Integer(5)]).cardinality()
1
```

(continues on next page)

(continued from previous page)

```
>>> StandardSuperTableaux([Integer(6),Integer(5),Integer(5),Integer(3)]).
↪cardinality()
6651216
>>> StandardSuperTableaux([]).cardinality()
1
```

class sage.combinat.super_tableau.**StandardSuperTableaux_size**(*n*)

Bases: *StandardSuperTableaux*, DisjointUnionEnumeratedSets

Standard super tableaux of fixed size n.

EXAMPLES:

```
sage: [ t for t in StandardSuperTableaux(1) ]
[[[1']]]
sage: [ t for t in StandardSuperTableaux(2) ]
[[[1', 1]], [[1'], [1]]]
sage: [ t for t in StandardSuperTableaux(3) ]
[[[1', 1, 2']], [[1', 2'], [1]], [[1', 1], [2']], [[1'], [1], [2']]]
sage: StandardSuperTableaux(4)[:]
[[[1', 1, 2', 2]],
 [[1', 2', 2], [1]],
 [[1', 1, 2], [2']],
 [[1', 1, 2'], [2]],
 [[1', 2'], [1, 2]],
 [[1', 1], [2', 2]],
 [[1', 2], [1], [2']],
 [[1', 2'], [1], [2]],
 [[1', 1], [2'], [2]],
 [[1'], [1], [2'], [2]]]
```

```
>>> from sage.all import *
>>> [ t for t in StandardSuperTableaux(Integer(1)) ]
[[[1']]]
>>> [ t for t in StandardSuperTableaux(Integer(2)) ]
[[[1', 1]], [[1'], [1]]]
>>> [ t for t in StandardSuperTableaux(Integer(3)) ]
[[[1', 1, 2']], [[1', 2'], [1]], [[1', 1], [2']], [[1'], [1], [2']]]
>>> StandardSuperTableaux(Integer(4))[:]
[[[1', 1, 2', 2]],
 [[1', 2', 2], [1]],
 [[1', 1, 2], [2']],
 [[1', 1, 2'], [2]],
 [[1', 2'], [1, 2]],
 [[1', 1], [2', 2]],
 [[1', 2], [1], [2']],
 [[1', 2'], [1], [2]],
 [[1', 1], [2'], [2]],
 [[1'], [1], [2'], [2]]]
```

cardinality()

Return the number of all standard super tableaux of size n.

The standard super tableaux of size n are in bijection with the corresponding standard tableaux (under the alphabet relabeling). Refer *sage.combinat.tableau.StandardTableaux_size* for more details.

EXAMPLES:

```
sage: StandardSuperTableaux(3).cardinality()
4
sage: ns = [1,2,3,4,5,6]
sage: sts = [StandardSuperTableaux(n) for n in ns]
sage: all(st.cardinality() == len(st.list()) for st in sts)
True
sage: StandardSuperTableaux(50).cardinality()   # long time
27886995605342342839104615869259776
```

```
>>> from sage.all import *
>>> StandardSuperTableaux(Integer(3)).cardinality()
4
>>> ns = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6)]
>>> sts = [StandardSuperTableaux(n) for n in ns]
>>> all(st.cardinality() == len(st.list()) for st in sts)
True
>>> StandardSuperTableaux(Integer(50)).cardinality()   # long time
27886995605342342839104615869259776
```

5.1.341 Super Partitions

AUTHORS:

- Mike Zabrocki

A super partition of size n and fermionic sector m is a pair consisting of a strict partition of some integer r of length m (that may end in a 0) and an integer partition of $n - r$.

This module provides tools for manipulating super partitions.

Super partitions are the indexing set for symmetric functions in super space.

Super partitions may be input in two different formats: one as a pair consisting of fermionic (strict partition) and a bosonic (partition) part and the other as a list of integer values where the negative entries come first and are listed in strict order followed by the positive values in weak order.

A super partition is displayed as two partitions separated by a semicolon as a default. Super partitions may also be displayed as a weakly increasing sequence of integers that are strict if the numbers are not positive.

These combinatorial objects index the space of symmetric polynomials in two sets of variables, one commuting and one anti-commuting, and they are known as symmetric functions in super space (hence the origin of the name super partitions).

EXAMPLES:

```
sage: SuperPartitions()
Super Partitions
sage: SuperPartitions(2)
Super Partitions of 2
sage: SuperPartitions(2).cardinality()
8
sage: SuperPartitions(4,2)
Super Partitions of 4 and of fermionic sector 2
sage: [[2,0],[1,1]] in SuperPartitions(4,2)
```

(continues on next page)

(continued from previous page)

```
True
sage: [[1,0],[1,1]] in SuperPartitions(4,2)
False
sage: [[1,0],[2,1]] in SuperPartitions(4)
True
sage: [[1,0],[2,2,1]] in SuperPartitions(4)
False
sage: [[1,0],[2,1]] in SuperPartitions()
True
sage: [[1,1],[2,1]] in SuperPartitions()
False
sage: [-2, 0, 1, 1] in SuperPartitions(4,2)
True
sage: [-1, 0, 1, 1] in SuperPartitions(4,2)
False
sage: [-2, -2, 2, 1] in SuperPartitions(7,2)
False
```

```
>>> from sage.all import *
>>> SuperPartitions()
Super Partitions
>>> SuperPartitions(Integer(2))
Super Partitions of 2
>>> SuperPartitions(Integer(2)).cardinality()
8
>>> SuperPartitions(Integer(4),Integer(2))
Super Partitions of 4 and of fermionic sector 2
>>> [[Integer(2),Integer(0)],[Integer(1),Integer(1)]] in SuperPartitions(Integer(4),
↪Integer(2))
True
>>> [[Integer(1),Integer(0)],[Integer(1),Integer(1)]] in SuperPartitions(Integer(4),
↪Integer(2))
False
>>> [[Integer(1),Integer(0)],[Integer(2),Integer(1)]] in SuperPartitions(Integer(4))
True
>>> [[Integer(1),Integer(0)],[Integer(2),Integer(2),Integer(1)]] in␣
↪SuperPartitions(Integer(4))
False
>>> [[Integer(1),Integer(0)],[Integer(2),Integer(1)]] in SuperPartitions()
True
>>> [[Integer(1),Integer(1)],[Integer(2),Integer(1)]] in SuperPartitions()
False
>>> [-Integer(2), Integer(0), Integer(1), Integer(1)] in SuperPartitions(Integer(4),
↪Integer(2))
True
>>> [-Integer(1), Integer(0), Integer(1), Integer(1)] in SuperPartitions(Integer(4),
↪Integer(2))
False
>>> [-Integer(2), -Integer(2), Integer(2), Integer(1)] in SuperPartitions(Integer(7),
↪Integer(2))
False
```

REFERENCES:

- [JL2016]

class sage.combinat.superpartition.**SuperPartition**(*parent*, *lst*, *check=True*, *immutable=True*)

Bases: ClonableArray

A super partition.

A *super partition* of size n and fermionic sector m is a pair consisting of a strict partition of some integer r of length m (that may end in a 0) and an integer partition of $n - r$.

EXAMPLES:

```
sage: sp = SuperPartition([[1,0],[2,2,1]]); sp
[1, 0; 2, 2, 1]
sage: sp[0]
(1, 0)
sage: sp[1]
(2, 2, 1)
sage: sp.fermionic_degree()
2
sage: sp.bosonic_degree()
6
sage: sp.length()
5
sage: sp.conjugate()
[4, 2; ]
```

```
>>> from sage.all import *
>>> sp = SuperPartition([[Integer(1),Integer(0)],[Integer(2),Integer(2),
↪Integer(1)]]); sp
[1, 0; 2, 2, 1]
>>> sp[Integer(0)]
(1, 0)
>>> sp[Integer(1)]
(2, 2, 1)
>>> sp.fermionic_degree()
2
>>> sp.bosonic_degree()
6
>>> sp.length()
5
>>> sp.conjugate()
[4, 2; ]
```

a_part()

The antisymmetric part as a list of strictly decreasing integers.

OUTPUT: list

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).antisymmetric_part()
[3, 1]
sage: SuperPartition([[2,1,0],[3,3]]).antisymmetric_part()
[2, 1, 0]
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).antisymmetric_part()
[3, 1]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).antisymmetric_part()
[2, 1, 0]
```

add_horizontal_border_strip_star(*h*)

Return a list of super partitions that differ from `self` by a horizontal strip.

The notion of horizontal strip comes from the Pieri rule for the Schur-star basis of symmetric functions in super space (see Theorem 7 from [JL2016]).

INPUT:

- `h` – number of cells in the horizontal strip

OUTPUT: list of super partitions

EXAMPLES:

```
sage: SuperPartition([[4,1],[3]]).add_horizontal_border_strip_star(3)
[[3, 1; 7],
 [4, 1; 6],
 [3, 0; 6, 2],
 [3, 1; 6, 1],
 [4, 0; 5, 2],
 [4, 1; 5, 1],
 [3, 0; 5, 3],
 [3, 1; 5, 2],
 [4, 0; 4, 3],
 [4, 1; 4, 2],
 [4, 1; 3, 3]]
sage: SuperPartition([[2,1],[3]]).add_horizontal_border_strip_star(2)
[[2, 1; 5], [2, 0; 4, 2], [2, 1; 4, 1], [2, 0; 3, 3], [2, 1; 3, 2]]
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(4),Integer(1)],[Integer(3)]]).add_horizontal_
↪border_strip_star(Integer(3))
[[3, 1; 7],
 [4, 1; 6],
 [3, 0; 6, 2],
 [3, 1; 6, 1],
 [4, 0; 5, 2],
 [4, 1; 5, 1],
 [3, 0; 5, 3],
 [3, 1; 5, 2],
 [4, 0; 4, 3],
 [4, 1; 4, 2],
 [4, 1; 3, 3]]
>>> SuperPartition([[Integer(2),Integer(1)],[Integer(3)]]).add_horizontal_
↪border_strip_star(Integer(2))
[[2, 1; 5], [2, 0; 4, 2], [2, 1; 4, 1], [2, 0; 3, 3], [2, 1; 3, 2]]
```

add_horizontal_border_strip_star_bar(*h*)

List super partitions that differ from `self` by a horizontal strip.

The notion of horizontal strip comes from the Pieri rule for the Schur-star-bar basis of symmetric functions in super space (see Theorem 10 from [JL2016]).

INPUT:

- `h` – number of cells in the horizontal strip

OUTPUT: list of super partitions

EXAMPLES:

```
sage: SuperPartition([[4,1],[5,4]]).add_horizontal_border_strip_star_bar(3)
[[4, 1; 8, 4],
 [4, 1; 7, 5],
 [4, 2; 7, 4],
 [4, 1; 7, 4, 1],
 [4, 2; 6, 5],
 [4, 1; 6, 5, 1],
 [4, 3; 6, 4],
 [4, 2; 6, 4, 1],
 [4, 1; 6, 4, 2],
 [4, 3; 5, 5],
 [4, 2; 5, 5, 1],
 [4, 1; 5, 5, 2],
 [4, 3; 5, 4, 1],
 [4, 1; 5, 4, 3]]
sage: SuperPartition([[3,1],[5]]).add_horizontal_border_strip_star_bar(2)
[[3, 1; 7],
 [4, 1; 6],
 [3, 2; 6],
 [3, 1; 6, 1],
 [4, 2; 5],
 [4, 1; 5, 1],
 [3, 2; 5, 1],
 [3, 1; 5, 2]]
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(4),Integer(1)],[Integer(5),Integer(4)]]).add_
↪horizontal_border_strip_star_bar(Integer(3))
[[4, 1; 8, 4],
 [4, 1; 7, 5],
 [4, 2; 7, 4],
 [4, 1; 7, 4, 1],
 [4, 2; 6, 5],
 [4, 1; 6, 5, 1],
 [4, 3; 6, 4],
 [4, 2; 6, 4, 1],
 [4, 1; 6, 4, 2],
 [4, 3; 5, 5],
 [4, 2; 5, 5, 1],
 [4, 1; 5, 5, 2],
 [4, 3; 5, 4, 1],
```

(continues on next page)

(continued from previous page)

```
 [4, 1; 5, 4, 3]]
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(5)]]).add_horizontal_
↪border_strip_star_bar(Integer(2))
[[3, 1; 7],
 [4, 1; 6],
 [3, 2; 6],
 [3, 1; 6, 1],
 [4, 2; 5],
 [4, 1; 5, 1],
 [3, 2; 5, 1],
 [3, 1; 5, 2]]
```

antisymmetric_part()

The antisymmetric part as a list of strictly decreasing integers.

OUTPUT: list

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).antisymmetric_part()
[3, 1]
sage: SuperPartition([[2,1,0],[3,3]]).antisymmetric_part()
[2, 1, 0]
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).antisymmetric_part()
[3, 1]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).antisymmetric_part()
[2, 1, 0]
```

bi_degree()

Return the bidegree of `self`, which is a pair consisting of the bosonic and fermionic degree.

OUTPUT: a tuple of two integers

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).bi_degree()
(9, 2)
sage: SuperPartition([[2,1,0],[3,3]]).bi_degree()
(9, 3)
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).bi_degree()
(9, 2)
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).bi_degree()
(9, 3)
```

bosonic_degree()

Return the bosonic degree of `self`.

The *bosonic degree* is the sum of the sizes of the antisymmetric and symmetric parts.

OUTPUT: integer

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).bosonic_degree()
9
sage: SuperPartition([[2,1,0],[3,3]]).bosonic_degree()
9
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).bosonic_degree()
9
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).bosonic_degree()
9
```

bosonic_length()

Return the length of the partition of the symmetric part.

OUTPUT: integer

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).bosonic_length()
3
sage: SuperPartition([[2,1,0],[3,3]]).bosonic_length()
2
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).bosonic_length()
3
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).bosonic_length()
2
```

check()

Check that `self` is a valid super partition.

EXAMPLES:

```
sage: SP = SuperPartition([[1],[1]])
sage: SP.check()
```

```
>>> from sage.all import *
>>> SP = SuperPartition([[Integer(1)],[Integer(1)]])
>>> SP.check()
```

conjugate()

Conjugate of a super partition.

The *conjugate* of a super partition is defined by conjugating the circled diagram.

OUTPUT: a *SuperPartition*

EXAMPLES:

```
sage: SuperPartition([[3, 1, 0], [4, 3, 2, 1]]).conjugate()
[6, 4, 1; 3]
sage: all(sp == sp.conjugate().conjugate() for sp in SuperPartitions(4))
True
sage: all(sp.conjugate() in SuperPartitions(3,2) for sp in SuperPartitions(3,
↪2))
True
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3), Integer(1), Integer(0)], [Integer(4),␣
↪Integer(3), Integer(2), Integer(1)]]).conjugate()
[6, 4, 1; 3]
>>> all(sp == sp.conjugate().conjugate() for sp in␣
↪SuperPartitions(Integer(4)))
True
>>> all(sp.conjugate() in SuperPartitions(Integer(3),Integer(2)) for sp in␣
↪SuperPartitions(Integer(3),Integer(2)))
True
```

degree()

Return the bosonic degree of `self`.

The *bosonic degree* is the sum of the sizes of the antisymmetric and symmetric parts.

OUTPUT: integer

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).bosonic_degree()
9
sage: SuperPartition([[2,1,0],[3,3]]).bosonic_degree()
9
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).bosonic_degree()
9
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).bosonic_degree()
9
```

dominates(*other*)

Return `True` if and only if `self` dominates `other`.

If the symmetric and anti-symmetric parts of `self` and `other` are not the same size then the result is `False`.

EXAMPLES:

```
sage: LA = SuperPartition([[2,1],[2,1,1]])
sage: LA.dominates([[2,1],[3,1]])
False
sage: LA.dominates([[2,1],[1,1,1,1]])
True
```

(continues on next page)

(continued from previous page)

```
sage: LA.dominates([[3],[2,1,1]])
False
sage: LA.dominates([[1],[1]*6])
False
```

```
>>> from sage.all import *
>>> LA = SuperPartition([[Integer(2),Integer(1)],[Integer(2),Integer(1),
↪Integer(1)]])
>>> LA.dominates([[Integer(2),Integer(1)],[Integer(3),Integer(1)]])
False
>>> LA.dominates([[Integer(2),Integer(1)],[Integer(1),Integer(1),Integer(1),
↪Integer(1)]])
True
>>> LA.dominates([[Integer(3)],[Integer(2),Integer(1),Integer(1)]])
False
>>> LA.dominates([[Integer(1)],[Integer(1)]*Integer(6)])
False
```

fermionic_degree()

Return the fermionic degree of `self`.

The *fermionic degree* is the length of the antisymmetric part.

OUTPUT: integer

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).fermionic_degree()
2
sage: SuperPartition([[2,1,0],[3,3]]).fermionic_degree()
3
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).fermionic_degree()
2
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).fermionic_degree()
3
```

fermionic_sector()

Return the fermionic degree of `self`.

The *fermionic degree* is the length of the antisymmetric part.

OUTPUT: integer

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).fermionic_degree()
2
sage: SuperPartition([[2,1,0],[3,3]]).fermionic_degree()
3
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).fermionic_degree()
2
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).fermionic_degree()
3
```

static from_circled_diagram(*shape*, *corners*)

Construct a super partition from a circled diagram.

A circled diagram consists of a partition of the concatenation of the antisymmetric and symmetric parts and a list of addable cells of the partition which indicate the location of the circled cells.

INPUT:

- `shape` – a partition or list of integers
- `corners` – list of removable cells of `shape`

OUTPUT: a *SuperPartition*

EXAMPLES:

```
sage: SuperPartition.from_circled_diagram([3, 2, 2, 1, 1], [(0, 3), (3, 1)])
[3, 1; 2, 2, 1]
sage: SuperPartition.from_circled_diagram([3, 3, 2, 1], [(2, 2), (3, 1), (4,␣
↪0)])
[2, 1, 0; 3, 3]
sage: from_cd = SuperPartition.from_circled_diagram
sage: all(sp == from_cd(*sp.to_circled_diagram()) for sp in␣
↪SuperPartitions(4))
True
```

```
>>> from sage.all import *
>>> SuperPartition.from_circled_diagram([Integer(3), Integer(2), Integer(2),␣
↪Integer(1), Integer(1)], [(Integer(0), Integer(3)), (Integer(3),␣
↪Integer(1))])
[3, 1; 2, 2, 1]
>>> SuperPartition.from_circled_diagram([Integer(3), Integer(3), Integer(2),␣
↪Integer(1)], [(Integer(2), Integer(2)), (Integer(3), Integer(1)),␣
↪(Integer(4), Integer(0))])
[2, 1, 0; 3, 3]
>>> from_cd = SuperPartition.from_circled_diagram
>>> all(sp == from_cd(*sp.to_circled_diagram()) for sp in␣
↪SuperPartitions(Integer(4)))
True
```

length()

Return the length of `self`, which is the sum of the lengths of the antisymmetric and symmetric part.

OUTPUT: integer

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).length()
5
sage: SuperPartition([[2,1,0],[3,3]]).length()
5
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
→Integer(1)]]).length()
5
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
→Integer(3)]]).length()
5
```

s_part()

The symmetric part as a list of weakly decreasing integers.

OUTPUT: list

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).symmetric_part()
[2, 2, 1]
sage: SuperPartition([[2,1,0],[3,3]]).symmetric_part()
[3, 3]
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
→Integer(1)]]).symmetric_part()
[2, 2, 1]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
→Integer(3)]]).symmetric_part()
[3, 3]
```

shape_circled_diagram()

A concatenated partition with an extra cell for each antisymmetric part

OUTPUT: a partition

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).shape_circled_diagram()
[4, 2, 2, 2, 1]
sage: SuperPartition([[2,1,0],[3,3]]).shape_circled_diagram()
[3, 3, 3, 2, 1]
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
→Integer(1)]]).shape_circled_diagram()
[4, 2, 2, 2, 1]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
→Integer(3)]]).shape_circled_diagram()
[3, 3, 3, 2, 1]
```

sign()

Return the sign of a permutation of cycle type the symmetric part of `self`.

OUTPUT: either 1 or -1

EXAMPLES:

```
sage: SuperPartition([[1,0],[3,1,1]]).sign()
-1
sage: SuperPartition([[1,0],[3,2,1]]).sign()
1
sage: sum(sp.sign()/sp.zee() for sp in SuperPartitions(6,0))
0
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(1),Integer(0)],[Integer(3),Integer(1),
↪Integer(1)]]).sign()
-1
>>> SuperPartition([[Integer(1),Integer(0)],[Integer(3),Integer(2),
↪Integer(1)]]).sign()
1
>>> sum(sp.sign()/sp.zee() for sp in SuperPartitions(Integer(6),Integer(0)))
0
```

symmetric_part()

The symmetric part as a list of weakly decreasing integers.

OUTPUT: list

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).symmetric_part()
[2, 2, 1]
sage: SuperPartition([[2,1,0],[3,3]]).symmetric_part()
[3, 3]
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).symmetric_part()
[2, 2, 1]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).symmetric_part()
[3, 3]
```

to_circled_diagram()

The shape of the circled diagram and a list of addable cells

A circled diagram consists of a partition for the outer shape and a list of removable cells of the partition indicating the location of the circled cells

OUTPUT: list consisting of a partition and a list of pairs of integers

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).to_circled_diagram()
[[3, 2, 2, 1, 1], [(0, 3), (3, 1)]]
```

(continues on next page)

(continued from previous page)

```
sage: SuperPartition([[2,1,0],[3,3]]).to_circled_diagram()
[[3, 3, 2, 1], [(2, 2), (3, 1), (4, 0)]]
sage: from_cd = SuperPartition.from_circled_diagram
sage: all(sp == from_cd(*sp.to_circled_diagram()) for sp in␣
↪SuperPartitions(4))
True
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).to_circled_diagram()
[[3, 2, 2, 1, 1], [(0, 3), (3, 1)]]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).to_circled_diagram()
[[3, 3, 2, 1], [(2, 2), (3, 1), (4, 0)]]
>>> from_cd = SuperPartition.from_circled_diagram
>>> all(sp == from_cd(*sp.to_circled_diagram()) for sp in␣
↪SuperPartitions(Integer(4)))
True
```

to_composition()

Concatenate the antisymmetric and symmetric parts to a composition.

OUTPUT:

- a (possibly weak) composition

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).to_composition()
[3, 1, 2, 2, 1]
sage: SuperPartition([[2,1,0],[3,3]]).to_composition()
[2, 1, 0, 3, 3]
sage: SuperPartition([[2,1,0],[3,3]]).to_composition().parent()
Compositions of nonnegative integers
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).to_composition()
[3, 1, 2, 2, 1]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).to_composition()
[2, 1, 0, 3, 3]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).to_composition().parent()
Compositions of nonnegative integers
```

to_list()

The list of two lists with the antisymmetric and symmetric parts.

EXAMPLES:

```
sage: SuperPartition([[1],[1]]).to_list()
[[1], [1]]
```

(continues on next page)

(continued from previous page)

```
sage: SuperPartition([[],[1]]).to_list()
[[], [1]]
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(1)],[Integer(1)]]).to_list()
[[1], [1]]
>>> SuperPartition([[],[Integer(1)]]).to_list()
[[], [1]]
```

to_partition()

Concatenate and sort the antisymmetric and symmetric parts to a partition.

OUTPUT: a partition

EXAMPLES:

```
sage: SuperPartition([[3,1],[2,2,1]]).to_partition()
[3, 2, 2, 1, 1]
sage: SuperPartition([[2,1,0],[3,3]]).to_partition()
[3, 3, 2, 1]
sage: SuperPartition([[2,1,0],[3,3]]).to_partition().parent()
Partitions
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(3),Integer(1)],[Integer(2),Integer(2),
↪Integer(1)]]).to_partition()
[3, 2, 2, 1, 1]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).to_partition()
[3, 3, 2, 1]
>>> SuperPartition([[Integer(2),Integer(1),Integer(0)],[Integer(3),
↪Integer(3)]]).to_partition().parent()
Partitions
```

zee()

Return the centralizer size of a permutation of cycle type symmetric part of `self`.

OUTPUT: a positive integer

EXAMPLES:

```
sage: SuperPartition([[1,0],[3,1,1]]).zee()
6
sage: SuperPartition([[1],[2,2,1]]).zee()
8
sage: sum(1/sp.zee() for sp in SuperPartitions(6,0))
1
```

```
>>> from sage.all import *
>>> SuperPartition([[Integer(1),Integer(0)],[Integer(3),Integer(1),
↪Integer(1)]]).zee()
6
>>> SuperPartition([[Integer(1)],[Integer(2),Integer(2),Integer(1)]]).zee()
```

(continues on next page)

(continued from previous page)

```
8
>>> sum(Integer(1)/sp.zee() for sp in SuperPartitions(Integer(6),Integer(0)))
1
```

class sage.combinat.superpartition.**SuperPartitions**(*is_infinite=False*)

Bases: UniqueRepresentation, Parent

Super partitions.

A super partition of size n and fermionic sector m is a pair consisting of a strict partition of some integer r of length m (that may end in a 0) and an integer partition of $n - r$.

INPUT:

- n – integer (default: None)
- m – if n is specified, an integer (optional: default None)

Super partitions are the indexing set for symmetric functions in super space.

EXAMPLES:

```
sage: SuperPartitions()
Super Partitions
sage: SuperPartitions(2)
Super Partitions of 2
sage: SuperPartitions(2).cardinality()
8
sage: SuperPartitions(4,2)
Super Partitions of 4 and of fermionic sector 2
sage: [[2,0],[1,1]] in SuperPartitions(4,2)
True
sage: [[1,0],[1,1]] in SuperPartitions(4,2)
False
sage: [[1,0],[2,1]] in SuperPartitions(4)
True
sage: [[1,0],[2,2,1]] in SuperPartitions(4)
False
sage: [[1,0],[2,1]] in SuperPartitions()
True
sage: [[1,1],[2,1]] in SuperPartitions()
False
```

```
>>> from sage.all import *
>>> SuperPartitions()
Super Partitions
>>> SuperPartitions(Integer(2))
Super Partitions of 2
>>> SuperPartitions(Integer(2)).cardinality()
8
>>> SuperPartitions(Integer(4),Integer(2))
Super Partitions of 4 and of fermionic sector 2
>>> [[Integer(2),Integer(0)],[Integer(1),Integer(1)]] in␣
↪SuperPartitions(Integer(4),Integer(2))
True
```

(continues on next page)

(continued from previous page)

```
>>> [[Integer(1),Integer(0)],[Integer(1),Integer(1)]] in␣
↪SuperPartitions(Integer(4),Integer(2))
False
>>> [[Integer(1),Integer(0)],[Integer(2),Integer(1)]] in␣
↪SuperPartitions(Integer(4))
True
>>> [[Integer(1),Integer(0)],[Integer(2),Integer(2),Integer(1)]] in␣
↪SuperPartitions(Integer(4))
False
>>> [[Integer(1),Integer(0)],[Integer(2),Integer(1)]] in SuperPartitions()
True
>>> [[Integer(1),Integer(1)],[Integer(2),Integer(1)]] in SuperPartitions()
False
```

Element

alias of `SuperPartition`

options = Current options for SuperPartition - display: default

class sage.combinat.superpartition.**SuperPartitions_all**

Bases: `SuperPartitions`

Initialize `self`.

class sage.combinat.superpartition.**SuperPartitions_n**(*n*)

Bases: `SuperPartitions`

Initialize `self`.

class sage.combinat.superpartition.**SuperPartitions_n_m**(*n*, *m*)

Bases: `SuperPartitions`

Initialize `self`.

5.1.342 Symmetric Group Algebra

sage.combinat.symmetric_group_algebra.**HeckeAlgebraSymmetricGroupT**(*R*, *n*, *q=None*)

Return the Hecke algebra of the symmetric group S_n on the T-basis with quantum parameter `q` over the ring R.

If R is a commutative ring and q is an invertible element of R, and if n is a nonnegative integer, then the Hecke algebra of the symmetric group S_n over R with quantum parameter q is defined as the algebra generated by the generators $T_1, T_2, \ldots, T_{n-1}$ with relations

$$T_i T_{i+1} T_i = T_{i+1} T_i T_{i+1}$$

for all $i < n-1$ ("braid relations"),

$$T_i T_j = T_j T_i$$

for all i and j such that $|i-j| > 1$ ("locality relations"), and

$$T_i^2 = q + (q-1) T_i$$

for all i (the "quadratic relations", also known in the form $(T_i+1)(T_i-q) = 0$). (This is only one of several existing definitions in literature, not all of which are fully equivalent. We are following the conventions of [Go1993].) For any permutation $w \in S_n$, we can define an element T_w of this Hecke algebra by setting $T_w = T_{i_1} T_{i_2} \cdots T_{i_k}$, where $w = s_{i_1} s_{i_2} \cdots s_{i_k}$ is a reduced word for w (with s_i meaning the transposition $(i, i+1)$, and the product of permutations being evaluated by first applying s_{i_k}, then $s_{i_{k-1}}$, etc.). This element is independent of the choice of the reduced decomposition, and can be computed in Sage by calling `H[w]` where `H` is the Hecke algebra and `w` is the permutation.

The Hecke algebra of the symmetric group S_n with quantum parameter q over R can be seen as a deformation of the group algebra RS_n; indeed, it becomes RS_n when $q = 1$.

Warning: The multiplication on the Hecke algebra of the symmetric group does *not* follow the global option `mult` of the `Permutations` class (see `options()`). It is always as defined above. It does not match the default option (`mult=l2r`) of the symmetric group algebra!

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3)
Hecke algebra of the symmetric group of order 3 on the T basis over Univariate␣
↪Polynomial Ring in q over Rational Field
```

```
>>> from sage.all import *
>>> HeckeAlgebraSymmetricGroupT(QQ, Integer(3))
Hecke algebra of the symmetric group of order 3 on the T basis over Univariate␣
↪Polynomial Ring in q over Rational Field
```

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3, 2)
Hecke algebra of the symmetric group of order 3 with q=2 on the T basis over␣
↪Rational Field
```

```
>>> from sage.all import *
>>> HeckeAlgebraSymmetricGroupT(QQ, Integer(3), Integer(2))
Hecke algebra of the symmetric group of order 3 with q=2 on the T basis over␣
↪Rational Field
```

The multiplication on the Hecke algebra follows a different convention than the one on the symmetric group algebra does by default:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: H3([1,3,2]) * H3([2,1,3])
T[3, 1, 2]
sage: S3 = SymmetricGroupAlgebra(QQ, 3)
sage: S3([1,3,2]) * S3([2,1,3])
[2, 3, 1]

sage: TestSuite(H3).run()
```

```
>>> from sage.all import *
>>> H3 = HeckeAlgebraSymmetricGroupT(QQ, Integer(3))
>>> H3([Integer(1),Integer(3),Integer(2)]) * H3([Integer(2),Integer(1),
↪Integer(3)])
T[3, 1, 2]
>>> S3 = SymmetricGroupAlgebra(QQ, Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> S3([Integer(1),Integer(3),Integer(2)]) * S3([Integer(2),Integer(1),
↪Integer(3)])
[2, 3, 1]

>>> TestSuite(H3).run()
```

Note

IwahoriHeckeAlgebra gives a different implementation of the Iwahori-Hecke algebras of a Coxeter system (W, S). This includes the Hecke algebras of the symmetric group as special case.

class sage.combinat.symmetric_group_algebra.**HeckeAlgebraSymmetricGroup_generic**(*R*, *n*, *q=None*)

Bases: *CombinatorialFreeModule*

one_basis()

Return the identity permutation.

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3).one()  # indirect doctest
T[1, 2, 3]
```

```
>>> from sage.all import *
>>> HeckeAlgebraSymmetricGroupT(QQ, Integer(3)).one()  # indirect doctest
T[1, 2, 3]
```

q()

Return the variable or parameter q.

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3).q()
q
sage: HeckeAlgebraSymmetricGroupT(QQ, 3, 2).q()
2
```

```
>>> from sage.all import *
>>> HeckeAlgebraSymmetricGroupT(QQ, Integer(3)).q()
q
>>> HeckeAlgebraSymmetricGroupT(QQ, Integer(3), Integer(2)).q()
2
```

class sage.combinat.symmetric_group_algebra.**HeckeAlgebraSymmetricGroup_t**(*R*, *n*, *q=None*)

Bases: *HeckeAlgebraSymmetricGroup_generic*

algebra_generators()

Return the generators of the algebra.

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ,3).algebra_generators()
[T[2, 1, 3], T[1, 3, 2]]
```

```
>>> from sage.all import *
>>> HeckeAlgebraSymmetricGroupT(QQ,Integer(3)).algebra_generators()
[T[2, 1, 3], T[1, 3, 2]]
```

jucys_murphy(*k*)

Return the Jucys-Murphy element J_k of the Hecke algebra.

These Jucys-Murphy elements are defined by

$$J_k = (T_{k-1}T_{k-2}\cdots T_1)(T_1T_2\cdots T_{k-1}).$$

More explicitly,

$$J_k = q^{k-1} + \sum_{l=1}^{k-1}(q^l - q^{l-1})T_{(l,k)}.$$

For generic q, the J_k generate a maximal commutative sub-algebra of the Hecke algebra.

Warning

The specialization $q = 1$ does *not* map these elements J_k to the Young-Jucys-Murphy elements of the group algebra RS_n. (Instead, it maps the "reduced" Jucys-Murphy elements $(J_k - q^{k-1})/(q-1)$ to the Young-Jucys-Murphy elements of RS_n.)

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ,3)
sage: j2 = H3.jucys_murphy(2); j2
q*T[1, 2, 3] + (q-1)*T[2, 1, 3]
sage: j3 = H3.jucys_murphy(3); j3
q^2*T[1, 2, 3] + (q^2-q)*T[1, 3, 2] + (q-1)*T[3, 2, 1]
sage: j2*j3 == j3*j2
True
sage: j0 = H3.jucys_murphy(1); j0 == H3.one()
True
sage: H3.jucys_murphy(0)
Traceback (most recent call last):
...
ValueError: k (= 0) must be between 1 and n (= 3)
```

```
>>> from sage.all import *
>>> H3 = HeckeAlgebraSymmetricGroupT(QQ,Integer(3))
>>> j2 = H3.jucys_murphy(Integer(2)); j2
q*T[1, 2, 3] + (q-1)*T[2, 1, 3]
>>> j3 = H3.jucys_murphy(Integer(3)); j3
q^2*T[1, 2, 3] + (q^2-q)*T[1, 3, 2] + (q-1)*T[3, 2, 1]
>>> j2*j3 == j3*j2
True
>>> j0 = H3.jucys_murphy(Integer(1)); j0 == H3.one()
```

(continues on next page)

(continued from previous page)

```
True
>>> H3.jucys_murphy(Integer(0))
Traceback (most recent call last):
...
ValueError: k (= 0) must be between 1 and n (= 3)
```

product_on_basis(*perm1*, *perm2*)

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3, 1)
sage: a = H3([2,1,3])+2*H3([1,2,3])-H3([3,2,1])
sage: a^2 #indirect doctest
6*T[1, 2, 3] + 4*T[2, 1, 3] - T[2, 3, 1]
- T[3, 1, 2] - 4*T[3, 2, 1]
```

```
>>> from sage.all import *
>>> H3 = HeckeAlgebraSymmetricGroupT(QQ, Integer(3), Integer(1))
>>> a = H3([Integer(2),Integer(1),Integer(3)])+Integer(2)*H3([Integer(1),
↪Integer(2),Integer(3)])-H3([Integer(3),Integer(2),Integer(1)])
>>> a**Integer(2) #indirect doctest
6*T[1, 2, 3] + 4*T[2, 1, 3] - T[2, 3, 1]
- T[3, 1, 2] - 4*T[3, 2, 1]
```

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = QS3([2,1,3])+2*QS3([1,2,3])-QS3([3,2,1])
sage: a^2
6*[1, 2, 3] + 4*[2, 1, 3] - [2, 3, 1] - [3, 1, 2] - 4*[3, 2, 1]
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> a = QS3([Integer(2),Integer(1),Integer(3)])+Integer(2)*QS3([Integer(1),
↪Integer(2),Integer(3)])-QS3([Integer(3),Integer(2),Integer(1)])
>>> a**Integer(2)
6*[1, 2, 3] + 4*[2, 1, 3] - [2, 3, 1] - [3, 1, 2] - 4*[3, 2, 1]
```

t(*i*)

Return the element T_i of the Hecke algebra `self`.

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ,3)
sage: H3.t(1)
T[2, 1, 3]
sage: H3.t(2)
T[1, 3, 2]
sage: H3.t(0)
Traceback (most recent call last):
...
ValueError: i (= 0) must be between 1 and n-1 (= 2)
```

```
>>> from sage.all import *
>>> H3 = HeckeAlgebraSymmetricGroupT(QQ,Integer(3))
>>> H3.t(Integer(1))
T[2, 1, 3]
>>> H3.t(Integer(2))
T[1, 3, 2]
>>> H3.t(Integer(0))
Traceback (most recent call last):
...
ValueError: i (= 0) must be between 1 and n-1 (= 2)
```

t_action(*a*, *i*)

Return the product $T_i \cdot a$.

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: a = H3([2,1,3])+2*H3([1,2,3])
sage: H3.t_action(a, 1)
q*T[1, 2, 3] + (q+1)*T[2, 1, 3]
sage: H3.t(1)*a
q*T[1, 2, 3] + (q+1)*T[2, 1, 3]
```

```
>>> from sage.all import *
>>> H3 = HeckeAlgebraSymmetricGroupT(QQ, Integer(3))
>>> a = H3([Integer(2),Integer(1),Integer(3)])+Integer(2)*H3([Integer(1),
↪Integer(2),Integer(3)])
>>> H3.t_action(a, Integer(1))
q*T[1, 2, 3] + (q+1)*T[2, 1, 3]
>>> H3.t(Integer(1))*a
q*T[1, 2, 3] + (q+1)*T[2, 1, 3]
```

t_action_on_basis(*perm*, *i*)

Return the product $T_i \cdot T_{perm}$, where `perm` is a permutation in the symmetric group S_n.

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: H3.t_action_on_basis(Permutation([2,1,3]), 1)
q*T[1, 2, 3] + (q-1)*T[2, 1, 3]
sage: H3.t_action_on_basis(Permutation([1,2,3]), 1)
T[2, 1, 3]
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3, 1)
sage: H3.t_action_on_basis(Permutation([2,1,3]), 1)
T[1, 2, 3]
sage: H3.t_action_on_basis(Permutation([1,3,2]), 2)
T[1, 2, 3]
```

```
>>> from sage.all import *
>>> H3 = HeckeAlgebraSymmetricGroupT(QQ, Integer(3))
>>> H3.t_action_on_basis(Permutation([Integer(2),Integer(1),Integer(3)]),␣
↪Integer(1))
q*T[1, 2, 3] + (q-1)*T[2, 1, 3]
```

(continues on next page)

(continued from previous page)

```
>>> H3.t_action_on_basis(Permutation([Integer(1),Integer(2),Integer(3)]),␣
↪Integer(1))
T[2, 1, 3]
>>> H3 = HeckeAlgebraSymmetricGroupT(QQ, Integer(3), Integer(1))
>>> H3.t_action_on_basis(Permutation([Integer(2),Integer(1),Integer(3)]),␣
↪Integer(1))
T[1, 2, 3]
>>> H3.t_action_on_basis(Permutation([Integer(1),Integer(3),Integer(2)]),␣
↪Integer(2))
T[1, 2, 3]
```

class sage.combinat.symmetric_group_algebra.**KLCellularBasis**(*SGA*)

Bases: *SGACellularBasis*

The Kazhdan-Lusztig C' basis (at $q = 1$) of the symmetric group algebra realized as a `cellular basis`

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 5)
sage: KL = SGA.kazhdan_lusztig_cellular_basis()
sage: for la in KL.simple_module_parameterization():
....:     CM = KL.cell_module(la)
....:     print(la, CM.dimension(), CM.simple_module().dimension())
[2, 2, 1] 5 4
[3, 1, 1] 6 6
[3, 2] 5 1
[4, 1] 4 4
[5] 1 1
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(5))
>>> KL = SGA.kazhdan_lusztig_cellular_basis()
>>> for la in KL.simple_module_parameterization():
...     CM = KL.cell_module(la)
...     print(la, CM.dimension(), CM.simple_module().dimension())
[2, 2, 1] 5 4
[3, 1, 1] 6 6
[3, 2] 5 1
[4, 1] 4 4
[5] 1 1
```

class sage.combinat.symmetric_group_algebra.**MurphyBasis**(*SGA*)

Bases: *SGACellularBasis*

The Murphy basis of a symmetric group algebra.

Let R be a commutative ring, and let $A = R[S_n]$ denote the group algebra (over R) of S_n. The *Murphy basis* is the basis of A defined as follows. Let S, T be standard tableaux of shape λ. Define T^λ as the standard tableau of shape λ with the first row filled with $1, \dots, \lambda_1$, the second row $\lambda_1 + 1, \dots, \lambda_1 + \lambda_2$, and so on. Let $d(S)$ be the unique permutation such that $S = T^\lambda d(S)$ under the natural action. Then the Murphy basis element indexed by S and T is

$$M_{S'T'} = d(S)^{-1} R_\lambda d(T),$$

where S' denotes the conjugate tableau. The Murphy basis is a `cellular basis` of A.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 5)
sage: M = SGA.murphy_basis()
sage: for la in M.simple_module_parameterization():
....:     CM = M.cell_module(la)
....:     print(la, CM.dimension(), CM.simple_module().dimension())
[2, 2, 1] 5 4
[3, 1, 1] 6 6
[3, 2] 5 1
[4, 1] 4 4
[5] 1 1
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(5))
>>> M = SGA.murphy_basis()
>>> for la in M.simple_module_parameterization():
...     CM = M.cell_module(la)
...     print(la, CM.dimension(), CM.simple_module().dimension())
[2, 2, 1] 5 4
[3, 1, 1] 6 6
[3, 2] 5 1
[4, 1] 4 4
[5] 1 1
```

REFERENCES:

- [DJM1998]
- [Mathas2004]

class sage.combinat.symmetric_group_algebra.**SGACellularBasis**(*SGA*)

Bases: `CellularBasis`

A cellular basis of the symmetric group algebra.

one()

Return the element 1 in `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 4)
sage: M = SGA.murphy_basis()
sage: M.one()
C([4], [[1, 2, 3, 4]], [[1, 2, 3, 4]])
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(4))
>>> M = SGA.murphy_basis()
>>> M.one()
C([4], [[1, 2, 3, 4]], [[1, 2, 3, 4]])
```

one_basis()

Return the index of the basis element for the multiplicative identity.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 4)
sage: M = SGA.murphy_basis()
sage: M.one_basis()
([4], [[1, 2, 3, 4]], [[1, 2, 3, 4]])
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(4))
>>> M = SGA.murphy_basis()
>>> M.one_basis()
([4], [[1, 2, 3, 4]], [[1, 2, 3, 4]])
```

sage.combinat.symmetric_group_algebra.**SymmetricGroupAlgebra**(*R*, *W*, *category=None*)

Return the symmetric group algebra of order `W` over the ring `R`.

INPUT:

- `W` – a symmetric group; alternatively an integer n can be provided, as shorthand for `Permutations(n)`.
- `R` – a base ring
- `category` – a category (default: the category of `W`)

This supports several implementations of the symmetric group. At this point this has been tested with `W=Permutations(n)` and `W=SymmetricGroup(n)`.

Warning

Some features are failing in the latter case, in particular if the domain of the symmetric group is not $1, \ldots, n$.

Note

The brave can also try setting `W=WeylGroup(['A',n-1])`, but little support for this currently exists.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3); QS3
Symmetric group algebra of order 3 over Rational Field
sage: QS3(1)
[1, 2, 3]
sage: QS3(2)
2*[1, 2, 3]
sage: basis = [QS3(p) for p in Permutations(3)]
sage: a = sum(basis); a
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: a^2
6*[1, 2, 3] + 6*[1, 3, 2] + 6*[2, 1, 3] + 6*[2, 3, 1]
+ 6*[3, 1, 2] + 6*[3, 2, 1]
sage: a^2 == 6*a
True
sage: b = QS3([3, 1, 2])
sage: b
[3, 1, 2]
```

(continues on next page)

(continued from previous page)

```
sage: b*a
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: b*a == a
True
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3)); QS3
Symmetric group algebra of order 3 over Rational Field
>>> QS3(Integer(1))
[1, 2, 3]
>>> QS3(Integer(2))
2*[1, 2, 3]
>>> basis = [QS3(p) for p in Permutations(Integer(3))]
>>> a = sum(basis); a
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
>>> a**Integer(2)
6*[1, 2, 3] + 6*[1, 3, 2] + 6*[2, 1, 3] + 6*[2, 3, 1]
+ 6*[3, 1, 2] + 6*[3, 2, 1]
>>> a**Integer(2) == Integer(6)*a
True
>>> b = QS3([Integer(3), Integer(1), Integer(2)])
>>> b
[3, 1, 2]
>>> b*a
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
>>> b*a == a
True
```

We now construct the symmetric group algebra by providing explicitly the underlying group:

```
sage: SGA = SymmetricGroupAlgebra(QQ, Permutations(4)); SGA
Symmetric group algebra of order 4 over Rational Field
sage: SGA.group()
Standard permutations of 4
sage: SGA.an_element()
[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 3*[1, 3, 2, 4] + [4, 1, 2, 3]

sage: SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(4)); SGA
Symmetric group algebra of order 4 over Rational Field
sage: SGA.group()
Symmetric group of order 4! as a permutation group
sage: SGA.an_element()
() + (2,3,4) + 2*(1,3)(2,4) + 3*(1,4)(2,3)

sage: SGA = SymmetricGroupAlgebra(QQ, WeylGroup(["A",3], prefix='s')); SGA
Symmetric group algebra of order 4 over Rational Field
sage: SGA.group()
Weyl Group of type ['A', 3] (as a matrix group acting
on the ambient space)
sage: SGA.an_element()
s1*s2*s3 + 3*s2*s3*s1*s2 + 2*s3*s1 + 1
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Permutations(Integer(4))); SGA
Symmetric group algebra of order 4 over Rational Field
>>> SGA.group()
Standard permutations of 4
>>> SGA.an_element()
[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 3*[1, 3, 2, 4] + [4, 1, 2, 3]

>>> SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(Integer(4))); SGA
Symmetric group algebra of order 4 over Rational Field
>>> SGA.group()
Symmetric group of order 4! as a permutation group
>>> SGA.an_element()
() + (2,3,4) + 2*(1,3)(2,4) + 3*(1,4)(2,3)

>>> SGA = SymmetricGroupAlgebra(QQ, WeylGroup(["A",Integer(3)], prefix='s')); SGA
Symmetric group algebra of order 4 over Rational Field
>>> SGA.group()
Weyl Group of type ['A', 3] (as a matrix group acting
on the ambient space)
>>> SGA.an_element()
s1*s2*s3 + 3*s2*s3*s1*s2 + 2*s3*s1 + 1
```

The preferred way to construct the symmetric group algebra is to go through the usual `algebra` method:

```
sage: SGA = Permutations(3).algebra(QQ); SGA
Symmetric group algebra of order 3 over Rational Field
sage: SGA.group()
Standard permutations of 3

sage: SGA = SymmetricGroup(3).algebra(QQ); SGA
Symmetric group algebra of order 3 over Rational Field
sage: SGA.group()
Symmetric group of order 3! as a permutation group
```

```
>>> from sage.all import *
>>> SGA = Permutations(Integer(3)).algebra(QQ); SGA
Symmetric group algebra of order 3 over Rational Field
>>> SGA.group()
Standard permutations of 3

>>> SGA = SymmetricGroup(Integer(3)).algebra(QQ); SGA
Symmetric group algebra of order 3 over Rational Field
>>> SGA.group()
Symmetric group of order 3! as a permutation group
```

The canonical embedding from the symmetric group algebra of order n to the symmetric group algebra of order $p > n$ is available as a coercion:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: QS4.coerce_map_from(QS3)
Generic morphism:
```

(continues on next page)

(continued from previous page)

```
  From: Symmetric group algebra of order 3 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field

sage: x3  = QS3([3,1,2]) + 2 * QS3([2,3,1]); x3
2*[2, 3, 1] + [3, 1, 2]
sage: QS4(x3)
2*[2, 3, 1, 4] + [3, 1, 2, 4]
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> QS4 = SymmetricGroupAlgebra(QQ, Integer(4))
>>> QS4.coerce_map_from(QS3)
Generic morphism:
  From: Symmetric group algebra of order 3 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field

>>> x3  = QS3([Integer(3),Integer(1),Integer(2)]) + Integer(2) * QS3([Integer(2),
↪Integer(3),Integer(1)]); x3
2*[2, 3, 1] + [3, 1, 2]
>>> QS4(x3)
2*[2, 3, 1, 4] + [3, 1, 2, 4]
```

This allows for mixed expressions:

```
sage: x4  = 3 * QS4([3, 1, 4, 2])
sage: x3 + x4
2*[2, 3, 1, 4] + [3, 1, 2, 4] + 3*[3, 1, 4, 2]

sage: QS0 = SymmetricGroupAlgebra(QQ, 0)
sage: QS1 = SymmetricGroupAlgebra(QQ, 1)
sage: x0 = QS0([])
sage: x1 = QS1([1])
sage: x0 * x1
[1]
sage: x3 - (2*x0 + x1) - x4
-3*[1, 2, 3, 4] + 2*[2, 3, 1, 4] + [3, 1, 2, 4] - 3*[3, 1, 4, 2]
```

```
>>> from sage.all import *
>>> x4  = Integer(3) * QS4([Integer(3), Integer(1), Integer(4), Integer(2)])
>>> x3 + x4
2*[2, 3, 1, 4] + [3, 1, 2, 4] + 3*[3, 1, 4, 2]

>>> QS0 = SymmetricGroupAlgebra(QQ, Integer(0))
>>> QS1 = SymmetricGroupAlgebra(QQ, Integer(1))
>>> x0 = QS0([])
>>> x1 = QS1([Integer(1)])
>>> x0 * x1
[1]
>>> x3 - (Integer(2)*x0 + x1) - x4
-3*[1, 2, 3, 4] + 2*[2, 3, 1, 4] + [3, 1, 2, 4] - 3*[3, 1, 4, 2]
```

Caveat: to achieve this, constructing `SymmetricGroupAlgebra(QQ, 10)` currently triggers the construction of all symmetric group algebras of smaller order. Is this a feature we really want to have?

> **Warning**
>
> The semantics of multiplication in symmetric group algebras with index set `Permutations(n)` is determined by the order in which permutations are multiplied, which currently defaults to "in such a way that multiplication is associative with permutations acting on integers from the right", but can be changed to the opposite order at runtime by setting the global variable `Permutations.options['mult']` (see `sage.combinat.permutation.Permutations.options()`). On the other hand, the semantics of multiplication in symmetric group algebras with index set `SymmetricGroup(n)` does not depend on this global variable. (This has the awkward consequence that the coercions between these two sorts of symmetric group algebras do not respect multiplication when this global variable is set to `'r2l'`.) In view of this, it is recommended that code not rely on the usual multiplication function, but rather use the methods `left_action_product()` and `right_action_product()` for multiplying permutations (these methods don't depend on the setting). See Issue #14885 for more information.

We conclude by constructing the algebra of the symmetric group as a monoid algebra:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3, category=Monoids())
sage: QS3.category()
Category of finite dimensional cellular monoid algebras
over Rational Field
sage: TestSuite(QS3).run(skip=['_test_construction'])
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3), category=Monoids())
>>> QS3.category()
Category of finite dimensional cellular monoid algebras
over Rational Field
>>> TestSuite(QS3).run(skip=['_test_construction'])
```

class `sage.combinat.symmetric_group_algebra.`**`SymmetricGroupAlgebra_n`**(*R*, *W*, *category*)

Bases: `GroupAlgebra_class`

`algebra_generators`()

Return generators of this group algebra (as algebra) as a list of permutations.

The generators used for the group algebra of S_n are the transposition $(2,1)$ and the n-cycle $(1,2,\ldots,n)$, unless $n \leq 1$ (in which case no generators are needed).

EXAMPLES:

```
sage: SymmetricGroupAlgebra(ZZ,5).algebra_generators()
Family ([2, 1, 3, 4, 5], [2, 3, 4, 5, 1])

sage: SymmetricGroupAlgebra(QQ,0).algebra_generators()
Family ()

sage: SymmetricGroupAlgebra(QQ,1).algebra_generators()
Family ()
```

```
>>> from sage.all import *
>>> SymmetricGroupAlgebra(ZZ,Integer(5)).algebra_generators()
```

(continues on next page)

(continued from previous page)

```
Family ([2, 1, 3, 4, 5], [2, 3, 4, 5, 1])

>>> SymmetricGroupAlgebra(QQ,Integer(0)).algebra_generators()
Family ()

>>> SymmetricGroupAlgebra(QQ,Integer(1)).algebra_generators()
Family ()
```

antipode(*x*)

Return the image of the element `x` of `self` under the antipode of the Hopf algebra `self` (where the comultiplication is the usual one on a group algebra).

Explicitly, this is obtained by replacing each permutation σ by σ^{-1} in `x` while keeping all coefficients as they are.

EXAMPLES:

```
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: QS4.antipode(2 * QS4([1, 3, 4, 2]) - 1/2 * QS4([1, 4, 2, 3]))
-1/2*[1, 3, 4, 2] + 2*[1, 4, 2, 3]
sage: all( QS4.antipode(QS4(p)) == QS4(p.inverse())
....:      for p in Permutations(4) )
True

sage: ZS3 = SymmetricGroupAlgebra(ZZ, 3)
sage: ZS3.antipode(ZS3.zero())
0
sage: ZS3.antipode(-ZS3(Permutation([2, 3, 1])))
-[3, 1, 2]
```

```
>>> from sage.all import *
>>> QS4 = SymmetricGroupAlgebra(QQ, Integer(4))
>>> QS4.antipode(Integer(2) * QS4([Integer(1), Integer(3), Integer(4),␣
↪Integer(2)]) - Integer(1)/Integer(2) * QS4([Integer(1), Integer(4),␣
↪Integer(2), Integer(3)]))
-1/2*[1, 3, 4, 2] + 2*[1, 4, 2, 3]
>>> all( QS4.antipode(QS4(p)) == QS4(p.inverse())
...      for p in Permutations(Integer(4)) )
True

>>> ZS3 = SymmetricGroupAlgebra(ZZ, Integer(3))
>>> ZS3.antipode(ZS3.zero())
0
>>> ZS3.antipode(-ZS3(Permutation([Integer(2), Integer(3), Integer(1)])))
-[3, 1, 2]
```

binary_unshuffle_sum(*k*)

Return the k-th binary unshuffle sum in the group algebra `self`.

The k-th binary unshuffle sum in the symmetric group algebra RS_n over a ring R is defined as the sum of all permutations $\sigma \in S_n$ satisfying $\sigma(1) < \sigma(2) < \cdots < \sigma(k)$ and $\sigma(k+1) < \sigma(k+2) < \cdots < \sigma(n)$.

This element has the property that, if it is denoted by t_k, and if the k-th semi-RSW element (see *semi_rsw_element()*) is denoted by s_k, then $s_k S(t_k)$ and $t_k S(s_k)$ both equal the k-th

Reiner-Saliola-Welker shuffling element of RS_n (see `rsw_shuffling_element()`).

The k-th binary unshuffle sum is the image of the complete non-commutative symmetric function $S^{(k,n-k)}$ in the ring of non-commutative symmetric functions under the canonical projection on the symmetric group algebra (through the descent algebra).

EXAMPLES:

The binary unshuffle sums on $\mathbf{Q}S_3$:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.binary_unshuffle_sum(0)
[1, 2, 3]
sage: QS3.binary_unshuffle_sum(1)
[1, 2, 3] + [2, 1, 3] + [3, 1, 2]
sage: QS3.binary_unshuffle_sum(2)
[1, 2, 3] + [1, 3, 2] + [2, 3, 1]
sage: QS3.binary_unshuffle_sum(3)
[1, 2, 3]
sage: QS3.binary_unshuffle_sum(4)
0
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> QS3.binary_unshuffle_sum(Integer(0))
[1, 2, 3]
>>> QS3.binary_unshuffle_sum(Integer(1))
[1, 2, 3] + [2, 1, 3] + [3, 1, 2]
>>> QS3.binary_unshuffle_sum(Integer(2))
[1, 2, 3] + [1, 3, 2] + [2, 3, 1]
>>> QS3.binary_unshuffle_sum(Integer(3))
[1, 2, 3]
>>> QS3.binary_unshuffle_sum(Integer(4))
0
```

Let us check the relation with the k-th Reiner-Saliola-Welker shuffling element stated in the docstring:

```
sage: def test_rsw(n):
....:     ZSn = SymmetricGroupAlgebra(ZZ, n)
....:     for k in range(1, n):
....:         a = ZSn.semi_rsw_element(k)
....:         b = ZSn.binary_unshuffle_sum(k)
....:         c = ZSn.left_action_product(a, ZSn.antipode(b))
....:         d = ZSn.left_action_product(b, ZSn.antipode(a))
....:         e = ZSn.rsw_shuffling_element(k)
....:         if c != e or d != e:
....:             return False
....:     return True
sage: test_rsw(3)
True
sage: test_rsw(4)  # long time
True
sage: test_rsw(5)  # long time
True
```

```
>>> from sage.all import *
>>> def test_rsw(n):
...     ZSn = SymmetricGroupAlgebra(ZZ, n)
...     for k in range(Integer(1), n):
...         a = ZSn.semi_rsw_element(k)
...         b = ZSn.binary_unshuffle_sum(k)
...         c = ZSn.left_action_product(a, ZSn.antipode(b))
...         d = ZSn.left_action_product(b, ZSn.antipode(a))
...         e = ZSn.rsw_shuffling_element(k)
...         if c != e or d != e:
...             return False
...     return True
>>> test_rsw(Integer(3))
True
>>> test_rsw(Integer(4))  # long time
True
>>> test_rsw(Integer(5))  # long time
True
```

Let us also check the statement about the complete non-commutative symmetric function:

```
sage: def test_rsw_ncsf(n):
....:     ZSn = SymmetricGroupAlgebra(ZZ, n)
....:     NSym = NonCommutativeSymmetricFunctions(ZZ)
....:     S = NSym.S()
....:     for k in range(1, n):
....:         a = S(Composition([k, n-k])).to_symmetric_group_algebra()
....:         if a != ZSn.binary_unshuffle_sum(k):
....:             return False
....:     return True
sage: test_rsw_ncsf(3)
True
sage: test_rsw_ncsf(4)
True
sage: test_rsw_ncsf(5)  # long time
True
```

```
>>> from sage.all import *
>>> def test_rsw_ncsf(n):
...     ZSn = SymmetricGroupAlgebra(ZZ, n)
...     NSym = NonCommutativeSymmetricFunctions(ZZ)
...     S = NSym.S()
...     for k in range(Integer(1), n):
...         a = S(Composition([k, n-k])).to_symmetric_group_algebra()
...         if a != ZSn.binary_unshuffle_sum(k):
...             return False
...     return True
>>> test_rsw_ncsf(Integer(3))
True
>>> test_rsw_ncsf(Integer(4))
True
>>> test_rsw_ncsf(Integer(5))  # long time
True
```

canonical_embedding(*other*)

Return the canonical coercion of `self` into a symmetric group algebra `other`.

INPUT:

- `other` – a symmetric group algebra with order p satisfying $p \geq n$, where n is the order of `self`, over a ground ring into which the ground ring of `self` coerces.

EXAMPLES:

```
sage: QS2 = SymmetricGroupAlgebra(QQ, 2)
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: phi = QS2.canonical_embedding(QS4); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field

sage: x = QS2([2,1]) + 2 * QS2([1,2])
sage: phi(x)
2*[1, 2, 3, 4] + [2, 1, 3, 4]

sage: loads(dumps(phi))
Generic morphism:
  From: Symmetric group algebra of order 2 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field

sage: ZS2 = SymmetricGroupAlgebra(ZZ, 2)
sage: phi = ZS2.canonical_embedding(QS4); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Integer Ring
  To:   Symmetric group algebra of order 4 over Rational Field

sage: phi = ZS2.canonical_embedding(QS2); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Integer Ring
  To:   Symmetric group algebra of order 2 over Rational Field

sage: QS4.canonical_embedding(QS2)
Traceback (most recent call last):
...
ValueError: There is no canonical embedding from Symmetric group
 algebra of order 2 over Rational Field to Symmetric group
 algebra of order 4 over Rational Field

sage: QS4g = SymmetricGroup(4).algebra(QQ)
sage: QS4.canonical_embedding(QS4g)(QS4([1,3,2,4]))
(2,3)
sage: QS4g.canonical_embedding(QS4)(QS4g((2,3)))
[1, 3, 2, 4]
sage: ZS2.canonical_embedding(QS4g)(ZS2([2,1]))
(1,2)
sage: ZS2g = SymmetricGroup(2).algebra(ZZ)
sage: ZS2g.canonical_embedding(QS4)(ZS2g((1,2)))
[2, 1, 3, 4]
```

```
>>> from sage.all import *
>>> QS2 = SymmetricGroupAlgebra(QQ, Integer(2))
>>> QS4 = SymmetricGroupAlgebra(QQ, Integer(4))
>>> phi = QS2.canonical_embedding(QS4); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field

>>> x = QS2([Integer(2),Integer(1)]) + Integer(2) * QS2([Integer(1),
↪Integer(2)])
>>> phi(x)
2*[1, 2, 3, 4] + [2, 1, 3, 4]

>>> loads(dumps(phi))
Generic morphism:
  From: Symmetric group algebra of order 2 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field

>>> ZS2 = SymmetricGroupAlgebra(ZZ, Integer(2))
>>> phi = ZS2.canonical_embedding(QS4); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Integer Ring
  To:   Symmetric group algebra of order 4 over Rational Field

>>> phi = ZS2.canonical_embedding(QS2); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Integer Ring
  To:   Symmetric group algebra of order 2 over Rational Field

>>> QS4.canonical_embedding(QS2)
Traceback (most recent call last):
...
ValueError: There is no canonical embedding from Symmetric group
 algebra of order 2 over Rational Field to Symmetric group
 algebra of order 4 over Rational Field

>>> QS4g = SymmetricGroup(Integer(4)).algebra(QQ)
>>> QS4.canonical_embedding(QS4g)(QS4([Integer(1),Integer(3),Integer(2),
↪Integer(4)]))
(2,3)
>>> QS4g.canonical_embedding(QS4)(QS4g((Integer(2),Integer(3))))
[1, 3, 2, 4]
>>> ZS2.canonical_embedding(QS4g)(ZS2([Integer(2),Integer(1)]))
(1,2)
>>> ZS2g = SymmetricGroup(Integer(2)).algebra(ZZ)
>>> ZS2g.canonical_embedding(QS4)(ZS2g((Integer(1),Integer(2))))
[2, 1, 3, 4]
```

cell_module(*la*, ***kwds*)

Return the cell module indexed by `la`.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: M = S.cell_module(Partition([2,1])); M
Cell module indexed by [2, 1] of Cellular basis of
 Symmetric group algebra of order 3 over Rational Field
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> M = S.cell_module(Partition([Integer(2),Integer(1)])); M
Cell module indexed by [2, 1] of Cellular basis of
 Symmetric group algebra of order 3 over Rational Field
```

We check that the input `la` is standardized:

```
sage: N = S.cell_module([2,1])
sage: M is N
True
```

```
>>> from sage.all import *
>>> N = S.cell_module([Integer(2),Integer(1)])
>>> M is N
True
```

cell_module_indices(*la*)

Return the indices of the cell module of `self` indexed by `la` .

This is the finite set $M(\lambda)$.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 4)
sage: S.cell_module_indices([3,1])
Standard tableaux of shape [3, 1]
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(4))
>>> S.cell_module_indices([Integer(3),Integer(1)])
Standard tableaux of shape [3, 1]
```

cell_poset()

Return the cell poset of `self`.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 4)
sage: S.cell_poset()
Finite poset containing 5 elements
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(4))
>>> S.cell_poset()
Finite poset containing 5 elements
```

central_orthogonal_idempotent(*la*, *block=True*)

Return the central idempotent for the symmetric group of order n corresponding to the indecomposable block to which the partition `la` is associated.

If `self.base_ring()` contains $\mathbf{Q}$, this corresponds to the classical central idempotent corresponding to the irreducible representation indexed by `la`.

Alternatively, if `self.base_ring()` has characteristic $p > 0$, then Theorem 2.8 in [Mur1983] provides that `la` is associated to an idempotent f_μ, where μ is the p-core of `la`. This f_μ is a sum of classical idempotents,

$$f_\mu = \sum_{c(\lambda)=\mu} e_\lambda,$$

where the sum ranges over the partitions λ of n with p-core equal to μ.

INPUT:

- `la` – a partition of `self.n` or a `self.base_ring().characteristic()`-core of such a partition
- `block` – boolean (default: `True`); when `False`, this returns the classical idempotent associated to `la` (defined over $\mathbf{Q}$)

OUTPUT:

If `block=False` and the corresponding coefficients are not defined over `self.base_ring()`, then return `None`. Otherwise return an element of `self`.

EXAMPLES:

Asking for block idempotents in any characteristic, by passing a partition of `self.n`:

```
sage: S0 = SymmetricGroup(4).algebra(QQ)
sage: S2 = SymmetricGroup(4).algebra(GF(2))
sage: S3 = SymmetricGroup(4).algebra(GF(3))
sage: S0.central_orthogonal_idempotent([2,1,1])
3/8*() - 1/8*(3,4) - 1/8*(2,3) - 1/8*(2,4) - 1/8*(1,2)
 - 1/8*(1,2)(3,4) + 1/8*(1,2,3,4) + 1/8*(1,2,4,3)
 + 1/8*(1,3,4,2) - 1/8*(1,3) - 1/8*(1,3)(2,4)
 + 1/8*(1,3,2,4) + 1/8*(1,4,3,2) - 1/8*(1,4)
 + 1/8*(1,4,2,3) - 1/8*(1,4)(2,3)
sage: S2.central_orthogonal_idempotent([2,1,1])
()
sage: idem = S3.central_orthogonal_idempotent([4]); idem
 () + (1,2)(3,4) + (1,3)(2,4) + (1,4)(2,3)
sage: idem == S3.central_orthogonal_idempotent([1,1,1,1])
True
sage: S3.central_orthogonal_idempotent([2,2])
() + (1,2)(3,4) + (1,3)(2,4) + (1,4)(2,3)
```

```
>>> from sage.all import *
>>> S0 = SymmetricGroup(Integer(4)).algebra(QQ)
>>> S2 = SymmetricGroup(Integer(4)).algebra(GF(Integer(2)))
>>> S3 = SymmetricGroup(Integer(4)).algebra(GF(Integer(3)))
>>> S0.central_orthogonal_idempotent([Integer(2),Integer(1),Integer(1)])
3/8*() - 1/8*(3,4) - 1/8*(2,3) - 1/8*(2,4) - 1/8*(1,2)
 - 1/8*(1,2)(3,4) + 1/8*(1,2,3,4) + 1/8*(1,2,4,3)
 + 1/8*(1,3,4,2) - 1/8*(1,3) - 1/8*(1,3)(2,4)
 + 1/8*(1,3,2,4) + 1/8*(1,4,3,2) - 1/8*(1,4)
```

(continues on next page)

(continued from previous page)

```
 + 1/8*(1,4,2,3) - 1/8*(1,4)(2,3)
>>> S2.central_orthogonal_idempotent([Integer(2),Integer(1),Integer(1)])
()
>>> idem = S3.central_orthogonal_idempotent([Integer(4)]); idem
 () + (1,2)(3,4) + (1,3)(2,4) + (1,4)(2,3)
>>> idem == S3.central_orthogonal_idempotent([Integer(1),Integer(1),
↪Integer(1),Integer(1)])
True
>>> S3.central_orthogonal_idempotent([Integer(2),Integer(2)])
() + (1,2)(3,4) + (1,3)(2,4) + (1,4)(2,3)
```

Asking for block idempotents in any characteristic, by passing p-cores:

```
sage: S0.central_orthogonal_idempotent([1,1])
Traceback (most recent call last):
...
ValueError: [1, 1] is not a partition of integer 4
sage: S2.central_orthogonal_idempotent([])
()
sage: S2.central_orthogonal_idempotent([1])
Traceback (most recent call last):
...
ValueError: the 2-core of [1] is not a 2-core of a partition of 4
sage: S3.central_orthogonal_idempotent([1])
() + (1,2)(3,4) + (1,3)(2,4) + (1,4)(2,3)
sage: S3.central_orthogonal_idempotent([7])
() + (1,2)(3,4) + (1,3)(2,4) + (1,4)(2,3)
```

```
>>> from sage.all import *
>>> S0.central_orthogonal_idempotent([Integer(1),Integer(1)])
Traceback (most recent call last):
...
ValueError: [1, 1] is not a partition of integer 4
>>> S2.central_orthogonal_idempotent([])
()
>>> S2.central_orthogonal_idempotent([Integer(1)])
Traceback (most recent call last):
...
ValueError: the 2-core of [1] is not a 2-core of a partition of 4
>>> S3.central_orthogonal_idempotent([Integer(1)])
() + (1,2)(3,4) + (1,3)(2,4) + (1,4)(2,3)
>>> S3.central_orthogonal_idempotent([Integer(7)])
() + (1,2)(3,4) + (1,3)(2,4) + (1,4)(2,3)
```

Asking for classical idempotents:

```
sage: S3.central_orthogonal_idempotent([2,2], block=False) is None
True
sage: S3.central_orthogonal_idempotent([2,1,1], block=False)
(3,4) + (2,3) + (2,4) + (1,2) + (1,2)(3,4) + 2*(1,2,3,4)
 + 2*(1,2,4,3) + 2*(1,3,4,2) + (1,3) + (1,3)(2,4)
 + 2*(1,3,2,4) + 2*(1,4,3,2) + (1,4) + 2*(1,4,2,3)
```

(continues on next page)

(continued from previous page)

```
 + (1,4)(2,3)
```

```
>>> from sage.all import *
>>> S3.central_orthogonal_idempotent([Integer(2),Integer(2)], block=False) is␣
↪None
True
>>> S3.central_orthogonal_idempotent([Integer(2),Integer(1),Integer(1)],␣
↪block=False)
(3,4) + (2,3) + (2,4) + (1,2) + (1,2)(3,4) + 2*(1,2,3,4)
 + 2*(1,2,4,3) + 2*(1,3,4,2) + (1,3) + (1,3)(2,4)
 + 2*(1,3,2,4) + 2*(1,4,3,2) + (1,4) + 2*(1,4,2,3)
 + (1,4)(2,3)
```

See also:

- *sage.combinat.partition.Partition.core()*

central_orthogonal_idempotents()

Return a maximal list of central orthogonal idempotents for `self`.

This method does not require that `self` be semisimple, relying on Nakayama's Conjecture whenever `self.base_ring()` has positive characteristic.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: a = QS3.central_orthogonal_idempotents()
sage: a[0]  # [3]
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1]
 + 1/6*[3, 1, 2] + 1/6*[3, 2, 1]
sage: a[1]  # [2, 1]
2/3*[1, 2, 3] - 1/3*[2, 3, 1] - 1/3*[3, 1, 2]
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ,Integer(3))
>>> a = QS3.central_orthogonal_idempotents()
>>> a[Integer(0)]  # [3]
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1]
 + 1/6*[3, 1, 2] + 1/6*[3, 2, 1]
>>> a[Integer(1)]  # [2, 1]
2/3*[1, 2, 3] - 1/3*[2, 3, 1] - 1/3*[3, 1, 2]
```

See also:

- *central_orthogonal_idempotent()*

dft(*form=None*, *mult='l2r'*)

Return the discrete Fourier transform for `self`.

See [Mur1983] for the construction of central primitive orthogonal idempotents. For each idempotent e_i we have a homomorphic projection $v \mapsto ve_i$. Choose a basis for each submodule spanned by $\{\sigma e_i | \sigma \in S_n\}$. The change-of-basis from the standard basis $\{\sigma\}_\sigma$ is returned.

INPUT:

- `mult` – string (default: $l2r$); if set to $r2l$, this causes the method to use the antipodes (`antipode()`) of the seminormal basis instead of the seminormal basis.
- `form` – string (default: `"modular"` if $p|n!$ else `"seminormal"`); one of the following:
  - `"seminormal"` – use the seminormal basis
  - `"modular"` – use the modular DFT, which uses the Peirce decomposition of the group algebra into blocks with central orthogonal idempotents
  - `"unitary"` – use the unitary DFT, which computes the extended Cholesky decomposition of an S_n-invariant symmetric bilinear form as a change-of-basis matrix (for positive characteristics, it must be a finite field of square order)

EXAMPLES:

The default is the seminormal DFT when the characteristic does not divide the group order:

```
sage: QQ_S3 = SymmetricGroupAlgebra(QQ, 3)
sage: QQ_S3.dft()
[   1    1    1    1    1    1]
[   1  1/2   -1 -1/2 -1/2  1/2]
[   0  3/4    0  3/4 -3/4 -3/4]
[   0    1    0   -1    1   -1]
[   1 -1/2    1 -1/2 -1/2 -1/2]
[   1   -1   -1    1    1   -1]
```

```
>>> from sage.all import *
>>> QQ_S3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> QQ_S3.dft()
[   1    1    1    1    1    1]
[   1  1/2   -1 -1/2 -1/2  1/2]
[   0  3/4    0  3/4 -3/4 -3/4]
[   0    1    0   -1    1   -1]
[   1 -1/2    1 -1/2 -1/2 -1/2]
[   1   -1   -1    1    1   -1]
```

The unitary form works in characteristic zero:

```
sage: U = QQ_S3.dft(form='unitary'); U
[-1/6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2 -1/
↪6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2]
[       1/3*sqrt3        1/6*sqrt3       -1/3*sqrt3       -1/6*sqrt3       -1/
↪6*sqrt3        1/6*sqrt3]
[               0              1/2                0              1/2              ␣
↪    -1/2             -1/2]
[               0              1/2                0             -1/2              ␣
↪     1/2             -1/2]
[       1/3*sqrt3       -1/6*sqrt3        1/3*sqrt3       -1/6*sqrt3       -1/
↪6*sqrt3       -1/6*sqrt3]
[-1/6*sqrt3*sqrt2  1/6*sqrt3*sqrt2  1/6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2 -1/
```

(continues on next page)

(continued from previous page)

```
↪6*sqrt3*sqrt2  1/6*sqrt3*sqrt2]
sage: U*U.H == 1
True
```

```
>>> from sage.all import *
>>> U = QQ_S3.dft(form='unitary'); U
[-1/6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2 -1/
↪6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2]
[       1/3*sqrt3        1/6*sqrt3       -1/3*sqrt3       -1/6*sqrt3       -1/
↪6*sqrt3        1/6*sqrt3]
[               0              1/2                0              1/2          ␣
↪   -1/2             -1/2]
[               0              1/2                0             -1/2          ␣
↪    1/2             -1/2]
[       1/3*sqrt3       -1/6*sqrt3        1/3*sqrt3       -1/6*sqrt3       -1/
↪6*sqrt3       -1/6*sqrt3]
[-1/6*sqrt3*sqrt2  1/6*sqrt3*sqrt2  1/6*sqrt3*sqrt2 -1/6*sqrt3*sqrt2 -1/
↪6*sqrt3*sqrt2  1/6*sqrt3*sqrt2]
>>> U*U.H == Integer(1)
True
```

Over finite fields of square order with characteristic $p > n$, we can perform the unitary DFT:

```
sage: GF25_S3 = SymmetricGroupAlgebra(GF(5**2), 3)
sage: U = GF25_S3.dft(form='unitary'); U
[       1        1        1        1        1        1]
[2*z2 + 4   z2 + 2 3*z2 + 1 4*z2 + 3 4*z2 + 3   z2 + 2]
[       0        2        0        2        3        3]
[       0   z2 + 1        0 4*z2 + 4   z2 + 1 4*z2 + 4]
[2*z2 + 4 4*z2 + 3 2*z2 + 4 4*z2 + 3 4*z2 + 3 4*z2 + 3]
[       1        4        4        1        1        4]
sage: U*U.H == 1
True
```

```
>>> from sage.all import *
>>> GF25_S3 = SymmetricGroupAlgebra(GF(Integer(5)**Integer(2)), Integer(3))
>>> U = GF25_S3.dft(form='unitary'); U
[       1        1        1        1        1        1]
[2*z2 + 4   z2 + 2 3*z2 + 1 4*z2 + 3 4*z2 + 3   z2 + 2]
[       0        2        0        2        3        3]
[       0   z2 + 1        0 4*z2 + 4   z2 + 1 4*z2 + 4]
[2*z2 + 4 4*z2 + 3 2*z2 + 4 4*z2 + 3 4*z2 + 3 4*z2 + 3]
[       1        4        4        1        1        4]
>>> U*U.H == Integer(1)
True
```

Over fields of characteristic $p > 0$ such that $p|n!$, we use the modular Fourier transform (Issue #37751):

```
sage: GF2_S3 = SymmetricGroupAlgebra(GF(2), 3)
sage: GF2_S3.dft()
[1 0 0 0 1 0]
[0 1 0 0 0 1]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0 0 1]
[0 0 0 1 1 0]
[1 0 0 1 1 0]
[0 1 1 0 0 1]
```

```
>>> from sage.all import *
>>> GF2_S3 = SymmetricGroupAlgebra(GF(Integer(2)), Integer(3))
>>> GF2_S3.dft()
[1 0 0 0 1 0]
[0 1 0 0 0 1]
[0 0 1 0 0 1]
[0 0 0 1 1 0]
[1 0 0 1 1 0]
[0 1 1 0 0 1]
```

epsilon_ik(*itab*, *ktab*, *star=0*, *mult='l2r'*)

Return the seminormal basis element of `self` corresponding to the pair of tableaux `itab` and `ktab` (or restrictions of these tableaux, if the optional variable `star` is set).

INPUT:

- `itab`, `ktab` – two standard tableaux of size n
- `star` – integer (default: 0)
- `mult` – string (default: $l2r$); if set to $r2l$, this causes the method to return the antipode (`antipode()`) of $\epsilon(I, K)$ instead of $\epsilon(I, K)$ itself.

OUTPUT:

The element $\epsilon(I, K)$, where I and K are the tableaux obtained by removing all entries higher than $n - \mathrm{star}$ from `itab` and `ktab`, respectively. Here, we are using the notations from `seminormal_basis()`.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = QS3.epsilon_ik([[1,2,3]], [[1,2,3]]); a
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2]␣
↪+ 1/6*[3, 2, 1]
sage: QS3.dft()*vector(a)
(1, 0, 0, 0, 0, 0)
sage: a = QS3.epsilon_ik([[1,2],[3]], [[1,2],[3]]); a
1/3*[1, 2, 3] - 1/6*[1, 3, 2] + 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2]␣
↪- 1/6*[3, 2, 1]
sage: QS3.dft()*vector(a)
(0, 0, 0, 0, 1, 0)
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> a = QS3.epsilon_ik([[Integer(1),Integer(2),Integer(3)]], [[Integer(1),
↪Integer(2),Integer(3)]]); a
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2]␣
↪+ 1/6*[3, 2, 1]
>>> QS3.dft()*vector(a)
(1, 0, 0, 0, 0, 0)
```

(continues on next page)

(continued from previous page)

```
>>> a = QS3.epsilon_ik([[Integer(1),Integer(2)],[Integer(3)]], [[Integer(1),
↪Integer(2)],[Integer(3)]]); a
1/3*[1, 2, 3] - 1/6*[1, 3, 2] + 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2]␣
↪- 1/6*[3, 2, 1]
>>> QS3.dft()*vector(a)
(0, 0, 0, 0, 1, 0)
```

Let us take some properties of the seminormal basis listed in the docstring of `seminormal_basis()`, and verify them on the situation of S_3.

First, check the formula

$$\epsilon(T) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{T}) e(T) \epsilon(\overline{T}).$$

In fact:

```
sage: from sage.combinat.symmetric_group_algebra import e
sage: def test_sn1(n):
....:     QSn = SymmetricGroupAlgebra(QQ, n)
....:     QSn1 = SymmetricGroupAlgebra(QQ, n - 1)
....:     for T in StandardTableaux(n):
....:         TT = T.restrict(n-1)
....:         eTT = QSn1.epsilon_ik(TT, TT)
....:         eT = QSn.epsilon_ik(T, T)
....:         kT = prod(T.shape().hooks())
....:         if kT * eT != eTT * e(T) * eTT:
....:             return False
....:     return True
sage: test_sn1(3)
True
sage: test_sn1(4)    # long time
True
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import e
>>> def test_sn1(n):
...     QSn = SymmetricGroupAlgebra(QQ, n)
...     QSn1 = SymmetricGroupAlgebra(QQ, n - Integer(1))
...     for T in StandardTableaux(n):
...         TT = T.restrict(n-Integer(1))
...         eTT = QSn1.epsilon_ik(TT, TT)
...         eT = QSn.epsilon_ik(T, T)
...         kT = prod(T.shape().hooks())
...         if kT * eT != eTT * e(T) * eTT:
...             return False
...     return True
>>> test_sn1(Integer(3))
True
>>> test_sn1(Integer(4))    # long time
True
```

Next, we check the identity

$$\epsilon(T,S) = \frac{1}{\kappa_{\mathrm{sh}(T)}}\epsilon(\overline{S})\pi_{T,S}e(T)\epsilon(\overline{T})$$

which we used to define $\epsilon(T,S)$. In fact:

```
sage: from sage.combinat.symmetric_group_algebra import e
sage: def test_sn2(n):
....:     QSn = SymmetricGroupAlgebra(QQ, n)
....:     mul = QSn.left_action_product
....:     QSn1 = SymmetricGroupAlgebra(QQ, n - 1)
....:     for lam in Partitions(n):
....:         k = prod(lam.hooks())
....:         for T in StandardTableaux(lam):
....:             for S in StandardTableaux(lam):
....:                 TT = T.restrict(n-1)
....:                 SS = S.restrict(n-1)
....:                 eTT = QSn1.epsilon_ik(TT, TT)
....:                 eSS = QSn1.epsilon_ik(SS, SS)
....:                 eTS = QSn.epsilon_ik(T, S)
....:                 piTS = [0] * n
....:                 for (i, j) in T.cells():
....:                     piTS[T[i][j] - 1] = S[i][j]
....:                 piTS = QSn(Permutation(piTS))
....:                 if k * eTS != mul(mul(eSS, piTS), mul(e(T), eTT)):
....:                     return False
....:     return True
sage: test_sn2(3)
True
sage: test_sn2(4)    # long time
True
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import e
>>> def test_sn2(n):
...     QSn = SymmetricGroupAlgebra(QQ, n)
...     mul = QSn.left_action_product
...     QSn1 = SymmetricGroupAlgebra(QQ, n - Integer(1))
...     for lam in Partitions(n):
...         k = prod(lam.hooks())
...         for T in StandardTableaux(lam):
...             for S in StandardTableaux(lam):
...                 TT = T.restrict(n-Integer(1))
...                 SS = S.restrict(n-Integer(1))
...                 eTT = QSn1.epsilon_ik(TT, TT)
...                 eSS = QSn1.epsilon_ik(SS, SS)
...                 eTS = QSn.epsilon_ik(T, S)
...                 piTS = [Integer(0)] * n
...                 for (i, j) in T.cells():
...                     piTS[T[i][j] - Integer(1)] = S[i][j]
...                 piTS = QSn(Permutation(piTS))
...                 if k * eTS != mul(mul(eSS, piTS), mul(e(T), eTT)):
...                     return False
```

(continues on next page)

(continued from previous page)

```
...     return True
>>> test_sn2(Integer(3))
True
>>> test_sn2(Integer(4))    # long time
True
```

Let us finally check the identity

$$\epsilon(T,S)\epsilon(U,V) = \delta_{T,V}\epsilon(U,S)$$

In fact:

```
sage: def test_sn3(lam):
....:     n = lam.size()
....:     QSn = SymmetricGroupAlgebra(QQ, n)
....:     mul = QSn.left_action_product
....:     for T in StandardTableaux(lam):
....:         for S in StandardTableaux(lam):
....:             for U in StandardTableaux(lam):
....:                 for V in StandardTableaux(lam):
....:                     lhs = mul(QSn.epsilon_ik(T, S), QSn.epsilon_ik(U,␣
↪V))
....:                     if T == V:
....:                         rhs = QSn.epsilon_ik(U, S)
....:                     else:
....:                         rhs = QSn.zero()
....:                     if rhs != lhs:
....:                         return False
....:     return True
sage: all( test_sn3(lam) for lam in Partitions(3) )
True
sage: all( test_sn3(lam) for lam in Partitions(4) )    # long time
True
```

```
>>> from sage.all import *
>>> def test_sn3(lam):
...     n = lam.size()
...     QSn = SymmetricGroupAlgebra(QQ, n)
...     mul = QSn.left_action_product
...     for T in StandardTableaux(lam):
...         for S in StandardTableaux(lam):
...             for U in StandardTableaux(lam):
...                 for V in StandardTableaux(lam):
...                     lhs = mul(QSn.epsilon_ik(T, S), QSn.epsilon_ik(U, V))
...                     if T == V:
...                         rhs = QSn.epsilon_ik(U, S)
...                     else:
...                         rhs = QSn.zero()
...                     if rhs != lhs:
...                         return False
...     return True
>>> all( test_sn3(lam) for lam in Partitions(Integer(3)) )
```

(continues on next page)

(continued from previous page)

```
True
>>> all( test_sn3(lam) for lam in Partitions(Integer(4)) )    # long time
True
```

garsia_procesi_module(*la*)

Return the *Garsia-Procesi module* of self indexed by la.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 6)
sage: SGA.garsia_procesi_module(Partition([2,2,1,1]))
Garsia-Procesi module of shape [2, 2, 1, 1] over Finite Field of size 2
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(6))
>>> SGA.garsia_procesi_module(Partition([Integer(2),Integer(2),Integer(1),
↪Integer(1)]))
Garsia-Procesi module of shape [2, 2, 1, 1] over Finite Field of size 2
```

jucys_murphy(*k*)

Return the Jucys-Murphy element J_k (also known as a Young-Jucys-Murphy element) for the symmetric group algebra self.

The Jucys-Murphy element J_k in the symmetric group algebra RS_n is defined for every $k \in \{1, 2, \ldots, n\}$ by

$$J_k = (1, k) + (2, k) + \cdots + (k - 1, k) \in RS_n,$$

where the addends are transpositions in S_n (regarded as elements of RS_n). We note that there is not a dependence on n, so it is often suppressed in the notation.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.jucys_murphy(1)
0
sage: QS3.jucys_murphy(2)
[2, 1, 3]
sage: QS3.jucys_murphy(3)
[1, 3, 2] + [3, 2, 1]

sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: j3 = QS4.jucys_murphy(3); j3
[1, 3, 2, 4] + [3, 2, 1, 4]
sage: j4 = QS4.jucys_murphy(4); j4
[1, 2, 4, 3] + [1, 4, 3, 2] + [4, 2, 3, 1]
sage: j3*j4 == j4*j3
True

sage: QS5 = SymmetricGroupAlgebra(QQ, 5)
sage: QS5.jucys_murphy(4)
[1, 2, 4, 3, 5] + [1, 4, 3, 2, 5] + [4, 2, 3, 1, 5]
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> QS3.jucys_murphy(Integer(1))
0
>>> QS3.jucys_murphy(Integer(2))
[2, 1, 3]
>>> QS3.jucys_murphy(Integer(3))
[1, 3, 2] + [3, 2, 1]

>>> QS4 = SymmetricGroupAlgebra(QQ, Integer(4))
>>> j3 = QS4.jucys_murphy(Integer(3)); j3
[1, 3, 2, 4] + [3, 2, 1, 4]
>>> j4 = QS4.jucys_murphy(Integer(4)); j4
[1, 2, 4, 3] + [1, 4, 3, 2] + [4, 2, 3, 1]
>>> j3*j4 == j4*j3
True

>>> QS5 = SymmetricGroupAlgebra(QQ, Integer(5))
>>> QS5.jucys_murphy(Integer(4))
[1, 2, 4, 3, 5] + [1, 4, 3, 2, 5] + [4, 2, 3, 1, 5]
```

kazhdan_lusztig_basis_element(*w*)

Return the Kazhdan-Lusztig C'_w basis element at $q = 1$.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: for w in SGA.group():
....:     print(w, SGA.kazhdan_lusztig_basis_element(w))
[1, 2, 3] [1, 2, 3]
[1, 3, 2] [1, 2, 3] + [1, 3, 2]
[2, 1, 3] [1, 2, 3] + [2, 1, 3]
[2, 3, 1] [1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1]
[3, 1, 2] [1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [3, 1, 2]
[3, 2, 1] [1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 
→1]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(3))
>>> for w in SGA.group():
...     print(w, SGA.kazhdan_lusztig_basis_element(w))
[1, 2, 3] [1, 2, 3]
[1, 3, 2] [1, 2, 3] + [1, 3, 2]
[2, 1, 3] [1, 2, 3] + [2, 1, 3]
[2, 3, 1] [1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1]
[3, 1, 2] [1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [3, 1, 2]
[3, 2, 1] [1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 
→1]
```

kazhdan_lusztig_cellular_basis()

Return the Kazhdan-Lusztig basis (at $q = 1$) of `self` as a cellular basis.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: KL = SGA.kazhdan_lusztig_cellular_basis()
sage: KL(SGA.an_element())
C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
 + C([2, 1], [[1, 3], [2]], [[1, 2], [3]])
 + 2*C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
 - 3*C([3], [[1, 2, 3]], [[1, 2, 3]])
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(3))
>>> KL = SGA.kazhdan_lusztig_cellular_basis()
>>> KL(SGA.an_element())
C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
 + C([2, 1], [[1, 3], [2]], [[1, 2], [3]])
 + 2*C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
 - 3*C([3], [[1, 2, 3]], [[1, 2, 3]])
```

ladder_idempotent(*la*)

Return the ladder idempotent of `self`.

Let F be a field of characteristic p. The *ladder idempotent* of shape λ is the idempotent of $F[S_n]$ defined as follows. Let T be the *ladder tableau* of shape λ. Let $[T]$ be the set of standard tableaux whose residue sequence is the same as for T. Let α be the sizes of the ladders of λ. Then the ladder idempotent is constructed as

$$\widetilde{e}_\lambda := \frac{1}{\alpha!}\left(\sum_{\sigma\in S_\alpha}\sigma\right)\left(\overline{\sum_{U\in[T]} E_U}\right),$$

where E_{UU} is the `seminormal_basis()` element over $\mathbf{Q}$ and we project the sum to F, S_α is the Young subgroup corresponding to α, and $\alpha! = \alpha_1!\cdots\alpha_k!$.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 4)
sage: for la in Partitions(SGA.n):
....:     idem = SGA.ladder_idempotent(la)
....:     print(la)
....:     print(idem)
....:     assert idem^2 == idem
[4]
2*[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 2*[2, 1, 3, 4] + 2*[2, 1, 4, 3]
 + 2*[3, 4, 1, 2] + 2*[3, 4, 2, 1] + 2*[4, 3, 1, 2] + 2*[4, 3, 2, 1]
[3, 1]
2*[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 2*[2, 1, 3, 4] + 2*[2, 1, 4, 3]
 + [3, 4, 1, 2] + [3, 4, 2, 1] + [4, 3, 1, 2] + [4, 3, 2, 1]
[2, 2]
2*[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 2*[2, 1, 3, 4] + 2*[2, 1, 4, 3]
 + 2*[3, 4, 1, 2] + 2*[3, 4, 2, 1] + 2*[4, 3, 1, 2] + 2*[4, 3, 2, 1]
[2, 1, 1]
2*[1, 2, 3, 4] + [1, 2, 4, 3] + 2*[1, 3, 2, 4] + [1, 3, 4, 2]
 + [1, 4, 2, 3] + 2*[1, 4, 3, 2] + 2*[2, 1, 3, 4] + [2, 1, 4, 3]
 + 2*[2, 3, 1, 4] + [2, 3, 4, 1] + [2, 4, 1, 3] + 2*[2, 4, 3, 1]
 + 2*[3, 1, 2, 4] + [3, 1, 4, 2] + 2*[3, 2, 1, 4] + [3, 2, 4, 1]
```

(continues on next page)

(continued from previous page)

```
 + [4, 1, 2, 3] + 2*[4, 1, 3, 2] + [4, 2, 1, 3] + 2*[4, 2, 3, 1]
[1, 1, 1, 1]
2*[1, 2, 3, 4] + [1, 2, 4, 3] + [2, 1, 3, 4] + 2*[2, 1, 4, 3]
 + 2*[3, 4, 1, 2] + [3, 4, 2, 1] + [4, 3, 1, 2] + 2*[4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(4))
>>> for la in Partitions(SGA.n):
...     idem = SGA.ladder_idempotent(la)
...     print(la)
...     print(idem)
...     assert idem**Integer(2) == idem
[4]
2*[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 2*[2, 1, 3, 4] + 2*[2, 1, 4, 3]
 + 2*[3, 4, 1, 2] + 2*[3, 4, 2, 1] + 2*[4, 3, 1, 2] + 2*[4, 3, 2, 1]
[3, 1]
2*[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 2*[2, 1, 3, 4] + 2*[2, 1, 4, 3]
 + [3, 4, 1, 2] + [3, 4, 2, 1] + [4, 3, 1, 2] + [4, 3, 2, 1]
[2, 2]
2*[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 2*[2, 1, 3, 4] + 2*[2, 1, 4, 3]
 + 2*[3, 4, 1, 2] + 2*[3, 4, 2, 1] + 2*[4, 3, 1, 2] + 2*[4, 3, 2, 1]
[2, 1, 1]
2*[1, 2, 3, 4] + [1, 2, 4, 3] + 2*[1, 3, 2, 4] + [1, 3, 4, 2]
 + [1, 4, 2, 3] + 2*[1, 4, 3, 2] + 2*[2, 1, 3, 4] + [2, 1, 4, 3]
 + 2*[2, 3, 1, 4] + [2, 3, 4, 1] + [2, 4, 1, 3] + 2*[2, 4, 3, 1]
 + 2*[3, 1, 2, 4] + [3, 1, 4, 2] + 2*[3, 2, 1, 4] + [3, 2, 4, 1]
 + [4, 1, 2, 3] + 2*[4, 1, 3, 2] + [4, 2, 1, 3] + 2*[4, 2, 3, 1]
[1, 1, 1, 1]
2*[1, 2, 3, 4] + [1, 2, 4, 3] + [2, 1, 3, 4] + 2*[2, 1, 4, 3]
 + 2*[3, 4, 1, 2] + [3, 4, 2, 1] + [4, 3, 1, 2] + 2*[4, 3, 2, 1]
```

When $p = 0$, these idempotents will generate all of the simple modules (which are the *Specht modules* and also projective modules):

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: for la in Partitions(SGA.n):
....:     idem = SGA.ladder_idempotent(la)
....:     assert idem^2 == idem
....:     print(la, SGA.principal_ideal(idem).dimension())
[5] 1
[4, 1] 4
[3, 2] 5
[3, 1, 1] 6
[2, 2, 1] 5
[2, 1, 1, 1] 4
[1, 1, 1, 1, 1] 1
sage: [StandardTableaux(la).cardinality() for la in Partitions(SGA.n)]
[1, 4, 5, 6, 5, 4, 1]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> for la in Partitions(SGA.n):
```

(continues on next page)

(continued from previous page)

```
...     idem = SGA.ladder_idempotent(la)
...     assert idem**Integer(2) == idem
...     print(la, SGA.principal_ideal(idem).dimension())
[5] 1
[4, 1] 4
[3, 2] 5
[3, 1, 1] 6
[2, 2, 1] 5
[2, 1, 1, 1] 4
[1, 1, 1, 1, 1] 1
>>> [StandardTableaux(la).cardinality() for la in Partitions(SGA.n)]
[1, 4, 5, 6, 5, 4, 1]
```

REFERENCES:

- [Ryom2015]

left_action_product (*left*, *right*)

Return the product of two elements `left` and `right` of `self`, where multiplication is defined in such a way that for two permutations p and q, the product pq is the permutation obtained by first applying q and then applying p. This definition of multiplication is tailored to make multiplication of permutations associative with their action on numbers if permutations are to act on numbers from the left.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: p1 = Permutation([2, 1, 3])
sage: p2 = Permutation([3, 1, 2])
sage: QS3.left_action_product(QS3(p1), QS3(p2))
[3, 2, 1]
sage: x = QS3([1, 2, 3]) - 2*QS3([1, 3, 2])
sage: y = 1/2 * QS3([3, 1, 2]) + 3*QS3([1, 2, 3])
sage: QS3.left_action_product(x, y)
3*[1, 2, 3] - 6*[1, 3, 2] - [2, 1, 3] + 1/2*[3, 1, 2]
sage: QS3.left_action_product(0, x)
0
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> p1 = Permutation([Integer(2), Integer(1), Integer(3)])
>>> p2 = Permutation([Integer(3), Integer(1), Integer(2)])
>>> QS3.left_action_product(QS3(p1), QS3(p2))
[3, 2, 1]
>>> x = QS3([Integer(1), Integer(2), Integer(3)]) -␣
↪Integer(2)*QS3([Integer(1), Integer(3), Integer(2)])
>>> y = Integer(1)/Integer(2) * QS3([Integer(3), Integer(1), Integer(2)]) +␣
↪Integer(3)*QS3([Integer(1), Integer(2), Integer(3)])
>>> QS3.left_action_product(x, y)
3*[1, 2, 3] - 6*[1, 3, 2] - [2, 1, 3] + 1/2*[3, 1, 2]
>>> QS3.left_action_product(Integer(0), x)
0
```

The method coerces its input into the algebra `self`:

```
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: QS4.left_action_product(QS3([1, 2, 3]), QS3([2, 1, 3]))
[2, 1, 3, 4]
sage: QS4.left_action_product(1, Permutation([4, 1, 2, 3]))
[4, 1, 2, 3]
```

```
>>> from sage.all import *
>>> QS4 = SymmetricGroupAlgebra(QQ, Integer(4))
>>> QS4.left_action_product(QS3([Integer(1), Integer(2), Integer(3)]),␣
→QS3([Integer(2), Integer(1), Integer(3)]))
[2, 1, 3, 4]
>>> QS4.left_action_product(Integer(1), Permutation([Integer(4), Integer(1),␣
→Integer(2), Integer(3)]))
[4, 1, 2, 3]
```

Warning

Note that coercion presently works from permutations of `n` into the `n`-th symmetric group algebra, and also from all smaller symmetric group algebras into the `n`-th symmetric group algebra, but not from permutations of integers smaller than `n` into the `n`-th symmetric group algebra.

monomial_from_smaller_permutation(*permutation*)

Convert `permutation` into a permutation, possibly extending it to the appropriate size, and return the corresponding basis element of `self`.

EXAMPLES:

```
sage: QS5 = SymmetricGroupAlgebra(QQ, 5)
sage: QS5.monomial_from_smaller_permutation([])
[1, 2, 3, 4, 5]
sage: QS5.monomial_from_smaller_permutation(Permutation([3,1,2]))
[3, 1, 2, 4, 5]
sage: QS5.monomial_from_smaller_permutation([5,3,4,1,2])
[5, 3, 4, 1, 2]
sage: QS5.monomial_from_smaller_permutation(SymmetricGroup(2)((1,2)))
[2, 1, 3, 4, 5]

sage: QS5g = SymmetricGroup(5).algebra(QQ)
sage: QS5g.monomial_from_smaller_permutation([2,1])
(1,2)
```

```
>>> from sage.all import *
>>> QS5 = SymmetricGroupAlgebra(QQ, Integer(5))
>>> QS5.monomial_from_smaller_permutation([])
[1, 2, 3, 4, 5]
>>> QS5.monomial_from_smaller_permutation(Permutation([Integer(3),Integer(1),
→Integer(2)]))
[3, 1, 2, 4, 5]
>>> QS5.monomial_from_smaller_permutation([Integer(5),Integer(3),Integer(4),
→Integer(1),Integer(2)])
[5, 3, 4, 1, 2]
```

(continues on next page)

(continued from previous page)

```
>>> QS5.monomial_from_smaller_
↪permutation(SymmetricGroup(Integer(2))((Integer(1),Integer(2))))
[2, 1, 3, 4, 5]

>>> QS5g = SymmetricGroup(Integer(5)).algebra(QQ)
>>> QS5g.monomial_from_smaller_permutation([Integer(2),Integer(1)])
(1,2)
```

murphy_basis()

Return the *Murphy basis* of self.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: M = SGA.murphy_basis()
sage: M(SGA.an_element())
-C([1, 1, 1], [[1], [2], [3]], [[1], [2], [3]])
 + C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
 + C([2, 1], [[1, 2], [3]], [[1, 3], [2]])
 + 2*C([2, 1], [[1, 3], [2]], [[1, 2], [3]])
 + 4*C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
 - 3*C([3], [[1, 2, 3]], [[1, 2, 3]])
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(3))
>>> M = SGA.murphy_basis()
>>> M(SGA.an_element())
-C([1, 1, 1], [[1], [2], [3]], [[1], [2], [3]])
 + C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
 + C([2, 1], [[1, 2], [3]], [[1, 3], [2]])
 + 2*C([2, 1], [[1, 3], [2]], [[1, 2], [3]])
 + 4*C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
 - 3*C([3], [[1, 2, 3]], [[1, 2, 3]])
```

murphy_basis_element(S, T)

Return the Murphy basis element indexed by S and T.

See also

MurphyBasis

EXAMPLES:

```
sage: import itertools
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: for S, T in itertools.product(StandardTableaux([2,1]), repeat=2):
....:     print(S, T, SGA.murphy_basis_element(S, T))
[[1, 3], [2]] [[1, 3], [2]] [1, 2, 3] + [2, 1, 3]
[[1, 3], [2]] [[1, 2], [3]] [1, 3, 2] + [3, 1, 2]
[[1, 2], [3]] [[1, 3], [2]] [1, 3, 2] + [2, 3, 1]
[[1, 2], [3]] [[1, 2], [3]] [1, 2, 3] + [3, 2, 1]
```

```
>>> from sage.all import *
>>> import itertools
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(3))
>>> for S, T in itertools.product(StandardTableaux([Integer(2),Integer(1)]),␣
↪repeat=Integer(2)):
...     print(S, T, SGA.murphy_basis_element(S, T))
[[1, 3], [2]] [[1, 3], [2]] [1, 2, 3] + [2, 1, 3]
[[1, 3], [2]] [[1, 2], [3]] [1, 3, 2] + [3, 1, 2]
[[1, 2], [3]] [[1, 3], [2]] [1, 3, 2] + [2, 3, 1]
[[1, 2], [3]] [[1, 2], [3]] [1, 2, 3] + [3, 2, 1]
```

retract_direct_product (f, m)

Return the direct-product retract of the element $f \in RS_n$ to RS_m, where $m \leq n$ (and where RS_n is `self`).

If m is a nonnegative integer less or equal to n, then the direct-product retract from S_n to S_m is defined as an R-linear map $S_n \to S_m$ which sends every permutation $p \in S_n$ to

$$\begin{cases} \mathrm{dret}(p) & \text{if } \mathrm{dret}(p) \text{ is defined;} \\ 0 & \text{otherwise} \end{cases}.$$

Here $\mathrm{dret}(p)$ denotes the direct-product retract of the permutation p to S_m, which is defined in *retract_direct_product()*.

EXAMPLES:

```
sage: SGA3 = SymmetricGroupAlgebra(QQ, 3)
sage: SGA3.retract_direct_product(2*SGA3([1,2,3]) - 4*SGA3([2,1,3]) +␣
↪7*SGA3([1,3,2]), 2)
2*[1, 2] - 4*[2, 1]
sage: SGA3.retract_direct_product(2*SGA3([1,3,2]) - 5*SGA3([2,3,1]), 2)
0

sage: SGA5 = SymmetricGroupAlgebra(QQ, 5)
sage: SGA5.retract_direct_product(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) +␣
↪11*SGA5([3,2,1,4,5]), 4)
11*[3, 2, 1, 4]
sage: SGA5.retract_direct_product(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) +␣
↪11*SGA5([3,2,1,4,5]), 3)
-6*[1, 3, 2] + 11*[3, 2, 1]
sage: SGA5.retract_direct_product(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) +␣
↪11*SGA5([3,2,1,4,5]), 2)
0
sage: SGA5.retract_direct_product(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) +␣
↪11*SGA5([3,2,1,4,5]), 1)
2*[1]

sage: SGA5.retract_direct_product(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]),␣
↪3)
8*[1, 2, 3] - 6*[1, 3, 2]
sage: SGA5.retract_direct_product(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]),␣
↪1)
2*[1]
sage: SGA5.retract_direct_product(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]),␣
```

(continues on next page)

(continued from previous page)

```
↪0)
2*[]
```

```
>>> from sage.all import *
>>> SGA3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> SGA3.retract_direct_product(Integer(2)*SGA3([Integer(1),Integer(2),
↪Integer(3)]) - Integer(4)*SGA3([Integer(2),Integer(1),Integer(3)]) +␣
↪Integer(7)*SGA3([Integer(1),Integer(3),Integer(2)]), Integer(2))
2*[1, 2] - 4*[2, 1]
>>> SGA3.retract_direct_product(Integer(2)*SGA3([Integer(1),Integer(3),
↪Integer(2)]) - Integer(5)*SGA3([Integer(2),Integer(3),Integer(1)]),␣
↪Integer(2))
0

>>> SGA5 = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SGA5.retract_direct_product(Integer(8)*SGA5([Integer(1),Integer(4),
↪Integer(2),Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),
↪Integer(2),Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),
↪Integer(2),Integer(1),Integer(4),Integer(5)]), Integer(4))
11*[3, 2, 1, 4]
>>> SGA5.retract_direct_product(Integer(8)*SGA5([Integer(1),Integer(4),
↪Integer(2),Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),
↪Integer(2),Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),
↪Integer(2),Integer(1),Integer(4),Integer(5)]), Integer(3))
-6*[1, 3, 2] + 11*[3, 2, 1]
>>> SGA5.retract_direct_product(Integer(8)*SGA5([Integer(1),Integer(4),
↪Integer(2),Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),
↪Integer(2),Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),
↪Integer(2),Integer(1),Integer(4),Integer(5)]), Integer(2))
0
>>> SGA5.retract_direct_product(Integer(8)*SGA5([Integer(1),Integer(4),
↪Integer(2),Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),
↪Integer(2),Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),
↪Integer(2),Integer(1),Integer(4),Integer(5)]), Integer(1))
2*[1]

>>> SGA5.retract_direct_product(Integer(8)*SGA5([Integer(1),Integer(2),
↪Integer(3),Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),
↪Integer(2),Integer(4),Integer(5)]), Integer(3))
8*[1, 2, 3] - 6*[1, 3, 2]
>>> SGA5.retract_direct_product(Integer(8)*SGA5([Integer(1),Integer(2),
↪Integer(3),Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),
↪Integer(2),Integer(4),Integer(5)]), Integer(1))
2*[1]
>>> SGA5.retract_direct_product(Integer(8)*SGA5([Integer(1),Integer(2),
↪Integer(3),Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),
↪Integer(2),Integer(4),Integer(5)]), Integer(0))
2*[]
```

See also

retract_plain(), *retract_okounkov_vershik()*

retract_okounkov_vershik(*f*, *m*)

Return the Okounkov-Vershik retract of the element $f \in RS_n$ to RS_m, where $m \leq n$ (and where RS_n is `self`).

If m is a nonnegative integer less or equal to n, then the Okounkov-Vershik retract from S_n to S_m is defined as an R-linear map $S_n \to S_m$ which sends every permutation $p \in S_n$ to the Okounkov-Vershik retract of the permutation p to S_m, which is defined in *retract_okounkov_vershik()*.

EXAMPLES:

```
sage: SGA3 = SymmetricGroupAlgebra(QQ, 3)
sage: SGA3.retract_okounkov_vershik(2*SGA3([1,2,3]) - 4*SGA3([2,1,3]) +
↪7*SGA3([1,3,2]), 2)
9*[1, 2] - 4*[2, 1]
sage: SGA3.retract_okounkov_vershik(2*SGA3([1,3,2]) - 5*SGA3([2,3,1]), 2)
2*[1, 2] - 5*[2, 1]

sage: SGA5 = SymmetricGroupAlgebra(QQ, 5)
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4])
↪+ 11*SGA5([3,2,1,4,5]), 4)
-6*[1, 3, 2, 4] + 8*[1, 4, 2, 3] + 11*[3, 2, 1, 4]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4])
↪+ 11*SGA5([3,2,1,4,5]), 3)
2*[1, 3, 2] + 11*[3, 2, 1]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4])
↪+ 11*SGA5([3,2,1,4,5]), 2)
13*[1, 2]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4])
↪+ 11*SGA5([3,2,1,4,5]), 1)
13*[1]

sage: SGA5.retract_okounkov_vershik(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]),
↪ 3)
8*[1, 2, 3] - 6*[1, 3, 2]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]),
↪ 1)
2*[1]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]),
↪ 0)
2*[]
```

```
>>> from sage.all import *
>>> SGA3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> SGA3.retract_okounkov_vershik(Integer(2)*SGA3([Integer(1),Integer(2),
↪Integer(3)]) - Integer(4)*SGA3([Integer(2),Integer(1),Integer(3)]) +
↪Integer(7)*SGA3([Integer(1),Integer(3),Integer(2)]), Integer(2))
9*[1, 2] - 4*[2, 1]
>>> SGA3.retract_okounkov_vershik(Integer(2)*SGA3([Integer(1),Integer(3),
↪Integer(2)]) - Integer(5)*SGA3([Integer(2),Integer(3),Integer(1)]),
↪Integer(2))
2*[1, 2] - 5*[2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> SGA5 = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SGA5.retract_okounkov_vershik(Integer(8)*SGA5([Integer(1),Integer(4),
→Integer(2),Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),
→Integer(2),Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),
→Integer(2),Integer(1),Integer(4),Integer(5)]), Integer(4))
-6*[1, 3, 2, 4] + 8*[1, 4, 2, 3] + 11*[3, 2, 1, 4]
>>> SGA5.retract_okounkov_vershik(Integer(8)*SGA5([Integer(1),Integer(4),
→Integer(2),Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),
→Integer(2),Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),
→Integer(2),Integer(1),Integer(4),Integer(5)]), Integer(3))
2*[1, 3, 2] + 11*[3, 2, 1]
>>> SGA5.retract_okounkov_vershik(Integer(8)*SGA5([Integer(1),Integer(4),
→Integer(2),Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),
→Integer(2),Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),
→Integer(2),Integer(1),Integer(4),Integer(5)]), Integer(2))
13*[1, 2]
>>> SGA5.retract_okounkov_vershik(Integer(8)*SGA5([Integer(1),Integer(4),
→Integer(2),Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),
→Integer(2),Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),
→Integer(2),Integer(1),Integer(4),Integer(5)]), Integer(1))
13*[1]

>>> SGA5.retract_okounkov_vershik(Integer(8)*SGA5([Integer(1),Integer(2),
→Integer(3),Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),
→Integer(2),Integer(4),Integer(5)]), Integer(3))
8*[1, 2, 3] - 6*[1, 3, 2]
>>> SGA5.retract_okounkov_vershik(Integer(8)*SGA5([Integer(1),Integer(2),
→Integer(3),Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),
→Integer(2),Integer(4),Integer(5)]), Integer(1))
2*[1]
>>> SGA5.retract_okounkov_vershik(Integer(8)*SGA5([Integer(1),Integer(2),
→Integer(3),Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),
→Integer(2),Integer(4),Integer(5)]), Integer(0))
2*[]
```

See also

retract_plain(), *retract_direct_product()*

retract_plain(f, m)

Return the plain retract of the element $f \in RS_n$ to RS_m, where $m \leq n$ (and where RS_n is `self`).

If m is a nonnegative integer less or equal to n, then the plain retract from S_n to S_m is defined as an R-linear map $S_n \to S_m$ which sends every permutation $p \in S_n$ to

$$\begin{cases} \mathrm{pret}(p) & \text{if } \mathrm{pret}(p) \text{ is defined;} \\ 0 & \text{otherwise} \end{cases}.$$

Here $\mathrm{pret}(p)$ denotes the plain retract of the permutation p to S_m, which is defined in *retract_plain()*.

EXAMPLES:

```
sage: SGA3 = SymmetricGroupAlgebra(QQ, 3)
sage: SGA3.retract_plain(2*SGA3([1,2,3]) - 4*SGA3([2,1,3]) + 7*SGA3([1,3,2]),␣
↪2)
2*[1, 2] - 4*[2, 1]
sage: SGA3.retract_plain(2*SGA3([1,3,2]) - 5*SGA3([2,3,1]), 2)
0

sage: SGA5 = SymmetricGroupAlgebra(QQ, 5)
sage: SGA5.retract_plain(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) +␣
↪11*SGA5([3,2,1,4,5]), 4)
11*[3, 2, 1, 4]
sage: SGA5.retract_plain(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) +␣
↪11*SGA5([3,2,1,4,5]), 3)
11*[3, 2, 1]
sage: SGA5.retract_plain(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) +␣
↪11*SGA5([3,2,1,4,5]), 2)
0
sage: SGA5.retract_plain(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) +␣
↪11*SGA5([3,2,1,4,5]), 1)
0

sage: SGA5.retract_plain(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 3)
8*[1, 2, 3] - 6*[1, 3, 2]
sage: SGA5.retract_plain(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 1)
8*[1]
sage: SGA5.retract_plain(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 0)
8*[]
```

```
>>> from sage.all import *
>>> SGA3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> SGA3.retract_plain(Integer(2)*SGA3([Integer(1),Integer(2),Integer(3)]) -␣
↪Integer(4)*SGA3([Integer(2),Integer(1),Integer(3)]) +␣
↪Integer(7)*SGA3([Integer(1),Integer(3),Integer(2)]), Integer(2))
2*[1, 2] - 4*[2, 1]
>>> SGA3.retract_plain(Integer(2)*SGA3([Integer(1),Integer(3),Integer(2)]) -␣
↪Integer(5)*SGA3([Integer(2),Integer(3),Integer(1)]), Integer(2))
0

>>> SGA5 = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SGA5.retract_plain(Integer(8)*SGA5([Integer(1),Integer(4),Integer(2),
↪Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),Integer(2),
↪Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),Integer(2),
↪Integer(1),Integer(4),Integer(5)]), Integer(4))
11*[3, 2, 1, 4]
>>> SGA5.retract_plain(Integer(8)*SGA5([Integer(1),Integer(4),Integer(2),
↪Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),Integer(2),
↪Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),Integer(2),
↪Integer(1),Integer(4),Integer(5)]), Integer(3))
11*[3, 2, 1]
>>> SGA5.retract_plain(Integer(8)*SGA5([Integer(1),Integer(4),Integer(2),
↪Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),Integer(2),
↪Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(4),Integer(5)]), Integer(2))
0
>>> SGA5.retract_plain(Integer(8)*SGA5([Integer(1),Integer(4),Integer(2),
↪Integer(5),Integer(3)]) - Integer(6)*SGA5([Integer(1),Integer(3),Integer(2),
↪Integer(5),Integer(4)]) + Integer(11)*SGA5([Integer(3),Integer(2),
↪Integer(1),Integer(4),Integer(5)]), Integer(1))
0

>>> SGA5.retract_plain(Integer(8)*SGA5([Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),Integer(2),
↪Integer(4),Integer(5)]), Integer(3))
8*[1, 2, 3] - 6*[1, 3, 2]
>>> SGA5.retract_plain(Integer(8)*SGA5([Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),Integer(2),
↪Integer(4),Integer(5)]), Integer(1))
8*[1]
>>> SGA5.retract_plain(Integer(8)*SGA5([Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]) - Integer(6)*SGA5([Integer(1),Integer(3),Integer(2),
↪Integer(4),Integer(5)]), Integer(0))
8*[]
```

See also

retract_direct_product(), *retract_okounkov_vershik()*

right_action_product(*left*, *right*)

Return the product of two elements `left` and `right` of `self`, where multiplication is defined in such a way that for two permutations p and q, the product pq is the permutation obtained by first applying p and then applying q. This definition of multiplication is tailored to make multiplication of permutations associative with their action on numbers if permutations are to act on numbers from the right.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: p1 = Permutation([2, 1, 3])
sage: p2 = Permutation([3, 1, 2])
sage: QS3.right_action_product(QS3(p1), QS3(p2))
[1, 3, 2]
sage: x = QS3([1, 2, 3]) - 2*QS3([1, 3, 2])
sage: y = 1/2 * QS3([3, 1, 2]) + 3*QS3([1, 2, 3])
sage: QS3.right_action_product(x, y)
3*[1, 2, 3] - 6*[1, 3, 2] + 1/2*[3, 1, 2] - [3, 2, 1]
sage: QS3.right_action_product(0, x)
0
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> p1 = Permutation([Integer(2), Integer(1), Integer(3)])
>>> p2 = Permutation([Integer(3), Integer(1), Integer(2)])
>>> QS3.right_action_product(QS3(p1), QS3(p2))
[1, 3, 2]
```

(continues on next page)

(continued from previous page)

```
>>> x = QS3([Integer(1), Integer(2), Integer(3)]) -␣
↪Integer(2)*QS3([Integer(1), Integer(3), Integer(2)])
>>> y = Integer(1)/Integer(2) * QS3([Integer(3), Integer(1), Integer(2)]) +␣
↪Integer(3)*QS3([Integer(1), Integer(2), Integer(3)])
>>> QS3.right_action_product(x, y)
3*[1, 2, 3] - 6*[1, 3, 2] + 1/2*[3, 1, 2] - [3, 2, 1]
>>> QS3.right_action_product(Integer(0), x)
0
```

The method coerces its input into the algebra `self`:

```
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: QS4.right_action_product(QS3([1, 2, 3]), QS3([2, 1, 3]))
[2, 1, 3, 4]
sage: QS4.right_action_product(1, Permutation([4, 1, 2, 3]))
[4, 1, 2, 3]
```

```
>>> from sage.all import *
>>> QS4 = SymmetricGroupAlgebra(QQ, Integer(4))
>>> QS4.right_action_product(QS3([Integer(1), Integer(2), Integer(3)]),␣
↪QS3([Integer(2), Integer(1), Integer(3)]))
[2, 1, 3, 4]
>>> QS4.right_action_product(Integer(1), Permutation([Integer(4), Integer(1),␣
↪Integer(2), Integer(3)]))
[4, 1, 2, 3]
```

Warning

Note that coercion presently works from permutations of `n` into the `n`-th symmetric group algebra, and also from all smaller symmetric group algebras into the `n`-th symmetric group algebra, but not from permutations of integers smaller than `n` into the `n`-th symmetric group algebra.

rsw_shuffling_element(k)

Return the k-th Reiner-Saliola-Welker shuffling element in the group algebra `self`.

The k-th Reiner-Saliola-Welker shuffling element in the symmetric group algebra RS_n over a ring R is defined as the sum $\sum_{\sigma \in S_n} \mathrm{noninv}_k(\sigma) \cdot \sigma$, where for every permutation σ, the number $\mathrm{noninv}_k(\sigma)$ is the number of all k-noninversions of σ (that is, the number of all k-element subsets of $\{1, 2, \ldots, n\}$ on which σ restricts to a strictly increasing map). See `sage.combinat.permutation.number_of_noninversions()` for the noninv map.

This element is more or less the operator $\nu_{k,1^{n-k}}$ introduced in [RSW2011]; more precisely, $\nu_{k,1^{n-k}}$ is the left multiplication by this element.

It is a nontrivial theorem (Theorem 1.1 in [RSW2011]) that the operators $\nu_{k,1^{n-k}}$ (for fixed n and varying k) pairwise commute. It is a conjecture (Conjecture 1.2 in [RSW2011]) that all their eigenvalues are integers (which, in light of their commutativity and easily established symmetry, yields that they can be simultaneously diagonalized over $\mathbf{Q}$ with only integer eigenvalues).

EXAMPLES:

The Reiner-Saliola-Welker shuffling elements on $\mathbf{Q}S_3$:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.rsw_shuffling_element(0)
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: QS3.rsw_shuffling_element(1)
3*[1, 2, 3] + 3*[1, 3, 2] + 3*[2, 1, 3] + 3*[2, 3, 1] + 3*[3, 1, 2] + 3*[3, 2,
↪ 1]
sage: QS3.rsw_shuffling_element(2)
3*[1, 2, 3] + 2*[1, 3, 2] + 2*[2, 1, 3] + [2, 3, 1] + [3, 1, 2]
sage: QS3.rsw_shuffling_element(3)
[1, 2, 3]
sage: QS3.rsw_shuffling_element(4)
0
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> QS3.rsw_shuffling_element(Integer(0))
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
>>> QS3.rsw_shuffling_element(Integer(1))
3*[1, 2, 3] + 3*[1, 3, 2] + 3*[2, 1, 3] + 3*[2, 3, 1] + 3*[3, 1, 2] + 3*[3, 2,
↪ 1]
>>> QS3.rsw_shuffling_element(Integer(2))
3*[1, 2, 3] + 2*[1, 3, 2] + 2*[2, 1, 3] + [2, 3, 1] + [3, 1, 2]
>>> QS3.rsw_shuffling_element(Integer(3))
[1, 2, 3]
>>> QS3.rsw_shuffling_element(Integer(4))
0
```

Checking the commutativity of Reiner-Saliola-Welker shuffling elements (we leave out the ones for which it is trivial):

```
sage: def test_rsw_comm(n):
....:     QSn = SymmetricGroupAlgebra(QQ, n)
....:     rsws = [QSn.rsw_shuffling_element(k) for k in range(2, n)]
....:     return all(ri * rsws[j] == rsws[j] * ri
....:                for i, ri in enumerate(rsws) for j in range(i))
sage: test_rsw_comm(3)
True
sage: test_rsw_comm(4)    # long time
True
sage: test_rsw_comm(5)    # not tested
True
```

```
>>> from sage.all import *
>>> def test_rsw_comm(n):
...     QSn = SymmetricGroupAlgebra(QQ, n)
...     rsws = [QSn.rsw_shuffling_element(k) for k in range(Integer(2), n)]
...     return all(ri * rsws[j] == rsws[j] * ri
...                for i, ri in enumerate(rsws) for j in range(i))
>>> test_rsw_comm(Integer(3))
True
>>> test_rsw_comm(Integer(4))    # long time
True
```

(continues on next page)

(continued from previous page)

```
>>> test_rsw_comm(Integer(5))    # not tested
True
```

Note: For large k (relative to n), it might be faster to call `QSn.left_action_product(QSn.semi_rsw_element(k), QSn.antipode(binary_unshuffle_sum(k)))` than `QSn.rsw_shuffling_element(n)`.

See also:

`semi_rsw_element()`, `binary_unshuffle_sum()`

semi_rsw_element(k)

Return the k-th semi-RSW element in the group algebra `self`.

The k-th semi-RSW element in the symmetric group algebra RS_n over a ring R is defined as the sum of all permutations $\sigma \in S_n$ satisfying $\sigma(1) < \sigma(2) < \cdots < \sigma(k)$.

This element has the property that, if it is denoted by s_k, then $s_k S(s_k)$ is $(n-k)!$ times the k-th Reiner-Saliola-Welker shuffling element of RS_n (see `rsw_shuffling_element()`). Here, S denotes the antipode of the group algebra RS_n.

The k-th semi-RSW element is the image of the complete non-commutative symmetric function $S^{(k,1^{n-k})}$ in the ring of non-commutative symmetric functions under the canonical projection on the symmetric group algebra (through the descent algebra).

EXAMPLES:

The semi-RSW elements on $\mathbf{Q}S_3$:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.semi_rsw_element(0)
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: QS3.semi_rsw_element(1)
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: QS3.semi_rsw_element(2)
[1, 2, 3] + [1, 3, 2] + [2, 3, 1]
sage: QS3.semi_rsw_element(3)
[1, 2, 3]
sage: QS3.semi_rsw_element(4)
0
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> QS3.semi_rsw_element(Integer(0))
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
>>> QS3.semi_rsw_element(Integer(1))
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
>>> QS3.semi_rsw_element(Integer(2))
[1, 2, 3] + [1, 3, 2] + [2, 3, 1]
```

(continues on next page)

(continued from previous page)

```
>>> QS3.semi_rsw_element(Integer(3))
[1, 2, 3]
>>> QS3.semi_rsw_element(Integer(4))
0
```

Let us check the relation with the k-th Reiner-Saliola-Welker shuffling element stated in the docstring:

```
sage: def test_rsw(n):
....:     ZSn = SymmetricGroupAlgebra(ZZ, n)
....:     for k in range(1, n):
....:         a = ZSn.semi_rsw_element(k)
....:         b = ZSn.left_action_product(a, ZSn.antipode(a))
....:         if factorial(n-k) * ZSn.rsw_shuffling_element(k) != b:
....:             return False
....:     return True
sage: test_rsw(3)
True
sage: test_rsw(4)
True
sage: test_rsw(5)  # long time
True
```

```
>>> from sage.all import *
>>> def test_rsw(n):
...     ZSn = SymmetricGroupAlgebra(ZZ, n)
...     for k in range(Integer(1), n):
...         a = ZSn.semi_rsw_element(k)
...         b = ZSn.left_action_product(a, ZSn.antipode(a))
...         if factorial(n-k) * ZSn.rsw_shuffling_element(k) != b:
...             return False
...     return True
>>> test_rsw(Integer(3))
True
>>> test_rsw(Integer(4))
True
>>> test_rsw(Integer(5))  # long time
True
```

Let us also check the statement about the complete non-commutative symmetric function:

```
sage: def test_rsw_ncsf(n):
....:     ZSn = SymmetricGroupAlgebra(ZZ, n)
....:     NSym = NonCommutativeSymmetricFunctions(ZZ)
....:     S = NSym.S()
....:     for k in range(1, n):
....:         a = S(Composition([k] + [1]*(n-k))).to_symmetric_group_algebra()
....:         if a != ZSn.semi_rsw_element(k):
....:             return False
....:     return True
sage: test_rsw_ncsf(3)
True
sage: test_rsw_ncsf(4)
```

(continues on next page)

(continued from previous page)

```
True
sage: test_rsw_ncsf(5)  # long time
True
```

```
>>> from sage.all import *
>>> def test_rsw_ncsf(n):
...     ZSn = SymmetricGroupAlgebra(ZZ, n)
...     NSym = NonCommutativeSymmetricFunctions(ZZ)
...     S = NSym.S()
...     for k in range(Integer(1), n):
...         a = S(Composition([k] + [Integer(1)]*(n-k))).to_symmetric_group_
↪algebra()
...         if a != ZSn.semi_rsw_element(k):
...             return False
...     return True
>>> test_rsw_ncsf(Integer(3))
True
>>> test_rsw_ncsf(Integer(4))
True
>>> test_rsw_ncsf(Integer(5))  # long time
True
```

seminormal_basis(*mult='l2r'*)

Return a list of the seminormal basis elements of `self`.

The seminormal basis of a symmetric group algebra is defined as follows:

Let n be a nonnegative integer. Let R be a $\mathbf{Q}$-algebra. In the following, we will use the "left action" convention for multiplying permutations. This means that for all permutations p and q in S_n, the product pq is defined in such a way that $(pq)(i) = p(q(i))$ for each $i \in \{1, 2, \ldots, n\}$ (this is the same convention as in `left_action_product()`, but not the default semantics of the $*$ operator on permutations in Sage). Thus, for instance, $s_2 s_1$ is the permutation obtained by first transposing 1 with 2 and then transposing 2 with 3 (where $s_i = (i, i+1)$).

For every partition λ of n, let

$$\kappa_\lambda = \frac{n!}{f^\lambda}$$

where f^λ is the number of standard Young tableaux of shape λ. Note that κ_λ is an integer, namely the product of all hook lengths of λ (by the hook length formula). In Sage, this integer can be computed by using `sage.combinat.symmetric_group_algebra.kappa()`.

Let T be a standard tableau of size n.

Let $a(T)$ denote the formal sum (in RS_n) of all permutations in S_n which stabilize the rows of T (as sets), i. e., which map each entry i of T to an entry in the same row as i. (See `sage.combinat.symmetric_group_algebra.a()` for an implementation of this.)

Let $b(T)$ denote the signed formal sum (in RS_n) of all permutations in S_n which stabilize the columns of T (as sets). Here, "signed" means that each permutation is multiplied with its sign. (This is implemented in `sage.combinat.symmetric_group_algebra.b()`.)

Define an element $e(T)$ of RS_n to be $a(T)b(T)$. (This is implemented in `sage.combinat.symmetric_group_algebra.e()` for $R = \mathbf{Q}$.)

Let $\mathrm{sh}(T)$ denote the shape of T. (See `shape()`.)

Let $\overline{T}$ denote the standard tableau of size $n-1$ obtained by removing the letter n (along with its cell) from T (if $n \geq 1$).

Now, we define an element $\epsilon(T)$ of RS_n. We define it by induction on the size n of T, so we set $\epsilon(\emptyset) = 1$ and only need to define $\epsilon(T)$ for $n \geq 1$, assuming that $\epsilon(\overline{T})$ is already defined. We do this by setting

$$\epsilon(T) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{T}) e(T) \epsilon(\overline{T}).$$

This element $\epsilon(T)$ is implemented as `sage.combinat.symmetric_group_algebra.epsilon()` for $R = \mathbf{Q}$, but it is also a particular case of the elements $\epsilon(T, S)$ defined below.

Now let S be a further tableau of the same shape as T (possibly equal to T). Let $\pi_{T,S}$ denote the permutation in S_n such that applying this permutation to the entries of T yields the tableau S. Define an element $\epsilon(T, S)$ of RS_n by

$$\epsilon(T, S) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{S}) \pi_{T,S} e(T) \epsilon(\overline{T}) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{S}) a(S) \pi_{T,S} b(T) \epsilon(\overline{T}).$$

This element $\epsilon(T, S)$ is called *Young's seminormal unit corresponding to the bitableau `(T, S)`*, and is the return value of `epsilon_ik()` applied to `T` and `S`. Note that $\epsilon(T, T) = \epsilon(T)$.

If we let λ run through all partitions of n, and (T, S) run through all pairs of tableaux of shape λ, then the elements $\epsilon(T, S)$ form a basis of RS_n. This basis is called *Young's seminormal basis* and has the properties that

$$\epsilon(T, S) \epsilon(U, V) = \delta_{T,V} \epsilon(U, S)$$

(where δ stands for the Kronecker delta).

Warning: Because of our convention, we are multiplying our elements in reverse of those given in some papers, for example [Ram1997]. Using the other convention of multiplying permutations, we would instead have $\epsilon(U, V) \epsilon(T, S) = \delta_{T,V} \epsilon(U, S)$.

In other words, Young's seminormal basis consists of the matrix units in a (particular) Artin-Wedderburn decomposition of RS_n into a direct product of matrix algebras over $\mathbf{Q}$.

The output of `seminormal_basis()` is a list of all elements of the seminormal basis of `self`.

INPUT:

- `mult` – string (default: `'l2r'`); if set to `'r2l'`, this causes the method to return the list of the antipodes (`antipode()`) of all $\epsilon(T, S)$ instead of the $\epsilon(T, S)$ themselves.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: QS3.seminormal_basis()
[1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1,␣
↪2] + 1/6*[3, 2, 1],
1/3*[1, 2, 3] + 1/6*[1, 3, 2] - 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2]␣
↪+ 1/6*[3, 2, 1],
1/3*[1, 3, 2] + 1/3*[2, 3, 1] - 1/3*[3, 1, 2] - 1/3*[3, 2, 1],
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1],
1/3*[1, 2, 3] - 1/6*[1, 3, 2] + 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2]␣
```

(continues on next page)

(continued from previous page)

```
↪- 1/6*[3, 2, 1],
1/6*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2]␣
↪- 1/6*[3, 2, 1]]
```

```
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ,Integer(3))
>>> QS3.seminormal_basis()
[1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1,␣
↪2] + 1/6*[3, 2, 1],
1/3*[1, 2, 3] + 1/6*[1, 3, 2] - 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2]␣
↪+ 1/6*[3, 2, 1],
1/3*[1, 3, 2] + 1/3*[2, 3, 1] - 1/3*[3, 1, 2] - 1/3*[3, 2, 1],
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1],
1/3*[1, 2, 3] - 1/6*[1, 3, 2] + 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2]␣
↪- 1/6*[3, 2, 1],
1/6*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2]␣
↪- 1/6*[3, 2, 1]]
```

simple_module(*la*)

Return the simple module of `self` indexed by the partition `la`.

Over a field of characteristic 0, this simply returns the Specht module.

See also

sage.combinat.specht_module.SimpleModule

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 5)
sage: D = SGA.simple_module(Partition([3,1,1]))
sage: D
Simple module of [3, 1, 1] over Finite Field of size 3
sage: D.brauer_character()
(6, 0, -2, 0, 1)
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(5))
>>> D = SGA.simple_module(Partition([Integer(3),Integer(1),Integer(1)]))
>>> D
Simple module of [3, 1, 1] over Finite Field of size 3
>>> D.brauer_character()
(6, 0, -2, 0, 1)
```

simple_module_dimension(*la*)

Return the dimension of the simple module of `self` indexed by the partition `la`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(5), 6)
sage: SGA.simple_module_dimension(Partition([4,1,1]))
10
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(5)), Integer(6))
>>> SGA.simple_module_dimension(Partition([Integer(4),Integer(1),Integer(1)]))
10
```

simple_module_parameterization()

Return a parameterization of the simple modules of `self`.

The symmetric group algebra of S_n over a field of characteristic p has its simple modules indexed by all p-regular partitions of n.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 6)
sage: SGA.simple_module_parameterization()
Partitions of the integer 6

sage: SGA = SymmetricGroupAlgebra(GF(2), 6)
sage: SGA.simple_module_parameterization()
2-Regular Partitions of the integer 6
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(6))
>>> SGA.simple_module_parameterization()
Partitions of the integer 6

>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(6))
>>> SGA.simple_module_parameterization()
2-Regular Partitions of the integer 6
```

specht_module(D)

Return the Specht module of `self` indexed by the diagram `D`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SM = SGA.specht_module(Partition([3,1,1]))
sage: SM
Specht module of [3, 1, 1] over Rational Field
sage: SM.frobenius_image()
s[3, 1, 1]

sage: SM = SGA.specht_module([(1,1),(1,3),(2,2),(3,1),(3,2)])
sage: SM
Specht module of [(1, 1), (1, 3), (2, 2), (3, 1), (3, 2)] over Rational Field
sage: SM.frobenius_image()
s[2, 2, 1] + s[3, 1, 1] + s[3, 2]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SM = SGA.specht_module(Partition([Integer(3),Integer(1),Integer(1)]))
>>> SM
Specht module of [3, 1, 1] over Rational Field
>>> SM.frobenius_image()
```

(continues on next page)

(continued from previous page)

```
s[3, 1, 1]

>>> SM = SGA.specht_module([(Integer(1),Integer(1)),(Integer(1),Integer(3)),
↪(Integer(2),Integer(2)),(Integer(3),Integer(1)),(Integer(3),Integer(2))])
>>> SM
Specht module of [(1, 1), (1, 3), (2, 2), (3, 1), (3, 2)] over Rational Field
>>> SM.frobenius_image()
s[2, 2, 1] + s[3, 1, 1] + s[3, 2]
```

specht_module_dimension(*D*)

Return the dimension of the Specht module of `self` indexed by `D`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SGA.specht_module_dimension(Partition([3,1,1]))
6
sage: SGA.specht_module_dimension([(1,1),(1,3),(2,2),(3,1),(3,2)])
16
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SGA.specht_module_dimension(Partition([Integer(3),Integer(1),Integer(1)]))
6
>>> SGA.specht_module_dimension([(Integer(1),Integer(1)),(Integer(1),
↪Integer(3)),(Integer(2),Integer(2)),(Integer(3),Integer(1)),(Integer(3),
↪Integer(2))])
16
```

tabloid_module(*D*)

Return the module of tabloids with the natural action of `self`.

See also

TabloidModule

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: TM = SGA.tabloid_module(Partition([3,1,1]))
sage: TM
Tabloid module of [3, 1, 1] over Rational Field
sage: s = SymmetricFunctions(QQ).s()
sage: s(TM.frobenius_image())
s[3, 1, 1] + s[3, 2] + 2*s[4, 1] + s[5]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> TM = SGA.tabloid_module(Partition([Integer(3),Integer(1),Integer(1)]))
>>> TM
Tabloid module of [3, 1, 1] over Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> s = SymmetricFunctions(QQ).s()
>>> s(TM.frobenius_image())
s[3, 1, 1] + s[3, 2] + 2*s[4, 1] + s[5]
```

young_symmetrizer(*la*)

Return the Young symmetrizer of shape `la` of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(3))
sage: SGA.young_symmetrizer([2,1])
() + (1,2) - (1,3,2) - (1,3)
sage: SGA = SymmetricGroupAlgebra(QQ, 4)
sage: SGA.young_symmetrizer([2,1,1])
[1, 2, 3, 4] - [1, 2, 4, 3] + [2, 1, 3, 4] - [2, 1, 4, 3]
 - [3, 1, 2, 4] + [3, 1, 4, 2] - [3, 2, 1, 4] + [3, 2, 4, 1]
 + [4, 1, 2, 3] - [4, 1, 3, 2] + [4, 2, 1, 3] - [4, 2, 3, 1]
sage: SGA.young_symmetrizer([5,1,1])
Traceback (most recent call last):
...
ValueError: the partition [5, 1, 1] is not of size 4
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(Integer(3)))
>>> SGA.young_symmetrizer([Integer(2),Integer(1)])
() + (1,2) - (1,3,2) - (1,3)
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(4))
>>> SGA.young_symmetrizer([Integer(2),Integer(1),Integer(1)])
[1, 2, 3, 4] - [1, 2, 4, 3] + [2, 1, 3, 4] - [2, 1, 4, 3]
 - [3, 1, 2, 4] + [3, 1, 4, 2] - [3, 2, 1, 4] + [3, 2, 4, 1]
 + [4, 1, 2, 3] - [4, 1, 3, 2] + [4, 2, 1, 3] - [4, 2, 3, 1]
>>> SGA.young_symmetrizer([Integer(5),Integer(1),Integer(1)])
Traceback (most recent call last):
...
ValueError: the partition [5, 1, 1] is not of size 4
```

`sage.combinat.symmetric_group_algebra.`**a**(*tableau*, *star=0*, *base_ring=Rational Field*)

The row projection operator corresponding to the Young tableau `tableau` (which is supposed to contain every integer from 1 to its size precisely once, but may and may not be standard).

This is the sum (in the group algebra of the relevant symmetric group over **Q**) of all the permutations which preserve the rows of `tableau`. It is called a_{tableau} in [EGHLSVY], Section 4.2.

INPUT:

- `tableau` – Young tableau which contains every integer from 1 to its size precisely once
- `star` – nonnegative integer (default: 0); when this optional variable is set, the method computes not the row projection operator of `tableau`, but the row projection operator of the restriction of `tableau` to the entries `1, 2, ..., tableau.size() - star` instead.
- `base_ring` – commutative ring (default: `QQ`); when this optional variable is set, the row projection operator is computed over a user-determined base ring instead of **Q**. (Note that symmetric group algebras currently don't preserve coercion, so e. g. a symmetric group algebra over **Z** does not coerce into the corresponding one over **Q**; so convert manually or choose your base rings wisely!)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import a
sage: a([[1,2]])
[1, 2] + [2, 1]
sage: a([[1],[2]])
[1, 2]
sage: a([])
[]
sage: a([[1, 5], [2, 3], [4]])
[1, 2, 3, 4, 5] + [1, 3, 2, 4, 5] + [5, 2, 3, 4, 1] + [5, 3, 2, 4, 1]
sage: a([[1,4], [2,3]], base_ring=ZZ)
[1, 2, 3, 4] + [1, 3, 2, 4] + [4, 2, 3, 1] + [4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import a
>>> a([[Integer(1),Integer(2)]])
[1, 2] + [2, 1]
>>> a([[Integer(1)],[Integer(2)]])
[1, 2]
>>> a([])
[]
>>> a([[Integer(1), Integer(5)], [Integer(2), Integer(3)], [Integer(4)]])
[1, 2, 3, 4, 5] + [1, 3, 2, 4, 5] + [5, 2, 3, 4, 1] + [5, 3, 2, 4, 1]
>>> a([[Integer(1),Integer(4)], [Integer(2),Integer(3)]], base_ring=ZZ)
[1, 2, 3, 4] + [1, 3, 2, 4] + [4, 2, 3, 1] + [4, 3, 2, 1]
```

The same with a skew tableau:

```
sage: a([[None,1,4], [2,3]], base_ring=ZZ)
[1, 2, 3, 4] + [1, 3, 2, 4] + [4, 2, 3, 1] + [4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> a([[None,Integer(1),Integer(4)], [Integer(2),Integer(3)]], base_ring=ZZ)
[1, 2, 3, 4] + [1, 3, 2, 4] + [4, 2, 3, 1] + [4, 3, 2, 1]
```

`sage.combinat.symmetric_group_algebra.`**b**(*tableau*, *star=0*, *base_ring=Rational Field*)

The column projection operator corresponding to the Young tableau `tableau` (which is supposed to contain every integer from 1 to its size precisely once, but may and may not be standard).

This is the signed sum (in the group algebra of the relevant symmetric group over $\mathbf{Q}$) of all the permutations which preserve the column of `tableau` (where the signs are the usual signs of the permutations). It is called b_{tableau} in [EGHLSVY], Section 4.2.

INPUT:

- `tableau` – Young tableau which contains every integer from 1 to its size precisely once
- `star` – nonnegative integer (default: 0). When this optional variable is set, the method computes not the column projection operator of `tableau`, but the column projection operator of the restriction of `tableau` to the entries `1, 2, ..., tableau.size() - star` instead.
- `base_ring` – commutative ring (default: `QQ`). When this optional variable is set, the column projection operator is computed over a user-determined base ring instead of $\mathbf{Q}$. (Note that symmetric group algebras currently don't preserve coercion, so e. g. a symmetric group algebra over $\mathbf{Z}$ does not coerce into the corresponding one over $\mathbf{Q}$; so convert manually or choose your base rings wisely!)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import b
sage: b([[1,2]])
[1, 2]
sage: b([[1],[2]])
[1, 2] - [2, 1]
sage: b([])
[]
sage: b([[1, 2, 4], [5, 3]])
[1, 2, 3, 4, 5] - [1, 3, 2, 4, 5] - [5, 2, 3, 4, 1] + [5, 3, 2, 4, 1]
sage: b([[1, 4], [2, 3]], base_ring=ZZ)
[1, 2, 3, 4] - [1, 2, 4, 3] - [2, 1, 3, 4] + [2, 1, 4, 3]
sage: b([[1, 4], [2, 3]], base_ring=Integers(5))
[1, 2, 3, 4] + 4*[1, 2, 4, 3] + 4*[2, 1, 3, 4] + [2, 1, 4, 3]
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import b
>>> b([[Integer(1),Integer(2)]])
[1, 2]
>>> b([[Integer(1)],[Integer(2)]])
[1, 2] - [2, 1]
>>> b([])
[]
>>> b([[Integer(1), Integer(2), Integer(4)], [Integer(5), Integer(3)]])
[1, 2, 3, 4, 5] - [1, 3, 2, 4, 5] - [5, 2, 3, 4, 1] + [5, 3, 2, 4, 1]
>>> b([[Integer(1), Integer(4)], [Integer(2), Integer(3)]], base_ring=ZZ)
[1, 2, 3, 4] - [1, 2, 4, 3] - [2, 1, 3, 4] + [2, 1, 4, 3]
>>> b([[Integer(1), Integer(4)], [Integer(2), Integer(3)]], base_
↪ring=Integers(Integer(5)))
[1, 2, 3, 4] + 4*[1, 2, 4, 3] + 4*[2, 1, 3, 4] + [2, 1, 4, 3]
```

The same with a skew tableau:

```
sage: b([[None, 2, 4], [1, 3], [5]])
[1, 2, 3, 4, 5] - [1, 3, 2, 4, 5] - [5, 2, 3, 4, 1] + [5, 3, 2, 4, 1]
```

```
>>> from sage.all import *
>>> b([[None, Integer(2), Integer(4)], [Integer(1), Integer(3)], [Integer(5)]])
[1, 2, 3, 4, 5] - [1, 3, 2, 4, 5] - [5, 2, 3, 4, 1] + [5, 3, 2, 4, 1]
```

With the `l2r` setting for multiplication, the unnormalized Young symmetrizer `e(tableau)` should be the product `b(tableau) * a(tableau)` for every `tableau`. Let us check this on the standard tableaux of size 5:

```
sage: from sage.combinat.symmetric_group_algebra import a, b, e
sage: all( e(t) == b(t) * a(t) for t in StandardTableaux(5) )
True
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import a, b, e
>>> all( e(t) == b(t) * a(t) for t in StandardTableaux(Integer(5)) )
True
```

`sage.combinat.symmetric_group_algebra.`**`e`**`(`*tableau*, *star=0*`)`

The unnormalized Young projection operator corresponding to the Young tableau `tableau` (which is supposed to contain every integer from 1 to its size precisely once, but may and may not be standard).

If n is a nonnegative integer, and T is a Young tableau containing every integer from 1 to n exactly once, then the unnormalized Young projection operator $e(T)$ is defined by

$$e(T) = a(T)b(T) \in \mathbf{Q}S_n,$$

where $a(T) \in \mathbf{Q}S_n$ is the sum of all permutations in S_n which fix the rows of T (as sets), and $b(T) \in \mathbf{Q}S_n$ is the signed sum of all permutations in S_n which fix the columns of T (as sets). Here, "signed" means that each permutation is multiplied with its sign; and the product on the group S_n is defined in such a way that $(pq)(i) = p(q(i))$ for any permutations p and q and any $1 \leq i \leq n$.

Note that the definition of $e(T)$ is not uniform across literature. Others define it as $b(T)a(T)$ instead, or include certain scalar factors (we do not, whence "unnormalized").

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e
sage: e([[1,2]])
[1, 2] + [2, 1]
sage: e([[1],[2]])
[1, 2] - [2, 1]
sage: e([])
[]
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import e
>>> e([[Integer(1),Integer(2)]])
[1, 2] + [2, 1]
>>> e([[Integer(1)],[Integer(2)]])
[1, 2] - [2, 1]
>>> e([])
[]
```

There are differing conventions for the order of the symmetrizers and antisymmetrizers. This example illustrates our conventions:

```
sage: e([[1,2],[3]])
[1, 2, 3] + [2, 1, 3] - [3, 1, 2] - [3, 2, 1]
```

```
>>> from sage.all import *
>>> e([[Integer(1),Integer(2)],[Integer(3)]])
[1, 2, 3] + [2, 1, 3] - [3, 1, 2] - [3, 2, 1]
```

To obtain the product $b(T)a(T)$, one has to take the antipode of this:

```
sage: QS3 = parent(e([[1,2],[3]]))
sage: QS3.antipode(e([[1,2],[3]]))
[1, 2, 3] + [2, 1, 3] - [2, 3, 1] - [3, 2, 1]
```

```
>>> from sage.all import *
>>> QS3 = parent(e([[Integer(1),Integer(2)],[Integer(3)]]))
>>> QS3.antipode(e([[Integer(1),Integer(2)],[Integer(3)]]))
[1, 2, 3] + [2, 1, 3] - [2, 3, 1] - [3, 2, 1]
```

And here is an example for a skew tableau:

```
sage: e([[None, 2, 1], [4, 3]])
[1, 2, 3, 4] + [1, 2, 4, 3] - [1, 3, 2, 4] - [1, 4, 2, 3]
 + [2, 1, 3, 4] + [2, 1, 4, 3] - [2, 3, 1, 4] - [2, 4, 1, 3]
```

```
>>> from sage.all import *
>>> e([[None, Integer(2), Integer(1)], [Integer(4), Integer(3)]])
[1, 2, 3, 4] + [1, 2, 4, 3] - [1, 3, 2, 4] - [1, 4, 2, 3]
 + [2, 1, 3, 4] + [2, 1, 4, 3] - [2, 3, 1, 4] - [2, 4, 1, 3]
```

See also

e_hat()

sage.combinat.symmetric_group_algebra.**e_hat**(*tab*, *star=0*)

The Young projection operator corresponding to the Young tableau tab (which is supposed to contain every integer from 1 to its size precisely once, but may and may not be standard). This is an idempotent in the rational group algebra.

If n is a nonnegative integer, and T is a Young tableau containing every integer from 1 to n exactly once, then the Young projection operator $\widehat{e}(T)$ is defined by

$$\widehat{e}(T) = \frac{1}{\kappa_\lambda} a(T) b(T) \in \mathbf{Q} S_n,$$

where λ is the shape of T, where κ_λ is $n!$ divided by the number of standard tableaux of shape λ, where $a(T) \in \mathbf{Q} S_n$ is the sum of all permutations in S_n which fix the rows of T (as sets), and where $b(T) \in \mathbf{Q} S_n$ is the signed sum of all permutations in S_n which fix the columns of T (as sets). Here, "signed" means that each permutation is multiplied with its sign; and the product on the group S_n is defined in such a way that $(pq)(i) = p(q(i))$ for any permutations p and q and any $1 \leq i \leq n$.

Note that the definition of $\widehat{e}(T)$ is not uniform across literature. Others define it as $\frac{1}{\kappa_\lambda} b(T) a(T)$ instead.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e_hat
sage: e_hat([[1,2,3]])
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/
↪6*[3, 2, 1]
sage: e_hat([[1],[2]])
1/2*[1, 2] - 1/2*[2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import e_hat
>>> e_hat([[Integer(1),Integer(2),Integer(3)]])
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/
↪6*[3, 2, 1]
>>> e_hat([[Integer(1)],[Integer(2)]])
1/2*[1, 2] - 1/2*[2, 1]
```

There are differing conventions for the order of the symmetrizers and antisymmetrizers. This example illustrates our conventions:

```
sage: e_hat([[1,2],[3]])
1/3*[1, 2, 3] + 1/3*[2, 1, 3] - 1/3*[3, 1, 2] - 1/3*[3, 2, 1]
```

```
>>> from sage.all import *
>>> e_hat([[Integer(1),Integer(2)],[Integer(3)]])
1/3*[1, 2, 3] + 1/3*[2, 1, 3] - 1/3*[3, 1, 2] - 1/3*[3, 2, 1]
```

See also

e()

sage.combinat.symmetric_group_algebra.**e_ik**(*itab*, *ktab*, *star=0*)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e_ik
sage: e_ik([[1,2,3]], [[1,2,3]])
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: e_ik([[1,2,3]], [[1,2,3]], star=1)
[1, 2] + [2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import e_ik
>>> e_ik([[Integer(1),Integer(2),Integer(3)]], [[Integer(1),Integer(2),
→Integer(3)]])
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
>>> e_ik([[Integer(1),Integer(2),Integer(3)]], [[Integer(1),Integer(2),
→Integer(3)]], star=Integer(1))
[1, 2] + [2, 1]
```

sage.combinat.symmetric_group_algebra.**epsilon**(*tab*, *star=0*)

The (T, T)-th element of the seminormal basis of the group algebra $\mathbf{Q}[S_n]$, where T is the tableau `tab` (with its `star` highest entries removed if the optional variable `star` is set).

See the docstring of *seminormal_basis()* for the notation used herein.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import epsilon
sage: epsilon([[1,2]])
1/2*[1, 2] + 1/2*[2, 1]
sage: epsilon([[1],[2]])
1/2*[1, 2] - 1/2*[2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import epsilon
>>> epsilon([[Integer(1),Integer(2)]])
1/2*[1, 2] + 1/2*[2, 1]
>>> epsilon([[Integer(1)],[Integer(2)]])
1/2*[1, 2] - 1/2*[2, 1]
```

sage.combinat.symmetric_group_algebra.**epsilon_ik**(*itab*, *ktab*, *star=0*)

Return the seminormal basis element of the symmetric group algebra $\mathbf{Q}S_n$ corresponding to the pair of tableaux itab and ktab (or restrictions of these tableaux, if the optional variable star is set).

INPUT:

- itab, ktab – two standard tableaux of same size
- star – integer (default: 0)

OUTPUT:

The element $\epsilon(I, K) \in \mathbf{Q}S_n$, where I and K are the tableaux obtained by removing all entries higher than $n -$ star from itab and ktab, respectively (where n is the size of itab and ktab). Here, we are using the notations from *seminormal_basis()*.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import epsilon_ik
sage: epsilon_ik([[1,2],[3]], [[1,3],[2]])
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1]
sage: epsilon_ik([[1,2],[3]], [[1,3],[2]], star=1)
Traceback (most recent call last):
...
ValueError: the two tableaux must be of the same shape
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import epsilon_ik
>>> epsilon_ik([[Integer(1),Integer(2)],[Integer(3)]], [[Integer(1),Integer(3)],
↪[Integer(2)]])
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1]
>>> epsilon_ik([[Integer(1),Integer(2)],[Integer(3)]], [[Integer(1),Integer(3)],
↪[Integer(2)]], star=Integer(1))
Traceback (most recent call last):
...
ValueError: the two tableaux must be of the same shape
```

sage.combinat.symmetric_group_algebra.**kappa**(*alpha*)

Return κ_α, which is $n!$ divided by the number of standard tableaux of shape α (where α is a partition of n).

INPUT:

- alpha – integer partition (can be encoded as a list)

OUTPUT:

The factorial of the size of alpha, divided by the number of standard tableaux of shape alpha. Equivalently, the product of all hook lengths of alpha.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import kappa
sage: kappa(Partition([2,1]))
3
sage: kappa([2,1])
3
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import kappa
>>> kappa(Partition([Integer(2),Integer(1)]))
```

(continues on next page)

(continued from previous page)

```
3
>>> kappa([Integer(2),Integer(1)])
3
```

sage.combinat.symmetric_group_algebra.**pi_ik**(*itab*, *ktab*)

Return the permutation p which sends every entry of the tableau `itab` to the respective entry of the tableau `ktab`, as an element of the corresponding symmetric group algebra.

This assumes that `itab` and `ktab` are tableaux (possibly given just as lists of lists) of the same shape. Both tableaux are allowed to be skew.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import pi_ik
sage: pi_ik([[1,3],[2]], [[1,2],[3]])
[1, 3, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import pi_ik
>>> pi_ik([[Integer(1),Integer(3)],[Integer(2)]], [[Integer(1),Integer(2)],
↪[Integer(3)]])
[1, 3, 2]
```

The same with skew tableaux:

```
sage: from sage.combinat.symmetric_group_algebra import pi_ik
sage: pi_ik([[None,1,3],[2]], [[None,1,2],[3]])
[1, 3, 2]
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import pi_ik
>>> pi_ik([[None,Integer(1),Integer(3)],[Integer(2)]], [[None,Integer(1),
↪Integer(2)],[Integer(3)]])
[1, 3, 2]
```

sage.combinat.symmetric_group_algebra.**seminormal_test**(*n*)

Run a variety of tests to verify that the construction of the seminormal basis works as desired. The numbers appearing are results in James and Kerber's 'Representation Theory of the Symmetric Group' [JK1981].

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import seminormal_test
sage: seminormal_test(3)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_algebra import seminormal_test
>>> seminormal_test(Integer(3))
True
```

5.1.343 Representations of the Symmetric Group

Todo

- construct the product of two irreducible representations.
- implement Induction/Restriction of representations.

Warning

This code uses a different convention than in Sagan's book "The Symmetric Group"

class sage.combinat.symmetric_group_representations.**GarsiaProcesiModule**(*SGA*, *shape*)

Bases: UniqueRepresentation, QuotientRing_generic, *SymmetricGroupRepresentation*

A Garsia-Procesi module.

Let λ be a partition of n and R be a commutative ring. The *Garsia-Procesi module* is defined by $R_\lambda := R[x_1, \ldots, x_n]/I_\lambda$, where

$$I_\lambda := \langle e_r(x_{i_1}, \ldots, x_{i_k}) \mid \{i_1, \ldots, i_k\} \subseteq [n] \text{ and } k \geq r > k - d_k(\lambda) \rangle,$$

with e_r being the r-the elementary symmetric function and $d_k(\lambda) = \lambda'_n + \cdots + \lambda'_{n+1-k}$, is the *Tanisaki ideal*.

If we consider $R = \mathbf{Q}$, then the Garsia-Procesi module has the following interpretation. Let $\mathcal{F}_n = GL_n/B$ denote the (complex type A) flag variety. Consider the Springer fiber $F_\lambda \subseteq \mathcal{F}_n$ associated to a nilpotent matrix with Jordan blocks sizes λ. Springer showed that the cohomology ring $H^*(F_\lambda)$ admits a graded S_n-action that agrees with the induced representation of the sign representation of the Young subgroup S_λ. From work of De Concini and Procesi, this S_n-representation is isomorphic to R_λ. Moreover, the graded Frobenius image is known to be a modified Hall-Littlewood polynomial.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 7)
sage: GP421 = SGA.garsia_procesi_module([4, 2, 1])
sage: GP421.dimension()
105
sage: v = GP421.an_element(); v
-gp1 - gp2 - gp3 - gp4 - gp5 - gp6
sage: SGA.an_element() * v
-6*gp1 - 6*gp2 - 6*gp3 - 6*gp4 - 6*gp5 - 5*gp6
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(7))
>>> GP421 = SGA.garsia_procesi_module([Integer(4), Integer(2), Integer(1)])
>>> GP421.dimension()
105
>>> v = GP421.an_element(); v
-gp1 - gp2 - gp3 - gp4 - gp5 - gp6
>>> SGA.an_element() * v
-6*gp1 - 6*gp2 - 6*gp3 - 6*gp4 - 6*gp5 - 5*gp6
```

We verify the result is a modified Hall-Littlewood polynomial by using the Q' Hall-Littlewood polynomials, replacing $q \mapsto q^{-1}$ and multiplying by the smallest power of q so the coefficients are again polynomials:

```
sage: GP421.graded_frobenius_image()
q^4*s[4, 2, 1] + q^3*s[4, 3] + q^3*s[5, 1, 1] + (q^3+q^2)*s[5, 2]
 + (q^2+q)*s[6, 1] + s[7]
sage: R.<q> = QQ[]
sage: Sym = SymmetricFunctions(R)
sage: s = Sym.s()
sage: Qp = Sym.hall_littlewood(q).Qp()
sage: mHL = s(Qp[4,2,1]); mHL
s[4, 2, 1] + q*s[4, 3] + q*s[5, 1, 1] + (q^2+q)*s[5, 2]
 + (q^3+q^2)*s[6, 1] + q^4*s[7]
sage: mHL.map_coefficients(lambda c: R(q^4*c(q^-1)))
q^4*s[4, 2, 1] + q^3*s[4, 3] + q^3*s[5, 1, 1] + (q^3+q^2)*s[5, 2]
 + (q^2+q)*s[6, 1] + s[7]
```

```
>>> from sage.all import *
>>> GP421.graded_frobenius_image()
q^4*s[4, 2, 1] + q^3*s[4, 3] + q^3*s[5, 1, 1] + (q^3+q^2)*s[5, 2]
 + (q^2+q)*s[6, 1] + s[7]
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> Sym = SymmetricFunctions(R)
>>> s = Sym.s()
>>> Qp = Sym.hall_littlewood(q).Qp()
>>> mHL = s(Qp[Integer(4),Integer(2),Integer(1)]); mHL
s[4, 2, 1] + q*s[4, 3] + q*s[5, 1, 1] + (q^2+q)*s[5, 2]
 + (q^3+q^2)*s[6, 1] + q^4*s[7]
>>> mHL.map_coefficients(lambda c: R(q**Integer(4)*c(q**-Integer(1))))
q^4*s[4, 2, 1] + q^3*s[4, 3] + q^3*s[5, 1, 1] + (q^3+q^2)*s[5, 2]
 + (q^2+q)*s[6, 1] + s[7]
```

We show that the maximal degree component corresponds to the Yamanouchi words of content λ:

```
sage: B = GP421.graded_decomposition(4).basis()
sage: top_deg = [Word([i+1 for i in b.lift().lift().exponents()[0]]) for b in B]
sage: yamanouchi = [P.to_packed_word() for P in OrderedSetPartitions(range(7), [4,
↪ 2, 1])
....:                 if P.to_packed_word().reversal().is_yamanouchi()]
sage: set(top_deg) == set(yamanouchi)
True
```

```
>>> from sage.all import *
>>> B = GP421.graded_decomposition(Integer(4)).basis()
>>> top_deg = [Word([i+Integer(1) for i in b.lift().lift().
↪exponents()[Integer(0)]]) for b in B]
>>> yamanouchi = [P.to_packed_word() for P in␣
↪OrderedSetPartitions(range(Integer(7)), [Integer(4), Integer(2), Integer(1)])
...                 if P.to_packed_word().reversal().is_yamanouchi()]
>>> set(top_deg) == set(yamanouchi)
True
```

class Element(*parent*, *rep*, *reduce=True*)

Bases: `QuotientRingElement`

degree()

Return the degree of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 4)
sage: GP22 = SGA.garsia_procesi_module([2, 2])
sage: for b in GP22.basis():
....:     print(b, b.degree())
gp2*gp3 2
gp1*gp3 2
gp3 1
gp2 1
gp1 1
1 0
sage: v = sum(GP22.basis())
sage: v.degree()
2
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(4))
>>> GP22 = SGA.garsia_procesi_module([Integer(2), Integer(2)])
>>> for b in GP22.basis():
...     print(b, b.degree())
gp2*gp3 2
gp1*gp3 2
gp3 1
gp2 1
gp1 1
1 0
>>> v = sum(GP22.basis())
>>> v.degree()
2
```

homogeneous_degree()

Return the (homogeneous) degree of `self` if homogeneous otherwise raise an error.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 4)
sage: GP31 = SGA.garsia_procesi_module([3, 1])
sage: for b in GP31.basis():
....:     print(b, b.homogeneous_degree())
gp3 1
gp2 1
gp1 1
1 0
sage: v = sum(GP31.basis()); v
gp1 + gp2 + gp3 + 1
sage: v.homogeneous_degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(4))
>>> GP31 = SGA.garsia_procesi_module([Integer(3), Integer(1)])
>>> for b in GP31.basis():
...     print(b, b.homogeneous_degree())
gp3 1
gp2 1
gp1 1
1 0
>>> v = sum(GP31.basis()); v
gp1 + gp2 + gp3 + 1
>>> v.homogeneous_degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
```

monomial_coefficients(*copy=None*)

Return the monomial coefficients of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 4)
sage: GP31 = SGA.garsia_procesi_module([3, 1])
sage: v = GP31.an_element(); v
-gp1 - gp2 - gp3
sage: v.monomial_coefficients()
{0: 2, 1: 2, 2: 2, 3: 0}
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(4))
>>> GP31 = SGA.garsia_procesi_module([Integer(3), Integer(1)])
>>> v = GP31.an_element(); v
-gp1 - gp2 - gp3
>>> v.monomial_coefficients()
{0: 2, 1: 2, 2: 2, 3: 0}
```

to_vector(*order=None*)

Return `self` as a (dense) free module vector.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 4)
sage: GP22 = SGA.garsia_procesi_module([2, 2])
sage: v = GP22.an_element(); v
-gp1 - gp2 - gp3
sage: v.to_vector()
(0, 0, 2, 2, 2, 0)
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(4))
>>> GP22 = SGA.garsia_procesi_module([Integer(2), Integer(2)])
>>> v = GP22.an_element(); v
-gp1 - gp2 - gp3
```

(continues on next page)

(continued from previous page)

```
>>> v.to_vector()
(0, 0, 2, 2, 2, 0)
```

basis()

Return a basis of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 4)
sage: GP = SGA.garsia_procesi_module([2, 2])
sage: GP.basis()
Family (gp2*gp3, gp1*gp3, gp3, gp2, gp1, 1)
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(4))
>>> GP = SGA.garsia_procesi_module([Integer(2), Integer(2)])
>>> GP.basis()
Family (gp2*gp3, gp1*gp3, gp3, gp2, gp1, 1)
```

dimension()

Return the dimension of `self`.

The graded Frobenius character of the Garsia-Procesi module R_λ is given by the modified Hall-Littlewood polynomial $\widetilde{H}_{\lambda'}(x; q)$.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: Qp = Sym.hall_littlewood(1).Qp()
sage: for la in Partitions(5):
....:     print(SGA.garsia_procesi_module(la).dimension(),
....:           sum(c * StandardTableaux(la).cardinality()
....:               for la, c in s(Qp[la])))
1 1
5 5
10 10
20 20
30 30
60 60
120 120
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.s()
>>> Qp = Sym.hall_littlewood(Integer(1)).Qp()
>>> for la in Partitions(Integer(5)):
...     print(SGA.garsia_procesi_module(la).dimension(),
...           sum(c * StandardTableaux(la).cardinality()
...               for la, c in s(Qp[la])))
1 1
```

(continues on next page)

(continued from previous page)

```
5 5
10 10
20 20
30 30
60 60
120 120
```

get_order()

Return the order of the elements in the basis.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 4)
sage: GP = SGA.garsia_procesi_module([2, 2])
sage: GP.get_order()
(0, 1, 2, 3, 4, 5)
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(4))
>>> GP = SGA.garsia_procesi_module([Integer(2), Integer(2)])
>>> GP.get_order()
(0, 1, 2, 3, 4, 5)
```

graded_brauer_character()

Return the graded Brauer character of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 5)
sage: GP311 = SGA.garsia_procesi_module([3, 1, 1])
sage: GP311.graded_brauer_character()
(6*q^3 + 9*q^2 + 4*q + 1, q + 1, q^3 - q^2 - q + 1)
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(5))
>>> GP311 = SGA.garsia_procesi_module([Integer(3), Integer(1), Integer(1)])
>>> GP311.graded_brauer_character()
(6*q^3 + 9*q^2 + 4*q + 1, q + 1, q^3 - q^2 - q + 1)
```

graded_character()

Return the graded character of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: GP = SGA.garsia_procesi_module([2, 2, 1])
sage: gchi = GP.graded_character(); gchi
(5*q^4 + 11*q^3 + 9*q^2 + 4*q + 1, -q^4 + q^3 + 3*q^2 + 2*q + 1,
 q^4 - q^3 + q^2 + 1, -q^4 - q^3 + q + 1, -q^4 + q^3 - q + 1,
 q^4 - q^3 - q^2 + 1, q^3 - q^2 - q + 1)
sage: R.<q> = QQ[]
sage: gchi == sum(q^d * D.character()
```

(continues on next page)

(continued from previous page)

```
....:               for d, D in GP.graded_decomposition().items())
True
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> GP = SGA.garsia_procesi_module([Integer(2), Integer(2), Integer(1)])
>>> gchi = GP.graded_character(); gchi
(5*q^4 + 11*q^3 + 9*q^2 + 4*q + 1, -q^4 + q^3 + 3*q^2 + 2*q + 1,
 q^4 - q^3 + q^2 + 1, -q^4 - q^3 + q + 1, -q^4 + q^3 - q + 1,
 q^4 - q^3 - q^2 + 1, q^3 - q^2 - q + 1)
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> gchi == sum(q**d * D.character()
...             for d, D in GP.graded_decomposition().items())
True
```

graded_decomposition(*k=None*)

Return the decomposition of `self` as a direct sum of representations given by a fixed grading.

INPUT:

- `k` – (optional) integer; if given, return the k-th graded part

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 5)
sage: GP32 = SGA.garsia_procesi_module([3, 2])
sage: decomp = GP32.graded_decomposition(); decomp
{0: Subrepresentation with basis {0} of Garsia-Procesi ...,
 1: Subrepresentation with basis {0, 1, 2, 3} of Garsia-Procesi ...,
 2: Subrepresentation with basis {0, 1, 2, 3, 4} of Garsia-Procesi ...}
sage: decomp[2] is GP32.graded_decomposition(2)
True
sage: GP32.graded_decomposition(10)
Subrepresentation with basis {} of Garsia-Procesi module
 of shape [3, 2] over Finite Field of size 2
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(5))
>>> GP32 = SGA.garsia_procesi_module([Integer(3), Integer(2)])
>>> decomp = GP32.graded_decomposition(); decomp
{0: Subrepresentation with basis {0} of Garsia-Procesi ...,
 1: Subrepresentation with basis {0, 1, 2, 3} of Garsia-Procesi ...,
 2: Subrepresentation with basis {0, 1, 2, 3, 4} of Garsia-Procesi ...}
>>> decomp[Integer(2)] is GP32.graded_decomposition(Integer(2))
True
>>> GP32.graded_decomposition(Integer(10))
Subrepresentation with basis {} of Garsia-Procesi module
 of shape [3, 2] over Finite Field of size 2
```

graded_frobenius_image()

Return the graded Frobenius image of `self`.

The graded Frobenius image is the sum of the `frobenius_image()` of each graded component, which is known to result in the modified Hall-Littlewood polynomial $\widetilde{H}_\lambda(x; q)$.

EXAMPLES:

We verify that the result is the modified Hall-Littlewood polynomial for $n = 5$:

```
sage: R.<q> = QQ[]
sage: Sym = SymmetricFunctions(R)
sage: s = Sym.s()
sage: Qp = Sym.hall_littlewood(q).Qp()
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: for la in Partitions(5):
....:     f = SGA.garsia_procesi_module(la).graded_frobenius_image()
....:     d = f[la].degree()
....:     assert f.map_coefficients(lambda c: R(c(~q)*q^d)) == s(Qp[la])
```

```
>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> Sym = SymmetricFunctions(R)
>>> s = Sym.s()
>>> Qp = Sym.hall_littlewood(q).Qp()
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> for la in Partitions(Integer(5)):
...     f = SGA.garsia_procesi_module(la).graded_frobenius_image()
...     d = f[la].degree()
...     assert f.map_coefficients(lambda c: R(c(~q)*q**d)) == s(Qp[la])
```

graded_representation_matrix(*elt*, *q=None*)

Return the matrix corresponding to the left action of the symmetric group (algebra) element `elt` on `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(3), 3)
sage: GP = SGA.garsia_procesi_module([1, 1, 1])
sage: elt = SGA.an_element(); elt
[1, 2, 3] + 2*[1, 3, 2] + [3, 1, 2]
sage: X = GP.graded_representation_matrix(elt); X
[  0   0   0   0   0   0]
[  0 q^2   0   0   0   0]
[  0 q^2   0   0   0   0]
[  0   0   0   q   0   0]
[  0   0   0   q   0   0]
[  0   0   0   0   0   1]
sage: X.parent()
Full MatrixSpace of 6 by 6 dense matrices over
 Univariate Polynomial Ring in q over Finite Field of size 3
sage: R.<q> = GF(3)[]
sage: t = R.quotient([q^2+2*q+1]).gen()
sage: GP.graded_representation_matrix(elt, t)
[       0        0        0        0        0        0]
[       0 qbar + 2        0        0        0        0]
[       0 qbar + 2        0        0        0        0]
[       0        0        0     qbar        0        0]
[       0        0        0     qbar        0        0]
[       0        0        0        0        0        1]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(3))
>>> GP = SGA.garsia_procesi_module([Integer(1), Integer(1), Integer(1)])
>>> elt = SGA.an_element(); elt
[1, 2, 3] + 2*[1, 3, 2] + [3, 1, 2]
>>> X = GP.graded_representation_matrix(elt); X
[  0    0    0    0    0    0]
[  0 q^2    0    0    0    0]
[  0 q^2    0    0    0    0]
[  0    0    0    q    0    0]
[  0    0    0    q    0    0]
[  0    0    0    0    0    1]
>>> X.parent()
Full MatrixSpace of 6 by 6 dense matrices over
 Univariate Polynomial Ring in q over Finite Field of size 3
>>> R = GF(Integer(3))['q'];  (q,) = R._first_ngens(1)
>>> t = R.quotient([q**Integer(2)+Integer(2)*q+Integer(1)]).gen()
>>> GP.graded_representation_matrix(elt, t)
[       0        0        0        0        0        0]
[       0 qbar + 2        0        0        0        0]
[       0 qbar + 2        0        0        0        0]
[       0        0        0     qbar        0        0]
[       0        0        0     qbar        0        0]
[       0        0        0        0        0        1]
```

one_basis()

Return the index of the basis element 1.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 4)
sage: GP = SGA.garsia_procesi_module([2, 2])
sage: GP.one_basis()
5
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(4))
>>> GP = SGA.garsia_procesi_module([Integer(2), Integer(2)])
>>> GP.one_basis()
5
```

class sage.combinat.symmetric_group_representations.**SpechtRepresentation**(*parent*, *partition*)

Bases: `SymmetricGroupRepresentation_generic_class`

representation_matrix(*permutation*)

Return the matrix representing the `permutation` in this irreducible representation.

Note

This method caches the results.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3,1], 'specht')
sage: spc.representation_matrix(Permutation([2,1,3,4]))
[ 0 -1  0]
[-1  0  0]
[ 0  0  1]
sage: spc.representation_matrix(Permutation([3,2,1,4]))
[0 0 1]
[0 1 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> spc = SymmetricGroupRepresentation([Integer(3),Integer(1)], 'specht')
>>> spc.representation_matrix(Permutation([Integer(2),Integer(1),Integer(3),
↪Integer(4)]))
[ 0 -1  0]
[-1  0  0]
[ 0  0  1]
>>> spc.representation_matrix(Permutation([Integer(3),Integer(2),Integer(1),
↪Integer(4)]))
[0 0 1]
[0 1 0]
[1 0 0]
```

scalar_product (*u*, *v*)

Return `0` if `u+v` is not a permutation, and the signature of the permutation otherwise.

This is the scalar product of a vertex `u` of the underlying Yang-Baxter graph with the vertex `v` in the 'dual' Yang-Baxter graph.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3,2], 'specht')
sage: spc.scalar_product((1,0,2,1,0),(0,3,0,3,0))
-1
sage: spc.scalar_product((1,0,2,1,0),(3,0,0,3,0))
0
```

```
>>> from sage.all import *
>>> spc = SymmetricGroupRepresentation([Integer(3),Integer(2)], 'specht')
>>> spc.scalar_product((Integer(1),Integer(0),Integer(2),Integer(1),
↪Integer(0)),(Integer(0),Integer(3),Integer(0),Integer(3),Integer(0)))
-1
>>> spc.scalar_product((Integer(1),Integer(0),Integer(2),Integer(1),
↪Integer(0)),(Integer(3),Integer(0),Integer(0),Integer(3),Integer(0)))
0
```

scalar_product_matrix (*permutation=None*)

Return the scalar product matrix corresponding to `permutation`.

The entries are given by the scalar products of `u` and `permutation.action(v)`, where `u` is a vertex in the underlying Yang-Baxter graph and `v` is a vertex in the dual graph.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3,1], 'specht')
sage: spc.scalar_product_matrix()
[ 1  0  0]
[ 0 -1  0]
[ 0  0  1]
```

```
>>> from sage.all import *
>>> spc = SymmetricGroupRepresentation([Integer(3),Integer(1)], 'specht')
>>> spc.scalar_product_matrix()
[ 1  0  0]
[ 0 -1  0]
[ 0  0  1]
```

class sage.combinat.symmetric_group_representations.**SpechtRepresentations**(*n*, *ring=None*, *cache_matrices=True*)

Bases: *SymmetricGroupRepresentations_class*

Element

alias of *SpechtRepresentation*

sage.combinat.symmetric_group_representations.**SymmetricGroupRepresentation**(*partition*, *implementation='specht'*, *ring=None*, *cache_matrices=True*)

The irreducible representation of the symmetric group corresponding to `partition`.

INPUT:

- `partition` – a partition of a positive integer
- `implementation` – string (default: `'specht'`); one of:
  - `'seminormal'` – for Young's seminormal representation
  - `'orthogonal'` – for Young's orthogonal representation
  - `'specht'` – for Specht's representation
  - `'unitary'` – for the unitary representation
- `ring` – the ring over which the representation is defined
- `cache_matrices` – boolean (default: `True`); if `True`, then any representation matrices that are computed are cached

EXAMPLES:

Young's orthogonal representation: the matrices are orthogonal.

```
sage: orth = SymmetricGroupRepresentation([2,1], "orthogonal"); orth          #␣
↪needs sage.symbolic
Orthogonal representation of the symmetric group corresponding to [2, 1]
sage: all(a*a.transpose() == a.parent().identity_matrix() for a in orth)      #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> orth = SymmetricGroupRepresentation([Integer(2),Integer(1)], "orthogonal");
↪orth              # needs sage.symbolic
Orthogonal representation of the symmetric group corresponding to [2, 1]
>>> all(a*a.transpose() == a.parent().identity_matrix() for a in orth)        #
↪needs sage.symbolic
True
```

```
sage: # needs sage.symbolic
sage: orth = SymmetricGroupRepresentation([3,2], "orthogonal"); orth
Orthogonal representation of the symmetric group corresponding to [3, 2]
sage: orth([2,1,3,4,5])
[ 1  0  0  0  0]
[ 0  1  0  0  0]
[ 0  0 -1  0  0]
[ 0  0  0  1  0]
[ 0  0  0  0 -1]
sage: orth([1,3,2,4,5])
[           1            0            0            0            0]
[           0         -1/2 1/2*sqrt(3)            0            0]
[           0 1/2*sqrt(3)          1/2            0            0]
[           0            0            0         -1/2 1/2*sqrt(3)]
[           0            0            0 1/2*sqrt(3)          1/2]
sage: orth([1,2,4,3,5])
[       -1/3 2/3*sqrt(2)            0            0            0]
[2/3*sqrt(2)          1/3            0            0            0]
[           0            0            1            0            0]
[           0            0            0            1            0]
[           0            0            0            0           -1]
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> orth = SymmetricGroupRepresentation([Integer(3),Integer(2)], "orthogonal");
↪orth
Orthogonal representation of the symmetric group corresponding to [3, 2]
>>> orth([Integer(2),Integer(1),Integer(3),Integer(4),Integer(5)])
[ 1  0  0  0  0]
[ 0  1  0  0  0]
[ 0  0 -1  0  0]
[ 0  0  0  1  0]
[ 0  0  0  0 -1]
>>> orth([Integer(1),Integer(3),Integer(2),Integer(4),Integer(5)])
[           1            0            0            0            0]
[           0         -1/2 1/2*sqrt(3)            0            0]
[           0 1/2*sqrt(3)          1/2            0            0]
[           0            0            0         -1/2 1/2*sqrt(3)]
[           0            0            0 1/2*sqrt(3)          1/2]
>>> orth([Integer(1),Integer(2),Integer(4),Integer(3),Integer(5)])
[       -1/3 2/3*sqrt(2)            0            0            0]
[2/3*sqrt(2)          1/3            0            0            0]
[           0            0            1            0            0]
[           0            0            0            1            0]
```

(continues on next page)

(continued from previous page)

```
[          0           0           0           0          -1]
```

The Specht representation:

```
sage: spc = SymmetricGroupRepresentation([3,2], "specht")
sage: spc.scalar_product_matrix(Permutation([1,2,3,4,5]))
[ 1  0  0  0  0]
[ 0 -1  0  0  0]
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[-1  0  0  0 -1]
sage: spc.scalar_product_matrix(Permutation([5,4,3,2,1]))
[ 1 -1  0  1  0]
[ 0  0  1  0 -1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[-1  0  0  0 -1]
sage: spc([5,4,3,2,1])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
sage: spc.verify_representation()
True
```

```
>>> from sage.all import *
>>> spc = SymmetricGroupRepresentation([Integer(3),Integer(2)], "specht")
>>> spc.scalar_product_matrix(Permutation([Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]))
[ 1  0  0  0  0]
[ 0 -1  0  0  0]
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[-1  0  0  0 -1]
>>> spc.scalar_product_matrix(Permutation([Integer(5),Integer(4),Integer(3),
↪Integer(2),Integer(1)]))
[ 1 -1  0  1  0]
[ 0  0  1  0 -1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[-1  0  0  0 -1]
>>> spc([Integer(5),Integer(4),Integer(3),Integer(2),Integer(1)])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
>>> spc.verify_representation()
True
```

The unitary representation:

```
sage: unitary = SymmetricGroupRepresentation([3,1], "unitary"); unitary
Unitary representation of the symmetric group corresponding to [3, 1]
sage: unitary_GF49 = SymmetricGroupRepresentation([3,1], "unitary",␣
↪ring=GF(7**2)); unitary_GF49
Unitary representation of the symmetric group corresponding to [3, 1]
sage: unitary_GF49([2,1,3,4])
[6 0 0]
[0 1 0]
[0 0 1]
sage: unitary_GF49([3,2,1,4])
[       4 2*z2 + 3        0]
[5*z2 + 5        3        0]
[       0        0        1]
sage: unitary_GF49.verify_representation()
True
```

```
>>> from sage.all import *
>>> unitary = SymmetricGroupRepresentation([Integer(3),Integer(1)], "unitary");␣
↪unitary
Unitary representation of the symmetric group corresponding to [3, 1]
>>> unitary_GF49 = SymmetricGroupRepresentation([Integer(3),Integer(1)], "unitary
↪", ring=GF(Integer(7)**Integer(2))); unitary_GF49
Unitary representation of the symmetric group corresponding to [3, 1]
>>> unitary_GF49([Integer(2),Integer(1),Integer(3),Integer(4)])
[6 0 0]
[0 1 0]
[0 0 1]
>>> unitary_GF49([Integer(3),Integer(2),Integer(1),Integer(4)])
[       4 2*z2 + 3        0]
[5*z2 + 5        3        0]
[       0        0        1]
>>> unitary_GF49.verify_representation()
True
```

By default, any representation matrices that are computed are cached:

```
sage: spc = SymmetricGroupRepresentation([3,2], "specht")
sage: spc([5,4,3,2,1])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
sage: spc._cache__representation_matrix
{(([5, 4, 3, 2, 1],), ()): [ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]}
```

```
>>> from sage.all import *
>>> spc = SymmetricGroupRepresentation([Integer(3),Integer(2)], "specht")
```

(continues on next page)

(continued from previous page)

```
>>> spc([Integer(5),Integer(4),Integer(3),Integer(2),Integer(1)])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
>>> spc._cache__representation_matrix
{(([5, 4, 3, 2, 1],), ()): [ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]}
```

This can be turned off with the keyword `cache_matrices`:

```
sage: spc = SymmetricGroupRepresentation([3,2], "specht", cache_matrices=False)
sage: spc([5,4,3,2,1])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
sage: hasattr(spc, '_cache__representation_matrix')
False
```

```
>>> from sage.all import *
>>> spc = SymmetricGroupRepresentation([Integer(3),Integer(2)], "specht", cache_
↪matrices=False)
>>> spc([Integer(5),Integer(4),Integer(3),Integer(2),Integer(1)])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
>>> hasattr(spc, '_cache__representation_matrix')
False
```

Note

The implementation is based on the paper [Las].

REFERENCES:

AUTHORS:

- Franco Saliola (2009-04-23)

`class sage.combinat.symmetric_group_representations.SymmetricGroupRepresentation_generic_class`(*parent*, *partition*)

Bases: `Element`

Generic methods for a representation of the symmetric group.

to_character()

Return the character of the representation.

EXAMPLES:

The trivial character:

```
sage: rho = SymmetricGroupRepresentation([3])
sage: chi = rho.to_character(); chi
Character of Symmetric group of order 3! as a permutation group
sage: chi.values()
[1, 1, 1]
sage: all(chi(g) == 1 for g in SymmetricGroup(3))
True
```

```
>>> from sage.all import *
>>> rho = SymmetricGroupRepresentation([Integer(3)])
>>> chi = rho.to_character(); chi
Character of Symmetric group of order 3! as a permutation group
>>> chi.values()
[1, 1, 1]
>>> all(chi(g) == Integer(1) for g in SymmetricGroup(Integer(3)))
True
```

The sign character:

```
sage: rho = SymmetricGroupRepresentation([1,1,1])
sage: chi = rho.to_character(); chi
Character of Symmetric group of order 3! as a permutation group
sage: chi.values()
[1, -1, 1]
sage: all(chi(g) == g.sign() for g in SymmetricGroup(3))
True
```

```
>>> from sage.all import *
>>> rho = SymmetricGroupRepresentation([Integer(1),Integer(1),Integer(1)])
>>> chi = rho.to_character(); chi
Character of Symmetric group of order 3! as a permutation group
>>> chi.values()
[1, -1, 1]
>>> all(chi(g) == g.sign() for g in SymmetricGroup(Integer(3)))
True
```

The defining representation:

```
sage: triv = SymmetricGroupRepresentation([4])
sage: hook = SymmetricGroupRepresentation([3,1])
sage: def_rep = lambda p : triv(p).block_sum(hook(p)).trace()
sage: list(map(def_rep, Permutations(4)))
[4, 2, 2, 1, 1, 2, 2, 0, 1, 0, 0, 1, 1, 0, 2, 1, 0, 0, 0, 1, 1, 2, 0, 0]
```

(continues on next page)

(continued from previous page)

```
sage: [p.to_matrix().trace() for p in Permutations(4)]
[4, 2, 2, 1, 1, 2, 2, 0, 1, 0, 0, 1, 1, 0, 2, 1, 0, 0, 0, 1, 1, 2, 0, 0]
```

```
>>> from sage.all import *
>>> triv = SymmetricGroupRepresentation([Integer(4)])
>>> hook = SymmetricGroupRepresentation([Integer(3),Integer(1)])
>>> def_rep = lambda p : triv(p).block_sum(hook(p)).trace()
>>> list(map(def_rep, Permutations(Integer(4))))
[4, 2, 2, 1, 1, 2, 2, 0, 1, 0, 0, 1, 1, 0, 2, 1, 0, 0, 0, 1, 1, 2, 0, 0]
>>> [p.to_matrix().trace() for p in Permutations(Integer(4))]
[4, 2, 2, 1, 1, 2, 2, 0, 1, 0, 0, 1, 1, 0, 2, 1, 0, 0, 0, 1, 1, 2, 0, 0]
```

verify_representation()

Verify the representation.

This tests that the images of the simple transpositions are involutions and tests that the braid relations hold.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([1,1,1])
sage: spc.verify_representation()
True
sage: spc = SymmetricGroupRepresentation([4,2,1])
sage: spc.verify_representation()
True
```

```
>>> from sage.all import *
>>> spc = SymmetricGroupRepresentation([Integer(1),Integer(1),Integer(1)])
>>> spc.verify_representation()
True
>>> spc = SymmetricGroupRepresentation([Integer(4),Integer(2),Integer(1)])
>>> spc.verify_representation()
True
```

sage.combinat.symmetric_group_representations.**SymmetricGroupRepresentations**(*n*, *implementation='specht'*, *ring=None*, *cache_matrices=True*)

Irreducible representations of the symmetric group.

INPUT:

- `n` – positive integer
- `implementation` – string (default: `'specht'`); one of:
  - `'seminormal'` – for Young's seminormal representation
  - `'orthogonal'` – for Young's orthogonal representation
  - `'specht'` – for Specht's representation
  - `'unitary'` – for the unitary representation
- `ring` – the ring over which the representation is defined

- `cache_matrices` – boolean (default: `True`); if `True`, then any representation matrices that are computed are cached

EXAMPLES:

Young's orthogonal representation: the matrices are orthogonal.

```
sage: orth = SymmetricGroupRepresentations(3, "orthogonal"); orth              #␣
↪needs sage.symbolic
Orthogonal representations of the symmetric group of order 3! over Symbolic Ring
sage: orth.list()                                                              #␣
↪needs sage.symbolic
[Orthogonal representation of the symmetric group corresponding to [3],
 Orthogonal representation of the symmetric group corresponding to [2, 1],
 Orthogonal representation of the symmetric group corresponding to [1, 1, 1]]
sage: orth([2,1])([1,2,3])                                                     #␣
↪needs sage.symbolic
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> orth = SymmetricGroupRepresentations(Integer(3), "orthogonal"); orth       ␣
↪     # needs sage.symbolic
Orthogonal representations of the symmetric group of order 3! over Symbolic Ring
>>> orth.list()                                                              #␣
↪needs sage.symbolic
[Orthogonal representation of the symmetric group corresponding to [3],
 Orthogonal representation of the symmetric group corresponding to [2, 1],
 Orthogonal representation of the symmetric group corresponding to [1, 1, 1]]
>>> orth([Integer(2),Integer(1)])([Integer(1),Integer(2),Integer(3)])          ␣
↪                                          # needs sage.symbolic
[1 0]
[0 1]
```

Young's seminormal representation.

```
sage: snorm = SymmetricGroupRepresentations(3, "seminormal"); snorm
Seminormal representations of the symmetric group of order 3! over Rational Field
sage: sgn = snorm([1,1,1]); sgn
Seminormal representation of the symmetric group corresponding to [1, 1, 1]
sage: list(map(sgn, Permutations(3)))
[[1], [-1], [-1], [1], [1], [-1]]
```

```
>>> from sage.all import *
>>> snorm = SymmetricGroupRepresentations(Integer(3), "seminormal"); snorm
Seminormal representations of the symmetric group of order 3! over Rational Field
>>> sgn = snorm([Integer(1),Integer(1),Integer(1)]); sgn
Seminormal representation of the symmetric group corresponding to [1, 1, 1]
>>> list(map(sgn, Permutations(Integer(3))))
[[1], [-1], [-1], [1], [1], [-1]]
```

The Specht Representation.

```
sage: spc = SymmetricGroupRepresentations(5, "specht"); spc
Specht representations of the symmetric group of order 5! over Integer Ring
sage: spc([3,2])([5,4,3,2,1])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
```

```
>>> from sage.all import *
>>> spc = SymmetricGroupRepresentations(Integer(5), "specht"); spc
Specht representations of the symmetric group of order 5! over Integer Ring
>>> spc([Integer(3),Integer(2)])([Integer(5),Integer(4),Integer(3),Integer(2),
↪Integer(1)])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
```

The unitary representation.

```
sage: unitary = SymmetricGroupRepresentations(3, "unitary"); unitary
Unitary representations of the symmetric group of order 3! over Symbolic Ring
sage: unitary_GF49 = SymmetricGroupRepresentations(4, "unitary", ring=GF(7**2));␣
↪unitary_GF49
Unitary representations of the symmetric group of order 4! over Finite Field in␣
↪z2 of size 7^2
sage: unitary_GF49([3,1])([2,1,3,4])
[6 0 0]
[0 1 0]
[0 0 1]
sage: unitary_GF49([3,1])([3,2,1,4])
[       4 2*z2 + 3        0]
[5*z2 + 5        3        0]
[       0        0        1]
```

```
>>> from sage.all import *
>>> unitary = SymmetricGroupRepresentations(Integer(3), "unitary"); unitary
Unitary representations of the symmetric group of order 3! over Symbolic Ring
>>> unitary_GF49 = SymmetricGroupRepresentations(Integer(4), "unitary",␣
↪ring=GF(Integer(7)**Integer(2))); unitary_GF49
Unitary representations of the symmetric group of order 4! over Finite Field in␣
↪z2 of size 7^2
>>> unitary_GF49([Integer(3),Integer(1)])([Integer(2),Integer(1),Integer(3),
↪Integer(4)])
[6 0 0]
[0 1 0]
[0 0 1]
>>> unitary_GF49([Integer(3),Integer(1)])([Integer(3),Integer(2),Integer(1),
↪Integer(4)])
[       4 2*z2 + 3        0]
```

(continues on next page)

(continued from previous page)

```
[5*z2 + 5         3         0]
[        0        0         1]
```

Note: The implementation is based on the paper [Las].

AUTHORS:

- Franco Saliola (2009-04-23)

class sage.combinat.symmetric_group_representations.**SymmetricGroupRepresentations_class**(*n*, *ring=None*, *cache_matrices=True*)

Bases: UniqueRepresentation, Parent

Generic methods for the CombinatorialClass of irreducible representations of the symmetric group.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: sp = SymmetricGroupRepresentations(4, "specht")
sage: sp.cardinality()
5
```

```
>>> from sage.all import *
>>> sp = SymmetricGroupRepresentations(Integer(4), "specht")
>>> sp.cardinality()
5
```

class sage.combinat.symmetric_group_representations.**UnitaryRepresentation**(*parent*, *partition*)

Bases: *SymmetricGroupRepresentation_generic_class*

A unitary representation of the symmetric group.

In characteristic zero, this is the same as the orthogonal representation since they are defined over the reals. In positive characteristic, we are able to construct representations when the field is finite, has square order, and the characteristic does not divide the order of the group. In this case, we construct the representation by first constructing the orthogonal representation ρ and then taking the extended Cholesky decomposition of the unique solution U to the equation $\rho(g)^T U \rho(g) = U$ for all g in G.

representation_matrix(*permutation*)

Return the matrix representing the `permutation` in this irreducible representation.

Note: This method caches the results.

EXAMPLES:

```
sage: unitary_specht = SymmetricGroupRepresentation([3,1], 'unitary',␣
↪ring=GF(7**2))
sage: unitary_specht.representation_matrix(Permutation([2,1,3,4]))
[6 0 0]
[0 1 0]
[0 0 1]
sage: unitary_specht.representation_matrix(Permutation([3,2,1,4]))
[       4 2*z2 + 3        0]
[5*z2 + 5        3        0]
[       0        0        1]
sage: all(A * A.H == 1 for A in [unitary_specht.representation_matrix(g)
....:                            for g in Permutations(4)])
True
```

```
>>> from sage.all import *
>>> unitary_specht = SymmetricGroupRepresentation([Integer(3),Integer(1)],
↪'unitary', ring=GF(Integer(7)**Integer(2)))
>>> unitary_specht.representation_matrix(Permutation([Integer(2),Integer(1),
↪Integer(3),Integer(4)]))
[6 0 0]
[0 1 0]
[0 0 1]
>>> unitary_specht.representation_matrix(Permutation([Integer(3),Integer(2),
↪Integer(1),Integer(4)]))
[       4 2*z2 + 3        0]
[5*z2 + 5        3        0]
[       0        0        1]
>>> all(A * A.H == Integer(1) for A in [unitary_specht.representation_
↪matrix(g)
...                            for g in Permutations(Integer(4))])
True
```

class sage.combinat.symmetric_group_representations.**UnitaryRepresentations**(*n*, *ring=None*, *cache_matrices=True*)

Bases: *SymmetricGroupRepresentations_class*

Element

alias of *UnitaryRepresentation*

class sage.combinat.symmetric_group_representations.**YoungRepresentation_Orthogonal**(*parent*, *partition*)

Bases: *YoungRepresentation_generic*

class sage.combinat.symmetric_group_representations.**YoungRepresentation_Seminormal**(*parent*, *partition*)

Bases: *YoungRepresentation_generic*

class sage.combinat.symmetric_group_representations.**YoungRepresentation_generic**(*parent*, *partition*)

Bases: *SymmetricGroupRepresentation_generic_class*

Generic methods for Young's representations of the symmetric group.

representation_matrix(*permutation*)

Return the matrix representing permutation.

EXAMPLES:

```
sage: orth = SymmetricGroupRepresentation([2,1], "orthogonal")              #␣
↪needs sage.symbolic
sage: orth.representation_matrix(Permutation([2,1,3]))                      #␣
↪needs sage.symbolic
[ 1  0]
[ 0 -1]
sage: orth.representation_matrix(Permutation([1,3,2]))                      #␣
↪needs sage.symbolic
[       -1/2 1/2*sqrt(3)]
[1/2*sqrt(3)         1/2]
```

```
>>> from sage.all import *
>>> orth = SymmetricGroupRepresentation([Integer(2),Integer(1)], "orthogonal
↪")              # needs sage.symbolic
>>> orth.representation_matrix(Permutation([Integer(2),Integer(1),
↪Integer(3)]))                    # needs sage.symbolic
[ 1  0]
[ 0 -1]
>>> orth.representation_matrix(Permutation([Integer(1),Integer(3),
↪Integer(2)]))                    # needs sage.symbolic
[       -1/2 1/2*sqrt(3)]
[1/2*sqrt(3)         1/2]
```

```
sage: norm = SymmetricGroupRepresentation([2,1], "seminormal")
sage: p = PermutationGroupElement([2,1,3])
sage: norm.representation_matrix(p)
[ 1  0]
[ 0 -1]
sage: p = PermutationGroupElement([1,3,2])
sage: norm.representation_matrix(p)
[-1/2  3/2]
[ 1/2  1/2]
```

```
>>> from sage.all import *
>>> norm = SymmetricGroupRepresentation([Integer(2),Integer(1)], "seminormal")
>>> p = PermutationGroupElement([Integer(2),Integer(1),Integer(3)])
>>> norm.representation_matrix(p)
[ 1  0]
[ 0 -1]
>>> p = PermutationGroupElement([Integer(1),Integer(3),Integer(2)])
>>> norm.representation_matrix(p)
[-1/2  3/2]
[ 1/2  1/2]
```

representation_matrix_for_simple_transposition(*i*)

Return the matrix representing the transposition that swaps `i` and `i+1`.

EXAMPLES:

```
sage: orth = SymmetricGroupRepresentation([2,1], "orthogonal")              #␣
↪needs sage.symbolic
sage: orth.representation_matrix_for_simple_transposition(1)                #␣
↪needs sage.symbolic
[ 1  0]
[ 0 -1]
sage: orth.representation_matrix_for_simple_transposition(2)                #␣
↪needs sage.symbolic
[       -1/2 1/2*sqrt(3)]
[1/2*sqrt(3)         1/2]

sage: norm = SymmetricGroupRepresentation([2,1], "seminormal")
sage: norm.representation_matrix_for_simple_transposition(1)
[ 1  0]
[ 0 -1]
sage: norm.representation_matrix_for_simple_transposition(2)
[-1/2  3/2]
[ 1/2  1/2]
```

```
>>> from sage.all import *
>>> orth = SymmetricGroupRepresentation([Integer(2),Integer(1)], "orthogonal
↪")                # needs sage.symbolic
>>> orth.representation_matrix_for_simple_transposition(Integer(1))        ␣
↪      # needs sage.symbolic
[ 1  0]
[ 0 -1]
>>> orth.representation_matrix_for_simple_transposition(Integer(2))        ␣
↪      # needs sage.symbolic
[       -1/2 1/2*sqrt(3)]
[1/2*sqrt(3)         1/2]

>>> norm = SymmetricGroupRepresentation([Integer(2),Integer(1)], "seminormal")
>>> norm.representation_matrix_for_simple_transposition(Integer(1))
[ 1  0]
[ 0 -1]
>>> norm.representation_matrix_for_simple_transposition(Integer(2))
[-1/2  3/2]
[ 1/2  1/2]
```

class sage.combinat.symmetric_group_representations.**YoungRepresentations_Orthogonal**(*n*, *ring=None*, *cache_matrices=True*)

Bases: *SymmetricGroupRepresentations_class*

Element

alias of *YoungRepresentation_Orthogonal*

class sage.combinat.symmetric_group_representations.**YoungRepresentations_Seminormal**(*n*, *ring=None*, *cache_matrices=True*)

Bases: *SymmetricGroupRepresentations_class*

Element

alias of *YoungRepresentation_Seminormal*

sage.combinat.symmetric_group_representations.**partition_to_vector_of_contents**(*partition*, *reverse=False*)

Return the "vector of contents" associated to partition.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_representations import partition_to_
↪vector_of_contents
sage: partition_to_vector_of_contents([3,2])
(0, 1, 2, -1, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.symmetric_group_representations import partition_to_vector_
↪of_contents
>>> partition_to_vector_of_contents([Integer(3),Integer(2)])
(0, 1, 2, -1, 0)
```

5.1.344 T-sequences

T-sequences are tuples of four (-1, 0, 1) sequences of length t where for every i exactly one sequence has a nonzero entry at index i and for which the nonperiodic autocorrelation function is equal to zero (i.e. they are complementary). See Definition 7.5 of [Seb2017].

These can be constructed from Turyn sequences. In particular, if Turyn sequences of length l exists, there will be T-sequences of length $4l - 1$ and $2l - 1$.

Turyn sequences are tuples of four (-1, +1) sequences X, U, Y, V of length $l, l, l-1, l-1$ with nonperiodic autocorrelation equal to zero and the additional constraints that:

- the first element of X is 1
- the last element of X is -1
- the last element of U is 1

The nonperiodic autocorrelation of a familiy of sequences $X = \{A_1, A_2, ..., A_n\}$ is defined as (see Definition 7.2 of [Seb2017]):

$$N_X(j) = \sum_{i=1}^{n-j}(a_{1,i}a_{1,i+j} + a_{2,i}a_{2,i+j} + ... + a_{n,i}a_{n,i+j})$$

AUTHORS:

- Matteo Cati (2022-11-16): initial version

sage.combinat.t_sequences.**T_sequences_construction_from_base_sequences**(*base_sequences*, *check=True*)

Construct T-sequences of length $2n + p$ from base sequences of length $n + p, n + p, n, n$.

Given base sequences A, B, C, D, the T-sequences are constructed as described in [KTR2005]:

$$T_1 = \frac{1}{2}(A + B); 0_n$$
$$T_2 = \frac{1}{2}(A - B); 0_n$$
$$T_3 = 0_{n+p} + \frac{1}{2}(C + D)$$
$$T_4 = 0_{n+p} + \frac{1}{2}(C - D)$$

INPUT:

- `base_sequences` – the base sequences that should be used to construct the T-sequences
- `check` – boolean (default: `True`); check that the sequences created are T-sequences before returning them

EXAMPLES:

```
sage: from sage.combinat.t_sequences import turyn_sequences_smallcases, T_
↪sequences_construction_from_base_sequences
sage: seqs = turyn_sequences_smallcases(4)
sage: T_sequences_construction_from_base_sequences(seqs)
[[1, 1, -1, 0, 0, 0, 0],
[0, 0, 0, -1, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 1],
[0, 0, 0, 0, 0, 1, 0]]
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import turyn_sequences_smallcases, T_sequences_
↪construction_from_base_sequences
>>> seqs = turyn_sequences_smallcases(Integer(4))
>>> T_sequences_construction_from_base_sequences(seqs)
[[1, 1, -1, 0, 0, 0, 0],
[0, 0, 0, -1, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 1],
[0, 0, 0, 0, 0, 1, 0]]
```

sage.combinat.t_sequences.**T_sequences_construction_from_turyn_sequences**(*turyn_sequences*, *check=True*)

Construct T-sequences of length $4l - 1$ from Turyn sequences of length l.

Given Turyn sequences X, U, Y, V, the T-sequences are constructed as described in theorem 7.7 of [Seb2017]:

$$T_1 = 1; 0_{4l-2}$$
$$T_2 = 0; X/Y; 0_{2l-1}$$
$$T_3 = 0_{2l}; U/0_{l-2}$$
$$T_4 = 0_{2l} + 0_l/V$$

INPUT:

- `turyn_sequences` – the Turyn sequences that should be used to construct the T-sequences
- `check` – boolean (default: `True`); check that the sequences created are T-sequences before returning them

EXAMPLES:

```
sage: from sage.combinat.t_sequences import turyn_sequences_smallcases, T_
↪sequences_construction_from_turyn_sequences, is_T_sequences_set
sage: seqs = turyn_sequences_smallcases(4)
sage: T_sequences_construction_from_turyn_sequences(seqs)
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 1, 1, 1, -1, 1, -1, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, -1, 0, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1, 0, 1, 0]]
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import turyn_sequences_smallcases, T_sequences_
↪construction_from_turyn_sequences, is_T_sequences_set
>>> seqs = turyn_sequences_smallcases(Integer(4))
>>> T_sequences_construction_from_turyn_sequences(seqs)
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 1, 1, 1, -1, 1, -1, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, -1, 0, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1, 0, 1, 0]]
```

sage.combinat.t_sequences.**T_sequences_smallcases**(*t*, *existence=False*, *check=True*)

Construct T-sequences for some small values of t.

This method will try to use the constructions defined in *T_sequences_construction_from_base_sequences()* and *T_sequences_construction_from_turyn_sequences()* together with the Turyn sequences stored in *turyn_sequences_smallcases()*, or base sequences created by *base_sequences_smallcases()*.

This function contains also some T-sequences taken directly from [CRSKKY1989].

INPUT:

- `t` – integer; the length of the T-sequences to construct
- `existence` – boolean (default: `False`); if `True`, this method only returns whether a T-sequences of the given size can be constructed
- `check` – boolean (default: `True`); check that the sequences are T-sequences before returning them

EXAMPLES:

By default, this method returns the four T-sequences

```
sage: from sage.combinat.t_sequences import T_sequences_smallcases, is_T_
↪sequences_set
sage: T_sequences_smallcases(9)
[[1, 1, 0, 1, 0, 0, 0, 0, 0],
[0, 0, -1, 0, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 1, 0, 0, -1],
[0, 0, 0, 0, 0, 0, 1, -1, 0]]
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import T_sequences_smallcases, is_T_sequences_
↪set
>>> T_sequences_smallcases(Integer(9))
[[1, 1, 0, 1, 0, 0, 0, 0, 0],
[0, 0, -1, 0, 1, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0, 0, 0, 1, 0, 0, -1],
[0, 0, 0, 0, 0, 0, 1, -1, 0]]
```

If the existence flag is passed, the method returns a boolean

```
sage: T_sequences_smallcases(9, existence=True)
True
```

```
>>> from sage.all import *
>>> T_sequences_smallcases(Integer(9), existence=True)
True
```

sage.combinat.t_sequences.**base_sequences_construction**(*turyn_type_seqs*, *check=True*)

Construct base sequences of length $2n-1, 2n-1, n, n$ from Turyn type sequences of length $n, n, n, n-1$.

Given Turyn type sequences X, Y, Z, W of length $n, n, n, n-1$, Theorem 1 of [KTR2005] shows that the following are base sequences of length $2n-1, 2n-1, n, n$:

$$\begin{aligned}A &= Z; W \\ B &= Z; -W \\ C &= X \\ D &= Y\end{aligned}$$

INPUT:

- `turyn_type_seqs` – the list of 4 Turyn type sequences that should be used to construct the base sequences
- `check` – boolean (default: `True`); check that the resulting sequences are base sequences before returning them

OUTPUT: list containing the four base sequences

EXAMPLES:

```
sage: from sage.combinat.t_sequences import base_sequences_construction
sage: X = [1,1,-1,1,-1,1,-1,1]
sage: Y = [1,-1,-1,-1,-1,-1,-1,1]
sage: Z = [1,-1,-1,1,1,1,1,-1]
sage: W = [1,1,1,-1,1,1,-1]
sage: base_sequences_construction([X, Y, Z, W])
[[1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1],
[1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, -1, -1, 1],
[1, 1, -1, 1, -1, 1, -1, 1],
[1, -1, -1, -1, -1, -1, -1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import base_sequences_construction
>>> X = [Integer(1),Integer(1),-Integer(1),Integer(1),-Integer(1),Integer(1),-
↪Integer(1),Integer(1)]
>>> Y = [Integer(1),-Integer(1),-Integer(1),-Integer(1),-Integer(1),-Integer(1),-
↪Integer(1),Integer(1)]
>>> Z = [Integer(1),-Integer(1),-Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),-Integer(1)]
>>> W = [Integer(1),Integer(1),Integer(1),-Integer(1),Integer(1),Integer(1),-
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)]
>>> base_sequences_construction([X, Y, Z, W])
[[1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1],
[1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, -1, -1, 1],
[1, 1, -1, 1, -1, 1, -1, 1],
[1, -1, -1, -1, -1, -1, -1, 1]]
```

See also

is_base_sequences_tuple()

sage.combinat.t_sequences.**base_sequences_smallcases**(*n*, *p*, *existence=False*, *check=True*)

Construct base sequences of length $n+p, n+p, n, n$ from available data.

The function uses the construction *base_sequences_construction()*, together with Turyn type sequences from *turyn_type_sequences_smallcases()* to construct base sequences with $p = n - 1$.

Furthermore, this function uses also Turyn sequences (i.e. base sequences with $p = 1$) from *turyn_sequences_smallcases()*.

INPUT:

- n – integer; the length of the last two base sequences
- p – integer; $n + p$ will be the length of the first two base sequences
- existence – boolean (default: False); if True, the function will only check whether the base sequences can be constructed
- check – boolean (default: True); check that the resulting sequences are base sequences before returning them

OUTPUT:

If existence is False, the function returns a list containing the four base sequences, or raises an error if the base sequences cannot be constructed. If existence is True, the function returns a boolean, which is True if the base sequences can be constructed and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.t_sequences import base_sequences_smallcases
sage: base_sequences_smallcases(8, 7)
[[1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1],
[1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, -1, -1, 1],
[1, 1, -1, 1, -1, 1, -1, 1],
[1, -1, -1, -1, -1, -1, -1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import base_sequences_smallcases
>>> base_sequences_smallcases(Integer(8), Integer(7))
[[1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1],
[1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, -1, -1, 1],
[1, 1, -1, 1, -1, 1, -1, 1],
[1, -1, -1, -1, -1, -1, -1, 1]]
```

If existence is True, the function returns a boolean

```
sage: base_sequences_smallcases(8, 7, existence=True)
True
sage: base_sequences_smallcases(7, 5, existence=True)
False
```

```
>>> from sage.all import *
>>> base_sequences_smallcases(Integer(8), Integer(7), existence=True)
True
>>> base_sequences_smallcases(Integer(7), Integer(5), existence=True)
False
```

sage.combinat.t_sequences.**is_T_sequences_set**(*sequences*, *verbose=False*)

Check if a family of sequences is composed of T-sequences.

Given 4 (-1, 0, +1) sequences, they will be T-sequences if (Definition 7.4 of [Seb2017]):

- they have all the same length t
- for each index i, exactly one sequence is nonzero at i
- the nonperiodic autocorrelation is equal to 0

INPUT:

- `sequences` – list of four sequences
- `verbose` – boolean (default: `False`); if `True` the function will be verbose when the sequences do not satisfy the constraints

EXAMPLES:

```
sage: from sage.combinat.t_sequences import is_T_sequences_set
sage: seqs = [[1, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, -1], [0, 0, 0, 0, 0]]
sage: is_T_sequences_set(seqs)
True
sage: seqs = [[1, 1, 0, 1, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, -1], [0, 0, 0, 0, 0]]
sage: is_T_sequences_set(seqs, verbose=True)
There should be exactly a nonzero element at every index, found 2 such
 elements at index 3
False
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import is_T_sequences_set
>>> seqs = [[Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)],
↪[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0)], [Integer(0),
↪Integer(0), Integer(0), Integer(1), -Integer(1)], [Integer(0), Integer(0),
↪Integer(0), Integer(0), Integer(0)]]
>>> is_T_sequences_set(seqs)
True
>>> seqs = [[Integer(1), Integer(1), Integer(0), Integer(1), Integer(0)],
↪[Integer(0), Integer(0), Integer(1), Integer(0), Integer(0)], [Integer(0),
↪Integer(0), Integer(0), Integer(1), -Integer(1)], [Integer(0), Integer(0),
↪Integer(0), Integer(0), Integer(0)]]
>>> is_T_sequences_set(seqs, verbose=True)
There should be exactly a nonzero element at every index, found 2 such
```

(continues on next page)

(continued from previous page)

```
 elements at index 3
False
```

sage.combinat.t_sequences.**is_base_sequences_tuple**(*base_sequences*, *verbose=False*)

Check if the given sequences are base sequences.

Four (-1, +1) sequences A, B, C, D of length $n + p, n + p, n, n$ are called base sequences if for all $j \geq 1$:

$$N_A(j) + N_B(j) + N_C(j) + N_D(j) = 0$$

where $N_X(j)$ is the nonperiodic autocorrelation (See definition in [KTR2005]).

INPUT:

- `base_sequences` – the list of 4 sequences that should be checked
- `verbose` – boolean (default: `False`); if `True` the function will be verbose when the sequences do not satisfy the constraints

EXAMPLES:

```
sage: from sage.combinat.t_sequences import is_base_sequences_tuple
sage: seqs = [[1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1],[1, -1, -1, 1, 1,
↪ 1, 1, -1, -1, -1, -1, 1, -1, -1, 1],[1, 1, -1, 1, -1, 1, -1, 1],[1, -1, -1, -1,
↪ -1, -1, -1, 1]]
sage: is_base_sequences_tuple(seqs)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import is_base_sequences_tuple
>>> seqs = [[Integer(1), -Integer(1), -Integer(1), Integer(1), Integer(1),␣
↪Integer(1), Integer(1), -Integer(1), Integer(1), Integer(1), Integer(1), -
↪Integer(1), Integer(1), Integer(1), -Integer(1)],[Integer(1), -Integer(1), -
↪Integer(1), Integer(1), Integer(1), Integer(1), Integer(1), -Integer(1), -
↪Integer(1), -Integer(1), -Integer(1), Integer(1), -Integer(1), -Integer(1),␣
↪Integer(1)],[Integer(1), Integer(1), -Integer(1), Integer(1), -Integer(1),␣
↪Integer(1), -Integer(1), Integer(1)],[Integer(1), -Integer(1), -Integer(1), -
↪Integer(1), -Integer(1), -Integer(1), -Integer(1), Integer(1)]]
>>> is_base_sequences_tuple(seqs)
True
```

If verbose is true, the function will be verbose

```
sage: seqs = [[1, -1], [1, 1], [-1], [2]]
sage: is_base_sequences_tuple(seqs, verbose=True)
Base sequences should only contain -1, +1, found 2
False
```

```
>>> from sage.all import *
>>> seqs = [[Integer(1), -Integer(1)], [Integer(1), Integer(1)], [-Integer(1)],␣
↪[Integer(2)]]
>>> is_base_sequences_tuple(seqs, verbose=True)
Base sequences should only contain -1, +1, found 2
False
```

See also

base_sequences_construction()

sage.combinat.t_sequences.**is_skew**(*seq*, *verbose=False*)

Check if the given sequence is skew.

A sequence $X = \{x_1, x_2, ..., x_n\}$ is defined skew (according to Definition 7.4 of [Seb2017]) if n is even and $x_i = -x_{n-i+1}$.

INPUT:

- `seq` – the sequence that should be checked
- `verbose` – boolean (default: `False`); if `True` the function will be verbose when the sequences do not satisfy the constraints

EXAMPLES:

```
sage: from sage.combinat.t_sequences import is_skew
sage: is_skew([1, -1, 1, -1, 1, -1])
True
sage: is_skew([1, -1, -1, -1], verbose=True)
Constraint not satisfied at index 1
False
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import is_skew
>>> is_skew([Integer(1), -Integer(1), Integer(1), -Integer(1), Integer(1), -
↪Integer(1)])
True
>>> is_skew([Integer(1), -Integer(1), -Integer(1), -Integer(1)], verbose=True)
Constraint not satisfied at index 1
False
```

sage.combinat.t_sequences.**is_symmetric**(*seq*, *verbose=False*)

Check if the given sequence is symmetric.

A sequence $X = \{x_1, x_2, ..., x_n\}$ is defined symmetric (according to Definition 7.4 of [Seb2017]) if n is odd and $x_i = x_{n-i+1}$.

INPUT:

- `seq` – the sequence that should be checked
- `verbose` – boolean (default: `False`); if `True` the function will be verbose when the sequences do not satisfy the constraints

EXAMPLES:

```
sage: from sage.combinat.t_sequences import is_symmetric
sage: is_symmetric([1, -1, 1, -1, 1])
True
sage: is_symmetric([1, -1, 1, 1, 1], verbose=True)
Constraint not satisfied at index 1
False
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import is_symmetric
>>> is_symmetric([Integer(1), -Integer(1), Integer(1), -Integer(1), Integer(1)])
True
>>> is_symmetric([Integer(1), -Integer(1), Integer(1), Integer(1), Integer(1)],␣
↪verbose=True)
Constraint not satisfied at index 1
False
```

sage.combinat.t_sequences.**turyn_sequences_smallcases**(*l*, *existence=False*)

Construction of Turyn sequences for small values of l.

The data is taken from [Seb2017] and [CRSKKY1989].

INPUT:

- `l` – integer; the length of the Turyn sequences
- `existence` – boolean (default: `False`); if `True`, only return whether the Turyn sequences are available for the given length

EXAMPLES:

By default, this method returns the four Turyn sequences

```
sage: from sage.combinat.t_sequences import turyn_sequences_smallcases
sage: turyn_sequences_smallcases(4)
[[1, 1, -1, -1], [1, 1, -1, 1], [1, 1, 1], [1, -1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import turyn_sequences_smallcases
>>> turyn_sequences_smallcases(Integer(4))
[[1, 1, -1, -1], [1, 1, -1, 1], [1, 1, 1], [1, -1, 1]]
```

If we pass the `existence` flag, the method will return a boolean

```
sage: turyn_sequences_smallcases(4, existence=True)
True
```

```
>>> from sage.all import *
>>> turyn_sequences_smallcases(Integer(4), existence=True)
True
```

sage.combinat.t_sequences.**turyn_type_sequences_smallcases**(*n*, *existence=False*)

Construction of Turyn type sequences for small values of n.

The data is taken from [KTR2005] for $n = 36$, and from [BDKR2013] for $n \leq 32$.

INPUT:

- `n` – integer; the length of the Turyn type sequences
- `existence` – boolean (default: `False`); if `True`, only return whether the Turyn type sequences are available for the given length

EXAMPLES:

By default, this method returns the four Turyn type sequences

```
sage: from sage.combinat.t_sequences import turyn_type_sequences_smallcases
sage: turyn_type_sequences_smallcases(4)
[[1, 1, 1, 1], [1, 1, -1, 1], [1, 1, -1, -1], [1, -1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.t_sequences import turyn_type_sequences_smallcases
>>> turyn_type_sequences_smallcases(Integer(4))
[[1, 1, 1, 1], [1, 1, -1, 1], [1, 1, -1, -1], [1, -1, 1]]
```

If we pass the `existence` flag, the method will return a boolean

```
sage: turyn_type_sequences_smallcases(4, existence=True)
True
```

```
>>> from sage.all import *
>>> turyn_type_sequences_smallcases(Integer(4), existence=True)
True
```

ALGORITHM:

The Turyn type sequences are stored in hexadecimal format. Given n hexadecimal digits $h_1, h_2, ..., h_n$, it is possible to get the Turyn type sequences by converting each h_i $(1 \leq i \leq n-1)$ into a four digits binary number. Then, the j-th binary digit is 0 if the i-th number in the j-th sequence is 1, and it is 1 if the number in the sequence is -1.

For the n-th digit, it should be converted to a 3 digits binary number, and then the same mapping as before can be used (see also [BDKR2013]).

5.1.345 Tableaux

AUTHORS:

- Mike Hansen (2007): initial version
- Jason Bandlow (2011): updated to use Parent/Element model, and many minor fixes
- Andrew Mathas (2012-13): completed the transition to the parent/element model begun by Jason Bandlow
- Travis Scrimshaw (11-22-2012): Added tuple options, changed `*katabolism*` to `*catabolism*`. Cleaned up documentation.
- Andrew Mathas (2016-08-11): Row standard tableaux added
- Oliver Pechenik (2018): Added increasing tableaux.

This file consists of the following major classes:

Element classes:

- *Tableau*
- *SemistandardTableau*
- *StandardTableau*
- *RowStandardTableau*
- *IncreasingTableau*

Factory classes:

- *Tableaux*

- *SemistandardTableaux*
- *StandardTableaux*
- *RowStandardTableaux*
- *IncreasingTableaux*

Parent classes:

- *Tableaux_all*
- *Tableaux_size*
- *SemistandardTableaux_all* (facade class)
- *SemistandardTableaux_size*
- *SemistandardTableaux_size_inf*
- *SemistandardTableaux_size_weight*
- *SemistandardTableaux_shape*
- *SemistandardTableaux_shape_inf*
- *SemistandardTableaux_shape_weight*
- *StandardTableaux_all* (facade class)
- *StandardTableaux_size*
- *StandardTableaux_shape*
- *IncreasingTableaux_all* (facade class)
- *IncreasingTableaux_size*
- *IncreasingTableaux_size_inf*
- *IncreasingTableaux_size_weight*
- *IncreasingTableaux_shape*
- *IncreasingTableaux_shape_inf*
- *IncreasingTableaux_shape_weight*
- *RowStandardTableaux_all* (facade class)
- *RowStandardTableaux_size*
- *RowStandardTableaux_shape*

For display options, see *Tableaux.options()*.

Todo

- Move methods that only apply to semistandard tableaux from tableau to semistandard tableau
- Copy/move functionality to skew tableaux
- Add a class for tableaux of a given shape (eg Tableaux_shape)

class sage.combinat.tableau.**IncreasingTableau**(*parent*, *t*, *check=True*)

Bases: *Tableau*

A class to model an increasing tableau.

INPUT:

- t – a tableau, a list of iterables, or an empty list

An *increasing tableau* is a tableau whose entries are positive integers that are strictly increasing across rows and strictly increasing down columns.

EXAMPLES:

```
sage: t = IncreasingTableau([[1,2,3],[2,3]]); t
[[1, 2, 3], [2, 3]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty printing
1 2 3
2 3
sage: t = Tableau([[1,2],[2]])
sage: s = IncreasingTableau(t); s
[[1, 2], [2]]
sage: IncreasingTableau([]) # The empty tableau
[]
```

```
>>> from sage.all import *
>>> t = IncreasingTableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),
↪Integer(3)]]); t
[[1, 2, 3], [2, 3]]
>>> t.shape()
[3, 2]
>>> t.pp() # pretty printing
1 2 3
2 3
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(2)]])
>>> s = IncreasingTableau(t); s
[[1, 2], [2]]
>>> IncreasingTableau([]) # The empty tableau
[]
```

You can also construct an *IncreasingTableau* from the appropriate Parent object:

```
sage: IT = IncreasingTableaux()
sage: IT([[1, 2, 3], [4, 5]])
[[1, 2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> IT = IncreasingTableaux()
>>> IT([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5)]])
[[1, 2, 3], [4, 5]]
```

See also

- *Tableaux*
- *Tableau*
- *SemistandardTableaux*
- *SemistandardTableau*
- *StandardTableaux*
- *StandardTableau*
- *IncreasingTableaux*

K_bender_knuth(*i*)

Return the i-th K-Bender-Knuth operator (as defined in [DPS2017]) applied to self.

The i-th K-Bender-Knuth operator swaps the letters i and $i + 1$ everywhere where doing so would not break increasingness.

EXAMPLES:

```
sage: T = IncreasingTableau([[1,3,4],[2,4,5]])
sage: T.K_bender_knuth(2)
[[1, 2, 4], [3, 4, 5]]
sage: T.K_bender_knuth(3)
[[1, 3, 4], [2, 4, 5]]
```

```
>>> from sage.all import *
>>> T = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(4),Integer(5)]])
>>> T.K_bender_knuth(Integer(2))
[[1, 2, 4], [3, 4, 5]]
>>> T.K_bender_knuth(Integer(3))
[[1, 3, 4], [2, 4, 5]]
```

K_evacuation(*ceiling=None*)

Return the K-evacuation involution from [TY2009] to self.

EXAMPLES:

```
sage: T = IncreasingTableau([[1,3,4],[2,4,5]])
sage: T.K_evacuation()
[[1, 2, 4], [2, 3, 5]]
sage: T.K_evacuation(6)
[[2, 3, 5], [3, 4, 6]]
sage: U = IncreasingTableau([[1,3,4],[3,4,5],[5]])
sage: U.K_evacuation()
[[1, 2, 3], [2, 3, 5], [3]]
```

```
>>> from sage.all import *
>>> T = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(4),Integer(5)]])
>>> T.K_evacuation()
[[1, 2, 4], [2, 3, 5]]
>>> T.K_evacuation(Integer(6))
```

(continues on next page)

(continued from previous page)

```
[[2, 3, 5], [3, 4, 6]]
>>> U = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(3),
↪Integer(4),Integer(5)],[Integer(5)]])
>>> U.K_evacuation()
[[1, 2, 3], [2, 3, 5], [3]]
```

K_promotion(*ceiling=None*)

Return the K-promotion operator from [Pec2014] applied to `self`.

EXAMPLES:

```
sage: T = IncreasingTableau([[1,3,4],[2,4,5]])
sage: T.K_promotion()
[[1, 2, 3], [3, 4, 5]]
sage: T.K_promotion(6)
[[1, 2, 3], [3, 4, 6]]
sage: U = IncreasingTableau([[1,3,4],[3,4,5],[5]])
sage: U.K_promotion()
[[2, 3, 4], [3, 4, 5], [4]]
```

```
>>> from sage.all import *
>>> T = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(4),Integer(5)]])
>>> T.K_promotion()
[[1, 2, 3], [3, 4, 5]]
>>> T.K_promotion(Integer(6))
[[1, 2, 3], [3, 4, 6]]
>>> U = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(3),
↪Integer(4),Integer(5)],[Integer(5)]])
>>> U.K_promotion()
[[2, 3, 4], [3, 4, 5], [4]]
```

K_promotion_inverse(*ceiling=None*)

Return the inverse of K-promotion operator applied to `self`.

EXAMPLES:

```
sage: T = IncreasingTableau([[1,3,4],[2,4,5]])
sage: T.K_promotion_inverse()
[[1, 2, 4], [3, 4, 5]]
sage: T.K_promotion_inverse(6)
[[2, 4, 5], [3, 5, 6]]
sage: U = IncreasingTableau([[1,3,4],[3,4,5],[5]])
sage: U.K_promotion_inverse()
[[1, 2, 4], [2, 4, 5], [4]]
```

```
>>> from sage.all import *
>>> T = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(4),Integer(5)]])
>>> T.K_promotion_inverse()
[[1, 2, 4], [3, 4, 5]]
>>> T.K_promotion_inverse(Integer(6))
```

(continues on next page)

(continued from previous page)

```
[[2, 4, 5], [3, 5, 6]]
>>> U = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(3),
↪Integer(4),Integer(5)],[Integer(5)]])
>>> U.K_promotion_inverse()
[[1, 2, 4], [2, 4, 5], [4]]
```

check()

Check that `self` is a valid increasing tableau.

descent_set()

Compute the descents of the increasing tableau `self` as defined in [DPS2017].

The number i is a *descent* of an increasing tableau if some instance of $i+1$ appears in a lower row than some instance of i.

> **Note**
>
> This notion is close to the notion of descent for a standard tableau but is unrelated to the notion for semistandard tableaux.

EXAMPLES:

```
sage: T = IncreasingTableau([[1,2,4],[3,5,6]])
sage: T.descent_set()
[2, 4]
sage: U = IncreasingTableau([[1,3,4],[2,4,5]])
sage: U.descent_set()
[1, 3, 4]
sage: V = IncreasingTableau([[1,3,4],[3,4,5],[4,5]])
sage: V.descent_set()
[3, 4]
```

```
>>> from sage.all import *
>>> T = IncreasingTableau([[Integer(1),Integer(2),Integer(4)],[Integer(3),
↪Integer(5),Integer(6)]])
>>> T.descent_set()
[2, 4]
>>> U = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(4),Integer(5)]])
>>> U.descent_set()
[1, 3, 4]
>>> V = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(3),
↪Integer(4),Integer(5)],[Integer(4),Integer(5)]])
>>> V.descent_set()
[3, 4]
```

dual_K_evacuation(*ceiling=None*)

Return the dual K-evacuation involution applied to `self`.

EXAMPLES:

```
sage: T = IncreasingTableau([[1,3,4],[2,4,5]])
sage: T.dual_K_evacuation()
[[1, 2, 4], [2, 3, 5]]
sage: T.dual_K_evacuation(6)
[[2, 3, 5], [3, 4, 6]]
sage: U = IncreasingTableau([[1,3,4],[3,4,5],[5]])
sage: U.dual_K_evacuation()
[[1, 2, 3], [2, 3, 5], [3]]
```

```
>>> from sage.all import *
>>> T = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(4),Integer(5)]])
>>> T.dual_K_evacuation()
[[1, 2, 4], [2, 3, 5]]
>>> T.dual_K_evacuation(Integer(6))
[[2, 3, 5], [3, 4, 6]]
>>> U = IncreasingTableau([[Integer(1),Integer(3),Integer(4)],[Integer(3),
↪Integer(4),Integer(5)],[Integer(5)]])
>>> U.dual_K_evacuation()
[[1, 2, 3], [2, 3, 5], [3]]
```

class sage.combinat.tableau.**IncreasingTableaux**(***kwds*)

Bases: `Tableaux`

A factory class for the various classes of increasing tableaux.

An *increasing tableau* is a tableau whose entries are positive integers that are strictly increasing across rows and strictly increasing down columns. Note that Sage uses the English convention for partitions and tableaux; the longer rows are displayed on top.

INPUT:

Keyword arguments:

- `size` – the size of the tableaux
- `shape` – the shape of the tableaux
- `eval` – the weight (also called binary content) of the tableaux, where values can be either 0 or 1 with position i being 1 if and only if i can appear in the tableaux
- `max_entry` – positive integer or infinity (`oo`); the maximum entry for the tableaux; if `size` or `shape` are specified, `max_entry` defaults to be `size` or the size of `shape`

Positional arguments:

- the first argument is interpreted as either `size` or `shape` according to whether it is an integer or a partition
- the second keyword argument will always be interpreted as `eval`

Warning

The `eval` is not the usual notion of `eval` or `weight`, where the i-th entry counts how many i's appear in the tableau.

EXAMPLES:

```
sage: IT = IncreasingTableaux([2,1]); IT
Increasing tableaux of shape [2, 1] and maximum entry 3
sage: IT.list()
[[[1, 3], [2]], [[1, 2], [3]], [[1, 2], [2]], [[1, 3], [3]], [[2, 3], [3]]]

sage: IT = IncreasingTableaux(3); IT
Increasing tableaux of size 3 and maximum entry 3
sage: IT.list()
[[[1, 2, 3]],
 [[1, 3], [2]],
 [[1, 2], [3]],
 [[1, 2], [2]],
 [[1, 3], [3]],
 [[2, 3], [3]],
 [[1], [2], [3]]]

sage: IT = IncreasingTableaux(3, max_entry=2); IT
Increasing tableaux of size 3 and maximum entry 2
sage: IT.list()
[[[1, 2], [2]]]

sage: IT = IncreasingTableaux(3, max_entry=4); IT
Increasing tableaux of size 3 and maximum entry 4
sage: IT.list()
[[[1, 2, 3]],
 [[1, 2, 4]],
 [[1, 3, 4]],
 [[2, 3, 4]],
 [[1, 3], [2]],
 [[1, 2], [3]],
 [[1, 4], [2]],
 [[1, 2], [4]],
 [[1, 2], [2]],
 [[1, 4], [3]],
 [[1, 3], [4]],
 [[1, 3], [3]],
 [[1, 4], [4]],
 [[2, 4], [3]],
 [[2, 3], [4]],
 [[2, 3], [3]],
 [[2, 4], [4]],
 [[3, 4], [4]],
 [[1], [2], [3]],
 [[1], [2], [4]],
 [[1], [3], [4]],
 [[2], [3], [4]]]

sage: IT = IncreasingTableaux(3, max_entry=oo); IT
Increasing tableaux of size 3
sage: IT[123]
[[5, 7], [6]]

sage: IT = IncreasingTableaux(max_entry=2)
```

(continues on next page)

(continued from previous page)

```
sage: list(IT)
[[], [[1]], [[2]], [[1, 2]], [[1], [2]]]
sage: IT[4]
[[1], [2]]

sage: IncreasingTableaux()[0]
[]
```

```
>>> from sage.all import *
>>> IT = IncreasingTableaux([Integer(2),Integer(1)]); IT
Increasing tableaux of shape [2, 1] and maximum entry 3
>>> IT.list()
[[[1, 3], [2]], [[1, 2], [3]], [[1, 2], [2]], [[1, 3], [3]], [[2, 3], [3]]]

>>> IT = IncreasingTableaux(Integer(3)); IT
Increasing tableaux of size 3 and maximum entry 3
>>> IT.list()
[[[1, 2, 3]],
 [[1, 3], [2]],
 [[1, 2], [3]],
 [[1, 2], [2]],
 [[1, 3], [3]],
 [[2, 3], [3]],
 [[1], [2], [3]]]

>>> IT = IncreasingTableaux(Integer(3), max_entry=Integer(2)); IT
Increasing tableaux of size 3 and maximum entry 2
>>> IT.list()
[[[1, 2], [2]]]

>>> IT = IncreasingTableaux(Integer(3), max_entry=Integer(4)); IT
Increasing tableaux of size 3 and maximum entry 4
>>> IT.list()
[[[1, 2, 3]],
 [[1, 2, 4]],
 [[1, 3, 4]],
 [[2, 3, 4]],
 [[1, 3], [2]],
 [[1, 2], [3]],
 [[1, 4], [2]],
 [[1, 2], [4]],
 [[1, 2], [2]],
 [[1, 4], [3]],
 [[1, 3], [4]],
 [[1, 3], [3]],
 [[1, 4], [4]],
 [[2, 4], [3]],
 [[2, 3], [4]],
 [[2, 3], [3]],
 [[2, 4], [4]],
 [[3, 4], [4]],
 [[1], [2], [3]],
```

(continues on next page)

(continued from previous page)

```
 [[1], [2], [4]],
 [[1], [3], [4]],
 [[2], [3], [4]]]

>>> IT = IncreasingTableaux(Integer(3), max_entry=oo); IT
Increasing tableaux of size 3
>>> IT[Integer(123)]
[[5, 7], [6]]

>>> IT = IncreasingTableaux(max_entry=Integer(2))
>>> list(IT)
[[], [[1]], [[2]], [[1, 2]], [[1], [2]]]
>>> IT[Integer(4)]
[[1], [2]]

>>> IncreasingTableaux()[Integer(0)]
[]
```

See also

- *Tableaux*
- *Tableau*
- *SemistandardTableaux*
- *SemistandardTableau*
- *StandardTableaux*
- *StandardTableau*
- *IncreasingTableau*

Element

alias of *IncreasingTableau*

class sage.combinat.tableau.**IncreasingTableaux_all**(*max_entry=None*)

Bases: *IncreasingTableaux*, DisjointUnionEnumeratedSets

All increasing tableaux.

EXAMPLES:

```
sage: T = IncreasingTableaux()
sage: T.cardinality()
+Infinity

sage: T = IncreasingTableaux(max_entry=3)
sage: list(T)
[[],
 [[1]],
 [[2]],
 [[3]],
```

(continues on next page)

(continued from previous page)

```
 [[1, 2]],
 [[1, 3]],
 [[2, 3]],
 [[1], [2]],
 [[1], [3]],
 [[2], [3]],
 [[1, 2, 3]],
 [[1, 3], [2]],
 [[1, 2], [3]],
 [[1, 2], [2]],
 [[1, 3], [3]],
 [[2, 3], [3]],
 [[1], [2], [3]]]
```

```
>>> from sage.all import *
>>> T = IncreasingTableaux()
>>> T.cardinality()
+Infinity

>>> T = IncreasingTableaux(max_entry=Integer(3))
>>> list(T)
[[],
 [[1]],
 [[2]],
 [[3]],
 [[1, 2]],
 [[1, 3]],
 [[2, 3]],
 [[1], [2]],
 [[1], [3]],
 [[2], [3]],
 [[1, 2, 3]],
 [[1, 3], [2]],
 [[1, 2], [3]],
 [[1, 2], [2]],
 [[1, 3], [3]],
 [[2, 3], [3]],
 [[1], [2], [3]]]
```

class sage.combinat.tableau.**IncreasingTableaux_shape**(*p*, *max_entry=None*)

Bases: *IncreasingTableaux*

Increasing tableaux of fixed shape p with a given max entry.

An increasing tableau with max entry i is required to have all its entries less or equal to i. It is not required to actually contain an entry i.

INPUT:

- p – a partition
- max_entry – the max entry; defaults to the size of p

class sage.combinat.tableau.**IncreasingTableaux_shape_inf**(*p*)

Bases: *IncreasingTableaux*

Increasing tableaux of fixed shape p and no maximum entry.

class sage.combinat.tableau.**IncreasingTableaux_shape_weight**(*p*, *wt*)

Bases: *IncreasingTableaux_shape*

Increasing tableaux of fixed shape p and binary weight wt.

class sage.combinat.tableau.**IncreasingTableaux_size**(*n*, *max_entry=None*)

Bases: *IncreasingTableaux*

Increasing tableaux of fixed size n.

class sage.combinat.tableau.**IncreasingTableaux_size_inf**(*n*)

Bases: *IncreasingTableaux*

Increasing tableaux of fixed size n with no maximum entry.

class sage.combinat.tableau.**IncreasingTableaux_size_weight**(*n*, *wt*)

Bases: *IncreasingTableaux*

Increasing tableaux of fixed size n and weight wt.

class sage.combinat.tableau.**RowStandardTableau**(*parent*, *t*, *check=True*)

Bases: *Tableau*

A class to model a row standard tableau.

A row standard tableau is a tableau whose entries are positive integers from 1 to m that increase along rows.

INPUT:

- t – a Tableau, a list of iterables, or an empty list

EXAMPLES:

```
sage: t = RowStandardTableau([[3,4,5],[1,2]]); t
[[3, 4, 5], [1, 2]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty printing
3  4  5
1  2
sage: t.is_standard()
False
sage: RowStandardTableau([]) # The empty tableau
[]
sage: RowStandardTableau([[3,4,5],[1,2]]) in StandardTableaux()
False
sage: RowStandardTableau([[1,2,5],[3,4]]) in StandardTableaux()
True
```

```
>>> from sage.all import *
>>> t = RowStandardTableau([[Integer(3),Integer(4),Integer(5)],[Integer(1),
↪Integer(2)]]); t
[[3, 4, 5], [1, 2]]
>>> t.shape()
[3, 2]
>>> t.pp() # pretty printing
3  4  5
```

(continues on next page)

(continued from previous page)

```
1  2
>>> t.is_standard()
False
>>> RowStandardTableau([]) # The empty tableau
[]
>>> RowStandardTableau([[Integer(3),Integer(4),Integer(5)],[Integer(1),
↪Integer(2)]]) in StandardTableaux()
False
>>> RowStandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]) in StandardTableaux()
True
```

When using code that will generate a lot of tableaux, it is more efficient to construct a `RowStandardTableau` from the appropriate `Parent` object:

```
sage: ST = RowStandardTableaux()
sage: ST([[3, 4, 5], [1, 2]])
[[3, 4, 5], [1, 2]]
```

```
>>> from sage.all import *
>>> ST = RowStandardTableaux()
>>> ST([[Integer(3), Integer(4), Integer(5)], [Integer(1), Integer(2)]])
[[3, 4, 5], [1, 2]]
```

See also:

- `Tableau`
- `StandardTableau`
- `SemistandardTableau`
- `Tableaux`
- `StandardTableaux`
- `RowStandardTableaux`
- `SemistandardTableaux`

check()

Check that `self` is a valid row standard tableau.

class sage.combinat.tableau.**RowStandardTableaux**

Bases: `Tableaux`

A factory for the various classes of row standard tableaux.

INPUT:

- either a nonnegative integer (possibly specified with the keyword `n`) or a partition

OUTPUT: with no argument, the class of all standard tableaux

- with a nonnegative integer argument, `n`, the class of all standard tableaux of size `n`
- with a partition argument, the class of all standard tableaux of that shape

A row standard tableau is a tableau that contains each of the entries from 1 to n exactly once and is increasing along rows.

All classes of row standard tableaux are iterable.

EXAMPLES:

```
sage: ST = RowStandardTableaux(3); ST
Row standard tableaux of size 3
sage: ST.first()                                                            #␣
↪needs sage.graphs
[[1, 2, 3]]
sage: ST.last()                                                             #␣
↪needs sage.graphs sage.modules
[[3], [1], [2]]
sage: ST.cardinality()                                                      #␣
↪needs sage.graphs sage.modules
10
sage: ST.list()                                                             #␣
↪needs sage.graphs sage.modules
[[[1, 2, 3]],
 [[2, 3], [1]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[3], [2], [1]],
 [[2], [3], [1]],
 [[1], [3], [2]],
 [[1], [2], [3]],
 [[2], [1], [3]],
 [[3], [1], [2]]]
```

```
>>> from sage.all import *
>>> ST = RowStandardTableaux(Integer(3)); ST
Row standard tableaux of size 3
>>> ST.first()                                                              #␣
↪needs sage.graphs
[[1, 2, 3]]
>>> ST.last()                                                               #␣
↪needs sage.graphs sage.modules
[[3], [1], [2]]
>>> ST.cardinality()                                                        #␣
↪needs sage.graphs sage.modules
10
>>> ST.list()                                                               #␣
↪needs sage.graphs sage.modules
[[[1, 2, 3]],
 [[2, 3], [1]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[3], [2], [1]],
 [[2], [3], [1]],
 [[1], [3], [2]],
 [[1], [2], [3]],
 [[2], [1], [3]],
```

(continues on next page)

(continued from previous page)

```
[[3], [1], [2]]]
```

> **See also**
>
> - *Tableaux*
> - *Tableau*
> - *SemistandardTableaux*
> - *SemistandardTableau*
> - *RowStandardTableau*
> - *StandardSkewTableaux*

Element

alias of *RowStandardTableau*

class sage.combinat.tableau.**RowStandardTableaux_all**

Bases: *RowStandardTableaux*, DisjointUnionEnumeratedSets

All row standard tableaux.

class sage.combinat.tableau.**RowStandardTableaux_shape**(*p*)

Bases: *RowStandardTableaux*

Row Standard tableaux of a fixed shape p.

cardinality()

Return the number of row standard tableaux of this shape.

This is just the index of the corresponding Young subgroup in the full symmetric group.

EXAMPLES:

```
sage: RowStandardTableaux([3,2,1]).cardinality()
60
sage: RowStandardTableaux([2,2]).cardinality()
6
sage: RowStandardTableaux([5]).cardinality()
1
sage: RowStandardTableaux([6,5,5,3]).cardinality()
1955457504
sage: RowStandardTableaux([]).cardinality()
1
```

```
>>> from sage.all import *
>>> RowStandardTableaux([Integer(3),Integer(2),Integer(1)]).cardinality()
60
>>> RowStandardTableaux([Integer(2),Integer(2)]).cardinality()
6
>>> RowStandardTableaux([Integer(5)]).cardinality()
1
>>> RowStandardTableaux([Integer(6),Integer(5),Integer(5),Integer(3)]).
```

(continues on next page)

(continued from previous page)

```
↪cardinality()
1955457504
>>> RowStandardTableaux([]).cardinality()
1
```

class sage.combinat.tableau.**RowStandardTableaux_size**(*n*)

Bases: *RowStandardTableaux*, DisjointUnionEnumeratedSets

Row standard tableaux of fixed size n.

EXAMPLES:

```
sage: [t for t in RowStandardTableaux(1)]                                   #␣
↪needs sage.graphs
[[[1]]]
sage: [t for t in RowStandardTableaux(2)]                                   #␣
↪needs sage.graphs
[[[1, 2]], [[2], [1]], [[1], [2]]]
sage: list(RowStandardTableaux(3))                                          #␣
↪needs sage.graphs
[[[1, 2, 3]],
 [[2, 3], [1]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[3], [2], [1]],
 [[2], [3], [1]],
 [[1], [3], [2]],
 [[1], [2], [3]],
 [[2], [1], [3]],
 [[3], [1], [2]]]
```

```
>>> from sage.all import *
>>> [t for t in RowStandardTableaux(Integer(1))]                            ␣
↪     # needs sage.graphs
[[[1]]]
>>> [t for t in RowStandardTableaux(Integer(2))]                            ␣
↪     # needs sage.graphs
[[[1, 2]], [[2], [1]], [[1], [2]]]
>>> list(RowStandardTableaux(Integer(3)))                                   ␣
↪     # needs sage.graphs
[[[1, 2, 3]],
 [[2, 3], [1]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[3], [2], [1]],
 [[2], [3], [1]],
 [[1], [3], [2]],
 [[1], [2], [3]],
 [[2], [1], [3]],
 [[3], [1], [2]]]
```

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: RowStandardTableaux(4).an_element()
[[1, 2, 3, 4]]
```

```
>>> from sage.all import *
>>> RowStandardTableaux(Integer(4)).an_element()
[[1, 2, 3, 4]]
```

class sage.combinat.tableau.**SemistandardTableau**(*parent*, *t*, *check=True*)

Bases: *Tableau*

A class to model a semistandard tableau.

INPUT:

- t – a tableau, a list of iterables, or an empty list

OUTPUT: a SemistandardTableau object constructed from t

A semistandard tableau is a tableau whose entries are positive integers, which are weakly increasing in rows and strictly increasing down columns.

EXAMPLES:

```
sage: t = SemistandardTableau([[1,2,3],[2,3]]); t
[[1, 2, 3], [2, 3]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty printing
1 2 3
2 3
sage: t = Tableau([[1,2],[2]])
sage: s = SemistandardTableau(t); s
[[1, 2], [2]]
sage: SemistandardTableau([]) # The empty tableau
[]
```

```
>>> from sage.all import *
>>> t = SemistandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),
→Integer(3)]]); t
[[1, 2, 3], [2, 3]]
>>> t.shape()
[3, 2]
>>> t.pp() # pretty printing
1 2 3
2 3
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(2)]])
>>> s = SemistandardTableau(t); s
[[1, 2], [2]]
>>> SemistandardTableau([]) # The empty tableau
[]
```

When using code that will generate a lot of tableaux, it is slightly more efficient to construct a SemistandardTableau from the appropriate Parent object:

```
sage: SST = SemistandardTableaux()
sage: SST([[1, 2, 3], [4, 5]])
[[1, 2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> SST = SemistandardTableaux()
>>> SST([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5)]])
[[1, 2, 3], [4, 5]]
```

See also

- *Tableaux*
- *Tableau*
- *SemistandardTableaux*
- *StandardTableaux*
- *StandardTableau*

check()

Check that `self` is a valid semistandard tableau.

class sage.combinat.tableau.**SemistandardTableaux**(***kwds*)

Bases: *Tableaux*

A factory class for the various classes of semistandard tableaux.

INPUT:

Keyword arguments:

- `size` – the size of the tableaux
- `shape` – the shape of the tableaux
- `eval` – the weight (also called content or evaluation) of the tableaux
- `max_entry` – a maximum entry for the tableaux. This can be a positive integer or infinity (`oo`). If `size` or `shape` are specified, `max_entry` defaults to be `size` or the size of `shape`.

Positional arguments:

- The first argument is interpreted as either `size` or `shape` according to whether it is an integer or a partition
- The second keyword argument will always be interpreted as `eval`

OUTPUT:

- The appropriate class, after checking basic consistency tests. (For example, specifying `eval` implies a value for `max_entry`).

A semistandard tableau is a tableau whose entries are positive integers, which are weakly increasing in rows and strictly increasing down columns. Note that Sage uses the English convention for partitions and tableaux; the longer rows are displayed on top.

Classes of semistandard tableaux can be iterated over if and only if there is some restriction.

EXAMPLES:

```
sage: SST = SemistandardTableaux([2,1]); SST
Semistandard tableaux of shape [2, 1] and maximum entry 3
sage: SST.list()                                                          #␣
↪needs sage.modules
[[[1, 1], [2]],
 [[1, 1], [3]],
 [[1, 2], [2]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[1, 3], [3]],
 [[2, 2], [3]],
 [[2, 3], [3]]]

sage: SST = SemistandardTableaux(3); SST
Semistandard tableaux of size 3 and maximum entry 3
sage: SST.list()                                                          #␣
↪needs sage.modules
[[[1, 1, 1]],
 [[1, 1, 2]],
 [[1, 1, 3]],
 [[1, 2, 2]],
 [[1, 2, 3]],
 [[1, 3, 3]],
 [[2, 2, 2]],
 [[2, 2, 3]],
 [[2, 3, 3]],
 [[3, 3, 3]],
 [[1, 1], [2]],
 [[1, 1], [3]],
 [[1, 2], [2]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[1, 3], [3]],
 [[2, 2], [3]],
 [[2, 3], [3]],
 [[1], [2], [3]]]

sage: SST = SemistandardTableaux(3, max_entry=2); SST
Semistandard tableaux of size 3 and maximum entry 2
sage: SST.list()                                                          #␣
↪needs sage.modules
[[[1, 1, 1]],
 [[1, 1, 2]],
 [[1, 2, 2]],
 [[2, 2, 2]],
 [[1, 1], [2]],
 [[1, 2], [2]]]

sage: SST = SemistandardTableaux(3, max_entry=oo); SST
Semistandard tableaux of size 3
sage: SST[123]                                                            #␣
↪needs sage.modules
[[3, 4], [6]]
```

(continues on next page)

(continued from previous page)

```
sage: SemistandardTableaux(max_entry=2)[11]                                    #␣
↪needs sage.modules
[[1, 1], [2]]

sage: SemistandardTableaux()[0]                                                #␣
↪needs sage.modules
[]
```

```
>>> from sage.all import *
>>> SST = SemistandardTableaux([Integer(2),Integer(1)]); SST
Semistandard tableaux of shape [2, 1] and maximum entry 3
>>> SST.list()                                                                 #␣
↪needs sage.modules
[[[1, 1], [2]],
 [[1, 1], [3]],
 [[1, 2], [2]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[1, 3], [3]],
 [[2, 2], [3]],
 [[2, 3], [3]]]

>>> SST = SemistandardTableaux(Integer(3)); SST
Semistandard tableaux of size 3 and maximum entry 3
>>> SST.list()                                                                 #␣
↪needs sage.modules
[[[1, 1, 1]],
 [[1, 1, 2]],
 [[1, 1, 3]],
 [[1, 2, 2]],
 [[1, 2, 3]],
 [[1, 3, 3]],
 [[2, 2, 2]],
 [[2, 2, 3]],
 [[2, 3, 3]],
 [[3, 3, 3]],
 [[1, 1], [2]],
 [[1, 1], [3]],
 [[1, 2], [2]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[1, 3], [3]],
 [[2, 2], [3]],
 [[2, 3], [3]],
 [[1], [2], [3]]]

>>> SST = SemistandardTableaux(Integer(3), max_entry=Integer(2)); SST
Semistandard tableaux of size 3 and maximum entry 2
>>> SST.list()                                                                 #␣
↪needs sage.modules
[[[1, 1, 1]],
```

(continues on next page)

(continued from previous page)

```
 [[1, 1, 2]],
 [[1, 2, 2]],
 [[2, 2, 2]],
 [[1, 1], [2]],
 [[1, 2], [2]]]

>>> SST = SemistandardTableaux(Integer(3), max_entry=oo); SST
Semistandard tableaux of size 3
>>> SST[Integer(123)]
↪     # needs sage.modules
[[3, 4], [6]]

>>> SemistandardTableaux(max_entry=Integer(2))[Integer(11)]
↪              # needs sage.modules
[[1, 1], [2]]

>>> SemistandardTableaux()[Integer(0)]
↪     # needs sage.modules
[]
```

See also:

- *Tableaux*
- *Tableau*
- *SemistandardTableau*
- *StandardTableaux*
- *StandardTableau*

Element

alias of *SemistandardTableau*

class sage.combinat.tableau.**SemistandardTableaux_all**(*max_entry=None*)

Bases: *SemistandardTableaux*, DisjointUnionEnumeratedSets

All semistandard tableaux.

list()

class sage.combinat.tableau.**SemistandardTableaux_shape**(*p*, *max_entry=None*)

Bases: *SemistandardTableaux*

Semistandard tableaux of fixed shape p with a given max entry.

A semistandard tableau with max entry i is required to have all its entries less or equal to i. It is not required to actually contain an entry i.

INPUT:

- `p` – a partition
- `max_entry` – the max entry; defaults to the size of p

cardinality(*algorithm='hook'*)

Return the cardinality of `self`.

INPUT:

- `algorithm` – (default: `'hook'`) any one of the following:
  - `'hook'` – use Stanley's hook length formula
  - `'sum'` – sum over the compositions of `max_entry` the number of semistandard tableau with `shape` and given weight vector

This is computed using *Stanley's hook length formula*:

$$f_{\lambda} = \prod_{u \in \lambda} \frac{n + c(u)}{h(u)}.$$

where n is the `max_entry`, $c(u)$ is the content of u, and $h(u)$ is the hook length of u. See [Sta-EC2] Corollary 7.21.4.

EXAMPLES:

```
sage: SemistandardTableaux([2,1]).cardinality()
8
sage: SemistandardTableaux([2,2,1]).cardinality()
75
sage: SymmetricFunctions(QQ).schur()([2,2,1]).expand(5)(1,1,1,1,1)  # cross␣
↪check       # needs sage.modules
75
sage: SemistandardTableaux([5]).cardinality()
126
sage: SemistandardTableaux([3,2,1]).cardinality()
896
sage: SemistandardTableaux([3,2,1], max_entry=7).cardinality()
2352
sage: SemistandardTableaux([6,5,4,3,2,1], max_entry=30).cardinality()
208361017592001331200
sage: ssts = [SemistandardTableaux(p, max_entry=6) for p in Partitions(5)]
sage: all(sst.cardinality() == sst.cardinality(algorithm='sum')          #␣
↪needs sage.modules
....:     for sst in ssts)
True
```

```
>>> from sage.all import *
>>> SemistandardTableaux([Integer(2),Integer(1)]).cardinality()
8
>>> SemistandardTableaux([Integer(2),Integer(2),Integer(1)]).cardinality()
75
>>> SymmetricFunctions(QQ).schur()([Integer(2),Integer(2),Integer(1)]).
↪expand(Integer(5))(Integer(1),Integer(1),Integer(1),Integer(1),Integer(1))
↪# cross check       # needs sage.modules
75
>>> SemistandardTableaux([Integer(5)]).cardinality()
126
>>> SemistandardTableaux([Integer(3),Integer(2),Integer(1)]).cardinality()
896
```

(continues on next page)

(continued from previous page)

```
>>> SemistandardTableaux([Integer(3),Integer(2),Integer(1)], max_
↪entry=Integer(7)).cardinality()
2352
>>> SemistandardTableaux([Integer(6),Integer(5),Integer(4),Integer(3),
↪Integer(2),Integer(1)], max_entry=Integer(30)).cardinality()
208361017592001331200
>>> ssts = [SemistandardTableaux(p, max_entry=Integer(6)) for p in␣
↪Partitions(Integer(5))]
>>> all(sst.cardinality() == sst.cardinality(algorithm='sum')          #␣
↪needs sage.modules
...     for sst in ssts)
True
```

random_element()

Return a uniformly distributed random tableau of the given `shape` and `max_entry`.

Uses the algorithm from [Kra1999] based on the Novelli-Pak-Stoyanovskii bijection
http://www.sciencedirect.com/science/article/pii/0012365X9290368P

EXAMPLES:

```
sage: S = SemistandardTableaux([2, 2, 1, 1])
sage: S.random_element() in S
True
sage: S = SemistandardTableaux([2, 2, 1, 1], max_entry=7)
sage: S.random_element() in S
True
```

```
>>> from sage.all import *
>>> S = SemistandardTableaux([Integer(2), Integer(2), Integer(1), Integer(1)])
>>> S.random_element() in S
True
>>> S = SemistandardTableaux([Integer(2), Integer(2), Integer(1), Integer(1)],
↪ max_entry=Integer(7))
>>> S.random_element() in S
True
```

class sage.combinat.tableau.**SemistandardTableaux_shape_inf**(*p*)

Bases: *SemistandardTableaux*

Semistandard tableaux of fixed shape p and no maximum entry.

class sage.combinat.tableau.**SemistandardTableaux_shape_weight**(*p*, *mu*)

Bases: *SemistandardTableaux_shape*

Semistandard tableaux of fixed shape p and weight μ.

cardinality()

Return the number of semistandard tableaux of the given shape and weight, as computed by `kostka_number` function of `symmetrica`.

EXAMPLES:

```
sage: # needs sage.modules
sage: SemistandardTableaux([2,2], [2, 1, 1]).cardinality()
```

(continues on next page)

(continued from previous page)

```
1
sage: SemistandardTableaux([2,2,2], [2, 2, 1,1]).cardinality()
1
sage: SemistandardTableaux([2,2,2], [2, 2, 2]).cardinality()
1
sage: SemistandardTableaux([3,2,1], [2, 2, 2]).cardinality()
2
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> SemistandardTableaux([Integer(2),Integer(2)], [Integer(2), Integer(1),
↪Integer(1)]).cardinality()
1
>>> SemistandardTableaux([Integer(2),Integer(2),Integer(2)], [Integer(2),
↪Integer(2), Integer(1),Integer(1)]).cardinality()
1
>>> SemistandardTableaux([Integer(2),Integer(2),Integer(2)], [Integer(2),
↪Integer(2), Integer(2)]).cardinality()
1
>>> SemistandardTableaux([Integer(3),Integer(2),Integer(1)], [Integer(2),
↪Integer(2), Integer(2)]).cardinality()
2
```

list()

Return a list of all semistandard tableaux in `self` generated by symmetrica.

EXAMPLES:

```
sage: # needs sage.modules
sage: SemistandardTableaux([2,2], [2, 1, 1]).list()
[[[1, 1], [2, 3]]]
sage: SemistandardTableaux([2,2,2], [2, 2, 1,1]).list()
[[[1, 1], [2, 2], [3, 4]]]
sage: SemistandardTableaux([2,2,2], [2, 2, 2]).list()
[[[1, 1], [2, 2], [3, 3]]]
sage: SemistandardTableaux([3,2,1], [2, 2, 2]).list()
[[[1, 1, 2], [2, 3], [3]], [[1, 1, 3], [2, 2], [3]]]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> SemistandardTableaux([Integer(2),Integer(2)], [Integer(2), Integer(1),
↪Integer(1)]).list()
[[[1, 1], [2, 3]]]
>>> SemistandardTableaux([Integer(2),Integer(2),Integer(2)], [Integer(2),
↪Integer(2), Integer(1),Integer(1)]).list()
[[[1, 1], [2, 2], [3, 4]]]
>>> SemistandardTableaux([Integer(2),Integer(2),Integer(2)], [Integer(2),
↪Integer(2), Integer(2)]).list()
[[[1, 1], [2, 2], [3, 3]]]
>>> SemistandardTableaux([Integer(3),Integer(2),Integer(1)], [Integer(2),
↪Integer(2), Integer(2)]).list()
[[[1, 1, 2], [2, 3], [3]], [[1, 1, 3], [2, 2], [3]]]
```

class sage.combinat.tableau.**SemistandardTableaux_size**(*n*, *max_entry=None*)

Bases: *SemistandardTableaux*

Semistandard tableaux of fixed size n.

cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: SemistandardTableaux(3).cardinality()
19
sage: SemistandardTableaux(4).cardinality()
116
sage: SemistandardTableaux(4, max_entry=2).cardinality()
9
sage: SemistandardTableaux(4, max_entry=10).cardinality()
4225
sage: ns = list(range(1, 6))
sage: ssts = [ SemistandardTableaux(n) for n in ns ]
sage: all(sst.cardinality() == len(sst.list()) for sst in ssts)              #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> SemistandardTableaux(Integer(3)).cardinality()
19
>>> SemistandardTableaux(Integer(4)).cardinality()
116
>>> SemistandardTableaux(Integer(4), max_entry=Integer(2)).cardinality()
9
>>> SemistandardTableaux(Integer(4), max_entry=Integer(10)).cardinality()
4225
>>> ns = list(range(Integer(1), Integer(6)))
>>> ssts = [ SemistandardTableaux(n) for n in ns ]
>>> all(sst.cardinality() == len(sst.list()) for sst in ssts)              #␣
↪needs sage.modules
True
```

random_element()

Generate a random *SemistandardTableau* with uniform probability.

The RSK algorithm gives a bijection between symmetric $k \times k$ matrices of nonnegative integers that sum to n and semistandard tableaux with size n and maximum entry k.

The number of $k \times k$ symmetric matrices of nonnegative integers having sum of elements on the diagonal i and sum of elements above the diagonal j is $\binom{k+i-1}{k-1}\binom{\binom{k}{2}+j-1}{\binom{k}{2}-1}$. We first choose the sum of the elements on the diagonal randomly weighted by the number of matrices having that trace. We then create random integer vectors of length k having that sum and use them to generate a $k \times k$ diagonal matrix. Then we take a random integer vector of length $\binom{k}{2}$ summing to half the remainder and distribute it symmetrically to the remainder of the matrix.

Applying RSK to the random symmetric matrix gives us a pair of identical *SemistandardTableau* of which we choose the first.

EXAMPLES:

```
sage: SemistandardTableaux(6).random_element()   # random                     #␣
↪needs sage.modules
[[1, 1, 2], [3, 5, 5]]
sage: SemistandardTableaux(6, max_entry=7).random_element()   # random        #␣
↪needs sage.modules
[[2, 4, 4, 6, 6, 6]]
```

```
>>> from sage.all import *
>>> SemistandardTableaux(Integer(6)).random_element()   # random              ␣
↪     # needs sage.modules
[[1, 1, 2], [3, 5, 5]]
>>> SemistandardTableaux(Integer(6), max_entry=Integer(7)).random_element()
↪# random       # needs sage.modules
[[2, 4, 4, 6, 6, 6]]
```

class sage.combinat.tableau.**SemistandardTableaux_size_inf**(*n*)

Bases: *SemistandardTableaux*

Semistandard tableaux of fixed size n with no maximum entry.

list()

class sage.combinat.tableau.**SemistandardTableaux_size_weight**(*n*, *mu*)

Bases: *SemistandardTableaux*

Semistandard tableaux of fixed size n and weight μ.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: SemistandardTableaux(3, [2,1]).cardinality()                            #␣
↪needs sage.modules
2
sage: SemistandardTableaux(4, [2,2]).cardinality()                            #␣
↪needs sage.modules
3
```

```
>>> from sage.all import *
>>> SemistandardTableaux(Integer(3), [Integer(2),Integer(1)]).cardinality()  ␣
↪                       # needs sage.modules
2
>>> SemistandardTableaux(Integer(4), [Integer(2),Integer(2)]).cardinality()  ␣
↪                       # needs sage.modules
3
```

class sage.combinat.tableau.**StandardTableau**(*parent*, *t*, *check=True*)

Bases: *SemistandardTableau*

A class to model a standard tableau.

INPUT:

- `t` – a Tableau, a list of iterables, or an empty list

A standard tableau is a semistandard tableau whose entries are exactly the positive integers from 1 to n, where n is the size of the tableau.

EXAMPLES:

```
sage: t = StandardTableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty printing
1 2 3
4 5
sage: t.is_standard()
True
sage: StandardTableau([]) # The empty tableau
[]
sage: StandardTableau([[1,2,3],[4,5]]) in RowStandardTableaux()
True
```

```
>>> from sage.all import *
>>> t = StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]]); t
[[1, 2, 3], [4, 5]]
>>> t.shape()
[3, 2]
>>> t.pp() # pretty printing
1 2 3
4 5
>>> t.is_standard()
True
>>> StandardTableau([]) # The empty tableau
[]
>>> StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])␣
↪in RowStandardTableaux()
True
```

When using code that will generate a lot of tableaux, it is more efficient to construct a StandardTableau from the appropriate `Parent` object:

```
sage: ST = StandardTableaux()
sage: ST([[1, 2, 3], [4, 5]])
[[1, 2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> ST = StandardTableaux()
>>> ST([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5)]])
[[1, 2, 3], [4, 5]]
```

See also

- *Tableaux*
- *Tableau*

- *SemistandardTableaux*
- *SemistandardTableau*
- *StandardTableaux*

check()

Check that `self` is a standard tableau.

dominates(*t*)

Return `True` if `self` dominates the tableau `t`.

That is, if the shape of the tableau restricted to k dominates the shape of `t` restricted to k, for $k = 1, 2, \ldots, n$.

When the two tableaux have the same shape, then this ordering coincides with the Bruhat ordering for the corresponding permutations.

INPUT:

- `t` – a tableau

EXAMPLES:

```
sage: s = StandardTableau([[1,2,3],[4,5]])
sage: t = StandardTableau([[1,2],[3,5],[4]])
sage: s.dominates(t)
True
sage: t.dominates(s)
False
sage: all(StandardTableau(s).dominates(t) for t in StandardTableaux([3,2]))
True
sage: s.dominates([[1,2,3,4,5]])
False
```

```
>>> from sage.all import *
>>> s = StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]])
>>> t = StandardTableau([[Integer(1),Integer(2)],[Integer(3),Integer(5)],
↪[Integer(4)]])
>>> s.dominates(t)
True
>>> t.dominates(s)
False
>>> all(StandardTableau(s).dominates(t) for t in StandardTableaux([Integer(3),
↪Integer(2)]))
True
>>> s.dominates([[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]])
False
```

down()

An iterator for all the standard tableaux that can be obtained from `self` by removing a cell. Note that this iterates just over a single tableau (or nothing if `self` is empty).

EXAMPLES:

```
sage: t = StandardTableau([[1,2],[3]])
sage: [x for x in t.down()]
[[[1, 2]]]
sage: t = StandardTableau([])
sage: [x for x in t.down()]
[]
```

```
>>> from sage.all import *
>>> t = StandardTableau([[Integer(1),Integer(2)],[Integer(3)]])
>>> [x for x in t.down()]
[[[1, 2]]]
>>> t = StandardTableau([])
>>> [x for x in t.down()]
[]
```

down_list()

Return a list of all the standard tableaux that can be obtained from `self` by removing a cell. Note that this is just a singleton list if `self` is nonempty, and an empty list otherwise.

EXAMPLES:

```
sage: t = StandardTableau([[1,2],[3]])
sage: t.down_list()
[[[1, 2]]]
sage: t = StandardTableau([])
sage: t.down_list()
[]
```

```
>>> from sage.all import *
>>> t = StandardTableau([[Integer(1),Integer(2)],[Integer(3)]])
>>> t.down_list()
[[[1, 2]]]
>>> t = StandardTableau([])
>>> t.down_list()
[]
```

is_standard()

Return `True` since `self` is a standard tableau.

EXAMPLES:

```
sage: StandardTableau([[1, 3], [2, 4]]).is_standard()
True
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1), Integer(3)], [Integer(2), Integer(4)]]).is_
↪standard()
True
```

promotion(*n=None*)

Return the image of `self` under the promotion operator.

The promotion operator, applied to a standard tableau t, does the following:

Remove the letter n from t, thus leaving a hole where it used to be. Apply jeu de taquin to move this hole southwest (in French notation) until it reaches the inner boundary of t. Fill 0 into the hole once jeu de taquin has completed. Finally, add 1 to each letter in the tableau. The resulting standard tableau is the image of t under the promotion operator.

This definition of promotion is precisely the one given in [Hai1992] (p. 90). It is the inverse of the maps called "promotion" in [Sag2011] (p. 23) and in [Stan2009].

See the `promotion()` method for a more general operator.

EXAMPLES:

```
sage: ST = StandardTableaux(7)
sage: all( st.promotion().promotion_inverse() == st for st in ST ) # long time
True
sage: all( st.promotion_inverse().promotion() == st for st in ST ) # long time
True
sage: st = StandardTableau([[1,2,5],[3,4]])
sage: parent(st.promotion())
Standard tableaux
```

```
>>> from sage.all import *
>>> ST = StandardTableaux(Integer(7))
>>> all( st.promotion().promotion_inverse() == st for st in ST ) # long time
True
>>> all( st.promotion_inverse().promotion() == st for st in ST ) # long time
True
>>> st = StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]])
>>> parent(st.promotion())
Standard tableaux
```

promotion_inverse(*n=None*)

Return the image of `self` under the inverse promotion operator. The optional variable m should be set to the size of `self` minus 1 for a minimal speedup; otherwise, it defaults to this number.

The inverse promotion operator, applied to a standard tableau t, does the following:

Remove the letter 1 from t, thus leaving a hole where it used to be. Apply jeu de taquin to move this hole northeast (in French notation) until it reaches the outer boundary of t. Fill $n + 1$ into this hole, where n is the size of t. Finally, subtract 1 from each letter in the tableau. This yields a new standard tableau.

This definition of inverse promotion is the map called "promotion" in [Sag2011] (p. 23) and in [Stan2009], and is the inverse of the map called "promotion" in [Hai1992] (p. 90).

See the `promotion_inverse()` method for a more general operator.

EXAMPLES:

```
sage: t = StandardTableau([[1,3],[2,4]])
sage: t.promotion_inverse()
[[1, 2], [3, 4]]
```

```
>>> from sage.all import *
>>> t = StandardTableau([[Integer(1),Integer(3)],[Integer(2),Integer(4)]])
>>> t.promotion_inverse()
[[1, 2], [3, 4]]
```

We check the equivalence of two definitions of inverse promotion on standard tableaux:

```
sage: ST = StandardTableaux(7)
sage: def bk_promotion_inverse7(st):
....:     st2 = st
....:     for i in range(1, 7):
....:         st2 = st2.bender_knuth_involution(i, check=False)
....:     return st2
sage: all( bk_promotion_inverse7(st) == st.promotion_inverse() for st in ST )
↪# long time
True
```

```
>>> from sage.all import *
>>> ST = StandardTableaux(Integer(7))
>>> def bk_promotion_inverse7(st):
...     st2 = st
...     for i in range(Integer(1), Integer(7)):
...         st2 = st2.bender_knuth_involution(i, check=False)
...     return st2
>>> all( bk_promotion_inverse7(st) == st.promotion_inverse() for st in ST ) #␣
↪long time
True
```

standard_descents()

Return a list of the integers i such that i appears strictly further north than $i + 1$ in `self` (this is not to say that i and $i + 1$ must be in the same column). The list is sorted in increasing order.

EXAMPLES:

```
sage: StandardTableau( [[1,3,4],[2,5]] ).standard_descents()
[1, 4]
sage: StandardTableau( [[1,2],[3,4]] ).standard_descents()
[2]
sage: StandardTableau( [[1,2,5],[3,4],[6,7],[8],[9]] ).standard_descents()
[2, 5, 7, 8]
sage: StandardTableau( [] ).standard_descents()
[]
```

```
>>> from sage.all import *
>>> StandardTableau( [[Integer(1),Integer(3),Integer(4)],[Integer(2),
↪Integer(5)]] ).standard_descents()
[1, 4]
>>> StandardTableau( [[Integer(1),Integer(2)],[Integer(3),Integer(4)]] ).
↪standard_descents()
[2]
>>> StandardTableau( [[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)],[Integer(6),Integer(7)],[Integer(8)],[Integer(9)]] ).standard_
↪descents()
[2, 5, 7, 8]
>>> StandardTableau( [] ).standard_descents()
[]
```

standard_major_index()

Return the major index of the standard tableau `self` in the standard meaning of the word. The major index is defined to be the sum of the descents of `self` (see *standard_descents()* for their definition).

EXAMPLES:

```
sage: StandardTableau( [[1,4,5],[2,6],[3]] ).standard_major_index()
8
sage: StandardTableau( [[1,2],[3,4]] ).standard_major_index()
2
sage: StandardTableau( [[1,2,3],[4,5]] ).standard_major_index()
3
```

```
>>> from sage.all import *
>>> StandardTableau( [[Integer(1),Integer(4),Integer(5)],[Integer(2),
↪Integer(6)],[Integer(3)]] ).standard_major_index()
8
>>> StandardTableau( [[Integer(1),Integer(2)],[Integer(3),Integer(4)]] ).
↪standard_major_index()
2
>>> StandardTableau( [[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]] ).standard_major_index()
3
```

standard_number_of_descents()

Return the number of all integers i such that i appears strictly further north than $i + 1$ in `self` (this is not to say that i and $i + 1$ must be in the same column). A list of these integers can be obtained using the *standard_descents()* method.

EXAMPLES:

```
sage: StandardTableau( [[1,2],[3,4],[5]] ).standard_number_of_descents()
2
sage: StandardTableau( [] ).standard_number_of_descents()
0
sage: tabs = StandardTableaux(5)
sage: all( t.standard_number_of_descents() == t.schuetzenberger_involution().
↪standard_number_of_descents() for t in tabs )
True
```

```
>>> from sage.all import *
>>> StandardTableau( [[Integer(1),Integer(2)],[Integer(3),Integer(4)],
↪[Integer(5)]] ).standard_number_of_descents()
2
>>> StandardTableau( [] ).standard_number_of_descents()
0
>>> tabs = StandardTableaux(Integer(5))
>>> all( t.standard_number_of_descents() == t.schuetzenberger_involution().
↪standard_number_of_descents() for t in tabs )
True
```

up()

An iterator for all the standard tableaux that can be obtained from `self` by adding a cell.

EXAMPLES:

```
sage: t = StandardTableau([[1,2]])
sage: [x for x in t.up()]
[[[1, 2, 3]], [[1, 2], [3]]]
```

```
>>> from sage.all import *
>>> t = StandardTableau([[Integer(1),Integer(2)]])
>>> [x for x in t.up()]
[[[1, 2, 3]], [[1, 2], [3]]]
```

up_list()

Return a list of all the standard tableaux that can be obtained from `self` by adding a cell.

EXAMPLES:

```
sage: t = StandardTableau([[1,2]])
sage: t.up_list()
[[[1, 2, 3]], [[1, 2], [3]]]
```

```
>>> from sage.all import *
>>> t = StandardTableau([[Integer(1),Integer(2)]])
>>> t.up_list()
[[[1, 2, 3]], [[1, 2], [3]]]
```

class sage.combinat.tableau.**StandardTableaux**(***kwds*)

Bases: *SemistandardTableaux*

A factory for the various classes of standard tableaux.

INPUT:

- Either a nonnegative integer (possibly specified with the keyword `n`) or a partition.

OUTPUT: with no argument, the class of all standard tableaux

- With a nonnegative integer argument, `n`, the class of all standard tableaux of size `n`
- With a partition argument, the class of all standard tableaux of that shape.

A standard tableau is a semistandard tableaux which contains each of the entries from 1 to `n` exactly once.

All classes of standard tableaux are iterable.

EXAMPLES:

```
sage: ST = StandardTableaux(3); ST
Standard tableaux of size 3
sage: ST.first()
[[1, 2, 3]]
sage: ST.last()
[[1], [2], [3]]
sage: ST.cardinality()
4
sage: ST.list()
[[[1, 2, 3]], [[1, 3], [2]], [[1, 2], [3]], [[1], [2], [3]]]
```

```
>>> from sage.all import *
>>> ST = StandardTableaux(Integer(3)); ST
Standard tableaux of size 3
>>> ST.first()
[[1, 2, 3]]
>>> ST.last()
[[1], [2], [3]]
>>> ST.cardinality()
4
>>> ST.list()
[[[1, 2, 3]], [[1, 3], [2]], [[1, 2], [3]], [[1], [2], [3]]]
```

See also:

- *Tableaux*
- *Tableau*
- *SemistandardTableaux*
- *SemistandardTableau*
- *StandardTableau*
- *StandardSkewTableaux*

Element

alias of *StandardTableau*

class sage.combinat.tableau.**StandardTableaux_all**

Bases: *StandardTableaux*, DisjointUnionEnumeratedSets

All standard tableaux.

class sage.combinat.tableau.**StandardTableaux_shape**(*p*)

Bases: *StandardTableaux*

Semistandard tableaux of a fixed shape p.

cardinality()

Return the number of standard Young tableaux of this shape.

This method uses the so-called *hook length formula*, a formula for the number of Young tableaux associated with a given partition. The formula says the following: Let λ be a partition. For each cell c of the Young diagram of λ, let the *hook length* of c be defined as 1 plus the number of cells horizontally to the right of c plus the number of cells vertically below c. The number of standard Young tableaux of shape λ is then $n!$ divided by the product of the hook lengths of the shape of λ, where $n = |\lambda|$.

For example, consider the partition `[3,2,1]` of `6` with Ferrers diagram:

```
# # #
# #
#
```

When we fill in the cells with their respective hook lengths, we obtain:

```
5 3 1
3 1
1
```

The hook length formula returns

$$\frac{6!}{5 \cdot 3 \cdot 1 \cdot 3 \cdot 1 \cdot 1} = 16.$$

EXAMPLES:

```
sage: StandardTableaux([3,2,1]).cardinality()
16
sage: StandardTableaux([2,2]).cardinality()
2
sage: StandardTableaux([5]).cardinality()
1
sage: StandardTableaux([6,5,5,3]).cardinality()
6651216
sage: StandardTableaux([]).cardinality()
1
```

```
>>> from sage.all import *
>>> StandardTableaux([Integer(3),Integer(2),Integer(1)]).cardinality()
16
>>> StandardTableaux([Integer(2),Integer(2)]).cardinality()
2
>>> StandardTableaux([Integer(5)]).cardinality()
1
>>> StandardTableaux([Integer(6),Integer(5),Integer(5),Integer(3)]).
↪cardinality()
6651216
>>> StandardTableaux([]).cardinality()
1
```

REFERENCES:

- http://mathworld.wolfram.com/HookLengthFormula.html

list()

Return a list of the standard Young tableaux of the specified shape.

EXAMPLES:

```
sage: StandardTableaux([2,2]).list()
[[[1, 3], [2, 4]], [[1, 2], [3, 4]]]
sage: StandardTableaux([5]).list()
[[[1, 2, 3, 4, 5]]]
sage: StandardTableaux([3,2,1]).list()
[[[1, 4, 6], [2, 5], [3]],
 [[1, 3, 6], [2, 5], [4]],
 [[1, 2, 6], [3, 5], [4]],
 [[1, 3, 6], [2, 4], [5]],
 [[1, 2, 6], [3, 4], [5]],
 [[1, 4, 5], [2, 6], [3]],
```

(continues on next page)

(continued from previous page)

```
 [[1, 3, 5], [2, 6], [4]],
 [[1, 2, 5], [3, 6], [4]],
 [[1, 3, 4], [2, 6], [5]],
 [[1, 2, 4], [3, 6], [5]],
 [[1, 2, 3], [4, 6], [5]],
 [[1, 3, 5], [2, 4], [6]],
 [[1, 2, 5], [3, 4], [6]],
 [[1, 3, 4], [2, 5], [6]],
 [[1, 2, 4], [3, 5], [6]],
 [[1, 2, 3], [4, 5], [6]]
```

```
>>> from sage.all import *
>>> StandardTableaux([Integer(2),Integer(2)]).list()
[[[1, 3], [2, 4]], [[1, 2], [3, 4]]]
>>> StandardTableaux([Integer(5)]).list()
[[[1, 2, 3, 4, 5]]]
>>> StandardTableaux([Integer(3),Integer(2),Integer(1)]).list()
[[[1, 4, 6], [2, 5], [3]],
 [[1, 3, 6], [2, 5], [4]],
 [[1, 2, 6], [3, 5], [4]],
 [[1, 3, 6], [2, 4], [5]],
 [[1, 2, 6], [3, 4], [5]],
 [[1, 4, 5], [2, 6], [3]],
 [[1, 3, 5], [2, 6], [4]],
 [[1, 2, 5], [3, 6], [4]],
 [[1, 3, 4], [2, 6], [5]],
 [[1, 2, 4], [3, 6], [5]],
 [[1, 2, 3], [4, 6], [5]],
 [[1, 3, 5], [2, 4], [6]],
 [[1, 2, 5], [3, 4], [6]],
 [[1, 3, 4], [2, 5], [6]],
 [[1, 2, 4], [3, 5], [6]],
 [[1, 2, 3], [4, 5], [6]]]
```

random_element()

Return a random standard tableau of the given shape using the Greene-Nijenhuis-Wilf Algorithm.

EXAMPLES:

```
sage: t = StandardTableaux([2,2]).random_element()
sage: t.shape()
[2, 2]
sage: StandardTableaux([]).random_element()
[]
```

```
>>> from sage.all import *
>>> t = StandardTableaux([Integer(2),Integer(2)]).random_element()
>>> t.shape()
[2, 2]
>>> StandardTableaux([]).random_element()
[]
```

class sage.combinat.tableau.**StandardTableaux_size**(*n*)

Bases: `StandardTableaux`, `DisjointUnionEnumeratedSets`

Standard tableaux of fixed size n.

EXAMPLES:

```
sage: [ t for t in StandardTableaux(1) ]
[[[1]]]
sage: [ t for t in StandardTableaux(2) ]
[[[1, 2]], [[1], [2]]]
sage: [ t for t in StandardTableaux(3) ]
[[[1, 2, 3]], [[1, 3], [2]], [[1, 2], [3]], [[1], [2], [3]]]
sage: StandardTableaux(4)[:]
[[[1, 2, 3, 4]],
 [[1, 3, 4], [2]],
 [[1, 2, 4], [3]],
 [[1, 2, 3], [4]],
 [[1, 3], [2, 4]],
 [[1, 2], [3, 4]],
 [[1, 4], [2], [3]],
 [[1, 3], [2], [4]],
 [[1, 2], [3], [4]],
 [[1], [2], [3], [4]]]
```

```
>>> from sage.all import *
>>> [ t for t in StandardTableaux(Integer(1)) ]
[[[1]]]
>>> [ t for t in StandardTableaux(Integer(2)) ]
[[[1, 2]], [[1], [2]]]
>>> [ t for t in StandardTableaux(Integer(3)) ]
[[[1, 2, 3]], [[1, 3], [2]], [[1, 2], [3]], [[1], [2], [3]]]
>>> StandardTableaux(Integer(4))[:]
[[[1, 2, 3, 4]],
 [[1, 3, 4], [2]],
 [[1, 2, 4], [3]],
 [[1, 2, 3], [4]],
 [[1, 3], [2, 4]],
 [[1, 2], [3, 4]],
 [[1, 4], [2], [3]],
 [[1, 3], [2], [4]],
 [[1, 2], [3], [4]],
 [[1], [2], [3], [4]]]
```

cardinality()

Return the number of all standard tableaux of size `n`.

The number of standard tableaux of size n is equal to the number of involutions in the symmetric group S_n. This is a consequence of the symmetry of the RSK correspondence, that if $\sigma \mapsto (P, Q)$, then $\sigma^{-1} \mapsto (Q, P)$. For more information, see Wikipedia article Robinson-Schensted-Knuth_correspondence#Symmetry.

ALGORITHM:

The algorithm uses the fact that standard tableaux of size `n` are in bijection with the involutions of size `n`, (see page 41 in section 4.1 of [Ful1997]). For each number of fixed points, you count the number of ways to choose those fixed points multiplied by the number of perfect matchings on the remaining values.

EXAMPLES:

```
sage: StandardTableaux(3).cardinality()
4
sage: ns = [1,2,3,4,5,6]
sage: sts = [StandardTableaux(n) for n in ns]
sage: all(st.cardinality() == len(st.list()) for st in sts)
True
```

```
>>> from sage.all import *
>>> StandardTableaux(Integer(3)).cardinality()
4
>>> ns = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6)]
>>> sts = [StandardTableaux(n) for n in ns]
>>> all(st.cardinality() == len(st.list()) for st in sts)
True
```

The cardinality can be computed without constructing all elements in this set, so this computation is fast (see also Issue #28273):

```
sage: StandardTableaux(500).cardinality()
423107565308608549951551753690...221285999236657443927937253376
```

```
>>> from sage.all import *
>>> StandardTableaux(Integer(500)).cardinality()
423107565308608549951551753690...221285999236657443927937253376
```

random_element()

Return a random StandardTableau with uniform probability.

This algorithm uses the fact that the Robinson-Schensted correspondence returns a pair of identical standard Young tableaux (SYTs) if and only if the permutation was an involution. Thus, generating a random SYT is equivalent to generating a random involution.

To generate an involution, we first need to choose its number of fixed points k (if the size of the involution is even, the number of fixed points will be even, and if the size is odd, the number of fixed points will be odd). To do this, we choose a random integer r between 0 and the number N of all involutions of size n. We then decompose the interval $\{1, 2, \ldots, N\}$ into subintervals whose lengths are the numbers of involutions of size n with respectively 0, 1, $\ldots$, $\lfloor N/2 \rfloor$ fixed points. The interval in which our random integer r lies then decides how many fixed points our random involution will have. We then place those fixed points randomly and then compute a perfect matching (an involution without fixed points) on the remaining values.

EXAMPLES:

```
sage: StandardTableaux(10).random_element() # random
[[1, 3, 6], [2, 5, 7], [4, 8], [9], [10]]
sage: StandardTableaux(0).random_element()
[]
sage: StandardTableaux(1).random_element()
[[1]]
```

```
>>> from sage.all import *
>>> StandardTableaux(Integer(10)).random_element() # random
[[1, 3, 6], [2, 5, 7], [4, 8], [9], [10]]
```

(continues on next page)

(continued from previous page)

```
>>> StandardTableaux(Integer(0)).random_element()
[]
>>> StandardTableaux(Integer(1)).random_element()
[[1]]
```

class sage.combinat.tableau.**Tableau**(*parent*, *t*, *check=True*)

Bases: ClonableList

A class to model a tableau.

INPUT:

- t – a Tableau, a list of iterables, or an empty list

OUTPUT: a Tableau object constructed from t

A tableau is abstractly a mapping from the cells in a partition to arbitrary objects (called entries). It is often represented as a finite list of nonempty lists (or, more generally an iterator of iterables) of weakly decreasing lengths. This list, in particular, can be empty, representing the empty tableau.

Note that Sage uses the English convention for partitions and tableaux; the longer rows are displayed on top.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty printing
1 2 3
4 5
sage: t.is_standard()
True

sage: Tableau([['a','c','b'],[],(2,1)]])
[['a', 'c', 'b'], [], (2, 1)]]
sage: Tableau([]) # The empty tableau
[]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]); t
[[1, 2, 3], [4, 5]]
>>> t.shape()
[3, 2]
>>> t.pp() # pretty printing
1 2 3
4 5
>>> t.is_standard()
True

>>> Tableau([['a','c','b'],[],(Integer(2),Integer(1))]])
[['a', 'c', 'b'], [], (2, 1)]]
>>> Tableau([]) # The empty tableau
[]
```

When using code that will generate a lot of tableaux, it is slightly more efficient to construct a Tableau from the appropriate Parent object:

```
sage: T = Tableaux()
sage: T([[1, 2, 3], [4, 5]])
[[1, 2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> T = Tableaux()
>>> T([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5)]])
[[1, 2, 3], [4, 5]]
```

See also

- *Tableaux*
- *SemistandardTableaux*
- *SemistandardTableau*
- *StandardTableaux*
- *StandardTableau*

add_entry(*cell*, *m*)

Return the result of setting the entry in cell `cell` equal to `m` in the tableau `self`.

This tableau has larger size than `self` if `cell` does not belong to the shape of `self`; otherwise, the tableau has the same shape as `self` and has the appropriate entry replaced.

INPUT:

- `cell` – a pair of nonnegative integers

OUTPUT:

The tableau `self` with the entry in cell `cell` set to `m`. This entry overwrites an existing entry if `cell` already belongs to `self`, or is added to the tableau if `cell` is a cocorner of the shape `self`. (Either way, the input is not modified.)

Note

Both coordinates of `cell` are interpreted as starting at 0. So, `cell == (0, 0)` corresponds to the northwesternmost cell.

EXAMPLES:

```
sage: s = StandardTableau([[1,2,5],[3,4]]); s.pp()
  1  2  5
  3  4
sage: t = s.add_entry( (1,2), 6); t.pp()
  1  2  5
  3  4  6
sage: t.category()
Category of elements of Standard tableaux
```

(continues on next page)

(continued from previous page)

```
sage: s.add_entry( (2,0), 6).pp()
  1  2  5
  3  4
  6
sage: u = s.add_entry( (1,2), 3); u.pp()
  1  2  5
  3  4  3
sage: u.category()
Category of elements of Tableaux
sage: s.add_entry( (2,2),3)
Traceback (most recent call last):
...
IndexError: (2, 2) is not an addable cell of the tableau
```

```
>>> from sage.all import *
>>> s = StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]); s.pp()
  1  2  5
  3  4
>>> t = s.add_entry( (Integer(1),Integer(2)), Integer(6)); t.pp()
  1  2  5
  3  4  6
>>> t.category()
Category of elements of Standard tableaux
>>> s.add_entry( (Integer(2),Integer(0)), Integer(6)).pp()
  1  2  5
  3  4
  6
>>> u = s.add_entry( (Integer(1),Integer(2)), Integer(3)); u.pp()
  1  2  5
  3  4  3
>>> u.category()
Category of elements of Tableaux
>>> s.add_entry( (Integer(2),Integer(2)),Integer(3))
Traceback (most recent call last):
...
IndexError: (2, 2) is not an addable cell of the tableau
```

anti_restrict(*n*)

Return the skew tableau formed by removing all of the cells from `self` that are filled with a number at most n.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
sage: t.anti_restrict(1)
[[None, 2, 3], [4, 5]]
sage: t.anti_restrict(2)
[[None, None, 3], [4, 5]]
sage: t.anti_restrict(3)
[[None, None, None], [4, 5]]
```

(continues on next page)

(continued from previous page)

```
sage: t.anti_restrict(4)
[[None, None, None], [None, 5]]
sage: t.anti_restrict(5)
[[None, None, None], [None, None]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]);
↪ t
[[1, 2, 3], [4, 5]]
>>> t.anti_restrict(Integer(1))
[[None, 2, 3], [4, 5]]
>>> t.anti_restrict(Integer(2))
[[None, None, 3], [4, 5]]
>>> t.anti_restrict(Integer(3))
[[None, None, None], [4, 5]]
>>> t.anti_restrict(Integer(4))
[[None, None, None], [None, 5]]
>>> t.anti_restrict(Integer(5))
[[None, None, None], [None, None]]
```

atom()

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).atom()
[2, 2]
sage: Tableau([[1,2,3],[4,5],[6]]).atom()
[3, 2, 1]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).atom()
[2, 2]
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)],
↪[Integer(6)]]).atom()
[3, 2, 1]
```

bender_knuth_involution(*k*, *rows=None*, *check=True*)

Return the image of `self` under the k-th Bender–Knuth involution, assuming `self` is a semistandard tableau.

Let T be a tableau, then a *lower free `k` in `T`* means a cell of T which is filled with the integer k and whose direct lower neighbor is not filled with the integer $k+1$ (in particular, this lower neighbor might not exist at all). Let an *upper free `k + 1` in `T`* mean a cell of T which is filled with the integer $k+1$ and whose direct upper neighbor is not filled with the integer k (in particular, this neighbor might not exist at all). It is clear that for any row r of T, the lower free k's and the upper free $k+1$'s in r together form a contiguous interval or r.

The *`k`-th Bender–Knuth switch at row `i`* changes the entries of the cells in this interval in such a way that if it used to have a entries of k and b entries of $k+1$, it will now have b entries of k and a entries of $k+1$. For fixed k, the k-th Bender–Knuth switches for different i commute. The composition of the k-th Bender–Knuth switches for all rows is called the *`k`-th Bender-Knuth involution*. This is used to show that the Schur functions defined by semistandard tableaux are symmetric functions.

INPUT:

- `k` – integer

- `rows` – (default: `None`) when set to `None`, the method computes the k-th Bender–Knuth involution as defined above. When an iterable, this computes the composition of the k-th Bender–Knuth switches at row i over all i in `rows`. When set to an integer i, the method computes the k-th Bender–Knuth switch at row i. Note the indexing of the rows starts with 1.
- `check` – (default: `True`) check to make sure `self` is semistandard; Set to `False` to avoid this check

OUTPUT:

The image of `self` under either the k-th Bender–Knuth involution, the k-th Bender–Knuth switch at a certain row, or the composition of such switches, as detailed in the INPUT section.

EXAMPLES:

```
sage: t = Tableau([[1,1,3,4,4,5,6,7],[2,2,4,6,7,7,7],[3,4,5,8,8,9],[6,6,7,10],
↪[7,8,8,11],[8]])
sage: t.bender_knuth_involution(1) == t
True
sage: t.bender_knuth_involution(2)
[[1, 1, 2, 4, 4, 5, 6, 7], [2, 3, 4, 6, 7, 7, 7], [3, 4, 5, 8, 8, 9], [6, 6,␣
↪7, 10], [7, 8, 8, 11], [8]]
sage: t.bender_knuth_involution(3)
[[1, 1, 3, 3, 3, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 4, 5, 8, 8, 9], [6, 6,␣
↪7, 10], [7, 8, 8, 11], [8]]
sage: t.bender_knuth_involution(4)
[[1, 1, 3, 4, 5, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 5, 5, 8, 8, 9], [6, 6,␣
↪7, 10], [7, 8, 8, 11], [8]]
sage: t.bender_knuth_involution(5)
[[1, 1, 3, 4, 4, 5, 6, 7], [2, 2, 4, 5, 7, 7, 7], [3, 4, 6, 8, 8, 9], [5, 5,␣
↪7, 10], [7, 8, 8, 11], [8]]
sage: t.bender_knuth_involution(666) == t
True
sage: t.bender_knuth_involution(4, 2) == t
True
sage: t.bender_knuth_involution(4, 3)
[[1, 1, 3, 4, 4, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 5, 5, 8, 8, 9], [6, 6,␣
↪7, 10], [7, 8, 8, 11], [8]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(3),Integer(4),Integer(4),
↪Integer(5),Integer(6),Integer(7)],[Integer(2),Integer(2),Integer(4),
↪Integer(6),Integer(7),Integer(7),Integer(7)],[Integer(3),Integer(4),
↪Integer(5),Integer(8),Integer(8),Integer(9)],[Integer(6),Integer(6),
↪Integer(7),Integer(10)],[Integer(7),Integer(8),Integer(8),Integer(11)],
↪[Integer(8)]])
>>> t.bender_knuth_involution(Integer(1)) == t
True
>>> t.bender_knuth_involution(Integer(2))
[[1, 1, 2, 4, 4, 5, 6, 7], [2, 3, 4, 6, 7, 7, 7], [3, 4, 5, 8, 8, 9], [6, 6,␣
↪7, 10], [7, 8, 8, 11], [8]]
>>> t.bender_knuth_involution(Integer(3))
[[1, 1, 3, 3, 3, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 4, 5, 8, 8, 9], [6, 6,␣
↪7, 10], [7, 8, 8, 11], [8]]
>>> t.bender_knuth_involution(Integer(4))
[[1, 1, 3, 4, 5, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 5, 5, 8, 8, 9], [6, 6,␣
```

(continues on next page)

(continued from previous page)

```
↪7, 10], [7, 8, 8, 11], [8]]
>>> t.bender_knuth_involution(Integer(5))
[[1, 1, 3, 4, 4, 5, 6, 7], [2, 2, 4, 5, 7, 7, 7], [3, 4, 6, 8, 8, 9], [5, 5,␣
↪7, 10], [7, 8, 8, 11], [8]]
>>> t.bender_knuth_involution(Integer(666)) == t
True
>>> t.bender_knuth_involution(Integer(4), Integer(2)) == t
True
>>> t.bender_knuth_involution(Integer(4), Integer(3))
[[1, 1, 3, 4, 4, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 5, 5, 8, 8, 9], [6, 6,␣
↪7, 10], [7, 8, 8, 11], [8]]
```

The `rows` keyword can be an iterator:

```
sage: t.bender_knuth_involution(6, iter([1,2])) == t
False
sage: t.bender_knuth_involution(6, iter([3,4])) == t
True
```

```
>>> from sage.all import *
>>> t.bender_knuth_involution(Integer(6), iter([Integer(1),Integer(2)])) == t
False
>>> t.bender_knuth_involution(Integer(6), iter([Integer(3),Integer(4)])) == t
True
```

The Bender–Knuth involution is an involution:

```
sage: T = SemistandardTableaux(shape=[3,1,1], max_entry=4)
sage: all(t.bender_knuth_involution(k).bender_knuth_involution(k) == t       #␣
↪needs sage.modules
....:     for k in range(1, 5) for t in T)
True
```

```
>>> from sage.all import *
>>> T = SemistandardTableaux(shape=[Integer(3),Integer(1),Integer(1)], max_
↪entry=Integer(4))
>>> all(t.bender_knuth_involution(k).bender_knuth_involution(k) == t       #␣
↪needs sage.modules
...     for k in range(Integer(1), Integer(5)) for t in T)
True
```

The same holds for the single switches:

```
sage: all(t.bender_knuth_involution(k, j).bender_knuth_involution(k, j) == t ␣
↪          # needs sage.modules
....:     for k in range(1, 5) for j in range(1, 5) for t in T)
True
```

```
>>> from sage.all import *
>>> all(t.bender_knuth_involution(k, j).bender_knuth_involution(k, j) == t   ␣
↪         # needs sage.modules
...     for k in range(Integer(1), Integer(5)) for j in range(Integer(1),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(5)) for t in T)
True
```

Locality of the Bender–Knuth involutions:

```
sage: all(t.bender_knuth_involution(k).bender_knuth_involution(l)              #␣
↪needs sage.modules
....:        == t.bender_knuth_involution(l).bender_knuth_involution(k)
....:     for k in range(1, 5) for l in range(1, 5) if abs(k - l) > 1
....:     for t in T)
True
```

```
>>> from sage.all import *
>>> all(t.bender_knuth_involution(k).bender_knuth_involution(l)              #␣
↪needs sage.modules
...        == t.bender_knuth_involution(l).bender_knuth_involution(k)
...     for k in range(Integer(1), Integer(5)) for l in range(Integer(1),␣
↪Integer(5)) if abs(k - l) > Integer(1)
...     for t in T)
True
```

Berenstein and Kirillov [KB1995] have shown that $(s_1 s_2)^6 = id$ (for tableaux of straight shape):

```
sage: p = lambda t, k: t.bender_knuth_involution(k).bender_knuth_involution(k␣
↪+ 1)
sage: all(p(p(p(p(p(p(t,1),1),1),1),1),1) == t for t in T)                     #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> p = lambda t, k: t.bender_knuth_involution(k).bender_knuth_involution(k +␣
↪Integer(1))
>>> all(p(p(p(p(p(p(t,Integer(1)),Integer(1)),Integer(1)),Integer(1)),
↪Integer(1)),Integer(1)) == t for t in T)                     # needs sage.
↪modules
True
```

However, $(s_2 s_3)^6 = id$ is false:

```
sage: p = lambda t, k: t.bender_knuth_involution(k).bender_knuth_involution(k␣
↪+ 1)
sage: t = Tableau([[1,2,2],[3,4]])
sage: x = t
sage: for i in range(6): x = p(x, 2)
sage: x
[[1, 2, 3], [2, 4]]
sage: x == t
False
```

```
>>> from sage.all import *
>>> p = lambda t, k: t.bender_knuth_involution(k).bender_knuth_involution(k +␣
↪Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> t = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(3),Integer(4)]])
>>> x = t
>>> for i in range(Integer(6)): x = p(x, Integer(2))
>>> x
[[1, 2, 3], [2, 4]]
>>> x == t
False
```

bump(*x*)

Insert `x` into `self` using Schensted's row-bumping (or row-insertion) algorithm.

EXAMPLES:

```
sage: t = Tableau([[1,2],[3]])
sage: t.bump(1)
[[1, 1], [2], [3]]
sage: t
[[1, 2], [3]]
sage: t.bump(2)
[[1, 2, 2], [3]]
sage: t.bump(3)
[[1, 2, 3], [3]]
sage: t
[[1, 2], [3]]
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t.bump(2)
[[1, 2, 2, 2], [2, 3, 3, 5], [4, 4, 5], [5, 6, 6]]
sage: t.bump(1)
[[1, 1, 2, 3], [2, 2, 5, 5], [3, 4, 6], [4, 6], [5]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(3)]])
>>> t.bump(Integer(1))
[[1, 1], [2], [3]]
>>> t
[[1, 2], [3]]
>>> t.bump(Integer(2))
[[1, 2, 2], [3]]
>>> t.bump(Integer(3))
[[1, 2, 3], [3]]
>>> t
[[1, 2], [3]]
>>> t = Tableau([[Integer(1),Integer(2),Integer(2),Integer(3)],[Integer(2),
↪Integer(3),Integer(5),Integer(5)],[Integer(4),Integer(4),Integer(6)],
↪[Integer(5),Integer(6)]])
>>> t.bump(Integer(2))
[[1, 2, 2, 2], [2, 3, 3, 5], [4, 4, 5], [5, 6, 6]]
>>> t.bump(Integer(1))
[[1, 1, 2, 3], [2, 2, 5, 5], [3, 4, 6], [4, 6], [5]]
```

bump_multiply(*other*)

Multiply two tableaux using Schensted's bump.

This product makes the set of semistandard tableaux into an associative monoid. The empty tableau is the unit in this monoid. See pp. 11-12 of [Ful1997].

The same product operation is implemented in a different way in *slide_multiply()*.

EXAMPLES:

```
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t2 = Tableau([[1,2],[3]])
sage: t.bump_multiply(t2)
[[1, 1, 2, 2, 3], [2, 2, 3, 5], [3, 4, 5], [4, 6, 6], [5]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(2),Integer(3)],[Integer(2),
→Integer(3),Integer(5),Integer(5)],[Integer(4),Integer(4),Integer(6)],
→[Integer(5),Integer(6)]])
>>> t2 = Tableau([[Integer(1),Integer(2)],[Integer(3)]])
>>> t.bump_multiply(t2)
[[1, 1, 2, 2, 3], [2, 2, 3, 5], [3, 4, 5], [4, 6, 6], [5]]
```

catabolism()

Remove the top row of `self` and insert it back in using column Schensted insertion (starting with the largest letter).

EXAMPLES:

```
sage: Tableau([]).catabolism()
[]
sage: Tableau([[1,2,3,4,5]]).catabolism()
[[1, 2, 3, 4, 5]]
sage: Tableau([[1,1,3,3],[2,3],[3]]).catabolism()
[[1, 1, 2, 3, 3, 3], [3]]
sage: Tableau([[1, 1, 2, 3, 3, 3], [3]]).catabolism()
[[1, 1, 2, 3, 3, 3, 3]]
```

```
>>> from sage.all import *
>>> Tableau([]).catabolism()
[]
>>> Tableau([[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]]).
→catabolism()
[[1, 2, 3, 4, 5]]
>>> Tableau([[Integer(1),Integer(1),Integer(3),Integer(3)],[Integer(2),
→Integer(3)],[Integer(3)]]).catabolism()
[[1, 1, 2, 3, 3, 3], [3]]
>>> Tableau([[Integer(1), Integer(1), Integer(2), Integer(3), Integer(3),␣
→Integer(3)], [Integer(3)]]).catabolism()
[[1, 1, 2, 3, 3, 3, 3]]
```

catabolism_projector(*parts*)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.catabolism_projector([[4,2,1]])
[[1, 1, 3, 3], [2, 3], [3]]
```

(continues on next page)

(continued from previous page)

```
sage: t.catabolism_projector([[1]])
[]
sage: t.catabolism_projector([[2,1],[1]])
[]
sage: t.catabolism_projector([[1,1],[4,1]])
[[1, 1, 3, 3], [2, 3], [3]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(3),Integer(3)],[Integer(2),
→Integer(3)],[Integer(3)]])
>>> t.catabolism_projector([[Integer(4),Integer(2),Integer(1)]])
[[1, 1, 3, 3], [2, 3], [3]]
>>> t.catabolism_projector([[Integer(1)]])
[]
>>> t.catabolism_projector([[Integer(2),Integer(1)],[Integer(1)]])
[]
>>> t.catabolism_projector([[Integer(1),Integer(1)],[Integer(4),Integer(1)]])
[[1, 1, 3, 3], [2, 3], [3]]
```

catabolism_sequence()

Perform *catabolism()* on `self` until it returns a tableau consisting of a single row.

EXAMPLES:

```
sage: t = Tableau([[1,2,3,4,5,6,8],[7,9]])
sage: t.catabolism_sequence()
[[[1, 2, 3, 4, 5, 6, 8], [7, 9]],
 [[1, 2, 3, 4, 5, 6, 7, 9], [8]],
 [[1, 2, 3, 4, 5, 6, 7, 8], [9]],
 [[1, 2, 3, 4, 5, 6, 7, 8, 9]]]
sage: Tableau([]).catabolism_sequence()
[[]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
→Integer(6),Integer(8)],[Integer(7),Integer(9)]])
>>> t.catabolism_sequence()
[[[1, 2, 3, 4, 5, 6, 8], [7, 9]],
 [[1, 2, 3, 4, 5, 6, 7, 9], [8]],
 [[1, 2, 3, 4, 5, 6, 7, 8], [9]],
 [[1, 2, 3, 4, 5, 6, 7, 8, 9]]]
>>> Tableau([]).catabolism_sequence()
[[]]
```

cells()

Return a list of the coordinates of the cells of `self`.

Coordinates start at 0, so the northwesternmost cell (in English notation) has coordinates $(0, 0)$.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).cells()
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).cells()
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

cells_containing(*i*)

Return the list of cells in which the letter i appears in the tableau `self`. The list is ordered with cells appearing from left to right.

Cells are given as pairs of coordinates (a, b), where both rows and columns are counted from 0 (so $a = 0$ means the cell lies in the leftmost column of the tableau, etc.).

EXAMPLES:

```
sage: t = Tableau([[1,1,3],[2,3,5],[4,5]])
sage: t.cells_containing(5)
[(2, 1), (1, 2)]
sage: t.cells_containing(4)
[(2, 0)]
sage: t.cells_containing(6)
[]

sage: t = Tableau([[1,1,2,4],[2,4,4],[4]])
sage: t.cells_containing(4)
[(2, 0), (1, 1), (1, 2), (0, 3)]

sage: t = Tableau([[1,1,2,8,9],[2,5,6,11],[3,7,7,13],[4,8,9],[5],[13],[14]])
sage: t.cells_containing(8)
[(3, 1), (0, 3)]

sage: Tableau([]).cells_containing(3)
[]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(3)],[Integer(2),Integer(3),
↪Integer(5)],[Integer(4),Integer(5)]])
>>> t.cells_containing(Integer(5))
[(2, 1), (1, 2)]
>>> t.cells_containing(Integer(4))
[(2, 0)]
>>> t.cells_containing(Integer(6))
[]

>>> t = Tableau([[Integer(1),Integer(1),Integer(2),Integer(4)],[Integer(2),
↪Integer(4),Integer(4)],[Integer(4)]])
>>> t.cells_containing(Integer(4))
[(2, 0), (1, 1), (1, 2), (0, 3)]

>>> t = Tableau([[Integer(1),Integer(1),Integer(2),Integer(8),Integer(9)],
↪[Integer(2),Integer(5),Integer(6),Integer(11)],[Integer(3),Integer(7),
↪Integer(7),Integer(13)],[Integer(4),Integer(8),Integer(9)],[Integer(5)],
↪[Integer(13)],[Integer(14)]])
>>> t.cells_containing(Integer(8))
[(3, 1), (0, 3)]
```

(continues on next page)

(continued from previous page)

```
>>> Tableau([]).cells_containing(Integer(3))
[]
```

charge()

Return the charge of the reading word of `self`. See `charge()` for more information.

EXAMPLES:

```
sage: Tableau([[1,1],[2,2],[3]]).charge()
0
sage: Tableau([[1,1,3],[2,2]]).charge()
1
sage: Tableau([[1,1,2],[2],[3]]).charge()
1
sage: Tableau([[1,1,2],[2,3]]).charge()
2
sage: Tableau([[1,1,2,3],[2]]).charge()
2
sage: Tableau([[1,1,2,2],[3]]).charge()
3
sage: Tableau([[1,1,2,2,3]]).charge()
4
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(1)],[Integer(2),Integer(2)],[Integer(3)]]).
↪charge()
0
>>> Tableau([[Integer(1),Integer(1),Integer(3)],[Integer(2),Integer(2)]]).
↪charge()
1
>>> Tableau([[Integer(1),Integer(1),Integer(2)],[Integer(2)],[Integer(3)]]).
↪charge()
1
>>> Tableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),Integer(3)]]).
↪charge()
2
>>> Tableau([[Integer(1),Integer(1),Integer(2),Integer(3)],[Integer(2)]]).
↪charge()
2
>>> Tableau([[Integer(1),Integer(1),Integer(2),Integer(2)],[Integer(3)]]).
↪charge()
3
>>> Tableau([[Integer(1),Integer(1),Integer(2),Integer(2),Integer(3)]]).
↪charge()
4
```

check()

Check that `self` is a valid straight-shape tableau.

EXAMPLES:

```
sage: t = Tableau([[1,1],[2]])
sage: t.check()

sage: t = Tableau([[None, None, 1], [2, 4], [3, 4, 5]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: a tableau must be a list of iterables of weakly decreasing length
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1)],[Integer(2)]])
>>> t.check()

>>> t = Tableau([[None, None, Integer(1)], [Integer(2), Integer(4)],␣
↪[Integer(3), Integer(4), Integer(5)]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: a tableau must be a list of iterables of weakly decreasing length
```

cocharge()

Return the cocharge of the reading word of `self`. See *cocharge()* for more information.

EXAMPLES:

```
sage: Tableau([[1,1],[2,2],[3]]).cocharge()
4
sage: Tableau([[1,1,3],[2,2]]).cocharge()
3
sage: Tableau([[1,1,2],[2],[3]]).cocharge()
3
sage: Tableau([[1,1,2],[2,3]]).cocharge()
2
sage: Tableau([[1,1,2,3],[2]]).cocharge()
2
sage: Tableau([[1,1,2,2],[3]]).cocharge()
1
sage: Tableau([[1,1,2,2,3]]).cocharge()
0
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(1)],[Integer(2),Integer(2)],[Integer(3)]]).
↪cocharge()
4
>>> Tableau([[Integer(1),Integer(1),Integer(3)],[Integer(2),Integer(2)]]).
↪cocharge()
3
>>> Tableau([[Integer(1),Integer(1),Integer(2)],[Integer(2)],[Integer(3)]]).
↪cocharge()
3
>>> Tableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),Integer(3)]]).
↪cocharge()
2
>>> Tableau([[Integer(1),Integer(1),Integer(2),Integer(3)],[Integer(2)]]).
↪cocharge()
```

(continues on next page)

(continued from previous page)

```
2
>>> Tableau([[Integer(1),Integer(1),Integer(2),Integer(2)],[Integer(3)]]).
↪cocharge()
1
>>> Tableau([[Integer(1),Integer(1),Integer(2),Integer(2),Integer(3)]]).
↪cocharge()
0
```

codegree(*e*, *multicharge=(0,)*)

Return the Brundan-Kleshchev-Wang [BKW2011] codegree of the standard tableau `self`.

The *codegree* of a tableau is an integer that is defined recursively by successively stripping off the number k, for $k = n, n-1, \ldots, 1$ and at stage adding the number of addable cell of the same residue minus the number of removable cells of the same residue as k and are above k in the diagram.

The codegree of the tableau T gives the degree of "dual" homogeneous basis element of the Graded Specht module that is indexed by T.

INPUT:

- `e` – the *quantum characteristic*
- `multicharge` – (default: `[0]`) the multicharge

OUTPUT: the codegree of the tableau `self`, which is an integer

EXAMPLES:

```
sage: StandardTableau([[1,3,5],[2,4]]).codegree(3)                          #␣
↪needs sage.groups
0
sage: StandardTableau([[1,2,5],[3,4]]).codegree(3)                          #␣
↪needs sage.groups
1
sage: StandardTableau([[1,2,5],[3,4]]).codegree(4)                          #␣
↪needs sage.groups
0
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(3),Integer(5)],[Integer(2),
↪Integer(4)]]).codegree(Integer(3))                          # needs sage.
↪groups
0
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).codegree(Integer(3))                          # needs sage.
↪groups
1
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).codegree(Integer(4))                          # needs sage.
↪groups
0
```

column_stabilizer()

Return the PermutationGroup corresponding to the column stabilizer of `self`.

This assumes that every integer from 1 to the size of `self` appears exactly once in `self`.

EXAMPLES:

```
sage: # needs sage.groups
sage: cs = Tableau([[1,2,3],[4,5]]).column_stabilizer()
sage: cs.order() == factorial(2)*factorial(2)
True
sage: PermutationGroupElement([(1,3,2),(4,5)]) in cs
False
sage: PermutationGroupElement([(1,4)]) in cs
True
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> cs = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]]).column_stabilizer()
>>> cs.order() == factorial(Integer(2))*factorial(Integer(2))
True
>>> PermutationGroupElement([(Integer(1),Integer(3),Integer(2)),(Integer(4),
↪Integer(5))]) in cs
False
>>> PermutationGroupElement([(Integer(1),Integer(4))]) in cs
True
```

components()

This function returns a list containing itself. It exists mainly for compatibility with *TableauTuple* as it allows constructions like the example below.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[4,5]])
sage: for s in t.components(): print(s.to_list())
[[1, 2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])
>>> for s in t.components(): print(s.to_list())
[[1, 2, 3], [4, 5]]
```

conjugate()

Return the conjugate of `self`.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).conjugate()
[[1, 3], [2, 4]]
sage: c = StandardTableau([[1,2],[3,4]]).conjugate()
sage: c.parent()
Standard tableaux
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).conjugate()
[[1, 3], [2, 4]]
>>> c = StandardTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).
```

(continues on next page)

(continued from previous page)

```
↪conjugate()
>>> c.parent()
Standard tableaux
```

content (*k*, *multicharge=[0]*)

Return the content of `k` in the standard tableau `self`.

The content of k is $c - r$ if k appears in row r and column c of the tableau.

The `multicharge` is a list of length 1 which gives an offset for all of the contents. It is included mainly for compatibility with *sage.combinat.tableau_tuple.TableauTuple()*.

EXAMPLES:

```
sage: StandardTableau([[1,2],[3,4]]).content(3)
-1

sage: StandardTableau([[1,2],[3,4]]).content(6)
Traceback (most recent call last):
...
ValueError: 6 does not appear in tableau
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).
↪content(Integer(3))
-1

>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).
↪content(Integer(6))
Traceback (most recent call last):
...
ValueError: 6 does not appear in tableau
```

corners ()

Return the corners of the tableau `self`.

EXAMPLES:

```
sage: Tableau([[1, 4, 6], [2, 5], [3]]).corners()
[(0, 2), (1, 1), (2, 0)]
sage: Tableau([[1, 3], [2, 4]]).corners()
[(1, 1)]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1), Integer(4), Integer(6)], [Integer(2), Integer(5)],␣
↪[Integer(3)]]).corners()
[(0, 2), (1, 1), (2, 0)]
>>> Tableau([[Integer(1), Integer(3)], [Integer(2), Integer(4)]]).corners()
[(1, 1)]
```

degree (*e*, *multicharge=(0,)*)

Return the Brundan-Kleshchev-Wang [BKW2011] degree of `self`.

The *degree* is an integer that is defined recursively by successively stripping off the number k, for $k = n, n -$

$1, \ldots, 1$ and at stage adding the number of addable cell of the same residue minus the number of removable cells of the same residue as k and which are below k in the diagram.

The degrees of the tableau T gives the degree of the homogeneous basis element of the graded Specht module that is indexed by T.

INPUT:

- `e` – the *quantum characteristic*
- `multicharge` – (default: `[0]`) the multicharge

OUTPUT: the degree of the tableau `self`, which is an integer

EXAMPLES:

```
sage: StandardTableau([[1,2,5],[3,4]]).degree(3)                              #␣
↪needs sage.groups
0
sage: StandardTableau([[1,2,5],[3,4]]).degree(4)                              #␣
↪needs sage.groups
1
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).degree(Integer(3))                              # needs sage.
↪groups
0
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).degree(Integer(4))                              # needs sage.
↪groups
1
```

descents()

Return a list of the cells `(i,j)` such that `self[i][j] > self[i-1][j]`.

> **Warning**
>
> This is not to be confused with the descents of a standard tableau.

EXAMPLES:

```
sage: Tableau( [[1,4],[2,3]] ).descents()
[(1, 0)]
sage: Tableau( [[1,2],[3,4]] ).descents()
[(1, 0), (1, 1)]
sage: Tableau( [[1,2,3],[4,5]] ).descents()
[(1, 0), (1, 1)]
```

```
>>> from sage.all import *
>>> Tableau( [[Integer(1),Integer(4)],[Integer(2),Integer(3)]] ).descents()
[(1, 0)]
>>> Tableau( [[Integer(1),Integer(2)],[Integer(3),Integer(4)]] ).descents()
[(1, 0), (1, 1)]
>>> Tableau( [[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]] ).
```

(continues on next page)

(continued from previous page)

```
↪descents()
[(1, 0), (1, 1)]
```

entries()

Return the tuple of all entries of `self`, in the order obtained by reading across the rows from top to bottom (in English notation).

EXAMPLES:

```
sage: t = Tableau([[1,3], [2]])
sage: t.entries()
(1, 3, 2)
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(3)], [Integer(2)]])
>>> t.entries()
(1, 3, 2)
```

entry(*cell*)

Return the entry of cell `cell` in the tableau `self`. Here, `cell` should be given as a tuple (i, j) of zero-based coordinates (so the northwesternmost cell in English notation is $(0, 0)$).

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,4]])
sage: t.entry( (0,0) )
1
sage: t.entry( (1,1) )
4
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
>>> t.entry( (Integer(0),Integer(0)) )
1
>>> t.entry( (Integer(1),Integer(1)) )
4
```

evacuation(*n=None*, *check=True*)

Return the evacuation of the tableau `self`.

This is an alias for *schuetzenberger_involution()*.

This method relies on the analogous method on words, which reverts the word and then complements all letters within the underlying ordered alphabet. If n is specified, the underlying alphabet is assumed to be $[1, 2, \ldots, n]$. If no alphabet is specified, n is the maximal letter appearing in `self`.

INPUT:

- `n` – integer specifying the maximal letter in the alphabet (optional)
- `check` – boolean (default: `True`); check to make sure `self` is semistandard

OUTPUT: a tableau, the evacuation of `self`

EXAMPLES:

```
sage: t = Tableau([[1,1,1],[2,2]])
sage: t.evacuation(3)
[[2, 2, 3], [3, 3]]

sage: t = Tableau([[1,2,3],[4,5]])
sage: t.evacuation()
[[1, 2, 5], [3, 4]]

sage: t = Tableau([[1,3,5,7],[2,4,6],[8,9]])
sage: t.evacuation()
[[1, 2, 6, 8], [3, 4, 9], [5, 7]]

sage: t = Tableau([])
sage: t.evacuation()
[]

sage: t = StandardTableau([[1,2,3],[4,5]])
sage: s = t.evacuation()
sage: s.parent()
Standard tableaux
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(2)]])
>>> t.evacuation(Integer(3))
[[2, 2, 3], [3, 3]]

>>> t = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])
>>> t.evacuation()
[[1, 2, 5], [3, 4]]

>>> t = Tableau([[Integer(1),Integer(3),Integer(5),Integer(7)],[Integer(2),
↪Integer(4),Integer(6)],[Integer(8),Integer(9)]])
>>> t.evacuation()
[[1, 2, 6, 8], [3, 4, 9], [5, 7]]

>>> t = Tableau([])
>>> t.evacuation()
[]

>>> t = StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]])
>>> s = t.evacuation()
>>> s.parent()
Standard tableaux
```

evaluation()

Return the weight of the tableau `self`. Trailing zeroes are omitted when returning the weight.

The weight of a tableau T is the sequence $(a_1, a_2, a_3, \ldots)$, where a_k is the number of entries of T equal to k. This sequence contains only finitely many nonzero entries.

The weight of a tableau T is the same as the weight of the reading word of T, for any reading order.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).weight()
[1, 1, 1, 1]

sage: Tableau([]).weight()
[]

sage: Tableau([[1,3,3,7],[4,2],[2,3]]).weight()
[1, 2, 3, 1, 0, 0, 1]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).weight()
[1, 1, 1, 1]

>>> Tableau([]).weight()
[]

>>> Tableau([[Integer(1),Integer(3),Integer(3),Integer(7)],[Integer(4),
↪Integer(2)],[Integer(2),Integer(3)]]).weight()
[1, 2, 3, 1, 0, 0, 1]
```

first_column_descent()

Return the first cell where `self` is not column standard.

Cells are ordered left to right along the rows and then top to bottom. That is, the cell (r, c) with r and c minimal such that the entry in position (r, c) is bigger than the entry in position $(r, c + 1)$. If there is no such cell then `None` is returned - in this case the tableau is column strict.

OUTPUT:

The first cell which there is a descent or `None` if no such cell exists.

EXAMPLES:

```
sage: Tableau([[1,4,5],[2,3]]).first_column_descent()
(0, 1)
sage: Tableau([[1,2,3],[4]]).first_column_descent() is None
True
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(4),Integer(5)],[Integer(2),Integer(3)]]).
↪first_column_descent()
(0, 1)
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4)]]).first_column_
↪descent() is None
True
```

first_row_descent()

Return the first cell where the tableau `self` is not row standard.

Cells are ordered left to right along the rows and then top to bottom. That is, the cell (r, c) with r and c minimal such that the entry in position (r, c) is bigger than the entry in position $(r, c + 1)$. If there is no such cell then `None` is returned - in this case the tableau is row strict.

OUTPUT:

The first cell which there is a descent or `None` if no such cell exists.

EXAMPLES:

```
sage: t = Tableau([[1,3,2],[4]]); t.first_row_descent()
(0, 1)
sage: Tableau([[1,2,3],[4]]).first_row_descent() is  None
True
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(3),Integer(2)],[Integer(4)]]); t.first_
↪row_descent()
(0, 1)
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4)]]).first_row_
↪descent() is  None
True
```

flush()

Return the number of flush segments in `self`, as in [Sal2014].

Let $1 \leq i < k \leq r+1$ and suppose ℓ is the smallest integer greater than k such that there exists an ℓ-segment in the $(i+1)$-st row of T. A k-segment in the i-th row of T is called *flush* if the leftmost box in the k-segment and the leftmost box of the ℓ-segment are in the same column of T. If, however, no such ℓ exists, then this k-segment is said to be *flush* if the number of boxes in the k-segment is equal to θ_i, where $\theta_i = \lambda_i - \lambda_{i+1}$ and the shape of T is $\lambda = (\lambda_1 > \lambda_2 > \cdots > \lambda_r)$. Denote the number of flush k-segments in T by $\mathrm{flush}(T)$.

EXAMPLES:

```
sage: t = Tableau([[1,1,2,3,5],[2,3,5,5],[3,4]])
sage: t.flush()
3

sage: B = crystals.Tableaux("A4", shape=[4,3,2,1])                          #␣
↪needs sage.modules
sage: t = B[32].to_tableau()                                                 #␣
↪needs sage.modules
sage: t.flush()                                                              #␣
↪needs sage.modules
4
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(2),Integer(3),Integer(5)],
↪[Integer(2),Integer(3),Integer(5),Integer(5)],[Integer(3),Integer(4)]])
>>> t.flush()
3

>>> B = crystals.Tableaux("A4", shape=[Integer(4),Integer(3),Integer(2),
↪Integer(1)])                             # needs sage.modules
>>> t = B[Integer(32)].to_tableau()                                          ␣
↪       # needs sage.modules
>>> t.flush()                                                              #␣
↪needs sage.modules
4
```

height()

Return the height of `self`.

EXAMPLES:

```
sage: Tableau([[1,2,3],[4,5]]).height()
2
sage: Tableau([[1,2,3]]).height()
1
sage: Tableau([]).height()
0
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]).
↪height()
2
>>> Tableau([[Integer(1),Integer(2),Integer(3)]]).height()
1
>>> Tableau([]).height()
0
```

hillman_grassl()

Return the image of the λ-array `self` under the Hillman-Grassl correspondence (as a *WeakReversePlanePartition*).

This relies on interpreting `self` as a λ-array in the sense of *hillman_grassl*.

Fix a partition λ (see *Partition()*). We draw all partitions and tableaux in English notation.

A λ-*array* will mean a tableau of shape λ whose entries are nonnegative integers. (No conditions on the order of these entries are made. Note that 0 is allowed.)

A *weak reverse plane partition of shape* λ (short: λ-*rpp*) will mean a λ-array whose entries weakly increase along each row and weakly increase along each column.

The Hillman-Grassl correspondence H is the map that sends a λ-array M to a λ-rpp $H(M)$ defined recursively as follows:

- If all entries of M are 0, then $H(M) = M$.
- Otherwise, let s be the index of the leftmost column of M containing a nonzero entry. Let r be the index of the bottommost nonzero entry in the s-th column of M. Let M' be the λ-array obtained from M by subtracting 1 from the (r, s)-th entry of M. Let $Q = (q_{i,j})$ be the image $H(M')$ (which is already defined by recursion).
- Define a sequence $((i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n))$ of boxes in the diagram of λ (actually a lattice path made of southward and westward steps) as follows: Set $(i_1, j_1) = (r, \lambda_r)$ (the rightmost box in the r-th row of λ). If (i_k, j_k) is defined for some $k \geq 1$, then (i_{k+1}, j_{k+1}) is constructed as follows: If q_{i_k+1,j_k} is well-defined and equals q_{i_k,j_k}, then we set $(i_{k+1}, j_{k+1}) = (i_k + 1, j_k)$. Otherwise, if $j_k = s$, then the sequence ends here. Otherwise, we set $(i_{k+1}, j_{k+1}) = (i_k, j_k - 1)$.
- Let $H(M)$ be the array obtained from Q by adding 1 to the (i_k, j_k)-th entry of Q for each $k \in \{1, 2, \ldots, n\}$.

See [Gans1981] (Section 3) for this construction.

See also

hillman_grassl() for the Hillman-Grassl correspondence as a standalone function.

hillman_grassl_inverse() for the inverse map.

EXAMPLES:

```
sage: a = Tableau([[2, 1, 1], [0, 2, 0], [1, 1]])
sage: A = a.hillman_grassl(); A
[[2, 2, 4], [2, 3, 4], [3, 5]]
sage: A.parent(), a.parent()
(Weak Reverse Plane Partitions, Tableaux)
```

```
>>> from sage.all import *
>>> a = Tableau([[Integer(2), Integer(1), Integer(1)], [Integer(0),␣
↪Integer(2), Integer(0)], [Integer(1), Integer(1)]])
>>> A = a.hillman_grassl(); A
[[2, 2, 4], [2, 3, 4], [3, 5]]
>>> A.parent(), a.parent()
(Weak Reverse Plane Partitions, Tableaux)
```

insert_word(*w*, *left=False*)

Insert the word `w` into the tableau `self` letter by letter using Schensted insertion. By default, the word `w` is being processed from left to right, and the insertion used is row insertion. If the optional keyword `left` is set to `True`, the word `w` is being processed from right to left, and column insertion is used instead.

EXAMPLES:

```
sage: t0 = Tableau([])
sage: w = [1,1,2,3,3,3,3]
sage: t0.insert_word(w)
[[1, 1, 2, 3, 3, 3, 3]]
sage: t0.insert_word(w,left=True)
[[1, 1, 2, 3, 3, 3, 3]]
sage: w.reverse()
sage: t0.insert_word(w)
[[1, 1, 3, 3], [2, 3], [3]]
sage: t0.insert_word(w,left=True)
[[1, 1, 3, 3], [2, 3], [3]]
sage: t1 = Tableau([[1,3],[2]])
sage: t1.insert_word([4,5])
[[1, 3, 4, 5], [2]]
sage: t1.insert_word([4,5], left=True)
[[1, 3], [2, 5], [4]]
```

```
>>> from sage.all import *
>>> t0 = Tableau([])
>>> w = [Integer(1),Integer(1),Integer(2),Integer(3),Integer(3),Integer(3),
↪Integer(3)]
>>> t0.insert_word(w)
[[1, 1, 2, 3, 3, 3, 3]]
>>> t0.insert_word(w,left=True)
[[1, 1, 2, 3, 3, 3, 3]]
>>> w.reverse()
>>> t0.insert_word(w)
[[1, 1, 3, 3], [2, 3], [3]]
>>> t0.insert_word(w,left=True)
[[1, 1, 3, 3], [2, 3], [3]]
```

(continues on next page)

(continued from previous page)

```
>>> t1 = Tableau([[Integer(1),Integer(3)],[Integer(2)]])
>>> t1.insert_word([Integer(4),Integer(5)])
[[1, 3, 4, 5], [2]]
>>> t1.insert_word([Integer(4),Integer(5)], left=True)
[[1, 3], [2, 5], [4]]
```

inversion_number()

Return the inversion number of `self`.

The inversion number is defined to be the number of inversions of `self` minus the sum of the arm lengths of the descents of `self` (see the *inversions()* and *descents()* methods for the relevant definitions).

> **Warning**
>
> This has none of the meanings in which the word "inversion" is used in the theory of standard tableaux.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,5]])
sage: t.inversion_number()
0
sage: t = Tableau([[1,2,4],[3,5]])
sage: t.inversion_number()
0
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(5)]])
>>> t.inversion_number()
0
>>> t = Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3),Integer(5)]])
>>> t.inversion_number()
0
```

inversions()

Return a list of the inversions of `self`.

Let T be a tableau. An inversion is an attacking pair (c, d) of the shape of T (see *attacking_pairs()* for a definition of this) such that the entry of c in T is greater than the entry of d.

> **Warning**
>
> Do not mistake this for the inversions of a standard tableau.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,5]])
sage: t.inversions()
[((1, 1), (0, 0))]
sage: t = Tableau([[1,4,3],[5,2],[2,6],[3]])
sage: t.inversions()
[((0, 1), (0, 2)), ((1, 0), (1, 1)), ((1, 1), (0, 0)), ((2, 1), (1, 0))]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(5)]])
>>> t.inversions()
[((1, 1), (0, 0))]
>>> t = Tableau([[Integer(1),Integer(4),Integer(3)],[Integer(5),Integer(2)],
↪[Integer(2),Integer(6)],[Integer(3)]])
>>> t.inversions()
[((0, 1), (0, 2)), ((1, 0), (1, 1)), ((1, 1), (0, 0)), ((2, 1), (1, 0))]
```

is_column_increasing(*weak=False*)

Return `True` if the entries in each column are in increasing order, and `False` otherwise.

By default, this checks for strictly increasing columns. Set `weak` to `True` to test for weakly increasing columns.

EXAMPLES:

```
sage: T = Tableau([[1, 1, 3], [1, 2]])
sage: T.is_column_increasing(weak=True)
True
sage: T.is_column_increasing()
False
sage: Tableau([[2], [1]]).is_column_increasing(weak=True)
False
```

```
>>> from sage.all import *
>>> T = Tableau([[Integer(1), Integer(1), Integer(3)], [Integer(1),␣
↪Integer(2)]])
>>> T.is_column_increasing(weak=True)
True
>>> T.is_column_increasing()
False
>>> Tableau([[Integer(2)], [Integer(1)]]).is_column_increasing(weak=True)
False
```

is_column_strict()

Return `True` if `self` is a column strict tableau and `False` otherwise.

A tableau is column strict if the entries in each column are in (strictly) increasing order.

EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_column_strict()
True
sage: Tableau([[1, 2], [2, 4]]).is_column_strict()
True
sage: Tableau([[2, 3], [2, 4]]).is_column_strict()
False
sage: Tableau([[5, 3], [2, 4]]).is_column_strict()
False
sage: Tableau([]).is_column_strict()
True
sage: Tableau([[1, 4, 2]]).is_column_strict()
True
sage: Tableau([[1, 4, 2], [2, 5]]).is_column_strict()
```

(continues on next page)

(continued from previous page)

```
True
sage: Tableau([[1, 4, 2], [2, 3]]).is_column_strict()
False
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1), Integer(3)], [Integer(2), Integer(4)]]).is_column_
↪strict()
True
>>> Tableau([[Integer(1), Integer(2)], [Integer(2), Integer(4)]]).is_column_
↪strict()
True
>>> Tableau([[Integer(2), Integer(3)], [Integer(2), Integer(4)]]).is_column_
↪strict()
False
>>> Tableau([[Integer(5), Integer(3)], [Integer(2), Integer(4)]]).is_column_
↪strict()
False
>>> Tableau([]).is_column_strict()
True
>>> Tableau([[Integer(1), Integer(4), Integer(2)]]).is_column_strict()
True
>>> Tableau([[Integer(1), Integer(4), Integer(2)], [Integer(2), Integer(5)]]).
↪is_column_strict()
True
>>> Tableau([[Integer(1), Integer(4), Integer(2)], [Integer(2), Integer(3)]]).
↪is_column_strict()
False
```

is_increasing()

Return `True` if `self` is an increasing tableau and `False` otherwise.

A tableau is increasing if it is both row strict and column strict.

EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_increasing()
True
sage: Tableau([[1, 2], [2, 4]]).is_increasing()
True
sage: Tableau([[2, 3], [2, 4]]).is_increasing()
False
sage: Tableau([[5, 3], [2, 4]]).is_increasing()
False
sage: Tableau([[1, 2, 3], [2, 3], [3]]).is_increasing()
True
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1), Integer(3)], [Integer(2), Integer(4)]]).is_
↪increasing()
True
>>> Tableau([[Integer(1), Integer(2)], [Integer(2), Integer(4)]]).is_
↪increasing()
```

(continues on next page)

(continued from previous page)

```
True
>>> Tableau([[Integer(2), Integer(3)], [Integer(2), Integer(4)]]).is_
↪increasing()
False
>>> Tableau([[Integer(5), Integer(3)], [Integer(2), Integer(4)]]).is_
↪increasing()
False
>>> Tableau([[Integer(1), Integer(2), Integer(3)], [Integer(2), Integer(3)],␣
↪[Integer(3)]]).is_increasing()
True
```

is_k_tableau(*k*)

Check whether `self` is a valid weak k-tableau.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,3],[3]])
sage: t.is_k_tableau(3)
True
sage: t = Tableau([[1,1,3],[2,2],[3]])
sage: t.is_k_tableau(3)
False
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(3)],
↪[Integer(3)]])
>>> t.is_k_tableau(Integer(3))
True
>>> t = Tableau([[Integer(1),Integer(1),Integer(3)],[Integer(2),Integer(2)],
↪[Integer(3)]])
>>> t.is_k_tableau(Integer(3))
False
```

is_key_tableau()

Return `True` if `self` is a key tableau or `False` otherwise.

A tableau is a *key tableau* if the set of entries in the j-th column is a subset of the set of entries in the $(j-1)$-st column.

REFERENCES:

- [LS1990]
- [Wil2010]

EXAMPLES:

```
sage: t = Tableau([[1,1,1],[2,3],[3]])
sage: t.is_key_tableau()
True

sage: t = Tableau([[1,1,2],[2,3],[3]])
sage: t.is_key_tableau()
False
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(3)],
→[Integer(3)]])
>>> t.is_key_tableau()
True

>>> t = Tableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),Integer(3)],
→[Integer(3)]])
>>> t.is_key_tableau()
False
```

is_rectangular()

Return True if the tableau self is rectangular and False otherwise.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).is_rectangular()
True
sage: Tableau([[1,2,3],[4,5],[6]]).is_rectangular()
False
sage: Tableau([]).is_rectangular()
True
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).is_
→rectangular()
True
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)],
→[Integer(6)]]).is_rectangular()
False
>>> Tableau([]).is_rectangular()
True
```

is_row_increasing(*weak=False*)

Return True if the entries in each row are in increasing order, and False otherwise.

By default, this checks for strictly increasing rows. Set weak to True to test for weakly increasing rows.

EXAMPLES:

```
sage: T = Tableau([[1, 1, 3], [1, 2]])
sage: T.is_row_increasing(weak=True)
True
sage: T.is_row_increasing()
False
sage: Tableau([[2, 1]]).is_row_increasing(weak=True)
False
```

```
>>> from sage.all import *
>>> T = Tableau([[Integer(1), Integer(1), Integer(3)], [Integer(1),␣
→Integer(2)]])
>>> T.is_row_increasing(weak=True)
True
>>> T.is_row_increasing()
```

(continues on next page)

(continued from previous page)

```
False
>>> Tableau([[Integer(2), Integer(1)]]).is_row_increasing(weak=True)
False
```

is_row_strict()

Return `True` if `self` is a row strict tableau and `False` otherwise.

A tableau is row strict if the entries in each row are in (strictly) increasing order.

EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_row_strict()
True
sage: Tableau([[1, 2], [2, 4]]).is_row_strict()
True
sage: Tableau([[2, 3], [2, 4]]).is_row_strict()
True
sage: Tableau([[5, 3], [2, 4]]).is_row_strict()
False
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1), Integer(3)], [Integer(2), Integer(4)]]).is_row_
↪strict()
True
>>> Tableau([[Integer(1), Integer(2)], [Integer(2), Integer(4)]]).is_row_
↪strict()
True
>>> Tableau([[Integer(2), Integer(3)], [Integer(2), Integer(4)]]).is_row_
↪strict()
True
>>> Tableau([[Integer(5), Integer(3)], [Integer(2), Integer(4)]]).is_row_
↪strict()
False
```

is_semistandard()

Return `True` if `self` is a semistandard tableau, and `False` otherwise.

A tableau is semistandard if its rows weakly increase and its columns strictly increase.

EXAMPLES:

```
sage: Tableau([[1,1],[1,2]]).is_semistandard()
False
sage: Tableau([[1,2],[1,2]]).is_semistandard()
False
sage: Tableau([[1,1],[2,2]]).is_semistandard()
True
sage: Tableau([[1,2],[2,3]]).is_semistandard()
True
sage: Tableau([[4,1],[3,2]]).is_semistandard()
False
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(1)],[Integer(1),Integer(2)]]).is_
↪semistandard()
False
>>> Tableau([[Integer(1),Integer(2)],[Integer(1),Integer(2)]]).is_
↪semistandard()
False
>>> Tableau([[Integer(1),Integer(1)],[Integer(2),Integer(2)]]).is_
↪semistandard()
True
>>> Tableau([[Integer(1),Integer(2)],[Integer(2),Integer(3)]]).is_
↪semistandard()
True
>>> Tableau([[Integer(4),Integer(1)],[Integer(3),Integer(2)]]).is_
↪semistandard()
False
```

is_standard()

Return `True` if `self` is a standard tableau and `False` otherwise.

EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_standard()
True
sage: Tableau([[1, 2], [2, 4]]).is_standard()
False
sage: Tableau([[2, 3], [2, 4]]).is_standard()
False
sage: Tableau([[5, 3], [2, 4]]).is_standard()
False
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1), Integer(3)], [Integer(2), Integer(4)]]).is_
↪standard()
True
>>> Tableau([[Integer(1), Integer(2)], [Integer(2), Integer(4)]]).is_
↪standard()
False
>>> Tableau([[Integer(2), Integer(3)], [Integer(2), Integer(4)]]).is_
↪standard()
False
>>> Tableau([[Integer(5), Integer(3)], [Integer(2), Integer(4)]]).is_
↪standard()
False
```

k_weight(k)

Return the k-weight of `self`.

A tableau has k-weight $\alpha = (\alpha_1, ..., \alpha_n)$ if there are exactly α_i distinct residues for the cells occupied by the letter i for each i. The residue of a cell in position (a, b) is $a - b$ modulo $k + 1$.

This definition is the one used in [Ive2012] (p. 12).

EXAMPLES:

```
sage: Tableau([[1,2],[2,3]]).k_weight(1)
[1, 1, 1]
sage: Tableau([[1,2],[2,3]]).k_weight(2)
[1, 2, 1]
sage: t = Tableau([[1,1,1,2,5],[2,3,6],[3],[4]])
sage: t.k_weight(1)
[2, 1, 1, 1, 1, 1]
sage: t.k_weight(2)
[3, 2, 2, 1, 1, 1]
sage: t.k_weight(3)
[3, 1, 2, 1, 1, 1]
sage: t.k_weight(4)
[3, 2, 2, 1, 1, 1]
sage: t.k_weight(5)
[3, 2, 2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(2),Integer(3)]]).k_
↪weight(Integer(1))
[1, 1, 1]
>>> Tableau([[Integer(1),Integer(2)],[Integer(2),Integer(3)]]).k_
↪weight(Integer(2))
[1, 2, 1]
>>> t = Tableau([[Integer(1),Integer(1),Integer(1),Integer(2),Integer(5)],
↪[Integer(2),Integer(3),Integer(6)],[Integer(3)],[Integer(4)]])
>>> t.k_weight(Integer(1))
[2, 1, 1, 1, 1, 1]
>>> t.k_weight(Integer(2))
[3, 2, 2, 1, 1, 1]
>>> t.k_weight(Integer(3))
[3, 1, 2, 1, 1, 1]
>>> t.k_weight(Integer(4))
[3, 2, 2, 1, 1, 1]
>>> t.k_weight(Integer(5))
[3, 2, 2, 1, 1, 1]
```

lambda_catabolism(*part*)

Return the `part`-catabolism of `self`, where `part` is a partition (which can be just given as an array).

For a partition λ and a tableau T, the λ-catabolism of T is defined by performing the following steps.

1. Truncate the parts of λ so that λ is contained in the shape of T. Let m be the length of this partition.
2. Let T_a be the first m rows of T, and T_b be the remaining rows.
3. Let S_a be the skew tableau T_a/λ.
4. Concatenate the reading words of S_a and T_b, and insert into a tableau.

EXAMPLES:

```
sage: Tableau([[1,1,3],[2,4,5]]).lambda_catabolism([2,1])
[[3, 5], [4]]
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.lambda_catabolism([])
```

(continues on next page)

(continued from previous page)

```
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.lambda_catabolism([1])
[[1, 2, 3, 3, 3], [3]]
sage: t.lambda_catabolism([1,1])
[[1, 3, 3, 3], [3]]
sage: t.lambda_catabolism([2,1])
[[3, 3, 3, 3]]
sage: t.lambda_catabolism([4,2,1])
[]
sage: t.lambda_catabolism([5,1])
[[3, 3]]
sage: t.lambda_catabolism([4,1])
[[3, 3]]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(1),Integer(3)],[Integer(2),Integer(4),
↪Integer(5)]]).lambda_catabolism([Integer(2),Integer(1)])
[[3, 5], [4]]
>>> t = Tableau([[Integer(1),Integer(1),Integer(3),Integer(3)],[Integer(2),
↪Integer(3)],[Integer(3)]])
>>> t.lambda_catabolism([])
[[1, 1, 3, 3], [2, 3], [3]]
>>> t.lambda_catabolism([Integer(1)])
[[1, 2, 3, 3, 3], [3]]
>>> t.lambda_catabolism([Integer(1),Integer(1)])
[[1, 3, 3, 3], [3]]
>>> t.lambda_catabolism([Integer(2),Integer(1)])
[[3, 3, 3, 3]]
>>> t.lambda_catabolism([Integer(4),Integer(2),Integer(1)])
[]
>>> t.lambda_catabolism([Integer(5),Integer(1)])
[[3, 3]]
>>> t.lambda_catabolism([Integer(4),Integer(1)])
[[3, 3]]
```

last_letter_lequal(*tab2*)

Return True if self is less than or equal to tab2 in the last letter ordering.

EXAMPLES:

```
sage: st = StandardTableaux([3,2])
sage: f = lambda b: 1 if b else 0
sage: matrix([[f(t1.last_letter_lequal(t2)) for t2 in st] for t1 in st])     #␣
↪needs sage.modules
[1 1 1 1 1]
[0 1 1 1 1]
[0 0 1 1 1]
[0 0 0 1 1]
[0 0 0 0 1]
```

```
>>> from sage.all import *
>>> st = StandardTableaux([Integer(3),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> f = lambda b: Integer(1) if b else Integer(0)
>>> matrix([[f(t1.last_letter_lequal(t2)) for t2 in st] for t1 in st])     #␣
↪needs sage.modules
[1 1 1 1 1]
[0 1 1 1 1]
[0 0 1 1 1]
[0 0 0 1 1]
[0 0 0 0 1]
```

left_key_tableau()

Return the left key tableau of `self`.

The left key tableau of a tableau T is the key tableau whose entries are weakly less than the corresponding entries in T, and whose column reading word is subject to certain conditions. See [LS1990] for the full definition.

ALGORITHM:

The following algorithm follows [Wil2010]. Note that if T is a key tableau then the output of the algorithm is T.

To compute the left key tableau L of a tableau T we iterate over the columns of T. Let T_j be the j-th column of T and iterate over the entries in T_j from bottom to top. Initialize the corresponding entry k in L as the largest entry in T_j. Scan the columns to the left of T_j and with each column update k to be the lowest entry in that column which is weakly less than k. Update T_j and all columns to the left by removing all scanned entries.

See also

- *is_key_tableau()*

EXAMPLES:

```
sage: t = Tableau([[1,2],[2,3]])
sage: t.left_key_tableau()
[[1, 1], [2, 2]]
sage: t = Tableau([[1,1,2,4],[2,3,3],[4],[5]])
sage: t.left_key_tableau()
[[1, 1, 1, 2], [2, 2, 2], [4], [5]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(2),Integer(3)]])
>>> t.left_key_tableau()
[[1, 1], [2, 2]]
>>> t = Tableau([[Integer(1),Integer(1),Integer(2),Integer(4)],[Integer(2),
↪Integer(3),Integer(3)],[Integer(4)],[Integer(5)]])
>>> t.left_key_tableau()
[[1, 1, 1, 2], [2, 2, 2], [4], [5]]
```

leq(*secondtab*)

Check whether each entry of `self` is less-or-equal to the corresponding entry of a further tableau `secondtab`.

INPUT:

- `secondtab` – a tableau of the same shape as `self`

EXAMPLES:

```
sage: T = Tableau([[1, 2], [3]])
sage: S = Tableau([[1, 3], [3]])
sage: G = Tableau([[2, 1], [4]])
sage: H = Tableau([[1, 2], [4]])
sage: T.leq(S)
True
sage: T.leq(T)
True
sage: T.leq(G)
False
sage: T.leq(H)
True
sage: S.leq(T)
False
sage: S.leq(G)
False
sage: S.leq(H)
False
sage: G.leq(H)
False
sage: H.leq(G)
False
```

```
>>> from sage.all import *
>>> T = Tableau([[Integer(1), Integer(2)], [Integer(3)]])
>>> S = Tableau([[Integer(1), Integer(3)], [Integer(3)]])
>>> G = Tableau([[Integer(2), Integer(1)], [Integer(4)]])
>>> H = Tableau([[Integer(1), Integer(2)], [Integer(4)]])
>>> T.leq(S)
True
>>> T.leq(T)
True
>>> T.leq(G)
False
>>> T.leq(H)
True
>>> S.leq(T)
False
>>> S.leq(G)
False
>>> S.leq(H)
False
>>> G.leq(H)
False
>>> H.leq(G)
False
```

level()

Return the level of `self`, which is always 1.

This function exists mainly for compatibility with *TableauTuple*.

EXAMPLES:

```
sage: Tableau([[1,2,3],[4,5]]).level()
1
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]).
↪level()
1
```

major_index()

Return the major index of `self`.

The major index of a tableau T is defined to be the sum of the number of descents of `T` (defined in *descents()*) with the sum of their legs' lengths.

> **Warning**
>
> This is not to be confused with the major index of a standard tableau.

EXAMPLES:

```
sage: Tableau( [[1,4],[2,3]] ).major_index()
1
sage: Tableau( [[1,2],[3,4]] ).major_index()
2
```

```
>>> from sage.all import *
>>> Tableau( [[Integer(1),Integer(4)],[Integer(2),Integer(3)]] ).major_index()
1
>>> Tableau( [[Integer(1),Integer(2)],[Integer(3),Integer(4)]] ).major_index()
2
```

If the major index would be defined in the sense of standard tableaux theory, then the following would give 3 for a result:

```
sage: Tableau( [[1,2,3],[4,5]] ).major_index()
2
```

```
>>> from sage.all import *
>>> Tableau( [[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]] ).
↪major_index()
2
```

plot(*descents=False*)

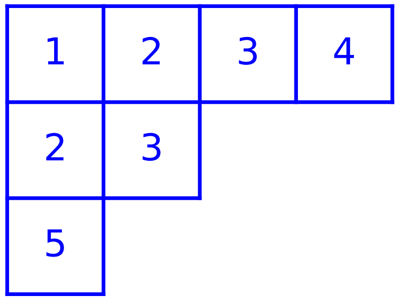
Return a plot `self`.

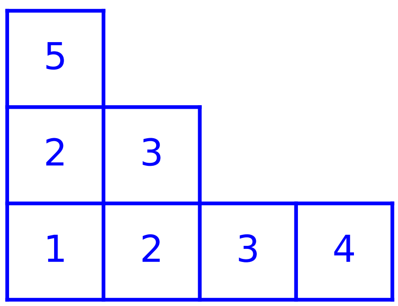
If English notation is set then the first row of the tableau is on the top:

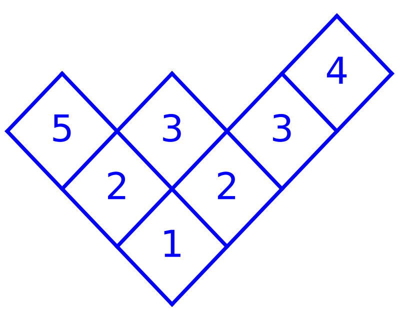
If French notation is set, the first row of the tableau is on the bottom:

If Russian notation is set, we tilt the French notation by 45 degrees:

INPUT:


1 2 3 4
2 3
5


5
2 3
1 2 3 4


4
5 3 3
2 2
1

- descents – boolean (default: False); if True, then the descents are marked in the tableau; only valid if self is a standard tableau

EXAMPLES:

```
sage: t = Tableau([[1,2,4],[3]])
sage: t.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 11 graphics primitives
sage: t.plot(descents=True)                                                 #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives

sage: t = Tableau([[2,2,4],[3]])
sage: t.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 11 graphics primitives
sage: t.plot(descents=True)                                                 #␣
↪needs sage.plot
Traceback (most recent call last):
...
ValueError: the tableau must be standard for 'descents=True'
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3)]])
>>> t.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 11 graphics primitives
>>> t.plot(descents=True)                                                 #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives

>>> t = Tableau([[Integer(2),Integer(2),Integer(4)],[Integer(3)]])
>>> t.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 11 graphics primitives
>>> t.plot(descents=True)                                                 #␣
↪needs sage.plot
Traceback (most recent call last):
...
ValueError: the tableau must be standard for 'descents=True'
```

pp()

Pretty print a string of the tableau.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[3,4],[5]])
sage: T.pp()
  1  2  3
  3  4
  5
sage: Tableaux.options.convention="french"
sage: T.pp()
```

(continues on next page)

(continued from previous page)

```
  5
  3  4
  1  2  3
sage: Tableaux.options.convention="russian"
sage: T.pp()
 5    4    3
   3    2
      1
sage: Tableaux.options._reset()
```

```
>>> from sage.all import *
>>> T = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(3),Integer(4)],
↪[Integer(5)]])
>>> T.pp()
  1  2  3
  3  4
  5
>>> Tableaux.options.convention="french"
>>> T.pp()
  5
  3  4
  1  2  3
>>> Tableaux.options.convention="russian"
>>> T.pp()
 5    4    3
   3    2
      1
>>> Tableaux.options._reset()
```

promotion(*n*)

Return the image of `self` under the promotion operator.

> **Warning**
>
> You might know this operator as the inverse promotion operator – literature does not agree on the name. You might also be looking for the Lapointe-Lascoux-Morse promotion operator (*promotion_operator()*).

The promotion operator, applied to a tableau t, does the following:

Iterate over all letters $n+1$ in the tableau t, from left to right. For each of these letters, do the following:

- Remove the letter from t, thus leaving a hole where it used to be.
- Apply jeu de taquin to move this hole southwest (in French notation) until it reaches the inner boundary of t.
- Fill 0 into the hole once jeu de taquin has completed.

Once this all is done, add 1 to each letter in the tableau. This is not always well-defined. Restricted to the class of semistandard tableaux whose entries are all $\leq n+1$, this is the usual promotion operator defined on this class.

When `self` is a standard tableau of size `n + 1`, this definition of promotion is precisely the one given in [Hai1992] (p. 90). It is the inverse of the maps called “promotion” in [Sag2011] (p. 23) and in [Stan2009].

Warning

To my (Darij’s) knowledge, the fact that the above promotion operator really is the inverse of the “inverse promotion operator” *promotion_inverse()* for semistandard tableaux has never been proven in literature. Corrections are welcome.

REFERENCES:

- [Hai1992]
- [Sag2011]

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,3]])
sage: t.promotion(2)
[[1, 1], [2, 3]]

sage: t = Tableau([[1,1,1],[2,2,3],[3,4,4]])
sage: t.promotion(3)
[[1, 1, 2], [2, 2, 3], [3, 4, 4]]

sage: t = Tableau([[1,2],[2]])
sage: t.promotion(3)
[[2, 3], [3]]

sage: t = Tableau([[1,1,3],[2,2]])
sage: t.promotion(2)
[[1, 2, 2], [3, 3]]

sage: t = Tableau([[1,1,3],[2,3]])
sage: t.promotion(2)
[[1, 1, 2], [2, 3]]

sage: t = Tableau([])
sage: t.promotion(2)
[]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(3)]])
>>> t.promotion(Integer(2))
[[1, 1], [2, 3]]

>>> t = Tableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(2),
↪Integer(3)],[Integer(3),Integer(4),Integer(4)]])
>>> t.promotion(Integer(3))
[[1, 1, 2], [2, 2, 3], [3, 4, 4]]

>>> t = Tableau([[Integer(1),Integer(2)],[Integer(2)]])
>>> t.promotion(Integer(3))
[[2, 3], [3]]
```

(continues on next page)

(continued from previous page)

```
>>> t = Tableau([[Integer(1),Integer(1),Integer(3)],[Integer(2),Integer(2)]])
>>> t.promotion(Integer(2))
[[1, 2, 2], [3, 3]]

>>> t = Tableau([[Integer(1),Integer(1),Integer(3)],[Integer(2),Integer(3)]])
>>> t.promotion(Integer(2))
[[1, 1, 2], [2, 3]]

>>> t = Tableau([])
>>> t.promotion(Integer(2))
[]
```

promotion_inverse(*n*)

Return the image of `self` under the inverse promotion operator.

> **Warning**
>
> You might know this operator as the promotion operator (without "inverse") – literature does not agree on the name.

The inverse promotion operator, applied to a tableau t, does the following:

Iterate over all letters 1 in the tableau t, from right to left. For each of these letters, do the following:

- Remove the letter from t, thus leaving a hole where it used to be.
- Apply jeu de taquin to move this hole northeast (in French notation) until it reaches the outer boundary of t.
- Fill $n + 2$ into the hole once jeu de taquin has completed.

Once this all is done, subtract 1 from each letter in the tableau. This is not always well-defined. Restricted to the class of semistandard tableaux whose entries are all $\leq n + 1$, this is the usual inverse promotion operator defined on this class.

When `self` is a standard tableau of size `n + 1`, this definition of inverse promotion is the map called "promotion" in [Sag2011] (p. 23) and in [Stan2009], and is the inverse of the map called "promotion" in [Hai1992] (p. 90).

> **Warning**
>
> To my (Darij's) knowledge, the fact that the above "inverse promotion operator" really is the inverse of the promotion operator `promotion()` for semistandard tableaux has never been proven in literature. Corrections are welcome.

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,3]])
sage: t.promotion_inverse(2)
[[1, 2], [2, 3]]

sage: t = Tableau([[1,2],[2,3]])
```

(continues on next page)

(continued from previous page)

```
sage: t.promotion_inverse(2)
[[1, 1], [2, 3]]

sage: t = Tableau([[1,2,5],[3,3,6],[4,7]])
sage: t.promotion_inverse(8)
[[1, 2, 4], [2, 5, 9], [3, 6]]

sage: t = Tableau([])
sage: t.promotion_inverse(2)
[]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(3)]])
>>> t.promotion_inverse(Integer(2))
[[1, 2], [2, 3]]

>>> t = Tableau([[Integer(1),Integer(2)],[Integer(2),Integer(3)]])
>>> t.promotion_inverse(Integer(2))
[[1, 1], [2, 3]]

>>> t = Tableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),Integer(3),
↪Integer(6)],[Integer(4),Integer(7)]])
>>> t.promotion_inverse(Integer(8))
[[1, 2, 4], [2, 5, 9], [3, 6]]

>>> t = Tableau([])
>>> t.promotion_inverse(Integer(2))
[]
```

promotion_operator(i)

Return a list of semistandard tableaux obtained by the i-th Lapointe-Lascoux-Morse promotion operator from the semistandard tableau `self`.

> **Warning**
>
> This is not Schuetzenberger's jeu de taquin promotion! For the latter, see *promotion()* and *promotion_inverse()*.

This operator is defined by taking the maximum entry m of T, then adding a horizontal i-strip to T in all possible ways, each time filling this strip with $m + 1$'s, and finally letting the permutation $\sigma_1\sigma_2\cdots\sigma_m = (2, 3, \ldots, m + 1, 1)$ act on each of the resulting tableaux via the Lascoux-Schuetzenberger action (*symmetric_group_action_on_values()*). This method returns the list of all resulting tableaux. See [LLM2003] for the purpose of this operator.

EXAMPLES:

```
sage: t = Tableau([[1,2],[3]])
sage: t.promotion_operator(1)
[[[1, 2, 4], [3]], [[1, 2], [3, 4]], [[1, 2], [3], [4]]]
sage: t.promotion_operator(2)
[[[1, 1, 2, 4], [3]],
```

(continues on next page)

(continued from previous page)

```
 [[1, 1, 4], [2, 3]],
 [[1, 1, 2], [3], [4]],
 [[1, 1], [2, 3], [4]]]
sage: Tableau([[1]]).promotion_operator(2)
[[[1, 1, 2]], [[1, 1], [2]]]
sage: Tableau([[1,1],[2]]).promotion_operator(3)
[[[1, 1, 1, 2, 3], [2]],
 [[1, 1, 1, 3], [2, 2]],
 [[1, 1, 1, 2], [2], [3]],
 [[1, 1, 1], [2, 2], [3]]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(3)]])
>>> t.promotion_operator(Integer(1))
[[[1, 2, 4], [3]], [[1, 2], [3, 4]], [[1, 2], [3], [4]]]
>>> t.promotion_operator(Integer(2))
[[[1, 1, 2, 4], [3]],
 [[1, 1, 4], [2, 3]],
 [[1, 1, 2], [3], [4]],
 [[1, 1], [2, 3], [4]]]
>>> Tableau([[Integer(1)]]).promotion_operator(Integer(2))
[[[1, 1, 2]], [[1, 1], [2]]]
>>> Tableau([[Integer(1),Integer(1)],[Integer(2)]]).promotion_
↪operator(Integer(3))
[[[1, 1, 1, 2, 3], [2]],
 [[1, 1, 1, 3], [2, 2]],
 [[1, 1, 1, 2], [2], [3]],
 [[1, 1, 1], [2, 2], [3]]]
```

The example from [LLM2003] p. 12:

```
sage: Tableau([[1,1],[2,2]]).promotion_operator(3)
[[[1, 1, 1, 3, 3], [2, 2]],
 [[1, 1, 1, 3], [2, 2], [3]],
 [[1, 1, 1], [2, 2], [3, 3]]]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(1)],[Integer(2),Integer(2)]]).promotion_
↪operator(Integer(3))
[[[1, 1, 1, 3, 3], [2, 2]],
 [[1, 1, 1, 3], [2, 2], [3]],
 [[1, 1, 1], [2, 2], [3, 3]]]
```

raise_action_from_words(*f*, **args*)

EXAMPLES:

```
sage: from sage.combinat.tableau import symmetric_group_action_on_values
sage: import functools
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: f = functools.partial(t.raise_action_from_words, symmetric_group_action_
↪on_values)
sage: f([1,2,3])
```

(continues on next page)

(continued from previous page)

```
[[1, 1, 3, 3], [2, 3], [3]]
sage: f([3,2,1])
[[1, 1, 1, 1], [2, 3], [3]]
sage: f([1,3,2])
[[1, 1, 2, 2], [2, 2], [3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau import symmetric_group_action_on_values
>>> import functools
>>> t = Tableau([[Integer(1),Integer(1),Integer(3),Integer(3)],[Integer(2),
↪Integer(3)],[Integer(3)]])
>>> f = functools.partial(t.raise_action_from_words, symmetric_group_action_
↪on_values)
>>> f([Integer(1),Integer(2),Integer(3)])
[[1, 1, 3, 3], [2, 3], [3]]
>>> f([Integer(3),Integer(2),Integer(1)])
[[1, 1, 1, 1], [2, 3], [3]]
>>> f([Integer(1),Integer(3),Integer(2)])
[[1, 1, 2, 2], [2, 2], [3]]
```

reading_word_permutation()

Return the permutation obtained by reading the entries of the standardization of `self` row by row, starting with the bottommost row (in English notation).

EXAMPLES:

```
sage: StandardTableau([[1,2],[3,4]]).reading_word_permutation()
[3, 4, 1, 2]
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).
↪reading_word_permutation()
[3, 4, 1, 2]
```

Check that Issue #14724 is fixed:

```
sage: SemistandardTableau([[1,1]]).reading_word_permutation()
[1, 2]
```

```
>>> from sage.all import *
>>> SemistandardTableau([[Integer(1),Integer(1)]]).reading_word_permutation()
[1, 2]
```

reduced_column_word()

Return the lexicographically minimal reduced expression for the permutation that maps the conjugate of the `initial_tableau()` to `self`.

Ths reduced expression is a minimal length coset representative for the corresponding Young subgroup. In one line notation, the permutation is obtained by concatenating the columns of the tableau in order from top to bottom.

EXAMPLES:

```
sage: StandardTableau([[1,4,6],[2,5],[3]]).reduced_column_word()
[]
sage: StandardTableau([[1,4,5],[2,6],[3]]).reduced_column_word()
[5]
sage: StandardTableau([[1,3,6],[2,5],[4]]).reduced_column_word()
[3]
sage: StandardTableau([[1,3,5],[2,6],[4]]).reduced_column_word()
[3, 5]
sage: StandardTableau([[1,2,5],[3,6],[4]]).reduced_column_word()
[3, 2, 5]
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(4),Integer(6)],[Integer(2),
↪Integer(5)],[Integer(3)]]).reduced_column_word()
[]
>>> StandardTableau([[Integer(1),Integer(4),Integer(5)],[Integer(2),
↪Integer(6)],[Integer(3)]]).reduced_column_word()
[5]
>>> StandardTableau([[Integer(1),Integer(3),Integer(6)],[Integer(2),
↪Integer(5)],[Integer(4)]]).reduced_column_word()
[3]
>>> StandardTableau([[Integer(1),Integer(3),Integer(5)],[Integer(2),
↪Integer(6)],[Integer(4)]]).reduced_column_word()
[3, 5]
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(6)],[Integer(4)]]).reduced_column_word()
[3, 2, 5]
```

reduced_lambda_catabolism(*part*)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.reduced_lambda_catabolism([])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.reduced_lambda_catabolism([1])
[[1, 2, 3, 3, 3], [3]]
sage: t.reduced_lambda_catabolism([1,1])
[[1, 3, 3, 3], [3]]
sage: t.reduced_lambda_catabolism([2,1])
[[3, 3, 3, 3]]
sage: t.reduced_lambda_catabolism([4,2,1])
[]
sage: t.reduced_lambda_catabolism([5,1])
0
sage: t.reduced_lambda_catabolism([4,1])
0
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(3),Integer(3)],[Integer(2),
↪Integer(3)],[Integer(3)]])
>>> t.reduced_lambda_catabolism([])
[[1, 1, 3, 3], [2, 3], [3]]
```

(continues on next page)

(continued from previous page)

```
>>> t.reduced_lambda_catabolism([Integer(1)])
[[1, 2, 3, 3, 3], [3]]
>>> t.reduced_lambda_catabolism([Integer(1),Integer(1)])
[[1, 3, 3, 3], [3]]
>>> t.reduced_lambda_catabolism([Integer(2),Integer(1)])
[[3, 3, 3, 3]]
>>> t.reduced_lambda_catabolism([Integer(4),Integer(2),Integer(1)])
[]
>>> t.reduced_lambda_catabolism([Integer(5),Integer(1)])
0
>>> t.reduced_lambda_catabolism([Integer(4),Integer(1)])
0
```

reduced_row_word()

Return the lexicographically minimal reduced expression for the permutation that maps the `initial_tableau()` to `self`.

Ths reduced expression is a minimal length coset representative for the corresponding Young subgroup. In one line notation, the permutation is obtained by concatenating the rows of the tableau in order from top to bottom.

EXAMPLES:

```
sage: StandardTableau([[1,2,3],[4,5],[6]]).reduced_row_word()
[]
sage: StandardTableau([[1,2,3],[4,6],[5]]).reduced_row_word()
[5]
sage: StandardTableau([[1,2,4],[3,6],[5]]).reduced_row_word()
[3, 5]
sage: StandardTableau([[1,2,5],[3,6],[4]]).reduced_row_word()
[3, 5, 4]
sage: StandardTableau([[1,2,6],[3,5],[4]]).reduced_row_word()
[3, 4, 5, 4]
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)],[Integer(6)]]).reduced_row_word()
[]
>>> StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(6)],[Integer(5)]]).reduced_row_word()
[5]
>>> StandardTableau([[Integer(1),Integer(2),Integer(4)],[Integer(3),
↪Integer(6)],[Integer(5)]]).reduced_row_word()
[3, 5]
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(6)],[Integer(4)]]).reduced_row_word()
[3, 5, 4]
>>> StandardTableau([[Integer(1),Integer(2),Integer(6)],[Integer(3),
↪Integer(5)],[Integer(4)]]).reduced_row_word()
[3, 4, 5, 4]
```

residue(*k*, *e*, *multicharge=(0,)*)

Return the residue of the integer `k` in the tableau `self`.

The *residue* of k in a standard tableau is $c - r + m$ in $\mathbf{Z}/e\mathbf{Z}$, where k appears in row r and column c of the tableau with multicharge m.

INPUT:

- `k` – integer in $\{1, 2, \ldots, n\}$
- `e` – integer in $\{0, 2, 3, 4, 5, \ldots\}$
- `multicharge` – (default: `[0]`) a list of length 1

Here n is its size of `self`.

The `multicharge` is a list of length 1 which gives an offset for all of the contents. It is included mainly for compatibility with `residue()`.

OUTPUT: the residue in $\mathbf{Z}/e\mathbf{Z}$

EXAMPLES:

```
sage: StandardTableau([[1,2,5],[3,4]]).residue(1,3)
0
sage: StandardTableau([[1,2,5],[3,4]]).residue(2,3)
1
sage: StandardTableau([[1,2,5],[3,4]]).residue(3,3)
2
sage: StandardTableau([[1,2,5],[3,4]]).residue(4,3)
0
sage: StandardTableau([[1,2,5],[3,4]]).residue(5,3)
2
sage: StandardTableau([[1,2,5],[3,4]]).residue(6,3)
Traceback (most recent call last):
...
ValueError: 6 does not appear in the tableau
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).residue(Integer(1),Integer(3))
0
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).residue(Integer(2),Integer(3))
1
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).residue(Integer(3),Integer(3))
2
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).residue(Integer(4),Integer(3))
0
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).residue(Integer(5),Integer(3))
2
>>> StandardTableau([[Integer(1),Integer(2),Integer(5)],[Integer(3),
↪Integer(4)]]).residue(Integer(6),Integer(3))
Traceback (most recent call last):
...
ValueError: 6 does not appear in the tableau
```

residue_sequence(*e*, *multicharge=(0,)*)

Return the *sage.combinat.tableau_residues.ResidueSequence* of the tableau self.

INPUT:

- e – integer in $\{0, 2, 3, 4, 5, \ldots\}$
- multicharge – (default: [0]) a sequence of integers of length 1

The $multicharge$ is a list of length 1 which gives an offset for all of the contents. It is included mainly for compatibility with residue().

OUTPUT:

The corresponding residue sequence of the tableau; see ResidueSequence.

EXAMPLES:

```
sage: StandardTableauTuple([[1,2],[3,4]]).residue_sequence(2)                    #␣
↪needs sage.groups
2-residue sequence (0,1,1,0) with multicharge (0)
sage: StandardTableauTuple([[1,2],[3,4]]).residue_sequence(3)                    #␣
↪needs sage.groups
3-residue sequence (0,1,2,0) with multicharge (0)
sage: StandardTableauTuple([[1,2],[3,4]]).residue_sequence(4)                    #␣
↪needs sage.groups
4-residue sequence (0,1,3,0) with multicharge (0)
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).
↪residue_sequence(Integer(2))                    # needs sage.groups
2-residue sequence (0,1,1,0) with multicharge (0)
>>> StandardTableauTuple([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).
↪residue_sequence(Integer(3))                    # needs sage.groups
3-residue sequence (0,1,2,0) with multicharge (0)
>>> StandardTableauTuple([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).
↪residue_sequence(Integer(4))                    # needs sage.groups
4-residue sequence (0,1,3,0) with multicharge (0)
```

restrict(*n*)

Return the restriction of the semistandard tableau self to n. If possible, the restricted tableau will have the same parent as this tableau.

If T is a semistandard tableau and n is a nonnegative integer, then the restriction of T to n is defined as the (semistandard) tableau obtained by removing all cells filled with entries greater than n from T.

> **Note**
>
> If only the shape of the restriction, rather than the whole restriction, is needed, then the faster method *restriction_shape()* is preferred.

EXAMPLES:

```
sage: Tableau([[1,2],[3],[4]]).restrict(3)
[[1, 2], [3]]
sage: StandardTableau([[1,2],[3],[4]]).restrict(2)
```

(continues on next page)

(continued from previous page)

```
[[1, 2]]
sage: Tableau([[1,2,3],[2,4,4],[3]]).restrict(0)
[]
sage: Tableau([[1,2,3],[2,4,4],[3]]).restrict(2)
[[1, 2], [2]]
sage: Tableau([[1,2,3],[2,4,4],[3]]).restrict(3)
[[1, 2, 3], [2], [3]]
sage: Tableau([[1,2,3],[2,4,4],[3]]).restrict(5)
[[1, 2, 3], [2, 4, 4], [3]]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3)],[Integer(4)]]).
↪restrict(Integer(3))
[[1, 2], [3]]
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3)],[Integer(4)]]).
↪restrict(Integer(2))
[[1, 2]]
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(4),
↪Integer(4)],[Integer(3)]]).restrict(Integer(0))
[]
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(4),
↪Integer(4)],[Integer(3)]]).restrict(Integer(2))
[[1, 2], [2]]
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(4),
↪Integer(4)],[Integer(3)]]).restrict(Integer(3))
[[1, 2, 3], [2], [3]]
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(4),
↪Integer(4)],[Integer(3)]]).restrict(Integer(5))
[[1, 2, 3], [2, 4, 4], [3]]
```

If possible the restricted tableau will belong to the same category as the original tableau:

```
sage: S = StandardTableau([[1,2,4,7],[3,5],[6]]); S.category()
Category of elements of Standard tableaux
sage: S.restrict(4).category()
Category of elements of Standard tableaux
sage: SS=StandardTableaux([4,2,1])([[1,2,4,7],[3,5],[6]]); SS.category()
Category of elements of Standard tableaux of shape [4, 2, 1]
sage: SS.restrict(4).category()
Category of elements of Standard tableaux

sage: Tableau([[1,2],[3],[4]]).restrict(3)
[[1, 2], [3]]
sage: Tableau([[1,2],[3],[4]]).restrict(2)
[[1, 2]]
sage: SemistandardTableau([[1,1],[2]]).restrict(1)
[[1, 1]]
sage: _.category()
Category of elements of Semistandard tableaux
```

```
>>> from sage.all import *
>>> S = StandardTableau([[Integer(1),Integer(2),Integer(4),Integer(7)],
```

(continues on next page)

(continued from previous page)

```
↪[Integer(3),Integer(5)],[Integer(6)]]); S.category()
Category of elements of Standard tableaux
>>> S.restrict(Integer(4)).category()
Category of elements of Standard tableaux
>>> SS=StandardTableaux([Integer(4),Integer(2),Integer(1)])([[Integer(1),
↪Integer(2),Integer(4),Integer(7)],[Integer(3),Integer(5)],[Integer(6)]]);
↪SS.category()
Category of elements of Standard tableaux of shape [4, 2, 1]
>>> SS.restrict(Integer(4)).category()
Category of elements of Standard tableaux

>>> Tableau([[Integer(1),Integer(2)],[Integer(3)],[Integer(4)]]).
↪restrict(Integer(3))
[[1, 2], [3]]
>>> Tableau([[Integer(1),Integer(2)],[Integer(3)],[Integer(4)]]).
↪restrict(Integer(2))
[[1, 2]]
>>> SemistandardTableau([[Integer(1),Integer(1)],[Integer(2)]]).
↪restrict(Integer(1))
[[1, 1]]
>>> _.category()
Category of elements of Semistandard tableaux
```

restriction_shape(*n*)

Return the shape of the restriction of the semistandard tableau `self` to `n`.

If T is a semistandard tableau and n is a nonnegative integer, then the restriction of T to n is defined as the (semistandard) tableau obtained by removing all cells filled with entries greater than n from T.

This method computes merely the shape of the restriction. For the restriction itself, use `restrict()`.

EXAMPLES:

```
sage: Tableau([[1,2],[2,3],[3,4]]).restriction_shape(3)
[2, 2, 1]
sage: StandardTableau([[1,2],[3],[4],[5]]).restriction_shape(2)
[2]
sage: Tableau([[1,3,3,5],[2,4,4],[17]]).restriction_shape(0)
[]
sage: Tableau([[1,3,3,5],[2,4,4],[17]]).restriction_shape(2)
[1, 1]
sage: Tableau([[1,3,3,5],[2,4,4],[17]]).restriction_shape(3)
[3, 1]
sage: Tableau([[1,3,3,5],[2,4,4],[17]]).restriction_shape(5)
[4, 3]

sage: all( T.restriction_shape(i) == T.restrict(i).shape()
....:      for T in StandardTableaux(5) for i in range(1, 5) )
True
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(2),Integer(3)],[Integer(3),
↪Integer(4)]]).restriction_shape(Integer(3))
```

(continues on next page)

(continued from previous page)

```
[2, 2, 1]
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3)],[Integer(4)],
↪[Integer(5)]]).restriction_shape(Integer(2))
[2]
>>> Tableau([[Integer(1),Integer(3),Integer(3),Integer(5)],[Integer(2),
↪Integer(4),Integer(4)],[Integer(17)]]).restriction_shape(Integer(0))
[]
>>> Tableau([[Integer(1),Integer(3),Integer(3),Integer(5)],[Integer(2),
↪Integer(4),Integer(4)],[Integer(17)]]).restriction_shape(Integer(2))
[1, 1]
>>> Tableau([[Integer(1),Integer(3),Integer(3),Integer(5)],[Integer(2),
↪Integer(4),Integer(4)],[Integer(17)]]).restriction_shape(Integer(3))
[3, 1]
>>> Tableau([[Integer(1),Integer(3),Integer(3),Integer(5)],[Integer(2),
↪Integer(4),Integer(4)],[Integer(17)]]).restriction_shape(Integer(5))
[4, 3]

>>> all( T.restriction_shape(i) == T.restrict(i).shape()
...      for T in StandardTableaux(Integer(5)) for i in range(Integer(1),
↪Integer(5)) )
True
```

reverse_bump(*loc*)

Reverse row bump the entry of `self` at the specified location `loc` (given as a row index or a corner `(r, c)` of the tableau).

This is the reverse of Schensted's row-insertion algorithm. See Section 1.1, page 8, of Fulton's [Ful1997].

INPUT:

- `loc` – can be either of the following:
  - The coordinates `(r, c)` of the square to reverse-bump (which must be a corner of the tableau);
  - The row index `r` of this square.

  Note that both `r` and `c` are 0-based, i.e., the topmost row and the leftmost column are the 0-th row and the 0-th column.

OUTPUT: an ordered pair consisting of:

1. The resulting (smaller) tableau;
2. The entry bumped out at the end of the process.

See also

bump()

EXAMPLES:

This is the reverse of Schensted's bump:

```
sage: T = Tableau([[1, 1, 2, 2, 4], [2, 3, 3], [3, 4], [4]])
sage: T.reverse_bump(2)
([[1, 1, 2, 3, 4], [2, 3, 4], [3], [4]], 2)
```

(continues on next page)

(continued from previous page)

```
sage: T == T.reverse_bump(2)[0].bump(2)
True
sage: T.reverse_bump((3, 0))
([[1, 2, 2, 2, 4], [3, 3, 3], [4, 4]], 1)
```

```
>>> from sage.all import *
>>> T = Tableau([[Integer(1), Integer(1), Integer(2), Integer(2), Integer(4)],
↪ [Integer(2), Integer(3), Integer(3)], [Integer(3), Integer(4)],
↪[Integer(4)]])
>>> T.reverse_bump(Integer(2))
([[1, 1, 2, 3, 4], [2, 3, 4], [3], [4]], 2)
>>> T == T.reverse_bump(Integer(2))[Integer(0)].bump(Integer(2))
True
>>> T.reverse_bump((Integer(3), Integer(0)))
([[1, 2, 2, 2, 4], [3, 3, 3], [4, 4]], 1)
```

Some errors caused by wrong input:

```
sage: T.reverse_bump((3, 1))
Traceback (most recent call last):
...
ValueError: invalid corner
sage: T.reverse_bump(4)
Traceback (most recent call last):
...
IndexError: list index out of range
sage: Tableau([[2, 2, 1], [3, 3]]).reverse_bump(0)
Traceback (most recent call last):
...
ValueError: reverse bumping is only defined for semistandard tableaux
```

```
>>> from sage.all import *
>>> T.reverse_bump((Integer(3), Integer(1)))
Traceback (most recent call last):
...
ValueError: invalid corner
>>> T.reverse_bump(Integer(4))
Traceback (most recent call last):
...
IndexError: list index out of range
>>> Tableau([[Integer(2), Integer(2), Integer(1)], [Integer(3), Integer(3)]]).
↪reverse_bump(Integer(0))
Traceback (most recent call last):
...
ValueError: reverse bumping is only defined for semistandard tableaux
```

Some edge cases:

```
sage: Tableau([[1]]).reverse_bump(0)
([], 1)
sage: Tableau([[1,1]]).reverse_bump(0)
([[1]], 1)
```

(continues on next page)

(continued from previous page)

```
sage: Tableau([]).reverse_bump(0)
Traceback (most recent call last):
...
IndexError: list index out of range
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1)]]).reverse_bump(Integer(0))
([], 1)
>>> Tableau([[Integer(1),Integer(1)]]).reverse_bump(Integer(0))
([[1]], 1)
>>> Tableau([]).reverse_bump(Integer(0))
Traceback (most recent call last):
...
IndexError: list index out of range
```

Note

Reverse row bumping is only implemented for tableaux with weakly increasing and strictly increasing columns (though the tableau does not need to be an instance of class *SemistandardTableau*).

right_key_tableau()

Return the right key tableau of `self`.

The right key tableau of a tableau T is a key tableau whose entries are weakly greater than the corresponding entries in T, and whose column reading word is subject to certain conditions. See [LS1990] for the full definition.

ALGORITHM:

The following algorithm follows [Wil2010]. Note that if T is a key tableau then the output of the algorithm is T.

To compute the right key tableau R of a tableau T we iterate over the columns of T. Let T_j be the j-th column of T and iterate over the entries in T_j from bottom to top. Initialize the corresponding entry k in R to be the largest entry in T_j. Scan the bottom of each column of T to the right of T_j, updating k to be the scanned entry whenever the scanned entry is weakly greater than k. Update T_j and all columns to the right by removing all scanned entries.

See also

- *is_key_tableau()*

EXAMPLES:

```
sage: t = Tableau([[1,2],[2,3]])
sage: t.right_key_tableau()
[[2, 2], [3, 3]]
sage: t = Tableau([[1,1,2,4],[2,3,3],[4],[5]])
sage: t.right_key_tableau()
[[2, 2, 2, 4], [3, 4, 4], [4], [5]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(2),Integer(3)]])
>>> t.right_key_tableau()
[[2, 2], [3, 3]]
>>> t = Tableau([[Integer(1),Integer(1),Integer(2),Integer(4)],[Integer(2),
↪Integer(3),Integer(3)],[Integer(4)],[Integer(5)]])
>>> t.right_key_tableau()
[[2, 2, 2, 4], [3, 4, 4], [4], [5]]
```

rotate_180()

Return the tableau obtained by rotating `self` by 180 degrees.

This only works for rectangular tableaux.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).rotate_180()
[[4, 3], [2, 1]]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).rotate_180()
[[4, 3], [2, 1]]
```

row_stabilizer()

Return the PermutationGroup corresponding to the row stabilizer of `self`.

This assumes that every integer from 1 to the size of `self` appears exactly once in `self`.

EXAMPLES:

```
sage: # needs sage.groups
sage: rs = Tableau([[1,2,3],[4,5]]).row_stabilizer()
sage: rs.order() == factorial(3)*factorial(2)
True
sage: PermutationGroupElement([(1,3,2),(4,5)]) in rs
True
sage: PermutationGroupElement([(1,4)]) in rs
False
sage: rs = Tableau([[1, 2],[3]]).row_stabilizer()
sage: PermutationGroupElement([(1,2),(3,)]) in rs
True
sage: rs.one().domain()
[1, 2, 3]
sage: rs = Tableau([[1],[2],[3]]).row_stabilizer()
sage: rs.order()
1
sage: rs = Tableau([[2,4,5],[1,3]]).row_stabilizer()
sage: rs.order()
12
sage: rs = Tableau([]).row_stabilizer()
sage: rs.order()
1
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> rs = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]]).row_stabilizer()
>>> rs.order() == factorial(Integer(3))*factorial(Integer(2))
True
>>> PermutationGroupElement([(Integer(1),Integer(3),Integer(2)),(Integer(4),
↪Integer(5))]) in rs
True
>>> PermutationGroupElement([(Integer(1),Integer(4))]) in rs
False
>>> rs = Tableau([[Integer(1), Integer(2)],[Integer(3)]]).row_stabilizer()
>>> PermutationGroupElement([(Integer(1),Integer(2)),(Integer(3),)]) in rs
True
>>> rs.one().domain()
[1, 2, 3]
>>> rs = Tableau([[Integer(1)],[Integer(2)],[Integer(3)]]).row_stabilizer()
>>> rs.order()
1
>>> rs = Tableau([[Integer(2),Integer(4),Integer(5)],[Integer(1),
↪Integer(3)]]).row_stabilizer()
>>> rs.order()
12
>>> rs = Tableau([]).row_stabilizer()
>>> rs.order()
1
```

schensted_insert(*i*, *left=False*)

Insert `i` into `self` using Schensted's row-bumping (or row-insertion) algorithm.

INPUT:

- `i` – a number to insert
- `left` – boolean (default: `False`); if set to `True`, the insertion will be done from the left. That is, if one thinks of the algorithm as appending a letter to the reading word of `self`, we append the letter to the left instead of the right

EXAMPLES:

```
sage: t = Tableau([[3,5],[7]])
sage: t.schensted_insert(8)
[[3, 5, 8], [7]]
sage: t.schensted_insert(8, left=True)
[[3, 5], [7], [8]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(3),Integer(5)],[Integer(7)]])
>>> t.schensted_insert(Integer(8))
[[3, 5, 8], [7]]
>>> t.schensted_insert(Integer(8), left=True)
[[3, 5], [7], [8]]
```

schuetzenberger_involution(*n=None*, *check=True*)

Return the Schuetzenberger involution of the tableau `self`.

This method relies on the analogous method on words, which reverts the word and then complements all letters within the underlying ordered alphabet. If n is specified, the underlying alphabet is assumed to be $[1, 2, \ldots, n]$. If no alphabet is specified, n is the maximal letter appearing in `self`.

INPUT:

- `n` – integer specifying the maximal letter in the alphabet (optional)
- `check` – boolean (default: `True`); check to make sure `self` is semistandard

OUTPUT: a tableau, the Schuetzenberger involution of `self`

EXAMPLES:

```
sage: t = Tableau([[1,1,1],[2,2]])
sage: t.schuetzenberger_involution(3)
[[2, 2, 3], [3, 3]]

sage: t = Tableau([[1,2,3],[4,5]])
sage: t.schuetzenberger_involution()
[[1, 2, 5], [3, 4]]

sage: t = Tableau([[1,3,5,7],[2,4,6],[8,9]])
sage: t.schuetzenberger_involution()
[[1, 2, 6, 8], [3, 4, 9], [5, 7]]

sage: t = Tableau([])
sage: t.schuetzenberger_involution()
[]

sage: t = StandardTableau([[1,2,3],[4,5]])
sage: s = t.schuetzenberger_involution()
sage: s.parent()
Standard tableaux
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(1)],[Integer(2),Integer(2)]])
>>> t.schuetzenberger_involution(Integer(3))
[[2, 2, 3], [3, 3]]

>>> t = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])
>>> t.schuetzenberger_involution()
[[1, 2, 5], [3, 4]]

>>> t = Tableau([[Integer(1),Integer(3),Integer(5),Integer(7)],[Integer(2),
↪Integer(4),Integer(6)],[Integer(8),Integer(9)]])
>>> t.schuetzenberger_involution()
[[1, 2, 6, 8], [3, 4, 9], [5, 7]]

>>> t = Tableau([])
>>> t.schuetzenberger_involution()
[]

>>> t = StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]])
>>> s = t.schuetzenberger_involution()
```

(continues on next page)

(continued from previous page)

```
>>> s.parent()
Standard tableaux
```

seg()

Return the total number of segments in `self`, as in [Sal2014].

Let T be a tableaux. We define a k-*segment* of T (in the i-th row) to be a maximal consecutive sequence of k-boxes in the i-th row for any $i+1 \le k \le r+1$. Denote the total number of k-segments in T by $\mathrm{seg}(T)$.

REFERENCES:

- [Sal2014]

EXAMPLES:

```
sage: t = Tableau([[1,1,2,3,5],[2,3,5,5],[3,4]])
sage: t.seg()
6

sage: B = crystals.Tableaux("A4", shape=[4,3,2,1])                          #␣
↪needs sage.modules
sage: t = B[31].to_tableau()                                                 #␣
↪needs sage.modules
sage: t.seg()                                                                #␣
↪needs sage.modules
3
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(2),Integer(3),Integer(5)],
↪[Integer(2),Integer(3),Integer(5),Integer(5)],[Integer(3),Integer(4)]])
>>> t.seg()
6

>>> B = crystals.Tableaux("A4", shape=[Integer(4),Integer(3),Integer(2),
↪Integer(1)])                                    # needs sage.modules
>>> t = B[Integer(31)].to_tableau()                                          ␣
↪      # needs sage.modules
>>> t.seg()                                                                #␣
↪needs sage.modules
3
```

shape()

Return the shape of a tableau `self`.

EXAMPLES:

```
sage: Tableau([[1,2,3],[4,5],[6]]).shape()
[3, 2, 1]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)],
↪[Integer(6)]]).shape()
[3, 2, 1]
```

size()

Return the size of the shape of the tableau `self`.

EXAMPLES:

```
sage: Tableau([[1, 4, 6], [2, 5], [3]]).size()
6
sage: Tableau([[1, 3], [2, 4]]).size()
4
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1), Integer(4), Integer(6)], [Integer(2), Integer(5)],␣
↪[Integer(3)]]).size()
6
>>> Tableau([[Integer(1), Integer(3)], [Integer(2), Integer(4)]]).size()
4
```

slide_multiply(*other*)

Multiply two tableaux using jeu de taquin.

This product makes the set of semistandard tableaux into an associative monoid. The empty tableau is the unit in this monoid.

See pp. 15 of [Ful1997].

The same product operation is implemented in a different way in *bump_multiply()*.

EXAMPLES:

```
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t2 = Tableau([[1,2],[3]])
sage: t.slide_multiply(t2)
[[1, 1, 2, 2, 3], [2, 2, 3, 5], [3, 4, 5], [4, 6, 6], [5]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2),Integer(2),Integer(3)],[Integer(2),
↪Integer(3),Integer(5),Integer(5)],[Integer(4),Integer(4),Integer(6)],
↪[Integer(5),Integer(6)]])
>>> t2 = Tableau([[Integer(1),Integer(2)],[Integer(3)]])
>>> t.slide_multiply(t2)
[[1, 1, 2, 2, 3], [2, 2, 3, 5], [3, 4, 5], [4, 6, 6], [5]]
```

socle()

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).socle()
2
sage: Tableau([[1,2,3,4]]).socle()
4
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).socle()
2
>>> Tableau([[Integer(1),Integer(2),Integer(3),Integer(4)]]).socle()
4
```

standardization(*check=True*)

Return the standardization of `self`, assuming `self` is a semistandard tableau.

The standardization of a semistandard tableau T is the standard tableau $\mathrm{st}(T)$ of the same shape as T whose reversed reading word is the standardization of the reversed reading word of T.

The standardization of a word w can be formed by replacing all 1's in w by $1, 2, \ldots, k_1$ from left to right, all 2's in w by $k_1 + 1, k_1 + 2, \ldots, k_2$, and repeating for all letters which appear in w. See also `Word.standard_permutation()`.

INPUT:

- `check` – (default: `True`) check to make sure `self` is semistandard; set to `False` to avoid this check

EXAMPLES:

```
sage: t = Tableau([[1,3,3,4],[2,4,4],[5,16]])
sage: t.standardization()
[[1, 3, 4, 7], [2, 5, 6], [8, 9]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(3),Integer(3),Integer(4)],[Integer(2),
↪Integer(4),Integer(4)],[Integer(5),Integer(16)]])
>>> t.standardization()
[[1, 3, 4, 7], [2, 5, 6], [8, 9]]
```

Standard tableaux are fixed under standardization:

```
sage: all((t == t.standardization() for t in StandardTableaux(6)))
True
sage: t = Tableau([])
sage: t.standardization()
[]
```

```
>>> from sage.all import *
>>> all((t == t.standardization() for t in StandardTableaux(Integer(6))))
True
>>> t = Tableau([])
>>> t.standardization()
[]
```

The reading word of the standardization is the standardization of the reading word:

```
sage: T = SemistandardTableaux(shape=[5,2,2,1], max_entry=4)
sage: all(t.to_word().standard_permutation() == t.standardization().reading_
↪word_permutation() for t in T) # long time
True
```

```
>>> from sage.all import *
>>> T = SemistandardTableaux(shape=[Integer(5),Integer(2),Integer(2),
↪Integer(1)], max_entry=Integer(4))
>>> all(t.to_word().standard_permutation() == t.standardization().reading_
↪word_permutation() for t in T) # long time
True
```

sulzgruber_correspondence()

Return the image of the λ-array `self` under the Sulzgruber correspondence (as a *WeakReversePlanePartition*).

This relies on interpreting `self` as a λ-array in the sense of *hillman_grassl*. See *hillman_grassl* for definitions of the objects involved.

The Sulzgruber correspondence is the map Φ_λ from [Sulzgr2017] Section 7, and is the map ξ_λ^{-1} from [Pak2002] Section 5. It is denoted by $\mathcal{RSK}$ in [Hopkins2017]. It is the inverse of the Pak correspondence (`pak_correspondence()`). The following description of the Sulzgruber correspondence follows [Hopkins2017] (which denotes it by $\mathcal{RSK}$):

Fix a partition λ (see *Partition()*). We draw all partitions and tableaux in English notation.

A λ*-array* will mean a tableau of shape λ whose entries are nonnegative integers. (No conditions on the order of these entries are made. Note that 0 is allowed.)

A *weak reverse plane partition of shape* λ (short: λ*-rpp*) will mean a λ-array whose entries weakly increase along each row and weakly increase along each column.

We shall also use the following notation: If (u, v) is a cell of λ, and if π is a λ-rpp, then:

- the *lower bound* of π at (u, v) (denoted by $\pi_{<(u,v)}$) is defined to be $\max\{\pi_{u-1,v}, \pi_{u,v-1}\}$ (where $\pi_{0,v}$ and $\pi_{u,0}$ are understood to mean 0).
- the *upper bound* of π at (u, v) (denoted by $\pi_{>(u,v)}$) is defined to be $\min\{\pi_{u+1,v}, \pi_{u,v+1}\}$ (where $\pi_{i,j}$ is understood to mean $+\infty$ if (i, j) is not in λ; thus, the upper bound at a corner cell is $+\infty$).
- *toggling* π at (u, v) means replacing the entry $\pi_{u,v}$ of π at (u, v) by $\pi_{<(u,v)} + \pi_{>(u,v)} - \pi_{u,v}$ (this is well-defined as long as (u, v) is not a corner of λ).

Note that every λ-rpp π and every cell (u, v) of λ satisfy $\pi_{<(u,v)} \leq \pi_{u,v} \leq \pi_{>(u,v)}$. Note that toggling a λ-rpp (at a cell that is not a corner) always results in a λ-rpp. Also, toggling is an involution).

The Pak correspondence ξ_λ sends a λ-rpp π to a λ-array $\xi_\lambda(\pi)$. It is defined by recursion on λ (that is, we assume that ξ_μ is already defined for every partition μ smaller than λ), and its definition proceeds as follows:

- If $\lambda = \varnothing$, then ξ_λ is the obvious bijection sending the only $\varnothing$-rpp to the only $\varnothing$-array.
- Pick any corner $c = (i, j)$ of λ, and let μ be the result of removing this corner c from the partition λ. (The exact choice of c is immaterial.)
- Let π' be what remains of π when the corner cell c is removed.
- For each positive integer k such that $(i - k, j - k)$ is a cell of λ, toggle π' at $(i - k, j - k)$. (All these togglings commute, so the order in which they are made is immaterial.)
- Let $M = \xi_\mu(\pi')$.
- Extend the μ-array M to a λ-array M' by adding the cell c and writing the number $\pi_{i,j} - \pi_{<(i,j)}$ into this cell.
- Set $\xi_\lambda(\pi) = M'$.

See also

sulzgruber_correspondence() for the Sulzgruber correspondence as a standalone function.

pak_correspondence() for the inverse map.

EXAMPLES:

```
sage: a = Tableau([[2, 1, 1], [0, 2, 0], [1, 1]])
sage: A = a.sulzgruber_correspondence(); A
[[0, 1, 4], [1, 5, 5], [3, 6]]
sage: A.parent(), a.parent()
(Weak Reverse Plane Partitions, Tableaux)

sage: a = Tableau([[1, 3], [0, 1]])
sage: a.sulzgruber_correspondence()
[[0, 4], [1, 5]]
```

```
>>> from sage.all import *
>>> a = Tableau([[Integer(2), Integer(1), Integer(1)], [Integer(0),␣
↪Integer(2), Integer(0)], [Integer(1), Integer(1)]])
>>> A = a.sulzgruber_correspondence(); A
[[0, 1, 4], [1, 5, 5], [3, 6]]
>>> A.parent(), a.parent()
(Weak Reverse Plane Partitions, Tableaux)

>>> a = Tableau([[Integer(1), Integer(3)], [Integer(0), Integer(1)]])
>>> a.sulzgruber_correspondence()
[[0, 4], [1, 5]]
```

symmetric_group_action_on_entries(*w*)

Return the tableau obtained form this tableau by acting by the permutation `w`.

Let T be a standard tableau of size n, then the action of $w \in S_n$ is defined by permuting the entries of T (recall they are $1, 2, \ldots, n$). In particular, suppose the entry at cell (i, j) is a, then the entry becomes $w(a)$. In general, the resulting tableau wT may *not* be standard.

> **Note**
>
> This is different than *symmetric_group_action_on_values()* which is defined on semistandard tableaux and is guaranteed to return a semistandard tableau.

INPUT:

- `w` – a permutation

EXAMPLES:

```
sage: StandardTableau([[1,2,4],[3,5]]).symmetric_group_action_on_entries(␣
↪Permutation(((4,5))) )
[[1, 2, 5], [3, 4]]
sage: _.category()
Category of elements of Standard tableaux
sage: StandardTableau([[1,2,4],[3,5]]).symmetric_group_action_on_entries(␣
↪Permutation(((1,2))) )
[[2, 1, 4], [3, 5]]
sage: _.category()
Category of elements of Tableaux
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(2),Integer(4)],[Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(5)]]).symmetric_group_action_on_entries( Permutation(((Integer(4),
↪Integer(5)))) )
[[1, 2, 5], [3, 4]]
>>> _.category()
Category of elements of Standard tableaux
>>> StandardTableau([[Integer(1),Integer(2),Integer(4)],[Integer(3),
↪Integer(5)]]).symmetric_group_action_on_entries( Permutation(((Integer(1),
↪Integer(2)))) )
[[2, 1, 4], [3, 5]]
>>> _.category()
Category of elements of Tableaux
```

symmetric_group_action_on_values(*perm*)

Return the image of the semistandard tableau `self` under the action of the permutation `perm` using the Lascoux-Schuetzenberger action of the symmetric group S_n on the semistandard tableaux with ceiling n.

If n is a nonnegative integer, then the Lascoux-Schuetzenberger action is a group action of the symmetric group S_n on the set of semistandard Young tableaux with ceiling n (that is, with entries taken from the set $\{1, 2, \ldots, n\}$). It is defined as follows:

Let $i \in \{1, 2, \ldots, n-1\}$, and let T be a semistandard tableau with ceiling n. Let w be the reading word (`to_word()`) of T. Replace all letters i in w by closing parentheses, and all letters $i+1$ in w by opening parentheses. Whenever an opening parenthesis stands left of a closing parenthesis without there being any parentheses in between (it is allowed to have letters in-between as long as they are not parentheses), consider these two parentheses as matched with each other, and replace them back by the letters $i+1$ and i. Repeat this procedure until there are no more opening parentheses standing left of closing parentheses. Then, let a be the number of opening parentheses in the word, and b the number of closing parentheses (notice that all opening parentheses are right of all closing parentheses). Replace the first a parentheses by the letters i, and replace the remaining b parentheses by the letters $i+1$. Let w' be the resulting word. Let T' be the tableau with the same shape as T but with reading word w'. This tableau T' can be shown to be semistandard. We define the image of T under the action of the simple transposition $s_i = (i, i+1) \in S_n$ to be this tableau T'. It can be shown that these actions of the transpositions $s_1, s_2, \ldots, s_{n-1}$ satisfy the Moore-Coxeter relations of S_n, and thus this extends to a unique action of the symmetric group S_n on the set of semistandard tableaux with ceiling n. This is the Lascoux-Schuetzenberger action.

This action of the symmetric group S_n on the set of all semistandard tableaux of given shape λ with entries in $\{1, 2, \ldots, n\}$ is the one defined in [Loth02] Theorem 5.6.3. In particular, the action of s_i is denoted by σ_i in said source. (Beware of the typo in the definition of σ_i: it should say $\sigma_i(a_i^r a_{i+1}^s) = a_i^s a_{i+1}^r$, not $\sigma_i(a_i^r a_{i+1}^s) = a_i^s a_{i+1}^s$.)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.symmetric_group_action_on_values([1,2,3])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([2,1,3])
[[1, 2, 3, 3], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([3,1,2])
[[1, 2, 2, 2], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([2,3,1])
[[1, 1, 1, 1], [2, 2], [3]]
sage: t.symmetric_group_action_on_values([3,2,1])
[[1, 1, 1, 1], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([1,3,2])
```

(continues on next page)

(continued from previous page)

```
[[1, 1, 2, 2], [2, 2], [3]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(1),Integer(3),Integer(3)],[Integer(2),
↪Integer(3)],[Integer(3)]])
>>> t.symmetric_group_action_on_values([Integer(1),Integer(2),Integer(3)])
[[1, 1, 3, 3], [2, 3], [3]]
>>> t.symmetric_group_action_on_values([Integer(2),Integer(1),Integer(3)])
[[1, 2, 3, 3], [2, 3], [3]]
>>> t.symmetric_group_action_on_values([Integer(3),Integer(1),Integer(2)])
[[1, 2, 2, 2], [2, 3], [3]]
>>> t.symmetric_group_action_on_values([Integer(2),Integer(3),Integer(1)])
[[1, 1, 1, 1], [2, 2], [3]]
>>> t.symmetric_group_action_on_values([Integer(3),Integer(2),Integer(1)])
[[1, 1, 1, 1], [2, 3], [3]]
>>> t.symmetric_group_action_on_values([Integer(1),Integer(3),Integer(2)])
[[1, 1, 2, 2], [2, 2], [3]]
```

to_Gelfand_Tsetlin_pattern()

Return the *Gelfand-Tsetlin pattern* corresponding to `self` when semistandard.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[2,3],[3]])
sage: G = T.to_Gelfand_Tsetlin_pattern(); G
[[3, 2, 1], [2, 1], [1]]
sage: G.to_tableau() == T
True
sage: T = Tableau([[1,3],[2]])
sage: T.to_Gelfand_Tsetlin_pattern()
[[2, 1, 0], [1, 1], [1]]
```

```
>>> from sage.all import *
>>> T = Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(3)],
↪[Integer(3)]])
>>> G = T.to_Gelfand_Tsetlin_pattern(); G
[[3, 2, 1], [2, 1], [1]]
>>> G.to_tableau() == T
True
>>> T = Tableau([[Integer(1),Integer(3)],[Integer(2)]])
>>> T.to_Gelfand_Tsetlin_pattern()
[[2, 1, 0], [1, 1], [1]]
```

to_chain(*max_entry=None*)

Return the chain of partitions corresponding to the (semi)standard tableau `self`.

The optional keyword parameter `max_entry` can be used to customize the length of the chain. Specifically, if this parameter is set to a nonnegative integer `n`, then the chain is constructed from the positions of the letters $1, 2, \ldots, n$ in the tableau.

EXAMPLES:

```
sage: Tableau([[1,2],[3],[4]]).to_chain()
[[], [1], [2], [2, 1], [2, 1, 1]]
sage: Tableau([[1,1],[2]]).to_chain()
[[], [2], [2, 1]]
sage: Tableau([[1,1],[3]]).to_chain()
[[], [2], [2], [2, 1]]
sage: Tableau([]).to_chain()
[[]]
sage: Tableau([[1,1],[2],[3]]).to_chain(max_entry=2)
[[], [2], [2, 1]]
sage: Tableau([[1,1],[2],[3]]).to_chain(max_entry=3)
[[], [2], [2, 1], [2, 1, 1]]
sage: Tableau([[1,1],[2],[3]]).to_chain(max_entry=4)
[[], [2], [2, 1], [2, 1, 1], [2, 1, 1]]
sage: Tableau([[1,1,2],[2,3],[4,5]]).to_chain(max_entry=6)
[[], [2], [3, 1], [3, 2], [3, 2, 1], [3, 2, 2], [3, 2, 2]]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3)],[Integer(4)]]).to_chain()
[[], [1], [2], [2, 1], [2, 1, 1]]
>>> Tableau([[Integer(1),Integer(1)],[Integer(2)]]).to_chain()
[[], [2], [2, 1]]
>>> Tableau([[Integer(1),Integer(1)],[Integer(3)]]).to_chain()
[[], [2], [2], [2, 1]]
>>> Tableau([]).to_chain()
[[]]
>>> Tableau([[Integer(1),Integer(1)],[Integer(2)],[Integer(3)]]).to_chain(max_
↪entry=Integer(2))
[[], [2], [2, 1]]
>>> Tableau([[Integer(1),Integer(1)],[Integer(2)],[Integer(3)]]).to_chain(max_
↪entry=Integer(3))
[[], [2], [2, 1], [2, 1, 1]]
>>> Tableau([[Integer(1),Integer(1)],[Integer(2)],[Integer(3)]]).to_chain(max_
↪entry=Integer(4))
[[], [2], [2, 1], [2, 1, 1], [2, 1, 1]]
>>> Tableau([[Integer(1),Integer(1),Integer(2)],[Integer(2),Integer(3)],
↪[Integer(4),Integer(5)]]).to_chain(max_entry=Integer(6))
[[], [2], [3, 1], [3, 2], [3, 2, 1], [3, 2, 2], [3, 2, 2]]
```

to_list()

Return `self` as a list of lists (not tuples!).

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,4]])
sage: l = t.to_list(); l
[[1, 2], [3, 4]]
sage: l[0][0] = 2
sage: t
[[1, 2], [3, 4]]
```

```
>>> from sage.all import *
>>> t = Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
```

(continues on next page)

(continued from previous page)

```
>>> l = t.to_list(); l
[[1, 2], [3, 4]]
>>> l[Integer(0)][Integer(0)] = Integer(2)
>>> t
[[1, 2], [3, 4]]
```

to_sign_matrix(*max_entry=None*)

Return the sign matrix of `self`.

A sign matrix is an $m \times n$ matrix of 0s, 1s, and -1s such that the partial sums of each column is either 0 or 1 and the partial sums of each row is nonnegative. [Ava2007]

INPUT:

- `max_entry` – nonnegative integer; the maximum allowable number in the tableau (defaults to the largest entry in the tableau if not specified)

EXAMPLES:

```
sage: t = SemistandardTableau([[1,1,1,2,4],[3,3,4],[4,5],[6,6]])
sage: t.to_sign_matrix(6)                                               #␣
↪needs sage.modules
[ 0  0  0  1  0  0]
[ 0  1  0 -1  0  0]
[ 1 -1  0  1  0  0]
[ 0  0  1 -1  1  1]
[ 0  0  0  1 -1  0]
sage: t = Tableau([[1,2,4],[3,5]])
sage: t.to_sign_matrix(7)                                               #␣
↪needs sage.modules
[ 0  0  0  1  0  0  0]
[ 0  1  0 -1  1  0  0]
[ 1 -1  1  0 -1  0  0]
sage: t = Tableau([(4,5,4,3),(2,1,3)])
sage: t.to_sign_matrix(5)                                               #␣
↪needs sage.modules
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[ 1  0 -1 -1  1]
[-1  1  0  1 -1]
sage: s = Tableau([(1,0,-2,4),(3,4,5)])
sage: s.to_sign_matrix(6)
Traceback (most recent call last):
...
ValueError: the entries must be nonnegative integers
```

```
>>> from sage.all import *
>>> t = SemistandardTableau([[Integer(1),Integer(1),Integer(1),Integer(2),
↪Integer(4)],[Integer(3),Integer(3),Integer(4)],[Integer(4),Integer(5)],
↪[Integer(6),Integer(6)]])
>>> t.to_sign_matrix(Integer(6))                                        ␣
↪      # needs sage.modules
[ 0  0  0  1  0  0]
[ 0  1  0 -1  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 1 -1  0  1  0  0]
[ 0  0  1 -1  1  1]
[ 0  0  0  1 -1  0]
>>> t = Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3),Integer(5)]])
>>> t.to_sign_matrix(Integer(7))                                          ␣
↪     # needs sage.modules
[ 0  0  0  1  0  0  0]
[ 0  1  0 -1  1  0  0]
[ 1 -1  1  0 -1  0  0]
>>> t = Tableau([(Integer(4),Integer(5),Integer(4),Integer(3)),(Integer(2),
↪Integer(1),Integer(3))])
>>> t.to_sign_matrix(Integer(5))                                          ␣
↪     # needs sage.modules
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[ 1  0 -1 -1  1]
[-1  1  0  1 -1]
>>> s = Tableau([(Integer(1),Integer(0),-Integer(2),Integer(4)),(Integer(3),
↪Integer(4),Integer(5))])
>>> s.to_sign_matrix(Integer(6))
Traceback (most recent call last):
...
ValueError: the entries must be nonnegative integers
```

to_word()

An alias for `to_word_by_row()`.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word()
word: 3412
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word()
word: 325146
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).to_word()
word: 3412
>>> Tableau([[Integer(1), Integer(4), Integer(6)], [Integer(2), Integer(5)],␣
↪[Integer(3)]]).to_word()
word: 325146
```

to_word_by_column()

Return the word obtained from a column reading of the tableau `self` (starting with the leftmost column, reading every column from bottom to top).

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word_by_column()
word: 3142
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word_by_column()
word: 321546
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).to_word_by_
↪column()
word: 3142
>>> Tableau([[Integer(1), Integer(4), Integer(6)], [Integer(2), Integer(5)],␣
↪[Integer(3)]]).to_word_by_column()
word: 321546
```

to_word_by_row()

Return the word obtained from a row reading of the tableau `self` (starting with the lowermost row, reading every row from left to right).

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word_by_row()
word: 3412
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word_by_row()
word: 325146
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).to_word_by_
↪row()
word: 3412
>>> Tableau([[Integer(1), Integer(4), Integer(6)], [Integer(2), Integer(5)],␣
↪[Integer(3)]]).to_word_by_row()
word: 325146
```

vertical_flip()

Return the tableau obtained by vertically flipping the tableau `self`.

This only works for rectangular tableaux.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).vertical_flip()
[[3, 4], [1, 2]]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).vertical_flip()
[[3, 4], [1, 2]]
```

weight()

Return the weight of the tableau `self`. Trailing zeroes are omitted when returning the weight.

The weight of a tableau T is the sequence $(a_1, a_2, a_3, \ldots)$, where a_k is the number of entries of T equal to k. This sequence contains only finitely many nonzero entries.

The weight of a tableau T is the same as the weight of the reading word of T, for any reading order.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).weight()
[1, 1, 1, 1]

sage: Tableau([]).weight()
```

(continues on next page)

(continued from previous page)

```
[]

sage: Tableau([[1,3,3,7],[4,2],[2,3]]).weight()
[1, 2, 3, 1, 0, 0, 1]
```

```
>>> from sage.all import *
>>> Tableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).weight()
[1, 1, 1, 1]

>>> Tableau([]).weight()
[]

>>> Tableau([[Integer(1),Integer(3),Integer(3),Integer(7)],[Integer(4),
↪Integer(2)],[Integer(2),Integer(3)]]).weight()
[1, 2, 3, 1, 0, 0, 1]
```

class sage.combinat.tableau.**Tableau_class**(*parent*, *t*, *check=True*)

Bases: *Tableau*

This exists solely for unpickling `Tableau_class` objects.

class sage.combinat.tableau.**Tableaux**

Bases: `UniqueRepresentation`, `Parent`

A factory class for the various classes of tableaux.

INPUT:

- `n` – (optional) nonnegative integer

OUTPUT:

- If `n` is specified, the class of tableaux of size `n`. Otherwise, the class of all tableaux.

A tableau in Sage is a finite list of lists, whose lengths are weakly decreasing, or an empty list, representing the empty tableau. The entries of a tableau can be any Sage objects. Because of this, no enumeration through the set of Tableaux is possible.

EXAMPLES:

```
sage: T = Tableaux(); T
Tableaux
sage: T3 = Tableaux(3); T3
Tableaux of size 3
sage: [['a','b']] in T
True
sage: [['a','b']] in T3
False
sage: t = T3([[1,1,1]]); t
[[1, 1, 1]]
sage: t in T
True
sage: t.parent()
Tableaux of size 3
sage: T([]) # the empty tableau
[]
```

(continues on next page)

(continued from previous page)

```
sage: T.category()
Category of sets
```

```
>>> from sage.all import *
>>> T = Tableaux(); T
Tableaux
>>> T3 = Tableaux(Integer(3)); T3
Tableaux of size 3
>>> [['a','b']] in T
True
>>> [['a','b']] in T3
False
>>> t = T3([[Integer(1),Integer(1),Integer(1)]]); t
[[1, 1, 1]]
>>> t in T
True
>>> t.parent()
Tableaux of size 3
>>> T([]) # the empty tableau
[]
>>> T.category()
Category of sets
```

See also

- *Tableau*
- *SemistandardTableaux*
- *SemistandardTableau*
- *StandardTableaux*
- *StandardTableau*

Element

alias of *Tableau*

options = Current options for Tableaux - ascii_art: repr - convention: English - display: list - latex: diagram

class sage.combinat.tableau.**Tableaux_all**

Bases: *Tableaux*

Initialize the class of all tableaux.

an_element()

Return a particular element of the class.

class sage.combinat.tableau.**Tableaux_size**(n)

Bases: *Tableaux*

Tableaux of a fixed size n.

an_element()

Return a particular element of the class.

sage.combinat.tableau.**from_chain**(*chain*)

Return a semistandard tableau from a chain of partitions.

EXAMPLES:

```
sage: from sage.combinat.tableau import from_chain
sage: from_chain([[], [2], [2, 1], [3, 2, 1]])
[[1, 1, 3], [2, 3], [3]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau import from_chain
>>> from_chain([[], [Integer(2)], [Integer(2), Integer(1)], [Integer(3),␣
↪Integer(2), Integer(1)]])
[[1, 1, 3], [2, 3], [3]]
```

sage.combinat.tableau.**from_shape_and_word**(*shape*, *w*, *convention='French'*)

Return a tableau from a shape and word.

INPUT:

- `shape` – a partition
- `w` – a word whose length equals that of the partition
- `convention` – string (default: `'French'`); can take values `'French'` or `'English'`

OUTPUT:

A tableau, whose shape is `shape` and whose reading word is `w`. If the `convention` is specified as `'French'`, the reading word is to be read starting from the top row in French convention (= the bottom row in English convention). If the `convention` is specified as `'English'`, the reading word is to be read starting with the top row in English convention.

EXAMPLES:

```
sage: from sage.combinat.tableau import from_shape_and_word
sage: t = Tableau([[1, 3], [2], [4]])
sage: shape = t.shape(); shape
[2, 1, 1]
sage: word = t.to_word(); word
word: 4213
sage: from_shape_and_word(shape, word)
[[1, 3], [2], [4]]
sage: word = Word(flatten(t))
sage: from_shape_and_word(shape, word, convention='English')
[[1, 3], [2], [4]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau import from_shape_and_word
>>> t = Tableau([[Integer(1), Integer(3)], [Integer(2)], [Integer(4)]])
>>> shape = t.shape(); shape
[2, 1, 1]
>>> word = t.to_word(); word
word: 4213
```

(continues on next page)

(continued from previous page)

```
>>> from_shape_and_word(shape, word)
[[1, 3], [2], [4]]
>>> word = Word(flatten(t))
>>> from_shape_and_word(shape, word, convention='English')
[[1, 3], [2], [4]]
```

sage.combinat.tableau.**symmetric_group_action_on_values**(*word*, *perm*)

Return the image of the word `word` under the Lascoux-Schuetzenberger action of the permutation `perm`.

See `Tableau.symmetric_group_action_on_values()` for the definition of the Lascoux-Schuetzenberger action on semistandard tableaux. The transformation that the reading word of the tableau undergoes in said definition is precisely the Lascoux-Schuetzenberger action on words.

EXAMPLES:

```
sage: from sage.combinat.tableau import symmetric_group_action_on_values
sage: symmetric_group_action_on_values([1,1,1],[1,3,2])
[1, 1, 1]
sage: symmetric_group_action_on_values([1,1,1],[2,1,3])
[2, 2, 2]
sage: symmetric_group_action_on_values([1,2,1],[2,1,3])
[2, 2, 1]
sage: symmetric_group_action_on_values([2,2,2],[2,1,3])
[1, 1, 1]
sage: symmetric_group_action_on_values([2,1,2],[2,1,3])
[2, 1, 1]
sage: symmetric_group_action_on_values([2,2,3,1,1,2,2,3],[1,3,2])
[2, 3, 3, 1, 1, 2, 3, 3]
sage: symmetric_group_action_on_values([2,1,1],[2,1])
[2, 1, 2]
sage: symmetric_group_action_on_values([2,2,1],[2,1])
[1, 2, 1]
sage: symmetric_group_action_on_values([1,2,1],[2,1])
[2, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau import symmetric_group_action_on_values
>>> symmetric_group_action_on_values([Integer(1),Integer(1),Integer(1)],
↪[Integer(1),Integer(3),Integer(2)])
[1, 1, 1]
>>> symmetric_group_action_on_values([Integer(1),Integer(1),Integer(1)],
↪[Integer(2),Integer(1),Integer(3)])
[2, 2, 2]
>>> symmetric_group_action_on_values([Integer(1),Integer(2),Integer(1)],
↪[Integer(2),Integer(1),Integer(3)])
[2, 2, 1]
>>> symmetric_group_action_on_values([Integer(2),Integer(2),Integer(2)],
↪[Integer(2),Integer(1),Integer(3)])
[1, 1, 1]
>>> symmetric_group_action_on_values([Integer(2),Integer(1),Integer(2)],
↪[Integer(2),Integer(1),Integer(3)])
[2, 1, 1]
>>> symmetric_group_action_on_values([Integer(2),Integer(2),Integer(3),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(2),Integer(2),Integer(3)],[Integer(1),Integer(3),Integer(2)])
[2, 3, 3, 1, 1, 2, 3, 3]
>>> symmetric_group_action_on_values([Integer(2),Integer(1),Integer(1)],
↪[Integer(2),Integer(1)])
[2, 1, 2]
>>> symmetric_group_action_on_values([Integer(2),Integer(2),Integer(1)],
↪[Integer(2),Integer(1)])
[1, 2, 1]
>>> symmetric_group_action_on_values([Integer(1),Integer(2),Integer(1)],
↪[Integer(2),Integer(1)])
[2, 2, 1]
```

sage.combinat.tableau.**unmatched_places**(*w*, *open*, *close*)

Given a word `w` and two letters `open` and `close` to be treated as opening and closing parentheses (respectively), return a pair `(xs, ys)` that encodes the positions of the unmatched parentheses after the standard parenthesis matching procedure is applied to `w`.

More precisely, `xs` will be the list of all `i` such that `w[i]` is an unmatched closing parenthesis, while `ys` will be the list of all `i` such that `w[i]` is an unmatched opening parenthesis. Both lists returned are in increasing order.

EXAMPLES:

```
sage: from sage.combinat.tableau import unmatched_places
sage: unmatched_places([2,2,2,1,1,1],2,1)
([], [])
sage: unmatched_places([1,1,1,2,2,2],2,1)
([0, 1, 2], [3, 4, 5])
sage: unmatched_places([], 2, 1)
([], [])
sage: unmatched_places([1,2,4,6,2,1,5,3],2,1)
([0], [1])
sage: unmatched_places([2,2,1,2,4,6,2,1,5,3], 2, 1)
([], [0, 3])
sage: unmatched_places([3,1,1,1,2,1,2], 2, 1)
([1, 2, 3], [6])
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau import unmatched_places
>>> unmatched_places([Integer(2),Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(1)],Integer(2),Integer(1))
([], [])
>>> unmatched_places([Integer(1),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(2)],Integer(2),Integer(1))
([0, 1, 2], [3, 4, 5])
>>> unmatched_places([], Integer(2), Integer(1))
([], [])
>>> unmatched_places([Integer(1),Integer(2),Integer(4),Integer(6),Integer(2),
↪Integer(1),Integer(5),Integer(3)],Integer(2),Integer(1))
([0], [1])
>>> unmatched_places([Integer(2),Integer(2),Integer(1),Integer(2),Integer(4),
↪Integer(6),Integer(2),Integer(1),Integer(5),Integer(3)], Integer(2), Integer(1))
([], [0, 3])
>>> unmatched_places([Integer(3),Integer(1),Integer(1),Integer(1),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(2)], Integer(2), Integer(1))
([1, 2, 3], [6])
```

5.1.346 Residue sequences of tableaux

A *residue sequence* for a `StandardTableau`, or `StandardTableauTuple`, of size n is an n-tuple $(i_1, i_2, \ldots, i_n)$ of elements of $\mathbf{Z}/e\mathbf{Z}$ for some positive integer $e \geq 1$. Such sequences arise in the representation theory of the symmetric group and the closely related cyclotomic Hecke algebras, and cyclotomic quiver Hecke algebras, where the residue sequences play a similar role to weights in the representations of Lie groups and Lie algebras. These Hecke algebras are semisimple when e is "large enough" and in these cases residue sequences are essentially the same as content sequences (see `sage.combinat.partition.Partition.content()`) and it is not difficult to see that residue sequences are in bijection with the set of standard tableaux. In the non-semisimple case, when e is "small", different standard tableaux can have the same residue sequence. In this case the residue sequences describe how to decompose modules into generalised eigenspaces for the Jucys-Murphy elements for these algebras.

By definition, if t is a `StandardTableau` of size n then the residue sequence of t is the n-tuple $(i_1, \ldots, i_n)$ where $i_m = c - r + e\mathbf{Z}$, if m appears in row r and column c of t. If p is prime then such sequence arise in the representation theory of the symmetric group n characteristic p. More generally, e-residue sequences arise in he representation theory of the Iwahori-Hecke algebra (see `IwahoriHeckeAlgebra`) the symmetric group with Hecke parameter at an e-th root of unity.

More generally, the e-residue sequence of a `StandardTableau` of size n and level l is the n-tuple $(i_1, \ldots, i_n)$ determined by e and a *multicharge* $\kappa = (\kappa_1, \ldots, \kappa_l)$ by setting $i_m = \kappa_k + c - r + e\mathbf{Z}$, if m appears in component k, row r and column c of t. These sequences arise in the representation theory of the cyclotomic Hecke algebras of type A, which are also known as Ariki-Koike algebras.

The residue classes are constructed from standard tableaux:

```
sage: StandardTableau([[1,2],[3,4]]).residue_sequence(2)
2-residue sequence (0,1,1,0) with multicharge (0)
sage: StandardTableau([[1,2],[3,4]]).residue_sequence(3)
3-residue sequence (0,1,2,0) with multicharge (0)

sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue_sequence(3,[0,0])
3-residue sequence (0,1,2,0,0) with multicharge (0,0)
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue_sequence(3,[0,1])
3-residue sequence (1,2,0,1,0) with multicharge (0,1)
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue_sequence(3,[0,2])
3-residue sequence (2,0,1,2,0) with multicharge (0,2)
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).residue_
↪sequence(Integer(2))
2-residue sequence (0,1,1,0) with multicharge (0)
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).residue_
↪sequence(Integer(3))
3-residue sequence (0,1,2,0) with multicharge (0)

>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue_sequence(Integer(3),[Integer(0),Integer(0)])
3-residue sequence (0,1,2,0,0) with multicharge (0,0)
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue_sequence(Integer(3),[Integer(0),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
3-residue sequence (1,2,0,1,0) with multicharge (0,1)
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue_sequence(Integer(3),[Integer(0),Integer(2)])
3-residue sequence (2,0,1,2,0) with multicharge (0,2)
```

One of the most useful functions of a `ResidueSequence` is that it can return the `StandardTableaux_residue` and `StandardTableaux_residue_shape` that contain all of the tableaux with this residue sequence. Again, these are best accessed via the standard tableaux classes:

```
sage: res = StandardTableau([[1,2],[3,4]]).residue_sequence(2)
sage: res.standard_tableaux()
Standard tableaux with 2-residue sequence (0,1,1,0) and multicharge (0)
sage: res.standard_tableaux()[:]
[[[1, 2, 4], [3]],
[[1, 2], [3, 4]],
[[1, 2], [3], [4]],
[[1, 3, 4], [2]],
[[1, 3], [2, 4]],
[[1, 3], [2], [4]]]
sage: res.standard_tableaux(shape=[4])
Standard (4)-tableaux with 2-residue sequence (0,1,1,0) and multicharge (0)
sage: res.standard_tableaux(shape=[4])[:]
[]

sage: res=StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue_sequence(3,[0,0])
sage: res.standard_tableaux()
Standard tableaux with 3-residue sequence (0,1,2,0,0) and multicharge (0,0)
sage: res.standard_tableaux(shape=[[1],[2,2]])[:]
[([[5]], [[1, 2], [3, 4]]), ([[4]], [[1, 2], [3, 5]])]
```

```
>>> from sage.all import *
>>> res = StandardTableau([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).residue_
↪sequence(Integer(2))
>>> res.standard_tableaux()
Standard tableaux with 2-residue sequence (0,1,1,0) and multicharge (0)
>>> res.standard_tableaux()[:]
[[[1, 2, 4], [3]],
[[1, 2], [3, 4]],
[[1, 2], [3], [4]],
[[1, 3, 4], [2]],
[[1, 3], [2, 4]],
[[1, 3], [2], [4]]]
>>> res.standard_tableaux(shape=[Integer(4)])
Standard (4)-tableaux with 2-residue sequence (0,1,1,0) and multicharge (0)
>>> res.standard_tableaux(shape=[Integer(4)])[:]
[]

>>> res=StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue_sequence(Integer(3),[Integer(0),Integer(0)])
>>> res.standard_tableaux()
Standard tableaux with 3-residue sequence (0,1,2,0,0) and multicharge (0,0)
>>> res.standard_tableaux(shape=[[Integer(1)],[Integer(2),Integer(2)]])[:]
```

(continues on next page)

(continued from previous page)

```
[([[5]], [[1, 2], [3, 4]]), ([[4]], [[1, 2], [3, 5]])]
```

These residue sequences are particularly useful in the graded representation theory of the cyclotomic KLR algebras and the cyclotomic Hecke algebras of type~A; see [DJM1998] and [BK2009].

This module implements the following classes:

- *ResidueSequence*
- *ResidueSequences*

See also

- *Partitions*
- *PartitionTuples*
- *StandardTableaux_residue*
- *StandardTableaux_residue_shape*
- *RowStandardTableauTuples_residue*
- *RowStandardTableauTuples_residue_shape*
- *StandardTableaux*
- *StandardTableauTuples*
- *Tableaux*
- *TableauTuples*

Todo

Strictly speaking this module implements residue sequences of type $A_e^{(1)}$. Residue sequences of other types also need to be implemented.

AUTHORS:

- Andrew Mathas (2016-07-01): Initial version

class sage.combinat.tableau_residues.**ResidueSequence**(*parent*, *residues*, *check*)

Bases: ClonableArray

A residue sequence.

The *residue sequence* of a tableau t (of partition or partition tuple shape) is the sequence $(i_1, i_2, \ldots, i_n)$ where i_k is the residue of l in t, for $k = 1, 2, \ldots, n$, where n is the size of t. Residue sequences are important in the representation theory of the cyclotomic Hecke algebras of type $G(r, 1, n)$, and of the cyclotomic quiver Hecke algebras, because they determine the eigenvalues of the Jucys-Murphy elements upon all modules. More precisely, they index and completely determine the irreducible representations of the (cyclotomic) Gelfand-Tsetlin algebras.

Rather than being called directly, residue sequences are best accessed via the standard tableaux classes *StandardTableau* and *StandardTableauTuple*.

INPUT:

Can be of the form:

- `ResidueSequence(e, res)`,
- `ResidueSequence(e, multicharge, res)`,

where `e` is a positive integer not equal to 1 and `res` is a sequence of integers (the residues).

EXAMPLES:

```
sage: res = StandardTableauTuple([[[1,3],[6]],[[2,7],[4],[5]]]).residue_
↪sequence(3,(0,5))
sage: res
3-residue sequence (0,2,1,1,0,2,0) with multicharge (0,2)
sage: res.quantum_characteristic()
3
sage: res.level()
2
sage: res.size()
7
sage: res.residues()
[0, 2, 1, 1, 0, 2, 0]
sage: res.restrict(2)
3-residue sequence (0,2) with multicharge (0,2)
sage: res.standard_tableaux([[2,1],[1],[2,1]])
Standard (2,1|1|2,1)-tableaux with 3-residue sequence (0,2,1,1,0,2,0) and␣
↪multicharge (0,2)
sage: res.standard_tableaux([[2,2],[3]]).list()
[]
sage: res.standard_tableaux([[2,2],[3]])[:]
[]
sage: res.standard_tableaux()
Standard tableaux with 3-residue sequence (0,2,1,1,0,2,0) and multicharge (0,2)
sage: res.standard_tableaux()[:10]
[([[1, 3, 6, 7], [2, 5], [4]], []),
 ([[1, 3, 6], [2, 5], [4], [7]], []),
 ([[1, 3], [2, 5], [4, 6], [7]], []),
 ([[1, 3], [2, 5], [4], [7]], [[6]]),
 ([[1, 3], [2, 5], [4]], [[6, 7]]),
 ([[1, 3, 6, 7], [2], [4], [5]], []),
 ([[1, 3, 6], [2, 7], [4], [5]], []),
 ([[1, 3], [2, 7], [4], [5], [6]], []),
 ([[1, 3], [2, 7], [4], [5]], [[6]]),
 ([[1, 3], [2], [4], [5]], [[6, 7]])]
```

```
>>> from sage.all import *
>>> res = StandardTableauTuple([[[Integer(1),Integer(3)],[Integer(6)]],
↪[[Integer(2),Integer(7)],[Integer(4)],[Integer(5)]]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(5)))
>>> res
3-residue sequence (0,2,1,1,0,2,0) with multicharge (0,2)
>>> res.quantum_characteristic()
3
>>> res.level()
2
>>> res.size()
7
```

(continues on next page)

(continued from previous page)

```
>>> res.residues()
[0, 2, 1, 1, 0, 2, 0]
>>> res.restrict(Integer(2))
3-residue sequence (0,2) with multicharge (0,2)
>>> res.standard_tableaux([[Integer(2),Integer(1)],[Integer(1)],[Integer(2),
↪Integer(1)]])
Standard (2,1|1|2,1)-tableaux with 3-residue sequence (0,2,1,1,0,2,0) and␣
↪multicharge (0,2)
>>> res.standard_tableaux([[Integer(2),Integer(2)],[Integer(3)]]).list()
[]
>>> res.standard_tableaux([[Integer(2),Integer(2)],[Integer(3)]])[:]
[]
>>> res.standard_tableaux()
Standard tableaux with 3-residue sequence (0,2,1,1,0,2,0) and multicharge (0,2)
>>> res.standard_tableaux()[:Integer(10)]
[([[1, 3, 6, 7], [2, 5], [4]], []),
 ([[1, 3, 6], [2, 5], [4], [7]], []),
 ([[1, 3], [2, 5], [4, 6], [7]], []),
 ([[1, 3], [2, 5], [4], [7]], [[6]]),
 ([[1, 3], [2, 5], [4]], [[6, 7]]),
 ([[1, 3, 6, 7], [2], [4], [5]], []),
 ([[1, 3, 6], [2, 7], [4], [5]], []),
 ([[1, 3], [2, 7], [4], [5], [6]], []),
 ([[1, 3], [2, 7], [4], [5]], [[6]]),
 ([[1, 3], [2], [4], [5]], [[6, 7]])]
```

The TestSuite fails `_test_pickling` because `__getitem__` does not support slices, so we skip this.

base_ring()

Return the base ring for the residue sequence.

If the *quantum_characteristic()* of the residue sequence `self` is e then the base ring for the sequence is $\mathbf{Z}/e\mathbf{Z}$, or $\mathbf{Z}$ if $e = 0$.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3, (0,0,1), [0,0,1,1,2,2,3,3]).base_ring()
Ring of integers modulo 3
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3), (Integer(0),Integer(0),Integer(1)),␣
↪[Integer(0),Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3)]).base_ring()
Ring of integers modulo 3
```

block()

Return a dictionary β that determines the block associated to the residue sequence `self`.

Two Specht modules for a cyclotomic Hecke algebra of type A belong to the same block, in this sense, if and only if the residue sequences of their standard tableaux have the same block in this sense. The blocks of these algebras are actually indexed by positive roots in the root lattice of an affine special linear group. Instead of than constructing the root lattice, this method simply returns a dictionary β where the keys are residues i and where the value of the key i is equal to the numbers of nodes in the residue sequence `self` that are equal to i. The dictionary β corresponds to the positive root:

$$\sum_{i\in I}\beta_i\alpha_i \in Q^+,$$

These positive roots also index the blocks of the cyclotomic KLR algebras of type A.

We return a dictionary because when the *quantum_characteristic()* is 0, the Cartan type is A_∞, in which case the simple roots are indexed by the integers, which is infinite.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3, [0,0,0], [0,1,2,0,1,2,0,1,2]).block()
{0: 3, 1: 3, 2: 3}
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3), [Integer(0),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(2),Integer(0),Integer(1),Integer(2),
↪Integer(0),Integer(1),Integer(2)]).block()
{0: 3, 1: 3, 2: 3}
```

check()

Raise a `ValueError` if `self` is not a residue sequence.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3, [0,0,1], [0,0,1,1,2,2,3,3]).check()
sage: ResidueSequence(3, [0,0,1], [2,0,1,1,2,2,3,3]).check()
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3), [Integer(0),Integer(0),Integer(1)],␣
↪[Integer(0),Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3)]).check()
>>> ResidueSequence(Integer(3), [Integer(0),Integer(0),Integer(1)],␣
↪[Integer(2),Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3)]).check()
```

level()

Return the level of the residue sequence. That is, the level of the corresponding (tuples of) standard tableaux.

The *level* of a residue sequence is the length of its *multicharge()*. This is the same as the level of the *standard_tableaux()* that belong to the residue class of tableaux determined by `self`.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3, (0,0,1), [0,0,1,1,2,2,3,3]).level()
3
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3), (Integer(0),Integer(0),Integer(1)),␣
↪[Integer(0),Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3)]).level()
3
```

multicharge()

Return the multicharge for the residue sequence `self`.

The e-residue sequences are associated with a cyclotomic Hecke algebra with Hecke parameter q of *quantum_characteristic()* e and multicharge $(\kappa_1, \ldots, \kappa_l)$. This means that the cyclotomic parameters of the Hecke algebra are $q^{\kappa_1}, \ldots, q^{\kappa_l}$. Equivalently, the Hecke algebra is determined by the dominant weight

$$\sum_{r \in \mathbf{Z}/e\mathbf{Z}} \kappa_r \Lambda_r \in P^+.$$

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3, (0,0,1), [0,0,1,1,2,2,3,3]).multicharge()
(0, 0, 1)
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3), (Integer(0),Integer(0),Integer(1)),␣
↪[Integer(0),Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3)]).multicharge()
(0, 0, 1)
```

negative()

Return the negative of the residue sequence `self`.

That is, if `self` is the residue sequence $(i_1, \ldots, i_n)$ then return $(-i_1, \ldots, -i_n)$. Taking the negative residue sequences is a shadow of tensoring with the sign representation from the cyclotomic Hecke algebras of type A.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3,[0,0,1],[0,0,1,1,2,2,3,3]).negative()
3-residue sequence (0,0,2,2,1,1,0,0) with multicharge (0,0,1)
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3),[Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
↪Integer(3)]).negative()
3-residue sequence (0,0,2,2,1,1,0,0) with multicharge (0,0,1)
```

quantum_characteristic()

Return the quantum characteristic of the residue sequence `self`.

The e-residue sequences are associated with a cyclotomic Hecke algebra that has a parameter q of *quantum characteristic* e. This is the smallest positive integer such that $1 + q + \cdots + q^{e-1} = 0$, or $e = 0$ if no such integer exists.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3, (0,0,1), [0,0,1,1,2,2,3,3]).quantum_characteristic()
3
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3), (Integer(0),Integer(0),Integer(1)),␣
→[Integer(0),Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),
→Integer(3),Integer(3)]).quantum_characteristic()
3
```

residues()

Return a list of the residue sequence.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3,(0,0,1),[0,0,1,1,2,2,3,3]).residues()
[0, 0, 1, 1, 2, 2, 0, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(1)),[Integer(0),
→Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
→Integer(3)]).residues()
[0, 0, 1, 1, 2, 2, 0, 0]
```

restrict(m)

Return the subsequence of this sequence of length m.

The residue sequence `self` is of the form $(r_1, \ldots, r_n)$. The function returns the residue sequence $(r_1, \ldots, r_m)$, with the same *quantum_characteristic()* and *multicharge()*.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3,(0,0,1),[0,0,1,1,2,2,3,3]).restrict(7)
3-residue sequence (0,0,1,1,2,2,0) with multicharge (0,0,1)
sage: ResidueSequence(3,(0,0,1),[0,0,1,1,2,2,3,3]).restrict(6)
3-residue sequence (0,0,1,1,2,2) with multicharge (0,0,1)
sage: ResidueSequence(3,(0,0,1),[0,0,1,1,2,2,3,3]).restrict(4)
3-residue sequence (0,0,1,1) with multicharge (0,0,1)
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(1)),[Integer(0),
→Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
→Integer(3)]).restrict(Integer(7))
3-residue sequence (0,0,1,1,2,2,0) with multicharge (0,0,1)
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(1)),[Integer(0),
→Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
→Integer(3)]).restrict(Integer(6))
```

(continues on next page)

(continued from previous page)

```
3-residue sequence (0,0,1,1,2,2) with multicharge (0,0,1)
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(1)),[Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
↪Integer(3)]).restrict(Integer(4))
3-residue sequence (0,0,1,1) with multicharge (0,0,1)
```

restrict_row(*cell*, *row*)

Return a residue sequence for the tableau obtained by swapping the row in ending in *cell* with the row that is row rows above it and which has the same length.

The residue sequence `self` is of the form $(r_1, \ldots, r_n)$. The function returns the residue sequence $(r_1, \ldots, r_m)$, with the same `quantum_characteristic()` and `multicharge()`.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3, [0,1,2,2,0,1]).restrict_row((1,2),1)
3-residue sequence (2,0,1,0,1) with multicharge (0)
sage: ResidueSequence(3, [1,0], [0,1,2,2,0,1]).restrict_row((1,1,2),1)
3-residue sequence (2,0,1,0,1) with multicharge (1,0)
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3), [Integer(0),Integer(1),Integer(2),Integer(2),
↪Integer(0),Integer(1)]).restrict_row((Integer(1),Integer(2)),Integer(1))
3-residue sequence (2,0,1,0,1) with multicharge (0)
>>> ResidueSequence(Integer(3), [Integer(1),Integer(0)], [Integer(0),
↪Integer(1),Integer(2),Integer(2),Integer(0),Integer(1)]).restrict_
↪row((Integer(1),Integer(1),Integer(2)),Integer(1))
3-residue sequence (2,0,1,0,1) with multicharge (1,0)
```

row_standard_tableaux(*shape=None*)

Return the residue-class of row standard tableaux that have residue sequence `self`.

INPUT:

- `shape` – (optional) a partition or partition tuple of the correct level

OUTPUT:

An iterator for the row standard tableaux with this residue sequence. If the `shape` is given then only tableaux of this shape are returned, otherwise all of the full residue-class of row standard tableaux, or row standard tableaux tuples, is returned. The residue sequence `self` specifies the `multicharge()` of the tableaux which, in turn, determines the `level()` of the tableaux in the residue class.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3,(0,0,0),[0,1,2,0,1,2,0,1,2]).row_standard_tableaux()
Row standard tableaux with 3-residue sequence (0,1,2,0,1,2,0,1,2) and␣
↪multicharge (0,0,0)
sage: ResidueSequence(3,(0,0,0),[0,1,2,0,1,2,0,1,2]).row_standard_
↪tableaux([[3],[3],[3]])
Row standard (3|3|3)-tableaux with 3-residue sequence (0,1,2,0,1,2,0,1,2) and␣
↪multicharge (0,0,0)
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(0)),[Integer(0),
↪Integer(1),Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),
↪Integer(1),Integer(2)]).row_standard_tableaux()
Row standard tableaux with 3-residue sequence (0,1,2,0,1,2,0,1,2) and␣
↪multicharge (0,0,0)
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(0)),[Integer(0),
↪Integer(1),Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),
↪Integer(1),Integer(2)]).row_standard_tableaux([[Integer(3)],[Integer(3)],
↪[Integer(3)]])
Row standard (3|3|3)-tableaux with 3-residue sequence (0,1,2,0,1,2,0,1,2) and␣
↪multicharge (0,0,0)
```

size()

Return the size of the residue sequence.

This is the size, or length, of the residue sequence, which is the same as the size of the *standard_tableaux()* that belong to the residue class of tableaux determined by `self`.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3, (0,0,1), [0,0,1,1,2,2,3,3]).size()
8
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3), (Integer(0),Integer(0),Integer(1)),␣
↪[Integer(0),Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3)]).size()
8
```

standard_tableaux(*shape=None*)

Return the residue-class of standard tableaux that have residue sequence `self`.

INPUT:

- `shape` – (optional) a partition or partition tuple of the correct level

OUTPUT:

An iterator for the standard tableaux with this residue sequence. If the `shape` is given then only tableaux of this shape are returned, otherwise all of the full residue-class of standard tableaux, or standard tableaux tuples, is returned. The residue sequence `self` specifies the *multicharge()* of the tableaux which, in turn, determines the *level()* of the tableaux in the residue class.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3,(0,0,0),[0,1,2,0,1,2,0,1,2]).standard_tableaux()
Standard tableaux with 3-residue sequence (0,1,2,0,1,2,0,1,2) and multicharge␣
↪(0,0,0)
sage: ResidueSequence(3,(0,0,0),[0,1,2,0,1,2,0,1,2]).standard_tableaux([[3],
↪[3],[3]])
```

(continues on next page)

(continued from previous page)

```
Standard (3|3|3)-tableaux with 3-residue sequence (0,1,2,0,1,2,0,1,2) and␣
↪multicharge (0,0,0)
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(0)),[Integer(0),
↪Integer(1),Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),
↪Integer(1),Integer(2)]).standard_tableaux()
Standard tableaux with 3-residue sequence (0,1,2,0,1,2,0,1,2) and multicharge␣
↪(0,0,0)
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(0)),[Integer(0),
↪Integer(1),Integer(2),Integer(0),Integer(1),Integer(2),Integer(0),
↪Integer(1),Integer(2)]).standard_tableaux([[Integer(3)],[Integer(3)],
↪[Integer(3)]])
Standard (3|3|3)-tableaux with 3-residue sequence (0,1,2,0,1,2,0,1,2) and␣
↪multicharge (0,0,0)
```

swap_residues(*i*, *j*)

Return the *new* residue sequence obtained by swapping the residues for `i` and `j`.

INPUT:

- `i`, `j` – two integers between 1 and the length of the residue sequence

If residue sequence `self` is of the form $(r_1, \ldots, r_n)$, and $i < j$, then the residue sequence $(r_1, \ldots, r_j, \ldots, r_i, \ldots, r_m)$, with the same `quantum_characteristic()` and `multicharge()`, is returned.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: res = ResidueSequence(3,(0,0,1),[0,0,1,1,2,2,3,3]); res
3-residue sequence (0,0,1,1,2,2,0,0) with multicharge (0,0,1)
sage: ser = res.swap_residues(2,6); ser
3-residue sequence (0,2,1,1,2,0,0,0) with multicharge (0,0,1)
sage: res == ser
False
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> res = ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(1)),
↪[Integer(0),Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(3)]); res
3-residue sequence (0,0,1,1,2,2,0,0) with multicharge (0,0,1)
>>> ser = res.swap_residues(Integer(2),Integer(6)); ser
3-residue sequence (0,2,1,1,2,0,0,0) with multicharge (0,0,1)
>>> res == ser
False
```

class sage.combinat.tableau_residues.**ResidueSequences**(*e*, *multicharge=(0,)*)

Bases: `UniqueRepresentation`, `Parent`

A parent class for `ResidueSequence`.

This class exists because `ResidueSequence` needs to have a parent. Apart form being a parent the only useful method that it provides is `cell_residue()`, which is a short-hand for computing the residue of a cell using the `ResidueSequence.quantum_characteristic()` and `ResidueSequence.multicharge()` for the residue class.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequences
sage: ResidueSequences(e=0, multicharge=(0,1,2))
0-residue sequences with multicharge (0, 1, 2)
sage: ResidueSequences(e=0, multicharge=(0,1,2)) == ResidueSequences(e=0,␣
↪multicharge=(0,1,2))
True
sage: ResidueSequences(e=0, multicharge=(0,1,2)) == ResidueSequences(e=3,␣
↪multicharge=(0,1,2))
False
sage: ResidueSequences(e=0, multicharge=(0,1,2)).element_class
<class 'sage.combinat.tableau_residues.ResidueSequences_with_category.element_
↪class'>
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequences
>>> ResidueSequences(e=Integer(0), multicharge=(Integer(0),Integer(1),Integer(2)))
0-residue sequences with multicharge (0, 1, 2)
>>> ResidueSequences(e=Integer(0), multicharge=(Integer(0),Integer(1),
↪Integer(2))) == ResidueSequences(e=Integer(0), multicharge=(Integer(0),
↪Integer(1),Integer(2)))
True
>>> ResidueSequences(e=Integer(0), multicharge=(Integer(0),Integer(1),
↪Integer(2))) == ResidueSequences(e=Integer(3), multicharge=(Integer(0),
↪Integer(1),Integer(2)))
False
>>> ResidueSequences(e=Integer(0), multicharge=(Integer(0),Integer(1),
↪Integer(2))).element_class
<class 'sage.combinat.tableau_residues.ResidueSequences_with_category.element_
↪class'>
```

Element

alias of `ResidueSequence`

an_element()

Return a particular element of `self`.

EXAMPLES:

```
sage: TableauTuples().an_element()
([[1]], [[2]], [[3]], [[4]], [[5]], [[6]], [[7]])
```

```
>>> from sage.all import *
>>> TableauTuples().an_element()
([[1]], [[2]], [[3]], [[4]], [[5]], [[6]], [[7]])
```

cell_residue(**args*)

Return the residue a cell with respect to the quantum characteristic and the multicharge of the residue sequence.

INPUT:

- `r`, `c` – the row and column indices in level one
- `k`, `r`, `c` – the component, row and column indices in higher levels

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequences
sage: ResidueSequences(3).cell_residue(1,1)
0
sage: ResidueSequences(3).cell_residue(2,1)
2
sage: ResidueSequences(3).cell_residue(3,1)
1
sage: ResidueSequences(3).cell_residue(3,2)
2
sage: ResidueSequences(3,(0,1,2)).cell_residue(0,0,0)
0
sage: ResidueSequences(3,(0,1,2)).cell_residue(0,1,0)
2
sage: ResidueSequences(3,(0,1,2)).cell_residue(0,1,2)
1
sage: ResidueSequences(3,(0,1,2)).cell_residue(1,0,0)
1
sage: ResidueSequences(3,(0,1,2)).cell_residue(1,1,0)
0
sage: ResidueSequences(3,(0,1,2)).cell_residue(1,0,1)
2
sage: ResidueSequences(3,(0,1,2)).cell_residue(2,0,0)
2
sage: ResidueSequences(3,(0,1,2)).cell_residue(2,1,0)
1
sage: ResidueSequences(3,(0,1,2)).cell_residue(2,0,1)
0
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequences
>>> ResidueSequences(Integer(3)).cell_residue(Integer(1),Integer(1))
0
>>> ResidueSequences(Integer(3)).cell_residue(Integer(2),Integer(1))
2
>>> ResidueSequences(Integer(3)).cell_residue(Integer(3),Integer(1))
1
>>> ResidueSequences(Integer(3)).cell_residue(Integer(3),Integer(2))
2
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(0),Integer(0),Integer(0))
0
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(0),Integer(1),Integer(0))
2
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(0),Integer(1),Integer(2))
1
```

(continues on next page)

(continued from previous page)

```
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(1),Integer(0),Integer(0))
1
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(1),Integer(1),Integer(0))
0
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(1),Integer(0),Integer(1))
2
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(2),Integer(0),Integer(0))
2
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(2),Integer(1),Integer(0))
1
>>> ResidueSequences(Integer(3),(Integer(0),Integer(1),Integer(2))).cell_
↪residue(Integer(2),Integer(0),Integer(1))
0
```

check_element(*element*)

Check that `element` is a residue sequence with multicharge `self.multicharge()`.

This is weak criteria in that we only require that `element` is a tuple of elements in the underlying base ring of `self`. Such a sequence is always a valid residue sequence, although there may be no tableaux with this residue sequence.

EXAMPLES:

```
sage: from sage.combinat.tableau_residues import ResidueSequence
sage: ResidueSequence(3,(0,0,1),[0,0,1,1,2,2,3,3]) # indirect doctest
3-residue sequence (0,0,1,1,2,2,0,0) with multicharge (0,0,1)
sage: ResidueSequence(3,(0,0,1),[2,0,1,4,2,2,5,3]) # indirect doctest
3-residue sequence (2,0,1,1,2,2,2,0) with multicharge (0,0,1)
sage: ResidueSequence(3,(0,0,1),[2,0,1,1,2,2,3,3]) # indirect doctest
3-residue sequence (2,0,1,1,2,2,0,0) with multicharge (0,0,1)
```

```
>>> from sage.all import *
>>> from sage.combinat.tableau_residues import ResidueSequence
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(1)),[Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
↪Integer(3)]) # indirect doctest
3-residue sequence (0,0,1,1,2,2,0,0) with multicharge (0,0,1)
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(1)),[Integer(2),
↪Integer(0),Integer(1),Integer(4),Integer(2),Integer(2),Integer(5),
↪Integer(3)]) # indirect doctest
3-residue sequence (2,0,1,1,2,2,2,0) with multicharge (0,0,1)
>>> ResidueSequence(Integer(3),(Integer(0),Integer(0),Integer(1)),[Integer(2),
↪Integer(0),Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
↪Integer(3)]) # indirect doctest
3-residue sequence (2,0,1,1,2,2,0,0) with multicharge (0,0,1)
```

5.1.347 TableauTuples

A `TableauTuple` is a tuple of tableaux. These objects arise naturally in representation theory of the wreath products of cyclic groups and the symmetric groups where the standard tableau tuples index bases for the ordinary irreducible representations. This generalises the well-known fact the ordinary irreducible representations of the symmetric groups have bases indexed by the standard tableaux of a given shape. More generally, `TableauTuples`, or multitableaux, appear in the representation theory of the degenerate and non-degenerate cyclotomic Hecke algebras and in the crystal theory of the integral highest weight representations of the affine special linear groups.

A `TableauTuple` is an ordered tuple $(t^{(1)}, t^{(2)}, \ldots, t^{(l)})$ of tableaux. The length of the tuple is its *level* and the tableaux $t^{(1)}, t^{(2)}, \ldots, t^{(l)}$ are the components of the `TableauTuple`.

A tableau can be thought of as the labelled diagram of a partition. Analogously, a `TableauTuple` is the labelled diagram of a `PartitionTuple`. That is, a `TableauTuple` is a tableau of `PartitionTuple` shape. As much as possible, `TableauTuples` behave in exactly the same way as `Tableaux`. There are obvious differences in that the cells of a partition are ordered pairs (r, c), where r is a row index and c a column index, whereas the cells of a `PartitionTuple` are ordered triples (k, r, c), with r and c as before and k indexes the component.

Frequently, we will call a `TableauTuple` a tableau, or a tableau of `PartitionTuple` shape. If the shape of the tableau is known this should not cause any confusion.

Warning

In sage the convention is that the (k, r, c)-th entry of a tableau tuple t is the entry in row r, column c and component k of the tableau. This is because it makes much more sense to let `t[k]` be component of the tableau. In particular, we want `t(k,r,c) == t[k][r][c]`. In the literature, the cells of a tableau tuple are usually written in the form (r, c, k), where r is the row index, c is the column index, and k is the component index.

The same convention applies to the cells of `PartitionTuples`.

Note

As with partitions and tableaux, the cells are 0-based. For example, the (lexicographically) first cell in any non-empty tableau tuple is `[0,0,0]`.

EXAMPLES:

```
sage: TableauTuple([[1,2,3],[4,5]])
[[1, 2, 3], [4, 5]]
sage: t = TableauTuple([ [[6,7],[8,9]],[[1,2,3],[4,5]] ]); t
([[6, 7], [8, 9]], [[1, 2, 3], [4, 5]])
sage: t.pp()
  6  7     1  2  3
  8  9     4  5
sage: t(0,0,1)
7
sage: t(1,0,1)
2
sage: t.shape()
([2, 2], [3, 2])
sage: t.size()
9
sage: t.level()
```

(continues on next page)

(continued from previous page)

```
2
sage: t.components()
[[[6, 7], [8, 9]], [[1, 2, 3], [4, 5]]]
sage: t.entries()
[6, 7, 8, 9, 1, 2, 3, 4, 5]
sage: t.parent()
Tableau tuples
sage: t.category()
Category of elements of Tableau tuples
```

```
>>> from sage.all import *
>>> TableauTuple([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])
[[1, 2, 3], [4, 5]]
>>> t = TableauTuple([ [[Integer(6),Integer(7)],[Integer(8),Integer(9)]],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(4),Integer(5)]] ]); t
([[6, 7], [8, 9]], [[1, 2, 3], [4, 5]])
>>> t.pp()
  6  7     1  2  3
  8  9     4  5
>>> t(Integer(0),Integer(0),Integer(1))
7
>>> t(Integer(1),Integer(0),Integer(1))
2
>>> t.shape()
([2, 2], [3, 2])
>>> t.size()
9
>>> t.level()
2
>>> t.components()
[[[6, 7], [8, 9]], [[1, 2, 3], [4, 5]]]
>>> t.entries()
[6, 7, 8, 9, 1, 2, 3, 4, 5]
>>> t.parent()
Tableau tuples
>>> t.category()
Category of elements of Tableau tuples
```

One reason for implementing `TableauTuples` is to be able to consider `StandardTableauTuples`. These objects arise in many areas of algebraic combinatorics. In particular, they index bases for the Specht modules of the cyclotomic Hecke algebras of type $G(r, 1, n)$. A `StandardTableauTuple` of tableau whose entries are increasing along rows and down columns in each component and which contain the numbers $1, 2, \ldots, n$, where the shape of the `StandardTableauTuple` is a `PartitionTuple` of n.

```
sage: s = StandardTableauTuple([ [[1,2],[3]],[[4,5]]])
sage: s.category()
Category of elements of Standard tableau tuples
sage: t = TableauTuple([ [[1,2],[3]],[[4,5]]])
sage: t.is_standard(), t.is_column_strict(), t.is_row_strict()
(True, True, True)
sage: t.category()
Category of elements of Tableau tuples
```

(continues on next page)

(continued from previous page)

```
sage: s == t
True
sage: s is t
False
sage: s == StandardTableauTuple(t)
True
sage: StandardTableauTuples([ [2,1],[1] ])[:]
[([[1, 2], [3]], [[4]]),
 ([[1, 3], [2]], [[4]]),
 ([[1, 2], [4]], [[3]]),
 ([[1, 3], [4]], [[2]]),
 ([[2, 3], [4]], [[1]]),
 ([[1, 4], [2]], [[3]]),
 ([[1, 4], [3]], [[2]]),
 ([[2, 4], [3]], [[1]])]
```

```
>>> from sage.all import *
>>> s = StandardTableauTuple([ [[Integer(1),Integer(2)],[Integer(3)]],[[Integer(4),
↪Integer(5)]]])
>>> s.category()
Category of elements of Standard tableau tuples
>>> t = TableauTuple([ [[Integer(1),Integer(2)],[Integer(3)]],[[Integer(4),
↪Integer(5)]]])
>>> t.is_standard(), t.is_column_strict(), t.is_row_strict()
(True, True, True)
>>> t.category()
Category of elements of Tableau tuples
>>> s == t
True
>>> s is t
False
>>> s == StandardTableauTuple(t)
True
>>> StandardTableauTuples([ [Integer(2),Integer(1)],[Integer(1)] ])[:]
[([[1, 2], [3]], [[4]]),
 ([[1, 3], [2]], [[4]]),
 ([[1, 2], [4]], [[3]]),
 ([[1, 3], [4]], [[2]]),
 ([[2, 3], [4]], [[1]]),
 ([[1, 4], [2]], [[3]]),
 ([[1, 4], [3]], [[2]]),
 ([[2, 4], [3]], [[1]])]
```

As tableaux (of partition shape) are in natural bijection with 1-tuples of tableaux all of the `TableauTuple` classes return an ordinary `Tableau` when given `TableauTuple` of level 1.

```
sage: TableauTuples( level=1 ) is Tableaux()
True
sage: TableauTuple([[1,2,3],[4,5]])
[[1, 2, 3], [4, 5]]
sage: TableauTuple([ [[1,2,3],[4,5]] ])
[[1, 2, 3], [4, 5]]
```

(continues on next page)

(continued from previous page)

```
sage: TableauTuple([[1,2,3],[4,5]]) == Tableau([[1,2,3],[4,5]])
True
```

```
>>> from sage.all import *
>>> TableauTuples( level=Integer(1) ) is Tableaux()
True
>>> TableauTuple([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])
[[1, 2, 3], [4, 5]]
>>> TableauTuple([ [[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]] ])
[[1, 2, 3], [4, 5]]
>>> TableauTuple([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]) ==
↪Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])
True
```

There is one situation where a 1-tuple of tableau is not actually a `Tableau`; tableaux generated by the `StandardTableauTuples()` iterators must have the correct parents, so in this one case 1-tuples of tableaux are different from `Tableaux`:

```
sage: StandardTableauTuples()[:10]                                          #
↪needs sage.libs.flint
[(),
 ([[1]]),
 ([], []),
 ([[1, 2]]),
 ([[1], [2]]),
 ([[1]], []),
 ([], [[1]]),
 ([], [], []),
 ([[1, 2, 3]]),
 ([[1, 3], [2]])]
```

```
>>> from sage.all import *
>>> StandardTableauTuples()[:Integer(10)]
↪     # needs sage.libs.flint
[(),
 ([[1]]),
 ([], []),
 ([[1, 2]]),
 ([[1], [2]]),
 ([[1]], []),
 ([], [[1]]),
 ([], [], []),
 ([[1, 2, 3]]),
 ([[1, 3], [2]])]
```

AUTHORS:

- Andrew Mathas (2012-10-09): Initial version – heavily based on `tableau.py` by Mike Hansen (2007) and Jason Bandlow (2011).
- Andrew Mathas (2016-08-11): Row standard tableaux added

Element classes:

- *TableauTuples*
- *StandardTableauTuples*
- *RowStandardTableauTuples*

Factory classes:

- *TableauTuples*
- *StandardTableauTuples*
- *RowStandardTableauTuples*

Parent classes:

- *TableauTuples_all*
- *TableauTuples_level*
- *TableauTuples_size*
- *TableauTuples_level_size*
- *StandardTableauTuples_all*
- *StandardTableauTuples_level*
- *StandardTableauTuples_size*
- *StandardTableauTuples_level_size*
- *StandardTableauTuples_shape*
- *StandardTableaux_residue*
- *StandardTableaux_residue_shape*
- *RowStandardTableauTuples_all*
- *RowStandardTableauTuples_level*
- *RowStandardTableauTuples_size*
- *RowStandardTableauTuples_level_size*
- *RowStandardTableauTuples_shape*
- *RowStandardTableauTuples_residue*
- *RowStandardTableauTuples_residue_shape*

See also:

- *Tableau*
- *StandardTableau*
- *Tableaux*
- *StandardTableaux*
- *Partitions*
- *PartitionTuples*
- ResidueSequence

Todo

Implement semistandard tableau tuples as defined in [DJM1998].

Much of the combinatorics implemented here is motivated by this and subsequent papers on the representation theory of these algebras.

class sage.combinat.tableau_tuple.**RowStandardTableauTuple**(*parent*, *t*, *check=True*)

Bases: *TableauTuple*

A class for row standard tableau tuples of shape a partition tuple.

A row standard tableau tuple of size n is an ordered tuple of row standard tableaux (see *RowStandardTableau*), with entries $1, 2, \ldots, n$ such that, in each component, the entries are in increasing order along each row. If the tableau in component k has shape $\lambda^{(k)}$ then $\lambda = (\lambda^{(1)}, \ldots, \lambda^{(l)}$ is a *PartitionTuple*.

Note

The tableaux appearing in a *RowStandardTableauTuple* are row strict, but individually they are not standard tableaux because the entries in any single component of a *RowStandardTableauTuple* will typically not be in bijection with $\{1, 2, \ldots, n\}$.

INPUT:

- `t` – a tableau, a list of (standard) tableau or an equivalent list

OUTPUT: a *RowStandardTableauTuple* object constructed from `t`

Note

Sage uses the English convention for (tuples of) partitions and tableaux: the longer rows are displayed on top. As with *PartitionTuple*, in sage the cells, or nodes, of partition tuples are 0-based. For example, the (lexicographically) first cell in any non-empty partition tuple is $[0, 0, 0]$. Further, the coordinates `[k,r,c]` in a *TableauTuple* refer to the component, row and column indices, respectively.

EXAMPLES:

```
sage: t = RowStandardTableauTuple([[[4,7],[3]],[[2,6,8],[1,5]],[[9]]]); t
([[4, 7], [3]], [[2, 6, 8], [1, 5]], [[9]])
sage: t.pp()
  4  7     2  6  8     9
  3        1  5
sage: t.shape()
([2, 1], [3, 2], [1])
sage: t[0].pp()   # pretty printing
  4  7
  3
sage: t.is_row_strict()
True
sage: t[0].is_standard()
False
sage: RowStandardTableauTuple([[],[],[]]) # An empty tableau tuple
([], [], [])
```

(continues on next page)

(continued from previous page)

```
sage: RowStandardTableauTuple([[[4,5],[6]],[[1,2,3]]]) in StandardTableauTuples()
True
sage: RowStandardTableauTuple([[[5,6],[4]],[[1,2,3]]]) in StandardTableauTuples()
False
```

```
>>> from sage.all import *
>>> t = RowStandardTableauTuple([[[Integer(4),Integer(7)],[Integer(3)]],
↪[[Integer(2),Integer(6),Integer(8)],[Integer(1),Integer(5)]],[[Integer(9)]]]); t
([[4, 7], [3]], [[2, 6, 8], [1, 5]], [[9]])
>>> t.pp()
  4  7     2  6  8     9
  3        1  5
>>> t.shape()
([2, 1], [3, 2], [1])
>>> t[Integer(0)].pp()  # pretty printing
  4  7
  3
>>> t.is_row_strict()
True
>>> t[Integer(0)].is_standard()
False
>>> RowStandardTableauTuple([[],[],[]]) # An empty tableau tuple
([], [], [])
>>> RowStandardTableauTuple([[[Integer(4),Integer(5)],[Integer(6)]],[[Integer(1),
↪Integer(2),Integer(3)]]]) in StandardTableauTuples()
True
>>> RowStandardTableauTuple([[[Integer(5),Integer(6)],[Integer(4)]],[[Integer(1),
↪Integer(2),Integer(3)]]]) in StandardTableauTuples()
False
```

When using code that will generate a lot of tableaux, it is slightly more efficient to construct a *RowStandardTableauTuple* from the appropriate parent object:

```
sage: RST = RowStandardTableauTuples()
sage: RST([[[4,5],[7]],[[1,2,3],[6,8]],[[9]]])
([[4, 5], [7]], [[1, 2, 3], [6, 8]], [[9]])
```

```
>>> from sage.all import *
>>> RST = RowStandardTableauTuples()
>>> RST([[[Integer(4),Integer(5)],[Integer(7)]],[[Integer(1),Integer(2),
↪Integer(3)],[Integer(6),Integer(8)]],[[Integer(9)]]])
([[4, 5], [7]], [[1, 2, 3], [6, 8]], [[9]])
```

See also:

- RowTableau
- RowTableaux
- *TableauTuples*
- *TableauTuple*

- *StandardTableauTuples*
- *StandardTableauTuple*
- *RowStandardTableauTuples*

codegree (*e*, *multicharge*)

Return the Brundan-Kleshchev-Wang [BKW2011] codegree of `self`.

The *codegree* of a tableau is an integer that is defined recursively by successively stripping off the number k, for $k = n, n-1, \ldots, 1$ and at stage adding the number of addable cell of the same residue minus the number of removable cells of the same residue as k and which are above k in the diagram.

The codegree of the tableau `self` gives the degree of "dual" homogeneous basis element of the graded Specht module which is indexed by `self`.

INPUT:

- `e` – the *quantum characteristic*
- `multicharge` – the multicharge

OUTPUT: the codegree of the tableau `self`, which is an integer

EXAMPLES:

```
sage: StandardTableauTuple([[[1]], [], []]).codegree(0,(0,0,0))
0
sage: StandardTableauTuple([[],[[1]], []]).codegree(0,(0,0,0))
1
sage: StandardTableauTuple([[], [], [[1]]]).codegree(0,(0,0,0))
2
sage: StandardTableauTuple([[[1]],[[2]], []]).codegree(0,(0,0,0))
-1
sage: StandardTableauTuple([[[1]], [], [[2]]]).codegree(0,(0,0,0))
0
sage: StandardTableauTuple([[],[[1]], [[2]]]).codegree(0,(0,0,0))
1
sage: StandardTableauTuple([[[2]],[[1]], []]).codegree(0,(0,0,0))
1
sage: StandardTableauTuple([[[2]], [], [[1]]]).codegree(0,(0,0,0))
2
sage: StandardTableauTuple([[],[[2]], [[1]]]).codegree(0,(0,0,0))
3
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[[Integer(1)]], [], []]).codegree(Integer(0),
↪(Integer(0),Integer(0),Integer(0)))
0
>>> StandardTableauTuple([[],[[Integer(1)]], []]).codegree(Integer(0),
↪(Integer(0),Integer(0),Integer(0)))
1
>>> StandardTableauTuple([[], [], [[Integer(1)]]]).codegree(Integer(0),
↪(Integer(0),Integer(0),Integer(0)))
2
>>> StandardTableauTuple([[[Integer(1)]],[[Integer(2)]], []]).
```

(continues on next page)

(continued from previous page)

```
↪codegree(Integer(0),(Integer(0),Integer(0),Integer(0)))
-1
>>> StandardTableauTuple([[[Integer(1)]], [], [[Integer(2)]]]).
↪codegree(Integer(0),(Integer(0),Integer(0),Integer(0)))
0
>>> StandardTableauTuple([[],[[Integer(1)]], [[Integer(2)]]]).
↪codegree(Integer(0),(Integer(0),Integer(0),Integer(0)))
1
>>> StandardTableauTuple([[[Integer(2)]],[[Integer(1)]], []]).
↪codegree(Integer(0),(Integer(0),Integer(0),Integer(0)))
1
>>> StandardTableauTuple([[[Integer(2)]], [], [[Integer(1)]]]).
↪codegree(Integer(0),(Integer(0),Integer(0),Integer(0)))
2
>>> StandardTableauTuple([[],[[Integer(2)]], [[Integer(1)]]]).
↪codegree(Integer(0),(Integer(0),Integer(0),Integer(0)))
3
```

degree(*e*, *multicharge*)

Return the Brundan-Kleshchev-Wang [BKW2011] degree of `self`.

The *degree* of a tableau is an integer that is defined recursively by successively stripping off the number k, for $k = n, n-1, \ldots, 1$, and at stage adding the count of the number of addable cell of the same residue minus the number of removable cells of them same residue as k and that are below k in the diagram.

Note that even though this degree function was defined by Brundan-Kleshchev-Wang [BKW2011] the underlying combinatorics is much older, going back at least to Misra and Miwa.

The degrees of the tableau T gives the degree of the homogeneous basis element of the graded Specht module which is indexed by T.

INPUT:

- `e` – the *quantum characteristic* `e`
- `multicharge` – (default: `[0]`) the multicharge

OUTPUT: the degree of the tableau `self`, which is an integer

EXAMPLES:

```
sage: StandardTableauTuple([[[1]], [], []]).degree(0,(0,0,0))
2
sage: StandardTableauTuple([[],[[1]], []]).degree(0,(0,0,0))
1
sage: StandardTableauTuple([[], [], [[1]]]).degree(0,(0,0,0))
0
sage: StandardTableauTuple([[[1]],[[2]], []]).degree(0,(0,0,0))
3
sage: StandardTableauTuple([[[1]], [], [[2]]]).degree(0,(0,0,0))
2
sage: StandardTableauTuple([[],[[1]], [[2]]]).degree(0,(0,0,0))
1
sage: StandardTableauTuple([[[2]],[[1]], []]).degree(0,(0,0,0))
1
sage: StandardTableauTuple([[[2]], [], [[1]]]).degree(0,(0,0,0))
```

(continues on next page)

(continued from previous page)

```
0
sage: StandardTableauTuple([[],[[2]], [[1]]]).degree(0,(0,0,0))
-1
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[[Integer(1)]], [], []]).degree(Integer(0),
↪(Integer(0),Integer(0),Integer(0)))
2
>>> StandardTableauTuple([[],[[Integer(1)]], []]).degree(Integer(0),
↪(Integer(0),Integer(0),Integer(0)))
1
>>> StandardTableauTuple([[], [], [[Integer(1)]]]).degree(Integer(0),
↪(Integer(0),Integer(0),Integer(0)))
0
>>> StandardTableauTuple([[[Integer(1)]],[[Integer(2)]], []]).
↪degree(Integer(0),(Integer(0),Integer(0),Integer(0)))
3
>>> StandardTableauTuple([[[Integer(1)]], [], [[Integer(2)]]]).
↪degree(Integer(0),(Integer(0),Integer(0),Integer(0)))
2
>>> StandardTableauTuple([[],[[Integer(1)]], [[Integer(2)]]]).
↪degree(Integer(0),(Integer(0),Integer(0),Integer(0)))
1
>>> StandardTableauTuple([[[Integer(2)]],[[Integer(1)]], []]).
↪degree(Integer(0),(Integer(0),Integer(0),Integer(0)))
1
>>> StandardTableauTuple([[[Integer(2)]], [], [[Integer(1)]]]).
↪degree(Integer(0),(Integer(0),Integer(0),Integer(0)))
0
>>> StandardTableauTuple([[],[[Integer(2)]], [[Integer(1)]]]).
↪degree(Integer(0),(Integer(0),Integer(0),Integer(0)))
-1
```

inverse(*k*)

Return the cell containing `k` in the tableau tuple `self`.

EXAMPLES:

```
sage: RowStandardTableauTuple([[[3,4],[1,2]],[[5,6,7],[8]],[[9,10],[11],
↪[12]]]).inverse(1)
(0, 1, 0)
sage: RowStandardTableauTuple([[[3,4],[1,2]],[[5,6,7],[8]],[[9,10],[11],
↪[12]]]).inverse(2)
(0, 1, 1)
sage: RowStandardTableauTuple([[[3,4],[1,2]],[[5,6,7],[8]],[[9,10],[11],
↪[12]]]).inverse(3)
(0, 0, 0)
sage: RowStandardTableauTuple([[[3,4],[1,2]],[[5,6,7],[8]],[[9,10],[11],
↪[12]]]).inverse(4)
(0, 0, 1)
sage: StandardTableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).
↪inverse(1)
```

(continues on next page)

(continued from previous page)

```
(0, 0, 0)
sage: StandardTableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).
↪inverse(2)
(0, 0, 1)
sage: StandardTableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).
↪inverse(3)
(0, 1, 0)
sage: StandardTableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).
↪inverse(12)
(2, 2, 0)
```

```
>>> from sage.all import *
>>> RowStandardTableauTuple([[[Integer(3),Integer(4)],[Integer(1),
↪Integer(2)]],[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),
↪Integer(10)],[Integer(11)],[Integer(12)]]]).inverse(Integer(1))
(0, 1, 0)
>>> RowStandardTableauTuple([[[Integer(3),Integer(4)],[Integer(1),
↪Integer(2)]],[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),
↪Integer(10)],[Integer(11)],[Integer(12)]]]).inverse(Integer(2))
(0, 1, 1)
>>> RowStandardTableauTuple([[[Integer(3),Integer(4)],[Integer(1),
↪Integer(2)]],[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),
↪Integer(10)],[Integer(11)],[Integer(12)]]]).inverse(Integer(3))
(0, 0, 0)
>>> RowStandardTableauTuple([[[Integer(3),Integer(4)],[Integer(1),
↪Integer(2)]],[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),
↪Integer(10)],[Integer(11)],[Integer(12)]]]).inverse(Integer(4))
(0, 0, 1)
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).inverse(Integer(1))
(0, 0, 0)
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).inverse(Integer(2))
(0, 0, 1)
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).inverse(Integer(3))
(0, 1, 0)
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).inverse(Integer(12))
(2, 2, 0)
```

residue_sequence(*e*, *multicharge*)

Return the *sage.combinat.tableau_residues.ResidueSequence* of `self`.

INPUT:

- `e` – integer in $\{0, 2, 3, 4, 5, \ldots\}$
- `multicharge` – a sequence of integers of length equal to the level/length of `self`

OUTPUT:

The *residue sequence* of the tableau.

EXAMPLES:

```
sage: RowStandardTableauTuple([[[5]],[[3,4],[1,2]]]).residue_sequence(3,[0,0])
3-residue sequence (2,0,0,1,0) with multicharge (0,0)
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue_sequence(3,[0,1])
3-residue sequence (1,2,0,1,0) with multicharge (0,1)
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue_sequence(3,[0,2])
3-residue sequence (2,0,1,2,0) with multicharge (0,2)
```

```
>>> from sage.all import *
>>> RowStandardTableauTuple([[[Integer(5)]],[[Integer(3),Integer(4)],
↪[Integer(1),Integer(2)]]]).residue_sequence(Integer(3),[Integer(0),
↪Integer(0)])
3-residue sequence (2,0,0,1,0) with multicharge (0,0)
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue_sequence(Integer(3),[Integer(0),Integer(1)])
3-residue sequence (1,2,0,1,0) with multicharge (0,1)
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue_sequence(Integer(3),[Integer(0),Integer(2)])
3-residue sequence (2,0,1,2,0) with multicharge (0,2)
```

class sage.combinat.tableau_tuple.**RowStandardTableauTuples**

Bases: *TableauTuples*

A factory class for the various classes of tuples of row standard tableau.

INPUT:

There are three optional arguments:

- `level` – the *level()* of the tuples of tableaux
- `size` – the *size()* of the tuples of tableaux
- `shape` – list or a partition tuple specifying the *shape()* of the row standard tableau tuples

It is not necessary to use the keywords. If they are not used then the first integer argument specifies the *level()* and the second the *size()* of the tableau tuples.

OUTPUT: the appropriate subclass of *RowStandardTableauTuples*

A tuple of row standard tableau is a tableau whose entries are positive integers which increase from left to right along the rows in each component. The entries do NOT need to increase from left to right along the components.

> **Note**
>
> Sage uses the English convention for (tuples of) partitions and tableaux: the longer rows are displayed on top. As with *PartitionTuple*, in sage the cells, or nodes, of partition tuples are 0-based. For example, the (lexicographically) first cell in any non-empty partition tuple is $[0, 0, 0]$.

EXAMPLES:

```
sage: tabs = RowStandardTableauTuples([[2],[1,1]]); tabs
Row standard tableau tuples of shape ([2], [1, 1])
sage: tabs.cardinality()
12
sage: tabs[:]                                                        #␣
↪needs sage.graphs sage.rings.finite_rings
[([[3, 4]], [[2], [1]]),
 ([[2, 4]], [[3], [1]]),
 ([[1, 4]], [[3], [2]]),
 ([[1, 2]], [[4], [3]]),
 ([[1, 3]], [[4], [2]]),
 ([[2, 3]], [[4], [1]]),
 ([[1, 4]], [[2], [3]]),
 ([[1, 3]], [[2], [4]]),
 ([[1, 2]], [[3], [4]]),
 ([[2, 3]], [[1], [4]]),
 ([[2, 4]], [[1], [3]]),
 ([[3, 4]], [[1], [2]])]

sage: tabs = RowStandardTableauTuples(level=3); tabs
Row standard tableau tuples of level 3
sage: tabs[100]                                                      #␣
↪needs sage.libs.flint
([], [], [[2, 3], [1]])

sage: RowStandardTableauTuples()[0]                                  #␣
↪needs sage.libs.flint
([])
```

```
>>> from sage.all import *
>>> tabs = RowStandardTableauTuples([[Integer(2)],[Integer(1),Integer(1)]]); tabs
Row standard tableau tuples of shape ([2], [1, 1])
>>> tabs.cardinality()
12
>>> tabs[:]                                                        #␣
↪needs sage.graphs sage.rings.finite_rings
[([[3, 4]], [[2], [1]]),
 ([[2, 4]], [[3], [1]]),
 ([[1, 4]], [[3], [2]]),
 ([[1, 2]], [[4], [3]]),
 ([[1, 3]], [[4], [2]]),
 ([[2, 3]], [[4], [1]]),
 ([[1, 4]], [[2], [3]]),
 ([[1, 3]], [[2], [4]]),
 ([[1, 2]], [[3], [4]]),
 ([[2, 3]], [[1], [4]]),
 ([[2, 4]], [[1], [3]]),
 ([[3, 4]], [[1], [2]])]

>>> tabs = RowStandardTableauTuples(level=Integer(3)); tabs
Row standard tableau tuples of level 3
>>> tabs[Integer(100)]                                               ␣
```

(continues on next page)

(continued from previous page)

```
↪      # needs sage.libs.flint
([], [], [[2, 3], [1]])

>>> RowStandardTableauTuples()[Integer(0)]
↪      # needs sage.libs.flint
([])
```

See also

- *TableauTuples*
- *Tableau*
- *RowStandardTableau*
- *RowStandardTableauTuples*

Element

alias of *RowStandardTableauTuple*

level_one_parent_class

alias of *RowStandardTableaux_all*

shape()

Return the shape of the set of *RowStandardTableauTuples*, or `None` if it is not defined.

EXAMPLES:

```
sage: tabs=RowStandardTableauTuples(shape=[[5,2],[3,2],[],[1,1,1],[3]]); tabs
Row standard tableau tuples of shape ([5, 2], [3, 2], [], [1, 1, 1], [3])
sage: tabs.shape()
([5, 2], [3, 2], [], [1, 1, 1], [3])
sage: RowStandardTableauTuples().shape() is None
True
```

```
>>> from sage.all import *
>>> tabs=RowStandardTableauTuples(shape=[[Integer(5),Integer(2)],[Integer(3),
↪Integer(2)],[],[Integer(1),Integer(1),Integer(1)],[Integer(3)]]); tabs
Row standard tableau tuples of shape ([5, 2], [3, 2], [], [1, 1, 1], [3])
>>> tabs.shape()
([5, 2], [3, 2], [], [1, 1, 1], [3])
>>> RowStandardTableauTuples().shape() is None
True
```

class sage.combinat.tableau_tuple.**RowStandardTableauTuples_all**

Bases: *RowStandardTableauTuples*, `DisjointUnionEnumeratedSets`

Default class of all *RowStandardTableauTuples* with an arbitrary *level()* and *size()*.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: RowStandardTableauTuples().an_element()
([[4, 5, 6, 7]], [[2, 3]], [[1]])
```

```
>>> from sage.all import *
>>> RowStandardTableauTuples().an_element()
([[4, 5, 6, 7]], [[2, 3]], [[1]])
```

class sage.combinat.tableau_tuple.**RowStandardTableauTuples_level**(*level*)

Bases: *RowStandardTableauTuples*, DisjointUnionEnumeratedSets

Class of all *RowStandardTableauTuples* with a fixed level and arbitrary size.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: RowStandardTableauTuples(2).an_element()
([[1]], [[2, 3]])
sage: RowStandardTableauTuples(3).an_element()
([[1]], [[2, 3]], [[4, 5, 6, 7]])
```

```
>>> from sage.all import *
>>> RowStandardTableauTuples(Integer(2)).an_element()
([[1]], [[2, 3]])
>>> RowStandardTableauTuples(Integer(3)).an_element()
([[1]], [[2, 3]], [[4, 5, 6, 7]])
```

class sage.combinat.tableau_tuple.**RowStandardTableauTuples_level_size**(*level*, *size*)

Bases: *RowStandardTableauTuples*, DisjointUnionEnumeratedSets

Class of all *RowStandardTableauTuples* with a fixed level and a fixed size.

an_element()

Return a particular element of self.

EXAMPLES:

```
sage: RowStandardTableauTuples(5, size=2).an_element()                    #␣
↪needs sage.libs.flint
([], [], [], [], [[1], [2]])
sage: RowStandardTableauTuples(2, size=4).an_element()                    #␣
↪needs sage.libs.flint
([[1]], [[2, 3], [4]])
```

```
>>> from sage.all import *
>>> RowStandardTableauTuples(Integer(5), size=Integer(2)).an_element()      ␣
↪               # needs sage.libs.flint
([], [], [], [], [[1], [2]])
>>> RowStandardTableauTuples(Integer(2), size=Integer(4)).an_element()      ␣
↪               # needs sage.libs.flint
([[1]], [[2, 3], [4]])
```

class sage.combinat.tableau_tuple.**RowStandardTableauTuples_residue**(*residue*)

Bases: *RowStandardTableauTuples*

Class of all row standard tableau tuples with a fixed residue sequence.

Implicitly, this also specifies the quantum characteristic, multicharge and hence the level and size of the tableaux.

> **Note**
>
> This class is not intended to be called directly, but rather, it is accessed through the row standard tableaux.

EXAMPLES:

```
sage: RowStandardTableau([[3,4,5],[1,2]]).residue_sequence(2).row_standard_
↪tableaux()
Row standard tableaux with 2-residue sequence (1,0,0,1,0) and multicharge (0)
sage: RowStandardTableau([[3,4,5],[1,2]]).residue_sequence(3).row_standard_
↪tableaux()
Row standard tableaux with 3-residue sequence (2,0,0,1,2) and multicharge (0)
sage: RowStandardTableauTuple([[[5,6],[7]],[[1,2,3],[4]]]).residue_sequence(2,(0,
↪0)).row_standard_tableaux()
Row standard tableaux with 2-residue sequence (0,1,0,1,0,1,1) and multicharge (0,
↪0)
sage: RowStandardTableauTuple([[[5,6],[7]],[[1,2,3],[4]]]).residue_sequence(3,(0,
↪1)).row_standard_tableaux()
Row standard tableaux with 3-residue sequence (1,2,0,0,0,1,2) and multicharge (0,
↪1)
```

```
>>> from sage.all import *
>>> RowStandardTableau([[Integer(3),Integer(4),Integer(5)],[Integer(1),
↪Integer(2)]]).residue_sequence(Integer(2)).row_standard_tableaux()
Row standard tableaux with 2-residue sequence (1,0,0,1,0) and multicharge (0)
>>> RowStandardTableau([[Integer(3),Integer(4),Integer(5)],[Integer(1),
↪Integer(2)]]).residue_sequence(Integer(3)).row_standard_tableaux()
Row standard tableaux with 3-residue sequence (2,0,0,1,2) and multicharge (0)
>>> RowStandardTableauTuple([[[Integer(5),Integer(6)],[Integer(7)]],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(4)]]]).residue_sequence(Integer(2),(Integer(0),
↪Integer(0))).row_standard_tableaux()
Row standard tableaux with 2-residue sequence (0,1,0,1,0,1,1) and multicharge (0,
↪0)
>>> RowStandardTableauTuple([[[Integer(5),Integer(6)],[Integer(7)]],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(4)]]]).residue_sequence(Integer(3),(Integer(0),
↪Integer(1))).row_standard_tableaux()
Row standard tableaux with 3-residue sequence (1,2,0,0,0,1,2) and multicharge (0,
↪1)
```

an_element()

Return a particular element of `self`.

EXAMPLES:

```
sage: RowStandardTableau([[2,3],[1]]).residue_sequence(3).row_standard_
↪tableaux().an_element()
```

(continues on next page)

(continued from previous page)

```
[[2, 3], [1]]
sage: StandardTableau([[1,3],[2]]).residue_sequence(3).row_standard_
↪tableaux().an_element()
[[1, 3], [2]]
sage: RowStandardTableauTuple([[[4]],[[2,3],[1]]]).residue_sequence(3,(0,1)).
↪row_standard_tableaux().an_element()                                        #␣
↪needs sage.libs.flint
sage: StandardTableauTuple([[[4]],[[1,3],[2]]]).residue_sequence(3,(0,1)).row_
↪standard_tableaux().an_element()                                           #␣
↪needs sage.libs.flint
([[4], [3], [1], [2]], [])
```

```
>>> from sage.all import *
>>> RowStandardTableau([[Integer(2),Integer(3)],[Integer(1)]]).residue_
↪sequence(Integer(3)).row_standard_tableaux().an_element()
[[2, 3], [1]]
>>> StandardTableau([[Integer(1),Integer(3)],[Integer(2)]]).residue_
↪sequence(Integer(3)).row_standard_tableaux().an_element()
[[1, 3], [2]]
>>> RowStandardTableauTuple([[[Integer(4)]],[[Integer(2),Integer(3)],
↪[Integer(1)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
↪standard_tableaux().an_element()                                  # needs␣
↪sage.libs.flint
>>> StandardTableauTuple([[[Integer(4)]],[[Integer(1),Integer(3)],
↪[Integer(2)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
↪standard_tableaux().an_element()                                           #␣
↪needs sage.libs.flint
([[4], [3], [1], [2]], [])
```

level()

Return the level of `self`.

EXAMPLES:

```
sage: RowStandardTableau([[2,3],[1]]).residue_sequence(3,(0,1)).row_standard_
↪tableaux().level()
2
sage: StandardTableau([[1,2],[3]]).residue_sequence(3,(0,1)).row_standard_
↪tableaux().level()
2
sage: RowStandardTableauTuple([[[4]],[[2,3],[1]]]).residue_sequence(3,(0,1)).
↪row_standard_tableaux().level()
2
sage: StandardTableauTuple([[[4]],[[1,3],[2]]]).residue_sequence(3,(0,1)).row_
↪standard_tableaux().level()
2
```

```
>>> from sage.all import *
>>> RowStandardTableau([[Integer(2),Integer(3)],[Integer(1)]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().level()
2
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3)]]).residue_
```

(continues on next page)

(continued from previous page)

```
↪sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().level()
2
>>> RowStandardTableauTuple([[[Integer(4)]],[[Integer(2),Integer(3)],
↪[Integer(1)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
↪standard_tableaux().level()
2
>>> StandardTableauTuple([[[Integer(4)]],[[Integer(1),Integer(3)],
↪[Integer(2)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
↪standard_tableaux().level()
2
```

multicharge()

Return the multicharge of `self`.

EXAMPLES:

```
sage: RowStandardTableau([[2,3],[1]]).residue_sequence(3,(0,1)).row_standard_
↪tableaux().multicharge()
(0, 1)
sage: StandardTableau([[1,2],[3]]).residue_sequence(3,(0,1)).row_standard_
↪tableaux().multicharge()
(0, 1)
sage: RowStandardTableauTuple([[[4]],[[2,3],[1]]]).residue_sequence(3,(0,1)).
↪row_standard_tableaux().multicharge()
(0, 1)
sage: StandardTableauTuple([[[4]],[[1,3],[2]]]).residue_sequence(3,(0,1)).row_
↪standard_tableaux().multicharge()
(0, 1)
```

```
>>> from sage.all import *
>>> RowStandardTableau([[Integer(2),Integer(3)],[Integer(1)]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().
↪multicharge()
(0, 1)
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3)]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().
↪multicharge()
(0, 1)
>>> RowStandardTableauTuple([[[Integer(4)]],[[Integer(2),Integer(3)],
↪[Integer(1)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
↪standard_tableaux().multicharge()
(0, 1)
>>> StandardTableauTuple([[[Integer(4)]],[[Integer(1),Integer(3)],
↪[Integer(2)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
↪standard_tableaux().multicharge()
(0, 1)
```

quantum_characteristic()

Return the quantum characteristic of `self`.

EXAMPLES:

```
sage: RowStandardTableau([[2,3],[1]]).residue_sequence(3,(0,1)).row_standard_
↪tableaux().quantum_characteristic()
3
sage: StandardTableau([[1,2],[3]]).residue_sequence(3,(0,1)).row_standard_
↪tableaux().quantum_characteristic()
3
sage: RowStandardTableauTuple([[[4]],[[2,3],[1]]]).residue_sequence(3,(0,1)).
↪row_standard_tableaux().quantum_characteristic()
3
sage: StandardTableauTuple([[[4]],[[1,3],[2]]]).residue_sequence(3,(0,1)).row_
↪standard_tableaux().quantum_characteristic()
3
```

```
>>> from sage.all import *
>>> RowStandardTableau([[Integer(2),Integer(3)],[Integer(1)]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().
↪quantum_characteristic()
3
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3)]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().
↪quantum_characteristic()
3
>>> RowStandardTableauTuple([[[Integer(4)]],[[Integer(2),Integer(3)],
↪[Integer(1)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
↪standard_tableaux().quantum_characteristic()
3
>>> StandardTableauTuple([[[Integer(4)]],[[Integer(1),Integer(3)],
↪[Integer(2)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
↪standard_tableaux().quantum_characteristic()
3
```

residue_sequence()

Return the residue sequence of `self`.

EXAMPLES:

```
sage: RowStandardTableau([[2,3],[1]]).residue_sequence(3,(0,1)).row_standard_
↪tableaux().residue_sequence()
3-residue sequence (2,0,1) with multicharge (0,1)
sage: StandardTableau([[1,2],[3]]).residue_sequence(3,(0,1)).row_standard_
↪tableaux().residue_sequence()
3-residue sequence (0,1,2) with multicharge (0,1)
sage: RowStandardTableauTuple([[[4]],[[2,3],[1]]]).residue_sequence(3,(0,1)).
↪row_standard_tableaux().residue_sequence()
3-residue sequence (0,1,2,0) with multicharge (0,1)
sage: StandardTableauTuple([[[4]],[[1,3],[2]]]).residue_sequence(3,(0,1)).row_
↪standard_tableaux().residue_sequence()
3-residue sequence (1,0,2,0) with multicharge (0,1)
```

```
>>> from sage.all import *
>>> RowStandardTableau([[Integer(2),Integer(3)],[Integer(1)]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().
↪residue_sequence()
```

(continues on next page)

(continued from previous page)

```
3-residue sequence (2,0,1) with multicharge (0,1)
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3)]]).residue_
→sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().
→residue_sequence()
3-residue sequence (0,1,2) with multicharge (0,1)
>>> RowStandardTableauTuple([[[Integer(4)]],[[Integer(2),Integer(3)],
→[Integer(1)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
→standard_tableaux().residue_sequence()
3-residue sequence (0,1,2,0) with multicharge (0,1)
>>> StandardTableauTuple([[[Integer(4)]],[[Integer(1),Integer(3)],
→[Integer(2)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
→standard_tableaux().residue_sequence()
3-residue sequence (1,0,2,0) with multicharge (0,1)
```

size()

Return the size of `self`.

EXAMPLES:

```
sage: RowStandardTableau([[2,3],[1]]).residue_sequence(3,(0,1)).row_standard_
→tableaux().size()
3
sage: StandardTableau([[1,2],[3]]).residue_sequence(3,(0,1)).row_standard_
→tableaux().size()
3
sage: RowStandardTableauTuple([[[4]],[[2,3],[1]]]).residue_sequence(3,(0,1)).
→row_standard_tableaux().size()
4
sage: StandardTableauTuple([[[4]],[[1,3],[2]]]).residue_sequence(3,(0,1)).row_
→standard_tableaux().size()
4
```

```
>>> from sage.all import *
>>> RowStandardTableau([[Integer(2),Integer(3)],[Integer(1)]]).residue_
→sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().size()
3
>>> StandardTableau([[Integer(1),Integer(2)],[Integer(3)]]).residue_
→sequence(Integer(3),(Integer(0),Integer(1))).row_standard_tableaux().size()
3
>>> RowStandardTableauTuple([[[Integer(4)]],[[Integer(2),Integer(3)],
→[Integer(1)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
→standard_tableaux().size()
4
>>> StandardTableauTuple([[[Integer(4)]],[[Integer(1),Integer(3)],
→[Integer(2)]]]).residue_sequence(Integer(3),(Integer(0),Integer(1))).row_
→standard_tableaux().size()
4
```

class sage.combinat.tableau_tuple.**RowStandardTableauTuples_residue_shape**(*residue*, *shape*)

Bases: *RowStandardTableauTuples_residue*

All row standard tableau tuples with a fixed residue and shape.

INPUT:

- `shape` – the shape of the partitions or partition tuples
- `residue` – the residue sequence of the label

EXAMPLES:

```
sage: res = RowStandardTableauTuple([[[3,6],[1]],[[5,7],[4],[2]]]).residue_
↪sequence(3,(0,0))
sage: tabs = res.row_standard_tableaux([[2,1],[2,1,1]]); tabs
Row standard (2,1|2,1^2)-tableaux with 3-residue sequence (2,1,0,2,0,1,1) and␣
↪multicharge (0,0)
sage: tabs.shape()
([2, 1], [2, 1, 1])
sage: tabs.level()
2
sage: tabs[:6]
[([[5, 7], [4]], [[3, 6], [1], [2]]),
 ([[5, 7], [1]], [[3, 6], [4], [2]]),
 ([[3, 7], [4]], [[5, 6], [1], [2]]),
 ([[3, 7], [1]], [[5, 6], [4], [2]]),
 ([[5, 6], [4]], [[3, 7], [1], [2]]),
 ([[5, 6], [1]], [[3, 7], [4], [2]])]
```

```
>>> from sage.all import *
>>> res = RowStandardTableauTuple([[[Integer(3),Integer(6)],[Integer(1)]],
↪[[Integer(5),Integer(7)],[Integer(4)],[Integer(2)]]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(0)))
>>> tabs = res.row_standard_tableaux([[Integer(2),Integer(1)],[Integer(2),
↪Integer(1),Integer(1)]]); tabs
Row standard (2,1|2,1^2)-tableaux with 3-residue sequence (2,1,0,2,0,1,1) and␣
↪multicharge (0,0)
>>> tabs.shape()
([2, 1], [2, 1, 1])
>>> tabs.level()
2
>>> tabs[:Integer(6)]
[([[5, 7], [4]], [[3, 6], [1], [2]]),
 ([[5, 7], [1]], [[3, 6], [4], [2]]),
 ([[3, 7], [4]], [[5, 6], [1], [2]]),
 ([[3, 7], [1]], [[5, 6], [4], [2]]),
 ([[5, 6], [4]], [[3, 7], [1], [2]]),
 ([[5, 6], [1]], [[3, 7], [4], [2]])]
```

class sage.combinat.tableau_tuple.**RowStandardTableauTuples_shape**(*shape*)

Bases: *RowStandardTableauTuples*

Class of all *RowStandardTableauTuples* of a fixed shape.

an_element()

Return a particular element of `self`.

EXAMPLES:

```
sage: RowStandardTableauTuples([[2],[2,1]]).an_element()                    #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
([[4, 5]], [[1, 3], [2]])
sage: RowStandardTableauTuples([[10],[],[]]).an_element()                    #␣
↪needs sage.graphs
([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]], [], [])
```

```
>>> from sage.all import *
>>> RowStandardTableauTuples([[Integer(2)],[Integer(2),Integer(1)]]).an_
↪element()                          # needs sage.graphs
([[4, 5]], [[1, 3], [2]])
>>> RowStandardTableauTuples([[Integer(10)],[],[]]).an_element()              ␣
↪       # needs sage.graphs
([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]], [], [])
```

cardinality()

Return the number of row standard tableau tuples of with the same shape as the partition tuple `self`.

This is just the index of the corresponding Young subgroup in the full symmetric group.

EXAMPLES:

```
sage: RowStandardTableauTuples([[3,2,1],[]]).cardinality()
60
sage: RowStandardTableauTuples([[1],[1],[1]]).cardinality()
6
sage: RowStandardTableauTuples([[2,1],[1],[1]]).cardinality()
60
```

```
>>> from sage.all import *
>>> RowStandardTableauTuples([[Integer(3),Integer(2),Integer(1)],[]]).
↪cardinality()
60
>>> RowStandardTableauTuples([[Integer(1)],[Integer(1)],[Integer(1)]]).
↪cardinality()
6
>>> RowStandardTableauTuples([[Integer(2),Integer(1)],[Integer(1)],
↪[Integer(1)]]).cardinality()
60
```

class sage.combinat.tableau_tuple.**RowStandardTableauTuples_size**(*size*)

Bases: *RowStandardTableauTuples*, `DisjointUnionEnumeratedSets`

Class of all *RowStandardTableauTuples* with an arbitrary `level` and a fixed `size`.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: RowStandardTableauTuples(size=2).an_element()
([[1]], [[2]], [], [])
sage: RowStandardTableauTuples(size=4).an_element()
([[1]], [[2, 3, 4]], [], [])
```

```
>>> from sage.all import *
>>> RowStandardTableauTuples(size=Integer(2)).an_element()
([[1]], [[2]], [], [])
>>> RowStandardTableauTuples(size=Integer(4)).an_element()
([[1]], [[2, 3, 4]], [], [])
```

class sage.combinat.tableau_tuple.**StandardTableauTuple**(*parent*, *t*, *check=True*)

Bases: *RowStandardTableauTuple*

A class to model a standard tableau of shape a partition tuple. This is a tuple of standard tableau with entries $1, 2, \ldots, n$, where n is the size of the underlying partition tuple, such that the entries increase along rows and down columns in each component of the tuple.

> sage: s = StandardTableauTuple([[1,2,3],[4,5]]) sage: t = StandardTableauTuple([[1,2],[3,5],[4]]) sage: s.dominates(t) True sage: t.dominates(s) False sage: StandardTableauTuple([[1,2,3],[4,5]]) in RowStandardTableauTuples() True

The tableaux appearing in a *StandardTableauTuple* are both row and column strict, but individually they are not standard tableaux because the entries in any single component of a *StandardTableauTuple* will typically not be in bijection with $\{1, 2, \ldots, n\}$.

INPUT:

- `t` – a tableau, a list of (standard) tableau or an equivalent list

OUTPUT: a *StandardTableauTuple* object constructed from `t`

> **Note**
>
> Sage uses the English convention for (tuples of) partitions and tableaux: the longer rows are displayed on top. As with *PartitionTuple*, in sage the cells, or nodes, of partition tuples are 0-based. For example, the (lexicographically) first cell in any non-empty partition tuple is $[0, 0, 0]$. Further, the coordinates `[k,r,c]` in a *TableauTuple* refer to the component, row and column indices, respectively.

EXAMPLES:

```
sage: t = TableauTuple([ [[1,3,4],[7,9]], [[2,8,11],[6]], [[5,10]] ])
sage: t
([[1, 3, 4], [7, 9]], [[2, 8, 11], [6]], [[5, 10]])
sage: t[0][0][0]
1
sage: t[1][1][0]
6
sage: t[2][0][0]
5
sage: t[2][0][1]
10

sage: t = StandardTableauTuple([[[4,5],[7]],[[1,2,3],[6,8]],[[9]]]); t
([[4, 5], [7]], [[1, 2, 3], [6, 8]], [[9]])
sage: t.pp()
  4  5     1  2  3     9
  7        6  8
sage: t.shape()
([2, 1], [3, 2], [1])
```

(continues on next page)

(continued from previous page)

```
sage: t[0].pp()  # pretty printing
  4  5
  7
sage: t.is_standard()
True
sage: t[0].is_standard()
False
sage: StandardTableauTuple([[],[],[]]) # An empty tableau tuple
([], [], [])
```

```
>>> from sage.all import *
>>> t = TableauTuple([ [[Integer(1),Integer(3),Integer(4)],[Integer(7),
↪Integer(9)]], [[Integer(2),Integer(8),Integer(11)],[Integer(6)]], [[Integer(5),
↪Integer(10)]] ])
>>> t
([[1, 3, 4], [7, 9]], [[2, 8, 11], [6]], [[5, 10]])
>>> t[Integer(0)][Integer(0)][Integer(0)]
1
>>> t[Integer(1)][Integer(1)][Integer(0)]
6
>>> t[Integer(2)][Integer(0)][Integer(0)]
5
>>> t[Integer(2)][Integer(0)][Integer(1)]
10

>>> t = StandardTableauTuple([[[Integer(4),Integer(5)],[Integer(7)]],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(6),Integer(8)]],[[Integer(9)]]]); t
([[4, 5], [7]], [[1, 2, 3], [6, 8]], [[9]])
>>> t.pp()
  4  5     1  2  3     9
  7        6  8
>>> t.shape()
([2, 1], [3, 2], [1])
>>> t[Integer(0)].pp()  # pretty printing
  4  5
  7
>>> t.is_standard()
True
>>> t[Integer(0)].is_standard()
False
>>> StandardTableauTuple([[],[],[]]) # An empty tableau tuple
([], [], [])
```

When using code that will generate a lot of tableaux, it is slightly more efficient to construct a `StandardTableauTuple` from the appropriate parent object:

```
sage: STT = StandardTableauTuples()
sage: STT([[[4,5],[7]],[[1,2,3],[6,8]],[[9]]])
([[4, 5], [7]], [[1, 2, 3], [6, 8]], [[9]])
```

```
>>> from sage.all import *
>>> STT = StandardTableauTuples()
```

(continues on next page)

(continued from previous page)

```
>>> STT([[[Integer(4),Integer(5)],[Integer(7)]],[[Integer(1),Integer(2),
↪Integer(3)],[Integer(6),Integer(8)]],[[Integer(9)]]])
([[4, 5], [7]], [[1, 2, 3], [6, 8]], [[9]])
```

See also

- *Tableau*
- *Tableaux*
- *TableauTuples*
- *TableauTuple*
- *StandardTableauTuples*

dominates(*t*)

Return `True` if the tableau (tuple) `self` dominates the tableau `t`. The two tableaux do not need to be of the same shape.

EXAMPLES:

```
sage: s = StandardTableauTuple([[1,2,3],[4,5]])
sage: t = StandardTableauTuple([[1,2],[3,5],[4]])
sage: s.dominates(t)
True
sage: t.dominates(s)
False
```

```
>>> from sage.all import *
>>> s = StandardTableauTuple([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]])
>>> t = StandardTableauTuple([[Integer(1),Integer(2)],[Integer(3),Integer(5)],
↪[Integer(4)]])
>>> s.dominates(t)
True
>>> t.dominates(s)
False
```

restrict(*m=None*)

Return the restriction of the standard tableau `self` to `m`, which defaults to one less than the current *size()*.

EXAMPLES:

```
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(6)
([[5]], [[1, 2], [3, 4]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(5)
([[5]], [[1, 2], [3, 4]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(4)
([], [[1, 2], [3, 4]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3)
([], [[1, 2], [3]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(2)
```

(continues on next page)

(continued from previous page)

```
([], [[1, 2]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(1)
([], [[1]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(0)
([], [])
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(6))
([[5]], [[1, 2], [3, 4]])
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(5))
([[5]], [[1, 2], [3, 4]])
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(4))
([], [[1, 2], [3, 4]])
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(3))
([], [[1, 2], [3]])
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(2))
([], [[1, 2]])
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(1))
([], [[1]])
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(0))
([], [])
```

Where possible the restricted tableau belongs to the same category as the tableau `self`:

```
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Tableau tuples
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Standard tableau tuples
sage: StandardTableauTuples([[1],[2,2]])([[[5]],[[1,2],[3,4]]]).restrict(3).
↪category()
Category of elements of Standard tableau tuples
sage: StandardTableauTuples(level=2)([[[5]],[[1,2],[3,4]]]).restrict(3).
↪category()
Category of elements of Standard tableau tuples of level 2
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(3)).category()
Category of elements of Tableau tuples
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(3)).category()
Category of elements of Standard tableau tuples
>>> StandardTableauTuples([[Integer(1)],[Integer(2),
↪Integer(2)]])([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict(Integer(3)).category()
```

(continues on next page)

(continued from previous page)

```
Category of elements of Standard tableau tuples
>>> StandardTableauTuples(level=Integer(2))([[[Integer(5)]],[[Integer(1),
↪Integer(2)],[Integer(3),Integer(4)]]]).restrict(Integer(3)).category()
Category of elements of Standard tableau tuples of level 2
```

to_chain()

Return the chain of partitions corresponding to the standard tableau tuple `self`.

EXAMPLES:

```
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).to_chain()
[([], []),
 ([], [1]),
 ([], [2]),
 ([], [2, 1]),
 ([], [2, 2]),
 ([1], [2, 2])]
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).to_chain()
[([], []),
 ([], [1]),
 ([], [2]),
 ([], [2, 1]),
 ([], [2, 2]),
 ([1], [2, 2])]
```

class sage.combinat.tableau_tuple.**StandardTableauTuples**

Bases: *RowStandardTableauTuples*

A factory class for the various classes of tuples of standard tableau.

INPUT:

There are three optional arguments:

- `level` – the *level()* of the tuples of tableaux
- `size` – the *size()* of the tuples of tableaux
- `shape` – list or a partition tuple specifying the *shape()* of the standard tableau tuples

It is not necessary to use the keywords. If they are not used then the first integer argument specifies the *level()* and the second the *size()* of the tableau tuples.

OUTPUT: the appropriate subclass of *StandardTableauTuples*

A tuple of standard tableau is a tableau whose entries are positive integers which increase from left to right along the rows, and from top to bottom down the columns, in each component. The entries do NOT need to increase from left to right along the components.

Note

Sage uses the English convention for (tuples of) partitions and tableaux: the longer rows are displayed on top. As with *PartitionTuple*, in sage the cells, or nodes, of partition tuples are 0-based. For example, the (lexicographically) first cell in any non-empty partition tuple is $[0, 0, 0]$.

EXAMPLES:

```
sage: tabs=StandardTableauTuples([[3],[2,2]]); tabs
Standard tableau tuples of shape ([3], [2, 2])
sage: tabs.cardinality()
70
sage: tabs[10:16]
[([[1, 2, 3]], [[4, 6], [5, 7]]),
 ([[1, 2, 4]], [[3, 6], [5, 7]]),
 ([[1, 3, 4]], [[2, 6], [5, 7]]),
 ([[2, 3, 4]], [[1, 6], [5, 7]]),
 ([[1, 2, 5]], [[3, 6], [4, 7]]),
 ([[1, 3, 5]], [[2, 6], [4, 7]])]

sage: tabs=StandardTableauTuples(level=3); tabs
Standard tableau tuples of level 3
sage: tabs[100]                                                               #␣
↪needs sage.libs.flint
([[1, 2], [3]], [], [[4]])

sage: StandardTableauTuples()[0]                                              #␣
↪needs sage.libs.flint
()
```

```
>>> from sage.all import *
>>> tabs=StandardTableauTuples([[Integer(3)],[Integer(2),Integer(2)]]); tabs
Standard tableau tuples of shape ([3], [2, 2])
>>> tabs.cardinality()
70
>>> tabs[Integer(10):Integer(16)]
[([[1, 2, 3]], [[4, 6], [5, 7]]),
 ([[1, 2, 4]], [[3, 6], [5, 7]]),
 ([[1, 3, 4]], [[2, 6], [5, 7]]),
 ([[2, 3, 4]], [[1, 6], [5, 7]]),
 ([[1, 2, 5]], [[3, 6], [4, 7]]),
 ([[1, 3, 5]], [[2, 6], [4, 7]])]

>>> tabs=StandardTableauTuples(level=Integer(3)); tabs
Standard tableau tuples of level 3
>>> tabs[Integer(100)]                                                        ␣
↪     # needs sage.libs.flint
([[1, 2], [3]], [], [[4]])

>>> StandardTableauTuples()[Integer(0)]                                       ␣
↪     # needs sage.libs.flint
()
```

See also

- *TableauTuples*
- *Tableau*
- *StandardTableau*
- *StandardTableauTuples*

Element

alias of *StandardTableauTuple*

level_one_parent_class

alias of *StandardTableaux_all*

shape()

Return the shape of the set of *StandardTableauTuples*, or `None` if it is not defined.

EXAMPLES:

```
sage: tabs=StandardTableauTuples(shape=[[5,2],[3,2],[],[1,1,1],[3]]); tabs
Standard tableau tuples of shape ([5, 2], [3, 2], [], [1, 1, 1], [3])
sage: tabs.shape()
([5, 2], [3, 2], [], [1, 1, 1], [3])
sage: StandardTableauTuples().shape() is None
True
```

```
>>> from sage.all import *
>>> tabs=StandardTableauTuples(shape=[[Integer(5),Integer(2)],[Integer(3),
↪Integer(2)],[],[Integer(1),Integer(1),Integer(1)],[Integer(3)]]); tabs
Standard tableau tuples of shape ([5, 2], [3, 2], [], [1, 1, 1], [3])
>>> tabs.shape()
([5, 2], [3, 2], [], [1, 1, 1], [3])
>>> StandardTableauTuples().shape() is None
True
```

class sage.combinat.tableau_tuple.**StandardTableauTuples_all**

Bases: *StandardTableauTuples*, `DisjointUnionEnumeratedSets`

Default class of all *StandardTableauTuples* with an arbitrary *level()* and *size()*.

class sage.combinat.tableau_tuple.**StandardTableauTuples_level**(*level*)

Bases: *StandardTableauTuples*, `DisjointUnionEnumeratedSets`

Class of all *StandardTableauTuples* with a fixed `level` and arbitrary `size`.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples(size=2).an_element()
([[1]], [[2]], [], [])
sage: StandardTableauTuples(size=4).an_element()
([[1]], [[2, 3, 4]], [], [])
```

```
>>> from sage.all import *
>>> StandardTableauTuples(size=Integer(2)).an_element()
([[1]], [[2]], [], [])
>>> StandardTableauTuples(size=Integer(4)).an_element()
([[1]], [[2, 3, 4]], [], [])
```

class sage.combinat.tableau_tuple.**StandardTableauTuples_level_size**(*level*, *size*)

Bases: *StandardTableauTuples*, DisjointUnionEnumeratedSets

Class of all *StandardTableauTuples* with a fixed level and a fixed size.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples(5, size=2).an_element()                     #␣
↪needs sage.libs.flint
([], [], [], [], [[1], [2]])
sage: StandardTableauTuples(2, size=4).an_element()                     #␣
↪needs sage.libs.flint
([[1]], [[2, 3], [4]])
```

```
>>> from sage.all import *
>>> StandardTableauTuples(Integer(5), size=Integer(2)).an_element()        ␣
↪               # needs sage.libs.flint
([], [], [], [], [[1], [2]])
>>> StandardTableauTuples(Integer(2), size=Integer(4)).an_element()        ␣
↪               # needs sage.libs.flint
([[1]], [[2, 3], [4]])
```

cardinality()

Return the number of elements in this set of tableaux.

EXAMPLES:

```
sage: StandardTableauTuples(3,2).cardinality()                           #␣
↪needs sage.libs.flint
12
sage: StandardTableauTuples(4,6).cardinality()                           #␣
↪needs sage.libs.flint
31936
```

```
>>> from sage.all import *
>>> StandardTableauTuples(Integer(3),Integer(2)).cardinality()             ␣
↪               # needs sage.libs.flint
12
>>> StandardTableauTuples(Integer(4),Integer(6)).cardinality()             ␣
↪               # needs sage.libs.flint
31936
```

class sage.combinat.tableau_tuple.**StandardTableauTuples_shape**(*shape*)

Bases: *StandardTableauTuples*

Class of all *StandardTableauTuples* of a fixed shape.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples([[2],[2,1]]).an_element()
([[2, 4]], [[1, 3], [5]])
sage: StandardTableauTuples([[10],[],[]]).an_element()
([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]], [], [])
```

```
>>> from sage.all import *
>>> StandardTableauTuples([[Integer(2)],[Integer(2),Integer(1)]]).an_element()
([[2, 4]], [[1, 3], [5]])
>>> StandardTableauTuples([[Integer(10)],[],[]]).an_element()
([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]], [], [])
```

cardinality()

Return the number of standard Young tableau tuples of with the same shape as the partition tuple `self`.

Let $\mu = (\mu^{(1)}, \ldots, \mu^{(l)})$ be the `shape` of the tableaux in `self` and let $m_k = |\mu^{(k)}|$, for $1 \leq k \leq l$. Multiplying by a (unique) coset representative of the Young subgroup $S_{m_1} \times \cdots \times S_{m_l}$ inside the symmetric group S_n, we can assume that t is standard and the numbers $1, 2..., n$ are entered in order from to right along the components of the tableau. Therefore, there are

$$\binom{n}{m_1, \ldots, m_l} \prod_{k=1}^{l} |\mathrm{Std}(\mu^{(k)})|$$

standard tableau tuples of this shape, where $|\mathrm{Std}(\mu^{(k)})|$ is the number of standard tableau of shape $\mu^{(k)}$, for $1 \leq k \leq l$. This is given by the hook length formula.

EXAMPLES:

```
sage: StandardTableauTuples([[3,2,1],[]]).cardinality()
16
sage: StandardTableauTuples([[1],[1],[1]]).cardinality()
6
sage: StandardTableauTuples([[2,1],[1],[1]]).cardinality()
40
sage: StandardTableauTuples([[3,2,1],[3,2]]).cardinality()
36960
```

```
>>> from sage.all import *
>>> StandardTableauTuples([[Integer(3),Integer(2),Integer(1)],[]]).
↪cardinality()
16
>>> StandardTableauTuples([[Integer(1)],[Integer(1)],[Integer(1)]]).
↪cardinality()
6
>>> StandardTableauTuples([[Integer(2),Integer(1)],[Integer(1)],
↪[Integer(1)]]).cardinality()
40
>>> StandardTableauTuples([[Integer(3),Integer(2),Integer(1)],[Integer(3),
↪Integer(2)]]).cardinality()
36960
```

last()

Return the last standard tableau tuple in self, with respect to the order that they are generated by the iterator.

This is just the standard tableau tuple with the numbers $1, 2, \ldots, n$, where n is *size()*, entered in order down the columns form right to left along the components.

EXAMPLES:

```
sage: StandardTableauTuples([[2],[2,2]]).last().pp()
  5  6     1  3
           2  4
```

```
>>> from sage.all import *
>>> StandardTableauTuples([[Integer(2)],[Integer(2),Integer(2)]]).last().pp()
  5  6     1  3
           2  4
```

random_element()

Return a random standard tableau in self.

We do this by randomly selecting addable nodes to place $1, 2, \ldots, n$. Of course we could do this recursively, but it is more efficient to keep track of the (changing) list of addable nodes as we go.

EXAMPLES:

```
sage: StandardTableauTuples([[2],[2,1]]).random_element()  # random
([[1, 2]], [[3, 4], [5]])
```

```
>>> from sage.all import *
>>> StandardTableauTuples([[Integer(2)],[Integer(2),Integer(1)]]).random_
↪element()  # random
([[1, 2]], [[3, 4], [5]])
```

class sage.combinat.tableau_tuple.**StandardTableauTuples_size**(*size*)

Bases: *StandardTableauTuples*, DisjointUnionEnumeratedSets

Class of all *StandardTableauTuples* with an arbitrary level and a fixed size.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples(size=2).an_element()
([[1]], [[2]], [], [])
sage: StandardTableauTuples(size=4).an_element()
([[1]], [[2, 3, 4]], [], [])
```

```
>>> from sage.all import *
>>> StandardTableauTuples(size=Integer(2)).an_element()
([[1]], [[2]], [], [])
>>> StandardTableauTuples(size=Integer(4)).an_element()
([[1]], [[2, 3, 4]], [], [])
```

class sage.combinat.tableau_tuple.**StandardTableaux_residue**(*residue*)

Bases: *StandardTableauTuples*

Class of all standard tableau tuples with a fixed residue sequence.

Implicitly, this also specifies the quantum characteristic, multicharge and hence the level and size of the tableaux.

> **Note**
>
> This class is not intended to be called directly, but rather, it is accessed through the standard tableaux.

EXAMPLES:

```
sage: StandardTableau([[1,2,3],[4,5]]).residue_sequence(2).standard_tableaux()
Standard tableaux with 2-residue sequence (0,1,0,1,0) and multicharge (0)
sage: StandardTableau([[1,2,3],[4,5]]).residue_sequence(3).standard_tableaux()
Standard tableaux with 3-residue sequence (0,1,2,2,0) and multicharge (0)
sage: StandardTableauTuple([[[5,6],[7]],[[1,2,3],[4]]]).residue_sequence(2,(0,0)).
↪standard_tableaux()
Standard tableaux with 2-residue sequence (0,1,0,1,0,1,1) and multicharge (0,0)
sage: StandardTableauTuple([[[5,6],[7]],[[1,2,3],[4]]]).residue_sequence(3,(0,1)).
↪standard_tableaux()
Standard tableaux with 3-residue sequence (1,2,0,0,0,1,2) and multicharge (0,1)
```

```
>>> from sage.all import *
>>> StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]).
↪residue_sequence(Integer(2)).standard_tableaux()
Standard tableaux with 2-residue sequence (0,1,0,1,0) and multicharge (0)
>>> StandardTableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]).
↪residue_sequence(Integer(3)).standard_tableaux()
Standard tableaux with 3-residue sequence (0,1,2,2,0) and multicharge (0)
>>> StandardTableauTuple([[[Integer(5),Integer(6)],[Integer(7)]],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(4)]]]).residue_sequence(Integer(2),(Integer(0),
↪Integer(0))).standard_tableaux()
Standard tableaux with 2-residue sequence (0,1,0,1,0,1,1) and multicharge (0,0)
>>> StandardTableauTuple([[[Integer(5),Integer(6)],[Integer(7)]],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(4)]]]).residue_sequence(Integer(3),(Integer(0),
↪Integer(1))).standard_tableaux()
Standard tableaux with 3-residue sequence (1,2,0,0,0,1,2) and multicharge (0,1)
```

class sage.combinat.tableau_tuple.**StandardTableaux_residue_shape**(*residue*, *shape*)

Bases: *StandardTableaux_residue*

All standard tableau tuples with a fixed residue and shape.

INPUT:

- `shape` – the shape of the partitions or partition tuples
- `residue` – the residue sequence of the label

EXAMPLES:

```
sage: res = StandardTableauTuple([[[1,3],[6]],[[2,7],[4],[5]]]).residue_
↪sequence(3,(0,0))
```

(continues on next page)

(continued from previous page)

```
sage: tabs = res.standard_tableaux([[2,1],[2,1,1]]); tabs
Standard (2,1|2,1^2)-tableaux with 3-residue sequence (0,0,1,2,1,2,1) and␣
↪multicharge (0,0)
sage: tabs.shape()
([2, 1], [2, 1, 1])
sage: tabs.level()
2
sage: tabs[:6]
[([[2, 7], [6]], [[1, 3], [4], [5]]),
 ([[1, 7], [6]], [[2, 3], [4], [5]]),
 ([[2, 3], [6]], [[1, 7], [4], [5]]),
 ([[1, 3], [6]], [[2, 7], [4], [5]]),
 ([[2, 5], [6]], [[1, 3], [4], [7]]),
 ([[1, 5], [6]], [[2, 3], [4], [7]])]
```

```
>>> from sage.all import *
>>> res = StandardTableauTuple([[[Integer(1),Integer(3)],[Integer(6)]],
↪[[Integer(2),Integer(7)],[Integer(4)],[Integer(5)]]]).residue_
↪sequence(Integer(3),(Integer(0),Integer(0)))
>>> tabs = res.standard_tableaux([[Integer(2),Integer(1)],[Integer(2),Integer(1),
↪Integer(1)]]); tabs
Standard (2,1|2,1^2)-tableaux with 3-residue sequence (0,0,1,2,1,2,1) and␣
↪multicharge (0,0)
>>> tabs.shape()
([2, 1], [2, 1, 1])
>>> tabs.level()
2
>>> tabs[:Integer(6)]
[([[2, 7], [6]], [[1, 3], [4], [5]]),
 ([[1, 7], [6]], [[2, 3], [4], [5]]),
 ([[2, 3], [6]], [[1, 7], [4], [5]]),
 ([[1, 3], [6]], [[2, 7], [4], [5]]),
 ([[2, 5], [6]], [[1, 3], [4], [7]]),
 ([[1, 5], [6]], [[2, 3], [4], [7]])]
```

an_element()

Return a particular element of `self`.

EXAMPLES:

```
sage: T = StandardTableau([[1,3],[2]]).residue_sequence(3).standard_
↪tableaux([2,1])
sage: T.an_element()
[[1, 3], [2]]
```

```
>>> from sage.all import *
>>> T = StandardTableau([[Integer(1),Integer(3)],[Integer(2)]]).residue_
↪sequence(Integer(3)).standard_tableaux([Integer(2),Integer(1)])
>>> T.an_element()
[[1, 3], [2]]
```

class sage.combinat.tableau_tuple.**TableauTuple**(*parent*, *t*, *check=True*)

Bases: *CombinatorialElement*

A class to model a tuple of tableaux.

INPUT:

- `t` – list or tuple of `Tableau`, a list or tuple of lists of lists

OUTPUT: the Tableau tuple object constructed from `t`

A `TableauTuple` is a tuple of tableau of shape a `PartitionTuple`. These combinatorial objects are useful is several areas of algebraic combinatorics. In particular, they are important in:

- the representation theory of the complex reflection groups of type $G(l, 1, n)$ and the representation theory of the associated (degenerate and non-degenerate) Hecke algebras. See, for example, [DJM1998]
- the crystal theory of (quantum) affine special linear groups and its integral highest weight modules and their canonical bases. See, for example, [BK2009].

These apparently different and unrelated contexts are, in fact, intimately related as in characteristic zero the cyclotomic Hecke algebras categorify the canonical bases of the integral highest weight modules of the quantum affine special linear groups.

The `level()` of a tableau tuple is the length of the tuples. This corresponds to the level of the corresponding highest weight module.

In sage a `TableauTuple` looks and behaves like a real tuple of (level 1) `Tableaux`. Many of the operations which are defined on `Tableau` extend to `TableauTuples`. Tableau tuples of level 1 are just ordinary `Tableau`.

In sage, the entries of `Tableaux` can be very general, including arbitrarily nested lists, so some lists can be interpreted either as a tuple of tableaux or simply as tableaux. If it is possible to interpret the input to `TableauTuple` as a tuple of tableaux then `TableauTuple` returns the corresponding tuple. Given a 1-tuple of tableaux the tableau itself is returned.

EXAMPLES:

```
sage: t = TableauTuple([ [[6,9,10],[11]], [[1,2,3],[4,5]], [[7],[8]] ]); t
([[6, 9, 10], [11]], [[1, 2, 3], [4, 5]], [[7], [8]])
sage: t.level()
3
sage: t.size()
11
sage: t.shape()
([3, 1], [3, 2], [1, 1])
sage: t.is_standard()
True
sage: t.pp() # pretty printing
 6  9 10     1  2  3     7
11           4  5        8
sage: t.category()
Category of elements of Tableau tuples
sage: t.parent()
Tableau tuples

sage: s = TableauTuple([ [['a','c','b'],['d','e']],[[(2,1)]]]); s
([['a', 'c', 'b'], ['d', 'e']], [[(2, 1)]])
sage: s.shape()
([3, 2], [1])
sage: s.size()
6
```

(continues on next page)

(continued from previous page)

```
sage: TableauTuple([[],[],[]])  # The empty 3-tuple of tableaux
([], [], [])

sage: TableauTuple([[1,2,3],[4,5]])
[[1, 2, 3], [4, 5]]
sage: TableauTuple([[1,2,3],[4,5]]) == Tableau([[1,2,3],[4,5]])
True
```

```
>>> from sage.all import *
>>> t = TableauTuple([ [[Integer(6),Integer(9),Integer(10)],[Integer(11)]],␣
↪[[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]], [[Integer(7)],
↪[Integer(8)]] ]); t
([[6, 9, 10], [11]], [[1, 2, 3], [4, 5]], [[7], [8]])
>>> t.level()
3
>>> t.size()
11
>>> t.shape()
([3, 1], [3, 2], [1, 1])
>>> t.is_standard()
True
>>> t.pp() # pretty printing
 6  9 10     1  2  3     7
11           4  5        8
>>> t.category()
Category of elements of Tableau tuples
>>> t.parent()
Tableau tuples

>>> s = TableauTuple([ [['a','c','b'],['d','e']],[[(Integer(2),Integer(1))]]]); s
([['a', 'c', 'b'], ['d', 'e']], [[(2, 1)]])
>>> s.shape()
([3, 2], [1])
>>> s.size()
6

>>> TableauTuple([[],[],[]])  # The empty 3-tuple of tableaux
([], [], [])

>>> TableauTuple([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])
[[1, 2, 3], [4, 5]]
>>> TableauTuple([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]) ==␣
↪Tableau([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]])
True
```

See also

- *StandardTableauTuple*
- *StandardTableauTuples*
- *StandardTableau*

- *StandardTableaux*
- *TableauTuple*
- *TableauTuples*
- *Tableau*
- *Tableaux*

Element

alias of *Tableau*

add_entry(*cell*, *m*)

Set the entry in `cell` equal to `m`. If the cell does not exist then extend the tableau, otherwise just replace the entry.

EXAMPLES:

```
sage: s = StandardTableauTuple([ [[3,4,7],[6,8]], [[9,13],[12]], [[1,5],[2,
↪11],[10]] ]); s.pp()
  3  4  7     9 13     1  5
  6  8       12        2 11
                      10
sage: t = s.add_entry( (0,0,3),14); t.pp(); t.category()
  3  4  7 14     9 13     1  5
  6  8          12        2 11
                         10
Category of elements of Standard tableau tuples
sage: t = s.add_entry( (0,0,3),15); t.pp(); t.category()
  3  4  7 15     9 13     1  5
  6  8          12        2 11
                         10
Category of elements of Tableau tuples
sage: t = s.add_entry( (1,1,1),14); t.pp(); t.category()
  3  4  7     9 13     1  5
  6  8       12 14     2 11
                      10
Category of elements of Standard tableau tuples
sage: t = s.add_entry( (2,1,1),14); t.pp(); t.category()
  3  4  7     9 13     1  5
  6  8       12        2 14
                      10
Category of elements of Tableau tuples
sage: t = s.add_entry( (2,1,2),14); t.pp(); t.category()
Traceback (most recent call last):
...
IndexError: (2, 1, 2) is not an addable cell of the tableau
```

```
>>> from sage.all import *
>>> s = StandardTableauTuple([ [[Integer(3),Integer(4),Integer(7)],
↪[Integer(6),Integer(8)]], [[Integer(9),Integer(13)],[Integer(12)]],␣
↪[[Integer(1),Integer(5)],[Integer(2),Integer(11)],[Integer(10)]] ]); s.pp()
  3  4  7     9 13     1  5
  6  8       12        2 11
```

(continues on next page)

(continued from previous page)

```
                        10
>>> t = s.add_entry( (Integer(0),Integer(0),Integer(3)),Integer(14)); t.pp();␣
↪t.category()
  3  4  7 14     9 13     1  5
  6  8          12        2 11
                         10
Category of elements of Standard tableau tuples
>>> t = s.add_entry( (Integer(0),Integer(0),Integer(3)),Integer(15)); t.pp();␣
↪t.category()
  3  4  7 15     9 13     1  5
  6  8          12        2 11
                         10
Category of elements of Tableau tuples
>>> t = s.add_entry( (Integer(1),Integer(1),Integer(1)),Integer(14)); t.pp();␣
↪t.category()
  3  4  7     9 13     1  5
  6  8       12 14     2 11
                      10
Category of elements of Standard tableau tuples
>>> t = s.add_entry( (Integer(2),Integer(1),Integer(1)),Integer(14)); t.pp();␣
↪t.category()
  3  4  7     9 13     1  5
  6  8       12        2 14
                      10
Category of elements of Tableau tuples
>>> t = s.add_entry( (Integer(2),Integer(1),Integer(2)),Integer(14)); t.pp();␣
↪t.category()
Traceback (most recent call last):
...
IndexError: (2, 1, 2) is not an addable cell of the tableau
```

cells_containing(*m*)

Return the list of cells in which the letter `m` appears in the tableau `self`.

The list is ordered with cells appearing from left to right.

EXAMPLES:

```
sage: t = TableauTuple([[[4,5]],[[1,1,2,4],[2,4,4],[4]],[[1,3,4],[3,4]]])
sage: t.cells_containing(4)
[(0, 0, 0),
 (1, 2, 0),
 (1, 1, 1),
 (1, 1, 2),
 (1, 0, 3),
 (2, 1, 1),
 (2, 0, 2)]
sage: t.cells_containing(6)
[]
```

```
>>> from sage.all import *
>>> t = TableauTuple([[[Integer(4),Integer(5)]],[[Integer(1),Integer(1),
↪Integer(2),Integer(4)],[Integer(2),Integer(4),Integer(4)],[Integer(4)]],
```

(continues on next page)

(continued from previous page)

```
↪[[Integer(1),Integer(3),Integer(4)],[Integer(3),Integer(4)]]])
>>> t.cells_containing(Integer(4))
[(0, 0, 0),
 (1, 2, 0),
 (1, 1, 1),
 (1, 1, 2),
 (1, 0, 3),
 (2, 1, 1),
 (2, 0, 2)]
>>> t.cells_containing(Integer(6))
[]
```

charge()

Return the charge of the reading word of `self`.

See *charge()* for more information.

EXAMPLES:

```
sage: TableauTuple([[[4,5]],[[1,1,2,4],[2,4,4],[4]],[[1,3,4],[3,4]]]).charge()
4
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(4),Integer(5)]],[[Integer(1),Integer(1),
↪Integer(2),Integer(4)],[Integer(2),Integer(4),Integer(4)],[Integer(4)]],
↪[[Integer(1),Integer(3),Integer(4)],[Integer(3),Integer(4)]]]).charge()
4
```

cocharge()

Return the cocharge of the reading word of `self`.

See *cocharge()* for more information.

EXAMPLES:

```
sage: TableauTuple([[[4,5]],[[1,1,2,4],[2,4,4],[4]],[[1,3,4],[3,4]]]).charge()
4
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(4),Integer(5)]],[[Integer(1),Integer(1),
↪Integer(2),Integer(4)],[Integer(2),Integer(4),Integer(4)],[Integer(4)]],
↪[[Integer(1),Integer(3),Integer(4)],[Integer(3),Integer(4)]]]).charge()
4
```

column_stabilizer()

Return the `PermutationGroup` corresponding to `self`. That is, return subgroup of the symmetric group of degree *size()* which is the column stabilizer of `self`.

EXAMPLES:

```
sage: # needs sage.groups
sage: t = TableauTuple([[[1,2,3],[4,5]],[[6,7]],[[8],[9]]])
sage: cs = t.column_stabilizer()
sage: cs.order()
```

(continues on next page)

(continued from previous page)

```
8
sage: PermutationGroupElement([(1,3,2),(4,5)]) in cs
False
sage: PermutationGroupElement([(1,4)]) in cs
True
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> t = TableauTuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]],[[Integer(6),Integer(7)]],[[Integer(8)],[Integer(9)]]])
>>> cs = t.column_stabilizer()
>>> cs.order()
8
>>> PermutationGroupElement([(Integer(1),Integer(3),Integer(2)),(Integer(4),
↪Integer(5))]) in cs
False
>>> PermutationGroupElement([(Integer(1),Integer(4))]) in cs
True
```

components()

Return a list of the components of tableau tuple `self`.

The *components* are the individual `Tableau` which are contained in the tuple `self`.

For compatibility with `TableauTuples` of `level()` 1, `components()` should be used to iterate over the components of `TableauTuples`.

EXAMPLES:

```
sage: for t in TableauTuple([[1,2,3],[4,5]]).components(): t.pp()
  1  2  3
  4  5
sage: for t in TableauTuple([ [[1,2,3],[4,5]], [[6,7],[8,9]] ]).components():␣
↪t.pp()
  1  2  3
  4  5
  6  7
  8  9
```

```
>>> from sage.all import *
>>> for t in TableauTuple([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]]).components(): t.pp()
  1  2  3
  4  5
>>> for t in TableauTuple([ [[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]], [[Integer(6),Integer(7)],[Integer(8),Integer(9)]] ]).
↪components(): t.pp()
  1  2  3
  4  5
  6  7
  8  9
```

conjugate()

Return the conjugate of the tableau tuple `self`.

The conjugate tableau tuple T' is the *TableauTuple* obtained from T by reversing the order of the components and conjugating each component – that is, swapping the rows and columns of the all of *Tableau* in T (see *sage.combinat.tableau.Tableau.conjugate()*).

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).
↪conjugate()
([[9, 11, 12], [10]], [[5, 8], [6], [7]], [[1, 3], [2, 4]])
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).conjugate()
([[9, 11, 12], [10]], [[5, 8], [6], [7]], [[1, 3], [2, 4]])
```

content (*k*, *multicharge*)

Return the content `k` in `self`.

The content of k in a standard tableau. That is, if k appears in row r and column c of the tableau, then we return $c - r + a_k$, where the multicharge is $(a_1, a_2, \ldots, a_l)$ and l is the level of the tableau.

The multicharge determines the dominant weight

$$\Lambda = \sum_{i=1}^{l} \Lambda_{a_i}$$

of the affine special linear group. In the combinatorics, the multicharge simply offsets the contents in each component so that the cell (k, r, c) has content $a_k + c - r$.

INPUT:

- `k` – integer in $\{1, 2, \ldots, n\}$
- `multicharge` – a sequence of integers of length l

Here l is the *level()* and n is the *size()* of `self`.

EXAMPLES:

```
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).content(3,[0,0])
-1
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).content(3,[0,1])
0
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).content(3,[0,2])
1
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).content(6,[0,2])
Traceback (most recent call last):
...
ValueError: 6 must be contained in the tableaux
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).content(Integer(3),[Integer(0),Integer(0)])
-1
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).content(Integer(3),[Integer(0),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
0
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).content(Integer(3),[Integer(0),Integer(2)])
1
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).content(Integer(6),[Integer(0),Integer(2)])
Traceback (most recent call last):
...
ValueError: 6 must be contained in the tableaux
```

entries()

Return a sorted list of all entries of `self`, in the order obtained by reading across the rows.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).entries()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: TableauTuple([[[1,2],[3,4]],[[9,10],[11],[12]],[[5,6,7],[8]]]).entries()
[1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8]
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).entries()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(9),Integer(10)],[Integer(11)],[Integer(12)]],[[Integer(5),
↪Integer(6),Integer(7)],[Integer(8)]]]).entries()
[1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8]
```

entry(l, r, c)

Return the entry of the cell `(l, r, c)` in `self`.

A cell is a tuple `(l, r, c)` of coordinates, where `l` is the component index, `r` is the row index, and `c` is the column index.

EXAMPLES:

```
sage: t = TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]])
sage: t.entry(1, 0, 0)
5
sage: t.entry(1, 1, 1)
Traceback (most recent call last):
...
IndexError: tuple index out of range
```

```
>>> from sage.all import *
>>> t = TableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]])
>>> t.entry(Integer(1), Integer(0), Integer(0))
5
>>> t.entry(Integer(1), Integer(1), Integer(1))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
IndexError: tuple index out of range
```

first_column_descent()

Return the first cell of `self` is not column standard.

Cells are ordered left to right along the rows and then top to bottom. That is, return the cell (k, r, c) with (k, r, c) minimal such that the entry in position (k, r, c) is bigger than the entry in position $(k, r, c+1)$. If there is no such cell then `None` is returned - in this case the tableau is column strict.

OUTPUT:

The cell corresponding to the first column descent or `None` if the tableau is column strict.

EXAMPLES:

```
sage: TableauTuple([[[3,5,6],[2,4,5]],[[1,4,5],[2,3]]]).first_column_descent()
(0, 0, 0)
sage: Tableau([[[1,2,3],[4]],[[5,6,7],[8,9]]]).first_column_descent() is None
True
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(3),Integer(5),Integer(6)],[Integer(2),Integer(4),
↪Integer(5)]],[[Integer(1),Integer(4),Integer(5)],[Integer(2),Integer(3)]]]).
↪first_column_descent()
(0, 0, 0)
>>> Tableau([[[Integer(1),Integer(2),Integer(3)],[Integer(4)]],[[Integer(5),
↪Integer(6),Integer(7)],[Integer(8),Integer(9)]]]).first_column_descent() is␣
↪None
True
```

first_row_descent()

Return the first cell of `self` that is not row standard.

Cells are ordered left to right along the rows and then top to bottom. That is, the cell minimal (k, r, c) such that the entry in position (k, r, c) is bigger than the entry in position $(k, r, c+1)$. If there is no such cell then `None` is returned - in this case the tableau is row strict.

OUTPUT:

The cell corresponding to the first row descent or `None` if the tableau is row strict.

EXAMPLES:

```
sage: TableauTuple([[[5,6,7],[1,2]],[[1,3,2],[4]]]).first_row_descent()
(1, 0, 1)
sage: TableauTuple([[[1,2,3],[4]],[[6,7,8],[1,2,3]],[[1,11]]]).first_row_
↪descent() is None
True
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(5),Integer(6),Integer(7)],[Integer(1),
↪Integer(2)]],[[Integer(1),Integer(3),Integer(2)],[Integer(4)]]]).first_row_
↪descent()
```

(continues on next page)

(continued from previous page)

```
(1, 0, 1)
>>> TableauTuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4)]],
↪[[Integer(6),Integer(7),Integer(8)],[Integer(1),Integer(2),Integer(3)]],
↪[[Integer(1),Integer(11)]]]).first_row_descent() is None
True
```

is_column_strict()

Return `True` if the tableau `self` is column strict and `False` otherwise.

A tableau tuple is *column strict* if the entries in each column of each component are in increasing order, when read from top to bottom.

EXAMPLES:

```
sage: TableauTuple([[[5,7],[8]],[[1, 3], [2, 4]],[[6]]]).is_column_strict()
True
sage: TableauTuple([[[1, 2], [2, 4]],[[4,5,6],[7,8]]]).is_column_strict()
True
sage: TableauTuple([[[1]],[[2, 3], [2, 4]]]).is_column_strict()
False
sage: TableauTuple([[[1]],[[2, 2], [4,5]]]]).is_column_strict()
True
sage: TableauTuple([[[1,2],[6,7]],[[4,8], [6, 9]],[]]).is_column_strict()
True
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(5),Integer(7)],[Integer(8)]],[[Integer(1),␣
↪Integer(3)], [Integer(2), Integer(4)]],[[Integer(6)]]]).is_column_strict()
True
>>> TableauTuple([[[Integer(1), Integer(2)], [Integer(2), Integer(4)]],
↪[[Integer(4),Integer(5),Integer(6)],[Integer(7),Integer(8)]]]).is_column_
↪strict()
True
>>> TableauTuple([[[Integer(1)]],[[Integer(2), Integer(3)], [Integer(2),␣
↪Integer(4)]]]).is_column_strict()
False
>>> TableauTuple([[[Integer(1)]],[[Integer(2), Integer(2)], [Integer(4),
↪Integer(5)]]]]).is_column_strict()
True
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(6),Integer(7)]],
↪[[Integer(4),Integer(8)], [Integer(6), Integer(9)]],[]]).is_column_strict()
True
```

is_row_strict()

Return `True` if the tableau `self` is row strict and `False` otherwise.

A tableau tuple is *row strict* if the entries in each row of each component are in increasing order, when read from left to right.

EXAMPLES:

```
sage: TableauTuple([[[5,7],[8]],[[1, 3], [2, 4]],[[6]]]).is_row_strict()
True
sage: TableauTuple([[[1, 2], [2, 4]],[[4,5,6],[7,8]]]).is_row_strict()
```

(continues on next page)

(continued from previous page)

```
True
sage: TableauTuple([[[1]],[[2, 3], [2, 4]]]).is_row_strict()
True
sage: TableauTuple([[[1]],[[2, 2], [4,5]]]).is_row_strict()
False
sage: TableauTuple([[[1,2],[6,7]],[[4,8], [6, 9]],[]]).is_row_strict()
True
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(5),Integer(7)],[Integer(8)]],[[Integer(1),␣
↪Integer(3)], [Integer(2), Integer(4)]],[[Integer(6)]]]).is_row_strict()
True
>>> TableauTuple([[[Integer(1), Integer(2)], [Integer(2), Integer(4)]],
↪[[Integer(4),Integer(5),Integer(6)],[Integer(7),Integer(8)]]]).is_row_
↪strict()
True
>>> TableauTuple([[[Integer(1)]],[[Integer(2), Integer(3)], [Integer(2),␣
↪Integer(4)]]]).is_row_strict()
True
>>> TableauTuple([[[Integer(1)]],[[Integer(2), Integer(2)], [Integer(4),
↪Integer(5)]]]).is_row_strict()
False
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(6),Integer(7)]],
↪[[Integer(4),Integer(8)], [Integer(6), Integer(9)]],[]]).is_row_strict()
True
```

is_standard()

Return `True` if the tableau `self` is a standard tableau and `False` otherwise.

A tableau tuple is *standard* if it is row standard, column standard and the entries in the tableaux are $1, 2, \ldots, n$, where n is the *size()* of the underlying partition tuple of `self`.

EXAMPLES:

```
sage: TableauTuple([[[5,7],[8]],[[1, 3], [2, 4]],[[6]]]).is_standard()
True
sage: TableauTuple([[[1, 2], [2, 4]],[[4,5,6],[7,8]]]).is_standard()
False
sage: TableauTuple([[[1]],[[2, 3], [2, 4]]]).is_standard()
False
sage: TableauTuple([[[1]],[[2, 2], [4,5]]]).is_row_strict()
False
sage: TableauTuple([[[1,2],[6,7]],[[4,8], [6, 9]],[]]).is_standard()
False
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(5),Integer(7)],[Integer(8)]],[[Integer(1),␣
↪Integer(3)], [Integer(2), Integer(4)]],[[Integer(6)]]]).is_standard()
True
>>> TableauTuple([[[Integer(1), Integer(2)], [Integer(2), Integer(4)]],
↪[[Integer(4),Integer(5),Integer(6)],[Integer(7),Integer(8)]]]).is_standard()
False
```

(continues on next page)

(continued from previous page)

```
>>> TableauTuple([[[Integer(1)]],[[Integer(2), Integer(3)], [Integer(2),␣
↪Integer(4)]]]).is_standard()
False
>>> TableauTuple([[[Integer(1)]],[[Integer(2), Integer(2)], [Integer(4),
↪Integer(5)]]]).is_row_strict()
False
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(6),Integer(7)]],
↪[[Integer(4),Integer(8)], [Integer(6), Integer(9)]],[]]).is_standard()
False
```

level()

Return the level of the tableau `self`.

This is just the number of components in the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[7,8,9]],[],[[1,2,3],[4,5],[6]]]).level()
3
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(7),Integer(8),Integer(9)]],[],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(4),Integer(5)],[Integer(6)]]]).level()
3
```

pp()

Pretty printing for the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([ [[1,2,3],[4,5]], [[1,2,3],[4,5]] ]).pp()
  1  2  3     1  2  3
  4  5        4  5
sage: TableauTuple([ [[1,2],[3],[4]],[],[[6,7,8],[10,11],[12],[13]]]).pp()
  1  2     -     6  7  8
  3             10 11
  4             12
                13
sage: t = TableauTuple([ [[1,2,3],[4,5],[6],[9]], [[1,2,3],[4,5,8]], [[11,12,
↪13],[14]] ])
sage: t.pp()
  1  2  3     1  2  3    11 12 13
  4  5        4  5  8    14
  6
  9
sage: TableauTuples.options(convention='french')
sage: t.pp()
  9
  6
  4  5        4  5  8    14
  1  2  3     1  2  3    11 12 13
sage: TableauTuples.options._reset()
```

```
>>> from sage.all import *
>>> TableauTuple([ [[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]], [[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5)]]␣
↪]).pp()
  1  2  3     1  2  3
  4  5        4  5
>>> TableauTuple([ [[Integer(1),Integer(2)],[Integer(3)],[Integer(4)]],[],
↪[[Integer(6),Integer(7),Integer(8)],[Integer(10),Integer(11)],[Integer(12)],
↪[Integer(13)]]]).pp()
  1  2     -     6  7  8
  3             10 11
  4             12
                13
>>> t = TableauTuple([ [[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)],[Integer(6)],[Integer(9)]], [[Integer(1),Integer(2),Integer(3)],
↪[Integer(4),Integer(5),Integer(8)]], [[Integer(11),Integer(12),Integer(13)],
↪[Integer(14)]] ])
>>> t.pp()
  1  2  3     1  2  3    11 12 13
  4  5        4  5  8    14
  6
  9
>>> TableauTuples.options(convention='french')
>>> t.pp()
  9
  6
  4  5        4  5  8    14
  1  2  3     1  2  3    11 12 13
>>> TableauTuples.options._reset()
```

reduced_column_word()

Return the lexicographically minimal reduced expression for the permutation that maps the `initial_column_tableau()` to `self`.

This reduced expression is a minimal length coset representative for the corresponding Young subgroup. In one line notation, the permutation is obtained by concatenating the rows of the tableau from top to bottom in each component, and then left to right along the components.

EXAMPLES:

```
sage: StandardTableauTuple([[[7,9],[8]],[[1,4,6],[2,5],[3]]]).reduced_column_
↪word()
[]
sage: StandardTableauTuple([[[7,9],[8]],[[1,3,6],[2,5],[4]]]).reduced_column_
↪word()
[3]
sage: StandardTableauTuple([[[6,9],[8]],[[1,3,7],[2,5],[4]]]).reduced_column_
↪word()
[3, 6]
sage: StandardTableauTuple([[[6,8],[9]],[[1,3,7],[2,5],[4]]]).reduced_column_
↪word()
[3, 6, 8]
sage: StandardTableauTuple([[[5,8],[9]],[[1,3,7],[2,6],[4]]]).reduced_column_
↪word()
```

(continues on next page)

(continued from previous page)

```
[3, 6, 5, 8]
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[[Integer(7),Integer(9)],[Integer(8)]],[[Integer(1),
↪Integer(4),Integer(6)],[Integer(2),Integer(5)],[Integer(3)]]]).reduced_
↪column_word()
[]
>>> StandardTableauTuple([[[Integer(7),Integer(9)],[Integer(8)]],[[Integer(1),
↪Integer(3),Integer(6)],[Integer(2),Integer(5)],[Integer(4)]]]).reduced_
↪column_word()
[3]
>>> StandardTableauTuple([[[Integer(6),Integer(9)],[Integer(8)]],[[Integer(1),
↪Integer(3),Integer(7)],[Integer(2),Integer(5)],[Integer(4)]]]).reduced_
↪column_word()
[3, 6]
>>> StandardTableauTuple([[[Integer(6),Integer(8)],[Integer(9)]],[[Integer(1),
↪Integer(3),Integer(7)],[Integer(2),Integer(5)],[Integer(4)]]]).reduced_
↪column_word()
[3, 6, 8]
>>> StandardTableauTuple([[[Integer(5),Integer(8)],[Integer(9)]],[[Integer(1),
↪Integer(3),Integer(7)],[Integer(2),Integer(6)],[Integer(4)]]]).reduced_
↪column_word()
[3, 6, 5, 8]
```

reduced_row_word()

Return the lexicographically minimal reduced expression for the permutation that maps the `initial_tableau()` to `self`.

This reduced expression is a minimal length coset representative for the corresponding Young subgroup. In one line notation, the permutation is obtained by concatenating the rows of the tableau from top to bottom in each component, and then left to right along the components.

EXAMPLES:

```
sage: StandardTableauTuple([[[1,2],[3]],[[4,5,6],[7,8],[9]]]).reduced_row_
↪word()
[]
sage: StandardTableauTuple([[[1,2],[3]],[[4,5,6],[7,9],[8]]]).reduced_row_
↪word()
[8]
sage: StandardTableauTuple([[[1,2],[3]],[[4,5,7],[6,9],[8]]]).reduced_row_
↪word()
[6, 8]
sage: StandardTableauTuple([[[1,2],[3]],[[4,5,8],[6,9],[7]]]).reduced_row_
↪word()
[6, 8, 7]
sage: StandardTableauTuple([[[1,2],[3]],[[4,5,9],[6,8],[7]]]).reduced_row_
↪word()
[6, 7, 8, 7]
sage: StandardTableauTuple([[[7,9],[8]],[[1,3,5],[2,6],[4]]]).reduced_row_
↪word()
[2, 3, 2, 1, 4, 3, 2, 5, 4, 3, 6, 5, 4, 3, 2, 7, 6, 5, 8, 7, 6, 5, 4]
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3)]],[[Integer(4),
↪Integer(5),Integer(6)],[Integer(7),Integer(8)],[Integer(9)]]]).reduced_row_
↪word()
[]
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3)]],[[Integer(4),
↪Integer(5),Integer(6)],[Integer(7),Integer(9)],[Integer(8)]]]).reduced_row_
↪word()
[8]
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3)]],[[Integer(4),
↪Integer(5),Integer(7)],[Integer(6),Integer(9)],[Integer(8)]]]).reduced_row_
↪word()
[6, 8]
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3)]],[[Integer(4),
↪Integer(5),Integer(8)],[Integer(6),Integer(9)],[Integer(7)]]]).reduced_row_
↪word()
[6, 8, 7]
>>> StandardTableauTuple([[[Integer(1),Integer(2)],[Integer(3)]],[[Integer(4),
↪Integer(5),Integer(9)],[Integer(6),Integer(8)],[Integer(7)]]]).reduced_row_
↪word()
[6, 7, 8, 7]
>>> StandardTableauTuple([[[Integer(7),Integer(9)],[Integer(8)]],[[Integer(1),
↪Integer(3),Integer(5)],[Integer(2),Integer(6)],[Integer(4)]]]).reduced_row_
↪word()
[2, 3, 2, 1, 4, 3, 2, 5, 4, 3, 6, 5, 4, 3, 2, 7, 6, 5, 8, 7, 6, 5, 4]
```

residue(*k*, *e*, *multicharge*)

Return the *residue* of the integer `k` in the tableau `self`.

The *residue* of k is $c - r + a_k$ in $\mathbf{Z}/e\mathbf{Z}$, where k appears in row r and column c of the tableau and the multicharge is $(a_1, a_2, \ldots, a_l)$.

The multicharge determines the dominant weight

$$\sum_{i=1}^{l} \Lambda_{a_i}$$

for the affine special linear group. In the combinatorics, it simply offsets the contents in each component so that the cell $(k, 0, 0)$ has content a_k.

INPUT:

- `k` – integer in $\{1, 2, \ldots, n\}$
- `e` – integer in $\{0, 2, 3, 4, 5, \ldots\}$
- `multicharge` – list of integers of length l

Here l is the `level()` and n is the `size()` of `self`.

OUTPUT: the residue of `k` in a standard tableau. That is,

EXAMPLES:

```
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue(1, 3,[0,0])
0
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue(1, 3,[0,1])
```

(continues on next page)

(continued from previous page)

```
1
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue(1, 3,[0,2])
2
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).residue(6, 3,[0,2])
Traceback (most recent call last):
...
ValueError: 6 must be contained in the tableaux
```

```
>>> from sage.all import *
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue(Integer(1), Integer(3),[Integer(0),Integer(0)])
0
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue(Integer(1), Integer(3),[Integer(0),Integer(1)])
1
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue(Integer(1), Integer(3),[Integer(0),Integer(2)])
2
>>> StandardTableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).residue(Integer(6), Integer(3),[Integer(0),Integer(2)])
Traceback (most recent call last):
...
ValueError: 6 must be contained in the tableaux
```

restrict(*m=None*)

Return the restriction of the standard tableau `self` to `m`.

The restriction is the subtableau of `self` whose entries are less than or equal to `m`.

By default, `m` is one less than the current size.

EXAMPLES:

```
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict()
([], [[1, 2], [3, 4]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(6)
([[5]], [[1, 2], [3, 4]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(5)
([[5]], [[1, 2], [3, 4]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(4)
([], [[1, 2], [3, 4]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3)
([], [[1, 2], [3]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(2)
([], [[1, 2]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(1)
([], [[1]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(0)
([], [])
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]]).restrict()
```

(continues on next page)

(continued from previous page)

```
([], [[1, 2], [3, 4]])
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(6))
([[5]], [[1, 2], [3, 4]])
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(5))
([[5]], [[1, 2], [3, 4]])
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(4))
([], [[1, 2], [3, 4]])
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(3))
([], [[1, 2], [3]])
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(2))
([], [[1, 2]])
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(1))
([], [[1]])
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(0))
([], [])
```

Where possible the restricted tableau belongs to the same category as the original tableaux:

```
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Tableau tuples
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Tableau tuples
sage: TableauTuples(level=2)([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Tableau tuples of level 2
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(3)).category()
Category of elements of Tableau tuples
>>> TableauTuple([[[Integer(5)]],[[Integer(1),Integer(2)],[Integer(3),
→Integer(4)]]]).restrict(Integer(3)).category()
Category of elements of Tableau tuples
>>> TableauTuples(level=Integer(2))([[[Integer(5)]],[[Integer(1),Integer(2)],
→[Integer(3),Integer(4)]]]).restrict(Integer(3)).category()
Category of elements of Tableau tuples of level 2
```

row_stabilizer()

Return the `PermutationGroup` corresponding to `self`. That is, return subgroup of the symmetric group of degree *size()* which is the row stabilizer of `self`.

EXAMPLES:

```
sage: # needs sage.groups
sage: t = TableauTuple([[[1,2,3],[4,5]],[[6,7]],[[8],[9]]])
sage: rs = t.row_stabilizer()
```

(continues on next page)

(continued from previous page)

```
sage: rs.order()
24
sage: PermutationGroupElement([(1,3,2),(4,5)]) in rs
True
sage: PermutationGroupElement([(1,4)]) in rs
False
sage: rs.one().domain()
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> t = TableauTuple([[[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]],[[Integer(6),Integer(7)]],[[Integer(8)],[Integer(9)]]])
>>> rs = t.row_stabilizer()
>>> rs.order()
24
>>> PermutationGroupElement([(Integer(1),Integer(3),Integer(2)),(Integer(4),
↪Integer(5))]) in rs
True
>>> PermutationGroupElement([(Integer(1),Integer(4))]) in rs
False
>>> rs.one().domain()
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

shape()

Return the *PartitionTuple* which is the shape of the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[7,8,9]],[],[[1,2,3],[4,5],[6]]]).shape()
([3], [], [3, 2, 1])
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(7),Integer(8),Integer(9)]],[],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(4),Integer(5)],[Integer(6)]]]).shape()
([3], [], [3, 2, 1])
```

size()

Return the size of the tableau tuple `self`.

This is just the number of boxes, or the size, of the underlying *PartitionTuple*.

EXAMPLES:

```
sage: TableauTuple([[[7,8,9]],[],[[1,2,3],[4,5],[6]]]).size()
9
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(7),Integer(8),Integer(9)]],[],[[Integer(1),
↪Integer(2),Integer(3)],[Integer(4),Integer(5)],[Integer(6)]]]).size()
9
```

symmetric_group_action_on_entries(*w*)

Return the action of a permutation `w` on `self`.

Consider a standard tableau tuple $T = (t^{(1)}, t^{(2)}, \ldots t^{(l)})$ of size n, then the action of $w \in S_n$ is defined by permuting the entries of T (recall they are $1, 2, \ldots, n$). In particular, suppose the entry at cell (k, i, j) is a, then the entry becomes $w(a)$. In general, the resulting tableau tuple wT may *not* be standard.

INPUT:

- `w` – a permutation

EXAMPLES:

```
sage: TableauTuple([[[1,2],[4]],[[3,5]]]).symmetric_group_action_on_entries(
↪Permutation(((4,5))) )
([[1, 2], [5]], [[3, 4]])
sage: TableauTuple([[[1,2],[4]],[[3,5]]]).symmetric_group_action_on_entries(
↪Permutation(((1,2))) )
([[2, 1], [4]], [[3, 5]])
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(4)]],[[Integer(3),
↪Integer(5)]]]).symmetric_group_action_on_entries( Permutation(((Integer(4),
↪Integer(5)))) )
([[1, 2], [5]], [[3, 4]])
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(4)]],[[Integer(3),
↪Integer(5)]]]).symmetric_group_action_on_entries( Permutation(((Integer(1),
↪Integer(2)))) )
([[2, 1], [4]], [[3, 5]])
```

to_list()

Return the list representation of the tableaux tuple `self`.

EXAMPLES:

```
sage: TableauTuple([ [[1,2,3],[4,5]], [[6,7],[8,9]] ]).to_list()
[[[1, 2, 3], [4, 5]], [[6, 7], [8, 9]]]
```

```
>>> from sage.all import *
>>> TableauTuple([ [[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5)]], [[Integer(6),Integer(7)],[Integer(8),Integer(9)]] ]).to_list()
[[[1, 2, 3], [4, 5]], [[6, 7], [8, 9]]]
```

to_permutation()

Return a permutation with the entries in the tableau tuple `self`.

The permutation is obtained from `self` by reading the entries of the tableau tuple in order from left to right along the rows, and then top to bottom, in each component and then left to right along the components.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).to_
↪permutation()
[12, 11, 9, 10, 8, 5, 6, 7, 3, 4, 1, 2]
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).to_permutation()
[12, 11, 9, 10, 8, 5, 6, 7, 3, 4, 1, 2]
```

to_word()

Return a word obtained from a row reading of the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).to_word_
↪by_row()
word: 12,11,9,10,8,5,6,7,3,4,1,2
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).to_word_by_row()
word: 12,11,9,10,8,5,6,7,3,4,1,2
```

to_word_by_column()

Return the word obtained from a column reading of the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).to_word_
↪by_column()
word: 12,11,9,10,8,5,6,7,3,1,4,2
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).to_word_by_column()
word: 12,11,9,10,8,5,6,7,3,1,4,2
```

to_word_by_row()

Return a word obtained from a row reading of the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).to_word_
↪by_row()
word: 12,11,9,10,8,5,6,7,3,4,1,2
```

```
>>> from sage.all import *
>>> TableauTuple([[[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪[[Integer(5),Integer(6),Integer(7)],[Integer(8)]],[[Integer(9),Integer(10)],
↪[Integer(11)],[Integer(12)]]]).to_word_by_row()
word: 12,11,9,10,8,5,6,7,3,4,1,2
```

up(*n=None*)

An iterator for all the *TableauTuple* that can be obtained from `self` by adding a cell with the label `n`. If `n` is not specified then a cell with label `n` will be added to the tableau tuple, where `n-1` is the size of the tableau tuple before any cells are added.

EXAMPLES:

```
sage: list(TableauTuple([[[1,2]],[[3]]]).up())
[([[1, 2, 4]], [[3]]),
  ([[1, 2], [4]], [[3]]),
  ([[1, 2]], [[3, 4]]),
  ([[1, 2]], [[3], [4]])]
```

```
>>> from sage.all import *
>>> list(TableauTuple([[[Integer(1),Integer(2)]],[[Integer(3)]]]).up())
[([[1, 2, 4]], [[3]]),
  ([[1, 2], [4]], [[3]]),
  ([[1, 2]], [[3, 4]]),
  ([[1, 2]], [[3], [4]])]
```

class sage.combinat.tableau_tuple.**TableauTuples**

Bases: `UniqueRepresentation`, `Parent`

A factory class for the various classes of tableau tuples.

INPUT:

There are three optional arguments:

- `shape` – determines a *`PartitionTuple`* which gives the shape of the *`TableauTuples`*
- `level` – the level of the tableau tuples (positive integer)
- `size` – the size of the tableau tuples (nonnegative integer)

It is not necessary to use the keywords. If they are not specified then the first integer argument specifies the `level` and the second the `size` of the tableaux.

OUTPUT: the corresponding class of tableau tuples

The entries of a tableau can be any sage object. Because of this, no enumeration of the set of *`TableauTuples`* is possible.

EXAMPLES:

```
sage: T3 = TableauTuples(3); T3
Tableau tuples of level 3
sage: [['a','b']] in TableauTuples()
True
sage: [['a','b']] in TableauTuples(level=3)
False
sage: t = TableauTuples(level=3)([[],[[1,1,1]],[]]); t
([], [[1, 1, 1]], [])
sage: t in T3
True
sage: t in TableauTuples()
True
sage: t in TableauTuples(size=3)
True
sage: t in TableauTuples(size=4)
False
```

(continues on next page)

(continued from previous page)

```
sage: t in StandardTableauTuples()
False
sage: t.parent()
Tableau tuples of level 3
sage: t.category()
Category of elements of Tableau tuples of level 3
```

```
>>> from sage.all import *
>>> T3 = TableauTuples(Integer(3)); T3
Tableau tuples of level 3
>>> [['a','b']] in TableauTuples()
True
>>> [['a','b']] in TableauTuples(level=Integer(3))
False
>>> t = TableauTuples(level=Integer(3))([[],[[Integer(1),Integer(1),Integer(1)]],
↪[]]); t
([], [[1, 1, 1]], [])
>>> t in T3
True
>>> t in TableauTuples()
True
>>> t in TableauTuples(size=Integer(3))
True
>>> t in TableauTuples(size=Integer(4))
False
>>> t in StandardTableauTuples()
False
>>> t.parent()
Tableau tuples of level 3
>>> t.category()
Category of elements of Tableau tuples of level 3
```

See also

- *Tableau*
- *StandardTableau*
- *StandardTableauTuples*

Element

alias of *TableauTuple*

level()

Return the `level` of a tableau tuple in `self`, or `None` if different tableau tuples in `self` can have different sizes. The `level` of a tableau tuple is just the level of the underlying *PartitionTuple*.

EXAMPLES:

```
sage: TableauTuples().level() is None
True
```

(continues on next page)

(continued from previous page)

```
sage: TableauTuples(7).level()
7
```

```
>>> from sage.all import *
>>> TableauTuples().level() is None
True
>>> TableauTuples(Integer(7)).level()
7
```

level_one_parent_class

alias of *Tableaux_all*

list()

If the set of tableau tuples self is finite then this function returns the list of these tableau tuples. If the class is infinite an error is returned.

EXAMPLES:

```
sage: StandardTableauTuples([[2,1],[2]]).list()
[([[1, 2], [3]], [[4, 5]]),
 ([[1, 3], [2]], [[4, 5]]),
 ([[1, 2], [4]], [[3, 5]]),
 ([[1, 3], [4]], [[2, 5]]),
 ([[2, 3], [4]], [[1, 5]]),
 ([[1, 4], [2]], [[3, 5]]),
 ([[1, 4], [3]], [[2, 5]]),
 ([[2, 4], [3]], [[1, 5]]),
 ([[1, 2], [5]], [[3, 4]]),
 ([[1, 3], [5]], [[2, 4]]),
 ([[2, 3], [5]], [[1, 4]]),
 ([[1, 4], [5]], [[2, 3]]),
 ([[2, 4], [5]], [[1, 3]]),
 ([[3, 4], [5]], [[1, 2]]),
 ([[1, 5], [2]], [[3, 4]]),
 ([[1, 5], [3]], [[2, 4]]),
 ([[2, 5], [3]], [[1, 4]]),
 ([[1, 5], [4]], [[2, 3]]),
 ([[2, 5], [4]], [[1, 3]]),
 ([[3, 5], [4]], [[1, 2]])]
```

```
>>> from sage.all import *
>>> StandardTableauTuples([[Integer(2),Integer(1)],[Integer(2)]]).list()
[([[1, 2], [3]], [[4, 5]]),
 ([[1, 3], [2]], [[4, 5]]),
 ([[1, 2], [4]], [[3, 5]]),
 ([[1, 3], [4]], [[2, 5]]),
 ([[2, 3], [4]], [[1, 5]]),
 ([[1, 4], [2]], [[3, 5]]),
 ([[1, 4], [3]], [[2, 5]]),
 ([[2, 4], [3]], [[1, 5]]),
 ([[1, 2], [5]], [[3, 4]]),
 ([[1, 3], [5]], [[2, 4]]),
```

(continues on next page)

(continued from previous page)

```
([[2, 3], [5]], [[1, 4]]),
([[1, 4], [5]], [[2, 3]]),
([[2, 4], [5]], [[1, 3]]),
([[3, 4], [5]], [[1, 2]]),
([[1, 5], [2]], [[3, 4]]),
([[1, 5], [3]], [[2, 4]]),
([[2, 5], [3]], [[1, 4]]),
([[1, 5], [4]], [[2, 3]]),
([[2, 5], [4]], [[1, 3]]),
([[3, 5], [4]], [[1, 2]])]
```

options = Current options for Tableaux - ascii_art: repr - convention: English - display: list - latex: diagram

size()

Return the size of a tableau tuple in self, or None if different tableau tuples in self can have different sizes. The size of a tableau tuple is just the size of the underlying *PartitionTuple*.

EXAMPLES:

```
sage: TableauTuples(size=14).size()
14
```

```
>>> from sage.all import *
>>> TableauTuples(size=Integer(14)).size()
14
```

class sage.combinat.tableau_tuple.**TableauTuples_all**

Bases: *TableauTuples*

The parent class of all *TableauTuples*, with arbitrary level and size.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: TableauTuples().an_element()
([[1]], [[2]], [[3]], [[4]], [[5]], [[6]], [[7]])
```

```
>>> from sage.all import *
>>> TableauTuples().an_element()
([[1]], [[2]], [[3]], [[4]], [[5]], [[6]], [[7]])
```

class sage.combinat.tableau_tuple.**TableauTuples_level**(*level*)

Bases: *TableauTuples*

Class of all *TableauTuples* with a fixed level and arbitrary size.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: TableauTuples(3).an_element()
([], [], [])
sage: TableauTuples(5).an_element()
([], [], [], [], [])
sage: T = TableauTuples(0)
Traceback (most recent call last):
...
ValueError: the level must be a positive integer
```

```
>>> from sage.all import *
>>> TableauTuples(Integer(3)).an_element()
([], [], [])
>>> TableauTuples(Integer(5)).an_element()
([], [], [], [], [])
>>> T = TableauTuples(Integer(0))
Traceback (most recent call last):
...
ValueError: the level must be a positive integer
```

class sage.combinat.tableau_tuple.**TableauTuples_level_size**(*level*, *size*)

Bases: *TableauTuples*

Class of all *TableauTuples* with a fixed `level` and a fixed `size`.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: TableauTuples(3,0).an_element()
([], [], [])
sage: TableauTuples(3,1).an_element()
([[1]], [], [])
sage: TableauTuples(3,2).an_element()
([[1, 2]], [], [])
```

```
>>> from sage.all import *
>>> TableauTuples(Integer(3),Integer(0)).an_element()
([], [], [])
>>> TableauTuples(Integer(3),Integer(1)).an_element()
([[1]], [], [])
>>> TableauTuples(Integer(3),Integer(2)).an_element()
([[1, 2]], [], [])
```

class sage.combinat.tableau_tuple.**TableauTuples_size**(*size*)

Bases: *TableauTuples*

Class of all *TableauTuples* with a arbitrary `level` and fixed `size`.

an_element()

Return a particular element of the class.

EXAMPLES:

```
sage: TableauTuples(size=3).an_element()
([], [[1, 2, 3]], [])
sage: TableauTuples(size=0).an_element()
([], [], [])
```

```
>>> from sage.all import *
>>> TableauTuples(size=Integer(3)).an_element()
([], [[1, 2, 3]], [])
>>> TableauTuples(size=Integer(0)).an_element()
([], [], [])
```

5.1.348 Generalized Tamari lattices

These lattices depend on three parameters a, b and m, where a and b are positive integers and m is a nonnegative integer.

The elements are `Dyck paths` in the $(a \times b)$-rectangle. The order relation depends on m.

To use the provided functionality, you should import Generalized Tamari lattices by typing:

```
sage: from sage.combinat.tamari_lattices import GeneralizedTamariLattice
```

```
>>> from sage.all import *
>>> from sage.combinat.tamari_lattices import GeneralizedTamariLattice
```

Then,

```
sage: GeneralizedTamariLattice(3,2)
Finite lattice containing 2 elements
sage: GeneralizedTamariLattice(4,3)
Finite lattice containing 5 elements
```

```
>>> from sage.all import *
>>> GeneralizedTamariLattice(Integer(3),Integer(2))
Finite lattice containing 2 elements
>>> GeneralizedTamariLattice(Integer(4),Integer(3))
Finite lattice containing 5 elements
```

The classical **Tamari lattices** are special cases of this construction and are also available directly using the catalogue of posets, as follows:

```
sage: posets.TamariLattice(3)
Finite lattice containing 5 elements
```

```
>>> from sage.all import *
>>> posets.TamariLattice(Integer(3))
Finite lattice containing 5 elements
```

See also

For more detailed information see `TamariLattice()`, `GeneralizedTamariLattice()`.

sage.combinat.tamari_lattices.**DexterSemilattice**(*n*)

Return the n-th Dexter meet-semilattice.

INPUT:

- n – nonnegative integer (the index)

OUTPUT: a finite meet-semilattice

The elements of the semilattice are Dyck paths in the $(n + 1 \times n)$-rectangle.

EXAMPLES:

```
sage: posets.DexterSemilattice(3)
Finite meet-semilattice containing 5 elements

sage: P = posets.DexterSemilattice(4); P
Finite meet-semilattice containing 14 elements
sage: len(P.maximal_chains())
15
sage: len(P.maximal_elements())
4
sage: P.chain_polynomial()
q^5 + 19*q^4 + 47*q^3 + 42*q^2 + 14*q + 1
```

```
>>> from sage.all import *
>>> posets.DexterSemilattice(Integer(3))
Finite meet-semilattice containing 5 elements

>>> P = posets.DexterSemilattice(Integer(4)); P
Finite meet-semilattice containing 14 elements
>>> len(P.maximal_chains())
15
>>> len(P.maximal_elements())
4
>>> P.chain_polynomial()
q^5 + 19*q^4 + 47*q^3 + 42*q^2 + 14*q + 1
```

REFERENCES:

- [Cha18]

sage.combinat.tamari_lattices.**GeneralizedTamariLattice**(*a*, *b*, *m=1*)

Return the (a, b)-Tamari lattice of parameter m.

INPUT:

- a, b – integers with $a \geq b$
- m – a nonnegative rational number such that $a \geq bm$

OUTPUT:

- a finite lattice (special case of the alt ν-Tamari lattices in [CC2023])

The elements of the lattice are Dyck paths in the $(a \times b)$-rectangle.

The parameter m (slope) is used only to define the covering relations. When the slope m is 0, two paths are comparable if and only if one is always above the other.

The usual Tamari lattice of index b is the special case $a = b + 1$ and $m = 1$.

Other special cases give the m-Tamari lattices studied in [BMFPR2011], or the rational Tamari lattices when a and b are coprime and m = a/b (see [PRV2017]).

EXAMPLES:

```
sage: from sage.combinat.tamari_lattices import GeneralizedTamariLattice
sage: GeneralizedTamariLattice(3,2)
Finite lattice containing 2 elements
sage: GeneralizedTamariLattice(4,3)
Finite lattice containing 5 elements
sage: GeneralizedTamariLattice(7,5,2)
Traceback (most recent call last):
...
ValueError: the condition a>=b*m does not hold
sage: P = GeneralizedTamariLattice(5,3); P
Finite lattice containing 7 elements
sage: P = GeneralizedTamariLattice(5, 3, m=5/3); P
Finite lattice containing 7 elements
```

```
>>> from sage.all import *
>>> from sage.combinat.tamari_lattices import GeneralizedTamariLattice
>>> GeneralizedTamariLattice(Integer(3),Integer(2))
Finite lattice containing 2 elements
>>> GeneralizedTamariLattice(Integer(4),Integer(3))
Finite lattice containing 5 elements
>>> GeneralizedTamariLattice(Integer(7),Integer(5),Integer(2))
Traceback (most recent call last):
...
ValueError: the condition a>=b*m does not hold
>>> P = GeneralizedTamariLattice(Integer(5),Integer(3)); P
Finite lattice containing 7 elements
>>> P = GeneralizedTamariLattice(Integer(5), Integer(3), m=Integer(5)/Integer(3));
↪ P
Finite lattice containing 7 elements
```

REFERENCES:

- [BMFPR2011]
- [PRV2017]
- [CC2023]

`sage.combinat.tamari_lattices.`**`TamariLattice`**`(n, m=1)`

Return the n-th Tamari lattice.

Using the slope parameter m, one can also get the m-Tamari lattices.

INPUT:

- `n` – nonnegative integer (the index)
- `m` – nonnegative integer (the slope, default: 1)

OUTPUT: a finite lattice

In the usual case, the elements of the lattice are `Dyck paths` in the $(n + 1 \times n)$-rectangle. For a general slope m, the elements are Dyck paths in the $(mn + 1 \times n)$-rectangle.

See Tamari lattice for mathematical background.

EXAMPLES:

```
sage: posets.TamariLattice(3)
Finite lattice containing 5 elements

sage: posets.TamariLattice(3, 2)
Finite lattice containing 12 elements
```

```
>>> from sage.all import *
>>> posets.TamariLattice(Integer(3))
Finite lattice containing 5 elements

>>> posets.TamariLattice(Integer(3), Integer(2))
Finite lattice containing 12 elements
```

REFERENCES:

- [BMFPR2011]

sage.combinat.tamari_lattices.**paths_in_triangle**(*i*, *j*, *a*, *b*)

Return all Dyck paths from $(0, 0)$ to (i, j) in the $(a \times b)$-rectangle.

This means that at each step of the path, one has $ay \geq bx$.

A path is represented by a sequence of 0 and 1, where 0 is an horizontal step $(1, 0)$ and 1 is a vertical step $(0, 1)$.

INPUT:

- a, b – integers with $a \geq b$
- i, j – nonnegative integers with $1 \geq \frac{j}{b} \geq \frac{i}{a} \geq 0$

OUTPUT: list of paths

EXAMPLES:

```
sage: from sage.combinat.tamari_lattices import paths_in_triangle
sage: paths_in_triangle(2,2,2,2)
[(1, 0, 1, 0), (1, 1, 0, 0)]
sage: paths_in_triangle(2,3,4,4)
[(1, 0, 1, 0, 1), (1, 1, 0, 0, 1), (1, 0, 1, 1, 0),
(1, 1, 0, 1, 0), (1, 1, 1, 0, 0)]
sage: paths_in_triangle(2,1,4,4)
Traceback (most recent call last):
...
ValueError: the endpoint is not valid
sage: paths_in_triangle(3,2,5,3)
[(1, 0, 1, 0, 0), (1, 1, 0, 0, 0)]
```

```
>>> from sage.all import *
>>> from sage.combinat.tamari_lattices import paths_in_triangle
>>> paths_in_triangle(Integer(2),Integer(2),Integer(2),Integer(2))
[(1, 0, 1, 0), (1, 1, 0, 0)]
>>> paths_in_triangle(Integer(2),Integer(3),Integer(4),Integer(4))
[(1, 0, 1, 0, 1), (1, 1, 0, 0, 1), (1, 0, 1, 1, 0),
(1, 1, 0, 1, 0), (1, 1, 1, 0, 0)]
>>> paths_in_triangle(Integer(2),Integer(1),Integer(4),Integer(4))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: the endpoint is not valid
>>> paths_in_triangle(Integer(3),Integer(2),Integer(5),Integer(3))
[(1, 0, 1, 0, 0), (1, 1, 0, 0, 0)]
```

sage.combinat.tamari_lattices.**swap**(*p*, *i*, *m=1*)

Perform a covering move in the (a, b)-Tamari lattice of slope parameter m.

The letter at position i in p must be a 0, followed by at least one 1.

INPUT:

- p – a Dyck path in the $(a \times b)$-rectangle
- i – integer between 0 and $a + b - 1$

OUTPUT: a Dyck path in the $(a \times b)$-rectangle

EXAMPLES:

```
sage: from sage.combinat.tamari_lattices import swap
sage: swap((1,0,1,0,0),1)
(1, 1, 0, 0, 0)
sage: swap((1,1,0,0,1,1,0,0,0),3)
(1, 1, 0, 1, 1, 0, 0, 0, 0)
sage: swap((1,0,1,0,1,0,0,0), 1, 1)
(1, 1, 0, 0, 1, 0, 0, 0)
sage: swap((1,0,1,0,1,0,0,0), 1, 5/3)
(1, 1, 0, 1, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.tamari_lattices import swap
>>> swap((Integer(1),Integer(0),Integer(1),Integer(0),Integer(0)),Integer(1))
(1, 1, 0, 0, 0)
>>> swap((Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(0),Integer(0)),Integer(3))
(1, 1, 0, 1, 1, 0, 0, 0, 0)
>>> swap((Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0)), Integer(1), Integer(1))
(1, 1, 0, 0, 1, 0, 0, 0)
>>> swap((Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0)), Integer(1), Integer(5)/Integer(3))
(1, 1, 0, 1, 0, 0, 0, 0)
```

sage.combinat.tamari_lattices.**swap_dexter**(*p*, *i*)

Perform covering moves in the (a, b)-Dexter posets.

The letter at position i in p must be a 0, followed by at least one 1.

INPUT:

- p – a Dyck path in the $(a \times b)$-rectangle
- i – integer between 0 and $a + b - 1$

OUTPUT:

- a list of Dyck paths in the $(a \times b)$-rectangle

EXAMPLES:

```
sage: from sage.combinat.tamari_lattices import swap_dexter
sage: swap_dexter((1,0,1,0,0),1)
[(1, 1, 0, 0, 0)]
sage: swap_dexter((1,1,0,0,1,1,0,0,0),3)
[(1, 1, 0, 1, 1, 0, 0, 0, 0), (1, 1, 1, 1, 0, 0, 0, 0, 0)]
sage: swap_dexter((1,1,0,1,0,0,0),2)
[]
```

```
>>> from sage.all import *
>>> from sage.combinat.tamari_lattices import swap_dexter
>>> swap_dexter((Integer(1),Integer(0),Integer(1),Integer(0),Integer(0)),
↪Integer(1))
[(1, 1, 0, 0, 0)]
>>> swap_dexter((Integer(1),Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0),Integer(0)),Integer(3))
[(1, 1, 0, 1, 1, 0, 0, 0, 0), (1, 1, 1, 1, 0, 0, 0, 0, 0)]
>>> swap_dexter((Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0)),Integer(2))
[]
```

5.1.349 Tiling Solver

Tiling a n-dimensional polyomino with n-dimensional polyominoes.

This module defines two classes:

- `sage.combinat.tiling.Polyomino` class, to represent polyominoes in arbitrary dimension. The goal of this class is to return all the rotated, reflected and/or translated copies of a polyomino that are contained in a certain box.
- `sage.combinat.tiling.TilingSolver` class, to solve the problem of tiling a n-dimensional polyomino with a set of n-dimensional polyominoes. One can specify if rotations and reflections are allowed or not and if pieces can be reused or not. This class convert the tiling data into rows of a matrix that are passed to the DLX solver. It also allows to compute the number of solutions.

This uses dancing links code which is in Sage. Dancing links were originally introduced by Donald Knuth in 2000 [Knuth1]. Knuth used dancing links to solve tilings of a region by 2d pentaminoes. Here we extend the method to any dimension.

In particular, the `sage.games.quantumino` module is based on the Tiling Solver and allows to solve the 3d Quantumino puzzle.

AUTHOR:

- Sébastien Labbé, June 2011, initial version
- Sébastien Labbé, July 2015, count solutions up to rotations
- Sébastien Labbé, April 2017, tiling a polyomino, not only a rectangular box

EXAMPLES:

2d Easy Example

Here is a 2d example. Let us try to fill the 3×2 rectangle with a 1×2 rectangle and a 2×2 square. Obviously, there are two solutions:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0), (0,1)])
sage: q = Polyomino([(0,0), (0,1), (1,0), (1,1)])
sage: T = TilingSolver([p,q], box=[3,2])
sage: it = T.solve()
sage: next(it)
[Polyomino: [(0, 0), (0, 1), (1, 0), (1, 1)], Color: gray,
 Polyomino: [(2, 0), (2, 1)], Color: gray]
sage: next(it)
[Polyomino: [(1, 0), (1, 1), (2, 0), (2, 1)], Color: gray,
 Polyomino: [(0, 0), (0, 1)], Color: gray]
sage: next(it)
Traceback (most recent call last):
...
StopIteration
sage: T.number_of_solutions()
2
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1))])
>>> q = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1)), (Integer(1),
↪Integer(0)), (Integer(1),Integer(1))])
>>> T = TilingSolver([p,q], box=[Integer(3),Integer(2)])
>>> it = T.solve()
>>> next(it)
[Polyomino: [(0, 0), (0, 1), (1, 0), (1, 1)], Color: gray,
 Polyomino: [(2, 0), (2, 1)], Color: gray]
>>> next(it)
[Polyomino: [(1, 0), (1, 1), (2, 0), (2, 1)], Color: gray,
 Polyomino: [(0, 0), (0, 1)], Color: gray]
>>> next(it)
Traceback (most recent call last):
...
StopIteration
>>> T.number_of_solutions()
2
```

Scott's pentamino problem

As mentioned in the introduction of [Knuth1], Scott's pentamino problem consists in tiling a chessboard leaving the center four squares vacant with the 12 distinct pentaminoes.

The 12 pentaminoes:

```
sage: from sage.combinat.tiling import Polyomino
sage: I = Polyomino([(0,0),(1,0),(2,0),(3,0),(4,0)], color='brown')
sage: N = Polyomino([(1,0),(1,1),(1,2),(0,2),(0,3)], color='yellow')
sage: L = Polyomino([(0,0),(1,0),(0,1),(0,2),(0,3)], color='magenta')
```

(continues on next page)

(continued from previous page)

```
sage: U = Polyomino([(0,0),(1,0),(0,1),(0,2),(1,2)], color='violet')
sage: X = Polyomino([(1,0),(0,1),(1,1),(1,2),(2,1)], color='pink')
sage: W = Polyomino([(2,0),(2,1),(1,1),(1,2),(0,2)], color='green')
sage: P = Polyomino([(1,0),(2,0),(0,1),(1,1),(2,1)], color='orange')
sage: F = Polyomino([(1,0),(1,1),(0,1),(2,1),(2,2)], color='gray')
sage: Z = Polyomino([(0,0),(1,0),(1,1),(1,2),(2,2)], color='yellow')
sage: T = Polyomino([(0,0),(0,1),(1,1),(2,1),(0,2)], color='red')
sage: Y = Polyomino([(0,0),(1,0),(2,0),(3,0),(2,1)], color='green')
sage: V = Polyomino([(0,0),(0,1),(0,2),(1,0),(2,0)], color='blue')
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> I = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),(Integer(2),
↪Integer(0)),(Integer(3),Integer(0)),(Integer(4),Integer(0))], color='brown')
>>> N = Polyomino([(Integer(1),Integer(0)),(Integer(1),Integer(1)),(Integer(1),
↪Integer(2)),(Integer(0),Integer(2)),(Integer(0),Integer(3))], color='yellow')
>>> L = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),(Integer(0),
↪Integer(1)),(Integer(0),Integer(2)),(Integer(0),Integer(3))], color='magenta')
>>> U = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),(Integer(0),
↪Integer(1)),(Integer(0),Integer(2)),(Integer(1),Integer(2))], color='violet')
>>> X = Polyomino([(Integer(1),Integer(0)),(Integer(0),Integer(1)),(Integer(1),
↪Integer(1)),(Integer(1),Integer(2)),(Integer(2),Integer(1))], color='pink')
>>> W = Polyomino([(Integer(2),Integer(0)),(Integer(2),Integer(1)),(Integer(1),
↪Integer(1)),(Integer(1),Integer(2)),(Integer(0),Integer(2))], color='green')
>>> P = Polyomino([(Integer(1),Integer(0)),(Integer(2),Integer(0)),(Integer(0),
↪Integer(1)),(Integer(1),Integer(1)),(Integer(2),Integer(1))], color='orange')
>>> F = Polyomino([(Integer(1),Integer(0)),(Integer(1),Integer(1)),(Integer(0),
↪Integer(1)),(Integer(2),Integer(1)),(Integer(2),Integer(2))], color='gray')
>>> Z = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),(Integer(1),
↪Integer(1)),(Integer(1),Integer(2)),(Integer(2),Integer(2))], color='yellow')
>>> T = Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(1),
↪Integer(1)),(Integer(2),Integer(1)),(Integer(0),Integer(2))], color='red')
>>> Y = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),(Integer(2),
↪Integer(0)),(Integer(3),Integer(0)),(Integer(2),Integer(1))], color='green')
>>> V = Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(1),Integer(0)),(Integer(2),Integer(0))], color='blue')
```

A 8×8 chessboard leaving the center four squares vacant:

```
sage: import itertools
sage: s = set(itertools.product(range(8), repeat=2))
sage: s.difference_update([(3,3), (3,4), (4,3), (4,4)])
sage: chessboard = Polyomino(s)
sage: len(chessboard)
60
```

```
>>> from sage.all import *
>>> import itertools
>>> s = set(itertools.product(range(Integer(8)), repeat=Integer(2)))
>>> s.difference_update([(Integer(3),Integer(3)), (Integer(3),Integer(4)),␣
↪(Integer(4),Integer(3)), (Integer(4),Integer(4))])
>>> chessboard = Polyomino(s)
```

(continues on next page)

(continued from previous page)

```
>>> len(chessboard)
60
```

This problem is represented by a matrix made of 1568 rows and 72 columns. It has 65 different solutions up to isometries:

```
sage: from sage.combinat.tiling import TilingSolver
sage: T = TilingSolver([I,N,L,U,X,W,P,F,Z,T,Y,V], box=chessboard, reflection=True)
sage: T
Tiling solver of 12 pieces into a box of size 60
Rotation allowed: True
Reflection allowed: True
Reusing pieces allowed: False
sage: len(T.rows())                 # long time
1568
sage: T.number_of_solutions()       # long time
520
sage: 520 / 8
65
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver
>>> T = TilingSolver([I,N,L,U,X,W,P,F,Z,T,Y,V], box=chessboard, reflection=True)
>>> T
Tiling solver of 12 pieces into a box of size 60
Rotation allowed: True
Reflection allowed: True
Reusing pieces allowed: False
>>> len(T.rows())                 # long time
1568
>>> T.number_of_solutions()       # long time
520
>>> Integer(520) / Integer(8)
65
```

Showing one solution:

```
sage: solution = next(T.solve())                                    # long time
sage: G = sum([piece.show2d() for piece in solution], Graphics())   # long time,␣
↪needs sage.plot
sage: G.show(aspect_ratio=1, axes=False)                            # long time,␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> solution = next(T.solve())                                    # long time
>>> G = sum([piece.show2d() for piece in solution], Graphics())   # long time, needs␣
↪sage.plot
>>> G.show(aspect_ratio=Integer(1), axes=False)                            # long␣
↪time, needs sage.plot
```

1d Easy Example

Here is an easy one dimensional example where we try to tile a stick of length 6 with three sticks of length 1, 2 and 3. There are six solutions:

```
sage: p = Polyomino([[0]])
sage: q = Polyomino([[0],[1]])
sage: r = Polyomino([[0],[1],[2]])
sage: T = TilingSolver([p,q,r], box=[6])
sage: len(T.rows())
15
sage: it = T.solve()
sage: next(it)
[Polyomino: [(0)], Color: gray,
 Polyomino: [(1), (2)], Color: gray,
 Polyomino: [(3), (4), (5)], Color: gray]
sage: next(it)
[Polyomino: [(0)], Color: gray,
 Polyomino: [(1), (2), (3)], Color: gray,
 Polyomino: [(4), (5)], Color: gray]
sage: T.number_of_solutions()
6
```

```
>>> from sage.all import *
>>> p = Polyomino([[Integer(0)]])
>>> q = Polyomino([[Integer(0)],[Integer(1)]])
>>> r = Polyomino([[Integer(0)],[Integer(1)],[Integer(2)]])
>>> T = TilingSolver([p,q,r], box=[Integer(6)])
>>> len(T.rows())
15
>>> it = T.solve()
>>> next(it)
[Polyomino: [(0)], Color: gray,
 Polyomino: [(1), (2)], Color: gray,
 Polyomino: [(3), (4), (5)], Color: gray]
>>> next(it)
[Polyomino: [(0)], Color: gray,
 Polyomino: [(1), (2), (3)], Color: gray,
 Polyomino: [(4), (5)], Color: gray]
>>> T.number_of_solutions()
6
```

2d Puzzle allowing reflections

The following is a puzzle owned by Florent Hivert:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: L = []
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3)], 'yellow'))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2)], "black"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,3)], "gray"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,3)],"cyan"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,1)],"red"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,1),(1,2)],"blue"))
```

(continues on next page)

(continued from previous page)

```
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,1),(1,3)],"green"))
sage: L.append(Polyomino([(0,1),(0,2),(0,3),(1,0),(1,1),(1,3)],"magenta"))
sage: L.append(Polyomino([(0,1),(0,2),(0,3),(1,0),(1,1),(1,2)],"orange"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)],"pink"))
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino, TilingSolver
>>> L = []
>>> L.append(Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(0),Integer(3)),(Integer(1),Integer(0)),(Integer(1),Integer(1)),
↪(Integer(1),Integer(2)),(Integer(1),Integer(3))], 'yellow'))
>>> L.append(Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(0),Integer(3)),(Integer(1),Integer(0)),(Integer(1),Integer(1)),
↪(Integer(1),Integer(2))], "black"))
>>> L.append(Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(0),Integer(3)),(Integer(1),Integer(0)),(Integer(1),Integer(1)),
↪(Integer(1),Integer(3))], "gray"))
>>> L.append(Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(0),Integer(3)),(Integer(1),Integer(0)),(Integer(1),
↪Integer(3))],"cyan"))
>>> L.append(Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(0),Integer(3)),(Integer(1),Integer(0)),(Integer(1),
↪Integer(1))],"red"))
>>> L.append(Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(0),Integer(3)),(Integer(1),Integer(1)),(Integer(1),
↪Integer(2))],"blue"))
>>> L.append(Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(0),Integer(3)),(Integer(1),Integer(1)),(Integer(1),
↪Integer(3))],"green"))
>>> L.append(Polyomino([(Integer(0),Integer(1)),(Integer(0),Integer(2)),(Integer(0),
↪Integer(3)),(Integer(1),Integer(0)),(Integer(1),Integer(1)),(Integer(1),
↪Integer(3))],"magenta"))
>>> L.append(Polyomino([(Integer(0),Integer(1)),(Integer(0),Integer(2)),(Integer(0),
↪Integer(3)),(Integer(1),Integer(0)),(Integer(1),Integer(1)),(Integer(1),
↪Integer(2))],"orange"))
>>> L.append(Polyomino([(Integer(0),Integer(0)),(Integer(0),Integer(1)),(Integer(0),
↪Integer(2)),(Integer(1),Integer(0)),(Integer(1),Integer(1)),(Integer(1),
↪Integer(2))],"pink"))
```

By default, rotations are allowed and reflections are not. In this case, there are no solutions for tiling a 8×8 rectangular box:

```
sage: T = TilingSolver(L, box=(8,8))
sage: T.number_of_solutions()                              # long time (2.5s)
0
```

```
>>> from sage.all import *
>>> T = TilingSolver(L, box=(Integer(8),Integer(8)))
>>> T.number_of_solutions()                              # long time (2.5s)
0
```

If reflections are allowed, there are solutions. Solve the puzzle and show one solution:

```
sage: T = TilingSolver(L, box=(8,8), reflection=True)
sage: solution = next(T.solve())                              # long time (7s)
sage: G = sum([piece.show2d() for piece in solution], Graphics())   # long time (<1s),
↪ needs sage.plot
sage: G.show(aspect_ratio=1, axes=False)                      # long time (2s),␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> T = TilingSolver(L, box=(Integer(8),Integer(8)), reflection=True)
>>> solution = next(T.solve())                                # long time (7s)
>>> G = sum([piece.show2d() for piece in solution], Graphics())    # long time (<1s),␣
↪needs sage.plot
>>> G.show(aspect_ratio=Integer(1), axes=False)                          # long␣
↪time (2s), needs sage.plot
```

Compute the number of solutions:

```
sage: T.number_of_solutions()                                 # long time (2.6s)
328
```

```
>>> from sage.all import *
>>> T.number_of_solutions()                                   # long time (2.6s)
328
```

Create a animation of all the solutions:

```
sage: a = T.animate()                         # not tested
sage: a                                       # not tested
Animation with 328 frames
```

```
>>> from sage.all import *
>>> a = T.animate()                         # not tested
>>> a                                       # not tested
Animation with 328 frames
```

3d Puzzle

The same thing done in 3d *without* allowing reflections this time:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: L = []
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,2,0),(1,
↪3,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,2,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,3,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,3,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,1,0),(1,2,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,1,0),(1,3,0)]))
sage: L.append(Polyomino([(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,3,0)]))
sage: L.append(Polyomino([(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,2,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(1,0,0),(1,1,0),(1,2,0)]))
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino, TilingSolver
>>> L = []
>>> L.append(Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(2),Integer(0)),(Integer(0),Integer(3),Integer(0)),
↪(Integer(1),Integer(0),Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),
↪Integer(2),Integer(0)),(Integer(1),Integer(3),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(2),Integer(0)),(Integer(0),Integer(3),Integer(0)),
↪(Integer(1),Integer(0),Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),
↪Integer(2),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(2),Integer(0)),(Integer(0),Integer(3),Integer(0)),
↪(Integer(1),Integer(0),Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),
↪Integer(3),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(2),Integer(0)),(Integer(0),Integer(3),Integer(0)),
↪(Integer(1),Integer(0),Integer(0)),(Integer(1),Integer(3),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(2),Integer(0)),(Integer(0),Integer(3),Integer(0)),
↪(Integer(1),Integer(0),Integer(0)),(Integer(1),Integer(1),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(2),Integer(0)),(Integer(0),Integer(3),Integer(0)),
↪(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(2),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(2),Integer(0)),(Integer(0),Integer(3),Integer(0)),
↪(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(3),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(1),Integer(0)),(Integer(0),Integer(2),
↪Integer(0)),(Integer(0),Integer(3),Integer(0)),(Integer(1),Integer(0),Integer(0)),
↪(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(3),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(1),Integer(0)),(Integer(0),Integer(2),
↪Integer(0)),(Integer(0),Integer(3),Integer(0)),(Integer(1),Integer(0),Integer(0)),
↪(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(2),Integer(0))]))
>>> L.append(Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(2),Integer(0)),(Integer(1),Integer(0),Integer(0)),
↪(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(2),Integer(0))]))
```

Solve the puzzle and show one solution:

```
sage: T = TilingSolver(L, box=(8,8,1))
sage: solution = next(T.solve())                                # long time (8s)
sage: G = sum([p.show3d(size=0.85) for p in solution], Graphics())  # long time (<1s),
↪ needs sage.plot
sage: G.show(aspect_ratio=1, viewer='tachyon')                  # long time (2s),␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> T = TilingSolver(L, box=(Integer(8),Integer(8),Integer(1)))
>>> solution = next(T.solve())                                # long time (8s)
>>> G = sum([p.show3d(size=RealNumber('0.85')) for p in solution], Graphics())  #␣
↪long time (<1s), needs sage.plot
>>> G.show(aspect_ratio=Integer(1), viewer='tachyon')                  # long␣
↪time (2s), needs sage.plot
```

Let us compute the number of solutions:

```
sage: T.number_of_solutions()                                          # long time (3s)
328
```

```
>>> from sage.all import *
>>> T.number_of_solutions()                                          # long time (3s)
328
```

Donald Knuth example : the Y pentamino

Donald Knuth [Knuth1] considered the problem of packing 45 Y pentaminoes into a 15×15 square:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: y = Polyomino([(0,0),(1,0),(2,0),(3,0),(2,1)])
sage: T = TilingSolver([y], box=(5,10), reusable=True, reflection=True)
sage: T.number_of_solutions()
10
sage: solution = next(T.solve())
sage: G = sum([p.show2d() for p in solution], Graphics())                            #␣
↪needs sage.plot
sage: G.show(aspect_ratio=1)                                           # long time (2s),␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino, TilingSolver
>>> y = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),(Integer(2),
↪Integer(0)),(Integer(3),Integer(0)),(Integer(2),Integer(1))])
>>> T = TilingSolver([y], box=(Integer(5),Integer(10)), reusable=True,␣
↪reflection=True)
>>> T.number_of_solutions()
10
>>> solution = next(T.solve())
>>> G = sum([p.show2d() for p in solution], Graphics())                            #␣
↪needs sage.plot
>>> G.show(aspect_ratio=Integer(1))                                          # long␣
↪time (2s), needs sage.plot
```

```
sage: T = TilingSolver([y], box=(15,15), reusable=True, reflection=True)
sage: T.number_of_solutions()                         # not tested
1696
```

```
>>> from sage.all import *
>>> T = TilingSolver([y], box=(Integer(15),Integer(15)), reusable=True,␣
↪reflection=True)
>>> T.number_of_solutions()                         # not tested
1696
```

Up to the symmetries of the square, there are 212 distinct solutions:

```
sage: 1696 // 8
212
```

```
>>> from sage.all import *
>>> Integer(1696) // Integer(8)
212
```

Animation of Donald Knuth's dancing links

Animation of the solutions:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: Y = Polyomino([(0,0),(1,0),(2,0),(3,0),(2,1)], color='yellow')
sage: T = TilingSolver([Y], box=(15,15), reusable=True, reflection=True)
sage: a = T.animate(stop=40); a                    # long time, optional -␣
↪imagemagick, needs sage.plot
Animation with 40 frames
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino, TilingSolver
>>> Y = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),(Integer(2),
↪Integer(0)),(Integer(3),Integer(0)),(Integer(2),Integer(1))], color='yellow')
>>> T = TilingSolver([Y], box=(Integer(15),Integer(15)), reusable=True,␣
↪reflection=True)
>>> a = T.animate(stop=Integer(40)); a                    # long time, optional -␣
↪imagemagick, needs sage.plot
Animation with 40 frames
```

Incremental animation of the solutions (one piece is removed/added at a time):

```
sage: a = T.animate('incremental', stop=40); a      # long time, optional -␣
↪imagemagick, needs sage.plot
Animation with 40 frames
sage: a.show(delay=50, iterations=1)                # long time, optional -␣
↪imagemagick, needs sage.plot
```

```
>>> from sage.all import *
>>> a = T.animate('incremental', stop=Integer(40)); a      # long time, optional -␣
↪imagemagick, needs sage.plot
Animation with 40 frames
>>> a.show(delay=Integer(50), iterations=Integer(1))                # long time,␣
↪optional - imagemagick, needs sage.plot
```

5d Easy Example

Here is a 5d example. Let us try to fill the $2 \times 2 \times 2 \times 2 \times 2$ rectangle with reusable $1 \times 1 \times 1 \times 1 \times 1$ rectangles. Obviously, there is one solution:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: p = Polyomino([(0,0,0,0,0)])
sage: T = TilingSolver([p], box=(2,2,2,2,2), reusable=True)
sage: rows = T.rows()                              # long time (3s)
sage: rows                                         # long time (fast)
[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12],
 [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24],
```

(continues on next page)

(continued from previous page)

```
 [25], [26], [27], [28], [29], [30], [31]]
sage: T.number_of_solutions()                        # long time (fast)
1
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino, TilingSolver
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0),Integer(0),Integer(0))])
>>> T = TilingSolver([p], box=(Integer(2),Integer(2),Integer(2),Integer(2),
↪Integer(2)), reusable=True)
>>> rows = T.rows()                                  # long time (3s)
>>> rows                                             # long time (fast)
[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12],
 [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24],
 [25], [26], [27], [28], [29], [30], [31]]
>>> T.number_of_solutions()                          # long time (fast)
1
```

REFERENCES:

class sage.combinat.tiling.**Polyomino**(*coords*, *color='gray'*, *dimension=None*)

Bases: SageObject

A polyomino in $\mathbf{Z}^d$.

The polyomino is the union of the unit square (or cube, or n-cube) centered at those coordinates. Such an object should be connected, but the code does not make this assumption.

INPUT:

- `coords` – iterable of integer coordinates in $\mathbf{Z}^d$
- `color` – string (default: `'gray'`); color for display
- `dimension` – integer (default: `None`); dimension of the space, if `None`, it is guessed from the `coords` if `coords` is non empty

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
Polyomino: [(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1)], Color: blue
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(1),Integer(1),Integer(0)), (Integer(1),Integer(1),
↪Integer(1))], color='blue')
Polyomino: [(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1)], Color: blue
```

boundary()

Return the boundary of a 2d polyomino.

INPUT:

- `self` – a 2d polyomino

OUTPUT:

- list of edges (an edge is a pair of adjacent 2d coordinates)

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0), (1,0), (0,1), (1,1)])
sage: sorted(p.boundary())
[((-0.5, -0.5), (-0.5, 0.5)), ((-0.5, -0.5), (0.5, -0.5)),
 ((-0.5, 0.5), (-0.5, 1.5)), ((-0.5, 1.5), (0.5, 1.5)),
 ((0.5, -0.5), (1.5, -0.5)), ((0.5, 1.5), (1.5, 1.5)),
 ((1.5, -0.5), (1.5, 0.5)), ((1.5, 0.5), (1.5, 1.5))]
sage: len(_)
8
sage: p = Polyomino([(5,5)])
sage: sorted(p.boundary())
[((4.5, 4.5), (4.5, 5.5)), ((4.5, 4.5), (5.5, 4.5)),
 ((4.5, 5.5), (5.5, 5.5)), ((5.5, 4.5), (5.5, 5.5))]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪(Integer(0),Integer(1)), (Integer(1),Integer(1))])
>>> sorted(p.boundary())
[((-0.5, -0.5), (-0.5, 0.5)), ((-0.5, -0.5), (0.5, -0.5)),
 ((-0.5, 0.5), (-0.5, 1.5)), ((-0.5, 1.5), (0.5, 1.5)),
 ((0.5, -0.5), (1.5, -0.5)), ((0.5, 1.5), (1.5, 1.5)),
 ((1.5, -0.5), (1.5, 0.5)), ((1.5, 0.5), (1.5, 1.5))]
>>> len(_)
8
>>> p = Polyomino([(Integer(5),Integer(5))])
>>> sorted(p.boundary())
[((4.5, 4.5), (4.5, 5.5)), ((4.5, 4.5), (5.5, 4.5)),
 ((4.5, 5.5), (5.5, 5.5)), ((5.5, 4.5), (5.5, 5.5))]
```

bounding_box()

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink
↪')
sage: p.bounding_box()
[[0, 0, 0], [1, 2, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(1),Integer(0),
↪Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(1),
↪Integer(1)),(Integer(1),Integer(2),Integer(0))], color='deeppink')
>>> p.bounding_box()
[[0, 0, 0], [1, 2, 1]]
```

canonical()

Return the translated copy of `self` having minimal and nonnegative coordinates

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink
↪')
sage: p
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color:␣
↪deeppink
sage: p.canonical()
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color:␣
↪deeppink
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(1),Integer(0),
↪Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(1),
↪Integer(1)),(Integer(1),Integer(2),Integer(0))], color='deeppink')
>>> p
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color:␣
↪deeppink
>>> p.canonical()
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color:␣
↪deeppink
```

canonical_isometric_copies(*orientation_preserving=True*, *mod_box_isometries=False*)

Return the list of image of `self` under isometries of the n-cube where the coordinates are all nonnegative and minimal.

INPUT:

- `orientation_preserving` – boolean (default: `True`); if `True`, the group of isometries of the n-cube is restricted to those that preserve the orientation, i.e. of determinant 1.
- `mod_box_isometries` – boolean (default: `False`); whether to quotient the group of isometries of the n-cube by the subgroup of isometries of the $a_1 \times a_2 \cdots \times a_n$ rectangular box where are the a_i are assumed to be distinct.

OUTPUT: set of Polyomino

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
sage: s = p.canonical_isometric_copies()
sage: len(s)
12
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(1),Integer(1),Integer(0)), (Integer(1),Integer(1),
↪Integer(1))], color='blue')
>>> s = p.canonical_isometric_copies()
>>> len(s)
12
```

With the non orientation-preserving:

```
sage: s = p.canonical_isometric_copies(orientation_preserving=False)
sage: len(s)
24
```

```
>>> from sage.all import *
>>> s = p.canonical_isometric_copies(orientation_preserving=False)
>>> len(s)
24
```

Modulo rotation by angle 180 degrees:

```
sage: s = p.canonical_isometric_copies(mod_box_isometries=True)
sage: len(s)
3
```

```
>>> from sage.all import *
>>> s = p.canonical_isometric_copies(mod_box_isometries=True)
>>> len(s)
3
```

center()

Return the center of the polyomino.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(0,0,1)])
sage: p.center()
(0, 0, 1/2)
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(0),
↪Integer(1))])
>>> p.center()
(0, 0, 1/2)
```

In 3d:

```
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink
↪')
sage: p.center()
(4/5, 4/5, 1/5)
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(1),Integer(0),
↪Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(1),
↪Integer(1)),(Integer(1),Integer(2),Integer(0))], color='deeppink')
>>> p.center()
(4/5, 4/5, 1/5)
```

In 2d:

```
sage: p = Polyomino([(0,0),(1,0),(1,1),(1,2)])
sage: p.center()
(3/4, 3/4)
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(1),Integer(1)),(Integer(1),Integer(2))])
>>> p.center()
(3/4, 3/4)
```

color(*color=None*)

Return or change the color of the polyomino.

INPUT:

- `color` – string, RBG tuple or `None` (default: `None`), if `None`, it returns the current color

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
sage: p.color()
'blue'
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(1),Integer(1),Integer(0)), (Integer(1),Integer(1),
↪Integer(1))], color='blue')
>>> p.color()
'blue'
```

frozenset()

Return the elements of $\mathbf{Z}^d$ in the polyomino as a frozenset.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='red')
sage: p.frozenset()
frozenset({(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1)})
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(1),Integer(1),Integer(0)), (Integer(1),Integer(1),
↪Integer(1))], color='red')
>>> p.frozenset()
frozenset({(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1)})
```

intersection(*other*)

Return the intersection of `self` and `other`.

INPUT:

- `other` – a polyomino

OUTPUT: polyomino

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: a = Polyomino([(0,0)])
sage: b = Polyomino([(0,0), (0,1), (1,1), (2,1)])
sage: a.intersection(b)
Polyomino: [(0, 0)], Color: gray
sage: a.intersection(b+(1,1))
Polyomino: [], Color: gray
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> a = Polyomino([(Integer(0),Integer(0))])
>>> b = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1)),
↪(Integer(1),Integer(1)), (Integer(2),Integer(1))])
>>> a.intersection(b)
Polyomino: [(0, 0)], Color: gray
>>> a.intersection(b+(Integer(1),Integer(1)))
Polyomino: [], Color: gray
```

isometric_copies(*box*, *orientation_preserving=True*, *mod_box_isometries=False*)

Return the translated and isometric images of `self` that lies in the box.

INPUT:

- `box` – polyomino or tuple of integers (size of a box)
- `orientation_preserving` – boolean (default: `True`); if `True`, the group of isometries of the n-cube is restricted to those that preserve the orientation, i.e. of determinant 1.
- `mod_box_isometries` – boolean (default: `False`); whether to quotient the group of isometries of the n-cube by the subgroup of isometries of the $a_1 \times a_2 \cdots \times a_n$ rectangular box where are the a_i are assumed to be distinct.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink
↪')
sage: L = list(p.isometric_copies(box=(5,8,2)))
sage: len(L)
360
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(1),Integer(0),
↪Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(1),
↪Integer(1)),(Integer(1),Integer(2),Integer(0))], color='deeppink')
>>> L = list(p.isometric_copies(box=(Integer(5),Integer(8),Integer(2))))
>>> len(L)
360
```

```
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,2,0),(1,2,1)], color='orange')
sage: L = list(p.isometric_copies(box=(5,8,2)))
sage: len(L)
180
sage: L = list(p.isometric_copies((5,8,2), False))
sage: len(L)
360
sage: L = list(p.isometric_copies((5,8,2), mod_box_isometries=True))
sage: len(L)
45
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(1),Integer(0),
→Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(2),
→Integer(0)),(Integer(1),Integer(2),Integer(1))], color='orange')
>>> L = list(p.isometric_copies(box=(Integer(5),Integer(8),Integer(2))))
>>> len(L)
180
>>> L = list(p.isometric_copies((Integer(5),Integer(8),Integer(2)), False))
>>> len(L)
360
>>> L = list(p.isometric_copies((Integer(5),Integer(8),Integer(2)), mod_box_
→isometries=True))
>>> len(L)
45
```

```
sage: p = Polyomino([(0,0), (1,0), (0,1)])
sage: b = Polyomino([(0,0), (1,0), (2,0), (0,1), (1,1), (0,2)])
sage: sorted(p.isometric_copies(b), key=lambda p: p.sorted_list())
[Polyomino: [(0, 0), (0, 1), (1, 0)], Color: gray,
 Polyomino: [(0, 0), (0, 1), (1, 1)], Color: gray,
 Polyomino: [(0, 0), (1, 0), (1, 1)], Color: gray,
 Polyomino: [(0, 1), (0, 2), (1, 1)], Color: gray,
 Polyomino: [(0, 1), (1, 0), (1, 1)], Color: gray,
 Polyomino: [(1, 0), (1, 1), (2, 0)], Color: gray]
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
→(Integer(0),Integer(1))])
>>> b = Polyomino([(Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
→(Integer(2),Integer(0)), (Integer(0),Integer(1)), (Integer(1),Integer(1)),␣
→(Integer(0),Integer(2))])
>>> sorted(p.isometric_copies(b), key=lambda p: p.sorted_list())
[Polyomino: [(0, 0), (0, 1), (1, 0)], Color: gray,
 Polyomino: [(0, 0), (0, 1), (1, 1)], Color: gray,
 Polyomino: [(0, 0), (1, 0), (1, 1)], Color: gray,
 Polyomino: [(0, 1), (0, 2), (1, 1)], Color: gray,
 Polyomino: [(0, 1), (1, 0), (1, 1)], Color: gray,
 Polyomino: [(1, 0), (1, 1), (2, 0)], Color: gray]
```

isometric_copies_intersection(*box*, *orientation_preserving=True*)

Return the set of non empty intersections of isometric images of `self` with a polyomino.

INPUT:

- `box` – polyomino or tuple of integers (size of a box)
- `orientation_preserving` – boolean (default: `True`); if `True`, the group of isometries of the n-cube is restricted to those that preserve the orientation, i.e. of determinant 1.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0),(1,0)], color='deeppink')
sage: sorted(sorted(a.frozenset()) for a in p.isometric_copies_
↪intersection(box=(2,3)))
[[(0, 0)],
 [(0, 0), (0, 1)],
 [(0, 0), (1, 0)],
 [(0, 1)],
 [(0, 1), (0, 2)],
 [(0, 1), (1, 1)],
 [(0, 2)],
 [(0, 2), (1, 2)],
 [(1, 0)],
 [(1, 0), (1, 1)],
 [(1, 1)],
 [(1, 1), (1, 2)],
 [(1, 2)]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0))], color=
↪'deeppink')
>>> sorted(sorted(a.frozenset()) for a in p.isometric_copies_
↪intersection(box=(Integer(2),Integer(3))))
[[(0, 0)],
 [(0, 0), (0, 1)],
 [(0, 0), (1, 0)],
 [(0, 1)],
 [(0, 1), (0, 2)],
 [(0, 1), (1, 1)],
 [(0, 2)],
 [(0, 2), (1, 2)],
 [(1, 0)],
 [(1, 0), (1, 1)],
 [(1, 1)],
 [(1, 1), (1, 2)],
 [(1, 2)]]
```

neighbor_edges()

Return an iterator over the pairs of neighbor coordinates inside of the polyomino.

Two points P and Q in the polyomino are neighbor if $P - Q$ has one coordinate equal to $+1$ or -1 and zero everywhere else.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(0,0,1)])
sage: [sorted(edge) for edge in p.neighbor_edges()]
[[(0, 0, 0), (0, 0, 1)]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(0),Integer(0),
↪Integer(1))])
>>> [sorted(edge) for edge in p.neighbor_edges()]
[[(0, 0, 0), (0, 0, 1)]]
```

In 3d:

```
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink
↪')
sage: L = sorted(sorted(edge) for edge in p.neighbor_edges())
sage: for a in L: a
[(0, 0, 0), (1, 0, 0)]
[(1, 0, 0), (1, 1, 0)]
[(1, 1, 0), (1, 1, 1)]
[(1, 1, 0), (1, 2, 0)]
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(1),Integer(0),
↪Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(1),
↪Integer(1)),(Integer(1),Integer(2),Integer(0))], color='deeppink')
>>> L = sorted(sorted(edge) for edge in p.neighbor_edges())
>>> for a in L: a
[(0, 0, 0), (1, 0, 0)]
[(1, 0, 0), (1, 1, 0)]
[(1, 1, 0), (1, 1, 1)]
[(1, 1, 0), (1, 2, 0)]
```

In 2d:

```
sage: p = Polyomino([(0,0),(1,0),(1,1),(1,2)])
sage: L = sorted(sorted(edge) for edge in p.neighbor_edges())
sage: for a in L: a
[(0, 0), (1, 0)]
[(1, 0), (1, 1)]
[(1, 1), (1, 2)]
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(1),Integer(1)),(Integer(1),Integer(2))])
>>> L = sorted(sorted(edge) for edge in p.neighbor_edges())
>>> for a in L: a
[(0, 0), (1, 0)]
[(1, 0), (1, 1)]
[(1, 1), (1, 2)]
```

self_surrounding(*radius*, *remove_incomplete_copies=True*, *ncpus=None*)

Return a list of isometric copies of `self` surrounding it with an annulus of given radius.

INPUT:

- `self` – a polyomino of dimension 2
- `radius` – integer
- `remove_incomplete_copies` – boolean (default: `True`); whether to keep only complete copies of `self` in the output
- `ncpus` – integer (default: `None`); maximal number of subprocesses to use at the same time. If `None`, it detects the number of effective CPUs in the system using `sage.parallel.ncpus.ncpus()`. If `ncpus=1`, the first solution is searched serially.

OUTPUT: list of polyominoes

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: H = Polyomino([(-1, 1), (-1, 4), (-1, 7), (0, 0), (0, 1), (0, 2),
....: (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (1, 1), (1, 2),
....: (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (2, 0), (2, 2),
....: (2, 3), (2, 5), (2, 6), (2, 8)])
sage: solution = H.self_surrounding(8)
sage: G = sum([p.show2d() for p in solution], Graphics())                    #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> H = Polyomino([(-Integer(1), Integer(1)), (-Integer(1), Integer(4)), (-
↪Integer(1), Integer(7)), (Integer(0), Integer(0)), (Integer(0), Integer(1)),
↪ (Integer(0), Integer(2)),
... (Integer(0), Integer(3)), (Integer(0), Integer(4)), (Integer(0),␣
↪Integer(5)), (Integer(0), Integer(6)), (Integer(0), Integer(7)),␣
↪(Integer(0), Integer(8)), (Integer(1), Integer(1)), (Integer(1),␣
↪Integer(2)),
... (Integer(1), Integer(3)), (Integer(1), Integer(4)), (Integer(1),␣
↪Integer(5)), (Integer(1), Integer(6)), (Integer(1), Integer(7)),␣
↪(Integer(1), Integer(8)), (Integer(2), Integer(0)), (Integer(2),␣
↪Integer(2)),
... (Integer(2), Integer(3)), (Integer(2), Integer(5)), (Integer(2),␣
↪Integer(6)), (Integer(2), Integer(8))])
>>> solution = H.self_surrounding(Integer(8))
>>> G = sum([p.show2d() for p in solution], Graphics())                    #␣
↪needs sage.plot
```

```
sage: solution = H.self_surrounding(8, remove_incomplete_copies=False)
sage: G = sum([p.show2d() for p in solution], Graphics())                    #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> solution = H.self_surrounding(Integer(8), remove_incomplete_copies=False)
>>> G = sum([p.show2d() for p in solution], Graphics())                    #␣
↪needs sage.plot
```

show2d(*size=0.7*, *color='black'*, *thickness=1*)

Return a 2d Graphic object representing the polyomino.

INPUT:

- `self` – a polyomino of dimension 2
- `size` – number (default: `0.7`); the size of each square
- `color` – color (default: `'black'`); color of the boundary line
- `thickness` – number (default: `1`); how thick the boundary line is

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0),(1,0),(1,1),(1,2)], color='deeppink')
sage: p.show2d()                              # long time (0.5s)                  #␣
↪needs sage.plot
Graphics object consisting of 17 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(1),Integer(1)),(Integer(1),Integer(2))], color='deeppink')
>>> p.show2d()                              # long time (0.5s)                  #␣
↪needs sage.plot
Graphics object consisting of 17 graphics primitives
```

show3d(*size=1*)

Return a 3d Graphic object representing the polyomino.

INPUT:

- `self` – a polyomino of dimension 3
- `size` – number (default: `1`); the size of each `1 \times 1 \times 1` cube. This does a homothety with respect to the center of the polyomino.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
sage: p.show3d()                              # long time (2s)                    #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(1),Integer(1),Integer(0)), (Integer(1),Integer(1),
↪Integer(1))], color='blue')
>>> p.show3d()                              # long time (2s)                    #␣
↪needs sage.plot
Graphics3d Object
```

sorted_list()

Return the color of the polyomino.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
sage: p.sorted_list()
[(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1)]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(1),Integer(1),Integer(0)), (Integer(1),Integer(1),
↪Integer(1))], color='blue')
>>> p.sorted_list()
[(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1)]
```

translated_copies(*box*)

Return an iterator over the translated images of `self` inside a polyomino.

INPUT:

- `box` – polyomino or tuple of integers (size of a box)

OUTPUT: iterator of 3d polyominoes

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink
↪')
sage: for t in p.translated_copies(box=(5,8,2)): t
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color:␣
↪deeppink
Polyomino: [(0, 1, 0), (1, 1, 0), (1, 2, 0), (1, 2, 1), (1, 3, 0)], Color:␣
↪deeppink
Polyomino: [(0, 2, 0), (1, 2, 0), (1, 3, 0), (1, 3, 1), (1, 4, 0)], Color:␣
↪deeppink
Polyomino: [(0, 3, 0), (1, 3, 0), (1, 4, 0), (1, 4, 1), (1, 5, 0)], Color:␣
↪deeppink
Polyomino: [(0, 4, 0), (1, 4, 0), (1, 5, 0), (1, 5, 1), (1, 6, 0)], Color:␣
↪deeppink
Polyomino: [(0, 5, 0), (1, 5, 0), (1, 6, 0), (1, 6, 1), (1, 7, 0)], Color:␣
↪deeppink
Polyomino: [(1, 0, 0), (2, 0, 0), (2, 1, 0), (2, 1, 1), (2, 2, 0)], Color:␣
↪deeppink
Polyomino: [(1, 1, 0), (2, 1, 0), (2, 2, 0), (2, 2, 1), (2, 3, 0)], Color:␣
↪deeppink
Polyomino: [(1, 2, 0), (2, 2, 0), (2, 3, 0), (2, 3, 1), (2, 4, 0)], Color:␣
↪deeppink
Polyomino: [(1, 3, 0), (2, 3, 0), (2, 4, 0), (2, 4, 1), (2, 5, 0)], Color:␣
↪deeppink
Polyomino: [(1, 4, 0), (2, 4, 0), (2, 5, 0), (2, 5, 1), (2, 6, 0)], Color:␣
↪deeppink
Polyomino: [(1, 5, 0), (2, 5, 0), (2, 6, 0), (2, 6, 1), (2, 7, 0)], Color:␣
↪deeppink
Polyomino: [(2, 0, 0), (3, 0, 0), (3, 1, 0), (3, 1, 1), (3, 2, 0)], Color:␣
```

(continues on next page)

(continued from previous page)

```
↪deeppink
Polyomino: [(2, 1, 0), (3, 1, 0), (3, 2, 0), (3, 2, 1), (3, 3, 0)], Color:␣
↪deeppink
Polyomino: [(2, 2, 0), (3, 2, 0), (3, 3, 0), (3, 3, 1), (3, 4, 0)], Color:␣
↪deeppink
Polyomino: [(2, 3, 0), (3, 3, 0), (3, 4, 0), (3, 4, 1), (3, 5, 0)], Color:␣
↪deeppink
Polyomino: [(2, 4, 0), (3, 4, 0), (3, 5, 0), (3, 5, 1), (3, 6, 0)], Color:␣
↪deeppink
Polyomino: [(2, 5, 0), (3, 5, 0), (3, 6, 0), (3, 6, 1), (3, 7, 0)], Color:␣
↪deeppink
Polyomino: [(3, 0, 0), (4, 0, 0), (4, 1, 0), (4, 1, 1), (4, 2, 0)], Color:␣
↪deeppink
Polyomino: [(3, 1, 0), (4, 1, 0), (4, 2, 0), (4, 2, 1), (4, 3, 0)], Color:␣
↪deeppink
Polyomino: [(3, 2, 0), (4, 2, 0), (4, 3, 0), (4, 3, 1), (4, 4, 0)], Color:␣
↪deeppink
Polyomino: [(3, 3, 0), (4, 3, 0), (4, 4, 0), (4, 4, 1), (4, 5, 0)], Color:␣
↪deeppink
Polyomino: [(3, 4, 0), (4, 4, 0), (4, 5, 0), (4, 5, 1), (4, 6, 0)], Color:␣
↪deeppink
Polyomino: [(3, 5, 0), (4, 5, 0), (4, 6, 0), (4, 6, 1), (4, 7, 0)], Color:␣
↪deeppink
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0)),(Integer(1),Integer(0),
↪Integer(0)),(Integer(1),Integer(1),Integer(0)),(Integer(1),Integer(1),
↪Integer(1)),(Integer(1),Integer(2),Integer(0))], color='deeppink')
>>> for t in p.translated_copies(box=(Integer(5),Integer(8),Integer(2))): t
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color:␣
↪deeppink
Polyomino: [(0, 1, 0), (1, 1, 0), (1, 2, 0), (1, 2, 1), (1, 3, 0)], Color:␣
↪deeppink
Polyomino: [(0, 2, 0), (1, 2, 0), (1, 3, 0), (1, 3, 1), (1, 4, 0)], Color:␣
↪deeppink
Polyomino: [(0, 3, 0), (1, 3, 0), (1, 4, 0), (1, 4, 1), (1, 5, 0)], Color:␣
↪deeppink
Polyomino: [(0, 4, 0), (1, 4, 0), (1, 5, 0), (1, 5, 1), (1, 6, 0)], Color:␣
↪deeppink
Polyomino: [(0, 5, 0), (1, 5, 0), (1, 6, 0), (1, 6, 1), (1, 7, 0)], Color:␣
↪deeppink
Polyomino: [(1, 0, 0), (2, 0, 0), (2, 1, 0), (2, 1, 1), (2, 2, 0)], Color:␣
↪deeppink
Polyomino: [(1, 1, 0), (2, 1, 0), (2, 2, 0), (2, 2, 1), (2, 3, 0)], Color:␣
↪deeppink
Polyomino: [(1, 2, 0), (2, 2, 0), (2, 3, 0), (2, 3, 1), (2, 4, 0)], Color:␣
↪deeppink
Polyomino: [(1, 3, 0), (2, 3, 0), (2, 4, 0), (2, 4, 1), (2, 5, 0)], Color:␣
↪deeppink
Polyomino: [(1, 4, 0), (2, 4, 0), (2, 5, 0), (2, 5, 1), (2, 6, 0)], Color:␣
↪deeppink
```

(continues on next page)

(continued from previous page)

```
Polyomino: [(1, 5, 0), (2, 5, 0), (2, 6, 0), (2, 6, 1), (2, 7, 0)], Color:␣
↪deeppink
Polyomino: [(2, 0, 0), (3, 0, 0), (3, 1, 0), (3, 1, 1), (3, 2, 0)], Color:␣
↪deeppink
Polyomino: [(2, 1, 0), (3, 1, 0), (3, 2, 0), (3, 2, 1), (3, 3, 0)], Color:␣
↪deeppink
Polyomino: [(2, 2, 0), (3, 2, 0), (3, 3, 0), (3, 3, 1), (3, 4, 0)], Color:␣
↪deeppink
Polyomino: [(2, 3, 0), (3, 3, 0), (3, 4, 0), (3, 4, 1), (3, 5, 0)], Color:␣
↪deeppink
Polyomino: [(2, 4, 0), (3, 4, 0), (3, 5, 0), (3, 5, 1), (3, 6, 0)], Color:␣
↪deeppink
Polyomino: [(2, 5, 0), (3, 5, 0), (3, 6, 0), (3, 6, 1), (3, 7, 0)], Color:␣
↪deeppink
Polyomino: [(3, 0, 0), (4, 0, 0), (4, 1, 0), (4, 1, 1), (4, 2, 0)], Color:␣
↪deeppink
Polyomino: [(3, 1, 0), (4, 1, 0), (4, 2, 0), (4, 2, 1), (4, 3, 0)], Color:␣
↪deeppink
Polyomino: [(3, 2, 0), (4, 2, 0), (4, 3, 0), (4, 3, 1), (4, 4, 0)], Color:␣
↪deeppink
Polyomino: [(3, 3, 0), (4, 3, 0), (4, 4, 0), (4, 4, 1), (4, 5, 0)], Color:␣
↪deeppink
Polyomino: [(3, 4, 0), (4, 4, 0), (4, 5, 0), (4, 5, 1), (4, 6, 0)], Color:␣
↪deeppink
Polyomino: [(3, 5, 0), (4, 5, 0), (4, 6, 0), (4, 6, 1), (4, 7, 0)], Color:␣
↪deeppink
```

This method is independent of the translation of the polyomino:

```
sage: q = Polyomino([(0,0,0), (1,0,0)])
sage: list(q.translated_copies((2,2,1)))
[Polyomino: [(0, 0, 0), (1, 0, 0)], Color: gray,
 Polyomino: [(0, 1, 0), (1, 1, 0)], Color: gray]
sage: q = Polyomino([(34,7,-9), (35,7,-9)])
sage: list(q.translated_copies((2,2,1)))
[Polyomino: [(0, 0, 0), (1, 0, 0)], Color: gray,
 Polyomino: [(0, 1, 0), (1, 1, 0)], Color: gray]
```

```
>>> from sage.all import *
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(1),Integer(0),
↪Integer(0))])
>>> list(q.translated_copies((Integer(2),Integer(2),Integer(1))))
[Polyomino: [(0, 0, 0), (1, 0, 0)], Color: gray,
 Polyomino: [(0, 1, 0), (1, 1, 0)], Color: gray]
>>> q = Polyomino([(Integer(34),Integer(7),-Integer(9)), (Integer(35),
↪Integer(7),-Integer(9))])
>>> list(q.translated_copies((Integer(2),Integer(2),Integer(1))))
[Polyomino: [(0, 0, 0), (1, 0, 0)], Color: gray,
 Polyomino: [(0, 1, 0), (1, 1, 0)], Color: gray]
```

Inside smaller boxes:

```
sage: list(p.translated_copies(box=(2,2,3)))
[]
sage: list(p.translated_copies(box=(2,3,2)))
[Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color:␣
↪deeppink]
sage: list(p.translated_copies(box=(3,2,2)))
[]
sage: list(p.translated_copies(box=(1,1,1)))
[]
```

```
>>> from sage.all import *
>>> list(p.translated_copies(box=(Integer(2),Integer(2),Integer(3))))
[]
>>> list(p.translated_copies(box=(Integer(2),Integer(3),Integer(2))))
[Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color:␣
↪deeppink]
>>> list(p.translated_copies(box=(Integer(3),Integer(2),Integer(2))))
[]
>>> list(p.translated_copies(box=(Integer(1),Integer(1),Integer(1))))
[]
```

Using a Polyomino as input:

```
sage: b = Polyomino([(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)])
sage: p = Polyomino([(0,0)])
sage: list(p.translated_copies(b))
[Polyomino: [(0, 0)], Color: gray,
 Polyomino: [(0, 1)], Color: gray,
 Polyomino: [(0, 2)], Color: gray,
 Polyomino: [(1, 0)], Color: gray,
 Polyomino: [(1, 1)], Color: gray,
 Polyomino: [(1, 2)], Color: gray]
```

```
>>> from sage.all import *
>>> b = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(0),Integer(2)), (Integer(1),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(1),Integer(2))])
>>> p = Polyomino([(Integer(0),Integer(0))])
>>> list(p.translated_copies(b))
[Polyomino: [(0, 0)], Color: gray,
 Polyomino: [(0, 1)], Color: gray,
 Polyomino: [(0, 2)], Color: gray,
 Polyomino: [(1, 0)], Color: gray,
 Polyomino: [(1, 1)], Color: gray,
 Polyomino: [(1, 2)], Color: gray]
```

```
sage: p = Polyomino([(0,0), (1,0), (0,1)])
sage: b = Polyomino([(0,0), (1,0), (2,0), (0,1), (1,1), (0,2)])
sage: list(p.translated_copies(b))
[Polyomino: [(0, 0), (0, 1), (1, 0)], Color: gray,
 Polyomino: [(0, 1), (0, 2), (1, 1)], Color: gray,
 Polyomino: [(1, 0), (1, 1), (2, 0)], Color: gray]
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
↪(Integer(0),Integer(1))])
>>> b = Polyomino([(Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
↪(Integer(2),Integer(0)), (Integer(0),Integer(1)), (Integer(1),Integer(1)),␣
↪(Integer(0),Integer(2))])
>>> list(p.translated_copies(b))
[Polyomino: [(0, 0), (0, 1), (1, 0)], Color: gray,
 Polyomino: [(0, 1), (0, 2), (1, 1)], Color: gray,
 Polyomino: [(1, 0), (1, 1), (2, 0)], Color: gray]
```

translated_copies_intersection(*box*)

Return the set of non empty intersections of translated images of `self` with a polyomino.

INPUT:

- `box` – polyomino or tuple of integers (size of a box)

OUTPUT: set of 3d polyominoes

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0),(1,0)], color='deeppink')
sage: sorted(sorted(a.frozenset())
....:         for a in p.translated_copies_intersection(box=(2,3)))
[[(0, 0)],
 [(0, 0), (1, 0)],
 [(0, 1)],
 [(0, 1), (1, 1)],
 [(0, 2)],
 [(0, 2), (1, 2)],
 [(1, 0)],
 [(1, 1)],
 [(1, 2)]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino
>>> p = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0))], color=
↪'deeppink')
>>> sorted(sorted(a.frozenset())
...         for a in p.translated_copies_intersection(box=(Integer(2),
↪Integer(3))))
[[(0, 0)],
 [(0, 0), (1, 0)],
 [(0, 1)],
 [(0, 1), (1, 1)],
 [(0, 2)],
 [(0, 2), (1, 2)],
 [(1, 0)],
 [(1, 1)],
 [(1, 2)]]
```

Using a Polyomino as input:

```
sage: b = Polyomino([(0,0), (0,1), (0,2), (1,0), (2,0)])
sage: p = Polyomino([(0,0), (1,0)])
sage: sorted(sorted(a.frozenset())
....:         for a in p.translated_copies_intersection(b))
[[(0, 0)], [(0, 0), (1, 0)], [(0, 1)], [(0, 2)], [(1, 0), (2, 0)], [(2, 0)]]
```

```
>>> from sage.all import *
>>> b = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(0),Integer(2)), (Integer(1),Integer(0)), (Integer(2),Integer(0))])
>>> p = Polyomino([(Integer(0),Integer(0)), (Integer(1),Integer(0))])
>>> sorted(sorted(a.frozenset())
...         for a in p.translated_copies_intersection(b))
[[(0, 0)], [(0, 0), (1, 0)], [(0, 1)], [(0, 2)], [(1, 0), (2, 0)], [(2, 0)]]
```

class sage.combinat.tiling.**TilingSolver**(*pieces*, *box*, *rotation=True*, *reflection=False*, *reusable=False*, *outside=False*)

Bases: `SageObject`

Tiling solver.

Solve the problem of tiling a polyomino with a certain number of polyominoes.

INPUT:

- `pieces` – iterable of Polyominoes
- `box` – polyomino or tuple of integers (size of a box)
- `rotation` – boolean (default: `True`); whether to allow rotations
- `reflection` – boolean (default: `False`); whether to allow reflections
- `reusable` – boolean (default: `False`); whether to allow the pieces to be reused
- `outside` – boolean (default: `False`); whether to allow pieces to partially go outside of the box (all non-empty intersection of the pieces with the box are considered)

EXAMPLES:

By default, rotations are allowed and reflections are not allowed:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T
Tiling solver of 3 pieces into a box of size 6
Rotation allowed: True
Reflection allowed: False
Reusing pieces allowed: False
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
```

(continues on next page)

(continued from previous page)

```
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> T
Tiling solver of 3 pieces into a box of size 6
Rotation allowed: True
Reflection allowed: False
Reusing pieces allowed: False
```

Solutions are given by an iterator:

```
sage: it = T.solve()
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
```

```
>>> from sage.all import *
>>> it = T.solve()
>>> for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
```

Another solution:

```
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
```

```
>>> from sage.all import *
>>> for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
```

Tiling of a polyomino by polyominoes:

```
sage: b = Polyomino([(0,0), (1,0), (1,1), (2,1), (1,2), (2,2), (0,3), (1,3)])
sage: p = Polyomino([(0,0), (1,0)])
sage: T = TilingSolver([p], box=b, reusable=True)
sage: T.number_of_solutions()
2
```

```
>>> from sage.all import *
>>> b = Polyomino([(Integer(0),Integer(0)), (Integer(1),Integer(0)), (Integer(1),
↪Integer(1)), (Integer(2),Integer(1)), (Integer(1),Integer(2)), (Integer(2),
↪Integer(2)), (Integer(0),Integer(3)), (Integer(1),Integer(3))])
>>> p = Polyomino([(Integer(0),Integer(0)), (Integer(1),Integer(0))])
>>> T = TilingSolver([p], box=b, reusable=True)
```

(continues on next page)

(continued from previous page)

```
>>> T.number_of_solutions()
2
```

animate(*partial=None*, *stop=None*, *size=0.75*, *axes=False*)

Return an animation of evolving solutions.

INPUT:

- `partial` – string (default: `None`); whether to include partial (incomplete) solutions. It can be one of the following:
  - `None` – include only complete solutions
  - `'common_prefix'` – common prefix between two consecutive solutions
  - `'incremental'` – one piece change at a time
- `stop` – integer (default: `None`); number of frames
- `size` – number (default: `0.75`); the size of each `1 \times 1` square. This does a homothety with respect to the center of each polyomino.
- `axes` – boolean (default: `False`); whether the x and y axes are shown

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: y = Polyomino([(0,0),(1,0),(2,0),(3,0),(2,1)], color='cyan')
sage: T = TilingSolver([y], box=(5,10), reusable=True, reflection=True)
sage: a = T.animate()                                                      #␣
↪needs sage.plot
sage: a                               # long time, optional - imagemagick,␣
↪needs sage.plot
Animation with 10 frames
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino, TilingSolver
>>> y = Polyomino([(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(2),Integer(0)),(Integer(3),Integer(0)),(Integer(2),Integer(1))],␣
↪color='cyan')
>>> T = TilingSolver([y], box=(Integer(5),Integer(10)), reusable=True,␣
↪reflection=True)
>>> a = T.animate()                                                       #␣
↪needs sage.plot
>>> a                                 # long time, optional - imagemagick, needs␣
↪sage.plot
Animation with 10 frames
```

Include partial solutions (common prefix between two consecutive solutions):

```
sage: a = T.animate('common_prefix')                                       #␣
↪needs sage.plot
sage: a                               # long time, optional - imagemagick,␣
↪needs sage.plot
Animation with 19 frames
```

```
>>> from sage.all import *
>>> a = T.animate('common_prefix')                                    #␣
↪needs sage.plot
>>> a                               # long time, optional - imagemagick, needs␣
↪sage.plot
Animation with 19 frames
```

Incremental solutions (one piece removed or added at a time):

```
sage: a = T.animate('incremental')  # long time (2s)                      #␣
↪needs sage.plot
sage: a                             # long time (2s), optional - imagemagick,␣
↪needs sage.plot
Animation with 123 frames
```

```
>>> from sage.all import *
>>> a = T.animate('incremental')  # long time (2s)                      #␣
↪needs sage.plot
>>> a                             # long time (2s), optional - imagemagick,␣
↪needs sage.plot
Animation with 123 frames
```

```
sage: a.show()                      # long time, optional - imagemagick,␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> a.show()                      # long time, optional - imagemagick, needs␣
↪sage.plot
```

The `show` function takes arguments to specify the delay between frames (measured in hundredths of a second, default value 20) and the number of iterations (default: 0, which means to iterate forever). To iterate 4 times with half a second between each frame:

```
sage: a.show(delay=50, iterations=4)        # long time, optional -␣
↪imagemagick, needs sage.plot
```

```
>>> from sage.all import *
>>> a.show(delay=Integer(50), iterations=Integer(4))        # long time,␣
↪optional - imagemagick, needs sage.plot
```

Limit the number of frames:

```
sage: a = T.animate('incremental', stop=13); a       # not tested          #␣
↪needs sage.plot
Animation with 13 frames
```

```
>>> from sage.all import *
>>> a = T.animate('incremental', stop=Integer(13)); a       # not tested      ␣
↪     # needs sage.plot
Animation with 13 frames
```

coord_to_int_dict()

Return a dictionary mapping coordinates to integers.

OUTPUT: dictionary

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: A = T.coord_to_int_dict()
sage: sorted(A.items())
[((0, 0, 0), 3), ((0, 0, 1), 4), ((0, 0, 2), 5),
 ((0, 0, 3), 6), ((0, 0, 4), 7), ((0, 0, 5), 8)]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> A = T.coord_to_int_dict()
>>> sorted(A.items())
[((0, 0, 0), 3), ((0, 0, 1), 4), ((0, 0, 2), 5),
 ((0, 0, 3), 6), ((0, 0, 4), 7), ((0, 0, 5), 8)]
```

Reusable pieces:

```
sage: p = Polyomino([(0,0), (0,1)])
sage: q = Polyomino([(0,0), (0,1), (1,0), (1,1)])
sage: T = TilingSolver([p,q], box=[3,2], reusable=True)
sage: B = T.coord_to_int_dict()
sage: sorted(B.items())
[((0, 0), 0), ((0, 1), 1), ((1, 0), 2), ((1, 1), 3),
 ((2, 0), 4), ((2, 1), 5)]
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1))])
>>> q = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(1),Integer(0)), (Integer(1),Integer(1))])
>>> T = TilingSolver([p,q], box=[Integer(3),Integer(2)], reusable=True)
>>> B = T.coord_to_int_dict()
>>> sorted(B.items())
[((0, 0), 0), ((0, 1), 1), ((1, 0), 2), ((1, 1), 3),
 ((2, 0), 4), ((2, 1), 5)]
```

dlx_solver()

Return the sage DLX solver of that tiling problem.

OUTPUT: dLX Solver

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T.dlx_solver()
Dancing links solver for 9 columns and 15 rows
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> T.dlx_solver()
Dancing links solver for 9 columns and 15 rows
```

int_to_coord_dict()

Return a dictionary mapping integers to coordinates.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: B = T.int_to_coord_dict()
sage: sorted(B.items())
[(3, (0, 0, 0)), (4, (0, 0, 1)), (5, (0, 0, 2)),
 (6, (0, 0, 3)), (7, (0, 0, 4)), (8, (0, 0, 5))]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> B = T.int_to_coord_dict()
>>> sorted(B.items())
[(3, (0, 0, 0)), (4, (0, 0, 1)), (5, (0, 0, 2)),
 (6, (0, 0, 3)), (7, (0, 0, 4)), (8, (0, 0, 5))]
```

Reusable pieces:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: p = Polyomino([(0,0), (0,1)])
sage: q = Polyomino([(0,0), (0,1), (1,0), (1,1)])
sage: T = TilingSolver([p,q], box=[3,2], reusable=True)
```

(continues on next page)

(continued from previous page)

```
sage: B = T.int_to_coord_dict()
sage: sorted(B.items())
[(0, (0, 0)), (1, (0, 1)), (2, (1, 0)),
 (3, (1, 1)), (4, (2, 0)), (5, (2, 1))]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import Polyomino, TilingSolver
>>> p = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1))])
>>> q = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(1),Integer(0)), (Integer(1),Integer(1))])
>>> T = TilingSolver([p,q], box=[Integer(3),Integer(2)], reusable=True)
>>> B = T.int_to_coord_dict()
>>> sorted(B.items())
[(0, (0, 0)), (1, (0, 1)), (2, (1, 0)),
 (3, (1, 1)), (4, (2, 0)), (5, (2, 1))]
```

is_suitable()

Return whether the volume of the box is equal to sum of the volume of the polyominoes and the number of rows sent to the DLX solver is larger than zero.

If these conditions are not verified, then the problem is not suitable in the sense that there are no solution.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T.is_suitable()
True
sage: T = TilingSolver([p,q,r], box=(1,1,7))
sage: T.is_suitable()
False
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> T.is_suitable()
True
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(7)))
>>> T.is_suitable()
False
```

nrows_per_piece()

Return the number of rows necessary by each piece.

OUTPUT: list

EXAMPLES:

```
sage: from sage.games.quantumino import QuantuminoSolver
sage: q = QuantuminoSolver(0)
sage: T = q.tiling_solver()
sage: T.nrows_per_piece()                 # long time (10s)
[360, 360, 360, 360, 360, 180, 180, 672, 672, 360, 360, 180, 180, 360, 360,
↪180]
```

```
>>> from sage.all import *
>>> from sage.games.quantumino import QuantuminoSolver
>>> q = QuantuminoSolver(Integer(0))
>>> T = q.tiling_solver()
>>> T.nrows_per_piece()                 # long time (10s)
[360, 360, 360, 360, 360, 180, 180, 672, 672, 360, 360, 180, 180, 360, 360,
↪180]
```

number_of_solutions()

Return the number of distinct solutions.

OUTPUT: integer

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0)])
sage: q = Polyomino([(0,0), (0,1)])
sage: r = Polyomino([(0,0), (0,1), (0,2)])
sage: T = TilingSolver([p,q,r], box=(1,6))
sage: T.number_of_solutions()
6
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0)), (Integer(0),Integer(1)),
↪(Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(6)))
>>> T.number_of_solutions()
6
```

```
sage: T = TilingSolver([p,q,r], box=(1,7))
sage: T.number_of_solutions()
0
```

```
>>> from sage.all import *
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(7)))
>>> T.number_of_solutions()
0
```

pieces()

Return the list of pieces.

OUTPUT: list of 3d polyominoes

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: for p in T._pieces: p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 0), (0, 0, 1), (0, 0, 2)], Color: gray
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> for p in T._pieces: p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 0), (0, 0, 1), (0, 0, 2)], Color: gray
```

row_to_polyomino(*row_number*)

Return a polyomino associated to a row.

INPUT:

- `row_number` – integer; the i-th row

OUTPUT: polyomino

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: a = Polyomino([(0,0,0), (0,0,1), (1,0,0)], color='blue')
sage: b = Polyomino([(0,0,0), (1,0,0), (0,1,0)], color='red')
sage: T = TilingSolver([a,b], box=(2,1,3))
sage: len(T.rows())
16
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> a = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(1),Integer(0),Integer(0))], color='blue')
>>> b = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(1),Integer(0),
↪Integer(0)), (Integer(0),Integer(1),Integer(0))], color='red')
>>> T = TilingSolver([a,b], box=(Integer(2),Integer(1),Integer(3)))
>>> len(T.rows())
16
```

```
sage: T.row_to_polyomino(7)
Polyomino: [(0, 0, 2), (1, 0, 1), (1, 0, 2)], Color: blue
```

```
>>> from sage.all import *
>>> T.row_to_polyomino(Integer(7))
Polyomino: [(0, 0, 2), (1, 0, 1), (1, 0, 2)], Color: blue
```

```
sage: T.row_to_polyomino(13)
Polyomino: [(0, 0, 1), (1, 0, 1), (1, 0, 2)], Color: red
```

```
>>> from sage.all import *
>>> T.row_to_polyomino(Integer(13))
Polyomino: [(0, 0, 1), (1, 0, 1), (1, 0, 2)], Color: red
```

rows()

Creation of the rows.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: rows = T.rows()
sage: for row in rows: row
[0, 3]
[0, 4]
[0, 5]
[0, 6]
[0, 7]
[0, 8]
[1, 3, 4]
[1, 4, 5]
[1, 5, 6]
[1, 6, 7]
[1, 7, 8]
[2, 3, 4, 5]
[2, 4, 5, 6]
[2, 5, 6, 7]
[2, 6, 7, 8]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> rows = T.rows()
>>> for row in rows: row
```

(continues on next page)

(continued from previous page)

```
[0, 3]
[0, 4]
[0, 5]
[0, 6]
[0, 7]
[0, 8]
[1, 3, 4]
[1, 4, 5]
[1, 5, 6]
[1, 6, 7]
[1, 7, 8]
[2, 3, 4, 5]
[2, 4, 5, 6]
[2, 5, 6, 7]
[2, 6, 7, 8]
```

rows_for_piece(*i*, *mod_box_isometries=False*)

Return the rows for the i-th piece.

INPUT:

- `i` – integer; the i-th piece
- `mod_box_isometries` – boolean (default: `False`); whether to consider only rows for positions up to the action of the quotient the group of isometries of the n-cube by the subgroup of isometries of the $a_1 \times a_2 \cdots \times a_n$ rectangular box where are the a_i are assumed to be distinct.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T.rows_for_piece(0)
[[0, 3], [0, 4], [0, 5], [0, 6], [0, 7], [0, 8]]
sage: T.rows_for_piece(1)
[[1, 3, 4], [1, 4, 5], [1, 5, 6], [1, 6, 7], [1, 7, 8]]
sage: T.rows_for_piece(2)
[[2, 3, 4, 5], [2, 4, 5, 6], [2, 5, 6, 7], [2, 6, 7, 8]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> T.rows_for_piece(Integer(0))
[[0, 3], [0, 4], [0, 5], [0, 6], [0, 7], [0, 8]]
>>> T.rows_for_piece(Integer(1))
[[1, 3, 4], [1, 4, 5], [1, 5, 6], [1, 6, 7], [1, 7, 8]]
```

(continues on next page)

(continued from previous page)

```
>>> T.rows_for_piece(Integer(2))
[[2, 3, 4, 5], [2, 4, 5, 6], [2, 5, 6, 7], [2, 6, 7, 8]]
```

Less rows when using `mod_box_isometries=True`:

```
sage: a = Polyomino([(0,0,0), (0,0,1), (1,0,0)])
sage: b = Polyomino([(0,0,0), (1,0,0), (0,1,0)])
sage: T = TilingSolver([a,b], box=(2,1,3))
sage: T.rows_for_piece(0)
[[0, 2, 3, 5],
 [0, 3, 4, 6],
 [0, 2, 3, 6],
 [0, 3, 4, 7],
 [0, 2, 5, 6],
 [0, 3, 6, 7],
 [0, 3, 5, 6],
 [0, 4, 6, 7]]
sage: T.rows_for_piece(0, mod_box_isometries=True)
[[0, 2, 3, 5], [0, 3, 4, 6]]
sage: T.rows_for_piece(1, mod_box_isometries=True)
[[1, 2, 3, 5], [1, 3, 4, 6]]
```

```
>>> from sage.all import *
>>> a = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(1),Integer(0),Integer(0))])
>>> b = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(1),Integer(0),
↪Integer(0)), (Integer(0),Integer(1),Integer(0))])
>>> T = TilingSolver([a,b], box=(Integer(2),Integer(1),Integer(3)))
>>> T.rows_for_piece(Integer(0))
[[0, 2, 3, 5],
 [0, 3, 4, 6],
 [0, 2, 3, 6],
 [0, 3, 4, 7],
 [0, 2, 5, 6],
 [0, 3, 6, 7],
 [0, 3, 5, 6],
 [0, 4, 6, 7]]
>>> T.rows_for_piece(Integer(0), mod_box_isometries=True)
[[0, 2, 3, 5], [0, 3, 4, 6]]
>>> T.rows_for_piece(Integer(1), mod_box_isometries=True)
[[1, 2, 3, 5], [1, 3, 4, 6]]
```

solve(*partial=None*)

Return an iterator of list of polyominoes that are an exact cover of the box.

INPUT:

- `partial` – string (default: `None`); whether to include partial (incomplete) solutions. It can be one of the following:
  - `None` – include only complete solution
  - `'common_prefix'` – common prefix between two consecutive solutions
  - `'incremental'` – one piece change at a time

OUTPUT: iterator of list of polyominoes

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: it = T.solve()
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
sage: for p in next(it): p
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 2), (0, 0, 3), (0, 0, 4)], Color: gray
Polyomino: [(0, 0, 5)], Color: gray
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> it = T.solve()
>>> for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
>>> for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
>>> for p in next(it): p
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 2), (0, 0, 3), (0, 0, 4)], Color: gray
Polyomino: [(0, 0, 5)], Color: gray
```

Including the partial solutions:

```
sage: it = T.solve(partial='common_prefix')
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
```

(continues on next page)

(continued from previous page)

```
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
sage: for p in next(it): p
sage: for p in next(it): p
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 2), (0, 0, 3), (0, 0, 4)], Color: gray
Polyomino: [(0, 0, 5)], Color: gray
```

```
>>> from sage.all import *
>>> it = T.solve(partial='common_prefix')
>>> for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
>>> for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
>>> for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
>>> for p in next(it): p
>>> for p in next(it): p
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 2), (0, 0, 3), (0, 0, 4)], Color: gray
Polyomino: [(0, 0, 5)], Color: gray
```

Colors are preserved when the polyomino can be reused:

```
sage: p = Polyomino([(0,0,0)], color='yellow')
sage: q = Polyomino([(0,0,0), (0,0,1)], color='yellow')
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)], color='yellow')
sage: T = TilingSolver([p,q,r], box=(1,1,6), reusable=True)
sage: it = T.solve()
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: yellow
Polyomino: [(0, 0, 1)], Color: yellow
Polyomino: [(0, 0, 2)], Color: yellow
Polyomino: [(0, 0, 3)], Color: yellow
Polyomino: [(0, 0, 4)], Color: yellow
Polyomino: [(0, 0, 5)], Color: yellow
```

```
>>> from sage.all import *
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))], color='yellow')
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))], color='yellow')
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))], color='yellow')
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)),␣
↪reusable=True)
>>> it = T.solve()
```

(continues on next page)

(continued from previous page)

```
>>> for p in next(it): p
Polyomino: [(0, 0, 0)], Color: yellow
Polyomino: [(0, 0, 1)], Color: yellow
Polyomino: [(0, 0, 2)], Color: yellow
Polyomino: [(0, 0, 3)], Color: yellow
Polyomino: [(0, 0, 4)], Color: yellow
Polyomino: [(0, 0, 5)], Color: yellow
```

space()

Return an iterator over all the nonnegative integer coordinates contained in the space to tile.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: list(T.space())
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 0, 4), (0, 0, 5)]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> list(T.space())
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 0, 4), (0, 0, 5)]
```

starting_rows()

Return the starting rows for each piece.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T.starting_rows()
[0, 6, 11, 15]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import TilingSolver, Polyomino
>>> p = Polyomino([(Integer(0),Integer(0),Integer(0))])
>>> q = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> r = Polyomino([(Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1)), (Integer(0),Integer(0),Integer(2))])
```

(continues on next page)

(continued from previous page)

```
>>> T = TilingSolver([p,q,r], box=(Integer(1),Integer(1),Integer(6)))
>>> T.starting_rows()
[0, 6, 11, 15]
```

`sage.combinat.tiling.`**`ncube_isometry_group`**`(`*n*, *orientation_preserving=True*`)`

Return the isometry group of the n-cube as a list of matrices.

INPUT:

- `n` – positive integer, dimension of the space
- `orientation_preserving` – boolean (default: `True`); whether the orientation is preserved

OUTPUT: list of matrices

EXAMPLES:

```
sage: from sage.combinat.tiling import ncube_isometry_group
sage: ncube_isometry_group(2)
[
[1 0]  [ 0  1]  [-1  0]  [ 0 -1]
[0 1], [-1  0], [ 0 -1], [ 1  0]
]
sage: ncube_isometry_group(2, orientation_preserving=False)
[
[1 0]  [ 0 -1]  [ 1  0]  [ 0  1]  [0 1]  [-1  0]  [ 0 -1]  [-1  0]
[0 1], [-1  0], [ 0 -1], [-1  0], [1 0], [ 0 -1], [ 1  0], [ 0  1]
]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import ncube_isometry_group
>>> ncube_isometry_group(Integer(2))
[
[1 0]  [ 0  1]  [-1  0]  [ 0 -1]
[0 1], [-1  0], [ 0 -1], [ 1  0]
]
>>> ncube_isometry_group(Integer(2), orientation_preserving=False)
[
[1 0]  [ 0 -1]  [ 1  0]  [ 0  1]  [0 1]  [-1  0]  [ 0 -1]  [-1  0]
[0 1], [-1  0], [ 0 -1], [-1  0], [1 0], [ 0 -1], [ 1  0], [ 0  1]
]
```

There are 24 orientation preserving isometries of the 3-cube:

```
sage: ncube_isometry_group(3)
[
[1 0 0]  [ 1  0  0]  [ 1  0  0]  [ 0  1  0]  [0 1 0]  [ 0  0  1]
[0 1 0]  [ 0  0  1]  [ 0  0 -1]  [-1  0  0]  [0 0 1]  [ 0 -1  0]
[0 0 1], [ 0 -1  0], [ 0  1  0], [ 0  0  1], [1 0 0], [ 1  0  0],

[-1  0  0]  [ 0 -1  0]  [-1  0  0]  [-1  0  0]  [ 0 -1  0]  [ 0  0 -1]
[ 0 -1  0]  [ 0  0 -1]  [ 0  0 -1]  [ 0  1  0]  [ 0  0  1]  [ 1  0  0]
[ 0  0  1], [ 1  0  0], [ 0 -1  0], [ 0  0 -1], [-1  0  0], [ 0 -1  0],
```

(continues on next page)

(continued from previous page)

```
[ 0  1  0]  [ 0  0  1]  [0 0 1]  [ 0 -1  0]  [ 0  0 -1]  [-1  0  0]
[ 1  0  0]  [ 0  1  0]  [1 0 0]  [ 1  0  0]  [ 0  1  0]  [ 0  0  1]
[ 0  0 -1], [-1  0  0], [0 1 0], [ 0  0  1], [ 1  0  0], [ 0  1  0],

[ 0 -1  0]  [ 0  0 -1]  [ 0  0  1]  [ 1  0  0]  [ 0  0 -1]  [ 0  1  0]
[-1  0  0]  [-1  0  0]  [-1  0  0]  [ 0 -1  0]  [ 0 -1  0]  [ 0  0 -1]
[ 0  0 -1], [ 0  1  0], [ 0 -1  0], [ 0  0 -1], [-1  0  0], [-1  0  0]
]
```

```
>>> from sage.all import *
>>> ncube_isometry_group(Integer(3))
[
[1 0 0]  [ 1  0  0]  [ 1  0  0]  [ 0  1  0]  [0 1 0]  [ 0  0  1]
[0 1 0]  [ 0  0  1]  [ 0  0 -1]  [-1  0  0]  [0 0 1]  [ 0 -1  0]
[0 0 1], [ 0 -1  0], [ 0  1  0], [ 0  0  1], [1 0 0], [ 1  0  0],
<BLANKLINE>
[-1  0  0]  [ 0 -1  0]  [-1  0  0]  [-1  0  0]  [ 0 -1  0]  [ 0  0 -1]
[ 0 -1  0]  [ 0  0 -1]  [ 0  0 -1]  [ 0  1  0]  [ 0  0  1]  [ 1  0  0]
[ 0  0  1], [ 1  0  0], [ 0 -1  0], [ 0  0 -1], [-1  0  0], [ 0 -1  0],
<BLANKLINE>
[ 0  1  0]  [ 0  0  1]  [0 0 1]  [ 0 -1  0]  [ 0  0 -1]  [-1  0  0]
[ 1  0  0]  [ 0  1  0]  [1 0 0]  [ 1  0  0]  [ 0  1  0]  [ 0  0  1]
[ 0  0 -1], [-1  0  0], [0 1 0], [ 0  0  1], [ 1  0  0], [ 0  1  0],
<BLANKLINE>
[ 0 -1  0]  [ 0  0 -1]  [ 0  0  1]  [ 1  0  0]  [ 0  0 -1]  [ 0  1  0]
[-1  0  0]  [-1  0  0]  [-1  0  0]  [ 0 -1  0]  [ 0 -1  0]  [ 0  0 -1]
[ 0  0 -1], [ 0  1  0], [ 0 -1  0], [ 0  0 -1], [-1  0  0], [-1  0  0]
]
```

sage.combinat.tiling.**ncube_isometry_group_cosets**(*orientation_preserving=True*)

Return the quotient of the isometry group of the n-cube by the the isometry group of the rectangular parallelepiped.

INPUT:

- `n` – positive integer, dimension of the space
- `orientation_preserving` – boolean (default: `True`); whether the orientation is preserved

OUTPUT: list of cosets, each coset being a sorted list of matrices

EXAMPLES:

```
sage: from sage.combinat.tiling import ncube_isometry_group_cosets
sage: sorted(ncube_isometry_group_cosets(2))
[[
[-1  0]  [1 0]
[ 0 -1], [0 1]
], [
[ 0 -1]  [ 0  1]
[ 1  0], [-1  0]
]]
sage: sorted(ncube_isometry_group_cosets(2, False))
[[
[-1  0]  [-1  0]  [ 1  0]  [1 0]
```

(continues on next page)

(continued from previous page)

```
[ 0 -1], [ 0  1], [ 0 -1], [0 1]
], [
[ 0 -1]  [ 0 -1]  [ 0  1]  [0 1]
[-1  0], [ 1  0], [-1  0], [1 0]
]]
```

```
>>> from sage.all import *
>>> from sage.combinat.tiling import ncube_isometry_group_cosets
>>> sorted(ncube_isometry_group_cosets(Integer(2)))
[[
[-1  0]  [1 0]
[ 0 -1], [0 1]
], [
[ 0 -1]  [ 0  1]
[ 1  0], [-1  0]
]]
>>> sorted(ncube_isometry_group_cosets(Integer(2), False))
[[
[-1  0]  [-1  0]  [ 1  0]  [1 0]
[ 0 -1], [ 0  1], [ 0 -1], [0 1]
], [
[ 0 -1]  [ 0 -1]  [ 0  1]  [0 1]
[-1  0], [ 1  0], [-1  0], [1 0]
]]
```

```
sage: sorted(ncube_isometry_group_cosets(3))
[[
[-1  0  0]  [-1  0  0]  [ 1  0  0]  [1 0 0]
[ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]  [0 1 0]
[ 0  0  1], [ 0  0 -1], [ 0  0 -1], [0 0 1]
], [
[-1  0  0]  [-1  0  0]  [ 1  0  0]  [ 1  0  0]
[ 0  0 -1]  [ 0  0  1]  [ 0  0 -1]  [ 0  0  1]
[ 0 -1  0], [ 0  1  0], [ 0  1  0], [ 0 -1  0]
], [
[ 0 -1  0]  [ 0 -1  0]  [ 0  1  0]  [ 0  1  0]
[-1  0  0]  [ 1  0  0]  [-1  0  0]  [ 1  0  0]
[ 0  0 -1], [ 0  0  1], [ 0  0  1], [ 0  0 -1]
], [
[ 0 -1  0]  [ 0 -1  0]  [ 0  1  0]  [0 1 0]
[ 0  0 -1]  [ 0  0  1]  [ 0  0 -1]  [0 0 1]
[ 1  0  0], [-1  0  0], [-1  0  0], [1 0 0]
], [
[ 0  0 -1]  [ 0  0 -1]  [ 0  0  1]  [0 0 1]
[-1  0  0]  [ 1  0  0]  [-1  0  0]  [1 0 0]
[ 0  1  0], [ 0 -1  0], [ 0 -1  0], [0 1 0]
], [
[ 0  0 -1]  [ 0  0 -1]  [ 0  0  1]  [ 0  0  1]
[ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]  [ 0  1  0]
[-1  0  0], [ 1  0  0], [ 1  0  0], [-1  0  0]
]]
```

```
>>> from sage.all import *
>>> sorted(ncube_isometry_group_cosets(Integer(3)))
[[
[-1  0  0]  [-1  0  0]  [ 1  0  0]  [1 0 0]
[ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]  [0 1 0]
[ 0  0  1], [ 0  0 -1], [ 0  0 -1], [0 0 1]
], [
[-1  0  0]  [-1  0  0]  [ 1  0  0]  [ 1  0  0]
[ 0  0 -1]  [ 0  0  1]  [ 0  0 -1]  [ 0  0  1]
[ 0 -1  0], [ 0  1  0], [ 0  1  0], [ 0 -1  0]
], [
[ 0 -1  0]  [ 0 -1  0]  [ 0  1  0]  [ 0  1  0]
[-1  0  0]  [ 1  0  0]  [-1  0  0]  [ 1  0  0]
[ 0  0 -1], [ 0  0  1], [ 0  0  1], [ 0  0 -1]
], [
[ 0 -1  0]  [ 0 -1  0]  [ 0  1  0]  [0 1 0]
[ 0  0 -1]  [ 0  0  1]  [ 0  0 -1]  [0 0 1]
[ 1  0  0], [-1  0  0], [-1  0  0], [1 0 0]
], [
[ 0  0 -1]  [ 0  0 -1]  [ 0  0  1]  [0 0 1]
[-1  0  0]  [ 1  0  0]  [-1  0  0]  [1 0 0]
[ 0  1  0], [ 0 -1  0], [ 0 -1  0], [0 1 0]
], [
[ 0  0 -1]  [ 0  0 -1]  [ 0  0  1]  [ 0  0  1]
[ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]  [ 0  1  0]
[-1  0  0], [ 1  0  0], [ 1  0  0], [-1  0  0]
]]
```

5.1.350 Transitive ideal closure tool

sage.combinat.tools.**transitive_ideal**(f, x)

Return a list of all elements reachable from x in the abstract reduction system whose reduction relation is given by the function f.

In more elementary terms:

If S is a set, and f is a function sending every element of S to a list of elements of S, then we can define a digraph on the vertex set S by drawing an edge from s to t for every $s \in S$ and every $t \in f(s)$.

If $x \in S$, then an element $y \in S$ is said to be reachable from x if there is a path $x \to y$ in this graph.

Given f and x, this method computes the list of all elements of S reachable from x.

Note that if there are infinitely many such elements, then this method will never halt.

For more powerful versions, see *sage.combinat.backtrack*

EXAMPLES:

```
sage: f = lambda x: [x-1] if x > 0 else []
sage: sage.combinat.tools.transitive_ideal(f, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> from sage.all import *
>>> f = lambda x: [x-Integer(1)] if x > Integer(0) else []
```

(continues on next page)

(continued from previous page)

```
>>> sage.combinat.tools.transitive_ideal(f, Integer(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

5.1.351 Combinatorial triangles for posets and fans

This provides several classes and methods to convert between them. Elements of the classes are polynomials in two variables x and y, possibly with other parameters. The conversion methods amount to specific invertible rational change-of-variables involving x and y.

These polynomial are called triangles because their supports, the sets of exponents where their coefficients can be nonzero, have a triangular shape.

The M-triangle class is motivated by the generating series of Möbius numbers for graded posets. A typical example is:

```
sage: W = SymmetricGroup(4)                                                     #␣
↪needs sage.groups
sage: posets.NoncrossingPartitions(W).M_triangle()                              #␣
↪needs sage.graphs sage.groups
M: x^3*y^3 - 6*x^2*y^3 + 6*x^2*y^2 + 10*x*y^3 - 16*x*y^2
- 5*y^3 + 6*x*y + 10*y^2 - 6*y + 1
sage: unicode_art(_)                                                            #␣
↪needs sage.graphs sage.groups sage.modules
⎛ -5  10  -6   1⎞
⎜ 10 -16   6   0⎟
⎜ -6   6   0   0⎟
⎝  1   0   0   0⎠
```

```
>>> from sage.all import *
>>> W = SymmetricGroup(Integer(4))                                              ␣
↪     # needs sage.groups
>>> posets.NoncrossingPartitions(W).M_triangle()                              #␣
↪needs sage.graphs sage.groups
M: x^3*y^3 - 6*x^2*y^3 + 6*x^2*y^2 + 10*x*y^3 - 16*x*y^2
- 5*y^3 + 6*x*y + 10*y^2 - 6*y + 1
>>> unicode_art(_)                                                            #␣
↪needs sage.graphs sage.groups sage.modules
⎛ -5  10  -6   1⎞
⎜ 10 -16   6   0⎟
⎜ -6   6   0   0⎟
⎝  1   0   0   0⎠
```

The F-triangle class is motivated by the generating series of pure simplicial complexes endowed with a distinguished facet. One can also think about complete fans endowed with a distinguished maximal cone. A typical example is:

```
sage: # needs sage.graphs sage.modules
sage: C = ClusterComplex(['A',3])
sage: f = C.greedy_facet()
sage: C.F_triangle(f)
F: 5*x^3 + 5*x^2*y + 3*x*y^2 + y^3 + 10*x^2 + 8*x*y + 3*y^2 + 6*x + 3*y + 1
sage: unicode_art(_)
⎛ 1  0  0  0⎞
⎜ 3  3  0  0⎟
```

(continues on next page)

(continued from previous page)

```
⎢ 3  8  5  0⎥
⎝ 1  6 10  5⎠
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.modules
>>> C = ClusterComplex(['A',Integer(3)])
>>> f = C.greedy_facet()
>>> C.F_triangle(f)
F: 5*x^3 + 5*x^2*y + 3*x*y^2 + y^3 + 10*x^2 + 8*x*y + 3*y^2 + 6*x + 3*y + 1
>>> unicode_art(_)
⎛ 1  0  0  0⎞
⎢ 3  3  0  0⎥
⎢ 3  8  5  0⎥
⎝ 1  6 10  5⎠
```

The H-triangles are related to the F-triangles by a relationship similar to the classical link between the f-vector and the h-vector of a simplicial complex.

The Gamma-triangles are related to the H-triangles by an analog of the relationship between gamma-vectors and h-vectors of flag simplicial complexes.

class sage.combinat.triangles_FHM.**F_triangle**(*poly*, *variables=None*)

Bases: *Triangle*

Class for the F-triangles.

h()

Return the associated H-triangle.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import F_triangle
sage: x,y = polygens(ZZ,'x,y')
sage: ft = F_triangle(1+x+y)
sage: ft.h()
H: x*y + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import F_triangle
>>> x,y = polygens(ZZ,'x,y')
>>> ft = F_triangle(Integer(1)+x+y)
>>> ft.h()
H: x*y + 1
```

m()

Return the associated M-triangle.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygens(ZZ,'x,y')
sage: H_triangle(1+x*y).f()
F: x + y + 1
sage: _.m()
```

(continues on next page)

(continued from previous page)

```
M: x*y - y + 1

sage: H_triangle(x^2*y^2 + 2*x*y + x + 1).f()
F: 2*x^2 + 2*x*y + y^2 + 3*x + 2*y + 1
sage: _.m()
M: x^2*y^2 - 3*x*y^2 + 3*x*y + 2*y^2 - 3*y + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> x, y = polygens(ZZ,'x,y')
>>> H_triangle(Integer(1)+x*y).f()
F: x + y + 1
>>> _.m()
M: x*y - y + 1

>>> H_triangle(x**Integer(2)*y**Integer(2) + Integer(2)*x*y + x + Integer(1)).
↪f()
F: 2*x^2 + 2*x*y + y^2 + 3*x + 2*y + 1
>>> _.m()
M: x^2*y^2 - 3*x*y^2 + 3*x*y + 2*y^2 - 3*y + 1
```

parabolic()

Return a parabolic version of the F-triangle.

This is obtained by replacing the variable y by $y - 1$.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygens(ZZ,'x,y')
sage: H_triangle(1+x*y).f()
F: x + y + 1
sage: _.parabolic()
F: x + y
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> x, y = polygens(ZZ,'x,y')
>>> H_triangle(Integer(1)+x*y).f()
F: x + y + 1
>>> _.parabolic()
F: x + y
```

vector()

Return the f-vector as a polynomial in one variable.

This is obtained by letting $y = x$.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import F_triangle
sage: x, y = polygen(ZZ,'x,y')
sage: ft = 2*x^2 + 2*x*y + y^2 + 3*x + 2*y + 1
```

(continues on next page)

(continued from previous page)

```
sage: F_triangle(ft).vector()
5*x^2 + 5*x + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import F_triangle
>>> x, y = polygen(ZZ,'x,y')
>>> ft = Integer(2)*x**Integer(2) + Integer(2)*x*y + y**Integer(2) +␣
↪Integer(3)*x + Integer(2)*y + Integer(1)
>>> F_triangle(ft).vector()
5*x^2 + 5*x + 1
```

class sage.combinat.triangles_FHM.**Gamma_triangle**(*poly*, *variables=None*)

Bases: *Triangle*

Class for the Gamma-triangles.

h()

Return the associated H-triangle.

The transition between Gamma-triangles and H-triangles is defined by

$$H(x,y) = (1+x)^d \sum_{0\leq i; 0\leq j\leq d-2i} \gamma_{i,j} \left(\frac{x}{(1+x)^2}\right)^i \left(\frac{1+xy}{1+x}\right)^j$$

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import Gamma_triangle
sage: x, y = polygen(ZZ,'x,y')
sage: g = y**2 + x
sage: Gamma_triangle(g).h()
H: x^2*y^2 + 2*x*y + x + 1

sage: a, b = polygen(ZZ, 'a, b')
sage: x, y = polygens(a.parent(),'x,y')
sage: g = Gamma_triangle(y**3+a*x*y+b*x,(x,y))
sage: hh = g.h()
sage: hh.gamma() == g
True
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import Gamma_triangle
>>> x, y = polygen(ZZ,'x,y')
>>> g = y**Integer(2) + x
>>> Gamma_triangle(g).h()
H: x^2*y^2 + 2*x*y + x + 1

>>> a, b = polygen(ZZ, 'a, b')
>>> x, y = polygens(a.parent(),'x,y')
>>> g = Gamma_triangle(y**Integer(3)+a*x*y+b*x,(x,y))
>>> hh = g.h()
>>> hh.gamma() == g
True
```

vector()

Return the gamma-vector as a polynomial in one variable.

This is obtained by letting $y = 1$.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import Gamma_triangle
sage: x, y = polygen(ZZ,'x,y')
sage: gt = y**2 + x
sage: Gamma_triangle(gt).vector()
x + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import Gamma_triangle
>>> x, y = polygen(ZZ,'x,y')
>>> gt = y**Integer(2) + x
>>> Gamma_triangle(gt).vector()
x + 1
```

class sage.combinat.triangles_FHM.**H_triangle**(*poly*, *variables=None*)

Bases: *Triangle*

Class for the H-triangles.

f()

Return the associated F-triangle.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygens(ZZ,'x,y')
sage: H_triangle(1+x*y).f()
F: x + y + 1
sage: H_triangle(x^2*y^2 + 2*x*y + x + 1).f()
F: 2*x^2 + 2*x*y + y^2 + 3*x + 2*y + 1
sage: flo = H_triangle(1+4*x+2*x**2+x*y*(4+8*x)+
....:   x**2*y**2*(6+4*x)+4*(x*y)**3+(x*y)**4).f(); flo
F: 7*x^4 + 12*x^3*y + 10*x^2*y^2 + 4*x*y^3 + y^4 + 20*x^3
+ 28*x^2*y + 16*x*y^2 + 4*y^3 + 20*x^2 + 20*x*y
+ 6*y^2 + 8*x + 4*y + 1
sage: flo(-1-x,-1-y) == flo
True
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> x, y = polygens(ZZ,'x,y')
>>> H_triangle(Integer(1)+x*y).f()
F: x + y + 1
>>> H_triangle(x**Integer(2)*y**Integer(2) + Integer(2)*x*y + x + Integer(1)).
→f()
F: 2*x^2 + 2*x*y + y^2 + 3*x + 2*y + 1
>>> flo = H_triangle(Integer(1)+Integer(4)*x+Integer(2)*x**Integer(2)+x*y*(Int
→eger(4)+Integer(8)*x)+
...   x**Integer(2)*y**Integer(2)*(Integer(6)+Integer(4)*x)+Inte-
```

(continues on next page)

(continued from previous page)

```
ger(4)*(x*y)**Integer(3)+(x*y)**Integer(4)).f(); flo
F: 7*x^4 + 12*x^3*y + 10*x^2*y^2 + 4*x*y^3 + y^4 + 20*x^3
+ 28*x^2*y + 16*x*y^2 + 4*y^3 + 20*x^2 + 20*x*y
+ 6*y^2 + 8*x + 4*y + 1
>>> flo(-Integer(1)-x,-Integer(1)-y) == flo
True
```

gamma()

Return the associated Gamma-triangle.

In some cases, this is a more condensed way to encode the same amount of information.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygen(ZZ,'x,y')
sage: ht = x**2*y**2 + 2*x*y + x + 1
sage: H_triangle(ht).gamma()
Γ: y^2 + x

sage: W = SymmetricGroup(5)                                               #␣
↪needs sage.groups
sage: P = posets.NoncrossingPartitions(W)                                 #␣
↪needs sage.graphs sage.groups
sage: P.M_triangle().h().gamma()                                          #␣
↪needs sage.graphs sage.groups
Γ: y^4 + 3*x*y^2 + 2*x^2 + 2*x*y + x
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> x, y = polygen(ZZ,'x,y')
>>> ht = x**Integer(2)*y**Integer(2) + Integer(2)*x*y + x + Integer(1)
>>> H_triangle(ht).gamma()
Γ: y^2 + x

>>> W = SymmetricGroup(Integer(5))                                          ␣
↪     # needs sage.groups
>>> P = posets.NoncrossingPartitions(W)                                   #␣
↪needs sage.graphs sage.groups
>>> P.M_triangle().h().gamma()                                            #␣
↪needs sage.graphs sage.groups
Γ: y^4 + 3*x*y^2 + 2*x^2 + 2*x*y + x
```

m()

Return the associated M-triangle.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: h = polygen(ZZ, 'h')
sage: x, y = polygens(h.parent(),'x,y')
sage: ht = H_triangle(x^2*y^2 + 2*x*y + 2*x*h - 4*x + 1, variables=[x,y])
sage: ht.m()
```

(continues on next page)

(continued from previous page)

```
M: x^2*y^2 + (-2*h + 2)*x*y^2 + (2*h - 2)*x*y
+ (2*h - 3)*y^2 + (-2*h + 2)*y + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> h = polygen(ZZ, 'h')
>>> x, y = polygens(h.parent(),'x,y')
>>> ht = H_triangle(x**Integer(2)*y**Integer(2) + Integer(2)*x*y +␣
↪Integer(2)*x*h - Integer(4)*x + Integer(1), variables=[x,y])
>>> ht.m()
M: x^2*y^2 + (-2*h + 2)*x*y^2 + (2*h - 2)*x*y
+ (2*h - 3)*y^2 + (-2*h + 2)*y + 1
```

transpose()

Return the transposed H-triangle.

OUTPUT: another H-triangle

This operation is an involution. When seen as a matrix, it performs a symmetry with respect to the northwest-southeast diagonal.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygens(ZZ,'x,y')
sage: H_triangle(1+x*y).transpose()
H: x*y + 1
sage: H_triangle(x^2*y^2 + 2*x*y + x + 1).transpose()
H: x^2*y^2 + x^2*y + 2*x*y + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> x, y = polygens(ZZ,'x,y')
>>> H_triangle(Integer(1)+x*y).transpose()
H: x*y + 1
>>> H_triangle(x**Integer(2)*y**Integer(2) + Integer(2)*x*y + x + Integer(1)).
↪transpose()
H: x^2*y^2 + x^2*y + 2*x*y + 1
```

vector()

Return the h-vector as a polynomial in one variable.

This is obtained by letting $y = 1$.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygen(ZZ,'x,y')
sage: ht = x**2*y**2 + 2*x*y + x + 1
sage: H_triangle(ht).vector()
x^2 + 3*x + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
```

(continues on next page)

(continued from previous page)

```
>>> x, y = polygen(ZZ,'x,y')
>>> ht = x**Integer(2)*y**Integer(2) + Integer(2)*x*y + x + Integer(1)
>>> H_triangle(ht).vector()
x^2 + 3*x + 1
```

class sage.combinat.triangles_FHM.**M_triangle**(*poly*, *variables=None*)

Bases: *Triangle*

Class for the M-triangles.

This is motivated by generating series of Möbius numbers of graded posets.

EXAMPLES:

```
sage: x, y = polygens(ZZ, 'x,y')
sage: P = Poset({2: [1]})                                                   #␣
↪needs sage.graphs
sage: P.M_triangle()                                                        #␣
↪needs sage.graphs
M: x*y - y + 1
```

```
>>> from sage.all import *
>>> x, y = polygens(ZZ, 'x,y')
>>> P = Poset({Integer(2): [Integer(1)]})                                  ␣
↪               # needs sage.graphs
>>> P.M_triangle()                                                        #␣
↪needs sage.graphs
M: x*y - y + 1
```

dual()

Return the dual M-triangle.

This is the M-triangle of the dual poset, hence an involution.

When seen as a matrix, this performs a symmetry with respect to the northwest-southeast diagonal.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import  M_triangle
sage: x, y = polygens(ZZ, 'x,y')
sage: mt = M_triangle(x*y - y + 1)
sage: mt.dual() == mt
True
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import  M_triangle
>>> x, y = polygens(ZZ, 'x,y')
>>> mt = M_triangle(x*y - y + Integer(1))
>>> mt.dual() == mt
True
```

f()

Return the associated F-triangle.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import M_triangle
sage: x, y = polygens(ZZ,'x,y')
sage: M_triangle(1-y+x*y).f()
F: x + y + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import M_triangle
>>> x, y = polygens(ZZ,'x,y')
>>> M_triangle(Integer(1)-y+x*y).f()
F: x + y + 1
```

h()

Return the associated H-triangle.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import M_triangle
sage: x, y = polygens(ZZ,'x,y')
sage: M_triangle(1-y+x*y).h()
H: x*y + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import M_triangle
>>> x, y = polygens(ZZ,'x,y')
>>> M_triangle(Integer(1)-y+x*y).h()
H: x*y + 1
```

transmute()

Return the image of `self` by an involution.

OUTPUT: another M-triangle

The involution is defined by converting to an H-triangle, transposing the matrix, and then converting back to an M-triangle.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import  M_triangle
sage: x, y = polygens(ZZ, 'x,y')
sage: nc3 = x^2*y^2 - 3*x*y^2 + 3*x*y + 2*y^2 - 3*y + 1
sage: m = M_triangle(nc3)
sage: m2 = m.transmute(); m2                                        #␣
↪needs sage.libs.flint
M: 2*x^2*y^2 - 3*x*y^2 + 2*x*y + y^2 - 2*y + 1
sage: m2.transmute() == m                                           #␣
↪needs sage.libs.flint
True
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import  M_triangle
>>> x, y = polygens(ZZ, 'x,y')
>>> nc3 = x**Integer(2)*y**Integer(2) - Integer(3)*x*y**Integer(2) +␣
↪Integer(3)*x*y + Integer(2)*y**Integer(2) - Integer(3)*y + Integer(1)
>>> m = M_triangle(nc3)
```

(continues on next page)

(continued from previous page)

```
>>> m2 = m.transmute(); m2                                                    #␣
↪needs sage.libs.flint
M: 2*x^2*y^2 - 3*x*y^2 + 2*x*y + y^2 - 2*y + 1
>>> m2.transmute() == m                                                       #␣
↪needs sage.libs.flint
True
```

class sage.combinat.triangles_FHM.**Triangle**(*poly*, *variables=None*)

Bases: `SageObject`

Common class for different kinds of triangles.

This serves as a base class for F-triangles, H-triangles, M-triangles and Gamma-triangles.

The user should use these subclasses directly.

The input is a polynomial in two variables. One can also give a polynomial with more variables and specify two chosen variables.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import Triangle
sage: x, y = polygens(ZZ, 'x,y')
sage: ht = Triangle(1+4*x+2*x*y)
sage: unicode_art(ht)                                                          #␣
↪needs sage.modules
⎛0 2⎞
⎝1 4⎠
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import Triangle
>>> x, y = polygens(ZZ, 'x,y')
>>> ht = Triangle(Integer(1)+Integer(4)*x+Integer(2)*x*y)
>>> unicode_art(ht)                                                          #␣
↪needs sage.modules
⎛0 2⎞
⎝1 4⎠
```

matrix()

Return the associated matrix for display.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygens(ZZ, 'x,y')
sage: h = H_triangle(1+2*x*y)
sage: h.matrix()                                                              #␣
↪needs sage.modules
[0 2]
[1 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> x, y = polygens(ZZ, 'x,y')
```

(continues on next page)

(continued from previous page)

```
>>> h = H_triangle(Integer(1)+Integer(2)*x*y)
>>> h.matrix()                                                          #␣
↪needs sage.modules
[0 2]
[1 0]
```

polynomial()

Return the triangle as a bare polynomial.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygens(ZZ, 'x,y')
sage: h = H_triangle(1+2*x*y)
sage: h.polynomial()
2*x*y + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> x, y = polygens(ZZ, 'x,y')
>>> h = H_triangle(Integer(1)+Integer(2)*x*y)
>>> h.polynomial()
2*x*y + 1
```

truncate(*d*)

Return the truncated triangle.

INPUT:

- d – integer

As a polynomial, this means that all monomials with a power of either x or y greater than or equal to d are dismissed.

EXAMPLES:

```
sage: from sage.combinat.triangles_FHM import H_triangle
sage: x, y = polygens(ZZ, 'x,y')
sage: h = H_triangle(1+2*x*y)
sage: h.truncate(2)
H: 2*x*y + 1
```

```
>>> from sage.all import *
>>> from sage.combinat.triangles_FHM import H_triangle
>>> x, y = polygens(ZZ, 'x,y')
>>> h = H_triangle(Integer(1)+Integer(2)*x*y)
>>> h.truncate(Integer(2))
H: 2*x*y + 1
```

5.1.352 Tuples

class sage.combinat.tuple.**Tuples**(*S*, *k*)

Bases: Parent, UniqueRepresentation

Return the enumerated set of ordered tuples of S of length k.

An ordered tuple of length k of set is an ordered selection with repetition and is represented by a list of length k containing elements of set.

EXAMPLES:

```
sage: S = [1,2]
sage: Tuples(S,3).list()
[(1, 1, 1), (2, 1, 1), (1, 2, 1), (2, 2, 1), (1, 1, 2),
 (2, 1, 2), (1, 2, 2), (2, 2, 2)]
sage: mset = ["s","t","e","i","n"]
sage: Tuples(mset,2).list()
[('s', 's'), ('t', 's'), ('e', 's'), ('i', 's'), ('n', 's'),
 ('s', 't'), ('t', 't'), ('e', 't'), ('i', 't'), ('n', 't'),
 ('s', 'e'), ('t', 'e'), ('e', 'e'), ('i', 'e'), ('n', 'e'),
 ('s', 'i'), ('t', 'i'), ('e', 'i'), ('i', 'i'), ('n', 'i'),
 ('s', 'n'), ('t', 'n'), ('e', 'n'), ('i', 'n'), ('n', 'n')]
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2)]
>>> Tuples(S,Integer(3)).list()
[(1, 1, 1), (2, 1, 1), (1, 2, 1), (2, 2, 1), (1, 1, 2),
 (2, 1, 2), (1, 2, 2), (2, 2, 2)]
>>> mset = ["s","t","e","i","n"]
>>> Tuples(mset,Integer(2)).list()
[('s', 's'), ('t', 's'), ('e', 's'), ('i', 's'), ('n', 's'),
 ('s', 't'), ('t', 't'), ('e', 't'), ('i', 't'), ('n', 't'),
 ('s', 'e'), ('t', 'e'), ('e', 'e'), ('i', 'e'), ('n', 'e'),
 ('s', 'i'), ('t', 'i'), ('e', 'i'), ('i', 'i'), ('n', 'i'),
 ('s', 'n'), ('t', 'n'), ('e', 'n'), ('i', 'n'), ('n', 'n')]
```

```
sage: K.<a> = GF(4, 'a')                                                    #␣
↪needs sage.rings.finite_rings
sage: mset = sorted((x for x in K if x != 0), key=str)                      #␣
↪needs sage.rings.finite_rings
sage: Tuples(mset, 2).list()                                                #␣
↪needs sage.rings.finite_rings
[(1, 1),     (a, 1),     (a + 1, 1),
 (1, a),     (a, a),     (a + 1, a),
 (1, a + 1), (a, a + 1), (a + 1, a + 1)]
```

```
>>> from sage.all import *
>>> K = GF(Integer(4), 'a', names=('a',)); (a,) = K._first_ngens(1)# needs sage.
↪rings.finite_rings
>>> mset = sorted((x for x in K if x != Integer(0)), key=str)               ␣
↪     # needs sage.rings.finite_rings
>>> Tuples(mset, Integer(2)).list()                                         ␣
↪     # needs sage.rings.finite_rings
[(1, 1),     (a, 1),     (a + 1, 1),
 (1, a),     (a, a),     (a + 1, a),
 (1, a + 1), (a, a + 1), (a + 1, a + 1)]
```

cardinality()

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: Tuples(S,2).cardinality()
25
sage: S = [1,1,2,3,4,5]
sage: Tuples(S,2).cardinality()
25
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]
>>> Tuples(S,Integer(2)).cardinality()
25
>>> S = [Integer(1),Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]
>>> Tuples(S,Integer(2)).cardinality()
25
```

unrank(i)

Return the i-th element of the set of Tuples.

INPUT:

- `i` – integer between 0 and $n - 1$, where n is the cardinality of this set

EXAMPLES:

```
sage: T = Tuples(range(7), 6)
sage: T[73451]
(0, 0, 1, 4, 2, 4)
```

```
>>> from sage.all import *
>>> T = Tuples(range(Integer(7)), Integer(6))
>>> T[Integer(73451)]
(0, 0, 1, 4, 2, 4)
```

sage.combinat.tuple.**Tuples_sk**

alias of `Tuples`

class sage.combinat.tuple.**UnorderedTuples**(S, k)

Bases: `Parent`, `UniqueRepresentation`

Return the enumerated set of unordered tuples of S of length k.

An unordered tuple of length k of set is a unordered selection with repetitions of set and is represented by a sorted list of length k containing elements from set.

EXAMPLES:

```
sage: S = [1,2]
sage: UnorderedTuples(S,3).list()
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
sage: UnorderedTuples(["a","b","c"],2).list()
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'),
 ('c', 'c')]
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2)]
>>> UnorderedTuples(S,Integer(3)).list()
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
>>> UnorderedTuples(["a","b","c"],Integer(2)).list()
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'),
 ('c', 'c')]
```

cardinality()

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: UnorderedTuples(S,2).cardinality()
15
```

```
>>> from sage.all import *
>>> S = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]
>>> UnorderedTuples(S,Integer(2)).cardinality()
15
```

sage.combinat.tuple.**UnorderedTuples_sk**

alias of *UnorderedTuples*

5.1.353 Introduction to combinatorics in Sage

This thematic tutorial is a translation by Hugh Thomas of the combinatorics chapter written by Nicolas M. Thiéry in the book "Calcul Mathématique avec Sage" [CMS2012]. It covers mainly the treatment in `Sage` of the following combinatorial problems: enumeration (how many elements are there in a set S?), listing (generate all the elements of S, or iterate through them), and random selection (choosing an element at random from a set S according to a given distribution, for example the uniform distribution). These questions arise naturally in the calculation of probabilities (what is the probability in poker of obtaining a straight or a four-of-a-kind of aces?), in statistical physics, and also in computer algebra (the number of elements in a finite field), or in the analysis of algorithms. Combinatorics covers a much wider domain (partial orders, representation theory, …) for which we only give a few pointers towards the possibilities offered by `Sage`.

Todo

Add link to some thematic tutorial on graphs

A characteristic of computational combinatorics is the profusion of types of objects and sets that one wants to manipulate. It would be impossible to describe them all or, a fortiori, to implement them all. After some *Initial examples*, this chapter illustrates the underlying method: supplying the basic building blocks to describe common combinatorial sets *Common enumerated sets*, tools for combining them to construct new examples *Constructions*, and generic algorithms for solving uniformly a large class of problems *Generic algorithms*.

This is a domain in which `Sage` has much more extensive capabilities than most computer algebra systems, and it is rapidly expanding; at the same time, it is still quite new, and has many unnecessary limitations and incoherences.

Initial examples

Poker and probability

We begin by solving a classic problem: enumerating certain combinations of cards in the game of poker, in order to deduce their probability.

A card in a poker deck is characterized by a suit (hearts, diamonds, spades, or clubs) and a value (2, 3, …, 10, jack, queen, king, ace). The game is played with a full deck, which consists of the Cartesian product of the set of suits and the set of values:

$$\text{Cards} = \text{Suits} \times \text{Values} = \{(s, v) \mid s \in \text{Suits} \text{ et } v \in \text{Values}\}.$$

We construct these examples in `Sage`:

```
sage: Suits = Set(["Hearts", "Diamonds", "Spades", "Clubs"])
sage: Values = Set([2, 3, 4, 5, 6, 7, 8, 9, 10,
....:               "Jack", "Queen", "King", "Ace"])
sage: Cards = cartesian_product([Values, Suits])
```

```
>>> from sage.all import *
>>> Suits = Set(["Hearts", "Diamonds", "Spades", "Clubs"])
>>> Values = Set([Integer(2), Integer(3), Integer(4), Integer(5), Integer(6),
↪Integer(7), Integer(8), Integer(9), Integer(10),
...               "Jack", "Queen", "King", "Ace"])
>>> Cards = cartesian_product([Values, Suits])
```

There are 4 suits and 13 possible values, and therefore $4 \times 13 = 52$ cards in the poker deck:

```
sage: Suits.cardinality()
4
sage: Values.cardinality()
13
sage: Cards.cardinality()
52
```

```
>>> from sage.all import *
>>> Suits.cardinality()
4
>>> Values.cardinality()
13
>>> Cards.cardinality()
52
```

Draw a card at random:

```
sage: Cards.random_element()                          # random
(6, 'Clubs')
```

```
>>> from sage.all import *
>>> Cards.random_element()                          # random
(6, 'Clubs')
```

Now we can define a set of cards:

```
sage: Set([Cards.random_element(), Cards.random_element()])  # random
{(2, 'Hearts'), (4, 'Spades')}
```

```
>>> from sage.all import *
>>> Set([Cards.random_element(), Cards.random_element()])  # random
{(2, 'Hearts'), (4, 'Spades')}
```

This problem should eventually disappear: it is planned to change the implementation of Cartesian products so that their elements are immutable by default.

Returning to our main topic, we will be considering a simplified version of poker, in which each player directly draws five cards, which form his *hand*. The cards are all distinct and the order in which they are drawn is irrelevant; a hand is therefore a subset of size 5 of the set of cards. To draw a hand at random, we first construct the set of all possible hands, and then we ask for a randomly chosen element:

```
sage: Hands = Subsets(Cards, 5)
sage: Hands.random_element()                        # random
{(4, 'Hearts', 4), (9, 'Diamonds'), (8, 'Spades'),
 (9, 'Clubs'), (7, 'Hearts')}
```

```
>>> from sage.all import *
>>> Hands = Subsets(Cards, Integer(5))
>>> Hands.random_element()                        # random
{(4, 'Hearts', 4), (9, 'Diamonds'), (8, 'Spades'),
 (9, 'Clubs'), (7, 'Hearts')}
```

The total number of hands is given by the number of subsets of size 5 of a set of size 52, which is given by the binomial coefficient $\binom{52}{5}$:

```
sage: binomial(52,5)
2598960
```

```
>>> from sage.all import *
>>> binomial(Integer(52),Integer(5))
2598960
```

One can also ignore the method of calculation, and simply ask for the size of the set of hands:

```
sage: Hands.cardinality()
2598960
```

```
>>> from sage.all import *
>>> Hands.cardinality()
2598960
```

The strength of a poker hand depends on the particular combination of cards present. One such combination is the *flush*; this is a hand all of whose cards have the same suit. (In principle, straight flushes should be excluded; this will be the goal of an exercise given below.) Such a hand is therefore characterized by the choice of five values from among the thirteen possibilities, and the choice of one of four suits. We will construct the set of all flushes, so as to determine how many there are:

```
sage: Flushes = cartesian_product([Subsets(Values, 5), Suits])
sage: Flushes.cardinality()
5148
```

```
>>> from sage.all import *
>>> Flushes = cartesian_product([Subsets(Values, Integer(5)), Suits])
>>> Flushes.cardinality()
5148
```

The probability of obtaining a flush when drawing a hand at random is therefore:

```
sage: Flushes.cardinality()  / Hands.cardinality()
33/16660
```

```
>>> from sage.all import *
>>> Flushes.cardinality()  / Hands.cardinality()
33/16660
```

or about two in a thousand:

```
sage: 1000.0 * Flushes.cardinality()  / Hands.cardinality()
1.98079231692677
```

```
>>> from sage.all import *
>>> RealNumber('1000.0') * Flushes.cardinality()  / Hands.cardinality()
1.98079231692677
```

We will now attempt a little numerical simulation. The following function tests whether a given hand is a flush or not:

```
sage: def is_flush(hand):
....:     return len(set(suit for (val, suit) in hand)) == 1
```

```
>>> from sage.all import *
>>> def is_flush(hand):
...     return len(set(suit for (val, suit) in hand)) == Integer(1)
```

We now draw 10000 hands at random, and count the number of flushes obtained (this takes about 10 seconds):

```
sage: n = 10000
sage: nflush = 0
sage: for i in range(n):                        # long time
....:    hand = Hands.random_element()
....:    if is_flush(hand):
....:        nflush += 1
sage: n, nflush                                 # random
(10000, 18)
```

```
>>> from sage.all import *
>>> n = Integer(10000)
>>> nflush = Integer(0)
>>> for i in range(n):                          # long time
...    hand = Hands.random_element()
...    if is_flush(hand):
...        nflush += Integer(1)
>>> n, nflush                                   # random
(10000, 18)
```

Exercises

A hand containing four cards of the same value is called a *four of a kind*. Construct the set of four of a kind hands (Hint: use `Arrangements` to choose a pair of distinct values at random, then choose a suit for the first value). Calculate the number of four of a kind hand, list them, and then determine the probability of obtaining a four of a kind when drawing a hand at random.

A hand all of whose cards have the same suit, and whose values are consecutive, is called a *straight flush* rather than a *flush*. Count the number of straight flushes, and then deduce the correct probability of obtaining a flush when drawing a hand at random.

Calculate the probability of each of the poker hands (see Wikipedia article Poker_hands), and compare them with the results of simulations.

Enumeration of trees using generating functions

In this section, we discuss the example of complete binary trees, and illustrate in this context many techniques of enumeration in which formal power series play a natural role. These techniques are quite general, and can be applied whenever the combinatorial objects in question admit a recursive definition (grammar) (see *Species, decomposable combinatorial classes* for an automated treatment). The goal is not a formal presentation of these methods; the calculations are rigorous, but most of the justifications will be skipped.

A *complete binary tree* is either a leaf L, or a node to which two complete binary trees are attached (see *Figure: The five complete binary trees with four leaves*).

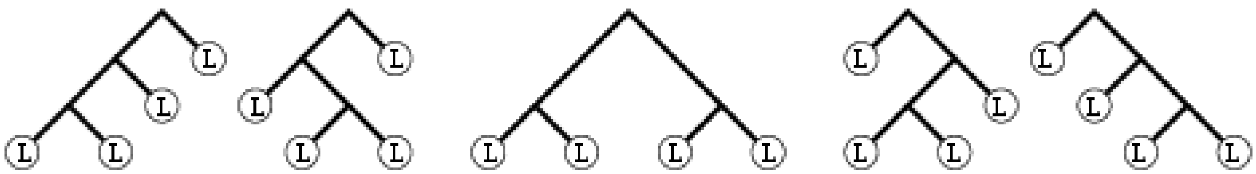


Fig. 1: Figure: The five complete binary trees with four leaves

Exercise: enumeration of binary trees

Find by hand all the complete binary trees with $n = 1, 2, 3, 4, 5$ leaves (see *Exercise: complete binary tree iterator* to find them using `Sage`).

Our goal is to determine the number c_n of complete binary trees with n leaves (in this section, except when explicitly stated otherwise, "trees" always means complete binary trees). This is a typical situation in which one is not only interested in a single set, but in a family of sets, typically parameterized by $n \in \mathbf{N}$.

According to the solution of *Exercise: enumeration of binary trees*, the first terms are given by $c_1, \dots, c_5 = 1, 1, 2, 5, 14$. The simple fact of knowing these few numbers is already very valuable. In fact, this permits research in a gold mine of information: the Online Encyclopedia of Integer Sequences, commonly called "Sloane", the name of its principal author, which contains more than 190000 sequences of integers:

```
sage: oeis([1,1,2,5,14])                            # optional -- internet
0: A000108: Catalan numbers: ...
1: ...
2: ...
```

```
>>> from sage.all import *
>>> oeis([Integer(1),Integer(1),Integer(2),Integer(5),Integer(14)])                                      
↪          # optional -- internet
0: A000108: Catalan numbers: ...
1: ...
2: ...
```

The result suggests that the trees are counted by one of the most famous sequences, the Catalan numbers. Looking through the references supplied by the Encyclopedia, we see that this is really the case: the few numbers above form a digital fingerprint of our objects, which enable us to find, in a few seconds, a precise result from within an abundant literature.

Our next goal is to recover this result using Sage. Let C_n be the set of trees with n leaves, so that $c_n = |C_n|$; by convention, we will define $C_0 = \emptyset$ and $c_0 = 0$. The set of all trees is then the disjoint union of the sets C_n:

$$C = \biguplus_{n\in\mathbb{N}} C_n \,.$$

Having named the set C of all trees, we can translate the recursive definition of trees into a set-theoretic equation:

$$C \quad \approx \quad \{\mathrm{L}\} \quad \uplus \quad C\times C\,.$$

In words: a tree t (which is by definition in C) is either a leaf (so in $\{\mathrm{L}\}$) or a node to which two trees t_1 and t_2 have been attached, and which we can therefore identify with the pair (t_1, t_2) (in the Cartesian product $C \times C$).

The founding idea of algebraic combinatorics, introduced by Euler in a letter to Goldbach of 1751 to treat a similar problem , is to manipulate all the numbers c_n simultaneously, by encoding them as coefficients in a formal power series, called the *generating function* of the c_n's:

$$C(z) = \sum_{n\in\mathbb{N}} c_n z^n \,,$$

where z is a formal variable (which means that we do not have to worry about questions of convergence). The beauty of this idea is that set-theoretic operations ($A \uplus B$, $A\times B$) translate naturally into algebraic operations on the corresponding series ($A(z) + B(z)$, $A(z) \cdot B(z)$), in such a way that the set-theoretic equation satisfied by C can be translated directly into an algebraic equation satisfied by $C(z)$:

$$C(z) = z + C(z)\cdot C(z)\,.$$

Now we can solve this equation with Sage. In order to do so, we introduce two variables, C and z, and we define the equation:

```
sage: C, z = var('C,z')                                                      #␣
↪needs sage.symbolic
sage: sys = [ C == z + C*C ]                                                 #␣
↪needs sage.symbolic
```

```
>>> from sage.all import *
>>> C, z = var('C,z')                                                      #␣
↪needs sage.symbolic
>>> sys = [ C == z + C*C ]                                                 #␣
↪needs sage.symbolic
```

There are two solutions, which happen to have closed forms:

```
sage: sol = solve(sys, C, solution_dict=True); sol                           #␣
↪needs sage.symbolic
[{C: -1/2*sqrt(-4*z + 1) + 1/2}, {C: 1/2*sqrt(-4*z + 1) + 1/2}]
sage: s0 = sol[0][C]; s1 = sol[1][C]                                         #␣
↪needs sage.symbolic
```

```
>>> from sage.all import *
>>> sol = solve(sys, C, solution_dict=True); sol                           #␣
↪needs sage.symbolic
[{C: -1/2*sqrt(-4*z + 1) + 1/2}, {C: 1/2*sqrt(-4*z + 1) + 1/2}]
>>> s0 = sol[Integer(0)][C]; s1 = sol[Integer(1)][C]                         ␣
↪              # needs sage.symbolic
```

and whose Taylor series begin as follows:

```
sage: s0.series(z, 6)                                                    #␣
↪needs sage.symbolic
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + Order(z^6)
sage: s1.series(z, 6)                                                    #␣
↪needs sage.symbolic
1 + (-1)*z + (-1)*z^2 + (-2)*z^3 + (-5)*z^4 + (-14)*z^5
+ Order(z^6)
```

```
>>> from sage.all import *
>>> s0.series(z, Integer(6))                                              ␣
↪     # needs sage.symbolic
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + Order(z^6)
>>> s1.series(z, Integer(6))                                              ␣
↪     # needs sage.symbolic
1 + (-1)*z + (-1)*z^2 + (-2)*z^3 + (-5)*z^4 + (-14)*z^5
+ Order(z^6)
```

The second solution is clearly aberrant, while the first one gives the expected coefficients. Therefore, we set:

```
sage: C = s0                                                             #␣
↪needs sage.symbolic
```

```
>>> from sage.all import *
>>> C = s0                                                             #␣
↪needs sage.symbolic
```

We can now calculate the next terms:

```
sage: C.series(z, 11)                                                    #␣
↪needs sage.symbolic
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 +
132*z^7 + 429*z^8 + 1430*z^9 + 4862*z^10 + Order(z^11)
```

```
>>> from sage.all import *
>>> C.series(z, Integer(11))                                              ␣
↪     # needs sage.symbolic
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 +
132*z^7 + 429*z^8 + 1430*z^9 + 4862*z^10 + Order(z^11)
```

or calculate, more or less instantaneously, the 100-th coefficient:

```
sage: C.series(z, 101).coefficient(z,100)                                #␣
↪needs sage.symbolic
227508830794229349661819540395688853956041682601541047340
```

```
>>> from sage.all import *
>>> C.series(z, Integer(101)).coefficient(z,Integer(100))                 ␣
↪              # needs sage.symbolic
227508830794229349661819540395688853956041682601541047340
```

It is unfortunate to have to recalculate everything if at some point we wanted the 101-st coefficient. Lazy power series (see `sage.rings.lazy_series_ring`) come into their own here, in that one can define them from a system of equations without solving it, and, in particular, without needing a closed form for the answer. We begin by defining the ring of lazy power series:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
```

```
>>> from sage.all import *
>>> L = LazyPowerSeriesRing(QQ, names=('z',)); (z,) = L._first_ngens(1)
```

Then we create a "free" power series, which we name, and which we then define by a recursive equation:

```
sage: C = L.undefined(valuation=1)
sage: C.define(z + C * C)
```

```
>>> from sage.all import *
>>> C = L.undefined(valuation=Integer(1))
>>> C.define(z + C * C)
```

```
sage: [C.coefficient(i) for i in range(11)]
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> from sage.all import *
>>> [C.coefficient(i) for i in range(Integer(11))]
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

At any point, one can ask for any coefficient without having to redefine C:

```
sage: C.coefficient(100)
227508830794229349661819540395688853956041682601541047340
sage: C.coefficient(200)
129013158064429114001222907669676675134349530552728882499810851598901419013348319045534580850847735528275750122188940
```

```
>>> from sage.all import *
>>> C.coefficient(Integer(100))
227508830794229349661819540395688853956041682601541047340
>>> C.coefficient(Integer(200))
129013158064429114001222907669676675134349530552728882499810851598901419013348319045534580850847735528275750122188940
```

We now return to the closed form of $C(z)$:

```
sage: z = var('z')                                                          # needs sage.symbolic
sage: C = s0; C                                                             # needs sage.symbolic
-1/2*sqrt(-4*z + 1) + 1/2
```

```
>>> from sage.all import *
>>> z = var('z')                                                          # needs sage.symbolic
>>> C = s0; C                                                             # needs sage.symbolic
-1/2*sqrt(-4*z + 1) + 1/2
```

The n-th coefficient in the Taylor series for $C(z)$ being given by $\frac{1}{n!}C(z)^{(n)}(0)$, we look at the successive derivatives $C(z)^{(n)}(z)$:

```
sage: derivative(C, z, 1)                                                   #␣
↪needs sage.symbolic
1/sqrt(-4*z + 1)
sage: derivative(C, z, 2)                                                   #␣
↪needs sage.symbolic
2/(-4*z + 1)^(3/2)
sage: derivative(C, z, 3)                                                   #␣
↪needs sage.symbolic
12/(-4*z + 1)^(5/2)
```

```
>>> from sage.all import *
>>> derivative(C, z, Integer(1))                                            ␣
↪     # needs sage.symbolic
1/sqrt(-4*z + 1)
>>> derivative(C, z, Integer(2))                                            ␣
↪     # needs sage.symbolic
2/(-4*z + 1)^(3/2)
>>> derivative(C, z, Integer(3))                                            ␣
↪     # needs sage.symbolic
12/(-4*z + 1)^(5/2)
```

This suggests the existence of a simple explicit formula, which we will now seek. The following small function returns $d_n = n!\, c_n$:

```
sage: def d(n): return derivative(s0, n).subs(z=0)
```

```
>>> from sage.all import *
>>> def d(n): return derivative(s0, n).subs(z=Integer(0))
```

Taking successive quotients:

```
sage: [ (d(n+1) / d(n)) for n in range(1,17) ]                              #␣
↪needs sage.symbolic
[2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62]
```

```
>>> from sage.all import *
>>> [ (d(n+Integer(1)) / d(n)) for n in range(Integer(1),Integer(17)) ]     ␣
↪                         # needs sage.symbolic
[2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62]
```

we observe that d_n satisfies the recurrence relation $d_{n+1} = (4n-2)d_n$, from which we deduce that c_n satisfies the recurrence relation $c_{n+1} = \frac{(4n-2)}{n+1}c_n$. Simplifying, we find that c_n is the $(n-1)$-th Catalan number:

$$c_n = \mathrm{Catalan}(n-1) = \frac{1}{n}\binom{2(n-1)}{n-1}.$$

We check this:

```
sage: n = var('n')                                                          #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
sage: c = 1/n*binomial(2*(n-1),n-1)                                          #␣
↪needs sage.symbolic
sage: [c.subs(n=k) for k in range(1, 11)]                                    #␣
↪needs sage.symbolic
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
sage: [catalan_number(k-1) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> from sage.all import *
>>> n = var('n')                                                             #␣
↪needs sage.symbolic
>>> c = Integer(1)/n*binomial(Integer(2)*(n-Integer(1)),n-Integer(1))         ␣
↪                                   # needs sage.symbolic
>>> [c.subs(n=k) for k in range(Integer(1), Integer(11))]                     ␣
↪                 # needs sage.symbolic
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
>>> [catalan_number(k-Integer(1)) for k in range(Integer(1), Integer(11))]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

We can now calculate coefficients much further; here we calculate c_{100000} which has more than 60000 digits:

```
sage: cc = c(n=100000)                                                       #␣
↪needs sage.symbolic
```

```
>>> from sage.all import *
>>> cc = c(n=Integer(100000))                                                 ␣
↪     # needs sage.symbolic
```

This takes a couple of seconds:

```
sage: %time cc = c(100000)                   # not tested                    #␣
↪needs sage.symbolic
CPU times: user 2.34 s, sys: 0.00 s, total: 2.34 s
Wall time: 2.34 s
sage: ZZ(cc).ndigits()                                                       #␣
↪needs sage.symbolic
60198
```

```
>>> from sage.all import *
>>> %time cc = c(Integer(100000))                   # not tested              ␣
↪     # needs sage.symbolic
CPU times: user 2.34 s, sys: 0.00 s, total: 2.34 s
Wall time: 2.34 s
>>> ZZ(cc).ndigits()                                                         #␣
↪needs sage.symbolic
60198
```

The methods which we have used generalize to all recursively defined objects: the system of set-theoretic equations can be translated into a system of equations on the generating function, which enables the recursive calculation of its coefficients. If the set-theoretic equations are simple enough (for example, if they only involve Cartesian products and disjoint unions), the equation for $C(z)$ is algebraic. This equation has, in general, no closed-form solution. However, using *confinement*, one can deduce a *linear* differential equation which $C(z)$ satisfies. This differential equation, in turn, can be translated into a recurrence relation of fixed length on its coefficients c_n. In this case, the series is called *D-finite*. After the initial calculation of this recurrence relation, the calculation of coefficients is very fast. All these steps are purely algorithmic, and it is planned to port into `Sage` the implementations which exist in `Maple` (the `gfun` and `combstruct` packages) or `MuPAD-Combinat` (the `decomposableObjects` library).

For the moment, we illustrate this general procedure in the case of complete binary trees. The generating function $C(z)$ is a solution to an algebraic equation $P(z, C(z)) = 0$, where $P = P(x, y)$ is a polynomial with coefficients in **Q**. In the present case, $P = y^2 - y + x$. We formally differentiate this equation with respect to z:

```
sage: # needs sage.symbolic
sage: x, y, z = var('x, y, z')
sage: P = function('P')(x, y)
sage: C = function('C')(z)
sage: equation =  P(x=z, y=C) == 0
sage: diff(equation, z)
diff(C(z), z)*D[1](P)(z, C(z)) + D[0](P)(z, C(z)) == 0
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> x, y, z = var('x, y, z')
>>> P = function('P')(x, y)
>>> C = function('C')(z)
>>> equation =  P(x=z, y=C) == Integer(0)
>>> diff(equation, z)
diff(C(z), z)*D[1](P)(z, C(z)) + D[0](P)(z, C(z)) == 0
```

or, in a more readable format,

$$\frac{dC(z)}{dz}\frac{\partial P}{\partial y}(z, C(z)) + \frac{\partial P}{\partial x}(z, C(z)) = 0$$

From this we deduce:

$$\frac{dC(z)}{dz} = -\frac{\frac{\partial P}{\partial x}}{\frac{\partial P}{\partial y}}(z, C(z)).$$

In the case of complete binary trees, this gives:

```
sage: P = y^2 - y + x                                                        #␣
↪needs sage.symbolic
sage: Px = diff(P, x); Py = diff(P, y)                                       #␣
↪needs sage.symbolic
sage: - Px / Py                                                              #␣
↪needs sage.symbolic
-1/(2*y - 1)
```

```
>>> from sage.all import *
>>> P = y**Integer(2) - y + x                                                 ␣
↪       # needs sage.symbolic
>>> Px = diff(P, x); Py = diff(P, y)                                        #␣
↪needs sage.symbolic
>>> - Px / Py                                                               #␣
↪needs sage.symbolic
-1/(2*y - 1)
```

Recall that $P(z, C(z)) = 0$. Thus, we can calculate this fraction mod P and, in this way, express the derivative of $C(z)$ as a *polynomial in* $C(z)$ *with coefficients in* $\mathbf{Q}(z)$. In order to achieve this, we construct the quotient ring $R = \mathbf{Q}(x)[y]/(P)$:

```
sage: Qx = QQ['x'].fraction_field()
sage: Qxy = Qx['y']
sage: R = Qxy.quo(P); R                                                     #␣
↪needs sage.symbolic
Univariate Quotient Polynomial Ring in ybar
over Fraction Field of Univariate Polynomial Ring in x
over Rational Field with modulus y^2 - y + x
```

```
>>> from sage.all import *
>>> Qx = QQ['x'].fraction_field()
>>> Qxy = Qx['y']
>>> R = Qxy.quo(P); R                                                     #␣
↪needs sage.symbolic
Univariate Quotient Polynomial Ring in ybar
over Fraction Field of Univariate Polynomial Ring in x
over Rational Field with modulus y^2 - y + x
```

Note: `ybar` is the name of the variable y in the quotient ring.

Todo

add link to some tutorial on quotient rings

We continue the calculation of this fraction in R:

```
sage: fraction = - R(Px) / R(Py); fraction                                  #␣
↪needs sage.symbolic
(1/2/(x - 1/4))*ybar - 1/4/(x - 1/4)
```

```
>>> from sage.all import *
>>> fraction = - R(Px) / R(Py); fraction                                  #␣
↪needs sage.symbolic
(1/2/(x - 1/4))*ybar - 1/4/(x - 1/4)
```

Note

The following variant does not work yet:

```
sage: fraction = R( - Px / Py ); fraction    # todo: not implemented
Traceback (most recent call last):
...
TypeError: denominator must be a unit
```

```
>>> from sage.all import *
>>> fraction = R( - Px / Py ); fraction    # todo: not implemented
Traceback (most recent call last):
...
TypeError: denominator must be a unit
```

We lift the result to $\mathbf{Q}(x)[y]$ and then substitute z and $C(z)$ to obtain an expression for $\frac{d}{dz}C(z)$:

```
sage: fraction = fraction.lift(); fraction                                    #␣
↪needs sage.symbolic
(1/2/(x - 1/4))*y - 1/4/(x - 1/4)
sage: fraction(x=z, y=C)                                                      #␣
↪needs sage.symbolic
2*C(z)/(4*z - 1) - 1/(4*z - 1)
```

```
>>> from sage.all import *
>>> fraction = fraction.lift(); fraction                                    #␣
↪needs sage.symbolic
(1/2/(x - 1/4))*y - 1/4/(x - 1/4)
>>> fraction(x=z, y=C)                                                      #␣
↪needs sage.symbolic
2*C(z)/(4*z - 1) - 1/(4*z - 1)
```

or, more legibly,

$$\frac{\partial C(z)}{\partial z} = \frac{1}{1-4z} - \frac{2}{1-4z}C(z)\,.$$

In this simple case, we can directly deduce from this expression a linear differential equation with coefficients in $\mathbf{Q}[z]$:

```
sage: # needs sage.symbolic
sage: equadiff = diff(C,z) == fraction(x=z, y=C)
sage: equadiff
diff(C(z), z) == 2*C(z)/(4*z - 1) - 1/(4*z - 1)
sage: equadiff = equadiff.simplify_rational()
sage: equadiff = equadiff * equadiff.rhs().denominator()
sage: equadiff = equadiff - equadiff.rhs()
sage: equadiff
(4*z - 1)*diff(C(z), z) - 2*C(z) + 1 == 0
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> equadiff = diff(C,z) == fraction(x=z, y=C)
>>> equadiff
diff(C(z), z) == 2*C(z)/(4*z - 1) - 1/(4*z - 1)
>>> equadiff = equadiff.simplify_rational()
>>> equadiff = equadiff * equadiff.rhs().denominator()
>>> equadiff = equadiff - equadiff.rhs()
>>> equadiff
(4*z - 1)*diff(C(z), z) - 2*C(z) + 1 == 0
```

or, more legibly,

$$(1-4z)\frac{\partial C(z)}{\partial z} + 2C(z) - 1 = 0\,.$$

It is trivial to verify this equation on the closed form:

```
sage: Cf = sage.symbolic.function_factory.function('C')                      #␣
↪needs sage.symbolic
sage: equadiff.substitute_function(Cf, s0.function(z))                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
(4*z - 1)/sqrt(-4*z + 1) + sqrt(-4*z + 1) == 0
sage: bool(equadiff.substitute_function(Cf, s0.function(z)))                    #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> Cf = sage.symbolic.function_factory.function('C')                            #␣
↪needs sage.symbolic
>>> equadiff.substitute_function(Cf, s0.function(z))                             #␣
↪needs sage.symbolic
(4*z - 1)/sqrt(-4*z + 1) + sqrt(-4*z + 1) == 0
>>> bool(equadiff.substitute_function(Cf, s0.function(z)))                       #␣
↪needs sage.symbolic
True
```

In the general case, one continues to calculate successive derivatives of $C(z)$. These derivatives are *confined* in the quotient ring $\mathbf{Q}(z)[C]/(P)$ which is of finite dimension $\deg P$ over $\mathbf{Q}(z)$. Therefore, one will eventually find a linear relation among the first $\deg P$ derivatives of $C(z)$. Putting it over a single denominator, we obtain a linear differential equation of degree $\leq \deg P$ with coefficients in $\mathbf{Q}[z]$. By extracting the coefficient of z^n in the differential equation, we obtain the desired recurrence relation on the coefficients; in this case we recover the relation we had already found, based on the closed form:

$$c_{n+1} = \frac{(4n-2)}{n+1} c_n$$

After fixing the correct initial conditions, it becomes possible to calculate the coefficients of $C(z)$ recursively:

```
sage: def C(n): return 1 if n <= 1 else (4*n-6)/n * C(n-1)
sage: [ C(i) for i in range(10) ]
[1, 1, 1, 2, 5, 14, 42, 132, 429, 1430]
```

```
>>> from sage.all import *
>>> def C(n): return Integer(1) if n <= Integer(1) else (Integer(4)*n-Integer(6))/n *␣
↪C(n-Integer(1))
>>> [ C(i) for i in range(Integer(10)) ]
[1, 1, 1, 2, 5, 14, 42, 132, 429, 1430]
```

If n is too large for the explicit calculation of c_n, a sequence asymptotically equivalent to the sequence of coefficients c_n may be sought. Here again, there are generic techniques. The central tool is complex analysis, specifically, the study of the generating function around its singularities. In the present instance, the singularity is at $z_0 = 1/4$ and one would obtain $c_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$.

Summary

We see here a general phenomenon of computer algebra: the best *data structure* to describe a complicated mathematical object (a real number, a sequence, a formal power series, a function, a set) is often an equation defining the object (or a system of equations, typically with some initial conditions). Attempting to find a closed-form solution to this equation is not necessarily of interest: on the one hand, such a closed form rarely exists (e.g., the problem of solving a polynomial by radicals), and on the other hand, the equation, in itself, contains all the necessary information to calculate algorithmically the properties of the object under consideration (e.g., a numerical approximation, the initial terms or elements, an asymptotic equivalent), or to calculate with the object itself (e.g., performing arithmetic on power series). Therefore, instead of solving the equation, we look for the equation describing the object which is best suited to the problem we want to solve.

As we saw in our example, confinement (for example, in a finite dimensional vector space) is a fundamental tool for studying such equations. This notion of confinement is widely applicable in elimination techniques (linear algebra, Gröbner bases, and their algebro-differential generalizations). The same tool is central in algorithms for automatic summation and automatic verification of identities (Gosper's algorithm, Zeilberger's algorithm, and their generalizations; see also *Exercise: alternating sign matrices*).

Todo

add link to some tutorial on summation

All these techniques and their many generalizations are at the heart of very active topics of research: automatic combinatorics and analytic combinatorics, with major applications in the analysis of algorithms. It is likely, and desirable, that they will be progressively implemented in `Sage`.

Common enumerated sets

First example: the subsets of a set

Fix a set E of size n and consider the subsets of E of size k. We know that these subsets are counted by the binomial coefficients $\binom{n}{k}$. We can therefore calculate the number of subsets of size $k = 2$ of $E = \{1, 2, 3, 4\}$ with the function `binomial`:

```
sage: binomial(4, 2)
6
```

```
>>> from sage.all import *
>>> binomial(Integer(4), Integer(2))
6
```

Alternatively, we can *construct* the set $\mathcal{P}_2(E)$ of all the subsets of size 2 of E, then ask its cardinality:

```
sage: S = Subsets([1,2,3,4], 2)
sage: S.cardinality()
6
```

```
>>> from sage.all import *
>>> S = Subsets([Integer(1),Integer(2),Integer(3),Integer(4)], Integer(2))
>>> S.cardinality()
6
```

Once `S` has been constructed, we can also obtain the list of its elements:

```
sage: S.list()
[{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
```

```
>>> from sage.all import *
>>> S.list()
[{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
```

or select an element at random:

```
sage: S.random_element()                  # random
{1, 4}
```

```
>>> from sage.all import *
>>> S.random_element()                    # random
{1, 4}
```

More precisely, the object `S` models the set $\mathcal{P}_2(E)$ equipped with a fixed order (here, lexicographic order). It is therefore possible to ask for its 5-th element, keeping in mind that, as with `Python` lists, the first element is numbered zero:

```
sage: S.unrank(4)
{2, 4}
```

```
>>> from sage.all import *
>>> S.unrank(Integer(4))
{2, 4}
```

As a shortcut, in this setting, one can also use the notation:

```
sage: S[4]
{2, 4}
```

```
>>> from sage.all import *
>>> S[Integer(4)]
{2, 4}
```

but this should be used with care because some sets have a natural indexing other than by $(0, 1, \dots)$.

Conversely, one can calculate the position of an object in this order:

```
sage: s = S([2,4]); s
{2, 4}
sage: S.rank(s)
4
```

```
>>> from sage.all import *
>>> s = S([Integer(2),Integer(4)]); s
{2, 4}
>>> S.rank(s)
4
```

Note that `S` is *not* the list of its elements. One can, for example, model the set $\mathcal{P}(\mathcal{P}(\mathcal{P}(E)))$ and calculate its cardinality ($2^{2^{2^4}}$):

```
sage: E = Set([1,2,3,4])
sage: S = Subsets(Subsets(Subsets(E))); S
Subsets of Subsets of Subsets of {1, 2, 3, 4}
sage: n = S.cardinality(); n
2003529930406846464979072351560255750447825475569751419265016973...
```

```
>>> from sage.all import *
>>> E = Set([Integer(1),Integer(2),Integer(3),Integer(4)])
>>> S = Subsets(Subsets(Subsets(E))); S
Subsets of Subsets of Subsets of {1, 2, 3, 4}
>>> n = S.cardinality(); n
2003529930406846464979072351560255750447825475569751419265016973...
```

which is roughly $2 \cdot 10^{19728}$:

```
sage: n.ndigits()
19729
```

```
>>> from sage.all import *
>>> n.ndigits()
19729
```

or ask for its 237102124-th element:

```
sage: S.unrank(237102123) # random print output
{{{2, 4}, {1, 4}, {}, {1, 3, 4}, {1, 2, 4}, {4}, {2, 3}, {1, 3}, {2}},
  {{1, 3}, {2, 4}, {1, 2, 4}, {}, {3, 4}}}
```

```
>>> from sage.all import *
>>> S.unrank(Integer(237102123)) # random print output
{{{2, 4}, {1, 4}, {}, {1, 3, 4}, {1, 2, 4}, {4}, {2, 3}, {1, 3}, {2}},
  {{1, 3}, {2, 4}, {1, 2, 4}, {}, {3, 4}}}
```

It would be physically impossible to construct explicitly all the elements of S, as there are many more of them than there are particles in the universe (estimated at 10^{82}).

Remark: it would be natural in `Python` to use `len(S)` to ask for the cardinality of `S`. This is not possible because `Python` requires that the result of `len` be an integer of type `int`; this could cause overflows, and would not permit the return of {Infinity} for infinite sets:

```
sage: len(S)
Traceback (most recent call last):
...
OverflowError: cannot fit 'int' into an index-sized integer
```

```
>>> from sage.all import *
>>> len(S)
Traceback (most recent call last):
...
OverflowError: cannot fit 'int' into an index-sized integer
```

Partitions of integers

We now consider another classic problem: given a positive integer n, in how many ways can it be written in the form of a sum $n = i_1 + i_2 + \cdots + i_\ell$, where $i_1, \ldots, i_\ell$ are positive integers? There are two cases to distinguish:

- the order of the elements in the sum is not important, in which case we call $(i_1, \ldots, i_\ell)$ a *partition* of n;
- the order of the elements in the sum is important, in which case we call $(i_1, \ldots, i_\ell)$ a *composition* of n.

We will begin with the partitions of $n = 5$; as before, we begin by constructing the set of these partitions:

```
sage: P5 = Partitions(5); P5
Partitions of the integer 5
```

```
>>> from sage.all import *
>>> P5 = Partitions(Integer(5)); P5
Partitions of the integer 5
```

then we ask for its cardinality:

```
sage: P5.cardinality()
7
```

```
>>> from sage.all import *
>>> P5.cardinality()
7
```

We look at these 7 partitions; the order being irrelevant, the entries are ordered, by convention, in decreasing order.

```
sage: P5.list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],
 [1, 1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> P5.list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],
 [1, 1, 1, 1, 1]]
```

The calculation of the number of partitions uses the Rademacher formula (Wikipedia article Partition_(number_theory)), implemented in `C` and highly optimized, which makes it very fast:

```
sage: Partitions(100000).cardinality()
27493510569775696512677516320986352688173429315980054758203125984302147328114964173055
↪050741660736621590157844774296248940493063070200461792764493033510116079342457190155
↪718943509725312466108452006369558934464248716828789832182345009262853831404597021307
↪130674510624419227311238999702284408609370935531629697851569569892196108480158600569
↪421098519
```

```
>>> from sage.all import *
>>> Partitions(Integer(100000)).cardinality()
27493510569775696512677516320986352688173429315980054758203125984302147328114964173055
↪050741660736621590157844774296248940493063070200461792764493033510116079342457190155
↪718943509725312466108452006369558934464248716828789832182345009262853831404597021307
↪130674510624419227311238999702284408609370935531629697851569569892196108480158600569
↪421098519
```

Partitions of integers are combinatorial objects naturally equipped with many operations. They are therefore returned as objects that are richer than simple lists:

```
sage: P7 = Partitions(7)
sage: p = P7.unrank(5); p
[4, 2, 1]
sage: type(p)
<class 'sage.combinat.partition.Partitions_n_with_category.element_class'>
```

```
>>> from sage.all import *
>>> P7 = Partitions(Integer(7))
>>> p = P7.unrank(Integer(5)); p
[4, 2, 1]
>>> type(p)
<class 'sage.combinat.partition.Partitions_n_with_category.element_class'>
```

For example, they can be represented graphically by a Ferrers diagram:

```
sage: print(p.ferrers_diagram())
****
**
*
```

```
>>> from sage.all import *
>>> print(p.ferrers_diagram())
****
**
*
```

We leave it to the user to explore by introspection the available operations.

Note that we can also construct a partition directly by:

```
sage: Partition([4,2,1])
[4, 2, 1]
```

```
>>> from sage.all import *
>>> Partition([Integer(4),Integer(2),Integer(1)])
[4, 2, 1]
```

or:

```
sage: P7([4,2,1])
[4, 2, 1]
```

```
>>> from sage.all import *
>>> P7([Integer(4),Integer(2),Integer(1)])
[4, 2, 1]
```

If one wants to restrict the possible values of the parts $i_1, \ldots, i_\ell$ of the partition as, for example, when giving change, one can use `WeightedIntegerVectors`. For example, the following calculation:

```
sage: WeightedIntegerVectors(8, [2,3,5]).list()
[[0, 1, 1], [1, 2, 0], [4, 0, 0]]
```

```
>>> from sage.all import *
>>> WeightedIntegerVectors(Integer(8), [Integer(2),Integer(3),Integer(5)]).list()
[[0, 1, 1], [1, 2, 0], [4, 0, 0]]
```

shows that to make 8 dollars using 2, 3, and 5 dollar bills, one can use a 3 and a 5 dollar bill, or a 2 and two 3 dollar bills, or four 2 dollar bills.

Compositions of integers are manipulated the same way:

```
sage: C5 = Compositions(5); C5
Compositions of 5
sage: C5.cardinality()
16
sage: C5.list()
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 3],
```

(continues on next page)

(continued from previous page)

```
 [1, 2, 1, 1], [1, 2, 2], [1, 3, 1], [1, 4], [2, 1, 1, 1],
 [2, 1, 2], [2, 2, 1], [2, 3], [3, 1, 1], [3, 2], [4, 1], [5]]
```

```
>>> from sage.all import *
>>> C5 = Compositions(Integer(5)); C5
Compositions of 5
>>> C5.cardinality()
16
>>> C5.list()
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 3],
 [1, 2, 1, 1], [1, 2, 2], [1, 3, 1], [1, 4], [2, 1, 1, 1],
 [2, 1, 2], [2, 2, 1], [2, 3], [3, 1, 1], [3, 2], [4, 1], [5]]
```

The number 16 above seems significant and suggests the existence of a formula. We look at the number of compositions of n ranging from 0 to 9:

```
sage: [Compositions(n).cardinality() for n in range(10)]
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256]
```

```
>>> from sage.all import *
>>> [Compositions(n).cardinality() for n in range(Integer(10))]
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256]
```

Similarly, if we consider the number of compositions of 5 by length, we find a line of Pascal's triangle:

```
sage: x = var('x')                                                          #␣
↪needs sage.symbolic
sage: sum(x^len(c) for c in C5)                                             #␣
↪needs sage.symbolic
x^5 + 4*x^4 + 6*x^3 + 4*x^2 + x
```

```
>>> from sage.all import *
>>> x = var('x')                                                          #␣
↪needs sage.symbolic
>>> sum(x**len(c) for c in C5)                                            #␣
↪needs sage.symbolic
x^5 + 4*x^4 + 6*x^3 + 4*x^2 + x
```

The above example uses a functionality which we have not seen yet: `C5` being iterable, it can be used like a list in a `for` loop or a comprehension (*Set comprehension and iterators*).

> Prove the formulas suggested by the above examples for the number of compositions of n and the number of compositions of n of length k; investigate by introspection whether `Sage` uses these formulas for calculating cardinalities.

Some other finite enumerated sets

Essentially, the principle is the same for all the finite sets with which one wants to do combinatorics in `Sage`; begin by constructing an object which models this set, and then supply appropriate methods, following a uniform interface[1]. We now give a few more typical examples.

Intervals of integers:

[1] Or at least that should be the case; there are still many corners to clean up.

```
sage: C = IntegerRange(3, 21, 2); C
{3, 5, ..., 19}
sage: C.cardinality()
9
sage: C.list()
[3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
>>> from sage.all import *
>>> C = IntegerRange(Integer(3), Integer(21), Integer(2)); C
{3, 5, ..., 19}
>>> C.cardinality()
9
>>> C.list()
[3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Permutations:

```
sage: C = Permutations(4); C
Standard permutations of 4
sage: C.cardinality()
24
sage: C.list()
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
 [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3],
 [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1],
 [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
 [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2],
 [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]
```

```
>>> from sage.all import *
>>> C = Permutations(Integer(4)); C
Standard permutations of 4
>>> C.cardinality()
24
>>> C.list()
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
 [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3],
 [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1],
 [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
 [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2],
 [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]
```

Set partitions:

```
sage: C = SetPartitions(["a", "b", "c"])
sage: C # random print output
Set partitions of {'a', 'c', 'b'}
sage: C.cardinality()
5
sage: C.list()
[{{'a', 'b', 'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'c'}, {'b'}},
```

(continues on next page)

(continued from previous page)

```
 {{'a'}, {'b', 'c'}},
 {{'a'}, {'b'}, {'c'}}]
```

```
>>> from sage.all import *
>>> C = SetPartitions(["a", "b", "c"])
>>> C # random print output
Set partitions of {'a', 'c', 'b'}
>>> C.cardinality()
5
>>> C.list()
[{{'a', 'b', 'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'c'}, {'b'}},
 {{'a'}, {'b', 'c'}},
 {{'a'}, {'b'}, {'c'}}]
```

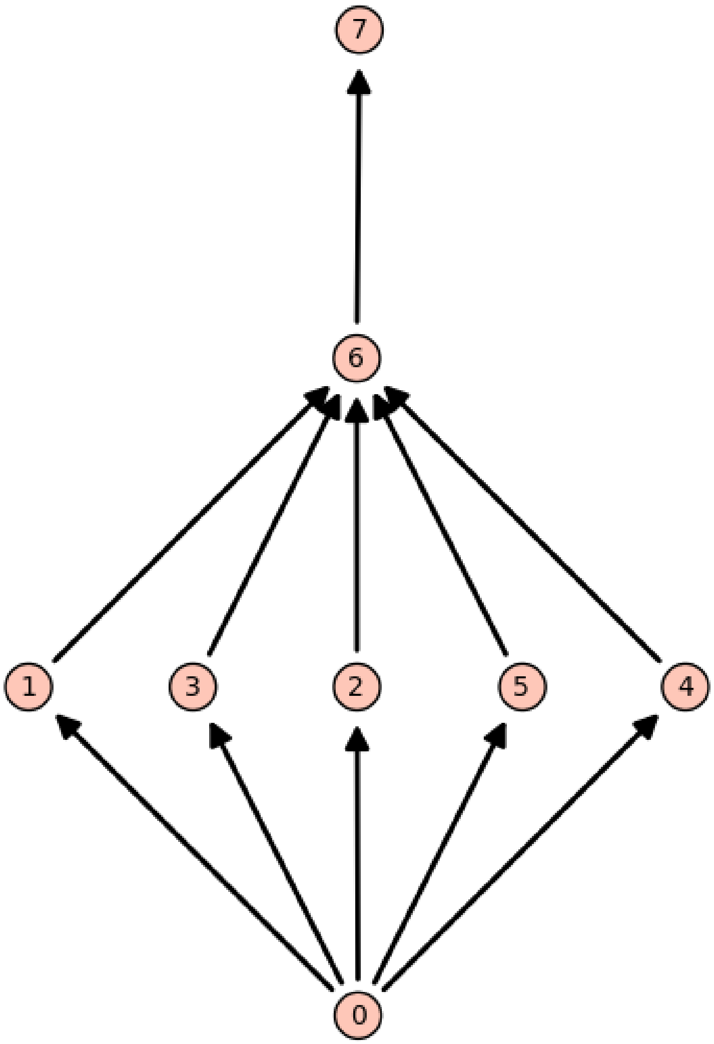
Partial orders on a set of 8 elements, up to isomorphism:

```
sage: C = Posets(7); C
Posets containing 7 elements
sage: C.cardinality()
2045
```

```
>>> from sage.all import *
>>> C = Posets(Integer(7)); C
Posets containing 7 elements
>>> C.cardinality()
2045
```

```
sage: C.unrank(20).plot()
Graphics object consisting of ... graphics primitives
```

```
>>> from sage.all import *
>>> C.unrank(Integer(20)).plot()
Graphics object consisting of ... graphics primitives
```



One can iterate through all graphs up to isomorphism. For example, there are 34 simple graphs with 5 vertices:

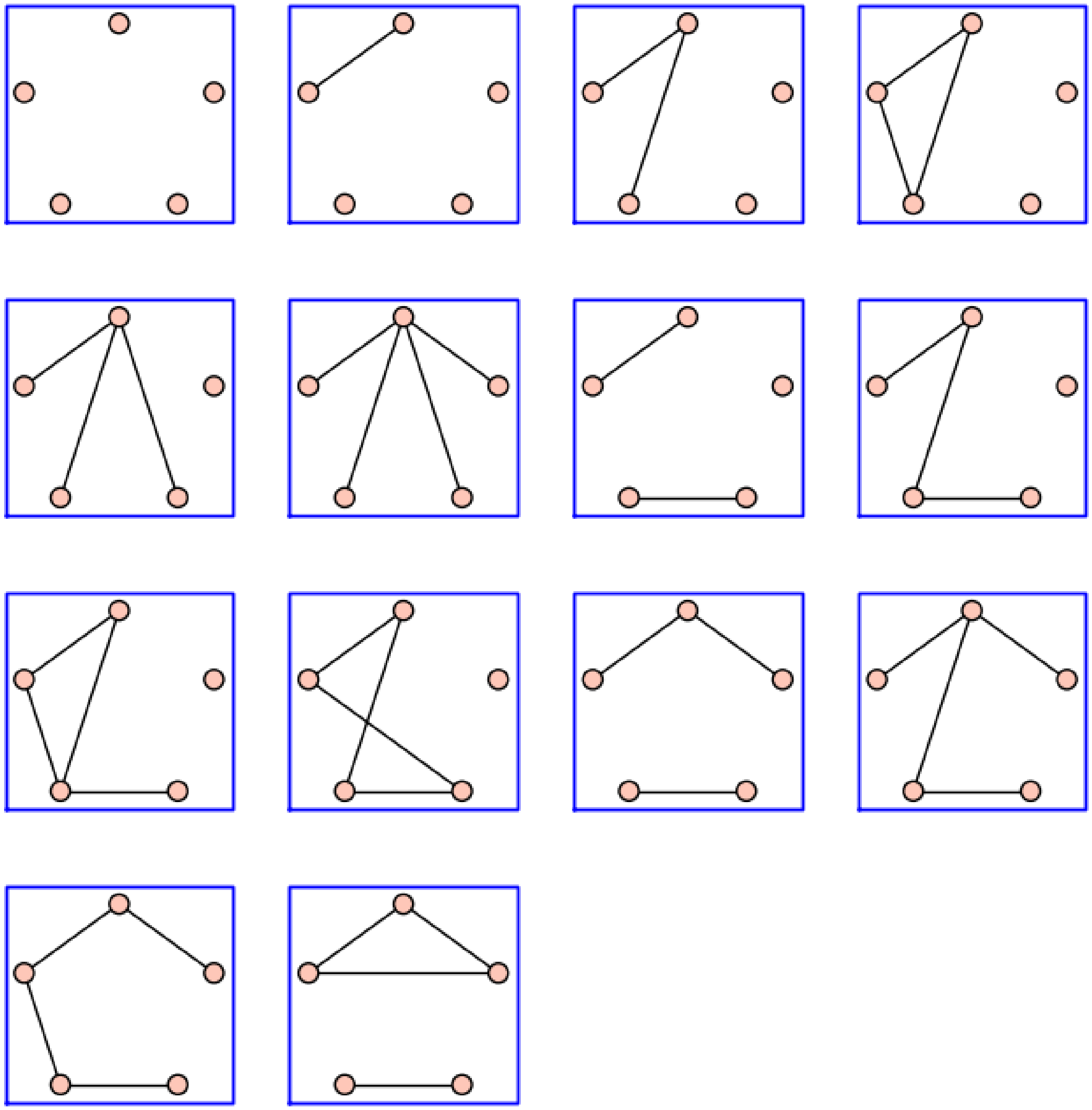
```
sage: len(list(graphs(5)))
34
```

```
>>> from sage.all import *
>>> len(list(graphs(Integer(5))))
34
```

Here are those with at most 4 edges:

```
sage: up_to_four_edges = list(graphs(5, lambda G: G.size() <= 4))
sage: pretty_print(*up_to_four_edges)
```

```
>>> from sage.all import *
>>> up_to_four_edges = list(graphs(Integer(5), lambda G: G.size() <= Integer(4)))
>>> pretty_print(*up_to_four_edges)
```



However, the *set* `C` of these graphs is not yet available in `Sage`; as a result, the following commands are not yet implemented:

```
sage: # not implemented
sage: C = Graphs(5)
sage: C.cardinality()
34
sage: Graphs(19).cardinality()
24637809253125004524383007491432768
sage: Graphs(19).random_element()
Graph on 19 vertices
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> # not implemented
>>> C = Graphs(Integer(5))
>>> C.cardinality()
34
>>> Graphs(Integer(19)).cardinality()
24637809253125004524383007491432768
>>> Graphs(Integer(19)).random_element()
Graph on 19 vertices
```

What we have seen so far also applies, in principle, to finite algebraic structures like the dihedral groups:

```
sage: G = DihedralGroup(4); G
Dihedral group of order 8 as a permutation group
sage: G.cardinality()
8
sage: G.list()
[(), (1,3)(2,4), (1,4,3,2), (1,2,3,4), (2,4), (1,3), (1,4)(2,3), (1,2)(3,4)]
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(4)); G
Dihedral group of order 8 as a permutation group
>>> G.cardinality()
8
>>> G.list()
[(), (1,3)(2,4), (1,4,3,2), (1,2,3,4), (2,4), (1,3), (1,4)(2,3), (1,2)(3,4)]
```

or the algebra of 2×2 matrices over the finite field $\mathbf{Z}/2\mathbf{Z}$:

```
sage: C = MatrixSpace(GF(2), 2)                                                  #␣
↪needs sage.modules sage.rings.finite_rings
sage: C.list()                                                                   #␣
↪needs sage.modules sage.rings.finite_rings
[
[0 0]  [1 0]  [0 1]  [0 0]  [0 0]  [1 1]  [1 0]  [1 0]  [0 1]  [0 1]
[0 0], [0 0], [0 0], [1 0], [0 1], [0 0], [1 0], [0 1], [1 0], [0 1],

[0 0]  [1 1]  [1 1]  [1 0]  [0 1]  [1 1]
[1 1], [1 0], [0 1], [1 1], [1 1], [1 1]
]
sage: C.cardinality()                                                            #␣
↪needs sage.modules sage.rings.finite_rings
16
```

```
>>> from sage.all import *
>>> C = MatrixSpace(GF(Integer(2)), Integer(2))                                   ␣
↪             # needs sage.modules sage.rings.finite_rings
>>> C.list()                                                                    #␣
↪needs sage.modules sage.rings.finite_rings
[
[0 0]  [1 0]  [0 1]  [0 0]  [0 0]  [1 1]  [1 0]  [1 0]  [0 1]  [0 1]
[0 0], [0 0], [0 0], [1 0], [0 1], [0 0], [1 0], [0 1], [1 0], [0 1],
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
[0 0]  [1 1]  [1 1]  [1 0]  [0 1]  [1 1]
[1 1], [1 0], [0 1], [1 1], [1 1], [1 1]
]
>>> C.cardinality()                                                      #␣
↪needs sage.modules sage.rings.finite_rings
16
```

Exercise

List all the monomials of degree 5 in three variables (see `IntegerVectors`). Manipulate the ordered set partitions `OrderedSetPartitions` and standard tableaux (`StandardTableaux`).

Exercise

List the alternating sign matrices of size 3, 4, and 5 (`AlternatingSignMatrices`), and try to guess the definition. The discovery and proof of the formula for the enumeration of these matrices (see the method `cardinality`), motivated by calculations of determinants in physics, is quite a story. In particular, the first proof, given by Zeilberger in 1992 was automatically produced by a computer program. It was 84 pages long, and required nearly a hundred people to verify it.

Exercise

Calculate by hand the number of vectors in $(\mathbf{Z}/2\mathbf{Z})^5$, and the number of matrices in $GL_3(\mathbf{Z}/2\mathbf{Z})$ (that is to say, the number of invertible 3×3 matrices with coefficients in $\mathbf{Z}/2\mathbf{Z}$). Verify your answer with `Sage`. Generalize to $GL_n(\mathbf{Z}/q\mathbf{Z})$.

Set comprehension and iterators

We will now show some of the possibilities offered by `Python` for constructing (and iterating through) sets, with a notation that is flexible and close to usual mathematical usage, and in particular the benefits this yields in combinatorics.

We begin by constructing the finite set $\{i^2 \parallel i \in \{1, 3, 7\}\}$:

```
sage: [ i^2 for i in [1, 3, 7] ]
[1, 9, 49]
```

```
>>> from sage.all import *
>>> [ i**Integer(2) for i in [Integer(1), Integer(3), Integer(7)] ]
[1, 9, 49]
```

and then the same set, but with i running from 1 to 9:

```
sage: [ i^2 for i in range(1,10) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> from sage.all import *
>>> [ i**Integer(2) for i in range(Integer(1),Integer(10)) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A construction of this form in `Python` is called *set comprehension*. A clause can be added to keep only those elements with i prime:

```
sage: [ i^2 for i in range(1,10) if is_prime(i) ]
[4, 9, 25, 49]
```

```
>>> from sage.all import *
>>> [ i**Integer(2) for i in range(Integer(1),Integer(10)) if is_prime(i) ]
[4, 9, 25, 49]
```

Combining more than one set comprehension, it is possible to construct the set $\{(i, j) \mid 1 \leq j < i < 5\}$:

```
sage: [ (i,j) for i in range(1,6) for j in range(1,i) ]
[(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3),
 (5, 1), (5, 2), (5, 3), (5, 4)]
```

```
>>> from sage.all import *
>>> [ (i,j) for i in range(Integer(1),Integer(6)) for j in range(Integer(1),i) ]
[(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3),
 (5, 1), (5, 2), (5, 3), (5, 4)]
```

or to produce Pascal's triangle:

```
sage: [[binomial(n,i) for i in range(n+1)] for n in range(10)]
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

```
>>> from sage.all import *
>>> [[binomial(n,i) for i in range(n+Integer(1))] for n in range(Integer(10))]
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

The execution of a set comprehension is accomplished in two steps; first an *iterator* is constructed, and then a list is filled with the elements successively produced by the iterator. Technically, an *iterator* is an object with a method `next` which returns a new value each time it is called, until it is exhausted. For example, the following iterator `it`:

```
sage: it = (binomial(3, i) for i in range(4))
```

```
>>> from sage.all import *
>>> it = (binomial(Integer(3), i) for i in range(Integer(4)))
```

returns successively the binomial coefficients $\binom{3}{i}$ with $i = 0, 1, 2, 3$:

```
sage: next(it)
1
sage: next(it)
3
sage: next(it)
3
sage: next(it)
1
```

```
>>> from sage.all import *
>>> next(it)
1
>>> next(it)
3
>>> next(it)
3
>>> next(it)
1
```

When the iterator is finally exhausted, an exception is raised:

```
sage: next(it)
Traceback (most recent call last):
  ...
StopIteration
```

```
>>> from sage.all import *
>>> next(it)
Traceback (most recent call last):
  ...
StopIteration
```

More generally, an *iterable* is a `Python` object `L` (a list, a set, ...) over whose elements it is possible to iterate. Technically, the iterator is constructed by `iter(L)`. In practice, the commands `iter` and `next` are used very rarely, since `for` loops and list comprehensions provide a much pleasanter syntax:

```
sage: for s in Subsets(3):
....:     print(s)
{}
{1}
{2}
{3}
{1, 2}
{1, 3}
{2, 3}
{1, 2, 3}
```

```
>>> from sage.all import *
>>> for s in Subsets(Integer(3)):
...     print(s)
{}
{1}
{2}
{3}
{1, 2}
{1, 3}
{2, 3}
{1, 2, 3}
```

```
sage: [ s.cardinality() for s in Subsets(3) ]
[0, 1, 1, 1, 2, 2, 2, 3]
```

```
>>> from sage.all import *
>>> [ s.cardinality() for s in Subsets(Integer(3)) ]
[0, 1, 1, 1, 2, 2, 2, 3]
```

What is the point of an iterator? Consider the following example:

```
sage: sum( [ binomial(8, i) for i in range(9) ] )
256
```

```
>>> from sage.all import *
>>> sum( [ binomial(Integer(8), i) for i in range(Integer(9)) ] )
256
```

When it is executed, a list of 9 elements is constructed, and then it is passed as an argument to `sum` to add them up. If, on the other hand, the iterator is passed directly to `sum` (note the absence of square brackets):

```
sage: sum( binomial(8, i) for i in range(9) )
256
```

```
>>> from sage.all import *
>>> sum( binomial(Integer(8), i) for i in range(Integer(9)) )
256
```

the function `sum` receives the iterator directly, and can short-circuit the construction of the intermediate list. If there are a large number of elements, this avoids allocating a large quantity of memory to fill a list which will be immediately destroyed.

Most functions that take a list of elements as input will also accept an iterator (or an iterable) instead. To begin with, one can obtain the list (or the tuple) of elements of an iterator as follows:

```
sage: list(binomial(8, i) for i in range(9))
[1, 8, 28, 56, 70, 56, 28, 8, 1]
sage: tuple(binomial(8, i) for i in range(9))
(1, 8, 28, 56, 70, 56, 28, 8, 1)
```

```
>>> from sage.all import *
>>> list(binomial(Integer(8), i) for i in range(Integer(9)))
```

(continues on next page)

(continued from previous page)

```
[1, 8, 28, 56, 70, 56, 28, 8, 1]
>>> tuple(binomial(Integer(8), i) for i in range(Integer(9)))
(1, 8, 28, 56, 70, 56, 28, 8, 1)
```

We now consider the functions `all` and `any` which denote respectively the n-ary *and* and *or*:

```
sage: all([True, True, True, True])
True
sage: all([True, False, True, True])
False
sage: any([False, False, False, False])
False
sage: any([False, False, True, False])
True
```

```
>>> from sage.all import *
>>> all([True, True, True, True])
True
>>> all([True, False, True, True])
False
>>> any([False, False, False, False])
False
>>> any([False, False, True, False])
True
```

The following example verifies that all primes from 3 to 99 are odd:

```
sage: all( is_odd(p) for p in range(3,100) if is_prime(p) )
True
```

```
>>> from sage.all import *
>>> all( is_odd(p) for p in range(Integer(3),Integer(100)) if is_prime(p) )
True
```

A *Mersenne prime* is a prime of the form $2^p - 1$. We verify that, for $p < 1000$, if $2^p - 1$ is prime, then p is also prime:

```
sage: def mersenne(p): return 2^p -1
sage: [ is_prime(p)
....:   for p in range(1000) if is_prime(mersenne(p)) ]
[True, True, True, True, True, True, True, True, True, True,
 True, True, True, True]
```

```
>>> from sage.all import *
>>> def mersenne(p): return Integer(2)**p -Integer(1)
>>> [ is_prime(p)
...   for p in range(Integer(1000)) if is_prime(mersenne(p)) ]
[True, True, True, True, True, True, True, True, True, True,
 True, True, True, True]
```

Is the converse true?

Exercise

Try the two following commands and explain the considerable difference in the length of the calculations:

```
sage: all(   is_prime(mersenne(p))
....:         for p in range(1000) if is_prime(p)   )
False
sage: all( [ is_prime(mersenne(p))
....:         for p in range(1000) if is_prime(p)] )
False
```

```
>>> from sage.all import *
>>> all(   is_prime(mersenne(p))
...         for p in range(Integer(1000)) if is_prime(p)   )
False
>>> all( [ is_prime(mersenne(p))
...         for p in range(Integer(1000)) if is_prime(p)] )
False
```

We now try to find the smallest counter-example. In order to do this, we use the `Sage` function `exists`:

```
sage: exists( (p for p in range(1000) if is_prime(p)),
....:          lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

```
>>> from sage.all import *
>>> exists( (p for p in range(Integer(1000)) if is_prime(p)),
...          lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

Alternatively, we could construct an iterator on the counter-examples:

```
sage: counter_examples = (p for p in range(1000)
....:       if is_prime(p) and not is_prime(mersenne(p)))
sage: next(counter_examples)
11
sage: next(counter_examples)
23
```

```
>>> from sage.all import *
>>> counter_examples = (p for p in range(Integer(1000))
...       if is_prime(p) and not is_prime(mersenne(p)))
>>> next(counter_examples)
11
>>> next(counter_examples)
23
```

Exercise

What do the following commands do?

```
sage: cubes = [t**3 for t in range(-999,1000)]
sage: exists([(x,y) for x in cubes for y in cubes],   # long time (3s, 2012)
```

```
....:         lambda x_y: x_y[0] + x_y[1] == 218)
(True, (-125, 343))
sage: exists(((x,y) for x in cubes for y in cubes),  # long time (2s, 2012)
....:         lambda x_y: x_y[0] + x_y[1] == 218)
(True, (-125, 343))
```

```
>>> from sage.all import *
>>> cubes = [t**Integer(3) for t in range(-Integer(999),Integer(1000))]
>>> exists([(x,y) for x in cubes for y in cubes],  # long time (3s, 2012)
...        lambda x_y: x_y[Integer(0)] + x_y[Integer(1)] == Integer(218))
(True, (-125, 343))
>>> exists(((x,y) for x in cubes for y in cubes),  # long time (2s, 2012)
...        lambda x_y: x_y[Integer(0)] + x_y[Integer(1)] == Integer(218))
(True, (-125, 343))
```

Which of the last two is more economical in terms of time? In terms of memory? By how much?

Exercise

Try each of the following commands, and explain its result. If possible, hide the result first and try to guess it before launching the command.

Todo

hide the results by default

Warning

it will be necessary to interrupt the execution of some of the commands

```
sage: x = var('x')                                                    #␣
↪needs sage.symbolic
sage: sum(x^len(s) for s in Subsets(8))                               #␣
↪needs sage.symbolic
x^8 + 8*x^7 + 28*x^6 + 56*x^5 + 70*x^4 + 56*x^3 + 28*x^2 + 8*x + 1
```

```
>>> from sage.all import *
>>> x = var('x')                                                    #␣
↪needs sage.symbolic
>>> sum(x**len(s) for s in Subsets(Integer(8)))                        ␣
↪   # needs sage.symbolic
x^8 + 8*x^7 + 28*x^6 + 56*x^5 + 70*x^4 + 56*x^3 + 28*x^2 + 8*x + 1
```

```
sage: sum(x^p.length() for p in Permutations(3))                      #␣
↪needs sage.symbolic
x^3 + 2*x^2 + 2*x + 1
```

```
>>> from sage.all import *
>>> sum(x**p.length() for p in Permutations(Integer(3)))               ␣
↪   # needs sage.symbolic
x^3 + 2*x^2 + 2*x + 1
```

```
sage: factor(sum(x^p.length() for p in Permutations(3)))                    #␣
↪needs sage.symbolic
(x^2 + x + 1)*(x + 1)
```

```
>>> from sage.all import *
>>> factor(sum(x**p.length() for p in Permutations(Integer(3))))            ␣
↪   # needs sage.symbolic
(x^2 + x + 1)*(x + 1)
```

```
sage: P = Permutations(5)
sage: all(p in P for p in P)
True
```

```
>>> from sage.all import *
>>> P = Permutations(Integer(5))
>>> all(p in P for p in P)
True
```

```
sage: for p in GL(2, 2): print(p); print("")
[1 0]
[0 1]

[0 1]
[1 0]

[0 1]
[1 1]

[1 1]
[0 1]

[1 1]
[1 0]

[1 0]
[1 1]
```

```
>>> from sage.all import *
>>> for p in GL(Integer(2), Integer(2)): print(p); print("")
[1 0]
[0 1]
<BLANKLINE>
[0 1]
[1 0]
<BLANKLINE>
[0 1]
[1 1]
<BLANKLINE>
[1 1]
[0 1]
<BLANKLINE>
[1 1]
[1 0]
<BLANKLINE>
[1 0]
[1 1]
<BLANKLINE>
```

```
sage: for p in Partitions(3): print(p)    # not tested
[3]
[2, 1]
[1, 1, 1]
...
```

```
>>> from sage.all import *
>>> for p in Partitions(Integer(3)): print(p)    # not tested
[3]
[2, 1]
[1, 1, 1]
...
```

```
sage: for p in Partitions(): print(p)     # not tested
[]
[1]
[2]
[1, 1]
[3]
...
```

```
>>> from sage.all import *
>>> for p in Partitions(): print(p)     # not tested
[]
[1]
[2]
[1, 1]
[3]
...
```

```
sage: for p in Primes(): print(p)         # not tested
2
3
5
7
...
```

```
>>> from sage.all import *
>>> for p in Primes(): print(p)         # not tested
2
3
5
7
...
```

```
sage: exists( Primes(), lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

```
>>> from sage.all import *
>>> exists( Primes(), lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

```
sage: counter_examples = (p for p in Primes()
....:                     if not is_prime(mersenne(p)))
sage: for p in counter_examples: print(p)    # not tested
11
23
29
37
41
43
47
...
```

```
>>> from sage.all import *
>>> counter_examples = (p for p in Primes()
...                     if not is_prime(mersenne(p)))
>>> for p in counter_examples: print(p)    # not tested
11
23
29
37
41
43
47
...
```

Operations on iterators

`Python` provides numerous tools for manipulating iterators; most of them are in the `itertools` library, which can be imported by:

```
sage: import itertools
```

```
>>> from sage.all import *
>>> import itertools
```

We will demonstrate some applications, taking as a starting point the permutations of 3:

```
sage: list(Permutations(3))
[[1, 2, 3], [1, 3, 2], [2, 1, 3],
 [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

```
>>> from sage.all import *
>>> list(Permutations(Integer(3)))
[[1, 2, 3], [1, 3, 2], [2, 1, 3],
 [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

We can list the elements of a set by numbering them:

```
sage: list(enumerate(Permutations(3)))
[(0, [1, 2, 3]), (1, [1, 3, 2]), (2, [2, 1, 3]),
 (3, [2, 3, 1]), (4, [3, 1, 2]), (5, [3, 2, 1])]
```

```
>>> from sage.all import *
>>> list(enumerate(Permutations(Integer(3))))
[(0, [1, 2, 3]), (1, [1, 3, 2]), (2, [2, 1, 3]),
 (3, [2, 3, 1]), (4, [3, 1, 2]), (5, [3, 2, 1])]
```

or select only the elements in positions 2, 3, and 4 (analogue of `l[1:4]`):

```
sage: import itertools
sage: list(itertools.islice(Permutations(3), int(1), int(4)))
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]
```

```
>>> from sage.all import *
>>> import itertools
>>> list(itertools.islice(Permutations(Integer(3)), int(Integer(1)), int(Integer(4))))
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]
```

To apply a function to all the elements, one can do:

```
sage: [z.cycle_type() for z in Permutations(3)]
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

```
>>> from sage.all import *
>>> [z.cycle_type() for z in Permutations(Integer(3))]
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

and similarly to select the elements satisfying a certain condition:

```
sage: [z for z in Permutations(3) if z.has_pattern([1,2])]
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]
```

```
>>> from sage.all import *
>>> [z for z in Permutations(Integer(3)) if z.has_pattern([Integer(1),Integer(2)])]
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]
```

Implementation of new iterators

It is easy to construct new iterators, using the keyword `yield` instead of `return` in a function:

```
sage: def f(n):
....:     for i in range(n):
....:         yield i
```

```
>>> from sage.all import *
>>> def f(n):
...     for i in range(n):
...         yield i
```

After the `yield`, execution is not halted, but only suspended, ready to be continued from the same point. The result of the function is therefore an iterator over the successive values returned by `yield`:

```
sage: g = f(4)
sage: next(g)
```

(continues on next page)

(continued from previous page)

```
0
sage: next(g)
1
sage: next(g)
2
sage: next(g)
3
```

```
>>> from sage.all import *
>>> g = f(Integer(4))
>>> next(g)
0
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
```

```
sage: next(g)
Traceback (most recent call last):
  ...
StopIteration
```

```
>>> from sage.all import *
>>> next(g)
Traceback (most recent call last):
  ...
StopIteration
```

The function could be used as follows:

```
sage: [ x for x in f(5) ]
[0, 1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> [ x for x in f(Integer(5)) ]
[0, 1, 2, 3, 4]
```

This model of computation, called *continuation*, is very useful in combinatorics, especially when combined with recursion. Here is how to generate all words of a given length on a given alphabet:

```
sage: def words(alphabet, l):
....:     if l == 0:
....:         yield []
....:     else:
....:         for word in words(alphabet, l-1):
....:             for l in alphabet:
....:                 yield word + [l]
sage: [ w for w in words(['a','b'], 3) ]
[['a', 'a', 'a'], ['a', 'a', 'b'], ['a', 'b', 'a'],
```

(continues on next page)

(continued from previous page)

```
 ['a', 'b', 'b'], ['b', 'a', 'a'], ['b', 'a', 'b'],
 ['b', 'b', 'a'], ['b', 'b', 'b']]
```

```
>>> from sage.all import *
>>> def words(alphabet, l):
...     if l == Integer(0):
...         yield []
...     else:
...         for word in words(alphabet, l-Integer(1)):
...             for l in alphabet:
...                 yield word + [l]
>>> [ w for w in words(['a','b'], Integer(3)) ]
[['a', 'a', 'a'], ['a', 'a', 'b'], ['a', 'b', 'a'],
 ['a', 'b', 'b'], ['b', 'a', 'a'], ['b', 'a', 'b'],
 ['b', 'b', 'a'], ['b', 'b', 'b']]
```

These words can then be counted by:

```
sage: sum(1 for w in words(['a','b','c','d'], 10))
1048576
```

```
>>> from sage.all import *
>>> sum(Integer(1) for w in words(['a','b','c','d'], Integer(10)))
1048576
```

Counting the words one by one is clearly not an efficient method in this case, since the formula n^ℓ is also available; note, though, that this is not the stupidest possible approach - it does, at least, avoid constructing the entire list in memory.

We now consider Dyck words, which are well-parenthesized words in the letters "(" and ")". The function below generates all the Dyck words of a given length (where the length is the number of pairs of parentheses), using the recursive definition which says that a Dyck word is either empty or of the form $(w_1)w_2$ where w_1 and w_2 are Dyck words:

```
sage: def dyck_words(l):
....:     if l==0:
....:         yield ''
....:     else:
....:         for k in range(l):
....:             for w1 in dyck_words(k):
....:                 for w2 in dyck_words(l-k-1):
....:                     yield '('+w1+')'+w2
```

```
>>> from sage.all import *
>>> def dyck_words(l):
...     if l==Integer(0):
...         yield ''
...     else:
...         for k in range(l):
...             for w1 in dyck_words(k):
...                 for w2 in dyck_words(l-k-Integer(1)):
...                     yield '('+w1+')'+w2
```

Here are all the Dyck words of length 4:

```
sage: list(dyck_words(4))
['()()()()', '()()(())', '()(())()', '()(()())', '()((()))',
 '(())()()', '(())(())', '(()())()', '((()))()', '(()()())',
 '(()(()))', '((())())', '((()()))', '(((())))']
```

```
>>> from sage.all import *
>>> list(dyck_words(Integer(4)))
['()()()()', '()()(())', '()(())()', '()(()())', '()((()))',
 '(())()()', '(())(())', '(()())()', '((()))()', '(()()())',
 '(()(()))', '((())())', '((()()))', '(((())))']
```

Counting them, we recover a well-known sequence:

```
sage: [ sum(1 for w in dyck_words(l)) for l in range(10) ]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> from sage.all import *
>>> [ sum(Integer(1) for w in dyck_words(l)) for l in range(Integer(10)) ]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

Exercise: complete binary tree iterator

Construct an iterator on the set C_n of complete binary trees with n leaves (see *Enumeration of trees using generating functions*).

Hint: `Sage` 4.8.2 does not yet have a native data structure to represent complete binary trees. One simple way to represent them is to define a formal variable `Leaf` for the leaves and a formal 2-ary function `Node`:

```
sage: var('Leaf')                                                    #␣
↪needs sage.symbolic
Leaf
sage: function('Node', nargs=2)                                      #␣
↪needs sage.symbolic
Node
```

```
>>> from sage.all import *
>>> var('Leaf')                                                    #␣
↪needs sage.symbolic
Leaf
>>> function('Node', nargs=Integer(2))                               ␣
↪   # needs sage.symbolic
Node
```

The second tree in *Figure: The five complete binary trees with four leaves* can be represented by the expression:

```
sage: tr = Node(Node(Leaf, Node(Leaf, Leaf)), Leaf)                  #␣
↪needs sage.symbolic
```

```
>>> from sage.all import *
>>> tr = Node(Node(Leaf, Node(Leaf, Leaf)), Leaf)                  #␣
↪needs sage.symbolic
```

Constructions

We will now see how to construct new sets starting from these building blocks. In fact, we have already begun to do this with the construction of $\mathcal{P}(\mathcal{P}(\mathcal{P}(\{1,2,3,4\})))$ in the previous section, and to construct the example of sets of cards in *Initial examples*.

Consider a large Cartesian product:

```
sage: C = cartesian_product([Compositions(8), Permutations(20)]); C
The Cartesian product of (Compositions of 8, Standard permutations of 20)
sage: C.cardinality()
311411457046609920000
```

```
>>> from sage.all import *
>>> C = cartesian_product([Compositions(Integer(8)), Permutations(Integer(20))]); C
The Cartesian product of (Compositions of 8, Standard permutations of 20)
>>> C.cardinality()
311411457046609920000
```

Clearly, it is impractical to construct the list of all the elements of this Cartesian product! And, in the following example, H is equipped with the usual combinatorial operations and also its structure as a product group:

```
sage: G = DihedralGroup(4)
sage: H = cartesian_product([G,G])
sage: H in Groups()
True
sage: H.an_element()
((1,3), (1,3))
sage: t = H([G.gen(0), G.gen(0)])
sage: t
((1,2,3,4), (1,2,3,4))
sage: t*t
((1,3)(2,4), (1,3)(2,4))
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(4))
>>> H = cartesian_product([G,G])
>>> H in Groups()
True
>>> H.an_element()
((1,3), (1,3))
>>> t = H([G.gen(Integer(0)), G.gen(Integer(0))])
>>> t
((1,2,3,4), (1,2,3,4))
>>> t*t
((1,3)(2,4), (1,3)(2,4))
```

We now construct the union of two existing disjoint sets:

```
sage: C = DisjointUnionEnumeratedSets(
....:       [ Compositions(4), Permutations(3)] )
sage: C
Disjoint union of Family (Compositions of 4,
Standard permutations of 3)
```

(continues on next page)

(continued from previous page)

```
sage: C.cardinality()
14
sage: C.list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2],
[3, 1], [4], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1],
[3, 1, 2], [3, 2, 1]]
```

```
>>> from sage.all import *
>>> C = DisjointUnionEnumeratedSets(
...         [ Compositions(Integer(4)), Permutations(Integer(3))] )
>>> C
Disjoint union of Family (Compositions of 4,
Standard permutations of 3)
>>> C.cardinality()
14
>>> C.list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2],
[3, 1], [4], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1],
[3, 1, 2], [3, 2, 1]]
```

It is also possible to take the union of more than two disjoint sets, or even an infinite number of them. We will now construct the set of all permutations, viewed as the union of the sets P_n of permutations of size n. We begin by constructing the infinite family $F = (P_n)_{n\in N}$:

```
sage: F = Family(NonNegativeIntegers(), Permutations); F
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in Non negative
→integers}
sage: F.keys()
Non negative integers
sage: F[1000]
Standard permutations of 1000
```

```
>>> from sage.all import *
>>> F = Family(NonNegativeIntegers(), Permutations); F
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in Non negative
→integers}
>>> F.keys()
Non negative integers
>>> F[Integer(1000)]
Standard permutations of 1000
```

Now we can construct the disjoint union $\bigcup_{n\in\mathbf{N}} P_n$:

```
sage: U = DisjointUnionEnumeratedSets(F); U
Disjoint union of
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in Non negative
→integers}
```

```
>>> from sage.all import *
>>> U = DisjointUnionEnumeratedSets(F); U
Disjoint union of
```

(continues on next page)

(continued from previous page)

```
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in Non negative
↪integers}
```

It is an infinite set:

```
sage: U.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> U.cardinality()
+Infinity
```

which doesn't prohibit iteration through its elements, though it will be necessary to interrupt it at some point:

```
sage: for p in U:                   # not tested
....:     print(p)
[]
[1]
[1, 2]
[2, 1]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
...
```

```
>>> from sage.all import *
>>> for p in U:                   # not tested
...     print(p)
[]
[1]
[1, 2]
[2, 1]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
...
```

Note: the above set could also have been constructed directly with:

```
sage: U = Permutations(); U
Standard permutations
```

```
>>> from sage.all import *
>>> U = Permutations(); U
Standard permutations
```

Summary

Sage provides a library of common enumerated sets, which can be combined by standard constructions, giving a toolbox that is flexible (but which could still be expanded). It is also possible to add new building blocks to Sage with a few lines (see the code in `FiniteEnumeratedSets().example()`). This is made possible by the uniformity of the interfaces and the fact that Sage is based on an object-oriented language. Also, very large or even infinite sets can be manipulated thanks to lazy evaluation strategies (iterators, etc.).

There is no magic to any of this: under the hood, Sage applies the usual rules (for example, that the cardinality of $E \times E$ is $|E|^2$); the added value comes from the capacity to manipulate complicated constructions. The situation is comparable to Sage's implementation of differential calculus: Sage applies the usual rules for differentiation of functions and their compositions, where the added value comes from the possibility of manipulating complicated formulas. In this sense, Sage implements a *calculus* of finite enumerated sets.

Generic algorithms

Lexicographic generation of lists of integers

Among the classic enumerated sets, especially in algebraic combinatorics, a certain number are composed of lists of integers of fixed sum, such as partitions, compositions, or integer vectors. These examples can also have supplementary constraints added to them. Here are some examples. We start with the integer vectors with sum 10 and length 3, with parts bounded below by 2, 4 and 2 respectively:

```
sage: IntegerVectors(10, 3, min_part=2, max_part=5,
....:                inner=[2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]
```

```
>>> from sage.all import *
>>> IntegerVectors(Integer(10), Integer(3), min_part=Integer(2), max_part=Integer(5),
...                inner=[Integer(2), Integer(4), Integer(2)]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]
```

The compositions of 5 with each part at most 3, and with length 2 or 3:

```
sage: Compositions(5, max_part=3,
....:              min_length=2, max_length=3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [1, 3, 1],
 [1, 2, 2], [1, 1, 3]]
```

```
>>> from sage.all import *
>>> Compositions(Integer(5), max_part=Integer(3),
...              min_length=Integer(2), max_length=Integer(3)).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [1, 3, 1],
 [1, 2, 2], [1, 1, 3]]
```

The strictly decreasing partitions of 5:

```
sage: Partitions(5, max_slope=-1).list()
[[5], [4, 1], [3, 2]]
```

```
>>> from sage.all import *
>>> Partitions(Integer(5), max_slope=-Integer(1)).list()
[[5], [4, 1], [3, 2]]
```

These sets share the same underlying algorithmic structure, implemented in the more general (and slightly more cumbersome) class `IntegerListsLex`. This class models sets of vectors $(\ell_0, \dots, \ell_k)$ of nonnegative integers, with constraints on the sum and the length, and bounds on the parts and on the consecutive differences between the parts. Here are some more examples:

```
sage: IntegerListsLex(10, length=3,
....:                 min_part=2, max_part=5,
....:                 floor=[2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]

sage: IntegerListsLex(5, min_part=1, max_part=3,
....:                 min_length=2, max_length=3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2],
 [1, 3, 1], [1, 2, 2], [1, 1, 3]]

sage: IntegerListsLex(5, min_part=1, max_slope=-1).list()
[[5], [4, 1], [3, 2]]

sage: list(Compositions(5, max_length=2))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]

sage: list(IntegerListsLex(5, max_length=2, min_part=1))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]
```

```
>>> from sage.all import *
>>> IntegerListsLex(Integer(10), length=Integer(3),
...                 min_part=Integer(2), max_part=Integer(5),
...                 floor=[Integer(2), Integer(4), Integer(2)]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]

>>> IntegerListsLex(Integer(5), min_part=Integer(1), max_part=Integer(3),
...                 min_length=Integer(2), max_length=Integer(3)).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2],
 [1, 3, 1], [1, 2, 2], [1, 1, 3]]

>>> IntegerListsLex(Integer(5), min_part=Integer(1), max_slope=-Integer(1)).list()
[[5], [4, 1], [3, 2]]

>>> list(Compositions(Integer(5), max_length=Integer(2)))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]

>>> list(IntegerListsLex(Integer(5), max_length=Integer(2), min_part=Integer(1)))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]
```

The point of the model of `IntegerListsLex` is in the compromise between generality and efficiency. The main algorithm permits iteration through the elements of such a set S in reverse lexicographic order with a good complexity in most practical use cases. Roughly speaking, the time needed to iterate through all the elements of S is proportional to the number of elements, where the proportion factor is controlled by the length l of the longest element of S. In addition, the memory usage is also controlled by l, which is to say negligible in practice.

This algorithm is based on a very general principle for traversing a decision tree, called *branch and bound*: at the top level, we run through all the possible choices for ℓ_0; for each of these choices, we run through all the possible choices for ℓ_1, and so on. Mathematically speaking, we have put the structure of a prefix tree on the elements of S: a node of the tree at depth k corresponds to a prefix $\ell_0, \dots, \ell_k$ of one (or more) elements of S (see *Figure: The prefix tree of the partitions* *of 5.*).

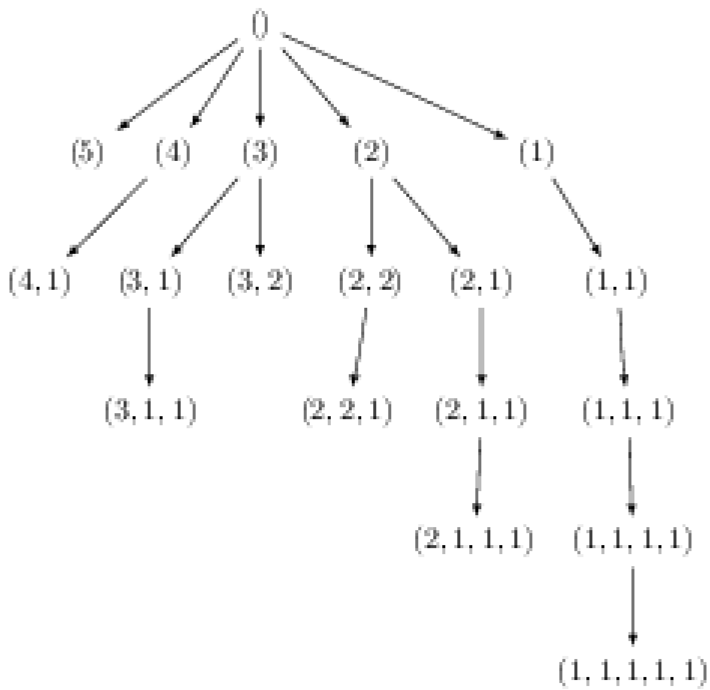


Fig. 2: Figure: The prefix tree of the partitions of 5.

The usual problem with this type of approach is to avoid bad decisions which lead to leaving the prefix tree and exploring dead branches; this is particularly problematic because the growth of the number of elements is usually exponential in the depth. It turns out that the constraints listed above are simple enough to be able to reasonably predict when a sequence $\ell_0, \dots, \ell_k$ is a prefix of some element S. Hence, most dead branches can be pruned.

Integer points in polytopes

Although the algorithm for iteration in `IntegerListsLex` is efficient, its counting algorithm is naive: it just iterates over all the elements.

There is an alternative approach to treating this problem: modelling the desired lists of integers as the set of integer points of a polytope, that is to say, the set of solutions with integer coordinates of a system of linear inequalities. This is a very general context in which there exist advanced counting algorithms (e.g. Barvinok), which are implemented in libraries like `LattE`. Iteration does not pose a hard problem in principle. However, there are two limitations that justify the existence of `IntegerListsLex`. The first is theoretical: lattice points in a polytope only allow modelling of problems of a fixed dimension (length). The second is practical: at the moment only the library `PALP` has a `Sage` interface, and though it offers multiple capabilities for the study of polytopes, in the present application it only produces a list of lattice points, without providing either an iterator or non-naive counting:

```
sage: A = random_matrix(ZZ, 6, 3, x=7)
sage: L = LatticePolytope(A.rows())
sage: L.points()                              # random
M(4, 1, 0),
M(0, 3, 5),
M(2, 2, 3),
M(6, 1, 3),
M(1, 3, 6),
```

(continues on next page)

(continued from previous page)

```
M(6, 2, 3),
M(3, 2, 4),
M(3, 2, 3),
M(4, 2, 4),
M(4, 2, 3),
M(5, 2, 3)
in 3-d lattice M
sage: L.npoints()                                    # random
11
```

```
>>> from sage.all import *
>>> A = random_matrix(ZZ, Integer(6), Integer(3), x=Integer(7))
>>> L = LatticePolytope(A.rows())
>>> L.points()                                  # random
M(4, 1, 0),
M(0, 3, 5),
M(2, 2, 3),
M(6, 1, 3),
M(1, 3, 6),
M(6, 2, 3),
M(3, 2, 4),
M(3, 2, 3),
M(4, 2, 4),
M(4, 2, 3),
M(5, 2, 3)
in 3-d lattice M
>>> L.npoints()                                      # random
11
```

This polytope can be visualized in 3D with `L.plot3d()` (see *Figure: The polytope L and its integer points, in cross-eyed stereographic perspective.*).

Species, decomposable combinatorial classes

In *Enumeration of trees using generating functions*, we showed how to use the recursive definition of binary trees to count them efficiently using generating functions. The techniques we used there are very general, and apply whenever the sets involved can be defined recursively (depending on who you ask, such a set is called a *decomposable combinatorial class* or, roughly speaking, a *combinatorial species*). This includes all the types of trees, but also permutations, compositions, functional graphs, etc.

Here, we illustrate just a few examples using the `Sage` library on combinatorial species:

```
sage: from sage.combinat.species.library import *
sage: o = var('o')                                                          #␣
↪needs sage.symbolic
```

```
>>> from sage.all import *
>>> from sage.combinat.species.library import *
>>> o = var('o')                                                          #␣
↪needs sage.symbolic
```

We begin by redefining the complete binary trees; to do so, we stipulate the recurrence relation directly on the sets:

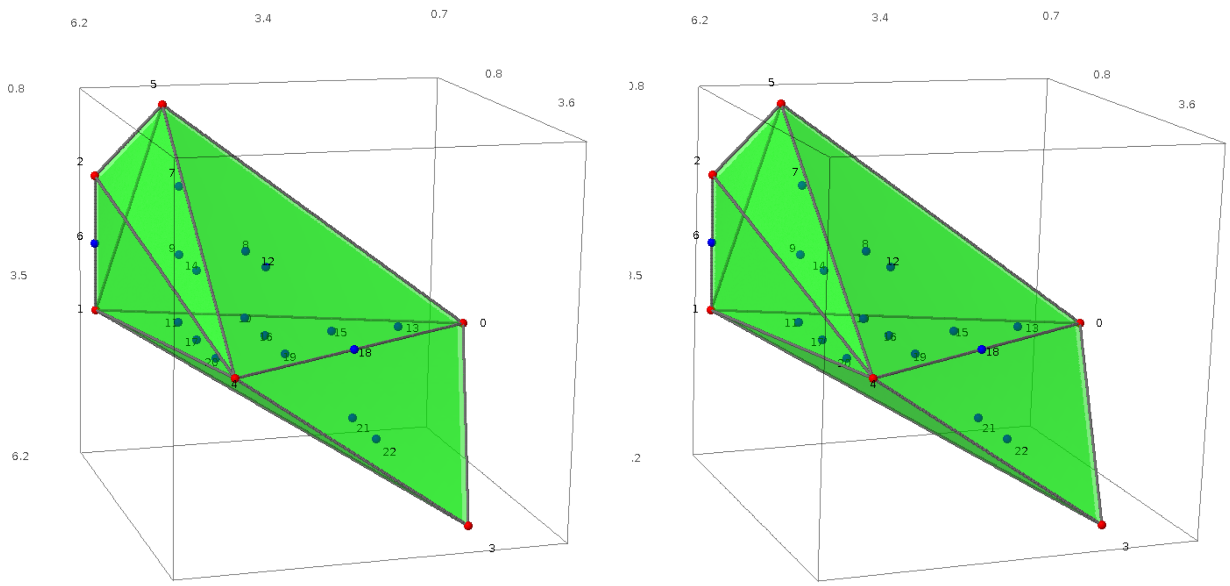


Fig. 3: Figure: The polytope L and its integer points, in cross-eyed stereographic perspective.

```
sage: BT = CombinatorialSpecies(min=1)
sage: Leaf =  SingletonSpecies()
sage: BT.define( Leaf + (BT*BT) )
```

```
>>> from sage.all import *
>>> BT = CombinatorialSpecies(min=Integer(1))
>>> Leaf =  SingletonSpecies()
>>> BT.define( Leaf + (BT*BT) )
```

Now we can construct the set of trees with five nodes, list them, count them…:

```
sage: BT5 = BT.isotypes([o]*5)                                              #␣
↪needs sage.symbolic
sage: BT5.cardinality()                                                     #␣
↪needs sage.symbolic
14
sage: BT5.list()                                                            #␣
↪needs sage.symbolic
[o*(o*(o*(o*o))), o*(o*((o*o)*o)), o*((o*o)*(o*o)),
 o*((o*(o*o))*o), o*(((o*o)*o)*o), (o*o)*(o*(o*o)),
 (o*o)*((o*o)*o), (o*(o*o))*(o*o), ((o*o)*o)*(o*o),
 (o*(o*(o*o)))*o, (o*((o*o)*o))*o, ((o*o)*(o*o))*o,
 ((o*(o*o))*o)*o, (((o*o)*o)*o)*o]
```

```
>>> from sage.all import *
>>> BT5 = BT.isotypes([o]*Integer(5))                                        ␣
↪     # needs sage.symbolic
>>> BT5.cardinality()                                                      #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
14
>>> BT5.list()                                                                      #␣
↪needs sage.symbolic
[o*(o*(o*(o*o))), o*(o*((o*o)*o)), o*((o*o)*(o*o)),
 o*((o*(o*o))*o), o*(((o*o)*o)*o), (o*o)*(o*(o*o)),
 (o*o)*((o*o)*o), (o*(o*o))*(o*o), ((o*o)*o)*(o*o),
 (o*(o*(o*o)))*o, (o*((o*o)*o))*o, ((o*o)*(o*o))*o,
 ((o*(o*o))*o)*o, (((o*o)*o)*o)*o]
```

The trees are constructed using a generic recursive structure; the display is therefore not wonderful. To do better, it would be necessary to provide `Sage` with a more specialized data structure with the desired display capabilities.

We recover the generating function for the Catalan numbers:

```
sage: g = BT.isotype_generating_series(); g
z + z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 + 132*z^7 + O(z^8)
```

```
>>> from sage.all import *
>>> g = BT.isotype_generating_series(); g
z + z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 + 132*z^7 + O(z^8)
```

which is returned in the form of a lazy power series:

```
sage: g[100]
227508830794229349661819540395688853956041682601541047340
```

```
>>> from sage.all import *
>>> g[Integer(100)]
227508830794229349661819540395688853956041682601541047340
```

We finish with the Fibonacci words, which are binary words without two consecutive "1"s. They admit a natural recursive definition:

```
sage: Eps =  EmptySetSpecies()
sage: Z0  =  SingletonSpecies()
sage: Z1  =  Eps*SingletonSpecies()
sage: FW  = CombinatorialSpecies()
sage: FW.define(Eps + Z0*FW  +  Z1*Eps + Z1*Z0*FW)
```

```
>>> from sage.all import *
>>> Eps =  EmptySetSpecies()
>>> Z0  =  SingletonSpecies()
>>> Z1  =  Eps*SingletonSpecies()
>>> FW  = CombinatorialSpecies()
>>> FW.define(Eps + Z0*FW  +  Z1*Eps + Z1*Z0*FW)
```

The Fibonacci sequence is easily recognized here, hence the name:

```
sage: L = FW.isotype_generating_series()[:15]; L
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

```
>>> from sage.all import *
>>> L = FW.isotype_generating_series()[:Integer(15)]; L
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

```
sage: oeis(L)                                    # optional -- internet
0: A000045: Fibonacci numbers: F(n) = F(n-1) + F(n-2) with F(0) = 0 and F(1) = 1.
1: ...
2: ...
```

```
>>> from sage.all import *
>>> oeis(L)                                    # optional -- internet
0: A000045: Fibonacci numbers: F(n) = F(n-1) + F(n-2) with F(0) = 0 and F(1) = 1.
1: ...
2: ...
```

This is an immediate consequence of the recurrence relation. One can also generate immediately all the Fibonacci words of a given length, with the same limitations resulting from the generic display.

```
sage: FW3 = FW.isotypes([o]*3)                                          #␣
↪needs sage.symbolic
sage: FW3.list()                                                        #␣
↪needs sage.symbolic
[o*(o*(o*{})), o*(o*(({}*o)*{})), o*((({}*o)*o)*{}),
 (({}*o)*o)*(o*{}), (({}*o)*o)*(({}*o)*{})]
```

```
>>> from sage.all import *
>>> FW3 = FW.isotypes([o]*Integer(3))                                    ␣
↪     # needs sage.symbolic
>>> FW3.list()                                                         #␣
↪needs sage.symbolic
[o*(o*(o*{})), o*(o*(({}*o)*{})), o*((({}*o)*o)*{}),
 (({}*o)*o)*(o*{}), (({}*o)*o)*(({}*o)*{})]
```

Graphs up to isomorphism

We saw in *Some other finite enumerated sets* that `Sage` could generate graphs and partial orders up to isomorphism. We will now describe the underlying algorithm, which is the same in both cases, and covers a substantially wider class of problems.

We begin by recalling some notions. A graph $G = (V, E)$ is a set V of vertices and a set E of edges connecting these vertices; an edge is described by a pair $\{u, v\}$ of distinct vertices of V. Such a graph is called labelled; its vertices are typically numbered by considering $V = \{1, 2, 3, 4, 5\}$.

In many problems, the labels on the vertices play no role. Typically a chemist wants to study all the possible molecules with a given composition, for example the alkanes with $n = 8$ atoms of carbon and $2n + 2 = 18$ atoms of hydrogen. He therefore wants to find all the graphs consisting of 8 vertices with 4 neighbours, and 18 vertices with a single neighbour. The different carbon atoms, however, are all considered to be identical, and the same for the hydrogen atoms. The problem of our chemist is not imaginary; this type of application is actually at the origin of an important part of the research in graph theory on isomorphism problems.

Working by hand on a small graph it is possible, as in the example of *Some other finite enumerated sets*, to make a drawing, erase the labels, and "forget" the geometrical information about the location of the vertices in the plane. However, to represent a graph in a computer program, it is necessary to introduce labels on the vertices so as to be able to describe how the edges connect them together. To compensate for the extra information which we have introduced, we then say that two labelled graphs g_1 and g_2 are *isomorphic* if there is a bijection from the vertices of g_1 to those of g_2, which maps bijectively the edges of g_1 to those of g_2; an *unlabelled graph* is then an equivalence class of labelled graphs.

In general, testing if two labelled graphs are isomorphic is expensive. However, the number of graphs, even unlabelled, grows very rapidly. Nonetheless, it is possible to list unlabelled graphs very efficiently considering their number. For example, the program `Nauty` can list the 12005168 simple graphs with 10 vertices in 20 seconds.

As in *Lexicographic generation of lists of integers*, the general principle of the algorithm is to organize the objects to be enumerated into a tree that one traverses.

For this, in each equivalence class of labelled graphs (that is to say, for each unlabelled graph) one fixes a convenient canonical representative. The following are the fundamental operations:

- Testing whether a labelled graph is canonical
- Calculating the canonical representative of a labelled graph

These unavoidable operations remain expensive; one therefore tries to minimize the number of calls to them.

The canonical representatives are chosen in such a way that, for each canonical labelled graph G, there is a canonical choice of an edge whose removal produces a canonical graph again, which is called the father of G. This property implies that it is possible to organize the set of canonical representatives as a tree: at the root, the graph with no edges; below it, its unique child, the graph with one edge; then the graphs with two edges, and so on. The set of children of a graph G can be constructed by *augmentation*, adding an edge in all the possible ways to G, and then selecting, from among those graphs, the ones that are still canonical[2]. Recursively, one obtains all the canonical graphs.

In what sense is this algorithm generic? Consider for example planar graphs (graphs which can be drawn in the plane without edges crossing): by removing an edge from a planar graph, one obtains another planar graph; so planar graphs form a subtree of the previous tree. To generate them, exactly the same algorithm can be used, selecting only the children which are planar:

```
sage: [len(list(graphs(n, property=lambda G: G.is_planar())))
....:  for n in range(7)]
[1, 1, 2, 4, 11, 33, 142]
```

```
>>> from sage.all import *
>>> [len(list(graphs(n, property=lambda G: G.is_planar())))
...  for n in range(Integer(7))]
[1, 1, 2, 4, 11, 33, 142]
```

In a similar fashion, one can generate any family of graphs closed under deletion of an edge, and in particular any family characterized by a forbidden subgraph. This includes for example forests (graphs without cycles), bipartite graphs (graphs without odd cycles), etc. This can be applied to generate:

- partial orders, via the bijection with Hasse diagrams which are oriented graphs without cycles and without edges implied by the transitivity of the order relation;
- lattices (not implemented in `Sage`), via the bijection with the meet semi-lattice obtained by deleting the maximal vertex; in this case an augmentation by vertices rather than by edges is used.

REFERENCES:

[2] In practice, an efficient implementation would exploit the symmetries of G, i.e., its automorphism group, to reduce the number of children to explore, and to reduce the cost of each test of canonicity.

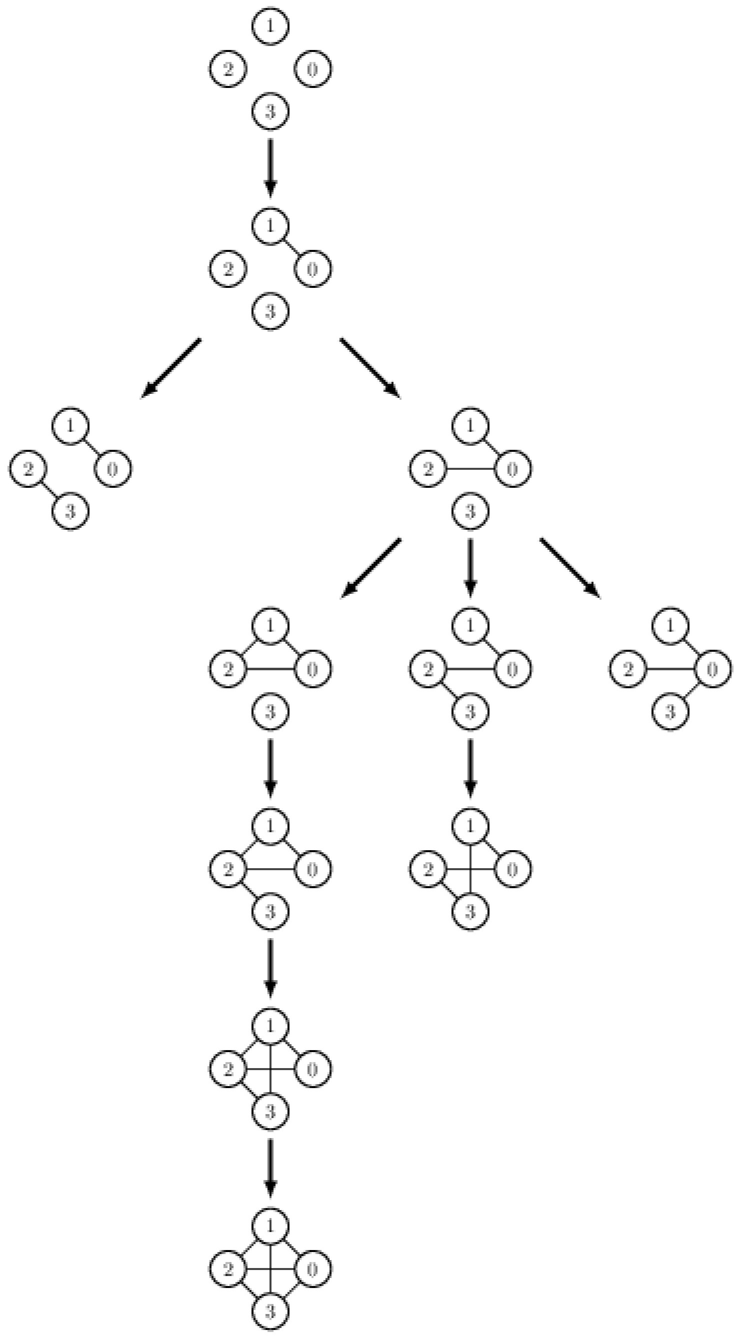


Fig. 4: Figure: The generation tree of simple graphs with 4 vertices.

5.1.354 Vector Partitions

AUTHORS:

- Amritanshu Prasad (2013): Initial version
- Shriya M (2022): Added new parameters such as `distinct`, `parts` and `is_repeatable`

`sage.combinat.vector_partition.`**`IntegerVectorsIterator`**`(vect, min=None)`

Return an iterator over the list of integer vectors which are componentwise less than or equal to `vect`, and lexicographically greater than or equal to `min`.

INPUT:

- `vect` – list of nonnegative integers
- `min` – list of nonnegative integers dominated elementwise by `vect`

OUTPUT:

A list in lexicographic order of all integer vectors (as lists) which are dominated elementwise by `vect` and are greater than or equal to `min` in lexicographic order.

EXAMPLES:

```
sage: from sage.combinat.vector_partition import IntegerVectorsIterator
sage: list(IntegerVectorsIterator([1, 1]))
[[0, 0], [0, 1], [1, 0], [1, 1]]

sage: list(IntegerVectorsIterator([1, 1], min = [1, 0]))
[[1, 0], [1, 1]]
```

```
>>> from sage.all import *
>>> from sage.combinat.vector_partition import IntegerVectorsIterator
>>> list(IntegerVectorsIterator([Integer(1), Integer(1)]))
[[0, 0], [0, 1], [1, 0], [1, 1]]

>>> list(IntegerVectorsIterator([Integer(1), Integer(1)], min = [Integer(1),␣
↪Integer(0)]))
[[1, 0], [1, 1]]
```

class `sage.combinat.vector_partition.`**`VectorPartition`**`(parent, vecpar)`

Bases: `CombinatorialElement`

A vector partition is a multiset of integer vectors.

`partition_at_vertex``(i)`

Return the partition obtained by sorting the `i`-th elements of the vectors in the vector partition.

EXAMPLES:

```
sage: V = VectorPartition([[1, 2, 1], [2, 4, 1]])
sage: V.partition_at_vertex(1)
[4, 2]
```

```
>>> from sage.all import *
>>> V = VectorPartition([[Integer(1), Integer(2), Integer(1)], [Integer(2),␣
↪Integer(4), Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> V.partition_at_vertex(Integer(1))
[4, 2]
```

sum()

Return the sum vector as a list.

EXAMPLES:

```
sage: V = VectorPartition([[3, 2, 1], [2, 2, 1]])
sage: V.sum()
[5, 4, 2]
```

```
>>> from sage.all import *
>>> V = VectorPartition([[Integer(3), Integer(2), Integer(1)], [Integer(2),␣
↪Integer(2), Integer(1)]])
>>> V.sum()
[5, 4, 2]
```

class sage.combinat.vector_partition.**VectorPartitions**(*vec*, *min=None*, *parts=None*, *distinct=False*, *is_repeatable=None*)

Bases: `UniqueRepresentation`, `Parent`

Class of all vector partitions of `vec` with all parts greater than or equal to `min` in lexicographic order, with parts from `parts`.

A vector partition of `vec` is a list of vectors with nonnegative integer entries whose sum is `vec`.

INPUT:

- `vec` – integer vector
- `min` – integer vector dominated elementwise by `vec`
- `parts` – finite list of possible parts
- `distinct` – boolean, set to `True` if only vector partitions with distinct parts are enumerated
- `is_repeatable` – boolean function on `parts` which gives `True` in parts that can be repeated

EXAMPLES:

If `min` is not specified, then the class of all vector partitions of `vec` is created:

```
sage: VP = VectorPartitions([2, 2])
sage: for vecpar in VP:
....:     print(vecpar)
[[0, 1], [0, 1], [1, 0], [1, 0]]
[[0, 1], [0, 1], [2, 0]]
[[0, 1], [1, 0], [1, 1]]
[[0, 1], [2, 1]]
[[0, 2], [1, 0], [1, 0]]
[[0, 2], [2, 0]]
[[1, 0], [1, 2]]
[[1, 1], [1, 1]]
[[2, 2]]
```

```
>>> from sage.all import *
>>> VP = VectorPartitions([Integer(2), Integer(2)])
>>> for vecpar in VP:
...     print(vecpar)
[[0, 1], [0, 1], [1, 0], [1, 0]]
[[0, 1], [0, 1], [2, 0]]
[[0, 1], [1, 0], [1, 1]]
[[0, 1], [2, 1]]
[[0, 2], [1, 0], [1, 0]]
[[0, 2], [2, 0]]
[[1, 0], [1, 2]]
[[1, 1], [1, 1]]
[[2, 2]]
```

If `distinct` is set to be True, then distinct part partitions are created:

```
sage: VP = VectorPartitions([2,2], distinct = True)
sage: list(VP)
[[[0, 1], [1, 0], [1, 1]],
 [[0, 1], [2, 1]],
 [[0, 2], [2, 0]],
 [[1, 0], [1, 2]],
 [[2, 2]]]
```

```
>>> from sage.all import *
>>> VP = VectorPartitions([Integer(2),Integer(2)], distinct = True)
>>> list(VP)
[[[0, 1], [1, 0], [1, 1]],
 [[0, 1], [2, 1]],
 [[0, 2], [2, 0]],
 [[1, 0], [1, 2]],
 [[2, 2]]]
```

If `min` is specified, then the class consists of only those vector partitions whose parts are all greater than or equal to `min` in lexicographic order:

```
sage: VP = VectorPartitions([2, 2], min = [1, 0])
sage: for vecpar in VP:
....:     print(vecpar)
[[1, 0], [1, 2]]
[[1, 1], [1, 1]]
[[2, 2]]
sage: VP = VectorPartitions([2, 2], min = [1, 0], distinct = True)
sage: for vecpar in VP:
....:     print(vecpar)
[[1, 0], [1, 2]]
[[2, 2]]
```

```
>>> from sage.all import *
>>> VP = VectorPartitions([Integer(2), Integer(2)], min = [Integer(1),
→Integer(0)])
>>> for vecpar in VP:
...     print(vecpar)
```

(continues on next page)

(continued from previous page)

```
[[1, 0], [1, 2]]
[[1, 1], [1, 1]]
[[2, 2]]
>>> VP = VectorPartitions([Integer(2), Integer(2)], min = [Integer(1),␣
↪Integer(0)], distinct = True)
>>> for vecpar in VP:
...     print(vecpar)
[[1, 0], [1, 2]]
[[2, 2]]
```

If `parts` is specified, then the class consists only of those vector partitions whose parts are from `parts`:

```
sage: Vec_Par = VectorPartitions([2,2], parts=[[0,1],[1,0],[1,1]])
sage: list(Vec_Par)
[[[0, 1], [0, 1], [1, 0], [1, 0]], [[0, 1], [1, 0], [1, 1]], [[1, 1], [1, 1]]]
```

```
>>> from sage.all import *
>>> Vec_Par = VectorPartitions([Integer(2),Integer(2)], parts=[[Integer(0),
↪Integer(1)],[Integer(1),Integer(0)],[Integer(1),Integer(1)]])
>>> list(Vec_Par)
[[[0, 1], [0, 1], [1, 0], [1, 0]], [[0, 1], [1, 0], [1, 1]], [[1, 1], [1, 1]]]
```

If `is_repeatable` is specified, then the parts which satisfy the boolean function `is_repeatable` are allowed to be repeated:

```
sage: Vector_Partitions = VectorPartitions([2,2], parts=[[0,1],[1,0],[1,1]], is_
↪repeatable=lambda vec: sum(vec)%2!=0)
sage: list(Vector_Partitions)
[[[0, 1], [0, 1], [1, 0], [1, 0]], [[0, 1], [1, 0], [1, 1]]]
```

```
>>> from sage.all import *
>>> Vector_Partitions = VectorPartitions([Integer(2),Integer(2)],␣
↪parts=[[Integer(0),Integer(1)],[Integer(1),Integer(0)],[Integer(1),Integer(1)]],
↪ is_repeatable=lambda vec: sum(vec)%Integer(2)!=Integer(0))
>>> list(Vector_Partitions)
[[[0, 1], [0, 1], [1, 0], [1, 0]], [[0, 1], [1, 0], [1, 1]]]
```

Element

alias of *VectorPartition*

`sage.combinat.vector_partition.`**`find_min`**`(`*vect*`)`

Return a string of `0`'s with one `1` at the location where the list `vect` has its last entry which is not equal to `0`.

INPUT:

- `vec` – list of integers

OUTPUT:

A list of the same length with `0`'s everywhere, except for a `1` at the last position where `vec` has an entry not equal to `0`.

EXAMPLES:

```
sage: from sage.combinat.vector_partition import find_min
sage: find_min([2, 1])
[0, 1]
sage: find_min([2, 1, 0])
[0, 1, 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.vector_partition import find_min
>>> find_min([Integer(2), Integer(1)])
[0, 1]
>>> find_min([Integer(2), Integer(1), Integer(0)])
[0, 1, 0]
```

5.1.355 Abstract word (finite or infinite)

This module gathers functions that works for both finite and infinite words.

AUTHORS:

- Sébastien Labbé
- Franco Saliola

EXAMPLES:

```
sage: a = 0.618
sage: g = words.CodingOfRotationWord(alpha=a, beta=1-a, x=a)
sage: f = words.FibonacciWord()
sage: p = f.longest_common_prefix(g, length='finite')
sage: p
word: 0100101001001010010100100101001001010010...
sage: p.length()
231
```

```
>>> from sage.all import *
>>> a = RealNumber('0.618')
>>> g = words.CodingOfRotationWord(alpha=a, beta=Integer(1)-a, x=a)
>>> f = words.FibonacciWord()
>>> p = f.longest_common_prefix(g, length='finite')
>>> p
word: 0100101001001010010100100101001001010010...
>>> p.length()
231
```

class sage.combinat.words.abstract_word.**Word_class**

Bases: SageObject

apply_morphism(*morphism*)

Return the word obtained by applying the morphism to self.

INPUT:

- morphism – can be an instance of WordMorphism, or anything that can be used to construct one

EXAMPLES:

```
sage: w = Word("ab")
sage: d = {'a':'ab', 'b':'ba'}
sage: w.apply_morphism(d)
word: abba
sage: w.apply_morphism(WordMorphism(d))
word: abba
```

```
>>> from sage.all import *
>>> w = Word("ab")
>>> d = {'a':'ab', 'b':'ba'}
>>> w.apply_morphism(d)
word: abba
>>> w.apply_morphism(WordMorphism(d))
word: abba
```

```
sage: w = Word('ababa')
sage: d = dict(a='ab', b='ba')
sage: d
{'a': 'ab', 'b': 'ba'}
sage: w.apply_morphism(d)
word: abbaabbaab
```

```
>>> from sage.all import *
>>> w = Word('ababa')
>>> d = dict(a='ab', b='ba')
>>> d
{'a': 'ab', 'b': 'ba'}
>>> w.apply_morphism(d)
word: abbaabbaab
```

For infinite words:

```
sage: t = words.ThueMorseWord([0,1]); t
word: 0110100110010110100101100110100110010110...
sage: t.apply_morphism({0:8,1:9})
word: 8998988998898998988989988998988998898998...
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord([Integer(0),Integer(1)]); t
word: 0110100110010110100101100110100110010110...
>>> t.apply_morphism({Integer(0):Integer(8),Integer(1):Integer(9)})
word: 8998988998898998988989988998988998898998...
```

complete_return_words_iterator(*fact*)

Return an iterator over all the complete return words of fact in `self` (without unicity).

A complete return words u of a factor v is a factor starting by the given factor v and ending just after the next occurrence of this factor v. See for instance [1].

INPUT:

- `fact` – a non empty finite word

OUTPUT: iterator

EXAMPLES:

```
sage: TM = words.ThueMorseWord()
sage: fact = Word([0,1,1,0,1])
sage: it = TM.complete_return_words_iterator(fact)
sage: next(it)
word: 01101001100101101
sage: next(it)
word: 01101001011001101
sage: next(it)
word: 011010011001011001101
sage: next(it)
word: 0110100101101
sage: next(it)
word: 01101001100101101
sage: next(it)
word: 01101001011001101
```

```
>>> from sage.all import *
>>> TM = words.ThueMorseWord()
>>> fact = Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1)])
>>> it = TM.complete_return_words_iterator(fact)
>>> next(it)
word: 01101001100101101
>>> next(it)
word: 01101001011001101
>>> next(it)
word: 011010011001011001101
>>> next(it)
word: 0110100101101
>>> next(it)
word: 01101001100101101
>>> next(it)
word: 01101001011001101
```

REFERENCES:

- [1] J. Justin, L. Vuillon, Return words in Sturmian and episturmian words, Theor. Inform. Appl. 34 (2000) 343–356.

delta()

Return the image of `self` under the delta morphism.

This is the word composed of the length of consecutive runs of the same letter in a given word.

OUTPUT: word over integers

EXAMPLES:

For finite words:

```
sage: W = Words('0123456789')
sage: W('22112122').delta()
word: 22112
sage: W('555008').delta()
word: 321
```

(continues on next page)

(continued from previous page)

```
sage: W().delta()
word:
sage: Word('aabbabaa').delta()
word: 22112
```

```
>>> from sage.all import *
>>> W = Words('0123456789')
>>> W('22112122').delta()
word: 22112
>>> W('555008').delta()
word: 321
>>> W().delta()
word:
>>> Word('aabbabaa').delta()
word: 22112
```

For infinite words:

```
sage: t = words.ThueMorseWord()
sage: t.delta()
word: 1211222112112112221122211222112112112221...
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord()
>>> t.delta()
word: 1211222112112112221122211222112112112221...
```

factor_occurrences_iterator(*fact*)

Return an iterator over all occurrences (including overlapping ones) of fact in `self` in their order of appearance.

INPUT:

- `fact` – a non empty finite word

OUTPUT: iterator

EXAMPLES:

```
sage: TM = words.ThueMorseWord()
sage: fact = Word([0,1,1,0,1])
sage: it = TM.factor_occurrences_iterator(fact)
sage: next(it)
0
sage: next(it)
12
sage: next(it)
24
```

```
>>> from sage.all import *
>>> TM = words.ThueMorseWord()
>>> fact = Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1)])
>>> it = TM.factor_occurrences_iterator(fact)
```

(continues on next page)

(continued from previous page)

```
>>> next(it)
0
>>> next(it)
12
>>> next(it)
24
```

```
sage: u = Word('121')
sage: w = Word('121213211213')
sage: list(w.factor_occurrences_iterator(u))
[0, 2, 8]
```

```
>>> from sage.all import *
>>> u = Word('121')
>>> w = Word('121213211213')
>>> list(w.factor_occurrences_iterator(u))
[0, 2, 8]
```

finite_differences(*mod=None*)

Return the word obtained by the differences of consecutive letters of `self`.

INPUT:

- `self` – a word over the integers
- **mod – (default: None) it can be one of the following:**
  - None or 0 : result is over the integers
  - integer : result is over the integers modulo `mod`.

EXAMPLES:

```
sage: w = Word([x^2 for x in range(10)])
sage: w.finite_differences()
word: 1,3,5,7,9,11,13,15,17
sage: w.finite_differences(mod=4)
word: 131313131
sage: w.finite_differences(mod=0)
word: 1,3,5,7,9,11,13,15,17
```

```
>>> from sage.all import *
>>> w = Word([x**Integer(2) for x in range(Integer(10))])
>>> w.finite_differences()
word: 1,3,5,7,9,11,13,15,17
>>> w.finite_differences(mod=Integer(4))
word: 131313131
>>> w.finite_differences(mod=Integer(0))
word: 1,3,5,7,9,11,13,15,17
```

first_occurrence(*other*, *start=0*)

Return the position of the first occurrence of `other` in `self`.

If `other` is not a factor of `self`, it returns `None` or loops forever when `self` is an infinite word.

INPUT:

- other – a finite word
- start – integer (default: 0) where the search starts

OUTPUT: integer or None

EXAMPLES:

```
sage: w = Word('01234567890123456789')
sage: w.first_occurrence(Word('3456'))
3
sage: w.first_occurrence(Word('3456'), start=7)
13
```

```
>>> from sage.all import *
>>> w = Word('01234567890123456789')
>>> w.first_occurrence(Word('3456'))
3
>>> w.first_occurrence(Word('3456'), start=Integer(7))
13
```

When the factor is not present, None is returned:

```
sage: w.first_occurrence(Word('3456'), start=17) is None
True
sage: w.first_occurrence(Word('3333')) is None
True
```

```
>>> from sage.all import *
>>> w.first_occurrence(Word('3456'), start=Integer(17)) is None
True
>>> w.first_occurrence(Word('3333')) is None
True
```

Also works for searching a finite word in an infinite word:

```
sage: w = Word('0123456789')^oo
sage: w.first_occurrence(Word('3456'))
3
sage: w.first_occurrence(Word('3456'), start=1000)
1003
```

```
>>> from sage.all import *
>>> w = Word('0123456789')**oo
>>> w.first_occurrence(Word('3456'))
3
>>> w.first_occurrence(Word('3456'), start=Integer(1000))
1003
```

But it will loop for ever if the factor is not found:

```
sage: w.first_occurrence(Word('3333')) # not tested -- infinite loop
```

```
>>> from sage.all import *
>>> w.first_occurrence(Word('3333')) # not tested -- infinite loop
```

The empty word occurs in a word:

```
sage: Word('123').first_occurrence(Word(''), 0)
0
sage: Word('').first_occurrence(Word(''), 0)
0
```

```
>>> from sage.all import *
>>> Word('123').first_occurrence(Word(''), Integer(0))
0
>>> Word('').first_occurrence(Word(''), Integer(0))
0
```

is_empty()

Return `True` if the length of `self` is zero, and `False` otherwise.

EXAMPLES:

```
sage: it = iter([])
sage: Word(it).is_empty()
True
sage: it = iter([1,2,3])
sage: Word(it).is_empty()
False
sage: from itertools import count
sage: Word(count()).is_empty()
False
```

```
>>> from sage.all import *
>>> it = iter([])
>>> Word(it).is_empty()
True
>>> it = iter([Integer(1),Integer(2),Integer(3)])
>>> Word(it).is_empty()
False
>>> from itertools import count
>>> Word(count()).is_empty()
False
```

is_finite()

Return whether this word is known to be finite.

Warning

A word defined by an iterator such that its end has never been reached will returns False.

EXAMPLES:

```
sage: Word([]).is_finite()
True
sage: Word('a').is_finite()
True
```

(continues on next page)

(continued from previous page)

```
sage: TM = words.ThueMorseWord()
sage: TM.is_finite()
False
```

```
>>> from sage.all import *
>>> Word([]).is_finite()
True
>>> Word('a').is_finite()
True
>>> TM = words.ThueMorseWord()
>>> TM.is_finite()
False
```

```
sage: w = Word(iter('a'*100))
sage: w.is_finite()
False
```

```
>>> from sage.all import *
>>> w = Word(iter('a'*Integer(100)))
>>> w.is_finite()
False
```

iterated_right_palindromic_closure(*f=None*, *algorithm='recursive'*)

Return the iterated (f-)palindromic closure of `self`.

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`; it must be callable on letters as well as words (e.g. WordMorphism)
- `algorithm` – string (default: `'recursive'`); specifying which algorithm to be used when computing the iterated palindromic closure. It must be one of the two following values:
  - `'definition'` – computed using the definition
  - `'recursive'` – computation based on an efficient formula that recursively computes the iterated right palindromic closure without having to recompute the longest f-palindromic suffix at each iteration [2].

OUTPUT: word; the iterated (f-)palindromic closure of `self`

EXAMPLES:

```
sage: Word('123').iterated_right_palindromic_closure()
word: 1213121
```

```
>>> from sage.all import *
>>> Word('123').iterated_right_palindromic_closure()
word: 1213121
```

```
sage: w = Word('abc')
sage: w.iterated_right_palindromic_closure()
word: abacaba
```

```
>>> from sage.all import *
>>> w = Word('abc')
>>> w.iterated_right_palindromic_closure()
word: abacaba
```

```
sage: w = Word('aaa')
sage: w.iterated_right_palindromic_closure()
word: aaa
```

```
>>> from sage.all import *
>>> w = Word('aaa')
>>> w.iterated_right_palindromic_closure()
word: aaa
```

```
sage: w = Word('abbab')
sage: w.iterated_right_palindromic_closure()
word: ababaabababaababa
```

```
>>> from sage.all import *
>>> w = Word('abbab')
>>> w.iterated_right_palindromic_closure()
word: ababaabababaababa
```

A right f-palindromic closure:

```
sage: f = WordMorphism('a->b,b->a')
sage: w = Word('abbab')
sage: w.iterated_right_palindromic_closure(f=f)
word: abbaabbaababbaabbaabbaababbaabbaab
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> w = Word('abbab')
>>> w.iterated_right_palindromic_closure(f=f)
word: abbaabbaababbaabbaabbaababbaabbaab
```

An infinite word:

```
sage: t = words.ThueMorseWord('ab')
sage: t.iterated_right_palindromic_closure()
word: ababaabababaababaabababaababaababaabab...
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord('ab')
>>> t.iterated_right_palindromic_closure()
word: ababaabababaababaabababaababaababaabab...
```

There are two implementations computing the iterated right f-palindromic closure, the latter being much more efficient:

```
sage: w = Word('abaab')
sage: u = w.iterated_right_palindromic_closure(algorithm='definition')
```

(continues on next page)

(continued from previous page)

```
sage: v = w.iterated_right_palindromic_closure(algorithm='recursive')
sage: u
word: abaabaababaabaaba
sage: u == v
True
sage: w = words.RandomWord(8)
sage: u = w.iterated_right_palindromic_closure(algorithm='definition')
sage: v = w.iterated_right_palindromic_closure(algorithm='recursive')
sage: u == v
True
```

```
>>> from sage.all import *
>>> w = Word('abaab')
>>> u = w.iterated_right_palindromic_closure(algorithm='definition')
>>> v = w.iterated_right_palindromic_closure(algorithm='recursive')
>>> u
word: abaabaababaabaaba
>>> u == v
True
>>> w = words.RandomWord(Integer(8))
>>> u = w.iterated_right_palindromic_closure(algorithm='definition')
>>> v = w.iterated_right_palindromic_closure(algorithm='recursive')
>>> u == v
True
```

REFERENCES:

- [1] A. de Luca, A. De Luca, Pseudopalindrome closure operators in free monoids, Theoret. Comput. Sci. 362 (2006) 282–300.
- [2] J. Justin, Episturmian morphisms and a Galois theorem on continued fractions, RAIRO Theoret. Informatics Appl. 39 (2005) 207-215.

length()

Return the length of `self`.

lex_greater(*other*)

Return `True` if `self` is lexicographically greater than `other`.

EXAMPLES:

```
sage: w = Word([1,2,3])
sage: u = Word([1,3,2])
sage: v = Word([3,2,1])
sage: w.lex_greater(u)
False
sage: v.lex_greater(w)
True
sage: a = Word("abba")
sage: b = Word("abbb")
sage: a.lex_greater(b)
False
sage: b.lex_greater(a)
True
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(2),Integer(3)])
>>> u = Word([Integer(1),Integer(3),Integer(2)])
>>> v = Word([Integer(3),Integer(2),Integer(1)])
>>> w.lex_greater(u)
False
>>> v.lex_greater(w)
True
>>> a = Word("abba")
>>> b = Word("abbb")
>>> a.lex_greater(b)
False
>>> b.lex_greater(a)
True
```

For infinite words:

```
sage: t = words.ThueMorseWord()
sage: t[:10].lex_greater(t)
False
sage: t.lex_greater(t[:10])
True
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord()
>>> t[:Integer(10)].lex_greater(t)
False
>>> t.lex_greater(t[:Integer(10)])
True
```

lex_less(*other*)

Return `True` if `self` is lexicographically less than `other`.

EXAMPLES:

```
sage: w = Word([1,2,3])
sage: u = Word([1,3,2])
sage: v = Word([3,2,1])
sage: w.lex_less(u)
True
sage: v.lex_less(w)
False
sage: a = Word("abba")
sage: b = Word("abbb")
sage: a.lex_less(b)
True
sage: b.lex_less(a)
False
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(2),Integer(3)])
>>> u = Word([Integer(1),Integer(3),Integer(2)])
>>> v = Word([Integer(3),Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> w.lex_less(u)
True
>>> v.lex_less(w)
False
>>> a = Word("abba")
>>> b = Word("abbb")
>>> a.lex_less(b)
True
>>> b.lex_less(a)
False
```

For infinite words:

```
sage: t = words.ThueMorseWord()
sage: t.lex_less(t[:10])
False
sage: t[:10].lex_less(t)
True
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord()
>>> t.lex_less(t[:Integer(10)])
False
>>> t[:Integer(10)].lex_less(t)
True
```

longest_common_prefix(*other*, *length='unknown'*)

Return the longest common prefix of `self` and `other`.

INPUT:

- `other` – word
- `length` – string (default: `'unknown'`) the length type of the resulting word if known. It may be one of the following:
  - `'unknown'`
  - `'finite'`
  - `'infinite'`

EXAMPLES:

```
sage: f = lambda n : add(Integer(n).digits(2)) % 2
sage: t = Word(f)
sage: u = t[:10]
sage: t.longest_common_prefix(u)
word: 0110100110
```

```
>>> from sage.all import *
>>> f = lambda n : add(Integer(n).digits(Integer(2))) % Integer(2)
>>> t = Word(f)
>>> u = t[:Integer(10)]
```

(continues on next page)

(continued from previous page)

```
>>> t.longest_common_prefix(u)
word: 0110100110
```

The longest common prefix of two equal infinite words:

```
sage: t1 = Word(f)
sage: t2 = Word(f)
sage: t1.longest_common_prefix(t2)
word: 0110100110010110100101100110100110010110...
```

```
>>> from sage.all import *
>>> t1 = Word(f)
>>> t2 = Word(f)
>>> t1.longest_common_prefix(t2)
word: 0110100110010110100101100110100110010110...
```

Useful to study the approximation of an infinite word:

```
sage: a = 0.618
sage: g = words.CodingOfRotationWord(alpha=a, beta=1-a, x=a)
sage: f = words.FibonacciWord()
sage: p = f.longest_common_prefix(g, length='finite')
sage: p.length()
231
```

```
>>> from sage.all import *
>>> a = RealNumber('0.618')
>>> g = words.CodingOfRotationWord(alpha=a, beta=Integer(1)-a, x=a)
>>> f = words.FibonacciWord()
>>> p = f.longest_common_prefix(g, length='finite')
>>> p.length()
231
```

longest_periodic_prefix(*period=1*)

Return the longest prefix of `self` having the given period.

INPUT:

- `period` – positive integer (default: 1)

OUTPUT: word

EXAMPLES:

```
sage: Word([]).longest_periodic_prefix()
word:
sage: Word([1]).longest_periodic_prefix()
word: 1
sage: Word([1,2]).longest_periodic_prefix()
word: 1
sage: Word([1,1,2]).longest_periodic_prefix()
word: 11
sage: Word([1,2,1,2,1,3]).longest_periodic_prefix(2)
word: 12121
```

(continues on next page)

(continued from previous page)

```
sage: type(_)
<class 'sage.combinat.words.word.FiniteWord_iter_with_caching'>
sage: Word(lambda n:0).longest_periodic_prefix()
word: 0000000000000000000000000000000000000000...
```

```
>>> from sage.all import *
>>> Word([]).longest_periodic_prefix()
word:
>>> Word([Integer(1)]).longest_periodic_prefix()
word: 1
>>> Word([Integer(1),Integer(2)]).longest_periodic_prefix()
word: 1
>>> Word([Integer(1),Integer(1),Integer(2)]).longest_periodic_prefix()
word: 11
>>> Word([Integer(1),Integer(2),Integer(1),Integer(2),Integer(1),Integer(3)]).
↪longest_periodic_prefix(Integer(2))
word: 12121
>>> type(_)
<class 'sage.combinat.words.word.FiniteWord_iter_with_caching'>
>>> Word(lambda n:Integer(0)).longest_periodic_prefix()
word: 0000000000000000000000000000000000000000...
```

palindrome_prefixes_iterator(*max_length=None*)

Return an iterator over the palindrome prefixes of `self`.

INPUT:

- `max_length` – nonnegative integer or `None` (default); the maximum length of the prefixes

OUTPUT: iterator

EXAMPLES:

```
sage: w = Word('abaaba')
sage: for pp in w.palindrome_prefixes_iterator(): pp
word:
word: a
word: aba
word: abaaba
sage: for pp in w.palindrome_prefixes_iterator(max_length=4): pp
word:
word: a
word: aba
```

```
>>> from sage.all import *
>>> w = Word('abaaba')
>>> for pp in w.palindrome_prefixes_iterator(): pp
word:
word: a
word: aba
word: abaaba
>>> for pp in w.palindrome_prefixes_iterator(max_length=Integer(4)): pp
word:
```

(continues on next page)

(continued from previous page)

```
word: a
word: aba
```

You can iterate over the palindrome prefixes of an infinite word:

```
sage: f = words.FibonacciWord()
sage: for pp in f.palindrome_prefixes_iterator(max_length=20): pp
word:
word: 0
word: 010
word: 010010
word: 01001010010
word: 0100101001001010010
```

```
>>> from sage.all import *
>>> f = words.FibonacciWord()
>>> for pp in f.palindrome_prefixes_iterator(max_length=Integer(20)): pp
word:
word: 0
word: 010
word: 010010
word: 01001010010
word: 0100101001001010010
```

parent()

Return the parent of `self`.

partial_sums(*start*, *mod=None*)

Return the word defined by the partial sums of its prefixes.

INPUT:

- `self` – a word over the integers
- `start` – integer; the first letter of the resulting word
- **mod – (default: None) it can be one of the following:**
  - None or 0 : result is over the integers
  - integer : result is over the integers modulo `mod`.

EXAMPLES:

```
sage: w = Word(range(10))
sage: w.partial_sums(0)
word: 0,0,1,3,6,10,15,21,28,36,45
sage: w.partial_sums(1)
word: 1,1,2,4,7,11,16,22,29,37,46
```

```
>>> from sage.all import *
>>> w = Word(range(Integer(10)))
>>> w.partial_sums(Integer(0))
word: 0,0,1,3,6,10,15,21,28,36,45
>>> w.partial_sums(Integer(1))
word: 1,1,2,4,7,11,16,22,29,37,46
```

```
sage: w = Word([1,2,3,1,2,3,2,2,2,2])
sage: w.partial_sums(0, mod=None)
word: 0,1,3,6,7,9,12,14,16,18,20
sage: w.partial_sums(0, mod=0)
word: 0,1,3,6,7,9,12,14,16,18,20
sage: w.partial_sums(0, mod=8)
word: 01367146024
sage: w.partial_sums(0, mod=4)
word: 01323102020
sage: w.partial_sums(0, mod=2)
word: 01101100000
sage: w.partial_sums(0, mod=1)
word: 00000000000
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(2),Integer(3),Integer(1),Integer(2),
↪Integer(3),Integer(2),Integer(2),Integer(2),Integer(2)])
>>> w.partial_sums(Integer(0), mod=None)
word: 0,1,3,6,7,9,12,14,16,18,20
>>> w.partial_sums(Integer(0), mod=Integer(0))
word: 0,1,3,6,7,9,12,14,16,18,20
>>> w.partial_sums(Integer(0), mod=Integer(8))
word: 01367146024
>>> w.partial_sums(Integer(0), mod=Integer(4))
word: 01323102020
>>> w.partial_sums(Integer(0), mod=Integer(2))
word: 01101100000
>>> w.partial_sums(Integer(0), mod=Integer(1))
word: 00000000000
```

prefixes_iterator(*max_length=None*)

Return an iterator over the prefixes of `self`.

INPUT:

- `max_length` – nonnegative integer or `None` (default); the maximum length of the prefixes

OUTPUT: iterator

EXAMPLES:

```
sage: w = Word('abaaba')
sage: for p in w.prefixes_iterator(): p
word:
word: a
word: ab
word: aba
word: abaa
word: abaab
word: abaaba
sage: for p in w.prefixes_iterator(max_length=3): p
word:
word: a
word: ab
word: aba
```

```
>>> from sage.all import *
>>> w = Word('abaaba')
>>> for p in w.prefixes_iterator(): p
word:
word: a
word: ab
word: aba
word: abaa
word: abaab
word: abaaba
>>> for p in w.prefixes_iterator(max_length=Integer(3)): p
word:
word: a
word: ab
word: aba
```

You can iterate over the prefixes of an infinite word:

```
sage: f = words.FibonacciWord()
sage: for p in f.prefixes_iterator(max_length=8): p
word:
word: 0
word: 01
word: 010
word: 0100
word: 01001
word: 010010
word: 0100101
word: 01001010
```

```
>>> from sage.all import *
>>> f = words.FibonacciWord()
>>> for p in f.prefixes_iterator(max_length=Integer(8)): p
word:
word: 0
word: 01
word: 010
word: 0100
word: 01001
word: 010010
word: 0100101
word: 01001010
```

return_words_iterator(*fact*)

Return an iterator over all the return words of fact in self (without unicity).

INPUT:

- `fact` – a non empty finite word

OUTPUT: iterator

EXAMPLES:

```
sage: w = Word('baccabccbacbca')
sage: b = Word('b')
sage: list(w.return_words_iterator(b))
[word: bacca, word: bcc, word: bac]
```

```
>>> from sage.all import *
>>> w = Word('baccabccbacbca')
>>> b = Word('b')
>>> list(w.return_words_iterator(b))
[word: bacca, word: bcc, word: bac]
```

```
sage: TM = words.ThueMorseWord()
sage: fact = Word([0,1,1,0,1])
sage: it = TM.return_words_iterator(fact)
sage: next(it)
word: 011010011001
sage: next(it)
word: 011010010110
sage: next(it)
word: 0110100110010110
sage: next(it)
word: 01101001
sage: next(it)
word: 011010011001
sage: next(it)
word: 011010010110
```

```
>>> from sage.all import *
>>> TM = words.ThueMorseWord()
>>> fact = Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1)])
>>> it = TM.return_words_iterator(fact)
>>> next(it)
word: 011010011001
>>> next(it)
word: 011010010110
>>> next(it)
word: 0110100110010110
>>> next(it)
word: 01101001
>>> next(it)
word: 011010011001
>>> next(it)
word: 011010010110
```

string_rep()

Return the (truncated) raw sequence of letters as a string.

EXAMPLES:

```
sage: Word('abbabaab').string_rep()
'abbabaab'
sage: Word([0, 1, 0, 0, 1]).string_rep()
```

(continues on next page)

(continued from previous page)

```
'01001'
sage: Word([0,1,10,101]).string_rep()
'0,1,10,101'
sage: WordOptions(letter_separator='-')
sage: Word([0,1,10,101]).string_rep()
'0-1-10-101'
sage: WordOptions(letter_separator=',')
```

```
>>> from sage.all import *
>>> Word('abbabaab').string_rep()
'abbabaab'
>>> Word([Integer(0), Integer(1), Integer(0), Integer(0), Integer(1)]).string_
↪rep()
'01001'
>>> Word([Integer(0),Integer(1),Integer(10),Integer(101)]).string_rep()
'0,1,10,101'
>>> WordOptions(letter_separator='-')
>>> Word([Integer(0),Integer(1),Integer(10),Integer(101)]).string_rep()
'0-1-10-101'
>>> WordOptions(letter_separator=',')
```

sum_digits(*base=2*, *mod=None*)

Return the sequence of the sum modulo `mod` of the digits written in base `base` of `self`.

INPUT:

- `self` – word over natural numbers
- `base` – integer (default: 2) greater or equal to 2
- `mod` – modulo (default: `None`); can take the following values: - `integer` – the modulo - `None` – the value `base` is considered for the modulo

EXAMPLES:

The Thue-Morse word:

```
sage: from itertools import count
sage: Word(count()).sum_digits()
word: 0110100110010110100101100110100110010110...
```

```
>>> from sage.all import *
>>> from itertools import count
>>> Word(count()).sum_digits()
word: 0110100110010110100101100110100110010110...
```

Sum of digits modulo 2 of the prime numbers written in base 2:

```
sage: Word(primes(1000)).sum_digits()                                      #␣
↪needs sage.libs.pari
word: 1001110100111010111011001011101110011011...
```

```
>>> from sage.all import *
>>> Word(primes(Integer(1000))).sum_digits()                                ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.libs.pari
word: 1001110100111010111011001011101110011011...
```

Sum of digits modulo 3 of the prime numbers written in base 3:

```
sage: Word(primes(1000)).sum_digits(base=3)                                    #␣
↪needs sage.libs.pari
word: 2100002020002221222121022221022122111022...
sage: Word(primes(1000)).sum_digits(base=3, mod=3)                             #␣
↪needs sage.libs.pari
word: 2100002020002221222121022221022122111022...
```

```
>>> from sage.all import *
>>> Word(primes(Integer(1000))).sum_digits(base=Integer(3))                    ␣
↪              # needs sage.libs.pari
word: 2100002020002221222121022221022122111022...
>>> Word(primes(Integer(1000))).sum_digits(base=Integer(3), mod=Integer(3))    ␣
↪                        # needs sage.libs.pari
word: 2100002020002221222121022221022122111022...
```

Sum of digits modulo 2 of the prime numbers written in base 3:

```
sage: Word(primes(1000)).sum_digits(base=3, mod=2)                             #␣
↪needs sage.libs.pari
word: 0111111111111111111111111111111111111111...
```

```
>>> from sage.all import *
>>> Word(primes(Integer(1000))).sum_digits(base=Integer(3), mod=Integer(2))    ␣
↪                        # needs sage.libs.pari
word: 0111111111111111111111111111111111111111...
```

Sum of digits modulo 7 of the prime numbers written in base 10:

```
sage: Word(primes(1000)).sum_digits(base=10, mod=7)                            #␣
↪needs sage.libs.pari
word: 2350241354435041006132432241353546006304...
```

```
>>> from sage.all import *
>>> Word(primes(Integer(1000))).sum_digits(base=Integer(10), mod=Integer(7))   ␣
↪                        # needs sage.libs.pari
word: 2350241354435041006132432241353546006304...
```

Negative entries:

```
sage: w = Word([-1,0,1,2,3,4,5])
sage: w.sum_digits()
Traceback (most recent call last):
...
NotImplementedError: nth digit of Thue-Morse word is not implemented for␣
↪negative value of n
```

```
>>> from sage.all import *
>>> w = Word([-Integer(1),Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)])
>>> w.sum_digits()
Traceback (most recent call last):
...
NotImplementedError: nth digit of Thue-Morse word is not implemented for␣
↪negative value of n
```

to_integer_word()

Return a word over the integers whose letters are those output by self._to_integer_iterator()

EXAMPLES:

```
sage: from itertools import count
sage: w = Word(count()); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
↪27,28,29,30,31,32,33,34,35,36,37,38,39,...
sage: w.to_integer_word()
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
↪27,28,29,30,31,32,33,34,35,36,37,38,39,...
sage: w = Word(iter("abbacabba"), length='finite'); w
word: abbacabba
sage: w.to_integer_word()
word: 011020110
sage: w = Word(iter("abbacabba"), length='unknown'); w
word: abbacabba
sage: w.to_integer_word()
word: 011020110
```

```
>>> from sage.all import *
>>> from itertools import count
>>> w = Word(count()); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
↪27,28,29,30,31,32,33,34,35,36,37,38,39,...
>>> w.to_integer_word()
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
↪27,28,29,30,31,32,33,34,35,36,37,38,39,...
>>> w = Word(iter("abbacabba"), length='finite'); w
word: abbacabba
>>> w.to_integer_word()
word: 011020110
>>> w = Word(iter("abbacabba"), length='unknown'); w
word: abbacabba
>>> w.to_integer_word()
word: 011020110
```

5.1.356 Combinatorics on words

Main modules and their methods:

- *Abstract word (finite or infinite)*
- *Finite word*

- *Infinite word*
- *Alphabet*
- *Set of words*
- *Word paths*
- *Word morphisms/substitutions*
- *Shuffle product of words*
- *Suffix Tries and Suffix Trees*

Main classes and functions meant to be used by the user:

`Word()`, `FiniteWords`, `InfiniteWords`, `Words()`, `Alphabet()`, `WordMorphism`, `WordPaths`.

A list of common words can be accessed through `words.<tab>` and are listed in the *words catalog*.

Internal representation of words:

- *Word classes*
- *Fast word datatype using an array of unsigned char*
- *Datatypes for finite words*
- *Datatypes for words defined by iterators and callables*

Options:

- *User-customizable options for words*

See `WordOptions()`.

5.1.357 Alphabet

AUTHORS:

- Franco Saliola (2008-12-17) : merged into sage
- Vincent Delecroix and Stepan Starosta (2012): remove classes for alphabet and use other Sage classes otherwise (TotallyOrderedFiniteSet, FiniteEnumeratedSet, …). More shortcut to standard alphabets.

EXAMPLES:

```
sage: build_alphabet("ab")
{'a', 'b'}
sage: build_alphabet([0,1,2])
{0, 1, 2}
sage: build_alphabet(name='PP')
Positive integers
sage: build_alphabet(name='NN')
Non negative integers
sage: build_alphabet(name='lower')
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q',
↪'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}
```

```
>>> from sage.all import *
>>> build_alphabet("ab")
{'a', 'b'}
>>> build_alphabet([Integer(0),Integer(1),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
{0, 1, 2}
>>> build_alphabet(name='PP')
Positive integers
>>> build_alphabet(name='NN')
Non negative integers
>>> build_alphabet(name='lower')
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q',
↪'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}
```

sage.combinat.words.alphabet.**Alphabet**(*data=None*, *names=None*, *name=None*)

Return an object representing an ordered alphabet.

INPUT:

- `data` – the letters of the alphabet; it can be:
  - a list/tuple/iterable of letters; the iterable may be infinite
  - an integer n to represent $\{1,\ldots,n\}$, or infinity to represent $\mathbf{N}$
- `names` – (optional) a list for the letters (i.e. variable names) or a string for prefix for all letters; if given a list, it must have the same cardinality as the set represented by `data`
- `name` – (optional) if given, then return a named set and can be equal to : `'lower'`, `'upper'`, `'space'`, `'underscore'`, `'punctuation'`, `'printable'`, `'binary'`, `'octal'`, `'decimal'`, `'hexadecimal'`, `'radix64'`.

  You can use many of them at once, separated by spaces : `'lower punctuation'` represents the union of the two alphabets `'lower'` and `'punctuation'`.

  Alternatively, `name` can be set to `'positive integers'` (or `'PP'`) or `'natural numbers'` (or `'NN'`).

  `name` cannot be combined with `data`.

EXAMPLES:

If the argument is a Set, it just returns it:

```
sage: build_alphabet(ZZ) is ZZ
True
sage: F = FiniteEnumeratedSet('abc')
sage: build_alphabet(F) is F
True
```

```
>>> from sage.all import *
>>> build_alphabet(ZZ) is ZZ
True
>>> F = FiniteEnumeratedSet('abc')
>>> build_alphabet(F) is F
True
```

If a list, tuple or string is provided, then it builds a proper Sage class (`TotallyOrderedFiniteSet`):

```
sage: build_alphabet([0,1,2])
{0, 1, 2}
sage: F = build_alphabet('abc'); F
{'a', 'b', 'c'}
```

(continues on next page)

(continued from previous page)

```
sage: print(type(F).__name__)
TotallyOrderedFiniteSet_with_category
```

```
>>> from sage.all import *
>>> build_alphabet([Integer(0),Integer(1),Integer(2)])
{0, 1, 2}
>>> F = build_alphabet('abc'); F
{'a', 'b', 'c'}
>>> print(type(F).__name__)
TotallyOrderedFiniteSet_with_category
```

If an integer and a set is given, then it constructs a `TotallyOrderedFiniteSet`:

```
sage: build_alphabet(3, ['a','b','c'])
{'a', 'b', 'c'}
```

```
>>> from sage.all import *
>>> build_alphabet(Integer(3), ['a','b','c'])
{'a', 'b', 'c'}
```

If an integer and a string is given, then it considers that string as a prefix:

```
sage: build_alphabet(3, 'x')
{'x0', 'x1', 'x2'}
```

```
>>> from sage.all import *
>>> build_alphabet(Integer(3), 'x')
{'x0', 'x1', 'x2'}
```

If no data is provided, `name` may be a string which describe an alphabet. The available names decompose into two families. The first one are 'positive integers', 'PP', 'natural numbers' or 'NN' which refer to standard set of numbers:

```
sage: build_alphabet(name="positive integers")
Positive integers
sage: build_alphabet(name='PP')
Positive integers
sage: build_alphabet(name="natural numbers")
Non negative integers
sage: build_alphabet(name="NN")
Non negative integers
```

```
>>> from sage.all import *
>>> build_alphabet(name="positive integers")
Positive integers
>>> build_alphabet(name='PP')
Positive integers
>>> build_alphabet(name="natural numbers")
Non negative integers
>>> build_alphabet(name="NN")
Non negative integers
```

The other families for the option `name` are among 'lower', 'upper', 'space', 'underscore', 'punctuation', 'printable', 'binary', 'octal', 'decimal', 'hexadecimal', 'radix64' which refer to standard set of characters. Theses names may be combined by separating them by a space:

```
sage: build_alphabet(name='lower')
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
↪'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}
sage: build_alphabet(name='hexadecimal')
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}
sage: build_alphabet(name="decimal punctuation")
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ', ',', '.', ';', ':', '!',
↪'?'}
```

```
>>> from sage.all import *
>>> build_alphabet(name='lower')
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
↪'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}
>>> build_alphabet(name='hexadecimal')
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}
>>> build_alphabet(name="decimal punctuation")
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ', ',', '.', ';', ':', '!',
↪'?'}
```

In the case the alphabet is built from a list or a tuple, the order on the alphabet is given by the elements themselves:

```
sage: A = build_alphabet([0,2,1])
sage: A(0) < A(2)
True
sage: A(2) < A(1)
False
```

```
>>> from sage.all import *
>>> A = build_alphabet([Integer(0),Integer(2),Integer(1)])
>>> A(Integer(0)) < A(Integer(2))
True
>>> A(Integer(2)) < A(Integer(1))
False
```

If a different order is needed, you may use `TotallyOrderedFiniteSet` and set the option `facade` to `False`. That way, the comparison fits the order of the input:

```
sage: A = TotallyOrderedFiniteSet([4,2,6,1], facade=False)
sage: A(4) < A(2)
True
sage: A(1) < A(6)
False
```

```
>>> from sage.all import *
>>> A = TotallyOrderedFiniteSet([Integer(4),Integer(2),Integer(6),Integer(1)],␣
↪facade=False)
>>> A(Integer(4)) < A(Integer(2))
True
>>> A(Integer(1)) < A(Integer(6))
False
```

Be careful, the element of the set in the last example are no more integers and do not compare equal with integers:

```
sage: type(A.an_element())
<class 'sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet_with_
→category.element_class'>
sage: A(1) == 1
False
sage: 1 == A(1)
False
```

```
>>> from sage.all import *
>>> type(A.an_element())
<class 'sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet_with_
→category.element_class'>
>>> A(Integer(1)) == Integer(1)
False
>>> Integer(1) == A(Integer(1))
False
```

We give an example of an infinite alphabet indexed by the positive integers and the prime numbers:

```
sage: build_alphabet(oo, 'x')
Lazy family (x(i))_{i in Non negative integers}
sage: build_alphabet(Primes(), 'y')
Lazy family (y(i))_{i in Set of all prime numbers: 2, 3, 5, 7, ...}
```

```
>>> from sage.all import *
>>> build_alphabet(oo, 'x')
Lazy family (x(i))_{i in Non negative integers}
>>> build_alphabet(Primes(), 'y')
Lazy family (y(i))_{i in Set of all prime numbers: 2, 3, 5, 7, ...}
```

`sage.combinat.words.alphabet.`**`build_alphabet`**(*data=None*, *names=None*, *name=None*)

Return an object representing an ordered alphabet.

INPUT:

- `data` – the letters of the alphabet; it can be:
  - a list/tuple/iterable of letters; the iterable may be infinite
  - an integer n to represent $\{1, \ldots, n\}$, or infinity to represent $\mathbf{N}$
- `names` – (optional) a list for the letters (i.e. variable names) or a string for prefix for all letters; if given a list, it must have the same cardinality as the set represented by `data`
- `name` – (optional) if given, then return a named set and can be equal to : `'lower'`, `'upper'`, `'space'`, `'underscore'`, `'punctuation'`, `'printable'`, `'binary'`, `'octal'`, `'decimal'`, `'hexadecimal'`, `'radix64'`.

  You can use many of them at once, separated by spaces : `'lower punctuation'` represents the union of the two alphabets `'lower'` and `'punctuation'`.

  Alternatively, `name` can be set to `'positive integers'` (or `'PP'`) or `'natural numbers'` (or `'NN'`).

  `name` cannot be combined with `data`.

EXAMPLES:

If the argument is a Set, it just returns it:

```
sage: build_alphabet(ZZ) is ZZ
True
sage: F = FiniteEnumeratedSet('abc')
sage: build_alphabet(F) is F
True
```

```
>>> from sage.all import *
>>> build_alphabet(ZZ) is ZZ
True
>>> F = FiniteEnumeratedSet('abc')
>>> build_alphabet(F) is F
True
```

If a list, tuple or string is provided, then it builds a proper Sage class (`TotallyOrderedFiniteSet`):

```
sage: build_alphabet([0,1,2])
{0, 1, 2}
sage: F = build_alphabet('abc'); F
{'a', 'b', 'c'}
sage: print(type(F).__name__)
TotallyOrderedFiniteSet_with_category
```

```
>>> from sage.all import *
>>> build_alphabet([Integer(0),Integer(1),Integer(2)])
{0, 1, 2}
>>> F = build_alphabet('abc'); F
{'a', 'b', 'c'}
>>> print(type(F).__name__)
TotallyOrderedFiniteSet_with_category
```

If an integer and a set is given, then it constructs a `TotallyOrderedFiniteSet`:

```
sage: build_alphabet(3, ['a','b','c'])
{'a', 'b', 'c'}
```

```
>>> from sage.all import *
>>> build_alphabet(Integer(3), ['a','b','c'])
{'a', 'b', 'c'}
```

If an integer and a string is given, then it considers that string as a prefix:

```
sage: build_alphabet(3, 'x')
{'x0', 'x1', 'x2'}
```

```
>>> from sage.all import *
>>> build_alphabet(Integer(3), 'x')
{'x0', 'x1', 'x2'}
```

If no data is provided, `name` may be a string which describe an alphabet. The available names decompose into two families. The first one are 'positive integers', 'PP', 'natural numbers' or 'NN' which refer to standard set of numbers:

```
sage: build_alphabet(name="positive integers")
Positive integers
sage: build_alphabet(name='PP')
Positive integers
sage: build_alphabet(name="natural numbers")
Non negative integers
sage: build_alphabet(name="NN")
Non negative integers
```

```
>>> from sage.all import *
>>> build_alphabet(name="positive integers")
Positive integers
>>> build_alphabet(name='PP')
Positive integers
>>> build_alphabet(name="natural numbers")
Non negative integers
>>> build_alphabet(name="NN")
Non negative integers
```

The other families for the option `name` are among 'lower', 'upper', 'space', 'underscore', 'punctuation', 'printable', 'binary', 'octal', 'decimal', 'hexadecimal', 'radix64' which refer to standard set of characters. Theses names may be combined by separating them by a space:

```
sage: build_alphabet(name='lower')
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
↪'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}
sage: build_alphabet(name='hexadecimal')
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}
sage: build_alphabet(name="decimal punctuation")
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ', ',', '.', ';', ':', '!',
↪'?'}
```

```
>>> from sage.all import *
>>> build_alphabet(name='lower')
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
↪'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}
>>> build_alphabet(name='hexadecimal')
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}
>>> build_alphabet(name="decimal punctuation")
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ', ',', '.', ';', ':', '!',
↪'?'}
```

In the case the alphabet is built from a list or a tuple, the order on the alphabet is given by the elements themselves:

```
sage: A = build_alphabet([0,2,1])
sage: A(0) < A(2)
True
sage: A(2) < A(1)
False
```

```
>>> from sage.all import *
>>> A = build_alphabet([Integer(0),Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> A(Integer(0)) < A(Integer(2))
True
>>> A(Integer(2)) < A(Integer(1))
False
```

If a different order is needed, you may use `TotallyOrderedFiniteSet` and set the option `facade` to `False`. That way, the comparison fits the order of the input:

```
sage: A = TotallyOrderedFiniteSet([4,2,6,1], facade=False)
sage: A(4) < A(2)
True
sage: A(1) < A(6)
False
```

```
>>> from sage.all import *
>>> A = TotallyOrderedFiniteSet([Integer(4),Integer(2),Integer(6),Integer(1)],␣
↪facade=False)
>>> A(Integer(4)) < A(Integer(2))
True
>>> A(Integer(1)) < A(Integer(6))
False
```

Be careful, the element of the set in the last example are no more integers and do not compare equal with integers:

```
sage: type(A.an_element())
<class 'sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet_with_
↪category.element_class'>
sage: A(1) == 1
False
sage: 1 == A(1)
False
```

```
>>> from sage.all import *
>>> type(A.an_element())
<class 'sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet_with_
↪category.element_class'>
>>> A(Integer(1)) == Integer(1)
False
>>> Integer(1) == A(Integer(1))
False
```

We give an example of an infinite alphabet indexed by the positive integers and the prime numbers:

```
sage: build_alphabet(oo, 'x')
Lazy family (x(i))_{i in Non negative integers}
sage: build_alphabet(Primes(), 'y')
Lazy family (y(i))_{i in Set of all prime numbers: 2, 3, 5, 7, ...}
```

```
>>> from sage.all import *
>>> build_alphabet(oo, 'x')
Lazy family (x(i))_{i in Non negative integers}
```

(continues on next page)

(continued from previous page)

```
>>> build_alphabet(Primes(), 'y')
Lazy family (y(i))_{i in Set of all prime numbers: 2, 3, 5, 7, ...}
```

5.1.358 Finite word

AUTHORS:

- Arnaud Bergeron
- Amy Glen
- Sébastien Labbé
- Franco Saliola
- Julien Leroy (March 2010): reduced_rauzy_graph

EXAMPLES:

Creation of a finite word

Finite words from Python strings, lists and tuples:

```
sage: Word("abbabaab")
word: abbabaab
sage: Word([0, 1, 1, 0, 1, 0, 0, 1])
word: 01101001
sage: Word( ('a', 0, 5, 7, 'b', 9, 8) )
word: a057b98
```

```
>>> from sage.all import *
>>> Word("abbabaab")
word: abbabaab
>>> Word([Integer(0), Integer(1), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(0), Integer(1)])
word: 01101001
>>> Word( ('a', Integer(0), Integer(5), Integer(7), 'b', Integer(9), Integer(8)) )
word: a057b98
```

Finite words from functions:

```
sage: f = lambda n : n%3
sage: Word(f, length=13)
word: 0120120120120
```

```
>>> from sage.all import *
>>> f = lambda n : n%Integer(3)
>>> Word(f, length=Integer(13))
word: 0120120120120
```

Finite words from iterators:

```
sage: from itertools import count
sage: Word(count(), length=10)
word: 0123456789
```

```
>>> from sage.all import *
>>> from itertools import count
>>> Word(count(), length=Integer(10))
word: 0123456789
```

```
sage: Word( iter('abbccdef') )
word: abbccdef
```

```
>>> from sage.all import *
>>> Word( iter('abbccdef') )
word: abbccdef
```

Finite words from words via concatenation:

```
sage: u = Word("abcccabba")
sage: v = Word([0, 4, 8, 8, 3])
sage: u * v
word: abcccabba04883
sage: v * u
word: 04883abcccabba
sage: u + v
word: abcccabba04883
sage: u^3 * v^(8/5)
word: abcccabbaabcccabbaabcccabba04883048
```

```
>>> from sage.all import *
>>> u = Word("abcccabba")
>>> v = Word([Integer(0), Integer(4), Integer(8), Integer(8), Integer(3)])
>>> u * v
word: abcccabba04883
>>> v * u
word: 04883abcccabba
>>> u + v
word: abcccabba04883
>>> u**Integer(3) * v**(Integer(8)/Integer(5))
word: abcccabbaabcccabbaabcccabba04883048
```

Finite words from infinite words:

```
sage: vv = v^Infinity
sage: vv[10000:10015]
word: 048830488304883
```

```
>>> from sage.all import *
>>> vv = v**Infinity
>>> vv[Integer(10000):Integer(10015)]
word: 048830488304883
```

Finite words in a specific combinatorial class:

```
sage: W = Words("ab")
sage: W
```

(continues on next page)

(continued from previous page)

```
Finite and infinite words over {'a', 'b'}
sage: W("abbabaab")
word: abbabaab
sage: W(["a","b","b","a","b","a","a","b"])
word: abbabaab
sage: W( iter('ababab') )
word: ababab
```

```
>>> from sage.all import *
>>> W = Words("ab")
>>> W
Finite and infinite words over {'a', 'b'}
>>> W("abbabaab")
word: abbabaab
>>> W(["a","b","b","a","b","a","a","b"])
word: abbabaab
>>> W( iter('ababab') )
word: ababab
```

Finite word as the image under a morphism:

```
sage: m = WordMorphism({0:[4,4,5,0],5:[0,5,5],4:[4,0,0,0]})
sage: m(0)
word: 4450
sage: m(0, order=2)
word: 400040000554450
sage: m(0, order=3)
word: 4000445044504450400044504450445044500550...
```

```
>>> from sage.all import *
>>> m = WordMorphism({Integer(0):[Integer(4),Integer(4),Integer(5),Integer(0)],
↪Integer(5):[Integer(0),Integer(5),Integer(5)],Integer(4):[Integer(4),Integer(0),
↪Integer(0),Integer(0)]})
>>> m(Integer(0))
word: 4450
>>> m(Integer(0), order=Integer(2))
word: 400040000554450
>>> m(Integer(0), order=Integer(3))
word: 4000445044504450400044504450445044500550...
```

Note

The following two finite words have the same string representation:

```
sage: w = Word('010120')
sage: z = Word([0, 1, 0, 1, 2, 0])
sage: w
word: 010120
sage: z
word: 010120
```

```
>>> from sage.all import *
>>> w = Word('010120')
>>> z = Word([Integer(0), Integer(1), Integer(0), Integer(1), Integer(2),␣
↪Integer(0)])
>>> w
word: 010120
>>> z
word: 010120
```

but are not equal:

```
sage: w == z
False
```

```
>>> from sage.all import *
>>> w == z
False
```

Indeed, w and z are defined on different alphabets:

```
sage: w[2]
'0'
sage: z[2]
0
```

```
>>> from sage.all import *
>>> w[Integer(2)]
'0'
>>> z[Integer(2)]
0
```

Functions and algorithms

There are more than 100 functions defined on a finite word. Here are some of them:

```
sage: w = Word('abaabbba'); w
word: abaabbba
sage: w.is_palindrome()
False
sage: w.is_lyndon()
False
sage: w.number_of_factors()
28
sage: w.critical_exponent()
3
```

```
>>> from sage.all import *
>>> w = Word('abaabbba'); w
word: abaabbba
>>> w.is_palindrome()
False
>>> w.is_lyndon()
False
```

(continues on next page)

(continued from previous page)

```
>>> w.number_of_factors()
28
>>> w.critical_exponent()
3
```

```
sage: print(w.lyndon_factorization())
(ab, aabbb, a)
sage: print(w.crochemore_factorization())
(a, b, a, ab, bb, a)
```

```
>>> from sage.all import *
>>> print(w.lyndon_factorization())
(ab, aabbb, a)
>>> print(w.crochemore_factorization())
(a, b, a, ab, bb, a)
```

```
sage: st = w.suffix_tree()
sage: st
Implicit Suffix Tree of the word: abaabbba
sage: st.show(word_labels=True)                                                 #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> st = w.suffix_tree()
>>> st
Implicit Suffix Tree of the word: abaabbba
>>> st.show(word_labels=True)                                                 #␣
↪needs sage.plot
```

```
sage: T = words.FibonacciWord('ab')
sage: T.longest_common_prefix(Word('abaababababbbbbb'))
word: abaababa
```

```
>>> from sage.all import *
>>> T = words.FibonacciWord('ab')
>>> T.longest_common_prefix(Word('abaababababbbbbb'))
word: abaababa
```

As matrix and many other sage objects, words have a parent:

```
sage: u = Word('xyxxyxyyy')
sage: u.parent()
Finite words over Set of Python objects of class 'object'
```

```
>>> from sage.all import *
>>> u = Word('xyxxyxyyy')
>>> u.parent()
Finite words over Set of Python objects of class 'object'
```

```
sage: v = Word('xyxxyxyyy', alphabet='xy')
sage: v.parent()
Finite words over {'x', 'y'}
```

```
>>> from sage.all import *
>>> v = Word('xyxxyxyyy', alphabet='xy')
>>> v.parent()
Finite words over {'x', 'y'}
```

Factors and Rauzy Graphs

Enumeration of factors, the successive values returned by `next(it)` can appear in a different order depending on hardware. Therefore we mark the three first results of the test `random`. The important test is that the iteration stops properly on the fourth call:

```
sage: w = Word([4,5,6])^7
sage: it = w.factor_iterator(4)
sage: next(it) # random
word: 6456
sage: next(it) # random
word: 5645
sage: next(it) # random
word: 4564
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

```
>>> from sage.all import *
>>> w = Word([Integer(4),Integer(5),Integer(6)])**Integer(7)
>>> it = w.factor_iterator(Integer(4))
>>> next(it) # random
word: 6456
>>> next(it) # random
word: 5645
>>> next(it) # random
word: 4564
>>> next(it)
Traceback (most recent call last):
...
StopIteration
```

The set of factors:

```
sage: sorted(w.factor_set(3))
[word: 456, word: 564, word: 645]
sage: sorted(w.factor_set(4))
[word: 4564, word: 5645, word: 6456]
sage: w.factor_set().cardinality()
61
```

```
>>> from sage.all import *
>>> sorted(w.factor_set(Integer(3)))
[word: 456, word: 564, word: 645]
>>> sorted(w.factor_set(Integer(4)))
[word: 4564, word: 5645, word: 6456]
>>> w.factor_set().cardinality()
61
```

Rauzy graphs:

```
sage: f = words.FibonacciWord()[:30]
sage: f.rauzy_graph(4)                                                   #␣
↪needs sage.graphs
Looped digraph on 5 vertices
sage: f.reduced_rauzy_graph(4)                                           #␣
↪needs sage.graphs
Looped multi-digraph on 2 vertices
```

```
>>> from sage.all import *
>>> f = words.FibonacciWord()[:Integer(30)]
>>> f.rauzy_graph(Integer(4))                                             ␣
↪     # needs sage.graphs
Looped digraph on 5 vertices
>>> f.reduced_rauzy_graph(Integer(4))                                     ␣
↪     # needs sage.graphs
Looped multi-digraph on 2 vertices
```

Left-special and bispecial factors:

```
sage: f.number_of_left_special_factors(7)
1
sage: f.bispecial_factors()
[word: , word: 0, word: 010, word: 010010, word: 01001010010]
```

```
>>> from sage.all import *
>>> f.number_of_left_special_factors(Integer(7))
1
>>> f.bispecial_factors()
[word: , word: 0, word: 010, word: 010010, word: 01001010010]
```

class sage.combinat.words.finite_word.**CallableFromListOfWords**(*words*)

Bases: tuple

A class to create a callable from a list of words. The concatenation of a list of words is obtained by creating a word from this callable.

class sage.combinat.words.finite_word.**Factorization**(*iterable=()*, */*)

Bases: list

A list subclass having a nicer representation for factorization of words.

class sage.combinat.words.finite_word.**FiniteWord_class**

Bases: *Word_class*

BWT()

Return the Burrows-Wheeler Transform (BWT) of `self`.

The *Burrows-Wheeler transform* of a finite word w is obtained from w by first listing the conjugates of w in lexicographic order and then concatenating the final letters of the conjugates in this order. See [BW1994].

EXAMPLES:

```
sage: Word('abaccaaba').BWT()
word: cbaabaaca
sage: Word('abaab').BWT()
word: bbaaa
sage: Word('bbabbaca').BWT()
word: cbbbbaaa
sage: Word('aabaab').BWT()
word: bbaaaa
sage: Word().BWT()
word:
sage: Word('a').BWT()
word: a
```

```
>>> from sage.all import *
>>> Word('abaccaaba').BWT()
word: cbaabaaca
>>> Word('abaab').BWT()
word: bbaaa
>>> Word('bbabbaca').BWT()
word: cbbbbaaa
>>> Word('aabaab').BWT()
word: bbaaaa
>>> Word().BWT()
word:
>>> Word('a').BWT()
word: a
```

LZ_decomposition()

Return the Crochemore factorization of `self` as an ordered list of factors.

The *Crochemore factorization* or the *Lempel-Ziv decomposition* of a finite word w is the unique factorization: $(x_1, x_2, \ldots, x_n)$ of w with each x_i satisfying either: C1. x_i is a letter that does not appear in $u = x_1 \ldots x_{i-1}$; C2. x_i is the longest prefix of $v = x_i \ldots x_n$ that also has an occurrence beginning within $u = x_1 \ldots x_{i-1}$. See [Cro1983].

EXAMPLES:

```
sage: x = Word('ababab')
sage: x.crochemore_factorization()
(a, b, abab, b)
sage: mul(x.crochemore_factorization()) == x
True
sage: y = Word('abaababacabba')
sage: y.crochemore_factorization()
(a, b, a, aba, ba, c, ab, ba)
sage: mul(y.crochemore_factorization()) == y
True
```

(continues on next page)

(continued from previous page)

```
sage: x = Word([0,1,0,1,0,1,1])
sage: x.crochemore_factorization()
(0, 1, 0101, 1)
sage: mul(x.crochemore_factorization()) == x
True
```

```
>>> from sage.all import *
>>> x = Word('abababb')
>>> x.crochemore_factorization()
(a, b, abab, b)
>>> mul(x.crochemore_factorization()) == x
True
>>> y = Word('abaababacabba')
>>> y.crochemore_factorization()
(a, b, a, aba, ba, c, ab, ba)
>>> mul(y.crochemore_factorization()) == y
True
>>> x = Word([Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(1)])
>>> x.crochemore_factorization()
(0, 1, 0101, 1)
>>> mul(x.crochemore_factorization()) == x
True
```

abelian_complexity(*n*)

Return the number of abelian vectors of factors of length `n` of `self`.

EXAMPLES:

```
sage: w = words.FibonacciWord()[:100]
sage: [w.abelian_complexity(i) for i in range(20)]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

```
>>> from sage.all import *
>>> w = words.FibonacciWord()[:Integer(100)]
>>> [w.abelian_complexity(i) for i in range(Integer(20))]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

```
sage: w = words.ThueMorseWord()[:100]
sage: [w.abelian_complexity(i) for i in range(20)]
[1, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2]
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(100)]
>>> [w.abelian_complexity(i) for i in range(Integer(20))]
[1, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2]
```

abelian_vector()

Return the abelian vector of `self` counting the occurrences of each letter.

The vector is defined w.r.t. the order of the alphabet of the parent. See also *evaluation_dict()*.

INPUT:

- self – word having a parent on a finite alphabet

OUTPUT: list

EXAMPLES:

```
sage: W = Words('ab')
sage: W('aaabbbbb').abelian_vector()
[3, 5]
sage: W('a').abelian_vector()
[1, 0]
sage: W().abelian_vector()
[0, 0]
```

```
>>> from sage.all import *
>>> W = Words('ab')
>>> W('aaabbbbb').abelian_vector()
[3, 5]
>>> W('a').abelian_vector()
[1, 0]
>>> W().abelian_vector()
[0, 0]
```

The result depends on the alphabet of the parent:

```
sage: W = Words('abc')
sage: W('aabaa').abelian_vector()
[4, 1, 0]
```

```
>>> from sage.all import *
>>> W = Words('abc')
>>> W('aabaa').abelian_vector()
[4, 1, 0]
```

abelian_vectors(*n*)

Return the abelian vectors of factors of length n of self.

The vectors are defined w.r.t. the order of the alphabet of the parent.

OUTPUT: a set of tuples

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: w = W([0,1,1,0,1,2,0,2,0,2])
sage: w.abelian_vectors(3)
{(1, 0, 2), (1, 1, 1), (1, 2, 0), (2, 0, 1)}
sage: w[:5].abelian_vectors(3)
{(1, 2, 0)}
sage: w[5:].abelian_vectors(3)
{(1, 0, 2), (2, 0, 1)}
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> w = W([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0),Integer(2),Integer(0),Integer(2)])
>>> w.abelian_vectors(Integer(3))
{(1, 0, 2), (1, 1, 1), (1, 2, 0), (2, 0, 1)}
>>> w[:Integer(5)].abelian_vectors(Integer(3))
{(1, 2, 0)}
>>> w[Integer(5):].abelian_vectors(Integer(3))
{(1, 0, 2), (2, 0, 1)}
```

```
sage: w = words.FibonacciWord()[:100]
sage: sorted(w.abelian_vectors(0))
[(0, 0)]
sage: sorted(w.abelian_vectors(1))
[(0, 1), (1, 0)]
sage: sorted(w.abelian_vectors(7))
[(4, 3), (5, 2)]
```

```
>>> from sage.all import *
>>> w = words.FibonacciWord()[:Integer(100)]
>>> sorted(w.abelian_vectors(Integer(0)))
[(0, 0)]
>>> sorted(w.abelian_vectors(Integer(1)))
[(0, 1), (1, 0)]
>>> sorted(w.abelian_vectors(Integer(7)))
[(4, 3), (5, 2)]
```

The word must be defined with a parent on a finite alphabet:

```
sage: from itertools import count
sage: w = Word(count(), alphabet=NN)
sage: w[:2].abelian_vectors(2)
Traceback (most recent call last):
...
TypeError: The alphabet of the parent is infinite; define the
word with a parent on a finite alphabet
```

```
>>> from sage.all import *
>>> from itertools import count
>>> w = Word(count(), alphabet=NN)
>>> w[:Integer(2)].abelian_vectors(Integer(2))
Traceback (most recent call last):
...
TypeError: The alphabet of the parent is infinite; define the
word with a parent on a finite alphabet
```

apply_permutation_to_letters(*permutation*)

Return the word obtained by applying the permutation `permutation` of the alphabet of `self` to each letter of `self`.

EXAMPLES:

```
sage: w = Words('abcd')('abcd')
sage: p = [2,1,4,3]
```

(continues on next page)

(continued from previous page)

```
sage: w.apply_permutation_to_letters(p)
word: badc
sage: u = Words('dabc')('abcd')
sage: u.apply_permutation_to_letters(p)
word: dcba
sage: w.apply_permutation_to_letters(Permutation(p))
word: badc
sage: w.apply_permutation_to_letters(PermutationGroupElement(p))          #␣
↪needs sage.groups
word: badc
```

```
>>> from sage.all import *
>>> w = Words('abcd')('abcd')
>>> p = [Integer(2),Integer(1),Integer(4),Integer(3)]
>>> w.apply_permutation_to_letters(p)
word: badc
>>> u = Words('dabc')('abcd')
>>> u.apply_permutation_to_letters(p)
word: dcba
>>> w.apply_permutation_to_letters(Permutation(p))
word: badc
>>> w.apply_permutation_to_letters(PermutationGroupElement(p))          #␣
↪needs sage.groups
word: badc
```

apply_permutation_to_positions(*permutation*)

Return the word obtained by permuting the positions of the letters in `self` according to the permutation `permutation`.

EXAMPLES:

```
sage: w = Words('abcd')('abcd')
sage: w.apply_permutation_to_positions([2,1,4,3])
word: badc
sage: u = Words('dabc')('abcd')
sage: u.apply_permutation_to_positions([2,1,4,3])
word: badc
sage: w.apply_permutation_to_positions(Permutation([2,1,4,3]))
word: badc
sage: w.apply_permutation_to_positions(PermutationGroupElement([2,1,4,3]))   #␣
↪needs sage.groups
word: badc
sage: Word([1,2,3,4]).apply_permutation_to_positions([3,4,2,1])
word: 3421
```

```
>>> from sage.all import *
>>> w = Words('abcd')('abcd')
>>> w.apply_permutation_to_positions([Integer(2),Integer(1),Integer(4),
↪Integer(3)])
word: badc
>>> u = Words('dabc')('abcd')
>>> u.apply_permutation_to_positions([Integer(2),Integer(1),Integer(4),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)])
word: badc
>>> w.apply_permutation_to_positions(Permutation([Integer(2),Integer(1),
↪Integer(4),Integer(3)]))
word: badc
>>> w.apply_permutation_to_positions(PermutationGroupElement([Integer(2),
↪Integer(1),Integer(4),Integer(3)]))   # needs sage.groups
word: badc
>>> Word([Integer(1),Integer(2),Integer(3),Integer(4)]).apply_permutation_to_
↪positions([Integer(3),Integer(4),Integer(2),Integer(1)])
word: 3421
```

balance()

Return the balance of `self`.

The balance of a word is the smallest number q such that `self` is q-balanced [FV2002].

A finite or infinite word w is said to be *q-balanced* if for any two factors u, v of w of the same length, the difference between the number of x's in each of u and v is at most q for all letters x in the alphabet of w. A 1-balanced word is simply said to be balanced. See Chapter 2 of [Lot2002].

OUTPUT: integer

EXAMPLES:

```
sage: Word('1111111').balance()
0
sage: Word('001010101011').balance()
2
sage: Word('0101010101').balance()
1
```

```
>>> from sage.all import *
>>> Word('1111111').balance()
0
>>> Word('001010101011').balance()
2
>>> Word('0101010101').balance()
1
```

```
sage: w = Word('11112222')
sage: w.is_balanced(2)
False
sage: w.is_balanced(3)
False
sage: w.is_balanced(4)
True
sage: w.is_balanced(5)
True
sage: w.balance()
4
```

```
>>> from sage.all import *
>>> w = Word('11112222')
>>> w.is_balanced(Integer(2))
False
>>> w.is_balanced(Integer(3))
False
>>> w.is_balanced(Integer(4))
True
>>> w.is_balanced(Integer(5))
True
>>> w.balance()
4
```

bispecial_factors(*n=None*)

Return the bispecial factors (of length `n`).

A factor u of a word w is *bispecial* if it is right special and left special.

INPUT:

- `n` – integer (default: `None`); if `None`, it returns all bispecial factors

OUTPUT: list of words

EXAMPLES:

```
sage: w = words.FibonacciWord()[:30]
sage: w.bispecial_factors()
[word: , word: 0, word: 010, word: 010010, word: 01001010010]
```

```
>>> from sage.all import *
>>> w = words.FibonacciWord()[:Integer(30)]
>>> w.bispecial_factors()
[word: , word: 0, word: 010, word: 010010, word: 01001010010]
```

```
sage: w = words.ThueMorseWord()[:30]
sage: for i in range(10):
....:     print("{} {}".format(i, sorted(w.bispecial_factors(i))))
0 [word: ]
1 [word: 0, word: 1]
2 [word: 01, word: 10]
3 [word: 010, word: 101]
4 [word: 0110, word: 1001]
5 []
6 [word: 011001, word: 100110]
7 []
8 [word: 10010110]
9 []
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(30)]
>>> for i in range(Integer(10)):
...     print("{} {}".format(i, sorted(w.bispecial_factors(i))))
0 [word: ]
```

(continues on next page)

(continued from previous page)

```
1 [word: 0, word: 1]
2 [word: 01, word: 10]
3 [word: 010, word: 101]
4 [word: 0110, word: 1001]
5 []
6 [word: 011001, word: 100110]
7 []
8 [word: 10010110]
9 []
```

bispecial_factors_iterator(*n=None*)

Return an iterator over the bispecial factors (of length n).

A factor u of a word w is *bispecial* if it is right special and left special.

INPUT:

- n – integer (default: None); if None, it returns an iterator over all bispecial factors

EXAMPLES:

```
sage: w = words.ThueMorseWord()[:30]
sage: for i in range(10):
....:     for u in sorted(w.bispecial_factors_iterator(i)):
....:         print("{} {}".format(i,u))
0
1 0
1 1
2 01
2 10
3 010
3 101
4 0110
4 1001
6 011001
6 100110
8 10010110
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(30)]
>>> for i in range(Integer(10)):
...     for u in sorted(w.bispecial_factors_iterator(i)):
...         print("{} {}".format(i,u))
0
1 0
1 1
2 01
2 10
3 010
3 101
4 0110
4 1001
6 011001
```

(continues on next page)

(continued from previous page)

```
6 100110
8 10010110
```

```
sage: key = lambda u : (len(u), u)
sage: for u in sorted(w.bispecial_factors_iterator(), key=key): u
word:
word: 0
word: 1
word: 01
word: 10
word: 010
word: 101
word: 0110
word: 1001
word: 011001
word: 100110
word: 10010110
```

```
>>> from sage.all import *
>>> key = lambda u : (len(u), u)
>>> for u in sorted(w.bispecial_factors_iterator(), key=key): u
word:
word: 0
word: 1
word: 01
word: 10
word: 010
word: 101
word: 0110
word: 1001
word: 011001
word: 100110
word: 10010110
```

border()

Return the longest word that is both a proper prefix and a proper suffix of `self`.

EXAMPLES:

```
sage: Word('121212').border()
word: 1212
sage: Word('12321').border()
word: 1
sage: Word().border() is None
True
```

```
>>> from sage.all import *
>>> Word('121212').border()
word: 1212
>>> Word('12321').border()
word: 1
```

(continues on next page)

(continued from previous page)

```
>>> Word().border() is None
True
```

charge(*check=True*)

Return the charge of `self`. This is defined as follows.

If w is a permutation of length n, (in other words, the evaluation of w is $(1, 1, \dots, 1)$), the statistic charge(w) is given by $\sum_{i=1}^{n} c_i(w)$ where $c_1(w) = 0$ and $c_i(w)$ is defined recursively by setting p_i equal to 1 if i appears to the right of $i - 1$ in w and 0 otherwise. Then we set $c_i(w) = c_{i-1}(w) + p_i$.

EXAMPLES:

```
sage: Word([1, 2, 3]).charge()
3
sage: Word([3, 5, 1, 4, 2]).charge() == 0 + 1 + 1 + 2 + 2
True
```

```
>>> from sage.all import *
>>> Word([Integer(1), Integer(2), Integer(3)]).charge()
3
>>> Word([Integer(3), Integer(5), Integer(1), Integer(4), Integer(2)]).
↪charge() == Integer(0) + Integer(1) + Integer(1) + Integer(2) + Integer(2)
True
```

If w is not a permutation, but the evaluation of w is a partition, the charge of w is defined to be the sum of its charge subwords (each of which will be a permutation). The first charge subword is found by starting at the end of w and moving left until the first 1 is found. This is marked, and we continue to move to the left until the first 2 is found, wrapping around from the beginning of the word back to the end, if necessary. We mark this 2, and continue on until we have marked the largest letter in w. The marked letters, with relative order preserved, form the first charge subword of w. This subword is removed, and the next charge subword is found in the same manner from the remaining letters. In the following example, $w1, w2, w3$ are the charge subwords of w.

EXAMPLES:

```
sage: w = Word([5,2,3,4,4,1,1,1,2,2,3])
sage: w1 = Word([5, 2, 4, 1, 3])
sage: w2 = Word([3, 4, 1, 2])
sage: w3 = Word([1, 2])
sage: w.charge() == w1.charge() + w2.charge() + w3.charge()
True
```

```
>>> from sage.all import *
>>> w = Word([Integer(5),Integer(2),Integer(3),Integer(4),Integer(4),
↪Integer(1),Integer(1),Integer(1),Integer(2),Integer(2),Integer(3)])
>>> w1 = Word([Integer(5), Integer(2), Integer(4), Integer(1), Integer(3)])
>>> w2 = Word([Integer(3), Integer(4), Integer(1), Integer(2)])
>>> w3 = Word([Integer(1), Integer(2)])
>>> w.charge() == w1.charge() + w2.charge() + w3.charge()
True
```

Finally, if w does not have partition content, we apply the Lascoux-Schützenberger standardization operators s_i in such a manner as to obtain a word with partition content. (The word we obtain is independent of the choice of operators.) The charge is then defined to be the charge of this word:

```
sage: Word([3,3,2,1,1]).charge()
0
sage: Word([1,2,3,1,2]).charge()
2
```

```
>>> from sage.all import *
>>> Word([Integer(3),Integer(3),Integer(2),Integer(1),Integer(1)]).charge()
0
>>> Word([Integer(1),Integer(2),Integer(3),Integer(1),Integer(2)]).charge()
2
```

Note that this differs from the definition of charge given in Macdonald's book. The difference amounts to a choice of reading a word from left-to-right or right-to-left. The choice in Sage was made to agree with the definition of a reading word of a tableau in Sage, and seems to be the more common convention in the literature.

See [Mac1995], [LLM2003], and [LLT].

cocharge()

Return the cocharge of `self`. For a word w, this can be defined as $n_{ev} - ch(w)$, where $ch(w)$ is the charge of w and ev is the evaluation of w, and n_{ev} is $\sum_{i<j} min(ev_i, ev_j)$.

EXAMPLES:

```
sage: Word([1,2,3]).cocharge()
0
sage: Word([3,2,1]).cocharge()
3
sage: Word([1,1,2]).cocharge()
0
sage: Word([2,1,2]).cocharge()
1
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(2),Integer(3)]).cocharge()
0
>>> Word([Integer(3),Integer(2),Integer(1)]).cocharge()
3
>>> Word([Integer(1),Integer(1),Integer(2)]).cocharge()
0
>>> Word([Integer(2),Integer(1),Integer(2)]).cocharge()
1
```

coerce(*other*)

Try to return a pair of words with a common parent; raise an exception if this is not possible.

This function begins by checking if both words have the same parent. If this is the case, then no work is done and both words are returned as-is.

Otherwise it will attempt to convert `other` to the domain of `self`. If that fails, it will attempt to convert `self` to the domain of `other`. If both attempts fail, it raises a `TypeError` to signal failure.

EXAMPLES:

```
sage: W1 = Words('abc'); W2 = Words('ab')
sage: w1 = W1('abc'); w2 = W2('abba'); w3 = W1('baab')
sage: w1.parent() is w2.parent()
False
sage: a, b = w1.coerce(w2)
sage: a.parent() is b.parent()
True
sage: w1.parent() is w2.parent()
False
```

```
>>> from sage.all import *
>>> W1 = Words('abc'); W2 = Words('ab')
>>> w1 = W1('abc'); w2 = W2('abba'); w3 = W1('baab')
>>> w1.parent() is w2.parent()
False
>>> a, b = w1.coerce(w2)
>>> a.parent() is b.parent()
True
>>> w1.parent() is w2.parent()
False
```

colored_vector(*x=0*, *y=0*, *width='default'*, *height=1*, *cmap='hsv'*, *thickness=1*, *label=None*)

Return a vector (Graphics object) illustrating `self`. Each letter is represented by a coloured rectangle.

If the parent of `self` is a class of words over a finite alphabet, then each letter in the alphabet is assigned a unique colour, and this colour will be the same every time this method is called. This is especially useful when plotting and comparing words defined on the same alphabet.

If the alphabet is infinite, then the letters appearing in the word are used as the alphabet.

INPUT:

- `x` – (default: `0`) bottom left x-coordinate of the vector
- `y` – (default: `0`) bottom left y-coordinate of the vector
- `width` – (default: `'default'`) width of the vector. By default, the width is the length of `self`.
- `height` – (default: `1`) height of the vector
- `thickness` – (default: `1`) thickness of the contour
- `cmap` – (default: `'hsv'`) color map; for available color map names type: `import matplotlib.cm; list(matplotlib.cm.datad)`
- `label` – string (default: `None`); a label to add on the colored vector

OUTPUT: Graphics

EXAMPLES:

```
sage: # needs sage.plot
sage: Word(range(20)).colored_vector()
Graphics object consisting of 21 graphics primitives
sage: Word(range(100)).colored_vector(0,0,10,1)
Graphics object consisting of 101 graphics primitives
sage: Words(range(100))(range(10)).colored_vector()
Graphics object consisting of 11 graphics primitives
```

(continues on next page)

(continued from previous page)

```
sage: w = Word('abbabaab')
sage: w.colored_vector()
Graphics object consisting of 9 graphics primitives
sage: w.colored_vector(cmap='autumn')
Graphics object consisting of 9 graphics primitives
sage: Word(range(20)).colored_vector(label='Rainbow')
Graphics object consisting of 23 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> Word(range(Integer(20))).colored_vector()
Graphics object consisting of 21 graphics primitives
>>> Word(range(Integer(100))).colored_vector(Integer(0),Integer(0),
↪Integer(10),Integer(1))
Graphics object consisting of 101 graphics primitives
>>> Words(range(Integer(100)))(range(Integer(10))).colored_vector()
Graphics object consisting of 11 graphics primitives
>>> w = Word('abbabaab')
>>> w.colored_vector()
Graphics object consisting of 9 graphics primitives
>>> w.colored_vector(cmap='autumn')
Graphics object consisting of 9 graphics primitives
>>> Word(range(Integer(20))).colored_vector(label='Rainbow')
Graphics object consisting of 23 graphics primitives
```

When two words are defined under the same parent, same letters are mapped to same colors:

```
sage: W = Words(range(20))
sage: w = W(range(20))
sage: y = W(range(10,20))
sage: y.colored_vector(y=1, x=10) + w.colored_vector()                    #␣
↪needs sage.plot
Graphics object consisting of 32 graphics primitives
```

```
>>> from sage.all import *
>>> W = Words(range(Integer(20)))
>>> w = W(range(Integer(20)))
>>> y = W(range(Integer(10),Integer(20)))
>>> y.colored_vector(y=Integer(1), x=Integer(10)) + w.colored_vector()       ␣
↪               # needs sage.plot
Graphics object consisting of 32 graphics primitives
```

commutes_with(*other*)

Return `True` if `self` commutes with `other`, and `False` otherwise.

EXAMPLES:

```
sage: Word('12').commutes_with(Word('12'))
True
sage: Word('12').commutes_with(Word('11'))
False
sage: Word().commutes_with(Word('21'))
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> Word('12').commutes_with(Word('12'))
True
>>> Word('12').commutes_with(Word('11'))
False
>>> Word().commutes_with(Word('21'))
True
```

complete_return_words(*fact*)

Return the set of complete return words of `fact` in `self`.

This is the set of all factors starting by the given factor and ending just after the next occurrence of this factor. See for instance [JV2000].

INPUT:

- `fact` – a non-empty finite word

OUTPUT: a Python set of finite words

EXAMPLES:

```
sage: s = Word('21331233213231').complete_return_words(Word('2'))
sage: sorted(s)
[word: 2132, word: 213312, word: 2332]
sage: Word('').complete_return_words(Word('213'))
set()
sage: Word('121212').complete_return_words(Word('1212'))
{word: 121212}
```

```
>>> from sage.all import *
>>> s = Word('21331233213231').complete_return_words(Word('2'))
>>> sorted(s)
[word: 2132, word: 213312, word: 2332]
>>> Word('').complete_return_words(Word('213'))
set()
>>> Word('121212').complete_return_words(Word('1212'))
{word: 121212}
```

concatenate(*other*)

Return the concatenation of `self` and `other`.

INPUT:

- `other` – a word over the same alphabet as `self`

EXAMPLES:

Concatenation may be made using `+` or `*` operations:

```
sage: w = Word('abadafd')
sage: y = Word([5,3,5,8,7])
sage: w * y
word: abadafd53587
```

(continues on next page)

(continued from previous page)

```
sage: w + y
word: abadafd53587
sage: w.concatenate(y)
word: abadafd53587
```

```
>>> from sage.all import *
>>> w = Word('abadafd')
>>> y = Word([Integer(5),Integer(3),Integer(5),Integer(8),Integer(7)])
>>> w * y
word: abadafd53587
>>> w + y
word: abadafd53587
>>> w.concatenate(y)
word: abadafd53587
```

Both words must be defined over the same alphabet:

```
sage: z = Word('12223', alphabet = '123')
sage: z + y
Traceback (most recent call last):
...
ValueError: 5 not in alphabet
```

```
>>> from sage.all import *
>>> z = Word('12223', alphabet = '123')
>>> z + y
Traceback (most recent call last):
...
ValueError: 5 not in alphabet
```

Eventually, it should work:

```
sage: z = Word('12223', alphabet = '123')
sage: z + y                   #todo: not implemented
word: 1222353587
```

```
>>> from sage.all import *
>>> z = Word('12223', alphabet = '123')
>>> z + y                   #todo: not implemented
word: 1222353587
```

conjugate(*pos*)

Return the conjugate at `pos` of `self`.

`pos` can be any integer, the distance used is the modulo by the length of `self`.

EXAMPLES:

```
sage: Word('12112').conjugate(1)
word: 21121
sage: Word().conjugate(2)
word:
```

(continues on next page)

(continued from previous page)

```
sage: Word('12112').conjugate(8)
word: 12121
sage: Word('12112').conjugate(-1)
word: 21211
```

```
>>> from sage.all import *
>>> Word('12112').conjugate(Integer(1))
word: 21121
>>> Word().conjugate(Integer(2))
word:
>>> Word('12112').conjugate(Integer(8))
word: 12121
>>> Word('12112').conjugate(-Integer(1))
word: 21211
```

conjugate_position(*other*)

Return the position where `self` is conjugate with `other`. Return `None` if there is no such position.

EXAMPLES:

```
sage: Word('12113').conjugate_position(Word('31211'))
1
sage: Word('12131').conjugate_position(Word('12113')) is None
True
sage: Word().conjugate_position(Word('123')) is None
True
```

```
>>> from sage.all import *
>>> Word('12113').conjugate_position(Word('31211'))
1
>>> Word('12131').conjugate_position(Word('12113')) is None
True
>>> Word().conjugate_position(Word('123')) is None
True
```

conjugates()

Return the list of unique conjugates of `self`.

EXAMPLES:

```
sage: Word(range(6)).conjugates()
[word: 012345,
 word: 123450,
 word: 234501,
 word: 345012,
 word: 450123,
 word: 501234]
sage: Word('cbbca').conjugates()
[word: cbbca, word: bbcac, word: bcacb, word: cacbb, word: acbbc]
```

```
>>> from sage.all import *
>>> Word(range(Integer(6))).conjugates()
```

(continues on next page)

(continued from previous page)

```
[word: 012345,
 word: 123450,
 word: 234501,
 word: 345012,
 word: 450123,
 word: 501234]
>>> Word('cbbca').conjugates()
[word: cbbca, word: bbcac, word: bcacb, word: cacbb, word: acbbc]
```

The result contains each conjugate only once:

```
sage: Word('abcabc').conjugates()
[word: abcabc, word: bcabca, word: cabcab]
```

```
>>> from sage.all import *
>>> Word('abcabc').conjugates()
[word: abcabc, word: bcabca, word: cabcab]
```

conjugates_iterator()

Return an iterator over the conjugates of `self`.

EXAMPLES:

```
sage: it = Word(range(4)).conjugates_iterator()
sage: for w in it: w
word: 0123
word: 1230
word: 2301
word: 3012
```

```
>>> from sage.all import *
>>> it = Word(range(Integer(4))).conjugates_iterator()
>>> for w in it: w
word: 0123
word: 1230
word: 2301
word: 3012
```

content(*n=None*)

Return content of `self`.

INPUT:

- `n` – (optional) an integer specifying the maximal letter in the alphabet

OUTPUT: list where the i-th entry indicates the multiplicity of the i-th letter in the alphabet in `self`

EXAMPLES:

```
sage: w = Word([1,2,4,3,2,2,2])
sage: w.content()
[1, 4, 1, 1]
sage: w = Word([3,1])
sage: w.content()
```

(continues on next page)

(continued from previous page)

```
[1, 1]
sage: w.content(n=3)
[1, 0, 1]
sage: w = Word([2,4],alphabet=[1,2,3,4])
sage: w.content(n=3)
[0, 1, 0]
sage: w.content()
[0, 1, 0, 1]
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(2),Integer(4),Integer(3),Integer(2),
↪Integer(2),Integer(2)])
>>> w.content()
[1, 4, 1, 1]
>>> w = Word([Integer(3),Integer(1)])
>>> w.content()
[1, 1]
>>> w.content(n=Integer(3))
[1, 0, 1]
>>> w = Word([Integer(2),Integer(4)],alphabet=[Integer(1),Integer(2),
↪Integer(3),Integer(4)])
>>> w.content(n=Integer(3))
[0, 1, 0]
>>> w.content()
[0, 1, 0, 1]
```

count (*letter*)

Return the number of occurrences of `letter` in `self`.

INPUT:

- `letter` – a letter

OUTPUT: integer

EXAMPLES:

```
sage: w = Word('abbabaab')
sage: w.number_of_letter_occurrences('a')
4
sage: w.number_of_letter_occurrences('ab')
0
```

```
>>> from sage.all import *
>>> w = Word('abbabaab')
>>> w.number_of_letter_occurrences('a')
4
>>> w.number_of_letter_occurrences('ab')
0
```

This methods is equivalent to `list(w).count(letter)` and `tuple(w).count(letter)`, thus `count` is an alias for the method `number_of_letter_occurrences`:

```
sage: list(w).count('a')
4
sage: w.count('a')
4
```

```
>>> from sage.all import *
>>> list(w).count('a')
4
>>> w.count('a')
4
```

But notice that if `s` and `w` are strings, `Word(s).count(w)` counts the number occurrences of `w` as a letter in `Word(s)` which is not the same as `s.count(w)` which counts the number of occurrences of the string `w` inside `s`:

```
sage: s = 'abbabaab'
sage: s.count('ab')
3
sage: Word(s).count('ab')
0
```

```
>>> from sage.all import *
>>> s = 'abbabaab'
>>> s.count('ab')
3
>>> Word(s).count('ab')
0
```

See also:

sage.combinat.words.finite_word.FiniteWord_class.number_of_factor_occurrences()

critical_exponent()

Return the critical exponent of `self`.

The *critical exponent* of a word is the supremum of the order of all its (finite) factors. See [Dej1972].

Note: The implementation here uses the suffix tree to enumerate all the factors. It should be improved (especially when the critical exponent is larger than 2).

EXAMPLES:

```
sage: Word('aaba').critical_exponent()
2
sage: Word('aabaa').critical_exponent()
2
sage: Word('aabaaba').critical_exponent()
7/3
```

(continues on next page)

(continued from previous page)

```
sage: Word('ab').critical_exponent()
1
sage: Word('aba').critical_exponent()
3/2
sage: words.ThueMorseWord()[:20].critical_exponent()
2
```

```
>>> from sage.all import *
>>> Word('aaba').critical_exponent()
2
>>> Word('aabaa').critical_exponent()
2
>>> Word('aabaaba').critical_exponent()
7/3
>>> Word('ab').critical_exponent()
1
>>> Word('aba').critical_exponent()
3/2
>>> words.ThueMorseWord()[:Integer(20)].critical_exponent()
2
```

For the Fibonacci word, the critical exponent is known to be $(5+\sqrt(5))/2$. With a prefix of length 500, we obtain a lower bound:

```
sage: words.FibonacciWord()[:500].critical_exponent()
320/89
```

```
>>> from sage.all import *
>>> words.FibonacciWord()[:Integer(500)].critical_exponent()
320/89
```

It is an error to compute the critical exponent of the empty word:

```
sage: Word('').critical_exponent()
Traceback (most recent call last):
...
ValueError: no critical exponent for empty word
```

```
>>> from sage.all import *
>>> Word('').critical_exponent()
Traceback (most recent call last):
...
ValueError: no critical exponent for empty word
```

crochemore_factorization()

Return the Crochemore factorization of `self` as an ordered list of factors.

The *Crochemore factorization* or the *Lempel-Ziv decomposition* of a finite word w is the unique factorization: $(x_1, x_2, \ldots, x_n)$ of w with each x_i satisfying either: C1. x_i is a letter that does not appear in $u = x_1 \ldots x_{i-1}$; C2. x_i is the longest prefix of $v = x_i \ldots x_n$ that also has an occurrence beginning within $u = x_1 \ldots x_{i-1}$. See [Cro1983].

EXAMPLES:

```
sage: x = Word('abababb')
sage: x.crochemore_factorization()
(a, b, abab, b)
sage: mul(x.crochemore_factorization()) == x
True
sage: y = Word('abaababacabba')
sage: y.crochemore_factorization()
(a, b, a, aba, ba, c, ab, ba)
sage: mul(y.crochemore_factorization()) == y
True
sage: x = Word([0,1,0,1,0,1,1])
sage: x.crochemore_factorization()
(0, 1, 0101, 1)
sage: mul(x.crochemore_factorization()) == x
True
```

```
>>> from sage.all import *
>>> x = Word('abababb')
>>> x.crochemore_factorization()
(a, b, abab, b)
>>> mul(x.crochemore_factorization()) == x
True
>>> y = Word('abaababacabba')
>>> y.crochemore_factorization()
(a, b, a, aba, ba, c, ab, ba)
>>> mul(y.crochemore_factorization()) == y
True
>>> x = Word([Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(1)])
>>> x.crochemore_factorization()
(0, 1, 0101, 1)
>>> mul(x.crochemore_factorization()) == x
True
```

defect (*f=None*)

Return the defect of `self`.

The *defect* of a finite word w is given by the difference between the maximum number of possible palindromic factors in a word of length $|w|$ and the actual number of palindromic factors contained in w. It is well known that the maximum number of palindromic factors in w is $|w|+1$ (see [DJP2001]).

An optional involution on letters `f` can be given. In that case, the *f-palindromic defect* (or *pseudopalindromic defect*, or *theta-palindromic defect*) of w is returned. It is a generalization of defect to f-palindromes. More precisely, the defect is $D(w) = |w| + 1 - g_f(w) - |PAL_f(w)|$, where $PAL_f(w)$ denotes the set of f-palindromic factors of w (including the empty word) and $g_f(w)$ is the number of pairs $\{a, f(a)\}$ such that a is a letter, a is not equal to $f(a)$, and a or $f(a)$ occurs in w. In the case of usual palindromes (i.e., for `f` not given or equal to the identity), $g_f(w) = 0$ for all w. See [BHNR2004] for usual palindromes and [Star2011] for f-palindromes.

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`. It must be callable on letters as well as words (e.g. `WordMorphism`). The default value corresponds to usual palindromes, i.e., `f` equal to the identity.

OUTPUT:

an integer – If `f` is `None`, the palindromic defect of `self`; otherwise, the `f`-palindromic defect of `self`.

EXAMPLES:

```
sage: Word('ara').defect()
0
sage: Word('abcacba').defect()
1
```

```
>>> from sage.all import *
>>> Word('ara').defect()
0
>>> Word('abcacba').defect()
1
```

It is known that Sturmian words (see [DJP2001]) have zero defect:

```
sage: words.FibonacciWord()[:100].defect()
0

sage: sa = WordMorphism('a->ab,b->b')
sage: sb = WordMorphism('a->a,b->ba')
sage: w = (sa*sb*sb*sa*sa*sa*sb).fixed_point('a')
sage: w[:30].defect()                                                          #␣
↪needs sage.modules
0
sage: w[110:140].defect()                                                      #␣
↪needs sage.modules
0
```

```
>>> from sage.all import *
>>> words.FibonacciWord()[:Integer(100)].defect()
0

>>> sa = WordMorphism('a->ab,b->b')
>>> sb = WordMorphism('a->a,b->ba')
>>> w = (sa*sb*sb*sa*sa*sa*sb).fixed_point('a')
>>> w[:Integer(30)].defect()                                                    ␣
↪      # needs sage.modules
0
>>> w[Integer(110):Integer(140)].defect()                                       ␣
↪               # needs sage.modules
0
```

It is even conjectured that the defect of an aperiodic word which is a fixed point of a primitive morphism is either 0 or infinite (see [BBGL2008]):

```
sage: w = words.ThueMorseWord()
sage: w[:50].defect()                                                          #␣
↪needs sage.modules
12
sage: w[:100].defect()                                                         #␣
↪needs sage.modules
16
```

(continues on next page)

(continued from previous page)

```
sage: w[:300].defect()                                                     #␣
↪needs sage.modules
52
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()
>>> w[:Integer(50)].defect()                                               ␣
↪     # needs sage.modules
12
>>> w[:Integer(100)].defect()                                              ␣
↪     # needs sage.modules
16
>>> w[:Integer(300)].defect()                                              ␣
↪     # needs sage.modules
52
```

For generalized defect with an involution different from the identity, there is always a letter which is not a palindrome! This is the reason for the modification of the definition:

```
sage: f = WordMorphism('a->b,b->a')
sage: Word('a').defect(f)
0
sage: Word('ab').defect(f)
0
sage: Word('aa').defect(f)
1
sage: Word('abbabaabbaababba').defect(f)
3
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> Word('a').defect(f)
0
>>> Word('ab').defect(f)
0
>>> Word('aa').defect(f)
1
>>> Word('abbabaabbaababba').defect(f)
3
```

```
sage: f = WordMorphism('a->b,b->a,c->c')
sage: Word('cabc').defect(f)
0
sage: Word('abcaab').defect(f)
2
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a,c->c')
>>> Word('cabc').defect(f)
0
>>> Word('abcaab').defect(f)
```

(continues on next page)

(continued from previous page)

```
2
```

Other examples:

```
sage: Word('000000000000').defect()
0
sage: Word('011010011001').defect()
2
sage: Word('0101001010001').defect()
0
sage: Word().defect()
0
sage: Word('abbabaabbaababba').defect()
2
```

```
>>> from sage.all import *
>>> Word('000000000000').defect()
0
>>> Word('011010011001').defect()
2
>>> Word('0101001010001').defect()
0
>>> Word().defect()
0
>>> Word('abbabaabbaababba').defect()
2
```

deg_inv_lex_less(*other*, *weights=None*)

Return `True` if the word `self` is degree inverse lexicographically less than `other`.

EXAMPLES:

```
sage: Word([1,2,4]).deg_inv_lex_less(Word([1,3,2]))
False
sage: Word([3,2,1]).deg_inv_lex_less(Word([1,2,3]))
True
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(2),Integer(4)]).deg_inv_lex_
↪less(Word([Integer(1),Integer(3),Integer(2)]))
False
>>> Word([Integer(3),Integer(2),Integer(1)]).deg_inv_lex_
↪less(Word([Integer(1),Integer(2),Integer(3)]))
True
```

deg_lex_less(*other*, *weights=None*)

Return `True` if `self` is degree lexicographically less than `other`, and `False` otherwise. The weight of each letter in the ordered alphabet is given by `weights`, which defaults to `[1, 2, 3, ...]`.

EXAMPLES:

```
sage: Word([1,2,3]).deg_lex_less(Word([1,3,2]))
True
```

(continues on next page)

(continued from previous page)

```
sage: Word([3,2,1]).deg_lex_less(Word([1,2,3]))
False
sage: W = Words(range(5))
sage: W([1,2,4]).deg_lex_less(W([1,3,2]))
False
sage: Word("abba").deg_lex_less(Word("abbb"), dict(a=1,b=2))
True
sage: Word("abba").deg_lex_less(Word("baba"), dict(a=1,b=2))
True
sage: Word("abba").deg_lex_less(Word("aaba"), dict(a=1,b=2))
False
sage: Word("abba").deg_lex_less(Word("aaba"), dict(a=1,b=0))
True
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(2),Integer(3)]).deg_lex_less(Word([Integer(1),
↪Integer(3),Integer(2)]))
True
>>> Word([Integer(3),Integer(2),Integer(1)]).deg_lex_less(Word([Integer(1),
↪Integer(2),Integer(3)]))
False
>>> W = Words(range(Integer(5)))
>>> W([Integer(1),Integer(2),Integer(4)]).deg_lex_less(W([Integer(1),
↪Integer(3),Integer(2)]))
False
>>> Word("abba").deg_lex_less(Word("abbb"), dict(a=Integer(1),b=Integer(2)))
True
>>> Word("abba").deg_lex_less(Word("baba"), dict(a=Integer(1),b=Integer(2)))
True
>>> Word("abba").deg_lex_less(Word("aaba"), dict(a=Integer(1),b=Integer(2)))
False
>>> Word("abba").deg_lex_less(Word("aaba"), dict(a=Integer(1),b=Integer(0)))
True
```

deg_rev_lex_less(*other*, *weights=None*)

Return `True` if `self` is degree reverse lexicographically less than `other`.

EXAMPLES:

```
sage: Word([3,2,1]).deg_rev_lex_less(Word([1,2,3]))
False
sage: Word([1,2,4]).deg_rev_lex_less(Word([1,3,2]))
False
sage: Word([1,2,3]).deg_rev_lex_less(Word([1,2,4]))
True
```

```
>>> from sage.all import *
>>> Word([Integer(3),Integer(2),Integer(1)]).deg_rev_lex_
↪less(Word([Integer(1),Integer(2),Integer(3)]))
False
>>> Word([Integer(1),Integer(2),Integer(4)]).deg_rev_lex_
↪less(Word([Integer(1),Integer(3),Integer(2)]))
```

(continues on next page)

(continued from previous page)

```
False
>>> Word([Integer(1),Integer(2),Integer(3)]).deg_rev_lex_
↪less(Word([Integer(1),Integer(2),Integer(4)]))
True
```

degree(*weights=None*)

Return the weighted degree of `self`, where the weighted degree of each letter in the ordered alphabet is given by `weights`, which defaults to `[1, 2, 3, ...]`.

INPUT:

- `weights` – list or tuple, or dictionary keyed by the letters occurring in `self`

EXAMPLES:

```
sage: Word([1,2,3]).degree()
6
sage: Word([3,2,1]).degree()
6
sage: Words("ab")("abba").degree()
6
sage: Words("ab")("abba").degree([0,2])
4
sage: Words("ab")("abba").degree([-1,-1])
-4
sage: Words("ab")("aabba").degree([1,1])
5
sage: Words([1,2,4])([1,2,4]).degree()
6
sage: Word([1,2,4]).degree()
7
sage: Word("aabba").degree({'a':1,'b':2})
7
sage: Word([0,1,0]).degree({0:17,1:0})
34
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(2),Integer(3)]).degree()
6
>>> Word([Integer(3),Integer(2),Integer(1)]).degree()
6
>>> Words("ab")("abba").degree()
6
>>> Words("ab")("abba").degree([Integer(0),Integer(2)])
4
>>> Words("ab")("abba").degree([-Integer(1),-Integer(1)])
-4
>>> Words("ab")("aabba").degree([Integer(1),Integer(1)])
5
>>> Words([Integer(1),Integer(2),Integer(4)])([Integer(1),Integer(2),
↪Integer(4)]).degree()
6
>>> Word([Integer(1),Integer(2),Integer(4)]).degree()
```

(continues on next page)

(continued from previous page)

```
7
>>> Word("aabba").degree({'a':Integer(1),'b':Integer(2)})
7
>>> Word([Integer(0),Integer(1),Integer(0)]).degree({Integer(0):Integer(17),
↪Integer(1):Integer(0)})
34
```

delta()

Return the image of `self` under the delta morphism.

The delta morphism, also known as the run-length encoding, is the word composed of the length of consecutive runs of the same letter in a given word.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('22112122').delta()
word: 22112
sage: W('555008').delta()
word: 321
sage: W().delta()
word:
sage: Word('aabbabaa').delta()
word: 22112
```

```
>>> from sage.all import *
>>> W = Words('0123456789')
>>> W('22112122').delta()
word: 22112
>>> W('555008').delta()
word: 321
>>> W().delta()
word:
>>> Word('aabbabaa').delta()
word: 22112
```

delta_derivate(*W=None*)

Return the derivative under delta for `self`.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate()
word: 22
sage: W('1').delta_derivate(Words([1]))
word: 1
sage: W('2112').delta_derivate()
word: 2
sage: W('2211').delta_derivate()
word: 22
sage: W('112').delta_derivate()
word: 2
sage: W('11222').delta_derivate(Words([1, 2, 3]))
word: 3
```

```
>>> from sage.all import *
>>> W = Words('12')
>>> W('12211').delta_derivate()
word: 22
>>> W('1').delta_derivate(Words([Integer(1)]))
word: 1
>>> W('2112').delta_derivate()
word: 2
>>> W('2211').delta_derivate()
word: 22
>>> W('112').delta_derivate()
word: 2
>>> W('11222').delta_derivate(Words([Integer(1), Integer(2), Integer(3)]))
word: 3
```

delta_derivate_left(*W=None*)

Return the derivative under delta for `self`.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate_left()
word: 22
sage: W('1').delta_derivate_left(Words([1]))
word: 1
sage: W('2112').delta_derivate_left()
word: 21
sage: W('2211').delta_derivate_left()
word: 22
sage: W('112').delta_derivate_left()
word: 21
sage: W('11222').delta_derivate_left(Words([1, 2, 3]))
word: 3
```

```
>>> from sage.all import *
>>> W = Words('12')
>>> W('12211').delta_derivate_left()
word: 22
>>> W('1').delta_derivate_left(Words([Integer(1)]))
word: 1
>>> W('2112').delta_derivate_left()
word: 21
>>> W('2211').delta_derivate_left()
word: 22
>>> W('112').delta_derivate_left()
word: 21
>>> W('11222').delta_derivate_left(Words([Integer(1), Integer(2),␣
↪Integer(3)]))
word: 3
```

delta_derivate_right(*W=None*)

Return the right derivative under delta for `self`.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate_right()
word: 122
sage: W('1').delta_derivate_right(Words([1]))
word: 1
sage: W('2112').delta_derivate_right()
word: 12
sage: W('2211').delta_derivate_right()
word: 22
sage: W('112').delta_derivate_right()
word: 2
sage: W('11222').delta_derivate_right(Words([1, 2, 3]))
word: 23
```

```
>>> from sage.all import *
>>> W = Words('12')
>>> W('12211').delta_derivate_right()
word: 122
>>> W('1').delta_derivate_right(Words([Integer(1)]))
word: 1
>>> W('2112').delta_derivate_right()
word: 12
>>> W('2211').delta_derivate_right()
word: 22
>>> W('112').delta_derivate_right()
word: 2
>>> W('11222').delta_derivate_right(Words([Integer(1), Integer(2),␣
↪Integer(3)]))
word: 23
```

delta_inv(*W=None*, *s=None*)

Lift `self` via the delta operator to obtain a word containing the letters in alphabet (default: `[0, 1]`). The letters used in the construction start with `s` (default: `alphabet[0]`) and cycle through alphabet.

INPUT:

- `alphabet` – an iterable
- `s` – an object in the iterable

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2, 2, 1, 1]).delta_inv()
word: 112212
sage: W([1, 1, 1, 1]).delta_inv(Words('123'))
word: 1231
sage: W([2, 2, 1, 1, 2]).delta_inv(s=2)
word: 22112122
```

```
>>> from sage.all import *
>>> W = Words([Integer(1), Integer(2)])
>>> W([Integer(2), Integer(2), Integer(1), Integer(1)]).delta_inv()
word: 112212
```

(continues on next page)

(continued from previous page)

```
>>> W([Integer(1), Integer(1), Integer(1), Integer(1)]).delta_inv(Words('123
↪'))
word: 1231
>>> W([Integer(2), Integer(2), Integer(1), Integer(1), Integer(2)]).delta_
↪inv(s=Integer(2))
word: 22112122
```

evaluation()

Return the abelian vector of `self` counting the occurrences of each letter.

The vector is defined w.r.t. the order of the alphabet of the parent. See also *evaluation_dict()*.

INPUT:

- `self` – word having a parent on a finite alphabet

OUTPUT: list

EXAMPLES:

```
sage: W = Words('ab')
sage: W('aaabbbbb').abelian_vector()
[3, 5]
sage: W('a').abelian_vector()
[1, 0]
sage: W().abelian_vector()
[0, 0]
```

```
>>> from sage.all import *
>>> W = Words('ab')
>>> W('aaabbbbb').abelian_vector()
[3, 5]
>>> W('a').abelian_vector()
[1, 0]
>>> W().abelian_vector()
[0, 0]
```

The result depends on the alphabet of the parent:

```
sage: W = Words('abc')
sage: W('aabaa').abelian_vector()
[4, 1, 0]
```

```
>>> from sage.all import *
>>> W = Words('abc')
>>> W('aabaa').abelian_vector()
[4, 1, 0]
```

evaluation_dict()

Return a dictionary keyed by the letters occurring in `self` with values the number of occurrences of the letter.

EXAMPLES:

```
sage: Word([2,1,4,2,3,4,2]).evaluation_dict()
{1: 1, 2: 3, 3: 1, 4: 2}
sage: Word('badbcdb').evaluation_dict()
{'a': 1, 'b': 3, 'c': 1, 'd': 2}
sage: Word().evaluation_dict()
{}
```

```
>>> from sage.all import *
>>> Word([Integer(2),Integer(1),Integer(4),Integer(2),Integer(3),Integer(4),
↪Integer(2)]).evaluation_dict()
{1: 1, 2: 3, 3: 1, 4: 2}
>>> Word('badbcdb').evaluation_dict()
{'a': 1, 'b': 3, 'c': 1, 'd': 2}
>>> Word().evaluation_dict()
{}
```

```
sage: f = Word('1213121').evaluation_dict() # keys appear in random order
{'1': 4, '2': 2, '3': 1}
```

```
>>> from sage.all import *
>>> f = Word('1213121').evaluation_dict() # keys appear in random order
{'1': 4, '2': 2, '3': 1}
```

evaluation_partition()

Return the evaluation of the word w as a partition.

EXAMPLES:

```
sage: Word("acdabda").evaluation_partition()
[3, 2, 1, 1]
sage: Word([2,1,4,2,3,4,2]).evaluation_partition()
[3, 2, 1, 1]
```

```
>>> from sage.all import *
>>> Word("acdabda").evaluation_partition()
[3, 2, 1, 1]
>>> Word([Integer(2),Integer(1),Integer(4),Integer(2),Integer(3),Integer(4),
↪Integer(2)]).evaluation_partition()
[3, 2, 1, 1]
```

evaluation_sparse()

Return a list representing the evaluation of `self`. The entries of the list are two-element lists `[a, n]`, where `a` is a letter occurring in `self` and `n` is the number of occurrences of `a` in `self`.

EXAMPLES:

```
sage: sorted(Word([4,4,2,5,2,1,4,1]).evaluation_sparse())
[(1, 2), (2, 2), (4, 3), (5, 1)]
sage: sorted(Word("abcaccab").evaluation_sparse())
[('a', 3), ('b', 2), ('c', 3)]
```

```
>>> from sage.all import *
>>> sorted(Word([Integer(4),Integer(4),Integer(2),Integer(5),Integer(2),
↪Integer(1),Integer(4),Integer(1)]).evaluation_sparse())
[(1, 2), (2, 2), (4, 3), (5, 1)]
>>> sorted(Word("abcaccab").evaluation_sparse())
[('a', 3), ('b', 2), ('c', 3)]
```

exponent()

Return the exponent of `self`.

OUTPUT: integer; the exponent

EXAMPLES:

```
sage: Word('1231').exponent()
1
sage: Word('121212').exponent()
3
sage: Word().exponent()
0
```

```
>>> from sage.all import *
>>> Word('1231').exponent()
1
>>> Word('121212').exponent()
3
>>> Word().exponent()
0
```

factor_complexity(*n*)

Return the number of distinct factors of length `n` of `self`.

INPUT:

- `n` – the length of the factors

EXAMPLES:

```
sage: w = words.FibonacciWord()[:100]
sage: [w.factor_complexity(i) for i in range(20)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
>>> from sage.all import *
>>> w = words.FibonacciWord()[:Integer(100)]
>>> [w.factor_complexity(i) for i in range(Integer(20))]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
sage: w = words.ThueMorseWord()[:1000]
sage: [w.factor_complexity(i) for i in range(20)]
[1, 2, 4, 6, 10, 12, 16, 20, 22, 24, 28, 32, 36, 40, 42, 44, 46, 48, 52, 56]
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(1000)]
```

(continues on next page)

(continued from previous page)

```
>>> [w.factor_complexity(i) for i in range(Integer(20))]
[1, 2, 4, 6, 10, 12, 16, 20, 22, 24, 28, 32, 36, 40, 42, 44, 46, 48, 52, 56]
```

factor_iterator(*n=None*)

Generate distinct factors of `self`.

INPUT:

- `n` – integer or `None`

OUTPUT:

If `n` is an integer, returns an iterator over all distinct factors of length `n`. If `n` is `None`, returns an iterator generating all distinct factors.

EXAMPLES:

```
sage: w = Word('1213121')
sage: sorted( w.factor_iterator(0) )
[word: ]
sage: sorted( w.factor_iterator(10) )
[]
sage: sorted( w.factor_iterator(1) )
[word: 1, word: 2, word: 3]
sage: sorted( w.factor_iterator(4) )
[word: 1213, word: 1312, word: 2131, word: 3121]
sage: sorted( w.factor_iterator() )
[word: , word: 1, word: 12, word: 121, word: 1213, word: 12131, word: 121312,␣
↪word: 1213121, word: 13, word: 131, word: 1312, word: 13121, word: 2, word:␣
↪21, word: 213, word: 2131, word: 21312, word: 213121, word: 3, word: 31,␣
↪word: 312, word: 3121]
```

```
>>> from sage.all import *
>>> w = Word('1213121')
>>> sorted( w.factor_iterator(Integer(0)) )
[word: ]
>>> sorted( w.factor_iterator(Integer(10)) )
[]
>>> sorted( w.factor_iterator(Integer(1)) )
[word: 1, word: 2, word: 3]
>>> sorted( w.factor_iterator(Integer(4)) )
[word: 1213, word: 1312, word: 2131, word: 3121]
>>> sorted( w.factor_iterator() )
[word: , word: 1, word: 12, word: 121, word: 1213, word: 12131, word: 121312,␣
↪word: 1213121, word: 13, word: 131, word: 1312, word: 13121, word: 2, word:␣
↪21, word: 213, word: 2131, word: 21312, word: 213121, word: 3, word: 31,␣
↪word: 312, word: 3121]
```

```
sage: u = Word([1,2,1,2,3])
sage: sorted( u.factor_iterator(0) )
[word: ]
sage: sorted( u.factor_iterator(10) )
[]
sage: sorted( u.factor_iterator(1) )
```

(continues on next page)

(continued from previous page)

```
[word: 1, word: 2, word: 3]
sage: sorted( u.factor_iterator(5) )
[word: 12123]
sage: sorted( u.factor_iterator() )
[word: , word: 1, word: 12, word: 121, word: 1212, word: 12123, word: 123,␣
↪word: 2, word: 21, word: 212, word: 2123, word: 23, word: 3]
```

```
>>> from sage.all import *
>>> u = Word([Integer(1),Integer(2),Integer(1),Integer(2),Integer(3)])
>>> sorted( u.factor_iterator(Integer(0)) )
[word: ]
>>> sorted( u.factor_iterator(Integer(10)) )
[]
>>> sorted( u.factor_iterator(Integer(1)) )
[word: 1, word: 2, word: 3]
>>> sorted( u.factor_iterator(Integer(5)) )
[word: 12123]
>>> sorted( u.factor_iterator() )
[word: , word: 1, word: 12, word: 121, word: 1212, word: 12123, word: 123,␣
↪word: 2, word: 21, word: 212, word: 2123, word: 23, word: 3]
```

```
sage: xxx = Word("xxx")
sage: sorted( xxx.factor_iterator(0) )
[word: ]
sage: sorted( xxx.factor_iterator(4) )
[]
sage: sorted( xxx.factor_iterator(2) )
[word: xx]
sage: sorted( xxx.factor_iterator() )
[word: , word: x, word: xx, word: xxx]
```

```
>>> from sage.all import *
>>> xxx = Word("xxx")
>>> sorted( xxx.factor_iterator(Integer(0)) )
[word: ]
>>> sorted( xxx.factor_iterator(Integer(4)) )
[]
>>> sorted( xxx.factor_iterator(Integer(2)) )
[word: xx]
>>> sorted( xxx.factor_iterator() )
[word: , word: x, word: xx, word: xxx]
```

```
sage: e = Word()
sage: sorted( e.factor_iterator(0) )
[word: ]
sage: sorted( e.factor_iterator(17) )
[]
sage: sorted( e.factor_iterator() )
[word: ]
```

```
>>> from sage.all import *
>>> e = Word()
>>> sorted( e.factor_iterator(Integer(0)) )
[word: ]
>>> sorted( e.factor_iterator(Integer(17)) )
[]
>>> sorted( e.factor_iterator() )
[word: ]
```

factor_occurrences_in(*other*)

Return an iterator over all occurrences (including overlapping ones) of `self` in `other` in their order of appearance.

> **Warning**
>
> This method is deprecated since 2020 and will be removed in a later version of SageMath. Use `factor_occurrences_iterator()` instead.

EXAMPLES:

```
sage: u = Word('121')
sage: w = Word('121213211213')
sage: list(u.factor_occurrences_in(w))
doctest:warning
...
DeprecationWarning: f.factor_occurrences_in(w) is deprecated.
Use w.factor_occurrences_iterator(f) instead.
See https://github.com/sagemath/sage/issues/30187 for details.
[0, 2, 8]
```

```
>>> from sage.all import *
>>> u = Word('121')
>>> w = Word('121213211213')
>>> list(u.factor_occurrences_in(w))
doctest:warning
...
DeprecationWarning: f.factor_occurrences_in(w) is deprecated.
Use w.factor_occurrences_iterator(f) instead.
See https://github.com/sagemath/sage/issues/30187 for details.
[0, 2, 8]
```

factor_set(*n=None*, *algorithm='suffix tree'*)

Return the set of factors (of length `n`) of `self`.

INPUT:

- `n` – integer or `None` (default: `None`)
- `algorithm` – string (default: `'suffix tree'`), takes the following values:
  - `'suffix tree'` – construct and use the suffix tree of the word
  - `'naive'` – algorithm uses a sliding window

OUTPUT:

If `n` is an integer, returns the set of all distinct factors of length `n`. If `n` is `None`, returns the set of all distinct factors.

EXAMPLES:

```
sage: w = Word('121')
sage: sorted(w.factor_set())
[word: , word: 1, word: 12, word: 121, word: 2, word: 21]
sage: sorted(w.factor_set(algorithm='naive'))
[word: , word: 1, word: 12, word: 121, word: 2, word: 21]
```

```
>>> from sage.all import *
>>> w = Word('121')
>>> sorted(w.factor_set())
[word: , word: 1, word: 12, word: 121, word: 2, word: 21]
>>> sorted(w.factor_set(algorithm='naive'))
[word: , word: 1, word: 12, word: 121, word: 2, word: 21]
```

```
sage: w = Word('1213121')
sage: for i in range(w.length()): sorted(w.factor_set(i))
[word: ]
[word: 1, word: 2, word: 3]
[word: 12, word: 13, word: 21, word: 31]
[word: 121, word: 131, word: 213, word: 312]
[word: 1213, word: 1312, word: 2131, word: 3121]
[word: 12131, word: 13121, word: 21312]
[word: 121312, word: 213121]
```

```
>>> from sage.all import *
>>> w = Word('1213121')
>>> for i in range(w.length()): sorted(w.factor_set(i))
[word: ]
[word: 1, word: 2, word: 3]
[word: 12, word: 13, word: 21, word: 31]
[word: 121, word: 131, word: 213, word: 312]
[word: 1213, word: 1312, word: 2131, word: 3121]
[word: 12131, word: 13121, word: 21312]
[word: 121312, word: 213121]
```

```
sage: w = Word([1,2,1,2,3])
sage: s = w.factor_set()
sage: sorted(s)
[word: , word: 1, word: 12, word: 121, word: 1212, word: 12123, word: 123,␣
↪word: 2, word: 21, word: 212, word: 2123, word: 23, word: 3]
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(2),Integer(1),Integer(2),Integer(3)])
>>> s = w.factor_set()
>>> sorted(s)
[word: , word: 1, word: 12, word: 121, word: 1212, word: 12123, word: 123,␣
↪word: 2, word: 21, word: 212, word: 2123, word: 23, word: 3]
```

find(*sub*, *start=0*, *end=None*)

Return the index of the first occurrence of `sub` in `self`, such that `sub` is contained within `self[start:end]`. Return -1 on failure.

INPUT:

- `sub` – string, list, tuple or word to search for
- `start` – nonnegative integer (default: 0) specifying the position from which to start the search
- `end` – nonnegative integer (default: `None`); specifying the position at which the search must stop. If `None`, then the search is performed up to the end of the string.

OUTPUT: nonnegative integer or -1

EXAMPLES:

```
sage: w = Word([0,1,0,0,1])
sage: w.find(Word([1,0]))
1
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(0),Integer(0),Integer(1)])
>>> w.find(Word([Integer(1),Integer(0)]))
1
```

The `sub` argument can also be a tuple or a list:

```
sage: w.find([1,0])
1
sage: w.find((1,0))
1
```

```
>>> from sage.all import *
>>> w.find([Integer(1),Integer(0)])
1
>>> w.find((Integer(1),Integer(0)))
1
```

Examples using `start` and `end`:

```
sage: w.find(Word([0,1]), start=1)
3
sage: w.find(Word([0,1]), start=1, end=5)
3
sage: w.find(Word([0,1]), start=1, end=4) == -1
True
sage: w.find(Word([1,1])) == -1
True
sage: w.find("aa")
-1
```

```
>>> from sage.all import *
>>> w.find(Word([Integer(0),Integer(1)]), start=Integer(1))
3
>>> w.find(Word([Integer(0),Integer(1)]), start=Integer(1), end=Integer(5))
3
```

(continues on next page)

(continued from previous page)

```
>>> w.find(Word([Integer(0),Integer(1)]), start=Integer(1), end=Integer(4))␣
↪== -Integer(1)
True
>>> w.find(Word([Integer(1),Integer(1)])) == -Integer(1)
True
>>> w.find("aa")
-1
```

Instances of `Word_str` handle string inputs as well:

```
sage: w = Word('abac')
sage: w.find('a')
0
sage: w.find('ba')
1
```

```
>>> from sage.all import *
>>> w = Word('abac')
>>> w.find('a')
0
>>> w.find('ba')
1
```

first_pos_in(*other*)

Return the position of the first occurrence of `self` in `other`, or `None` if `self` is not a factor of `other`.

Warning

This method is deprecated since 2020 and will be removed in a later version of SageMath. Use `first_occurrence()` instead.

EXAMPLES:

```
sage: Word('12').first_pos_in(Word('131231'))
doctest:warning
...
DeprecationWarning: f.first_pos_in(w) is deprecated.
Use w.first_occurrence(f) instead.
See https://github.com/sagemath/sage/issues/30187 for details.
2
sage: Word('32').first_pos_in(Word('131231')) is None
True
```

```
>>> from sage.all import *
>>> Word('12').first_pos_in(Word('131231'))
doctest:warning
...
DeprecationWarning: f.first_pos_in(w) is deprecated.
Use w.first_occurrence(f) instead.
See https://github.com/sagemath/sage/issues/30187 for details.
2
```

(continues on next page)

(continued from previous page)

```
>>> Word('32').first_pos_in(Word('131231')) is None
True
```

foata_bijection()

Return word `self` under the Foata bijection.

The Foata bijection ϕ is a bijection on the set of words of given content (by a slight generalization of Section 2 in [FS1978]). It can be defined by induction on the size of the word: Given a word $w_1 w_2 \cdots w_n$, start with $\phi(w_1) = w_1$. At the i-th step, if $\phi(w_1 w_2 \cdots w_i) = v_1 v_2 \cdots v_i$, we define $\phi(w_1 w_2 \cdots w_i w_{i+1})$ by placing w_{i+1} on the end of the word $v_1 v_2 \cdots v_i$ and breaking the word up into blocks as follows. If $w_{i+1} \geq v_i$, place a vertical line to the right of each v_k for which $w_{i+1} \geq v_k$. Otherwise, if $w_{i+1} < v_i$, place a vertical line to the right of each v_k for which $w_{i+1} < v_k$. In either case, place a vertical line at the start of the word as well. Now, within each block between vertical lines, cyclically shift the entries one place to the right.

For instance, to compute $\phi([4, 1, 5, 4, 2, 2, 3])$, the sequence of words is

- 4,
- $|4|1 \to 41$,
- $|4|1|5 \to 415$,
- $|415|4 \to 5414$,
- $|5|4|14|2 \to 54412$,
- $|5441|2|2 \to 154422$,
- $|1|5442|2|3 \to 1254423$.

So $\phi([4, 1, 5, 4, 2, 2, 3]) = [1, 2, 5, 4, 4, 2, 3]$.

See also

Foata bijection on Permutations.

EXAMPLES:

```
sage: w = Word([2,2,2,1,1,1])
sage: w.foata_bijection()
word: 112221
sage: w = Word([2,2,1,2,2,2,1,1,2,1])
sage: w.foata_bijection()
word: 2122212211
sage: w = Word([4,1,5,4,2,2,3])
sage: w.foata_bijection()
word: 1254423
```

```
>>> from sage.all import *
>>> w = Word([Integer(2),Integer(2),Integer(2),Integer(1),Integer(1),
↪Integer(1)])
>>> w.foata_bijection()
word: 112221
>>> w = Word([Integer(2),Integer(2),Integer(1),Integer(2),Integer(2),
↪Integer(2),Integer(1),Integer(1),Integer(2),Integer(1)])
>>> w.foata_bijection()
```

(continues on next page)

(continued from previous page)

```
word: 2122212211
>>> w = Word([Integer(4),Integer(1),Integer(5),Integer(4),Integer(2),
↪Integer(2),Integer(3)])
>>> w.foata_bijection()
word: 1254423
```

good_suffix_table()

Return a table of the maximum skip you can do in order not to miss a possible occurrence of `self` in a word.

This is a part of the Boyer-Moore algorithm to find factors. See [BM1977].

EXAMPLES:

```
sage: Word('121321').good_suffix_table()
[5, 5, 5, 5, 3, 3, 1]
sage: Word('12412').good_suffix_table()
[3, 3, 3, 3, 3, 1]
```

```
>>> from sage.all import *
>>> Word('121321').good_suffix_table()
[5, 5, 5, 5, 3, 3, 1]
>>> Word('12412').good_suffix_table()
[3, 3, 3, 3, 3, 1]
```

has_period(*p*)

Return `True` if `self` has the period p, `False` otherwise.

Note

By convention, integers greater than the length of `self` are periods of `self`.

INPUT:

- `p` – integer to check if it is a period of `self`

EXAMPLES:

```
sage: w = Word('ababa')
sage: w.has_period(2)
True
sage: w.has_period(3)
False
sage: w.has_period(4)
True
sage: w.has_period(-1)
False
sage: w.has_period(5)
True
sage: w.has_period(6)
True
```

```
>>> from sage.all import *
>>> w = Word('ababa')
>>> w.has_period(Integer(2))
True
>>> w.has_period(Integer(3))
False
>>> w.has_period(Integer(4))
True
>>> w.has_period(-Integer(1))
False
>>> w.has_period(Integer(5))
True
>>> w.has_period(Integer(6))
True
```

has_prefix(*other*)

Test whether `self` has `other` as a prefix.

INPUT:

- `other` – a word, or data describing a word

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: u = Word("abbab")
sage: w.has_prefix(u)
True
sage: u.has_prefix(w)
False
sage: u.has_prefix("abbab")
True
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> u = Word("abbab")
>>> w.has_prefix(u)
True
>>> u.has_prefix(w)
False
>>> u.has_prefix("abbab")
True
```

```
sage: w = Word([0,1,1,0,1,0,0,1,0,1,0,1,0])
sage: u = Word([0,1,1,0,1])
sage: w.has_prefix(u)
True
sage: u.has_prefix(w)
False
sage: u.has_prefix([0,1,1,0,1])
True
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0)])
>>> u = Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1)])
>>> w.has_prefix(u)
True
>>> u.has_prefix(w)
False
>>> u.has_prefix([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1)])
True
```

has_suffix(*other*)

Test whether `self` has `other` as a suffix.

> **Note**
>
> Some word datatype classes, like `WordDatatype_str`, override this method.

INPUT:

- `other` – a word, or data describing a word

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: u = Word("ababa")
sage: w.has_suffix(u)
True
sage: u.has_suffix(w)
False
sage: u.has_suffix("ababa")
True
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> u = Word("ababa")
>>> w.has_suffix(u)
True
>>> u.has_suffix(w)
False
>>> u.has_suffix("ababa")
True
```

```
sage: w = Word([0,1,1,0,1,0,0,1,0,1,0,1,0])
sage: u = Word([0,1,0,1,0])
sage: w.has_suffix(u)
True
sage: u.has_suffix(w)
False
```

(continues on next page)

(continued from previous page)

```
sage: u.has_suffix([0,1,0,1,0])
True
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0)])
>>> u = Word([Integer(0),Integer(1),Integer(0),Integer(1),Integer(0)])
>>> w.has_suffix(u)
True
>>> u.has_suffix(w)
False
>>> u.has_suffix([Integer(0),Integer(1),Integer(0),Integer(1),Integer(0)])
True
```

implicit_suffix_tree()

Return the implicit suffix tree of `self`.

The *suffix tree* of a word w is a compactification of the suffix trie for w. The compactification removes all nodes that have exactly one incoming edge and exactly one outgoing edge. It consists of two components: a tree and a word. Thus, instead of labelling the edges by factors of w, we can label them by indices of the occurrence of the factors in w.

Type `sage.combinat.words.suffix_trees.ImplicitSuffixTree?` for more information.

EXAMPLES:

```
sage: w = Word("cacao")
sage: w.implicit_suffix_tree()
Implicit Suffix Tree of the word: cacao
```

```
>>> from sage.all import *
>>> w = Word("cacao")
>>> w.implicit_suffix_tree()
Implicit Suffix Tree of the word: cacao
```

```
sage: w = Word([0,1,0,1,1])
sage: w.implicit_suffix_tree()
Implicit Suffix Tree of the word: 01011
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(0),Integer(1),Integer(1)])
>>> w.implicit_suffix_tree()
Implicit Suffix Tree of the word: 01011
```

inv_lex_less(*other*)

Return `True` if `self` is inverse lexicographically less than `other`.

EXAMPLES:

```
sage: Word([1,2,4]).inv_lex_less(Word([1,3,2]))
False
```

(continues on next page)

(continued from previous page)

```
sage: Word([3,2,1]).inv_lex_less(Word([1,2,3]))
True
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(2),Integer(4)]).inv_lex_less(Word([Integer(1),
↪Integer(3),Integer(2)]))
False
>>> Word([Integer(3),Integer(2),Integer(1)]).inv_lex_less(Word([Integer(1),
↪Integer(2),Integer(3)]))
True
```

inversions()

Return a list of the inversions of `self`. An inversion is a pair (i, j) of nonnegative integers $i < j$ such that `self[i] > self[j]`.

EXAMPLES:

```
sage: Word([1,2,3,2,2,1]).inversions()
[[1, 5], [2, 3], [2, 4], [2, 5], [3, 5], [4, 5]]
sage: Words([3,2,1])([1,2,3,2,2,1]).inversions()
[[0, 1], [0, 2], [0, 3], [0, 4], [1, 2]]
sage: Word('abbaba').inversions()
[[1, 3], [1, 5], [2, 3], [2, 5], [4, 5]]
sage: Words('ba')('abbaba').inversions()
[[0, 1], [0, 2], [0, 4], [3, 4]]
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(2),Integer(3),Integer(2),Integer(2),Integer(1)]).
↪inversions()
[[1, 5], [2, 3], [2, 4], [2, 5], [3, 5], [4, 5]]
>>> Words([Integer(3),Integer(2),Integer(1)])([Integer(1),Integer(2),
↪Integer(3),Integer(2),Integer(2),Integer(1)]).inversions()
[[0, 1], [0, 2], [0, 3], [0, 4], [1, 2]]
>>> Word('abbaba').inversions()
[[1, 3], [1, 5], [2, 3], [2, 5], [4, 5]]
>>> Words('ba')('abbaba').inversions()
[[0, 1], [0, 2], [0, 4], [3, 4]]
```

is_balanced(*q=1*)

Return `True` if `self` is `q`-balanced, and `False` otherwise.

A finite or infinite word w is said to be *q-balanced* if for any two factors u, v of w of the same length, the difference between the number of x's in each of u and v is at most q for all letters x in the alphabet of w. A 1-balanced word is simply said to be balanced. See for instance [CFZ2000] and Chapter 2 of [Lot2002].

INPUT:

- `q` – integer (default: 1); the balance level

EXAMPLES:

```
sage: Word('1213121').is_balanced()
True
sage: Word('1122').is_balanced()
```

(continues on next page)

(continued from previous page)

```
False
sage: Word('121333121').is_balanced()
False
sage: Word('121333121').is_balanced(2)
False
sage: Word('121333121').is_balanced(3)
True
sage: Word('121122121').is_balanced()
False
sage: Word('121122121').is_balanced(2)
True
```

```
>>> from sage.all import *
>>> Word('1213121').is_balanced()
True
>>> Word('1122').is_balanced()
False
>>> Word('121333121').is_balanced()
False
>>> Word('121333121').is_balanced(Integer(2))
False
>>> Word('121333121').is_balanced(Integer(3))
True
>>> Word('121122121').is_balanced()
False
>>> Word('121122121').is_balanced(Integer(2))
True
```

is_cadence(*seq*)

Return `True` if `seq` is a cadence of `self`, and `False` otherwise.

A *cadence* is an increasing sequence of indexes that all map to the same letter.

EXAMPLES:

```
sage: Word('121132123').is_cadence([0, 2, 6])
True
sage: Word('121132123').is_cadence([0, 1, 2])
False
sage: Word('121132123').is_cadence([])
True
```

```
>>> from sage.all import *
>>> Word('121132123').is_cadence([Integer(0), Integer(2), Integer(6)])
True
>>> Word('121132123').is_cadence([Integer(0), Integer(1), Integer(2)])
False
>>> Word('121132123').is_cadence([])
True
```

is_christoffel()

Return `True` if `self` is a Christoffel word, and `False` otherwise.

The *Christoffel word* of slope p/q is obtained from the Cayley graph of $\mathbf{Z}/(p+q)\mathbf{Z}$ with generator q as follows. If $u \rightarrow v$ is an edge in the Cayley graph, then, $v = u + p \mod p + q$. Let a,`b` be the alphabet of w. Label the edge $u \rightarrow v$ by a if $u < v$ and b otherwise. The Christoffel word is the word obtained by reading the edge labels along the cycle beginning from 0.

Equivalently, w is a Christoffel word iff w is a symmetric non-empty word and $w[1 : n-1]$ is a palindrome.

See for instance [Ber2007] and [BLRS2009].

INPUT:

- `self` – word

OUTPUT: boolean; `True` if `self` is a Christoffel word, `False` otherwise

EXAMPLES:

```
sage: Word('00100101').is_christoffel()
True
sage: Word('aab').is_christoffel()
True
sage: Word().is_christoffel()
False
sage: Word('123123123').is_christoffel()
False
sage: Word('00100').is_christoffel()
False
sage: Word('0').is_christoffel()
True
```

```
>>> from sage.all import *
>>> Word('00100101').is_christoffel()
True
>>> Word('aab').is_christoffel()
True
>>> Word().is_christoffel()
False
>>> Word('123123123').is_christoffel()
False
>>> Word('00100').is_christoffel()
False
>>> Word('0').is_christoffel()
True
```

is_conjugate_with(*other*)

Return `True` if `self` is a conjugate of `other`, and `False` otherwise.

INPUT:

- `other` – a finite word

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word([0..20])
sage: z = Word([7..20] + [0..6])
sage: w
```

(continues on next page)

(continued from previous page)

```
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
sage: z
word: 7,8,9,10,11,12,13,14,15,16,17,18,19,20,0,1,2,3,4,5,6
sage: w.is_conjugate_with(z)
True
sage: z.is_conjugate_with(w)
True
sage: u = Word([4]*21)
sage: u.is_conjugate_with(w)
False
sage: u.is_conjugate_with(z)
False
```

```
>>> from sage.all import *
>>> w = Word((ellipsis_range(Integer(0),Ellipsis,Integer(20))))
>>> z = Word((ellipsis_range(Integer(7),Ellipsis,Integer(20))) + (ellipsis_
↪range(Integer(0),Ellipsis,Integer(6))))
>>> w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
>>> z
word: 7,8,9,10,11,12,13,14,15,16,17,18,19,20,0,1,2,3,4,5,6
>>> w.is_conjugate_with(z)
True
>>> z.is_conjugate_with(w)
True
>>> u = Word([Integer(4)]*Integer(21))
>>> u.is_conjugate_with(w)
False
>>> u.is_conjugate_with(z)
False
```

Both words must be finite:

```
sage: w = Word(iter([2]*100),length='unknown')
sage: z = Word([2]*100)
sage: z.is_conjugate_with(w) #TODO: Not implemented for word of unknown length
True
sage: wf = Word(iter([2]*100),length='finite')
sage: z.is_conjugate_with(wf)
True
sage: wf.is_conjugate_with(z)
True
```

```
>>> from sage.all import *
>>> w = Word(iter([Integer(2)]*Integer(100)),length='unknown')
>>> z = Word([Integer(2)]*Integer(100))
>>> z.is_conjugate_with(w) #TODO: Not implemented for word of unknown length
True
>>> wf = Word(iter([Integer(2)]*Integer(100)),length='finite')
>>> z.is_conjugate_with(wf)
True
>>> wf.is_conjugate_with(z)
```

(continues on next page)

(continued from previous page)

```
True
```

is_cube()

Return `True` if `self` is a cube, and `False` otherwise.

EXAMPLES:

```
sage: Word('012012012').is_cube()
True
sage: Word('01010101').is_cube()
False
sage: Word().is_cube()
True
sage: Word('012012').is_cube()
False
```

```
>>> from sage.all import *
>>> Word('012012012').is_cube()
True
>>> Word('01010101').is_cube()
False
>>> Word().is_cube()
True
>>> Word('012012').is_cube()
False
```

is_cube_free()

Return `True` if `self` does not contain cubes, and `False` otherwise.

EXAMPLES:

```
sage: Word('12312').is_cube_free()
True
sage: Word('32221').is_cube_free()
False
sage: Word().is_cube_free()
True
```

```
>>> from sage.all import *
>>> Word('12312').is_cube_free()
True
>>> Word('32221').is_cube_free()
False
>>> Word().is_cube_free()
True
```

is_empty()

Return `True` if the length of `self` is zero, and `False` otherwise.

EXAMPLES:

```
sage: Word([]).is_empty()
True
```

(continues on next page)

(continued from previous page)

```
sage: Word('a').is_empty()
False
```

```
>>> from sage.all import *
>>> Word([]).is_empty()
True
>>> Word('a').is_empty()
False
```

is_factor(*other*)

Return `True` if `self` is a factor of `other`, and `False` otherwise.

A finite word $u \in A^*$ is a *factor* of a finite word $v \in A^*$ if there exists $p, s \in A^*$ such that $v = pus$.

EXAMPLES:

```
sage: u = Word('2113')
sage: w = Word('123121332131233121132123')
sage: u.is_factor(w)
True
sage: u = Word('321')
sage: w = Word('1231241231312312312')
sage: u.is_factor(w)
False
```

```
>>> from sage.all import *
>>> u = Word('2113')
>>> w = Word('123121332131233121132123')
>>> u.is_factor(w)
True
>>> u = Word('321')
>>> w = Word('1231241231312312312')
>>> u.is_factor(w)
False
```

The empty word is factor of another word:

```
sage: Word().is_factor(Word())
True
sage: Word().is_factor(Word('a'))
True
sage: Word().is_factor(Word([1,2,3]))
True
sage: Word().is_factor(Word(lambda n:n, length=5))
True
```

```
>>> from sage.all import *
>>> Word().is_factor(Word())
True
>>> Word().is_factor(Word('a'))
True
>>> Word().is_factor(Word([Integer(1),Integer(2),Integer(3)]))
```

(continues on next page)

(continued from previous page)

```
True
>>> Word().is_factor(Word(lambda n:n, length=Integer(5)))
True
```

is_finite()

Return True.

EXAMPLES:

```
sage: Word([]).is_finite()
True
sage: Word('a').is_finite()
True
```

```
>>> from sage.all import *
>>> Word([]).is_finite()
True
>>> Word('a').is_finite()
True
```

is_full(*f=None*)

Return True if self has defect 0, and False otherwise.

A word is *full* (or *rich*) if its defect is zero (see [BHNR2004]).

If f is given, then the f-palindromic defect is used (see [PeSt2011]).

INPUT:

- f – involution (default: None) on the alphabet of self; it must be callable on letters as well as words (e.g. WordMorphism)

OUTPUT:

boolean – If f is None, whether self is full; otherwise, whether self is full of f-palindromes.

EXAMPLES:

```
sage: words.ThueMorseWord()[:100].is_full()
False
sage: words.FibonacciWord()[:100].is_full()
True
sage: Word('000000000000000').is_full()
True
sage: Word('011010011001').is_full()
False
sage: Word('2194').is_full()
True
sage: Word().is_full()
True
```

```
>>> from sage.all import *
>>> words.ThueMorseWord()[:Integer(100)].is_full()
False
>>> words.FibonacciWord()[:Integer(100)].is_full()
```

(continues on next page)

(continued from previous page)

```
True
>>> Word('000000000000000').is_full()
True
>>> Word('011010011001').is_full()
False
>>> Word('2194').is_full()
True
>>> Word().is_full()
True
```

```
sage: f = WordMorphism('a->b,b->a')
sage: Word().is_full(f)
True
sage: w = Word('ab')
sage: w.is_full()
True
sage: w.is_full(f)
True
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> Word().is_full(f)
True
>>> w = Word('ab')
>>> w.is_full()
True
>>> w.is_full(f)
True
```

```
sage: f = WordMorphism('a->b,b->a')
sage: Word('abab').is_full(f)
True
sage: Word('abba').is_full(f)
False
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> Word('abab').is_full(f)
True
>>> Word('abba').is_full(f)
False
```

A simple example of an infinite word full of f-palindromes:

```
sage: p = WordMorphism({0:'abc',1:'ab'})
sage: f = WordMorphism('a->b,b->a,c->c')
sage: p(words.FibonacciWord()[:50]).is_full(f)
True
sage: p(words.FibonacciWord()[:150]).is_full(f)
True
```

```
>>> from sage.all import *
>>> p = WordMorphism({Integer(0):'abc',Integer(1):'ab'})
>>> f = WordMorphism('a->b,b->a,c->c')
>>> p(words.FibonacciWord()[:Integer(50)]).is_full(f)
True
>>> p(words.FibonacciWord()[:Integer(150)]).is_full(f)
True
```

is_lyndon()

Return `True` if `self` is a Lyndon word, and `False` otherwise.

A *Lyndon word* is a non-empty word that is lexicographically smaller than each of its proper suffixes (for the given order on its alphabet). That is, w is a Lyndon word if w is non-empty and for each factorization $w = uv$ (with u, v both non-empty), we have $w < v$.

Equivalently, w is a Lyndon word iff w is a non-empty word that is lexicographically smaller than each of its proper conjugates for the given order on its alphabet.

See for instance [Lot1983].

EXAMPLES:

```
sage: Word('123132133').is_lyndon()
True
sage: Word().is_lyndon()
False
sage: Word('122112').is_lyndon()
False
```

```
>>> from sage.all import *
>>> Word('123132133').is_lyndon()
True
>>> Word().is_lyndon()
False
>>> Word('122112').is_lyndon()
False
```

is_overlap()

Return `True` if `self` is an overlap, and `False` otherwise.

EXAMPLES:

```
sage: Word('12121').is_overlap()
True
sage: Word('123').is_overlap()
False
sage: Word('1231').is_overlap()
False
sage: Word('123123').is_overlap()
False
sage: Word('1231231').is_overlap()
True
sage: Word().is_overlap()
False
```

```
>>> from sage.all import *
>>> Word('12121').is_overlap()
True
>>> Word('123').is_overlap()
False
>>> Word('1231').is_overlap()
False
>>> Word('123123').is_overlap()
False
>>> Word('1231231').is_overlap()
True
>>> Word().is_overlap()
False
```

is_palindrome(*f=None*)

Return True if self is a palindrome (or a f-palindrome), and False otherwise.

Let $f : \Sigma \to \Sigma$ be an involution that extends to a morphism on Σ^*. We say that $w \in \Sigma^*$ is a *`f`-palindrome* if $w = f(\tilde{w})$ [Lab2008]. Also called *`f`-pseudo-palindrome* [AZZ2005].

INPUT:

- f – involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. *WordMorphism*). The default value corresponds to usual palindromes, i.e., f equal to the identity.

EXAMPLES:

```
sage: Word('esope reste ici et se repose').is_palindrome()
False
sage: Word('esoperesteicietserepose').is_palindrome()
True
sage: Word('I saw I was I').is_palindrome()
True
sage: Word('abbcbba').is_palindrome()
True
sage: Word('abcbdba').is_palindrome()
False
```

```
>>> from sage.all import *
>>> Word('esope reste ici et se repose').is_palindrome()
False
>>> Word('esoperesteicietserepose').is_palindrome()
True
>>> Word('I saw I was I').is_palindrome()
True
>>> Word('abbcbba').is_palindrome()
True
>>> Word('abcbdba').is_palindrome()
False
```

Some f-palindromes:

```
sage: f = WordMorphism('a->b,b->a')
sage: Word('aababb').is_palindrome(f)
True
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> Word('aababb').is_palindrome(f)
True
```

```
sage: f = WordMorphism('a->b,b->a,c->c')
sage: Word('abacbacbab').is_palindrome(f)
True
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a,c->c')
>>> Word('abacbacbab').is_palindrome(f)
True
```

```
sage: f = WordMorphism({'a':'b','b':'a'})
sage: Word('aababb').is_palindrome(f)
True
```

```
>>> from sage.all import *
>>> f = WordMorphism({'a':'b','b':'a'})
>>> Word('aababb').is_palindrome(f)
True
```

```
sage: f = WordMorphism({0:[1],1:[0]})
sage: w = words.ThueMorseWord()[:8]; w
word: 01101001
sage: w.is_palindrome(f)
True
```

```
>>> from sage.all import *
>>> f = WordMorphism({Integer(0):[Integer(1)],Integer(1):[Integer(0)]})
>>> w = words.ThueMorseWord()[:Integer(8)]; w
word: 01101001
>>> w.is_palindrome(f)
True
```

The word must be in the domain of the involution:

```
sage: f = WordMorphism('a->a')
sage: Word('aababb').is_palindrome(f)
Traceback (most recent call last):
...
KeyError: 'b'
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->a')
>>> Word('aababb').is_palindrome(f)
Traceback (most recent call last):
...
KeyError: 'b'
```

is_prefix(*other*)

Return `True` if `self` is a prefix of `other`, and `False` otherwise.

EXAMPLES:

```
sage: w = Word('0123456789')
sage: y = Word('012345')
sage: y.is_prefix(w)
True
sage: w.is_prefix(y)
False
sage: w.is_prefix(Word())
False
sage: Word().is_prefix(w)
True
sage: Word().is_prefix(Word())
True
```

```
>>> from sage.all import *
>>> w = Word('0123456789')
>>> y = Word('012345')
>>> y.is_prefix(w)
True
>>> w.is_prefix(y)
False
>>> w.is_prefix(Word())
False
>>> Word().is_prefix(w)
True
>>> Word().is_prefix(Word())
True
```

is_primitive()

Return `True` if `self` is primitive, and `False` otherwise.

A finite word w is *primitive* if it is not a positive integer power of a shorter word.

EXAMPLES:

```
sage: Word('1231').is_primitive()
True
sage: Word('111').is_primitive()
False
```

```
>>> from sage.all import *
>>> Word('1231').is_primitive()
True
>>> Word('111').is_primitive()
False
```

is_proper_prefix(*other*)

Return `True` if `self` is a proper prefix of `other`, and `False` otherwise.

EXAMPLES:

```
sage: Word('12').is_proper_prefix(Word('123'))
True
sage: Word('12').is_proper_prefix(Word('12'))
False
sage: Word().is_proper_prefix(Word('123'))
True
sage: Word('123').is_proper_prefix(Word('12'))
False
sage: Word().is_proper_prefix(Word())
False
```

```
>>> from sage.all import *
>>> Word('12').is_proper_prefix(Word('123'))
True
>>> Word('12').is_proper_prefix(Word('12'))
False
>>> Word().is_proper_prefix(Word('123'))
True
>>> Word('123').is_proper_prefix(Word('12'))
False
>>> Word().is_proper_prefix(Word())
False
```

is_proper_suffix(*other*)

Return `True` if `self` is a proper suffix of `other`, and `False` otherwise.

EXAMPLES:

```
sage: Word('23').is_proper_suffix(Word('123'))
True
sage: Word('12').is_proper_suffix(Word('12'))
False
sage: Word().is_proper_suffix(Word('123'))
True
sage: Word('123').is_proper_suffix(Word('12'))
False
```

```
>>> from sage.all import *
>>> Word('23').is_proper_suffix(Word('123'))
True
>>> Word('12').is_proper_suffix(Word('12'))
False
>>> Word().is_proper_suffix(Word('123'))
True
>>> Word('123').is_proper_suffix(Word('12'))
False
```

is_quasiperiodic()

Return `True` if `self` is quasiperiodic, and `False` otherwise.

A finite or infinite word w is *quasiperiodic* if it can be constructed by concatenations and superpositions of one of its proper factors u, which is called a *quasiperiod* of w. See for instance [AE1993], [Mar2004], and [GLR2008].

EXAMPLES:

```
sage: Word('abaababaabaababaaba').is_quasiperiodic()
True
sage: Word('abacaba').is_quasiperiodic()
False
sage: Word('a').is_quasiperiodic()
False
sage: Word().is_quasiperiodic()
False
sage: Word('abaaba').is_quasiperiodic()
True
```

```
>>> from sage.all import *
>>> Word('abaababaabaababaaba').is_quasiperiodic()
True
>>> Word('abacaba').is_quasiperiodic()
False
>>> Word('a').is_quasiperiodic()
False
>>> Word().is_quasiperiodic()
False
>>> Word('abaaba').is_quasiperiodic()
True
```

is_rich(*f=None*)

Return `True` if `self` has defect 0, and `False` otherwise.

A word is *full* (or *rich*) if its defect is zero (see [BHNR2004]).

If `f` is given, then the `f`-palindromic defect is used (see [PeSt2011]).

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`; it must be callable on letters as well as words (e.g. `WordMorphism`)

OUTPUT:

boolean – If `f` is `None`, whether `self` is full; otherwise, whether `self` is full of `f`-palindromes.

EXAMPLES:

```
sage: words.ThueMorseWord()[:100].is_full()
False
sage: words.FibonacciWord()[:100].is_full()
True
sage: Word('000000000000000').is_full()
True
sage: Word('011010011001').is_full()
False
sage: Word('2194').is_full()
True
sage: Word().is_full()
True
```

```
>>> from sage.all import *
>>> words.ThueMorseWord()[:Integer(100)].is_full()
False
>>> words.FibonacciWord()[:Integer(100)].is_full()
True
>>> Word('000000000000000').is_full()
True
>>> Word('011010011001').is_full()
False
>>> Word('2194').is_full()
True
>>> Word().is_full()
True
```

```
sage: f = WordMorphism('a->b,b->a')
sage: Word().is_full(f)
True
sage: w = Word('ab')
sage: w.is_full()
True
sage: w.is_full(f)
True
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> Word().is_full(f)
True
>>> w = Word('ab')
>>> w.is_full()
True
>>> w.is_full(f)
True
```

```
sage: f = WordMorphism('a->b,b->a')
sage: Word('abab').is_full(f)
True
sage: Word('abba').is_full(f)
False
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> Word('abab').is_full(f)
True
>>> Word('abba').is_full(f)
False
```

A simple example of an infinite word full of f-palindromes:

```
sage: p = WordMorphism({0:'abc',1:'ab'})
sage: f = WordMorphism('a->b,b->a,c->c')
sage: p(words.FibonacciWord()[:50]).is_full(f)
True
```

(continues on next page)

(continued from previous page)

```
sage: p(words.FibonacciWord()[:150]).is_full(f)
True
```

```
>>> from sage.all import *
>>> p = WordMorphism({Integer(0):'abc',Integer(1):'ab'})
>>> f = WordMorphism('a->b,b->a,c->c')
>>> p(words.FibonacciWord()[:Integer(50)]).is_full(f)
True
>>> p(words.FibonacciWord()[:Integer(150)]).is_full(f)
True
```

is_smooth_prefix()

Return `True` if `self` is the prefix of a smooth word, and `False` otherwise.

Let $A_k = \{1, \ldots, k\}$, $k \geq 2$. An infinite word w in A_k^ω is said to be *smooth* if and only if for all positive integers m, $\Delta^m(w)$ is in A_k^ω, where $\Delta(w)$ is the word obtained from w by composing the length of consecutive runs of the same letter in w. See for instance [BL2003] and [BDLV2006].

INPUT:

- `self` – must be a word over the integers to get something other than `False`

OUTPUT: boolean; whether `self` is a smooth prefix or not

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([1, 1, 2, 2, 1, 2, 1, 1]).is_smooth_prefix()
True
sage: W([1, 2, 1, 2, 1, 2]).is_smooth_prefix()
False
```

```
>>> from sage.all import *
>>> W = Words([Integer(1), Integer(2)])
>>> W([Integer(1), Integer(1), Integer(2), Integer(2), Integer(1), Integer(2),
↪ Integer(1), Integer(1)]).is_smooth_prefix()
True
>>> W([Integer(1), Integer(2), Integer(1), Integer(2), Integer(1),␣
↪Integer(2)]).is_smooth_prefix()
False
```

is_square()

Return `True` if `self` is a square, and `False` otherwise.

EXAMPLES:

```
sage: Word([1,0,0,1]).is_square()
False
sage: Word('1212').is_square()
True
sage: Word('1213').is_square()
False
sage: Word('12123').is_square()
False
```

(continues on next page)

(continued from previous page)

```
sage: Word().is_square()
True
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(0),Integer(0),Integer(1)]).is_square()
False
>>> Word('1212').is_square()
True
>>> Word('1213').is_square()
False
>>> Word('12123').is_square()
False
>>> Word().is_square()
True
```

is_square_free()

Return `True` if `self` does not contain squares, and `False` otherwise.

EXAMPLES:

```
sage: Word('12312').is_square_free()
True
sage: Word('31212').is_square_free()
False
sage: Word().is_square_free()
True
```

```
>>> from sage.all import *
>>> Word('12312').is_square_free()
True
>>> Word('31212').is_square_free()
False
>>> Word().is_square_free()
True
```

is_sturmian_factor()

Tell whether `self` is a factor of a Sturmian word.

The finite word `self` must be defined on a two-letter alphabet.

Equivalently, tells whether `self` is balanced. The advantage over the `is_balanced` method is that this one runs in linear time whereas `is_balanced` runs in quadratic time.

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word('0111011011011101101',alphabet='01')
sage: w.is_sturmian_factor()
True
```

```
>>> from sage.all import *
>>> w = Word('0111011011011101101',alphabet='01')
```

(continues on next page)

(continued from previous page)

```
>>> w.is_sturmian_factor()
True
```

```
sage: words.LowerMechanicalWord(random(),alphabet='01')[:100].is_sturmian_
↪factor()
True
sage: words.CharacteristicSturmianWord(random())[:100].is_sturmian_factor()  ␣
↪            # needs sage.rings.real_mpfr
True
```

```
>>> from sage.all import *
>>> words.LowerMechanicalWord(random(),alphabet='01')[:Integer(100)].is_
↪sturmian_factor()
True
>>> words.CharacteristicSturmianWord(random())[:Integer(100)].is_sturmian_
↪factor()              # needs sage.rings.real_mpfr
True
```

```
sage: w = Word('aabb',alphabet='ab')
sage: w.is_sturmian_factor()
False

sage: s1 = WordMorphism('a->ab,b->b')
sage: s2 = WordMorphism('a->ba,b->b')
sage: s3 = WordMorphism('a->a,b->ba')
sage: s4 = WordMorphism('a->a,b->ab')
sage: W = Words('ab')
sage: w = W('ab')
sage: for i in range(8): w = choice([s1,s2,s3,s4])(w)
sage: w.is_sturmian_factor()
True
```

```
>>> from sage.all import *
>>> w = Word('aabb',alphabet='ab')
>>> w.is_sturmian_factor()
False

>>> s1 = WordMorphism('a->ab,b->b')
>>> s2 = WordMorphism('a->ba,b->b')
>>> s3 = WordMorphism('a->a,b->ba')
>>> s4 = WordMorphism('a->a,b->ab')
>>> W = Words('ab')
>>> w = W('ab')
>>> for i in range(Integer(8)): w = choice([s1,s2,s3,s4])(w)
>>> w.is_sturmian_factor()
True
```

Famous words:

```
sage: words.FibonacciWord()[:100].is_sturmian_factor()
True
```

(continues on next page)

(continued from previous page)

```
sage: words.ThueMorseWord()[:1000].is_sturmian_factor()
False
sage: words.KolakoskiWord()[:1000].is_sturmian_factor()
False
```

```
>>> from sage.all import *
>>> words.FibonacciWord()[:Integer(100)].is_sturmian_factor()
True
>>> words.ThueMorseWord()[:Integer(1000)].is_sturmian_factor()
False
>>> words.KolakoskiWord()[:Integer(1000)].is_sturmian_factor()
False
```

See [Arn2002], [Ser1985], and [SU2009].

AUTHOR:

- Thierry Monteil

is_subword_of(*other*)

Return `True` if `self` is a subword of `other`, and `False` otherwise.

A finite word u is a *subword* of a finite word v if u is a subsequence of v. See Chapter 6 on Subwords in [Lot1997].

Some references define subword as a consecutive subsequence. Use *is_factor()* if this is what you need.

INPUT:

- `other` – a finite word

EXAMPLES:

```
sage: Word('bb').is_subword_of(Word('ababa'))
True
sage: Word('bbb').is_subword_of(Word('ababa'))
False
```

```
>>> from sage.all import *
>>> Word('bb').is_subword_of(Word('ababa'))
True
>>> Word('bbb').is_subword_of(Word('ababa'))
False
```

```
sage: Word().is_subword_of(Word('123'))
True
sage: Word('123').is_subword_of(Word('3211333213233321'))
True
sage: Word('321').is_subword_of(Word('11122212112122133111222332'))
False
```

```
>>> from sage.all import *
>>> Word().is_subword_of(Word('123'))
True
>>> Word('123').is_subword_of(Word('3211333213233321'))
```

(continues on next page)

(continued from previous page)

```
True
>>> Word('321').is_subword_of(Word('11122212112122133111222332'))
False
```

See also

longest_common_subword() number_of_subword_occurrences() is_factor()

is_suffix(*other*)

Return `True` if `self` is a suffix of `other`, and `False` otherwise.

EXAMPLES:

```
sage: w = Word('0123456789')
sage: y = Word('56789')
sage: y.is_suffix(w)
True
sage: w.is_suffix(y)
False
sage: Word('579').is_suffix(w)
False
sage: Word().is_suffix(y)
True
sage: w.is_suffix(Word())
False
sage: Word().is_suffix(Word())
True
```

```
>>> from sage.all import *
>>> w = Word('0123456789')
>>> y = Word('56789')
>>> y.is_suffix(w)
True
>>> w.is_suffix(y)
False
>>> Word('579').is_suffix(w)
False
>>> Word().is_suffix(y)
True
>>> w.is_suffix(Word())
False
>>> Word().is_suffix(Word())
True
```

is_symmetric(*f=None*)

Return `True` if `self` is symmetric (or `f`-symmetric), and `False` otherwise.

A word is *symmetric* (resp. *f-symmetric*) if it is the product of two palindromes (resp. f-palindromes). See [BHNR2004] and [DeLuca2006].

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`; it must be callable on letters as well as words (e.g. `WordMorphism`)

EXAMPLES:

```
sage: Word('abbabab').is_symmetric()
True
sage: Word('ababa').is_symmetric()
True
sage: Word('aababaabba').is_symmetric()
False
sage: Word('aabbbaababba').is_symmetric()
False
sage: f = WordMorphism('a->b,b->a')
sage: Word('aabbbaababba').is_symmetric(f)
True
```

```
>>> from sage.all import *
>>> Word('abbabab').is_symmetric()
True
>>> Word('ababa').is_symmetric()
True
>>> Word('aababaabba').is_symmetric()
False
>>> Word('aabbbaababba').is_symmetric()
False
>>> f = WordMorphism('a->b,b->a')
>>> Word('aabbbaababba').is_symmetric(f)
True
```

is_tangent()

Tell whether `self` is a tangent word.

The finite word `self` must be defined on a two-letter alphabet.

A binary word is said to be *tangent* if it can appear in infinitely many cutting sequences of a smooth curve, where each cutting sequence is observed on a progressively smaller grid.

This class of words strictly contains the class of 1-balanced words, and is strictly contained in the class of 2-balanced words.

This method runs in linear time.

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word('01110110110111011101',alphabet='01')
sage: w.is_tangent()
True
```

```
>>> from sage.all import *
>>> w = Word('01110110110111011101',alphabet='01')
>>> w.is_tangent()
True
```

Some tangent words may not be balanced:

```
sage: Word('aabb',alphabet='ab').is_balanced()
False
sage: Word('aabb',alphabet='ab').is_tangent()
True
```

```
>>> from sage.all import *
>>> Word('aabb',alphabet='ab').is_balanced()
False
>>> Word('aabb',alphabet='ab').is_tangent()
True
```

Some 2-balanced words may not be tangent:

```
sage: Word('aaabb',alphabet='ab').is_tangent()
False
sage: Word('aaabb',alphabet='ab').is_balanced(2)
True
```

```
>>> from sage.all import *
>>> Word('aaabb',alphabet='ab').is_tangent()
False
>>> Word('aaabb',alphabet='ab').is_balanced(Integer(2))
True
```

Famous words:

```
sage: words.FibonacciWord()[:100].is_tangent()
True
sage: words.ThueMorseWord()[:1000].is_tangent()
True
sage: words.KolakoskiWord()[:1000].is_tangent()
False
```

```
>>> from sage.all import *
>>> words.FibonacciWord()[:Integer(100)].is_tangent()
True
>>> words.ThueMorseWord()[:Integer(1000)].is_tangent()
True
>>> words.KolakoskiWord()[:Integer(1000)].is_tangent()
False
```

See [Mon2010].

AUTHOR:

- Thierry Monteil

is_yamanouchi(*n=None*)

Return whether `self` is Yamanouchi.

A word w is Yamanouchi if, when read from right to left, it always has weakly more i's than $i + 1$'s for all i that appear in w.

INPUT:

- `n` – (optional) an integer specifying the maximal letter in the alphabet

EXAMPLES:

```
sage: w = Word([1,2,4,3,2,2,2])
sage: w.is_yamanouchi()
False
sage: w = Word([2,3,4,3,1,2,1,1,2,1])
sage: w.is_yamanouchi()
True
sage: w = Word([3,1])
sage: w.is_yamanouchi(n=3)
False
sage: w.is_yamanouchi()
True
sage: w = Word([3,1],alphabet=[1,2,3])
sage: w.is_yamanouchi()
False
sage: w = Word([2,1,1,2])
sage: w.is_yamanouchi()
False
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(2),Integer(4),Integer(3),Integer(2),
↪Integer(2),Integer(2)])
>>> w.is_yamanouchi()
False
>>> w = Word([Integer(2),Integer(3),Integer(4),Integer(3),Integer(1),
↪Integer(2),Integer(1),Integer(1),Integer(2),Integer(1)])
>>> w.is_yamanouchi()
True
>>> w = Word([Integer(3),Integer(1)])
>>> w.is_yamanouchi(n=Integer(3))
False
>>> w.is_yamanouchi()
True
>>> w = Word([Integer(3),Integer(1)],alphabet=[Integer(1),Integer(2),
↪Integer(3)])
>>> w.is_yamanouchi()
False
>>> w = Word([Integer(2),Integer(1),Integer(1),Integer(2)])
>>> w.is_yamanouchi()
False
```

iterated_left_palindromic_closure(*f=None*)

Return the iterated left (f-)palindromic closure of self.

INPUT:

- f – involution (default: None) on the alphabet of self; it must be callable on letters as well as words (e.g. WordMorphism)

OUTPUT: word; the left iterated f-palindromic closure of self

EXAMPLES:

```
sage: Word('123').iterated_left_palindromic_closure()
word: 3231323
sage: f = WordMorphism('a->b,b->a')
sage: Word('ab').iterated_left_palindromic_closure(f=f)
word: abbaab
sage: Word('aab').iterated_left_palindromic_closure(f=f)
word: abbaabbaab
```

```
>>> from sage.all import *
>>> Word('123').iterated_left_palindromic_closure()
word: 3231323
>>> f = WordMorphism('a->b,b->a')
>>> Word('ab').iterated_left_palindromic_closure(f=f)
word: abbaab
>>> Word('aab').iterated_left_palindromic_closure(f=f)
word: abbaabbaab
```

lacunas(*f=None*)

Return the list of all the lacunas of `self`.

A *lacuna* is a position in a word where the longest (f-)palindromic suffix is not unioccurrent (see [BMBL2008]).

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`. It must be callable on letters as well as words (e.g. `WordMorphism`). The default value corresponds to usual palindromes, i.e., `f` equal to the identity.

OUTPUT: list of all the lacunas of self

EXAMPLES:

```
sage: w = Word([0,1,1,2,3,4,5,1,13,3])
sage: w.lacunas()
[7, 9]
sage: words.ThueMorseWord()[:100].lacunas()
[8, 9, 24, 25, 32, 33, 34, 35, 36, 37, 38, 39, 96, 97, 98, 99]
sage: f = WordMorphism({0:[1],1:[0]})
sage: words.ThueMorseWord()[:50].lacunas(f)
[0, 2, 4, 12, 16, 17, 18, 19, 48, 49]
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(1),Integer(13),Integer(3)])
>>> w.lacunas()
[7, 9]
>>> words.ThueMorseWord()[:Integer(100)].lacunas()
[8, 9, 24, 25, 32, 33, 34, 35, 36, 37, 38, 39, 96, 97, 98, 99]
>>> f = WordMorphism({Integer(0):[Integer(1)],Integer(1):[Integer(0)]})
>>> words.ThueMorseWord()[:Integer(50)].lacunas(f)
[0, 2, 4, 12, 16, 17, 18, 19, 48, 49]
```

last_position_dict()

Return a dictionary that contains the last position of each letter in `self`.

EXAMPLES:

```
sage: Word('1231232').last_position_dict()
{'1': 3, '2': 6, '3': 5}
```

```
>>> from sage.all import *
>>> Word('1231232').last_position_dict()
{'1': 3, '2': 6, '3': 5}
```

left_special_factors(*n=None*)

Return the left special factors (of length `n`).

A factor u of a word w is *left special* if there are two distinct letters a and b such that au and bu are factors of w.

INPUT:

- `n` – integer (default: `None`); if `None`, it returns all left special factors

OUTPUT: list of words

EXAMPLES:

```
sage: alpha, beta, x = 0.54, 0.294, 0.1415
sage: w = words.CodingOfRotationWord(alpha, beta, x)[:40]
sage: for i in range(5):
....:     print("{} {}".format(i, sorted(w.left_special_factors(i))))
0 [word: ]
1 [word: 0]
2 [word: 00, word: 01]
3 [word: 000, word: 010]
4 [word: 0000, word: 0101]
```

```
>>> from sage.all import *
>>> alpha, beta, x = RealNumber('0.54'), RealNumber('0.294'), RealNumber('0.
↪1415')
>>> w = words.CodingOfRotationWord(alpha, beta, x)[:Integer(40)]
>>> for i in range(Integer(5)):
...     print("{} {}".format(i, sorted(w.left_special_factors(i))))
0 [word: ]
1 [word: 0]
2 [word: 00, word: 01]
3 [word: 000, word: 010]
4 [word: 0000, word: 0101]
```

left_special_factors_iterator(*n=None*)

Return an iterator over the left special factors (of length `n`).

A factor u of a word w is *left special* if there are two distinct letters a and b such that au and bu are factors of w.

INPUT:

- `n` – integer (default: `None`); if `None`, it returns an iterator over all left special factors

EXAMPLES:

```
sage: alpha, beta, x = 0.54, 0.294, 0.1415
sage: w = words.CodingOfRotationWord(alpha, beta, x)[:40]
sage: sorted(w.left_special_factors_iterator(3))
[word: 000, word: 010]
sage: sorted(w.left_special_factors_iterator(4))
[word: 0000, word: 0101]
sage: sorted(w.left_special_factors_iterator(5))
[word: 00000, word: 01010]
```

```
>>> from sage.all import *
>>> alpha, beta, x = RealNumber('0.54'), RealNumber('0.294'), RealNumber('0.
↪1415')
>>> w = words.CodingOfRotationWord(alpha, beta, x)[:Integer(40)]
>>> sorted(w.left_special_factors_iterator(Integer(3)))
[word: 000, word: 010]
>>> sorted(w.left_special_factors_iterator(Integer(4)))
[word: 0000, word: 0101]
>>> sorted(w.left_special_factors_iterator(Integer(5)))
[word: 00000, word: 01010]
```

length()

Return the length of `self`.

length_border()

Return the length of the border of `self`.

The *border* of a word is the longest word that is both a proper prefix and a proper suffix of `self`.

EXAMPLES:

```
sage: Word('121').length_border()
1
sage: Word('1').length_border()
0
sage: Word('1212').length_border()
2
sage: Word('111').length_border()
2
sage: Word().length_border() is None
True
```

```
>>> from sage.all import *
>>> Word('121').length_border()
1
>>> Word('1').length_border()
0
>>> Word('1212').length_border()
2
>>> Word('111').length_border()
2
>>> Word().length_border() is None
True
```

length_maximal_palindrome(*j*, *m=None*, *f=None*)

Return the length of the longest palindrome centered at position `j`.

INPUT:

- `j` – rational; position of the symmetry axis of the palindrome. Must return an integer when doubled. It is an integer when the center of the palindrome is a letter.
- `m` – integer (default: `None`); minimal length of palindrome, if known. The parity of `m` can't be the same as the parity of `2j`.
- `f` – involution (default: `None`) on the alphabet; it must be callable on letters as well as words (e.g. `WordMorphism`)

OUTPUT: length of the longest `f`-palindrome centered at position `j`

EXAMPLES:

```
sage: Word('01001010').length_maximal_palindrome(3/2)
0
sage: Word('01101001').length_maximal_palindrome(3/2)
4
sage: Word('01010').length_maximal_palindrome(j=3, f='0->1,1->0')
0
sage: Word('01010').length_maximal_palindrome(j=2.5, f='0->1,1->0')
4
sage: Word('0222220').length_maximal_palindrome(3, f='0->1,1->0,2->2')
5
```

```
>>> from sage.all import *
>>> Word('01001010').length_maximal_palindrome(Integer(3)/Integer(2))
0
>>> Word('01101001').length_maximal_palindrome(Integer(3)/Integer(2))
4
>>> Word('01010').length_maximal_palindrome(j=Integer(3), f='0->1,1->0')
0
>>> Word('01010').length_maximal_palindrome(j=RealNumber('2.5'), f='0->1,1->0
↪')
4
>>> Word('0222220').length_maximal_palindrome(Integer(3), f='0->1,1->0,2->2')
5
```

```
sage: w = Word('abcdcbaxyzzyx')
sage: w.length_maximal_palindrome(3)
7
sage: w.length_maximal_palindrome(3, 3)
7
sage: w.length_maximal_palindrome(3.5)
0
sage: w.length_maximal_palindrome(9.5)
6
sage: w.length_maximal_palindrome(9.5, 2)
6
```

```
>>> from sage.all import *
>>> w = Word('abcdcbaxyzzyx')
```

(continues on next page)

(continued from previous page)

```
>>> w.length_maximal_palindrome(Integer(3))
7
>>> w.length_maximal_palindrome(Integer(3), Integer(3))
7
>>> w.length_maximal_palindrome(RealNumber('3.5'))
0
>>> w.length_maximal_palindrome(RealNumber('9.5'))
6
>>> w.length_maximal_palindrome(RealNumber('9.5'), Integer(2))
6
```

lengths_maximal_palindromes(*f=None*)

Return the length of maximal palindromes centered at each position.

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`; it must be callable on letters as well as words (e.g. `WordMorphism`)

OUTPUT: list; the length of the maximal palindrome (or `f`-palindrome) with a given symmetry axis (letter or space between two letters)

EXAMPLES:

```
sage: Word('01101001').lengths_maximal_palindromes()
[0, 1, 0, 1, 4, 1, 0, 3, 0, 3, 0, 1, 4, 1, 0, 1, 0]
sage: Word('00000').lengths_maximal_palindromes()
[0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0]
sage: Word('0').lengths_maximal_palindromes()
[0, 1, 0]
sage: Word('').lengths_maximal_palindromes()
[0]
sage: Word().lengths_maximal_palindromes()
[0]
sage: f = WordMorphism('a->b,b->a')
sage: Word('abbabaab').lengths_maximal_palindromes(f)
[0, 0, 2, 0, 0, 0, 2, 0, 8, 0, 2, 0, 0, 0, 2, 0, 0]
```

```
>>> from sage.all import *
>>> Word('01101001').lengths_maximal_palindromes()
[0, 1, 0, 1, 4, 1, 0, 3, 0, 3, 0, 1, 4, 1, 0, 1, 0]
>>> Word('00000').lengths_maximal_palindromes()
[0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0]
>>> Word('0').lengths_maximal_palindromes()
[0, 1, 0]
>>> Word('').lengths_maximal_palindromes()
[0]
>>> Word().lengths_maximal_palindromes()
[0]
>>> f = WordMorphism('a->b,b->a')
>>> Word('abbabaab').lengths_maximal_palindromes(f)
[0, 0, 2, 0, 0, 0, 2, 0, 8, 0, 2, 0, 0, 0, 2, 0, 0]
```

lengths_unioccurrent_lps(*f=None*)

Return the list of the lengths of the unioccurrent longest (`f`)-palindromic suffixes (lps) for each non-empty prefix of `self`. No unioccurrent lps are indicated by `None`.

It corresponds to the function H_w defined in [BMBL2008] and [BMBFLR2008].

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`. It must be callable on letters as well as words (e.g. `WordMorphism`). The default value corresponds to usual palindromes, i.e., `f` equal to the identity.

OUTPUT:

a list – list of the length of the unioccurrent longest palindromic suffix (lps) for each non-empty prefix of `self`. No unioccurrent lps are indicated by `None`.

EXAMPLES:

```
sage: w = Word([0,1,1,2,3,4,5,1,13,3])
sage: w.lengths_unioccurrent_lps()
[1, 1, 2, 1, 1, 1, 1, None, 1, None]
sage: f = words.FibonacciWord()[:20]
sage: f.lengths_unioccurrent_lps() == f.lps_lengths()[1:]
True
sage: t = words.ThueMorseWord()
sage: t[:20].lengths_unioccurrent_lps()
[1, 1, 2, 4, 3, 3, 2, 4, None, None, 6, 8, 10, 12, 14, 16, 6, 8, 10, 12]
sage: f = WordMorphism({1:[0],0:[1]})
sage: t[:15].lengths_unioccurrent_lps(f)
[None, 2, None, 2, None, 4, 6, 8, 4, 6, 4, 6, None, 4, 6]
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(1),Integer(13),Integer(3)])
>>> w.lengths_unioccurrent_lps()
[1, 1, 2, 1, 1, 1, 1, None, 1, None]
>>> f = words.FibonacciWord()[:Integer(20)]
>>> f.lengths_unioccurrent_lps() == f.lps_lengths()[Integer(1):]
True
>>> t = words.ThueMorseWord()
>>> t[:Integer(20)].lengths_unioccurrent_lps()
[1, 1, 2, 4, 3, 3, 2, 4, None, None, 6, 8, 10, 12, 14, 16, 6, 8, 10, 12]
>>> f = WordMorphism({Integer(1):[Integer(0)],Integer(0):[Integer(1)]})
>>> t[:Integer(15)].lengths_unioccurrent_lps(f)
[None, 2, None, 2, None, 4, 6, 8, 4, 6, 4, 6, None, 4, 6]
```

letters()

Return the list of letters that appear in this word, listed in the order of first appearance.

EXAMPLES:

```
sage: Word([0,1,1,0,1,0,0,1]).letters()
[0, 1]
sage: Word("cacao").letters()
['c', 'a', 'o']
```

```
>>> from sage.all import *
>>> Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1)]).letters()
[0, 1]
>>> Word("cacao").letters()
['c', 'a', 'o']
```

longest_backward_extension(*x*, *y*)

Compute the length of the longest factor of `self` that ends at `x` and that matches a factor that ends at `y`.

INPUT:

- `x`, `y` – positions in `self`

EXAMPLES:

```
sage: w = Word('0011001')
sage: w.longest_backward_extension(6, 2)
3
sage: w.longest_backward_extension(1, 4)
1
sage: w.longest_backward_extension(1, 3)
0
```

```
>>> from sage.all import *
>>> w = Word('0011001')
>>> w.longest_backward_extension(Integer(6), Integer(2))
3
>>> w.longest_backward_extension(Integer(1), Integer(4))
1
>>> w.longest_backward_extension(Integer(1), Integer(3))
0
```

The method also accepts negative positions indicating the distance from the end of the word (in order to be consist with how negative indices work with lists). For instance, for a word of length 7, using positions 6 and -5 is the same as using positions 6 and 2:

```
sage: w.longest_backward_extension(6, -5)
3
sage: w.longest_backward_extension(-6, 4)
1
```

```
>>> from sage.all import *
>>> w.longest_backward_extension(Integer(6), -Integer(5))
3
>>> w.longest_backward_extension(-Integer(6), Integer(4))
1
```

longest_common_subword(*other*)

Return a longest subword of `self` and `other`.

A subword of a word is a subset of the word's letters, read in the order in which they appear in the word.

For more information, see Wikipedia article Longest_common_subsequence_problem.

INPUT:

- other – a word

ALGORITHM:

For any indices i, j, we compute the longest common subword lcs[i,j] of self[:i] and other[:j]. This can be easily obtained as the longest of

- lcs[i-1,j]
- lcs[i,j-1]
- lcs[i-1,j-1]+self[i] if self[i]==other[j]

EXAMPLES:

```
sage: v1 = Word("abc")
sage: v2 = Word("ace")
sage: v1.longest_common_subword(v2)
word: ac

sage: w1 = Word("1010101010101010101010101010101010101010")
sage: w2 = Word("0011001100110011001100110011001100110011")
sage: w1.longest_common_subword(w2)
word: 00110011001100110011010101010
```

```
>>> from sage.all import *
>>> v1 = Word("abc")
>>> v2 = Word("ace")
>>> v1.longest_common_subword(v2)
word: ac

>>> w1 = Word("1010101010101010101010101010101010101010")
>>> w2 = Word("0011001100110011001100110011001100110011")
>>> w1.longest_common_subword(w2)
word: 00110011001100110011010101010
```

See also

is_subword_of()

longest_common_suffix(*other*)

Return the longest common suffix of self and other.

EXAMPLES:

```
sage: w = Word('112345678')
sage: u = Word('1115678')
sage: w.longest_common_suffix(u)
word: 5678
sage: u.longest_common_suffix(u)
word: 1115678
sage: u.longest_common_suffix(w)
word: 5678
sage: w.longest_common_suffix(w)
word: 112345678
```

(continues on next page)

(continued from previous page)

```
sage: y = Word('549332345')
sage: w.longest_common_suffix(y)
word:
```

```
>>> from sage.all import *
>>> w = Word('112345678')
>>> u = Word('1115678')
>>> w.longest_common_suffix(u)
word: 5678
>>> u.longest_common_suffix(u)
word: 1115678
>>> u.longest_common_suffix(w)
word: 5678
>>> w.longest_common_suffix(w)
word: 112345678
>>> y = Word('549332345')
>>> w.longest_common_suffix(y)
word:
```

longest_forward_extension(*x*, *y*)

Compute the length of the longest factor of `self` that starts at `x` and that matches a factor that starts at `y`.

INPUT:

- `x`, `y` – positions in `self`

EXAMPLES:

```
sage: w = Word('0011001')
sage: w.longest_forward_extension(0, 4)
3
sage: w.longest_forward_extension(0, 2)
0
```

```
>>> from sage.all import *
>>> w = Word('0011001')
>>> w.longest_forward_extension(Integer(0), Integer(4))
3
>>> w.longest_forward_extension(Integer(0), Integer(2))
0
```

The method also accepts negative positions indicating the distance from the end of the word (in order to be consist with how negative indices work with lists). For instance, for a word of length 7, using positions -3 and 2 is the same as using positions 4 and 2:

```
sage: w.longest_forward_extension(1, -2)
2
sage: w.longest_forward_extension(4, -3)
3
```

```
>>> from sage.all import *
>>> w.longest_forward_extension(Integer(1), -Integer(2))
2
```

(continues on next page)

(continued from previous page)

```
>>> w.longest_forward_extension(Integer(4), -Integer(3))
3
```

lps(*f=None*, *l=None*)

Return the longest palindromic (or `f`-palindromic) suffix of `self`.

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`. It must be callable on letters as well as words (e.g. `WordMorphism`)
- `l` – integer (default: `None`); the length of the longest palindrome suffix of `self[:-1]`, if known

OUTPUT: word; if `f` is `None`, the longest palindromic suffix of `self`. Otherwise, the longest `f`-palindromic suffix of `self`.

EXAMPLES:

```
sage: Word('0111').lps()
word: 111
sage: Word('011101').lps()
word: 101
sage: Word('6667').lps()
word: 7
sage: Word('abbabaab').lps()
word: baab
sage: Word().lps()
word:
sage: f = WordMorphism('a->b,b->a')
sage: Word('abbabaab').lps(f=f)
word: abbabaab
sage: w = Word('33412321')
sage: w.lps(l=3)
word: 12321
sage: Y = Word
sage: w = Y('01101001')
sage: w.lps(l=2)
word: 1001
sage: w.lps()
word: 1001
sage: w.lps(l=None)
word: 1001
sage: Y().lps(l=2)
Traceback (most recent call last):
...
IndexError: list index out of range
sage: v = Word('abbabaab')
sage: pal = v[:0]
sage: for i in range(1, v.length()+1):
....:     pal = v[:i].lps(l=pal.length())
....:     pal
word: a
word: b
word: bb
```

(continues on next page)

(continued from previous page)

```
word: abba
word: bab
word: aba
word: aa
word: baab
sage: f = WordMorphism('a->b,b->a')
sage: v = Word('abbabaab')
sage: pal = v[:0]
sage: for i in range(1, v.length()+1):
....:   pal = v[:i].lps(f=f, l=pal.length())
....:   pal
word:
word: ab
word:
word: ba
word: ab
word: baba
word: bbabaa
word: abbabaab
```

```
>>> from sage.all import *
>>> Word('0111').lps()
word: 111
>>> Word('011101').lps()
word: 101
>>> Word('6667').lps()
word: 7
>>> Word('abbabaab').lps()
word: baab
>>> Word().lps()
word:
>>> f = WordMorphism('a->b,b->a')
>>> Word('abbabaab').lps(f=f)
word: abbabaab
>>> w = Word('33412321')
>>> w.lps(l=Integer(3))
word: 12321
>>> Y = Word
>>> w = Y('01101001')
>>> w.lps(l=Integer(2))
word: 1001
>>> w.lps()
word: 1001
>>> w.lps(l=None)
word: 1001
>>> Y().lps(l=Integer(2))
Traceback (most recent call last):
...
IndexError: list index out of range
>>> v = Word('abbabaab')
>>> pal = v[:Integer(0)]
>>> for i in range(Integer(1), v.length()+Integer(1)):
```

(continues on next page)

(continued from previous page)

```
...     pal = v[:i].lps(l=pal.length())
...     pal
word: a
word: b
word: bb
word: abba
word: bab
word: aba
word: aa
word: baab
>>> f = WordMorphism('a->b,b->a')
>>> v = Word('abbabaab')
>>> pal = v[:Integer(0)]
>>> for i in range(Integer(1), v.length()+Integer(1)):
...     pal = v[:i].lps(f=f, l=pal.length())
...     pal
word:
word: ab
word:
word: ba
word: ab
word: baba
word: bbabaa
word: abbabaab
```

lps_lengths(*f=None*)

Return the length of the longest palindromic suffix of each prefix.

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`. It must be callable on letters as well as words (e.g. `WordMorphism`).

OUTPUT: list; the length of the longest palindromic (or `f`-palindromic) suffix of each prefix of `self`

EXAMPLES:

```
sage: Word('01101001').lps_lengths()
[0, 1, 1, 2, 4, 3, 3, 2, 4]
sage: Word('00000').lps_lengths()
[0, 1, 2, 3, 4, 5]
sage: Word('0').lps_lengths()
[0, 1]
sage: Word('').lps_lengths()
[0]
sage: Word().lps_lengths()
[0]
sage: f = WordMorphism('a->b,b->a')
sage: Word('abbabaab').lps_lengths(f)
[0, 0, 2, 0, 2, 2, 4, 6, 8]
```

```
>>> from sage.all import *
>>> Word('01101001').lps_lengths()
[0, 1, 1, 2, 4, 3, 3, 2, 4]
```

(continues on next page)

(continued from previous page)

```
>>> Word('00000').lps_lengths()
[0, 1, 2, 3, 4, 5]
>>> Word('0').lps_lengths()
[0, 1]
>>> Word('').lps_lengths()
[0]
>>> Word().lps_lengths()
[0]
>>> f = WordMorphism('a->b,b->a')
>>> Word('abbabaab').lps_lengths(f)
[0, 0, 2, 0, 2, 2, 4, 6, 8]
```

lyndon_factorization()

Return the Lyndon factorization of `self`.

The *Lyndon factorization* of a finite word w is the unique factorization of w as a non-increasing product of Lyndon words, i.e., $w = l_1 \cdots l_n$ where each l_i is a Lyndon word and $l_1 \geq \cdots \geq l_n$. See for instance [Duv1983].

OUTPUT: the list $[l_1, \ldots, l_n]$ of factors obtained

EXAMPLES:

```
sage: Word('010010010001000').lyndon_factorization()
(01, 001, 001, 0001, 0, 0, 0)
sage: Words('10')('010010010001000').lyndon_factorization()
(0, 10010010001000)
sage: Word('abbababbaababba').lyndon_factorization()
(abb, ababb, aababb, a)
sage: Words('ba')('abbababbaababba').lyndon_factorization()
(a, bbababbaaba, bba)
sage: Word([1,2,1,3,1,2,1]).lyndon_factorization()
(1213, 12, 1)
```

```
>>> from sage.all import *
>>> Word('010010010001000').lyndon_factorization()
(01, 001, 001, 0001, 0, 0, 0)
>>> Words('10')('010010010001000').lyndon_factorization()
(0, 10010010001000)
>>> Word('abbababbaababba').lyndon_factorization()
(abb, ababb, aababb, a)
>>> Words('ba')('abbababbaababba').lyndon_factorization()
(a, bbababbaaba, bba)
>>> Word([Integer(1),Integer(2),Integer(1),Integer(3),Integer(1),Integer(2),
↪Integer(1)]).lyndon_factorization()
(1213, 12, 1)
```

major_index(*final_descent=False*)

Return the major index of `self`.

The major index of a word w is the sum of the descents of w.

With the `final_descent` option, the last position of a non-empty word is also considered as a descent.

See also

major index on Permutations.

EXAMPLES:

```
sage: w = Word([2,1,3,3,2])
sage: w.major_index()
5
sage: w = Word([2,1,3,3,2])
sage: w.major_index(final_descent=True)
10
```

```
>>> from sage.all import *
>>> w = Word([Integer(2),Integer(1),Integer(3),Integer(3),Integer(2)])
>>> w.major_index()
5
>>> w = Word([Integer(2),Integer(1),Integer(3),Integer(3),Integer(2)])
>>> w.major_index(final_descent=True)
10
```

minimal_conjugate()

Return the lexicographically minimal conjugate of this word (see Wikipedia article Lexicographically_minimal_string_rotation).

EXAMPLES:

```
sage: Word('213').minimal_conjugate()
word: 132
sage: Word('11').minimal_conjugate()
word: 11
sage: Word('12112').minimal_conjugate()
word: 11212
sage: Word('211').minimal_conjugate()
word: 112
sage: Word('211211211').minimal_conjugate()
word: 112112112
```

```
>>> from sage.all import *
>>> Word('213').minimal_conjugate()
word: 132
>>> Word('11').minimal_conjugate()
word: 11
>>> Word('12112').minimal_conjugate()
word: 11212
>>> Word('211').minimal_conjugate()
word: 112
>>> Word('211211211').minimal_conjugate()
word: 112112112
```

minimal_period()

Return the period of `self`.

Let A be an alphabet. An integer $p \geq 1$ is a *period* of a word $w = a_1 a_2 \cdots a_n$ where $a_i \in A$ if $a_i = a_{i+p}$ for $i = 1, \ldots, n - p$. The smallest period of w is called *the* period of w. See Chapter 1 of [Lot2002].

EXAMPLES:

```
sage: Word('aba').minimal_period()
2
sage: Word('abab').minimal_period()
2
sage: Word('ababa').minimal_period()
2
sage: Word('ababaa').minimal_period()
5
sage: Word('ababac').minimal_period()
6
sage: Word('aaaaaa').minimal_period()
1
sage: Word('a').minimal_period()
1
sage: Word().minimal_period()
1
```

```
>>> from sage.all import *
>>> Word('aba').minimal_period()
2
>>> Word('abab').minimal_period()
2
>>> Word('ababa').minimal_period()
2
>>> Word('ababaa').minimal_period()
5
>>> Word('ababac').minimal_period()
6
>>> Word('aaaaaa').minimal_period()
1
>>> Word('a').minimal_period()
1
>>> Word().minimal_period()
1
```

nb_factor_occurrences_in(*other*)

Return the number of times `self` appears as a factor in `other`.

> **Warning**
>
> This method is deprecated since 2020 and will be removed in a later version of SageMath. Use *number_of_factor_occurrences()* instead.

EXAMPLES:

```
sage: Word('123').nb_factor_occurrences_in(Word('112332312313112332121123'))
doctest:warning
...
```

(continues on next page)

(continued from previous page)

```
DeprecationWarning: f.nb_factor_occurrences_in(w) is deprecated.
Use w.number_of_factor_occurrences(f) instead.
See https://github.com/sagemath/sage/issues/30187 for details.
4
sage: Word('321').nb_factor_occurrences_in(Word('11233231231311233221123'))
0
```

```
>>> from sage.all import *
>>> Word('123').nb_factor_occurrences_in(Word('112332312313112332121123'))
doctest:warning
...
DeprecationWarning: f.nb_factor_occurrences_in(w) is deprecated.
Use w.number_of_factor_occurrences(f) instead.
See https://github.com/sagemath/sage/issues/30187 for details.
4
>>> Word('321').nb_factor_occurrences_in(Word('11233231231311233221123'))
0
```

An error is raised for the empty word:

```
sage: Word().nb_factor_occurrences_in(Word('123'))
Traceback (most recent call last):
...
NotImplementedError: The factor must be non empty
```

```
>>> from sage.all import *
>>> Word().nb_factor_occurrences_in(Word('123'))
Traceback (most recent call last):
...
NotImplementedError: The factor must be non empty
```

nb_subword_occurrences_in(*other*)

Return the number of times `self` appears in `other` as a subword.

This corresponds to the notion of $binomial coefficient$ of two finite words whose properties are presented in the chapter of Lothaire's book written by Sakarovitch and Simon [Lot1997].

> **Warning**
>
> This method is deprecated since 2020 and will be removed in a later version of SageMath. Use *number_of_subword_occurrences()* instead.

INPUT:

- `other` – finite word

EXAMPLES:

```
sage: tm = words.ThueMorseWord()

sage: u = Word([0,1,0,1])
sage: u.nb_subword_occurrences_in(tm[:1000])
```

(continues on next page)

(continued from previous page)

```
doctest:warning
...
DeprecationWarning: f.nb_subword_occurrences_in(w) is deprecated.
Use w.number_of_subword_occurrences(f) instead.
See https://github.com/sagemath/sage/issues/30187 for details.
2604124996

sage: u = Word([0,1,0,1,1,0])
sage: u.nb_subword_occurrences_in(tm[:100])
20370432
```

```
>>> from sage.all import *
>>> tm = words.ThueMorseWord()

>>> u = Word([Integer(0),Integer(1),Integer(0),Integer(1)])
>>> u.nb_subword_occurrences_in(tm[:Integer(1000)])
doctest:warning
...
DeprecationWarning: f.nb_subword_occurrences_in(w) is deprecated.
Use w.number_of_subword_occurrences(f) instead.
See https://github.com/sagemath/sage/issues/30187 for details.
2604124996

>>> u = Word([Integer(0),Integer(1),Integer(0),Integer(1),Integer(1),
↪Integer(0)])
>>> u.nb_subword_occurrences_in(tm[:Integer(100)])
20370432
```

Note

This code, based on [MSSY2001], actually compute the number of occurrences of all prefixes of `self` as subwords in all prefixes of `other`. In particular, its complexity is bounded by `len(self) * len(other)`.

number_of_factor_occurrences(*other*)

Return the number of times `other` appears as a factor in `self`.

INPUT:

- `other` – a non empty word

EXAMPLES:

```
sage: w = Word('112332312313112332121123')
sage: w.number_of_factor_occurrences(Word('123'))
4
sage: w = Word('112332312313112332121123')
sage: w.number_of_factor_occurrences(Word('321'))
0
```

```
>>> from sage.all import *
>>> w = Word('112332312313112332121123')
```

(continues on next page)

(continued from previous page)

```
>>> w.number_of_factor_occurrences(Word('123'))
4
>>> w = Word('112332312313112332121123')
>>> w.number_of_factor_occurrences(Word('321'))
0
```

```
sage: Word().number_of_factor_occurrences(Word('123'))
0
```

```
>>> from sage.all import *
>>> Word().number_of_factor_occurrences(Word('123'))
0
```

An error is raised for the empty word:

```
sage: Word('123').number_of_factor_occurrences(Word())
Traceback (most recent call last):
...
NotImplementedError: The factor must be non empty
```

```
>>> from sage.all import *
>>> Word('123').number_of_factor_occurrences(Word())
Traceback (most recent call last):
...
NotImplementedError: The factor must be non empty
```

number_of_factors(*n=None*, *algorithm='suffix tree'*)

Count the number of distinct factors of `self`.

INPUT:

- `n` – integer or `None`
- `algorithm` – string (default: `'suffix tree'`); takes the following values:
  - `'suffix tree'` – construct and use the suffix tree of the word
  - `'naive'` – algorithm uses a sliding window

OUTPUT:

If `n` is an integer, returns the number of distinct factors of length `n`. If `n` is `None`, returns the total number of distinct factors.

EXAMPLES:

```
sage: w = Word([1,2,1,2,3])
sage: w.number_of_factors()
13
sage: [w.number_of_factors(i) for i in range(6)]
[1, 3, 3, 3, 2, 1]
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(2),Integer(1),Integer(2),Integer(3)])
>>> w.number_of_factors()
```

(continues on next page)

(continued from previous page)

```
13
>>> [w.number_of_factors(i) for i in range(Integer(6))]
[1, 3, 3, 3, 2, 1]
```

```
sage: w = words.ThueMorseWord()[:100]
sage: [w.number_of_factors(i) for i in range(10)]
[1, 2, 4, 6, 10, 12, 16, 20, 22, 24]
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(100)]
>>> [w.number_of_factors(i) for i in range(Integer(10))]
[1, 2, 4, 6, 10, 12, 16, 20, 22, 24]
```

```
sage: Word('1213121').number_of_factors()
22
sage: Word('1213121').number_of_factors(1)
3
```

```
>>> from sage.all import *
>>> Word('1213121').number_of_factors()
22
>>> Word('1213121').number_of_factors(Integer(1))
3
```

```
sage: Word('a'*100).number_of_factors()
101
sage: Word('a'*100).number_of_factors(77)
1
```

```
>>> from sage.all import *
>>> Word('a'*Integer(100)).number_of_factors()
101
>>> Word('a'*Integer(100)).number_of_factors(Integer(77))
1
```

```
sage: Word().number_of_factors()
1
sage: Word().number_of_factors(17)
0
```

```
>>> from sage.all import *
>>> Word().number_of_factors()
1
>>> Word().number_of_factors(Integer(17))
0
```

```
sage: blueberry = Word("blueberry")
sage: blueberry.number_of_factors()
43
```

(continues on next page)

(continued from previous page)

```
sage: [blueberry.number_of_factors(i) for i in range(10)]
[1, 6, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> blueberry = Word("blueberry")
>>> blueberry.number_of_factors()
43
>>> [blueberry.number_of_factors(i) for i in range(Integer(10))]
[1, 6, 8, 7, 6, 5, 4, 3, 2, 1]
```

number_of_inversions()

Return the number of inversions in `self`.

An inversion of a word $w = w_1 \ldots w_n$ is a pair of indices (i, j) with $i < j$ and $w_i > w_j$.

> **See also**
>
> *number of inversions on Permutations*.

EXAMPLES:

```
sage: w = Word([2,1,3,3,2])
sage: w.number_of_inversions()
3
```

```
>>> from sage.all import *
>>> w = Word([Integer(2),Integer(1),Integer(3),Integer(3),Integer(2)])
>>> w.number_of_inversions()
3
```

number_of_left_special_factors(n)

Return the number of left special factors of length n.

A factor u of a word w is *left special* if there are two distinct letters a and b such that au and bu are factors of w.

INPUT:

- `n` – integer

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: w = words.FibonacciWord()[:100]
sage: [w.number_of_left_special_factors(i) for i in range(10)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> w = words.FibonacciWord()[:Integer(100)]
>>> [w.number_of_left_special_factors(i) for i in range(Integer(10))]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
sage: w = words.ThueMorseWord()[:100]
sage: [w.number_of_left_special_factors(i) for i in range(10)]
[1, 2, 2, 4, 2, 4, 4, 2, 2, 4]
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(100)]
>>> [w.number_of_left_special_factors(i) for i in range(Integer(10))]
[1, 2, 2, 4, 2, 4, 4, 2, 2, 4]
```

number_of_letter_occurrences(*letter*)

Return the number of occurrences of `letter` in `self`.

INPUT:

- `letter` – a letter

OUTPUT: integer

EXAMPLES:

```
sage: w = Word('abbabaab')
sage: w.number_of_letter_occurrences('a')
4
sage: w.number_of_letter_occurrences('ab')
0
```

```
>>> from sage.all import *
>>> w = Word('abbabaab')
>>> w.number_of_letter_occurrences('a')
4
>>> w.number_of_letter_occurrences('ab')
0
```

This methods is equivalent to `list(w).count(letter)` and `tuple(w).count(letter)`, thus `count` is an alias for the method `number_of_letter_occurrences`:

```
sage: list(w).count('a')
4
sage: w.count('a')
4
```

```
>>> from sage.all import *
>>> list(w).count('a')
4
>>> w.count('a')
4
```

But notice that if `s` and `w` are strings, `Word(s).count(w)` counts the number occurrences of `w` as a letter in `Word(s)` which is not the same as `s.count(w)` which counts the number of occurrences of the string `w` inside `s`:

```
sage: s = 'abbabaab'
sage: s.count('ab')
3
```

(continues on next page)

(continued from previous page)

```
sage: Word(s).count('ab')
0
```

```
>>> from sage.all import *
>>> s = 'abbabaab'
>>> s.count('ab')
3
>>> Word(s).count('ab')
0
```

See also

sage.combinat.words.finite_word.FiniteWord_class.number_of_factor_occurrences()

number_of_right_special_factors(*n*)

Return the number of right special factors of length `n`.

A factor u of a word w is *right special* if there are two distinct letters a and b such that ua and ub are factors of w.

INPUT:

- `n` – integer

OUTPUT: nonnegative integer

EXAMPLES:

```
sage: w = words.FibonacciWord()[:100]
sage: [w.number_of_right_special_factors(i) for i in range(10)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> w = words.FibonacciWord()[:Integer(100)]
>>> [w.number_of_right_special_factors(i) for i in range(Integer(10))]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
sage: w = words.ThueMorseWord()[:100]
sage: [w.number_of_right_special_factors(i) for i in range(10)]
[1, 2, 2, 4, 2, 4, 4, 2, 2, 4]
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(100)]
>>> [w.number_of_right_special_factors(i) for i in range(Integer(10))]
[1, 2, 2, 4, 2, 4, 4, 2, 2, 4]
```

number_of_subword_occurrences(*other*)

Return the number of times `other` appears in `self` as a subword.

This corresponds to the notion of $binomial coefficient$ of two finite words whose properties are presented in the chapter of Lothaire's book written by Sakarovitch and Simon [Lot1997].

INPUT:

- `other` – finite word

EXAMPLES:

```
sage: tm = words.ThueMorseWord()
sage: u = Word([0,1,0,1])
sage: tm[:1000].number_of_subword_occurrences(u)
2604124996

sage: u = Word([0,1,0,1,1,0])
sage: tm[:100].number_of_subword_occurrences(u)
20370432
```

```
>>> from sage.all import *
>>> tm = words.ThueMorseWord()
>>> u = Word([Integer(0),Integer(1),Integer(0),Integer(1)])
>>> tm[:Integer(1000)].number_of_subword_occurrences(u)
2604124996

>>> u = Word([Integer(0),Integer(1),Integer(0),Integer(1),Integer(1),
↪Integer(0)])
>>> tm[:Integer(100)].number_of_subword_occurrences(u)
20370432
```

Note

This code, based on [MSSY2001], actually compute the number of occurrences of all prefixes of `self` as subwords in all prefixes of `other`. In particular, its complexity is bounded by `len(self) * len(other)`.

order()

Return the order of `self`.

Let $p(w)$ be the period of a word w. The positive rational number $|w|/p(w)$ is the *order* of w. See Chapter 8 of [Lot2002].

OUTPUT: rational; the order

EXAMPLES:

```
sage: Word('abaaba').order()
2
sage: Word('ababaaba').order()
8/5
sage: Word('a').order()
1
sage: Word('aa').order()
2
sage: Word().order()
0
```

```
>>> from sage.all import *
>>> Word('abaaba').order()
```

(continues on next page)

(continued from previous page)

```
2
>>> Word('abababaaba').order()
8/5
>>> Word('a').order()
1
>>> Word('aa').order()
2
>>> Word().order()
0
```

overlap_partition(*other*, *delay=0*, *p=None*, *involution=None*)

Return the partition of the alphabet induced by the overlap of `self` and `other` with the given `delay`.

The partition of the alphabet is given by the equivalence relation obtained from the symmetric, reflexive and transitive closure of the set of pairs of letters $R_{u,v,d} = \{(u_k, v_{k-d}) : 0 \leq k < n, 0 \leq k - d < m\}$ where $u = u_0 u_1 \cdots u_{n-1}$, $v = v_0 v_1 \cdots v_{m-1}$ are two words on the alphabet A and d is an integer.

The equivalence relation defined by R is inspired from [Lab2008].

INPUT:

- `other` – word on the same alphabet as `self`
- `delay` – integer (default: 0)
- `p` – disjoint sets data structure (default: `None`), a partition of the alphabet into disjoint sets to start with. If `None`, each letter start in distinct equivalence classes.
- `involution` – callable (default: `None`); an involution on the alphabet. If `involution` is not `None`, the relation $R_{u,v,d} \cup R_{involution(u),involution(v),d}$ is considered.

OUTPUT: a disjoint set data structure

EXAMPLES:

```
sage: W = Words(list('abc012345'))
sage: u = W('abc')
sage: v = W('01234')
sage: u.overlap_partition(v)
{{'0', 'a'}, {'1', 'b'}, {'2', 'c'}, {'3'}, {'4'}, {'5'}}
sage: u.overlap_partition(v, 2)
{{'0', 'c'}, {'1'}, {'2'}, {'3'}, {'4'}, {'5'}, {'a'}, {'b'}}
sage: u.overlap_partition(v, -1)
{{'0'}, {'1', 'a'}, {'2', 'b'}, {'3', 'c'}, {'4'}, {'5'}}
```

```
>>> from sage.all import *
>>> W = Words(list('abc012345'))
>>> u = W('abc')
>>> v = W('01234')
>>> u.overlap_partition(v)
{{'0', 'a'}, {'1', 'b'}, {'2', 'c'}, {'3'}, {'4'}, {'5'}}
>>> u.overlap_partition(v, Integer(2))
{{'0', 'c'}, {'1'}, {'2'}, {'3'}, {'4'}, {'5'}, {'a'}, {'b'}}
>>> u.overlap_partition(v, -Integer(1))
{{'0'}, {'1', 'a'}, {'2', 'b'}, {'3', 'c'}, {'4'}, {'5'}}
```

You can re-use the same disjoint set and do more than one overlap:

```
sage: p = u.overlap_partition(v, 2)
sage: p
{{'0', 'c'}, {'1'}, {'2'}, {'3'}, {'4'}, {'5'}, {'a'}, {'b'}}
sage: u.overlap_partition(v, 1, p)
{{'0', '1', 'b', 'c'}, {'2'}, {'3'}, {'4'}, {'5'}, {'a'}}
```

```
>>> from sage.all import *
>>> p = u.overlap_partition(v, Integer(2))
>>> p
{{'0', 'c'}, {'1'}, {'2'}, {'3'}, {'4'}, {'5'}, {'a'}, {'b'}}
>>> u.overlap_partition(v, Integer(1), p)
{{'0', '1', 'b', 'c'}, {'2'}, {'3'}, {'4'}, {'5'}, {'a'}}
```

The function `overlap_partition` can be used to study equations on words. For example, if a word w overlaps itself with delay d, then d is a period of w:

```
sage: W = Words(range(20))
sage: w = W(range(14)); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13
sage: d = 5
sage: p = w.overlap_partition(w, d)
sage: m = WordMorphism(p.element_to_root_dict())
sage: w2 = m(w); w2
word: 56789567895678
sage: w2.minimal_period() == d
True
```

```
>>> from sage.all import *
>>> W = Words(range(Integer(20)))
>>> w = W(range(Integer(14))); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13
>>> d = Integer(5)
>>> p = w.overlap_partition(w, d)
>>> m = WordMorphism(p.element_to_root_dict())
>>> w2 = m(w); w2
word: 56789567895678
>>> w2.minimal_period() == d
True
```

If a word is equal to its reversal, then it is a palindrome:

```
sage: W = Words(range(20))
sage: w = W(range(17)); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
sage: p = w.overlap_partition(w.reversal(), 0)
sage: m = WordMorphism(p.element_to_root_dict())
sage: w2 = m(w); w2
word: 01234567876543210
sage: w2.parent()
Finite words over {0, 1, 2, 3, 4, 5, 6, 7, 8, 17, 18, 19}
sage: w2.is_palindrome()
True
```

```
>>> from sage.all import *
>>> W = Words(range(Integer(20)))
>>> w = W(range(Integer(17))); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
>>> p = w.overlap_partition(w.reversal(), Integer(0))
>>> m = WordMorphism(p.element_to_root_dict())
>>> w2 = m(w); w2
word: 01234567876543210
>>> w2.parent()
Finite words over {0, 1, 2, 3, 4, 5, 6, 7, 8, 17, 18, 19}
>>> w2.is_palindrome()
True
```

If the reversal of a word w is factor of its square w^2, then w is symmetric, i.e. the product of two palindromes:

```
sage: W = Words(range(10))
sage: w = W(range(10)); w
word: 0123456789
sage: p = (w*w).overlap_partition(w.reversal(), 4)
sage: m = WordMorphism(p.element_to_root_dict())
sage: w2 = m(w); w2
word: 0110456654
sage: w2.is_symmetric()
True
```

```
>>> from sage.all import *
>>> W = Words(range(Integer(10)))
>>> w = W(range(Integer(10))); w
word: 0123456789
>>> p = (w*w).overlap_partition(w.reversal(), Integer(4))
>>> m = WordMorphism(p.element_to_root_dict())
>>> w2 = m(w); w2
word: 0110456654
>>> w2.is_symmetric()
True
```

If the image of the reversal of a word w under an involution f is factor of its square w^2, then w is f-symmetric:

```
sage: W = Words([-11,-9,..,11])
sage: w = W([1,3,..,11])
sage: w
word: 1,3,5,7,9,11
sage: inv = lambda x:-x
sage: f = WordMorphism(dict( (a, inv(a)) for a in W.alphabet()))
sage: p = (w*w).overlap_partition(f(w).reversal(), 2, involution=f)
sage: m = WordMorphism(p.element_to_root_dict())
sage: m(w)
word: 1,-1,5,7,-7,-5
sage: m(w).is_symmetric(f)
True
```

```
>>> from sage.all import *
>>> W = Words((ellipsis_range(-Integer(11),-Integer(9),Ellipsis,Integer(11))))
```

(continues on next page)

(continued from previous page)

```
>>> w = W((ellipsis_range(Integer(1),Integer(3),Ellipsis,Integer(11))))
>>> w
word: 1,3,5,7,9,11
>>> inv = lambda x:-x
>>> f = WordMorphism(dict( (a, inv(a)) for a in W.alphabet()))
>>> p = (w*w).overlap_partition(f(w).reversal(), Integer(2), involution=f)
>>> m = WordMorphism(p.element_to_root_dict())
>>> m(w)
word: 1,-1,5,7,-7,-5
>>> m(w).is_symmetric(f)
True
```

palindrome_prefixes()

Return a list of all palindrome prefixes of `self`.

EXAMPLES:

```
sage: w = Word('abaaba')
sage: w.palindrome_prefixes()
[word: , word: a, word: aba, word: abaaba]
sage: w = Word('abbbbbbbbbb')
sage: w.palindrome_prefixes()
[word: , word: a]
```

```
>>> from sage.all import *
>>> w = Word('abaaba')
>>> w.palindrome_prefixes()
[word: , word: a, word: aba, word: abaaba]
>>> w = Word('abbbbbbbbbb')
>>> w.palindrome_prefixes()
[word: , word: a]
```

palindromes(*f=None*)

Return the set of all palindromic (or `f`-palindromic) factors of `self`.

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`; it must be callable on letters as well as words (e.g. `WordMorphism`).

OUTPUT: a set – If `f` is `None`, the set of all palindromic factors of `self`; otherwise, the set of all `f`-palindromic factors of `self`

EXAMPLES:

```
sage: sorted(Word('01101001').palindromes())
[word: , word: 0, word: 00, word: 010, word: 0110, word: 1, word: 1001, word:␣
↪101, word: 11]
sage: sorted(Word('00000').palindromes())
[word: , word: 0, word: 00, word: 000, word: 0000, word: 00000]
sage: sorted(Word('0').palindromes())
[word: , word: 0]
sage: sorted(Word('').palindromes())
[word: ]
```

(continues on next page)

(continued from previous page)

```
sage: sorted(Word().palindromes())
[word: ]
sage: f = WordMorphism('a->b,b->a')
sage: sorted(Word('abbabaab').palindromes(f))
[word: , word: ab, word: abbabaab, word: ba, word: baba, word: bbabaa]
```

```
>>> from sage.all import *
>>> sorted(Word('01101001').palindromes())
[word: , word: 0, word: 00, word: 010, word: 0110, word: 1, word: 1001, word:␣
↪101, word: 11]
>>> sorted(Word('00000').palindromes())
[word: , word: 0, word: 00, word: 000, word: 0000, word: 00000]
>>> sorted(Word('0').palindromes())
[word: , word: 0]
>>> sorted(Word('').palindromes())
[word: ]
>>> sorted(Word().palindromes())
[word: ]
>>> f = WordMorphism('a->b,b->a')
>>> sorted(Word('abbabaab').palindromes(f))
[word: , word: ab, word: abbabaab, word: ba, word: baba, word: bbabaa]
```

palindromic_closure(*side='right'*, *f=None*)

Return the shortest palindrome having `self` as a prefix (or as a suffix if `side` is `'left'`).

See [DeLuca2006].

INPUT:

- `side` – `'right'` or `'left'` (default: `'right'`) the direction of the closure
- `f` – involution (default: `None`) on the alphabet of `self`; it must be callable on letters as well as words (e.g. `WordMorphism`)

OUTPUT:

a word – If `f` is `None`, the right palindromic closure of `self`; otherwise, the right `f`-palindromic closure of `self`. If `side` is `'left'`, the left palindromic closure.

EXAMPLES:

```
sage: Word('1233').palindromic_closure()
word: 123321
sage: Word('12332').palindromic_closure()
word: 123321
sage: Word('0110343').palindromic_closure()
word: 01103430110
sage: Word('0110343').palindromic_closure(side='left')
word: 3430110343
sage: Word('01105678').palindromic_closure(side='left')
word: 876501105678
sage: w = Word('abbaba')
sage: w.palindromic_closure()
word: abbababba
```

```
>>> from sage.all import *
>>> Word('1233').palindromic_closure()
word: 123321
>>> Word('12332').palindromic_closure()
word: 123321
>>> Word('0110343').palindromic_closure()
word: 01103430110
>>> Word('0110343').palindromic_closure(side='left')
word: 3430110343
>>> Word('01105678').palindromic_closure(side='left')
word: 876501105678
>>> w = Word('abbaba')
>>> w.palindromic_closure()
word: abbababba
```

```
sage: f = WordMorphism('a->b,b->a')
sage: w.palindromic_closure(f=f)
word: abbabaab
sage: w.palindromic_closure(f=f, side='left')
word: babaabbaba
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> w.palindromic_closure(f=f)
word: abbabaab
>>> w.palindromic_closure(f=f, side='left')
word: babaabbaba
```

palindromic_complexity(*n*)

Return the number of distinct palindromic factors of length `n` of `self`.

INPUT:

- `n` – the length of the factors

EXAMPLES:

```
sage: w = words.FibonacciWord()[:100]
sage: [w.palindromic_complexity(i) for i in range(20)]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

```
>>> from sage.all import *
>>> w = words.FibonacciWord()[:Integer(100)]
>>> [w.palindromic_complexity(i) for i in range(Integer(20))]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

```
sage: w = words.ThueMorseWord()[:1000]
sage: [w.palindromic_complexity(i) for i in range(20)]
[1, 2, 2, 2, 2, 0, 4, 0, 4, 0, 4, 0, 4, 0, 2, 0, 2, 0, 4, 0]
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(1000)]
```

(continues on next page)

(continued from previous page)

```
>>> [w.palindromic_complexity(i) for i in range(Integer(20))]
[1, 2, 2, 2, 2, 0, 4, 0, 4, 0, 4, 0, 4, 0, 2, 0, 2, 0, 4, 0]
```

palindromic_lacunas_study(*f=None*)

Return interesting statistics about longest (`f`-)palindromic suffixes and lacunas of `self` (see [BMBL2008] and [BMBFLR2008]).

Note that a word w has at most $|w|+1$ different palindromic factors (see [DJP2001]). For f-palindromes (or pseudopalindromes or theta-palindromes), the maximum number of f-palindromic factors is $|w|+1-g_f(w)$, where $g_f(w)$ is the number of pairs $\{a, f(a)\}$ such that a is a letter, a is not equal to $f(a)$, and a or $f(a)$ occurs in w, see [Star2011].

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`. It must be callable on letters as well as words (e.g. `WordMorphism`). The default value corresponds to usual palindromes, i.e., `f` equal to the identity.

OUTPUT:

- `list` – list of the length of the longest palindromic suffix (lps) for each non-empty prefix of `self`
- `list` – list of all the lacunas, i.e. positions where there is no unioccurrent lps
- `set` – set of palindromic factors of `self`

EXAMPLES:

```
sage: a,b,c = Word('abbabaabbaab').palindromic_lacunas_study()
sage: a
[1, 1, 2, 4, 3, 3, 2, 4, 2, 4, 6, 8]
sage: b
[8, 9]
sage: c          # random order
set([word: , word: b, word: bab, word: abba, word: bb, word: aa, word:␣
↪baabbaab, word: baab, word: aba, word: aabbaa, word: a])
```

```
>>> from sage.all import *
>>> a,b,c = Word('abbabaabbaab').palindromic_lacunas_study()
>>> a
[1, 1, 2, 4, 3, 3, 2, 4, 2, 4, 6, 8]
>>> b
[8, 9]
>>> c          # random order
set([word: , word: b, word: bab, word: abba, word: bb, word: aa, word:␣
↪baabbaab, word: baab, word: aba, word: aabbaa, word: a])
```

```
sage: f = WordMorphism('a->b,b->a')
sage: a,b,c = Word('abbabaab').palindromic_lacunas_study(f=f)
sage: a
[0, 2, 0, 2, 2, 4, 6, 8]
sage: b
[0, 2, 4]
sage: c          # random order
set([word: , word: ba, word: baba, word: ab, word: bbabaa, word: abbabaab])
sage: c == set([Word(), Word('ba'), Word('baba'), Word('ab'), Word('bbabaa'),␣
```

(continues on next page)

(continued from previous page)

```
↪Word('abbabaab')])
True
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->b,b->a')
>>> a,b,c = Word('abbabaab').palindromic_lacunas_study(f=f)
>>> a
[0, 2, 0, 2, 2, 4, 6, 8]
>>> b
[0, 2, 4]
>>> c            # random order
set([word: , word: ba, word: baba, word: ab, word: bbabaa, word: abbabaab])
>>> c == set([Word(), Word('ba'), Word('baba'), Word('ab'), Word('bbabaa'),␣
↪Word('abbabaab')])
True
```

periods(*divide_length=False*)

Return a list containing the periods of `self` between 1 and $n - 1$, where n is the length of `self`.

INPUT:

- `divide_length` – boolean (default: `False`); when set to `True`, then only periods that divide the length of `self` are considered

OUTPUT: list of positive integers

EXAMPLES:

```
sage: w = Word('ababab')
sage: w.periods()
[2, 4]
sage: w.periods(divide_length=True)
[2]
sage: w = Word('ababa')
sage: w.periods()
[2, 4]
sage: w.periods(divide_length=True)
[]
```

```
>>> from sage.all import *
>>> w = Word('ababab')
>>> w.periods()
[2, 4]
>>> w.periods(divide_length=True)
[2]
>>> w = Word('ababa')
>>> w.periods()
[2, 4]
>>> w.periods(divide_length=True)
[]
```

phi()

Apply the phi function to `self` and return the result. This is the word obtained by taking the first letter of the words obtained by iterating delta on `self`.

OUTPUT: a word – the result of the phi function

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2,2,1,1,2,1,2,2,1,2,2,1,1,2]).phi()
word: 222222
sage: W([2,1,2,2,1,2,2,1,2,1]).phi()
word: 212113
sage: W().phi()
word:
sage: Word([2,1,2,2,1,2,2,1,2,1]).phi()
word: 212113
sage: Word([2,3,1,1,2,1,2,3,1,2,2,3,1,2]).phi()
word: 21215
sage: Word("aabbabaabaabba").phi()
word: a22222
sage: w = Word([2,3,1,1,2,1,2,3,1,2,2,3,1,2])
```

```
>>> from sage.all import *
>>> W = Words([Integer(1), Integer(2)])
>>> W([Integer(2),Integer(2),Integer(1),Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(2),Integer(1),Integer(2),Integer(2),Integer(1),
↪Integer(1),Integer(2)]).phi()
word: 222222
>>> W([Integer(2),Integer(1),Integer(2),Integer(2),Integer(1),Integer(2),
↪Integer(2),Integer(1),Integer(2),Integer(1)]).phi()
word: 212113
>>> W().phi()
word:
>>> Word([Integer(2),Integer(1),Integer(2),Integer(2),Integer(1),Integer(2),
↪Integer(2),Integer(1),Integer(2),Integer(1)]).phi()
word: 212113
>>> Word([Integer(2),Integer(3),Integer(1),Integer(1),Integer(2),Integer(1),
↪Integer(2),Integer(3),Integer(1),Integer(2),Integer(2),Integer(3),
↪Integer(1),Integer(2)]).phi()
word: 21215
>>> Word("aabbabaabaabba").phi()
word: a22222
>>> w = Word([Integer(2),Integer(3),Integer(1),Integer(1),Integer(2),
↪Integer(1),Integer(2),Integer(3),Integer(1),Integer(2),Integer(2),
↪Integer(3),Integer(1),Integer(2)])
```

See [BL2003] and [BDLV2006].

phi_inv(*W=None*)

Apply the inverse of the phi function to `self`.

INPUT:

- `self` – a word over the integers
- `W` – a parent object of words defined over integers

OUTPUT: a word – the inverse of the phi function

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2, 2, 2, 2, 1, 2]).phi_inv()
word: 22112122
sage: W([2, 2, 2]).phi_inv(Words([2, 3]))
word: 2233
```

```
>>> from sage.all import *
>>> W = Words([Integer(1), Integer(2)])
>>> W([Integer(2), Integer(2), Integer(2), Integer(2), Integer(1),␣
↪Integer(2)]).phi_inv()
word: 22112122
>>> W([Integer(2), Integer(2), Integer(2)]).phi_inv(Words([Integer(2),␣
↪Integer(3)]))
word: 2233
```

prefix_function_table()

Return a vector containing the length of the proper prefix-suffixes for all the non-empty prefixes of `self`.

EXAMPLES:

```
sage: Word('121321').prefix_function_table()
[0, 0, 1, 0, 0, 1]
sage: Word('1241245').prefix_function_table()
[0, 0, 0, 1, 2, 3, 0]
sage: Word().prefix_function_table()
[]
```

```
>>> from sage.all import *
>>> Word('121321').prefix_function_table()
[0, 0, 1, 0, 0, 1]
>>> Word('1241245').prefix_function_table()
[0, 0, 0, 1, 2, 3, 0]
>>> Word().prefix_function_table()
[]
```

primitive()

Return the primitive of `self`.

EXAMPLES:

```
sage: Word('12312').primitive()
word: 12312
sage: Word('121212').primitive()
word: 12
```

```
>>> from sage.all import *
>>> Word('12312').primitive()
word: 12312
>>> Word('121212').primitive()
word: 12
```

primitive_length()

Return the length of the primitive of `self`.

EXAMPLES:

```
sage: Word('1231').primitive_length()
4
sage: Word('121212').primitive_length()
2
```

```
>>> from sage.all import *
>>> Word('1231').primitive_length()
4
>>> Word('121212').primitive_length()
2
```

quasiperiods()

Return the quasiperiods of `self` as a list ordered from shortest to longest.

Let w be a finite or infinite word. A *quasiperiod* of w is a proper factor u of w such that the occurrences of u in w entirely cover w, i.e., every position of w falls within some occurrence of u in w. See for instance [AE1993], [Mar2004], and [GLR2008].

EXAMPLES:

```
sage: Word('abaababaabaababaaba').quasiperiods()
[word: aba, word: abaaba, word: abaababaaba]
sage: Word('abaaba').quasiperiods()
[word: aba]
sage: Word('abacaba').quasiperiods()
[]
```

```
>>> from sage.all import *
>>> Word('abaababaabaababaaba').quasiperiods()
[word: aba, word: abaaba, word: abaababaaba]
>>> Word('abaaba').quasiperiods()
[word: aba]
>>> Word('abacaba').quasiperiods()
[]
```

rauzy_graph(n)

Return the Rauzy graph of the factors of length `n` of `self`.

The vertices are the factors of length n and there is an edge from u to v if $ua = bv$ is a factor of length $n + 1$ for some letters a and b.

INPUT:

- `n` – integer

EXAMPLES:

```
sage: w = Word(range(10)); w
word: 0123456789
sage: g = w.rauzy_graph(3); g                                               #␣
↪needs sage.graphs
Looped digraph on 8 vertices
sage: WordOptions(identifier='')
sage: g.vertices(sort=True)                                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[012, 123, 234, 345, 456, 567, 678, 789]
sage: g.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(012, 123, 3),
 (123, 234, 4),
 (234, 345, 5),
 (345, 456, 6),
 (456, 567, 7),
 (567, 678, 8),
 (678, 789, 9)]
sage: WordOptions(identifier='word: ')
```

```
>>> from sage.all import *
>>> w = Word(range(Integer(10))); w
word: 0123456789
>>> g = w.rauzy_graph(Integer(3)); g                                          ␣
↪     # needs sage.graphs
Looped digraph on 8 vertices
>>> WordOptions(identifier='')
>>> g.vertices(sort=True)                                                   #␣
↪needs sage.graphs
[012, 123, 234, 345, 456, 567, 678, 789]
>>> g.edges(sort=True)                                                      #␣
↪needs sage.graphs
[(012, 123, 3),
 (123, 234, 4),
 (234, 345, 5),
 (345, 456, 6),
 (456, 567, 7),
 (567, 678, 8),
 (678, 789, 9)]
>>> WordOptions(identifier='word: ')
```

```
sage: f = words.FibonacciWord()[:100]
sage: f.rauzy_graph(8)                                                      #␣
↪needs sage.graphs
Looped digraph on 9 vertices
```

```
>>> from sage.all import *
>>> f = words.FibonacciWord()[:Integer(100)]
>>> f.rauzy_graph(Integer(8))                                                 ␣
↪     # needs sage.graphs
Looped digraph on 9 vertices
```

```
sage: w = Word('1111111')
sage: g = w.rauzy_graph(3)                                                  #␣
↪needs sage.graphs
sage: g.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(word: 111, word: 111, word: 1)]
```

```
>>> from sage.all import *
>>> w = Word('1111111')
>>> g = w.rauzy_graph(Integer(3))                                       ␣
↪     # needs sage.graphs
>>> g.edges(sort=True)                                                #␣
↪needs sage.graphs
[(word: 111, word: 111, word: 1)]
```

```
sage: w = Word('111')
sage: for i in range(5): w.rauzy_graph(i)                              #␣
↪needs sage.graphs
Looped multi-digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 0 vertices
```

```
>>> from sage.all import *
>>> w = Word('111')
>>> for i in range(Integer(5)): w.rauzy_graph(i)                       ␣
↪     # needs sage.graphs
Looped multi-digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 0 vertices
```

Multi-edges are allowed for the empty word:

```
sage: W = Words('abcde')
sage: w = W('abc')
sage: w.rauzy_graph(0)                                                 #␣
↪needs sage.graphs
Looped multi-digraph on 1 vertex
sage: _.edges(sort=True)                                               #␣
↪needs sage.graphs
[(word: , word: , word: a),
 (word: , word: , word: b),
 (word: , word: , word: c)]
```

```
>>> from sage.all import *
>>> W = Words('abcde')
>>> w = W('abc')
>>> w.rauzy_graph(Integer(0))                                           ␣
↪     # needs sage.graphs
Looped multi-digraph on 1 vertex
>>> _.edges(sort=True)                                                #␣
↪needs sage.graphs
[(word: , word: , word: a),
 (word: , word: , word: b),
 (word: , word: , word: c)]
```

reduced_rauzy_graph(*n*)

Return the reduced Rauzy graph of order n of self.

INPUT:

- n – nonnegative integer; every vertex of a reduced Rauzy graph of order n is a factor of length n of self

OUTPUT: a looped multi-digraph

DEFINITION:

For infinite periodic words (resp. for finite words of type $u^i u[0:j]$), the reduced Rauzy graph of order n (resp. for n smaller or equal to $(i-1)|u|+j$) is the directed graph whose unique vertex is the prefix p of length n of self and which has an only edge which is a loop on p labelled by $w[n+1:|w|]p$ where w is the unique return word to p.

In other cases, it is the directed graph defined as followed. Let G_n be the Rauzy graph of order n of self. The vertices are the vertices of G_n that are either special or not prolongable to the right or to the left. For each couple (u, v) of such vertices and each directed path in G_n from u to v that contains no other vertices that are special, there is an edge from u to v in the reduced Rauzy graph of order n whose label is the label of the path in G_n.

Note: In the case of infinite recurrent non-periodic words, this definition corresponds to the following one that can be found in [BDLGZ2009] and [BPS2008] where a simple path is a path that begins with a special factor, ends with a special factor and contains no other vertices that are special:

The reduced Rauzy graph of factors of length n is obtained from G_n by replacing each simple path $P = v_1 v_2 ... v_\ell$ with an edge $v_1 v_\ell$ whose label is the concatenation of the labels of the edges of P.

EXAMPLES:

```
sage: w = Word(range(10)); w
word: 0123456789
sage: g = w.reduced_rauzy_graph(3); g                                       #␣
↪needs sage.graphs
Looped multi-digraph on 2 vertices
sage: g.vertices(sort=True)                                                 #␣
↪needs sage.graphs
[word: 012, word: 789]
sage: g.edges(sort=True)                                                    #␣
↪needs sage.graphs
[(word: 012, word: 789, word: 3456789)]
```

```
>>> from sage.all import *
>>> w = Word(range(Integer(10))); w
word: 0123456789
>>> g = w.reduced_rauzy_graph(Integer(3)); g                                  ␣
↪     # needs sage.graphs
Looped multi-digraph on 2 vertices
>>> g.vertices(sort=True)                                                   #␣
↪needs sage.graphs
[word: 012, word: 789]
>>> g.edges(sort=True)                                                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[(word: 012, word: 789, word: 3456789)]
```

For the Fibonacci word:

```
sage: f = words.FibonacciWord()[:100]
sage: g = f.reduced_rauzy_graph(8);g                                    #␣
↪needs sage.graphs
Looped multi-digraph on 2 vertices
sage: g.vertices(sort=True)                                             #␣
↪needs sage.graphs
[word: 01001010, word: 01010010]
sage: g.edges(sort=True)                                                #␣
↪needs sage.graphs
[(word: 01001010, word: 01010010, word: 010),
 (word: 01010010, word: 01001010, word: 01010),
 (word: 01010010, word: 01001010, word: 10)]
```

```
>>> from sage.all import *
>>> f = words.FibonacciWord()[:Integer(100)]
>>> g = f.reduced_rauzy_graph(Integer(8));g                              ␣
↪     # needs sage.graphs
Looped multi-digraph on 2 vertices
>>> g.vertices(sort=True)                                              #␣
↪needs sage.graphs
[word: 01001010, word: 01010010]
>>> g.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(word: 01001010, word: 01010010, word: 010),
 (word: 01010010, word: 01001010, word: 01010),
 (word: 01010010, word: 01001010, word: 10)]
```

For periodic words:

```
sage: from itertools import cycle
sage: w = Word(cycle('abcd'))[:100]
sage: g = w.reduced_rauzy_graph(3)                                      #␣
↪needs sage.graphs
sage: g.edges(sort=True)                                                #␣
↪needs sage.graphs
[(word: abc, word: abc, word: dabc)]
```

```
>>> from sage.all import *
>>> from itertools import cycle
>>> w = Word(cycle('abcd'))[:Integer(100)]
>>> g = w.reduced_rauzy_graph(Integer(3))                                ␣
↪     # needs sage.graphs
>>> g.edges(sort=True)                                                 #␣
↪needs sage.graphs
[(word: abc, word: abc, word: dabc)]
```

```
sage: w = Word('111')
sage: for i in range(5): w.reduced_rauzy_graph(i)                               #␣
↪needs sage.graphs
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped multi-digraph on 1 vertex
Looped multi-digraph on 0 vertices
```

```
>>> from sage.all import *
>>> w = Word('111')
>>> for i in range(Integer(5)): w.reduced_rauzy_graph(i)                        ␣
↪     # needs sage.graphs
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped multi-digraph on 1 vertex
Looped multi-digraph on 0 vertices
```

For ultimately periodic words:

```
sage: sigma = WordMorphism('a->abcd,b->cd,c->cd,d->cd')
sage: w = sigma.fixed_point('a')[:100]; w                                       #␣
↪needs sage.modules
word: abcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcd...
sage: g = w.reduced_rauzy_graph(5)                                              #␣
↪needs sage.graphs
sage: g.vertices(sort=True)                                                     #␣
↪needs sage.graphs
[word: abcdc, word: cdcdc]
sage: g.edges(sort=True)                                                        #␣
↪needs sage.graphs
[(word: abcdc, word: cdcdc, word: dc), (word: cdcdc, word: cdcdc, word: dc)]
```

```
>>> from sage.all import *
>>> sigma = WordMorphism('a->abcd,b->cd,c->cd,d->cd')
>>> w = sigma.fixed_point('a')[:Integer(100)]; w                                ␣
↪     # needs sage.modules
word: abcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcd...
>>> g = w.reduced_rauzy_graph(Integer(5))                                       ␣
↪     # needs sage.graphs
>>> g.vertices(sort=True)                                                     #␣
↪needs sage.graphs
[word: abcdc, word: cdcdc]
>>> g.edges(sort=True)                                                        #␣
↪needs sage.graphs
[(word: abcdc, word: cdcdc, word: dc), (word: cdcdc, word: cdcdc, word: dc)]
```

AUTHOR:

Julien Leroy (March 2010): initial version

return_words(*fact*)

Return the set of return words of `fact` in `self`.

This is the set of all factors starting by the given factor and ending just before the next occurrence of this factor. See [Dur1998] and [HZ1999].

INPUT:

- `fact` – a non-empty finite word

OUTPUT: a Python set of finite words

EXAMPLES:

```
sage: Word('21331233213231').return_words(Word('2'))
{word: 213, word: 21331, word: 233}
sage: Word().return_words(Word('213'))
set()
sage: Word('121212').return_words(Word('1212'))
{word: 12}
```

```
>>> from sage.all import *
>>> Word('21331233213231').return_words(Word('2'))
{word: 213, word: 21331, word: 233}
>>> Word().return_words(Word('213'))
set()
>>> Word('121212').return_words(Word('1212'))
{word: 12}
```

```
sage: TM = words.ThueMorseWord()[:1000]
sage: sorted(TM.return_words(Word([0])))
[word: 0, word: 01, word: 011]
```

```
>>> from sage.all import *
>>> TM = words.ThueMorseWord()[:Integer(1000)]
>>> sorted(TM.return_words(Word([Integer(0)])))
[word: 0, word: 01, word: 011]
```

return_words_derivate(*fact*)

Return the word generated by mapping a letter to each occurrence of the return words for the given factor dropping any dangling prefix and suffix. See for instance [Dur1998].

EXAMPLES:

```
sage: Word('12131221312313122').return_words_derivate(Word('1'))
word: 123242
```

```
>>> from sage.all import *
>>> Word('12131221312313122').return_words_derivate(Word('1'))
word: 123242
```

rev_lex_less(*other*)

Return `True` if the word `self` is reverse lexicographically less than `other`.

EXAMPLES:

```
sage: Word([1,2,4]).rev_lex_less(Word([1,3,2]))
True
```

(continues on next page)

(continued from previous page)

```
sage: Word([3,2,1]).rev_lex_less(Word([1,2,3]))
False
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(2),Integer(4)]).rev_lex_less(Word([Integer(1),
↪Integer(3),Integer(2)]))
True
>>> Word([Integer(3),Integer(2),Integer(1)]).rev_lex_less(Word([Integer(1),
↪Integer(2),Integer(3)]))
False
```

reversal()

Return the reversal of `self`.

EXAMPLES:

```
sage: Word('124563').reversal()
word: 365421
```

```
>>> from sage.all import *
>>> Word('124563').reversal()
word: 365421
```

rfind(*sub*, *start=0*, *end=None*)

Return the index of the last occurrence of `sub` in `self`, such that `sub` is contained within `self[start:end]`. Return `-1` on failure.

INPUT:

- `sub` – string, list, tuple or word to search for
- `start` – nonnegative integer (default: 0); specifying the position at which the search must stop
- `end` – nonnegative integer (default: `None`); specifying the position from which to start the search. If `None`, then the search is performed up to the end of the string.

OUTPUT: nonnegative integer or -1

EXAMPLES:

```
sage: w = Word([0,1,0,0,1])
sage: w.rfind(Word([0,1]))
3
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(0),Integer(0),Integer(1)])
>>> w.rfind(Word([Integer(0),Integer(1)]))
3
```

The `sub` parameter can also be a list or a tuple:

```
sage: w.rfind([0,1])
3
sage: w.rfind((0,1))
3
```

```
>>> from sage.all import *
>>> w.rfind([Integer(0),Integer(1)])
3
>>> w.rfind((Integer(0),Integer(1)))
3
```

Examples using the argument `start` and `end`:

```
sage: w.rfind(Word([0,1]), end=4)
0
sage: w.rfind(Word([0,1]), end=5)
3
sage: w.rfind(Word([0,0]), start=2, end=5)
2
sage: w.rfind(Word([0,0]), start=3, end=5)
-1
```

```
>>> from sage.all import *
>>> w.rfind(Word([Integer(0),Integer(1)]), end=Integer(4))
0
>>> w.rfind(Word([Integer(0),Integer(1)]), end=Integer(5))
3
>>> w.rfind(Word([Integer(0),Integer(0)]), start=Integer(2), end=Integer(5))
2
>>> w.rfind(Word([Integer(0),Integer(0)]), start=Integer(3), end=Integer(5))
-1
```

Instances of `Word_str` handle string inputs as well:

```
sage: w = Word('abac')
sage: w.rfind('a')
2
sage: w.rfind(Word('a'))
2
sage: w.rfind([0,1])
-1
```

```
>>> from sage.all import *
>>> w = Word('abac')
>>> w.rfind('a')
2
>>> w.rfind(Word('a'))
2
>>> w.rfind([Integer(0),Integer(1)])
-1
```

right_special_factors(*n=None*)

Return the right special factors (of length `n`).

A factor u of a word w is *right special* if there are two distinct letters a and b such that ua and ub are factors of w.

INPUT:

- `n` – integer (default: `None`); if `None`, it returns all right special factors

OUTPUT: list of words

EXAMPLES:

```
sage: w = words.ThueMorseWord()[:30]
sage: for i in range(5):
....:     print("{} {}".format(i, sorted(w.right_special_factors(i))))
0 [word: ]
1 [word: 0, word: 1]
2 [word: 01, word: 10]
3 [word: 001, word: 010, word: 101, word: 110]
4 [word: 0110, word: 1001]
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(30)]
>>> for i in range(Integer(5)):
...     print("{} {}".format(i, sorted(w.right_special_factors(i))))
0 [word: ]
1 [word: 0, word: 1]
2 [word: 01, word: 10]
3 [word: 001, word: 010, word: 101, word: 110]
4 [word: 0110, word: 1001]
```

right_special_factors_iterator(*n=None*)

Return an iterator over the right special factors (of length `n`).

A factor u of a word w is *right special* if there are two distinct letters a and b such that ua and ub are factors of w.

INPUT:

- `n` – integer (default: `None`); if `None`, it returns an iterator over all right special factors

EXAMPLES:

```
sage: alpha, beta, x = 0.61, 0.54, 0.3
sage: w = words.CodingOfRotationWord(alpha, beta, x)[:40]
sage: sorted(w.right_special_factors_iterator(3))
[word: 010, word: 101]
sage: sorted(w.right_special_factors_iterator(4))
[word: 0101, word: 1010]
sage: sorted(w.right_special_factors_iterator(5))
[word: 00101, word: 11010]
```

```
>>> from sage.all import *
>>> alpha, beta, x = RealNumber('0.61'), RealNumber('0.54'), RealNumber('0.3')
>>> w = words.CodingOfRotationWord(alpha, beta, x)[:Integer(40)]
>>> sorted(w.right_special_factors_iterator(Integer(3)))
[word: 010, word: 101]
>>> sorted(w.right_special_factors_iterator(Integer(4)))
[word: 0101, word: 1010]
>>> sorted(w.right_special_factors_iterator(Integer(5)))
[word: 00101, word: 11010]
```

robinson_schensted()

Return the semistandard tableau and standard tableau pair obtained by running the Robinson-Schensted algorithm on `self`.

This can also be done by running *RSK()* on `self`.

EXAMPLES:

```
sage: Word([1,1,3,1,2,3,1]).robinson_schensted()
[[[1, 1, 1, 1, 3], [2], [3]], [[1, 2, 3, 5, 6], [4], [7]]]
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(1),Integer(3),Integer(1),Integer(2),Integer(3),
↪Integer(1)]).robinson_schensted()
[[[1, 1, 1, 1, 3], [2], [3]], [[1, 2, 3, 5, 6], [4], [7]]]
```

schuetzenberger_involution(*n=None*)

Return the Schützenberger involution of the word `self`, which is obtained by reverting the word and then complementing all letters within the underlying ordered alphabet. If `n` is specified, the underlying alphabet is assumed to be $[1, 2, \ldots, n]$. If no alphabet is specified, n is the maximal letter appearing in `self`.

INPUT:

- `self` – a word
- `n` – integer specifying the maximal letter in the alphabet (optional)

OUTPUT: a word, the Schützenberger involution of `self`

EXAMPLES:

```
sage: w = Word([9,7,4,1,6,2,3])
sage: v = w.schuetzenberger_involution(); v
word: 7849631
sage: v.parent()
Finite words over Set of Python objects of class 'object'

sage: w = Word([1,2,3],alphabet=[1,2,3,4,5])
sage: v = w.schuetzenberger_involution();v
word: 345
sage: v.parent()
Finite words over {1, 2, 3, 4, 5}

sage: w = Word([1,2,3])
sage: v = w.schuetzenberger_involution(n=5);v
word: 345
sage: v.parent()
Finite words over Set of Python objects of class 'object'

sage: w = Word([11,32,69,2,53,1,2,3,18,41])
sage: w.schuetzenberger_involution()
word: 29,52,67,68,69,17,68,1,38,59

sage: w = Word([],alphabet=[1,2,3,4,5])
sage: w.schuetzenberger_involution()
word:
```

(continues on next page)

(continued from previous page)

```
sage: w = Word([])
sage: w.schuetzenberger_involution()
word:
```

```
>>> from sage.all import *
>>> w = Word([Integer(9),Integer(7),Integer(4),Integer(1),Integer(6),
↪Integer(2),Integer(3)])
>>> v = w.schuetzenberger_involution(); v
word: 7849631
>>> v.parent()
Finite words over Set of Python objects of class 'object'

>>> w = Word([Integer(1),Integer(2),Integer(3)],alphabet=[Integer(1),
↪Integer(2),Integer(3),Integer(4),Integer(5)])
>>> v = w.schuetzenberger_involution();v
word: 345
>>> v.parent()
Finite words over {1, 2, 3, 4, 5}

>>> w = Word([Integer(1),Integer(2),Integer(3)])
>>> v = w.schuetzenberger_involution(n=Integer(5));v
word: 345
>>> v.parent()
Finite words over Set of Python objects of class 'object'

>>> w = Word([Integer(11),Integer(32),Integer(69),Integer(2),Integer(53),
↪Integer(1),Integer(2),Integer(3),Integer(18),Integer(41)])
>>> w.schuetzenberger_involution()
word: 29,52,67,68,69,17,68,1,38,59

>>> w = Word([],alphabet=[Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)])
>>> w.schuetzenberger_involution()
word:

>>> w = Word([])
>>> w.schuetzenberger_involution()
word:
```

shifted_shuffle(*other*, *shift=None*)

Return the combinatorial class representing the shifted shuffle product between words `self` and `other`. This is the same as the shuffle product of `self` with the word obtained from `other` by incrementing its values (i.e. its letters) by the given `shift`.

INPUT:

- `other` – finite word over the integers
- `shift` – integer or `None` (default: `None`); added to each letter of `other`. When `shift` is `None`, it is replaced by `self.length()`

OUTPUT: combinatorial class of shifted shuffle products of `self` and `other`

EXAMPLES:

```
sage: w = Word([0,1,1])
sage: sp = w.shifted_shuffle(w); sp
Shuffle product of word: 011 and word: 344
sage: sp = w.shifted_shuffle(w, 2); sp
Shuffle product of word: 011 and word: 233
sage: sp.cardinality()
20
sage: WordOptions(identifier='')
sage: sp.list()
[011233, 012133, 012313, 012331, 021133, 021313, 021331, 023113, 023131,
→023311, 201133, 201313, 201331, 203113, 203131, 203311, 230113, 230131,
→230311, 233011]
sage: WordOptions(identifier='word: ')
sage: y = Word('aba')
sage: y.shifted_shuffle(w,2)
Traceback (most recent call last):
...
ValueError: for shifted shuffle, words must only contain integers as letters
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(1)])
>>> sp = w.shifted_shuffle(w); sp
Shuffle product of word: 011 and word: 344
>>> sp = w.shifted_shuffle(w, Integer(2)); sp
Shuffle product of word: 011 and word: 233
>>> sp.cardinality()
20
>>> WordOptions(identifier='')
>>> sp.list()
[011233, 012133, 012313, 012331, 021133, 021313, 021331, 023113, 023131,
→023311, 201133, 201313, 201331, 203113, 203131, 203311, 230113, 230131,
→230311, 233011]
>>> WordOptions(identifier='word: ')
>>> y = Word('aba')
>>> y.shifted_shuffle(w,Integer(2))
Traceback (most recent call last):
...
ValueError: for shifted shuffle, words must only contain integers as letters
```

shuffle(*other*, *overlap=0*)

Return the combinatorial class representing the shuffle product between words `self` and `other`. This consists of all words of length `self.length()+other.length()` that have both `self` and `other` as subwords.

If `overlap` is nonzero, then the combinatorial class representing the shuffle product with overlaps is returned. The calculation of the shift in each overlap is done relative to the order of the alphabet. For example, a shifted by a is b in the alphabet $[a, b, c]$ and 0 shifted by 1 in $[0, 1, 2, 3]$ is 2.

INPUT:

- `other` – finite word
- `overlap` – (default: `0`) integer or `True`

OUTPUT: combinatorial class of shuffle product of `self` and `other`

EXAMPLES:

```
sage: ab = Word("ab")
sage: cd = Word("cd")
sage: sp = ab.shuffle(cd); sp
Shuffle product of word: ab and word: cd
sage: sp.cardinality()
6
sage: sp.list()
[word: abcd, word: acbd, word: acdb, word: cabd, word: cadb, word: cdab]
sage: w = Word([0,1])
sage: u = Word([2,3])
sage: w.shuffle(w)
Shuffle product of word: 01 and word: 01
sage: u.shuffle(u)
Shuffle product of word: 23 and word: 23
sage: w.shuffle(u)
Shuffle product of word: 01 and word: 23
sage: sp2 = w.shuffle(u,2); sp2
Overlapping shuffle product of word: 01 and word: 23 with 2 overlaps
sage: list(sp2)
[word: 24]
```

```
>>> from sage.all import *
>>> ab = Word("ab")
>>> cd = Word("cd")
>>> sp = ab.shuffle(cd); sp
Shuffle product of word: ab and word: cd
>>> sp.cardinality()
6
>>> sp.list()
[word: abcd, word: acbd, word: acdb, word: cabd, word: cadb, word: cdab]
>>> w = Word([Integer(0),Integer(1)])
>>> u = Word([Integer(2),Integer(3)])
>>> w.shuffle(w)
Shuffle product of word: 01 and word: 01
>>> u.shuffle(u)
Shuffle product of word: 23 and word: 23
>>> w.shuffle(u)
Shuffle product of word: 01 and word: 23
>>> sp2 = w.shuffle(u,Integer(2)); sp2
Overlapping shuffle product of word: 01 and word: 23 with 2 overlaps
>>> list(sp2)
[word: 24]
```

squares()

Return a set of all distinct squares of `self`.

EXAMPLES:

```
sage: sorted(Word('cacao').squares())
[word: , word: caca]
sage: sorted(Word('1111').squares())
[word: , word: 11, word: 1111]
sage: w = Word('00110011010')
```

(continues on next page)

(continued from previous page)

```
sage: sorted(w.squares())
[word: , word: 00, word: 00110011, word: 01100110, word: 1010, word: 11]
```

```
>>> from sage.all import *
>>> sorted(Word('cacao').squares())
[word: , word: caca]
>>> sorted(Word('1111').squares())
[word: , word: 11, word: 1111]
>>> w = Word('00110011010')
>>> sorted(w.squares())
[word: , word: 00, word: 00110011, word: 01100110, word: 1010, word: 11]
```

standard_factorization()

Return the standard factorization of `self`.

The *standard factorization* of a word w of length greater than 1 is the factorization $w = uv$ where v is the longest proper suffix of w that is a Lyndon word.

Note that if w is a Lyndon word of length greater than 1 with standard factorization $w = uv$, then u and v are also Lyndon words and $u < v$.

See for instance [CFL1958], [Duv1983] and [Lot2002].

INPUT:

- `self` – finite word of length greater than 1

OUTPUT: 2-tuple (u, v)

EXAMPLES:

```
sage: Words('01')('0010110011').standard_factorization()
(word: 001011, word: 0011)
sage: Words('123')('1223312').standard_factorization()
(word: 12233, word: 12)
sage: Word([3,2,1]).standard_factorization()
(word: 32, word: 1)
```

```
>>> from sage.all import *
>>> Words('01')('0010110011').standard_factorization()
(word: 001011, word: 0011)
>>> Words('123')('1223312').standard_factorization()
(word: 12233, word: 12)
>>> Word([Integer(3),Integer(2),Integer(1)]).standard_factorization()
(word: 32, word: 1)
```

```
sage: w = Word('0010110011',alphabet='01')
sage: w.standard_factorization()
(word: 001011, word: 0011)
sage: w = Word('0010110011',alphabet='10')
sage: w.standard_factorization()
(word: 001011001, word: 1)
sage: w = Word('1223312',alphabet='123')
sage: w.standard_factorization()
(word: 12233, word: 12)
```

```
>>> from sage.all import *
>>> w = Word('0010110011',alphabet='01')
>>> w.standard_factorization()
(word: 001011, word: 0011)
>>> w = Word('0010110011',alphabet='10')
>>> w.standard_factorization()
(word: 001011001, word: 1)
>>> w = Word('1223312',alphabet='123')
>>> w.standard_factorization()
(word: 12233, word: 12)
```

standard_permutation()

Return the standard permutation of the word `self` on the ordered alphabet. It is defined as the permutation with exactly the same inversions as `self`. Equivalently, it is the permutation of minimal length whose inverse sorts `self`.

EXAMPLES:

```
sage: w = Word([1,2,3,2,2,1]); w
word: 123221
sage: p = w.standard_permutation(); p
[1, 3, 6, 4, 5, 2]
sage: v = Word(p.inverse().action(w)); v
word: 112223
sage: [q for q in Permutations(w.length())
....:      if q.length() <= p.length() and
....:      q.inverse().action(w) == list(v)]
[[1, 3, 6, 4, 5, 2]]
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(2),Integer(3),Integer(2),Integer(2),
↪Integer(1)]); w
word: 123221
>>> p = w.standard_permutation(); p
[1, 3, 6, 4, 5, 2]
>>> v = Word(p.inverse().action(w)); v
word: 112223
>>> [q for q in Permutations(w.length())
...      if q.length() <= p.length() and
...      q.inverse().action(w) == list(v)]
[[1, 3, 6, 4, 5, 2]]
```

```
sage: w = Words([1,2,3])([1,2,3,2,2,1,2,1]); w
word: 12322121
sage: p = w.standard_permutation(); p
[1, 4, 8, 5, 6, 2, 7, 3]
sage: Word(p.inverse().action(w))
word: 11122223
```

```
>>> from sage.all import *
>>> w = Words([Integer(1),Integer(2),Integer(3)])([Integer(1),Integer(2),
↪Integer(3),Integer(2),Integer(2),Integer(1),Integer(2),Integer(1)]); w
```

(continues on next page)

(continued from previous page)

```
word: 12322121
>>> p = w.standard_permutation(); p
[1, 4, 8, 5, 6, 2, 7, 3]
>>> Word(p.inverse().action(w))
word: 11122223
```

```
sage: w = Words([3,2,1])([1,2,3,2,2,1,2,1]); w
word: 12322121
sage: p = w.standard_permutation(); p
[6, 2, 1, 3, 4, 7, 5, 8]
sage: Word(p.inverse().action(w))
word: 32222111
```

```
>>> from sage.all import *
>>> w = Words([Integer(3),Integer(2),Integer(1)])([Integer(1),Integer(2),
↪Integer(3),Integer(2),Integer(2),Integer(1),Integer(2),Integer(1)]); w
word: 12322121
>>> p = w.standard_permutation(); p
[6, 2, 1, 3, 4, 7, 5, 8]
>>> Word(p.inverse().action(w))
word: 32222111
```

```
sage: w = Words('ab')('abbaba'); w
word: abbaba
sage: p = w.standard_permutation(); p
[1, 4, 5, 2, 6, 3]
sage: Word(p.inverse().action(w))
word: aaabbb
```

```
>>> from sage.all import *
>>> w = Words('ab')('abbaba'); w
word: abbaba
>>> p = w.standard_permutation(); p
[1, 4, 5, 2, 6, 3]
>>> Word(p.inverse().action(w))
word: aaabbb
```

```
sage: w = Words('ba')('abbaba'); w
word: abbaba
sage: p = w.standard_permutation(); p
[4, 1, 2, 5, 3, 6]
sage: Word(p.inverse().action(w))
word: bbbaaa
```

```
>>> from sage.all import *
>>> w = Words('ba')('abbaba'); w
word: abbaba
>>> p = w.standard_permutation(); p
[4, 1, 2, 5, 3, 6]
>>> Word(p.inverse().action(w))
word: bbbaaa
```

sturmian_desubstitute_as_possible()

Sturmian-desubstitute the word `self` as much as possible.

The finite word `self` must be defined on a two-letter alphabet or use at most two letters.

It can be Sturmian desubstituted if one letter appears isolated: the Sturmian desubstitution consists in removing one letter per run of the non-isolated letter. The accelerated Sturmian desubstitution consists in removing a run equal to the length of the shortest inner run from any run of the non-isolated letter (including possible leading and trailing runs even if they have shorter length). The (accelerated) Sturmian desubstitution is done as much as possible. A word is a factor of a Sturmian word if, and only if, the result is the empty word.

OUTPUT: a finite word defined on a two-letter alphabet

EXAMPLES:

```
sage: u = Word('10111101101110111',alphabet='01') ; u
word: 10111101101110111
sage: v = u.sturmian_desubstitute_as_possible() ; v
word: 01100101
sage: v == v.sturmian_desubstitute_as_possible()
True

sage: Word('azaazaaazaaazaazaaaz', alphabet='az').sturmian_desubstitute_as_
↪possible()
word:
```

```
>>> from sage.all import *
>>> u = Word('10111101101110111',alphabet='01') ; u
word: 10111101101110111
>>> v = u.sturmian_desubstitute_as_possible() ; v
word: 01100101
>>> v == v.sturmian_desubstitute_as_possible()
True

>>> Word('azaazaaazaaazaazaaaz', alphabet='az').sturmian_desubstitute_as_
↪possible()
word:
```

AUTHOR:

- Thierry Monteil

subword_complementaries(*other*)

Return the possible complementaries `other` minus `self` if `self` is a subword of `other` (empty list otherwise). The complementary is made of all the letters that are in `other` once we removed the letters of `self`. There can be more than one.

To check whether `self` is a subword of `other` (without knowing its complementaries), use `self.is_subword_of(other)`, and to count the number of occurrences of `self` in `other`, use `other.number_of_subword_occurrences(self)`.

INPUT:

- `other` – finite word

OUTPUT: list of all the complementary subwords of `self` in `other`

EXAMPLES:

```
sage: Word('tamtam').subword_complementaries(Word('ta'))
[]

sage: Word('mta').subword_complementaries(Word('tamtam'))
[word: tam]

sage: Word('ta').subword_complementaries(Word('tamtam'))
[word: mtam, word: amtm, word: tamm]

sage: Word('a').subword_complementaries(Word('a'))
[word: ]
```

```
>>> from sage.all import *
>>> Word('tamtam').subword_complementaries(Word('ta'))
[]

>>> Word('mta').subword_complementaries(Word('tamtam'))
[word: tam]

>>> Word('ta').subword_complementaries(Word('tamtam'))
[word: mtam, word: amtm, word: tamm]

>>> Word('a').subword_complementaries(Word('a'))
[word: ]
```

suffix_tree()

Alias for `implicit_suffix_tree()`.

EXAMPLES:

```
sage: Word('abbabaab').suffix_tree()
Implicit Suffix Tree of the word: abbabaab
```

```
>>> from sage.all import *
>>> Word('abbabaab').suffix_tree()
Implicit Suffix Tree of the word: abbabaab
```

suffix_trie()

Return the suffix trie of `self`.

The *suffix trie* of a finite word w is a data structure representing the factors of w. It is a tree whose edges are labelled with letters of w, and whose leafs correspond to suffixes of w.

Type `sage.combinat.words.suffix_trees.SuffixTrie?` for more information.

EXAMPLES:

```
sage: w = Word("cacao")
sage: w.suffix_trie()
Suffix Trie of the word: cacao
```

```
>>> from sage.all import *
>>> w = Word("cacao")
```

(continues on next page)

(continued from previous page)

```
>>> w.suffix_trie()
Suffix Trie of the word: cacao
```

```
sage: w = Word([0,1,0,1,1])
sage: w.suffix_trie()
Suffix Trie of the word: 01011
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(0),Integer(1),Integer(1)])
>>> w.suffix_trie()
Suffix Trie of the word: 01011
```

swap(*i*, *j=None*)

Return the word w with entries at positions `i` and `j` swapped. By default, `j = i+1`.

EXAMPLES:

```
sage: Word([1,2,3]).swap(0,2)
word: 321
sage: Word([1,2,3]).swap(1)
word: 132
sage: Word("abba").swap(1,-1)
word: aabb
```

```
>>> from sage.all import *
>>> Word([Integer(1),Integer(2),Integer(3)]).swap(Integer(0),Integer(2))
word: 321
>>> Word([Integer(1),Integer(2),Integer(3)]).swap(Integer(1))
word: 132
>>> Word("abba").swap(Integer(1),-Integer(1))
word: aabb
```

swap_decrease(*i*)

Return the word with positions `i` and `i+1` exchanged if `self[i] < self[i+1]`. Otherwise, it returns `self`.

EXAMPLES:

```
sage: w = Word([1,3,2])
sage: w.swap_decrease(0)
word: 312
sage: w.swap_decrease(1)
word: 132
sage: w.swap_decrease(1) is w
True
sage: Words("ab")("abba").swap_decrease(0)
word: baba
sage: Words("ba")("abba").swap_decrease(0)
word: abba
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(3),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> w.swap_decrease(Integer(0))
word: 312
>>> w.swap_decrease(Integer(1))
word: 132
>>> w.swap_decrease(Integer(1)) is w
True
>>> Words("ab")("abba").swap_decrease(Integer(0))
word: baba
>>> Words("ba")("abba").swap_decrease(Integer(0))
word: abba
```

swap_increase(*i*)

Return the word with positions `i` and `i+1` exchanged if `self[i] > self[i+1]`. Otherwise, it returns `self`.

EXAMPLES:

```
sage: w = Word([1,3,2])
sage: w.swap_increase(1)
word: 123
sage: w.swap_increase(0)
word: 132
sage: w.swap_increase(0) is w
True
sage: Words("ab")("abba").swap_increase(0)
word: abba
sage: Words("ba")("abba").swap_increase(0)
word: baba
```

```
>>> from sage.all import *
>>> w = Word([Integer(1),Integer(3),Integer(2)])
>>> w.swap_increase(Integer(1))
word: 123
>>> w.swap_increase(Integer(0))
word: 132
>>> w.swap_increase(Integer(0)) is w
True
>>> Words("ab")("abba").swap_increase(Integer(0))
word: abba
>>> Words("ba")("abba").swap_increase(Integer(0))
word: baba
```

to_integer_list()

Return a list of integers from `[0,1,...,self.length()-1]` in the same relative order as the letters in `self` in the parent.

EXAMPLES:

```
sage: from itertools import count
sage: w = Word('abbabaab')
sage: w.to_integer_list()
[0, 1, 1, 0, 1, 0, 0, 1]
sage: w = Word(iter("cacao"), length='finite')
```

(continues on next page)

(continued from previous page)

```
sage: w.to_integer_list()
[1, 0, 1, 0, 2]
sage: w = Words([3,2,1])([2,3,3,1])
sage: w.to_integer_list()
[1, 0, 0, 2]
```

```
>>> from sage.all import *
>>> from itertools import count
>>> w = Word('abbabaab')
>>> w.to_integer_list()
[0, 1, 1, 0, 1, 0, 0, 1]
>>> w = Word(iter("cacao"), length='finite')
>>> w.to_integer_list()
[1, 0, 1, 0, 2]
>>> w = Words([Integer(3),Integer(2),Integer(1)])([Integer(2),Integer(3),
↪Integer(3),Integer(1)])
>>> w.to_integer_list()
[1, 0, 0, 2]
```

to_integer_word()

Return a word over the alphabet `[0,1,...,self.length()-1]` whose letters are in the same relative order as the letters of `self` in the parent.

EXAMPLES:

```
sage: from itertools import count
sage: w = Word('abbabaab')
sage: w.to_integer_word()
word: 01101001
sage: w = Word(iter("cacao"), length='finite')
sage: w.to_integer_word()
word: 10102

sage: w = Words([3,2,1])([2,3,3,1])
sage: w.to_integer_word()
word: 1002
```

```
>>> from sage.all import *
>>> from itertools import count
>>> w = Word('abbabaab')
>>> w.to_integer_word()
word: 01101001
>>> w = Word(iter("cacao"), length='finite')
>>> w.to_integer_word()
word: 10102

>>> w = Words([Integer(3),Integer(2),Integer(1)])([Integer(2),Integer(3),
↪Integer(3),Integer(1)])
>>> w.to_integer_word()
word: 1002
```

to_monoid_element()

Return `self` as an element of the free monoid with the same alphabet as `self`.

EXAMPLES:

```
sage: w = Word('aabb')
sage: w.to_monoid_element()
a^2*b^2
sage: W = Words('abc')
sage: w = W(w)
sage: w.to_monoid_element()
a^2*b^2
```

```
>>> from sage.all import *
>>> w = Word('aabb')
>>> w.to_monoid_element()
a^2*b^2
>>> W = Words('abc')
>>> w = W(w)
>>> w.to_monoid_element()
a^2*b^2
```

to_ordered_set_partition()

Return the ordered set partition correspond to `self`.

If w is a finite word of length n, then the corresponding ordered set partition is an ordered set partition $(P_1, P_2, \ldots, P_k)$ of $\{1, 2, \ldots, n\}$, where each block P_i is the set of positions at which the i-th smallest letter occurring in w occurs in w.

EXAMPLES:

```
sage: w = Word('abbabaab')
sage: w.to_ordered_set_partition()
[{1, 4, 6, 7}, {2, 3, 5, 8}]
sage: Word([-10, 3, -10, 2]).to_ordered_set_partition()
[{1, 3}, {4}, {2}]
sage: Word([]).to_ordered_set_partition()
[]
sage: Word('aaaaa').to_ordered_set_partition()
[{1, 2, 3, 4, 5}]
```

```
>>> from sage.all import *
>>> w = Word('abbabaab')
>>> w.to_ordered_set_partition()
[{1, 4, 6, 7}, {2, 3, 5, 8}]
>>> Word([-Integer(10), Integer(3), -Integer(10), Integer(2)]).to_ordered_set_
→partition()
[{1, 3}, {4}, {2}]
>>> Word([]).to_ordered_set_partition()
[]
>>> Word('aaaaa').to_ordered_set_partition()
[{1, 2, 3, 4, 5}]
```

topological_entropy(n)

Return the topological entropy for the factors of length `n`.

The topological entropy of a sequence u is defined as the exponential growth rate of the complexity of u as the length increases: $H_{top}(u) = \lim_{n\to\infty} \frac{\log_d(p_u(n))}{n}$ where d denotes the cardinality of the alphabet and

$p_u(n)$ is the complexity function, i.e. the number of factors of length n in the sequence u [Fog2002].

INPUT:

- `self` – a word defined over a finite alphabet
- `n` – positive integer

OUTPUT: real number (a symbolic expression)

EXAMPLES:

```
sage: W = Words([0, 1])
sage: w = W([0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1])
sage: t = w.topological_entropy(3); t                                   #␣
↪needs sage.symbolic
1/3*log(7)/log(2)
sage: n(t)                                                              #␣
↪needs sage.symbolic
0.935784974019201
```

```
>>> from sage.all import *
>>> W = Words([Integer(0), Integer(1)])
>>> w = W([Integer(0), Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1)])
>>> t = w.topological_entropy(Integer(3)); t                             ␣
↪     # needs sage.symbolic
1/3*log(7)/log(2)
>>> n(t)                                                               #␣
↪needs sage.symbolic
0.935784974019201
```

```
sage: w = words.ThueMorseWord()[:100]
sage: topo = w.topological_entropy
sage: for i in range(0, 41, 5):                                         #␣
↪needs sage.symbolic
....:     print("{} {}".format(i, n(topo(i), digits=5)))
0 1.0000
5 0.71699
10 0.48074
15 0.36396
20 0.28774
25 0.23628
30 0.20075
35 0.17270
40 0.14827
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()[:Integer(100)]
>>> topo = w.topological_entropy
>>> for i in range(Integer(0), Integer(41), Integer(5)):                 ␣
↪                   # needs sage.symbolic
...     print("{} {}".format(i, n(topo(i), digits=Integer(5))))
0 1.0000
```

(continues on next page)

(continued from previous page)

```
5 0.71699
10 0.48074
15 0.36396
20 0.28774
25 0.23628
30 0.20075
35 0.17270
40 0.14827
```

If no alphabet is specified, an error is raised:

```
sage: w = Word(range(20))
sage: w.topological_entropy(3)
Traceback (most recent call last):
...
TypeError: The word must be defined over a finite alphabet
```

```
>>> from sage.all import *
>>> w = Word(range(Integer(20)))
>>> w.topological_entropy(Integer(3))
Traceback (most recent call last):
...
TypeError: The word must be defined over a finite alphabet
```

The following is ok:

```
sage: W = Words(range(20))
sage: w = W(range(20))
sage: w.topological_entropy(3)                                          #␣
↪needs sage.symbolic
1/3*log(18)/log(20)
```

```
>>> from sage.all import *
>>> W = Words(range(Integer(20)))
>>> w = W(range(Integer(20)))
>>> w.topological_entropy(Integer(3))                                    ␣
↪     # needs sage.symbolic
1/3*log(18)/log(20)
```

sage.combinat.words.finite_word.**evaluation_dict**(*w*)

Return a dictionary keyed by the letters occurring in `w` with values the number of occurrences of the letter.

INPUT:

- `w` – a word

sage.combinat.words.finite_word.**word_to_ordered_set_partition**(*w*)

Return the ordered set partition corresponding to a finite word w.

If w is a finite word of length n, then the corresponding ordered set partition is an ordered set partition $(P_1, P_2, \ldots, P_k)$ of $\{1, 2, \ldots, n\}$, where each block P_i is the set of positions at which the i-th smallest letter occurring in w occurs in w. (Positions are 1-based.)

This is the same functionality that `to_ordered_set_partition()` provides, but without the wrapping: The input w can be given as a list or tuple, not necessarily as a word; and the output is returned as a list of lists (which are the blocks of the ordered set partition in increasing order), not as an ordered set partition.

EXAMPLES:

```
sage: from sage.combinat.words.finite_word import word_to_ordered_set_partition
sage: word_to_ordered_set_partition([3, 6, 3, 1])
[[4], [1, 3], [2]]
sage: word_to_ordered_set_partition((1, 3, 3, 7))
[[1], [2, 3], [4]]
sage: word_to_ordered_set_partition("noob")
[[4], [1], [2, 3]]
sage: word_to_ordered_set_partition(Word("hell"))
[[2], [1], [3, 4]]
sage: word_to_ordered_set_partition([1])
[[1]]
sage: word_to_ordered_set_partition([])
[]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.finite_word import word_to_ordered_set_partition
>>> word_to_ordered_set_partition([Integer(3), Integer(6), Integer(3),␣
↪Integer(1)])
[[4], [1, 3], [2]]
>>> word_to_ordered_set_partition((Integer(1), Integer(3), Integer(3),␣
↪Integer(7)))
[[1], [2, 3], [4]]
>>> word_to_ordered_set_partition("noob")
[[4], [1], [2, 3]]
>>> word_to_ordered_set_partition(Word("hell"))
[[2], [1], [3, 4]]
>>> word_to_ordered_set_partition([Integer(1)])
[[1]]
>>> word_to_ordered_set_partition([])
[]
```

5.1.359 Infinite word

AUTHORS:

- Sebastien Labbe
- Franco Saliola

EXAMPLES:

Creation of an infinite word

Periodic infinite words:

```
sage: v = Word([0, 4, 8, 8, 3])
sage: vv = v^Infinity
sage: vv
word: 0488304883048830488304883048830488304883...
```

```
>>> from sage.all import *
>>> v = Word([Integer(0), Integer(4), Integer(8), Integer(8), Integer(3)])
>>> vv = v**Infinity
>>> vv
word: 0488304883048830488304883048830488304883...
```

Infinite words from a function $f : \mathbb{N} \to A$ over an alphabet A:

```
sage: Word(lambda n: n%3)
word: 0120120120120120120120120120120120120120...
```

```
>>> from sage.all import *
>>> Word(lambda n: n%Integer(3))
word: 0120120120120120120120120120120120120120...
```

```
sage: def t(n):
....:     return add(Integer(n).digits(base=2)) % 2
sage: Word(t, alphabet = [0, 1])
word: 0110100110010110100101100110100110010110...
```

```
>>> from sage.all import *
>>> def t(n):
...     return add(Integer(n).digits(base=Integer(2))) % Integer(2)
>>> Word(t, alphabet = [Integer(0), Integer(1)])
word: 0110100110010110100101100110100110010110...
```

or as a one-liner:

```
sage: Word(lambda n : add(Integer(n).digits(base=2)) % 2, alphabet = [0, 1])
word: 0110100110010110100101100110100110010110...
```

```
>>> from sage.all import *
>>> Word(lambda n : add(Integer(n).digits(base=Integer(2))) % Integer(2), alphabet =␣
↪[Integer(0), Integer(1)])
word: 0110100110010110100101100110100110010110...
```

Infinite words from iterators:

```
sage: from itertools import count,repeat
sage: Word( repeat(4) )
word: 4444444444444444444444444444444444444444...
sage: Word( count() )
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
↪30,31,32,33,34,35,36,37,38,39,...
```

```
>>> from sage.all import *
>>> from itertools import count,repeat
>>> Word( repeat(Integer(4)) )
word: 4444444444444444444444444444444444444444...
>>> Word( count() )
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
↪30,31,32,33,34,35,36,37,38,39,...
```

Infinite words from morphism

For example, let $A = \{a, b\}$ and $\mu : A^* \to A^*$ be the morphism defined by $a \mapsto ab, b \mapsto ba$:

```
sage: mu = WordMorphism('a->ab,b->ba'); mu
WordMorphism: a->ab, b->ba
sage: mu.fixed_point('a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

```
>>> from sage.all import *
>>> mu = WordMorphism('a->ab,b->ba'); mu
WordMorphism: a->ab, b->ba
>>> mu.fixed_point('a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

Infinite words in a specific combinatorial class:

```
sage: W = InfiniteWords("ab"); W
Infinite words over {'a', 'b'}
sage: f = lambda n : 'a' if n % 2 == 1 else 'b'
sage: W(f)
word: babababababababababababababababababababa...
```

```
>>> from sage.all import *
>>> W = InfiniteWords("ab"); W
Infinite words over {'a', 'b'}
>>> f = lambda n : 'a' if n % Integer(2) == Integer(1) else 'b'
>>> W(f)
word: babababababababababababababababababababa...
```

class sage.combinat.words.infinite_word.**InfiniteWord_class**

Bases: *Word_class*

length()

Return the length of `self`.

EXAMPLES:

```
sage: f = lambda n : n % 6
sage: w = Word(f); w
word: 0123450123450123450123450123450123450123...
sage: w.length()
+Infinity
```

```
>>> from sage.all import *
>>> f = lambda n : n % Integer(6)
>>> w = Word(f); w
word: 0123450123450123450123450123450123450123...
>>> w.length()
+Infinity
```

5.1.360 Lyndon words

sage.combinat.words.lyndon_word.**LyndonWord**(*data*, *check=True*)

Construction of a Lyndon word.

INPUT:

- data – list
- check – boolean (default: True); if True, check that the input data represents a Lyndon word

OUTPUT: a Lyndon word

EXAMPLES:

```
sage: LyndonWord([1,2,2])
word: 122
sage: LyndonWord([1,2,3])
word: 123
sage: LyndonWord([2,1,2,3])
Traceback (most recent call last):
...
ValueError: not a Lyndon word
```

```
>>> from sage.all import *
>>> LyndonWord([Integer(1),Integer(2),Integer(2)])
word: 122
>>> LyndonWord([Integer(1),Integer(2),Integer(3)])
word: 123
>>> LyndonWord([Integer(2),Integer(1),Integer(2),Integer(3)])
Traceback (most recent call last):
...
ValueError: not a Lyndon word
```

If check is False, then no verification is done:

```
sage: LyndonWord([2,1,2,3], check=False)
word: 2123
```

```
>>> from sage.all import *
>>> LyndonWord([Integer(2),Integer(1),Integer(2),Integer(3)], check=False)
word: 2123
```

sage.combinat.words.lyndon_word.**LyndonWords**(*e=None*, *k=None*)

Return the combinatorial class of Lyndon words.

A Lyndon word w is a word that is lexicographically less than all of its rotations. Equivalently, whenever w is split into two non-empty substrings, w is lexicographically less than the right substring.

See Wikipedia article Lyndon_word

INPUT:

- no input at all

or

- e – integer; size of alphabet
- k – integer; length of the words

or

- e – a composition

OUTPUT: a combinatorial class of Lyndon words

EXAMPLES:

```
sage: LyndonWords()
Lyndon words
```

```
>>> from sage.all import *
>>> LyndonWords()
Lyndon words
```

If e is an integer, then e specifies the length of the alphabet; k must also be specified in this case:

```
sage: LW = LyndonWords(3, 4); LW
Lyndon words from an alphabet of size 3 of length 4
sage: LW.first()
word: 1112
sage: LW.last()
word: 2333
sage: LW.random_element()  # random                                      #␣
↪needs sage.libs.pari
word: 1232
sage: LW.cardinality()                                                    #␣
↪needs sage.libs.pari
18
```

```
>>> from sage.all import *
>>> LW = LyndonWords(Integer(3), Integer(4)); LW
Lyndon words from an alphabet of size 3 of length 4
>>> LW.first()
word: 1112
>>> LW.last()
word: 2333
>>> LW.random_element()  # random                                      #␣
↪needs sage.libs.pari
word: 1232
>>> LW.cardinality()                                                    #␣
↪needs sage.libs.pari
18
```

If e is a (weak) composition, then it returns the class of Lyndon words that have evaluation e:

```
sage: LyndonWords([2, 0, 1]).list()
[word: 113]
sage: LyndonWords([2, 0, 1, 0, 1]).list()
[word: 1135, word: 1153, word: 1315]
sage: LyndonWords([2, 1, 1]).list()
[word: 1123, word: 1132, word: 1213]
```

```
>>> from sage.all import *
>>> LyndonWords([Integer(2), Integer(0), Integer(1)]).list()
```

(continues on next page)

(continued from previous page)

```
[word: 113]
>>> LyndonWords([Integer(2), Integer(0), Integer(1), Integer(0), Integer(1)]).
↪list()
[word: 1135, word: 1153, word: 1315]
>>> LyndonWords([Integer(2), Integer(1), Integer(1)]).list()
[word: 1123, word: 1132, word: 1213]
```

class sage.combinat.words.lyndon_word.**LyndonWords_class**(*alphabet=None*)

Bases: UniqueRepresentation, Parent

The set of all Lyndon words.

class sage.combinat.words.lyndon_word.**LyndonWords_evaluation**(*e*)

Bases: UniqueRepresentation, Parent

The set of Lyndon words on a fixed multiset of letters.

EXAMPLES:

```
sage: L = LyndonWords([1,2,1])
sage: L
Lyndon words with evaluation [1, 2, 1]
sage: L.list()
[word: 1223, word: 1232, word: 1322]
```

```
>>> from sage.all import *
>>> L = LyndonWords([Integer(1),Integer(2),Integer(1)])
>>> L
Lyndon words with evaluation [1, 2, 1]
>>> L.list()
[word: 1223, word: 1232, word: 1322]
```

cardinality()

Return the number of Lyndon words with the evaluation e.

EXAMPLES:

```
sage: LyndonWords([]).cardinality()
0
sage: LyndonWords([2,2]).cardinality()                                    #␣
↪needs sage.libs.pari
1
sage: LyndonWords([2,3,2]).cardinality()                                  #␣
↪needs sage.libs.pari
30
```

```
>>> from sage.all import *
>>> LyndonWords([]).cardinality()
0
>>> LyndonWords([Integer(2),Integer(2)]).cardinality()                    ␣
↪               # needs sage.libs.pari
1
>>> LyndonWords([Integer(2),Integer(3),Integer(2)]).cardinality()         ␣
```

(continues on next page)

(continued from previous page)

```
↪                                # needs sage.libs.pari
30
```

Check to make sure that the count matches up with the number of Lyndon words generated:

```
sage: comps = [[],[2,2],[3,2,7],[4,2]] + Compositions(4).list()
sage: lws = [LyndonWords(comp) for comp in comps]
sage: all(lw.cardinality() == len(lw.list()) for lw in lws)                #␣
↪needs sage.libs.pari
True
```

```
>>> from sage.all import *
>>> comps = [[],[Integer(2),Integer(2)],[Integer(3),Integer(2),Integer(7)],
↪[Integer(4),Integer(2)]] + Compositions(Integer(4)).list()
>>> lws = [LyndonWords(comp) for comp in comps]
>>> all(lw.cardinality() == len(lw.list()) for lw in lws)                #␣
↪needs sage.libs.pari
True
```

class sage.combinat.words.lyndon_word.**LyndonWords_nk**(*n*, *k*)

Bases: UniqueRepresentation, Parent

Lyndon words of fixed length k over the alphabet $\{1, 2, \ldots, n\}$.

INPUT:

- n – the size of the alphabet
- k – the length of the words

EXAMPLES:

```
sage: L = LyndonWords(3, 4)
sage: L.list()
[word: 1112,
 word: 1113,
 word: 1122,
 word: 1123,
 ...
 word: 1333,
 word: 2223,
 word: 2233,
 word: 2333]
```

```
>>> from sage.all import *
>>> L = LyndonWords(Integer(3), Integer(4))
>>> L.list()
[word: 1112,
 word: 1113,
 word: 1122,
 word: 1123,
 ...
 word: 1333,
 word: 2223,
```

(continues on next page)

(continued from previous page)

```
word: 2233,
word: 2333]
```

cardinality()

sage.combinat.words.lyndon_word.**StandardBracketedLyndonWords**(*n*, *k*)

Return the combinatorial class of standard bracketed Lyndon words from [1, …, n] of length k.

These are in one to one correspondence with the Lyndon words and form a basis for the subspace of degree k of the free Lie algebra of rank n.

EXAMPLES:

```
sage: SBLW33 = StandardBracketedLyndonWords(3,3); SBLW33
Standard bracketed Lyndon words from an alphabet of size 3 of length 3
sage: SBLW33.first()
[1, [1, 2]]
sage: SBLW33.last()
[[2, 3], 3]
sage: SBLW33.cardinality()
8
sage: SBLW33.random_element() in SBLW33
True
```

```
>>> from sage.all import *
>>> SBLW33 = StandardBracketedLyndonWords(Integer(3),Integer(3)); SBLW33
Standard bracketed Lyndon words from an alphabet of size 3 of length 3
>>> SBLW33.first()
[1, [1, 2]]
>>> SBLW33.last()
[[2, 3], 3]
>>> SBLW33.cardinality()
8
>>> SBLW33.random_element() in SBLW33
True
```

class sage.combinat.words.lyndon_word.**StandardBracketedLyndonWords_nk**(*n*, *k*)

Bases: UniqueRepresentation, Parent

cardinality()

EXAMPLES:

```
sage: StandardBracketedLyndonWords(3, 3).cardinality()
8
sage: StandardBracketedLyndonWords(3, 4).cardinality()
18
```

```
>>> from sage.all import *
>>> StandardBracketedLyndonWords(Integer(3), Integer(3)).cardinality()
8
>>> StandardBracketedLyndonWords(Integer(3), Integer(4)).cardinality()
18
```

sage.combinat.words.lyndon_word.**standard_bracketing**(*lw*)

Return the standard bracketing of a Lyndon word `lw`.

EXAMPLES:

```
sage: import sage.combinat.words.lyndon_word as lyndon_word
sage: [lyndon_word.standard_bracketing(u) for u in LyndonWords(3,3)]
[[1, [1, 2]],
 [1, [1, 3]],
 [[1, 2], 2],
 [1, [2, 3]],
 [[1, 3], 2],
 [[1, 3], 3],
 [2, [2, 3]],
 [[2, 3], 3]]
```

```
>>> from sage.all import *
>>> import sage.combinat.words.lyndon_word as lyndon_word
>>> [lyndon_word.standard_bracketing(u) for u in LyndonWords(Integer(3),
↪Integer(3))]
[[1, [1, 2]],
 [1, [1, 3]],
 [[1, 2], 2],
 [1, [2, 3]],
 [[1, 3], 2],
 [[1, 3], 3],
 [2, [2, 3]],
 [[2, 3], 3]]
```

sage.combinat.words.lyndon_word.**standard_unbracketing**(*sblw*)

Return flattened `sblw` if it is a standard bracketing of a Lyndon word, otherwise raise an error.

EXAMPLES:

```
sage: from sage.combinat.words.lyndon_word import standard_unbracketing
sage: standard_unbracketing([1, [2, 3]])
word: 123
sage: standard_unbracketing([[1, 2], 3])
Traceback (most recent call last):
...
ValueError: not a standard bracketing of a Lyndon word
```

```
>>> from sage.all import *
>>> from sage.combinat.words.lyndon_word import standard_unbracketing
>>> standard_unbracketing([Integer(1), [Integer(2), Integer(3)]])
word: 123
>>> standard_unbracketing([[Integer(1), Integer(2)], Integer(3)])
Traceback (most recent call last):
...
ValueError: not a standard bracketing of a Lyndon word
```

5.1.361 Word morphisms/substitutions

This module implements morphisms over finite and infinite words.

AUTHORS:

- Sébastien Labbé (2007-06-01): initial version
- Sébastien Labbé (2008-07-01): merged into sage-words
- Sébastien Labbé (2008-12-17): merged into sage
- Sébastien Labbé (2009-02-03): words next generation
- Sébastien Labbé (2009-11-20): allowing the choice of the datatype of the image. Doc improvements.
- Stepan Starosta (2012-11-09): growing letters

EXAMPLES:

Creation of a morphism from a dictionary or a string:

```
sage: n = WordMorphism({0:[0,2,2,1],1:[0,2],2:[2,2,1]})
```

```
>>> from sage.all import *
>>> n = WordMorphism({Integer(0):[Integer(0),Integer(2),Integer(2),Integer(1)],
↪Integer(1):[Integer(0),Integer(2)],Integer(2):[Integer(2),Integer(2),Integer(1)]})
```

```
sage: m = WordMorphism('x->xyxsxss,s->xyss,y->ys')
```

```
>>> from sage.all import *
>>> m = WordMorphism('x->xyxsxss,s->xyss,y->ys')
```

```
sage: n
WordMorphism: 0->0221, 1->02, 2->221
sage: m
WordMorphism: s->xyss, x->xyxsxss, y->ys
```

```
>>> from sage.all import *
>>> n
WordMorphism: 0->0221, 1->02, 2->221
>>> m
WordMorphism: s->xyss, x->xyxsxss, y->ys
```

The codomain may be specified:

```
sage: WordMorphism({0:[0,2,2,1],1:[0,2],2:[2,2,1]}, codomain=Words([0,1,2,3,4]))
WordMorphism: 0->0221, 1->02, 2->221
```

```
>>> from sage.all import *
>>> WordMorphism({Integer(0):[Integer(0),Integer(2),Integer(2),Integer(1)],
↪Integer(1):[Integer(0),Integer(2)],Integer(2):[Integer(2),Integer(2),Integer(1)]},␣
↪codomain=Words([Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)]))
WordMorphism: 0->0221, 1->02, 2->221
```

Power of a morphism:

```
sage: n^2
WordMorphism: 0->022122122102, 1->0221221, 2->22122102
```

```
>>> from sage.all import *
>>> n**Integer(2)
WordMorphism: 0->022122122102, 1->0221221, 2->22122102
```

Image under a morphism:

```
sage: m('y')
word: ys
sage: m('xxxsy')
word: xyxsxssxyxsxssxyxsxssxyssys
```

```
>>> from sage.all import *
>>> m('y')
word: ys
>>> m('xxxsy')
word: xyxsxssxyxsxssxyxsxssxyssys
```

Iterated image under a morphism:

```
sage: m('y', 3)
word: ysxyssxyxsxssysxyssxyss
```

```
>>> from sage.all import *
>>> m('y', Integer(3))
word: ysxyssxyxsxssysxyssxyss
```

See more examples in the documentation of the call method (`m.__call__?`).

Infinite fixed point of morphism:

```
sage: fix = m.fixed_point('x')
sage: fix
word: xyxsxssysxyxsxssxyssxyxsxssxyssxyssysxys...
sage: fix.length()
+Infinity
```

```
>>> from sage.all import *
>>> fix = m.fixed_point('x')
>>> fix
word: xyxsxssysxyxsxssxyssxyxsxssxyssxyssysxys...
>>> fix.length()
+Infinity
```

Incidence matrix:

```
sage: matrix(m)                                                          #␣
↪needs sage.modules
[2 3 1]
[1 3 0]
[1 1 1]
```

```
>>> from sage.all import *
>>> matrix(m)                                                                   #␣
↪needs sage.modules
[2 3 1]
[1 3 0]
[1 1 1]
```

Many other functionalities…:

```
sage: m.is_identity()
False
sage: m.is_endomorphism()
True
```

```
>>> from sage.all import *
>>> m.is_identity()
False
>>> m.is_endomorphism()
True
```

class sage.combinat.words.morphism.**PeriodicPointIterator**(*m*, *cycle*)

Bases: object

(Lazy) constructor of the periodic points of a word morphism.

This class is mainly used in *WordMorphism.periodic_point* and *WordMorphism.periodic_points*.

EXAMPLES:

```
sage: from sage.combinat.words.morphism import PeriodicPointIterator
sage: s = WordMorphism('a->bacca,b->cba,c->aab')
sage: p = PeriodicPointIterator(s, ['a','b','c'])
sage: p._cache[0]
lazy list ['a', 'a', 'b', ...]
sage: p._cache[1]
lazy list ['b', 'a', 'c', ...]
sage: p._cache[2]
lazy list ['c', 'b', 'a', ...]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.morphism import PeriodicPointIterator
>>> s = WordMorphism('a->bacca,b->cba,c->aab')
>>> p = PeriodicPointIterator(s, ['a','b','c'])
>>> p._cache[Integer(0)]
lazy list ['a', 'a', 'b', ...]
>>> p._cache[Integer(1)]
lazy list ['b', 'a', 'c', ...]
>>> p._cache[Integer(2)]
lazy list ['c', 'b', 'a', ...]
```

get_iterator(*i*)

Internal method.

EXAMPLES:

```
sage: from sage.combinat.words.morphism import PeriodicPointIterator
sage: s = WordMorphism('a->bacca,b->cba,c->aab')
sage: p = PeriodicPointIterator(s, ['a','b','c'])
sage: p.get_iterator(0)
<generator object ...get_iterator at ...>
```

```
>>> from sage.all import *
>>> from sage.combinat.words.morphism import PeriodicPointIterator
>>> s = WordMorphism('a->bacca,b->cba,c->aab')
>>> p = PeriodicPointIterator(s, ['a','b','c'])
>>> p.get_iterator(Integer(0))
<generator object ...get_iterator at ...>
```

class sage.combinat.words.morphism.**WordMorphism**(*data*, *domain=None*, *codomain=None*)

Bases: SageObject

WordMorphism class.

INPUT:

- `data` – dictionary or string or an instance of WordMorphism, the map giving the image of letters
- `domain` – (optional:`None`) set of words over a given alphabet. If `None`, the domain alphabet is computed from `data` and is *sorted*.
- `codomain` – (optional:`None`) set of words over a given alphabet. If `None`, the codomain alphabet is computed from `data` and is *sorted*.

Note

When the domain or the codomain are not explicitly given, it is expected that the letters are comparable because the alphabets of the domain and of the codomain are sorted.

EXAMPLES:

From a dictionary:

```
sage: n = WordMorphism({0:[0,2,2,1],1:[0,2],2:[2,2,1]})
sage: n
WordMorphism: 0->0221, 1->02, 2->221
```

```
>>> from sage.all import *
>>> n = WordMorphism({Integer(0):[Integer(0),Integer(2),Integer(2),Integer(1)],
↪Integer(1):[Integer(0),Integer(2)],Integer(2):[Integer(2),Integer(2),
↪Integer(1)]})
>>> n
WordMorphism: 0->0221, 1->02, 2->221
```

From a string with `'->'` as separation:

```
sage: m = WordMorphism('x->xyxsxss,s->xyss,y->ys')
sage: m
WordMorphism: s->xyss, x->xyxsxss, y->ys
sage: m.domain()
```

(continues on next page)

(continued from previous page)

```
Finite words over {'s', 'x', 'y'}
sage: m.codomain()
Finite words over {'s', 'x', 'y'}
```

```
>>> from sage.all import *
>>> m = WordMorphism('x->xyxsxss,s->xyss,y->ys')
>>> m
WordMorphism: s->xyss, x->xyxsxss, y->ys
>>> m.domain()
Finite words over {'s', 'x', 'y'}
>>> m.codomain()
Finite words over {'s', 'x', 'y'}
```

Specifying the domain and codomain:

```
sage: W = FiniteWords([0,1,2])
sage: d = {0:[0,1], 1:[0,1,0], 2:[0]}
sage: m = WordMorphism(d, domain=W, codomain=W)
sage: m([0]).parent()
Finite words over {0, 1, 2}
```

```
>>> from sage.all import *
>>> W = FiniteWords([Integer(0),Integer(1),Integer(2)])
>>> d = {Integer(0):[Integer(0),Integer(1)], Integer(1):[Integer(0),Integer(1),
↪Integer(0)], Integer(2):[Integer(0)]}
>>> m = WordMorphism(d, domain=W, codomain=W)
>>> m([Integer(0)]).parent()
Finite words over {0, 1, 2}
```

When the alphabet is non-sortable, the domain and/or codomain must be explicitly given:

```
sage: W = FiniteWords(['a',6])
sage: d = {'a':['a',6,'a'],6:[6,6,6,'a']}
sage: WordMorphism(d, domain=W, codomain=W)
WordMorphism: 6->666a, a->a6a
```

```
>>> from sage.all import *
>>> W = FiniteWords(['a',Integer(6)])
>>> d = {'a':['a',Integer(6),'a'],Integer(6):[Integer(6),Integer(6),Integer(6),'a
↪']}
>>> WordMorphism(d, domain=W, codomain=W)
WordMorphism: 6->666a, a->a6a
```

abelian_rotation_subspace()

Return the subspace on which the incidence matrix of `self` acts by roots of unity.

EXAMPLES:

```
sage: # needs sage.modules
sage: WordMorphism('0->1,1->0').abelian_rotation_subspace()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
```

(continues on next page)

(continued from previous page)

```
[1 0]
[0 1]
sage: WordMorphism('0->01,1->10').abelian_rotation_subspace()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: WordMorphism('0->01,1->1').abelian_rotation_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
sage: WordMorphism('1->122,2->211').abelian_rotation_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1]
sage: WordMorphism('0->1,1->102,2->3,3->4,4->2').abelian_rotation_subspace()
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> WordMorphism('0->1,1->0').abelian_rotation_subspace()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
>>> WordMorphism('0->01,1->10').abelian_rotation_subspace()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
>>> WordMorphism('0->01,1->1').abelian_rotation_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
>>> WordMorphism('1->122,2->211').abelian_rotation_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1]
>>> WordMorphism('0->1,1->102,2->3,3->4,4->2').abelian_rotation_subspace()
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

The domain needs to be equal to the codomain:

```
sage: WordMorphism('0->1,1->',codomain=Words('01')).abelian_rotation_
↪subspace()          # needs sage.modules
Vector space of degree 2 and dimension 0 over Rational Field
```

(continues on next page)

(continued from previous page)

```
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> WordMorphism('0->1,1->',codomain=Words('01')).abelian_rotation_subspace()␣
↪        # needs sage.modules
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

codomain()

Return the codomain of `self`.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').codomain()
Finite words over {'a', 'b'}
sage: WordMorphism('6->ab,y->5,0->asd').codomain()
Finite words over {'5', 'a', 'b', 'd', 's'}
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->a').codomain()
Finite words over {'a', 'b'}
>>> WordMorphism('6->ab,y->5,0->asd').codomain()
Finite words over {'5', 'a', 'b', 'd', 's'}
```

conjugate(*pos*)

Return the morphism where the image of the letter by `self` is conjugated of parameter `pos`.

INPUT:

- `pos` – integer

EXAMPLES:

```
sage: m = WordMorphism('a->abcde')
sage: m.conjugate(0) == m
True
sage: m.conjugate(1)
WordMorphism: a->bcdea
sage: m.conjugate(3)
WordMorphism: a->deabc
sage: WordMorphism('').conjugate(4)
WordMorphism:
sage: m = WordMorphism('a->abcde,b->xyz')
sage: m.conjugate(2)
WordMorphism: a->cdeab, b->zxy
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->abcde')
>>> m.conjugate(Integer(0)) == m
True
>>> m.conjugate(Integer(1))
```

(continues on next page)

(continued from previous page)

```
WordMorphism: a->bcdea
>>> m.conjugate(Integer(3))
WordMorphism: a->deabc
>>> WordMorphism('').conjugate(Integer(4))
WordMorphism:
>>> m = WordMorphism('a->abcde,b->xyz')
>>> m.conjugate(Integer(2))
WordMorphism: a->cdeab, b->zxy
```

domain()

Return domain of `self`.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').domain()
Finite words over {'a', 'b'}
sage: WordMorphism('b->ba,a->ab').domain()
Finite words over {'a', 'b'}
sage: WordMorphism('6->ab,y->5,0->asd').domain()
Finite words over {'0', '6', 'y'}
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->a').domain()
Finite words over {'a', 'b'}
>>> WordMorphism('b->ba,a->ab').domain()
Finite words over {'a', 'b'}
>>> WordMorphism('6->ab,y->5,0->asd').domain()
Finite words over {'0', '6', 'y'}
```

dual_map(*k=1*)

Return the dual map E_k^* of `self` (see [1]).

Note: It is actually implemented only for $k = 1$.

INPUT:

- `self` – unimodular endomorphism defined on integers `1, 2, \ldots, d`
- `k` – integer (default: 1)

OUTPUT: an instance of E1Star - the dual map

EXAMPLES:

```
sage: sigma = WordMorphism({1: [2], 2: [3], 3: [1,2]})
sage: sigma.dual_map()                                                  #␣
↪needs sage.modules
E_1^*(1->2, 2->3, 3->12)
```

```
>>> from sage.all import *
>>> sigma = WordMorphism({Integer(1): [Integer(2)], Integer(2): [Integer(3)],␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(3): [Integer(1),Integer(2)]})
>>> sigma.dual_map()                                                            #␣
↪needs sage.modules
E_1^*(1->2, 2->3, 3->12)
```

```
sage: sigma.dual_map(k=2)
Traceback (most recent call last):
...
NotImplementedError: the dual map E_k^* is implemented only for k = 1 (not 2)
```

```
>>> from sage.all import *
>>> sigma.dual_map(k=Integer(2))
Traceback (most recent call last):
...
NotImplementedError: the dual map E_k^* is implemented only for k = 1 (not 2)
```

REFERENCES:

- [1] Sano, Y., Arnoux, P. and Ito, S., Higher dimensional extensions of substitutions and their dual maps, Journal d'Analyse Mathématique 83 (2001), 183-206.

extend_by(*other*)

Return `self` extended by `other`.

Let $\varphi_1 : A^* \to B^*$ and $\varphi_2 : C^* \to D^*$ be two morphisms. A morphism $\mu : (A \cup C)^* \to (B \cup D)^*$ corresponds to φ_1 *extended by* φ_2 if $\mu(a) = \varphi_1(a)$ if $a \in A$ and $\mu(a) = \varphi_2(a)$ otherwise.

INPUT:

- `other` – a WordMorphism

OUTPUT: WordMorphism

EXAMPLES:

```
sage: m = WordMorphism('a->ab,b->ba')
sage: n = WordMorphism({'0':'1','1':'0','a':'5'})
sage: m.extend_by(n)
WordMorphism: 0->1, 1->0, a->ab, b->ba
sage: n.extend_by(m)
WordMorphism: 0->1, 1->0, a->5, b->ba
sage: m.extend_by(m)
WordMorphism: a->ab, b->ba
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->ab,b->ba')
>>> n = WordMorphism({'0':'1','1':'0','a':'5'})
>>> m.extend_by(n)
WordMorphism: 0->1, 1->0, a->ab, b->ba
>>> n.extend_by(m)
WordMorphism: 0->1, 1->0, a->5, b->ba
>>> m.extend_by(m)
WordMorphism: a->ab, b->ba
```

fixed_point(*letter*)

Return the fixed point of `self` beginning by the given `letter`.

A fixed point of morphism φ is a word w such that $\varphi(w) = w$.

INPUT:

- `self` – an endomorphism (or more generally a self-composable morphism), must be prolongable on `letter`
- `letter` – in the domain of `self`, the first letter of the fixed point

OUTPUT: `word` – the fixed point of `self` beginning with `letter`

EXAMPLES:

```
sage: W = FiniteWords('abc')
```

```
>>> from sage.all import *
>>> W = FiniteWords('abc')
```

1. Infinite fixed point:

```
sage: WordMorphism('a->ab,b->ba').fixed_point(letter='a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: WordMorphism('a->ab,b->a').fixed_point(letter='a')
word: abaababaabaababaababaabaababaabaababaaba...
sage: WordMorphism('a->ab,b->b,c->ba', codomain=W).fixed_point(letter='a')
word: abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->ba').fixed_point(letter='a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
>>> WordMorphism('a->ab,b->a').fixed_point(letter='a')
word: abaababaabaababaababaabaababaabaababaaba...
>>> WordMorphism('a->ab,b->b,c->ba', codomain=W).fixed_point(letter='a')
word: abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...
```

2. Infinite fixed point of an erasing morphism:

```
sage: WordMorphism('a->ab,b->,c->ba', codomain=W).fixed_point(letter='a')
word: ab
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->,c->ba', codomain=W).fixed_point(letter='a')
word: ab
```

3. Finite fixed point:

```
sage: WordMorphism('a->ab,b->b,c->ba', codomain=W).fixed_point(letter='b')
word: b
sage: _.parent()
Finite words over {'a', 'b', 'c'}

sage: WordMorphism('a->ab,b->cc,c->', codomain=W).fixed_point(letter='a')
```

(continues on next page)

(continued from previous page)

```
word: abcc
sage: _.parent()
Finite words over {'a', 'b', 'c'}

sage: m = WordMorphism('a->abc,b->,c->')
sage: fp = m.fixed_point('a'); fp
word: abc

sage: m = WordMorphism('a->ba,b->')
sage: m('ba')
word: ba
sage: m.fixed_point('a') #todo: not implemented
word: ba
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->b,c->ba', codomain=W).fixed_point(letter='b')
word: b
>>> _.parent()
Finite words over {'a', 'b', 'c'}

>>> WordMorphism('a->ab,b->cc,c->', codomain=W).fixed_point(letter='a')
word: abcc
>>> _.parent()
Finite words over {'a', 'b', 'c'}

>>> m = WordMorphism('a->abc,b->,c->')
>>> fp = m.fixed_point('a'); fp
word: abc

>>> m = WordMorphism('a->ba,b->')
>>> m('ba')
word: ba
>>> m.fixed_point('a') #todo: not implemented
word: ba
```

5. Fixed point of a power of a morphism:

```
sage: m = WordMorphism('a->ba,b->ab')
sage: (m^2).fixed_point(letter='a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->ba,b->ab')
>>> (m**Integer(2)).fixed_point(letter='a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

6. With a self-composable but not endomorphism

   sage: m = WordMorphism('a->cbc,b->bc,c->b') sage: m.is_endomorphism() False sage: m.fixed_point('b') word: bcbbcbcbbcbbcbcbbcbcbbcbbcbcbbcbbcbcbbcb…

fixed_points()

Return the list of all fixed points of `self`.

EXAMPLES:

```
sage: f = WordMorphism('a->ab,b->ba')
sage: for w in f.fixed_points(): print(w)
abbabaabbaababbabaababbaabbabaabbaababba...
baababbaabbabaababbabaabbaababbaabbabaab...

sage: f = WordMorphism('a->ab,b->c,c->a')
sage: for w in f.fixed_points(): print(w)
abcaababcabcaabcaababcaababcabcaababcabc...

sage: f = WordMorphism('a->ab,b->cab,c->bcc')
sage: for w in f.fixed_points(): print(w)
abcabbccabcabcabbccbccabcabbccabcabbccab...
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->ab,b->ba')
>>> for w in f.fixed_points(): print(w)
abbabaabbaababbabaababbaabbabaabbaababba...
baababbaabbabaababbabaabbaababbaabbabaab...

>>> f = WordMorphism('a->ab,b->c,c->a')
>>> for w in f.fixed_points(): print(w)
abcaababcabcaabcaababcaababcabcaababcabc...

>>> f = WordMorphism('a->ab,b->cab,c->bcc')
>>> for w in f.fixed_points(): print(w)
abcabbccabcabcabbccbccabcabbccabcabbccab...
```

This shows that issue Issue #13668 has been resolved:

```
sage: d = {1:[1,2],2:[2,3],3:[4],4:[5],5:[6],6:[7],7:[8],8:[9],9:[10],10:[1]}
sage: s = WordMorphism(d)
sage: s7 = s^7
sage: s7.fixed_points()
[word: 12232342..., word: 2,3,4,5,6,7,8...]
sage: s7r = s7.reversal()
sage: s7r.periodic_point(2)
word: 2,1,1,10,9,8,7,6,5,4,3,2,1,10,9,8,7,6,5,4,3,2,10,9,8,7,6,5,4,3,2,9,8,7,
↪6,5,4,3,2,8,...
```

```
>>> from sage.all import *
>>> d = {Integer(1):[Integer(1),Integer(2)],Integer(2):[Integer(2),
↪Integer(3)],Integer(3):[Integer(4)],Integer(4):[Integer(5)],
↪Integer(5):[Integer(6)],Integer(6):[Integer(7)],Integer(7):[Integer(8)],
↪Integer(8):[Integer(9)],Integer(9):[Integer(10)],Integer(10):[Integer(1)]}
>>> s = WordMorphism(d)
>>> s7 = s**Integer(7)
>>> s7.fixed_points()
[word: 12232342..., word: 2,3,4,5,6,7,8...]
```

(continues on next page)

(continued from previous page)

```
>>> s7r = s7.reversal()
>>> s7r.periodic_point(Integer(2))
word: 2,1,1,10,9,8,7,6,5,4,3,2,1,10,9,8,7,6,5,4,3,2,10,9,8,7,6,5,4,3,2,9,8,7,
↪6,5,4,3,2,8,...
```

This shows that issue Issue #13668 has been resolved:

```
sage: s = "1->321331332133133,2->133321331332133133,3->
↪2133133133321331332133133"
sage: s = WordMorphism(s)
sage: (s^2).fixed_points()
[]
```

```
>>> from sage.all import *
>>> s = "1->321331332133133,2->133321331332133133,3->2133133133321331332133133
↪"
>>> s = WordMorphism(s)
>>> (s**Integer(2)).fixed_points()
[]
```

growing_letters()

Return the list of growing letters.

See *is_growing()* for more information.

EXAMPLES:

```
sage: WordMorphism('0->01,1->10').growing_letters()
['0', '1']
sage: WordMorphism('0->01,1->1').growing_letters()
['0']
sage: WordMorphism('0->01,1->0,2->1',codomain=Words('012')).growing_letters()
['0', '1', '2']
sage: WordMorphism('a->b,b->a').growing_letters()
[]
sage: WordMorphism('a->b,b->c,c->d,d->c', codomain=Words('abcd')).growing_
↪letters()
[]
```

```
>>> from sage.all import *
>>> WordMorphism('0->01,1->10').growing_letters()
['0', '1']
>>> WordMorphism('0->01,1->1').growing_letters()
['0']
>>> WordMorphism('0->01,1->0,2->1',codomain=Words('012')).growing_letters()
['0', '1', '2']
>>> WordMorphism('a->b,b->a').growing_letters()
[]
>>> WordMorphism('a->b,b->c,c->d,d->c', codomain=Words('abcd')).growing_
↪letters()
[]
```

has_conjugate_in_classP(*f=None*)

Return `True` if `self` has a conjugate in class f-P.

DEFINITION : Let A be an alphabet. We say that a primitive substitution S is in the *class P* if there exists a palindrome p and for each $b \in A$ a palindrome q_b such that $S(b) = pq_b$ for all $b \in A$. [1]

Let f be an involution on A. We say that a morphism φ is in class f-P if there exists an f-palindrome p and for each $\alpha \in A$ there exists an f-palindrome q_α such that $\varphi(\alpha) = pq_\alpha$. [2]

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`; it must be callable on letters as well as words (e.g. WordMorphism)

REFERENCES:

- [1] Hof, A., O. Knill et B. Simon, Singular continuous spectrum for palindromic Schrödinger operators, Commun. Math. Phys. 174 (1995) 149-159.
- [2] Labbe, Sebastien. Proprietes combinatoires des f-palindromes, Memoire de maitrise en Mathematiques, Montreal, UQAM, 2008, 109 pages.

EXAMPLES:

```
sage: fibo = WordMorphism('a->ab,b->a')
sage: fibo.has_conjugate_in_classP()
True
sage: (fibo^2).is_in_classP()
False
sage: (fibo^2).has_conjugate_in_classP()
True
```

```
>>> from sage.all import *
>>> fibo = WordMorphism('a->ab,b->a')
>>> fibo.has_conjugate_in_classP()
True
>>> (fibo**Integer(2)).is_in_classP()
False
>>> (fibo**Integer(2)).has_conjugate_in_classP()
True
```

has_left_conjugate()

Return `True` if all the non empty images of `self` begins with the same letter.

EXAMPLES:

```
sage: m = WordMorphism('a->abcde,b->xyz')
sage: m.has_left_conjugate()
False
sage: WordMorphism('b->xyz').has_left_conjugate()
True
sage: WordMorphism('').has_left_conjugate()
True
sage: WordMorphism('a->,b->xyz').has_left_conjugate()
True
sage: WordMorphism('a->abbab,b->abb').has_left_conjugate()
True
```

(continues on next page)

(continued from previous page)

```
sage: WordMorphism('a->abbab,b->abb,c->').has_left_conjugate()
True
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->abcde,b->xyz')
>>> m.has_left_conjugate()
False
>>> WordMorphism('b->xyz').has_left_conjugate()
True
>>> WordMorphism('').has_left_conjugate()
True
>>> WordMorphism('a->,b->xyz').has_left_conjugate()
True
>>> WordMorphism('a->abbab,b->abb').has_left_conjugate()
True
>>> WordMorphism('a->abbab,b->abb,c->').has_left_conjugate()
True
```

has_right_conjugate()

Return `True` if all the non empty images of `self` ends with the same letter.

EXAMPLES:

```
sage: m = WordMorphism('a->abcde,b->xyz')
sage: m.has_right_conjugate()
False
sage: WordMorphism('b->xyz').has_right_conjugate()
True
sage: WordMorphism('').has_right_conjugate()
True
sage: WordMorphism('a->,b->xyz').has_right_conjugate()
True
sage: WordMorphism('a->abbab,b->abb').has_right_conjugate()
True
sage: WordMorphism('a->abbab,b->abb,c->').has_right_conjugate()
True
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->abcde,b->xyz')
>>> m.has_right_conjugate()
False
>>> WordMorphism('b->xyz').has_right_conjugate()
True
>>> WordMorphism('').has_right_conjugate()
True
>>> WordMorphism('a->,b->xyz').has_right_conjugate()
True
>>> WordMorphism('a->abbab,b->abb').has_right_conjugate()
True
>>> WordMorphism('a->abbab,b->abb,c->').has_right_conjugate()
True
```

image(*letter*)

Return the image of a letter.

INPUT:

- `letter` – a letter in the domain alphabet

OUTPUT: word

Note: The letter is assumed to be in the domain alphabet (no check done). Hence, this method is faster than the `__call__` method suitable for words input.

EXAMPLES:

```
sage: m = WordMorphism('a->ab,b->ac,c->a')
sage: m.image('b')
word: ac
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->ab,b->ac,c->a')
>>> m.image('b')
word: ac
```

```
sage: s = WordMorphism({('a', 1):[('a', 1), ('a', 2)], ('a', 2):[('a', 1)]})
sage: s.image(('a',1))
word: ('a', 1),('a', 2)
```

```
>>> from sage.all import *
>>> s = WordMorphism({('a', Integer(1)):[('a', Integer(1)), ('a',␣
↪Integer(2))], ('a', Integer(2)):[('a', Integer(1))]})
>>> s.image(('a',Integer(1)))
word: ('a', 1),('a', 2)
```

```
sage: s = WordMorphism({'b':[1,2], 'a':(2,3,4), 'z':[9,8,7]})
sage: s.image('b')
word: 12
sage: s.image('a')
word: 234
sage: s.image('z')
word: 987
```

```
>>> from sage.all import *
>>> s = WordMorphism({'b':[Integer(1),Integer(2)], 'a':(Integer(2),Integer(3),
↪Integer(4)), 'z':[Integer(9),Integer(8),Integer(7)]})
>>> s.image('b')
word: 12
>>> s.image('a')
word: 234
>>> s.image('z')
word: 987
```

images()

Return the list of all the images of the letters of the alphabet under `self`.

EXAMPLES:

```
sage: sorted(WordMorphism('a->ab,b->a').images())
[word: a, word: ab]
sage: sorted(WordMorphism('6->ab,y->5,0->asd').images())
[word: 5, word: ab, word: asd]
```

```
>>> from sage.all import *
>>> sorted(WordMorphism('a->ab,b->a').images())
[word: a, word: ab]
>>> sorted(WordMorphism('6->ab,y->5,0->asd').images())
[word: 5, word: ab, word: asd]
```

immortal_letters()

Return the list of immortal letters.

A letter a is *immortal* for the morphism s if the length of the iterates of $|s^n(a)|$ is larger than zero as n goes to infinity.

Requires this morphism to be self-composable.

EXAMPLES:

```
sage: WordMorphism('a->a').immortal_letters()
['a']
sage: WordMorphism('a->b,b->a').immortal_letters()
['a', 'b']
sage: WordMorphism('a->abcd,b->cd,c->dd,d->').immortal_letters()
['a']
sage: WordMorphism('a->bc,b->cac,c->de,d->,e->').immortal_letters()
['a', 'b']
sage: WordMorphism('a->', domain=Words('a'), codomain=Words('a')).immortal_
↪letters()
[]

sage: WordMorphism('a->').immortal_letters()
[]
```

```
>>> from sage.all import *
>>> WordMorphism('a->a').immortal_letters()
['a']
>>> WordMorphism('a->b,b->a').immortal_letters()
['a', 'b']
>>> WordMorphism('a->abcd,b->cd,c->dd,d->').immortal_letters()
['a']
>>> WordMorphism('a->bc,b->cac,c->de,d->,e->').immortal_letters()
['a', 'b']
>>> WordMorphism('a->', domain=Words('a'), codomain=Words('a')).immortal_
↪letters()
[]
```

(continues on next page)

(continued from previous page)

```
>>> WordMorphism('a->').immortal_letters()
[]
```

incidence_matrix()

Return the incidence matrix of the morphism. The order of the rows and column are given by the order defined on the alphabet of the domain and the codomain.

The matrix returned is over the integers. If a different ring is desired, use either the `change_ring` function or the `matrix` function.

EXAMPLES:

```
sage: m = WordMorphism('a->abc,b->a,c->c')
sage: m.incidence_matrix()                                                      #␣
↪needs sage.modules
[1 1 0]
[1 0 0]
[1 0 1]
sage: m = WordMorphism('a->abc,b->a,c->c,d->abbccccabca,e->abc')
sage: m.incidence_matrix()                                                      #␣
↪needs sage.modules
[1 1 0 3 1]
[1 0 0 3 1]
[1 0 1 5 1]
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->abc,b->a,c->c')
>>> m.incidence_matrix()                                                      #␣
↪needs sage.modules
[1 1 0]
[1 0 0]
[1 0 1]
>>> m = WordMorphism('a->abc,b->a,c->c,d->abbccccabca,e->abc')
>>> m.incidence_matrix()                                                      #␣
↪needs sage.modules
[1 1 0 3 1]
[1 0 0 3 1]
[1 0 1 5 1]
```

infinite_repetitions_primitive_roots(*w=None*, *allow_growing=None*)

Return the set of primitive roots (up to conjugacy) of infinite repetitions from the language $\{m^n(w) | n \geq 0\}$, where m is this morphism and w is a word inputted as a parameter.

Requires this morphism to be an endomorphism.

The word v^ω is an infinite repetition (in other words, an infinite periodic factor) of a language, if v is a non-empty word and for each positive integer k the word v^k is a factor of some word from the language. It turns out that a language created by iterating a morphism has a finite number of primitive roots of infinite repetitions.

If v is a primitive root of an infinite repetition, then all its conjugations are also primitive roots of an infinite repetition. For simplicity's sake this method returns only the lexicographically minimal one from each conjugacy class.

INPUT:

- `w` – finite iterable (default: `self.domain().alphabet()`); represents a word used to start the language
- `allow_growing` – boolean or `None` (default: `None`); if `False`, return only the primitive roots that contain no growing letters. If `True`, return only the primitive roots that contain at least one growing letter. If `None`, return both.

ALGORITHM:

The algorithm used is described in detail in [KS2015].

EXAMPLES:

```
sage: m = WordMorphism('a->aba,b->aba,c->cd,d->e,e->d')
sage: inf_reps = m.infinite_repetitions_primitive_roots('ac')
sage: sorted(inf_reps)
[word: aab, word: de]
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->aba,b->aba,c->cd,d->e,e->d')
>>> inf_reps = m.infinite_repetitions_primitive_roots('ac')
>>> sorted(inf_reps)
[word: aab, word: de]
```

`allow_growing` parameter:

```
sage: sorted(m.infinite_repetitions_primitive_roots('ac', True))
[word: aab]
sage: sorted(m.infinite_repetitions_primitive_roots('ac', False))
[word: de]
```

```
>>> from sage.all import *
>>> sorted(m.infinite_repetitions_primitive_roots('ac', True))
[word: aab]
>>> sorted(m.infinite_repetitions_primitive_roots('ac', False))
[word: de]
```

Incomplete check that these words are indeed the primitive roots of infinite repetitions:

```
sage: SL = m._language_naive(10, Word('ac'))
sage: all(x in SL for x in inf_reps)
True
sage: all(x^2 in SL for x in inf_reps)
True
sage: all(x^3 in SL for x in inf_reps)
True
```

```
>>> from sage.all import *
>>> SL = m._language_naive(Integer(10), Word('ac'))
>>> all(x in SL for x in inf_reps)
True
>>> all(x**Integer(2) in SL for x in inf_reps)
True
>>> all(x**Integer(3) in SL for x in inf_reps)
True
```

Large example:

```
sage: m = WordMorphism('a->1b5,b->fcg,c->dae,d->432,e->678,f->f,g->g,1->2,2->
↪3,3->4,4->1,5->6,6->7,7->8,8->5')
sage: sorted(m.infinite_repetitions_primitive_roots('a'))
[word: 1432f2143f3214f4321f, word: 5678g8567g7856g6785g]
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->1b5,b->fcg,c->dae,d->432,e->678,f->f,g->g,1->2,2->3,
↪3->4,4->1,5->6,6->7,7->8,8->5')
>>> sorted(m.infinite_repetitions_primitive_roots('a'))
[word: 1432f2143f3214f4321f, word: 5678g8567g7856g6785g]
```

is_empty()

Return `True` if the cardinality of the domain is zero and `False` otherwise.

EXAMPLES:

```
sage: WordMorphism('').is_empty()
True
sage: WordMorphism('a->a').is_empty()
False
```

```
>>> from sage.all import *
>>> WordMorphism('').is_empty()
True
>>> WordMorphism('a->a').is_empty()
False
```

is_endomorphism()

Return whether `self` is an endomorphism, that is if the domain coincide with the codomain.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').is_endomorphism()
True
sage: WordMorphism('6->ab,y->5,0->asd').is_endomorphism()
False
sage: WordMorphism('a->a,b->aa,c->aaa').is_endomorphism()
False
sage: Wabc = Words('abc')
sage: m = WordMorphism('a->a,b->aa,c->aaa',codomain = Wabc)
sage: m.is_endomorphism()
True
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->a').is_endomorphism()
True
>>> WordMorphism('6->ab,y->5,0->asd').is_endomorphism()
False
>>> WordMorphism('a->a,b->aa,c->aaa').is_endomorphism()
False
>>> Wabc = Words('abc')
>>> m = WordMorphism('a->a,b->aa,c->aaa',codomain = Wabc)
```

(continues on next page)

(continued from previous page)

```
>>> m.is_endomorphism()
True
```

We check that Issue #8674 is fixed:

```
sage: P = WordPaths('abcd')                                              #␣
↪needs sage.modules
sage: m = WordMorphism('a->adab,b->ab,c->cbcd,d->cd',                    #␣
↪needs sage.modules
....:                  domain=P, codomain=P)
sage: m.is_endomorphism()                                                #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> P = WordPaths('abcd')                                              #␣
↪needs sage.modules
>>> m = WordMorphism('a->adab,b->ab,c->cbcd,d->cd',                    #␣
↪needs sage.modules
...                  domain=P, codomain=P)
>>> m.is_endomorphism()                                                #␣
↪needs sage.modules
True
```

is_erasing()

Return `True` if `self` is an erasing morphism, i.e. the image of a letter is the empty word.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').is_erasing()
False
sage: WordMorphism('6->ab,y->5,0->asd').is_erasing()
False
sage: WordMorphism('6->ab,y->5,0->asd,7->').is_erasing()
True
sage: WordMorphism('').is_erasing()
False
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->a').is_erasing()
False
>>> WordMorphism('6->ab,y->5,0->asd').is_erasing()
False
>>> WordMorphism('6->ab,y->5,0->asd,7->').is_erasing()
True
>>> WordMorphism('').is_erasing()
False
```

is_growing(*letter=None*)

Return `True` if `letter` is a growing letter.

A letter a is *growing* for the morphism s if the length of the iterates of $|s^n(a)|$ tend to infinity as n goes to infinity.

INPUT:

- `letter` – `None` or a letter in the domain of `self`

Note

If letter is `None`, this returns `True` if `self` is everywhere growing, i.e., all letters are growing letters (see [CassNic10]), and that `self` **must** be an endomorphism.

EXAMPLES:

```
sage: WordMorphism('0->01,1->1').is_growing('0')
True
sage: WordMorphism('0->01,1->1').is_growing('1')
False
sage: WordMorphism('0->01,1->10').is_growing()
True
sage: WordMorphism('0->1,1->2,2->01').is_growing()
True
sage: WordMorphism('0->01,1->1').is_growing()
False
```

```
>>> from sage.all import *
>>> WordMorphism('0->01,1->1').is_growing('0')
True
>>> WordMorphism('0->01,1->1').is_growing('1')
False
>>> WordMorphism('0->01,1->10').is_growing()
True
>>> WordMorphism('0->1,1->2,2->01').is_growing()
True
>>> WordMorphism('0->01,1->1').is_growing()
False
```

The domain needs to be equal to the codomain:

```
sage: WordMorphism('0->01,1->0,2->1',codomain=Words('012')).is_growing()
True
```

```
>>> from sage.all import *
>>> WordMorphism('0->01,1->0,2->1',codomain=Words('012')).is_growing()
True
```

Test of erasing morphisms:

```
sage: WordMorphism('0->01,1->').is_growing('0')
False
sage: m = WordMorphism('a->bc,b->bcc,c->',codomain=Words('abc'))
sage: m.is_growing('a')
False
sage: m.is_growing('b')
False
```

(continues on next page)

(continued from previous page)

```
sage: m.is_growing('c')
False
```

```
>>> from sage.all import *
>>> WordMorphism('0->01,1->').is_growing('0')
False
>>> m = WordMorphism('a->bc,b->bcc,c->',codomain=Words('abc'))
>>> m.is_growing('a')
False
>>> m.is_growing('b')
False
>>> m.is_growing('c')
False
```

REFERENCES:

is_identity()

Return `True` if `self` is the identity morphism.

EXAMPLES:

```
sage: m = WordMorphism('a->a,b->b,c->c,d->e')
sage: m.is_identity()
False
sage: WordMorphism('a->a,b->b,c->c').is_identity()
True
sage: WordMorphism('a->a,b->b,c->cb').is_identity()
False
sage: m = WordMorphism('a->b,b->c,c->a')
sage: (m^2).is_identity()
False
sage: (m^3).is_identity()
True
sage: (m^4).is_identity()
False
sage: WordMorphism('').is_identity()
True
sage: WordMorphism({0:[0],1:[1]}).is_identity()
True
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->a,b->b,c->c,d->e')
>>> m.is_identity()
False
>>> WordMorphism('a->a,b->b,c->c').is_identity()
True
>>> WordMorphism('a->a,b->b,c->cb').is_identity()
False
>>> m = WordMorphism('a->b,b->c,c->a')
>>> (m**Integer(2)).is_identity()
False
>>> (m**Integer(3)).is_identity()
```

(continues on next page)

(continued from previous page)

```
True
>>> (m**Integer(4)).is_identity()
False
>>> WordMorphism('').is_identity()
True
>>> WordMorphism({Integer(0):[Integer(0)],Integer(1):[Integer(1)]}).is_
↪identity()
True
```

We check that Issue #8618 is fixed:

```
sage: t = WordMorphism({'a1':['a2'], 'a2':['a1']})
sage: (t*t).is_identity()
True
```

```
>>> from sage.all import *
>>> t = WordMorphism({'a1':['a2'], 'a2':['a1']})
>>> (t*t).is_identity()
True
```

is_in_classP(*f=None*)

Return `True` if `self` is in class P (or f-P).

DEFINITION : Let A be an alphabet. We say that a primitive substitution S is in the *class P* if there exists a palindrome p and for each $b \in A$ a palindrome q_b such that $S(b) = pq_b$ for all $b \in A$. [1]

Let f be an involution on A. "We say that a morphism φ is in class f-P if there exists an f-palindrome p and for each $\alpha \in A$ there exists an f-palindrome q_α such that $\varphi(\alpha) = pq_\alpha$. [2]

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`; it must be callable on letters as well as words (e.g. WordMorphism)

REFERENCES:

- [1] Hof, A., O. Knill et B. Simon, Singular continuous spectrum for palindromic Schrödinger operators, Commun. Math. Phys. 174 (1995) 149-159.
- [2] Labbe, Sebastien. Proprietes combinatoires des f-palindromes, Memoire de maitrise en Mathematiques, Montreal, UQAM, 2008, 109 pages.

EXAMPLES:

```
sage: WordMorphism('a->bbaba,b->bba').is_in_classP()
True
sage: tm = WordMorphism('a->ab,b->ba')
sage: tm.is_in_classP()
False
sage: f = WordMorphism('a->b,b->a')
sage: tm.is_in_classP(f=f)
True
sage: (tm^2).is_in_classP()
True
sage: (tm^2).is_in_classP(f=f)
False
```

(continues on next page)

(continued from previous page)

```
sage: fibo = WordMorphism('a->ab,b->a')
sage: fibo.is_in_classP()
True
sage: fibo.is_in_classP(f=f)
False
sage: (fibo^2).is_in_classP()
False
sage: f = WordMorphism('a->b,b->a,c->c')
sage: WordMorphism('a->acbcc,b->acbab,c->acbba').is_in_classP(f)
True
```

```
>>> from sage.all import *
>>> WordMorphism('a->bbaba,b->bba').is_in_classP()
True
>>> tm = WordMorphism('a->ab,b->ba')
>>> tm.is_in_classP()
False
>>> f = WordMorphism('a->b,b->a')
>>> tm.is_in_classP(f=f)
True
>>> (tm**Integer(2)).is_in_classP()
True
>>> (tm**Integer(2)).is_in_classP(f=f)
False
>>> fibo = WordMorphism('a->ab,b->a')
>>> fibo.is_in_classP()
True
>>> fibo.is_in_classP(f=f)
False
>>> (fibo**Integer(2)).is_in_classP()
False
>>> f = WordMorphism('a->b,b->a,c->c')
>>> WordMorphism('a->acbcc,b->acbab,c->acbba').is_in_classP(f)
True
```

is_injective()

Return whether this morphism is injective.

ALGORITHM:

Uses a version of Wikipedia article Sardinas–Patterson_algorithm. Time complexity is on average quadratic with regards to the size of the morphism.

EXAMPLES:

```
sage: WordMorphism('a->0,b->10,c->110,d->111').is_injective()
True
sage: WordMorphism('a->00,b->01,c->012,d->20001').is_injective()
False
```

```
>>> from sage.all import *
>>> WordMorphism('a->0,b->10,c->110,d->111').is_injective()
True
```

(continues on next page)

(continued from previous page)

```
>>> WordMorphism('a->00,b->01,c->012,d->20001').is_injective()
False
```

is_involution()

Return `True` if `self` is an involution, i.e. its square is the identity.

INPUT:

- `self` – an endomorphism

EXAMPLES:

```
sage: WordMorphism('a->b,b->a').is_involution()
True
sage: WordMorphism('a->b,b->ba').is_involution()
False
sage: WordMorphism({0:[1],1:[0]}).is_involution()
True
```

```
>>> from sage.all import *
>>> WordMorphism('a->b,b->a').is_involution()
True
>>> WordMorphism('a->b,b->ba').is_involution()
False
>>> WordMorphism({Integer(0):[Integer(1)],Integer(1):[Integer(0)]}).is_
↪involution()
True
```

is_primitive()

Return `True` if `self` is primitive.

A morphism φ is *primitive* if there exists an positive integer k such that for all $\alpha \in \Sigma$, $\varphi^k(\alpha)$ contains all the letters of Σ.

INPUT:

- `self` – an endomorphism

ALGORITHM:

Exercices 8.7.8, p.281 in [1]: (c) Let $y(M)$ be the least integer e such that M^e has all positive entries. Prove that, for all primitive matrices M, we have $y(M) \leq (d-1)^2 + 1$. (d) Prove that the bound $y(M) \leq (d-1)^2 + 1$ is best possible.

EXAMPLES:

```
sage: tm = WordMorphism('a->ab,b->ba')
sage: tm.is_primitive()                                                    #␣
↪needs sage.modules
True
sage: fibo = WordMorphism('a->ab,b->a')
sage: fibo.is_primitive()                                                  #␣
↪needs sage.modules
True
sage: m = WordMorphism('a->bb,b->aa')
sage: m.is_primitive()                                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
False
sage: f = WordMorphism({0:[1],1:[0]})
sage: f.is_primitive()                                                    #␣
↪needs sage.modules
False
```

```
>>> from sage.all import *
>>> tm = WordMorphism('a->ab,b->ba')
>>> tm.is_primitive()                                                     #␣
↪needs sage.modules
True
>>> fibo = WordMorphism('a->ab,b->a')
>>> fibo.is_primitive()                                                   #␣
↪needs sage.modules
True
>>> m = WordMorphism('a->bb,b->aa')
>>> m.is_primitive()                                                      #␣
↪needs sage.modules
False
>>> f = WordMorphism({Integer(0):[Integer(1)],Integer(1):[Integer(0)]})
>>> f.is_primitive()                                                      #␣
↪needs sage.modules
False
```

```
sage: s = WordMorphism('a->b,b->c,c->ab')
sage: s.is_primitive()                                                    #␣
↪needs sage.modules
True
sage: s = WordMorphism('a->b,b->c,c->d,d->e,e->f,f->g,g->h,h->ab')
sage: s.is_primitive()                                                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> s = WordMorphism('a->b,b->c,c->ab')
>>> s.is_primitive()                                                      #␣
↪needs sage.modules
True
>>> s = WordMorphism('a->b,b->c,c->d,d->e,e->f,f->g,g->h,h->ab')
>>> s.is_primitive()                                                      #␣
↪needs sage.modules
True
```

REFERENCES:

- [1] Jean-Paul Allouche and Jeffrey Shallit, Automatic Sequences: Theory, Applications, Generalizations, Cambridge University Press, 2003.

is_prolongable(*letter*)

Return `True` if `self` is prolongable on `letter`.

A morphism φ is prolongable on a letter a if a is a prefix of $\varphi(a)$.

INPUT:

- `self` – its codomain must be an instance of Words
- `letter` – a letter in the domain alphabet

OUTPUT: boolean

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').is_prolongable(letter='a')
True
sage: WordMorphism('a->ab,b->a').is_prolongable(letter='b')
False
sage: WordMorphism('a->ba,b->ab').is_prolongable(letter='b')
False
sage: (WordMorphism('a->ba,b->ab')^2).is_prolongable(letter='b')
True
sage: WordMorphism('a->ba,b->').is_prolongable(letter='b')
False
sage: WordMorphism('a->bb,b->aac').is_prolongable(letter='a')
False
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->a').is_prolongable(letter='a')
True
>>> WordMorphism('a->ab,b->a').is_prolongable(letter='b')
False
>>> WordMorphism('a->ba,b->ab').is_prolongable(letter='b')
False
>>> (WordMorphism('a->ba,b->ab')**Integer(2)).is_prolongable(letter='b')
True
>>> WordMorphism('a->ba,b->').is_prolongable(letter='b')
False
>>> WordMorphism('a->bb,b->aac').is_prolongable(letter='a')
False
```

We check that Issue #8595 is fixed:

```
sage: s = WordMorphism({('a', 1) : [('a', 1), ('a', 2)], ('a', 2) : [('a',␣
↪1)]})
sage: s.is_prolongable(('a',1))
True
```

```
>>> from sage.all import *
>>> s = WordMorphism({('a', Integer(1)) : [('a', Integer(1)), ('a',␣
↪Integer(2))], ('a', Integer(2)) : [('a', Integer(1))]})
>>> s.is_prolongable(('a',Integer(1)))
True
```

is_pushy(*w=None*)

Return whether the language $\{m^n(w)|n \geq 0\}$ is pushy, where m is this morphism and w is a word inputted as a parameter.

Requires this morphism to be an endomorphism.

A language created by iterating a morphism is pushy, if its words contain an infinite number of factors containing no growing letters. It turns out that this is equivalent to having at least one infinite repetition containing no growing letters.

See *infinite_repetitions_primitive_roots()* and *is_growing()*.

INPUT:

- w – finite iterable (default: `self.domain().alphabet()`); represents a word used to start the language

EXAMPLES:

```
sage: WordMorphism('a->abca,b->bc,c->').is_pushy()
False
sage: WordMorphism('a->abc,b->,c->bcb').is_pushy()
True
```

```
>>> from sage.all import *
>>> WordMorphism('a->abca,b->bc,c->').is_pushy()
False
>>> WordMorphism('a->abc,b->,c->bcb').is_pushy()
True
```

is_repetitive(*w=None*)

Return whether the language $\{m^n(w)|n \geq 0\}$ is repetitive, where m is this morphism and w is a word inputted as a parameter.

Requires this morphism to be an endomorphism.

A language is repetitive, if for each positive integer k there exists a word u such that u^k is a factor of some word of the language.

It turns out that for languages created by iterating a morphism this is equivalent to having at least one infinite repetition (this property is also known as strong repetitiveness).

See *infinite_repetitions_primitive_roots()*.

INPUT:

- w – finite iterable (default: `self.domain().alphabet()`); represents a word used to start the language

EXAMPLES:

This method can be used to check whether a purely morphic word is not k-power free for all positive integers k. For example, the language containing just the Thue-Morse word and its prefixes is not repetitive, since the Thue-Morse word is cube-free:

```
sage: WordMorphism('a->ab,b->ba').is_repetitive('a')
False
```

```
>>> from sage.all import *
>>> WordMorphism('a->ab,b->ba').is_repetitive('a')
False
```

Similarly, the Hanoi word is square-free:

```
sage: WordMorphism('a->aC,A->ac,b->cB,B->cb,c->bA,C->ba').is_repetitive('a')
False
```

```
>>> from sage.all import *
>>> WordMorphism('a->aC,A->ac,b->cB,B->cb,c->bA,C->ba').is_repetitive('a')
False
```

However, this method solves a more general problem, as it can be called on any morphism m and with any word w:

```
sage: WordMorphism('a->c,b->cda,c->a,d->abc').is_repetitive('bd')
True
```

```
>>> from sage.all import *
>>> WordMorphism('a->c,b->cda,c->a,d->abc').is_repetitive('bd')
True
```

is_self_composable()

Return whether the codomain of `self` is contained in the domain.

EXAMPLES:

```
sage: f = WordMorphism('a->a,b->a')
sage: f.is_endomorphism()
False
sage: f.is_self_composable()
True
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->a,b->a')
>>> f.is_endomorphism()
False
>>> f.is_self_composable()
True
```

is_unboundedly_repetitive(*w=None*)

Return whether the language $\{m^n(w)|n \geq 0\}$ is unboundedly repetitive, where m is this morphism and w is a word inputted as a parameter.

Requires this morphism to be an endomorphism.

A language created by iterating a morphism is unboundedly repetitive, if it has at least one infinite repetition containing at least one growing letter.

See *infinite_repetitions_primitive_roots()* and *is_growing()*.

INPUT:

- `w` – finite iterable (default: `self.domain().alphabet()`); represents a word used to start the language

EXAMPLES:

```
sage: WordMorphism('a->abca,b->bc,c->').is_unboundedly_repetitive()
True
sage: WordMorphism('a->abc,b->,c->bcb').is_unboundedly_repetitive()
False
```

```
>>> from sage.all import *
>>> WordMorphism('a->abca,b->bc,c->').is_unboundedly_repetitive()
True
>>> WordMorphism('a->abc,b->,c->bcb').is_unboundedly_repetitive()
False
```

is_uniform(*k=None*)

Return True if self is a k-uniform morphism.

Let k be a positive integer. A morphism ϕ is called k-uniform if for every letter α, we have $|\phi(\alpha)| = k$. In other words, all images have length k. A morphism is called uniform if it is k-uniform for some positive integer k.

INPUT:

- k – positive integer or None. If set to a positive integer, then the function return True if self is k-uniform. If set to None, then the function return True if self is uniform.

EXAMPLES:

```
sage: phi = WordMorphism('a->ab,b->a')
sage: phi.is_uniform()
False
sage: phi.is_uniform(k=1)
False
sage: tau = WordMorphism('a->ab,b->ba')
sage: tau.is_uniform()
True
sage: tau.is_uniform(k=1)
False
sage: tau.is_uniform(k=2)
True
```

```
>>> from sage.all import *
>>> phi = WordMorphism('a->ab,b->a')
>>> phi.is_uniform()
False
>>> phi.is_uniform(k=Integer(1))
False
>>> tau = WordMorphism('a->ab,b->ba')
>>> tau.is_uniform()
True
>>> tau.is_uniform(k=Integer(1))
False
>>> tau.is_uniform(k=Integer(2))
True
```

language(*n*, *u=None*)

Return the words of length n in the language generated by this substitution.

Given a non-erasing substitution s and a word u the DOL-language generated by s and u is the union of the factors of $s^n(u)$ where n is a nonnegative integer.

INPUT:

- n – nonnegative integer; length of the words in the language

- u – a word or None (default: None); if set to None some letter of the alphabet is used

OUTPUT: a Python set

EXAMPLES:

The fibonacci morphism:

```
sage: s = WordMorphism({0: [0,1], 1: [0]})
sage: sorted(s.language(3))                                         #␣
↪needs sage.modules
[word: 001, word: 010, word: 100, word: 101]
sage: len(s.language(1000))                                         #␣
↪needs sage.modules
1001
sage: all(len(s.language(n)) == n+1 for n in range(100))           #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> s = WordMorphism({Integer(0): [Integer(0),Integer(1)], Integer(1):␣
↪[Integer(0)]})
>>> sorted(s.language(Integer(3)))                                  ␣
↪     # needs sage.modules
[word: 001, word: 010, word: 100, word: 101]
>>> len(s.language(Integer(1000)))                                  ␣
↪     # needs sage.modules
1001
>>> all(len(s.language(n)) == n+Integer(1) for n in range(Integer(100)))    ␣
↪              # needs sage.modules
True
```

A growing but non-primitive example. The DOL-languages generated by 0 and 2 are different:

```
sage: s = WordMorphism({0: [0,1], 1: [0], 2: [2,0,2]})

sage: u = s.fixed_point(0)
sage: A0 = u[:200].factor_set(5)                                    #␣
↪needs sage.modules
sage: B0 = s.language(5, [0])                                       #␣
↪needs sage.modules
sage: set(A0) == B0                                                 #␣
↪needs sage.modules
True

sage: v = s.fixed_point(2)
sage: A2 = v[:200].factor_set(5)                                    #␣
↪needs sage.modules
sage: B2 = s.language(5, [2])                                       #␣
↪needs sage.modules
sage: set(A2) == B2                                                 #␣
↪needs sage.modules
True

sage: len(A0), len(A2)                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
(6, 20)
```

```
>>> from sage.all import *
>>> s = WordMorphism({Integer(0): [Integer(0),Integer(1)], Integer(1):␣
↪[Integer(0)], Integer(2): [Integer(2),Integer(0),Integer(2)]})

>>> u = s.fixed_point(Integer(0))
>>> A0 = u[:Integer(200)].factor_set(Integer(5))                        ␣
↪               # needs sage.modules
>>> B0 = s.language(Integer(5), [Integer(0)])                           ␣
↪               # needs sage.modules
>>> set(A0) == B0                                                     #␣
↪needs sage.modules
True

>>> v = s.fixed_point(Integer(2))
>>> A2 = v[:Integer(200)].factor_set(Integer(5))                        ␣
↪               # needs sage.modules
>>> B2 = s.language(Integer(5), [Integer(2)])                           ␣
↪               # needs sage.modules
>>> set(A2) == B2                                                     #␣
↪needs sage.modules
True

>>> len(A0), len(A2)                                                  #␣
↪needs sage.modules
(6, 20)
```

The Chacon transformation (non-primitive):

```
sage: s = WordMorphism({0: [0,0,1,0], 1:[1]})
sage: sorted(s.language(10))                                            #␣
↪needs sage.modules
[word: 0001000101,
 word: 0001010010,
 ...
 word: 1010010001,
 word: 1010010100]
```

```
>>> from sage.all import *
>>> s = WordMorphism({Integer(0): [Integer(0),Integer(0),Integer(1),
↪Integer(0)], Integer(1):[Integer(1)]})
>>> sorted(s.language(Integer(10)))                                     ␣
↪      # needs sage.modules
[word: 0001000101,
 word: 0001010010,
 ...
 word: 1010010001,
 word: 1010010100]
```

latex_layout(*layout=None*)

Get or set the actual latex layout (oneliner vs array).

INPUT:

- `layout` – string (default: `None`); can take one of the following values:
  - `None` – returns the actual latex layout; by default, the layout is `'array'`
  - `'oneliner'` – set the layout to `'oneliner'`
  - `'array'` – set the layout to `'array'`

EXAMPLES:

```
sage: s = WordMorphism('a->ab,b->ba')
sage: s.latex_layout()
'array'
sage: s.latex_layout('oneliner')
sage: s.latex_layout()
'oneliner'
```

```
>>> from sage.all import *
>>> s = WordMorphism('a->ab,b->ba')
>>> s.latex_layout()
'array'
>>> s.latex_layout('oneliner')
>>> s.latex_layout()
'oneliner'
```

letter_growth_types()

Return the mortal, polynomial and exponential growing letters.

The growth of $|s^n(a)|$ as n goes to ∞ is always of the form $\alpha^n n^\beta$ (where α is a Perron number and β an integer).

Without doing any linear algebra three cases can be differentiated: mortal (ultimately empty or $\alpha = 0$); polynomial ($\alpha = 1$); exponential ($\alpha > 1$). This is what is done in this method.

It requires this morphism to be an endomorphism.

OUTPUT:

The output is a 3-tuple of lists (mortal, polynomial, exponential) where:

- `mortal` – list of mortal letters
- `polynomial` – list of lists where `polynomial[i]` is the list of letters with growth n^i
- `exponential` – list of at least exponentionally growing letters

EXAMPLES:

```
sage: s = WordMorphism('a->abc,b->bc,c->c')
sage: mortal, poly, expo = s.letter_growth_types()
sage: mortal
[]
sage: poly
[['c'], ['b'], ['a']]
sage: expo
[]
```

```
>>> from sage.all import *
>>> s = WordMorphism('a->abc,b->bc,c->c')
>>> mortal, poly, expo = s.letter_growth_types()
>>> mortal
[]
>>> poly
[['c'], ['b'], ['a']]
>>> expo
[]
```

When three mortal letters (c, d, and e), and two letters (a, b) are not growing:

```
sage: s = WordMorphism('a->bc,b->cac,c->de,d->,e->')
sage: s^20
WordMorphism: a->cacde, b->debcde, c->, d->, e->
sage: mortal, poly, expo = s.letter_growth_types()
sage: mortal
['c', 'd', 'e']
sage: poly
[['a', 'b']]
sage: expo
[]
```

```
>>> from sage.all import *
>>> s = WordMorphism('a->bc,b->cac,c->de,d->,e->')
>>> s**Integer(20)
WordMorphism: a->cacde, b->debcde, c->, d->, e->
>>> mortal, poly, expo = s.letter_growth_types()
>>> mortal
['c', 'd', 'e']
>>> poly
[['a', 'b']]
>>> expo
[]
```

```
sage: s = WordMorphism('a->abcd,b->bc,c->c,d->a')
sage: mortal, poly, expo = s.letter_growth_types()
sage: mortal
[]
sage: poly
[['c'], ['b']]
sage: expo
['a', 'd']
```

```
>>> from sage.all import *
>>> s = WordMorphism('a->abcd,b->bc,c->c,d->a')
>>> mortal, poly, expo = s.letter_growth_types()
>>> mortal
[]
>>> poly
[['c'], ['b']]
>>> expo
```

(continues on next page)

(continued from previous page)

```
['a', 'd']
```

list_of_conjugates()

Return the list of all the conjugate morphisms of `self`.

DEFINITION:

Recall from Lothaire [1] (Section 2.3.4) that φ is *right conjugate* of φ', noted $\varphi \triangleleft \varphi'$, if there exists $u \in \Sigma^*$ such that

$$\varphi(\alpha)u = u\varphi'(\alpha),$$

for all $\alpha \in \Sigma$, or equivalently that $\varphi(x)u = u\varphi'(x)$, for all words $x \in \Sigma^*$. Clearly, this relation is not symmetric so that we say that two morphisms φ and φ' are *conjugate*, noted $\varphi \bowtie \varphi'$, if $\varphi \triangleleft \varphi'$ or $\varphi' \triangleleft \varphi$. It is easy to see that conjugacy of morphisms is an equivalence relation.

REFERENCES:

- [1] M. Lothaire, Algebraic Combinatorics on words, Cambridge University Press, 2002.

EXAMPLES:

```
sage: m = WordMorphism('a->abbab,b->abb')
sage: m.list_of_conjugates()
[WordMorphism: a->babba, b->bab,
WordMorphism: a->abbab, b->abb,
WordMorphism: a->bbaba, b->bba,
WordMorphism: a->babab, b->bab,
WordMorphism: a->ababb, b->abb,
WordMorphism: a->babba, b->bba,
WordMorphism: a->abbab, b->bab]
sage: m = WordMorphism('a->aaa,b->aa')
sage: m.list_of_conjugates()
[WordMorphism: a->aaa, b->aa]
sage: WordMorphism('').list_of_conjugates()
[WordMorphism: ]
sage: m = WordMorphism('a->aba,b->aba')
sage: m.list_of_conjugates()
[WordMorphism: a->baa, b->baa,
WordMorphism: a->aab, b->aab,
WordMorphism: a->aba, b->aba]
sage: m = WordMorphism('a->abb,b->abbab,c->')
sage: m.list_of_conjugates()
[WordMorphism: a->bab, b->babba, c->,
WordMorphism: a->abb, b->abbab, c->,
WordMorphism: a->bba, b->bbaba, c->,
WordMorphism: a->bab, b->babab, c->,
WordMorphism: a->abb, b->ababb, c->,
WordMorphism: a->bba, b->babba, c->,
WordMorphism: a->bab, b->abbab, c->]
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->abbab,b->abb')
>>> m.list_of_conjugates()
```

(continues on next page)

(continued from previous page)

```
[WordMorphism: a->babba, b->bab,
WordMorphism: a->abbab, b->abb,
WordMorphism: a->bbaba, b->bba,
WordMorphism: a->babab, b->bab,
WordMorphism: a->ababb, b->abb,
WordMorphism: a->babba, b->bba,
WordMorphism: a->abbab, b->bab]
>>> m = WordMorphism('a->aaa,b->aa')
>>> m.list_of_conjugates()
[WordMorphism: a->aaa, b->aa]
>>> WordMorphism('').list_of_conjugates()
[WordMorphism: ]
>>> m = WordMorphism('a->aba,b->aba')
>>> m.list_of_conjugates()
[WordMorphism: a->baa, b->baa,
WordMorphism: a->aab, b->aab,
WordMorphism: a->aba, b->aba]
>>> m = WordMorphism('a->abb,b->abbab,c->')
>>> m.list_of_conjugates()
[WordMorphism: a->bab, b->babba, c->,
WordMorphism: a->abb, b->abbab, c->,
WordMorphism: a->bba, b->bbaba, c->,
WordMorphism: a->bab, b->babab, c->,
WordMorphism: a->abb, b->ababb, c->,
WordMorphism: a->bba, b->babba, c->,
WordMorphism: a->bab, b->abbab, c->]
```

partition_of_domain_alphabet()

Return a partition of the domain alphabet.

Let $\varphi : \Sigma^* \rightarrow \Sigma^*$ be an involution. There exists a triple of sets (A, B, C) such that

- $A \cup B \cup C = \Sigma$;
- A, B and C are mutually disjoint and
- $\varphi(A) = B$, $\varphi(B) = A$, $\varphi(C) = C$.

These sets are not unique.

INPUT:

- `self` – an involution

OUTPUT: a tuple of three sets

EXAMPLES:

```
sage: m = WordMorphism('a->b,b->a')
sage: m.partition_of_domain_alphabet()   # random ordering
({'a'}, {'b'}, {})
sage: m = WordMorphism('a->b,b->a,c->c')
sage: m.partition_of_domain_alphabet()   # random ordering
({'a'}, {'b'}, {'c'})
sage: m = WordMorphism('a->a,b->b,c->c')
sage: m.partition_of_domain_alphabet()   # random ordering
```

(continues on next page)

(continued from previous page)

```
({}, {}, {'a', 'c', 'b'})
sage: m = WordMorphism('A->T,T->A,C->G,G->C')
sage: m.partition_of_domain_alphabet()   # random ordering
({'A', 'C'}, {'T', 'G'}, {})
sage: I = WordMorphism({0:oo,oo:0,1:-1,-1:1,2:-2,-2:2,3:-3,-3:3})
sage: I.partition_of_domain_alphabet()   # random ordering
({0, -1, -3, -2}, {1, 2, 3, +Infinity}, {})
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->b,b->a')
>>> m.partition_of_domain_alphabet()   # random ordering
({'a'}, {'b'}, {})
>>> m = WordMorphism('a->b,b->a,c->c')
>>> m.partition_of_domain_alphabet()   # random ordering
({'a'}, {'b'}, {'c'})
>>> m = WordMorphism('a->a,b->b,c->c')
>>> m.partition_of_domain_alphabet()   # random ordering
({}, {}, {'a', 'c', 'b'})
>>> m = WordMorphism('A->T,T->A,C->G,G->C')
>>> m.partition_of_domain_alphabet()   # random ordering
({'A', 'C'}, {'T', 'G'}, {})
>>> I = WordMorphism({Integer(0):oo,oo:Integer(0),Integer(1):-Integer(1),-
↪Integer(1):Integer(1),Integer(2):-Integer(2),-Integer(2):Integer(2),
↪Integer(3):-Integer(3),-Integer(3):Integer(3)})
>>> I.partition_of_domain_alphabet()   # random ordering
({0, -1, -3, -2}, {1, 2, 3, +Infinity}, {})
```

periodic_point(*letter*)

Return the periodic point of `self` that starts with `letter`.

EXAMPLES:

```
sage: f = WordMorphism('a->bab,b->ab')
sage: f.periodic_point('a')
word: abbababbababbabababbabababbabababbababbaba...
sage: f.fixed_point('a')
Traceback (most recent call last):
...
TypeError: self must be prolongable on a
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->bab,b->ab')
>>> f.periodic_point('a')
word: abbababbababbabababbabababbabababbababbaba...
>>> f.fixed_point('a')
Traceback (most recent call last):
...
TypeError: self must be prolongable on a
```

Make sure that Issue #31759 is fixed:

```
sage: WordMorphism('a->b,b->a').periodic_point('a')
word: a
```

```
>>> from sage.all import *
>>> WordMorphism('a->b,b->a').periodic_point('a')
word: a
```

periodic_points()

Return the periodic points of `f` as a list of tuples where each tuple is a periodic orbit of `f`.

EXAMPLES:

```
sage: f = WordMorphism('a->aba,b->baa')
sage: for p in f.periodic_points():
....:     print("{} , {}".format(len(p), p[0]))
1 , ababaaababaaabaababaaababaaabaababab...
1 , baaabaabaababaaabaababaaabaababaaababaaa...

sage: f = WordMorphism('a->bab,b->aa')
sage: for p in f.periodic_points():
....:     print("{} , {}".format(len(p), p[0]))
2 , aababaaaababaababbabaababaababbabaababaa...
sage: f.fixed_points()
[]
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->aba,b->baa')
>>> for p in f.periodic_points():
...     print("{} , {}".format(len(p), p[Integer(0)]))
1 , ababaaababaaabaababaaababaaabaababab...
1 , baaabaabaababaaabaababaaabaababaaababaaa...

>>> f = WordMorphism('a->bab,b->aa')
>>> for p in f.periodic_points():
...     print("{} , {}".format(len(p), p[Integer(0)]))
2 , aababaaaababaababbabaababaababbabaababaa...
>>> f.fixed_points()
[]
```

This shows that issue Issue #13668 has been resolved:

```
sage: d = {1:[1,2],2:[2,3],3:[4],4:[5],5:[6],6:[7],7:[8],8:[9],9:[10],10:[1]}
sage: s = WordMorphism(d)
sage: s7 = s^7
sage: s7r = s7.reversal()
sage: for p in s7r.periodic_points(): p
[word: 1,10,9,8,7,6,5,4,3,2,10,9,8,7,6,5,4,3,2,...,
 word: 8765432765432654325432432322176543265432...,
 word: 5,4,3,2,4,3,2,3,2,2,1,4,3,2,3,2,2,1,3,2,...,
 word: 2,1,1,10,9,8,7,6,5,4,3,2,1,10,9,8,7,6,5,...,
 word: 9876543287654327654326543254324323221876...,
 word: 6543254324323221543243232214323221322121...,
 word: 3,2,2,1,2,1,1,10,9,8,7,6,5,4,3,2,2,1,1,1...,
```

(continues on next page)

(continued from previous page)

```
word: 10,9,8,7,6,5,4,3,2,9,8,7,6,5,4,3,2,8,7,6...,
word: 7654326543254324323221654325432432322154...,
word: 4,3,2,3,2,2,1,3,2,2,1,2,1,1,10,9,8,7,6,5...]
```

```
>>> from sage.all import *
>>> d = {Integer(1):[Integer(1),Integer(2)],Integer(2):[Integer(2),
↪Integer(3)],Integer(3):[Integer(4)],Integer(4):[Integer(5)],
↪Integer(5):[Integer(6)],Integer(6):[Integer(7)],Integer(7):[Integer(8)],
↪Integer(8):[Integer(9)],Integer(9):[Integer(10)],Integer(10):[Integer(1)]}
>>> s = WordMorphism(d)
>>> s7 = s**Integer(7)
>>> s7r = s7.reversal()
>>> for p in s7r.periodic_points(): p
[word: 1,10,9,8,7,6,5,4,3,2,10,9,8,7,6,5,4,3,2,...,
 word: 8765432765432654325432432322176543265432...,
 word: 5,4,3,2,4,3,2,3,2,2,1,4,3,2,3,2,2,1,3,2,...,
 word: 2,1,1,10,9,8,7,6,5,4,3,2,1,10,9,8,7,6,5,...,
 word: 9876543287654327654326543254324323221876...,
 word: 6543254324323221543243232214323221322121...,
 word: 3,2,2,1,2,1,1,10,9,8,7,6,5,4,3,2,2,1,1,1...,
 word: 10,9,8,7,6,5,4,3,2,9,8,7,6,5,4,3,2,8,7,6...,
 word: 7654326543254324323221654325432432322154...,
 word: 4,3,2,3,2,2,1,3,2,2,1,2,1,1,10,9,8,7,6,5...]
```

Make sure that Issue #31454 is fixed:

```
sage: WordMorphism('a->a,b->bb').periodic_points()
[[word: bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...]]
```

```
>>> from sage.all import *
>>> WordMorphism('a->a,b->bb').periodic_points()
[[word: bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...]]
```

pisot_eigenvector_left()

Return the left eigenvector of the incidence matrix associated to the largest eigenvalue (in absolute value).

Unicity of the result is guaranteed when the multiplicity of the largest eigenvalue is one, for example when `self` is a Pisot irreducible substitution.

A substitution is Pisot irreducible if the characteristic polynomial of its incidence matrix is irreducible over $\mathbf{Q}$ and has all roots, except one, of modulus strictly smaller than 1.

INPUT:

- `self` – a Pisot irreducible substitution

EXAMPLES:

```
sage: m = WordMorphism('a->aaaabbc,b->aaabbc,c->aabc')
sage: matrix(m)                                                        #␣
↪needs sage.modules
[4 3 2]
[2 2 1]
[1 1 1]
```

(continues on next page)

(continued from previous page)

```
sage: m.pisot_eigenvector_left()                                             #␣
↪needs sage.modules sage.rings.number_field
(1, 0.8392867552141611?, 0.5436890126920763?)
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->aaaabbc,b->aaabbc,c->aabc')
>>> matrix(m)                                                                #␣
↪needs sage.modules
[4 3 2]
[2 2 1]
[1 1 1]
>>> m.pisot_eigenvector_left()                                               #␣
↪needs sage.modules sage.rings.number_field
(1, 0.8392867552141611?, 0.5436890126920763?)
```

pisot_eigenvector_right()

Return the right eigenvector of the incidence matrix associated to the largest eigenvalue (in absolute value).

Unicity of the result is guaranteed when the multiplicity of the largest eigenvalue is one, for example when `self` is a Pisot irreductible substitution.

A substitution is Pisot irreducible if the characteristic polynomial of its incidence matrix is irreducible over **Q** and has all roots, except one, of modulus strictly smaller than 1.

INPUT:

- `self` – a Pisot irreducible substitution

EXAMPLES:

```
sage: m = WordMorphism('a->aaaabbc,b->aaabbc,c->aabc')
sage: matrix(m)                                                              #␣
↪needs sage.modules
[4 3 2]
[2 2 1]
[1 1 1]
sage: m.pisot_eigenvector_right()                                            #␣
↪needs sage.modules sage.rings.number_field
(1, 0.5436890126920763?, 0.2955977425220848?)
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->aaaabbc,b->aaabbc,c->aabc')
>>> matrix(m)                                                                #␣
↪needs sage.modules
[4 3 2]
[2 2 1]
[1 1 1]
>>> m.pisot_eigenvector_right()                                              #␣
↪needs sage.modules sage.rings.number_field
(1, 0.5436890126920763?, 0.2955977425220848?)
```

rauzy_fractal_plot(*n=None*, *exchange=False*, *eig=None*, *translate=None*, *prec=53*, *colormap='hsv'*, *opacity=None*, *plot_origin=None*, *plot_basis=False*, *point_size=None*)

Return a plot of the Rauzy fractal associated with a substitution.

The substitution does not have to be irreducible. The usual definition of a Rauzy fractal requires that its dominant eigenvalue is a Pisot number but the present method doesn't require this, allowing to plot some interesting pictures in the non-Pisot case (see the examples below).

For more details about the definition of the fractal and the projection which is used, see Section 3.1 of [1].

Plots with less than 100,000 points take a few seconds, and several millions of points can be plotted in reasonable time.

Other ways to draw Rauzy fractals (and more generally projections of paths) can be found in *sage.combinat.words.paths.FiniteWordPath_all.plot_projection()* or in *sage.combinat.e_one_star()*.

OUTPUT: a Graphics object

INPUT:

- `n` – integer (default: `None`) The number of points used to plot the fractal. Default values: `1000` for a 1D fractal, `50000` for a 2D fractal, `10000` for a 3D fractal.
- `exchange` – boolean (default: `False`); plot the Rauzy fractal with domain exchange
- `eig` – a real element of `QQbar` of degree >= 2 (default: `None`); the eigenvalue used to plot the fractal. It must be an eigenvalue of `self.incidence_matrix()`. The one used by default the maximal eigenvalue of `self.incidence_matrix()` (usually a Pisot number), but for substitutions with more than 3 letters other interesting choices are sometimes possible.
- `translate` – list of vectors of `RR^size_alphabet`, or a dictionary from the alphabet to lists of vectors (default: `None`). Plot translated copies of the fractal. This option allows to plot tilings easily. The projection used for these vectors is the same as the projection used for the canonical basis to plot the fractal. If the input is a list, all the pieces will be translated and plotted. If the input is a dictionary, each piece will be translated and plotted accordingly to the vectors associated with each letter in the dictionary. Note: by default, the Rauzy fractal placed at the origin is not plotted with the `translate` option; the vector `(0,0,...,0)` has to be added manually.
- `prec` – integer (default: 53); the number of bits used in the floating point representations of the points of the fractal
- `colormap` – color map or dictionary (default: `'hsv'`). It can be one of the following:
  - `string` – a coloring map. For available coloring map names type: `sorted(colormaps)`
  - `dict` – dictionary of the alphabet mapped to colors
- `opacity` – dictionary from the alphabet to the real interval [0,1] (default: `None`); if none is specified, all letters are plotted with opacity `1`
- `plot_origin` – a couple `(k,c)` (default: `None`); if specified, mark the origin by a point of size `k` and color `c`
- `plot_basis` – boolean (default: `False`); plot the projection of the canonical basis with the fractal
- `point_size` – float (default: `None`); the size of the points used to plot the fractal

EXAMPLES:

1. The Rauzy fractal of the Tribonacci substitution:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: s.rauzy_fractal_plot()        # long time                          ␣
↪  # needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->1')
>>> s.rauzy_fractal_plot()         # long time
↪# needs sage.plot
Graphics object consisting of 3 graphics primitives
```

2. The "Hokkaido" fractal. We tweak the plot using the plotting options to get a nice reusable picture, in which we mark the origin by a black dot:

```
sage: s = WordMorphism('a->ab,b->c,c->d,d->e,e->a')
sage: G = s.rauzy_fractal_plot(n=100000, point_size=3,          # not␣
↪tested
....:                          plot_origin=(50,"black"))
sage: G.show(figsize=10, axes=false) # not tested
```

```
>>> from sage.all import *
>>> s = WordMorphism('a->ab,b->c,c->d,d->e,e->a')
>>> G = s.rauzy_fractal_plot(n=Integer(100000), point_size=Integer(3),    ␣
↪      # not tested
...                          plot_origin=(Integer(50),"black"))
>>> G.show(figsize=Integer(10), axes=false) # not tested
```

3. Another "Hokkaido" fractal and its domain exchange:

```
sage: s = WordMorphism({1:[2], 2:[4,3], 3:[4], 4:[5,3], 5:[6], 6:[1]})
sage: s.rauzy_fractal_plot()                                    # not␣
↪tested (> 1 second)
sage: s.rauzy_fractal_plot(exchange=True)                       # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism({Integer(1):[Integer(2)], Integer(2):[Integer(4),
↪Integer(3)], Integer(3):[Integer(4)], Integer(4):[Integer(5),
↪Integer(3)], Integer(5):[Integer(6)], Integer(6):[Integer(1)]})
>>> s.rauzy_fractal_plot()                                    # not␣
↪tested (> 1 second)
>>> s.rauzy_fractal_plot(exchange=True)                       # not␣
↪tested (> 1 second)
```

4. A three-dimensional Rauzy fractal:

```
sage: s = WordMorphism('1->12,2->13,3->14,4->1')
sage: s.rauzy_fractal_plot()                                    # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->14,4->1')
>>> s.rauzy_fractal_plot()                                    # not␣
↪tested (> 1 second)
```

5. A one-dimensional Rauzy fractal (very scattered):

```
sage: s = WordMorphism('1->2122,2->1')
sage: s.rauzy_fractal_plot().show(figsize=20)                        # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->2122,2->1')
>>> s.rauzy_fractal_plot().show(figsize=Integer(20))                      #␣
↪not tested (> 1 second)
```

6. A high resolution plot of a complicated fractal:

```
sage: s = WordMorphism('1->23,2->123,3->1122233')
sage: G = s.rauzy_fractal_plot(n=300000)                             # not␣
↪tested (> 1 second)
sage: G.show(axes=false, figsize=20)                                 # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->23,2->123,3->1122233')
>>> G = s.rauzy_fractal_plot(n=Integer(300000))                           #␣
↪not tested (> 1 second)
>>> G.show(axes=false, figsize=Integer(20))                               #␣
↪not tested (> 1 second)
```

7. A nice colorful animation of a domain exchange:

```
sage: s = WordMorphism('1->21,2->3,3->4,4->25,5->6,6->7,7->1')
sage: L = [s.rauzy_fractal_plot(),                                   # not␣
↪tested (> 1 second)
....:      s.rauzy_fractal_plot(exchange=True)]
sage: animate(L, axes=false).show(delay=100)                         # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->21,2->3,3->4,4->25,5->6,6->7,7->1')
>>> L = [s.rauzy_fractal_plot(),                                   # not␣
↪tested (> 1 second)
...      s.rauzy_fractal_plot(exchange=True)]
>>> animate(L, axes=false).show(delay=Integer(100))                       #␣
↪not tested (> 1 second)
```

8. Plotting with only one color:

```
sage: s = WordMorphism('1->12,2->31,3->1')
sage: cm = {'1':'black', '2':'black', '3':'black'}
sage: s.rauzy_fractal_plot(colormap=cm)                              # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->31,3->1')
>>> cm = {'1':'black', '2':'black', '3':'black'}
>>> s.rauzy_fractal_plot(colormap=cm)                              # not␣
↪tested (> 1 second)
```

9. Different fractals can be obtained by choosing another (non-Pisot) eigenvalue:

```
sage: s = WordMorphism('1->12,2->3,3->45,4->5,5->6,6->7,7->8,8->1')
sage: E = s.incidence_matrix().eigenvalues()                              ␣
↪  # needs sage.modules
sage: x = [x for x in E if -0.8 < x < -0.7][0]                            ␣
↪  # needs sage.modules
sage: s.rauzy_fractal_plot()                                       # not␣
↪tested (> 1 second)
sage: s.rauzy_fractal_plot(eig=x)                                  # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->3,3->45,4->5,5->6,6->7,7->8,8->1')
>>> E = s.incidence_matrix().eigenvalues()
↪# needs sage.modules
>>> x = [x for x in E if -RealNumber('0.8') < x < -RealNumber('0.7
↪')][Integer(0)]                                # needs sage.modules
>>> s.rauzy_fractal_plot()                                         # not␣
↪tested (> 1 second)
>>> s.rauzy_fractal_plot(eig=x)                                    # not␣
↪tested (> 1 second)
```

10. A Pisot reducible substitution with seemingly overlapping tiles:

```
sage: s = WordMorphism({1:[1,2], 2:[2,3], 3:[4], 4:[5], 5:[6],
....:                   6:[7], 7:[8], 8:[9], 9:[10], 10:[1]})
sage: s.rauzy_fractal_plot()                                       # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(2),Integer(3)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(5)], Integer(5):[Integer(6)],
...                   Integer(6):[Integer(7)], Integer(7):[Integer(8)],␣
↪Integer(8):[Integer(9)], Integer(9):[Integer(10)],␣
↪Integer(10):[Integer(1)]})
>>> s.rauzy_fractal_plot()                                         # not␣
↪tested (> 1 second)
```

11. A non-Pisot reducible substitution with a strange Rauzy fractal:

```
sage: s = WordMorphism({1:[3,2], 2:[3,3], 3:[4], 4:[1]})
sage: s.rauzy_fractal_plot()                                       # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism({Integer(1):[Integer(3),Integer(2)],␣
↪Integer(2):[Integer(3),Integer(3)], Integer(3):[Integer(4)],␣
↪Integer(4):[Integer(1)]})
>>> s.rauzy_fractal_plot()                                         # not␣
↪tested (> 1 second)
```

12. A substitution with overlapping tiles. We use the options `colormap` and `opacity` to study how the tiles

overlap:

```
sage: s = WordMorphism('1->213,2->4,3->5,4->1,5->21')
sage: s.rauzy_fractal_plot()                                          # not␣
↪tested (> 1 second)
sage: s.rauzy_fractal_plot(colormap={'1':'red', '4':'purple'})   # not␣
↪tested (> 1 second)
sage: s.rauzy_fractal_plot(n=150000,                                  # not␣
↪tested (> 1 second)
....:                      opacity={'1':0.1,'2':1,'3':0.1,'4':0.1,'5':0.1}
↪)
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->213,2->4,3->5,4->1,5->21')
>>> s.rauzy_fractal_plot()                                          # not␣
↪tested (> 1 second)
>>> s.rauzy_fractal_plot(colormap={'1':'red', '4':'purple'})   # not␣
↪tested (> 1 second)
>>> s.rauzy_fractal_plot(n=Integer(150000),                           #␣
↪not tested (> 1 second)
...                      opacity={'1':RealNumber('0.1'),'2':Integer(1),'3
↪':RealNumber('0.1'),'4':RealNumber('0.1'),'5':RealNumber('0.1')})
```

13. Funny experiments by playing with the precision of the float numbers used to plot the fractal:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: s.rauzy_fractal_plot(prec=6)                                    # not␣
↪tested
sage: s.rauzy_fractal_plot(prec=9)                                    # not␣
↪tested
sage: s.rauzy_fractal_plot(prec=15)                                   # not␣
↪tested
sage: s.rauzy_fractal_plot(prec=19)                                   # not␣
↪tested
sage: s.rauzy_fractal_plot(prec=25)                                   # not␣
↪tested
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->1')
>>> s.rauzy_fractal_plot(prec=Integer(6))                             #␣
↪not tested
>>> s.rauzy_fractal_plot(prec=Integer(9))                             #␣
↪not tested
>>> s.rauzy_fractal_plot(prec=Integer(15))                            #␣
↪not tested
>>> s.rauzy_fractal_plot(prec=Integer(19))                            #␣
↪not tested
>>> s.rauzy_fractal_plot(prec=Integer(25))                            #␣
↪not tested
```

14. Using the `translate` option to plot periodic tilings:

```
sage: s = WordMorphism('1->12,2->13,3->1')
```

(continues on next page)

(continued from previous page)

```
sage: s.rauzy_fractal_plot(n=10000,                                       # not␣
↪tested (> 1 second)
....:                      translate=[(0,0,0),(-1,0,1),(0,-1,1),(1,-1,0),
....:                                 (1,0,-1),(0,1,-1),(-1,1,0)])
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->1')
>>> s.rauzy_fractal_plot(n=Integer(10000),                                  #␣
↪not tested (> 1 second)
...                      translate=[(Integer(0),Integer(0),Integer(0)),(-
↪Integer(1),Integer(0),Integer(1)),(Integer(0),-Integer(1),Integer(1)),
↪(Integer(1),-Integer(1),Integer(0)),
...                                 (Integer(1),Integer(0),-Integer(1)),
↪(Integer(0),Integer(1),-Integer(1)),(-Integer(1),Integer(1),
↪Integer(0))])
```

```
sage: t = WordMorphism("a->aC,b->d,C->de,d->a,e->ab")   # substitution␣
↪found by Julien Bernat
sage: V = [vector((0,0,1,0,-1)), vector((0,0,1,-1,0))]                     ␣
↪  # needs sage.modules
sage: S = set(map(tuple, [i*V[0] + j*V[1]                                  ␣
↪  # needs sage.modules
....:                     for i in [-1,0,1] for j in [-1,0,1]]))
sage: t.rauzy_fractal_plot(n=10000,                                       # not␣
↪tested (> 1 second)
....:                      translate=S, exchange=true)
```

```
>>> from sage.all import *
>>> t = WordMorphism("a->aC,b->d,C->de,d->a,e->ab")   # substitution␣
↪found by Julien Bernat
>>> V = [vector((Integer(0),Integer(0),Integer(1),Integer(0),-
↪Integer(1))), vector((Integer(0),Integer(0),Integer(1),-Integer(1),
↪Integer(0)))]                    # needs sage.modules
>>> S = set(map(tuple, [i*V[Integer(0)] + j*V[Integer(1)]                  ␣
↪                  # needs sage.modules
...                     for i in [-Integer(1),Integer(0),Integer(1)] for␣
↪j in [-Integer(1),Integer(0),Integer(1)]]))
>>> t.rauzy_fractal_plot(n=Integer(10000),                                  #␣
↪not tested (> 1 second)
...                      translate=S, exchange=true)
```

15. Using the `translate` option to plot arbitrary tilings with the fractal pieces. This can be used for example to plot the self-replicating tiling of the Rauzy fractal:

```
sage: s = WordMorphism({1:[1,2], 2:[3], 3:[4,3], 4:[5], 5:[6], 6:[1]})
sage: s.rauzy_fractal_plot()                                              # not␣
↪tested (> 1 second)
sage: D = {1: [(0,0,0,0,0,0), (0,1,0,0,0,0)],
....:      3: [(0,0,0,0,0,0), (0,1,0,0,0,0)], 6: [(0,1,0,0,0,0)]}
sage: s.rauzy_fractal_plot(n=30000, translate=D)                          # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism({Integer(1):[Integer(1),Integer(2)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(4),Integer(3)],␣
↪Integer(4):[Integer(5)], Integer(5):[Integer(6)],␣
↪Integer(6):[Integer(1)]})
>>> s.rauzy_fractal_plot()                                    # not␣
↪tested (> 1 second)
>>> D = {Integer(1): [(Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0)), (Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0))],
...      Integer(3): [(Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0)), (Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0))], Integer(6): [(Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0),Integer(0))]}
>>> s.rauzy_fractal_plot(n=Integer(30000), translate=D)             #␣
↪not tested (> 1 second)
```

16. Plot the projection of the canonical basis with the fractal:

```
sage: s = WordMorphism({1:[2,1], 2:[3], 3:[6,4], 4:[5,1],
....:                   5:[6], 6:[7], 7:[8], 8:[9], 9:[1]})
sage: s.rauzy_fractal_plot(plot_basis=True)                     # not␣
↪tested (> 1 second)
```

```
>>> from sage.all import *
>>> s = WordMorphism({Integer(1):[Integer(2),Integer(1)],␣
↪Integer(2):[Integer(3)], Integer(3):[Integer(6),Integer(4)],␣
↪Integer(4):[Integer(5),Integer(1)],
...                   Integer(5):[Integer(6)], Integer(6):[Integer(7)],␣
↪Integer(7):[Integer(8)], Integer(8):[Integer(9)],␣
↪Integer(9):[Integer(1)]})
>>> s.rauzy_fractal_plot(plot_basis=True)                     # not␣
↪tested (> 1 second)
```

REFERENCES:

- [1] Valerie Berthe and Anne Siegel, Tilings associated with beta-numeration and substitutions, Integers 5 (3), 2005. http://www.integers-ejcnt.org/vol5-3.html

AUTHOR:

Timo Jolivet (2012-06-16)

rauzy_fractal_points(*n=None*, *exchange=False*, *eig=None*, *translate=None*, *prec=53*)

Return a dictionary of list of points associated with the pieces of the Rauzy fractal of `self`.

INPUT:

See the method *rauzy_fractal_plot()* for a description of the options and more examples.

OUTPUT: dictionary of list of points

EXAMPLES:

The Rauzy fractal of the Tribonacci substitution and the number of points in the piece of the fractal associated with `'1'`, `'2'` and `'3'` are respectively:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.rauzy_fractal_points(n=100)                                    #␣
↪needs sage.modules
sage: len(D['1'])                                                          #␣
↪needs sage.modules
54
sage: len(D['2'])                                                          #␣
↪needs sage.modules
30
sage: len(D['3'])                                                          #␣
↪needs sage.modules
16
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->1')
>>> D = s.rauzy_fractal_points(n=Integer(100))                             ␣
↪     # needs sage.modules
>>> len(D['1'])                                                          #␣
↪needs sage.modules
54
>>> len(D['2'])                                                          #␣
↪needs sage.modules
30
>>> len(D['3'])                                                          #␣
↪needs sage.modules
16
```

AUTHOR:

Timo Jolivet (2012-06-16)

rauzy_fractal_projection(*eig=None*, *prec=53*)

Return a dictionary giving the projection of the canonical basis.

See the method *rauzy_fractal_plot()* for more details about the projection.

INPUT:

- `eig` – a real element of `QQbar` of degree >= 2 (default: `None`). The eigenvalue used for the projection. It must be an eigenvalue of `self.incidence_matrix()`. The one used by default is the maximal eigenvalue of `self.incidence_matrix()` (usually a Pisot number), but for substitutions with more than 3 letters other interesting choices are sometimes possible.
- `prec` – integer (default: 53); the number of bits used in the floating point representations of the coordinates

OUTPUT:

dictionary, letter -> vector, giving the projection

EXAMPLES:

The projection for the Rauzy fractal of the Tribonacci substitution is:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: s.rauzy_fractal_projection()                                         #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
{'1': (1.00000000000000, 0.000000000000000),
 '2': (-1.41964337760708, -0.606290729207199),
 '3': (-0.771844506346038, 1.11514250803994)}
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->1')
>>> s.rauzy_fractal_projection()                                    #␣
↪needs sage.modules
{'1': (1.00000000000000, 0.000000000000000),
 '2': (-1.41964337760708, -0.606290729207199),
 '3': (-0.771844506346038, 1.11514250803994)}
```

AUTHOR:

Timo Jolivet (2012-06-16)

restrict_domain(*alphabet*)

Return a restriction of `self` to the given alphabet.

INPUT:

- `alphabet` – an iterable

OUTPUT: WordMorphism

EXAMPLES:

```
sage: m = WordMorphism('a->b,b->a')
sage: m.restrict_domain('a')
WordMorphism: a->b
sage: m.restrict_domain('')
WordMorphism:
sage: m.restrict_domain('A')
WordMorphism:
sage: m.restrict_domain('Aa')
WordMorphism: a->b
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->b,b->a')
>>> m.restrict_domain('a')
WordMorphism: a->b
>>> m.restrict_domain('')
WordMorphism:
>>> m.restrict_domain('A')
WordMorphism:
>>> m.restrict_domain('Aa')
WordMorphism: a->b
```

The input alphabet must be iterable:

```
sage: m.restrict_domain(66)
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

```
>>> from sage.all import *
>>> m.restrict_domain(Integer(66))
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

reversal()

Return the reversal of `self`.

EXAMPLES:

```
sage: WordMorphism('6->ab,y->5,0->asd').reversal()
WordMorphism: 0->dsa, 6->ba, y->5
sage: WordMorphism('a->ab,b->a').reversal()
WordMorphism: a->ba, b->a
```

```
>>> from sage.all import *
>>> WordMorphism('6->ab,y->5,0->asd').reversal()
WordMorphism: 0->dsa, 6->ba, y->5
>>> WordMorphism('a->ab,b->a').reversal()
WordMorphism: a->ba, b->a
```

simplify_alphabet_size(*Z=None*)

If this morphism is simplifiable, return morphisms h and k such that this morphism is simplifiable with respect to h and k, otherwise raise `ValueError`.

This method is quite fast if this morphism is non-injective, but very slow if it is injective.

Let $f : X^* \to Y^*$ be a morphism. Then f is simplifiable with respect to morphisms $h : X^* \to Z^*$ and $k : Z^* \to Y^*$, if $f = k \circ h$ and $|Z| < |X|$. If also $Y \subseteq X$, then the morphism $g : Z^* \to Z^* = h \circ k$ is a simplification of f (with respect to h and k).

Loosely speaking, a morphism is simplifiable if it contains "more letters than is needed". Non-injectivity implies simplifiability. Simplification preserves some properties of the original morphism (e.g. repetitiveness).

For more information see Section 3 in [KO2000].

INPUT:

- `Z` – iterable (default: `self.domain().alphabet()`) whose elements are used as an alphabet for the simplification

EXAMPLES:

Example of a simplifiable (non-injective) morphism:

```
sage: f = WordMorphism('a->aca,b->badc,c->acab,d->adc')
sage: h, k = f.simplify_alphabet_size('xyz'); h, k
(WordMorphism: a->x, b->zy, c->xz, d->y, WordMorphism: x->aca, y->adc, z->b)
sage: k * h == f
True
sage: g = h * k; g
WordMorphism: x->xxzx, y->xyxz, z->zy
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->aca,b->badc,c->acab,d->adc')
```

(continues on next page)

(continued from previous page)

```
>>> h, k = f.simplify_alphabet_size('xyz'); h, k
(WordMorphism: a->x, b->zy, c->xz, d->y, WordMorphism: x->aca, y->adc, z->b)
>>> k * h == f
True
>>> g = h * k; g
WordMorphism: x->xxzx, y->xyxz, z->zy
```

Example of a simplifiable (injective) morphism:

```
sage: f = WordMorphism('a->abcc,b->abcd,c->abdc,d->abdd')
sage: h, k = f.simplify_alphabet_size('xyz'); h, k
(WordMorphism: a->xyy, b->xyz, c->xzy, d->xzz, WordMorphism: x->ab, y->c, z->
↪d)
sage: k * h == f
True
sage: g = h * k; g
WordMorphism: x->xyyxyz, y->xzy, z->xzz
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->abcc,b->abcd,c->abdc,d->abdd')
>>> h, k = f.simplify_alphabet_size('xyz'); h, k
(WordMorphism: a->xyy, b->xyz, c->xzy, d->xzz, WordMorphism: x->ab, y->c, z->
↪d)
>>> k * h == f
True
>>> g = h * k; g
WordMorphism: x->xyyxyz, y->xzy, z->xzz
```

Example of a non-simplifiable morphism:

```
sage: WordMorphism('a->aa').simplify_alphabet_size()
Traceback (most recent call last):
...
ValueError: self (a->aa) is not simplifiable
```

```
>>> from sage.all import *
>>> WordMorphism('a->aa').simplify_alphabet_size()
Traceback (most recent call last):
...
ValueError: self (a->aa) is not simplifiable
```

Example of an erasing morphism:

```
sage: f = WordMorphism('a->abc,b->cc,c->')
sage: h, k = f.simplify_alphabet_size(); h, k
(WordMorphism: a->a, b->b, c->, WordMorphism: a->abc, b->cc)
sage: k * h == f
True
sage: g = h * k; g
WordMorphism: a->ab, b->
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->abc,b->cc,c->')
>>> h, k = f.simplify_alphabet_size(); h, k
(WordMorphism: a->a, b->b, c->, WordMorphism: a->abc, b->cc)
>>> k * h == f
True
>>> g = h * k; g
WordMorphism: a->ab, b->
```

Example of a morphism, that is not an endomorphism:

```
sage: f = WordMorphism('a->xx,b->xy,c->yx,d->yy')
sage: h, k = f.simplify_alphabet_size(NN); h, k
(WordMorphism: a->00, b->01, c->10, d->11, WordMorphism: 0->x, 1->y)
sage: k * h == f
True
sage: len(k.domain().alphabet()) < len(f.domain().alphabet())
True
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->xx,b->xy,c->yx,d->yy')
>>> h, k = f.simplify_alphabet_size(NN); h, k
(WordMorphism: a->00, b->01, c->10, d->11, WordMorphism: 0->x, 1->y)
>>> k * h == f
True
>>> len(k.domain().alphabet()) < len(f.domain().alphabet())
True
```

simplify_until_injective()

Return a quadruplet (g, h, k, i), where g is an injective simplification of this morphism with respect to h, k and i.

Requires this morphism to be an endomorphism.

This methods basically calls *simplify_alphabet_size()* until the returned simplification is injective. If this morphism is already injective, a quadruplet (g, h, k, i) is still returned, where g is this morphism, h and k are the identity morphisms and i is 0.

Let $f : X^* \to Y^*$ be a morphism and $Y \subseteq X$. Then $g : Z^* \to Z^*$ is an injective simplification of f with respect to morphisms $h : X^* \to Z^*$ and $k : Z^* \to Y^*$ and a positive integer i, if g is injective, $|Z| < |X|$, $g^i = h \circ k$ and $f^i = k \circ h$.

For more information see Section 4 in [KO2000].

EXAMPLES:

```
sage: f = WordMorphism('a->abc,b->a,c->bc')
sage: g, h, k, i = f.simplify_until_injective(); g, h, k, i
(WordMorphism: a->aa, WordMorphism: a->aa, b->a, c->a, WordMorphism: a->abc,␣
↪2)
sage: g.is_injective()
True
sage: g**i == h * k
True
```

(continues on next page)

(continued from previous page)

```
sage: f**i == k * h
True
```

```
>>> from sage.all import *
>>> f = WordMorphism('a->abc,b->a,c->bc')
>>> g, h, k, i = f.simplify_until_injective(); g, h, k, i
(WordMorphism: a->aa, WordMorphism: a->aa, b->a, c->a, WordMorphism: a->abc,␣
↪2)
>>> g.is_injective()
True
>>> g**i == h * k
True
>>> f**i == k * h
True
```

sage.combinat.words.morphism.**get_cycles**(*f*, *domain*)

Return the list of cycles of the function `f` contained in `domain`.

INPUT:

- `f` – function
- `domain` – iterable, a subdomain of the domain of definition of `f`

EXAMPLES:

```
sage: from sage.combinat.words.morphism import get_cycles
sage: get_cycles(lambda i: (i+1)%3, [0,1,2])
[(0, 1, 2)]
sage: get_cycles(lambda i: [0,0,0][i], [0,1,2])
[(0,)]
sage: get_cycles(lambda i: [1,1,1][i], [0,1,2])
[(1,)]
sage: get_cycles(lambda i: [2,3,0][i], [0,1,2])
[(0, 2)]
sage: d = {'a': 'a', 'b': 'b'}
sage: get_cycles(d.__getitem__, 'ba')
[('b',), ('a',)]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.morphism import get_cycles
>>> get_cycles(lambda i: (i+Integer(1))%Integer(3), [Integer(0),Integer(1),
↪Integer(2)])
[(0, 1, 2)]
>>> get_cycles(lambda i: [Integer(0),Integer(0),Integer(0)][i], [Integer(0),
↪Integer(1),Integer(2)])
[(0,)]
>>> get_cycles(lambda i: [Integer(1),Integer(1),Integer(1)][i], [Integer(0),
↪Integer(1),Integer(2)])
[(1,)]
>>> get_cycles(lambda i: [Integer(2),Integer(3),Integer(0)][i], [Integer(0),
↪Integer(1),Integer(2)])
[(0, 2)]
```

(continues on next page)

(continued from previous page)

```
>>> d = {'a': 'a', 'b': 'b'}
>>> get_cycles(d.__getitem__, 'ba')
[('b',), ('a',)]
```

5.1.362 Word paths

This module implements word paths, which is an application of Combinatorics on Words to Discrete Geometry. A word path is the representation of a word as a discrete path in a vector space using a one-to-one correspondence between the alphabet and a set of vectors called steps. Many problems surrounding 2d lattice polygons (such as questions of self-intersection, area, inertia moment, etc.) can be solved in linear time (linear in the length of the perimeter) using theory from Combinatorics on Words.

On the square grid, the encoding of a path using a four-letter alphabet (for East, North, West and South directions) is also known as the Freeman chain code [1,2] (see [3] for further reading).

AUTHORS:

- Arnaud Bergeron (2008) : Initial version, path on the square grid
- Sébastien Labbé (2009-01-14) : New classes and hierarchy, doc and functions.

EXAMPLES:

The combinatorial class of all paths defined over three given steps:

```
sage: P = WordPaths('abc', steps=[(1,2), (-3,4), (0,-3)]); P
Word Paths over 3 steps
```

```
>>> from sage.all import *
>>> P = WordPaths('abc', steps=[(Integer(1),Integer(2)), (-Integer(3),Integer(4)),␣
↪(Integer(0),-Integer(3))]); P
Word Paths over 3 steps
```

This defines a one-to-one correspondence between alphabet and steps:

```
sage: d = P.letters_to_steps()
sage: sorted(d.items())
[('a', (1, 2)), ('b', (-3, 4)), ('c', (0, -3))]
```

```
>>> from sage.all import *
>>> d = P.letters_to_steps()
>>> sorted(d.items())
[('a', (1, 2)), ('b', (-3, 4)), ('c', (0, -3))]
```

Creation of a path from the combinatorial class P defined above:

```
sage: p = P('abaccba'); p
Path: abaccba
```

```
>>> from sage.all import *
>>> p = P('abaccba'); p
Path: abaccba
```

Many functions can be used on p: the coordinates of its trajectory, ask whether p is a closed path, plot it and many other:

```
sage: list(p.points())
[(0, 0), (1, 2), (-2, 6), (-1, 8), (-1, 5), (-1, 2), (-4, 6), (-3, 8)]
sage: p.is_closed()
False
sage: p.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> list(p.points())
[(0, 0), (1, 2), (-2, 6), (-1, 8), (-1, 5), (-1, 2), (-4, 6), (-3, 8)]
>>> p.is_closed()
False
>>> p.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

To obtain a list of all the available word path specific functions, use `help(p)`:

```
sage: help(p)
Help on FiniteWordPath_2d_str in module sage.combinat.words.paths object:
...
Methods inherited from FiniteWordPath_2d:
...
Methods inherited from FiniteWordPath_all:
...
```

```
>>> from sage.all import *
>>> help(p)
Help on FiniteWordPath_2d_str in module sage.combinat.words.paths object:
...
Methods inherited from FiniteWordPath_2d:
...
Methods inherited from FiniteWordPath_all:
...
```

Since p is a finite word, many functions from the word library are available:

```
sage: p.crochemore_factorization()
(a, b, a, c, c, ba)
sage: p.is_palindrome()
False
sage: p[:3]
Path: aba
sage: len(p)
7
```

```
>>> from sage.all import *
>>> p.crochemore_factorization()
(a, b, a, c, c, ba)
>>> p.is_palindrome()
False
```

(continues on next page)

(continued from previous page)

```
>>> p[:Integer(3)]
Path: aba
>>> len(p)
7
```

P also herits many functions from Words:

```
sage: P = WordPaths('rs', steps=[(1,2), (-1,4)]); P
Word Paths over 2 steps
sage: P.alphabet()
{'r', 's'}
sage: list(P.iterate_by_length(3))
[Path: rrr,
 Path: rrs,
 Path: rsr,
 Path: rss,
 Path: srr,
 Path: srs,
 Path: ssr,
 Path: sss]
```

```
>>> from sage.all import *
>>> P = WordPaths('rs', steps=[(Integer(1),Integer(2)), (-Integer(1),Integer(4))]); P
Word Paths over 2 steps
>>> P.alphabet()
{'r', 's'}
>>> list(P.iterate_by_length(Integer(3)))
[Path: rrr,
 Path: rrs,
 Path: rsr,
 Path: rss,
 Path: srr,
 Path: srs,
 Path: ssr,
 Path: sss]
```

When the number of given steps is half the size of alphabet, the opposite of vectors are used:

```
sage: P = WordPaths('abcd', [(1,0), (0,1)])
sage: sorted(P.letters_to_steps().items())
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
```

```
>>> from sage.all import *
>>> P = WordPaths('abcd', [(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> sorted(P.letters_to_steps().items())
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
```

Some built-in combinatorial classes of paths:

```
sage: P = WordPaths('abAB', steps='square_grid'); P
Word Paths on the square grid
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB', steps='square_grid'); P
Word Paths on the square grid
```

```
sage: D = WordPaths('()', steps='dyck'); D
Finite Dyck paths
sage: d = D('()()()(())'); d
Path: ()()()(())
sage: d.plot()                                                                  #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> D = WordPaths('()', steps='dyck'); D
Finite Dyck paths
>>> d = D('()()()(())'); d
Path: ()()()(())
>>> d.plot()                                                                #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
sage: P = WordPaths('abcdef', steps='triangle_grid')
sage: p = P('babaddefadabcadefaadfafabacdefa')
sage: p.plot()                                                                  #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> P = WordPaths('abcdef', steps='triangle_grid')
>>> p = P('babaddefadabcadefaadfafabacdefa')
>>> p.plot()                                                                #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

Vector steps may be in more than 2 dimensions:

```
sage: d = [(1,0,0), (0,1,0), (0,0,1)]
sage: P = WordPaths(alphabet='abc', steps=d); P
Word Paths over 3 steps
sage: p = P('abcabcabcabcaabacabcababcacbabacacabcaccbcac')
sage: p.plot()                                                                  #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> d = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),Integer(0)),␣
↪(Integer(0),Integer(0),Integer(1))]
>>> P = WordPaths(alphabet='abc', steps=d); P
Word Paths over 3 steps
>>> p = P('abcabcabcabcaabacabcababcacbabacacabcaccbcac')
>>> p.plot()                                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics3d Object
```

```
sage: d = [(1,3,5,1), (-5,1,-6,0), (0,0,1,9), (4,2,-1,0)]
sage: P = WordPaths(alphabet='rstu', steps=d); P
Word Paths over 4 steps
sage: p = P('rtusuusususuturrsust'); p
Path: rtusuusususuturrsust
sage: p.end_point()
(5, 31, -26, 30)
```

```
>>> from sage.all import *
>>> d = [(Integer(1),Integer(3),Integer(5),Integer(1)), (-Integer(5),Integer(1),-
↪Integer(6),Integer(0)), (Integer(0),Integer(0),Integer(1),Integer(9)), (Integer(4),
↪Integer(2),-Integer(1),Integer(0))]
>>> P = WordPaths(alphabet='rstu', steps=d); P
Word Paths over 4 steps
>>> p = P('rtusuusususuturrsust'); p
Path: rtusuusususuturrsust
>>> p.end_point()
(5, 31, -26, 30)
```

```
sage: CubePaths = WordPaths('abcABC', steps='cube_grid'); CubePaths
Word Paths on the cube grid
sage: CubePaths('abcabaabcabAAAAA').plot()                                    #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> CubePaths = WordPaths('abcABC', steps='cube_grid'); CubePaths
Word Paths on the cube grid
>>> CubePaths('abcabaabcabAAAAA').plot()                                    #␣
↪needs sage.plot
Graphics3d Object
```

The input data may be a str, a list, a tuple, a callable or a finite iterator:

```
sage: P = WordPaths([0, 1, 2, 3])
sage: P([0,1,2,3,2,1,2,3,2])
Path: 012321232
sage: P((0,1,2,3,2,1,2,3,2))
Path: 012321232
sage: P(lambda n:n%4, length=10)
Path: 0123012301
sage: P(iter([0,3,2,1]), length='finite')
Path: 0321
```

```
>>> from sage.all import *
>>> P = WordPaths([Integer(0), Integer(1), Integer(2), Integer(3)])
>>> P([Integer(0),Integer(1),Integer(2),Integer(3),Integer(2),Integer(1),Integer(2),
↪Integer(3),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
Path: 012321232
>>> P((Integer(0),Integer(1),Integer(2),Integer(3),Integer(2),Integer(1),Integer(2),
↪Integer(3),Integer(2)))
Path: 012321232
>>> P(lambda n:n%Integer(4), length=Integer(10))
Path: 0123012301
>>> P(iter([Integer(0),Integer(3),Integer(2),Integer(1)]), length='finite')
Path: 0321
```

REFERENCES:

- [1] Freeman, H.: *On the encoding of arbitrary geometric configurations*. IRE Trans. Electronic Computer 10 (1961) 260-268.
- [2] Freeman, H.: *Boundary encoding and processing*. In Lipkin, B., Rosenfeld, A., eds.: Picture Processing and Psychopictorics, Academic Press, New York (1970) 241-266.
- [3] Braquelaire, J.P., Vialard, A.: *Euclidean paths: A new representation of boundary of discrete regions*. Graphical Models and Image Processing 61 (1999) 16-43.
- [4] Wikipedia article Regular_tiling
- [5] Wikipedia article Dyck_word

class sage.combinat.words.paths.**FiniteWordPath_2d**

Bases: *FiniteWordPath_all*

animate()

Return an animation object illustrating the path growing step by step.

EXAMPLES:

```
sage: P = WordPaths('abAB')
sage: p = P('aaababbb')
sage: a = p.animate(); print(a)                                         #␣
↪needs sage.plot
Animation with 9 frames
sage: show(a)                         # long time, optional - imagemagick,␣
↪needs sage.plot
sage: show(a, delay=35, iterations=3)       # long time, optional -␣
↪imagemagick, needs sage.plot
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB')
>>> p = P('aaababbb')
>>> a = p.animate(); print(a)                                         #␣
↪needs sage.plot
Animation with 9 frames
>>> show(a)                         # long time, optional - imagemagick, needs␣
↪sage.plot
>>> show(a, delay=Integer(35), iterations=Integer(3))       # long time,␣
↪optional - imagemagick, needs sage.plot
```

```
sage: P = WordPaths('abcdef',steps='triangle')
sage: p =  P('abcdef')
```

(continues on next page)

(continued from previous page)

```
sage: a = p.animate(); print(a)                                              #␣
↪needs sage.plot
Animation with 8 frames
sage: show(a)                    # long time, optional - imagemagick,␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> P = WordPaths('abcdef',steps='triangle')
>>> p =  P('abcdef')
>>> a = p.animate(); print(a)                                              #␣
↪needs sage.plot
Animation with 8 frames
>>> show(a)                    # long time, optional - imagemagick, needs␣
↪sage.plot
```

If the path is closed, the plain polygon is added at the end of the animation:

```
sage: P = WordPaths('abAB')
sage: p = P('ababAbABABaB')
sage: a = p.animate(); print(a)                                              #␣
↪needs sage.plot
Animation with 14 frames
sage: show(a)                    # long time, optional - imagemagick,␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB')
>>> p = P('ababAbABABaB')
>>> a = p.animate(); print(a)                                              #␣
↪needs sage.plot
Animation with 14 frames
>>> show(a)                    # long time, optional - imagemagick, needs␣
↪sage.plot
```

Another example illustrating a Fibonacci tile:

```
sage: w = words.fibonacci_tile(2)
sage: a = w.animate(); print(a)                                              #␣
↪needs sage.plot
Animation with 54 frames
sage: show(a)                    # long time, optional - imagemagick,␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> w = words.fibonacci_tile(Integer(2))
>>> a = w.animate(); print(a)                                              #␣
↪needs sage.plot
Animation with 54 frames
>>> show(a)                    # long time, optional - imagemagick, needs␣
↪sage.plot
```

The first 4 Fibonacci tiles in an animation:

```
sage: # needs sage.plot
sage: a = words.fibonacci_tile(0).animate()
sage: b = words.fibonacci_tile(1).animate()
sage: c = words.fibonacci_tile(2).animate()
sage: d = words.fibonacci_tile(3).animate()
sage: print(a*b*c*d)
Animation with 296 frames
sage: show(a*b*c*d)                    # long time, optional - imagemagick
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> a = words.fibonacci_tile(Integer(0)).animate()
>>> b = words.fibonacci_tile(Integer(1)).animate()
>>> c = words.fibonacci_tile(Integer(2)).animate()
>>> d = words.fibonacci_tile(Integer(3)).animate()
>>> print(a*b*c*d)
Animation with 296 frames
>>> show(a*b*c*d)                    # long time, optional - imagemagick
```

Note

If ImageMagick is not installed, you will get an error message like this:

```
convert: not found

Error: ImageMagick does not appear to be installed. Saving an
animation to a GIF file or displaying an animation requires
ImageMagick, so please install it and try again.
```

See www.imagemagick.org, for example.

area()

Return the area of a closed path.

INPUT:

- `self` – a closed path

EXAMPLES:

```
sage: P = WordPaths('abcd',steps=[(1,1),(-1,1),(-1,-1),(1,-1)])
sage: p = P('abcd')
sage: p.area()          #todo: not implemented
2
```

```
>>> from sage.all import *
>>> P = WordPaths('abcd',steps=[(Integer(1),Integer(1)),(-Integer(1),
↪Integer(1)),(-Integer(1),-Integer(1)),(Integer(1),-Integer(1))])
>>> p = P('abcd')
>>> p.area()          #todo: not implemented
2
```

height()

Return the height of `self`.

The height of a $2d$-path is merely the difference between the highest and the lowest y-coordinate of each points traced by it.

OUTPUT: nonnegative real number

EXAMPLES:

```
sage: Freeman = WordPaths('abAB')
sage: Freeman('aababaabbbAA').height()
5
```

```
>>> from sage.all import *
>>> Freeman = WordPaths('abAB')
>>> Freeman('aababaabbbAA').height()
5
```

The function is well-defined if `self` is not simple or close:

```
sage: Freeman('aabAAB').height()
1
sage: Freeman('abbABa').height()
2
```

```
>>> from sage.all import *
>>> Freeman('aabAAB').height()
1
>>> Freeman('abbABa').height()
2
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,0),(1,1)])
sage: p = Paths('abbaa')
sage: p.height()
2
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.height()
2
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.height()
2.59807621135332
```

```
>>> from sage.all import *
>>> Paths = WordPaths('ab', steps=[(Integer(1),Integer(0)),(Integer(1),
↪Integer(1))])
>>> p = Paths('abbaa')
>>> p.height()
2
>>> DyckPaths = WordPaths('ab', steps='dyck')
>>> p = DyckPaths('abaabb')
>>> p.height()
2
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
```

(continues on next page)

(continued from previous page)

```
>>> w.height()
2.59807621135332
```

height_vector()

Return the height at each point.

EXAMPLES:

```
sage: Paths = WordPaths('ab', steps=[(1,0),(0,1)])
sage: p = Paths('abbba')
sage: p.height_vector()
[0, 0, 1, 2, 3, 3]
```

```
>>> from sage.all import *
>>> Paths = WordPaths('ab', steps=[(Integer(1),Integer(0)),(Integer(0),
↪Integer(1))])
>>> p = Paths('abbba')
>>> p.height_vector()
[0, 0, 1, 2, 3, 3]
```

plot(*pathoptions={'rgbcolor': 'red', 'thickness': 3}, fill=True, filloptions={'rgbcolor': 'red', 'alpha': 0.2}, startpoint=True, startoptions={'rgbcolor': 'red', 'pointsize': 100}, endarrow=True, arrowoptions={'rgbcolor': 'red', 'arrowsize': 20, 'width': 3}, gridlines=False, gridoptions={}*)

Return a 2d Graphics illustrating the path.

INPUT:

- `pathoptions` – (dict, default:dict(rgbcolor='red',thickness=3)), options for the path drawing
- `fill` – boolean (default: `True`); if fill is `True` and if the path is closed, the inside is colored
- `filloptions` – (dict, default:dict(rgbcolor='red',alpha=0.2)), options for the inside filling
- `startpoint` – boolean (default: `True`); draw the start point?
- `startoptions` – (dict, default:dict(rgbcolor='red',pointsize=100)) options for the start point drawing
- `endarrow` – boolean (default: `True`); draw an arrow end at the end?
- `arrowoptions` – (dict, default:dict(rgbcolor='red',arrowsize=20, width=3)) options for the end point arrow
- `gridlines` – boolean (default: `False`); show gridlines?
- `gridoptions` – (dict, default: {}), options for the gridlines

EXAMPLES:

A non closed path on the square grid:

```
sage: P = WordPaths('abAB')
sage: P('ababAABAB').plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB')
>>> P('ababAABAB').plot()                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

A closed path on the square grid:

```
sage: P('abababAABABB').plot()                                                #␣
↪needs sage.plot
Graphics object consisting of 4 graphics primitives
```

```
>>> from sage.all import *
>>> P('abababAABABB').plot()                                                #␣
↪needs sage.plot
Graphics object consisting of 4 graphics primitives
```

A Dyck path:

```
sage: P = WordPaths('()', steps='dyck')
sage: P('()()()((()))').plot()                                                #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> P = WordPaths('()', steps='dyck')
>>> P('()()()((()))').plot()                                                #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

A path in the triangle grid:

```
sage: P = WordPaths('abcdef', steps='triangle_grid')
sage: P('abcdedededefab').plot()                                                #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> P = WordPaths('abcdef', steps='triangle_grid')
>>> P('abcdedededefab').plot()                                                #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

A polygon of length 220 that tiles the plane in two ways:

```
sage: P = WordPaths('abAB')
sage: P('aBababAbabaBaBABaBabaBaBABAbABABaBabaBaBABaBababAbabaBaBABaBabaBaBABA
↪bABABaBABAbAbabAbABABaBABAbABABaBabaBaBABAbABABaBABAbAbabAbABAbAbabaBababAbA
↪BAbAbabAbABABaBABAbAbabAbABAbAbabaBababAbabaBaBABaBababAbabaBababAbABAbAbab'
↪).plot()   # needs sage.plot
Graphics object consisting of 4 graphics primitives
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB')
>>> P('aBababAbabaBaBABaBabaBaBABAbABABaBabaBaBABaBababAbabaBaBABaBabaBaBABAbA
```

(continues on next page)

(continued from previous page)

```
↪BABaBABAbAbabAbABABaBABAbABABaBabaBaBABAbABABaBABAbAbabAbABAbAbabaBababAbABA
↪bAbabAbABABaBABAbAbabAbABAbAbabaBababAbabaBaBABaBababAbabaBababAbABAbAbab').
↪plot()   # needs sage.plot
Graphics object consisting of 4 graphics primitives
```

With gridlines:

```
sage: P('ababababab').plot(gridlines=True)                                    #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> P('ababababab').plot(gridlines=True)                                    #␣
↪needs sage.plot
```

plot_directive_vector(*options={'rgbcolor': 'blue'}*)

Return an arrow 2d graphics that goes from the start of the path to the end.

INPUT:

- `options` – dictionary, default: {'rgbcolor': 'blue'} graphic options for the arrow

If the start is the same as the end, a single point is returned.

EXAMPLES:

```
sage: P = WordPaths('abcd'); P
Word Paths on the square grid
sage: p = P('aaaccaccacacacaccccccbbdd'); p
Path: aaaccaccacacacaccccccbbdd
sage: R = p.plot() + p.plot_directive_vector()                                #␣
↪needs sage.plot
sage: R.axes(False)                                                           #␣
↪needs sage.plot
sage: R.set_aspect_ratio(1)                                                   #␣
↪needs sage.plot
sage: R.plot()                                                                #␣
↪needs sage.plot
Graphics object consisting of 4 graphics primitives
```

```
>>> from sage.all import *
>>> P = WordPaths('abcd'); P
Word Paths on the square grid
>>> p = P('aaaccaccacacacaccccccbbdd'); p
Path: aaaccaccacacacaccccccbbdd
>>> R = p.plot() + p.plot_directive_vector()                                #␣
↪needs sage.plot
>>> R.axes(False)                                                           #␣
↪needs sage.plot
>>> R.set_aspect_ratio(Integer(1))                                            ␣
↪     # needs sage.plot
>>> R.plot()                                                                #␣
↪needs sage.plot
Graphics object consisting of 4 graphics primitives
```

width()

Return the width of `self`.

The height of a $2d$-path is merely the difference between the rightmost and the leftmost x-coordinate of each points traced by it.

OUTPUT: nonnegative real number

EXAMPLES:

```
sage: Freeman = WordPaths('abAB')
sage: Freeman('aababaabbbAA').width()
5
```

```
>>> from sage.all import *
>>> Freeman = WordPaths('abAB')
>>> Freeman('aababaabbbAA').width()
5
```

The function is well-defined if `self` is not simple or close:

```
sage: Freeman('aabAAB').width()
2
sage: Freeman('abbABa').width()
1
```

```
>>> from sage.all import *
>>> Freeman('aabAAB').width()
2
>>> Freeman('abbABa').width()
1
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,0),(1,1)])
sage: p = Paths('abbaa')
sage: p.width()
5
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.width()
6
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.width()
4.50000000000000
```

```
>>> from sage.all import *
>>> Paths = WordPaths('ab', steps=[(Integer(1),Integer(0)),(Integer(1),
↪Integer(1))])
>>> p = Paths('abbaa')
>>> p.width()
5
>>> DyckPaths = WordPaths('ab', steps='dyck')
>>> p = DyckPaths('abaabb')
```

(continues on next page)

(continued from previous page)

```
>>> p.width()
6
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.width()
4.50000000000000
```

width_vector()

Return the width at each point.

EXAMPLES:

```
sage: Paths = WordPaths('ab', steps=[(1,0),(0,1)])
sage: p = Paths('abbba')
sage: p.width_vector()
[0, 1, 1, 1, 1, 2]
```

```
>>> from sage.all import *
>>> Paths = WordPaths('ab', steps=[(Integer(1),Integer(0)),(Integer(0),
↪Integer(1))])
>>> p = Paths('abbba')
>>> p.width_vector()
[0, 1, 1, 1, 1, 2]
```

xmax()

Return the maximum of the x-coordinates of the path.

EXAMPLES:

```
sage: P = WordPaths('0123')
sage: p = P('0101013332')
sage: p.xmax()
3
```

```
>>> from sage.all import *
>>> P = WordPaths('0123')
>>> p = P('0101013332')
>>> p.xmax()
3
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,-1),(-1,1)])
sage: p = Paths('ababa')
sage: p.xmax()
1
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.xmax()
6
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.xmax()
4.50000000000000
```

```
>>> from sage.all import *
>>> Paths = WordPaths('ab', steps=[(Integer(1),-Integer(1)),(-Integer(1),
↪Integer(1))])
>>> p = Paths('ababa')
>>> p.xmax()
1
>>> DyckPaths = WordPaths('ab', steps='dyck')
>>> p = DyckPaths('abaabb')
>>> p.xmax()
6
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.xmax()
4.50000000000000
```

xmin()

Return the minimum of the x-coordinates of the path.

EXAMPLES:

```
sage: P = WordPaths('0123')
sage: p = P('0101013332')
sage: p.xmin()
0
```

```
>>> from sage.all import *
>>> P = WordPaths('0123')
>>> p = P('0101013332')
>>> p.xmin()
0
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,0),(-1,1)])
sage: p = Paths('abbba')
sage: p.xmin()
-2
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.xmin()
0
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.xmin()
0.000000000000000
```

```
>>> from sage.all import *
>>> Paths = WordPaths('ab', steps=[(Integer(1),Integer(0)),(-Integer(1),
↪Integer(1))])
>>> p = Paths('abbba')
>>> p.xmin()
-2
>>> DyckPaths = WordPaths('ab', steps='dyck')
>>> p = DyckPaths('abaabb')
>>> p.xmin()
```

(continues on next page)

(continued from previous page)

```
0
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.xmin()
0.000000000000000
```

ymax()

Return the maximum of the y-coordinates of the path.

EXAMPLES:

```
sage: P = WordPaths('0123')
sage: p = P('0101013332')
sage: p.ymax()
3
```

```
>>> from sage.all import *
>>> P = WordPaths('0123')
>>> p = P('0101013332')
>>> p.ymax()
3
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,-1),(-1,1)])
sage: p = Paths('ababa')
sage: p.ymax()
0
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.ymax()
2
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.ymax()
2.59807621135332
```

```
>>> from sage.all import *
>>> Paths = WordPaths('ab', steps=[(Integer(1),-Integer(1)),(-Integer(1),
↪Integer(1))])
>>> p = Paths('ababa')
>>> p.ymax()
0
>>> DyckPaths = WordPaths('ab', steps='dyck')
>>> p = DyckPaths('abaabb')
>>> p.ymax()
2
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.ymax()
2.59807621135332
```

ymin()

Return the minimum of the y-coordinates of the path.

EXAMPLES:

```
sage: P = WordPaths('0123')
sage: p = P('0101013332')
sage: p.ymin()
0
```

```
>>> from sage.all import *
>>> P = WordPaths('0123')
>>> p = P('0101013332')
>>> p.ymin()
0
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,-1),(-1,1)])
sage: p = Paths('ababa')
sage: p.ymin()
-1
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.ymin()
0
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.ymin()
0.000000000000000
```

```
>>> from sage.all import *
>>> Paths = WordPaths('ab', steps=[(Integer(1),-Integer(1)),(-Integer(1),
↪Integer(1))])
>>> p = Paths('ababa')
>>> p.ymin()
-1
>>> DyckPaths = WordPaths('ab', steps='dyck')
>>> p = DyckPaths('abaabb')
>>> p.ymin()
0
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.ymin()
0.000000000000000
```

class sage.combinat.words.paths.**FiniteWordPath_2d_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_2d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_2d_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_2d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_2d_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_2d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_2d_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_2d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_2d_list**

Bases: *WordDatatype_list*, *FiniteWordPath_2d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_2d_str**

Bases: *WordDatatype_str*, *FiniteWordPath_2d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_2d_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_2d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_3d**

Bases: *FiniteWordPath_all*

plot(*pathoptions={'rgbcolor': 'red', 'arrow_head': True, 'thickness': 3}, startpoint=True, startoptions={'rgbcolor': 'red', 'size': 10}*)

INPUT:

- `pathoptions` – (dict, default:dict(rgbcolor='red',arrow_head=True, thickness=3)), options for the path drawing
- `startpoint` – boolean (default: `True`); draw the start point?
- `startoptions` – (dict, default:dict(rgbcolor='red',size=10)) options for the start point drawing

EXAMPLES:

```
sage: d = ( vector((1,3,2)), vector((2,-4,5)) )
sage: P = WordPaths(alphabet='ab', steps=d); P
Word Paths over 2 steps
sage: p = P('ababab'); p
Path: ababab
sage: p.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object

sage: P = WordPaths('abcABC', steps='cube_grid')
sage: p = P('abcabcAABBC')
sage: p.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> d = ( vector((Integer(1),Integer(3),Integer(2))), vector((Integer(2),-
↪Integer(4),Integer(5))) )
>>> P = WordPaths(alphabet='ab', steps=d); P
Word Paths over 2 steps
>>> p = P('ababab'); p
Path: ababab
>>> p.plot()                                                        #␣
↪needs sage.plot
Graphics3d Object

>>> P = WordPaths('abcABC', steps='cube_grid')
>>> p = P('abcabcAABBC')
>>> p.plot()                                                        #␣
↪needs sage.plot
Graphics3d Object
```

class sage.combinat.words.paths.**FiniteWordPath_3d_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_3d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_3d_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_3d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_3d_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_3d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_3d_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_3d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_3d_list**

Bases: *WordDatatype_list*, *FiniteWordPath_3d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_3d_str**

Bases: *WordDatatype_str*, *FiniteWordPath_3d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_3d_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_3d*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_all**

Bases: SageObject

directive_vector()

Return the directive vector of self.

The directive vector is the vector starting at the start point and ending at the end point of the path self.

EXAMPLES:

```
sage: WordPaths('abcdef')('abababab').directive_vector()
(6, 2*sqrt3)
sage: WordPaths('abAB')('abababab').directive_vector()
(4, 4)
sage: P = WordPaths('abcABC', steps='cube_grid')
sage: P('abababababCC').directive_vector()
(4, 4, -2)
sage: WordPaths('abcdef')('abcdef').directive_vector()
(0, 0)
sage: P = WordPaths('abc', steps=[(1,3,7,9),(-4,1,0,0),(0,32,1,8)])
sage: P('abcabababacaacccbbcac').directive_vector()
(-16, 254, 63, 128)
```

```
>>> from sage.all import *
>>> WordPaths('abcdef')('abababab').directive_vector()
(6, 2*sqrt3)
>>> WordPaths('abAB')('abababab').directive_vector()
(4, 4)
>>> P = WordPaths('abcABC', steps='cube_grid')
>>> P('abababababCC').directive_vector()
(4, 4, -2)
>>> WordPaths('abcdef')('abcdef').directive_vector()
(0, 0)
```

(continues on next page)

(continued from previous page)

```
>>> P = WordPaths('abc', steps=[(Integer(1),Integer(3),Integer(7),Integer(9)),
↪(-Integer(4),Integer(1),Integer(0),Integer(0)),(Integer(0),Integer(32),
↪Integer(1),Integer(8))])
>>> P('abcababababacaacccbbcac').directive_vector()
(-16, 254, 63, 128)
```

end_point()

Return the end point of the path.

EXAMPLES:

```
sage: WordPaths('abcdef')('abababab').end_point()
(6, 2*sqrt3)
sage: WordPaths('abAB')('abababab').end_point()
(4, 4)
sage: P = WordPaths('abcABC', steps='cube_grid')
sage: P('ababababCC').end_point()
(4, 4, -2)
sage: WordPaths('abcdef')('abcdef').end_point()
(0, 0)
sage: P = WordPaths('abc', steps=[(1,3,7,9),(-4,1,0,0),(0,32,1,8)])
sage: P('abcababababacaacccbbcac').end_point()
(-16, 254, 63, 128)
```

```
>>> from sage.all import *
>>> WordPaths('abcdef')('abababab').end_point()
(6, 2*sqrt3)
>>> WordPaths('abAB')('abababab').end_point()
(4, 4)
>>> P = WordPaths('abcABC', steps='cube_grid')
>>> P('ababababCC').end_point()
(4, 4, -2)
>>> WordPaths('abcdef')('abcdef').end_point()
(0, 0)
>>> P = WordPaths('abc', steps=[(Integer(1),Integer(3),Integer(7),Integer(9)),
↪(-Integer(4),Integer(1),Integer(0),Integer(0)),(Integer(0),Integer(32),
↪Integer(1),Integer(8))])
>>> P('abcababababacaacccbbcac').end_point()
(-16, 254, 63, 128)
```

is_closed()

Return `True` if the path is closed.

A path is closed if the origin and the end of the path are equal.

EXAMPLES:

```
sage: P = WordPaths('abcd', steps=[(1,0),(0,1),(-1,0),(0,-1)])
sage: P('abcd').is_closed()
True
sage: P('abc').is_closed()
False
sage: P().is_closed()
```

(continues on next page)

(continued from previous page)

```
True
sage: P('aacacc').is_closed()
True
```

```
>>> from sage.all import *
>>> P = WordPaths('abcd', steps=[(Integer(1),Integer(0)),(Integer(0),
→Integer(1)),(-Integer(1),Integer(0)),(Integer(0),-Integer(1))])
>>> P('abcd').is_closed()
True
>>> P('abc').is_closed()
False
>>> P().is_closed()
True
>>> P('aacacc').is_closed()
True
```

is_simple()

Return `True` if the path is simple.

A path is simple if all its points are distinct.

If the path is closed, the last point is not considered.

EXAMPLES:

```
sage: P = WordPaths('abcdef',steps='triangle_grid');P
Word Paths on the triangle grid
sage: P('abc').is_simple()
True
sage: P('abcde').is_simple()
True
sage: P('abcdef').is_simple()
True
sage: P('ad').is_simple()
True
sage: P('aabdee').is_simple()
False
```

```
>>> from sage.all import *
>>> P = WordPaths('abcdef',steps='triangle_grid');P
Word Paths on the triangle grid
>>> P('abc').is_simple()
True
>>> P('abcde').is_simple()
True
>>> P('abcdef').is_simple()
True
>>> P('ad').is_simple()
True
>>> P('aabdee').is_simple()
False
```

is_tangent()

The is_tangent() method, which is implemented for words, has an extended meaning for word paths, which is not implemented yet.

AUTHOR:

- Thierry Monteil

plot_projection(*v=None*, *letters=None*, *color=None*, *ring=None*, *size=12*, *kind='right'*)

Return an image of the projection of the successive points of the path into the space orthogonal to the given vector.

INPUT:

- `self` – a word path in a 3 or 4 dimension vector space
- `v` – vector (default: `None`); if `None`, the directive vector (i.e. the end point minus starting point) of the path is considered.
- `letters` – iterable (default: `None`); of the letters to be projected. If `None`, then all the letters are considered.
- `color` – dictionary (default: `None`); of the letters mapped to colors. If `None`, automatic colors are chosen.
- `ring` – ring (default: `None`); where to do the computations. If `None`, RealField(53) is used.
- `size` – number (default: `12`); size of the points
- `kind` – string (default: `'right'`); either `'right'` or `'left'`. The color of a letter is given to the projected prefix to the right or the left of the letter.

OUTPUT: 2d or 3d Graphic object

EXAMPLES:

The Rauzy fractal:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.fixed_point('1')
sage: v = s.pisot_eigenvector_right()
sage: P = WordPaths('123',[(1,0,0),(0,1,0),(0,0,1)])
sage: w = P(D[:200])
sage: w.plot_projection(v)   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->1')
>>> D = s.fixed_point('1')
>>> v = s.pisot_eigenvector_right()
>>> P = WordPaths('123',[(Integer(1),Integer(0),Integer(0)),(Integer(0),
↪Integer(1),Integer(0)),(Integer(0),Integer(0),Integer(1))])
>>> w = P(D[:Integer(200)])
>>> w.plot_projection(v)   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

In this case, the abelianized vector doesn't give a good projection:

```
sage: w.plot_projection()   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

```
>>> from sage.all import *
>>> w.plot_projection()   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

You can project only the letters you want:

```
sage: w.plot_projection(v, letters='12')   # long time (2s)
Graphics object consisting of 168 graphics primitives
```

```
>>> from sage.all import *
>>> w.plot_projection(v, letters='12')   # long time (2s)
Graphics object consisting of 168 graphics primitives
```

You can increase or decrease the precision of the computations by changing the ring of the projection matrix:

```
sage: w.plot_projection(v, ring=RealField(20))   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

```
>>> from sage.all import *
>>> w.plot_projection(v, ring=RealField(Integer(20)))   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

You can change the size of the points:

```
sage: w.plot_projection(v, size=30)   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

```
>>> from sage.all import *
>>> w.plot_projection(v, size=Integer(30))   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

You can assign the color of a letter to the projected prefix to the right or the left of the letter:

```
sage: w.plot_projection(v, kind='left')   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

```
>>> from sage.all import *
>>> w.plot_projection(v, kind='left')   # long time (2s)
Graphics object consisting of 200 graphics primitives
```

To remove the axis, do like this:

```
sage: r = w.plot_projection(v)                                              #␣
↪needs sage.plot
sage: r.axes(False)                                                         #␣
↪needs sage.plot
sage: r                              # long time (2s)                       #␣
↪needs sage.plot
Graphics object consisting of 200 graphics primitives
```

```
>>> from sage.all import *
>>> r = w.plot_projection(v)                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
>>> r.axes(False)                                                       #␣
↪needs sage.plot
>>> r                            # long time (2s)                      #␣
↪needs sage.plot
Graphics object consisting of 200 graphics primitives
```

You can assign different colors to each letter:

```
sage: color = {'1': 'purple', '2': (.2,.3,.4), '3': 'magenta'}
sage: w.plot_projection(v, color=color)      # long time (2s)             #␣
↪needs sage.plot
Graphics object consisting of 200 graphics primitives
```

```
>>> from sage.all import *
>>> color = {'1': 'purple', '2': (RealNumber('.2'),RealNumber('.3'),
↪RealNumber('.4')), '3': 'magenta'}
>>> w.plot_projection(v, color=color)      # long time (2s)             #␣
↪needs sage.plot
Graphics object consisting of 200 graphics primitives
```

The 3d-Rauzy fractal:

```
sage: s = WordMorphism('1->12,2->13,3->14,4->1')
sage: D = s.fixed_point('1')
sage: v = s.pisot_eigenvector_right()
sage: P = WordPaths('1234',[(1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)])
sage: w = P(D[:200])
sage: w.plot_projection(v)                                                 #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->14,4->1')
>>> D = s.fixed_point('1')
>>> v = s.pisot_eigenvector_right()
>>> P = WordPaths('1234',[(Integer(1),Integer(0),Integer(0),Integer(0)),␣
↪(Integer(0),Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(0),Integer(1))])
>>> w = P(D[:Integer(200)])
>>> w.plot_projection(v)                                                 #␣
↪needs sage.plot
Graphics3d Object
```

The dimension of vector space of the parent must be 3 or 4:

```
sage: P = WordPaths('ab', [(1, 0), (0, 1)])
sage: p = P('aabbabbab')
sage: p.plot_projection()                                                  #␣
↪needs sage.plot
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
TypeError: The dimension of the vector space (=2) must be 3 or 4
```

```
>>> from sage.all import *
>>> P = WordPaths('ab', [(Integer(1), Integer(0)), (Integer(0), Integer(1))])
>>> p = P('aabbabbab')
>>> p.plot_projection()                                                    #␣
↪needs sage.plot
Traceback (most recent call last):
...
TypeError: The dimension of the vector space (=2) must be 3 or 4
```

points(*include_last=True*)

Return an iterator yielding a list of points used to draw the path represented by this word.

INPUT:

- `include_last` – boolean (default: `True`); whether to include the last point

EXAMPLES:

A simple closed square:

```
sage: P = WordPaths('abAB')
sage: list(P('abAB').points())
[(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB')
>>> list(P('abAB').points())
[(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]
```

A simple closed square without the last point:

```
sage: list(P('abAB').points(include_last=False))
[(0, 0), (1, 0), (1, 1), (0, 1)]
```

```
>>> from sage.all import *
>>> list(P('abAB').points(include_last=False))
[(0, 0), (1, 0), (1, 1), (0, 1)]
```

```
sage: list(P('abaB').points())
[(0, 0), (1, 0), (1, 1), (2, 1), (2, 0)]
```

```
>>> from sage.all import *
>>> list(P('abaB').points())
[(0, 0), (1, 0), (1, 1), (2, 1), (2, 0)]
```

projected_path(*v=None, ring=None*)

Return the path projected into the space orthogonal to the given vector.

INPUT:

- `v` – vector (default: `None`); if `None`, the directive vector (i.e. the end point minus starting point) of the path is considered.

- `ring` – ring (default: `None`); where to do the computations. If `None`, RealField(53) is used.

OUTPUT: word path

EXAMPLES:

The projected path of the tribonacci word:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.fixed_point('1')
sage: v = s.pisot_eigenvector_right()
sage: P = WordPaths('123',[(1,0,0),(0,1,0),(0,0,1)])
sage: w = P(D[:1000])
sage: p = w.projected_path(v)
sage: p
Path: 1213121121312121312112131213121121312121...
sage: p[:20].plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->1')
>>> D = s.fixed_point('1')
>>> v = s.pisot_eigenvector_right()
>>> P = WordPaths('123',[(Integer(1),Integer(0),Integer(0)),(Integer(0),
↪Integer(1),Integer(0)),(Integer(0),Integer(0),Integer(1))])
>>> w = P(D[:Integer(1000)])
>>> p = w.projected_path(v)
>>> p
Path: 1213121121312121312112131213121121312121...
>>> p[:Integer(20)].plot()                                             ␣
↪     # needs sage.plot
Graphics object consisting of 3 graphics primitives
```

The `ring` argument allows to change the precision of the projected steps:

```
sage: p = w.projected_path(v, RealField(10))
sage: p
Path: 1213121121312121312112131213121121312121...
sage: p.parent().letters_to_steps()
{'1': (-0.53, 0.00), '2': (0.75, -0.48), '3': (0.41, 0.88)}
```

```
>>> from sage.all import *
>>> p = w.projected_path(v, RealField(Integer(10)))
>>> p
Path: 1213121121312121312112131213121121312121...
>>> p.parent().letters_to_steps()
{'1': (-0.53, 0.00), '2': (0.75, -0.48), '3': (0.41, 0.88)}
```

projected_point_iterator(*v=None*, *ring=None*)

Return an iterator of the projection of the orbit points of the path into the space orthogonal to the given vector.

INPUT:

- `v` – vector (default: `None`); if `None`, the directive vector (i.e. the end point minus starting point) of the path is considered

- `ring` – ring (default: `None`); where to do the computations. If `None`, RealField(53) is used.

OUTPUT: iterator of points

EXAMPLES:

Projected points of the Rauzy fractal:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.fixed_point('1')
sage: v = s.pisot_eigenvector_right()
sage: P = WordPaths('123',[(1,0,0),(0,1,0),(0,0,1)])
sage: w = P(D[:200])
sage: it = w.projected_point_iterator(v)
sage: for i in range(6): next(it)
(0.000000000000000, 0.000000000000000)
(-0.526233343362516, 0.000000000000000)
(0.220830337618112, -0.477656250512816)
(-0.305403005744404, -0.477656250512816)
(0.100767309386062, 0.400890564600664)
(-0.425466033976454, 0.400890564600664)
```

```
>>> from sage.all import *
>>> s = WordMorphism('1->12,2->13,3->1')
>>> D = s.fixed_point('1')
>>> v = s.pisot_eigenvector_right()
>>> P = WordPaths('123',[(Integer(1),Integer(0),Integer(0)),(Integer(0),
↪Integer(1),Integer(0)),(Integer(0),Integer(0),Integer(1))])
>>> w = P(D[:Integer(200)])
>>> it = w.projected_point_iterator(v)
>>> for i in range(Integer(6)): next(it)
(0.000000000000000, 0.000000000000000)
(-0.526233343362516, 0.000000000000000)
(0.220830337618112, -0.477656250512816)
(-0.305403005744404, -0.477656250512816)
(0.100767309386062, 0.400890564600664)
(-0.425466033976454, 0.400890564600664)
```

Projected points of a 2d path:

```
sage: P = WordPaths('ab','ne')
sage: p = P('aabbabbab')
sage: it = p.projected_point_iterator(ring=RealField(20))
sage: for i in range(8): next(it)
(0.00000)
(0.78087)
(1.5617)
(0.93704)
(0.31235)
(1.0932)
(0.46852)
(-0.15617)
```

```
>>> from sage.all import *
>>> P = WordPaths('ab','ne')
```

(continues on next page)

(continued from previous page)

```
>>> p = P('aabbabbab')
>>> it = p.projected_point_iterator(ring=RealField(Integer(20)))
>>> for i in range(Integer(8)): next(it)
(0.00000)
(0.78087)
(1.5617)
(0.93704)
(0.31235)
(1.0932)
(0.46852)
(-0.15617)
```

start_point()

Return the starting point of `self`.

OUTPUT: vector

EXAMPLES:

```
sage: WordPaths('abcdef')('abcdef').start_point()
(0, 0)
sage: WordPaths('abcdef', steps='cube_grid')('abcdef').start_point()
(0, 0, 0)
sage: P = WordPaths('ab', steps=[(1,0,0,0),(0,1,0,0)])
sage: P('abbba').start_point()
(0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> WordPaths('abcdef')('abcdef').start_point()
(0, 0)
>>> WordPaths('abcdef', steps='cube_grid')('abcdef').start_point()
(0, 0, 0)
>>> P = WordPaths('ab', steps=[(Integer(1),Integer(0),Integer(0),Integer(0)),
↪(Integer(0),Integer(1),Integer(0),Integer(0))])
>>> P('abbba').start_point()
(0, 0, 0, 0)
```

tikz_trajectory()

Return the trajectory of `self` as a `tikz` string.

EXAMPLES:

```
sage: P = WordPaths('abcdef')
sage: p = P('abcde')
sage: p.tikz_trajectory()
'(0.000, 0.000) -- (1.00, 0.000) -- (1.50, 0.866) -- (1.00, 1.73) -- (0.000,
↪1.73) -- (-0.500, 0.866)'
```

```
>>> from sage.all import *
>>> P = WordPaths('abcdef')
>>> p = P('abcde')
>>> p.tikz_trajectory()
```

(continues on next page)

(continued from previous page)

```
'(0.000, 0.000) -- (1.00, 0.000) -- (1.50, 0.866) -- (1.00, 1.73) -- (0.000,␣
↪1.73) -- (-0.500, 0.866)'
```

class sage.combinat.words.paths.**FiniteWordPath_all_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_all*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_all_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_all*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_all_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_all*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_all_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_all*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_all_list**

Bases: *WordDatatype_list*, *FiniteWordPath_all*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_all_str**

Bases: *WordDatatype_str*, *FiniteWordPath_all*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_all_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_all*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_cube_grid**

Bases: *FiniteWordPath_3d*

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_cube_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_cube_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_cube_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_cube_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_list**

Bases: *WordDatatype_list*, *FiniteWordPath_cube_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_str**

Bases: *WordDatatype_str*, *FiniteWordPath_cube_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_cube_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_dyck**

Bases: *FiniteWordPath_2d*

class sage.combinat.words.paths.**FiniteWordPath_dyck_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_dyck*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_dyck_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_dyck*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_dyck_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_dyck*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_dyck_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_dyck*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_dyck_list**

Bases: *WordDatatype_list*, *FiniteWordPath_dyck*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_dyck_str**

Bases: *WordDatatype_str*, *FiniteWordPath_dyck*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_dyck_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_dyck*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid**(*parent*, **args*, ***kwds*)

Bases: *FiniteWordPath_triangle_grid*

INPUT:

- `parent` – a parent object inheriting from Words_all that has the alphabet attribute defined
- `*args`, `**kwds` – arguments accepted by AbstractWord

EXAMPLES:

```
sage: F = WordPaths('abcdef', steps='hexagon'); F
Word Paths on the hexagonal grid
sage: f = F('aaabbbccddef'); f
Path: aaabbbccddef
```

```
>>> from sage.all import *
>>> F = WordPaths('abcdef', steps='hexagon'); F
Word Paths on the hexagonal grid
>>> f = F('aaabbbccddef'); f
Path: aaabbbccddef
```

```
sage: f == loads(dumps(f))
True
```

```
>>> from sage.all import *
>>> f == loads(dumps(f))
True
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_hexagonal_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_hexagonal_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_hexagonal_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_hexagonal_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_list**

Bases: *WordDatatype_list*, *FiniteWordPath_hexagonal_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_str**

Bases: *WordDatatype_str*, *FiniteWordPath_hexagonal_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_hexagonal_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_north_east**

Bases: *FiniteWordPath_2d*

class sage.combinat.words.paths.**FiniteWordPath_north_east_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_north_east*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_north_east_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_north_east*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_north_east_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_north_east*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_north_east_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_north_east*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_north_east_list**

Bases: *WordDatatype_list*, *FiniteWordPath_north_east*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_north_east_str**

Bases: *WordDatatype_str*, *FiniteWordPath_north_east*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_north_east_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_north_east*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_square_grid**

Bases: *FiniteWordPath_2d*

area()

Return the area of a closed path.

INPUT:

- `self` – a closed path

EXAMPLES:

```
sage: P = WordPaths('abAB', steps='square_grid')
sage: P('abAB').area()
1
sage: P('aabbAABB').area()
4
sage: P('aabbABAB').area()
3
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB', steps='square_grid')
>>> P('abAB').area()
1
>>> P('aabbAABB').area()
4
>>> P('aabbABAB').area()
3
```

The area of the Fibonacci tiles:

```
sage: [words.fibonacci_tile(i).area() for i in range(6)]
[1, 5, 29, 169, 985, 5741]
sage: [words.dual_fibonacci_tile(i).area() for i in range(6)]
[1, 5, 29, 169, 985, 5741]
sage: oeis(_)[0]                                # optional -- internet
A001653: Numbers k such that 2*k^2 - 1 is a square.
sage: _.first_terms()                           # optional -- internet
(1,
 5,
 29,
 169,
 985,
 5741,
 33461,
 195025,
 1136689,
 6625109,
 38613965,
 225058681,
 1311738121,
 7645370045,
 44560482149,
 259717522849,
 1513744654945,
 8822750406821,
 51422757785981,
 299713796309065,
```

(continues on next page)

(continued from previous page)

```
 1746860020068409,
 10181446324101389,
 59341817924539925)
```

```
>>> from sage.all import *
>>> [words.fibonacci_tile(i).area() for i in range(Integer(6))]
[1, 5, 29, 169, 985, 5741]
>>> [words.dual_fibonacci_tile(i).area() for i in range(Integer(6))]
[1, 5, 29, 169, 985, 5741]
>>> oeis(_)[Integer(0)]                                   # optional -- internet
A001653: Numbers k such that 2*k^2 - 1 is a square.
>>> _.first_terms()                             # optional -- internet
(1,
 5,
 29,
 169,
 985,
 5741,
 33461,
 195025,
 1136689,
 6625109,
 38613965,
 225058681,
 1311738121,
 7645370045,
 44560482149,
 259717522849,
 1513744654945,
 8822750406821,
 51422757785981,
 299713796309065,
 1746860020068409,
 10181446324101389,
 59341817924539925)
```

is_closed()

Return whether `self` represents a closed path.

EXAMPLES:

```
sage: P = WordPaths('abAB', steps='square_grid')
sage: P('aA').is_closed()
True
sage: P('abAB').is_closed()
True
sage: P('ababAABB').is_closed()
True
sage: P('aaabbbAABB').is_closed()
False
sage: P('ab').is_closed()
False
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB', steps='square_grid')
>>> P('aA').is_closed()
True
>>> P('abAB').is_closed()
True
>>> P('ababAABB').is_closed()
True
>>> P('aaabbbAABB').is_closed()
False
>>> P('ab').is_closed()
False
```

is_simple()

Return whether the path is simple.

A path is simple if all its points are distinct.

If the path is closed, the last point is not considered.

Note: The linear algorithm described in the thesis of Xavier Provençal should be implemented here.

EXAMPLES:

```
sage: P = WordPaths('abAB', steps='square_grid')
sage: P('abab').is_simple()
True
sage: P('abAB').is_simple()
True
sage: P('abA').is_simple()
True
sage: P('aabABB').is_simple()
False
sage: P().is_simple()
True
sage: P('A').is_simple()
True
sage: P('aA').is_simple()
True
sage: P('aaA').is_simple()
False
```

```
>>> from sage.all import *
>>> P = WordPaths('abAB', steps='square_grid')
>>> P('abab').is_simple()
True
>>> P('abAB').is_simple()
True
>>> P('abA').is_simple()
True
```

(continues on next page)

(continued from previous page)

```
>>> P('aabABB').is_simple()
False
>>> P().is_simple()
True
>>> P('A').is_simple()
True
>>> P('aA').is_simple()
True
>>> P('aaA').is_simple()
False
```

REFERENCES:

- Provençal, X., *Combinatoires des mots, géometrie discrète et pavages*, Thèse de doctorat en Mathématiques, Montréal, UQAM, septembre 2008, 115 pages.

tikz_trajectory()

Return the trajectory of `self` as a `tikz` string.

EXAMPLES:

```
sage: f = words.fibonacci_tile(1)
sage: f.tikz_trajectory()
'(0, 0) -- (0, -1) -- (-1, -1) -- (-1, -2) -- (0, -2) -- (0, -3) -- (1, -3) --
↪ (1, -2) -- (2, -2) -- (2, -1) -- (1, -1) -- (1, 0) -- (0, 0)'
```

```
>>> from sage.all import *
>>> f = words.fibonacci_tile(Integer(1))
>>> f.tikz_trajectory()
'(0, 0) -- (0, -1) -- (-1, -1) -- (-1, -2) -- (0, -2) -- (0, -3) -- (1, -3) --
↪ (1, -2) -- (2, -2) -- (2, -1) -- (1, -1) -- (1, 0) -- (0, 0)'
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_square_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_square_grid_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_square_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_square_grid_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_square_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_square_grid_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_square_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_square_grid_list**

Bases: *WordDatatype_list*, *FiniteWordPath_square_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_square_grid_str**

Bases: *WordDatatype_str*, *FiniteWordPath_square_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_square_grid_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_square_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid**

Bases: *FiniteWordPath_2d*

xmax()

Return the maximum of the x-coordinates of the path.

EXAMPLES:

```
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.xmax()
4.50000000000000
sage: w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
sage: w.xmax()
4.00000000000000
```

```
>>> from sage.all import *
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.xmax()
4.50000000000000
>>> w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
>>> w.xmax()
4.00000000000000
```

xmin()

Return the minimum of the x-coordinates of the path.

EXAMPLES:

```
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.xmin()
0.000000000000000
sage: w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
sage: w.xmin()
-3.00000000000000
```

```
>>> from sage.all import *
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.xmin()
0.000000000000000
>>> w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
>>> w.xmin()
-3.00000000000000
```

ymax()

Return the maximum of the y-coordinates of the path.

EXAMPLES:

```
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.ymax()
2.59807621135332
sage: w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
```

(continues on next page)

(continued from previous page)

```
sage: w.ymax()
8.66025403784439
```

```
>>> from sage.all import *
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.ymax()
2.59807621135332
>>> w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
>>> w.ymax()
8.66025403784439
```

ymin()

Return the minimum of the y-coordinates of the path.

EXAMPLES:

```
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.ymin()
0.000000000000000
sage: w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
sage: w.ymin()
-0.866025403784439
```

```
>>> from sage.all import *
>>> w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
>>> w.ymin()
0.000000000000000
>>> w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
>>> w.ymin()
-0.866025403784439
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWordPath_triangle_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWordPath_triangle_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWordPath_triangle_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWordPath_triangle_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_list**

Bases: *WordDatatype_list*, *FiniteWordPath_triangle_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_str**

Bases: *WordDatatype_str*, *FiniteWordPath_triangle_grid*, *FiniteWord_class*

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_tuple**

Bases: *WordDatatype_tuple*, *FiniteWordPath_triangle_grid*, *FiniteWord_class*

sage.combinat.words.paths.**WordPaths**(*alphabet*, *steps=None*)

Return the combinatorial class of paths of the given type of steps.

INPUT:

- `alphabet` – ordered alphabet
- `steps` – (default: `None`) it can be one of the following:
  - an iterable ordered container of as many vectors as there are letters in the alphabet. The vectors are associated to the letters according to their order in steps. The vectors can be a tuple or anything that can be passed to vector function.
  - an iterable ordered container of k vectors where k is half the size of alphabet. The vectors and their opposites are associated to the letters according to their order in steps (given vectors first, opposite vectors after).
  - `None` – in this case, the type of steps are guessed from the length of alphabet
  - `'square_grid'` or `'square'` – (default when size of alphabet is 4) The order is : East, North, West, South.
  - `'triangle_grid'` or `'triangle'`
  - `'hexagonal_grid'` or `'hexagon'` – (default when size of alphabet is 6)
  - `'cube_grid'` or `'cube'`
  - `'north_east'`, `'ne'` or `'NE'` – (the default when size of alphabet is 2)
  - `'dyck'`

OUTPUT: the combinatorial class of all paths of the given type

EXAMPLES:

The steps can be given explicitly:

```
sage: WordPaths('abc', steps=[(1,2), (-1,4), (0,-3)])
Word Paths over 3 steps
```

```
>>> from sage.all import *
>>> WordPaths('abc', steps=[(Integer(1),Integer(2)), (-Integer(1),Integer(4)),␣
↪(Integer(0),-Integer(3))])
Word Paths over 3 steps
```

Different type of input alphabet:

```
sage: WordPaths(range(3), steps=[(1,2), (-1,4), (0,-3)])
Word Paths over 3 steps
sage: WordPaths(['cric','crac','croc'], steps=[(1,2), (1,4), (0,3)])
Word Paths over 3 steps
```

```
>>> from sage.all import *
>>> WordPaths(range(Integer(3)), steps=[(Integer(1),Integer(2)), (-Integer(1),
↪Integer(4)), (Integer(0),-Integer(3))])
Word Paths over 3 steps
>>> WordPaths(['cric','crac','croc'], steps=[(Integer(1),Integer(2)), (Integer(1),
↪Integer(4)), (Integer(0),Integer(3))])
Word Paths over 3 steps
```

Directions can be in three dimensions as well:

```
sage: WordPaths('ab', steps=[(1,2,2),(-1,4,2)])
Word Paths over 2 steps
```

```
>>> from sage.all import *
>>> WordPaths('ab', steps=[(Integer(1),Integer(2),Integer(2)),(-Integer(1),
↪Integer(4),Integer(2))])
Word Paths over 2 steps
```

When the number of given steps is half the size of alphabet, the opposite of vectors are used:

```
sage: P = WordPaths('abcd', [(1,0), (0,1)])
sage: P
Word Paths over 4 steps
sage: sorted(P.letters_to_steps().items())
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
```

```
>>> from sage.all import *
>>> P = WordPaths('abcd', [(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> P
Word Paths over 4 steps
>>> sorted(P.letters_to_steps().items())
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
```

When no steps are given, default classes are returned:

```
sage: WordPaths('ab')
Word Paths in North and East steps
sage: WordPaths(range(4))
Word Paths on the square grid
sage: WordPaths(range(6))
Word Paths on the hexagonal grid
```

```
>>> from sage.all import *
>>> WordPaths('ab')
Word Paths in North and East steps
>>> WordPaths(range(Integer(4)))
Word Paths on the square grid
>>> WordPaths(range(Integer(6)))
Word Paths on the hexagonal grid
```

There are many type of built-in steps…

On a two letters alphabet:

```
sage: WordPaths('ab', steps='north_east')
Word Paths in North and East steps
sage: WordPaths('()', steps='dyck')
Finite Dyck paths
```

```
>>> from sage.all import *
>>> WordPaths('ab', steps='north_east')
Word Paths in North and East steps
>>> WordPaths('()', steps='dyck')
Finite Dyck paths
```

On a four letters alphabet:

```
sage: WordPaths('ruld', steps='square_grid')
Word Paths on the square grid
```

```
>>> from sage.all import *
>>> WordPaths('ruld', steps='square_grid')
Word Paths on the square grid
```

On a six letters alphabet:

```
sage: WordPaths('abcdef', steps='hexagonal_grid')
Word Paths on the hexagonal grid
sage: WordPaths('abcdef', steps='triangle_grid')
Word Paths on the triangle grid
sage: WordPaths('abcdef', steps='cube_grid')
Word Paths on the cube grid
```

```
>>> from sage.all import *
>>> WordPaths('abcdef', steps='hexagonal_grid')
Word Paths on the hexagonal grid
>>> WordPaths('abcdef', steps='triangle_grid')
Word Paths on the triangle grid
>>> WordPaths('abcdef', steps='cube_grid')
Word Paths on the cube grid
```

class sage.combinat.words.paths.**WordPaths_all**(*alphabet*, *steps*)

Bases: `FiniteWords`

The combinatorial class of all paths, i.e of all words over an alphabet where each letter is mapped to a step (a vector).

letters_to_steps()

Return the dictionary mapping letters to vectors (steps).

EXAMPLES:

```
sage: d = WordPaths('ab').letters_to_steps()
sage: sorted(d.items())
[('a', (0, 1)), ('b', (1, 0))]
sage: d = WordPaths('abcd').letters_to_steps()
sage: sorted(d.items())
```

(continues on next page)

(continued from previous page)

```
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
sage: d = WordPaths('abcdef').letters_to_steps()
sage: sorted(d.items())
[('a', (1, 0)),
 ('b', (1/2, 1/2*sqrt3)),
 ('c', (-1/2, 1/2*sqrt3)),
 ('d', (-1, 0)),
 ('e', (-1/2, -1/2*sqrt3)),
 ('f', (1/2, -1/2*sqrt3))]
```

```
>>> from sage.all import *
>>> d = WordPaths('ab').letters_to_steps()
>>> sorted(d.items())
[('a', (0, 1)), ('b', (1, 0))]
>>> d = WordPaths('abcd').letters_to_steps()
>>> sorted(d.items())
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
>>> d = WordPaths('abcdef').letters_to_steps()
>>> sorted(d.items())
[('a', (1, 0)),
 ('b', (1/2, 1/2*sqrt3)),
 ('c', (-1/2, 1/2*sqrt3)),
 ('d', (-1, 0)),
 ('e', (-1/2, -1/2*sqrt3)),
 ('f', (1/2, -1/2*sqrt3))]
```

vector_space()

Return the vector space over which the steps of the paths are defined.

EXAMPLES:

```
sage: WordPaths('ab',steps='dyck').vector_space()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: WordPaths('ab',steps='north_east').vector_space()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: WordPaths('abcd',steps='square_grid').vector_space()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: WordPaths('abcdef',steps='hexagonal_grid').vector_space()
Vector space of dimension 2 over Number Field in sqrt3 with defining␣
↪polynomial x^2 - 3 with sqrt3 = 1.732050807568878?
sage: WordPaths('abcdef',steps='cube_grid').vector_space()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: WordPaths('abcdef',steps='triangle_grid').vector_space()
Vector space of dimension 2 over Number Field in sqrt3 with defining␣
↪polynomial x^2 - 3 with sqrt3 = 1.732050807568878?
```

```
>>> from sage.all import *
>>> WordPaths('ab',steps='dyck').vector_space()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
>>> WordPaths('ab',steps='north_east').vector_space()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
>>> WordPaths('abcd',steps='square_grid').vector_space()
```

(continues on next page)

(continued from previous page)

```
Ambient free module of rank 2 over the principal ideal domain Integer Ring
>>> WordPaths('abcdef',steps='hexagonal_grid').vector_space()
Vector space of dimension 2 over Number Field in sqrt3 with defining
↪polynomial x^2 - 3 with sqrt3 = 1.732050807568878?
>>> WordPaths('abcdef',steps='cube_grid').vector_space()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> WordPaths('abcdef',steps='triangle_grid').vector_space()
Vector space of dimension 2 over Number Field in sqrt3 with defining
↪polynomial x^2 - 3 with sqrt3 = 1.732050807568878?
```

class sage.combinat.words.paths.**WordPaths_cube_grid**(*alphabet*)

Bases: *WordPaths_all*

The combinatorial class of all paths on the cube grid.

class sage.combinat.words.paths.**WordPaths_dyck**(*alphabet*)

Bases: *WordPaths_all*

The combinatorial class of all Dyck paths.

class sage.combinat.words.paths.**WordPaths_hexagonal_grid**(*alphabet*)

Bases: *WordPaths_triangle_grid*

The combinatorial class of all paths on the hexagonal grid.

class sage.combinat.words.paths.**WordPaths_north_east**(*alphabet*)

Bases: *WordPaths_all*

The combinatorial class of all paths using North and East directions.

class sage.combinat.words.paths.**WordPaths_square_grid**(*alphabet*)

Bases: *WordPaths_all*

The combinatorial class of all paths on the square grid.

class sage.combinat.words.paths.**WordPaths_triangle_grid**(*alphabet*)

Bases: *WordPaths_all*

The combinatorial class of all paths on the triangle grid.

5.1.363 Shuffle product of words

See also

The module *sage.combinat.shuffle* contains a more general implementation of shuffle product.

class sage.combinat.words.shuffle_product.**ShuffleProduct_shifted**(*w1*, *w2*, *check=True*)

Bases: *ShuffleProduct_w1w2*

Shifted shuffle product of `w1` with `w2`.

This is the shuffle product of `w1` with the word obtained by adding the length of `w1` to every letter of `w2`.

Note that this class is meant to be used for words; it misbehaves when `w1` is a permutation or composition.

INPUT:

- `check` – boolean (default: `True`); whether to check that all words in the shuffle product belong to the correct parent

EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_shifted
sage: w, u = Word([1,2]), Word([3,4])
sage: S = ShuffleProduct_shifted(w,u)
sage: S == loads(dumps(S))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.words.shuffle_product import ShuffleProduct_shifted
>>> w, u = Word([Integer(1),Integer(2)]), Word([Integer(3),Integer(4)])
>>> S = ShuffleProduct_shifted(w,u)
>>> S == loads(dumps(S))
True
```

class sage.combinat.words.shuffle_product.**ShuffleProduct_w1w2**(*w1*, *w2*, *check=True*)

Bases: `Parent`, `UniqueRepresentation`

The shuffle product of the two words `w1` and `w2`.

If u and v are two words, then the *shuffle product* of u and v is a certain multiset of words defined as follows: Let a and b be the lengths of u and v, respectively. For every a-element subset I of $\{1, 2, \cdots, a+b\}$, let $w(I)$ be the length-$a+b$ word such that:

- for every $1 \leq k \leq a$, the i_k-th letter of $w(I)$ is the k-th letter of u, where i_k is the k-th smallest element of I;
- for every $1 \leq l \leq b$, the j_l-th letter of $w(I)$ is the l-th letter of v, where j_l is the l-th smallest element of $\{1, 2, \cdots, a+b\} \setminus I$.

The shuffle product of u and v is then the multiset of all $w(I)$ with I ranging over the a-element subsets of $\{1, 2, \cdots, a+b\}$.

INPUT:

- `check` – boolean (default: `True`); whether to check that all words in the shuffle product belong to the correct parent

EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_w1w2
sage: W = Words([1,2,3,4])
sage: s = ShuffleProduct_w1w2(W([1,2]),W([3,4]))
sage: sorted(s)
[word: 1234, word: 1324, word: 1342, word: 3124,
 word: 3142, word: 3412]
sage: s == loads(dumps(s))
True
sage: TestSuite(s).run()

sage: s = ShuffleProduct_w1w2(W([1,4,3]),W([2]))
sage: sorted(s)
[word: 1243, word: 1423, word: 1432, word: 2143]

sage: s = ShuffleProduct_w1w2(W([1,4,3]),W([]))
```

(continues on next page)

(continued from previous page)

```
sage: sorted(s)
[word: 143]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.shuffle_product import ShuffleProduct_w1w2
>>> W = Words([Integer(1),Integer(2),Integer(3),Integer(4)])
>>> s = ShuffleProduct_w1w2(W([Integer(1),Integer(2)]),W([Integer(3),Integer(4)]))
>>> sorted(s)
[word: 1234, word: 1324, word: 1342, word: 3124,
 word: 3142, word: 3412]
>>> s == loads(dumps(s))
True
>>> TestSuite(s).run()

>>> s = ShuffleProduct_w1w2(W([Integer(1),Integer(4),Integer(3)]),W([Integer(2)]))
>>> sorted(s)
[word: 1243, word: 1423, word: 1432, word: 2143]

>>> s = ShuffleProduct_w1w2(W([Integer(1),Integer(4),Integer(3)]),W([]))
>>> sorted(s)
[word: 143]
```

cardinality()

Return the number of words in the shuffle product of `w1` and `w2`.

This is understood as a multiset cardinality, not as a set cardinality; it does not count the distinct words only.

It is given by $\binom{l_1+l_2}{l_1}$, where l_1 is the length of `w1` and where l_2 is the length of `w2`.

EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_w1w2
sage: w, u = map(Words("abcd"), ["ab", "cd"])
sage: S = ShuffleProduct_w1w2(w,u)
sage: S.cardinality()
6

sage: w, u = map(Words("ab"), ["ab", "ab"])
sage: S = ShuffleProduct_w1w2(w,u)
sage: S.cardinality()
6
```

```
>>> from sage.all import *
>>> from sage.combinat.words.shuffle_product import ShuffleProduct_w1w2
>>> w, u = map(Words("abcd"), ["ab", "cd"])
>>> S = ShuffleProduct_w1w2(w,u)
>>> S.cardinality()
6

>>> w, u = map(Words("ab"), ["ab", "ab"])
>>> S = ShuffleProduct_w1w2(w,u)
>>> S.cardinality()
6
```

5.1.364 Suffix Tries and Suffix Trees

class sage.combinat.words.suffix_trees.**DecoratedSuffixTree**(*w*)

Bases: *ImplicitSuffixTree*

The decorated suffix tree of a word.

A *decorated suffix tree* of a word w is the suffix tree of w marked with the end point of all squares in the w.

The symbol `$` is appended to `w` to ensure that each final state is a leaf of the suffix tree.

INPUT:

- `w` – a finite word

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import DecoratedSuffixTree
sage: w = Word('0011001')
sage: DecoratedSuffixTree(w)
Decorated suffix tree of : 0011001$
sage: w = Word('0011001', '01')
sage: DecoratedSuffixTree(w)
Decorated suffix tree of : 0011001$
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import DecoratedSuffixTree
>>> w = Word('0011001')
>>> DecoratedSuffixTree(w)
Decorated suffix tree of : 0011001$
>>> w = Word('0011001', '01')
>>> DecoratedSuffixTree(w)
Decorated suffix tree of : 0011001$
```

ALGORITHM:

When using `'pair'` as output, the squares are retrieved in linear time. The algorithm is an implementation of the one proposed in [DS2004].

square_vocabulary(*output='pair'*)

Return the list of distinct squares of `self.word`.

Two types of outputs are available $pair$ and $word$. The algorithm is only truly linear if $output$ is set to $pair$. A pair is a tuple (i, l) that indicates the factor `self.word()[i:i+l]`. The option `'word'` return word objects.

INPUT:

- `output` – (default: `'pair'`) either `'pair'` or `'word'`

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import DecoratedSuffixTree
sage: w = Word('aabb')
sage: sorted(DecoratedSuffixTree(w).square_vocabulary())
[(0, 0), (0, 2), (2, 2)]
sage: w = Word('00110011010')
sage: sorted(DecoratedSuffixTree(w).square_vocabulary(output='word'))
[word: , word: 00, word: 00110011, word: 01100110, word: 1010, word: 11]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import DecoratedSuffixTree
>>> w = Word('aabb')
>>> sorted(DecoratedSuffixTree(w).square_vocabulary())
[(0, 0), (0, 2), (2, 2)]
>>> w = Word('00110011010')
>>> sorted(DecoratedSuffixTree(w).square_vocabulary(output='word'))
[word: , word: 00, word: 00110011, word: 01100110, word: 1010, word: 11]
```

class sage.combinat.words.suffix_trees.**ImplicitSuffixTree**(*word*)

Bases: SageObject

Construct the implicit suffix tree of a word w.

The suffix tree of a word w is a compactification of the suffix trie for w. The compactification removes all nodes that have exactly one incoming edge and exactly one outgoing edge. It consists of two components: a tree and a word. Thus, instead of labelling the edges by factors of w, we can labelled them by indices of the occurrence of the factors in w.

The following is a straightforward implementation of Ukkonen's on-line algorithm for constructing the implicit suffix tree [Ukko1995]. It constructs the suffix tree for w[:i] from that of w[:i-1].

GENERAL IDEA. The suffix tree of w[:i+1] can be obtained from that of w[:i] by visiting each node corresponding to a suffix of w[:i] and modifying the tree by applying one of two rules (either append a new node to the tree, or split an edge into two). The "active state" is the node where the algorithm begins and the "suffix link" carries us to the next node that needs to be dealt with.

TREE. The tree is modelled as an automaton, which is stored as a dictionary of dictionaries: it is keyed by the nodes of the tree, and the corresponding dictionary is keyed by pairs (i, j) of integers representing the word w[i-1:j]. This makes it faster to look up a particular transition beginning at a specific node.

STATES/NODES. The states will always be -1, 0, 1, …, n. The state -1 is special and is only used for the purposes of the algorithm. All transitions map -1 to 0, so this information is not explicitly stored in the transition function.

EXPLICIT/IMPLICIT NODES. By definition, some of the nodes will not be states, but merely locations along an edge; these are called implicit nodes. A node r (implicit or explicit) is referenced as a pair (s,(k,p)) where s is an ancestor of r and w[k-1:p] is the word read by transitioning from s to r in the suffix trie. A reference pair is canonical if s is the closest ancestor of r.

SUFFIX LINK. The algorithm makes use of a map from (some) nodes to other nodes, called the suffix link. This is stored as a dictionary.

ACTIVE STATE. We store as ._active_state the active state of the tree, the state where the algorithm will begin when processing the next letter.

RUNNING TIME. The running time and storage space of the algorithm is linear in the length of the word w (whereas for a suffix tree it is quadratic).

REFERENCES:

- [Ukko1995]

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("aco")("cacao")
sage: t = ImplicitSuffixTree(w); t
Implicit Suffix Tree of the word: cacao
sage: ababb = Words([0,1])([0,1,0,1,1])
```

(continues on next page)

(continued from previous page)

```
sage: s = ImplicitSuffixTree(ababb); s
Implicit Suffix Tree of the word: 01011
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> w = Words("aco")("cacao")
>>> t = ImplicitSuffixTree(w); t
Implicit Suffix Tree of the word: cacao
>>> ababb = Words([Integer(0),Integer(1)])([Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(1)])
>>> s = ImplicitSuffixTree(ababb); s
Implicit Suffix Tree of the word: 01011
```

LZ_decomposition()

Return a list of index of the beginning of the block of the Lempel-Ziv decomposition of `self.word`

The *Lempel-Ziv decomposition* is the factorisation $u_1 ... u_k$ of a word $w = x_1 ... x_n$ such that u_i is the longest prefix of $u_i ... u_k$ that has an occurrence starting before u_i or a letter if this prefix is empty.

OUTPUT:

Return a list `iB` of index such that the blocks of the decomposition are `self.word()[iB[k]:iB[k+1]]`

EXAMPLES:

```
sage: w = Word('abababb')
sage: T = w.suffix_tree()
sage: T.LZ_decomposition()
[0, 1, 2, 6, 7]
sage: w = Word('abaababacabba')
sage: T = w.suffix_tree()
sage: T.LZ_decomposition()
[0, 1, 2, 3, 6, 8, 9, 11, 13]
sage: w = Word([0, 0, 0, 1, 1, 0, 1])
sage: T = w.suffix_tree()
sage: T.LZ_decomposition()
[0, 1, 3, 4, 5, 7]
sage: w = Word('0000100101')
sage: T = w.suffix_tree()
sage: T.LZ_decomposition()
[0, 1, 4, 5, 9, 10]
```

```
>>> from sage.all import *
>>> w = Word('abababb')
>>> T = w.suffix_tree()
>>> T.LZ_decomposition()
[0, 1, 2, 6, 7]
>>> w = Word('abaababacabba')
>>> T = w.suffix_tree()
>>> T.LZ_decomposition()
[0, 1, 2, 3, 6, 8, 9, 11, 13]
>>> w = Word([Integer(0), Integer(0), Integer(0), Integer(1), Integer(1),␣
↪Integer(0), Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> T = w.suffix_tree()
>>> T.LZ_decomposition()
[0, 1, 3, 4, 5, 7]
>>> w = Word('0000100101')
>>> T = w.suffix_tree()
>>> T.LZ_decomposition()
[0, 1, 4, 5, 9, 10]
```

active_state()

Return the active state of the suffix tree.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.active_state()
(0, (6, 6))
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> t = ImplicitSuffixTree(W([Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(2)]))
>>> t.active_state()
(0, (6, 6))
```

edge_iterator()

Return an iterator over the edges of the suffix tree.

The edge from u to v labelled by (i, j) is yielded as the tuple $(u, v, (i, j))$.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: sorted( ImplicitSuffixTree(Word("aaaaa")).edge_iterator() )
[(0, 1, (0, None))]
sage: sorted( ImplicitSuffixTree(Word([0,1,0,1])).edge_iterator() )
[(0, 1, (0, None)), (0, 2, (1, None))]
sage: sorted( ImplicitSuffixTree(Word()).edge_iterator() )
[]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> sorted( ImplicitSuffixTree(Word("aaaaa")).edge_iterator() )
[(0, 1, (0, None))]
>>> sorted( ImplicitSuffixTree(Word([Integer(0),Integer(1),Integer(0),
↪Integer(1)])).edge_iterator() )
[(0, 1, (0, None)), (0, 2, (1, None))]
>>> sorted( ImplicitSuffixTree(Word()).edge_iterator() )
[]
```

factor_iterator(*n=None*)

Generate distinct factors of `self`.

INPUT:

- `n` – integer or `None`

OUTPUT:

If `n` is an integer, returns an iterator over all distinct factors of length `n`. If `n` is `None`, returns an iterator generating all distinct factors.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator() )
[word: , word: a, word: ac, word: aca, word: acao, word: ao, word: c, word:
↪ca, word: cac, word: caca, word: cacao, word: cao, word: o]
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator(1) )
[word: a, word: c, word: o]
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator(2) )
[word: ac, word: ao, word: ca]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator() )
[word: , word: 0, word: 00, word: 000]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator(2) )
[word: 00]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator(0) )
[word: ]
sage: sorted( ImplicitSuffixTree(Word()).factor_iterator() )
[word: ]
sage: sorted( ImplicitSuffixTree(Word()).factor_iterator(2) )
[]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator() )
[word: , word: a, word: ac, word: aca, word: acao, word: ao, word: c, word:
↪ca, word: cac, word: caca, word: cacao, word: cao, word: o]
>>> sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator(Integer(1)) )
[word: a, word: c, word: o]
>>> sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator(Integer(2)) )
[word: ac, word: ao, word: ca]
>>> sorted( ImplicitSuffixTree(Word([Integer(0),Integer(0),Integer(0)])).
↪factor_iterator() )
[word: , word: 0, word: 00, word: 000]
>>> sorted( ImplicitSuffixTree(Word([Integer(0),Integer(0),Integer(0)])).
↪factor_iterator(Integer(2)) )
[word: 00]
>>> sorted( ImplicitSuffixTree(Word([Integer(0),Integer(0),Integer(0)])).
↪factor_iterator(Integer(0)) )
[word: ]
>>> sorted( ImplicitSuffixTree(Word()).factor_iterator() )
[word: ]
>>> sorted( ImplicitSuffixTree(Word()).factor_iterator(Integer(2)) )
[]
```

leftmost_covering_set()

Compute the leftmost covering set of square pairs in `self.word()`. Return a square as a pair `(i,l)` designating factor `self.word()[i:i+l]`.

A leftmost covering set is a set such that the leftmost occurrence (j, l) of a square in `self.word()` is covered by a pair (i, l) in the set for all types of squares. We say that (j, l) is covered by (i, l) if (i, l) (i+1,l), ldots, (j,l)` are all squares.

The set is returned in the form of a list `P` such that `P[i]` contains all the lengths of squares starting at `i` in the set. The lists `P[i]` are sorted in decreasing order.

The algorithm used is described in [DS2004].

EXAMPLES:

```
sage: w = Word('abaabaabbaaabaaba')
sage: T = w.suffix_tree()
sage: T.leftmost_covering_set()
[[6], [6], [2], [], [], [], [], [2], [], [], [6, 2], [], [], [], [], [], []]
sage: w = Word('abaca')
sage: T = w.suffix_tree()
sage: T.leftmost_covering_set()
[[], [], [], [], []]
sage: T = Word('aaaaa').suffix_tree()
sage: T.leftmost_covering_set()
[[4, 2], [], [], [], []]
```

```
>>> from sage.all import *
>>> w = Word('abaabaabbaaabaaba')
>>> T = w.suffix_tree()
>>> T.leftmost_covering_set()
[[6], [6], [2], [], [], [], [], [2], [], [], [6, 2], [], [], [], [], [], []]
>>> w = Word('abaca')
>>> T = w.suffix_tree()
>>> T.leftmost_covering_set()
[[], [], [], [], []]
>>> T = Word('aaaaa').suffix_tree()
>>> T.leftmost_covering_set()
[[4, 2], [], [], [], []]
```

number_of_factors(*n=None*)

Count the number of distinct factors of `self.word()`.

INPUT:

- `n` – integer or `None`

OUTPUT:

If `n` is an integer, returns the number of distinct factors of length `n`. If `n` is `None`, returns the total number of distinct factors.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: t = ImplicitSuffixTree(Word([1,2,1,3,1,2,1]))
sage: t.number_of_factors()
22
sage: t.number_of_factors(1)
3
sage: t.number_of_factors(9)
```

(continues on next page)

(continued from previous page)

```
0
sage: t.number_of_factors(0)
1
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> t = ImplicitSuffixTree(Word([Integer(1),Integer(2),Integer(1),Integer(3),
↪Integer(1),Integer(2),Integer(1)]))
>>> t.number_of_factors()
22
>>> t.number_of_factors(Integer(1))
3
>>> t.number_of_factors(Integer(9))
0
>>> t.number_of_factors(Integer(0))
1
```

```
sage: t = ImplicitSuffixTree(Word("cacao"))
sage: t.number_of_factors()
13
sage: list(map(t.number_of_factors, range(10)))
[1, 3, 3, 3, 2, 1, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> t = ImplicitSuffixTree(Word("cacao"))
>>> t.number_of_factors()
13
>>> list(map(t.number_of_factors, range(Integer(10))))
[1, 3, 3, 3, 2, 1, 0, 0, 0, 0]
```

```
sage: t = ImplicitSuffixTree(Word("c"*1000))
sage: t.number_of_factors()
1001
sage: t.number_of_factors(17)
1
sage: t.number_of_factors(0)
1
```

```
>>> from sage.all import *
>>> t = ImplicitSuffixTree(Word("c"*Integer(1000)))
>>> t.number_of_factors()
1001
>>> t.number_of_factors(Integer(17))
1
>>> t.number_of_factors(Integer(0))
1
```

```
sage: ImplicitSuffixTree(Word()).number_of_factors()
1
```

```
>>> from sage.all import *
>>> ImplicitSuffixTree(Word()).number_of_factors()
1
```

```
sage: blueberry = ImplicitSuffixTree(Word("blueberry"))
sage: blueberry.number_of_factors()
43
sage: list(map(blueberry.number_of_factors, range(10)))
[1, 6, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> blueberry = ImplicitSuffixTree(Word("blueberry"))
>>> blueberry.number_of_factors()
43
>>> list(map(blueberry.number_of_factors, range(Integer(10))))
[1, 6, 8, 7, 6, 5, 4, 3, 2, 1]
```

plot(*word_labels=False*, *layout='tree'*, *tree_root=0*, *tree_orientation='up'*, *vertex_colors=None*, *edge_labels=True*, **args*, ***kwds*)

Return a Graphics object corresponding to the transition graph of the suffix tree.

INPUT:

- `word_labels` – boolean (default: `False`); if `False`, labels the edges by pairs (i, j); if `True`, labels the edges by `word[i:j]`
- `layout` – (default: `'tree'`)
- `tree_root` – (default: 0)
- `tree_orientation` – (default: `'up'`)
- `vertex_colors` – (default: `None`)
- `edge_labels` – (default: `True`)

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: ImplicitSuffixTree(Word('cacao')).plot(word_labels=True)              #␣
↪needs sage.graphs sage.plot
Graphics object consisting of 23 graphics primitives
sage: ImplicitSuffixTree(Word('cacao')).plot(word_labels=False)             #␣
↪needs sage.graphs sage.plot
Graphics object consisting of 23 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> ImplicitSuffixTree(Word('cacao')).plot(word_labels=True)              #␣
↪needs sage.graphs sage.plot
Graphics object consisting of 23 graphics primitives
>>> ImplicitSuffixTree(Word('cacao')).plot(word_labels=False)             #␣
↪needs sage.graphs sage.plot
Graphics object consisting of 23 graphics primitives
```

process_letter(*letter*)

Modify the current implicit suffix tree producing the implicit suffix tree for `self.word() + letter`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("aco")("cacao")
sage: t = ImplicitSuffixTree(w[:-1]); t
Implicit Suffix Tree of the word: caca
sage: t.process_letter(w[-1]); t
Implicit Suffix Tree of the word: cacao
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> w = Words("aco")("cacao")
>>> t = ImplicitSuffixTree(w[:-Integer(1)]); t
Implicit Suffix Tree of the word: caca
>>> t.process_letter(w[-Integer(1)]); t
Implicit Suffix Tree of the word: cacao
```

```
sage: W = Words([0,1])
sage: s = ImplicitSuffixTree(W([0,1,0,1])); s
Implicit Suffix Tree of the word: 0101
sage: s.process_letter(W([1])[0]); s
Implicit Suffix Tree of the word: 01011
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1)])
>>> s = ImplicitSuffixTree(W([Integer(0),Integer(1),Integer(0),Integer(1)]));␣
↪s
Implicit Suffix Tree of the word: 0101
>>> s.process_letter(W([Integer(1)])[Integer(0)]); s
Implicit Suffix Tree of the word: 01011
```

show(*word_labels=None*, **args*, ***kwds*)

Display the output of `plot()`.

INPUT:

- `word_labels` – (default: `None`) if `False`, labels the edges by pairs (i, j); if `True`, labels the edges by `word[i:j]`

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("cao")("cacao")
sage: t = ImplicitSuffixTree(w)
sage: t.show(word_labels=True)                                              #␣
↪needs sage.plot
sage: t.show(word_labels=False)                                             #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
```

(continues on next page)

(continued from previous page)

```
>>> w = Words("cao")("cacao")
>>> t = ImplicitSuffixTree(w)
>>> t.show(word_labels=True)                                                  #␣
↪needs sage.plot
>>> t.show(word_labels=False)                                                 #␣
↪needs sage.plot
```

states()

Return the states (explicit nodes) of the suffix tree.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.states()
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> t = ImplicitSuffixTree(W([Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(2)]))
>>> t.states()
[0, 1, 2, 3, 4, 5, 6, 7]
```

suffix_link(*state*)

Evaluate the suffix link map of the implicit suffix tree on `state`.

Note that the suffix link is not defined for all states.

The suffix link of a state x' that corresponds to the suffix x is defined to be -1 is x' is the root (0) and y' otherwise, where y' is the state corresponding to the suffix `x[1:]`.

INPUT:

- `state` – a state

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.suffix_link(3)
5
sage: t.suffix_link(5)
0
sage: t.suffix_link(0)
-1
sage: t.suffix_link(-1)
Traceback (most recent call last):
    ...
TypeError: there is no suffix link from -1
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> t = ImplicitSuffixTree(W([Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(2)]))
>>> t.suffix_link(Integer(3))
5
>>> t.suffix_link(Integer(5))
0
>>> t.suffix_link(Integer(0))
-1
>>> t.suffix_link(-Integer(1))
Traceback (most recent call last):
    ...
TypeError: there is no suffix link from -1
```

suffix_walk(*edge*, *l*)

Return the state of "w" if the input state is "aw".

If the input state (edge, l) is path labeled "aw" with "a" a letter, the output is the state which is path labeled "w".

INPUT:

- edge – the edge containing the state
- l – the string-depth of the state on edge (l>0)

OUTPUT:

Return ("explicit", end_node) if the state of w is an explicit state and ("implicit", edge, d) is obtained by reading d letters on edge.

EXAMPLES:

```
sage: T = Word('00110111011').suffix_tree()
sage: T.suffix_walk((0, 5), 1)
('explicit', 0)
sage: T.suffix_walk((7, 3), 1)
('implicit', (9, 4), 1)
```

```
>>> from sage.all import *
>>> T = Word('00110111011').suffix_tree()
>>> T.suffix_walk((Integer(0), Integer(5)), Integer(1))
('explicit', 0)
>>> T.suffix_walk((Integer(7), Integer(3)), Integer(1))
('implicit', (9, 4), 1)
```

to_digraph(*word_labels=False*)

Return a DiGraph object of the transition graph of the suffix tree.

INPUT:

- word_labels – boolean (default: False); if False, labels the edges by pairs (i, j). If True, labels the edges by word[i:j]

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.to_digraph()                                                    #␣
↪needs sage.graphs
Digraph on 8 vertices
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> t = ImplicitSuffixTree(W([Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(2)]))
>>> t.to_digraph()                                                    #␣
↪needs sage.graphs
Digraph on 8 vertices
```

to_explicit_suffix_tree()

Convert `self` to an explicit suffix tree.

It is obtained by processing an end of string letter as if it were a regular letter, except that no new leaf nodes are created (thus, the only thing that happens is that some implicit nodes become explicit).

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("aco")("cacao")
sage: t = ImplicitSuffixTree(w)
sage: t.to_explicit_suffix_tree()
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> w = Words("aco")("cacao")
>>> t = ImplicitSuffixTree(w)
>>> t.to_explicit_suffix_tree()
```

```
sage: W = Words([0,1])
sage: s = ImplicitSuffixTree(W([0,1,0,1,1]))
sage: s.to_explicit_suffix_tree()
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1)])
>>> s = ImplicitSuffixTree(W([Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(1)]))
>>> s.to_explicit_suffix_tree()
```

transition_function(*word*, *node=0*)

Return the node obtained by starting from `node` and following the edges labelled by the letters of `word`.

OUTPUT:

`("explicit", end_node)` if we end at `end_node`, or `("implicit", edge, d)` if we end d spots along an edge.

INPUT:

- `word` – a word
- `node` – (default: 0) starting node

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.transition_function(W([0,1,0]))
('implicit', (3, 1), 1)
sage: t.transition_function(W([0,1,2]))
('explicit', 4)
sage: t.transition_function(W([0,1,2]), 5)
('explicit', 2)
sage: t.transition_function(W([0,1]), 5)
('implicit', (5, 2), 2)
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> t = ImplicitSuffixTree(W([Integer(0),Integer(1),Integer(0),Integer(1),
→Integer(2)]))
>>> t.transition_function(W([Integer(0),Integer(1),Integer(0)]))
('implicit', (3, 1), 1)
>>> t.transition_function(W([Integer(0),Integer(1),Integer(2)]))
('explicit', 4)
>>> t.transition_function(W([Integer(0),Integer(1),Integer(2)]), Integer(5))
('explicit', 2)
>>> t.transition_function(W([Integer(0),Integer(1)]), Integer(5))
('implicit', (5, 2), 2)
```

transition_function_dictionary()

Return the transition function as a dictionary of dictionaries.

The format is consistent with the input format for `DiGraph`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words("aco")
sage: t = ImplicitSuffixTree(W("cac"))
sage: t.transition_function_dictionary()
{0: {1: (0, None), 2: (1, None)}}
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree
>>> W = Words("aco")
>>> t = ImplicitSuffixTree(W("cac"))
>>> t.transition_function_dictionary()
{0: {1: (0, None), 2: (1, None)}}
```

```
sage: W = Words([0,1])
sage: t = ImplicitSuffixTree(W([0,1,0]))
```

(continues on next page)

(continued from previous page)

```
sage: t.transition_function_dictionary()
{0: {1: (0, None), 2: (1, None)}}
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1)])
>>> t = ImplicitSuffixTree(W([Integer(0),Integer(1),Integer(0)]))
>>> t.transition_function_dictionary()
{0: {1: (0, None), 2: (1, None)}}
```

trie_type_dict()

Return a dictionary in a format compatible with that of the suffix trie transition function.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree,␣
↪SuffixTrie
sage: W = Words("ab")
sage: t = ImplicitSuffixTree(W("aba"))
sage: d = t.trie_type_dict()
sage: len(d)
5
sage: d                             # random
{(4, word: b): 5, (0, word: a): 4, (0, word: b): 3, (5, word: a): 1, (3,␣
↪word: a): 2}
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree,␣
↪SuffixTrie
>>> W = Words("ab")
>>> t = ImplicitSuffixTree(W("aba"))
>>> d = t.trie_type_dict()
>>> len(d)
5
>>> d                             # random
{(4, word: b): 5, (0, word: a): 4, (0, word: b): 3, (5, word: a): 1, (3,␣
↪word: a): 2}
```

uncompactify()

Return the tree obtained from `self` by splitting edges so that they are labelled by exactly one letter.

The resulting tree is isomorphic to the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree,␣
↪SuffixTrie
sage: abbab = Words("ab")("abbab")
sage: s = SuffixTrie(abbab)
sage: t = ImplicitSuffixTree(abbab)
sage: t.uncompactify().is_isomorphic(s.to_digraph())                       #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import ImplicitSuffixTree,␣
↪SuffixTrie
>>> abbab = Words("ab")("abbab")
>>> s = SuffixTrie(abbab)
>>> t = ImplicitSuffixTree(abbab)
>>> t.uncompactify().is_isomorphic(s.to_digraph())                      #␣
↪needs sage.graphs
True
```

word()

Return the word whose implicit suffix tree this is.

class sage.combinat.words.suffix_trees.**SuffixTrie**(*word*)

Bases: SageObject

Construct the suffix trie of the word w.

The suffix trie of a finite word w is a data structure representing the factors of w. It is a tree whose edges are labelled with letters of w, and whose leafs correspond to suffixes of w.

This is a straightforward implementation of Algorithm 1 from [Ukko1995]. It constructs the suffix trie of w[:i] from that of w[:i-1].

A suffix trie is modelled as a deterministic finite-state automaton together with the suffix_link map. The set of states corresponds to factors of the word (below we write x' for the state corresponding to x); these are always 0, 1, …. The state 0 is the initial state, and it corresponds to the empty word. For the purposes of the algorithm, there is also an auxiliary state -1. The transition function t is defined as:

```
t(-1,a) = 0 for all letters a; and
t(x',a) = y' for all x',y' \in Q such that y = xa,
```

and the suffix link function is defined as:

```
suffix_link(0) = -1;
suffix_link(x') = y', if x = ay for some letter a.
```

REFERENCES:

- [Ukko1995]

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w); t
Suffix Trie of the word: cacao
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("cao")("cacao")
>>> t = SuffixTrie(w); t
Suffix Trie of the word: cacao
```

```
sage: e = Words("ab")()
sage: t = SuffixTrie(e); t
```

(continues on next page)

(continued from previous page)

```
Suffix Trie of the word:
sage: t.process_letter("a"); t
Suffix Trie of the word: a
sage: t.process_letter("b"); t
Suffix Trie of the word: ab
sage: t.process_letter("a"); t
Suffix Trie of the word: aba
```

```
>>> from sage.all import *
>>> e = Words("ab")()
>>> t = SuffixTrie(e); t
Suffix Trie of the word:
>>> t.process_letter("a"); t
Suffix Trie of the word: a
>>> t.process_letter("b"); t
Suffix Trie of the word: ab
>>> t.process_letter("a"); t
Suffix Trie of the word: aba
```

active_state()

Return the active state of the suffix trie.

This is the state corresponding to the word as a suffix of itself.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: t.active_state()
8
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("cao")("cacao")
>>> t = SuffixTrie(w)
>>> t.active_state()
8
```

```
sage: u = Words([0,1])([0,1,1,0,1,0,0,1])
sage: s = SuffixTrie(u)
sage: s.active_state()
22
```

```
>>> from sage.all import *
>>> u = Words([Integer(0),Integer(1)])([Integer(0),Integer(1),Integer(1),
↪Integer(0),Integer(1),Integer(0),Integer(0),Integer(1)])
>>> s = SuffixTrie(u)
>>> s.active_state()
22
```

final_states()

Return the set of final states of the suffix trie.

These are the states corresponding to the suffixes of `self.word()`. They are obtained be repeatedly following the suffix link from the active state until we reach 0.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: t.final_states() == Set([8, 9, 10, 11, 12, 0])
True
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("cao")("cacao")
>>> t = SuffixTrie(w)
>>> t.final_states() == Set([Integer(8), Integer(9), Integer(10), Integer(11),
↪ Integer(12), Integer(0)])
True
```

has_suffix(*word*)

Return `True` if and only if `word` is a suffix of `self.word()`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: [t.has_suffix(w[i:]) for i in range(w.length()+1)]
[True, True, True, True, True, True]
sage: [t.has_suffix(w[:i]) for i in range(w.length()+1)]
[True, False, False, False, False, True]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("cao")("cacao")
>>> t = SuffixTrie(w)
>>> [t.has_suffix(w[i:]) for i in range(w.length()+Integer(1))]
[True, True, True, True, True, True]
>>> [t.has_suffix(w[:i]) for i in range(w.length()+Integer(1))]
[True, False, False, False, False, True]
```

node_to_word(*state=0*)

Return the word obtained by reading the edge labels from 0 to `state`.

INPUT:

- `state` – (default: 0) a state

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("abc")("abcba")
sage: t = SuffixTrie(w)
sage: t.node_to_word(10)
word: abcba
```

(continues on next page)

(continued from previous page)

```
sage: t.node_to_word(7)
word: abcb
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("abc")("abcba")
>>> t = SuffixTrie(w)
>>> t.node_to_word(Integer(10))
word: abcba
>>> t.node_to_word(Integer(7))
word: abcb
```

plot (*layout='tree'*, *tree_root=0*, *tree_orientation='up'*, *vertex_colors=None*, *edge_labels=True*, **args*, ***kwds*)

Return a Graphics object corresponding to the transition graph of the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: SuffixTrie(Word("cacao")).plot()                                          #␣
↪needs sage.plot
Graphics object consisting of 38 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> SuffixTrie(Word("cacao")).plot()                                            #␣
↪needs sage.plot
Graphics object consisting of 38 graphics primitives
```

process_letter (*letter*)

Modify `self` to produce the suffix trie for `self.word() + letter`.

> **Note**
>
> `letter` must occur within the alphabet of the word.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("ab")("ababba")
sage: t = SuffixTrie(w); t
Suffix Trie of the word: ababba
sage: t.process_letter("a"); t
Suffix Trie of the word: ababbaa
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("ab")("ababba")
>>> t = SuffixTrie(w); t
Suffix Trie of the word: ababba
>>> t.process_letter("a"); t
Suffix Trie of the word: ababbaa
```

show(**args*, ***kwds*)

Display the output of `plot()`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cac")
sage: t = SuffixTrie(w)
sage: t.show()                                                              #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("cao")("cac")
>>> t = SuffixTrie(w)
>>> t.show()                                                              #␣
↪needs sage.plot
```

states()

Return the states of the automaton defined by the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words([0,1])([0,1,1])
sage: t = SuffixTrie(w)
sage: t.states()
[0, 1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words([Integer(0),Integer(1)])([Integer(0),Integer(1),Integer(1)])
>>> t = SuffixTrie(w)
>>> t.states()
[0, 1, 2, 3, 4]
```

```
sage: u = Words("aco")("cacao")
sage: s = SuffixTrie(u)
sage: s.states()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
>>> from sage.all import *
>>> u = Words("aco")("cacao")
>>> s = SuffixTrie(u)
>>> s.states()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

suffix_link(*state*)

Evaluate the suffix link map of the suffix trie on `state`.

Note that the suffix link map is not defined on -1.

INPUT:

- `state` – a state

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: list(map(t.suffix_link, range(13)))
[-1, 0, 3, 0, 5, 1, 7, 2, 9, 10, 11, 12, 0]
sage: t.suffix_link(0)
-1
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("cao")("cacao")
>>> t = SuffixTrie(w)
>>> list(map(t.suffix_link, range(Integer(13))))
[-1, 0, 3, 0, 5, 1, 7, 2, 9, 10, 11, 12, 0]
>>> t.suffix_link(Integer(0))
-1
```

to_digraph()

Return a `DiGraph` object of the transition graph of the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cac")
sage: t = SuffixTrie(w)
sage: d = t.to_digraph(); d                                        #␣
↪needs sage.graphs
Digraph on 6 vertices
sage: d.adjacency_matrix()                                         #␣
↪needs sage.graphs sage.modules
[0 1 0 1 0 0]
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("cao")("cac")
>>> t = SuffixTrie(w)
>>> d = t.to_digraph(); d                                        #␣
↪needs sage.graphs
Digraph on 6 vertices
>>> d.adjacency_matrix()                                         #␣
↪needs sage.graphs sage.modules
[0 1 0 1 0 0]
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
```

transition_function(*node*, *word*)

Return the state reached by beginning at `node` and following the arrows in the transition graph labelled by the letters of `word`.

INPUT:

- `node` – a node
- `word` – a word

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words([0,1])([0,1,0,1,1])
sage: t = SuffixTrie(w)
sage: all(t.transition_function(u, letter) == v
....:     for (u, letter), v in t._transition_function.items())
True
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words([Integer(0),Integer(1)])([Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(1)])
>>> t = SuffixTrie(w)
>>> all(t.transition_function(u, letter) == v
...     for (u, letter), v in t._transition_function.items())
True
```

word()

Return the word whose suffix tree this is.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("abc")("abcba")
sage: t = SuffixTrie(w)
sage: t.word()
word: abcba
sage: t.word() == w
True
```

```
>>> from sage.all import *
>>> from sage.combinat.words.suffix_trees import SuffixTrie
>>> w = Words("abc")("abcba")
>>> t = SuffixTrie(w)
>>> t.word()
word: abcba
>>> t.word() == w
True
```

5.1.365 Word classes

AUTHORS:

- Arnaud Bergeron

- Amy Glen
- Sébastien Labbé
- Franco Saliola

class sage.combinat.words.word.**FiniteWord_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *FiniteWord_class*

Finite word represented by a callable.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

EXAMPLES:

```
sage: f = lambda n : 3 if n > 8 else 6
sage: w = Word(f, length=30, caching=False)
sage: w
word: 666666666333333333333333333333
sage: w.is_symmetric()
True
```

```
>>> from sage.all import *
>>> f = lambda n : Integer(3) if n > Integer(8) else Integer(6)
>>> w = Word(f, length=Integer(30), caching=False)
>>> w
word: 666666666333333333333333333333
>>> w.is_symmetric()
True
```

class sage.combinat.words.word.**FiniteWord_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *FiniteWord_class*

Finite word represented by a callable (with caching).

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

EXAMPLES:

```
sage: f = lambda n : n % 3
sage: w = Word(f, length=32)
sage: w
word: 01201201201201201201201201201201
sage: w.border()
word: 01201201201201201201201201201
```

```
>>> from sage.all import *
>>> f = lambda n : n % Integer(3)
>>> w = Word(f, length=Integer(32))
>>> w
word: 01201201201201201201201201201201
>>> w.border()
word: 01201201201201201201201201201
```

class sage.combinat.words.word.**FiniteWord_char**

Bases: *WordDatatype_char*, *FiniteWord_class*

Finite word represented by an array `unsigned char *` (i.e. integers between 0 and 255).

For any word `w`, type `w.<TAB>` to see the functions that can be applied to `w`.

EXAMPLES:

```
sage: W = Words(range(20))

sage: w = W(list(range(1, 10)) * 2)
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_char'>
sage: w
word: 123456789123456789

sage: w.is_palindrome()
False
sage: (w*w[::-1]).is_palindrome()
True
sage: (w[:-1:]*w[::-1]).is_palindrome()
True

sage: w.is_lyndon()
False
sage: W(list(range(10)) + [10, 10]).is_lyndon()
True

sage: w.is_square_free()
False
sage: w[:-1].is_square_free()
True

sage: u = W([randint(0,10) for i in range(10)])
sage: (u*u).is_square()
True
sage: (u*u*u).is_cube()
True

sage: len(w.factor_set())
127
sage: w.rauzy_graph(5)                                                  #␣
↪needs sage.graphs
Looped digraph on 9 vertices

sage: u = W([1,2,3])
sage: w.first_occurrence(u)
0
sage: w.first_occurrence(u, start=1)
9
```

```
>>> from sage.all import *
>>> W = Words(range(Integer(20)))

>>> w = W(list(range(Integer(1), Integer(10))) * Integer(2))
>>> type(w)
<class 'sage.combinat.words.word.FiniteWord_char'>
>>> w
```

(continues on next page)

(continued from previous page)

```
word: 123456789123456789

>>> w.is_palindrome()
False
>>> (w*w[::-Integer(1)]).is_palindrome()
True
>>> (w[:-Integer(1):]*w[::-Integer(1)]).is_palindrome()
True

>>> w.is_lyndon()
False
>>> W(list(range(Integer(10))) + [Integer(10), Integer(10)]).is_lyndon()
True

>>> w.is_square_free()
False
>>> w[:-Integer(1)].is_square_free()
True

>>> u = W([randint(Integer(0),Integer(10)) for i in range(Integer(10))])
>>> (u*u).is_square()
True
>>> (u*u*u).is_cube()
True

>>> len(w.factor_set())
127
>>> w.rauzy_graph(Integer(5))
↪     # needs sage.graphs
Looped digraph on 9 vertices

>>> u = W([Integer(1),Integer(2),Integer(3)])
>>> w.first_occurrence(u)
0
>>> w.first_occurrence(u, start=Integer(1))
9
```

class sage.combinat.words.word.**FiniteWord_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *FiniteWord_class*

Finite word represented by an iterator.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word(iter(range(10)), caching=False)
sage: w
word: 0123456789
sage: w.finite_differences()
word: 111111111
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> w = Word(iter(range(Integer(10))), caching=False)
>>> w
word: 0123456789
>>> w.finite_differences()
word: 111111111
```

class sage.combinat.words.word.**FiniteWord_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *FiniteWord_class*

Finite word represented by an iterator (with caching).

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word(iter('abcdef'))
sage: w.conjugate(2)
word: cdefab
```

```
>>> from sage.all import *
>>> w = Word(iter('abcdef'))
>>> w.conjugate(Integer(2))
word: cdefab
```

class sage.combinat.words.word.**FiniteWord_list**

Bases: *WordDatatype_list*, *FiniteWord_class*

Finite word represented by a Python list.

For any word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word(range(10))
sage: w.iterated_right_palindromic_closure()
word: 0102010301020104010201030102010501020103...
```

```
>>> from sage.all import *
>>> w = Word(range(Integer(10)))
>>> w.iterated_right_palindromic_closure()
word: 0102010301020104010201030102010501020103...
```

class sage.combinat.words.word.**FiniteWord_morphic**(*parent*, *morphism*, *letter*, *coding=None*, *length=+Infinity*)

Bases: `WordDatatype_morphic`, *FiniteWord_class*

Finite morphic word.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

EXAMPLES:

```
sage: m = WordMorphism("a->ab,b->")
sage: w = m.fixed_point("a")
sage: w
word: ab
```

```
>>> from sage.all import *
>>> m = WordMorphism("a->ab,b->")
>>> w = m.fixed_point("a")
>>> w
word: ab
```

class sage.combinat.words.word.**FiniteWord_str**

Bases: *WordDatatype_str*, *FiniteWord_class*

Finite word represented by a Python str.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word('abcdef')
sage: w.is_square()
False
```

```
>>> from sage.all import *
>>> w = Word('abcdef')
>>> w.is_square()
False
```

class sage.combinat.words.word.**FiniteWord_tuple**

Bases: *WordDatatype_tuple*, *FiniteWord_class*

Finite word represented by a Python tuple.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word(())
sage: w.is_empty()
True
```

```
>>> from sage.all import *
>>> w = Word(())
>>> w.is_empty()
True
```

class sage.combinat.words.word.**InfiniteWord_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*, *InfiniteWord_class*

Infinite word represented by a callable.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: w = Word(lambda n:n, caching=False)
sage: w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,
```

(continues on next page)

(continued from previous page)

```
↪28,29,30,31,32,33,34,35,36,37,38,39,...
sage: w.iterated_right_palindromic_closure()
word: 0102010301020104010201030102010501020103...
```

```
>>> from sage.all import *
>>> w = Word(lambda n:n, caching=False)
>>> w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,
↪28,29,30,31,32,33,34,35,36,37,38,39,...
>>> w.iterated_right_palindromic_closure()
word: 0102010301020104010201030102010501020103...
```

class sage.combinat.words.word.**InfiniteWord_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable_with_caching*, *InfiniteWord_class*

Infinite word represented by a callable (with caching).

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: w = Word(lambda n:n)
sage: factor = w[4:13]
sage: factor
word: 4,5,6,7,8,9,10,11,12
```

```
>>> from sage.all import *
>>> w = Word(lambda n:n)
>>> factor = w[Integer(4):Integer(13)]
>>> factor
word: 4,5,6,7,8,9,10,11,12
```

class sage.combinat.words.word.**InfiniteWord_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *InfiniteWord_class*

Infinite word represented by an iterable.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: from itertools import chain, cycle
sage: w = Word(chain('letsgo', cycle('forever')), caching=False)
sage: w
word: letsgoforeverforeverforeverforeverforeve...
sage: prefix = w[:100]
sage: prefix
word: letsgoforeverforeverforeverforeverforeve...
```

(continues on next page)

(continued from previous page)

```
sage: prefix.is_lyndon()
False
```

```
>>> from sage.all import *
>>> from itertools import chain, cycle
>>> w = Word(chain('letsgo', cycle('forever')), caching=False)
>>> w
word: letsgoforeverforeverforeverforeverforeve...
>>> prefix = w[:Integer(100)]
>>> prefix
word: letsgoforeverforeverforeverforeverforeve...
>>> prefix.is_lyndon()
False
```

class sage.combinat.words.word.**InfiniteWord_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *InfiniteWord_class*

Infinite word represented by an iterable (with caching).

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: from itertools import cycle
sage: w = Word(cycle([9,8,4]))
sage: w
word: 9849849849849849849849849849849849849849...
sage: prefix = w[:23]
sage: prefix
word: 98498498498498498498498
sage: prefix.minimal_period()
3
```

```
>>> from sage.all import *
>>> from itertools import cycle
>>> w = Word(cycle([Integer(9),Integer(8),Integer(4)]))
>>> w
word: 9849849849849849849849849849849849849849...
>>> prefix = w[:Integer(23)]
>>> prefix
word: 98498498498498498498498
>>> prefix.minimal_period()
3
```

class sage.combinat.words.word.**InfiniteWord_morphic**(*parent*, *morphism*, *letter*, *coding=None*, *length=+Infinity*)

Bases: `WordDatatype_morphic`, *InfiniteWord_class*

Morphic word of infinite length.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: m = WordMorphism('a->ab,b->a')
sage: w = m.fixed_point('a')
sage: w
word: abaababaabaababaababaabaababaabaababaaba...
```

```
>>> from sage.all import *
>>> m = WordMorphism('a->ab,b->a')
>>> w = m.fixed_point('a')
>>> w
word: abaababaabaababaababaabaababaabaababaaba...
```

`sage.combinat.words.word.Word`(*data=None*, *alphabet=None*, *length=None*, *datatype=None*, *caching=True*, *RSK_data=None*)

Construct a word.

INPUT:

- `data` – (default: `None`) list, string, tuple, iterator, free monoid element, `None` (shorthand for `[]`), or a callable defined on `[0,1,...,length]`
- `alphabet` – any argument accepted by Words
- `length` – (default: `None`) this is dependent on the type of data. It is ignored for words defined by lists, strings, tuples, etc., because they have a naturally defined length. For callables, this defines the domain of definition, which is assumed to be `[0, 1, 2, ..., length-1]`. For iterators: Infinity if you know the iterator will not terminate (default); `'unknown'` if you do not know whether the iterator terminates; `'finite'` if you know that the iterator terminates, but do not know the length.
- `datatype` – (default: `None`) `None`, `'list'`, `'str'`, `'tuple'`, `'iter'`, `'callable'`; if `None`, then the function tries to guess this from the data
- `caching` – boolean (default: `True`); whether to keep a cache of the letters computed by an iterator or callable
- `RSK_data` – (default: `None`) semistandard and a standard Young tableau to run the inverse RSK bijection on

Note

Be careful when defining words using callables and iterators. It appears that islice does not pickle correctly causing various errors when reloading. Also, most iterators do not support copying and should not support pickling by extension.

EXAMPLES:

Empty word:

```
sage: Word()
word:
```

```
>>> from sage.all import *
>>> Word()
word:
```

Word with string:

```
sage: Word("abbabaab")
word: abbabaab
```

```
>>> from sage.all import *
>>> Word("abbabaab")
word: abbabaab
```

Word with string constructed from other types:

```
sage: Word([0,1,1,0,1,0,0,1], datatype='str')
word: 01101001
sage: Word((0,1,1,0,1,0,0,1), datatype='str')
word: 01101001
```

```
>>> from sage.all import *
>>> Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1)], datatype='str')
word: 01101001
>>> Word((Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1)), datatype='str')
word: 01101001
```

Word with list:

```
sage: Word([0,1,1,0,1,0,0,1])
word: 01101001
```

```
>>> from sage.all import *
>>> Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1)])
word: 01101001
```

Word with list constructed from other types:

```
sage: Word("01101001", datatype='list')
word: 01101001
sage: Word((0,1,1,0,1,0,0,1), datatype='list')
word: 01101001
```

```
>>> from sage.all import *
>>> Word("01101001", datatype='list')
word: 01101001
>>> Word((Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1)), datatype='list')
word: 01101001
```

Word with tuple:

```
sage: Word((0,1,1,0,1,0,0,1))
word: 01101001
```

```
>>> from sage.all import *
>>> Word((Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1)))
word: 01101001
```

Word with tuple constructed from other types:

```
sage: Word([0,1,1,0,1,0,0,1], datatype='tuple')
word: 01101001
sage: Word("01101001", datatype='str')
word: 01101001
```

```
>>> from sage.all import *
>>> Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(1)], datatype='tuple')
word: 01101001
>>> Word("01101001", datatype='str')
word: 01101001
```

Word with iterator:

```
sage: from itertools import count
sage: Word(count())
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,
↪28,29,30,31,32,33,34,35,36,37,38,39,...
sage: Word(iter("abbabaab")) # iterators default to infinite words
word: abbabaab
sage: Word(iter("abbabaab"), length='unknown')
word: abbabaab
sage: Word(iter("abbabaab"), length='finite')
word: abbabaab
```

```
>>> from sage.all import *
>>> from itertools import count
>>> Word(count())
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,
↪28,29,30,31,32,33,34,35,36,37,38,39,...
>>> Word(iter("abbabaab")) # iterators default to infinite words
word: abbabaab
>>> Word(iter("abbabaab"), length='unknown')
word: abbabaab
>>> Word(iter("abbabaab"), length='finite')
word: abbabaab
```

Word with function (a 'callable'):

```
sage: f = lambda n : add(Integer(n).digits(2)) % 2
sage: Word(f)
word: 0110100110010110100101100110100110010110...
sage: Word(f, length=8)
word: 01101001
```

```
>>> from sage.all import *
>>> f = lambda n : add(Integer(n).digits(Integer(2))) % Integer(2)
>>> Word(f)
word: 0110100110010110100101100110100110010110...
>>> Word(f, length=Integer(8))
word: 01101001
```

Word over a string with a parent:

```
sage: w = Word("abbabaab", alphabet='abc'); w
word: abbabaab
sage: w.parent()
Finite words over {'a', 'b', 'c'}
```

```
>>> from sage.all import *
>>> w = Word("abbabaab", alphabet='abc'); w
word: abbabaab
>>> w.parent()
Finite words over {'a', 'b', 'c'}
```

Word from a free monoid element:

```
sage: M.<x,y,z> = FreeMonoid(3)
sage: Word(x^3*y*x*z^2*x)
word: xxxyxzzx
```

```
>>> from sage.all import *
>>> M = FreeMonoid(Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = M._first_
↪ngens(3)
>>> Word(x**Integer(3)*y*x*z**Integer(2)*x)
word: xxxyxzzx
```

The default parent is the combinatorial class of all words:

```
sage: w = Word("abbabaab"); w
word: abbabaab
sage: w.parent()
Finite words over Set of Python objects of class 'object'
```

```
>>> from sage.all import *
>>> w = Word("abbabaab"); w
word: abbabaab
>>> w.parent()
Finite words over Set of Python objects of class 'object'
```

We can also input a semistandard tableau and a standard tableau to obtain a word from the inverse RSK algorithm using the `RSK_data` option:

```
sage: p = Tableau([[1,2,2],[3]]); q = Tableau([[1,2,4],[3]])
sage: Word(RSK_data=[p, q])
word: 1322
```

```
>>> from sage.all import *
>>> p = Tableau([[Integer(1),Integer(2),Integer(2)],[Integer(3)]]); q =␣
↪Tableau([[Integer(1),Integer(2),Integer(4)],[Integer(3)]])
>>> Word(RSK_data=[p, q])
word: 1322
```

class sage.combinat.words.word.**Word_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*, *Word_class*

Word of unknown length (finite or infinite) represented by an iterable.

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

Words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: w = Word(iter([1,1,4,9]*1000), length='unknown', caching=False)
sage: w
word: 114911491149114911491149114911491149...
sage: w.delta()
word: 211211211211211211211211211211211211...
```

```
>>> from sage.all import *
>>> w = Word(iter([Integer(1),Integer(1),Integer(4),Integer(9)]*Integer(1000)),␣
↪length='unknown', caching=False)
>>> w
word: 114911491149114911491149114911491149...
>>> w.delta()
word: 211211211211211211211211211211211211...
```

class sage.combinat.words.word.**Word_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter_with_caching*, *Word_class*

Word of unknown length (finite or infinite) represented by an iterable (with caching).

For such word w, type `w.` and hit `Tab` key to see the list of functions defined on w.

Words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: w = Word(iter([1,2,3]*1000), length='unknown')
sage: w
word: 123123123123123123123123123123123123...
sage: w.finite_differences(mod=2)
word: 110110110110110110110110110110110110...
```

```
>>> from sage.all import *
>>> w = Word(iter([Integer(1),Integer(2),Integer(3)]*Integer(1000)), length=
↪'unknown')
>>> w
word: 123123123123123123123123123123123123...
>>> w.finite_differences(mod=Integer(2))
word: 110110110110110110110110110110110110...
```

5.1.366 Fast word datatype using an array of unsigned char

class sage.combinat.words.word_char.**WordDatatype_char**

Bases: *WordDatatype*

A Fast class for words represented by an array `unsigned char *`.

Currently, only handles letters in [0,255].

concatenate(*other*)

Concatenation of `self` and `other`.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: W([0,2,1]).concatenate([0,0,0])
word: 021000
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> W([Integer(0),Integer(2),Integer(1)]).concatenate([Integer(0),Integer(0),
↪Integer(0)])
word: 021000
```

has_prefix(*other*)

Test whether `other` is a prefix of `self`.

INPUT:

- `other` – a word or a sequence (e.g. tuple, list)

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: w = W([0,1,1,0,1,2,0])
sage: w.has_prefix([0,1,1])
True
sage: w.has_prefix([0,1,2])
False
sage: w.has_prefix(w)
True
sage: w.has_prefix(w[:-1])
True
sage: w.has_prefix(w[1:])
False
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> w = W([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1),Integer(2),
↪Integer(0)])
>>> w.has_prefix([Integer(0),Integer(1),Integer(1)])
True
>>> w.has_prefix([Integer(0),Integer(1),Integer(2)])
False
>>> w.has_prefix(w)
True
```

(continues on next page)

(continued from previous page)

```
>>> w.has_prefix(w[:-Integer(1)])
True
>>> w.has_prefix(w[Integer(1):])
False
```

is_empty()

Return whether the word is empty.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: W([0,1,2,2]).is_empty()
False
sage: W([]).is_empty()
True
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> W([Integer(0),Integer(1),Integer(2),Integer(2)]).is_empty()
False
>>> W([]).is_empty()
True
```

is_square()

Return `True` if `self` is a square, and `False` otherwise.

EXAMPLES:

```
sage: w = Word([n % 4 for n in range(48)], alphabet=[0,1,2,3])
sage: w.is_square()
True
```

```
>>> from sage.all import *
>>> w = Word([n % Integer(4) for n in range(Integer(48))],␣
↪alphabet=[Integer(0),Integer(1),Integer(2),Integer(3)])
>>> w.is_square()
True
```

```
sage: w = Word([n % 4 for n in range(49)], alphabet=[0,1,2,3])
sage: w.is_square()
False
sage: (w*w).is_square()
True
```

```
>>> from sage.all import *
>>> w = Word([n % Integer(4) for n in range(Integer(49))],␣
↪alphabet=[Integer(0),Integer(1),Integer(2),Integer(3)])
>>> w.is_square()
False
>>> (w*w).is_square()
True
```

length()

Return the length of the word as a Sage integer.

EXAMPLES:

```
sage: W = Words([0,1,2,3,4])
sage: w = W([0,1,2,0,3,2,1])
sage: w.length()
7
sage: type(w.length())
<class 'sage.rings.integer.Integer'>
sage: type(len(w))
<class 'int'>
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)])
>>> w = W([Integer(0),Integer(1),Integer(2),Integer(0),Integer(3),Integer(2),
↪Integer(1)])
>>> w.length()
7
>>> type(w.length())
<class 'sage.rings.integer.Integer'>
>>> type(len(w))
<class 'int'>
```

letters()

Return the list of letters that appear in this word, listed in the order of first appearance.

EXAMPLES:

```
sage: W = Words(5)
sage: W([1,3,1,2,2,3,1]).letters()
[1, 3, 2]
```

```
>>> from sage.all import *
>>> W = Words(Integer(5))
>>> W([Integer(1),Integer(3),Integer(1),Integer(2),Integer(2),Integer(3),
↪Integer(1)]).letters()
[1, 3, 2]
```

longest_common_prefix(*other*)

Return the longest common prefix of this word and `other`.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: W([0,1,0,2]).longest_common_prefix([0,1])
word: 01
sage: u = W([0,1,0,0,1])
sage: v = W([0,1,0,2])
sage: u.longest_common_prefix(v)
word: 010
sage: v.longest_common_prefix(u)
word: 010
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> W([Integer(0),Integer(1),Integer(0),Integer(2)]).longest_common_
↪prefix([Integer(0),Integer(1)])
word: 01
>>> u = W([Integer(0),Integer(1),Integer(0),Integer(0),Integer(1)])
>>> v = W([Integer(0),Integer(1),Integer(0),Integer(2)])
>>> u.longest_common_prefix(v)
word: 010
>>> v.longest_common_prefix(u)
word: 010
```

Using infinite words is also possible (and the return type is also a of the same type as `self`):

```
sage: W([0,1,0,0]).longest_common_prefix(words.FibonacciWord())
word: 0100
sage: type(_)
<class 'sage.combinat.words.word.FiniteWord_char'>
```

```
>>> from sage.all import *
>>> W([Integer(0),Integer(1),Integer(0),Integer(0)]).longest_common_
↪prefix(words.FibonacciWord())
word: 0100
>>> type(_)
<class 'sage.combinat.words.word.FiniteWord_char'>
```

An example of an intensive usage:

```
sage: W = Words([0,1])
sage: w = words.FibonacciWord()
sage: w = W(list(w[:5000]))
sage: L = [[len(w[n:].longest_common_prefix(w[n+fibonacci(i):]))           #␣
↪needs sage.libs.pari
....:      for i in range(5,15)] for n in range(1,1000)]
sage: for n,l in enumerate(L):                                              #␣
↪needs sage.libs.pari
....:     if l.count(0) > 4:
....:         print("{} {}".format(n+1,l))
375 [0, 13, 0, 34, 0, 89, 0, 233, 0, 233]
376 [0, 12, 0, 33, 0, 88, 0, 232, 0, 232]
608 [8, 0, 21, 0, 55, 0, 144, 0, 377, 0]
609 [7, 0, 20, 0, 54, 0, 143, 0, 376, 0]
985 [0, 13, 0, 34, 0, 89, 0, 233, 0, 610]
986 [0, 12, 0, 33, 0, 88, 0, 232, 0, 609]
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1)])
>>> w = words.FibonacciWord()
>>> w = W(list(w[:Integer(5000)]))
>>> L = [[len(w[n:].longest_common_prefix(w[n+fibonacci(i):]))           #␣
↪needs sage.libs.pari
...      for i in range(Integer(5),Integer(15))] for n in range(Integer(1),
↪Integer(1000))]
```

(continues on next page)

(continued from previous page)

```
>>> for n,l in enumerate(L):                                                  #␣
↪needs sage.libs.pari
...     if l.count(Integer(0)) > Integer(4):
...         print("{} {}".format(n+Integer(1),l))
375 [0, 13, 0, 34, 0, 89, 0, 233, 0, 233]
376 [0, 12, 0, 33, 0, 88, 0, 232, 0, 232]
608 [8, 0, 21, 0, 55, 0, 144, 0, 377, 0]
609 [7, 0, 20, 0, 54, 0, 143, 0, 376, 0]
985 [0, 13, 0, 34, 0, 89, 0, 233, 0, 610]
986 [0, 12, 0, 33, 0, 88, 0, 232, 0, 609]
```

longest_common_suffix(*other*)

Return the longest common suffix between this word and `other`.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: W([0,1,0,2]).longest_common_suffix([2,0,2])
word: 02
sage: u = W([0,1,0,0,1])
sage: v = W([1,2,0,0,1])
sage: u.longest_common_suffix(v)
word: 001
sage: v.longest_common_suffix(u)
word: 001
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> W([Integer(0),Integer(1),Integer(0),Integer(2)]).longest_common_
↪suffix([Integer(2),Integer(0),Integer(2)])
word: 02
>>> u = W([Integer(0),Integer(1),Integer(0),Integer(0),Integer(1)])
>>> v = W([Integer(1),Integer(2),Integer(0),Integer(0),Integer(1)])
>>> u.longest_common_suffix(v)
word: 001
>>> v.longest_common_suffix(u)
word: 001
```

sage.combinat.words.word_char.**reversed_word_iterator**(*w*)

This function exists only because it is not possible to use yield in the special method `__reversed__`.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: w = W([0,1,0,0,1,2])
sage: list(reversed(w)) # indirect doctest
[2, 1, 0, 0, 1, 0]
```

```
>>> from sage.all import *
>>> W = Words([Integer(0),Integer(1),Integer(2)])
>>> w = W([Integer(0),Integer(1),Integer(0),Integer(0),Integer(1),Integer(2)])
>>> list(reversed(w)) # indirect doctest
[2, 1, 0, 0, 1, 0]
```

5.1.367 Datatypes for finite words

class sage.combinat.words.word_datatypes.**WordDatatype**

Bases: object

The generic WordDatatype class.

Any word datatype must contain two attributes (at least):

- _parent
- _hash

They are automatically defined here and it's not necessary (and forbidden) to define them anywhere else.

class sage.combinat.words.word_datatypes.**WordDatatype_list**

Bases: *WordDatatype*

Datatype class for words defined by lists.

length()

Return the length of the word.

EXAMPLES:

```
sage: w = Word([0,1,1,0])
sage: w.length()
4
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(1),Integer(0)])
>>> w.length()
4
```

number_of_letter_occurrences(*a*)

Return the number of occurrences of the letter a in the word self.

INPUT:

- a – a letter

OUTPUT: integer

EXAMPLES:

```
sage: w = Word([0,1,1,0,1])
sage: w.number_of_letter_occurrences(0)
2
sage: w.number_of_letter_occurrences(1)
3
sage: w.number_of_letter_occurrences(2)
0
```

```
>>> from sage.all import *
>>> w = Word([Integer(0),Integer(1),Integer(1),Integer(0),Integer(1)])
>>> w.number_of_letter_occurrences(Integer(0))
2
>>> w.number_of_letter_occurrences(Integer(1))
3
```

(continues on next page)

(continued from previous page)

```
>>> w.number_of_letter_occurrences(Integer(2))
0
```

See also

sage.combinat.words.finite_word.FiniteWord_class.number_of_factor_occurrences()

class sage.combinat.words.word_datatypes.**WordDatatype_str**

Bases: *WordDatatype*

Datatype for words defined by strings.

find(*sub*, *start=0*, *end=None*)

Return the index of the first occurrence of sub in self, such that sub is contained within self[start:end]. Returns -1 on failure.

INPUT:

- `sub` – string or word to search for
- `start` – nonnegative integer (default: 0) specifying the position from which to start the search.
- `end` – nonnegative integer (default: `None`); specifying the position at which the search must stop. If `None`, then the search is performed up to the end of the string.

OUTPUT: nonnegative integer or -1

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: w.find("a")
0
sage: w.find("a", 4)
5
sage: w.find("a", 4, 5)
-1
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> w.find("a")
0
>>> w.find("a", Integer(4))
5
>>> w.find("a", Integer(4), Integer(5))
-1
```

has_prefix(*other*)

Test whether `self` has `other` as a prefix.

INPUT:

- `other` – a word (an instance of `Word_class`) or a `str`

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: u = Word("abbab")
sage: w.has_prefix(u)
True
sage: u.has_prefix(w)
False
sage: u.has_prefix("abbab")
True
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> u = Word("abbab")
>>> w.has_prefix(u)
True
>>> u.has_prefix(w)
False
>>> u.has_prefix("abbab")
True
```

has_suffix(*other*)

Test whether `self` has `other` as a suffix.

INPUT:

- `other` – a word (an instance of `Word_class`) or a `str`

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: u = Word("ababa")
sage: w.has_suffix(u)
True
sage: u.has_suffix(w)
False
sage: u.has_suffix("ababa")
True
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> u = Word("ababa")
>>> w.has_suffix(u)
True
>>> u.has_suffix(w)
False
>>> u.has_suffix("ababa")
True
```

is_prefix(*other*)

Test whether `self` is a prefix of `other`.

INPUT:

- `other` – a word (an instance of `Word_class`) or a `str`

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: u = Word("abbab")
sage: w.is_prefix(u)
False
sage: u.is_prefix(w)
True
sage: u.is_prefix("abbabaabababa")
True
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> u = Word("abbab")
>>> w.is_prefix(u)
False
>>> u.is_prefix(w)
True
>>> u.is_prefix("abbabaabababa")
True
```

is_suffix(*other*)

Test whether `self` is a suffix of `other`.

INPUT:

- `other` – a word (an instance of `Word_class`) or a `str`

OUTPUT: boolean

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: u = Word("ababa")
sage: w.is_suffix(u)
False
sage: u.is_suffix(w)
True
sage: u.is_suffix("abbabaabababa")
True
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> u = Word("ababa")
>>> w.is_suffix(u)
False
>>> u.is_suffix(w)
True
>>> u.is_suffix("abbabaabababa")
True
```

length()

Return the length of the word.

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: w.length()
13
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> w.length()
13
```

number_of_letter_occurrences(*letter*)

Count the number of occurrences of `letter`.

INPUT:

- `letter` – a letter

OUTPUT: integer

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: w.number_of_letter_occurrences('a')
7
sage: w.number_of_letter_occurrences('b')
6
sage: w.number_of_letter_occurrences('c')
0
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> w.number_of_letter_occurrences('a')
7
>>> w.number_of_letter_occurrences('b')
6
>>> w.number_of_letter_occurrences('c')
0
```

```
sage: w.number_of_letter_occurrences('abb')
0
```

```
>>> from sage.all import *
>>> w.number_of_letter_occurrences('abb')
0
```

See also

sage.combinat.words.finite_word.FiniteWord_class.number_of_factor_occurrences()

partition(*sep*)

Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. The concatenation of the terms in the list gives back the initial word.

See also the split method.

Note

This just wraps Python's builtin `str::partition()` for `str`.

INPUT:

- `sep` – string or word

EXAMPLES:

```
sage: w = Word("MyTailorIsPoor")
sage: w.partition("Tailor")
[word: My, word: Tailor, word: IsPoor]
```

```
>>> from sage.all import *
>>> w = Word("MyTailorIsPoor")
>>> w.partition("Tailor")
[word: My, word: Tailor, word: IsPoor]
```

```
sage: w = Word("3230301030323212323032321210121232121010")
sage: l = w.partition("323")
sage: print(l)
[word: , word: 323, word: 0301030323212323032321210121232121010]
sage: sum(l, Word('')) == w
True
```

```
>>> from sage.all import *
>>> w = Word("3230301030323212323032321210121232121010")
>>> l = w.partition("323")
>>> print(l)
[word: , word: 323, word: 0301030323212323032321210121232121010]
>>> sum(l, Word('')) == w
True
```

If the separator is not a string an error is raised:

```
sage: w = Word("le papa du papa du papa etait un petit pioupiou")
sage: w.partition(Word(['p','a','p','a']))
Traceback (most recent call last):
...
ValueError: the separator must be a string
```

```
>>> from sage.all import *
>>> w = Word("le papa du papa du papa etait un petit pioupiou")
>>> w.partition(Word(['p','a','p','a']))
Traceback (most recent call last):
...
ValueError: the separator must be a string
```

rfind(*sub*, *start=0*, *end=None*)

Return the index of the last occurrence of sub in self, such that sub is contained within self[start:end]. Returns -1 on failure.

INPUT:

- `sub` – string or word to search for
- `start` – nonnegative integer (default: 0) specifying the position at which the search must stop.
- `end` – nonnegative integer (default: `None`); specifying the position from which to start the search. If `None`, then the search is performed up to the end of the string.

OUTPUT: nonnegative integer or -1

EXAMPLES:

```
sage: w = Word("abbabaabababa")
sage: w.rfind("a")
12
sage: w.rfind("a", 4, 8)
6
sage: w.rfind("a", 4, 5)
-1
```

```
>>> from sage.all import *
>>> w = Word("abbabaabababa")
>>> w.rfind("a")
12
>>> w.rfind("a", Integer(4), Integer(8))
6
>>> w.rfind("a", Integer(4), Integer(5))
-1
```

split(*sep=None*, *maxsplit=None*)

Return a list of words, using sep as a delimiter string. If maxsplit is given, at most maxsplit splits are done.

See also the partition method.

> **Note**
>
> This just wraps Python's builtin `str::split()` for `str`.

INPUT:

- `sep` – string or word (default: `None`)
- `maxsplit` – positive integer (default: `None`)

OUTPUT: list of words

EXAMPLES:

You can split along white space to find words in a sentence:

```
sage: w = Word("My tailor is poor")
sage: w.split(" ")
[word: My, word: tailor, word: is, word: poor]
```

```
>>> from sage.all import *
>>> w = Word("My tailor is poor")
>>> w.split(" ")
[word: My, word: tailor, word: is, word: poor]
```

The python behavior is kept when no argument is given:

```
sage: w.split()
[word: My, word: tailor, word: is, word: poor]
```

```
>>> from sage.all import *
>>> w.split()
[word: My, word: tailor, word: is, word: poor]
```

You can split in two words letters to get the length of blocks in the other letter:

```
sage: w = Word("ababbabaaba")
sage: w.split('a')
[word: , word: b, word: bb, word: b, word: , word: b, word: ]
sage: w.split('b')
[word: a, word: a, word: , word: a, word: aa, word: a]
```

```
>>> from sage.all import *
>>> w = Word("ababbabaaba")
>>> w.split('a')
[word: , word: b, word: bb, word: b, word: , word: b, word: ]
>>> w.split('b')
[word: a, word: a, word: , word: a, word: aa, word: a]
```

You can split along words:

```
sage: w = Word("3230301030323212323032321")
sage: w.split("32")
[word: , word: 30301030, word: , word: 12, word: 30, word: , word: 1]
```

```
>>> from sage.all import *
>>> w = Word("3230301030323212323032321")
>>> w.split("32")
[word: , word: 30301030, word: , word: 12, word: 30, word: , word: 1]
```

If the separator is not a string a `ValueError` is raised:

```
sage: w = Word("le papa du papa du papa etait un petit pioupiou")
sage: w.split(Word(['p','a','p','a']))
Traceback (most recent call last):
...
ValueError: the separator must be a string
```

```
>>> from sage.all import *
>>> w = Word("le papa du papa du papa etait un petit pioupiou")
>>> w.split(Word(['p','a','p','a']))
Traceback (most recent call last):
...
ValueError: the separator must be a string
```

class sage.combinat.words.word_datatypes.**WordDatatype_tuple**

Bases: *WordDatatype*

Datatype class for words defined by tuples.

length()

Return the length of the word.

EXAMPLES:

```
sage: w = Word((0,1,1,0))
sage: w.length()
4
```

```
>>> from sage.all import *
>>> w = Word((Integer(0),Integer(1),Integer(1),Integer(0)))
>>> w.length()
4
```

5.1.368 Common words

AUTHORS:

- Franco Saliola (2008-12-17): merged into sage
- Sébastien Labbé (2008-12-17): merged into sage
- Arnaud Bergeron (2008-12-17): merged into sage
- Amy Glen (2008-12-17): merged into sage
- Sébastien Labbé (2009-12-19): Added S-adic words (Issue #7543)

USE:

To see a list of all word constructors, type `words.` and then press the `Tab` key. The documentation for each constructor includes information about each word, which provides a useful reference.

REFERENCES:

EXAMPLES:

```
sage: t = words.ThueMorseWord(); t
word: 0110100110010110100101100110100110010110...
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord(); t
word: 0110100110010110100101100110100110010110...
```

class sage.combinat.words.word_generators.**LowerChristoffelWord**(*p*, *q*, *alphabet=(0, 1)*, *algorithm='cf'*)

Bases: *FiniteWord_list*

Return the lower Christoffel word of slope p/q, where p and q are relatively prime nonnegative integers, over the given two-letter alphabet.

The *Christoffel word of slope `p/q`* is obtained from the Cayley graph of $\mathbf{Z}/(p+q)\mathbf{Z}$ with generator q as follows. If $u \to v$ is an edge in the Cayley graph, then $v = u + p \mod p + q$. Label the edge $u \to v$ by `alphabet[1]` if $u < v$ and `alphabet[0]` otherwise. The Christoffel word is the word obtained by reading the edge labels along the cycle beginning from 0.

EXAMPLES:

```
sage: words.LowerChristoffelWord(4,7)
word: 00100100101
```

```
>>> from sage.all import *
>>> words.LowerChristoffelWord(Integer(4),Integer(7))
word: 00100100101
```

```
sage: words.LowerChristoffelWord(4,7,alphabet='ab')
word: aabaabaabab
```

```
>>> from sage.all import *
>>> words.LowerChristoffelWord(Integer(4),Integer(7),alphabet='ab')
word: aabaabaabab
```

markoff_number()

Return the Markoff number associated to the Christoffel word `self`.

The *Markoff number* of a Christoffel word w is $trace(M(w))/3$, where $M(w)$ is the 2×2 matrix obtained by applying the morphism: 0 -> matrix(2,[2,1,1,1]) 1 -> matrix(2,[5,2,2,1])

EXAMPLES:

```
sage: w0 = words.LowerChristoffelWord(4,7)
sage: w1, w2 = w0.standard_factorization()
sage: (m0,m1,m2) = (w.markoff_number() for w in (w0,w1,w2))                 #␣
↪needs sage.modules
sage: (m0,m1,m2)                                                             #␣
↪needs sage.modules
(294685, 13, 7561)
sage: m0**2 + m1**2 + m2**2 == 3*m0*m1*m2                                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> w0 = words.LowerChristoffelWord(Integer(4),Integer(7))
>>> w1, w2 = w0.standard_factorization()
>>> (m0,m1,m2) = (w.markoff_number() for w in (w0,w1,w2))                 #␣
↪needs sage.modules
>>> (m0,m1,m2)                                                             #␣
↪needs sage.modules
(294685, 13, 7561)
>>> m0**Integer(2) + m1**Integer(2) + m2**Integer(2) == Integer(3)*m0*m1*m2  ␣
↪                                  # needs sage.modules
True
```

standard_factorization()

Return the standard factorization of the Christoffel word `self`.

The *standard factorization* of a Christoffel word w is the unique factorization of w into two Christoffel words.

EXAMPLES:

```
sage: w = words.LowerChristoffelWord(5,9)
sage: w
word: 00100100100101
sage: w1, w2 = w.standard_factorization()
sage: w1
word: 001
sage: w2
word: 00100100101
```

```
>>> from sage.all import *
>>> w = words.LowerChristoffelWord(Integer(5),Integer(9))
>>> w
word: 00100100100101
>>> w1, w2 = w.standard_factorization()
>>> w1
word: 001
>>> w2
word: 00100100101
```

```
sage: w = words.LowerChristoffelWord(51,37)
sage: w1, w2 = w.standard_factorization()
sage: w1
word: 0101011010101101011
sage: w2
word: 01010110101011010110101011010101101011...
sage: w1 * w2 == w
True
```

```
>>> from sage.all import *
>>> w = words.LowerChristoffelWord(Integer(51),Integer(37))
>>> w1, w2 = w.standard_factorization()
>>> w1
word: 0101011010101101011
>>> w2
word: 01010110101011010110101011010101101011...
>>> w1 * w2 == w
True
```

class sage.combinat.words.word_generators.**WordGenerator**

Bases: `object`

Constructor of several famous words.

EXAMPLES:

```
sage: words.ThueMorseWord()
word: 0110100110010110100101100110100110010110...
```

```
>>> from sage.all import *
>>> words.ThueMorseWord()
word: 0110100110010110100101100110100110010110...
```

```
sage: words.FibonacciWord()
word: 0100101001001010010100100101001001010010...
```

```
>>> from sage.all import *
>>> words.FibonacciWord()
word: 0100101001001010010100100101001001010010...
```

```
sage: words.ChristoffelWord(5, 8)
word: 0010010100101
```

```
>>> from sage.all import *
>>> words.ChristoffelWord(Integer(5), Integer(8))
word: 0010010100101
```

```
sage: words.RandomWord(10, 4)     # not tested random
word: 1311131221
```

```
>>> from sage.all import *
>>> words.RandomWord(Integer(10), Integer(4))     # not tested random
word: 1311131221
```

```
sage: words.CodingOfRotationWord(alpha=0.618, beta=0.618)
word: 1010110101101101011010110110101101101011...
```

```
>>> from sage.all import *
>>> words.CodingOfRotationWord(alpha=RealNumber('0.618'), beta=RealNumber('0.618
↪'))
word: 1010110101101101011010110110101101101011...
```

```
sage: tm = WordMorphism('a->ab,b->ba')
sage: fib = WordMorphism('a->ab,b->a')
sage: tmword = words.ThueMorseWord([0, 1])
sage: from itertools import repeat
sage: words.s_adic(tmword, repeat('a'), {0:tm, 1:fib})
word: abbaababbaabbaabbaababbaaababbaabbaababba...
```

```
>>> from sage.all import *
>>> tm = WordMorphism('a->ab,b->ba')
>>> fib = WordMorphism('a->ab,b->a')
>>> tmword = words.ThueMorseWord([Integer(0), Integer(1)])
>>> from itertools import repeat
>>> words.s_adic(tmword, repeat('a'), {Integer(0):tm, Integer(1):fib})
word: abbaababbaabbaabbaababbaaababbaabbaababba...
```

Note

To see a list of all word constructors, type `words.` and then hit the `Tab` key. The documentation for each constructor includes information about each word, which provides a useful reference.

BaumSweetWord()

Return the Baum-Sweet Word.

The Baum-Sweet Sequence is an infinite word over the alphabet $\{0,1\}$ defined by the following string substitution rules:

$00 \rightarrow 0000$

$01 \rightarrow 1001$

$10 \rightarrow 0100$

$11 \rightarrow 1101$

The substitution rule above can be considered as a morphism on the submonoid of $\{0,1\}$ generated by $\{00,01,10,11\}$ (which is a free monoid on these generators).

It is also defined as the concatenation of the terms from the Baum-Sweet Sequence:

$$b_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } m \text{ is even} \\ b_{\frac{m-1}{2}}, & \text{if } m \text{ is odd} \end{cases}$$

where $n = m4^k$ and m is not divisible by 4 if $m \neq 0$.

The individual terms of the Baum-Sweet Sequence are also given by:

$$b_n = \begin{cases} 1, & \text{if the binary representation of} n \text{ contains no block of consecutive 0s of odd length} \\ 0, & \text{otherwise} \end{cases}$$

for $n > 0$ with $b_0 = 1$.

For more information see: Wikipedia article Baum-Sweet_sequence.

EXAMPLES:

Baum-Sweet Word:

```
sage: w = words.BaumSweetWord(); w
word: 1101100101001001100100000100100101001001...
```

```
>>> from sage.all import *
>>> w = words.BaumSweetWord(); w
word: 1101100101001001100100000100100101001001...
```

Block Definition:

```
sage: w = words.BaumSweetWord()
sage: f = lambda n: '1' if all(len(x)%2==0 for x in bin(n)[2:].split('1'))␣
↪else '0'
sage: all(f(i) == w[i] for i in range(1,100))
True
```

```
>>> from sage.all import *
>>> w = words.BaumSweetWord()
>>> f = lambda n: '1' if all(len(x)%Integer(2)==Integer(0) for x in␣
↪bin(n)[Integer(2):].split('1')) else '0'
>>> all(f(i) == w[i] for i in range(Integer(1),Integer(100)))
True
```

CharacteristicSturmianWord(*slope*, *alphabet=(0, 1)*, *bits=None*)

Return the characteristic Sturmian word (also called standard Sturmian word) of given slope.

Over a binary alphabet $\{a, b\}$, the characteristic Sturmian word c_α of irrational slope α is the infinite word satisfying $s_{\alpha,0} = ac_\alpha$ and $s'_{\alpha,0} = bc_\alpha$, where $s_{\alpha,0}$ and $s'_{\alpha,0}$ are respectively the lower and upper mechanical words with slope α and intercept 0. Equivalently, for irrational α, $c_\alpha = s_{\alpha,\alpha} = s'_{\alpha,\alpha}$.

Let $\alpha = [0, d_1 + 1, d_2, d_3, \ldots]$ be the continued fraction expansion of α. It has been shown that the characteristic Sturmian word of slope α is also the limit of the sequence: $s_0 = b, s_1 = a, \ldots, s_{n+1} = s_n^{d_n} s_{n-1}$ for $n > 0$.

See Section 2.1 of [Loth02] for more details.

INPUT:

- `slope` – the slope of the word. It can be one of the following:
  - real number in $]0, 1[$
  - iterable over the continued fraction expansion of a real number in $]0, 1[$
- `alphabet` – any container of length two that is suitable to build an instance of OrderedAlphabet (`list`, `tuple`, `str`, …)
- `bits` – integer (optional and considered only if `slope` is a real number); the number of bits to consider when computing the continued fraction

OUTPUT: word

ALGORITHM:

Let $[0, d_1 + 1, d_2, d_3, \ldots]$ be the continued fraction expansion of α. Then, the characteristic Sturmian word of slope α is the limit of the sequence: $s_0 = b$, $s_1 = a$ and $s_{n+1} = s_n^{d_n} s_{n-1}$ for $n > 0$.

EXAMPLES:

From real slope:

```
sage: words.CharacteristicSturmianWord(1/golden_ratio^2)                       #␣
↪needs sage.symbolic
word: 0100101001001010010100100101001001010010...
sage: words.CharacteristicSturmianWord(4/5)                                    #␣
↪needs sage.rings.real_mpfr
word: 11110
sage: words.CharacteristicSturmianWord(5/14)                                   #␣
↪needs sage.rings.real_mpfr
word: 01001001001001
sage: words.CharacteristicSturmianWord(pi - 3)                                 #␣
↪needs sage.symbolic
word: 0000001000000100000010000001000000100000...
```

```
>>> from sage.all import *
>>> words.CharacteristicSturmianWord(Integer(1)/golden_ratio**Integer(2))     ␣
↪                # needs sage.symbolic
word: 0100101001001010010100100101001001010010...
>>> words.CharacteristicSturmianWord(Integer(4)/Integer(5))                   ␣
↪               # needs sage.rings.real_mpfr
word: 11110
>>> words.CharacteristicSturmianWord(Integer(5)/Integer(14))                  ␣
↪               # needs sage.rings.real_mpfr
```

(continues on next page)

(continued from previous page)

```
word: 01001001001001
>>> words.CharacteristicSturmianWord(pi - Integer(3))
↪     # needs sage.symbolic
word: 0000001000000100000010000001000000100000...
```

From an iterator of the continued fraction expansion of a real:

```
sage: def cf():
....:     yield 0
....:     yield 2
....:     while True: yield 1
sage: F = words.CharacteristicSturmianWord(cf()); F                     #␣
↪needs sage.rings.real_mpfr
word: 0100101001001010010100100101001001010010...
sage: Fib = words.FibonacciWord(); Fib
word: 0100101001001010010100100101001001010010...
sage: F[:10000] == Fib[:10000]                                          #␣
↪needs sage.rings.real_mpfr
True
```

```
>>> from sage.all import *
>>> def cf():
...     yield Integer(0)
...     yield Integer(2)
...     while True: yield Integer(1)
>>> F = words.CharacteristicSturmianWord(cf()); F                      #␣
↪needs sage.rings.real_mpfr
word: 0100101001001010010100100101001001010010...
>>> Fib = words.FibonacciWord(); Fib
word: 0100101001001010010100100101001001010010...
>>> F[:Integer(10000)] == Fib[:Integer(10000)]
↪                # needs sage.rings.real_mpfr
True
```

The alphabet may be specified:

```
sage: words.CharacteristicSturmianWord(cf(), 'rs')                      #␣
↪needs sage.rings.real_mpfr
word: rsrrsrsrrsrrsrsrrsrsrrsrrsrsrrsrrsrsrrsr...
```

```
>>> from sage.all import *
>>> words.CharacteristicSturmianWord(cf(), 'rs')                       #␣
↪needs sage.rings.real_mpfr
word: rsrrsrsrrsrrsrsrrsrsrrsrrsrsrrsrrsrsrrsr...
```

The characteristic sturmian word of slope $(\sqrt{3}-1)/2$:

```
sage: words.CharacteristicSturmianWord((sqrt(3)-1)/2)                   #␣
↪needs sage.symbolic
word: 0100100101001001001010010010010100100101...
```

```
>>> from sage.all import *
>>> words.CharacteristicSturmianWord((sqrt(Integer(3))-Integer(1))/
↪Integer(2))                                # needs sage.symbolic
word: 0100100101001001001010010010010100100101...
```

The same word defined from the continued fraction expansion of $(\sqrt{3}-1)/2$:

```
sage: from itertools import cycle, chain
sage: it = chain([0], cycle([2, 1]))
sage: words.CharacteristicSturmianWord(it)
word: 0100100101001001001010010010010100100101...
```

```
>>> from sage.all import *
>>> from itertools import cycle, chain
>>> it = chain([Integer(0)], cycle([Integer(2), Integer(1)]))
>>> words.CharacteristicSturmianWord(it)
word: 0100100101001001001010010010010100100101...
```

The first terms of the standard sequence of the characteristic sturmian word of slope $(\sqrt{3}-1)/2$:

```
sage: words.CharacteristicSturmianWord([0,2])
word: 01
sage: words.CharacteristicSturmianWord([0,2,1])
word: 010
sage: words.CharacteristicSturmianWord([0,2,1,2])
word: 01001001
sage: words.CharacteristicSturmianWord([0,2,1,2,1])
word: 01001001010
sage: words.CharacteristicSturmianWord([0,2,1,2,1,2])
word: 010010010100100100101001001001
sage: words.CharacteristicSturmianWord([0,2,1,2,1,2,1])
word: 0100100101001001001010010010010100100101...
```

```
>>> from sage.all import *
>>> words.CharacteristicSturmianWord([Integer(0),Integer(2)])
word: 01
>>> words.CharacteristicSturmianWord([Integer(0),Integer(2),Integer(1)])
word: 010
>>> words.CharacteristicSturmianWord([Integer(0),Integer(2),Integer(1),
↪Integer(2)])
word: 01001001
>>> words.CharacteristicSturmianWord([Integer(0),Integer(2),Integer(1),
↪Integer(2),Integer(1)])
word: 01001001010
>>> words.CharacteristicSturmianWord([Integer(0),Integer(2),Integer(1),
↪Integer(2),Integer(1),Integer(2)])
word: 010010010100100100101001001001
>>> words.CharacteristicSturmianWord([Integer(0),Integer(2),Integer(1),
↪Integer(2),Integer(1),Integer(2),Integer(1)])
word: 0100100101001001001010010010010100100101...
```

ChristoffelWord

alias of *LowerChristoffelWord*

CodingOfRotationWord(*alpha*, *beta*, *x=0*, *alphabet=(0, 1)*)

Return the infinite word obtained from the coding of rotation of parameters (α, β, x) over the given two-letter alphabet.

The *coding of rotation* corresponding to the parameters (α, β, x) is the symbolic sequence $u = (u_n)_{n\geq 0}$ defined over the binary alphabet $\{0, 1\}$ by $u_n = 1$ if $x + n\alpha \in [0, \beta[$ and $u_n = 0$ otherwise. See [AC03].

EXAMPLES:

```
sage: alpha = 0.45
sage: beta = 0.48
sage: words.CodingOfRotationWord(0.45, 0.48)
word: 1101010101001010101011010101010010101010...
```

```
>>> from sage.all import *
>>> alpha = RealNumber('0.45')
>>> beta = RealNumber('0.48')
>>> words.CodingOfRotationWord(RealNumber('0.45'), RealNumber('0.48'))
word: 1101010101001010101011010101010010101010...
```

```
sage: words.CodingOfRotationWord(0.45, 0.48, alphabet='xy')
word: yyxyxyxyxyxxyxyxyxyxyyxyxyxyxyxxyxyxyxyx...
```

```
>>> from sage.all import *
>>> words.CodingOfRotationWord(RealNumber('0.45'), RealNumber('0.48'),␣
↪alphabet='xy')
word: yyxyxyxyxyxxyxyxyxyxyyxyxyxyxyxxyxyxyxyx...
```

FibonacciWord(*alphabet=(0, 1)*, *construction_method='recursive'*)

Return the Fibonacci word on the given two-letter alphabet.

INPUT:

- `alphabet` – any container of length two that is suitable to build an instance of OrderedAlphabet (`list`, `tuple`, `str`, ...)
- `construction_method` – can be any of the following: "recursive", "fixed point", "function" (see below for definitions)

Recursive construction: the Fibonacci word is the limit of the following sequence of words: $S_0 = 0$, $S_1 = 01$, $S_n = S_{n-1}S_{n-2}$ for $n \geq 2$.

Fixed point construction: the Fibonacci word is the fixed point of the morphism: $0 \mapsto 01$ and $1 \mapsto 0$. Hence, it can be constructed by the following read-write process:

1. beginning at the first letter of 01,
2. if the next letter is 0, append 01 to the word;
3. if the next letter is 1, append 1 to the word;
4. move to the next letter of the word.

Function: Over the alphabet $\{1, 2\}$, the n-th letter of the Fibonacci word is $\lfloor (n + 2)\varphi\rfloor - \lfloor (n + 1)\varphi\rfloor$ where $\varphi = (1 + \sqrt{5})/2$ is the golden ratio.

EXAMPLES:

```
sage: w = words.FibonacciWord(construction_method='recursive'); w
word: 0100101001001010010100100101001001010010...
```

```
>>> from sage.all import *
>>> w = words.FibonacciWord(construction_method='recursive'); w
word: 0100101001001010010100100101001001010010...
```

```
sage: v = words.FibonacciWord(construction_method='recursive', alphabet='ab');
↪ v
word: abaababaabaababaababaabaababaabaababaaba...
```

```
>>> from sage.all import *
>>> v = words.FibonacciWord(construction_method='recursive', alphabet='ab'); v
word: abaababaabaababaababaabaababaabaababaaba...
```

```
sage: u = words.FibonacciWord(construction_method="fixed point"); u
word: 0100101001001010010100100101001001010010...
```

```
>>> from sage.all import *
>>> u = words.FibonacciWord(construction_method="fixed point"); u
word: 0100101001001010010100100101001001010010...
```

```
sage: words.FibonacciWord(construction_method="fixed point", alphabet=[4, 1])
word: 4144141441441414414141441441414414414141...
```

```
>>> from sage.all import *
>>> words.FibonacciWord(construction_method="fixed point",␣
↪alphabet=[Integer(4), Integer(1)])
word: 4144141441441414414141441441414414414141...
```

```
sage: words.FibonacciWord([0,1], 'function')                                #␣
↪needs sage.symbolic
word: 0100101001001010010100100101001001010010...
sage: words.FibonacciWord('ab', 'function')                                 #␣
↪needs sage.symbolic
word: abaababaabaababaababaabaababaabaababaaba...
```

```
>>> from sage.all import *
>>> words.FibonacciWord([Integer(0),Integer(1)], 'function')                  ␣
↪              # needs sage.symbolic
word: 0100101001001010010100100101001001010010...
>>> words.FibonacciWord('ab', 'function')                                 #␣
↪needs sage.symbolic
word: abaababaabaababaababaabaababaabaababaaba...
```

FixedPointOfMorphism(*morphism*, *first_letter*)

Return the fixed point of the morphism beginning with `first_letter`.

A *fixed point* of a morphism φ is a word w such that $\varphi(w) = w$.

INPUT:

- `morphism` – endomorphism prolongable on `first_letter`. It must be something that WordMorphism's constructor understands (dict, str, …).
- `first_letter` – the first letter of the fixed point

OUTPUT: the fixed point of the morphism beginning with `first_letter`

EXAMPLES:

```
sage: mu = {0:[0,1], 1:[1,0]}
sage: tm = words.FixedPointOfMorphism(mu,0); tm
word: 0110100110010110100101100110100110010110...
sage: TM = words.ThueMorseWord()
sage: tm[:1000] == TM[:1000]                                                    #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> mu = {Integer(0):[Integer(0),Integer(1)], Integer(1):[Integer(1),
↪Integer(0)]}
>>> tm = words.FixedPointOfMorphism(mu,Integer(0)); tm
word: 0110100110010110100101100110100110010110...
>>> TM = words.ThueMorseWord()
>>> tm[:Integer(1000)] == TM[:Integer(1000)]                                    ␣
↪                # needs sage.modules
True
```

```
sage: mu = {0:[0,1], 1:[0]}
sage: f = words.FixedPointOfMorphism(mu,0); f
word: 0100101001001010010100100101001001010010...
sage: F = words.FibonacciWord(); F
word: 0100101001001010010100100101001001010010...
sage: f[:1000] == F[:1000]                                                      #␣
↪needs sage.modules
True
```

```
>>> from sage.all import *
>>> mu = {Integer(0):[Integer(0),Integer(1)], Integer(1):[Integer(0)]}
>>> f = words.FixedPointOfMorphism(mu,Integer(0)); f
word: 0100101001001010010100100101001001010010...
>>> F = words.FibonacciWord(); F
word: 0100101001001010010100100101001001010010...
>>> f[:Integer(1000)] == F[:Integer(1000)]                                      ␣
↪                # needs sage.modules
True
```

```
sage: fp = words.FixedPointOfMorphism('a->abc,b->,c->','a'); fp
word: abc
```

```
>>> from sage.all import *
>>> fp = words.FixedPointOfMorphism('a->abc,b->,c->','a'); fp
word: abc
```

KolakoskiWord(*alphabet=(1, 2)*)

Return the Kolakoski word over the given alphabet and starting with the first letter of the alphabet.

Let $A = \{a, b\}$ be an alphabet, where a and b are two distinct positive integers. The Kolakoski word $K_{a,b}$ over A and starting with a is the unique infinite word w such that $w = \Delta(w)$, where $\Delta(w)$ is the word encoding the runs of w (see `delta()` method on words for more details).

Note that $K_{a,b} \neq K_{b,a}$. On the other hand, the words $K_{a,b}$ and $K_{b,a}$ are the unique two words over A that are fixed by Δ.

Also note that the Kolakoski word is also known as the Oldenburger word.

INPUT:

- `alphabet` – (default: (1,2)) an iterable of two positive integers

OUTPUT: infinite word

EXAMPLES:

The usual Kolakoski word:

```
sage: w = words.KolakoskiWord()
sage: w
word: 1221121221221121122121121221121121221221...
sage: w.delta()
word: 1221121221221121122121121221121121221221...
```

```
>>> from sage.all import *
>>> w = words.KolakoskiWord()
>>> w
word: 1221121221221121122121121221121121221221...
>>> w.delta()
word: 1221121221221121122121121221121121221221...
```

The other Kolakoski word on the same alphabet:

```
sage: w = words.KolakoskiWord(alphabet = (2,1))
sage: w
word: 2211212212211211221211212211211212212211...
sage: w.delta()
word: 2211212212211211221211212211211212212211...
```

```
>>> from sage.all import *
>>> w = words.KolakoskiWord(alphabet = (Integer(2),Integer(1)))
>>> w
word: 2211212212211211221211212211211212212211...
>>> w.delta()
word: 2211212212211211221211212211211212212211...
```

It is naturally generalized to any two integers alphabet:

```
sage: w = words.KolakoskiWord(alphabet = (2,5))
sage: w
word: 2255222225555522552255225555522222555552...
sage: w.delta()
word: 2255222225555522552255225555522222555552...
```

```
>>> from sage.all import *
>>> w = words.KolakoskiWord(alphabet = (Integer(2),Integer(5)))
>>> w
word: 2255222225555522552255225555522222555552...
>>> w.delta()
word: 2255222225555522552255225555522222555552...
```

REFERENCES:

LowerChristoffelWord

alias of *LowerChristoffelWord*

LowerMechanicalWord(*alpha*, *rho=0*, *alphabet=None*)

Return the lower mechanical word with slope α and intercept ρ

The lower mechanical word $s_{\alpha,\rho}$ with slope α and intercept ρ is defined by $s_{\alpha,\rho}(n) = \lfloor\alpha(n+1)+\rho\rfloor - \lfloor\alpha n+\rho\rfloor$. [Loth02]

INPUT:

- `alpha` – real number such that $0 \leq \alpha \leq 1$
- `rho` – real number (default: 0)
- `alphabet` – iterable of two elements or `None` (default: `None`)

OUTPUT: infinite word

EXAMPLES:

```
sage: words.LowerMechanicalWord(1/golden_ratio^2)                          #␣
↪needs sage.symbolic
word: 0010010100100101001010010010100100101001...
sage: words.LowerMechanicalWord(1/5)                                       #␣
↪needs sage.symbolic
word: 0000100001000010000100001000010000100001...
sage: words.LowerMechanicalWord(1/pi)                                      #␣
↪needs sage.symbolic
word: 0001001001001001001001000100100100100100...
```

```
>>> from sage.all import *
>>> words.LowerMechanicalWord(Integer(1)/golden_ratio**Integer(2))         ␣
↪                # needs sage.symbolic
word: 0010010100100101001010010010100100101001...
>>> words.LowerMechanicalWord(Integer(1)/Integer(5))                       ␣
↪               # needs sage.symbolic
word: 0000100001000010000100001000010000100001...
>>> words.LowerMechanicalWord(Integer(1)/pi)                               ␣
↪     # needs sage.symbolic
word: 0001001001001001001001000100100100100100...
```

MinimalSmoothPrefix(*n*)

This function finds and returns the minimal smooth prefix of length `n`.

See [BMP2007] for a definition.

INPUT:

- `n` – the desired length of the prefix

OUTPUT: word; the prefix

Note: Be patient, this function can take a really long time if asked for a large prefix.

EXAMPLES:

```
sage: words.MinimalSmoothPrefix(10)
word: 1212212112
```

```
>>> from sage.all import *
>>> words.MinimalSmoothPrefix(Integer(10))
word: 1212212112
```

PalindromicDefectWord(*k=1*, *alphabet='ab'*)

Return the finite word $w = ab^k ab^{k-1} aab^{k-1} ab^k a$.

As described by Brlek, Hamel, Nivat and Reutenauer in [BHNR2004], this finite word w is such that the infinite periodic word w^ω has palindromic defect `k`.

INPUT:

- `k` – positive integer (default: 1)
- `alphabet` – iterable of size two (default: `'ab'`)

OUTPUT: finite word

EXAMPLES:

```
sage: words.PalindromicDefectWord(10)
word: abbbbbbbbbbabbbbbbbbbaabbbbbbbbbabbbbbbbb...
```

```
>>> from sage.all import *
>>> words.PalindromicDefectWord(Integer(10))
word: abbbbbbbbbbabbbbbbbbbaabbbbbbbbbabbbbbbbb...
```

```
sage: w = words.PalindromicDefectWord(3)
sage: w
word: abbbabbaabbabbba
sage: w.defect()
0
sage: (w^2).defect()
3
sage: (w^3).defect()
3
```

```
>>> from sage.all import *
>>> w = words.PalindromicDefectWord(Integer(3))
>>> w
word: abbbabbaabbabbba
>>> w.defect()
0
```

(continues on next page)

(continued from previous page)

```
>>> (w**Integer(2)).defect()
3
>>> (w**Integer(3)).defect()
3
```

On other alphabets:

```
sage: words.PalindromicDefectWord(3, alphabet='cd')
word: cdddcddccddcdddc
sage: words.PalindromicDefectWord(3, alphabet=['c', 3])
word: c333c33cc33c333c
```

```
>>> from sage.all import *
>>> words.PalindromicDefectWord(Integer(3), alphabet='cd')
word: cdddcddccddcdddc
>>> words.PalindromicDefectWord(Integer(3), alphabet=['c', Integer(3)])
word: c333c33cc33c333c
```

RandomWord(*n*, *m=2*, *alphabet=None*)

Return a random word of length n over the given m-letter alphabet.

INPUT:

- `n` – integer; the length of the word
- `m` – integer (default: 2); the size of the output alphabet
- `alphabet` – (default: $\{0, 1, ..., m-1\}$) any container of length m that is suitable to build an instance of OrderedAlphabet (`list`, `tuple`, `str`, ...)

EXAMPLES:

```
sage: words.RandomWord(10)         # random results
word: 0110100101
sage: words.RandomWord(10, 4)      # random results
word: 0322313320
sage: words.RandomWord(100, 7)     # random results
word: 2630644023642516442650025611300034413310...
sage: words.RandomWord(100, 7, range(-3,4))  # random results
word: 1,3,-1,-1,3,2,2,0,1,-2,1,-1,-3,-2,2,0,3,0,-3,0,3,0,-2,-2,2,0,1,-3,2,-2,-
↪2,2,0,2,1,-2,-3,-2,-1,0,...
sage: words.RandomWord(100, 5, "abcde") # random results
word: acebeaaccdbedbbbdeadeebbdeeebeaaacbadaac...
sage: words.RandomWord(17, 5, "abcde")     # random results
word: dcacbbecbddebaadd
```

```
>>> from sage.all import *
>>> words.RandomWord(Integer(10))         # random results
word: 0110100101
>>> words.RandomWord(Integer(10), Integer(4))      # random results
word: 0322313320
>>> words.RandomWord(Integer(100), Integer(7))     # random results
word: 2630644023642516442650025611300034413310...
>>> words.RandomWord(Integer(100), Integer(7), range(-Integer(3),Integer(4)))␣
```

(continues on next page)

(continued from previous page)

```
↪ # random results
word: 1,3,-1,-1,3,2,2,0,1,-2,1,-1,-3,-2,2,0,3,0,-3,0,3,0,-2,-2,2,0,1,-3,2,-2,-
↪2,2,0,2,1,-2,-3,-2,-1,0,...
>>> words.RandomWord(Integer(100), Integer(5), "abcde") # random results
word: acebeaaccdbedbbbdeadeebbdeeebeaaacbadaac...
>>> words.RandomWord(Integer(17), Integer(5), "abcde")      # random results
word: dcacbbecbddebaadd
```

StandardEpisturmianWord(*directive_word*)

Return the standard episturmian word (or epistandard word) directed by directive_word. Over a 2-letter alphabet, this function gives characteristic Sturmian words.

An infinite word w over a finite alphabet A is said to be *standard episturmian* (or *epistandard*) iff there exists an infinite word $x_1x_2x_3\cdots$ over A (called the *directive word* of w) such that w is the limit as n goes to infinity of $Pal(x_1\cdots x_n)$, where Pal is the iterated palindromic closure function.

Note that an infinite word is *episturmian* if it has the same set of factors as some epistandard word.

See for instance [DJP2001], [JP2002], and [GJ2007].

INPUT:

- `directive_word` – an infinite word or a period of a periodic infinite word

EXAMPLES:

```
sage: Fibonacci = words.StandardEpisturmianWord(Words('ab')('ab')); Fibonacci
word: abaababaabaababaababaabaababaabaababaaba...
sage: Tribonacci = words.StandardEpisturmianWord(Words('abc')('abc'));␣
↪Tribonacci
word: abacabaabacababacabaabacabacabaabacababa...
sage: S = words.StandardEpisturmianWord(Words('abcd')('aabcabada')); S
word: aabaacaabaaabaacaabaabaacaabaaabaacaabaa...
sage: S = words.StandardEpisturmianWord(Fibonacci); S
word: abaabaababaabaabaababaabaababaabaabaabab...
sage: S[:25]
word: abaabaababaabaabaababaaba
sage: S = words.StandardEpisturmianWord(Tribonacci); S
word: abaabacabaabaabacabaababaabacabaabaabaca...
sage: words.StandardEpisturmianWord(123)
Traceback (most recent call last):
...
TypeError: directive_word is not a word, so it cannot be used to build an␣
↪episturmian word
sage: words.StandardEpisturmianWord(Words('ab'))
Traceback (most recent call last):
...
TypeError: directive_word is not a word, so it cannot be used to build an␣
↪episturmian word
```

```
>>> from sage.all import *
>>> Fibonacci = words.StandardEpisturmianWord(Words('ab')('ab')); Fibonacci
word: abaababaabaababaababaabaababaabaababaaba...
>>> Tribonacci = words.StandardEpisturmianWord(Words('abc')('abc'));␣
↪Tribonacci
```

(continues on next page)

(continued from previous page)

```
word: abacabaabacababacabaabacabacabaabacababa...
>>> S = words.StandardEpisturmianWord(Words('abcd')('aabcabada')); S
word: aabaacaabaaabaacaabaabaacaabaaabaacaabaa...
>>> S = words.StandardEpisturmianWord(Fibonacci); S
word: abaababaabaababaababaabaababaabaababaabab...
>>> S[:Integer(25)]
word: abaababaabaababaababaaba
>>> S = words.StandardEpisturmianWord(Tribonacci); S
word: abaabacabaabaabacabaababaabacabaabaabaca...
>>> words.StandardEpisturmianWord(Integer(123))
Traceback (most recent call last):
...
TypeError: directive_word is not a word, so it cannot be used to build an␣
↪episturmian word
>>> words.StandardEpisturmianWord(Words('ab'))
Traceback (most recent call last):
...
TypeError: directive_word is not a word, so it cannot be used to build an␣
↪episturmian word
```

ThueMorseWord(*alphabet=(0, 1)*, *base=2*)

Return the (Generalized) Thue-Morse word over the given alphabet.

There are several ways to define the Thue-Morse word t. We use the following definition: $t[n]$ is the sum modulo m of the digits in the given base expansion of n.

See [BmBGL07], [Brlek89], and [MH38].

INPUT:

- `alphabet` – (default: `(0, 1)`) any container that is suitable to build an instance of OrderedAlphabet (`list`, `tuple`, `str`, ...)
- `base` – integer (default: 2); greater than or equal to 2

EXAMPLES:

Thue-Morse word:

```
sage: t = words.ThueMorseWord(); t
word: 0110100110010110100101100110100110010110...
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord(); t
word: 0110100110010110100101100110100110010110...
```

Thue-Morse word on other alphabets:

```
sage: t = words.ThueMorseWord('ab'); t
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord('ab'); t
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

```
sage: t = words.ThueMorseWord(['L1', 'L2'])
sage: t[:8]
word: L1,L2,L2,L1,L2,L1,L1,L2
```

```
>>> from sage.all import *
>>> t = words.ThueMorseWord(['L1', 'L2'])
>>> t[:Integer(8)]
word: L1,L2,L2,L1,L2,L1,L1,L2
```

Generalized Thue Morse word:

```
sage: words.ThueMorseWord(alphabet=(0,1,2), base=2)
word: 0112122012202001122020012001011212202001...
sage: t = words.ThueMorseWord(alphabet=(0,1,2), base=5); t
word: 0120112012201200120112012120122012001201...
sage: t[100:130].critical_exponent()
10/3
```

```
>>> from sage.all import *
>>> words.ThueMorseWord(alphabet=(Integer(0),Integer(1),Integer(2)),␣
↪base=Integer(2))
word: 0112122012202001122020012001011212202001...
>>> t = words.ThueMorseWord(alphabet=(Integer(0),Integer(1),Integer(2)),␣
↪base=Integer(5)); t
word: 0120112012201200120112012120122012001201...
>>> t[Integer(100):Integer(130)].critical_exponent()
10/3
```

REFERENCES:

UpperChristoffelWord(*p*, *q*, *alphabet=(0, 1)*)

Return the upper Christoffel word of slope p/q, where p and q are relatively prime nonnegative integers, over the given alphabet.

The *upper Christoffel word of slope `p/q`* is equal to the reversal of the lower Christoffel word of slope p/q. Equivalently, if xuy is the lower Christoffel word of slope p/q, where x and y are letters, then yux is the upper Christoffel word of slope p/q (because u is a palindrome).

INPUT:

- `alphabet` – any container of length two that is suitable to build an instance of OrderedAlphabet (`list`, `tuple`, `str`, …)

EXAMPLES:

```
sage: words.UpperChristoffelWord(1,0)
word: 1
```

```
>>> from sage.all import *
>>> words.UpperChristoffelWord(Integer(1),Integer(0))
word: 1
```

```
sage: words.UpperChristoffelWord(0,1)
word: 0
```

```
>>> from sage.all import *
>>> words.UpperChristoffelWord(Integer(0),Integer(1))
word: 0
```

```
sage: words.UpperChristoffelWord(1,1)
word: 10
```

```
>>> from sage.all import *
>>> words.UpperChristoffelWord(Integer(1),Integer(1))
word: 10
```

```
sage: words.UpperChristoffelWord(4,7)
word: 10100100100
```

```
>>> from sage.all import *
>>> words.UpperChristoffelWord(Integer(4),Integer(7))
word: 10100100100
```

UpperMechanicalWord(*alpha*, *rho=0*, *alphabet=None*)

Return the upper mechanical word with slope α and intercept ρ

The upper mechanical word $s'_{\alpha,\rho}$ with slope α and intercept ρ is defined by $s'_{\alpha,\rho}(n) = \lceil \alpha(n+1) + \rho \rceil - \lceil \alpha n + \rho \rceil$. [Loth02]

INPUT:

- `alpha` – real number such that $0 \leq \alpha \leq 1$
- `rho` – real number (default: 0)
- `alphabet` – iterable of two elements or `None` (default: `None`)

OUTPUT: infinite word

EXAMPLES:

```
sage: words.UpperMechanicalWord(1/golden_ratio^2)                          #␣
↪needs sage.symbolic
word: 1010010100100101001010010010100100101001...
sage: words.UpperMechanicalWord(1/5)                                       #␣
↪needs sage.symbolic
word: 1000010000100001000010000100001000010000...
sage: words.UpperMechanicalWord(1/pi)                                      #␣
↪needs sage.symbolic
word: 1001001001001001001001000100100100100100...
```

```
>>> from sage.all import *
>>> words.UpperMechanicalWord(Integer(1)/golden_ratio**Integer(2))          ␣
↪               # needs sage.symbolic
word: 1010010100100101001010010010100100101001...
>>> words.UpperMechanicalWord(Integer(1)/Integer(5))                        ␣
↪              # needs sage.symbolic
word: 1000010000100001000010000100001000010000...
>>> words.UpperMechanicalWord(Integer(1)/pi)                                ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.symbolic
word: 1001001001001001001001000100100100100100...
```

dual_fibonacci_tile(*n*)

Return the n-th dual Fibonacci Tile [BmBGL09].

EXAMPLES:

```
sage: for i in range(4): words.dual_fibonacci_tile(i)                          #␣
↪needs sage.modules
Path: 3210
Path: 32123032301030121012
Path: 3212303230103230321232101232123032123210...
Path: 3212303230103230321232101232123032123210...
```

```
>>> from sage.all import *
>>> for i in range(Integer(4)): words.dual_fibonacci_tile(i)                  ␣
↪     # needs sage.modules
Path: 3210
Path: 32123032301030121012
Path: 3212303230103230321232101232123032123210...
Path: 3212303230103230321232101232123032123210...
```

fibonacci_tile(*n*)

Return the n-th Fibonacci Tile [BmBGL09].

EXAMPLES:

```
sage: for i in range(3): words.fibonacci_tile(i)                               #␣
↪needs sage.modules
Path: 3210
Path: 323030101212
Path: 3230301030323212323032321210121232121010...
```

```
>>> from sage.all import *
>>> for i in range(Integer(3)): words.fibonacci_tile(i)                       ␣
↪     # needs sage.modules
Path: 3210
Path: 323030101212
Path: 3230301030323212323032321210121232121010...
```

s_adic(*sequence*, *letters*, *morphisms=None*)

Return the s-adic infinite word obtained from a sequence of morphisms applied on a letter.

DEFINITION (from [Fogg]):

Let w be a infinite word over an alphabet $A = A_0$. A standard representation of w is obtained from a sequence of substitutions $\sigma_k : A_{k+1} \to A_k$ and a sequence of letters $a_k \in A_k$ such that:

$$\lim_{k\to\infty} \sigma_0 \circ \sigma_1 \circ \cdots \sigma_k(a_k).$$

Given a set of substitutions S, we say that the representation is S-adic standard if the substitutions are chosen in S.

INPUT:

- `sequence` – an iterable sequence of indices or of morphisms. It may be finite or infinite. If `sequence` is infinite, the image of the $(i+1)$-th letter under the $(i+1)$-th morphism must start with the i-th letter.
- `letters` – a letter or a sequence of letters
- `morphisms` – dict, list, callable or `None` (default: `None`) an object that maps indices to morphisms. If `None`, then `sequence` must consist of morphisms.

OUTPUT: a word

EXAMPLES:

Let us define three morphisms and compute the first nested successive prefixes of the s-adic word:

```
sage: m1 = WordMorphism('e->gh,f->hg')
sage: m2 = WordMorphism('c->ef,d->e')
sage: m3 = WordMorphism('a->cd,b->dc')
sage: words.s_adic([m1],'e')
word: gh
sage: words.s_adic([m1,m2],'ec')
word: ghhg
sage: words.s_adic([m1,m2,m3],'eca')
word: ghhggh
```

```
>>> from sage.all import *
>>> m1 = WordMorphism('e->gh,f->hg')
>>> m2 = WordMorphism('c->ef,d->e')
>>> m3 = WordMorphism('a->cd,b->dc')
>>> words.s_adic([m1],'e')
word: gh
>>> words.s_adic([m1,m2],'ec')
word: ghhg
>>> words.s_adic([m1,m2,m3],'eca')
word: ghhggh
```

When the given sequence of morphism is finite, one may simply give the last letter, i.e. `'a'`, instead of giving all of them, i.e. `'eca'`:

```
sage: words.s_adic([m1,m2,m3],'a')
word: ghhggh
sage: words.s_adic([m1,m2,m3],'b')
word: ghghhg
```

```
>>> from sage.all import *
>>> words.s_adic([m1,m2,m3],'a')
word: ghhggh
>>> words.s_adic([m1,m2,m3],'b')
word: ghghhg
```

If the letters don't satisfy the hypothesis of the algorithm (nested prefixes), an error is raised:

```
sage: words.s_adic([m1,m2,m3],'ecb')
Traceback (most recent call last):
...
ValueError: the hypothesis of the algorithm used is not satisfied; the image
```

(continues on next page)

(continued from previous page)

```
↪of the 3-th letter (=b) under the 3-th morphism (=a->cd, b->dc) should␣
↪start with the 2-th letter (=c)
```

```
>>> from sage.all import *
>>> words.s_adic([m1,m2,m3],'ecb')
Traceback (most recent call last):
...
ValueError: the hypothesis of the algorithm used is not satisfied; the image␣
↪of the 3-th letter (=b) under the 3-th morphism (=a->cd, b->dc) should␣
↪start with the 2-th letter (=c)
```

Let's define the Thue-Morse morphism and the Fibonacci morphism which will be used below to illustrate more examples and let's import the `repeat` tool from the `itertools`:

```
sage: tm = WordMorphism('a->ab,b->ba')
sage: fib = WordMorphism('a->ab,b->a')
sage: from itertools import repeat
```

```
>>> from sage.all import *
>>> tm = WordMorphism('a->ab,b->ba')
>>> fib = WordMorphism('a->ab,b->a')
>>> from itertools import repeat
```

Two trivial examples of infinite s-adic words:

```
sage: words.s_adic(repeat(tm),repeat('a'))
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

```
>>> from sage.all import *
>>> words.s_adic(repeat(tm),repeat('a'))
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

```
sage: words.s_adic(repeat(fib),repeat('a'))
word: abaababaabaababaababaabaababaabaababaaba...
```

```
>>> from sage.all import *
>>> words.s_adic(repeat(fib),repeat('a'))
word: abaababaabaababaababaabaababaabaababaaba...
```

A less trivial infinite s-adic word:

```
sage: D = {4:tm,5:fib}
sage: tmword = words.ThueMorseWord([4,5])
sage: it = (D[a] for a in tmword)
sage: words.s_adic(it, repeat('a'))
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

```
>>> from sage.all import *
>>> D = {Integer(4):tm,Integer(5):fib}
>>> tmword = words.ThueMorseWord([Integer(4),Integer(5)])
>>> it = (D[a] for a in tmword)
```

(continues on next page)

(continued from previous page)

```
>>> words.s_adic(it, repeat('a'))
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

The same thing using a sequence of indices:

```
sage: tmword = words.ThueMorseWord([0,1])
sage: words.s_adic(tmword, repeat('a'), [tm,fib])
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

```
>>> from sage.all import *
>>> tmword = words.ThueMorseWord([Integer(0),Integer(1)])
>>> words.s_adic(tmword, repeat('a'), [tm,fib])
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

The correspondence of the indices may be given as a dict:

```
sage: words.s_adic(tmword, repeat('a'), {0:tm,1:fib})
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

```
>>> from sage.all import *
>>> words.s_adic(tmword, repeat('a'), {Integer(0):tm,Integer(1):fib})
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

because dict are more versatile for indices:

```
sage: tmwordTF = words.ThueMorseWord('TF')
sage: words.s_adic(tmwordTF, repeat('a'), {'T':tm,'F':fib})
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

```
>>> from sage.all import *
>>> tmwordTF = words.ThueMorseWord('TF')
>>> words.s_adic(tmwordTF, repeat('a'), {'T':tm,'F':fib})
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

or by a callable:

```
sage: f = lambda n: tm if n == 0 else fib
sage: words.s_adic(words.ThueMorseWord(), repeat('a'), f)
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

```
>>> from sage.all import *
>>> f = lambda n: tm if n == Integer(0) else fib
>>> words.s_adic(words.ThueMorseWord(), repeat('a'), f)
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

Random infinite s-adic words:

```
sage: from sage.misc.prandom import randint
sage: def it():
....:   while True: yield randint(0,1)
sage: words.s_adic(it(), repeat('a'), [tm,fib])  # random
word: abbaabababbaababbaabbaababbaabababbaabba...
```

(continues on next page)

(continued from previous page)

```
sage: words.s_adic(it(), repeat('a'), [tm,fib])  # random
word: abbaababbaabbaababbaababbaabbaababbaabba...
sage: words.s_adic(it(), repeat('a'), [tm,fib])  # random
word: abaaababaabaabaaababaabaaababaaababaabaa...
```

```
>>> from sage.all import *
>>> from sage.misc.prandom import randint
>>> def it():
...   while True: yield randint(Integer(0),Integer(1))
>>> words.s_adic(it(), repeat('a'), [tm,fib])  # random
word: abbaabababbaababbaabbaababbaabababbaabba...
>>> words.s_adic(it(), repeat('a'), [tm,fib])  # random
word: abbaababbaabbaababbaababbaabbaababbaabba...
>>> words.s_adic(it(), repeat('a'), [tm,fib])  # random
word: abaaababaabaabaaababaabaaababaaababaabaa...
```

An example where the sequences cycle on two morphisms and two letters:

```
sage: G = WordMorphism('a->cd,b->dc')
sage: H = WordMorphism('c->ab,d->ba')
sage: from itertools import cycle
sage: words.s_adic([G,H],'ac')
word: cddc
sage: words.s_adic(cycle([G,H]),cycle('ac'))
word: cddcdccddccdcddcdccdcddccddcdccddccdcddc...
```

```
>>> from sage.all import *
>>> G = WordMorphism('a->cd,b->dc')
>>> H = WordMorphism('c->ab,d->ba')
>>> from itertools import cycle
>>> words.s_adic([G,H],'ac')
word: cddc
>>> words.s_adic(cycle([G,H]),cycle('ac'))
word: cddcdccddccdcddcdccdcddccddcdccddccdcddc...
```

The morphism $\sigma : a \mapsto ba, b \mapsto b$ can't satisfy the hypothesis of the nested prefixes, but one may compute arbitrarily long finite words having the limit $\sigma^\omega(a)$:

```
sage: sigma = WordMorphism('a->ba,b->b')
sage: words.s_adic(repeat(sigma),repeat('a'))
Traceback (most recent call last):
...
ValueError: the hypothesis of the algorithm used is not satisfied; the image␣
↪of the 2-th letter (=a) under the 2-th morphism (=a->ba, b->b) should start␣
↪with the 1-th letter (=a)
sage: words.s_adic([sigma],'a')
word: ba
sage: words.s_adic([sigma,sigma],'a')
word: bba
sage: words.s_adic([sigma]*3,'a')
word: bbba
sage: words.s_adic([sigma]*4,'a')
```

(continues on next page)

(continued from previous page)

```
word: bbbba
sage: words.s_adic([sigma]*5,'a')
word: bbbbba
sage: words.s_adic([sigma]*6,'a')
word: bbbbbba
sage: words.s_adic([sigma]*7,'a')
word: bbbbbbba
```

```
>>> from sage.all import *
>>> sigma = WordMorphism('a->ba,b->b')
>>> words.s_adic(repeat(sigma),repeat('a'))
Traceback (most recent call last):
...
ValueError: the hypothesis of the algorithm used is not satisfied; the image␣
↪of the 2-th letter (=a) under the 2-th morphism (=a->ba, b->b) should start␣
↪with the 1-th letter (=a)
>>> words.s_adic([sigma],'a')
word: ba
>>> words.s_adic([sigma,sigma],'a')
word: bba
>>> words.s_adic([sigma]*Integer(3),'a')
word: bbba
>>> words.s_adic([sigma]*Integer(4),'a')
word: bbbba
>>> words.s_adic([sigma]*Integer(5),'a')
word: bbbbba
>>> words.s_adic([sigma]*Integer(6),'a')
word: bbbbbba
>>> words.s_adic([sigma]*Integer(7),'a')
word: bbbbbbba
```

The following examples illustrates an S-adic word defined over an infinite set S of morphisms x_h:

```
sage: x = lambda h:WordMorphism({1:[2],2:[3]+[1]*(h+1),3:[3]+[1]*h})
sage: for h in [0,1,2,3]:
....:     print("{} {}".format(h, x(h)))
0 1->2, 2->31, 3->3
1 1->2, 2->311, 3->31
2 1->2, 2->3111, 3->311
3 1->2, 2->31111, 3->3111
sage: w = Word(lambda n : valuation(n+1, 2) ); w
word: 0102010301020104010201030102010501020103...
sage: s = words.s_adic(w, repeat(3), x); s
word: 3232232232322322322323223223232232232232...
sage: prefixe = s[:10000]
sage: list(map(prefixe.number_of_factors, range(15)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: [_[i+1] - _[i] for i in range(len(_)-1)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> x = lambda h:WordMorphism({Integer(1):[Integer(2)],
```

(continues on next page)

(continued from previous page)

```
↪Integer(2):[Integer(3)]+[Integer(1)]*(h+Integer(1)),
↪Integer(3):[Integer(3)]+[Integer(1)]*h})
>>> for h in [Integer(0),Integer(1),Integer(2),Integer(3)]:
...     print("{} {}".format(h, x(h)))
0 1->2, 2->31, 3->3
1 1->2, 2->311, 3->31
2 1->2, 2->3111, 3->311
3 1->2, 2->31111, 3->3111
>>> w = Word(lambda n : valuation(n+Integer(1), Integer(2)) ); w
word: 0102010301020104010201030102010501020103...
>>> s = words.s_adic(w, repeat(Integer(3)), x); s
word: 3232232232322322322323223223232232232232...
>>> prefixe = s[:Integer(10000)]
>>> list(map(prefixe.number_of_factors, range(Integer(15))))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> [_[i+Integer(1)] - _[i] for i in range(len(_)-Integer(1))]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

AUTHORS:

- Sébastien Labbé (2009-12-18): initial version

5.1.369 Datatypes for words defined by iterators and callables

class sage.combinat.words.word_infinite_datatypes.**WordDatatype_callable**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype*

Datatype for a word defined by a callable.

class sage.combinat.words.word_infinite_datatypes.**WordDatatype_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: *WordDatatype_callable*

Datatype for a word defined by a callable.

flush()

Empty the associated cache of letters.

EXAMPLES:

The first 40 (by default) values are always cached:

```
sage: w = words.ThueMorseWord()
sage: w._letter_cache
{0: 0, 1: 1, 2: 1, 3: 0, 4: 1, 5: 0, 6: 0, 7: 1, 8: 1, 9: 0, 10: 0, 11: 1,␣
↪12: 0, 13: 1, 14: 1, 15: 0, 16: 1, 17: 0, 18: 0, 19: 1, 20: 0, 21: 1, 22: 1,
↪ 23: 0, 24: 0, 25: 1, 26: 1, 27: 0, 28: 1, 29: 0, 30: 0, 31: 1, 32: 1, 33:␣
↪0, 34: 0, 35: 1, 36: 0, 37: 1, 38: 1, 39: 0}
sage: w[100]
1
sage: w._letter_cache
{0: 0, 1: 1, 2: 1, 3: 0, 4: 1, 5: 0, 6: 0, 7: 1, 8: 1, 9: 0, 10: 0, 11: 1,␣
```

(continues on next page)

(continued from previous page)

```
↪12: 0, 13: 1, 14: 1, 15: 0, 16: 1, 17: 0, 18: 0, 19: 1, 20: 0, 21: 1, 22: 1,
↪ 23: 0, 24: 0, 25: 1, 26: 1, 27: 0, 28: 1, 29: 0, 30: 0, 31: 1, 32: 1, 33:␣
↪0, 34: 0, 35: 1, 36: 0, 37: 1, 38: 1, 39: 0, 100: 1}
sage: w.flush()
sage: w._letter_cache
{}
```

```
>>> from sage.all import *
>>> w = words.ThueMorseWord()
>>> w._letter_cache
{0: 0, 1: 1, 2: 1, 3: 0, 4: 1, 5: 0, 6: 0, 7: 1, 8: 1, 9: 0, 10: 0, 11: 1,␣
↪12: 0, 13: 1, 14: 1, 15: 0, 16: 1, 17: 0, 18: 0, 19: 1, 20: 0, 21: 1, 22: 1,
↪ 23: 0, 24: 0, 25: 1, 26: 1, 27: 0, 28: 1, 29: 0, 30: 0, 31: 1, 32: 1, 33:␣
↪0, 34: 0, 35: 1, 36: 0, 37: 1, 38: 1, 39: 0}
>>> w[Integer(100)]
1
>>> w._letter_cache
{0: 0, 1: 1, 2: 1, 3: 0, 4: 1, 5: 0, 6: 0, 7: 1, 8: 1, 9: 0, 10: 0, 11: 1,␣
↪12: 0, 13: 1, 14: 1, 15: 0, 16: 1, 17: 0, 18: 0, 19: 1, 20: 0, 21: 1, 22: 1,
↪ 23: 0, 24: 0, 25: 1, 26: 1, 27: 0, 28: 1, 29: 0, 30: 0, 31: 1, 32: 1, 33:␣
↪0, 34: 0, 35: 1, 36: 0, 37: 1, 38: 1, 39: 0, 100: 1}
>>> w.flush()
>>> w._letter_cache
{}
```

class sage.combinat.words.word_infinite_datatypes.**WordDatatype_iter**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype*

INPUT:

- parent – a parent
- iter – an iterator
- length – (default: None) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length='unknown', caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
sage: w = Word(iter(s), length='unknown', caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True
```

```
>>> from sage.all import *
>>> w = Word(iter("abbabaab"), length='unknown', caching=False); w
word: abbabaab
>>> isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
>>> w.length() is None
False
>>> w.length()
8
>>> s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
>>> w = Word(iter(s), length='unknown', caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
>>> w.length() is None
True
```

```
sage: w = Word(iter("abbabaab"), length='finite', caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

```
>>> from sage.all import *
>>> w = Word(iter("abbabaab"), length='finite', caching=False); w
word: abbabaab
>>> isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
>>> w.length()
8
>>> w = Word(iter("abbabaab"), length=Integer(8), caching=False); w
word: abbabaab
>>> isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
>>> w.length()
8
```

class `sage.combinat.words.word_infinite_datatypes.`**`WordDatatype_iter_with_caching`**(*parent*, *iter*, *length=None*)

Bases: *WordDatatype_iter*

INPUT:

- `parent` – a parent
- `iter` – an iterator
- `length` – (default: `None`) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
sage: w = Word(iter("abbabaab"), length='finite'); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length='unknown'); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

```
>>> from sage.all import *
>>> import itertools
>>> Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
>>> w = Word(iter("abbabaab"), length='finite'); w
word: abbabaab
>>> w.length()
8
>>> w = Word(iter("abbabaab"), length='unknown'); w
word: abbabaab
>>> w.length()
8
>>> list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
>>> w.length()
8
>>> w = Word(iter("abbabaab"), length=Integer(8))
>>> w._len
8
```

flush()

Delete the cached values.

EXAMPLES:

```
sage: from itertools import count
sage: w = Word(count())
sage: w._last_index, len(w._list)
(39, 40)
sage: w[43]
43
sage: w._last_index, len(w._list)
(43, 44)
```

(continues on next page)

(continued from previous page)

```
sage: w.flush()
sage: w._last_index, w._list
(-1, [])
```

```
>>> from sage.all import *
>>> from itertools import count
>>> w = Word(count())
>>> w._last_index, len(w._list)
(39, 40)
>>> w[Integer(43)]
43
>>> w._last_index, len(w._list)
(43, 44)
>>> w.flush()
>>> w._last_index, w._list
(-1, [])
```

5.1.370 User-customizable options for words

`sage.combinat.words.word_options.WordOptions(**kwargs)`

Set the global options for elements of the word class. The defaults are for words to be displayed in list notation.

INPUT:

- `display` – 'string' (default), or 'list', words are displayed in string or list notation
- `truncate` – boolean (default: `True`); whether to truncate the string output of long words (see truncate_length below)
- `truncate_length` – integer (default: 40); if the length of the word is greater than this integer, then the word is truncated
- `letter_separator` – string (default: `','`); if the string representation of letters have length greater than 1, then the letters are separated by this string in the string representation of the word

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: w = Word([2,1,3,12])
sage: u = Word("abba")
sage: WordOptions(display='list')
sage: w
word: [2, 1, 3, 12]
sage: u
word: ['a', 'b', 'b', 'a']
sage: WordOptions(display='string')
sage: w
word: 2,1,3,12
sage: u
word: abba
```

```
>>> from sage.all import *
>>> w = Word([Integer(2),Integer(1),Integer(3),Integer(12)])
```

(continues on next page)

(continued from previous page)

```
>>> u = Word("abba")
>>> WordOptions(display='list')
>>> w
word: [2, 1, 3, 12]
>>> u
word: ['a', 'b', 'b', 'a']
>>> WordOptions(display='string')
>>> w
word: 2,1,3,12
>>> u
word: abba
```

5.1.371 Set of words

To define a new class of words, please refer to the documentation file: sage/combinat/words/notes/word_inheritance_howto.rst

AUTHORS:

- Franco Saliola (2008-12-17): merged into sage
- Sebastien Labbe (2008-12-17): merged into sage
- Arnaud Bergeron (2008-12-17): merged into sage
- Sebastien Labbe (2009-07-21): Improved morphism iterator (Issue #6571).
- Vincent Delecroix (2015): classes simplifications (Issue #19619)

EXAMPLES:

```
sage: Words()
Finite and infinite words over Set of Python objects of class 'object'
sage: Words(4)
Finite and infinite words over {1, 2, 3, 4}
sage: Words(4,5)
Words of length 5 over {1, 2, 3, 4}

sage: FiniteWords('ab')
Finite words over {'a', 'b'}
sage: InfiniteWords('natural numbers')
Infinite words over Non negative integers
```

```
>>> from sage.all import *
>>> Words()
Finite and infinite words over Set of Python objects of class 'object'
>>> Words(Integer(4))
Finite and infinite words over {1, 2, 3, 4}
>>> Words(Integer(4),Integer(5))
Words of length 5 over {1, 2, 3, 4}

>>> FiniteWords('ab')
Finite words over {'a', 'b'}
>>> InfiniteWords('natural numbers')
Infinite words over Non negative integers
```

class sage.combinat.words.words.**AbstractLanguage**(*alphabet=None*, *category=None*)

Bases: Parent

Abstract base class.

This is *not* to be used by any means. This class gather previous features of set of words (prior to Issue #19619). In the future that class might simply disappear or become a common base class for all languages. In the latter case, its name would possibly change to Language.

alphabet()

EXAMPLES:

```
sage: Words(NN).alphabet()
Non negative integer semiring

sage: InfiniteWords([1,2,3]).alphabet()
{1, 2, 3}
sage: InfiniteWords('ab').alphabet()
{'a', 'b'}

sage: FiniteWords([1,2,3]).alphabet()
{1, 2, 3}
sage: FiniteWords().alphabet()
Set of Python objects of class 'object'
```

```
>>> from sage.all import *
>>> Words(NN).alphabet()
Non negative integer semiring

>>> InfiniteWords([Integer(1),Integer(2),Integer(3)]).alphabet()
{1, 2, 3}
>>> InfiniteWords('ab').alphabet()
{'a', 'b'}

>>> FiniteWords([Integer(1),Integer(2),Integer(3)]).alphabet()
{1, 2, 3}
>>> FiniteWords().alphabet()
Set of Python objects of class 'object'
```

identity_morphism()

Return the identity morphism from self to itself.

EXAMPLES:

```
sage: W = Words('ab')
sage: W.identity_morphism()
WordMorphism: a->a, b->b
```

```
>>> from sage.all import *
>>> W = Words('ab')
>>> W.identity_morphism()
WordMorphism: a->a, b->b
```

```
sage: W = Words(range(3))
sage: W.identity_morphism()
WordMorphism: 0->0, 1->1, 2->2
```

```
>>> from sage.all import *
>>> W = Words(range(Integer(3)))
>>> W.identity_morphism()
WordMorphism: 0->0, 1->1, 2->2
```

There is no support yet for infinite alphabet:

```
sage: W = Words(alphabet=Alphabet(name='NN'))
sage: W
Finite and infinite words over Non negative integers
sage: W.identity_morphism()
Traceback (most recent call last):
...
NotImplementedError: size of alphabet must be finite
```

```
>>> from sage.all import *
>>> W = Words(alphabet=Alphabet(name='NN'))
>>> W
Finite and infinite words over Non negative integers
>>> W.identity_morphism()
Traceback (most recent call last):
...
NotImplementedError: size of alphabet must be finite
```

class `sage.combinat.words.words.`**`FiniteOrInfiniteWords`**(*alphabet*)

Bases: `AbstractLanguage`

INPUT:

- `alphabet` – the underlying alphabet

`cardinality`()

Return the cardinality of this set of words.

EXAMPLES:

```
sage: Words('abcd').cardinality()
+Infinity
sage: Words('a').cardinality()
+Infinity
sage: Words('').cardinality()
1
```

```
>>> from sage.all import *
>>> Words('abcd').cardinality()
+Infinity
>>> Words('a').cardinality()
+Infinity
>>> Words('').cardinality()
1
```

factors()

Return the set of finite words.

EXAMPLES:

```
sage: Words('ab').finite_words()
Finite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> Words('ab').finite_words()
Finite words over {'a', 'b'}
```

finite_words()

Return the set of finite words.

EXAMPLES:

```
sage: Words('ab').finite_words()
Finite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> Words('ab').finite_words()
Finite words over {'a', 'b'}
```

infinite_words()

Return the set of infinite words.

EXAMPLES:

```
sage: Words('ab').infinite_words()
Infinite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> Words('ab').infinite_words()
Infinite words over {'a', 'b'}
```

iterate_by_length(*length*)

Return an iterator over the words of given length.

EXAMPLES:

```
sage: [w.string_rep() for w in Words('ab').iterate_by_length(3)]
['aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']
```

```
>>> from sage.all import *
>>> [w.string_rep() for w in Words('ab').iterate_by_length(Integer(3))]
['aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']
```

shift()

Return the set of infinite words.

EXAMPLES:

```
sage: Words('ab').infinite_words()
Infinite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> Words('ab').infinite_words()
Infinite words over {'a', 'b'}
```

class sage.combinat.words.words.**FiniteWords**(*alphabet=None*, *category=None*)

Bases: *AbstractLanguage*

The set of finite words over a fixed alphabet.

EXAMPLES:

```
sage: W = FiniteWords('ab')
sage: W
Finite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> W = FiniteWords('ab')
>>> W
Finite words over {'a', 'b'}
```

cardinality()

Return the cardinality of this set.

EXAMPLES:

```
sage: FiniteWords('').cardinality()
1
sage: FiniteWords('a').cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> FiniteWords('').cardinality()
1
>>> FiniteWords('a').cardinality()
+Infinity
```

factors()

Return itself.

EXAMPLES:

```
sage: FiniteWords('ab').factors()
Finite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> FiniteWords('ab').factors()
Finite words over {'a', 'b'}
```

is_empty()

Return `False`, because the empty word is in the set.

iter_morphisms(*arg=None*, *codomain=None*, *min_length=1*)

Iterate over all morphisms with domain `self` and the given codomain.

INPUT:

- `arg` – (default: `None`) it can be one of the following:
  - `None` – then the method iterates through all morphisms
  - tuple (a, b) of two integers – it specifies the range `range(a, b)` of values to consider for the sum of the length of the image of each letter in the alphabet
  - list of nonnegative integers – the length of the list must be equal to the size of the alphabet, and the i-th integer of `arg` determines the length of the word mapped to by the i-th letter of the (ordered) alphabet
- `codomain` – (default: `None`) a combinatorial class of words; by default, `codomain` is `self`
- `min_length` – nonnegative integer (default: 1); if `arg` is not specified, then iterate through all the morphisms where the length of the images of each letter in the alphabet is at least `min_length`. This is ignored if `arg` is a list.

OUTPUT: iterator

EXAMPLES:

Iterator over all non-erasing morphisms:

```
sage: W = FiniteWords('ab')
sage: it = W.iter_morphisms()
sage: for _ in range(7): next(it)
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
```

```
>>> from sage.all import *
>>> W = FiniteWords('ab')
>>> it = W.iter_morphisms()
>>> for _ in range(Integer(7)): next(it)
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
```

Iterator over all morphisms including erasing morphisms:

```
sage: W = FiniteWords('ab')
sage: it = W.iter_morphisms(min_length=0)
sage: for _ in range(7): next(it)
WordMorphism: a->, b->
WordMorphism: a->a, b->
WordMorphism: a->b, b->
WordMorphism: a->, b->a
WordMorphism: a->, b->b
WordMorphism: a->aa, b->
WordMorphism: a->ab, b->
```

```
>>> from sage.all import *
>>> W = FiniteWords('ab')
>>> it = W.iter_morphisms(min_length=Integer(0))
>>> for _ in range(Integer(7)): next(it)
WordMorphism: a->, b->
WordMorphism: a->a, b->
WordMorphism: a->b, b->
WordMorphism: a->, b->a
WordMorphism: a->, b->b
WordMorphism: a->aa, b->
WordMorphism: a->ab, b->
```

Iterator over morphisms where the sum of the lengths of the images of the letters is in a specific range:

```
sage: for m in W.iter_morphisms((0, 3), min_length=0): m
WordMorphism: a->aa, b->
WordMorphism: a->ab, b->
WordMorphism: a->ba, b->
WordMorphism: a->bb, b->
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
WordMorphism: a->a, b->
WordMorphism: a->b, b->
WordMorphism: a->, b->aa
WordMorphism: a->, b->ab
WordMorphism: a->, b->ba
WordMorphism: a->, b->bb
WordMorphism: a->, b->a
WordMorphism: a->, b->b
WordMorphism: a->, b->
```

```
>>> from sage.all import *
>>> for m in W.iter_morphisms((Integer(0), Integer(3)), min_
↪length=Integer(0)): m
WordMorphism: a->aa, b->
WordMorphism: a->ab, b->
WordMorphism: a->ba, b->
WordMorphism: a->bb, b->
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
WordMorphism: a->a, b->
WordMorphism: a->b, b->
WordMorphism: a->, b->aa
WordMorphism: a->, b->ab
WordMorphism: a->, b->ba
WordMorphism: a->, b->bb
WordMorphism: a->, b->a
WordMorphism: a->, b->b
WordMorphism: a->, b->
```

```
sage: for m in W.iter_morphisms( (2, 4) ): m
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
WordMorphism: a->ab, b->b
WordMorphism: a->ba, b->a
WordMorphism: a->ba, b->b
WordMorphism: a->bb, b->a
WordMorphism: a->bb, b->b
WordMorphism: a->a, b->aa
WordMorphism: a->a, b->ab
WordMorphism: a->a, b->ba
WordMorphism: a->a, b->bb
WordMorphism: a->b, b->aa
WordMorphism: a->b, b->ab
WordMorphism: a->b, b->ba
WordMorphism: a->b, b->bb
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
```

```
>>> from sage.all import *
>>> for m in W.iter_morphisms( (Integer(2), Integer(4)) ): m
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
WordMorphism: a->ab, b->b
WordMorphism: a->ba, b->a
WordMorphism: a->ba, b->b
WordMorphism: a->bb, b->a
WordMorphism: a->bb, b->b
WordMorphism: a->a, b->aa
WordMorphism: a->a, b->ab
WordMorphism: a->a, b->ba
WordMorphism: a->a, b->bb
WordMorphism: a->b, b->aa
WordMorphism: a->b, b->ab
WordMorphism: a->b, b->ba
WordMorphism: a->b, b->bb
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
```

Iterator over morphisms with specific image lengths:

```
sage: for m in W.iter_morphisms([0, 0]): m
WordMorphism: a->, b->
sage: for m in W.iter_morphisms([0, 1]): m
WordMorphism: a->, b->a
WordMorphism: a->, b->b
sage: for m in W.iter_morphisms([2, 1]): m
```

(continues on next page)

(continued from previous page)

```
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
WordMorphism: a->ab, b->b
WordMorphism: a->ba, b->a
WordMorphism: a->ba, b->b
WordMorphism: a->bb, b->a
WordMorphism: a->bb, b->b
sage: for m in W.iter_morphisms([2, 2]): m
WordMorphism: a->aa, b->aa
WordMorphism: a->aa, b->ab
WordMorphism: a->aa, b->ba
WordMorphism: a->aa, b->bb
WordMorphism: a->ab, b->aa
WordMorphism: a->ab, b->ab
WordMorphism: a->ab, b->ba
WordMorphism: a->ab, b->bb
WordMorphism: a->ba, b->aa
WordMorphism: a->ba, b->ab
WordMorphism: a->ba, b->ba
WordMorphism: a->ba, b->bb
WordMorphism: a->bb, b->aa
WordMorphism: a->bb, b->ab
WordMorphism: a->bb, b->ba
WordMorphism: a->bb, b->bb
```

```
>>> from sage.all import *
>>> for m in W.iter_morphisms([Integer(0), Integer(0)]): m
WordMorphism: a->, b->
>>> for m in W.iter_morphisms([Integer(0), Integer(1)]): m
WordMorphism: a->, b->a
WordMorphism: a->, b->b
>>> for m in W.iter_morphisms([Integer(2), Integer(1)]): m
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
WordMorphism: a->ab, b->b
WordMorphism: a->ba, b->a
WordMorphism: a->ba, b->b
WordMorphism: a->bb, b->a
WordMorphism: a->bb, b->b
>>> for m in W.iter_morphisms([Integer(2), Integer(2)]): m
WordMorphism: a->aa, b->aa
WordMorphism: a->aa, b->ab
WordMorphism: a->aa, b->ba
WordMorphism: a->aa, b->bb
WordMorphism: a->ab, b->aa
WordMorphism: a->ab, b->ab
WordMorphism: a->ab, b->ba
WordMorphism: a->ab, b->bb
WordMorphism: a->ba, b->aa
WordMorphism: a->ba, b->ab
```

(continues on next page)

(continued from previous page)

```
WordMorphism: a->ba, b->ba
WordMorphism: a->ba, b->bb
WordMorphism: a->bb, b->aa
WordMorphism: a->bb, b->ab
WordMorphism: a->bb, b->ba
WordMorphism: a->bb, b->bb
```

The codomain may be specified as well:

```
sage: Y = FiniteWords('xyz')
sage: for m in W.iter_morphisms([0, 2], codomain=Y): m
WordMorphism: a->, b->xx
WordMorphism: a->, b->xy
WordMorphism: a->, b->xz
WordMorphism: a->, b->yx
WordMorphism: a->, b->yy
WordMorphism: a->, b->yz
WordMorphism: a->, b->zx
WordMorphism: a->, b->zy
WordMorphism: a->, b->zz
sage: for m in Y.iter_morphisms([0,2,1], codomain=W): m
WordMorphism: x->, y->aa, z->a
WordMorphism: x->, y->aa, z->b
WordMorphism: x->, y->ab, z->a
WordMorphism: x->, y->ab, z->b
WordMorphism: x->, y->ba, z->a
WordMorphism: x->, y->ba, z->b
WordMorphism: x->, y->bb, z->a
WordMorphism: x->, y->bb, z->b
sage: it = W.iter_morphisms(codomain=Y)
sage: for _ in range(10): next(it)
WordMorphism: a->x, b->x
WordMorphism: a->x, b->y
WordMorphism: a->x, b->z
WordMorphism: a->y, b->x
WordMorphism: a->y, b->y
WordMorphism: a->y, b->z
WordMorphism: a->z, b->x
WordMorphism: a->z, b->y
WordMorphism: a->z, b->z
WordMorphism: a->xx, b->x
```

```
>>> from sage.all import *
>>> Y = FiniteWords('xyz')
>>> for m in W.iter_morphisms([Integer(0), Integer(2)], codomain=Y): m
WordMorphism: a->, b->xx
WordMorphism: a->, b->xy
WordMorphism: a->, b->xz
WordMorphism: a->, b->yx
WordMorphism: a->, b->yy
WordMorphism: a->, b->yz
WordMorphism: a->, b->zx
```

(continues on next page)

(continued from previous page)

```
WordMorphism: a->, b->zy
WordMorphism: a->, b->zz
>>> for m in Y.iter_morphisms([Integer(0),Integer(2),Integer(1)],␣
↪codomain=W): m
WordMorphism: x->, y->aa, z->a
WordMorphism: x->, y->aa, z->b
WordMorphism: x->, y->ab, z->a
WordMorphism: x->, y->ab, z->b
WordMorphism: x->, y->ba, z->a
WordMorphism: x->, y->ba, z->b
WordMorphism: x->, y->bb, z->a
WordMorphism: x->, y->bb, z->b
>>> it = W.iter_morphisms(codomain=Y)
>>> for _ in range(Integer(10)): next(it)
WordMorphism: a->x, b->x
WordMorphism: a->x, b->y
WordMorphism: a->x, b->z
WordMorphism: a->y, b->x
WordMorphism: a->y, b->y
WordMorphism: a->y, b->z
WordMorphism: a->z, b->x
WordMorphism: a->z, b->y
WordMorphism: a->z, b->z
WordMorphism: a->xx, b->x
```

iterate_by_length(*l=1*)

Return an iterator over all the words of `self` of length l.

INPUT:

- `l` – integer (default: 1); the length of the desired words

EXAMPLES:

```
sage: W = FiniteWords('ab')
sage: list(W.iterate_by_length(1))
[word: a, word: b]
sage: list(W.iterate_by_length(2))
[word: aa, word: ab, word: ba, word: bb]
sage: list(W.iterate_by_length(3))
[word: aaa,
 word: aab,
 word: aba,
 word: abb,
 word: baa,
 word: bab,
 word: bba,
 word: bbb]
sage: list(W.iterate_by_length('a'))
Traceback (most recent call last):
...
TypeError: the parameter l (='a') must be an integer
```

```
>>> from sage.all import *
>>> W = FiniteWords('ab')
>>> list(W.iterate_by_length(Integer(1)))
[word: a, word: b]
>>> list(W.iterate_by_length(Integer(2)))
[word: aa, word: ab, word: ba, word: bb]
>>> list(W.iterate_by_length(Integer(3)))
[word: aaa,
 word: aab,
 word: aba,
 word: abb,
 word: baa,
 word: bab,
 word: bba,
 word: bbb]
>>> list(W.iterate_by_length('a'))
Traceback (most recent call last):
...
TypeError: the parameter l (='a') must be an integer
```

random_element(*length=None*, **args*, ***kwds*)

Return a random finite word on the given alphabet.

INPUT:

- `length` – (optional) the length of the word; if not set, will use a uniformly random number between 0 and 10
- all other argument are transmitted to the random generator of the alphabet

EXAMPLES:

```
sage: W = FiniteWords(5)
sage: W.random_element() # random
word: 5114325445423521544531411434451152142155...

sage: W = FiniteWords(ZZ)
sage: W.random_element() # random
word: 5,-1,-1,-1,0,0,0,0,-3,-11
sage: W.random_element(length=4, x=0, y=4) # random
word: 1003
```

```
>>> from sage.all import *
>>> W = FiniteWords(Integer(5))
>>> W.random_element() # random
word: 5114325445423521544531411434451152142155...

>>> W = FiniteWords(ZZ)
>>> W.random_element() # random
word: 5,-1,-1,-1,0,0,0,0,-3,-11
>>> W.random_element(length=Integer(4), x=Integer(0), y=Integer(4)) # random
word: 1003
```

shift()

Return the set of infinite words on the same alphabet.

EXAMPLES:

```
sage: FiniteWords('ab').shift()
Infinite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> FiniteWords('ab').shift()
Infinite words over {'a', 'b'}
```

class sage.combinat.words.words.**InfiniteWords**(*alphabet=None*, *category=None*)

Bases: *AbstractLanguage*

cardinality()

Return the cardinality of this set.

EXAMPLES:

```
sage: InfiniteWords('ab').cardinality()
+Infinity
sage: InfiniteWords('a').cardinality()
1
sage: InfiniteWords('').cardinality()
0
```

```
>>> from sage.all import *
>>> InfiniteWords('ab').cardinality()
+Infinity
>>> InfiniteWords('a').cardinality()
1
>>> InfiniteWords('').cardinality()
0
```

factors()

Return the set of finite words on the same alphabet.

EXAMPLES:

```
sage: InfiniteWords('ab').factors()
Finite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> InfiniteWords('ab').factors()
Finite words over {'a', 'b'}
```

random_element(**args*, ***kwds*)

Return a random infinite word.

EXAMPLES:

```
sage: W = InfiniteWords('ab')
sage: W.random_element()  # random
word: abbbabbaabbbabbabbaabaabbabbbbbbbbaabbbb...

sage: W = InfiniteWords(ZZ)
```

(continues on next page)

(continued from previous page)

```
sage: W.random_element(x=2,y=4) # random
word: 333322332223223333322332322232222233233233...
```

```
>>> from sage.all import *
>>> W = InfiniteWords('ab')
>>> W.random_element() # random
word: abbbabbaabbbabbabbaabaabbabbbbbbbbaabbbb...

>>> W = InfiniteWords(ZZ)
>>> W.random_element(x=Integer(2),y=Integer(4)) # random
word: 333322332223223333322332322232222233233233...
```

shift()

Return itself.

EXAMPLES:

```
sage: InfiniteWords('ab').shift()
Infinite words over {'a', 'b'}
```

```
>>> from sage.all import *
>>> InfiniteWords('ab').shift()
Infinite words over {'a', 'b'}
```

sage.combinat.words.words.**Words**(*alphabet=None*, *length=None*, *finite=True*, *infinite=True*)

Return the combinatorial class of words of length k over an alphabet.

EXAMPLES:

```
sage: Words()
Finite and infinite words over Set of Python objects of class 'object'
sage: Words(length=7)
Words of length 7 over Set of Python objects of class 'object'
sage: Words(5)
Finite and infinite words over {1, 2, 3, 4, 5}
sage: Words(5, 3)
Words of length 3 over {1, 2, 3, 4, 5}
sage: Words(5, infinite=False)
Finite words over {1, 2, 3, 4, 5}
sage: Words(5, finite=False)
Infinite words over {1, 2, 3, 4, 5}
sage: Words('ab')
Finite and infinite words over {'a', 'b'}
sage: Words('ab', 2)
Words of length 2 over {'a', 'b'}
sage: Words('ab', infinite=False)
Finite words over {'a', 'b'}
sage: Words('ab', finite=False)
Infinite words over {'a', 'b'}
sage: Words('positive integers', finite=False)
Infinite words over Positive integers
sage: Words('natural numbers')
Finite and infinite words over Non negative integers
```

```
>>> from sage.all import *
>>> Words()
Finite and infinite words over Set of Python objects of class 'object'
>>> Words(length=Integer(7))
Words of length 7 over Set of Python objects of class 'object'
>>> Words(Integer(5))
Finite and infinite words over {1, 2, 3, 4, 5}
>>> Words(Integer(5), Integer(3))
Words of length 3 over {1, 2, 3, 4, 5}
>>> Words(Integer(5), infinite=False)
Finite words over {1, 2, 3, 4, 5}
>>> Words(Integer(5), finite=False)
Infinite words over {1, 2, 3, 4, 5}
>>> Words('ab')
Finite and infinite words over {'a', 'b'}
>>> Words('ab', Integer(2))
Words of length 2 over {'a', 'b'}
>>> Words('ab', infinite=False)
Finite words over {'a', 'b'}
>>> Words('ab', finite=False)
Infinite words over {'a', 'b'}
>>> Words('positive integers', finite=False)
Infinite words over Positive integers
>>> Words('natural numbers')
Finite and infinite words over Non negative integers
```

class sage.combinat.words.words.**Words_n**(*words*, *n*)

Bases: Parent

The set of words of fixed length on a given alphabet.

alphabet()

Return the underlying alphabet.

EXAMPLES:

```
sage: Words([0,1], 4).alphabet()
{0, 1}
```

```
>>> from sage.all import *
>>> Words([Integer(0),Integer(1)], Integer(4)).alphabet()
{0, 1}
```

cardinality()

Return the number of words of length n from alphabet.

EXAMPLES:

```
sage: Words(['a','b','c'], 4).cardinality()
81
sage: Words(3, 4).cardinality()
81
sage: Words(0,0).cardinality()
1
```

(continues on next page)

(continued from previous page)

```
sage: Words(5,0).cardinality()
1
sage: Words(['a','b','c'],0).cardinality()
1
sage: Words(0,1).cardinality()
0
sage: Words(5,1).cardinality()
5
sage: Words(['a','b','c'],1).cardinality()
3
sage: Words(7,13).cardinality()
96889010407
sage: Words(['a','b','c','d','e','f','g'],13).cardinality()
96889010407
```

```
>>> from sage.all import *
>>> Words(['a','b','c'], Integer(4)).cardinality()
81
>>> Words(Integer(3), Integer(4)).cardinality()
81
>>> Words(Integer(0),Integer(0)).cardinality()
1
>>> Words(Integer(5),Integer(0)).cardinality()
1
>>> Words(['a','b','c'],Integer(0)).cardinality()
1
>>> Words(Integer(0),Integer(1)).cardinality()
0
>>> Words(Integer(5),Integer(1)).cardinality()
5
>>> Words(['a','b','c'],Integer(1)).cardinality()
3
>>> Words(Integer(7),Integer(13)).cardinality()
96889010407
>>> Words(['a','b','c','d','e','f','g'],Integer(13)).cardinality()
96889010407
```

iterate_by_length(*length*)

All words in this class are of the same length, so use iterator instead.

list()

Return a list of all the words contained in `self`.

EXAMPLES:

```
sage: Words(0,0).list()
[word: ]
sage: Words(5,0).list()
[word: ]
sage: Words(['a','b','c'],0).list()
[word: ]
sage: Words(5,1).list()
```

(continues on next page)

(continued from previous page)

```
[word: 1, word: 2, word: 3, word: 4, word: 5]
sage: Words(['a','b','c'],2).list()
[word: aa, word: ab, word: ac, word: ba, word: bb, word: bc, word: ca, word:
↪cb, word: cc]
```

```
>>> from sage.all import *
>>> Words(Integer(0),Integer(0)).list()
[word: ]
>>> Words(Integer(5),Integer(0)).list()
[word: ]
>>> Words(['a','b','c'],Integer(0)).list()
[word: ]
>>> Words(Integer(5),Integer(1)).list()
[word: 1, word: 2, word: 3, word: 4, word: 5]
>>> Words(['a','b','c'],Integer(2)).list()
[word: aa, word: ab, word: ac, word: ba, word: bb, word: bc, word: ca, word:
↪cb, word: cc]
```

random_element(**args*, ***kwds*)

Return a random word in this set.

EXAMPLES:

```
sage: W = Words('ab', 4)
sage: W.random_element()   # random
word: bbab
sage: W.random_element() in W
True

sage: W = Words(ZZ, 5)
sage: W.random_element()   # random
word: 1,2,2,-1,12
sage: W.random_element() in W
True
```

```
>>> from sage.all import *
>>> W = Words('ab', Integer(4))
>>> W.random_element()   # random
word: bbab
>>> W.random_element() in W
True

>>> W = Words(ZZ, Integer(5))
>>> W.random_element()   # random
word: 1,2,2,-1,12
>>> W.random_element() in W
True
```

5.1.372 Yang-Baxter Graphs

class sage.combinat.yang_baxter_graph.**SwapIncreasingOperator**(*i*)

Bases: *SwapOperator*

class sage.combinat.yang_baxter_graph.**SwapOperator**(*i*)

Bases: SageObject

The operator that swaps the items in positions i and i+1.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapOperator
sage: s3 = SwapOperator(3)
sage: s3 == loads(dumps(s3))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapOperator
>>> s3 = SwapOperator(Integer(3))
>>> s3 == loads(dumps(s3))
True
```

position()

Return i where self is the operator that swaps positions i and i+1.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapOperator
sage: s3 = SwapOperator(3)
sage: s3.position()
3
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapOperator
>>> s3 = SwapOperator(Integer(3))
>>> s3.position()
3
```

sage.combinat.yang_baxter_graph.**YangBaxterGraph**(*partition=None*, *root=None*, *operators=None*)

Construct the Yang-Baxter graph from root by repeated application of operators, or the Yang-Baxter graph associated to partition.

INPUT:

The user needs to provide either partition or both root and operators, where

- partition – a partition of a positive integer
- root – the root vertex
- operator – a function that maps vertices u to a list of tuples of the form (v, l) where v is a successor of u and l is the label of the edge from u to v.

OUTPUT: either:

- *YangBaxterGraph_partition* – if partition is defined
- *YangBaxterGraph_generic* – if partition is None

EXAMPLES:

The Yang-Baxter graph defined by a partition $[p_1, \ldots, p_k]$ is the labelled directed graph with vertex set obtained by bubble-sorting $(p_k - 1, p_k - 2, \ldots, 0, \ldots, p_1 - 1, p_1 - 2, \ldots, 0)$; there is an arrow from u to v labelled by i if v is obtained by swapping the i-th and $(i + 1)$-th elements of u. For example, if the partition is $[3, 1]$, then we begin with $(0, 2, 1, 0)$ and generate all tuples obtained from it by swapping two adjacent entries if they are increasing:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: bubbleswaps = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=bubbleswaps); Y
Yang-Baxter graph with root vertex (0, 2, 1, 0)
sage: Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> bubbleswaps = [SwapIncreasingOperator(i) for i in range(Integer(3))]
>>> Y = YangBaxterGraph(root=(Integer(0),Integer(2),Integer(1),Integer(0)),␣
↪operators=bubbleswaps); Y
Yang-Baxter graph with root vertex (0, 2, 1, 0)
>>> Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
```

The `partition` keyword is a shorthand for the above construction:

```
sage: Y = YangBaxterGraph(partition=[3,1]); Y                                #␣
↪needs sage.combinat
Yang-Baxter graph of [3, 1], with top vertex (0, 2, 1, 0)
sage: Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
```

```
>>> from sage.all import *
>>> Y = YangBaxterGraph(partition=[Integer(3),Integer(1)]); Y                 ␣
↪               # needs sage.combinat
Yang-Baxter graph of [3, 1], with top vertex (0, 2, 1, 0)
>>> Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
```

The permutahedron can be realized as a Yang-Baxter graph:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: swappers = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(1,2,3,4), operators=swappers); Y
Yang-Baxter graph with root vertex (1, 2, 3, 4)
sage: Y.plot()                                                               #␣
↪needs sage.plot
Graphics object consisting of 97 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> swappers = [SwapIncreasingOperator(i) for i in range(Integer(3))]
>>> Y = YangBaxterGraph(root=(Integer(1),Integer(2),Integer(3),Integer(4)),␣
↪operators=swappers); Y
```

(continues on next page)

(continued from previous page)

```
Yang-Baxter graph with root vertex (1, 2, 3, 4)
>>> Y.plot()                                                                   #␣
↪needs sage.plot
Graphics object consisting of 97 graphics primitives
```

The Cayley graph of a finite group can be realized as a Yang-Baxter graph:

```
sage: # needs sage.groups
sage: def left_multiplication_by(g):
....:     return lambda h: h*g
sage: G = CyclicPermutationGroup(4)
sage: operators = [ left_multiplication_by(gen) for gen in G.gens() ]
sage: Y = YangBaxterGraph(root=G.identity(), operators=operators); Y
Yang-Baxter graph with root vertex ()
sage: Y.plot(edge_labels=False)                                                #␣
↪needs sage.plot
Graphics object consisting of 9 graphics primitives

sage: # needs sage.groups
sage: G = SymmetricGroup(4)
sage: operators = [left_multiplication_by(gen) for gen in G.gens()]
sage: Y = YangBaxterGraph(root=G.identity(), operators=operators); Y
Yang-Baxter graph with root vertex ()
sage: Y.plot(edge_labels=False)                                                #␣
↪needs sage.plot
Graphics object consisting of 96 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> def left_multiplication_by(g):
...     return lambda h: h*g
>>> G = CyclicPermutationGroup(Integer(4))
>>> operators = [ left_multiplication_by(gen) for gen in G.gens() ]
>>> Y = YangBaxterGraph(root=G.identity(), operators=operators); Y
Yang-Baxter graph with root vertex ()
>>> Y.plot(edge_labels=False)                                                  #␣
↪needs sage.plot
Graphics object consisting of 9 graphics primitives

>>> # needs sage.groups
>>> G = SymmetricGroup(Integer(4))
>>> operators = [left_multiplication_by(gen) for gen in G.gens()]
>>> Y = YangBaxterGraph(root=G.identity(), operators=operators); Y
Yang-Baxter graph with root vertex ()
>>> Y.plot(edge_labels=False)                                                  #␣
↪needs sage.plot
Graphics object consisting of 96 graphics primitives
```

AUTHORS:

- Franco Saliola (2009-04-23)

class sage.combinat.yang_baxter_graph.**YangBaxterGraph_generic**(*root*, *operators*)

Bases: SageObject

A class to model the Yang-Baxter graph defined by `root` and `operators`.

INPUT:

- `root` – the root vertex of the graph
- `operators` – list of callables that map vertices to (new) vertices

> **Note**
>
> This is a lazy implementation: the digraph is only computed when it is needed.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(4)]
sage: Y = YangBaxterGraph(root=(1,0,2,1,0), operators=ops); Y
Yang-Baxter graph with root vertex (1, 0, 2, 1, 0)
sage: loads(dumps(Y)) == Y
True
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(4))]
>>> Y = YangBaxterGraph(root=(Integer(1),Integer(0),Integer(2),Integer(1),
↪Integer(0)), operators=ops); Y
Yang-Baxter graph with root vertex (1, 0, 2, 1, 0)
>>> loads(dumps(Y)) == Y
True
```

AUTHORS:

- Franco Saliola (2009-04-23)

edges()

Return the (labelled) edges of `self`.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1,␣
↪0), (2, 1, 0, 0), Swap-if-increasing at position 1)]
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(3))]
>>> Y = YangBaxterGraph(root=(Integer(0),Integer(2),Integer(1),Integer(0)),␣
↪operators=ops)
>>> Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1,␣
↪0), (2, 1, 0, 0), Swap-if-increasing at position 1)]
```

plot(**args*, ***kwds*)

Plot `self` as a digraph.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(4)]
sage: Y = YangBaxterGraph(root=(1,0,2,1,0), operators=ops)
sage: Y.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 16 graphics primitives
sage: Y.plot(edge_labels=False)                                         #␣
↪needs sage.plot
Graphics object consisting of 11 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(4))]
>>> Y = YangBaxterGraph(root=(Integer(1),Integer(0),Integer(2),Integer(1),
↪Integer(0)), operators=ops)
>>> Y.plot()                                                            #␣
↪needs sage.plot
Graphics object consisting of 16 graphics primitives
>>> Y.plot(edge_labels=False)                                           #␣
↪needs sage.plot
Graphics object consisting of 11 graphics primitives
```

relabel_edges(*edge_dict*, *inplace=True*)

Relabel the edges of `self`.

INPUT:

- `edge_dict` – dictionary keyed by the (unlabelled) edges

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: def relabel_op(op, u):
....:     i = op.position()
....:     return u[:i] + u[i:i+2][::-1] + u[i+2:]
sage: Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1,␣
↪0), (2, 1, 0, 0), Swap-if-increasing at position 1)]
sage: d = {((0,2,1,0),(2,0,1,0)):17, ((2,0,1,0),(2,1,0,0)):27}
sage: Y.relabel_edges(d, inplace=False).edges()
[((0, 2, 1, 0), (2, 0, 1, 0), 17), ((2, 0, 1, 0), (2, 1, 0, 0), 27)]
sage: Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1,␣
↪0), (2, 1, 0, 0), Swap-if-increasing at position 1)]
sage: Y.relabel_edges(d, inplace=True)
sage: Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), 17), ((2, 0, 1, 0), (2, 1, 0, 0), 27)]
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(3))]
>>> Y = YangBaxterGraph(root=(Integer(0),Integer(2),Integer(1),Integer(0)),␣
↪operators=ops)
>>> def relabel_op(op, u):
...     i = op.position()
...     return u[:i] + u[i:i+Integer(2)][::-Integer(1)] + u[i+Integer(2):]
>>> Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1,␣
↪0), (2, 1, 0, 0), Swap-if-increasing at position 1)]
>>> d = {((Integer(0),Integer(2),Integer(1),Integer(0)),(Integer(2),
↪Integer(0),Integer(1),Integer(0))):Integer(17), ((Integer(2),Integer(0),
↪Integer(1),Integer(0)),(Integer(2),Integer(1),Integer(0),
↪Integer(0))):Integer(27)}
>>> Y.relabel_edges(d, inplace=False).edges()
[((0, 2, 1, 0), (2, 0, 1, 0), 17), ((2, 0, 1, 0), (2, 1, 0, 0), 27)]
>>> Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1,␣
↪0), (2, 1, 0, 0), Swap-if-increasing at position 1)]
>>> Y.relabel_edges(d, inplace=True)
>>> Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), 17), ((2, 0, 1, 0), (2, 1, 0, 0), 27)]
```

relabel_vertices(*v*, *relabel_operator*, *inplace=True*)

Relabel the vertices `u` of `self` by the object obtained from `u` by applying the `relabel_operator` to `v` along a path from `self.root()` to `u`.

Note that the `self.root()` is paired with `v`.

INPUT:

- `v` – tuple, Permutation, etc.
- `inplace` – if `True`, modifies `self`; otherwise returns a modified copy of `self`

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: def relabel_op(op, u):
....:     i = op.position()
....:     return u[:i] + u[i:i+2][::-1] + u[i+2:]
sage: d = Y.relabel_vertices((1,2,3,4), relabel_op, inplace=False); d
Yang-Baxter graph with root vertex (1, 2, 3, 4)
sage: Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
sage: e = Y.relabel_vertices((1,2,3,4), relabel_op); e
sage: Y.vertices(sort=True)
[(1, 2, 3, 4), (2, 1, 3, 4), (2, 3, 1, 4)]
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(3))]
```

(continues on next page)

(continued from previous page)

```
>>> Y = YangBaxterGraph(root=(Integer(0),Integer(2),Integer(1),Integer(0)),␣
↪operators=ops)
>>> def relabel_op(op, u):
...     i = op.position()
...     return u[:i] + u[i:i+Integer(2)][::-Integer(1)] + u[i+Integer(2):]
>>> d = Y.relabel_vertices((Integer(1),Integer(2),Integer(3),Integer(4)),␣
↪relabel_op, inplace=False); d
Yang-Baxter graph with root vertex (1, 2, 3, 4)
>>> Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
>>> e = Y.relabel_vertices((Integer(1),Integer(2),Integer(3),Integer(4)),␣
↪relabel_op); e
>>> Y.vertices(sort=True)
[(1, 2, 3, 4), (2, 1, 3, 4), (2, 3, 1, 4)]
```

root()

Return the root vertex of `self`.

If `self` is the Yang-Baxter graph of the partition $[p_1, p_2, \dots, p_k]$, then this is the vertex $(p_k - 1, p_k - 2, \dots, 0, \dots, p_1 - 1, p_1 - 2, \dots, 0)$.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(4)]
sage: Y = YangBaxterGraph(root=(1,0,2,1,0), operators=ops)
sage: Y.root()
(1, 0, 2, 1, 0)
sage: Y = YangBaxterGraph(root=(0,1,0,2,1,0), operators=ops)
sage: Y.root()
(0, 1, 0, 2, 1, 0)
sage: Y = YangBaxterGraph(root=(1,0,3,2,1,0), operators=ops)
sage: Y.root()
(1, 0, 3, 2, 1, 0)
sage: Y = YangBaxterGraph(partition=[3,2])                              #␣
↪needs sage.combinat
sage: Y.root()                                                          #␣
↪needs sage.combinat
(1, 0, 2, 1, 0)
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(4))]
>>> Y = YangBaxterGraph(root=(Integer(1),Integer(0),Integer(2),Integer(1),
↪Integer(0)), operators=ops)
>>> Y.root()
(1, 0, 2, 1, 0)
>>> Y = YangBaxterGraph(root=(Integer(0),Integer(1),Integer(0),Integer(2),
↪Integer(1),Integer(0)), operators=ops)
>>> Y.root()
(0, 1, 0, 2, 1, 0)
>>> Y = YangBaxterGraph(root=(Integer(1),Integer(0),Integer(3),Integer(2),
↪Integer(1),Integer(0)), operators=ops)
```

(continues on next page)

(continued from previous page)

```
>>> Y.root()
(1, 0, 3, 2, 1, 0)
>>> Y = YangBaxterGraph(partition=[Integer(3),Integer(2)])
↪              # needs sage.combinat
>>> Y.root()                                                    #␣
↪needs sage.combinat
(1, 0, 2, 1, 0)
```

successors(*v*)

Return the successors of the vertex v.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(4)]
sage: Y = YangBaxterGraph(root=(1,0,2,1,0), operators=ops)
sage: Y.successors(Y.root())
[(1, 2, 0, 1, 0)]
sage: sorted(Y.successors((1, 2, 0, 1, 0)))
[(1, 2, 1, 0, 0), (2, 1, 0, 1, 0)]
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(4))]
>>> Y = YangBaxterGraph(root=(Integer(1),Integer(0),Integer(2),Integer(1),
↪Integer(0)), operators=ops)
>>> Y.successors(Y.root())
[(1, 2, 0, 1, 0)]
>>> sorted(Y.successors((Integer(1), Integer(2), Integer(0), Integer(1),␣
↪Integer(0))))
[(1, 2, 1, 0, 0), (2, 1, 0, 1, 0)]
```

vertex_relabelling_dict(*v*, *relabel_operator*)

Return a dictionary pairing vertices u of self with the object obtained from v by applying the relabel_operator along a path from the root to u.

Note that the root is paired with v.

INPUT:

- v – an object
- relabel_operator – function mapping a vertex and a label to the image of the vertex

OUTPUT: dictionary pairing vertices with the corresponding image of v

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: def relabel_operator(op, u):
....:     i = op.position()
....:     return u[:i] + u[i:i+2][::-1] + u[i+2:]
sage: Y.vertex_relabelling_dict((1,2,3,4), relabel_operator)
```

(continues on next page)

(continued from previous page)

```
{(0, 2, 1, 0): (1, 2, 3, 4),
 (2, 0, 1, 0): (2, 1, 3, 4),
 (2, 1, 0, 0): (2, 3, 1, 4)}
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(3))]
>>> Y = YangBaxterGraph(root=(Integer(0),Integer(2),Integer(1),Integer(0)),␣
↪operators=ops)
>>> def relabel_operator(op, u):
...     i = op.position()
...     return u[:i] + u[i:i+Integer(2)][::-Integer(1)] + u[i+Integer(2):]
>>> Y.vertex_relabelling_dict((Integer(1),Integer(2),Integer(3),Integer(4)),␣
↪relabel_operator)
{(0, 2, 1, 0): (1, 2, 3, 4),
 (2, 0, 1, 0): (2, 1, 3, 4),
 (2, 1, 0, 0): (2, 3, 1, 4)}
```

vertices(*sort=False*)

Return the vertices of `self`.

INPUT:

- `sort` – boolean (default: `False`); whether to sort the vertices

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
```

```
>>> from sage.all import *
>>> from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
>>> ops = [SwapIncreasingOperator(i) for i in range(Integer(3))]
>>> Y = YangBaxterGraph(root=(Integer(0),Integer(2),Integer(1),Integer(0)),␣
↪operators=ops)
>>> Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
```

class sage.combinat.yang_baxter_graph.**YangBaxterGraph_partition**(*partition*)

Bases: *YangBaxterGraph_generic*

A class to model the Yang-Baxter graph of a partition.

The Yang-Baxter graph defined by a partition $[p_1, \ldots, p_k]$ is the labelled directed graph with vertex set obtained by bubble-sorting $(p_k - 1, p_k - 2, \ldots, 0, \ldots, p_1 - 1, p_1 - 2, \ldots, 0)$; there is an arrow from u to v labelled by i if v is obtained by swapping the i-th and $(i+1)$-th elements of u.

Note

This is a lazy implementation: the digraph is only computed when it is needed.

EXAMPLES:

```
sage: Y = YangBaxterGraph(partition=[3,2,1]); Y                          #␣
↪needs sage.combinat
Yang-Baxter graph of [3, 2, 1], with top vertex (0, 1, 0, 2, 1, 0)
sage: loads(dumps(Y)) == Y                                                #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> Y = YangBaxterGraph(partition=[Integer(3),Integer(2),Integer(1)]); Y        ␣
↪                    # needs sage.combinat
Yang-Baxter graph of [3, 2, 1], with top vertex (0, 1, 0, 2, 1, 0)
>>> loads(dumps(Y)) == Y                                                # needs␣
↪sage.combinat
True
```

AUTHORS:

- Franco Saliola (2009-04-23)

relabel_vertices(*v*, *inplace=True*)

Relabel the vertices of `self` with the object obtained from `v` by applying the transpositions corresponding to the edge labels along some path from the root to the vertex.

INPUT:

- `v` – tuple, Permutation, etc.
- `inplace` – if `True`, modifies `self`; otherwise returns a modified copy of `self`

EXAMPLES:

```
sage: # needs sage.combinat
sage: Y = YangBaxterGraph(partition=[3,1]); Y
Yang-Baxter graph of [3, 1], with top vertex (0, 2, 1, 0)
sage: d = Y.relabel_vertices((1,2,3,4), inplace=False); d
Digraph on 3 vertices
sage: Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
sage: e = Y.relabel_vertices((1,2,3,4)); e
sage: Y.vertices(sort=True)
[(1, 2, 3, 4), (2, 1, 3, 4), (2, 3, 1, 4)]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> Y = YangBaxterGraph(partition=[Integer(3),Integer(1)]); Y
Yang-Baxter graph of [3, 1], with top vertex (0, 2, 1, 0)
>>> d = Y.relabel_vertices((Integer(1),Integer(2),Integer(3),Integer(4)),␣
↪inplace=False); d
Digraph on 3 vertices
>>> Y.vertices(sort=True)
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
>>> e = Y.relabel_vertices((Integer(1),Integer(2),Integer(3),Integer(4))); e
>>> Y.vertices(sort=True)
[(1, 2, 3, 4), (2, 1, 3, 4), (2, 3, 1, 4)]
```

vertex_relabelling_dict(*v*)

Return a dictionary pairing vertices `u` of `self` with the object obtained from `v` by applying transpositions corresponding to the edges labels along a path from the root to `u`.

Note that the root is paired with `v`.

INPUT:

- `v` – an object

OUTPUT: dictionary pairing vertices with the corresponding image of `v`

EXAMPLES:

```
sage: Y = YangBaxterGraph(partition=[3,1])                                #␣
↪needs sage.combinat
sage: Y.vertex_relabelling_dict((1,2,3,4))                                #␣
↪needs sage.combinat
{(0, 2, 1, 0): (1, 2, 3, 4),
 (2, 0, 1, 0): (2, 1, 3, 4),
 (2, 1, 0, 0): (2, 3, 1, 4)}
sage: Y.vertex_relabelling_dict((4,3,2,1))                                #␣
↪needs sage.combinat
{(0, 2, 1, 0): (4, 3, 2, 1),
 (2, 0, 1, 0): (3, 4, 2, 1),
 (2, 1, 0, 0): (3, 2, 4, 1)}
```

```
>>> from sage.all import *
>>> Y = YangBaxterGraph(partition=[Integer(3),Integer(1)])                 ␣
↪                # needs sage.combinat
>>> Y.vertex_relabelling_dict((Integer(1),Integer(2),Integer(3),Integer(4))) ␣
↪                               # needs sage.combinat
{(0, 2, 1, 0): (1, 2, 3, 4),
 (2, 0, 1, 0): (2, 1, 3, 4),
 (2, 1, 0, 0): (2, 3, 1, 4)}
>>> Y.vertex_relabelling_dict((Integer(4),Integer(3),Integer(2),Integer(1))) ␣
↪                               # needs sage.combinat
{(0, 2, 1, 0): (4, 3, 2, 1),
 (2, 0, 1, 0): (3, 4, 2, 1),
 (2, 1, 0, 0): (3, 2, 4, 1)}
```

5.1.373 C-Finite Sequences

C-finite infinite sequences satisfy homogeneous linear recurrences with constant coefficients:

$$a_{n+d} = c_0 a_n + c_1 a_{n+1} + \cdots + c_{d-1} a_{n+d-1}, \quad d > 0.$$

CFiniteSequences are completely defined by their ordinary generating function (o.g.f., which is always a `fraction` of `polynomials` over $\mathbf{Z}$ or $\mathbf{Q}$).

EXAMPLES:

```
sage: fibo = CFiniteSequence(x/(1-x-x^2))        # the Fibonacci sequence
sage: fibo
C-finite sequence, generated by -x/(x^2 + x - 1)
sage: fibo.parent()
```

(continues on next page)

(continued from previous page)

```
The ring of C-Finite sequences in x over Rational Field
sage: fibo.parent().category()
Category of commutative rings
sage: C.<x> = CFiniteSequences(QQ)
sage: fibo.parent() == C
True
sage: C
The ring of C-Finite sequences in x over Rational Field
sage: C(x/(1-x-x^2))
C-finite sequence, generated by -x/(x^2 + x - 1)
sage: C(x/(1-x-x^2)) == fibo
True
sage: var('y')
y
sage: CFiniteSequence(y/(1-y-y^2))
C-finite sequence, generated by -y/(y^2 + y - 1)
sage: CFiniteSequence(y/(1-y-y^2)) == fibo
False
```

```
>>> from sage.all import *
>>> fibo = CFiniteSequence(x/(Integer(1)-x-x**Integer(2)))        # the Fibonacci␣
↪sequence
>>> fibo
C-finite sequence, generated by -x/(x^2 + x - 1)
>>> fibo.parent()
The ring of C-Finite sequences in x over Rational Field
>>> fibo.parent().category()
Category of commutative rings
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> fibo.parent() == C
True
>>> C
The ring of C-Finite sequences in x over Rational Field
>>> C(x/(Integer(1)-x-x**Integer(2)))
C-finite sequence, generated by -x/(x^2 + x - 1)
>>> C(x/(Integer(1)-x-x**Integer(2))) == fibo
True
>>> var('y')
y
>>> CFiniteSequence(y/(Integer(1)-y-y**Integer(2)))
C-finite sequence, generated by -y/(y^2 + y - 1)
>>> CFiniteSequence(y/(Integer(1)-y-y**Integer(2))) == fibo
False
```

Finite subsets of the sequence are accessible via python slices:

```
sage: fibo[137]                 #the 137th term of the Fibonacci sequence
19134702400093278081449423917
sage: fibo[137] == fibonacci(137)
True
sage: fibo[0:12]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

(continues on next page)

(continued from previous page)

```
sage: fibo[14:4:-2]
[377, 144, 55, 21, 8]
```

```
>>> from sage.all import *
>>> fibo[Integer(137)]                    #the 137th term of the Fibonacci sequence
19134702400093278081449423917
>>> fibo[Integer(137)] == fibonacci(Integer(137))
True
>>> fibo[Integer(0):Integer(12)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo[Integer(14):Integer(4):-Integer(2)]
[377, 144, 55, 21, 8]
```

They can be created also from the coefficients and start values of a recurrence:

```
sage: r = C.from_recurrence([1,1],[0,1])
sage: r == fibo
True
```

```
>>> from sage.all import *
>>> r = C.from_recurrence([Integer(1),Integer(1)],[Integer(0),Integer(1)])
>>> r == fibo
True
```

Given enough values, the o.g.f. of a C-finite sequence can be guessed:

```
sage: r = C.guess([0,1,1,2,3,5,8])
sage: r == fibo
True
```

```
>>> from sage.all import *
>>> r = C.guess([Integer(0),Integer(1),Integer(1),Integer(2),Integer(3),Integer(5),
↪Integer(8)])
>>> r == fibo
True
```

See also

fibonacci(), *BinaryRecurrenceSequence*

AUTHORS:

- Ralf Stephan (2014): initial version

REFERENCES:

- [GK1982]
- [KP2011]
- [SZ1994]
- [Zei2011]

class sage.rings.cfinite_sequence.**CFiniteSequence**(*parent*, *ogf*)

Bases: FieldElement

Create a C-finite sequence given its ordinary generating function.

INPUT:

- ogf – a rational function, the ordinary generating function (can be an element from the symbolic ring, fraction field or polynomial ring)

OUTPUT: a CFiniteSequence object

EXAMPLES:

```
sage: CFiniteSequence((2-x)/(1-x-x^2))      # the Lucas sequence
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
sage: CFiniteSequence(x/(1-x)^3)            # triangular numbers
C-finite sequence, generated by -x/(x^3 - 3*x^2 + 3*x - 1)
```

```
>>> from sage.all import *
>>> CFiniteSequence((Integer(2)-x)/(Integer(1)-x-x**Integer(2)))      # the Lucas␣
↪sequence
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
>>> CFiniteSequence(x/(Integer(1)-x)**Integer(3))            # triangular numbers
C-finite sequence, generated by -x/(x^3 - 3*x^2 + 3*x - 1)
```

Polynomials are interpreted as finite sequences, or recurrences of degree 0:

```
sage: CFiniteSequence(x^2-4*x^5)
Finite sequence [1, 0, 0, -4], offset = 2
sage: CFiniteSequence(1)
Finite sequence [1], offset = 0
```

```
>>> from sage.all import *
>>> CFiniteSequence(x**Integer(2)-Integer(4)*x**Integer(5))
Finite sequence [1, 0, 0, -4], offset = 2
>>> CFiniteSequence(Integer(1))
Finite sequence [1], offset = 0
```

This implementation allows any polynomial fraction as o.g.f. by interpreting any power of x dividing the o.g.f. numerator or denominator as a right or left shift of the sequence offset:

```
sage: CFiniteSequence(x^2+3/x)
Finite sequence [3, 0, 0, 1], offset = -1
sage: CFiniteSequence(1/x+4/x^3)
Finite sequence [4, 0, 1], offset = -3
sage: P = LaurentPolynomialRing(QQ.fraction_field(), 'X')
sage: X=P.gen()
sage: CFiniteSequence(1/(1-X))
C-finite sequence, generated by -1/(X - 1)
```

```
>>> from sage.all import *
>>> CFiniteSequence(x**Integer(2)+Integer(3)/x)
Finite sequence [3, 0, 0, 1], offset = -1
>>> CFiniteSequence(Integer(1)/x+Integer(4)/x**Integer(3))
```

(continues on next page)

(continued from previous page)

```
Finite sequence [4, 0, 1], offset = -3
>>> P = LaurentPolynomialRing(QQ.fraction_field(), 'X')
>>> X=P.gen()
>>> CFiniteSequence(Integer(1)/(Integer(1)-X))
C-finite sequence, generated by -1/(X - 1)
```

The o.g.f. is always normalized to get a denominator constant coefficient of $+1$:

```
sage: CFiniteSequence(1/(x-2))
C-finite sequence, generated by 1/(x - 2)
```

```
>>> from sage.all import *
>>> CFiniteSequence(Integer(1)/(x-Integer(2)))
C-finite sequence, generated by 1/(x - 2)
```

The given `ogf` is used to create an appropriate parent: it can be a symbolic expression, a polynomial , or a fraction field element as long as it can be coerced into a proper fraction field over the rationals:

```
sage: var('x')
x
sage: f1 = CFiniteSequence((2-x)/(1-x-x^2))
sage: P.<x> = QQ[]
sage: f2 = CFiniteSequence((2-x)/(1-x-x^2))
sage: f1 == f2
True
sage: f1.parent()
The ring of C-Finite sequences in x over Rational Field
sage: f1.ogf().parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: CFiniteSequence(log(x))
Traceback (most recent call last):
...
TypeError: unable to convert log(x) to a rational
```

```
>>> from sage.all import *
>>> var('x')
x
>>> f1 = CFiniteSequence((Integer(2)-x)/(Integer(1)-x-x**Integer(2)))
>>> P = QQ['x']; (x,) = P._first_ngens(1)
>>> f2 = CFiniteSequence((Integer(2)-x)/(Integer(1)-x-x**Integer(2)))
>>> f1 == f2
True
>>> f1.parent()
The ring of C-Finite sequences in x over Rational Field
>>> f1.ogf().parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
>>> CFiniteSequence(log(x))
Traceback (most recent call last):
...
TypeError: unable to convert log(x) to a rational
```

closed_form(*n='n'*)

Return a symbolic expression in `n`, which equals the n-th term of the sequence.

It is a well-known property of C-finite sequences `a_n` that they have a closed form of the type:

$$a_n = \sum_{i=1}^{d} c_i(n) \cdot r_i^n,$$

where `r_i` are the roots of the characteristic equation and `c_i(n)` is a polynomial (whose degree equals the multiplicity of `r_i` minus one). This is a natural generalization of Binet's formula for Fibonacci numbers. See, for instance, [KP2011, Theorem 4.1].

Note that if the o.g.f. has a polynomial part, that is, if the numerator degree is not strictly less than the denominator degree, then this closed form holds only when `n` exceeds the degree of that polynomial part. In that case, the returned expression will differ from the sequence for small `n`.

EXAMPLES:

```
sage: CFiniteSequence(1/(1-x)).closed_form()
1
sage: CFiniteSequence(x^2/(1-x)).closed_form()
1
sage: CFiniteSequence(1/(1-x^2)).closed_form()
1/2*(-1)^n + 1/2
sage: CFiniteSequence(1/(1+x^3)).closed_form()
1/3*(-1)^n + 1/3*(1/2*I*sqrt(3) + 1/2)^n + 1/3*(-1/2*I*sqrt(3) + 1/2)^n
sage: CFiniteSequence(1/(1-x)/(1-2*x)/(1-3*x)).closed_form()
9/2*3^n - 4*2^n + 1/2
```

```
>>> from sage.all import *
>>> CFiniteSequence(Integer(1)/(Integer(1)-x)).closed_form()
1
>>> CFiniteSequence(x**Integer(2)/(Integer(1)-x)).closed_form()
1
>>> CFiniteSequence(Integer(1)/(Integer(1)-x**Integer(2))).closed_form()
1/2*(-1)^n + 1/2
>>> CFiniteSequence(Integer(1)/(Integer(1)+x**Integer(3))).closed_form()
1/3*(-1)^n + 1/3*(1/2*I*sqrt(3) + 1/2)^n + 1/3*(-1/2*I*sqrt(3) + 1/2)^n
>>> CFiniteSequence(Integer(1)/(Integer(1)-x)/(Integer(1)-Integer(2)*x)/
↪(Integer(1)-Integer(3)*x)).closed_form()
9/2*3^n - 4*2^n + 1/2
```

Binet's formula for the Fibonacci numbers:

```
sage: CFiniteSequence(x/(1-x-x^2)).closed_form()
sqrt(1/5)*(1/2*sqrt(5) + 1/2)^n - sqrt(1/5)*(-1/2*sqrt(5) + 1/2)^n
sage: [_.subs(n=k).full_simplify() for k in range(6)]
[0, 1, 1, 2, 3, 5]

sage: CFiniteSequence((4*x+3)/(1-2*x-5*x^2)).closed_form()
1/2*(sqrt(6) + 1)^n*(7*sqrt(1/6) + 3) - 1/2*(-sqrt(6) + 1)^n*(7*sqrt(1/6) - 3)
```

```
>>> from sage.all import *
>>> CFiniteSequence(x/(Integer(1)-x-x**Integer(2))).closed_form()
sqrt(1/5)*(1/2*sqrt(5) + 1/2)^n - sqrt(1/5)*(-1/2*sqrt(5) + 1/2)^n
>>> [_.subs(n=k).full_simplify() for k in range(Integer(6))]
[0, 1, 1, 2, 3, 5]
```

(continues on next page)

(continued from previous page)

```
>>> CFiniteSequence((Integer(4)*x+Integer(3))/(Integer(1)-Integer(2)*x-
↪Integer(5)*x**Integer(2))).closed_form()
1/2*(sqrt(6) + 1)^n*(7*sqrt(1/6) + 3) - 1/2*(-sqrt(6) + 1)^n*(7*sqrt(1/6) - 3)
```

Examples with multiple roots:

```
sage: CFiniteSequence(x*(x^2+4*x+1)/(1-x)^5).closed_form()
1/4*n^4 + 1/2*n^3 + 1/4*n^2
sage: CFiniteSequence((1+2*x-x^2)/(1-x)^4/(1+x)^2).closed_form()
1/12*n^3 - 1/8*(-1)^n*(n + 1) + 3/4*n^2 + 43/24*n + 9/8
sage: CFiniteSequence(1/(1-x)^3/(1-2*x)^4).closed_form()
4/3*(n^3 - 3*n^2 + 20*n - 36)*2^n + 1/2*n^2 + 19/2*n + 49
sage: CFiniteSequence((x/(1-x-x^2))^2).closed_form()
1/5*(n - sqrt(1/5))*(1/2*sqrt(5) + 1/2)^n + 1/5*(n + sqrt(1/5))*(-1/2*sqrt(5)␣
↪+ 1/2)^n
```

```
>>> from sage.all import *
>>> CFiniteSequence(x*(x**Integer(2)+Integer(4)*x+Integer(1))/(Integer(1)-
↪x)**Integer(5)).closed_form()
1/4*n^4 + 1/2*n^3 + 1/4*n^2
>>> CFiniteSequence((Integer(1)+Integer(2)*x-x**Integer(2))/(Integer(1)-
↪x)**Integer(4)/(Integer(1)+x)**Integer(2)).closed_form()
1/12*n^3 - 1/8*(-1)^n*(n + 1) + 3/4*n^2 + 43/24*n + 9/8
>>> CFiniteSequence(Integer(1)/(Integer(1)-x)**Integer(3)/(Integer(1)-
↪Integer(2)*x)**Integer(4)).closed_form()
4/3*(n^3 - 3*n^2 + 20*n - 36)*2^n + 1/2*n^2 + 19/2*n + 49
>>> CFiniteSequence((x/(Integer(1)-x-x**Integer(2)))**Integer(2)).closed_
↪form()
1/5*(n - sqrt(1/5))*(1/2*sqrt(5) + 1/2)^n + 1/5*(n + sqrt(1/5))*(-1/2*sqrt(5)␣
↪+ 1/2)^n
```

coefficients()

Return the coefficients of the recurrence representation of the C-finite sequence.

OUTPUT: list of values

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: lucas = C((2-x)/(1-x-x^2))    # the Lucas sequence
sage: lucas.coefficients()
[1, 1]
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> lucas = C((Integer(2)-x)/(Integer(1)-x-x**Integer(2)))    # the Lucas␣
↪sequence
>>> lucas.coefficients()
[1, 1]
```

denominator()

Return the numerator of the o.g.f of `self`.

EXAMPLES:

```
sage: f = CFiniteSequence((2-x)/(1-x-x^2)); f
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
sage: f.denominator()
x^2 + x - 1
```

```
>>> from sage.all import *
>>> f = CFiniteSequence((Integer(2)-x)/(Integer(1)-x-x**Integer(2))); f
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
>>> f.denominator()
x^2 + x - 1
```

numerator()

Return the numerator of the o.g.f of `self`.

EXAMPLES:

```
sage: f = CFiniteSequence((2-x)/(1-x-x^2)); f
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
sage: f.numerator()
x - 2
```

```
>>> from sage.all import *
>>> f = CFiniteSequence((Integer(2)-x)/(Integer(1)-x-x**Integer(2))); f
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
>>> f.numerator()
x - 2
```

ogf()

Return the ordinary generating function associated with the CFiniteSequence.

This is always a fraction of polynomials in the base ring.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: r = C.from_recurrence([2],[1])
sage: r.ogf()
-1/2/(x - 1/2)
sage: C(0).ogf()
0
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> r = C.from_recurrence([Integer(2)],[Integer(1)])
>>> r.ogf()
-1/2/(x - 1/2)
>>> C(Integer(0)).ogf()
0
```

recurrence_repr()

Return a string with the recurrence representation of the C-finite sequence.

OUTPUT: string

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C((2-x)/(1-x-x^2)).recurrence_repr()
'homogeneous linear recurrence with constant coefficients of degree 2: a(n+2)␣
↪= a(n+1) + a(n), starting a(0...) = [2, 1]'
sage: C(x/(1-x)^3).recurrence_repr()
'homogeneous linear recurrence with constant coefficients of degree 3: a(n+3)␣
↪= 3*a(n+2) - 3*a(n+1) + a(n), starting a(1...) = [1, 3, 6]'
sage: C(1).recurrence_repr()
'Finite sequence [1], offset 0'
sage: r = C((-2*x^3 + x^2 - x + 1)/(2*x^2 - 3*x + 1))
sage: r.recurrence_repr()
'homogeneous linear recurrence with constant coefficients of degree 2: a(n+2)␣
↪= 3*a(n+1) - 2*a(n), starting a(0...) = [1, 2, 5, 9]'
sage: r = CFiniteSequence(x^3/(1-x-x^2))
sage: r.recurrence_repr()
'homogeneous linear recurrence with constant coefficients of degree 2: a(n+2)␣
↪= a(n+1) + a(n), starting a(3...) = [1, 1, 2, 3]'
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C((Integer(2)-x)/(Integer(1)-x-x**Integer(2))).recurrence_repr()
'homogeneous linear recurrence with constant coefficients of degree 2: a(n+2)␣
↪= a(n+1) + a(n), starting a(0...) = [2, 1]'
>>> C(x/(Integer(1)-x)**Integer(3)).recurrence_repr()
'homogeneous linear recurrence with constant coefficients of degree 3: a(n+3)␣
↪= 3*a(n+2) - 3*a(n+1) + a(n), starting a(1...) = [1, 3, 6]'
>>> C(Integer(1)).recurrence_repr()
'Finite sequence [1], offset 0'
>>> r = C((-Integer(2)*x**Integer(3) + x**Integer(2) - x + Integer(1))/
↪(Integer(2)*x**Integer(2) - Integer(3)*x + Integer(1)))
>>> r.recurrence_repr()
'homogeneous linear recurrence with constant coefficients of degree 2: a(n+2)␣
↪= 3*a(n+1) - 2*a(n), starting a(0...) = [1, 2, 5, 9]'
>>> r = CFiniteSequence(x**Integer(3)/(Integer(1)-x-x**Integer(2)))
>>> r.recurrence_repr()
'homogeneous linear recurrence with constant coefficients of degree 2: a(n+2)␣
↪= a(n+1) + a(n), starting a(3...) = [1, 1, 2, 3]'
```

series(n)

Return the Laurent power series associated with the CFiniteSequence, with precision n.

INPUT:

- n – nonnegative integer

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: r = C.from_recurrence([-1,2],[0,1])
sage: s = r.series(4); s
x + 2*x^2 + 3*x^3 + 4*x^4 + O(x^5)
sage: type(s)
<class 'sage.rings.laurent_series_ring_element.LaurentSeries'>
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> r = C.from_recurrence([-Integer(1),Integer(2)],[Integer(0),Integer(1)])
>>> s = r.series(Integer(4)); s
x + 2*x^2 + 3*x^3 + 4*x^4 + O(x^5)
>>> type(s)
<class 'sage.rings.laurent_series_ring_element.LaurentSeries'>
```

`sage.rings.cfinite_sequence.CFiniteSequences(base_ring, names=None, category=None)`

Return the commutative ring of C-Finite sequences.

The ring is defined over a base ring ($\mathbf{Z}$ or $\mathbf{Q}$) and each element is represented by its ordinary generating function (ogf) which is a rational function over the base ring.

INPUT:

- `base_ring` – the base ring to construct the fraction field representing the C-Finite sequences
- `names` – (optional) the list of variables

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C
The ring of C-Finite sequences in x over Rational Field
sage: C.an_element()
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
sage: C.category()
Category of commutative rings
sage: C.one()
Finite sequence [1], offset = 0
sage: C.zero()
Constant infinite sequence 0.
sage: C(x)
Finite sequence [1], offset = 1
sage: C(1/x)
Finite sequence [1], offset = -1
sage: C((-x + 2)/(-x^2 - x + 1))
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C
The ring of C-Finite sequences in x over Rational Field
>>> C.an_element()
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
>>> C.category()
Category of commutative rings
>>> C.one()
Finite sequence [1], offset = 0
>>> C.zero()
Constant infinite sequence 0.
>>> C(x)
Finite sequence [1], offset = 1
>>> C(Integer(1)/x)
```

(continues on next page)

(continued from previous page)

```
Finite sequence [1], offset = -1
>>> C((-x + Integer(2))/(-x**Integer(2) - x + Integer(1)))
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
```

class sage.rings.cfinite_sequence.**CFiniteSequences_generic**(*polynomial_ring*, *category*)

Bases: Parent, UniqueRepresentation

The class representing the ring of C-Finite Sequences.

Element

alias of *CFiniteSequence*

an_element()

Return an element of C-Finite Sequences.

OUTPUT: the Lucas sequence

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.an_element()
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C.an_element()
C-finite sequence, generated by (x - 2)/(x^2 + x - 1)
```

fraction_field()

Return the fraction field used to represent the elements of `self`.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.fraction_field()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C.fraction_field()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

from_recurrence(*coefficients*, *values*)

Create a C-finite sequence given the coefficients c and starting values a of a homogeneous linear recurrence.

$$a_{n+d} = c_0 a_n + c_1 a_{n+1} + \cdots + c_{d-1} a_{n+d-1}, \quad d \geq 0.$$

INPUT:

- `coefficients` – list of rationals
- `values` – start values, a list of rationals

OUTPUT: a CFiniteSequence object

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.from_recurrence([1,1],[0,1])    # Fibonacci numbers
C-finite sequence, generated by -x/(x^2 + x - 1)
sage: C.from_recurrence([-1,2],[0,1])     # natural numbers
C-finite sequence, generated by x/(x^2 - 2*x + 1)
sage: r = C.from_recurrence([-1],[1])
sage: s = C.from_recurrence([-1],[1,-1])
sage: r == s
True
sage: r = C(x^3/(1-x-x^2))
sage: s = C.from_recurrence([1,1],[0,0,0,1,1])
sage: r == s
True
sage: C.from_recurrence(1,1)
Traceback (most recent call last):
...
ValueError: Wrong type for recurrence coefficient list.
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C.from_recurrence([Integer(1),Integer(1)],[Integer(0),Integer(1)])    #
↪Fibonacci numbers
C-finite sequence, generated by -x/(x^2 + x - 1)
>>> C.from_recurrence([-Integer(1),Integer(2)],[Integer(0),Integer(1)])     #
↪natural numbers
C-finite sequence, generated by x/(x^2 - 2*x + 1)
>>> r = C.from_recurrence([-Integer(1)],[Integer(1)])
>>> s = C.from_recurrence([-Integer(1)],[Integer(1),-Integer(1)])
>>> r == s
True
>>> r = C(x**Integer(3)/(Integer(1)-x-x**Integer(2)))
>>> s = C.from_recurrence([Integer(1),Integer(1)],[Integer(0),Integer(0),
↪Integer(0),Integer(1),Integer(1)])
>>> r == s
True
>>> C.from_recurrence(Integer(1),Integer(1))
Traceback (most recent call last):
...
ValueError: Wrong type for recurrence coefficient list.
```

gen(*i=0*)

Return the i-th generator of `self`.

INPUT:

- `i` – integer (default: 0)

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.gen()
x
sage: x == C.gen()
True
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C.gen()
x
>>> x == C.gen()
True
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.gens()
(x,)
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C.gens()
(x,)
```

guess(*sequence*, *algorithm='sage'*)

Return the minimal CFiniteSequence that generates the sequence.

Assume the first value has index 0.

INPUT:

- `sequence` – list of integers
- `algorithm` – string; one of - `'sage'` – the default is to use Sage's matrix kernel function - `'pari'` – use Pari's implementation of LLL - `'bm'` – use Sage's Berlekamp-Massey algorithm

OUTPUT: a CFiniteSequence, or 0 if none could be found

With the default kernel method, trailing zeroes are chopped off before a guessing attempt. This may reduce the data below the accepted length of six values.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.guess([1,2,4,8,16,32])
C-finite sequence, generated by -1/2/(x - 1/2)
sage: r = C.guess([1,2,3,4,5])
Traceback (most recent call last):
...
ValueError: sequence too short for guessing
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C.guess([Integer(1),Integer(2),Integer(4),Integer(8),Integer(16),
↪Integer(32)])
C-finite sequence, generated by -1/2/(x - 1/2)
>>> r = C.guess([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: sequence too short for guessing
```

With Berlekamp-Massey, if an odd number of values is given, the last one is dropped. So with an odd number of values the result may not generate the last value:

```
sage: r = C.guess([1,2,4,8,9], algorithm='bm'); r
C-finite sequence, generated by -1/2/(x - 1/2)
sage: r[0:5]
[1, 2, 4, 8, 16]
```

```
>>> from sage.all import *
>>> r = C.guess([Integer(1),Integer(2),Integer(4),Integer(8),Integer(9)],␣
↪algorithm='bm'); r
C-finite sequence, generated by -1/2/(x - 1/2)
>>> r[Integer(0):Integer(5)]
[1, 2, 4, 8, 16]
```

Using pari:

```
sage: r = C.guess([1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 12, 16, 21, 28], algorithm=
↪'pari'); r
C-finite sequence, generated by (-x - 1)/(x^3 + x^2 - 1)
sage: r[0:5]
[1, 1, 1, 2, 2]
```

```
>>> from sage.all import *
>>> r = C.guess([Integer(1), Integer(1), Integer(1), Integer(2), Integer(2),␣
↪Integer(3), Integer(4), Integer(5), Integer(7), Integer(9), Integer(12),␣
↪Integer(16), Integer(21), Integer(28)], algorithm='pari'); r
C-finite sequence, generated by (-x - 1)/(x^3 + x^2 - 1)
>>> r[Integer(0):Integer(5)]
[1, 1, 1, 2, 2]
```

ngens()

Return the number of generators of `self`.

EXAMPLES:

```
sage: from sage.rings.cfinite_sequence import CFiniteSequences
sage: C.<x> = CFiniteSequences(QQ)
sage: C.ngens()
1
```

```
>>> from sage.all import *
>>> from sage.rings.cfinite_sequence import CFiniteSequences
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C.ngens()
1
```

polynomial_ring()

Return the polynomial ring used to represent the elements of `self`.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> C = CFiniteSequences(QQ, names=('x',)); (x,) = C._first_ngens(1)
>>> C.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field
```

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[Rei2002] Markus Reineke, *The Harder-Narasimhan system in quantum groups and cohomology of quiver moduli*, arXiv math/0204059

[RSW2004] Reiner, Stanton, White - *The cyclic sieving phenomenon*, Journal of Combinatorial Theory A 108 (2004)

[ClaytonSmith] On the existence of $(v, 5, 1)$-BIBD. http://www.argilo.net/files/bibd.pdf Clayton Smith

[Denniston69] R. H. F. Denniston, Some maximal arcs in finite projective planes. Journal of Combinatorial Theory 6, no. 3 (1969): 317-319. doi:10.1016/S0021-9800(69)80095-5

[AndHonk97] A short course in Combinatorial Designs, Ian Anderson, Iiro Honkala, Internet Editions, Spring 1997, http://www.utu.fi/~honkala/designs.ps

[Stinson91] D.R. Stinson, A survey of Kirkman triple systems and related designs, Volume 92, Issues 1-3, 17 November 1991, Pages 371-393, Discrete Mathematics, doi:10.1016/0012-365X(91)90294-C

[RCW71] D. K. Ray-Chaudhuri, R. M. Wilson, Solution of Kirkman's schoolgirl problem, Volume 19, Pages 187-203, Proceedings of Symposia in Pure Mathematics

[BJL99] T. Beth, D. Jungnickel, H. Lenz, Design Theory 2ed. Cambridge University Press 1999

[Hu57] Daniel R. Hughes, "A class of non-Desarguesian projective planes", The Canadian Journal of Mathematics (1957), http://cms.math.ca/cjm/v9/p378

[We07] Charles Weibel, "Survey of Non-Desarguesian planes" (2007), notices of the AMS, vol. 54 num. 10, pages 1294–1303

[CvL] P. Cameron, J. H. van Lint, Designs, graphs, codes and their links, London Math. Soc., 1991.

[DesignHandbook] Handbook of Combinatorial Designs (2ed) Charles Colbourn, Jeffrey Dinitz Chapman & Hall/CRC 2012

[Aschbacher71] M. Aschbacher, On collineation groups of symmetric block designs. J. Combinatorial Theory Ser. A 11 (1971), pp. 272–281.

[Hall71] M. Hall, Jr., Combinatorial designs and groups. Actes du Congrès International des Mathématiciens (Nice, 1970), v.3, pp. 217–222. Gauthier-Villars, Paris, 1971.

[HT95] W. Huffman and V. Tonchev, The existence of extremal self-dual $[50, 25, 10]$ codes and quasi-symmetric $2-(49, 9, 6)$ designs, Designs, Codes and Cryptography September 1995, Volume 6, Issue 2, pp 97-106

[Hanani75] Haim Hanani, Balanced incomplete block designs and related designs, doi:10.1016/0012-365X(75)90040-0, Discrete Mathematics, Volume 11, Issue 3, 1975, Pages 255-369.

[JulianAbel13] Existence of Five MOLS of Orders 18 and 60 R. Julian R. Abel Journal of Combinatorial Designs 2013

[KY04] S. Klee and L. Yates, Tight Subdesigns of the Higman-Sims Design, Rose-Hulman Undergraduate Math. J 5.2 (2004). https://www.rose-hulman.edu/mathjournal/archives/2004/vol5-n2/paper9/v5n2-9pd.pdf

[Todorov12] D.T. Todorov, Four mutually orthogonal Latin squares of order 14, Journal of Combinatorial Designs 2012, vol.20 n.8 pp.363-367

[BJL99-1] T. Beth, D. Jungnickel, H. Lenz "Design theory Vol. I." Second edition. Encyclopedia of Mathematics and its Applications, 69. Cambridge University Press, (1999).

[BLJ99-2] T. Beth, D. Jungnickel, H. Lenz "Design theory Vol. II." Second edition. Encyclopedia of Mathematics and its Applications, 78. Cambridge University Press, (1999).

[Bo39] R. C. Bose, "On the construction of balanced incomplete block designs", Ann. Eugenics, 9 (1939), 353–399.

[Bu95] M. Buratti "On simple radical difference families", J. Combinatorial Designs, 3 (1995) 161–168.

[Tu1965] R. J. Turyn "Character sum and difference sets" Pacific J. Math. 15 (1965) 319–346.

[Tu1984] R. J. Turyn "A special class of Williamson matrices and difference sets" J. Combinatorial Theory (A) 36 (1984) 111–115.

[Wi72] R. M. Wilson "Cyclotomy and difference families in elementary Abelian groups", J. Number Theory, 4 (1972) 17–47.

[McF1973] Robert L. McFarland "A family of difference sets in non-cyclic groups" J. Combinatorial Theory (A) 15 (1973) 1–10. doi:10.1016/0097-3165(73)90031-9

[Stinson2004] Douglas R. Stinson, *Combinatorial designs: construction and analysis*, Springer, 2004.

[ColDin01] Charles Colbourn, Jeffrey Dinitz, *Mutually orthogonal latin squares: a brief survey of constructions*, Volume 95, Issues 1-2, Pages 9-48, Journal of Statistical Planning and Inference, Springer, 1 May 2001.

[HananiBIBD] Balanced incomplete block designs and related designs, Haim Hanani, Discrete Mathematics 11.3 (1975) pages 255-369.

[Brouwer80] A Series of Separable Designs with Application to Pairwise Orthogonal Latin Squares, Andries E. Brouwer, Vol. 1, n. 1, pp. 39-41, European Journal of Combinatorics, 1980 http://www.sciencedirect.com/science/article/pii/S0195669880800199

[Greig99] Designs from projective planes and PBD bases Malcolm Greig Journal of Combinatorial Designs vol. 7, num. 5, pp. 341–374 1999

[DukesLing14] A three-factor product construction for mutually orthogonal latin squares, Peter J. Dukes, Alan C.H. Ling, arXiv 1401.1466

[Rees00] Truncated Transversal Designs: A New Lower Bound on the Number of Idempotent MOLS of Side, Rolf S. Rees, Journal of Combinatorial Theory, Series A 90.2 (2000): 257-266.

[Rees93] Two new direct product-type constructions for resolvable group-divisible designs, Rolf S. Rees, Journal of Combinatorial Designs 1.1 (1993): 15-26.

[Thwarts] Thwarts in transversal designs Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas. Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.

[OS64] Finite projective planes with affine subplanes, T. G. Ostrom and F. A. Sherk. Canad. Math. Bull vol7 num.4 (1964)

[AC07] Concerning eight mutually orthogonal latin squares Julian R. Abel, Nicholas Cavenagh Journal of Combinatorial Designs Vol. 15, n.3, pp. 255-261 2007

[Naz96] Maxim Nazarov, Young's Orthogonal Form for Brauer's Centralizer Algebra. Journal of Algebra 182 (1996), 664–693.

[GL1996] J.J. Graham and G.I. Lehrer, Cellular algebras. Inventiones mathematicae 123 (1996), 1–34.

[Sta-EC2] Richard P. Stanley. *Enumerative Combinatorics*, Volume 2. Cambridge University Press, 2001.

[StaCat98] Richard Stanley. *Exercises on Catalan and Related Numbers excerpted from Enumerative Combinatorics, vol. 2 (CUP 1999)*, version of 23 June 1998. http://www-math.mit.edu/~rstan/ec/catalan.pdf

[Hag2008] James Haglund. *The q, t – Catalan Numbers and the Space of Diagonal Harmonics: With an Appendix on the Combinatorics of Macdonald Polynomials*. University of Pennsylvania, Philadelphia – AMS, 2008, 167 pp.

[BK2001] J. Bandlow, K. Killpatrick – *An area-to_inv bijection between Dyck paths and 312-avoiding permutations*, Electronic Journal of Combinatorics, Volume 8, Issue 1 (2001).

[EP2004] S. Elizalde, I. Pak. *Bijections for refined restricted permutations**. JCTA 105(2) 2004.

[CK2008] A. Claesson, S. Kitaev. *Classification of bijections between `321`- and `132`- avoiding permutations*. Séminaire Lotharingien de Combinatoire **60** 2008. arXiv 0805.1325.

[Knu1973] D. Knuth. *The Art of Computer Programming, Vol. III*. Addison-Wesley. Reading, MA. 1973.

[Kra2001] C. Krattenthaler – *Permutations with restricted patterns and Dyck paths*, Adv. Appl. Math. 27 (2001), 510–530.

[DS1992] A. Denise, R. Simion, *Two combinatorial statistics on Dyck paths*, Discrete Math 137 (1992), 155–176.

[AI] P. Arnoux, S. Ito, Pisot substitutions and Rauzy fractals, Bull. Belg. Math. Soc. 8 (2), 2001, pp. 181–207

[SAI] Y. Sano, P. Arnoux, S. Ito, Higher dimensional extensions of substitutions and their dual maps, J. Anal. Math. 83, 2001, pp. 183–206

[HKP2015] Clemens Heuberger, Sara Kropf, and Helmut Prodinger, *Output sum of transducers: Limiting distribution and periodic fluctuation*, Electron. J. Combin. 22 (2015), #P2.19.

[HKW2015] Clemens Heuberger, Sara Kropf and Stephan Wagner, *Variances and Covariances in the Central Limit Theorem for the Output of a Transducer*, European J. Combin. 49 (2015), 167-187, doi:10.1016/j.ejc.2015.03.004.

[HKP2015a] Clemens Heuberger, Sara Kropf, and Helmut Prodinger, *Analysis of Carries in Signed Digit Expansions*, arXiv 1503.08816.

[P1964] William Parry, *Intrinsic Markov chains*, Transactions of the American Mathematical Society 112, 1964, pp. 55-66. doi:10.1090/S0002-9947-1964-0161372-1.

[S1948] Claude E. Shannon, *A mathematical theory of communication*, The Bell System Technical Journal 27, 1948, 379-423, doi:10.1002/j.1538-7305.1948.tb01338.x.

[HP2007] Clemens Heuberger and Helmut Prodinger, *The Hamming Weight of the Non-Adjacent-Form under Various Input Statistics*, Periodica Mathematica Hungarica Vol. 55 (1), 2007, pp. 81–96, doi:10.1007/s10998-007-3081-z.

[FGT1992] Philippe Flajolet, Danièle Gardy, Loÿs Thimonier, *Birthday paradox, coupon collectors, caching algorithms and self-organizing search*, Discrete Appl. Math. 39 (1992), 207–229, doi:10.1016/0166-218X(92)90177-C.

[FHP2015] Uta Freiberg, Clemens Heuberger, Helmut Prodinger, *Application of Smirnov Words to Waiting Time Distributions of Runs*, arXiv 1503.08096.

[S1986] Gábor J. Székely, *Paradoxes in Probability Theory and Mathematical Statistics*, D. Reidel Publishing Company.

[BaWo2012] Javier Baliosian and Dina Wonsever, *Finite State Transducers*, chapter in *Handbook of Finite State Based Models and Applications*, edited by Jiacun Wang, Chapman and Hall/CRC, 2012.

[BBF] B. Brubaker, D. Bump, and S. Friedberg. Weyl Group Multiple Dirichlet Series: Type A Combinatorial Theory. Ann. of Math. Stud., vol. 175, Princeton Univ. Press, New Jersey, 2011.

[GC50] I. M. Gelfand and M. L. Cetlin. Finite-Dimensional Representations of the Group of Unimodular Matrices. Dokl. Akad. Nauk SSSR **71**, pp. 825–828, 1950.

[Tok88] T. Tokuyama. A Generating Function of Strict Gelfand Patterns and Some Formulas on Characters of General Linear Groups. J. Math. Soc. Japan **40** (4), pp. 671–685, 1988.

[Ryser63] H. J. Ryser, Combinatorial Mathematics, Carus Monographs, MAA, 1963.

[Gale57] D. Gale, A theorem on flows in networks, Pacific J. Math. 7(1957)1073-1082.

[KnutsonPurbhoo10] A. Knutson, K. Purbhoo, Product and puzzle formulae for GL_n Belkale-Kumar coefficients, arXiv 1008.4979

[KT2003] Allen Knutson, Terence Tao, Puzzles and (equivariant) cohomology of Grassmannians, Duke Math. J. 119 (2003) 221

[CoskunVakil06] I. Coskun, R. Vakil, Geometric positivity in the cohomology of homogeneous spaces and generalized Schubert calculus, arXiv math/0610538

[KTW] Allen Knutson, Terence Tao, Christopher Woodward, The honeycomb model of GL(n) tensor products II: Puzzles determine facets of the Littlewood-Richardson cone, arXiv math/0107011

[BuchKreschTamvakis03] A. Buch, A. Kresch, H. Tamvakis, Gromov-Witten invariants on Grassmannian, arXiv math/0306388

[Buch00] A. Buch, A Littlewood-Richardson rule for the K-theory of Grassmannians, arXiv math.AG/0004137

[JacMat96] Mark T. Jacobson and Peter Matthews, "Generating uniformly distributed random Latin squares", Journal of Combinatorial Designs, 4 (1996)

[NCSF] Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, *Noncommutative Symmetric Functions*, Adv. Math. 112 (1995), no. 2, 218-348.

[QSCHUR] Haglund, Luoto, Mason, van Willigenburg, *Quasisymmetric Schur functions*, J. Comb. Theory Ser. A 118 (2011), 463-490. http://www.sciencedirect.com/science/article/pii/S0097316509001745 , arXiv 0810.2489v2.

[Tev2007] Lenny Tevlin, *Noncommutative Analogs of Monomial Symmetric Functions, Cauchy Identity, and Hall Scalar Product*, arXiv 0712.2201v1.

[Ges] I. Gessel, *Multipartite P-partitions and inner products of skew Schur functions*, Contemp. Math. **34** (1984), 289-301. http://people.brandeis.edu/~gessel/homepage/papers/multipartite.pdf

[MR] C. Malvenuto and C. Reutenauer, *Duality between quasi-symmetric functions and the Solomon descent algebra*, J. Algebra **177** (1995), no. 3, 967-982. http://www.mat.uniroma1.it/people/malvenuto/Duality.pdf

[Mal1993] Claudia Malvenuto, *Produits et coproduits des fonctions quasi-symétriques et de l'algèbre des descentes*, thesis, November 1993. http://www1.mat.uniroma1.it/people/malvenuto/Thesis.pdf

[Haz2004] Michiel Hazewinkel, *Explicit polynomial generators for the ring of quasisymmetric functions over the integers*. arXiv math/0410366v1

[Rad1979] David E. Radford, *A natural ring basis for the shuffle algebra and an application to group schemes*, J. Algebra **58** (1979), 432-454.

[NCSF1] Israel Gelfand, D. Krob, Alain Lascoux, B. Leclerc, V. S. Retakh, J.-Y. Thibon, *Noncommutative symmetric functions*. arXiv hep-th/9407124v1

[NCSF2] D. Krob, B. Leclerc, J.-Y. Thibon, *Noncommutative symmetric functions II: Transformations of alphabets*. http://www-igm.univ-mlv.fr/~jyt/ARTICLES/NCSF2.ps

[HLNT09] F. Hivert, J.-G. Luque, J.-C. Novelli, J.-Y. Thibon, *The (1-E)-transform in combinatorial Hopf algebras*. arXiv math/0912.0184v2

[LMvW13] Kurt Luoto, Stefan Mykytiuk and Stephanie van Willigenburg, *An introduction to quasisymmetric Schur functions – Hopf algebras, quasisymmetric functions, and Young composition tableaux*, May 23, 2013, Springer. http://www.math.ubc.ca/%7Esteph/papers/QuasiSchurBook.pdf

[BBSSZ2012] Chris Berg, Nantel Bergeron, Franco Saliola, Luis Serrano, Mike Zabrocki, *A lift of the Schur and Hall-Littlewood bases to non-commutative symmetric functions*, arXiv 1208.5191v3.

[Hoff2015] Michael Hoffman. *Quasi-symmetric functions and mod p multiple harmonic sums*. Kyushu J. Math. **69** (2015), pp. 345-366. doi:10.2206/kyushujm.69.345, arXiv math/0401319v3.

[BDHMN2017] Cristina Ballantine, Zajj Daugherty, Angela Hicks, Sarah Mason, Elizabeth Niese. *Quasisymmetric power sums*. arXiv 1710.11613.

[AHM2018] Edward Allen, Joshua Hallam, Sarah Mason, *Dual Immaculate Quasisymmetric Functions Expand Positively into Young Quasisymmetric Schur Functions*. arXiv 1606.03519

[SW2010] John Shareshian and Michelle Wachs. *Eulerian quasisymmetric functions*. (2010). arXiv 0812.0764v2

[HHL05] *A combinatorial formula for Macdonald polynomials*. Haiman, Haglund, and Loehr. J. Amer. Math. Soc. 18 (2005), no. 3, 735-761.

[LW12] *Quasisymmetric expansions of Schur-function plethysms*. Loehr and Warrington. Proc. Amer. Math. Soc. 140 (2012), no. 4, 1159-1171.

[KT97] *Noncommutative symmetric functions IV: Quantum linear groups and Hecke algebras at $q = 0$*. Krob and Thibon. Journal of Algebraic Combinatorics 6 (1997), 339-376.

[HNT06] F. Hivert, J.-C. Novelli, J.-Y. Thibon. *Commutative combinatorial Hopf algebras*. (2006). arXiv 0605262v1.

[BZ05] N. Bergeron, M. Zabrocki. *The Hopf algebra of symmetric functions and quasisymmetric functions in non-commutative variables are free and cofree*. (2005). arXiv math/0509265v3.

[BHRZ06] N. Bergeron, C. Hohlweg, M. Rosas, M. Zabrocki. *Grothendieck bialgebras, partition lattices, and symmetric functions in noncommutative variables*. Electronic Journal of Combinatorics. **13** (2006).

[RS06] M. Rosas, B. Sagan. *Symmetric functions in noncommuting variables*. Trans. Amer. Math. Soc. **358** (2006). no. 1, 215-232. arXiv math/0208168.

[BRRZ08] N. Bergeron, C. Reutenauer, M. Rosas, M. Zabrocki. *Invariants and coinvariants of the symmetric group in noncommuting variables*. Canad. J. Math. **60** (2008). 266-296. arXiv math/0502082

[BT13] N. Bergeron, N. Thiem. *A supercharacter table decomposition via power-sum symmetric functions*. Int. J. Algebra Comput. **23**, 763 (2013). doi:10.1142/S0218196713400171. arXiv 1112.4901.

[Beck] M. Beck, Stanford Math Circle - Parking Functions, October 2010, http://math.stanford.edu/circle/parkingBeck.pdf

[Hag08] The q, t – Catalan Numbers and the Space of Diagonal Harmonics: With an Appendix on the Combinatorics of Macdonald Polynomials, James Haglund, University of Pennsylvania, Philadelphia – AMS, 2008, 167 pp.

[Shin] H. Shin, Forests and Parking Functions, slides from talk September 24, 2008, http://www.emis.de/journals/SLC/wpapers/s61vortrag/shin.pdf

[GXZ] A. M. Garsia, G. Xin, M. Zabrocki, A three shuffle case of the compositional parking function conjecture, arXiv 1208.5796v1

[MV] combinatorics of orthogonal polynomials (A. de Medicis et X.Viennot, Moments des q-polynômes de Laguerre et la bijection de Foata-Zeilberger, Adv. Appl. Math., 15 (1994), 262-304)

[McD] combinatorics of hyperoctahedral group, double coset algebra and zonal polynomials (I. G. Macdonald, Symmetric functions and Hall polynomials, Oxford University Press, second edition, 1995, chapter VII).

[CM] Benoit Collins, Sho Matsumoto, *On some properties of orthogonal Weingarten functions*, arXiv 0903.5143.

[Gec81] Fundamentals of Computation Theory Gecseg, F. Proceedings of the 1981 International Fct-Conference Szeged, Hungaria, August 24-28, vol 117 Springer-Verlag, 1981

[Thom2006] Hugh Thomas, *An analogue of distributivity for ungraded lattices*. Order 23 (2006), no. 2-3, 249-269.

[Solomon67] Louis Solomon. *The Burnside Algebra of a Finite Group*. Journal of Combinatorial Theory, **2**, 1967. doi:10.1016/S0021-9800(67)80064-4.

[Greene73] Curtis Greene. *On the Möbius algebra of a partially ordered set*. Advances in Mathematics, **10**, 1973. doi:10.1016/0001-8708(73)90106-0.

[Etienne98] Gwihen Etienne. *On the Möbius algebra of geometric lattices*. European Journal of Combinatorics, **19**, 1998. doi:10.1006/eujc.1998.0227.

[Feig1986] Joan Feigenbaum, *Directed Cartesian-Product Graphs have Unique Factorizations that can be computed in Polynomial Time*, Discrete Applied Mathematics 15 (1986) 105-110 doi:10.1016/0166-218X(86)90023-5

[CH2006] William Y.C. Chen and Qing-Hu Hou, *Factors of the Gaussian coefficients*, Discrete Mathematics 306 (2006), 1446-1449. doi:10.1016/j.disc.2006.03.031

[Bu87] Butler, Lynne M. *A unimodality result in the enumeration of subgroups of a finite abelian group.* Proceedings of the American Mathematical Society 101, no. 4 (1987): 771-775. doi:10.1090/S0002-9939-1987-0911049-8

[Delsarte48] S. Delsarte, *Fonctions de Möbius Sur Les Groupes Abéliens Finis*, Annals of Mathematics, second series, Vol. 45, No. 3, (Jul 1948), pp. 600-609. http://www.jstor.org/stable/1969047

[Ca1948] Leonard Carlitz, "q-Bernoulli numbers and polynomials". Duke Math J. 15, 987-1000 (1948), doi:10.1215/S0012-7094-48-01588-9

[Ca1954] Leonard Carlitz, "q-Bernoulli and Eulerian numbers". Trans Am Soc. 76, 332-350 (1954), doi:10.1090/S0002-9947-1954-0060538-2

[vanLeeuwen91] Marc. A. A. van Leeuwen, *Edge sequences, ribbon tableaux, and an action of affine permutations*. Europe J. Combinatorics. **20** (1999). http://wwwmathlabo.univ-poitiers.fr/~maavl/pdf/edgeseqs.pdf

[RC-MLT] Ben Salisbury and Travis Scrimshaw. *Connecting marginally large tableaux and rigged configurations via crystals*. Preprint. arXiv 1505.07040.

[Kleber1] Michael Kleber. *Combinatorial structure of finite dimensional representations of Yangians: the simply-laced case*. Internat. Math. Res. Notices. (1997) no. 4. 187-201.

[Kleber2] Michael Kleber. *Finite dimensional representations of quantum affine algebras*. Ph.D. dissertation at University of California Berkeley. (1998). arXiv math.QA/9809087.

[OSS03] Masato Okado, Anne Schilling, and Mark Shimozono. *Virtual crystals and Klebers algorithm*. Commun. Math. Phys. **238** (2003). 187-209. arXiv math.QA/0209082.

[OSS13] Masato Okado, Reiho Sakamoto, and Anne Schilling. *Affine crystal structure on rigged configurations of type* $D_n^{(1)}$. J. Algebraic Combinatorics, **37** (2013). 571-599. arXiv 1109.3523.

[HKOTT2002] G. Hatayama, A. Kuniba, M. Okado, T. Takagi, Z. Tsuboi. Paths, Crystals and Fermionic Formulae. Prog. Math. Phys. **23** (2002) Pages 205-272.

[CrysStructSchilling06] Anne Schilling. Crystal structure on rigged configurations. International Mathematics Research Notices. Volume 2006. (2006) Article ID 97376. Pages 1-27.

[RigConBijection] Masato Okado, Anne Schilling, Mark Shimozono. A crystal to rigged configuration bijection for non-exceptional affine algebras. Algebraic Combinatorics and Quantum Groups. Edited by N. Jing. World Scientific. (2003) Pages 85-124.

[BijectionDn] Anne Schilling. A bijection between type $D_n^{(1)}$ crystals and rigged configurations. J. Algebra. **285** (2005) 292-334

[BijectionLRT] Anatol N. Kirillov, Anne Schilling, Mark Shimozono. A bijection between Littlewood-Richardson tableaux and rigged configurations. Selecta Mathematica (N.S.). **8** (2002) Pages 67-135. (MathSciNet MR1890195).

[OSS2003] Masato Okado, Anne Schilling, and Mark Shimozono. Virtual crystals and fermionic formulas of type $D_{n+1}^{(2)}$, $A_{2n}^{(2)}$, and $C_n^{(1)}$. Representation Theory. **7** (2003) arXiv math.QA/0105017.

[Sakamoto13] Reiho Sakamoto. Rigged configurations and Kashiwara operators. (2013) arXiv 1302.4562v1.

[OSS2011] Masato Okado, Reiho Sakamoto, Anne Schilling, Affine crystal structure on rigged configurations of type $D_n^{(1)}$, J. Algebraic Combinatorics 37(3) (2013) 571-599 (arXiv 1109.3523 [math.QA])

[FSS07] G. Fourier, A. Schilling, and M. Shimozono, *Demazure structure inside Kirillov-Reshetikhin crystals*, J. Algebra, Vol. 309, (2007), p. 386-404 arXiv math/0605451

[HST09] F. Hivert, A. Schilling, and N. M. Thiery, *Hecke group algebras as quotients of affine Hecke algebras at level 0*, JCT A, Vol. 116, (2009) p. 844-863 arXiv 0804.3781

[KMPS] Kass, Moody, Patera and Slansky, *Affine Lie algebras, weight multiplicities, and branching rules*. Vols. 1, 2. University of California Press, Berkeley, CA, 1990.

[KacPeterson] Kac and Peterson. *Infinite-dimensional Lie algebras, theta functions and modular forms*. Adv. in Math. 53 (1984), no. 2, 125-264.

[Carter] Carter, *Lie algebras of finite and affine type*. Cambridge University Press, 2005

[HaimanICM] M. Haiman, Cherednik algebras, Macdonald polynomials and combinatorics, Proceedings of the International Congress of Mathematicians, Madrid 2006, Vol. III, 843-872.

[HHL06] J. Haglund, M. Haiman and N. Loehr, A combinatorial formula for nonsymmetric Macdonald polynomials, Amer. J. Math. 130, No. 2 (2008), 359-383.

[LNSSS12] C. Lenart, S. Naito, D. Sagaki, A. Schilling, M. Shimozono, A uniform model for Kirillov-Reshetikhin crystals I: Lifting the parabolic quantum Bruhat graph, preprint arXiv 1211.2042 [math.QA]

[Haiman06] M. Haiman, Cherednik algebras, Macdonald polynomials and combinatorics, ICM 2006.

[Lusztig1985] G. Lusztig, *Equivariant K-theory and representations of Hecke algebras*, Proc. Amer. Math. Soc. 94 (1985), no. 2, 337-342.

[Cherednik1995] I. Cherednik, *Nonsymmetric Macdonald polynomials*. IMRN 10, 483-515 (1995).

[Kumar1987] S. Kumar, Demazure character formula in arbitrary Kac-Moody setting, Invent. Math. 89 (1987), no. 2, 395-423.

[Lascoux2003] Alain Lascoux, Symmetric functions and combinatorial operators on polynomials, CBMS Regional Conference Series in Mathematics, 99, 2003.

[Reiner97] Victor Reiner. *Non-crossing partitions for classical reflection groups*. Discrete Mathematics 177 (1997)

[Arm06] Drew Armstrong. *Generalized Noncrossing Partitions and Combinatorics of Coxeter Groups*. arXiv math/0611106

[Iwahori] Iwahori, *Generalized Tits system (Bruhat decomposition) on p-adic semisimple groups*. 1966 Algebraic Groups and Discontinuous Subgroups (AMS Proc. Symp. Pure Math.., 1965) pp. 71-83 Amer. Math. Soc., Providence, R.I.

[Bour] Bourbaki, *Lie Groups and Lie Algebras* IV.2

[OSShimo03] M. Okado, A. Schilling, M. Shimozono. "Virtual crystals and fermionic formulas for type $D_{n+1}^{(2)}$, $A_{2n}^{(2)}$, and $C_n^{(1)}$". Representation Theory. **7** (2003). 101-163. doi:10.1.1.192.2095, arXiv 0810.5067.

[SL000081] Sloane's OEIS sequence A000081

[Knu1970] Donald E. Knuth. *Permutations, matrices, and generalized Young tableaux*. Pacific J. Math. Volume 34, Number 3 (1970), pp. 709-727. http://projecteuclid.org/euclid.pjm/1102971948

[EG1987] Paul Edelman, Curtis Greene. *Balanced Tableaux*. Advances in Mathematics 63 (1987), pp. 42-99. doi:10.1016/0001-8708(87)90063-6

[BKSTY06] A. Buch, A. Kresch, M. Shimozono, H. Tamvakis, and A. Yong. *Stable Grothendieck polynomials and K-theoretic factor sequences*. Math. Ann. **340** Issue 2, (2008), pp. 359–382. arXiv math/0601514v1.

[GR2018v5sol] Darij Grinberg, Victor Reiner. *Hopf Algebras In Combinatorics*, arXiv 1409.8356v5, available with solutions at https://arxiv.org/src/1409.8356v5/anc/HopfComb-v73-with-solutions.pdf

[OZ2015] R. Orellana, M. Zabrocki, *Symmetric group characters as symmetric functions*, arXiv 1510.00438.

[Jack1970] H. Jack, *A class of symmetric functions with a parameter*, Proc. R. Soc. Edinburgh (A), 69, 1-18.

[Ma1995] I. G. Macdonald, *Symmetric functions and Hall polynomials*, second ed., The Clarendon Press, Oxford University Press, New York, 1995, With contributions by A. Zelevinsky, Oxford Science Publications.

[Mc1995] I. G. Macdonald, Symmetric functions and Hall polynomials, second ed., The Clarendon Press, Oxford University Press, New York, 1995, With contributions by A. Zelevinsky, Oxford Science Publications.

[Lam2006] T. Lam, Schubert polynomials for the affine Grassmannian, J. Amer. Math. Soc., 21 (2008), 259-281.

[LLMSSZ] T. Lam, L. Lapointe, J. Morse, A. Schilling, M. Shimozono, M. Zabrocki, k-Schur functions and affine Schubert calculus.

[LLT1997] Alain Lascoux, Bernard Leclerc, Jean-Yves Thibon, Ribbon tableaux, Hall-Littlewood functions, quantum affine algebras, and unipotent varieties, J. Math. Phys. 38 (1997), no. 2, 1041-1068, arXiv q-alg/9512031v1 [math.q.alg]

[LT2000] Bernard Leclerc and Jean-Yves Thibon, Littlewood-Richardson coefficients and Kazhdan-Lusztig polynomials, in: Combinatorial methods in representation theory (Kyoto) Adv. Stud. Pure Math., vol. 28, Kinokuniya, Tokyo, 2000, pp 155-220 arXiv math/9809122v3 [math.q-alg]

[GH1993] A. Garsia, M. Haiman, A graded representation module for Macdonald's polynomials, Proc. Nat. Acad. U.S.A. no. 90, 3607–3610.

[BGHT1999] F. Bergeron, A. M. Garsia, M. Haiman, and G. Tesler, Identities and positivity conjectures for some remarkable operators in the theory of symmetric functions, Methods Appl. Anal. 6 (1999), no. 3, 363–420.

[LLM1998] L. Lapointe, A. Lascoux, J. Morse, Determinantal Expressions for Macdonald Polynomials, IRMN no. 18 (1998). arXiv math/9808050.

[BH2013] F. Bergeron, M. Haiman, Tableaux Formulas for Macdonald Polynomials, Special edition in honor of Christophe Reutenauer 60 birthday, International Journal of Algebra and Computation, Volume 23, Issue 4, (2013), pp. 833-852.

[Morse11] J. Morse, Combinatorics of the K-theory of affine Grassmannians, Adv. in Math., Volume 229, Issue 5, pp. 2950–2984.

[LamSchillingShimozono10] T. Lam, A. Schilling, M.Shimozono, K-theory Schubert calculus of the affine Grassmannian, Compositio Math. 146 (2010), 811-852.

[Morse2011] J. Morse, Combinatorics of the K-theory of affine Grassmannians, Adv. in Math., Volume 229, Issue 5, pp. 2950–2984.

[LamSchillingShimozono2010] T. Lam, A. Schilling, M.Shimozono, K-theory Schubert calculus of the affine Grassmannian, Compositio Math. 146 (2010), 811-852.

[ClSt03] Peter Clifford, Richard P. Stanley, *Bottom Schur functions*. arXiv math/0311382v2.

[ChariKleber2000] Vyjayanthi Chari and Michael Kleber. *Symmetric functions and representations of quantum affine algebras*. arXiv math/0011161v1

[KoikeTerada1987] K. Koike, I. Terada, *Young-diagrammatic methods for the representation theory of the classical groups of type Bn, Cn, Dn*. J. Algebra 107 (1987), no. 2, 466-511.

[ShimozonoZabrocki2006] Mark Shimozono and Mike Zabrocki. *Deformed universal characters for classical and affine algebras*. Journal of Algebra, **299** (2006). arXiv math/0404288.

[FD06] Francois Descouens, Making research on symmetric functions using MuPAD-Combinat. In Andres Iglesias and Nobuki Takayama, editors, 2nd International Congress on Mathematical Software (ICMS'06), volume 4151 of LNCS, pages 407-418, Castro Urdiales, Spain, September 2006. Springer-Verlag. arXiv 0806.1873

[HT04] Florent Hivert and Nicolas M. Thiery, MuPAD-Combinat, an open-source package for research in algebraic combinatorics. Sem. Lothar. Combin., 51 :Art. B51z, 70 pp. (electronic), 2004. http://mupad-combinat.sf.net/.

[MAC] Ian Macdonald, Symmetric Functions and Orthogonal Polynomials, Second edition. With contributions by A. Zelevinsky. Oxford Mathematical Monographs. Oxford Science Publications. The Clarendon Press, Oxford University Press, New York, 1995. x+475 pp. ISBN: 0-19-853489-2

[STA] Richard Stanley, Enumerative combinatorics. Vol. 2. With a foreword by Gian-Carlo Rota and appendix 1 by Sergey Fomin. Cambridge Studies in Advanced Mathematics, 62. Cambridge University Press, Cambridge, 1999. xii+581 pp. ISBN: 0-521-56069-1; 0-521-78987-7

[ST94] Scharf, Thomas, Thibon, Jean-Yves, A Hopf-algebra approach to inner plethysm. Adv. Math. 104 (1994), no. 1, 30-58. doi:10.1006/aima.1994.1019

[SZ2001] M. Shimozono, M. Zabrocki, Hall-Littlewood vertex operators and generalized Kostka polynomials. Adv. Math. 158 (2001), no. 1, 66-85.

[King] King, R. Branching rules for $GL_m \supset \Sigma_n$ and the evaluation of inner plethysms. J. Math. Phys. 15, 258 (1974) doi:10.1063/1.1666632

[SchaThi1994] Thomas Scharf, Jean-Yves Thibon. *A Hopf-algebra approach to inner plethysm*. Advances in Mathematics 104 (1994), pp. 30-58. ftp://ftp.mathe2.uni-bayreuth.de/axel/papers/scharf:a_hopf_algebra_approach_to_inner_plethysm.ps.gz

[ShaWach2014] John Shareshian, Michelle L. Wachs. *Chromatic quasisymmetric functions*. arXiv 1405.4629v2.

[HazWitt1] Michiel Hazewinkel. *Witt vectors. Part 1*. arXiv 0804.3888v1

[DoranIV1996] William F. Doran IV. *A Proof of Reutenauer's `-q_{(n)}` Conjecture*. Journal of combinatorial theory, Series A 74, pp. 342-344 (1996), article no. 0056. doi:10.1006/jcta.1996.0056

[BorWi2004] James Borger, Ben Wieland. *Plethystic algebra*. arXiv math/0407227v1

[Banc2011] E. E. Bancroft, *Shard Intersections and Cambrian Congruence Classes in Type A.*, Ph.D. Thesis, North Carolina State University. 2011.

[Pete2013] T. Kyle Petersen, *On the shard intersection order of a Coxeter group*, SIAM J. Discrete Math. 27 (2013), no. 4, 1880-1912.

[Read2011] N. Reading, *Noncrossing partitions and the shard intersection order*, J. Algebraic Combin., 33 (2011), 483-530.

[EilLan53] On the groups $H(\pi, n)$, I, Samuel Eilenberg and Saunders Mac Lane, 1953.

[Green55] Green, J. A. *The characters of the finite general linear groups*. Trans. Amer. Math. Soc. 80 (1955), 402–447. doi:10.1090/S0002-9947-1955-0072878-2

[Morrison06] Morrison, Kent E. *Integer sequences and matrices over finite fields*. J. Integer Seq. 9 (2006), no. 2, Article 06.2.1, 28 pp. https://cs.uwaterloo.ca/journals/JIS/VOL9/Morrison/morrison37.html

[PSS13] Prasad, A., Singla, P., and Spallone, S., *Similarity of matrices over local rings of length two*. arXiv 1212.6157

[PR22] Prasad, A., Ram, S., *Splitting subspaces and a finite field interpretation of the Touchard-Riordan formula*. arXiv 2205.11076

[R17] Ramaré, O., *Rationality of the zeta function of the subgroups of abelian p-groups*. Publ. Math. Debrecen 90.1-2. doi:10.5486/PMD.2017.7466

[NS] T. Nakanishi, S. Stella, Wonder of sine-Gordon Y-systems, to appear in Trans. Amer. Math. Soc., arXiv 1212.6853

[KnuMil] Knutson and Miller. *Subword complexes in Coxeter groups*. Adv. Math., 184(1):161-176, 2004.

[PilStu] Pilaud and Stump. *Brick polytopes of spherical subword complexes and generalized associahedra*. Adv. Math. 276:1-61, 2015.

[Las] Alain Lascoux, 'Young representations of the symmetric group.' http://phalanstere.univ-mlv.fr/~al/ARTICLES/ProcCrac.ps.gz

[Knuth1] Knuth, Donald (2000). "Dancing links". arXiv cs/0011047.

[CMS2012] Alexandre Casamayou, Nathann Cohen, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet, Nicolas M. Thiéry, Paul Zimmermann *Calcul Mathématique avec Sage* https://www.sagemath.org/sagebook/french.html

[CassNic10] Cassaigne J., Nicolas F. Factor complexity. Combinatorics, automata and number theory, 163–247, Encyclopedia Math. Appl., 135, Cambridge Univ. Press, Cambridge, 2010.

[AC03] B. Adamczewski, J. Cassaigne, On the transcendence of real numbers with a regular expansion, J. Number Theory 103 (2003) 27–37.

[BmBGL07] A. Blondin-Massé, S. Brlek, A. Glen, and S. Labbé. On the critical exponent of generalized Thue-Morse words. *Discrete Math. Theor. Comput. Sci.* 9 (1):293–304, 2007.

[BmBGL09] A. Blondin-Massé, S. Brlek, A. Garon, and S. Labbé. Christoffel and Fibonacci Tiles, DGCI 2009, Montreal, to appear in LNCS.

[Loth02] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

[Fogg] Pytheas Fogg, https://www.lirmm.fr/arith/wiki/PytheasFogg/S-adiques.

[Kolakoski66] William Kolakoski, proposal 5304, American Mathematical Monthly 72 (1965), 674; for a partial solution, see "Self Generating Runs," by Necdet Üçoluk, Amer. Math. Mon. 73 (1966), 681-2.

[Brlek89] Brlek, S. 1989. «Enumeration of the factors in the Thue-Morse word», *Discrete Appl. Math.*, vol. 24, p. 83–96.

[MH38] Morse, M., et G. A. Hedlund. 1938. «Symbolic dynamics», *American Journal of Mathematics*, vol. 60, p. 815–866.

PYTHON MODULE INDEX

C

r

INDEX

Non-alphabetical

A

B

C

D

E

F

H

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

X

Y

Z